

بسمه تعالی

آموزش لینوکس برای مبتدیان

تالیف:

مهندس مهدی فولادگر

نسخه ۰،۹

در حال ویرایش

فصل یک : لینوکس برای مبتدیان

یونیکس چیست؟ به معنای دقیق، این برنامه هسته سیستم عامل اشتراک زمانی، است، یعنی برنامه‌ای که منابع کامپیوتر را کنترل می‌کند و آنها را در بین کاربران، تخصیص می‌دهد. این برنامه به کاربران اجازه می‌دهد که برنامه‌هایشان را اجرا کنند، و وسایل جانبی (دیسکها، پایانه‌ها، چاپگر و از این قبیل وسایل) را که به سیستم ارتباط دارد را کنترل می‌کند و فایل سیستمی را فراهم می‌آورد که ذخیره سازی طولانی مدت اطلاعاتی همچون: برنامه‌ها، داده‌ها و اسناد را کنترل کند. به معنای کلی، یونیکس غالباً نه تنها شامل اساس و شالوده است بلکه، شامل برنامه‌های ضروری همچون: مترجم (مفسر)، ویراستار، زبانهای فرمانها، برنامه‌هایی جهت کپی و چاپ فایلها و از این قبیل خدمات است. باز هم به معنای وسیعتر، یونیکس حتی شامل برنامه‌های توسعه یافته توسط شما و یا دیگر کاربران است که این برنامه‌ها را در سیستم شما، بدون وقفه اجرا می‌شوند، برنامه‌هایی همچون: ابزارهایی برای تهیه اسناد، مراحلی برای تجزیه تحلیل آماری و بسته‌های نرم افزاری تصاویر (طرحها). کدامیک از این استفاده‌هایی که از سیستم یونیکس مطرح شد، با توجه به سیستمی که شما در حال استفاده از آن هستید، صحیح است. زمانی که ما از اصطلاح یونیکس در مابقی این کتاب استفاده می‌کنیم، محتوای کتاب باید آن معنی را که ضمنی است، مطرح کند. سیستم یونیکس بعضی اوقات در مقایسه با آنچه که هست، مشکل تر به نظر می‌رسد، یعنی برای مبتدیان درک چگونگی بهترین استفاده از امکانات موجود، مشکل است. اما خوشبختانه، شروع کار با آن سخت نیست، تنها اطلاع و دانش پیرامون چندین برنامه باعث راه افتادن شما در استفاده از این سیستم می‌شود. این فصل وسیله‌ای است برای کمک به شما در جهت شروع هر چه سریعتر استفاده از سیستم. این فصل به عنوان بررسی کلی است نه راهنمای استفاده، ما در نظر داریم که مطالب را به صورت جزئی‌تر در فصول بعدی دوباره، مطرح کنیم. ما در نظر داریم که در مورد این عرصه‌های وسیع صحبت کنیم:

- مبانی اصلی: شامل وارد شدن و خارج شدن از سیستم، فرامین ساده، تصحیح خطاهای تایپ، پست الکترونیکی، ارتباطات بین پایانه‌ای، است.

- استفاده روز به روز، شامل موارد زیر است: شامل، فایلها و فایل سیستم، فایلهای چاپگر، فهرست راهنماها، فرامین مورد استفاده عادی.

- فرمان مترجم (مفسر) یا shell شامل موارد زیر است:

مختصر نویسی نام فایل، تغییر جهت ورودی و خروجی، Pipes، پاک کننده‌های محیط و از بین بردن نمادها و تعریف مسیر جستجو برای فرامین.

اگر شما از یک سیستم یونیکس استفاده می‌کنید، (از آنجایی که اغلب قسمتهای این فصل برای شما آشنا است) مستقیماً به سراغ فصل 2 بروید. شما به یک کپی از فهرست راهنمای برنامه‌ساز یونیکس نیاز خواهید داشت، حتی زمانی که شما این فصل را مطالعه می‌کنید، غالباً بیان این امر که بهتر است، حتی زمانی که شما این فصل را مطالعه می‌کنید، غالباً بیان این امر که بهتر است شما برخی از مطالب موجود در فهرست را به جای تکرار آنها در اینجا مورد مطالعه قرار دهید، برای ما امری آسانتر به نظر می‌رسد. این کتاب در نظر ندارد که آن مطالب را جایگزین کند، بلکه در نظر دارد که به شما چگونگی بهترین استفاده از فرامین توصیف شده در آن، را نشان دهد. علاوه بر این، ممکن است تفاوت‌های بین آنچه که ما در اینجا می‌گوئیم و آنچه که در سیستم شما صحیح به نظر می‌رسد، وجود داشته باشد. فهرست راهنما، در شروع آنچه که برای یافتن برنامه‌های صحیح جهت کاربرد در رفع مشکل و فراگیری استفاده از آن، ضروری به نظر می‌رسد، دارای فهرست جابه‌جا شده‌ای است. نهایتاً، یک توصیه‌ای که لازم به نظر می‌رسد این است که از آزمایش کردن، نترسید. اگر

شما یک مبتدی هستید، مسائل تصادفی زیادی وجود دارد که شما می‌توانید با انجام آنها به خودتان یا دیگر کاربران آسیب برسانید. بنابراین، چگونگی عملکرد آن مسائل را با تلاش و کوشش بر روی آنها فرا بگیرید. این فصل، فصل بسیار طولانی است و بهترین راه برای مطالعه آن این است که هر دفعه چند صفحه را مطالعه کنید و مادامی که به پیش می‌روید، مطالب را مورد آزمایش و بررسی قرار دهید.

بخش 1.1 شروع کار

برخی از مقدمات در مورد پایانه‌ها و تایپ به منظور اجتناب از تشریح هر مورد پیرامون استفاده از کامپیوترها، ما باید فرض کنیم که شما آشنایی مختصری با پایانه‌های کامپیوتر و چگونگی استفاده از آنها دارید. در صورتی که هر کدام از جملات زیر مبهم و پیچیده باشد، ما باید به منظور درک آن از کارشناس سؤال کنیم. سیستم یونیکس، سیستم دو طرفه کامل است، یعنی علائم و حروفی که شما از طریق صفحه کلید، تایپ می‌کنید به سیستم انتقال پیدا می‌کند، به نحوی که سیستم آنها را به پایانه‌ها به منظور چاپ بر روی صفحه نمایش، بر می‌گرداند. به طور طبیعی، این فرایند انعکاس، علائم و حروف را به طور مستقیم در صفحه نمایش، کپی می‌کند، بنابراین شما می‌توانید آنچه را که شما در حال تایپ آن هستید را ببینید اما، بعضی اوقات همچون زمانی که شما در حال تایپ یک کلمه رمز هستید، فرایند انعکاس قطع می‌شود، بنابراین علائم و حروف بر روی صفحه نمایش، نمایان نمی‌شود. اغلب علائم و حروف صفحه کلید، بدون هیچ مفهوم خاصی از جمله علائم چاپی معمولی و رایج به شمار می‌آیند، اما تعداد کمی از آنها به کامپیوتر چگونگی تفسیر تایپ شما را اعلام می‌کند. به مراتب، مهمترین این موارد، کلید Return است. کلید Return به معنای پایان خط ورودی است، سیستم این معنا را با حرکت مکان نما پایانه به ابتدای خط بعدی در صفحه نمایش، منعکس می‌سازد. کلید Return باید قبل از اینکه سیستم بخواهد علائمی که شما تایپ کرده‌اید را تفسیر و تعبیر کند، فشار داده شود. کلید Return نمونه‌ای از علائم کنترلی است، یعنی یک علامت غیر قابل رؤیتی که برخی از جنبه‌های ورودی - خروجی را بر روی پایانه‌ها، کنترل می‌کند. بر روی هر کدام از پایانه‌های مستدل (منطقی)، Return دارای یک کلید مخصوص به خود است. اما اغلب علائم کنترلی چنین نیستند. در عوض آنها باید به وسیله پایین نگه داشتن کلید Control تایپ شوند، بعضی اوقات تحت عنوان Ctl یا CNTL یا CL RL از آنها یاد می‌شود. بعد از آن، کلید دیگری را فشار دهید که معمولاً آن کلید شامل یک حرف است. برای مثال: «Return» ممکن است به وسیله فشار دادن کلید Return یا شرایط مشابه به آن یا پایین نگه داشتن کلید Control و تایپ 'm' تایپ شود. بنابراین Return ممکن است شامل Ctl-d باشد که بیان کننده برنامه‌ای است که دیگر شامل ورودی نیست، Ctl-g که شامل صدای زنگ موجود بر روی پایانه‌ها است، Ctl-h غالباً، کلید backspace نامیده می‌شود که می‌تواند به منظور تصحیح خطاهای کپی به کار برده شود و Ctl-I که غالباً کلید Tab نامیده می‌شود که مکان‌نما را به نقطه Tab بعدی به جلو می‌برند، که این فرایند بیشتر شبیه به ماشین تحریر تنظیمی، عمل می‌کند. نقطه Tab در سیستم‌های یونیکس به اندازه هشت (8) کاراکتر از یکدیگر فاصله دارند. هر دو علامت backspace و Tab در اغلب پایانه‌ها دارای کلیدهای مخصوص به خود هستند. دو کلید دیگر وجود دارد که دارای معنای مشابه به یکدیگر می‌باشد. کلید delete که بعضی اوقات rub-out یا برخی دیگر از علائم اختصاری و break نامیده میشوند، و بعضی اوقات interrupt نامیده می‌شوند. در اغلب سیستم‌های یونیکس، کلید delete بدون لحظه‌ای درنگ برای پایان دادن برنامه، آن را بلافاصله متوقف می‌سازد. در برخی از سیستم‌ها، Ctl-c این کار را انجام می‌دهد. و در برخی از سیستم‌ها، با توجه به این نکته که پایانه‌ها چگونه متصل می‌شوند، کلید break برای delete یا Ctl-c به عنوان کلیدی هم معنا و دارای همان فعالیت، معرفی می‌شود.

بحث پیرامون یونیکس

اجازه دهید که با گفتگویی پیرامون شما و سیستم یونیکس تان ، بحث را آغاز کنیم، از طریق مثالهای موجود در این کتاب ، آنچه را که شما تایپ کرده‌اید در حروفهایی به صورت موب چاپ شده است ، پاسخهای کامپیوتر به صورت علائمی به سک ماشین تحریر است و توضیحات و تعریف به صورت موب است

ارتباطی را ایجاد کنید : شماره ای بگیرید یا دکمه‌ای را در صورت لزوم روشن کنید. سیستمتان باید این موارد را اعلام کند :

Establish a connection: dial a phone or turn on a switch as necessary.

Your system should say

```
login: You          Type your name, then press RETURN
password:          your password won't be echoed as you type it
you have mail.     There's mail to be read after you log in
$                 The system is now ready for your commands
$                 press RETURN a couple of times
$ date            What's the date and time?
sun sep 25 23:02:57 EDT 1983
$ who              who's using the machine?
jlp    tty0    sep 25 13:59
you    tty2    sep 25 23:01
mary   tty4    sep 25 19:03
doug   tty5    sep 25 19:22
egb    tty7    sep 25 17:17
bob    tty8    sep 25 20:48
$ mail              Read your mail
From doug sun sep 25 20:53 EDT 1983
give me a call sometime monday
?                  RETURN moves on to the next message
from mary sun sep 25 19:07 EDT 1983  Next message
Lunch at noon tomorrow?
? d                Delete this message
$                  No more this message
$ mail mary        send mail to mary
lunch at 12 is fine
ctl-d              End of mail
$                  Hang up phone or turn off terminal
and that's the end
```

بعضی اوقات تمام مواردی که در اینجا بیان شد شامل یک مرحله است، همانگونه که بسیاری از افراد چنین می‌کند. مابقی این بخش در مورد قسمت بالا به علاوه دیگر برنامه‌هایی که آن برای انجام کارهای مفید ممکن می‌سازد، بحث می‌کند.

ورود به سیستم

شما باید یک نام و یک کلمه رمز ورودی داشته باشید، شما می‌توانید این نام را از اسم یا برنامه نرم‌افزاری سیستم‌تان بگیرید. سیستم یونیکس قابلیت ارتباط با انواع وسیع پایانه‌ها را دارد، اما این سیستم به طرز قابل توجهی در جهت ابزارهایی با حروف کوچک است، یعنی حروفی که مطالب را از یکدیگر مجزا می‌سازد، اگر پایانه شما تنها حروف بزرگ ایجاد کند، (مثل برخی از ویدئو و پایانه‌های قابل حمل) زندگی آنچنان سخت خواهد شد که شما باید به دنبال پایانه دیگری باشید. از دکمه‌هایی که به طرز مناسبی بر روی وسیله شما قرار گرفته است، مطمئن شوید، از حروف بزرگ و کوچک، سیستم دو طرفه کامل و دیگر تجهیزاتی که کارشناسان به آنها توصیه می‌کنند، مثل سرعت یا سرعت باود نیز اطمینان حاصل کنید. با استفاده از هر عمل شگفت‌انگیزی که برای پایانه شما نیاز است، ارتباطی را ایجاد کنید، این فرایند ممکن است شامل ارتباط تلفنی یا صرفاً ضربه زدن به یک دکمه باشد. در موارد دیگر، سیستم باید عمل تایپ را انجام دهد.

برقراری ارتباط

اگر سیستم اطلاعات غیرمفید، تایپ کند، احتمالاً نشان دهنده این است که شما سرعت غلطی را انتخاب کرده‌اید، محیط مربوط به سرعت را و نیز دیگر دکمه‌ها را چک کنید. اگر این کار فایده نداشت، کلید `break` یا `interrupt` را چند بار آهسته، فشار دهید. اگر هیچکدام از این کارها پیام ارتباط را نمایان نکرد، فرد متخصصی را به کمک بگیرید. زمانی که شما ارتباط برقرار کردید، پیامی به این مضمون نمایان می‌شود:

اسم ارتباط‌تان را با حروف کوچک تایپ کنید. به دنبال آن، کلید `Return` را فشار دهید. اگر کلمه رمز نیاز باشد، در مورد آن از شما سؤال می‌کند و عمل چاپ مادامی که شما آن را (کلمه رمز) تایپ می‌کنید قطع می‌شود. حداکثر تلاش‌های برقراری ارتباط‌تان به صورت پیام واره انجام می‌شود. معمولاً به صورت یک علامت تنها، پیام واره نشان دهنده این است که سیستم برای پذیرش فرامینی از جانب شما آماده است. پیام واره غالباً شبیه یک علامت دلار \$ یا علامت درصد (%) است. پیام واره به وسیله برنامه‌ای که مترجم (مفسر) پیامها یا `shell` نامیده می‌شود، چاپ می‌شود که این برنامه واسط اصلی شما برای سیستم به شمار می‌آید. شاید پیامی از روز قبل از پیام کنونی موجود باشد یا اطلاعیه‌ای مبنی بر این که شما نامه دارید وجود داشته باشد. شاید پرسید که شما اکنون در حال استفاده از چه نوع پایانه‌ای هستید؟ پاسخ شما به سیستم برای استفاده از هر گونه خصوصیات که پایانه آنها را داراست، کمک می‌کند.

تایپ فرامین

زمانی که شما پیام واره ای را دریافت می‌کنید، می‌توانید فرامینی را تایپ کنید که این فرامین عبارتند از درخواستهایی که سیستم آنها را انجام می‌دهد. ما در نظر داریم که از برنامه‌ها به عنوان برنامه‌هایی هم معنا و مشابه فرامین استفاده کنیم. سیستم باید با زمان و تاریخ پاسخ دهد. سپس پیام دیگری را چاپ کند، بنابراین تمام مذاکرات شبیه به این مورد در پایانه شما خواهد بود:

\$ date

Mon sep 26 12:2057 EDT 1983

\$

Return را فراموش نکنید و \$ را تایپ نکنید.

اگر فکر می‌کنید که عمل را اشتباه انجام داده‌اید، کلید Return را فشار دهید، در این مرحله باید اتفاقی پیش آید. Return بار دیگر نمایان نمی‌شود اما شما در پایان هر خط به آن نیاز دارید. فرمان دیگری که باید انجام دهید، عبارتست از who که این فرمان هر فردی را که در حال برقراری ارتباط است را معرفی می‌کند :

\$ who

rlm tty0 sep 26 11:17

pjw tty4 sep 26 11:30

gerard tty7 sep 26 10:27

mark tty9 sep 26 07:59

you ttya sep 26 12:20

\$

اولین ستون نام کاربران است، دومین ستون نام سیستم برای ارتباط مورد استفاده است. [(tty) به معنای دور تحریر (tele type) است، هم معنی قدیمی برای پایانه]. مابقی نشان دهنده زمانی است کاربر ارتباط برقرار کرده است. ممکن است شما عمل زیر را انجام دهید.

\$ who am i

you ttya sep 26 12:20

\$

اگر شما نام فرمان را اشتباه تایپ کنید، و به سراغ یک فرمانی که وجود ندارد بروید، ممکن است با خود بگوئید که هیچ فرمانی که تحت این عنوان باشد، یافت نمی‌شود :

\$ whom

Misspelled command name

whom: not found

.... so system didn't know how to run it

\$

البته اگر شما از روی بی دقتی، نام فرمان اصلی را تایپ کرده باشید، آن برنامه اجراء خواهد شد، البته شاید با نتایجی مبهم.

رفتارهای غیر متداول پایانه ها

بعضی اوقات پایانه شما ممکن است به طرز غیر متداولی عمل کند، برای مثال هر حرف ممکن است دو بار تایپ شود، یا Return ممکن است، مکان نما را در ابتدای ستون خط بعدی قرار ندهد. شما معمولاً می‌توانید این جریان را با خاموش و روشن نمودن پایانه تثبیت کنید یا با خارج شدن از ارتباط و سپس برقراری در ارتباط مجدد. یا می‌توانید توصیف فرمان STTy (تنظیم انتخابهای پایانه) که در بخش اول فهرست راهنما است، را مطالعه کنید. به منظور کنترل ارتباطات از طریق مشخصه Tab در صورتی که پایانه شما دارای Tabs نباشد، فرمان را تایپ کنید.

\$ stty – tabs

و سیستم Tabs را به تعداد صحیحی از spaces تبدیل خواهد کرد. اگر پایانه های شما دارای نقاط Tab قابل تنظیم کامپیوتری باشند، فرمان Tabs آنها را به طرز صحیحی برای شما تنظیم خواهد کرد. (شما باید در مورد تایپ پایانه \$ Tabs به دقت عمل کنید، تا این عمل انجام شود، توصیف فرامین Tabs در فهرست راهنما را مطالعه کنید).

خطا در فرایند تایپ

اگر شما خطای تایپی داشته باشید و آن را قبل از اینکه کلید Return را فشار دهید، ببینید، دو روش برای اصلاح آن وجود دارد:

علائم پاک کردن در همان زمان یا از بین بردن تمام سطر و تایپ مجدد آن. اگر شما علامت از بین بردن سطر را تایپ کرده باشید (پیش فرض علامت @) این عمل باعث می شود که تمام سطر حذف شود، درست مثل زمانی که شما هرگز آن سطر را تایپ نکرده باشید و یا آن عمل (تایپ) را از سطر جدیدی شروع کرده باشید:

```
$ ddate@                               Completely botched;start over
date                                   on a new line
Mon sep 26 12:23:39 EDT 1983
$
```

علامت # sharp باعث پاک کردن آخرین حرف / علامت تایپ شده می شود، هر # تنها یک حرف را پاک می کند، و به ابتدای سطر بر می گردد (البته نه فراتر از آن سطر). بنابراین اگر شما اشتبهاً تایپ کنید، شما می توانید آن را به همین ترتیب اصلاح نمایید.

```
$ dd#atte##e                          Fx it as you go
Mon sep 26 12:24:02 EDT 1983
$
```

پاک کننده های خاص و علائم از بین بردن سطر، سیستم های بسیار وابسته هستند. در بسیاری از سیستمها (از جمله سیستمی که ما از آن استفاده می کنیم)، علامت پاک کن تغییر یافته است و تبدیل به back space (یا پسبرد) شده است، که در پایانه های ویدئو به خوبی عمل می کند. شما سریعاً می توانید متوجه شوید که کدام مورد سیستم شما قرار گرفته است:

```
$ datee                                Try
datee :not found                       It`s not
$ datee#                                Try #
Mon sep 26 12:26:08                   it is #
$
```

ما علامت backspace یا پسبرد را به صورت \backslash چاپ کرده ایم، بنابراین می توانید آن را ببینید). انتخاب شایع دیگر عبارتست از CTL-C برای از بین بردن سطر. ما از علامت # sharp به عنوان مشخصه پاک کن برای مابقی این بخش استفاده می کنیم، چرا که این علامت قابل رؤیت است، اما در صورتی که سیستم شما متفاوت باشد، تنظیمات پایانه ای را انجام دهید. بعد، در فرایند «اندازه گیری محیط» ما به شما می گوئیم که چگونه پاک کن و علامت از بین بردن سطر را برای آنچه که شما دوست دارید یک بار یا برای همیشه تنظیم کنید. برای چه چیزی شما باید پاک کن یا علامت از بین بردن سطر را به عنوان بخشی از متن، وارد کنید؟ اگر شما # یا علامت @ را به وسیله back slash جلوتر قرار دهید، این امر باعث می شود که آن علامت معنای خاص خودش را از دست بدهد.

بنابراین به منظور وارد کردن # یا @

@ \ یا \ @

را تایپ کنید. سیستم ممکن است مکان نمای پایانه را به سطر بعدی بعد از @ شما جلو ببرد، حتی اگر آن توسط backslash جلوتر واقع شده باشد. در صورتی که علامت @ ثبت شود، نگران نباشید. Backslash بعضی اوقات علامت escape نامیده می شود، که به

میزان وسیع برای نشان دادن این مطلب که علائم زیر به طریقی خاص و ویژه هستند، مورد استفاده قرار می‌گیرد. به منظور پاک کردن `backslash` شما باید دو علامت پاک‌کن را تایپ کنید. یعنی `# \` آیا می‌دانید چرا؟ علائمی که شما تایپ کردید مورد بررسی قرار گرفته و بوسیله توالی برنامه‌ها قبل از اینکه آنها به مقصدشان برسند، تعبیر و تفسیر می‌شوند.

و اینکه آنها دقیقاً چگونه تعبیر و تفسیر می‌شوند نه تنها به جایی که پایان می‌پذیرند بستگی ندارد بلکه به چگونگی رسیدن به آن مرحله نیز بستگی ندارد. هر علامتی را که تایپ می‌کنید، بلافاصله در قسمت پایانه انعکاس پیدا می‌کند مگر اینکه روند انعکاس‌یابی پایان پذیرد (پایانه خاموش شود) که این شرایط بسیار نادر است. زمانیکه `Return` را فشار دهید علامتها به صورت موقتی توسط هسته اصلی (`kernel`) کنترل می‌شوند، در نتیجه غلطها تایپی می‌توانند با پاک‌کن و مشخصه از بین برنده سطرها، تصحیح شوند. زمانی که پاک‌کن یا علامت از بین برنده سطر توسط `backslash` تقدم یابد، هسته اصلی (`kernel`) باعث می‌شود که `backslash` کنار زده شده و بقیه علامتها را بدون تعبیر و تفسیر کنترل کند. زمانیکه `Return` را فشار دهید، علائم حفظ شده به برنامه ای فرستاده می‌شوند که از پایان خوانده می‌شود. آن برنامه نیز به نوبه خود ممکن است علائم و مشخصه‌ها را به روشهای خاصی تعبیر و تفسیر کند، برای مثال: برنامه `shell` هرگونه تعبیر خاصی از علائم را در صورتی که توسط `backslash` تقدم یافته باشد را از بین می‌برد. در مورد این موضوع در فصل 3 بحث خواهیم کرد. اکنون، باید به خاطر داشته باشید که `kernel` (هسته اصلی) پاک‌کن و علامت از بین برنده سطرها را پردازش می‌کند و `backslash` تنها در صورتیکه پاک‌کن و علامت از بین برنده سطر را در تقدم قرار داده باشد، هر علامتی را که بعد از آن رها شده باشد را به وسیله برنامه‌های دیگری به خوبی تعبیر و تفسیر می‌کند.

تمرین 1-1. توضیح دهید در صورتی که `$ \date @` چه اتفاقی می‌افتد.

تمرین 2-1. اغلب برنامه‌های `shell` (اگر چه `shell` چاپ هفتم شامل آن نمی‌شود) علامت `#` را به عنوان معرفی به موضوع تعبیر می‌کنند، و تمام متن را از علامت `#` تا پایان سطر نادیده می‌گیرند. با توجه به این موضوع، نسخه زیر را توضیح دهید، فرض کنید که علامت پاک‌کن نیز به صورت `#` است:

```
$ date
Mon sep 12 : 39 : 56 EDT 1983
$ # date
Mon sep 26 12 : 24: 21 EDT 1983
$ \ # date
$ \ \ # date
# date : not found (یافت نشد)
$
```

جلوتر تایپ کردن

هسته اصلی (`kernel`) آنچه را که شما تایپ می‌کنید را همزمان با تایپ آن می‌خواند، حتی اگر هسته اصلی مشغول کار دیگری باشد، بنابراین با سرعت مورد دلخواه مطلب مورد نظران را تایپ کنید، و یا حتی هر زمان که می‌خواهید، حتی زمانی که برخی از فرمانها به نفع شما چاپ نشوند. در صورتیکه هنگام عمل چاپ سیستم، شما تایپ انجام دهید، علائم ورودی شما در ترکیب با علائم خروجی به نظر می‌رسند اما، آنها ذخیره شده و به ترتیب صحیح تفسیر می‌شوند. می‌توانید فرامین را یکی پس از دیگری بدون انتظار کشیدن برای اتمام آنها یا حتی شروع آنها، تایپ کنید.

توقف برنامه

شما می‌توانید اغلب فرامین را با تایپ علامت `delete` متوقف سازید. کلید `Break` که در اغلب پایانه‌ها یافت می‌شوند نیز می‌تواند این کار را انجام دهد، اگر چه این عمل به سیستم بستگی دارد. در چندین برنامه همچون برنامه ویراستاری متن، کلید `delete` هر آنچه را که برنامه انجام داده است را متوقف می‌سازد اما شما را در آن برنامه رها می‌کند. خاموش کردن پایانه یا برداشتن تلفن، اغلب برنامه‌ها را متوقف می‌سازد. اگر شما تنها بخواهید از حالت توقف بیرون آئید، برای مثال: برای حفظ برخی از مسائل ضروری از ناپدید شدن در صفحه نمایش، فرمان `CTL-S` را تایپ کنید. فرایند بازدهی تقریباً سریع متوقف می‌شود، بدین ترتیب برنامه شما موقتاً تا زمانیکه شما دوباره آن را شروع کنید، متوقف می‌شود. زمانیکه شما بخواهید دوباره برنامه را ادامه دهید، فرمان `CTL-q` را تایپ کنید.

قطع کردن ارتباط

ار دادن `delete` خارج شوید. توجه کنید که خطاهای تایپی شما بر روی پایانه مری ظاهر نمی‌شود. اگر شما در نظر دارید که به فردی که در ارتباط نیست نامه بنویسید یا به فردی که نمی‌خواهد با کراه مناسب برای قطع کردن ارتباط عبارتست از تایپ فرمان `CTL-d` به جای فرمان وجود ندارد. به منظور پاسخ دهی، فرمان `write Mary $` را تایپ کنید. این عمل مسیر ارتباطی دو طرفه‌ای را بوجود می‌آورد. اکنون سطرهایی را که مری بر روی پایانه خودش تایپ کرده بر روی پایانه شما نیز تایپ می‌شود و بر عکس، اگر چه این مسیر آرام و آهسته عمل می‌کند اما تا حدی بیه به `talking to the moon` است. اگر در اواسط کاری قرار دارید، شما مجبورید در شرایطی قرار بگیرید که بتوانید فرمانی را تایپ کنید. طبیعتاً هر برنامه ایی که شما در حال اجرای آن هستید باید متوقف شود اما برخی از برنامه‌ها همچون ویراستار `write` به خودی خود، دارای فرمان `(!)` هستند که این فرمان به طور موقتی از برنامه `shell` خارج می‌شود، جدول 2 را در قسمت ضمیمه 1 مورد مطالعه قرار دهید. فرمان `write` هیچ قانونی را وضع نمی‌کند، بنابراین برای حفظ آنچه تایپ می‌کنیم و جلوگیری از مخدوش شدن با آنچه که مری تایپ می‌کند، دستورالعملی لازم است. یک قانون این است که چرخشی کرده و هر چرخش را با `(0)` پایان دهد که این علامت برای کلمه `(over)` در نظر گرفته شده و برای مشخص نمودن هدف‌تان، با علامت `(00)`، آن را متوقف سازید و از آن خارج شوید، این علامت برای `out` و `over` در نظر گرفته شده است.

\$ Eof

\$

همچنین می‌توانید از `write` با فشار دادن `delete` خارج شوید. توجه کنید که خطاهای تایپی شما بر روی پایانه مری ظاهر نمی‌شود. اگر شما در نظر دارید که به فردی که در ارتباط نیست نامه بنویسید یا به فردی که نمی‌خواهد با کراه مناسب برای قطع کردن ارتباط عبارتست از تایپ فرمان `CTL-d` به جای فرمان وجود ندارد. به منظور پاسخ دهی، فرمان `write Mary $` را تایپ کنید. این عمل مسیر ارتباطی دو طرفه‌ای را بوجود می‌آورد. اکنون سطرهایی را که مری بر روی پایانه خودش تایپ کرده بر روی پایانه شما نیز تایپ می‌شود و بر عکس، اگر چه این مسیر آرام و آهسته عمل می‌کند اما تا حدی بیه به `talking to the moon` است. اگر در اواسط کاری قرار دارید، شما مجبورید در شرایطی قرار بگیرید که بتوانید فرمانی را تایپ کنید. طبیعتاً هر برنامه ایی که شما در حال اجرای آن هستید باید متوقف شود اما برخی از برنامه‌ها همچون ویراستار `write` به خودی خود، دارای فرمان `(!)` هستند که این فرمان به طور موقتی از برنامه `shell` خارج می‌شود، جدول 2 را در قسمت ضمیمه 1 مورد مطالعه قرار دهید. فرمان `write` هیچ قانونی را وضع نمی‌کند،

بنابراین برای حفظ آنچه تایپ می‌کنیم و جلوگیری از مخدوش شدن با آنچه که مری تایپ می‌کند، دستورالعملی لازم است. یک قانون این است که چرخشی کرده و هر چرخش را با (0) پایان دهد که این علامت برای کلمه (over) در نظر گرفته شده و برای مشخص نمودن هدفتان، با علامت (00)، آن را متوقف سازید و از آن خارج شوید، این علامت برای out و over در نظر گرفته شده است. سی ارتباط داشته باشد، در این صوسی ارتباط داشته باشد، در این صورت باید به شما بگوید. در صورتیکه هدف مورد نظر ارتباط برقرار کرده باشد اما پس از یک وقفه مناسب پاسخ ندهد، در این صورت این فرد ممکن است مشغول کاری بوده و یا از پایانه دور بوده باشد، در این حالت CTI-d یا delete را تایپ کنید. اگر می‌خواهید فردی مزاحم شما نشود، از فرمان (msg1) استفاده کنید.

اخبار

بسیاری از سیستمهای یونیکس، سرویسهای خبری دارند، و به این ترتیب کاربران را پهلو به پهلو و وقایع جالب و نه چندان جالب حفظ می‌کنند. فرمان \$ news را تایپ کنید. همچنین شبکه وسیعی از سیستمهای یونیکس وجود دارد که از طریق خطوط تلفن در تماس قرار می‌گیرند، در مورد شبکه‌های خبری و useNet از یک کارشناس سؤال کنید.

فهرست راهنما

راهنمای برنامه‌ساز یونیکس اغلب آنچه را که شما نیازمند دانستن پیرامون سیستم هستید را توصیف می‌کند. بخش 1، با فرامین ارتباط دارد یعنی شامل فرامینی است که ما در این بخش در مورد آنها بحث می‌کنیم. بخش 2، سیستم صوت را توصیف می‌کند، موضوع فصل 7 و بخش 6، اطلاعاتی پیرامون بازیها دارد. ما بقی بخشها در مورد نقشهای استفاده از برنامه های-C، فرمت فایل و حفظ سیستم صحبت می‌کند. (تعداد این بخشها از سیستمی به سیستم دیگر فرق می‌کند).

فهرستهای جابه جا شده در ابتدای کتاب راهنما را فراموش نکنید، شما می‌توانید سریعاً از آن گذشته و برای فرامینی که ممکن است به آنچه که شما می‌خواهید انجام دهید مربوط باشد، مطالعه‌ای گذرا از آن انجام دهید. همچنین مقدمه‌ای در مورد سیستم وجود دارد که اطلاعاتی در مورد چگونگی عملکرد آن ارائه می‌دهد. غالباً فهرست راهنما بر روی خط حفظ می‌شود، بنابراین، شما می‌توانید آن را از طریق پایانه خودتان مطالعه کنید. اگر شما غرق در کاری شدید و نتوانستید فرد متخصصی را به کمک بگیرید، می‌توانید صفحه فهرست راهنما را بر روی پایانه خودتان با فرمان man comman-name چاپ کنید.

بنابراین برای اطلاع از اینکه فرمان چه کسی بوده است: فرمان

دیگری؛ و این موضوع به shell می‌گوید که ورودی مسیر دیگری موجود نمی‌باشد. (اینکه چگونه این عمل واقعاً صورت می‌پذیرد در فصل بعدی توضیح داده می‌شود) معمولاً می‌توانید پایانه را خاموش و یا گوشی تلفن را بردارید اما، این که این عمل واقعاً ارتباط شما را قطع می‌کند یا نه، به سیستم تان بستگی دارد.

پست الکترونیک

سیستم باعث می‌شود تا سیستم پستی جهت برقراری با کار بر دیگری، فراهم شود. بنابراین در صورتیکه روزی ارتباط برقرار شود شما و پیام زیر را خواهید دید:

«نامه دارید»

و این پیام را قبل از اولین برقراری ارتباط خواهید دید. به منظور خواندن نامه‌تان فرمان `mail$` را تایپ کنید. نامه شما چاپ خواهد شد و پیامی که پدیدار می‌شود به عنوان اولین و جدیدترین پیام است. بعد از هر عنوان نامه صبر کنید تا شما آنچه را که می‌خواهید در مورد آن اجرا کنید را اعلام کنید. دو جواب اصلی عبارتند از `d` که این جواب پیام را حذف کنید و دیگری `Return` که کاری انجام نمی‌دهد (بنابراین نامه باقی می‌ماند تا هر زمان دیگری که خواستید نامه‌تان را بخوانید) جوابهای دیگر عبارتند از `P` که باعث می‌شود پیام دوباره چاپ شود، نام فایل `S` که باعث می‌شود نامه در فایللی که شما به آن نام داده‌اید ذخیره شود و `q` که برای رد شدن از نامه، طراحی شده است. (در صورتیکه شما ندانید که یک فایل شامل چه چیزی است به فکر مکان خاصی باشید که بتوانید اطلاعات را تحت یک نام انتخابی ذخیره کنید و آن را بعداً بازیابی کنید. فایلها موضوع بخش 102 بوده و در واقع بخش عظیمی از کتاب را به خود اختصاص داده‌اند. پست الکترونیکی یکی از آن برنامه‌هایی است که احتمالاً متفاوت از آنچه می‌باشد که ما در اینجا توصیف می‌کنیم. انواع بسیاری از آنها وجود دارد. جهت اطلاع از جزئیات به کتاب راهنما مراجعه کنید. فرستادن نامه به افراد کار ساده‌ای است. فرض کنید نامه به فردی با نام ارتباطی `nico` فرستاده می‌شود. راحت ترین و ساده‌ترین روش عبارتست از:

`$ mail nico`

اکنون در متن نامه هر تعداد سطری که می‌خواهید تایپ کنید بعد از آخرین سطر نامه فرمان `Gntrd-d` را تایپ کنید.

`$Ctl - d`

`Ctl-d` به معنای پایان نامه به وسیله فرمان `mail` است که بیان می‌دارد هیچ ورودی دیگری موجود نمی‌باشد اگر شما در نیمه راه نظراتان را مبنی بر ترکیب نامه تغییر دادید، کلید `delete` را به جای `ctl-d` فشار دهید. نامه‌ای که به صورت نیمه کاره شکل گرفته است در فایللی که به نام `dead.letter` است به جای فرمان انتقال (`Send`) ذخیره می‌شود. به عنوان تمرین، نامه‌ای را برای خودتان بفرستید، سپس، علامت `mail` را جهت خواندن نامه تایپ کنید.

(این عمل به آن اندازه‌ای که به نظر می‌رسد گمراه کننده نیست، این کار مکانیسم یادآوری مناسبی است). روشهای دیگری برای فرستادن نامه وجود دارد شما می‌توانید نامه‌ای را که از قبل آماده شده است را بفرستید، می‌توانید نامه‌ای را به تعدادی از افراد مورد نظرتان در یک زمان خاص بفرستید و ممکن است قادر باشید نامه را به افرادی در سیستمهای دیگر بفرستید. جهت اطلاع از جزئیات بیشتر نسخه فرمان `mail` را در بخش 1 فهرست راهنمای برنامه‌ساز یونیکس مطالعه کنید. از اینجا به بعد ما از مفهوم `mail (1)` به معنای صفحاتی که `mail` را در بخش 1 کتاب راهنما توصیف می‌کنند، استفاده می‌کنیم همچنین ممکن است سرویس تقویم نیز وجود داشته باشد (بخش ، تقویم 1 را مطالعه کنید) و ما در فصل 4 به شما نشان خواهیم داد که نامه چگونه در صورتی که پیش از این کاری بر روی آن انجام نگرفته است، تنظیم می‌شود.

نوشتن نامه به کاربر دیگری

از سیستم یونیکس شما دارای چندین کاربر است. روزی، ناگهان، پایانه شما پیغامی شبیه به این موضوع چاپ خواهد کرد:

message from mary tty 7

که این پیغام با صدای بوق قابل توجهی همراه است. مری می‌خواهد که به شما نامه‌ای بنویسد، اما در صورتی که شما عمل واضحی انجام ندهید، امکان اینکه شما قادر به پاسخ دهی باشید، \$ man who را تایپ کنید و البته، \$ man man در مورد فرمان فرد مورد نظر اطلاعاتی می‌دهد.

دستورالعمل‌های کمک کننده به کامپیوتر

سیستم شما ممکن است دارای فرمانی باشد که Learn نام داد که این فرمان دستورالعمل‌های کمک دهنده به کامپیوتر را بر روی فایل سیستم و فایل‌های اصلی ویراستار، آماده سازی اسناد و حتی برنامه C فراهم می‌آورد. فرمان \$ learn را تایپ کنید. اگر فرمان Learn در سیستم شما باشد، در این صورت آنچه را که شما باید از آنجا انجام دهید را به شما اعلام می‌کند. در صورتی که این فرمان اجرا نشود در مورد فرمان teach اقدام کنید.

بازیه‌ها

بازیها همیشه پذیرفته شده نیستند اما یکی از بهترین راهها برای کسب آسایش و راحتی از کامپیوترها و پایانه‌ها، اجرای بازیها است. سیستم یونیکس با تجهیزات معمولی از بازیها همراه است و غالباً این تجهیزات به صورت محلی ضمیمه می‌شوند. اطراف را مورد بررسی قراردادده یا بخش 6 فهرست راهنما را مورد مطالعه قرار دهید.

2-1 (استفاده روز به روز: فایلها و فرامین معمول)

اطلاعات در سیستم یونیکس در فایلها ذخیره می‌شود، که بیشتر شبیه به فایل‌های اداری معمولی هستند. هر فایل دارای نام، محتوا و مکانی برای حفظ آن است و برخی از اطلاعات اداری (اجرایی) همچون فردی که مالک آن است و میزان بزرگی آن، می‌باشد فایل ممکن است دارای نامه یا لیستی از نامها و آدرسها باشد یا منبع اطلاعات برنامه‌ها یا اطلاعاتی که باید به به وسیله برنامه استفاده شود یا حتی دارای برنامه‌ها به شکل قابل توجهی و دیگر موضوعات غیر متنی باشد. فایل سیستم یونیکس به صورت سازمان یافته است. بنابراین شما می‌توانید فایل‌های شخصی خودتان را بدون مداخلت با فایل‌های متعلق به افراد دیگر حفظ کنید و افراد را از دخالت پیدا کردن در مورد فایل‌های خودتان منع نمائید. تعداد زیادی برنامه وجود دارد که فایلها را کنترل می‌کند. اما اکنون ما تنها به اکثر آنها می‌پردازیم که به طور مکرر توسط افراد مورد استفاده قرار می‌گیرد، سرورکار داریم. فصل 2 شامل بحث سیستماتیکی در مورد فایل سیستم است و بسیاری از دیگر فرامین مربوط به فایلها را معرفی می‌کند.

ایجاد فایل - فایل ویراستار

در صورتی که بخواهید برگه یا نامه و یا یک برنامه را تایپ نمائید. چگونه اطلاعات ذخیره شده در ماشین- (سیستم) را بدست می‌آورید؟

اغلب این کارها با فایل ویراستار متن که برنامه‌ای برای ذخیره کردن و کنترل اطلاعات در کامپیوتر است انجام می‌گیرد، تقریباً هر سیستم یونیکس دارای ویراستار صفحه نمایش است و ویراستاری که مزیت‌هایی را از پایانه‌های مدرن گرفته و بدین وسیله تأثیرات تغییرات ویراستاری شما را در متن همزمان با بوجود آوردن متن، نشان می‌دهد. دو نمونه از مشهورترین این فایلها عبارتند از emacs,vi ما در

نظر نداریم که هیچ نوع ویراستار صفحه نمایش خاصی را در اینجا تشریح کنیم، و این امر تا حدی به دلیل محدودیتهای حروف چینی و تا حدی به دلیل عدم وود نوعی استاندارد است. با این وجود یک نمونه ویراستار قدیمی تر که ed نامیده می شود وجود دارد که مطمئناً بر روی سیستم شما موجود است. این نوع ویراستار هیچ گونه مزیتی از خصوصیات پایانه های دریافت نکرده است، بنابراین بر روی هر گونه پایانه ای کار خواهد کرد. این فایل همچنین اصولی از دیگر برنامه های ضروری را تشکیل می دهد (شامل برخی از ویراستارهای صفحه نمایش) بنابراین فایل ارزش یادگیری را دارد. ضمیمه 1 شامل نسخه مختصری از این فایل است. این که شما چه نوع ویراستاری را ترجیح می دهید مهم نیست، بلکه مهم است که شما آن را به خوبی فرا گرفته تا بتوانید فایل هایی را ایجاد نمایید. در اینجا ما از فایل ed جهت ارائه بحث استفاده می کنیم و مطمئن هستیم که شما می توانید مثال های ما را بر روی سیستم تان اجرا نمایید اما حتماً از ویراستاری که بهتر دوست دارید استفاده نمایید. به منظور استفاده از ed جهت ایجاد فایل های junk نامیده می شود (به همراه متونی در آن) کارهای زیر را انجام دهید:

فرمان ed ویراستار متن را به منظور
افزودن متن به کمک بطلید a

اکنون در هر متنی که می خواهید

تایپ کنید.

w junk

به وسیله خودش ' . ' را به منظور متوقف نمودن

39

افزایش متن، تایپ کنید. متن تان را در فایل های که

ed

تحت عنوان junk است، بنویسید. q

\$

تعدادی از علائم نوشته شده را چاپ می کند.

از ed خارج شوید.

فرمان a ("ضمیمه سازی") به ed می گوید شروع به جمع آوری متن کند.

" . " که نشان دهنده پایان متن است باید در شروع خط (سطر) به وسیله خودش تایپ شود. آن را تا زمانیکه تایپ شود فراموش نکنید. دیگر فرامین ed نباید سازماندهی شوند هر چیزی که شما تایپ می کنید مادامی که متن افزوده می شود مورد بررسی و کنترل قرار خواهد گرفت) فرمان ویراستاری w ("نوشتن") اطلاعاتی را که شما تایپ می کنید را ذخیره می کند "w junk" آن را در فایل های junk نامیده می شود و ما junk را از این رو انتخاب کرده ایم تا پیشنهاد کنیم که این فایل خیلی مهم نیست. Ed با تعدادی از علائمی که آن را در فایل قرار می دهد پاسخ می دهد. تا زمانیکه w فرمان ندهد، هیچ چیزی به صورت دائمی ذخیره نمی شود بنابراین اگر شما گوش را گذاشته و به خانه بروید، اطلاعات در فایل ذخیره شده است. (اگر شما گوش می دهید را مادامی که در حال ویرایش هست بگذارید، اطلاعاتی که در حال پردازش است، در فایل های ed.hup نامیده می شود ذخیره می شود که بدین ترتیب شما می توانید به فعالیت متنی خودتان ادامه دهید.) اگر سیستم دچار سروصدا شد (برای مثال به صورت غیرمنتظره و به دلی اشکالات سخت افزاری و نرم افزاری بایستید) البته زمانیکه شما در حال ویراستاری هستید، فایل شما تنها دارای آنچیزی است که

آخرین فرمان نوشتاری آنجا قرار داده است. اما پس از اینکه w (نوشتن) اطلاعات به صورت دائمی ثبت شد، شما می‌توانید بعداً با تایپ فرمان $\$ ed junk$ دوباره به آنها دسترسی پیدا کنید البته می‌توانید متنی را که شما تایپ کرده‌اید را ویراستاری کنید و یا خطاهای املائی آن را تصحیح نمایید. کلمات را تغییر دهید، پاراگرافها را دوباره مرتب کنید و کارهایی از این قبیل، زمانی که شما این کارها را انجام دادید، فرمان q (خروج) اعلام می‌دارد که از فایل ویراستاری خارج شوید.

چه فایل‌هایی خارج از برنامه هستند؟

اجازه دهید دو فایل ایجاد کنیم فایل $Temp$, $junk$ بنابراین ما آنچه را که داریم می‌شناسیم.

$\$ ed$

a

باشد یا نباشد

•

$W junk$

19

q

$\$ ed$

A

سوالی مطرح است

•

$w Temp$

22

q

$\$$

علامتی که از ed منظور می‌شود عبارتست از علامتی در پایان هر خط (سطر) و که خط جدید یا سطر جدید نامیده می‌شود این علامت چگونگی نشان دادن $Return$ سیستم را نشان می‌دهد. فرمان Is (نه محتوا) از فایلها را فهرست می‌کند:

$\$ ls$

$junk$

$Temp$

$\$$

که این نامها، دوفایلی هستند که تنها ایجاد شده‌اند (ممکن است فایل‌های دیگری نیز وجود داشته باشند که شما ایجاد نکرده باشید) نامها به ترتیب حروف الفبایی به صورت اتوماتیک طبقه بندی می‌شوند. Is شبیه اغلب فرامین، دارای انتخابهایی است که ممکن است برای تغییر رفتارهای غلط به کار برده شوند. انتخابها نام فرامین را در سطر فرمانها پی گیری می‌کنند و معمولاً از علامت تفریق ' - ' و از یک حرف تنها که معمولاً دارای معنا است، تشکیل شده است. برای مثال، $Is-t$ باعث می‌شود که فایلها به ترتیب زمانی فهرست شوند: ترتیبی که در آن‌هایی که آخر تغییر یافته‌اند غالباً جدیدتر بوده و در اولویت آخر هستند.

$\$ ls-t$

$Temp$

$Junk$

$\$$

انتخاب 1 - فهرست طولی را نشان می‌دهد که اطلاعات زیادی در مورد هر فایل فراهم می‌آورد.

\$ 1s-1

Total 2

- rw-r--r--

1 jou 19 sep 26 16:25 junk

- rw-r--r--

22 sep 26 16:26 Temp

- rw-r--r--

" total 2 " نشان می دهد که چه تعداد از بلوکهای فضای دیسک در یک فایل اشغال شده است و یک بلوک معمولاً شامل 512 یا 1024 کاراکتر است.

ردیف -rw-r--r-- نشان می دهد که چه کسی اجازه خواندن و نوشتن در یک فایل را دارد. در این مورد مالک اصلی (شما) می توانید بخوانید و بنویسید، اما دیگران تنها می توانند آن فایل را بخوانند عدد "1" که به دنبال آن می آید نشان دهنده تعداد ارتباط به فایل است از این موضوع تا رسیدن به فصل 2 صرف نظر کنید. " شما " مالک فایل هستید، یعنی ، فردی که آن فایل را ایجاد کرده است. 22 و 19 تعداد کاراکترهایی است که با فایلها در مکاتبه بودند. که این تعداد با تعدادی که شما از فایل ed دریافت کرده اید، هماهنگی دارد. تاریخ و زمان نشان می دهد که چه زمانی فایل آخرین تغییر را پیدا کرده است انتخابها می تواند گروه بندی شود: Is, It داده هایی همانند 1s-1 را ارائه می دهد اما بر اساس آخرین فایلها طبقه بندی می شود. انتخاب -u اطلاعاتی را در مورد زمان خاصی که فایل مورد استفاده قرار گرفته است ارائه می دهد:

Is - lut لیست طولی (-l) را به ترتیب آخرین استفاده نشان می دهد. انتخاب مورد -r ترتیب خروجی را بر عکس می کند. بنابراین، Is-It به ترتیب آخرین مورد استفاده شده فهرست می شود. همچنین شما می توانید فایلها را به نامهایی که مورد علاقه تان است، نام گذاری کنید و Is تنها اطلاعاتی در مورد خودشان را ، فهرست می کنند:

\$ 1s - 1 junk

- rw-r--r-- 1jou 19 sep 26 16:25 junk

\$

ردیف یا رشته هایی که برنامه ها را ادامه می دهند بر اساس سطر فرمان نام گذاری می شوند، مثل 1- , junk در نمونه بالا، این موارد تحت عنوان «شناسنامه برنامه» نام گذاری و شناخته می شوند. شناسنامه معمولاً شامل انتخابها یا نامهای فایلهایی هستند که بوسیله فرمانها مورد استفاده قرار می گیرند. انتخابها را به وسیله علامت منها (-) و یک حرف تنها مشخص کنید مثل -t یا به صورت ترکیبی -It که این مورد بیشتر شایع است. به طور کلی در صورتیکه یک فرمان پذیرفته شود، مثل شناسه انتخابی، آنها هر گونه شناسه نام فایلها را در تقدم قرار می دهند اما ممکن است در هر ترتیب دیگری جور دیگری به نظر برسد. اما برنامه های یونیکس در کنترلشان پیرامون انتخابهای چندگانه، بی نظم هستند. برای مثال هفتمین چاپ استاندارد Is مورد زی را

\$ 1s - 1 -t

در چاپ هفتم فعال نمی باشد

به عنوان معادلی برای Is-It نمی پذیرد، هر چند دنبال برنامه ها نیازمندند انتخابهای چند گانه جهت مجزا شدن می باشند.

مادامی که اطلاعات بیشتری فراگیرید، متوجه خواهید شد که نظم کم وجود داشته یا سیستم در حالت شناسه انتخابی قرار دارد. هر زمان دارای ویژگی منحصر به فرد خود و انتخاب های مخصوص به خود در مورد آن خروجی که به معنای آن است می باشد (غالباً این موارد متفاوت از همان نقش در فرامین دیگر می باشند) این رفتارهای غیرقابل پیش بینی آزار دهنده بوده و غالباً به عنوان نقصان و ضعف بزرگی در یک سیستم مطرح می باشند. اگر چه این شرایط در حال بهبودیابی هستند (انواع جدید غالباً دارای یکنواختی بیشتری می باشد) و ما تمام آنچه که ما می توانیم پیشنهاد کنیم این است که شما تلاش کنید تا زمانیکه که در برنامه شخصیتان چیزی می نویسید به بهترین نحو این کار را انجام دهید و در این میان یک کپی از مهارتتان را حفظ کنید.

فایل‌های چاپگر - cat , pr

اکنون شما دارای برخی از فایل‌ها هستید، اکنون چگونه شما محتوای آنها را مورد بررسی قرار دهید، برنامه‌های زیادی وجود دارد که این کار را انجام می‌دهد، که احتمالاً این عمل بیشتر از مورد نیاز است. یکی از این امکانات این است که از ویراستار استفاده کنید:

ed به تعداد 19 کاراکتر را در junk ثبت می‌کند

ed junk \$
سطر 1 تا آخر را چاپ می‌کند

19
p\$, 1
و فایل تنها یک سطر دارد

باشد یا نباشد
تمام کارها انجام گرفته است

q

\$

ed با ثبت تعدادی از کاراکترها در junk شروع به کار می‌کند فرمان \$p , 1 ظاهر دارد که آن تمام سطرها را در فایل چاپ کند. پس از اینکه فراگرفتید چگونه از ویراستار استفاده کنید، می‌توانید به صورت انتخابی در مورد بخش‌هایی که چاپ می‌کنید، عمل کنید، زمانی که امکان استفاده از ویراستار برای چاپ وجود نداشته باشد موقعیتهایی وجود دارد. برای مثال، در مورد میزان بزرگی فایل ed که بتواند مورد استفاده قرار گیرد، محدودیتهایی وجود دارد (چندین هزار سطر) علاوه بر این، این فایل تنها یک فایل را در یک زمان چاپ می‌کند، و بعضی اوقات شما می‌خواهید چندین فایل را چاپ کنید به صورتی که یکی پس از دیگری بوده و هیچ‌گونه توقفی در حین چاپ وجود نداشته باشد. بنابراین در این زمینه مجموعه‌ای از راه‌حل‌ها وجود دارد. اولین مورد فایل cou است، که این فایل از تمام فرامین چاپ ساده‌تر است. اولین مورد فایل تمام فایل‌هایی که توسط شناسه‌هایشان نام گذاری شده‌اند را چاپ می‌کنید:

\$ cat junk

To be or not to be

\$ cat Temp

That is the question

\$ cat junk Temp

To be or not to be

That is the question

\$

فایل نام گذاری شده یا فایل‌ها در پایانه یکی پس از دیگری بدون وقفه و فاصله الحاق می‌شوند،

(از این رو نام «cat» انتخاب شده است) در مورد فایل‌های کوتاه مشکلی وجود ندارد اما برای فایل‌های بزرگ مشکل پیش می‌آید در صورتی که شما با سرعت بسیار بالا به کامپیوترتان متصل شوید، باید سریعاً Ctl-s را برای متوقف کردن خروج از Cat قبل از اینکه این برنامه از صفحه نمایش شما خارج شود اجرا کنید هیچ فرمان استاندارد برای چاپ یک فایل بر روی پایانه ویدئویی که در یک زمان دارای صفحه نمایش کامل است وجود ندارد، اگر چه تقریباً هر سیستم یونیکس دارای یک نمونه از این فرمان می‌باشد. سیستم شما ممکن است دارای فرمان make یا pg باشد سیستم ما دارای فرمانی است که p نامیده می‌شود، ما به شما اجراء آن را در فصل 6 نشان خواهیم داد. همچون cat فرمان pr محتوای تمام فایل‌هایی که در یک لیست، نام گذاری شده‌اند را چاپ می‌کند اما به شکل مناسبی برای چاپگرهای سطری، هر صفحه دارای 66 سطر است (11 اینچ) و نیز دارای تاریخ و زمانی است که آن فایل تغییر یافته است، شماره صفحه و نام فایل در بالای هر صفحه و تعدادی سطر اضافی به منظور رد کردن تاخوردگی در کاغذ از جمله دیگر امکانات این فایل است. بنابراین junk را به صورت مرتب چاپ کرده و سپس به بالای صفحه جدید رفته و Temp را به صورت مرتب چاپ کنید:

\$ pr junk Temp

Sep 26 16:25 1983 Junk page 1

To be or not To be

(60 more blank lines)

Sep 26 16:26 1983 Temp page 1

That is the question

(60 more blank lines)

\$

pr نیز می تواند خروجیهای چند ستونی بوجود آورد:

\$ pr- 3 filename

هر فایل را در طرحهای سه ستونی چاپ کنید. می توانید هر تعداد مورد قبول را در محل " 3 " به کار ببرید و pr این کار را به بهترین نحو انجام می دهد. کلمه filename مکان مشخصی برای لیست نامهای فایلها می باشد (pr-m مجموعه ای از فایلها را در ستونهای موازی چاپ می کند. (1)pr را مورد مطالعه قرار دهید. این نکته باید مورد توجه قرار گیرد که pr یک برنامه قالب بندی به معنای حقیقی تنظیم سطرها و فاصله بندی حاشیه ها نمی باشد برنامه قالب بندی صحیح nroff و Troff است که در فصل 9 مورد بحث قرار می گیرد. همچنین فرامینی وجود دارد که فایلها را با سرعت بالای چاپگر، چاپ می کند. به فهرست مورد نظرتان تحت نامهایی شبیه Lp 1 , pr نگاه کنید یا " چاپگر " را در فهرست جابجا شده مورد جستجو قرار دهید، آن چیزی که باید مورد استفاده قرار گیرد به تجهیزاتی بستگی دارد که به ماشین شما (سیستم) متصل است. pr او غالباً به صورت توأم مورد استفاده قرار می گیرد پس از اینکه pr اطلاعات را به صورت کامل فرمت کرد، pr 1 مکانیسم گرفتن آنها را از چاپگر خطی، کنترل می کند. در مراحل بعدی به این موضوع می پردازیم.

انتقال، کپی، پاک کردن فایلها *mr, cp, rm*

اجازه دهید به فرامین دیگری نیز پردازیم. اولین موضوع این است که نام فایل را عوض کنیم نامگذاری جدید يك فایل به وسیله " انتقال " آن از يك نام به نام دیگر انجام می پذیرد. مثل این مورد:

```
$ mv junk precious
```

این مورد به این معناست که فایلی که برای نامیدن junk به کار رفته است، اکنون تحت عنوان precious می باشد و محتوای آنها تغییر نکرده است. اگر شما IS را اکنون اجرا کنید، فهرست متفاوتی را خواهید دید: junk دیگر وجود ندارد اما precious موجود می باشد.

```
$ 1s
```

```
Precious
```

```
Temp
```

```
$ cat junk
```

```
cat con,t open junk
```

```
$
```

آگاه باشید که اگر فایلی را به فایل دیگری که از قبل وجود داشته است انتقال دهید فایل مورد نظر، جایگزین می شود. به منظور گرفتن کپی از فایل (یعنی داشتن دو نسخه از چیزی) از فرمان cp استفاده کنید:

```
$ cp precious precious save
```

کپی دوتایی از precious از Percious.save تهیه کنید.

نهایتاً زمانی که ما از ایجاد و انتقال فایلها خسته شدید، فرمان rm، تمام فایلهایی را که شما نام گاری کرده اید را پاک می کند. \$ rm

Temp junk

Rm:junk nonexistent

\$

در صورتیکه یکی از فایل‌هایی که باید پاک شود موجود نباشد، باید اطلاع داشته باشید اما rm مثل اغلب فرامین یونیکس این کار را به آرامی انجام می‌دهد. هیچ دای ناموزونی در این حین به گوش نمی‌رسد و پیغامهای خطا محدود بوده و بعضی اوقات سودند لغی باشند. خلاصه نویسی می‌تواند برای افراد تازه وارد، دردسرساز باشد اما کاربران با تجربه فرامین طویل و پرحرف را آزار دهنده تلقی می‌کنند.

درون نام یک فایل چه چیزی است؟

تاکنون از نام فایلها بدون اینکه بگوییم یک نام قانونی چیست؟ استفاده کرده‌ایم، اکنون زمان مجموعه‌ای از قوانین است اول اینکه نامهای فایلها به 14 کاراکتر محدود می‌باشند. دوم اینکه، اگر چه می‌توانید تقریباً از هر کاراکتری در نام‌گذاری فایل استفاده کنید اما عقل سلیم می‌گوید شما باید به کاراکترهایی متوجه شوید که مشهود هستند و از کاراکترهایی که ممکن است به دیگر معناها به کار روند، بپرهیزید. برای مثال، پیش از این دیدیم که در فرمان $ls -t$ ، $ls -t$ به معنای فهرست فایلها به ترتیب زمانی است. بنابراین اگر شما فایلی دارید که نام آن t باشد در نتیجه برای فهرست آن بر اساس نام وقت زیادی باید صرف کنید. (چگونه باید این کار را انجام دهید) علاوه بر علامت منها (-) به عنوان اولین کاراکتر، کاراکترهای دیگری با معنای خاص وجود دارد. به منظور اجتناب از ایجاد مشکل، باید در استفاده از حروف تنها، اعداد، دوره و زیر خط دار کردن تا زمانیکه با موقعیت آشنایی پیدا کنید، به خوبی و با احتیاط عمل کنید (دوره و زیر خط دار کردن (under score) به صورت قراردادی به منظور تقسیم نام فایلها به بخشهایی همچون Precious. Save مورد استفاده قرار می‌گیرد) نهایتاً فراموش نکنید که هر مورد، موضوع مجزایی به شمار می‌رود، برای مثال Junk, junk, Junk هر کدام سه نام متفاوت می‌باشد.

تعدادی از فرامین مفید

اکنون که شما از اصول ایجاد فایل، فهرست نامهای آنها و چاپ محتویات آنها اطلاع پیدا کردید، می‌توانیم شش مورد از فرامین پردازش فایلها را مورد بررسی قرار دهیم. به منظور اینکه بحث جلدی داشته باشیم. از فایلی استفاده خواهیم کرد که (شعر) poem نام دارد و حاوی شعر مشهوری از Augustus de morgan است. اجازه دهید این فایل را با ed ایجاد کنیم.

\$ ed

a

Great fleas have little fleas
Upon their back to bite em,
And little fleas have lesser fleas,
And so ad infinitum.
And the great fleas themselves , in turn
Have greater fleas to go on

While these again have greater still,
And greauer still , and so on.

W poem

263

q
\$

اولین فرمان سطرها، کلمات و کاراکترها را در یک یا تعداد بیشتری از فایلها می‌شمرد و این فرمان پس از نقش شمارش لغات تحت عنوان wc از آن یاد می‌شود

\$ wc poem

8 46 263 poem

\$

یعنی، شعر 8 سطر، 46 لغت و 263 کاراکتر دارد. تعریف " کلمه " بسیار ساده است هر رشته (زدیفی) از کاراکترها که حاوی جای خالی فاصله یا سطر جدید نباشد کلمه نامیده می‌شود. Wc بیش از یک فایل برای شما می‌شمارد. (و کل آن را چاپ می‌کند) و همچنین هر کدام از شمارشها را در صورتی که شما از آن بخواهید، متوقف می‌سازد. (wc(1 را مورد مطالعه قرار دهید.

دومین فرامین grep نامیده می‌شود و این فرمان فایلها یا سطرهایی که یک الگو را هماهنگ می‌سازد را جستجو می‌کند (نام آن از فرمان ed اقتباس می‌شود که در بخش ضمیمه 1 توضیح داده شده است) فرض کنید شما می‌خواهید به دنبال کلمه " flea " در شعر بگردید:

\$ grep fleas poem

greu flead have little flead
And little fleas have gesser fleas,
Have gret fleas themselves , in turn,
Have greater fleas to go on,
\$

grep به دنبال سطرهایی می‌گردد که با الگو مطابق نیستند، البته زمانیکه انتخاب -v به کار برده شود. (این انتخاب درس از فرمان ویراستاری " V " نامیده می‌شود، می‌توانید در مورد آن به عنوان معکوس موضوع تطابق، فکر کنید).

\$ grep - V fleas poem

upon their backs To bile, em,
and so an infinitum.
While these again have greater still,
And greater still , and so on.
\$

grep می‌تواند برای جستجوی چندین فایل مورد استفاده قرار گیرد، در این صورت، این فرمان نام فایل را در هر سطر که با آن هماهنگ است به پیشوند می‌آورد بنابراین شما می‌توانید تشخیص دهید که این هماهنگی و تطابق در کجا اتفاق افتاده است. همچنین انتخابهای برای شمارش، تعداد و از این قبیل موارد، وجود دارد grep همچنین تعداد زیادی از الگوهای پیچیده را تحت کنترل در می‌آورد تا اینکه تنها یک کلمه مثل " fleas " را کنترل کند اما ما نظرم را در مورد آن تا رسیدن به فصل 4 به تأخیر می‌اندازیم. سومین فرمان sort است که ورودیهای درون خود را به ترتیب الفبا سطر به سطر طبقه‌بندی می‌کند. این موضوع برای شعر چندان جالب نیست، اما اجازه دهید که اینکار را به صورت دیگری انجام دهیم و فقط ببینید که این مورد شبیه به چیست:

\$ sort poem

and greater sTill, and so on.

And so ad infinitum.
 Have greater fleas To go on,
 Upon their backs to bite,em,
 And little fleas have lesser fleas,
 And the greater fleas themselves , in Turn,
 great fleas have little fleas,
 while these again greater still,
 \$

طبقه بندی به صورت سطر به سطر است اما، ترتیب طبقه بندی ناقص باعث می شود که در ابتدا جای خالی قرار گیرد، سپس حروف بزرگ و سپس حروف کوچک، بنابراین این شرایط دقیقاً بر اساس الفبا نمی باشد sort دارای هزاران انتخاب جهت کنترل ترتیب طبقه بندی است که عبارتند از:

ترتیب معکوس، ترتیب عددی، ترتیب لغت نامه ای، نادیده گرفتن فضاها، خالی مهم، طبقه بندی بر اساس فایل های درون سطر و غیره اما معمولاً فرد باید جهت اطمینان از آنها، آن انتخابها را بررسی کند. در اینجا، فهرستی از معمولترین طبقه بندی وجود دارد.

Sort - v

معکوس کردن ترتیب معمولی

Sort - n

طبقه بندی به ترتیب عددی

Sort - nr

طبقه بندی به ترتیب عددی معکوس

Sort-f

Fold upper و حروف کوچک با هم دیگر

n+1-st Sort +n

شروع طبقه بندی در میدان

فصل 4 دارای اطلاعات بیشتری پیرامون Sort است.

فرمان بررسی فایل دیگری نیز وجود دارد که تحت عنوان Tail است که 10 سطر آخر فایل را چاپ می کند. این فرمان برای هشت سطر شعر ما کفایت می کند. اما بریا فایل های بزرگتر مناسب است. علاوه بر این، Tail برای مشخص نمودن تعداد سطرها دارای انتخاب است، بنابراین به منظور چاپ آخرین سطر شعر:

\$ Tail -1 poem
 and greater still, and so on

\$

Tail همچنین می تواند برای چاپ فایل که در یک سطر مشخص شده شروع می شود، به کار برده شوند:

\$ tail +3 filename

چاپ با خط سوم شروع می شود (به معکوس بودن طبیعی علامت منها برای شناسه ها توجه کنید) جفت فرامین نهایی برای مقایسه فایلها در نظر گرفته شده اند. فرض کنید که ما نوعی شعر در فایل New - poem داریم:

\$ cat poem
 Great fleas have little fleas
 upon their back to bite em ,
 And little fleas have lesser fleas,
 and so ad infinitum.
 And the great fleas themselves , in turn,
 Have greater fleas to go on,
 While these again have greater still,
 And greater still , and so on.


```
$ cat Newpoem
Great fleas have little fleas
Upen their back to bite them,
And little fleas have greater fleas,
and so on ad infinitum
And the great fleas themselves, in Turn,
have greater fleas to go on,
while these again have greater still,
and greater still ,and so on.
$
```

تفاوت چندانی بین دو فایل وجود ندارد، در حقیقت، شما برای پیدا کردن آن باید سخت به دنبال آن بگردید، این همان فرامین مقایسه فایلها است که یک فرمان مناسب و کاربردی است. `cmp` اولین جایی را که دو فایل از یکدیگر متفاوت هستند را پیدا می کند:

```
$ cmp poem new-poem
Poem new-poem differ : char 58, line 2
```

این فرمان می گوید که این فایلها در سطر دوم از یکدیگر متفاوتند که این مورد کاملاً صحیح است. اما این فرمان نمی گوید که تفاوت در چیست و علاوه بر این هیچ گونه از تفاوتهایی را که وجود دارد مشخص نمی کند یکی دیگر از فرامین مقایسه فایلها فرمان `diff` است که در مورد تمام سطری که تغییر یافته اند، اضافه شده اند و یا حذف شده اند، گزارش می دهد:

```
$ diff poem new-poem
2c2
< upon their backs to bileem,
...
>upon Their backs to bite them,
4 c4
<and so ad infinitum.
...
>and so on ad infinitum.
$
```

این فرمان می گوید که سطر 2 در اولین فایل (فایل شعر) به سطر 2 فایل (شعر جدید) تغییر یافته است و به همین صورت در مورد سطر چهارم. به طور کلی - `cmp` زمانی استفاده می شود که شما بخواهید مطمئن شوید که دو فایل به طور واقعی دارای یک محتوا هستند. کار این فرمان سریع بوده و در هر نوع فایلی کار می کند نه در متن تنها `diff` زمانی استفاده می شود که فایلها تا حدی متفاوت به نظر برسند و شما بخواهید واقعاً بدانید که کدام سطر متفاوت است. فرمان `diff` تنها در فایلهای متنی اجرا می شود.

خلاصه ای از فرامین فایلهای سیستم

جدول 1-1 خلاصه ای از فرامینی است که ما تاکنون در ارتباط با فایلها از آنها صحبت کرده ایم.

3-1 اطلاعات بیشتری در مورد فایلها فهرست راهنما

سیستم فایل شما را که تحت عنوان `junk` است از فایل فرد دیگری که به همین نام است، متمایز می شناسد. این تمایز به وسیله

گروه‌بندی فایلها به فهرستها، صورت می‌پذیرد که این فرآیند نسبتاً به ترتیبی است که کتابها در کتابخانه‌ها بر روی قفسه‌ها جای داده می‌شوند، بنابراین فایلهایی که در فهرستهای متفاوت هستند می‌توانند بدون هیچ گونه تضادی دارای یک نام باشند. به طور کلی هر کاربر دارای یک فهرست شخصی و خصوصی است، بعضی اوقات تحت عنوان فهرست login از آن یاد می‌شود، که تنها حاوی فایلهایی است که متعلق به آن فرد است. زمانی که شما با سیستم ارتباط برقرار می‌کنید، در واقع شما در فهرست شخصی خودتان هستید.

ممکن است فهرستی را که شما در حال کارکردن در آن هستید را تغییر دهید غالباً فهرست کنونی و آن فهرستی را که در حال فعالیت بر روی آن هستید را مشخص کنید اما در این شرایط فهرست شخصی شما هنوز به همان صورت قبلی باقی است. مگر اینکه شما عمل خاصی را انجام دهید مثلاً زمانی که یک فایل جدید ایجاد کنید در واقع این فایل وارد فهرست کنونی شما می‌شود از آنجایی که این فایل به عنوان فهرست شخصی شما است، در واقع این فایل به فایلی با همان نام که ممکن است در فهرست شخص دیگری باشد، در واقع این فایل به فایلی با همان نام که ممکن است در فهرست شخص دیگری باشد، مربوط نمی‌شود. فهرست می‌تواند حاوی فهرستهای دیگر و نیز افیلهای معمولی باشد (فهرستهای بزرگ دارای فهرستهای کوچکتر هستند).

روش معمول برای به تصویر کشیدن این سازماندهی شبیه به درختی از فهرستها و فایلها است. این امکان وجود دارد که پیرامون هر کدام از این درختها حرکت کرده و هر کدام از این فایلهای درون سیستم را به وسیله شروع مسیر از ریشه درخت و حرکت به دنبال شاخه‌های مناسب، پیدا کنیم. بر عکس شما می‌توانید از جایی که در آن قرار گرفته‌اید شروع کرده و به سمت ریشه حرکت کنید. اجازه دهید که مورد اخیر را اول انجام دهیم. ابزار اصلی ما فرمان `pwd` است (فهرست فعالیت چاپ) که نام فهرستهایی را که شما اکنون در آنها هستید را چاپ می‌کند:

جدول 1-1 فرمانهای رایج فایل سیستم

نام‌های تمام فایل‌های موجود در فهرست (دایرکتوری) را فهرست می‌کند	ls-l
تنها فایل‌های نام‌گذاری شده را فهرست می‌کند	ls filenames
به ترتیب زمانی فهرست می‌کند، آنهایی که اخیراً بوجود آمده در ابتدا قرار گرفته‌اند	ls-t
لیستی طولی و شامل اطلاعات زیاد و نیز ls-lt می باشد	ls-l
بر اساس آخرین زمان مورد استفاده قرار گرفتن فهرست می‌شود و نیز شامل	ls-u
1s-lu, 1s-lut نیز می‌باشد.	
به ترتیب معکوس فهرست می‌شود و نیز شامل rt,-rtl- و غیره می‌باشد	ls-r
فایل‌های نام گذاری شده را ویراستاری می‌کند	ed - filename
File را به 2file کپی کرده و در صورتی که	cp file file2
1 file 2	
file 2 آن را جانویسی می‌کند.	
فایل‌های نام‌گذاری شده را پاک کرده، البته این عمل	rm filename
به صورت تغییر ناپذیر صورت می‌گیرد	
محتوای فایل‌های نام‌گذاری شده را چاپ می‌کند	cat filenames
محتواها را با سرآمد چاپ می‌کند	pr filenames
66 سطر را در هر صفحه	
در ستون‌های n چاپ می‌کند	pr-n filenames
فایل‌های نام‌گذاری شده را در کنار هم چاپ می‌کند	pr-m flie
(ستونهای چندگانه)	
سطرها، لغات و کاراکترها را برای هر فایل می‌شمرد	wc
سطرها را برای هر فایل می‌شمارد	wc-l filenames
سطرهای هماهنگ با الگو با چاپ می‌کند	grep pattern
سطرهایی که هماهنگ با الگو نباشد را چاپ می‌کند	grep-v
فایلها را به صورت الفبایی و به صورت سطر چاپ می‌کند	sort
10 سطر آخر فایل را چاپ می‌کند	tail filename
سطر آخر فایل را چاپ می‌کند	filename Tail - n
چاپ فایل را در سطر n شروع می کند	Tail +n filename

```

cmp file filez
diff file filez

```

محل اولین تمایز را چاپ می‌کند
تمام تفاوت‌های موجود بین فایلها را چاپ می‌کند.
pwd \$
usr/you/
\$

این فرمان می‌گوید که شما در حال حاضر در فهرست خودتان و در دایرکتوری `usr` هستید که به نوبه خود این فهرست در فهرست ریشه قرار دارد، که این فهرست به صورت قراردادی «م» نامیده می‌شود. علامت / اجزاء نام را از یکدیگر جدا می‌کند؛ محدوده 14 کاراکتری که در بالا ذکر شد برای هر جزء که چنین نامی حاوی فهرستهای تمام کاربردن سیستم است. (حتی اگر فهرست اصلی شما به صورت `usr/you/` نباشد، در این صورت `pwd` چیزی شبیه به آن را چاپ خواهد کرد و بنابراین شما باید قادر باشید، آنچه را که در زیر اتفاق می‌افتد، پیگیری کنید). اگر اکنون شما

\$ `1S/usr/you` را تایپ کنید، دقیقاً همان فهرست نامهای فایلی را دریافت می‌کنید که شبیه به آنچه می‌باشد که از `1S` معمولی دریافت کرده‌اید. زمانیکه هیچ شناسه‌ای فراهم نباشد و `1S` محتواهای فهرستهای معمول را فهرست می‌کند، و نام دایرکتوریها را نشان می‌دهد و محتوای آن دایرکتوری را فهرست می‌کند. سپس برای اجرای `$ 1S/usr` تلاش کنید، این فرمان باید یک سری طویلی از نامها را چاپ کند که در بین آنها دایرکتوری متعلق به شما نیز وجود دارد. مرحله بعدی تلاش برای فهرست کردن خود ریشه است. در این وضعیت جوابی شبیه به این نمونه دریافت می‌کنید:

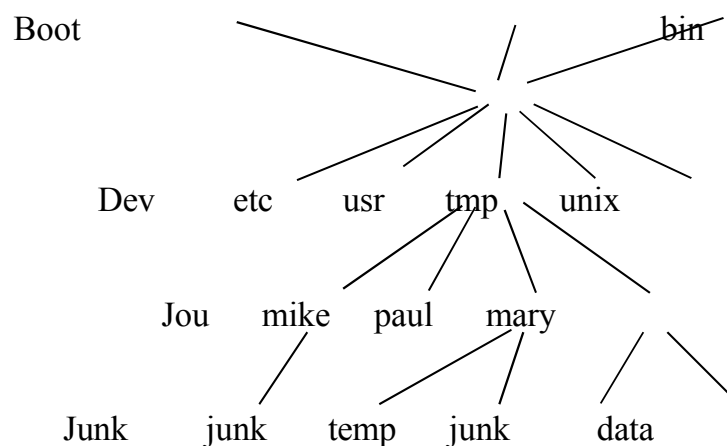
```

/ 1S $
bin
boot
dev
etc
lib
tmp
unix
usr
$

```

(در مورد دو معنای /: دچار سردرگمی نشوید، این دو معنا، نام ریشه بوده و در نامهای فاینها نقش جداکننده دارد). اغلب اینها فهرستها هستند، اما یونیکس فایلی است که حاوی شکل قابل اجراء هسته اصلی یونیکس است. اطلاعات بیشتر پیرامون این موضوع در فصل 2 آورده شده است. اکنون فرمان `$ cat /usr/usr/you/junk` را انجام دهید.
(اگر `junk` هنوز در فهرست شما وجود دارد).

نام `usr/you/junk/` تحت عنوان `path name` (نام مسیر) فایل نام دارد. `Pathnam` دارای معنای مستقیم است، و نام کامل مسیر را از ریشه (در سرتاسر درخت دایرکتوری) به فایل خاص، نشان می‌دهد. این موضوع، قانون کلی در سیستم یونیکس است که شما می‌توانید از نام فایل معمولی یا از نام مسیر استفاده کنید. فایل سیستم شبیه به شجره نام شکل گرفته است: در اینجا تصویری وجود دارد که این موضوع را شفافتر می‌سازد.



فایل شما که `junk` نام دارد به فایل `paul` یا `Mary` مربوط نمی‌باشد. نامهای مسیره‌ها در صورتی موجود نمی‌باشند که تمام فایل‌های مورد توجه در دایرکتوری شما باشند، اما در صورتی که شما با دایرکتوری فرد دیگری یا با چندین طرح به صورت همزمان سروکار داشته باشید، به راستی آنها قابل استفاده می‌شوند. برای مثال دوست شما می‌تواند `junk` شما را با اظهار `$ cat/usr/you/junk` چاپ کند، به طر مشابهی، شما نیز می‌توانید با اظهار

```
$ 1S/usr/mary
data
junk
$
```

متوجه شوید که `Mary` چه فایل‌هایی دارد، یک کپی از فایل‌های او را برای خود تهیه کنید، که این عمل با فرمان `$ cp/usr/mary/data data` انجام می‌شود، یا فایل او را ویراستاری کنید:

`$ ed/usr/mary/data` در صورتی که `Mary` نخواهد شما پیرامون فایل‌های جستجو کنید و یا برعکس، محیط خصوصی می‌تواند در این شرایط تنظیم شود. هر فایل و دایرکتوری اجازه اجراء خواندن - نوشتن برای مالک آن، گروه و هر فرد دیگری را دارد که می‌تواند جهت کنترل دسترسی به آن، مورد استفاده قرار گیرد. (1 - 1S را به خاطر آورید.) در سیستمهای محلی ما اغلب کاربران، اغلب اوقات از محیط باز نسبت به محیط خصوصی، مزایای بیشتری دریافت می‌کنند اما این روش شاید در سیستم شما متفاوت باشد، بنابراین راجع به این موضوع در فصل 2 مجدداً بحث خواهیم کرد. به عنوان آخرین مجموعه تجربیات در مورد ناچه‌های مسیره‌ها، فرمان `$`

1S/bin/usr/bin را اجرا کنید. آیا برخی از نامها آشنا به نظر می‌رسند؟ زمانی که شما یک فرمان را به وسیله تایپ کردن نام آن پس از آماده شدن، اجرا می‌کنید، سیستم برای فایلی که به آن نام است به جستجو می‌پردازد. به طور طبیعی این فایل ابتدا در فهرست (دایرکتوری) شما ظاهر می‌شود، (احتمالاً آن فایل را در کجا نمی‌توان یافت)، بعد از دایرکتوری شما در bin/ و نهایتاً در usr/bin/ یافت می‌شود. هیچ موضوع خاصی پیرامون فرامینی شبیه 1S یا cat وجود ندارد، بر غیر از اینکه، آنها در یک جفت دایرکتوری جمع آوری شده‌اند، تا اینکه جهت فرایند جستجو و اجرا به آسانی پیدا شوند. به منظور اثبات این موضوع، تلاش کنید که برخی از این برنامه‌ها را با استفاده از نامهای مسر کاملشان اجرا کنید:

```
bin/date
Mon sep      26   23 : 29 : 32 EDT 1983
bin/who/ $
Srm   tty/   sep    26      22:20
Crw   ttJ4   sep    26      22:40
you   tty5   sep    26      23:04
$
```

تمرین 3-1: \$ /usr/games/ را اجرا کنید.

و هر آنچه را که به طور طبیعی پیش می‌آید را انجام دهید. این اتفاقات ممکن است خارج از زمان کاری طبیعی، سرگرم کننده‌تر به نظر برسد.

تغییر دادن دایرکتوری - cd

در صورتی که شما به‌صورت منظم با مری (Hary) در مورد اطلاعات موجود در فهرستش در ارتباط باشید، می‌توانید بگوئید «من می‌خواهم به جای کارکردن بر روی فایل خودم، بر روی فایل مری کار کنم». این فرایند با تعویض فهرست خودتان بافرمان cd امکان‌پذیر است: \$ cd/usr/Mary

اکنون زمانیکه شما از یک نام نام (بدون S /) به عنوان شناسه برای برای cat یا pr استفاده کنید، این عمل بر فایلی در فهرست Mary، دلالت دارد. تغییر دایرکتوریها تحت تأثیر هیچ‌گونه اجازه‌مربوط به فایلها نمی‌باشد، در صورتی که شما نتوانید به فایلی از دایرکتوری خودتان دسترسی پیدا کنید، آن را به دایرکتوری دیگری که تغییر نخواهد کرد، تغییر دهید. این موضوع کاملاً مناسب است که فایل‌های شخصی‌تان را منظم کنید، بنابراین تمام فایل‌هایی که به یک چیز مربوط می‌شوند در یک دایرکتوری جداگانه‌ای از دیگر موضوعات قرار می‌گیرد. برای مثال: در صورتیکه شما بخواهید کتابی بنویسید، ممکن است بخواهید تمام متن را در یک دایرکتوری که کتاب نام دارد، حفظ و نگهداری کنید. فرمان mkdir باعث بوجود آمدن دایرکتوری جدید می‌شود.

```
mkdir book $      دایرکتوری جدید بساز
cd book $         رفتن به محل موردنظر
pwd $             از اینکه در محل صحیح قرار گرفته‌اید اطمینان حاصل کنید.
```

usr/you/book/

نوشتن کتاب موردنظر (پس از گذشتن چند دقیقه)

...

به میزان یک سطح بالا رفتن (جابه جا شدن) در فایل سیستم

\$ cd...

\$ pwd

/usr/you

\$

'..' به منشاء همان دایرکتوری که شما معمولاً در آن قرار دارید دلالت می کند و دایرکتوری یک سطح به ریشه نزدیک تر می شود. '0' مترادفی است برای دایرکتوری فعلی، به دایرکتوری اصلی برمی گردد.

cd \$

تمام آنها به همراه خود دایرکتوری شما را به دایرکتوری اصلی برمی گرداند، یعنی همان دایرکتوری که شما با آن مرتبط هستید. زمانیکه کتاب شما چاپ شد، می توانید آن فایلها را پاک کنید. به منظور پاک کردن دایرکتوری کتاب، تمام فایلها موجود در آن را حذف کنید (روش سریعتر و کوتاهتر را به شما خواهیم آموخت)، سپس فرمان cd را در دایرکتوری اصلی کتاب اجرا کرده و \$ rmdir book را تایپ کنید، rmdir تنها دایرکتوری خالی را حذف خواهد کرد.

Shell 1 - 4

زمانیکه سیستم پیغام \$ را چاپ کرد و شما فرامینی را تایپ کردید که اجرا شوند. این هسته اصلی (kernel) نیست که با شما مرتبط می شود بلکه، go-between فرمان تعبیر و تفسیر یل Shell را فراخوانی می کند. Shell تنها یک برنامه معمولی شبیه date یل who است، اگر این برنامه می تواند کارهای قابل توجهی انجام دهد. این حقیقت که Shell بین شما و امکانات هسته اصلی (Kernel) جای گرفته است، واقعیت دارد، ما در مورد برخی از موارد آن در اینجا صحبت خواهیم کرد. سه نکته اصلی وجود دارد:

• مختصر-نویسی نام-فایلها: می توانید مجموعه-کاملی-از-نامهای-فایلها-را-به-عنوان شناسه در برنامه ای به وسیله خصوصی سازی الگویی برای نامها، بدست آورید، Shell نامهای فایلهایی را پیدا می کند که با الگویی شما هماهنگی دارد.

• تغییر مسیر ورودی - خروجی : می توانید خروجی هر برنامه را به جای رفتن به پایانه، طوری تنظیم کنید که به فایل وارد شود و در مورد ورودی طوری ترتیب دهید که ورودی به جای پایانه از-فایل نشأت گیرد. ورودی و-خروجی حتی می توانند به دیگر برنامه ها نیز متصل شوند.

• خصوصی سازی محیط: می توانید فرامین شخصی تا و مختصر نویسی را بدین وسیله تشریح کنید.

مختصر نویسی نام فایل

اجازه دهید با الگوهای نام فایلها شروع کنیم. فرض کنید در حال تایپ کردن سند بزرگی همچون کتاب هستید. از لحاظ منطقی این سند باید به قطعات بسیار کوچک مثل فصل و شاید بخش تقسیم شود. اساساً چنین تقسیم‌بندی باید صورت گیرد، این فرایند به منظور ویراستاری فایل‌های بزرگ امری دشوار است. بنابراین، باید آن سند را به صورت تعداد فایلها تایپ کنید. ممکن است فایلها را برای هر فصل جدا کنید و آنها را فصل 1، فصل 2 و ... نام‌گذاری کنید. یا در صورتیکه هر فصل به بخشهایی تقسیم شود، ممکن است فایل‌هایی ایجاد کنید که

```
فصل 1.1 فصل 2.1
فصل 1.2 فصل 2.1
فصل 1.3 .....
```

نام داشته باشد که این عمل سازماندهی خاصی است که ما برای این کتاب استفاده کرده‌ایم. به کمک یک سیستم نام‌گذاری قراردادی می‌توانید با یک نگاه اجمالی تعیین کنید که یک فایل خاص در کدام محل با کل فایلها تناسب پیدا می‌کند. چه بخشهایی از کل کتاب را می‌خواهید چاپ کنید؟ در این مورد می‌توانید بگوئید :

```
$ ch 1.1 ch 1.2 ch 1.3 pr .....
```

اما به زودی از تایپ کردن نامهای فایلها خسته شده و شروع به غلط نوشتن می‌کنید. این جا همان مرحله‌ایی است که مختصر نویسی نام فایلها وارد عمل می‌شود. در صورتی که بگوئید `*pr ch $`

Shell علامت * رابه معنای 'هر ردیفی از کاراکترها' تلقی می‌کند، بنابراین `*ch` الگویی است که تمام نامهای فایلها را که در دایرکتوری فعلی همراه `ch` هستند را هماهنگ می‌کند. Shell لیستی فراهم می‌آورد که به ترتیب الفبایی است و لیست را به `pr` انتقال می‌دهد. فرمان `pr` هرگز علامت * را نمی‌بیند، الگوهایی که Shell را هماهنگ کرده‌اند در دایرکتوری فعلی لیستی از ردیفهای کاراکتری را بوجود می‌آورند که به `pr` انتقال داده می‌شود. مرحله بحرانی، عبارتست از مرحله‌ایی که مختصر نویسی نام فایلها خصوصیت فرمان `pr` نباشد، اما کار برنامه Shell باشد. بنابراین می‌توانید از آن برای تولید توالی نامهای فایلها برای هر فرمان استفاده کنید. برای مثال، برای شمردن کلمات در اولین فصل:

```
$ wc ch.1.*
113      562      3200      chl.0
935      4081     22435     chl.1
974      4191     22756     chl.2
378      1561     8481      chl.3
1293     5298     28841     chl.4
33       194      2030      chl.5
75       323      2030      chl.6
(88933   16210    3801 (کل  ToTal
```

برنامه‌ای به نام `echo` وجود دارد که برای بررسی معنای کاراکترهای مختصرنویسی بسیار مفید است. همانطور که حدس می‌زنید، `echo` چیزی به غیر از انعکاس شناسه‌هایش انجام نمی‌دهد:

```
$ echo hello world
hello world
$
```


اما شناسه‌ها می‌توانند به وسیله تطابق الگویی، گسترش یابند:

\$ echo ch1*

تمام نامهای، کلیه فایل‌های موجود در فصل 1 را فهرست کنید.

\$ echo *

تمام نامهای فایل‌های موجود در دایرکتوری فعلی را به ترتیب حروف الفبایی فهرست کنید.

\$ pr *

تمام فایل‌های خودتان (به ترتیب حروف الفبایی) را چاپ کنید، و

\$ rm *

تمام فایل‌های موجود در دایرکتوری فعلی خود را پاک کنید. (بهتر است که از آنچه که می‌خواهید بگوئید، مطمئن شوید!) . علامت * به

آخرین وضعیت در نام فایل‌ها محدود نمی‌شود - S * هر جایی می‌تواند باشد و چندین بار نیز می‌تواند، اتفاق افتد. بنابراین \$ rm

save * تمام فایل‌ها را که با save پایان می‌یابد را پاک می‌کند. توجه داشته باشید که نامهای فایل‌ها به ترتیب الفبایی طبقه بندی شده‌اند،

که این طبقه بندی همانند ترتیب عددی نیست. اگر کتابتان دارای 10 فصل است، ترتیب آن ممکن است آنچیزی که مورد نظر شما

است، نباشد، از این رو فصل 10 قبل از فصل 2 قرار می‌گیرد:

\$ * echo

Ch 1.1 ch1.2... ch10.1 ch10.2 ...ch2.1 ch2.2...

\$

علامت * تنها خصوصیت تطابق الگو نیست که به وسیله Shell فراهم می‌شود، اگرچه این علامت به میزان زیادی مورد استفاده قرار

می‌گیرد. الگوی [...] هر کدام اگر کاراکترهای موجود در براکت را هماهنگ می‌سازد. محدوده‌ایی از حروف متوالی یا ارقام می‌توانند به

صورت مختصر نوشته شوند.

\$ pr ch [1 2 3 4 6 7 8 9] *

فصلهای 9 و 8 و 7 و 6 و 4 و 3 و 2 و 1 به غیر از 5 را چاپ کن

* [pr ch [1-46-9 \$

همین عمل را انجام بده

[rm Temp [a-z \$ که Tempz, Temp

هرکدام از

موجود است را پاک کن

الگوی ؟ هرگونه کاراکتر تنها (مجزا) را مطابقت می‌دهد:

? 1S \$

فایلها را با نامهای مجزای کاراکترها فهرست کن

1S-1 ch? .1 \$

فصل 1-1 ، فصل 2-1 ، فصل 3-1 و ...

را فهرست کن اما فصل 1-10 را فهرست نکن

?rm Temp \$

فایلهای Tempa ، Temp 1 ... و غیره را پاک کن

توجه داشته باشید که الگوها تنها نامهای فایل‌های موجود را تطبیق می‌دهند. به ویژه شما نمی‌توانید نامهای فایل‌های جدید را با استفاده از

الگوها، تنظیم کنید. برای مثال، اگر بخواهید ch را هر نام فایل به chapter گسترش دهید، نمی‌توانید این کار را به این نحو انجام دهید:

.mv ch. chapter \$ عملی نیست!

زیرا * chapter هیچ نام فایل موجودی را تطبیق نمی‌دهد کاراکترهای الگو شبیه * می‌توانید در نامهای مسیر و نیز نامهای فایل‌ها ساده

استفاده شوند و این تطابق برای هر جزء مسیر که حاوی کاراکتر خاص باشد صورت می‌پذیرد . بنابراین /usr/Mary/ این تطابق را در

usr/mary/ انجام می‌دهد و usr/*/calendar/ لیست نامهای فایلها تمام کاربران فایلهای calendar را توسعه می‌دهد. در صورتی که بخواهید برای همیشه معنای خاص * ، ؟ و غیره را استفاده نکنید، در این صورت کل شناسه را در علامت نقل قول ، همانند

```
? 1S $
```

قرار دهید. همچنین می‌توانید، کاراکتر خاصی را با backslash در تقدم قرار دهید

```
? \ 1s $
```

(به خاطر داشته باشید، از آنجایی که؟ علامت از بین بردن سطر یا پاک کن نیست، این backslash به وسیله shell تعبیر و تفسیر می‌شود نه بوسیله هسته اصلی «kernd»). نقل قول به طور مفصل در فصل 3، بحث شده است.

تمرین 4-1) تفاوت‌های موجود بین این فرامین چیست؟

```
$ 1S junk          $ echo junk
$ 1S /            $ echo /
$ 1S              $ echo
$ 1S              $ echo *
$ 1S *            $ echo *
```

تغییر جهت ورودی - خروجی

اغلب فرمانهایی که ما تاکنون دیده‌ایم خروجی را در پایان بوجود می‌آورند و برخی نیز مانند ویراستار، ورودی آنها از پایانه نشأت گرفته است. این امر تقریباً کلی است که پایانه بتواند به وسیله فایل برای ورودی و خروجی به صورت دوتایی یا برای هکردام به صورت مجزا جایگزین شود. به عنوان مثال: \$ 1S لیستی از نامهای فایلها بر روی پایانه شما ایجاد می‌کند. اما در صورتی که بگوئید

```
$ 1S > file list
```

چنین لیست یکسانی از نامهای فایلها در عوض در فایل filelist ، قرار خواهد گرفت. علامت < به معنای این است که:

«خروجی را در فایل بعدی قرار دهید به جای اینکه در پایانه قرار دهید»

فایل در صورتی ایجاد خواهد شد که پیش از این وجود نداشته باشد یا محتوای قبل در صورتی که آن فایل وجود داشته باشد، روی هم نوشته شده باشد.

هیچ چیزی بر روی پایانه شما ایجاد نمی‌شود. به عنوان مثال دیگر، می‌توانید چندین فایل

را در يك فایل به وسیله اشغال (ذخیره سازی) خروجی cat در يك فایل ترکیب کنید:

```
cat f1 f2 f3 > temp $
```

علامت << بیشتر کاری شبیه < را انجام می‌دهد، به غیر از این مورد این علامت به معنای «افزودن به قسمت انتهای فایل» نیز می‌باشد. یعنی:

```
$ cat f1f2f3 >> temp
```

محتوای f1 f2 f3 را در پایان هرآنچه که از قبل در Temp وجود داشته، به جای نوشتن بر روی محتواهای موجود، کپی می‌کند. در مورد < (در صورتی که Temp وجود نداشته باشد) در ابتدا باعث بوجود آوردن يك فایل خالی برای شما می‌شود. به روشی مشابه ، علامت > به معنای گرفتن ورودی از پایانه گرفته شود.

بنابراین، می‌توانید نامهای را در فایل let آماده کرده و سپس آن را به وسیله \$ mail mary joe Tom bob<let بفردین نفر بفرستید. در تمام این مثالها، جاهای خالی در هر طرف از <or> اختیاری می‌باشد، اما قالب بندی ما، قدیمی است. با نشان دادن قابلیت تغییر مسیر خروجی با <، باعث می‌شود که این فرایند برای ترکیب فرمانهای به منظور تأثیر گذاری که به هیچ نحو دیگر امکان پذیر نیست، ممکن و میسر شود. برای مثال، به منظور چاپ فهرست الفبایی از کاربران،

```
$ who > temp
```

```
$ sort < temp
```

از این رو who یک سطر از خروجی را در هر ارتباط کاربر چاپ می‌کند و wc-1 سطرها را می‌شمارد (شمارش کلمات و کاراکترها را متوقف سازید) به این ترتیب می‌توانید کاربران را با

```
$ who > temp
```

```
$ wc - 1 < temp
```

شمارش کنید. می‌توانید فایل‌های موجود در دایرکتوری فعلی را با

```
$ Is > temp
```

```
$ wc - 1 < temp
```

با این وجود این فایل شامل نام فایل temp است که خودش هم در شمارش محسوب می‌شود. می‌توانید نامهای فایلها را در ستونهای درختی شکل با

```
$ IS > temp
```

```
$ pr - 3 < temp
```

چاپ کنید. و می‌توانید ببینید که کاربر ویژه‌ایی به وسیله ترکیب who و grep ارتباط برقرار کرده است

```
$ who > temp
```

```
$ grep mary < temp
```

در تمام این مثالها، کاراکترهای الگوی نام فایل مثل *، این نکته مهمی است که به خاطر آورید، تعبیر و تعریف علامت < و > به وسیله shell صورت می‌پذیرد نه به وسیله برنامه‌های خاص. تمرکز بخشی امکانات در-shell به این معناست که تغییر جهت ورودی و خروجی می‌تواند با هر برنامه ای مورد استفاده قرار گیرد؛ خود برنامه متوجه نیست که اتفاقی در حال رخ دادن است.

این امر باعث می‌شود قانون مهمی مطرح شود. فرمان \$ sort < temp محتوای فایل temp را طبقه‌بندی می‌کند، همانگونه که \$

sort temp این کار را انجام می‌دهد، اما تفاوتی در این فرایند وجود دارد. از آنجایی که رشته <temp> به وسیله shell تفسیر می‌شود،

sort نام فایل temp را به عنوان یک شناسنامه، تشخیص نمی‌دهد و آن در عوض ورودی استاندارد خودش را طبقه‌بندی می‌کند، به

نحوی که shell دارای برنامه تغییر مسیر است، بنابراین این فرایند نشأت گرفته از فایل است. آخرین نمونه نام temp را به عنوان

شناسه به sort انتقال می‌دهد که باعث خواندن فایل و طبقه‌بندی آن می‌شود. Sort می‌تواند فهرستی از نامهای فایلها را مثل \$ sort

temp1 temp2 temp3 ارائه دهد اما، اگر هیچ نام فایلی ارائه نشود، این برنامه ورودیهای استاندارد خودش را طبقه‌بندی می‌کند.

این فرایند خصوصیت ضروری اغلب فرمانها است: در صورتی که هیچ نام فایلی مشخص نشود، ورودی استاندارد، پردازش می‌شود.

این امر به این معناست که شما می‌توانید به سادگی برای اینکه متوجه شوید آنها چگونه کار می‌کنند، در فرمانها تایپ کنید. برای مثال

```
sort $
```

```
ghi
```

```
abc
```

```
CTL -d
```

```
Abc
```

Def
ghi
\$

در بخش بعدی، ما خواهیم دید که این اصل چگونه مورد استفاده قرار می‌گیرد.

تمرین 6-1، خروجی از `$ Temp > Temp WC` را شرح دهید.

در صورتی که نام فرمان را غلط نوشته باشید، همانند `$ Temp > who` چه اتفاقی می‌افتد؟

تمام مثالها در پایان بخش قبلی بر روی یک راهکار تکیه دارد: خروجی یک برنامه را در ورودی برنامه دیگری از طریق یک فایل موقتی قرار دهید. اما فایل موقتی هیچ هدف دیگری ندارد، به راستی این فایل به منظور مورد استفاده قرار گرفتن چنین فایلی نامناسب است. این نتیجه منجر به یکی از بخشهای اصلی سیستم یونیکس که همان لوله است می‌شود. لوله روشی است برای اتصال خروجی یک برنامه به ورودی برنامه دیگر بدون هیچ فایل موقتی؛ خط لوله عبارتست از اتصال دو یا تعداد بیشتر برنامه‌ها از طریق لوله‌ها. اجازه دهید برخی از مثالهای پیشین را به منظور استفاده از لوله‌ها به جای فایل‌های موقتی، مورد بررسی مجدد قرار دهیم. کاراکترهای ستونهای عمودی به `Shell` اعلام می‌کند که خط لوله را تنظیم کند:

لیست فهرست بندی شده کاربران را چاپ کنید. `who : srt $`

کاربران را شمارش کنید `who : wc-1 $`

لیست 3 ستونی از نامهای فایلها `1S : wc-1 $`

به دنبال کاربر ویژه جستجو کردن `who : grep mary $`

هر برنامه‌ای که از پایانه خوانده شود می‌تواند به جای آن از لوله نیز خوانده شود و هر برنامه‌ای که در پایانه نوشته شود می‌تواند در لوله نیز نوشته شود. این نقطه جایی است که قرارداد خوانده عبارتست از ورودی استاندارد، البته زمانی که هیچ فایل نامگذاری شده سودمند نباشد.

هر برنامه‌ای که بر این قرارداد پیوندد می‌تواند در خطوط لوله مورد استفاده قرار گیرد. `Grep, pr, sort` و `wc` همگی از روشی استفاده می‌کنند که در خط لوله ذکر شده می‌توانید تعداد بسیاری از برنامه‌های موجود در خط لوله را داشته باشید:

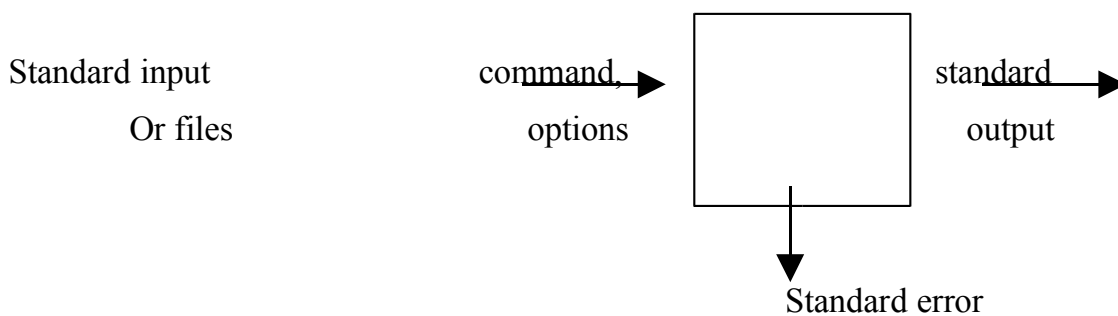
`$ |S : pr-3: | pr`

این فرمان لیست سه ستونی از نامهای فایلها بر روی مطر چاپگر ایجاد می‌کند و

`$ who : grep mary : we -1`

تعداد دفعاتی که `Mary` ارتباط برقرار کرده است را می‌شمرد. برنامه‌ها در خط لوله به همان تعداد می‌رسند، البته نه یکی پس از دیگری. این فرایند به این معناست که برنامه‌ها در خط لوله می‌توانند دوسویه عمل کنند، هسته اصلی (Kernel) هرآنچه که برنامه‌ریزی و هماهنگ سازی به منظور آماده کردن آن برای انجام تمام کارها، لازم دارند را مورد تعقیب قرار می‌دهد. زمانیکه شما دچار حدس و گمان شوید، `Shell` کارها را هنگامی که شما درخواست لوله کنید، تنظیم می‌کند؛ برنامه‌های خاصی وجود دارند که از آن بی‌اطلاع بوده و به منظور تغییر مسیر به کار می‌روند. البته برنامه‌ها در صورتی که به این ترتیب ترکیب شوند، به صورت معقولانه عمل خواهند نمود. اغلب فرمانها طرح معمولی و عادی را دنبال می‌کنند، بنابراین آنها به طرز مناسبی در خطوط لوله و در هر موقعیتی، تطبیق می‌یابند. به طور طبیعی، استمداد و کمک گرفتن از فرمانی به این صورت می‌باشد:

Command optional - arguments - Tialal - filenames در صورتی که هیچ نام فایلی ارائه نشود، فرمان ورودی استاندارد خودش را که ناشی از نقص پایانه است (جهت آزمایش مناسب است) را میخواند اما، این موضوع میتواند به منظور ناشی شدن از یک فایل یا لوله تغییر جهت دهد. در همین زمان، در قسمت خروجی، اغلب فرمانها خروجیشان را بر روی یک خروجی استاندارد، مینویسند، که این جریان همراه با نقصی به پایانه انتقال مینماید. اما این خروجی میتواند در یک فایل یا لوله تغییر جهت پیدا کند. پیغامهای خطا از ناحیه فرمانها به طر متفاوتی کنترل میشوند، و یا ممکن است آنها در یک فایل را در قسمت پایین لوله ناپدید شوند. بنابراین هر فرمان دارای یک خروجی خطای استاندارد است که به طور طبیعی به سمت پایانه شما جهت گرفته است. یا به صورت تصویری عبارتست از:



تقریباً تمام فرمانهایی که ما تاکنون در مورد آنها صحبت کردیم با این مدل سازگار میباشند؛ تنها استثناء در فرمانهایی همچون date و who است که هیچ ورودی را نمیخوانند، و به میزان کمی شبیهه cmp و diff است که دارد تعداد ثابتی از ورودیهای فایل هستند. (اما به انتخاب '۱' در این موارد توجه کنید).

تمرین 7-1) تفاوت بین
 \$ who / sort
 \$ who > sort

را توضیح دهید.

پردازش

Shell علاوه بر تنظیم لولهها چند کارایی دیگر نیز دارد. اجازه دهید به طور خلاصه بیشتر به جای پرداختن به برنامهها به نوبت به اصول اجرای آنها پردازیم، چرا که ما پیش از این

تأحدودي به این موضوع در ارتباط با لوله‌ها پرداختیم برای مثال، می‌توانید دو برنامه را با يك فرمان به وسیله جدا کردن فرمانها با استفاده از نقطه ویرگول (؛) اجرا کنید، Shell نقطه ویرگول (؛) را تشخیص داده و خط فرمان را به دو فرمان مجزا می‌شکند:

```
date ; who $
```

```
Tue  sep  27  01:03:17  EDT  1983
    Ten      TTJo      sep  27      00:43
    Dmr      Tly 1      sep  26      23:45
    Rob      TTJ2      sep  26      23:59
    Bwk      TTJ3      sep  27      00:06
    Jj       TTJ4      sep  26      23:31
    You      Tty5      sep  26      23:04
    Ber      TTJ7      sep  26      23:34
                                                $
```

هر دو فرمانها (به ترتیب) قبل از اینکه sell با يك کاراکتر پیغامی برگردد، اجرا می‌شوند. شما نیز می‌توانید به صورت همزمان در صورت تمایل بیش از يك برنامه اجرا کنید. برای مثال: فرض کنید می‌خواهید کار وقت‌گیری مثل شمردن لغات کتابتان را انجام دهید، اما نمی‌خواهید که صبر کنید تا WC قبل از اینکه شما مشغول کار دیگری شوید، تمام شود. در این صورت می‌توانید بگوئید:

```
& WC ch * > . out $
```

```
6944
```

فرایند id به وسیله shell چاپ می‌شود.

```
$
```

آمپرساند & در پایان خط فرمان به shell می‌گوید: «اجرای این فرمان را شروع کن، سپس فرمانهای بیشتری را از پایانه سریعاً بگیر»، یعنی منتظر تکمیل آن نشو. بنابراین، فرمان شروع خواهد شد، اما شما درحالی‌که این فرمان در حال اجرا است می‌توانید کار دیگری انجام دهید. جهت‌گیری خروجی به فایل WC.out آن را از مداخله با هر آنچه که شما در همان فرمان در حال انجام آن هستید، حفظ می‌کند. نمونه‌ای از اجرای برنامه، پردازش نامیده می‌شود. اعدادی که به وسیله shell برای فرمانها چاپ می‌شود، با & آغاز شد و

تحت عنوان process-id نام گرفت؛ می‌توانید آن در فرمانهای دیگری برای رجوع به برنامه در حال اجرای خاص دیگری استفاده کنید. این نکته مهمی است که بین برنامه‌ها و پردازش‌ها تمایز قائل شویم. WC یک برنامه است، هر بار که شما برنامه WC را اجرا کنید، این برنامه پردازش جدیدی ایجاد می‌کند. اگر چندین نمونه از برنامه‌های یکسان در یک زمان در حال اجرا باشند، هر کدام با process-id متفاوتی به صورت جداگانه پردازش می‌شوند. در صورتی که خط لوله با & شروع شود، مثل

```
& pr ch* |pr $
precess-id of |pr /695
$
```

پردازش در آن، در یک زمان شروع می‌شود. & در تمام خطوط لوله بکار برده می‌شود. تنها یک process-id پردازشهای نهایی در توالیها، چاپ می‌شود. فرمان - wait تا زمانی صبر می‌کند که تمام پردازشهای آغاز شده با & پایان پذیرد. در صورتی که این پردازش سریعاً باز نگردد، به این معناست که شما هنوز فرمان در حال اجرا دارید. می‌توانید فرمان wait را با delete قطع کنید. می‌توانید از process-id که بوسیله shell چاپ می‌شود برای متوقف کردن فرایند پردازش آغاز شده با & ، استفاده کنید:

```
kill 6944 $
```

در صورتیکه شما process-id را فراموش کنید، می‌توانید از فرمان ps برای اطلاع یافتن پیرامون هر آنچه که شما در نظر دارید اجرا کنید، استفاده کنید. اگر کار شما بی‌نتیجه بود، kill O تمام پردازشهای شما را به غیر از ارتباط شما با shell را از بین می‌برد. اگر شما در مورد آنچه که دیگر کاربران انجام می‌دهند کنجکاو هستید، ps-ag به شما در مورد تمام پردازشهایی که در حال اجرا است اطلاعاتی می‌دهد. در اینجا نمونه‌هایی از خروجیها آورده شده است:

```
ps - ag $
```

PID	Tty	TIME	CMD
Co	6:29		/etc/Cron 36
		5	6423
			Sh - 0:02
Sh -	0:04	1	6704

	vi paper	0:12	1	6722
	Sh -	0:03	2	4430
	fi paper	0:12	1	6722
	Sh-	0:03	2	4430
	Sh-	0:03	7	6612
	rogue	1:13	7	6628
	write dmr	0:02	2	6843
	login birnmler	0:01	4	6949
pr	ch1.1 ch 1.2 ch 1.3 ch1.4	0:08	5	6952
	1pr	0:03	5	6951
	ps – ag	0:02	5	6950
	write reb	0:02	1	6844
				\$

PID عبارتست از TTJ ، process-id پایانه‌ای همراه با پردازش است (همانند TIME ; who) عبارتست از پردازشگر زمان که در دقایق و ثانیه ها به کار برده می‌شود، و مابقی فرمانی است که باید اجراء شود. PS یکی از آن فرمانهایی است که در انواع متفاوت سیستمها ، متفاوت است، بنابراین خروجی شما ممکن نیست که مانند این مورد قالب بندی شود. در صورتیکه شناسه‌ها ممکن است متفاوت باشند (در این مورد به صفحه فهرست (1) Ps مراجعه کنید). پردازشها دارای يك نوع طبقه بندی ساختار سلسله مانند هستند که فایلها نیز دارند. هر پردازش دارای مبداء بود و شاید دارای انشعابات نیز باشد. برنامه Shell سیستم شما با پردازشی همراه با هر آنچه که شما به خط پایانه سیستم متصل کرده‌اید، ایجاد می‌شود. همان زمانی که شما فرمانها را اجراء می‌کنید، آن پردازشها انشعابات برنامه Shell سیستم شما را کنترل و هدایت می‌کنند. در صورتیکه شما یکی از برنامه‌های موجود در یکی از آن پردازشها را اجراء کنید، برای مثال فرمان 1 به منظور رهایی از ed،

در این صورت آن پردازش پردازش انشعاب متعلق به خودش که انشعاب اصلی Shell است را ایجاد می‌کند. بعضی اوقات پردازش تا زمانی طول می‌کشد که شما بخواهید اجراء برنامه‌ای را شروع کنید و سپس پایانه را خاموش کرده و بدون منتظر ماندن برای پایان

پذیری آن به خانه بروید.

اما اگر شما پایانه‌تان را خاموش کنید یا ارتباط خود را قطع کنید، پردازش به طور طبیعی حتی اگر شما از- & استفاده کنید، از بین می‌رود. فرمان `nohup` (قطع نکردن) به منظور ارتباط با این موقعیت ایجاد شده است؛ در صورتی شما اعلام کنید `& nohup command` ، فرمان حتی اگر شما ارتباط خود را قطع کنید، تا زمان اجراء ادامه پیدا می‌کند. هر گونه خروجی از فرمان در فایلی که `nohup.out` نامیده می‌شود، ذخیره می‌شود. هیچ روشی برای `nohup` به منظور برگشت پذیری فرمان، وجود ندارد. در صورتیکه پردازش شما تعداد زیادی از پردازشگرهای منابع اصلی را شامل شود، در واقع این شرایط لطفی است در حق کسانی که با سیستم شما برای به جریان انداختن شغلتان با هزینه کمتری نسبت به قبل، اشریک هستند، این فرآیند به وسیله برنامه دیگری که `nice` نامیده می‌شود، انجام می‌شود؛- `& nice expencive command` به طرز خودکار `nice` نامیده می‌شود، چرا که اگر شما در نظر داشته باشید که ارتباط خود را قطع نمائید، می‌توانید برای داشتن فرمانی که کمی بیشتر طول بکشد، از این فرمان استفاده کنید، نهایتاً ، می‌توانید به وضوح به سیستم بگوئید که پردازش را زمان صبح یعنی که افراد طبیعتاً در خواب به سر می‌برند، نه در محاسبه کردن، شروع کند، دین فرمان `(at 1)` نام دارد.

`at tme $`

`Whaterer commands`

`Jeu want`

`Cti-d`

`$`

این مورد کاربرد رایج است اما مسلماً فرمانها می‌توانند از فایل نشأت بگیرند:

`at 3am < fle $`

`$`

زمانها می‌توانند به سبک- 24 ساعته نوشته شود، شبیه- 2130 و یا به سبک- 12 ساعته . 630Pm

مناسب کردن محیط

یکی از مزایای سیستم یونیکس این است که چندی روش برای نزدیک کردن آن به علاقه شخصی شما یا قراردادهای محیط محاسبه محلی شما، دارد. برای مثال ما قبل از این مسئله استانداردهای متفاوت را برای پاک کن یا کاراکترهای از بین برنده سطرها که با توجه به نقص عبارتند از `و @` ، ذکر کردیم. شما می‌توانید هر زمان که خواستید این مورد را با فرمان

```
STTJ erase , Kill K $
```

عوض کنید ، (تغییر دهید). در این فرمان `e` شامل هر کاراکتری است که شما برای پاک کن در نظر گرفته‌اید و `K` برای کاراکتر از بین برنده سطرها در نظر گرفته شده است. اما تایپ نمودن این فرمان هر زمان که با سیستم مرتبط هستید، دردسرساز است. Shell در این زمینه باعث رهایی از این دردسر می‌شود. اگر فایلی تحت عنوان `profile` در دایرکتوری ارتباط شما وجود داشته باشد، Shell قبل از چاپ اولین پیغام فرمانها را در آن، هنگامی که شما ارتباط برقرار کرده‌اید، اجراء می‌کند. بنابراین، شما می‌توانید فرمانها را در `profile` به منظور تنظیم محیط پیرامونتان به همان صورت که دوست دارید، قرار دهید و آنها (فرمانها) هر زمان که شما ارتباط برقرار می‌کنید، اجراء خواهند شد.

اولین چیزی که اغلب افراد در `profile` خود قرار می‌دهند، عبارتست از:

```
STTJ erase
```

در اینجا ما از `^` استفاده می‌کنیم. بنابراین می‌توانید آن را ببینید اما می‌توانید کلید برگشت حروف به عقب (backspace) را در `profile` خود قرار دهید. `STTy` نیز `x` را برای `CTL - x` می‌فهمد، بنابراین، می‌توانید همان اثر را با `erase /h STTy` دریافت کنید. چرا که `CTL-h` همان backspace است. (کاراکتر `^` به عنوان یک هم معنی قدیمی برای اپراتور لوله است،

بنابراین شما باید از آن با علامت نقل قول محافظت کنید). در صورتیکه پایانه شما نقطه Tab حساس نداشته باشد ، می‌توانید `Tabs` را به خط `STTy` اضافه کنید:

```
STTy erase ^h - Tabs
```

اگر علاقه‌مند هستید که بفهمید اشغال بودن سیستم هنگامی که با آن مرتبط هستید چگونه است، `1 - who/wc` را اضافه کنید، تا اینکه این فرمان تعداد کاربران را بشمارد. در صورتیکه سرویس خبری وجود داشته باشد، می‌توانید `news` را اضافه کنید. بعضی از

مردم، آدمه‌های خوش شانس را دوست دارند، بنابراین `usr/games/fortune/` بعد از مدتی ممکن است متوجه شوید که ارتباط برقرار کردن شما، زمان زیادی وقت برده است (بیشتر طول کشیده است). پس به این منظور `profile` خود را قطع کرده و در صورت لزوم دوباره ارتباط برقرار کنید. برخی از خصوصیات `Shell` بوسیله متغیرهای `Shell` کنترل می‌شوند، البته با معیارهایی که شما بتوانید به آنها دسترسی پیدا کرده و سیستم خود را تنظیم کنید.

برای مثال، رشته پیغامهایی که ما به صورت `$` نشان داده‌ایم، در متغیرهای `Shell` که `ps1` نامیده می‌شوند ذخیره می‌شوند و شما می‌توانید آن را در هر جایی که دوست دارید تنظیم کنید، مثل این مورد:

```
?Ps1= 'yes dear
```

علامت نقل قول لازم است چرا که فضاهایی در رشته پیغامها وجود دارد.

فضاها پیرامون `=` در این ساختار، مجاز نمی‌باشد. `Shell` همچنین متغیرهای `Hmoe` و `MaIL` را مورد بررسی قرار می‌دهد. `HOME` نام دایرکتوری اصلی شما است و این دایرکتوری به طور طبیعی بدون مجبور بودن به قرار گرفتن در `profile` تنظیم می‌شود. متغیرها `MaIL`، نام فایل‌های استاندارد است یعنی جایی که نام‌های شما حفظ می‌شود. در صورتی که شما آن را برای `Shell` تعریف کنید، بعد از هر فرمان در صورتیکه نامه جدیدی برسد به شما اطلاع خواهد داد.

احتمالاً مفیدترین متغیرهای `Shell` عبارتند از متغیرهایی که جاهایی که `Shell` به دنبال فرمانها می‌گردد را کنترل می‌کنند. به خاطر داشته باشید که زمانیکه شما نام فرمان را تایپ می‌کنید، `Shell` به طور طبیعتی اول در دایرکتوری کنونی به دنبال آن می‌گردد و سپس در `bin/` و بعد از آن در `usr/bin/`. این توالی فهرستها «مسیر جستجو» نامیده می‌شوند و در متغیر `Shell` که `PATH` نامیده می‌شود، ذخیره می‌شود. اگر مسیر جستجو آن مسیری نباشد که شما می‌خواهید، می‌توانید آن را تغییر دهید، و معمولاً در `profile`.

1- این عمل در `Shell` به نحو بدی انجام می‌گیرد. پس از اینکه هر فرمان به صورت به بار سیستم افزوده شد، فایل را مورد بررسی قرار دهید. همچنین، در صورتیکه شما برای مدت طولانی در حال کار کردن بر روی برنامه ویراستاری باشید اطلاعاتی در مورد نامه جدید فرا نخواهید گرفت چرا که شما فرمانهای جدید را با ارتباط برقرار کردن با `Shell` اجراء نکرده‌اید. طرح و راهکار بهتر این است که هر چند دقیقه یکبار به جای بعد از هر فرمان، اوضاع را مورد بررسی قرار دهید. فصل 7 و 5 نشان می‌دهد که این نوع از مرورگرهای نامه چگونه عمل می‌کنند. سومین امکان که برای هر فردی موجود نمی‌باشد، این است که تنها برنامه پست الکترونیکی خود شما را مطلع بسازد: این شرایط مطمئناً زمانی میسر می‌شود که نامه تنها برای شما فرستاده شده باشد.

سیستم‌تان این عمل انجام می‌گیرد. برای مثال: این سطر مسیر را به سمت يك مسیر استاندارد به علاوه /usr/games تنظیم می‌کند.

```
one way : /usr/games : /usr/bin : /bin : ..PATH:
```

این فرآیند کمی عجیب به نظر می‌رسد: توالی نام دایرکتوریها به وسیله : از یکدیگر جدا می‌شوند. به خاطر داشته باشید که ' ' همان دایرکتوری فعلی است. شما می‌توانید ' ' را حذف کنید و جزء صفر در PATH به معنای دایرکتوری فعلی است. راه دیگر برای تنظیم مسیر (PATH) در این مورد خاص عبارتست از افزایش دادن معیار قبلی (PATH= \$ PATH: /usr) (/games Another way) می‌توانید مقدار هرگونه متغیرهای Shell را به وسیله پیشوند کردن نام آن با \$ ، بدست آورید. در مثال بالا، حالت \$ PATH مقادیر فعلی را اصلاح می‌کند، به نحوی که بخش جدید، افزوده می‌شود و نتایج به PATH برگردانده می‌شود. شما می‌توانید این فرآیند را با echo اثبات کنید:

```
echo PATH is $ PATH $
```

```
PATH is :/bin:/usr/bin:/usr/games
```

```
echo $ HOME $
```

```
usr/you
```

```
your login directory/
```

```
$
```

اگر شما برخی از فرمانهای مربوط به خودتان را در اختیار داشته باشید، ممکن است بخواهید آنها را در دایرکتوری مربوط به خودتان جمع‌آوری کرده و آنها را به مسیر جستجو شخصی‌تان اضافه نمایید. در این صورت، PATH شما ممکن است چنین به نظر برسد:

```
PATH= : $ HOME /bin:/bin:/usr/bin:/usr/games
```

ما در مورد نوشتن فرمانهای شخصی‌تان در فصل 3 خواهیم کرد. متغیر دیگر که غالباً به وسیله تزئینات ویراستاری متن استفاده می‌شود تا به وسیله ed ، عبارتست از TERM که نام نوعی از پایانه‌هایی است که شما از آن استفاده می‌کنید. چنین اطلاعاتی ممکن است این امکان را برای برنامه‌ها فراهم آورد که صفحه نمایش شما را به میزان کارآمدتری کنترل کند. بنابراین ممکن است چیزی شبیه به TERM = a dm3 را به فایل profile خود اضافه نمایید. همچنین این امکان وجود دارد که از متغیرها برای اختصار نویسی استفاده شود. اگر مکرراً به برخی از دایرکتوریها با نامهای طولانی مراجعه داشته

باشید، این امر شایسته است که سطرهای همچون

```
d= /harribly / lony / dinectory / name
```

به profile خود اضافه نمائید، بنابراین می‌توانید موردی شبیه \$ cd \$ d را داشته باشید. متغیرهای شخصی همچون d به منظور متمایز نمودن آنها از مواردی که با خود Shell مورد استفاده قرار می‌گیرند همچون PATH، به صورت قراردادی با حروف کوچک نوشته می‌شوند. نهایتاً، لازم است به Shell اعلام کنید که در نظر دارید از متغیرها در دیگر برنامه‌ها استفاده کنید، این عمل با فرمان export انجام می‌پذیرد، که در مورد این موضوع در فصل 3 بحث خواهیم کرد.

```
Export MAIL PATH TERM
```

به منظور خلاصه نمودن، در این مرحله فایل profile، معمولی وجود دارد که شبیه به این مورد است:

```
$ cat . profile
STLy erase ^h - Tabs
MAIL = /usr/ spool/ mail / you
PATH= $ HOME /bin : /bin: / usr/ bin:/usr/ games
B=$ HOME / book
Export MAIL PATH TERM b
Date
Who / WC-1
$
```

به هیچ وجه ما خدماتی را که Shell فراهم آورده است را نادیده نمی‌گیریم. یکی از مفیدترین خدمات این است که شما می‌توانید فرمانهای شخصی خودتان را به وسیله بسته بندی فرمانهای موجود در یک فایل به منظور پردازش شدن بوسیله shell ایجاد کنید. این نکته قابل توجه است که تا چه اندازه این امر می‌تواند به وسیله این مکانیسم ساده به نتیجه برسد. بحث ما در این مورد از فصل 3 شروع می‌شود.

5-1 مابقی اطلاعات سیستم یونیکس

نکات بیشتری در مورد سیستم یونیکس در مقایسه با آنچه که ما در این فصل ارائه کردیم وجود دارد، اما اطلاعات بیشتر در این کل این کتاب مطرح شده است. اکنون، باید در مورد این سیستم احساس راحتی کرده باشید به ویژه در مورد کتاب راهنما. زمانیکه شما سوالات خاصی در مورد زمان و چگونگی استفاده فرمانها دارید، کتاب راهنما جایگاه بررسی پاسخ شماست. همچنین این نکته مفید و باارزشی است که گاهی در کتاب راهنما جستجو صورت گیرد، به این منظور که دانش شما از آشنایی با فرمانها و کشف اطلاعات جدید، به روز شود. کتاب راهنما بسیاری از برنامه‌هایی که ما نباید توضیح دهیم و عبارتند از گردآورندگان برای زبانهایی همچون FORTRAN 71، برنامه‌های ماشین حساب همچون cu(1)، bc(1) و uncp(1) برای ارتباطات بین ماشینی، بسته‌بندیهای طرحها، برنامه‌های آماری و رمزهایی همچون units(1)، را تشریح می‌کند. همانگونه که پیش از این نیز گفته شد این کتاب جایگزین برای راهنما نمی‌باشد بلکه آن را تکمیل می‌کند. در فصلهایی که در پیش است، ما به بخشها و برنامه‌های سیستم یونیکس اشاره خواهیم کرد که این فرایند از اطلاعاتی در راهنما آغاز می‌شود اما رشته‌هایی را پی گیری می‌کند که پیوند دهنده اجزاء و بخشهای کتاب است. اگرچه ارتباطات برنامه‌ها هرگز در کتاب راهنما به وضوح مشخص نشده است اما آنها اصل محیط برنامه سازی یونیکس را تشکیل می‌دهند.

تاریخچه و نکات مربوط به تألیفات

مقاله اصلی یونیکس توسط K.L.thompson , D.M.Ritchie تهیه شده است. اصل سیستم اشتراک زمانی یونیکس تحت عنوان ارتباطات-ACM در جولای سال-1974 تهیه شد و در-CACM و در ماه ژانویه-1983 دوباره چاپ شد. (صفحه-89 چاپ جدید موضوع بحث ماه مارس سال 1983 قرار گرفت). این بررسی کلی از سیستم برای افراد علاقه مند به سیستم عامل جهت خواندن توسط هر فردی که برنامه نویس است، ارزشمند است. مجله تکنیکی سیستم (Bell (BSTJ که موضوع ویژه‌ای پیرامون سیستم یونیکس است (جولای 1978) حاوی مقالات بسیاری است که تشریح کنند تحول بعدی است و برخی از موضوعات گذشته شامل مقالات جدید مجله CA CM است که توسط Ritchie و thompson تهیه شده است. دومین موضوع خاص RSTJ حاوی مقالات جدید سیستم یونیکس است که به منظور چاپ شدن در سال 1984 طرح ریزی شده است. محیط برنامه یونیکس که توسط J.R.Mashey , B.W.kernighan تنظیم شده (مجله کامپیوتر IEEE . ماه آوریل سال 1981) جهت انتقال خصوصیات اصلی سیستم به برنامه سازان تلاش می‌کند. راهنمای برنامه ساز یونیکس در هر نوع برای سیستم شما، فهرست فرمانها، سیستم ها عادی و مشترک، قالب بندی فایل و روندهای حفظ و نگهداری مناسب هستند. شما بدون این سیستم برای مدت طولانی نمی‌توانید دوام بیاورید اگرچه شما احتمالاً تنها به خواندن بخشهای از جلد 1 تا زمان شروع برنامه‌سازی نیازمند باشید. جلد 1 چاپ هفتم کتاب راهنما توسط Holt , Rinehart و winston به چاپ رسید. جلد 2 راهنمای برنامه ساز یونیکس تحت عنوان سندی برای استفاده از سیستم اشتراک زمانی یونیکس نام دارد و حاوی نکات آموزشی و مرجع برای اکثر فرمانها می‌باشد. به ویژه این کتاب نکات مقدماتی برنامه‌ها و ابزارهای توسعه برنامه‌ها را به تفصیل بیان می‌کند. ممکن است شما بخواهید اغلب این مطالب را هرچند وقت مطالعه کنید. کتاب مقدماتی یونیکس که توسط Aun و (1983 prentice - Hall - Nico Lomuto) تهیه شده است، مقدمه مناسبی برای افراد مبتدی بی تجربه به ویژه افرادی که برنامه نویس نیستند، به شمار می‌رود.

فصل 2 : سیستم فایل

همه چیز در سیستم یونیکس به صورت یک فایل است. این سیستم بسیار ساده تر از آن است که شما فکر می کنید. زمانیکه اولین نسخه این سیستم طراحی می شد، قبل از آن که نامی داشته باشد، بحثها بروی سیستم فایل متمرکز شد که برای استفاده، تمیز و ساده باشد. سیستم فایل مرکز موفقیت و سادگی سیستم یونیکس است. این سیستم یکی از بهترین مثالهای فلسفه «ساده تر کنید» است؛ که نشان دهنده توان عملکرد پیاده سازی چند ایده درست انتخاب شده می باشد.

برای راحتتر صحبت کردن در مورد دستورات و روابط آنها نیاز به پیش زمینه مناسبی از ساختار و عملکرد بیرونی فایل سیستم داریم. این فصل بسیاری از جزئیات استفاده از سیستم فایل را در بر دارد- فایل چیست، چگونه بازنمایی می شود، شاخه ها و سلسله مراتب فایل سیستم، مجوزها، inode (رکورد داخلی سیستم فایل) و فایل های دستگاه جانبی.

چون بسیاری از کاربردهای یونیکس منجر به دستکاری فایل های می شود، فرمان ها و دستورات زیادی برای فایل بررسی و تنظیم فایل ها دارد؛ در این فصل به بررسی پرکاربردترین آنها پرداخته می شود.

مبانی فایل ها

یک فایل رشته ای از بایتها است. (بایت تکه کوچکی از اطلاعات است، که عموماً هشت بیت طول دارد. یک بایت را می توان معادل با یک کاراکتر دانست.) سیستم هیچ ساختاری بر روی فایل و هیچ معنایی برای محتوای آن در نظر نمی گیرد؛ مفهوم بایتها فقط به برنامه ای که فایل را تفسیر می کند، بستگی دارد. علاوه بر این همانطور که خواهید دید، این موضوع نه تنها در مورد فایل های ذخیره شده بر روی دیسکت بلکه در مورد دستگاه های جانبی نیز صادق است. نوار مغناطیسی، نامه ها، کاراکترهای تایپ شده روی صفحه کلید، خروجی به چاپگر خطی، داده های جاری بر روی خط لوله، تا جایی که به سیستم و برنامه های داخل آن مربوط باشد همگی فایل و معادل رشته ای از بایتها می باشند.

بهترین راه درک فایل بازی کردن با آنها است، بنابراین با ایجاد یک فایل کوچک شروع می کنیم:

```
$ed
a
now is the time
for all good people
.
w junk
36
q
$ ls -l junk
-rw-rw-r-- 1 mahdi mahdi 36 Jul 4 11:21 junk
$
```

Junk فایل 36 بایتی است. (36 کاراکتری که شما هنگام ایجاد فایل تایپ کرده اید؛ البته به جز اصلاحات اشتباهات تایپی). برای دیدن فایل،

```
$ cat junk
now is the time
for all good people
$
```

دستور cat نشان می‌دهد که محتوای فایل چیست. دستور 2od تمام بایت‌های فایل را بصورتی قابل رؤیت باز نمایش می‌کند:

```
$ od -c junk
0000000 n o w   i s   t h e   t i m e \n
0000020 f o r   a l l   g o o d   p e o
0000040 p l e \n
0000044
$
```

گزینه c به معنی تفسیر(نمایش) بایتها به صورت کارکتری است. انتخاب گزینه b، بایتها را به اکتال (octal) نیز نشان می‌دهد:

```
od -cb junk $
0000000 n o w   i s   t h e   t i m e \n
          156 157 167 040 151 163 040 164 150 145 040 164 151 155 145 012
0000020 f o r   a l l   g o o d   p e o
          146 157 162 040 141 154 154 040 147 157 157 144 040 160 145 157
0000040 p l e \n
          160 154 145 012
0000044
$
```

اعداد هفت رقمی که در پائین سمت چپ نمایش داده می‌شوند، شماره ترتیب کاراکتر در فایل برای اولین کاراکتر نمایش داده شده در آن خط است.

به هر ترتیب، تأکید بر روی شماره‌های اکتال، یک نظریه پابرجا از PDP-11 است، که در آن اکتال نمادگذاری پیشنهادی بوده است. مبنای 16 سازگاری بیشتری با سایر سخت افزارها دارد؛ گزینه x به od می‌گوید که اطلاعات به صورت هگز چاپ کند. توجه داشته باشید که در هر خط با یک کاراکتر 012 اکتال پایان می‌یابد. این کاراکتر newline (خط جدید) اسکی است؛ چیزی که سیستم در اثر فشار کلید enter دریافت می‌کند. بنابر یک عرف وام گرفته شده از زبان C کاراکتر خط جدید با \n نمایش داده می‌شود. اما باید توجه داشت که این نوع نمایش قراردادی برای برنامه‌هایی شبیه od است تا راحتتر بتوان آن را خواند؛ در سیستم یک بایت با مقدار 012 ذخیره می‌گردد.

کاراکتر newline رایج‌ترین مثال کاراکترهای خاص است. سایر کاراکترها با بعضی از عملیات کنترلی پایانه مرتبط می‌باشند، که شامل backspace (پس برد به عقب) (با مقدار اکتال 010 ، چاپ شده به صورت \b) کاراکتر (t، 011) \tab و کاراکتر سر خط³ (r، 015) \r می‌باشد. نکته بسیار مهم درک و تفکیک نحوه ذخیره سازی کارکترها در فایل و چگونگی تفسیر آنها در موقعیت‌های مختلف است. برای مثال وقتی شما backspace را روی صفحه کلید فشار می‌دهید هسته آن را به معنای حذف کاراکتر قبلی تفسیر می‌کند. کاراکتر قبلی و backspace ناپدید می‌شوند، اما backspace به پایانه شما انعکاس داده می‌شود تا مکان نما یک موقعیت به عقب برگردد.

اگر شما رشته ← را تایپ کنید (\ و به دنبال آن یک backspace) به هر حال هسته تفسیر معمول خود از backspace را دارد؛ بنابراین \ حذف می‌شود و بایت 010 از فایل شما محو می‌شود. انعکاس backspace بروی پایانه مکان نما را یک خانه به عقب برده و بروی \ قرار می‌دهد.

چاپ فایلی که شامل backspace است؛ موجب می‌شود که backspace بدون تفسیر به ترمینال منعکس شود و در نتیجه مکان نما یک

خانه به عقب برگردد. در صورت استفاده از `od` برای نمایش فایل‌ای که دارای `backspace` است، این کاراکتر به صورت `010` در صورت استفاده از گزینه `d` و به صورت `\b` در صورت استفاده از گزینه `c` نمایش داده خواهد شد.

داستان `tab` نیز به همین صورت است: در هنگام تایپ یک `tab`، این کاراکتر به پایانه منعکس شده و همچنین به برنامه‌ای که ورودی را مخواند فرستاده می‌شود. در خروجی، `tab` به سادگی به پایانه فرستاده می‌شود تا در آنجا تفسیر شود. البته در تفسیر `tab` برای نمایش تفاوت‌هایی وجود دارد. امکان تنظیم مشخصات سیستم برای نمایش این کاراکتر به صورت‌های مختلف توسط هسته پیش‌بینی شده است. بطور معمول هر کاراکتر `tab` با تعدادی فضای خالی جایگزین می‌شود تا مکان نما به سر خانه جدولی بعدی برود. سر خانه‌های جدول بر روی ستون‌های 9، 17، 25، ... قرار دارند. دستور `stty -tabs` منجر می‌شود که از این به بعد `tab` با تعدادی فضای خالی جایگزین شود. برای کسب اطلاعات دقیقتر به راهنمای `(1)stty` مراجعه کنید.

رفتار `enter` قیاسی است. هسته `enter` را به صورت یک کاراکتر سر خط و یک `newline` منعکس می‌کند؛ اما تنها `newline` را در ورودی ذخیره می‌کند. برای خروجی نیز `newline` به آن دو کاراکتر بسط داده می‌شود.

روش یونیکس برای نمایش اطلاعات کنترلی به خصوص در مورد کاربرد `newline` به منظور اعلام پایان خط متعارف نیست. در بعضی از سیستم‌ها به جای قراردادن هر رکورد در یک خط تعداد کارکترهای رکورد را نیز نگهداری می‌کنند (البته بدون هیچ علامت پایان خط یا رکوردی).

بقیه سیستم‌ها پایان هر خط را با کاراکتر سر خط و `newline` نمایش می‌دهند؛ زیرا این کارکترها برای خیلی از پایانه‌ها ضروری است. (واژه `linefeed` معادل این دو کاراکتر است که این معادل غالباً «`CRLF`» خوانده می‌شود که تقریباً قابل تلفظ است.) سیستم یونیکس هرگز دارای رکورد، شمارنده رکورد و یا هر اطلاعی که شما یا برنامه در فایل قرار نداده باشید، نمی‌باشد. یک `newline` زمان انعکاس به پایانه به کاراکتر سر خط و `newline` بسط داده می‌شود؛ اما برنامه‌ها نیاز دارند که تنها با کاراکتر `newline` سروکار داشته باشد زیرا این همه چیز نیست که آن‌ها می‌بینند. این طرح ساده چیزی است که دقیقاً برای بسیاری از مقاصد خواسته شده است. در صورت نیاز به ساختاری بسیار پیچیده می‌تواند آن را بر روی این ایجاد کرد. انجام برعکس این عمل، ایجاد سادگی از پیچیدگی، سخت‌تر است.

تا زمانی که انتهای خط با کاراکتر `newline` علامت‌گذاری می‌شود ممکن است انتظار داشته باشید که فایل به وسیله کاراکتر خاص دیگری خاتمه پیدا کند. نماد `(\e)` را برای انتهای فایل در نظر بگیرید. در خروجی `od` کاراکتر خاصی در انتهای فایل نمایش داده نمی‌شود؛ بلکه تنها نمایش متوقف می‌شود. بجای استفاده از کدی خاص، سیستم با اعلام عدم وجود داده بیشتر در فایل خاتمه یافتن فایل را مشخص می‌کند. هسته طول فایل را نگهداری می‌کند، بنابراین برنامه پس از پردازش تمام بایت‌های فایل با پایان فایل مواجه می‌شود.

برنامه با یک فراخوانی سیستمی (روالی از هسته) که `read` نام دارد داده‌ای درون فایل را بازیابی می‌کند. هر بار که `read` فرا خوانده می‌شود، بخش بعدی فایل مثلاً خط بعدی که روی پایانه تایپ شده است، را بر می‌گرداند. علاوه بر این `read` می‌گوید که چه تعداد از بایت‌های فایل برگردانده شده است. بنابراین انتهای فایل زمانی که `read` تعداد بایت‌های خوانده شده را صفر اعلام می‌کند در نظر گرفته می‌شود. اگر مقداری بایت باقی مانده باشد، `read` تعدادی از آن‌ها را برمی‌گرداند. واقعا این احساس ایجاد می‌شود که پایان فایل را با یک مقدار خاص نباید نمایش داد، زیرا همانطور که قبلاً بیان شد مفهوم بایت‌ها به نحوه تفسیر بایت‌ها بستگی دارد. اما همه فایلها باید به پایان برسند و چون همه فایلها باید به کمک `read` خوانده شوند، برگشت صفر یک راه مستقل از تفسیر برای نمایش پایان فایل است

بدون اینکه نیاز به معرفی یک کاراکتر جدید برای اینکار باشد.

وقتی که یک برنامه از پایانه شما می خواند هر خط ورودی وقتی که کارکتر `newline` تایپ شده بوسیله هسته به برنامه داده می شود. بنابراین وقتی که یک اشتباه تایپی مرتکب می شوید می توانید برگردید و آن را تصحیح کنید. البته اگر اشتباه را قبل از فشار `Enter` تشخیص دهید. بعد از آن خط توسط سیستم خوانده شده و شما نمی توانید آن را اصلاح کنید. با استفاده از برنامه `cat` می توان ورود اطلاعات به صورت هر بار یک خط را مشاهده کرد. در بعضی از سیستمهای قدیمی `cat` به طور معمول برای افزایش کارایی خروجی خود را بافر می کند. برای رفع این مشکل از گزینه `u` می توان استفاده کرد:

```
$ cat -u
123
123
456
456
789
789
ctl-d
$
```

زمانیکه `return` را فشار دهید `cat` خط را دریافت می کند؛ بدون بافر کردن اطلاعات بلافاصله پس از دریافت آن را چاپ می کند.

حالا موارد متفاوت را امتحان کنید؛ تعدادی کاراکتر و بعد از آن `ctl-d` بجای `Enter` تایپ کنید:

```
$ cat -u
123ctl-d123
```

`Cat` کاراکترها را بلافاصله چاپ می کند. تایپ `ctl-d` می گوید «کاراکترها بلافاصله بفرست؛ من برای برنامه ای که از پایانه می خواند تایپ کرده ام».

برعکس `Enter` خود `ctl-d` به برنامه فرستاده نمی شود. حالا دومین `ctl-d` را بدون کاراکترهای دیگر تایپ کنید:

```
$ cat -u
123ctl-d123ctl-d$
```

شل با یک پیامواره پاسخ می دهد؛ زیرا `Cat` هیچ کاراکتری را نمی خواند، قطعاً این به معنی پایان فایل است در نتیجه متوقف می شود. `ctl-d` هر چیزی را که شما تایپ کرده اید به برنامه ای که از پایانه می خواند می فرستد. اگر شما چیزی تایپ نکرده باشید، بنابراین برنامه هیچ کاراکتری را نمی خواند و این چیزی شبیه به انتهای فایل است. به همین دلیل تایپ کردن `ctl-d` شما را از سیستم خارج می کند، زیرا شل ورودی دیگری نمی بیند. البته `ctl-d` برای مشخص کردن انتهای فایل استفاده می شود. البته عملکرد عمومی بیشتری دارد.

در فایل چه چیزی هست؟

قالب بندی فایل به وسیله برنامه هایی که از آن استفاده می کنید معین می شود؛ انواع فایل ها شدیداً متنوعند، شاید به خاطر اینکه برنامه ها متنوعی وجود دارد. ولی تا زمانیکه نوع فایل توسط فایل سیستم مشخص نشود، هسته نمی تواند نوع فایل را مشخص کند؛ زیرا در مورد آن چیزی نمی داند. دستور `file` یک حدس مناسب ارائه می کند:

```
$ file /bin /bin/ed /usr/share/man/man1/ed.1.gz
```

```
/bin: directory
```

```
/bin/ed: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
```

2.2.5, dynamically linked (uses shared libs), stripped /usr/share/man/man1/ed.1.gz: gzip compressed data, from Unix, max compression

\$

اینها سه فایل نسبتاً نوعی هستند که همه به ویراستار ed مرتبطند: شاخه‌ای که برنامه در آن موجود است، باینری یا برنامه قابل اجرا و صفحه راهنمای برنامه.

برای مشخص کردن نوع، file توجهی به نام ندارد، زیرا عرفه نامگذاری، تنها یک قرارداد و کاملاً غیر حقیقی است. برای مثال پسوند c. برای کدهای زبان C در نظر گرفته شده است ولی هیچ دلیلی وجود ندارد که شما از فایلی به نام a.c با محتوایی دلخواه اجتناب کنید. در عوض file، چند صد بایت اول فایل را می‌خواند و به دنبال سر نخهایی برای تعیین نوع فایل می‌گردد. (همانطور که قبلاً بیان کردیم در بعضی مواقع مثلاً در مورد شاخه‌ها file می‌تواند از سیستم سؤال کند؛ البته حتی در همین موارد هم فایل با خواندن اول شاخه می‌تواند آن را تشخیص دهد.)

بعضی مواقع سر نخ‌ها مشخص‌اند، یک برنامه قابل اجرا توسط یک شماره جادویی در ابتدای خود علامت‌گذاری می‌شود. od بدون هیچ گزینه‌ای فایل را به صورت 16 بیتی یا 2 بیتی نمایش داده و شماره جادویی را قابل رویت می‌کند:

```
$ od /bin/ed
```

```
0000000 042577 043114 000401 000001 000000 000000 000000 000000
0000020 000002 000003 000001 000000 107140 004004 000064 000000
0000040 127244 000000 000000 000000 000064 000040 000007 000050
```

عدد 042577، نشانگر وجود برنامه‌ای قابل اجراست. یک کد اجرایی ممکن است توسط تعدادی پردازش همزمان اجرا شود. الگوی بیتی که با 042577 نشان داده می‌شود، یک متن ASCII نیست. و این مقدار نمی‌تواند توسط برنامه‌ای شبیه ویراستار ساخته شود. اما شما می‌توانید فایلی توسط برنامه‌های خاص خود بسازید، و سیستم فکر کند که آن یک برنامه اجرایی است.

برای فایل‌های متنی، سر نخ‌ها ممکن است عمیق‌تر باشند. بنابراین file به دنبال واژه‌ای شبیه #include می‌گردد تا مشخص کند فایل مورد نظر کد C است.

شما ممکن است از خود پرسید که چرا سیستم نوع فایل را با دقت در نظر نمی‌گیرد. برای مثال sort فایل /bin/ed را بعنوان ورودی نگیرد. یکی از دلایل خودداری در جلوگیری از بعضی از محاسبات مفید است.

شاخه‌ها و اسامی فایلها

همه فایل‌های شما، اسامی واضح و غیرمبهمی دارند که با /home/you شروع می‌شوند. اما اگر junk تنها فایل شما باشد و ls را اجرا کنید به صورت /usr/you/junk تایپ نمی‌شود. نام فایل بدون هیچ پیشوندی اینگونه تایپ می‌گردد.

```
$ IS
junk
```

\$

به همین دلیل برنامه در جریان که یک فرایند محسوب می‌شود، یک دایرکتوری جاری دارد. و تمامی نام فایل‌ها به صورت صنفی ایگونه فرض می‌شود که با نام آن، دایرکتوری آغاز شده است. مگر اینکه آنها مستقیماً با اسلش آغاز شوند. بنابراین راه‌یابی به سیستم عامل و IS یک دایرکتوری جاری دارد. فرمان «کارکرد دایرکتوری را چاپ کن» دایرکتوری جاری را شناسایی می‌کند.

```
$ pwd
/usr/you
```

\$

راهنمای جاری (دایرکتوری جاری) مشخصه‌ای از یک فرایند است نه از یک شخص یا برنامه. عموماً همه دایرکتوری ورود به سیستم

عامل را دارند و فرایندها دارای دایرکتوری جاری هستند. اگر فرایند، یک فرایند مولد ایجاد کند، این مولد، دایرکتوری جاری والد خود را با تمامی مشخصاتش به دست می آورد: البته اگر مولد یک راهنمای جدید (دایرکتوری جدید) تغییر یافت تأثیر بر روی والد خود نمی گذارد. دایرکتوری جاری به همان شکل باقی می ماند و به چگونگی مولد اهمیتی نمی دهد.

تصویری از یک راهنمای جاری (دایرکتوری جاری) مطمئناً یک نمادگذاری قراردادیست. زیرا می تواند بسیاری از نوع سازی ها را ذخیره سازد. اما هدف واقعی و اصلی آن سازمانی و از پیش برنامه ریزی شده است. فایل های مرتبط در دایرکتوری های یکسان، به یکدیگر تعلق دارند. /usr معمولا دایرکتوری بالایی از یک کاربر سیستم فایل است. (usr علامت اختصاری user است) وقتی که شما وارد سیستم عامل می شوید /usr /you / دایرکتوری ورودی و دایرکتوری جاری شما هستند:

/usr / src / شامل منبعی از برنامه های سیستم است. /usr / src / smd / sh / شامل منبعی از فرمان یونیکس است. شامل فایل اصلی برای شل است و غیره. وقتی که شما دست به یک پروژه جدید می زنید و یا وقتی یک سری فایل مرتبط دارید و یک سری دستورالعمل بیان می کنید می توانید با mkdir یک دایرکتوری جدید بسازید و آن را در فایل ها قرار دهید.

```
$ pwd
/ usr / you
$ mk dir recipes
$ cd recifes
$ pwd
/ usr / you / recipes
$ mkdir pie cookie
$ ed pie / aplole
.....
$ ed cookier / choc. Chip
.....
$
```

توجه داشته باشید که رجوع کردن به یک دایرکتوری فرعی بسیار ساده است. pie/apple یک مفهوم کاملا واضحی دارد. دستورالعمل apple pie در دایرکتوری بدین شکل است.

```
/ recipes / apple pie
```

اما قرار گرفتن تمامی pies با هم سازمان یافته تر به نظر می رسد. برای مثال دستورالعمل crust به جای کپی گرفتن از آن در هر سیستم pie به صورت recipes / pie اجرا شود.

گرچه سیستم فایل یک ابزار سازمان یافته قدرتمند است شما ممکن است فراموش کنید که فایل را کجا قرار داده اید و یا به چه فایلی دست یافته اید. یک راه حل واضح و روشن برای یافتن فایل یک یا دو فرمان جستجو کردن در میان دایرکتوری هاست. زمان IS برای یافتن فایل بسیار مؤثر است ، بدون اینکه به دایرکتوری فرعی مراجعه شود.

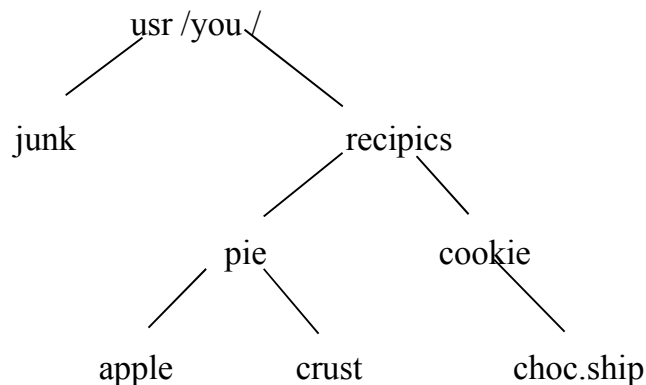
```
$ cd
$ ls
junk
recipes
$ file *
jund ascit text
```

```

recips :          directory
$ IS re cipes
cookie
pie
$ IS recipes / Pie
apple
crust
$\

```

تصویری از یک سیستم فایل در زیر نشان داده شده است.



فرمان `du` (کاربرد دیسک) نوشته شده است تا بیان کند چه مقدار از فضای دیسک توسط فایل‌های داخل دایرکتوری اشغال شده، که شامل تمامی فایل‌های فرعی نیز می‌باشد.

```

$ du -a
2      . / recipes / pie / apple
3      . / recipes / pie / crust
6      . / recipes / pie
3      . / recipes / cookie/ choc.chip
4      . / recipes / cookie
11     . / recipes /
1      . / junk
13     .
$

```

نام فایل‌ها واضح و مشخص است. این اعداد شماره بلوک‌های دیسک ذخیره‌سازی به ازای هر فایل محسوب می‌گذارند - تقریباً به ازای هر 1024 یا 512 بایت - اندازه هر دایرکتوری نشان می‌دهد چند بلوک توسط فایل‌ها در دایرکتوری و دایرکتوری فرعی اشغال شده که شامل دایرکتوری نیز می‌شود. `Du` برای «`wll`» یک گزینه `a` دارد که باعث می‌شود تمامی فایل‌ها در دایرکتوری چاپ شود. اگر یکی از آن‌ها دایرکتوری، `du` را اینگونه پردازش می‌کند.

خروجی `dw - a` می‌تواند برای جستجوی فایل‌های ویژه مسیری به طرف `grep` ایجاد کند.

```

$ du -a / grep choc
$      . / recipes / cookie / choc. Chip
$

```

نگاهی دوباره به قسمت یک داشته باشید. نام (‘.’) مدخل دایرکتوری می‌باشد که به خود دایرکتوری رجوع پیدا می‌کند و دستیابی به

دایرکتوری را بدون دانستن نام آن مجاز می شمرد. Du برای یافتن فایل به دایرکتوری سری می زند، اگر شما به او نگوئید کدام دایرکتوری او فرض را بر این علامت (‘.’) می گذارد. بنابراین junk و / junk نام فایل های مشابهند. علاوه بر مشخصات بنیادی داخل هسته ، دایرکتوری ها در سیستم فاعلی به عنوان فایل های معمولی قرار می گیرند. آن ها می توانند به عنوان فایل های معمولی نیز خوانده شوند. به منظور حفظ اعتدال و فایل های کاربر، هسته کنترل همه اطلاعات دایرکتوری را در دست می گیرد.

تایم نگاهی به بایت ها در راهنما می اندازد :

به نام فایل های پنهان شده در آنجا نگاهی کنید. فرمت دایرکتوری مخلوطی از دودویی و داده های در متن است. یک دایرکتوری شامل قسمت های 16 بایتی می باشند و 14 بایت آخر که نام فایل ها را در بر دارند توسط ASCII Nul's بسط داده می شود. (که ارزش صفر دارد). و دو تای اولی بیان می کند که سیستم اطلاعات اداری را در کجای فایل قرار می دهد. ما بعداً به این مطلب دوباره اشاره خواهیم کرد. هر دایرکتوری با دو ورودی آغاز می شود (“dot”) and (“.”) . ‘dot-dot’ دایرکتوری ، ریشه (Root) سیستم عامل خوانده می شود. هر فایلی در سیستم در دایرکتوری ریشه و یا یکی از دایرکتوری های فرعی آن است و ریشه (Root) ، دایرکتوری والد آن است.

4 - 2 مجوزها

هر فایلی یک سری مجوزهایی مربوط به خود دارد که تعیین می کند چه کسی چه کاری با فایل انجام می دهد. شاید شما تصمیم دارید نامه های عاشقانه خود را در سیستمی نگه دارید و با سلسله مراتبی در دایرکتوری مرتب کنید. و نمی خواهید دیگران آن ها را بخوانند. بنابراین شما می توانید مجوزهای آن نامه ها را به طرح های نامفهوم تغییر دهید. و یا می توانید مجوزها را به دایرکتوری هایی که شامل نامه هاست تغییر دهید و بدین وسیله آن ها را از دسترس دیگران دور کنید.

اما در اینجا هشدار می دهیم که یک کاربر ویژه بر روی سیستم یونیکس قرار دارد که super-user خوانده می شود. که می تواند هر فایلی را بر روی سیستم بخواند یا شناسایی کند. Root که نام ویژه و رودیست مزیت های super-user را در بر دارد که با اداره کننده سیستم در حین مراقبت از سیستم استفاده می شود. اگر شما نام رمز Root را بدانید. یک فرمان به نام Su موقعیت shper-user را گزارش می دهد. هر کس رمز super-user را بداند می تواند نامه های شما را بخواند. بنابراین موارد حساس را در این فایل قرار ندهید. اگر شما قصد پنهان کاری و اختفا دارید می توانید داده های داخل فایل را تغییر داده تا super-user قادر به خواندن آن نباشد (و یا حداقل درک نکنند). از فرمان crypt استفاده کنید. البته crypt نیز نمی توند کاملاً امن و مطمئن باشد supper-hser می تواند فرمان crypt را نیز تغییر دهد و یک عملیات رمزنگاری بر روی الگوریتم crypt انجام دهد. روش اولی امکان خطا دارد بنابراین crypt در عمل نسبتاً ایمن است.

در زندگی واقعی امنیت و اطمینان مستلزم رمزنگاری است تا لو داده نشود و به آسانی نیز حدس زده نشود. گاهی اوقات سیستم اجرایی خطاها برای کاربرهای خطاکار این امکان را به وجود می آورد تا مجوز super-user را به دست آورد. نتیجه و کارایی این امنیت در متن های ذکر شده در کتابشناسی انتهای این فصل مورد بحث قرار می گیرد.

وقتی که شما وارد سیستم عامل می شوید اسمی را تایپ می کنید و تأکید دارید که شما آن شخصی هستید اسم رمز را تایپ کرده اید. این اسم منطبق بر اسم ورودی شماست. ligin-id. البته سیستم شما را توسط شماره ها تشخیص می دهد که user-id و یا uid خوانده

می شود.

در حقیقت login-id متفاوت ممکن است uid یکسان داشته باشد، که آن را برای سیستم غیر قابل تشخیص می سازد. البته این عمل نسبتاً نادر، به دلایل امنیتی نامطلوب است. گذشته از uid، شما یک هویت شناسی گروهی group-id ترتیب می دهید که شما را در رده کاربرها قرار می دهد. همه کاربرها بر روی بسیاری از سیستمها (بر خلاف logging-id مثل roat) در گروهی به نام other قرار دارند. ولی سیستم شما ممکن است متفاوت باشد. سیستم فایل و سیستم یونیکس به طور کلی تشخیص می دهند که شما با مجوز داده شده با uid و group-id چه می توانید بکنید.

فایل etc/passwd یک سیستم رمزنگار است. و تمام یاطلاعات ورودی را شامل می شود. شما می تونید uid و group-id را توسط یافتن نامتان در etc/passwd پیدا کرده همانگونه که سیستم عمل می کند.

```
$ grep you / etc / passwd
you : g k m b c T r j o A com : 604 .1 : y. O. A . people : / usr/ you
$
```

زمینه ها و فیلدها در فایل رمزنگار توسط علامت « : » مجزا می شوند و به صورت زیر طرح ریزی می شوند.

```
login-id : encrypted – passwd : uid : group – ld : miscell any
: login – directery : shell
```

فایل های یک متن معمولی هستند و تفکیک ها و تعریف های فیلد قراردادی اند که مطابقت می کند با برنامه های که از اطلاعات داخل فایل استفاده می کنند.

فیلد شل، اغلب خالیست و مستلزم اینست که شما از شل پیش فرض به این صورت استفاده کنید.

bin / sh / فیلدهای متفرقه ممکن است هر چیزی را در بر داشته باشد و اغلب شامل اسم و آدرس و شماره تلفن شما می باشد. توجه داشته باشید اسم رمز شما بر روی فیلد دوم ظاهر می شود، البته تنها به صورت یک فرم رمزسازی شده. هر کسی می تواند اسم رمز را بخواند. بنابراین به کمک رمز هر کس می تواند به جای شما از آن استفاده کند. وقتی شما اسم رمز را به login می دهید او آن را رمزسازی کرده و نتیجه را مقایسه می کند با اسم رمز، رمزسازی شده etc/passwd. اگر آنها مطابقت کردند شما می توانید وارد سیستم شوید. این مکانیسم مؤثر است زیرا الگوریتم رمزسازی دارای یک ویژگیست که آن ورود از یک فرم خالی به یک فرم رمزسازی شده به آسانی صورت می گیرد ولی معکوس این عمل به سختی صورت پذیر است. ممکن است به صورت g k m b c t r y o 4 رمزسازی شود، اما بعد از دریافت نامه، برگشتن به حالت اولیه آسان نیست.

هسته تشخیص می دهد که به شما اجازه خواندن etc / passwd داده شده البته توسط نگاه کردن به مجوزهای فایل. در اینجا برای هر فایل سه نوع مجوز وجود دارد. بخوانید (متن را امتحان کنید) بنویسید (متن را تغییر دهید) و اجرا کنید (به عنوان یک بنا آن را اجرا کنید). به علاوه مجوزهای متفاوت توسط افراد متفاوت اجرا می شود. به عنوان ماسک فایل شما یک سری مجوز خواندن، نوشتن و اجرا کردن دارید. گروه (group) شما یک دستگاه مجزا دارد، هر شخص دیگری سه دستگاه دارد.

گزینه IS-1 اطلاعات مجوز را در میان دیگر چیزها چاپ می کند.

```
$ IS - 1 / etc / passwd
-rw - r - r - - 1 root 5115 Aug 30 10 :40
```

```
$ IS - Ig / etc / passwd
-rw - r - - r - - 1 adm
```

این دو خط ممکن است مجموعاً اینگونه تفسیر شود : `login-id root` و `group adm`

`etc / passwd /` می باشند که به طول 5115 بایت است و در تاریخ 30 آگوست و ساعت 45 : 10 دقیقه صبح اصلاح شده و دارای یک پیوند است. اولین اسم در سیستم فایل ممکن است پیوند را در قسمت بعدی مورد بحث قرار دهد (بسیاری از طرح های IS در یک سیستم، مالک و گروه را به دست می دهد).

خط `rw - r - - r - -` نشان می دهد که چگونه IS مجوزها را بر روی فایل نمایش می دهد. ابتدا نشان می دهد که این یک فایل معمولی است ، اگر یک دایرکتوری وجود داشت ، علامت `ok` در آنجا وجود دارد. همه کارکتر دیگر بعدی یعنی خواندن، نوشتن و اجرا کردن یک مجوز، صاحب فایل را رمزگذاری می کند. (بر مبنای `rw` . یعنی `uid` (root) مالک می خواند و می نویسد اما فایل را اجرا نمی کند یک فایل قابل اجرا به جای خط تیره `x` دارد.

سه کاراکتر بعدی (`- - r`) مجوزهای گروه را رمزگذاری می کنند، در این حالت هر شخص در گروه و هر سیستم اداره کننده ای می تواند فایل را بخواند ولی نمی تواند بنویسد یا اجرا کند. سه تایی بعدی (`- - also r`) مجوزها را برای بقیه کاربرهای روی سیستم تعریف می کند، در این ماشین، `root` اطلاعات ورودی برای کاربر را تغییر می دهد. اما هر کسی می تواند فایل را برای یافتن اطلاعات بخواند. یک انتخاب به ظاهر درست برای گروه `adm` می تواند نوشتن مجوزها روی `etc/passwd /` باشد.

فایل `etc / group /` اسم های گروه و `group - id` را رمزگذاری می کند. و مشخص می کند کدام کاربر در کدام گروه است. `etc / passwd` تنها گروه ورودی را شناسایی می کند. فرمان `newgrp` مجوزهای گروه را به گروه دیگری تغییر داده :
هر کسی می تواند بیان کند . `$ ed / etc / passwd` و فایل `passwd` را ویرایش کند. اما تنها `root` می تواند معکوس تغییرها را بنویسد.

ممکن است از خود پرسید که چگونه اسم رمز را می توانید تغییر دهید در حالیکه اسم رمز فایل در حال ویراستن است. برنامه ای که اسم رمز را تغییر می دهد `passwd` خوانده می شود. که ممکن است شما آن را در `/bin/` پیدا کنید :

```
$ IS - 1 / bin / passwd
- rwsr-x r-x 1 root      8454 Jan  4 1983 / bin / passwd
$
```

توجه داشته باشید که `etc / passwd /` فایل متنی است که شامل اطلاعات ورودیست. در حالیکه `bin / passwd /` در دایرکتوری متفاوتی فایلی است که شامل برنامه های قابل اجراست که به شما اجازه می دهد اسم رمز اطلاعات خود را تغییر دهید. مجوزها بیان می کنند که هر کسی می تواند فرمان را اجرا کند اما تنها `root` می تواند اسم رمز زمان را تغییر دهد. `S` که به جای `X` در فیلد اجرا برای فیلد ماسک قرار می گیرد بیان می کند که وقتی فرمان اجرا می شود. در حالت `root` ، مجوزهایی که با مالک فایل هم خوانی داشته باشد به دست می آیند .

دستگاه `uid-bit` یک نظریه ساده اما ظریف است. که شمارش مشکلات ایمنی را حل می کند. برای مثال نویسنده برنامه های بازی می تواند برنامه `set-uid` را برای صاحبش بسازد. بنابراین او می تواند یک فایل خطی را که از دسترس دیگر کاربرها دور شده به روز کند. اما مفهوم `set-uid` خطرناک است. `bin / passwd /` باید درست و صحیح باشد. اما اگر درست نبود می تواند اطلاعات سیستم را که تحت نظر `root` است را از بین ببرد. اگر او مجوز `rw sr wx rwx` را داشته به وسیله هر کاربری که می تواند فایل را به هر برنامه ای که هر کاری می کند رونوشت شود. این موضوع برای `set-uid` بسیار ضروری است. زیرا `root` به مجوزهای هر سیستمی فایلی

دسترسی دارد. وقتی که فایلی تغییر می کند ، سیستم `set - unix bit` ، یونیکس را قطع می کند تا خطر حوزه ایمنی را کاهش دهد. دستگاه `uid bit` قدرتمند است، اما به طور ابتدائی برای تعداد معدودی سیستم برنامه های نظیر `passwd` به کار می رود. بیائید به تعدادی فایل معمولی نگاهی بیندازیم.

```
IS - 1 / bin / who $
rwxrwxv-w 1 root 6348 mar 29 1983 / bin / who -
```

`who` چیز است که توسط هر شخصی قابل اجراست و توسط `root` قابل نوشتن است مصاحب گروهی می باشد. منظور از قابل اجرا بودن چیست وقتی که شما برای `shell` تایپ می کنید :

```
who $
```

به نظرمی رسد برای فایل هایی که `who` نامیده می شود، در یک سری دایرکتوری، یکی از آن ها `bin /` است. اگر او شبیه به یک فایل بود و فایل نیز یک مجوز قابل اجرا داشت ، شل به هسته فرمان اجرا شدنش را می دهد. هسته مجوزها را کنترل می کند و اگر معتبر بودند برنامه را اجرا می کند - توجه داشته باشید که یک برنامه تنها یک فایل است با مجوز قابل اجرا. در فصل بعدی ما به شما برنامه هایی نشان خواهیم داد که تنها یک فایل متنی باشند و می توانند مانند فرمان اجرا شوند زیرا دارای دستگاه مجوز اجرا هستند. مجوز دایرکتوری مقداری متفاوت عمل می کند، اما مبنای نظریه یکی است.

```
$ IS - Id
drwxrwxr 3 y-u 80 sep 27 06 : 11
$
```

گزینه `IS-d` بیشتر در مورد دایرکتوری صحبت می کند تا در مورد اطلاعاتش و منجر شدن `d` به یک علامت خروجی. یک فیلد `r` یعنی ، شما می توانید دایرکتوری را بخوانید. در نتیجه شما می توانید بفهمید که چه فایلی در `IS` (و یا `od`) وجود دارد. `W` یعنی شما می توانید فایل ها را در این دایرکتوری حذف یا اضافه کنید. زیرا او فایل دایرکتوری را تغییر می دهد و سپس می نویسد. در واقع شما نمی توانید به راحتی روی یک دایرکتوری بنویسد حتی اقدام `root` برای انجام این کار ممنوع است.

```
$ who try to wer write ‘.’
.: cannot creat you can’t
$
```

در عوض سیستمی وجود دارد که می تواند فایل ها را حذف و اضافه کند و تنها در حین آن ممکن است اطلاعات دایرکتوری را تغییر داد. نظریه مجوزها اینگونه کاربرد دارد که فیلد `w` بیان می کند که چه کسی می تواند از روال سیستم برای تغییر و یا اصلاح دایرکتوری استفاده کند.

اجازه حذف فایل به خود فایل بستگی ندارد. اگر شما مجوز را بر روی دایرکتوری نوشته اید، می توانید همانجا فایل را حذف کنید، حتی اگر فایل در قبال نوشتن محافظت شده باشد. فرمان `rm` قبل از حذف کردن یک فایل حفاظت شده، اجازه می گیرد. به هر حال به منظور اطمینان یافتن از اینکه شما واقعاً می خواهید حذف کنید، در مواقع نادری برنامه یونیکس مقصود شما را دوباره کنترل می کند. (علامت `F` به روی `rm` باعث می شود که حذف فایل بدون اجازه دوباره انجام پذیرد) فیلد `X` در مجوز برای یک دایرکتوری به معنی اجرا شدن نمی باشد. بلکه معنی آن جستجو کردن است مجوز اجرا، برای دایرکتوری مشخص می کند که آیا دایرکتوری برای فایل جستجو شده است. بنابراین ایجاد یک دایرکتوری با وضعیت (`- - X`) برای بقیه کاربرها امکان دارد. که این مستلزم آن است که کاربرها به هر فایلی که آن ها در مورد دایرکتوری اش می دانند دسترسی داشته باشند. ولی ممکن است `IS` را اجرا نکند و حتی آن را نخواند و نمی داند که فایل کجا قرار دارد. مجوز دایرکتوری (`- - r`) نیز همینگونه است. کاربر می تواند `IS` را ببیند اما نمی تواند از محتوای دایرکتوری استفاده

کند. بسیاری از تأسیسات از این وسیله برای قطع کردن / `usr games` در طول ساعات شلوغ استفاده می کنند.

فرمان `chmod` (وضعیت متغیر) مجوزهای روی فایل را تغییر می دهد.

\$ `chomod permissions filemond`

نحو و دستور زبان مجوزها بد ترکیب و ناهنجار است. آن‌ها می توانند به دو روش مشخص شوند. یکی به روش اعداد اکتال و یا با توصیفات نمادی. اعداد اکتال برای استفاده آسانتر هستند. اما توصیفات نمادین بعضی وقت‌ها مناسب تر است. زیرا آنها تغییرات مربوطه را در مجوزها مشخص می کنند بهتر است شما بگوئید :

```
$ chmod rw - rw - rw - junk
```

تا اینکه بگوئید

```
$ chmod G G G junk
```

مدهای اکتال توسط بهم بستن با یکدیگر مشخص می شوند. عدد 4 برای خواندن ، 2 برای نوشتن و 1 برای اجرا کردن. این سه رقم مشخص می کنند که IS مجوزی برای مالک ، گروه و هر شخص دیگریست. کدهای نمادی برای شرح و بسط دادن بسیار مشکل هستند. شما برای دستیابی به توصیفات کاملتر می توانید به `chmod (1)` نگاه کنید. برای اهداف، توجه داشته باشید که + یعنی وصل کردن مجوز - یعنی قطع کردن مجوز برای مثال ...

```
$ chmod + x command
```

شخص را وارد کنید تا این فرمان را اجرا کند.

```
$ chmod - w file
```

و سپس مجوز نوشتن را که شامل مالک فایل است را قطع کنید. به جز موارد عدم تعهد در مورد `super - user` مالک فایل مجوز فایل را بدون توجه به مجوزها تغییر می دهد. حتی اگر شخص دیگری شما را وادار کند که فایل را بنویسید، سیستم به شما اجازه نمی دهد که بیت‌های مجوزش را تغییر دهید.

```
$ IS - Id / user / mary
d r w x r w a r w x 5 mary 704 sep 25
```

```
$ chmod 444 /usr / mary
chmod : can't change /usr / mary
```

```
$
```

اگر دایرکتوری قابل نوشتن باشد می توانید فایل را بدون توجه به مجوز روی فایل‌ها حذف کنید. اگر شما می خواهید مطمئن شوید که شما یا دوستان فایل را هرگز از روی دایرکتوری حذف نمی کنید، مجوز وشتن را از آن حذف کنید.

```
$ cd
```

```
$ date > temp
```

```
$ chmod - w.
```

```
$ Is - Id .
```

```
dr - xr - xr - x 3 you
```

```
rm : temp not removed
```

```
$ chmod 77 5
```

```
$ Is - Id
```

```
d r w x r w x r - x 3 you
```

```
$ rm temp
```

\$

temp در اینجا آمده شده است. توجه داشته باشید که تغییر دادن مجوز روی دایرکتوری بر روی دیت‌های متغیر تأثیری ندارد. دیت‌های متغیر بر روی تغییرات محتوای فایل مؤثر است، و نه بر وضعیتش. مجوزها و دیت‌ها بر روی خود فایل ذخیره نمی‌شوند، بلکه روی ساختار سیستم که ثره شاخص (index nod) نامیده می‌شود ذخیره می‌شود.

: inodes 2 - 5

هر فایلی دارای عناصر متعددی است. نام و محتوا، اطلاعات اداره کننده مانند مجوزها و زمان‌های اصلاح اطلاعات ذخیره کننده که در inode ذخیره می‌شود و با ضرورت داده‌های سیستم سروکار دارد مانند: کجای دیسک محتوای فایل ذخیره شده و چقدر طولانی می‌باشد و غیره.....

سه زمان در inode وجود دارد. طول مدتی که محتوای فایل اصلاح یافت و نوشته شد، طول مدتی که فایل استفاده شد (خوانده شد یا اجرا شد) و مدتی که inode تغییر یافت. که در این مورد می‌توان تشکیل مجوزها را مثال زد.

\$ date.

The sep 27 12 : 07 : 24 EDT 1983

\$ date > junk

- rw - rw - rw - l you

\$ ls -lw junk

- rw - rw - rw junk

\$ ls -lc junk

- rw - rw - rw - 7 you

\$

تغییر محتوای فایل به زمان کاربرد آن تأثیری ندارد. همانگونه که گفته شد توسط ls-lu و تغییر مجوزها تنها بر روی زمان تغییر inode مؤثر است طبق گفته‌های پیشین توسط ls-lc.

\$ chmod 444 junk

\$ ls -lu junk

-r--r--r-- 1 you 29 sep 27 06:11 junk

\$ ls -lc junk

-r--r--r-- 1 you 29 sep 27 12:11 junk

\$ chmod 666 junk

\$

گزینه ls-t، که فایل‌ها را برطبق زمان ذخیره می‌کند توسط پیش‌بینی مدت اصلاح، می‌تواند ترکیبی از C یا u باشد. که بیان می‌کند کدام inodes تغییر کرده و یا کدام فایل خوانده شده.

\$ ls recipes

cookie

pie

\$ ls -lwt

total 2

d r w x r w x r w x 4 you

- rw - rw - rw - l you

\$

recipes اخیراً خیلی استفاده شده بود. و ما نگاهی به محتوایش نیز انداختیم. درک کردن inodes بسیار مهم است، نه تنها به خاطر

فهمیدن گزینه Is بلکه به خاطر اینکه در یک مفهوم قوی inoder یک فایل است. سلسله مراتب دایرکتوری - نام‌های مناسبی برای فایل‌ها فراهم می‌سازد. نام داخلی هر سیستم برای هر فایل I-number است. شماره‌های inode اطلاعات فایل را در بر دارد. I-1 بیانگر I-number در سیستم‌دهی (مبنا ده) است.

```
$ date > x
$ Is - i
1 5 7 6 8 junk
1 5 2 7 4 recipes
1 5 8 5 2 x
$
```

دو بایت اول در ورودی دایرکتوری تنها رابطه بین نام فایل و محتوایش می‌باشد. نام فایل‌ها در هر دایرکتوری یک پیوند خوانده می‌شود. زیرا او یک نام از سلسله مراتب دایرکتوری را به inode پیوند می‌دهد. i-number یکسان می‌تواند در بیش از یک دایرکتوری ظاهر شود. فرمان rm , inode را حذف نمی‌کند. او ورودی دایرکتوری یا پیوندها را حذف می‌کند. تنها وقتی آخرین پیوند با فایل ناپدید شود، سیستم، inode را حذف می‌کند. اگر i-number در یک ورودی سیستم صفر باشد، به این معنی است که پیوند حذف شده است. نه محتوای فایل. ممکن است پیوندهای دیگری نیز وجود داشته باشد. شما می‌توانید اثبات کنید که i-number از طریق حذف کردن، به طرف صفر می‌رود.

فایل بعدی که در این دایرکتوری ساخته شده است، به سوی یک شکاف بلا استفاده پیش خواهد رفت. گرچه ممکن است i-number متفاوتی داشته باشد.

فرمان In یک پیوند با فایل کنونی می‌سازد. \$ Inold - file new file

هدف از پیوند، دادن دو نام به یک فایل یکسان است. زیرا اغلب می‌تواند به صورت دایرکتوری متفاوتی ظاهر شود. در بسیاری از سیستم‌ها پیوند با / e / bine / ed called / bin وجود دارد، که می‌تواند ویراستارها را e بخواند. دو پیوند با فایل به inode مشابهی اشاره دارد.

```
$ In junk likt ojunk
$ Is - Ii
total 3
1 5 7 6 8 - rw - rw - rw - 2 you
1 5 7 6 8 - rw - rw - rw - 2 you
1 5 2 7 4 d r w x r w x r w x 4 you
$
```

عدد صحیح چاپ شده بی نمجوزها و ماسک‌ها عدد پیوند با فایل است. زیرا هر پیوندی به inode اشاره دارد. همه پیوندها با طور یکسانی اهمیت دارند. هیچگونه تفاوتی میان اولین پیوند و پیوند بعدی وجود ندارد. توجه داشته باشید که مجموع فضای دیسک که با Is محاسبه شده اشتباه است زیرا دوبار حساب شده است.

وقتی که فایلی را تغییر می‌دهید، دستیابی به فایل، تغییرها را آشکار می‌سازد. البته تا زمانیکه پیوندها به فایل مشابهی اشاره داشته باشند.

```
$ echo x > junk
$ Is - I
total 3
- rw - rw - rw - 2 you
- rw - rw - rw - 2 you
d r w x r w x r x x 4 you
```

```
$ rm link to junk
total 2      -rw -rw -rw - 1  you
drw x rw x rw x 4  you
$
```

بعد از اینکه link to junk حذف شود، شمار پیوندها به یک باز می‌گردد همانطور که قبلاً نیز گفتیم عملیات rm به روی فایل تنها پیوند را می‌شکند. فایل تا وقتی که آخرین پیوند وجود داشته باشد حفظ می‌شود. البته در عمل، بسیاری از فایل‌ها تنها یک پیوند دارند. اما ما دوباره شاهد نظریه ساده‌ای هستیم که انعطاف پذیری بزرگی را فراهم می‌کند.

پیوندی با فایل برقرار می‌شود، داده‌ها غیر قابل برگشتند. فایل حذف شده رو به نابودی و سوختن می‌رود. و هیچ راهی برای بازگرداندن آن‌ها از خاکستر شدن نیست. (امید ضعیفی برای تجدید حیات وجود دارد، بسیاری از سیستم‌های یونیکس بزرگ دارای فرایند پشتیبانی هستند که از تغییرات فایل‌ها مانند نوار مغناطیسی کپی می‌گیرند. از این به بعد آن‌ها می‌توانند قابل برگشت باشند. برای اطمینان خاطر شما باید بدانید که پشتیبان تا چه حد از سیستم شما حفاظت می‌کند. اگر پشتیبانی وجود نداشته باشد مشاهده می‌کنیم که دیسک دچار خرابی بدی می‌شود.

پیوند با فایل قابل دسترسی است، وقتی دو نفر می‌خواهند در یک فایل با هم سهیم باشند. اگر گاهی شما می‌خواهید فایل‌ها را تفکیک کنید (یک فایل متفاوت با اطلاعات یکسان) شما می‌توانید از مدارک کپی بگیرید قبل از آنکه تغییرات زیادی در آن ایجاد شود. برای مثال شما می‌توانید اطلاعات اصلی را بدون هیچ تغییری دوباره ذخیره کنید. ساختن پیوند کمکی نمی‌کند، زیرا وقتی داده‌ها تغییر کند پیوندها نیز تغییر می‌کنند. CP کپی‌های فایل را تشکیل می‌دهند.

```
$ cp junk copy of junk
$ ls -li
total 3
1 5 8 5 0  - rw -rw -rw- 1  you
1 5 7 6 8  - rw -rw -rw - 1  you
1 5 2 7 4  drw x  rw x rw x 4  you
$
```

i-number برای junk و copy of junk متفاوت است. زیرا فایل‌های متفاوتی هستند. حتی اگر دارای محتوای یکسانی باشند. اغلب یک نظریه خوب به یک مجوز بر روی کپی پشتیبان تغییر می‌یابد، بنابراین حذف کردن آن به طور تصادفی بسیار سخت است.

```
$ chmoe -w copy of junk
```

```
⋮
```

تغییر دادن کپی‌های فایل و حذف آن‌ها تغییری در اصل و مبدأ ایجاد نمی‌کند. توجه داشته باشید که چون copy of junk نوشته است که مجوزها قطع شده‌اند، rm برای حذف فایل نیاز به تأیید دارد.

برای اجرای عملیات فایل‌ها، بیش از یک فرمان وجود دارد. mv فایل‌ها را با مرتب کردن پیوندها دوباره نامگذاری می‌کند. دستورالعمل آن به صورت cp و In است.

```
$ mv junk sam old junk
$ ls -li
total 2
1 5 2 7 4  drw x  rw x rw x 4  you
1 5 7 6 8  - rw -rw -rw - 1  you
```

\$
 some old junk شبیه همان فایل junk قبلی است. به i-number نگاه کنید تنها نامش تغییر یافته (ورودی دایرکتوری با 1 5 7 8 inode مرتبط است)

In تمامی این فایل‌ها را که در یک دایرکتوری به صورت به هم ریخته بود، اجرا کردیم. اما آن تنها در راستای دایرکتوری عمل می‌کند. برای برقراری ارتباط با نام‌های یکسان در دایرکتوری‌های متعددی استفاده می‌شود. مانند وقتی که افراد زیادی در یک برنامه شرکت دارند. mv می‌تاند فایل دایرکتوری را به دایرکتی دیگری انتقال دهد. در حقیقت آن‌ها اطلاعات رایجی هستند که mv و cp دستور نحو خاصی برای آن‌ها دارند.

mv (or cp) file 1 file 2 ... directory

فرمان بالا یک یا دو فایل را به دایرکتوری که در آخرین مرحله است حرکت می‌دهد. کپی‌ها و پیوندها با نام فایل‌های مشابهی ساخته می‌شوند. برای مثال، اگر می‌خواهید، مهارت خود را در ویراستاری امتحان کنید باید با بیان این فرمان آغاز کنید:

\$ cp / usr / src / cmd / ed

اگر شما قصد دارید در یک شل کار کنید شماره فایل‌های منابع متفاوت را اینگونه بیان کنید.

\$ mk dir sh

\$ cp / usr / src / end / sh * sh

cp از تمامی شل فایل‌های منبع از دایرکتوری فرعی sh کپی برداری می‌کند. (فرض می‌کنیم ساختار دایرکتوری فرعی به صورت /usr / src / cnd / sh نمی‌باشد و cp نیز خیلی کارآمد نیست).

در بسیاری از سیستم‌ها، In بحث‌های فایل چندگانه را می‌پذیرد. همچنین در مورد دایرکتوری‌ها نیز همین گونه عمل می‌کند. و در بسیاری از سیستم‌ها mv و cp و In تنها به یک فایل مرتبط است که نام خود را برای اجرای عملیات امتحان می‌کند.

6 - 2 - سلسله مراتب دایرکتوری

bin

در فصل اول نگاهی بر روی سلسله مراتب سیستم فایل انداختیم که از /usr / you / شروع می‌شود. این بار قصد داریم در مورد آن با یک روش برنامه‌ریزی شده تحقیق و بررسی کنیم. که با بیش از سه ریشه (root) آغاز می‌شود.

\$ Is / boot

dev

etc

lib

tmp

unix

usr

\$

/ یونیکس برای خود هسته یونیکس یک برنامه است. وقتی سیستم شروع به کار می‌کند / unix را از روی دیسک خوانده می‌شود. این فرایند در دو مرحله رخ می‌دهد. ابتدا فایل / boot خوانده می‌شود، سپس از روی / unix می‌خواند. اطلاعات زیادی در مورد این فرایند سیستم خود راه‌انداز ممکن است در 8 boot) یافت شود. بقیه فایل در اسلش دایرکتوری است که هر کدام قسمت‌های کاملی هستند که

مجموع سیستم فایل را به وجود می آورند.

بیت ها را در دایرکتوری مذکور پیدا کنید. شما با طرح کلی سیستم فایل آشنایی زیادی دارید و می توانید از آن استفاده کنید. جدول 1 - 2 فضاهای مناسبی را برای تماشا پیشنهاد کرده گرچه بسیاری از اسم ها به سیستم وابسته است.

`bin /` را که قبلا دیده ایم . آن یک دایرکتوریست و جایی قرار می گیرد که برنامه های اصلی نظیر `who` و `ed` قرار می گیرند. `dev (devies /` که در بخش بعدی در موردش بحث خواهیم کرد.

`etc (et ceterw /` که آن را قبلا دیده ایم که شامل فایل های اداره کننده متعددی نظیر `passwd` و بسیاری از برنامه های سیستمی نظیر `getty / etc` که همانند ابتدای کار ارزش دارد و با

`bin/login /` مرتبط است.

`rc / etc` فایل فرمان های شل است که بعد از سیستم خود راه انداز، اجرا می شود.

`group / etc` تعدادی از این گروه را فهرست بندی کرده است.

`libclibrar / lib / cpp` شامل قسمتهایی از کامپایلر است. مانند

`tmp (temporaries0 /` که مخزنی برای فایل های کوتاه مدت که در طول اجرای برنامه ساخته شده، می باشد. وقتی که شما ویراستار `ed` را راه اندازی می کنید، برای مثال، او فایللی با نام `tmp/e00512` می سازد که تمامی کپی های فایل را در بر دارد. در این صورت شما به جای اینکه با اصل فایل سروکار داشته باشید، فایل را ویرایش کرده اید. البته این فایل می توانست در دایرکتوری جاری شما ساخته شود ولی بهتر است که در `tmp /` قرار گیرد. گرچه ناخوشایند به نظر می رسد. شما ممکن است فایللی به نام `e 00512` در دایرکتوری خود داشته باشید. وقتی سیستم شروع به کار می کند، `tmp /` به طور خودکار پاک می شود. بنابراین اگر سیستم خراب شود، دایرکتوری شما یک فایل ناخواسته به دست نمی آورد. اغلب `tmp /` بر روی دیسک مرتب می شود تا سهل الوصول باشد.

در این جا مشکلی وجود دارد که وقتی در `tmp /` برنامه های متعددی ساخته می شود ممکن است آن ها به فایل های یکدیگر رجوع کند. به همین دلیل فایل موقت `ed` نام ویژه ای دارد و تضمی می شود که برنامه دیگری برای فایل موقت خود نام مشابه آن را انتخاب نکند. در فصل 5 و 6 چگونگی این عمل را خواهیم دید.

`usr /` سیستم فایل کاربر خوانده می شود، گرچه با کاربرهای این سیستم کار زیادی نمی توان انجام داد. در ماشین های ما، دایرکتوری ورودی، `usr / rob /`، `usr / buk /` می باشند. وقتی فایل شخصی شما در دایرکتوری فرعی `usr /` قرار بگیرد، چیزهای زیادی در آنجا وجود دارد که شما دوست دارید آنها را پیدا کنید. در `usr / bin /` دایرکتورهایی به نام های `usr / lib /` و `usr / tmp /` قرار دارد که عملیاتی شبیه به نام های خود را اجرا می کنند و شامل برنامه هایی هستند که کمتر برای سیستم بحران ایجاد می کند. برای مثال `nroff` در `usr / bin /` یافت می شود و نه در `bin /`؛ مجموعه روال `fortRan` نیز در `usr / lib /` قرار دارد. چیزی که باید دانست این است که بحران ها در سیستم های مختلف، متفاوتند. بعضی سیستم ها مانند `7 th Edition` که گسترش یافته نیز هست، تمامی برنامه های `bin /` را داراست و تمامی `usr / bin /` را لغو می کند و از بین می برد. بقیه نیز به خاطر استفاده مکرر `usr / bin /` را به دو دایرکتوری تقسیم می کند.

دایرکتهای دیگر در `usr / adm /`، `usr /` می باشند که شمارش اطلاعات را شامل می شوند.

`usr / dic /` که یک فرهنگ لغت معمولی را در بر دارد. راهنمای خط - وصل، در `usr / man /` نگهداری می شود. برای مثال نگاهی

ببندازید به

`usr / man / man 1 / spen 1.`

اگر سیستم شما، خط - وصل منع کدرا داراست، شما می توانید آن را در `usr/src/` پیدا کنید. صر وقت برای یافتن فایل جدید ارزش زیادی دارد به خصوص در مورد `usr/` که چگونگی سازمان یافتن یک فایل و در کجا قرار داشتن آن را نشان می دهد.

7 - 2 - دستگاه ها :

ما قبلاً گذری بر `dev/` داشته ایم. همان طور که می شود حدس زد `dev/` شامل فایل دستگاه است یکی از ظریف ترین نظریات در مورد سیستم یونیکس این است که با دستگاه جانبی سروکار دارد. مانند دیسک، دستگاه های نوار مغناطیسی ، چاپگر خطی، پایانه و ...
فایلی به نام `dev / mto /` وجود دارد که در داخل هسته، چیزهایی که به این فایل ارجاع داده می شود به یک فرمان سخت افزاری تبدیل می شود تا دستیابی به نوار مغناطیسی را ممکن سازد.
اگر برنامه بخواند `dev / mto /` اطلاعات نوار مغناطیسی که به یک نوار ران نصب شده باز می گردد.

```
$ cp / dev / mto junk
```

فرمان بالا اطلاعات نوار مغناطیسی را بر روی فایلی به نام `junk` کپی می کند. در اینجا در مورد `cp` صحبتی نمی کنیم. روی صحبتمان تنها `dev / mto /` است. این فایلی ست که دارای بایت های هم تراز است.

اولین چیزی که باید بدان توجه داشت اینست که به جای شمارش بایت ها یک جفت عدد صحیح وجود دارد و اینکه اولین کاراکترهای مد همیشه `b` و یا `c` بوده. به این صورت IS اطلاعات را از یک `inode` تایپ می کند و دستگاه را از یک فایل عادی تغییر می دهد. `Inode` های یک فایل عادی شامل لیستی از بلوک های دیسک است. که اطلاعات فایل را ذخیره می کند. برای دستگاه فایل، `inode` به جای اینکه شامل اسم های داخلی برای دستگاه باشد و به جای یک جفت شماره، به صورت مینور (`minor`) و یا میژور (`major`) خوانده می شود. دیسک و نوار مغناطیسی ، دستگاه های بلوک هستند و هر چیز دیگری مثل پایانه و یا چاپگر و خط تلفن، دستگاه کارکتری هستند.

شماره میژور نوع دستگاه را کدگذاری می کند، در حالیکه شماره مینور/ فواصل متفاوت دستگاه را تعیین می کند. برای مثال `dev / ttyo /` و `dev / tty 1 /` دارای دو دهانه در کنترل کننده پایانه است. بنابراین آن ها دارای شماره میژور مساوی اما شماره مینور متفاوتی هستند. نام فایل های دیسک از دگرگونی سخت افزاری گرفته شده است. `dev / rpo 0/` و `dev / rpo 1 /` نام خود را از دیسک گردان `D E C RP06` که وارد سیستم شده است، گرفته اند.

تنها یک نوار دگرگون وجود دارد که به طور منطقی به دو سیستم فایل تقسیم شده است. اگر نوار گردان دومی وجود داشت، فایل های مرتبط خود را اینگونه نام گذاری می کردند.

```
/ dev / rp 10 / dev / rp 11
```

اولین رقم، فعالیت نوار گردان مشخص می کند و دومین رقم بخش های آن را.

ممکن است از خود بپرسید چرا به جای یک فایل دستگاه دیسک، انواع متنوعی از فایل های دستگاه دیسک وجود دارد. به دلایل تاریخی و نگهداری آسان آن، سیستم فایل به دو سیستم فرعی کوچک تبدیل می شود. فایل های داخل سیستم های فرعی به دایرکتری داخل سیستم اصلی دسترسی دارد. برنامه `etc / mount /` بیان می کند که فایل دستگاه و دایرکتری با هم، هم خوانی دارند.

```
$ /etc/mount
rp 01 on /usr
$
```


در این حالت سیستم `dev / rp 00 /` ، `root` را اشغال می کند (که این توسط دایرکتوری های فرعی یش - بر روی `dev / rp 01` ، قرار می گیرد.

سیستم فایل `root` باید به سیستم نشان داده شود، تا اجرا شود. `dev / bin /` و `etc` در سیستم `root` ادامه پیدا می کند. و بسیاری از سیستم های `bin / sh /` باید اجرا شود. در حین این عملیات خود راه انداز، همه سیستم های فایلی چک می شوند و به سیستم فایل می پیوندند این عملیات پیوند `mounting` (نصب کردن) نام دارد. یک نرم افزار با نصب یک بسته دیسک جدید بر دیسک ران برابری می کند که توسط `super - user` انجام می گیرد بعد از اینکه `dev / rp 01 /` به عنوان `usr` نصب شد، فایل در سیستم فایل کاربر قابل دسترسی است گویی که آن ها جزئی از سیستم `root` هستند.

یک مشکلی که وجود دارد اینست که شماره های `inode` در سیستم های فایلی متفاوت، منحصر به فرد نیستند. هر سیستم فرعی دارای سایزها و `inode` های مشخص و معینی است. (شماره بلوک قابل دسترسی در هر فایل). اگر سیستم فرعی پر شود بزرگ کردن فایل در سیستم فرعی غیر ممکن است تا زمانی که مکان جدیدی احیا شود. فرمان `df` (مکان باز دیسک) فضای مناسبی را روی سیستم فایل نصب شده درخواست می کند.

```
$ df
/dev / rp 0 0      1989
/dev / rp 01      21257
```

`usr` ، `21257 /` بلوک آزاد دارد. اینکه آیا این یک فضای وسیع است یا نه به چگونگی استفاده از سیستم بستگی دارد. بعضی از این تأسیسات به فضای بیشتری در فایل نیاز دارند. فرمان `df` تنوع وسیعی در فرمت خروجی دارد. خروجی `df` شما ممکن است کمی متفاوت به نظر برسد. وقتی که شما وارد سیستم عامل می شوید یک خط پایانه و یک فایل `dev /` به دست می آورید که تمامی کارکترهایی را که تایپ کرده و به دست آورده اید را ارسال می کند. فرمان `tty` بیان می کند که شما از کدام پایانه استفاده کرده اید.

```
$ who am 1
you      tty 0      sep 28 01 : 02
$ tty
/dev / ttyo
$ ls -l /dev / ttyo
crw - - w - - w - 1 you      1 , 12 sep 28 02 : 04 / deve / ttyo
$ date > /dev / ttyo
wed sep 28 02 : 40 : 51 Edt 1983
$
```

توجه داشته باشید که شما مالک دستگاه هستید و تنها شما اجازه خواندن آن را دارید به عبارت دیگر هیچ کس دیگری نمی تواند به طور مستقیم کارکترهایی را که تایپ کرده اید بخواند ولی ممکن است بتواند بر روی پایانه چیزی بنویسد. برای جلوگیری از این عمل شما می توانید از `chmod` و یا `mesg` استفاده کنید.

```
$ mesg N
$ ls -l /dev / ttyo
crw - - - - - 1 you
$ mesg y
$
```

می توان توسط نام به پایانه ای که از آن استفاده می کنیم رجوع کرد. دستگاه `dev / tty /` مفهومی برای پایانه ورودی شما محسوب می شود و می توان فهمید که از کدام پایانه استفاده می کنیم.

```
$ date > / dev / tty
wed sep 28 02 : 42 : 23 EDT 1983
```

\$
وقتی که برنامه‌ای به واکنش در مقابل کاربر نیاز دارد می‌توان از `dev / tty /` استفاده کرد حتی اگر ورودی و خروجی استاندارد آن با فایل‌ها در ارتباط باشد. `Crypt` اولین برنامه‌ای است که از `dev / tty /` استفاده می‌کند. یک متن واضح «`clear text`» که از ورودی استاندارد وارد می‌شود و داده‌های رمزگذاری شده به خروجی استاندارد می‌روند بنابراین `crypt` کلید رمز را از `dev / tty /` می‌خواند.

```
$ crypt < clear text > crypted text
```

Enter key :

\$
استفاده از `dev / tty /` در این مثال صریح نیست ولی در آن وجود دارد. اگر `crypt` کلید را از ورودی استاندارد بخواند، پس اولین خط متن پاک «`cleartext`» را می‌تواند بخواند. بنابراین به جای اینکه `crypt` رمز `dev / tty /` را باز کند، کارکترهای خودکار را قطع می‌کند و کلید رمز شما روی صفحه پدیدار نمی‌شود. در فصل 5 و 6 به این مطلب بیشتر اشاره خواهیم کرد.
گاهی شما می‌خواهید برنامه‌ای را اجرا کنید، اما اهمیتی نمی‌دهید که از کدام ورودی استفاده می‌کنید. برای مثال شما به اخبار امروز نگاهی انداخته‌اید و دیگر نمی‌خواهید آن را بخوانید. اگر فرمان `news` را به فایل `dev / nu 11 /` بدهید باعث می‌شود خروجی آن حذف شود.

```
$ news > / dev / nu 11
```

\$
داده‌های نوشته شده در `dev / nu 11 /` بدون هیچ دستوری وقتی حذف می‌شوند که برنامه‌های خوانده شده از `dev / nu 11 /` تمام شود، زیرا خواندن `dev / nu 11 /`، بایت‌ها را صفر می‌کند.

استفاده رایج از `dev / nu 11 /` حذف کردن یک خروجی عادیست. بنابراین پس از آن پیغام خطاشناسی پدیدار می‌شود. برای مثال فرمان `time` کاربرد `cpu` را برای یک برنامه بیان می‌کند. اطلاعات بر روی خطای استاندارد تایپ می‌شود.

```
$ ls -l /usr/dict/words
-r--r--r-- 1 bin
```

```
$ time grep e /usr/dict/words > / dev / nu 11
```

```
real 8.0
```

```
user 3.9
```

```
sys 2.8
```

```
$
```

اعداد در خروجی زمان، سپری شدن زمان را نشان می‌دهند و زمان `cpu` در برنامه سپری می‌شود و هنگامی که برنامه در جریان است زمان `cpu` در هسته سپری می‌شود.

`Egrep` گونه قدرتمندی از `grep` می‌باشد که در بخش 4 در موردش بحث خواهیم کرد و هنگام جستجوی یک فایل بزرگ سرعتش دو برابر `grep` می‌شود. اگر فرم خروجی `grep` و `egrep` به

`dev / nu 11 /` فرستاده نشد ما می‌توانیم منتظر صدها هزار کارکتری باشیم که ظاهر می‌شوند روی پایانه قبل از یافتن اطلاعات زمان.

فصل ۳: استفاده از شل

شل - برنامه‌ای که درخواستهای شما را برای اجرای برنامه‌ها تفسیر می‌کند - مهمترین برنامه برای اکثر استفاده‌کنندگان از یونیکس می‌باشند؛ با استثنای ممکن از ویراستار متن مورد علاقه خود، شما زمان بیشتری را درخصوص کار با شل نسبت به هر برنامه دیگری سپری می‌کنید. در این فصل و در فصل ۵، ما زمان قابل توجهی را به توانایی‌های شل اختصاص می‌دهیم. نکته مهمی که ما می‌خواهیم نشان دهیم این است که شما می‌توانید عملکردهای زیادی را بدون کار خیلی سخت و مطمئناً بدون ذخیره کردن مجدد برای برنامه‌نویسی در یک زبان قراردادی مانند C انجام دهید، اگر شما بدانید که چگونه از شل استفاده کنید.

ما شمول خود درخصوص شل را به دو فصل تقسیم کرده‌ایم. این فصل یک مرحله را فراتر از ضرورت‌های ارائه شده در فصل ۱ برای برخی از ویژگیهای تصویری شل اما عموماً استفاده شده، بیان می‌کند. مانند فراکاراکترها، نقل قول کردن، ایجاد فرمانهای جدید، عبور آرگومان‌ها درون آنها، استفاده از متغیرهای شل و برخی از روند کنترل اولیه. موضوعاتی وجود دارند که شما باید برای استفاده خود از شل بدانید. اطلاعات موجود در فصل ۵، عمیق‌تر بیان می‌شوند - این فصل برای نوشتن برنامه‌های مهم شل اختصاص دارد، برنامه‌هایی که برای استفاده توسط دیگران ضد گلوله می‌باشند. تقسیم بین دو فصل تا اندازه‌ای اختیاری می‌باشد، البته هر دو فصل نهایتاً باید مطالعه شوند.

۳.۱ ساختار سطر فرمان

در ادامه، ما نیاز به اندکی درک بهتر در این خصوص داریم که یک فرمان چیست و چگونه توسط مثل تفسیر می‌شود. این بخش یک شمول رسمی‌تر، با اندکی اطلاعات جدید از مبنای شل معرفی شده در فصل اول می‌باشد.

ساده‌ترین فرمان، تنها یک کلمه است که معمولاً نامیدن یک فایل برای اجرا می‌باشد (بعداً ما انواع دیگری از فرمانها را خواهیم دید):

اجرای فایل /bin/Who

```
$who
You          tty ۲          sep   ۲۸   ۰۷:۵۱
jpl          tty ۴          sep   ۲۸   ۰۸:۳۲
$
```

یک فرمان معمولاً با یک سطر جدید پایان می‌پذیرد، اما یک سمی کالن؛ نیز یک پایان‌نمای فرمان می‌باشد:

```
$ date ;
web sep ۲۸ ۰۹:۰۷:۱۵ EDT ۱۹۸۳
$ date ; who
web sep ۲۸ ۰۹:۲۳ EDT ۱۹۸۳
you          tty ۲          sep ۲۸   ۰۷:۵۱
jpl          tty ۴          sep ۲۸   ۰۸:۳۲
$
```

اگر چه سمی کالنها می‌توانند برای پایان دادن به فرمانها استفاده شوند، اما معمولاً تا زمانی که شما RE TV RN را تایپ نکنید هیچ اتفاقی نمی‌افتد. توجه داشته باشید که شل فقط یک پیامواره را پس از فرمانهای متعدد چاپ می‌کند، اما به جز برای پیامواره،

\$ date ; who

برابر با تایپ دو فرمان بر روی سطرهای متفاوت می باشد. بویژه ، who اجرا نمی شود تا زمانی که date پایان یافته باشد.
ارسال خروجی « date ; who » را درون یک لوله، بررسی کنید :

```
$ date ; who | wc
wed sep ۲۸ ۰۹:۰۸:۴۸ EDT ۱۹۸۳
      ۲          ۱۰          ۶۰
$
```

این نباید چیزی باشد که شما انتظار داشتید، چون فقط خروجی who ، وارد wc می شود. اتصال who و wc با یک لوله، یک فرمان را تشکیل می دهد که خط لوله ای نامیده می شود و پس از date، اجرا می شود. حق تقدم | بیشتر از « ; » می باشد، همچنانکه شل سطر فرمان شما را تجزیه می کند.

پرانتزها می توانند برای گروه بندی کردن فرمانها استفاده شوند :

```
$ (date ; who)
wed sep ۲۸ ۰۹:۱۱:۰۹ EDT ۱۹۸۳
you          tty ۲          sep ۲۸ ۰۷:۵۱
jpl          tty ۴          sep ۲۸ ۰۸:۳۲
$ (date ; who) | wc
      ۳          ۱۴          ۸۹
$
```

خروجیهای date و who ، در یک مسیل منفردی به هم ملحق می شوند که می تواند یک لوله را به پائین ارسال کند.
جریان یافتن داده ها درون یک لوله می تواند ضبط شود و در یک فایل (اما نه در یک لوله دیگر) با فرمان tee قرار گیرد، فرمانی که بخشی از شل نمی باشد، اما با این وجود، قابل استفاده برای لوله های انجام عملیات می باشد. یک استفاده از آن، ذخیره کردن خروجی واسطه در یک فایل می باشد :

```
$ (date ; who) | tee save | wc
      ۳          ۴          ۸۹          خروجی از wc
$
```

```
$ cat save
wed sep ۲۸ ۰۹:۱۳:۲۲ EDT ۱۹۸۳
you          tty ۲          sep ۲۸ ۰۷:۵۱
jpl          tty ۴          sep ۲۸ ۰۸:۳۲
$ wc < save
      ۳          ۱۶          ۸۹
$
```

tee از ورودی خود برای فایل یا فایل های نامگذاری شده و نیز برای خروجی خود کپی برمی دارد، در نتیجه wc همان داده ها را دریافت می کند، گویا اینکه tee در خط لوله نمی باشد.

پایان نمای دیگر فرمان، آپرساند \$ می باشد. این پایان نما، دقیقاً شبیه سمی کالن یا سطر جدید می باشد، به جز اینکه این علامت به شل می گوید که منتظر کامل شدن فرمان نباشد. اساساً، \$ برای اجرای یک فرمان طولانی مدت « در زمینه » استفاده می شود، همچنانکه

شما تایپ فرمانهای محاوره‌ای را ادامه می‌دهید :

طولانی مدت فرمان

\$ long – running - command

\$ id long – running – commend - پردازش

پیامواره سریعاً ظاهر می‌شود

۵۲۷۳

\$

با توجه به توانایی برای گروه بندی کردن فرمانها، استفاده‌های جالب دیگری از پردازش‌های زمینه وجود دارد. فرمان sleep ، چند ثانیه مشخص قبل از خروج، منتظر می‌ماند :

\$ sleep 5

\$

چنج ثانیه قبل از پیامواره طی می‌شود

\$ (sleep 5 ; date) \$ date

5278

wed sep 28 09 : 18 : 20 EDT 1983

خروجی از date دوم

\$ wed sep 28 09 : 18 : 25 EDT 1983

پیامواره ظاهر می‌شود، سپس

ثانیه، بعداً می‌آید ۵ date

فرآیند زمینه آغاز می‌شود، اما سریعاً به خواب می‌رود؛ در این میان، فرمان date دوم، زمان جاری و پیامواره‌های شل را برای یک فرمان جدید پرینت می‌کند. پنج ثانیه بعد، sleep خارج می‌شود و اولین date، زمان جدید را پرینت می‌کند. نشان دادن عبور زمان بر روی کاغذ دشوار می‌باشد، در نتیجه شما باید این مثال را بررسی کنید. (بر اساس اینکه ماشین شما چگونه مشغول است و نیز بر اساس سایر جزئیات ، تفاوت بین دو زمان، ممکن است دقیقاً 5 ثانیه نباشد).

این یک روش آسان برای اجرای یک فرمان در آینده می‌باشد؛ در نظر بگیرید

\$ (sleep 300 ; echo Tea is ready) \$ Tea will be ready in smunutes

5291

\$

به عنوان یک مکانیسم باقیمانده قابل استفاده می‌باشد . (یک ctl-g در رشته، که پژواک شده باشد، زنگ پایانه را به صدا در می‌آورد زمانی که پرینت می‌شود). پراترها در این مثالها لازم هستند، چون حق تقدم \$ بیشتر از « ; » می‌باشد.

پایان نمای \$ برای فرمانها بکار می‌رود و چون خطوط لوله‌ای فرمانها، هستند در نتیجه شما نیاز به پراترها برای اجرای خطوط لوله‌ای در زمینه ندارید:

\$ pr file | lpr \$

برای پرینت فایل بر روی چاپگر سطر ترتیب داده می‌شود بدون اینکه شما را برای خاتمه یافتن فرمان منتظر بگذارد. وارد پراتر کردن خط لوله‌ای نیز دارای همین اثر می‌باشد، اما نیازمند تایپ بیشتر است :

\$ (pr file | lpr) \$

همانند مثال قبل

اکثر برنامه‌ها، آرگومانها را روی سطر فرمان می‌پذیرند، مانند فایل (یک آرگومان برای pr) در مثال بالا. آرگومانها، کلمات هستند، توسط

فاصله‌ها و جدول بندیها از هم جدا می‌شوند، و اساساً فایل‌هایی را که باید توسط فرمان پردازش شوند، نامگذاری می‌کنند، اما آنها رشته‌هایی هستند که می‌توانند به هر روشی که برنامه متناسب به نظر می‌رسد، تفسیر شوند. برای مثال، `pr`، اسامی فایلها برای پرینت را می‌پذیرد، پژواک آرگومانهای خود را بدون تفسیر پژواک می‌کند و اولین آرگومان `grep` یک نمونه متن را برای جستجو مشخص می‌کند. و البته، اکثر برنامه‌ها نیز دارای انتخاب‌هایی هستند که توسط آرگومان‌هایی نشان داده می‌شوند که با علامت منها آغاز می‌شوند. کاراکترهای متعدد خاص که توسط شل تفسیر می‌شوند مانند `<`، `>`، `|`، `;` و `$`. آرگومانهای برنامه‌هایی نیستند که شل اجرا می‌کند. در عوض آنها، کنترل می‌کنند. که چگونه شل آنها را اجرا می‌کند. برای مثال،

```
$ echo hello > junk
```

به ما می‌گوید که شل، پژواک را تنها با آرگومان `Hello` اجرا می‌کند و خروجی را در `junk` فایل قرار می‌دهد. `String > junk`، یک آرگومان برای پژواک نمی‌باشد؛ بلکه توسط شل تفسیر می‌شود و هرگز توسط پژواک مشاهده نمی‌شود. در حقیقت، این عبارت نباید آخرین رشته در فرمان باشد:

```
$ > junk echo Hello
```

نیز شبیه به آن می‌باشد اما کمتر بدیهی است.

تمرین ۳.۱. تفاوت‌های میان سه فرمان زیر چه هستند؟

```
$ cat file | pr
```

```
$ pr < file
```

```
$ pr file
```

(در طی سالها اپراتور جهت دهی مجدد `<`، تا حدودی زمینه خود را برای لوله‌ها از دست داده است؛ به نظر می‌رسد افراد طبیعتاً بیشتر از `<file` به دنبال `<cat file` می‌باشند.)

۳.۲ فرا کاراکترها

شل، تعدادی دیگر از کاراکترهای خاص را تشخیص می‌دهد؛ و عمومی‌ترین کاراکتر مورد استفاده، کاراکتر ستاره `*` می‌باشد که به شل می‌گوید که به جستجوی فهرست برای اسامی فایل‌هایی پردازد که در آنها هر رشته از کاراکترها در موفقیت `*` رخ می‌دهد. برای مثال،

```
$ echo*
```

یک پیام نمای ضعیف از `ls` می‌باشد. چیزی که ما را فصل اول ذکر نکردیم این است که کاراکترهای تطبیق دهنده نام فایل، به بررسی اسامی فایل‌هایی که با یک نقطه شروع می‌شوند نمی‌پردازند، و این به خاطر اجتناب از مشکلات در خصوص اسامی `« . »` و `« .. »` می‌باشد که در هر فهرستی وجود دارند.

دستور این است: کاراکترهای تطبیق دهنده اسم فایل فقط اسامی فایل‌هایی را تطبیق دهند که با یک دوره شروع می‌شوند، اگر دوره آشکارا در نمونه ذخیره می‌شود. به طور معمول، یک `echo` یا دو `echo` درست چیزی را که رخ می‌دهد، شرح می‌دهند:

```
ls$
```

```
.Proflie
```

```
Junk
```

```
Temp
```

```
$ echo *
```

```
Junk Temp
```

```
$ echo . *
```

```
. .. profile
```

```
$
```

کاراکترهای شبیه ستاره * که دارای ویژگیهای خاص می باشند، فراکاراکتر نامیده می شوند. تعداد زیادی از آنها وجود دارد: جدول ۳.۱ یک فهرست کامل است، اگر چه تعداد کمی از آنها تا فصل ۵ مورد بررسی قرار نمی گیرند.

با توجه به تعداد فراکاراکترها، راههایی برای جواب دادن به شکل وجود دارد، « آن را تنها بگذارید »، بهترین و آسانترین راه برای حمایت کاراکترهای خاص از تفسیر شدن، ضمیمه کردن آنها در کاراکترهای نقل قولی منفرد می باشد:

```
$ echo '***'
```

```
***
```

```
$
```

همچنین این امکان وجود دارد که از نقل قولهای دو برابر «...» استفاده کنیم، اما شل، حقیقتاً درون این نقل قولها را برای جستجوی \$ ، «...» و \ می خواند، بنابراین «...» استفاده نمی کند، مگر اینکه شما قصد پردازش رشته نقل قول شده را داشته باشید.

سومین راه ممکن، قرار دادن یک پس کیج خط \ در جلوی هر کاراکتری می باشد که می خواهید آن را از شل محافظت کنید، مانند

```
$ echo \*\**
```

اگر چه *** ، خیلی انگلیسی نمی باشد، اما واژگان شل برای آن، هنوز یک کلمه است که هر رشته منفردی می باشد که شل آن را به عنوان یک واحد می پذیرد، همراه با فاصله ها، البته اگر آنها نقل قول شوند.

نقل قولهای یک نوع ، از نقل قولهای نوع دیگر حمایت می کنند:

```
$ echo " Don't do that !"
```

```
Don't do that
```

```
$
```

```
$ echo X * 'y
```

```
X * y
```

```
$ echo '* A' P '
```

```
* A?
```

```
$
```

جدول ۳.۱: فراکاراکترهای شل

> Prog >	فایل، خروجی استاندارد را به فایل هدایت می کند
>> Prog >	فایل، خروجی استاندارد را به فایل ضمیمه می کند
< Prog <	فایل، ورودی استاندارد را از فایل می گیرد
P ₂ P ₁	خروجی استاندارد P ₁ را به ورودی استاندارد
<<str	ورودی استاندارد تا Str بعدی روی یک خط توسط خودش : سند در اینجا ادامه می یابد.
*	هر رشته از صفر تا چند کاراکتر را در اسامی فایلها تطبیق می دهد
?	هر کاراکتر منفرد را در اسامی فایلها تطبیق می دهد
[ccc]	هر کاراکتر منفرد را در اسامی فایلها تطبیق می دهد ؛
	گستره هایی شبیه 0_9 یا a-z مجاز هستند
;	پایان نمای فرمان : P ₂ ؛ P ₁ ، P ₁ را انجام دهد و سپس P ₂ را.

\$	شبیبه ; می باشد، اما تا اتمام P ₂ منتظر نمی ماند.
'...'	دستورات را در ...؛ اجرا می کند خروجی را جایگزین '...' می کند
(...)	دستورات را در ... اجرا می کند، در یک زیر شل
{...}	دستورات را در ... اجرا می کند، در شل جاری (به ندرت استفاده می شود)
\$1 , \$2 etc.	\$0 ... \$4 توسط آرگومانها برای فایل شل جایگزین می شود
\$ var	مقدار متغیر شل var
\${var}	مقدار var؛ از بی نظمی اجتناب می کند، زمانی که به متن محلوق می شود؛ همچنین به جدول ۵.۳ مراجعه کنید.
\	\C کاراکتر C را به صورت حرفی می گیرد، \ سطر جدید، حذف می شود
«...»	... را به صورت حرفی پس از \$، '...' و \ تفسیر شده، می گیرد،
#	اگر # کلمه را آغاز کند، مابقی سطر یک توضیح می باشد (در هفتمین ویرایش وجود ندارد)
Var = value	به متغیر var نسبت داده می شود.
P ₁ \$\$ P ₂	P ₁ ؛ اجرا می شود، اگر موفق بود، P ₂ اجرا می شود
P ₁ P ₂	P ₁ ؛ اجرا می شود، اگر ناموفق بود، P ₂ اجرا می شود.

در مثال اخیر، چون نقل قولها، پس از اینکه کار خود را انجام دادند حذف می شوند، در نتیجه echo به صورت یک آرگومان منفرد به نظر می رسد که فاقد نقل قول می باشد.

رشته های نقل قول شده می توانند شامل سطرهای جدید باشند :

```
$ echo 'hello
> world'
hello
world
$
```

رشته «>» یک پیامواره ثانویه چاپ شده توسط شل می باشد. زمانی که از شما انتظار می رود که ورودی بیشتری را برای کامل کردن یک فرمان تایپ کنید. در این مثال : نقل قول روی اولین سطر باید با نقل قول دیگر متوازن شود. رشته پیامواره ثانویه در متغیر ps₂ ذخیره می شود و می تواند بر طبق سلیقه تغییر داده شود.

در همه این مثالها، نقل قول یک فرا کاراکتر، مانع تلاش شل برای تفسیر آن می شود. فرمان

```
$ echo x * y
```

همه اسامی فایل هایی را که با x آغاز و با y پایان می پذیرند، پژواک می کند. مثل همیشه، پژواک چیزی در مورد فایلها یا فرا کاراکترهای شل نمی داند؛ تفسیر *، اگر وجود داشته باشد، توسل شل ذخیره می شود.

چه اتفاقی می افتد، اگر هیچ فایل با نمونه تطبیق داده نشود؟ شل به جای شکایت کردن (همچنانکه در نسخه های قبلی این کار را انجام

داد)، از رشته عبور می کند، گویا اینکه نقل قول شده است. معمولاً این عقیده بدی است که به این رفتار تکیه کنیم، اما چنین رفتاری می تواند برای یادگیری وجود فایل هایی که یک طرح را تطبیق می کنند، استفاده شود :

```
$ ls x * y
x * y          یافت نمی شود
$ > xyzzy
xyzzy          وجود می آید
$ ls x *y
xy zzy        فایل xyzzy، با x*y تطبیق می شود
$ ls 'x * y'
'x * y'       *، ls تفسیر نمی کند
x * y          یافت نمی شود
$
```

یک پس کج خط در پایان یک سطر، باعث می شود که سطر ادامه یابد؛ و این یک روش برای نشان دادن یک سطر بسیار طولانی در شل می باشد.

```
$ echo abc \
> def \
> ghi
a b c d e f g h i
$
```

توجه داشته باشید که سطر جدید حذف می شود، زمانی که قبل از پس کج خط می آید، اما زمانی که به صورت نقل قول ظاهر می شود، باقی می ماند.

فرا کاراکتر # تقریباً به صورت جهانی برای توضیحات شل استفاده می شود؛ اگر یک کلمه شل با # آغاز شود، ما بقی سطر نادیده گرفته می شود :

```
$ echo hello # there
hello
$ echo hello # there
hello # there
$
```

بخشی از هفتمین ویرایش اصلی نبود، اما عمیقاً پذیرفته شده است و ما از آن در مابقی کتاب استفاده خواهیم کرد. تمرین ۳.۲ خروجی تهیه شده توسط * \$ ls را شرح دهید.

یک انحراف بر روی پژواک

اگر چه صراحتاً مورد پی گیری واقع نمی شود، اما یک سطر ج دید نهایی، توسط پژواک تهیه می شود. یک طرح محسوس و شاید تمیزتر برای پژواک، پرینت کردن فقط چیزی است که درخواست می شود. چنین چیزی، انتشار پیامواره ها را از شل آسان می سازد :

\$ pure - echo Enter یک فرمان :

سطر جدید پسین وجود ندارد \$: یک فرمان Enter

اما عیب آن ، این است که عمومی ترین مورد - تهیه یک سطر جدید - پیش فرض نمی باشد و نیاز به تایپ بیشتر دارد :

```
$ pure - echo 'Hello !
```

```
>'
```

```
Hello !
```

```
$
```

چون یک فرمان باید با پیش فرض عمومی ترین عملکرد استفاده شده خود را اجرا کند، پژواک واقعی، به طور خودکار ضمیمه سطر جدید نهایی می شود.

اما اگر مطلوب نباشد چه پیش می آید؟ هفتمین ویرایش پژواک دارای یک اختیار منفرد، n- و برای حذف آخرین سطر جدید می باشد :

\$ echo - n Enter : یک فرمان

پیامواره بر روی همان سطر \$ Enter : یک فرمان

```
$ echo -
```

خاص می باشد ، n- فقط

```
$
```

: می باشد که توسط یک سطر جدید دنبال می شود n- تنها مورد دشوار ، پژواک

```
$ echo - n' - n
```

```
>'
```

```
- n
```

```
$
```

این حالت بدترکیب است اما کار می کند و به هر حال این یک موقعیت نادر است. یک روش متفاوت ، ارائه شده در سیستم V ، برای پژواک در خصوص تفسیر C می باشد - شبیه توالیهای پس کج خط ، مانند \b برای پسبرد و \c (که حقیقتاً در زبان c نمی باشد) برای حذف آخرین سطر جدید :

نسخه سیستم V \C' : یک فرمان \$ echo 'Enter

\$: یک فرمان Enter

اگر چه این مکانیسم از بی نظمی درخصوص پژواک یک علامت منها اجتناب می کند، اما دارای مشکلات دیگر می باشد. پژواک اغلب به عنوان یک راهنمای تشخیص استفاده می شود و پس کج خطها توسط تعداد زیادی از برنامه هایی تفسیر می شوند که پژواک آنقدر به بررسی آنها می پردازد که به بی نظمی اضافه می کند.

با این حال، هر دو طرح پژواک دارای نکات خوب و بد می باشند. ما از هفتمین نسخه ویرایش (n-) استفاده می کنیم، بنابراین اگر پژواک محلی شما از یک قرارداد متفاوت اطاعت کند، یک جفت از برنامه های ما نیاز به تجدید نظر جزئی دارند.

پرسش دیگر در خصوص این نظریه این است که پژواک چه باید انجام دهد اگر هیچ آرگومانی ارائه نشود - بویژه آیا باید یک نسخه خالی را پرینت کند و یا اینکه اصلاً چیزی پرینت نکند؟ همه پیاده سازیهای رایج پژواک را که ما می شناسیم، یک سطر خالی پرینت می کنند، اما نسخه های قبلی چنین کاری را انجام نمی دادند و زمانی مباحثات زیادی درخصوص این موضوع وجود داشت. داگ مککروی ، احساسات واقعی عرفانی را در بحث و بررسی خود درخصوص موضوع ارائه داد :

یونیکس و پژواک

یونیکس در سرزمین نیوجرسی ساکن بود، یک خدمتکار زیبایی که دانشمندان از جاهای دور برای تعریف و تمجید از او مسافرت می کردند. همه مبهوت از عفت و پاکدامنی او، در صدد ازدواج با او بودند، یکی به خاطر زیبایی دخترانه او، دیگری برای فرهنگ آراسته او، اما فردی به خاطر زرنگی او در جرای دقیق وظایفی که به ندرت انجام می شد، حتی در سرزمینهای ثروتمندتر. بنابراین قلب پر احساس و خوش برخوردی یونیکس بود که همه خواستگاران غیرقابل تحمل خود را می پذیرفت. به زودی بسیاری از فرزندان بزرگ شدند و موفق شدند در تمام زمین پخش شدند.

طبیعت لبخند زد و مشتاق تر از سایر مخلوقات به یونیکس جواب داد. مخلوقات متواضع تر که اندکی از رفتارهای آراسته تر می دانستند، از پژواک او خوشحال می شدند، بنابراین آنقدر واضح و روشن بود که آنها معتقدند که او می تواند از همان سنگ ها و چوبهایی پاسخ بشنود که فریادهای آنها را در میان صحرا خراب و مخدوش می کردند. و یونیکس مطیع، مجبور به اجرای پژواکهای کامل از هر چیزی بود که از او درخواست می شد.

زمانی که یک عاشق بی صبر از یونیکس درخواست می کرد «چیزی را پژواک نکن» یونیکس از روی محبت دهانش را باز میکرد، چیزی را پژواک نمی کرد و دوباره دهانش را می بست. جوانان به او می گفتند؛ منظور شما چیست که دهانت را اینگونه باز می کنی؟ از این به بعد، هرگز دهانت را باز نکن زمانی که نمی خواهی چیزی را پژواک کنی! و یونیکس اطاعت کرد.

یک جوان حساس خواهش کرد؛ اما من یک عملکرد کامل می خواهم، حتی وقتی که شما چیزی را پژواک نمی کنی و پژواکهای کامل نمی توانند از یک دهان بسته خارج شوند. یونیکس نمی خواست حتی یک نفر را ناراحت کند و پذیرفت که چیزهای متفاوتی را برای جوان بی صبر و جوان حساس بیان کند. او جوان حساس را با «n\» فرا می خواند.

زمانی که او «n\» را بیان می کرد، او حقیقتاً چیزی نمی گفت بنابراین باید دهانش را دوبار باز می کرد یک بار برای گفتن «n\» و بار دیگر برای گفتن هیچ چیزی، و بنابراین او جوان حساس را راضی نکرد و جوان حساس بلافاصله گفت، صداهای n\ یک پژواک کامل برای من هستند اما بار دوم آن را خراب می کند. من از تو می خواهم که یکی از آنها را انتخاب کنی، بنابراین یونیکس که نمی توانست با ایجاد دردسر و مزاحمت زندگی کند، موافقت کرد که برخی از پژواکها را از بین ببرد و آن را «c\» بنامد. اکنون جوان حساس می توانست با درخواست «c\» و «n\» با هم، یک پژواک کامل از چیزی داشته باشد. اما آنها میگویند که او به خاطر تعداد زیاد نمادگذاری ها فوت کرد، قبل از اینکه او حتی یکی از آنها را بشنود.

تمرین ۳.۳. پیش بینی کنید که کدام یک از فرمانهای grep زیر انجام خواهند شد، سپس درک خود را عملاً بسنجید.

\\	grep	\\$ grep
\\ \\	grep	\\ \\$ grep
" \\$ \"	grep	\\ \\$ grep
' \$ \"	grep	' \\$ ' grep
" \$ \"	grep	' \\$ ' grep

فایلی که حاوی این فرمانها باشد، یک مورد آزمون خوبی می باشد، اگر شما بخواهید امتحان کنید.

تمرین ۳.۴. شما چگونه به grep می گوید که به جستجوی یک طرحی پردازد که بل «-a» آغاز می شود؟ چرانقل قول کردن

آرگومان کمک نمی کند؟ نکته : در مورد انتخاب C- تحقیق کنید.

تمرین ۳-۵ توجه کنید به

\$ echo */*

آیا این فرمان همه اسامی را در همه جهتها تهیه می کند؟ اسامی بر طبق چه نظمی آشکار می شوند؟

تمرین ۳-۶ (پرسش راهکار). شما چگونه یک / را در یک اسم فایل قرار می دهید (یعنی ، یک / که قطعات مسیر را جدا نمی کند)؟

تمرین ۳-۷ با فرمان

\$ cat xy / y

و با فرمان

\$ cat x >> x

چه اتفاقی رخ می دهد؟

قبل از اینکه به بررسی آنها پردازید فکر کنید.

تمرین ۳-۸ اگر شما فرمان

\$ rm *

را تایپ کنید، چرا rm نمی تواند به شما اخطار دهد که شما در حال حذف همه فایل های خود هستید؟

۳.۳ ایجاد فرمانهای جدید

اکنون وقت آن رسیده است که به سراغ چیزی برویم که در فصل اول قول دادیم - چگونه فرمانهای جدید را خارج از فرمانهای قبلی ایجاد کنیم.

با توجه به یک توالی از فرمانهایی که بیشتر از چندین بار تکرار می شوند، بهتر است که این توالی را در یک فرمان جدید با نام خودش قرار دهیم، در نتیجه شما می توانید از آن به عنوان یک فرمان منظم استفاده کنید. برای روشن شدن این موضوع، فرض کنید که قصد شمارش کاربرها را غالباً با خط لوله ای

\$ who | wc - l

دارید که در فصل اول ذکر شد و شما می خواهید یک برنامه جدید nu را برای انجام آن بوجود آورید.

اولین مرحله ایجاد یک فایل معمولی است که شامل ' who | wc - l ' می باشد شما می توانید از یک ویراستار مطلوب استفاده کنید و یا می توانید از قوه خلاقیت استفاده کنید :

\$ echo ' who | wc - l ' > nu

(بدون نقل قولها ، چه چیزی در nu ظاهر می شود؟)

همانگونه که در فصل اول بیان کردیم ، شل یک برنامه شبیه یک ویراستار یا who و یا wc می باشد؛ و نام آن sh است. و چون یک برنامه است، شما می توانید آن را اجرا کنید و به ورودی آن مجدداً جهت دهید. بنابراین شل را با ورودی منتج از فایل nu آن به جای پایانه، اجرا کنید:

\$ who

07:51	28	sep	2 you	tty
10:02	28	sep	4 you	tty
09:38	28	sep	5 you	tty

```
10:17 28      sep      4 tty      you
$ cat na
Who : wc -l
$ sh>nu
4
$
```

خروجی شبیه همان چیزی خواهد بود که در هنگام تایپ `who | wc -l` در پایانه می باشد.

دوباره همانند سایر - برنامه ها، شل ورودی خود را از یک فایل می گیرد، اگر فایلی به عنوان یک آرگومان نامگذاری شود ؛ شما می توانید برای رسیدن به همان نتیجه بنویسید

```
$ sh nu
```

اما تایپ کردن « sh » در هر مورد، درد سر و زحمت می باشد:

چون طولانی تر است و تمایزی را بین برنامه های نوشته شده در `c`، `say` و برنامه های نوشته شده از طریق اتصال برنامه ها با شل، بوجود می آورد. بنابراین، اگر یک فایل اجراپذیر باشد و اگر شامل متن باشد، در نتیجه شل، آن را به عنوان یک فایل از فرمانهای شل تصور می کند. چنین فایلی، فایل شل نامیده می شود.

همه کاری که شما باید انجام دهید، اجرا پذیر کردن `nu` می باشد و در ابتدا:

```
$ chmod +x nu
```

و پس از آن شما می توانید آن را با `$ nu`، راه اندازی کنید.

از اکنون کاربرهای `nu` فقط با اجرا کردن آن نمی توانند بگویند که شما آن را به این روش آسان پیاده کنید.

روشی که شل حقیقتاً `nu` را اجرا می کند، ایجاد یک فرآیند جدید شکل دقیقاً به گونه ای که شما `$ sh nu` را تایپ کرده اید می باشد. این شل مولود، زیر شل نامیده می شود - یک فرآیند شلی که توسط شل جاری شما راه اندازی می شود. `Shnu`، همانند `sh<nu` نمی باشد، چون ورودی استاندارد آن، هنوز متصل به پایانه می باشد.

همانگونه که هست، `nu` فقط زمانی که کار می کند که در فهرست جاری شما باشد (البته مشروط بر اینکه فهرست جاری در مسیر شما باشد، که از حالا به بعد ما اینگونه تصور می کنیم). برای ایجاد `nu` به عنوان بخشی از مجموعه عملکردهای خود، بدون توجه به اینکه شما در کدام فهرست قرار دارید، به فهرست `bin` اختصاصی خود وارد شوید و `/usr/you/bin/` را به مسیر جستجوی خود اضافه کنید:

```
pwd $      usr/you u /
```

بسازید اگر قبلا این کار را انجام نداده اید `bin` یک

```
$ mkdir bin
```

مسیر را برای اطمینان کنترل کنید

```
echo $ PATH $
```

باید اینگونه باشد

```
usr/you/bin:/bin:/usr/bin /:
```

را نصب کنید `nu`

```
mv nu bin $
```

```
$ ls nu
```

یافت نمی شود

`nu` حقیقتاً از فهرست جاری می آید

```
$ nu
```

4

اما توسط شل یافت می شود

```
$
```

البته، مسیر `PATK` شما باید به طور صحیح با برش عرضی `profile` شما متناسب باشد،

بنابراین شما نباید هرزمانی که به سیستم وارد می‌شوید، آن را ریست سازی کنید. فرمانهای ساده دیگری وجود دارند که شما می‌توانید به این روش برای مناسب کردن محیط خود بر طبق سلیقه خود بوجود آورید. برخی از فرمانهایی که به نظر ما مناسب می‌باشند، شامل موارد زیر هستند:

Cs0 که توالی صحیح کاراکترهای مرموز را برای پاک کردن نمایشگر روی پایانه شما پژواک می‌کند (۲۴ سطر جدید یک اجرای کلی مطلوب می‌باشد)؛

What 0 ، که who و ps - a را برای بیان اینکه چه کسی به سیستم وارد می‌شود و چه کاری انجام می‌دهد اجرا می‌کند؛

Where 0 که شناسایی اسم سیستم یونیکسی را که شما از آن استفاده می‌کنید پرینت می‌کند - این فرمان قابل استفاده می‌باشد اگر شما به طور منظم از آن استفاده کنید. (تنظیم ps1 نیز به همین منظور می‌باشد).

تمرین ۳.۹ . به فرمان `bin./usr/bin/` مراجعه کنید برای اینکه ببینید چند فرمان حقیقتاً فایل‌های شل می‌باشند. آیا می‌توانید این کار را با یک فرمان انجام دهید؟ نکته : فایل (1). حدسیات بر اساس طول فایل چقدر صحیح می‌باشند؟

۳.۴ . پارامترها و آرگومانهای فرمان

اگر چه `nu` همانگونه که هست، مناسب می‌باشد، اما اکثر برنامه‌های شل، آرگومونها را تفسیر می‌کنند، در نتیجه برای مثال، اسامی فایلها و انتخابها می‌توانند خاص باشند زمانی که برنامه اجرا می‌شود.

فرض کنید ما می‌خواهیم برنامه‌ای را بسازیم که `cx` نامیده می‌شود برای اینکه حالت یک فایل را برای قابل اجرا شدن تغییر دهیم، بنابراین

`cx nu $`

یک صورت مختصر برای

`chomd+xnu $`

می‌باشد، ما به میزان کافی برای انجام چنین چیزی اطلاعات داریم. ما به یک فایل با نام `cx` نیاز داریم که محتوای آن

`Chomd + x filename`

می‌باشد. تنها مورد جدیدی که ما باید بدانیم این است که چگونه به `cx` بگوییم که نام فایل چیست، چون در هر زمانی که `cx` اجرا می‌شود، نام فایل متفاوت است. زمانی که شل، یک فایل از فرمانها را اجرا می‌کند، هر رخداد از `$1`، توسط اولین آرگومان جایگزین می‌شود، هر `$2` توسط دومین آرگومان جایگزین می‌شود و به همین ترتیب تا `$9` ادامه می‌یابد. بنابراین اگر فایل `cx` شامل

```
Chomd+x $1
```

باشد، زمانی که فرمان `cx nu` اجرا می‌شود، زیر شل، `" $1 "` را با اولین آرگومان خود یعنی `« nu »` جایگزین می‌کند.

اکنون می‌خواهیم به توالی کل عملکردها نگاهی بیندازیم:

```
$ echo chomd + x $1' >cx
```

```
$ sh cx cx
```

```
$ echo echo Hi, there !>hello
```

```
$ hello
```

```
hello :
```

```
$ cx hello
```

```
$ hello
```

کار می‌کند `Hi, there! Cx` را نصب کنید

```
$ rm hello
```

توجه داشته باشید

```
$
```

`Cx` را به شکل اولیه ایجاد کنید

`Cx` را قابل اجرا کنید

یک برنامه آزمایشی ایجاد کنید

آن را بررسی کنید

نمی‌تواند اجرا شود

آن را قابل اجرا کنید

دوباره بررسی کنید

کار می‌کند

```
$ mv CX/urr/you/bin
```

پاکسازی کنید

که ما گفتیم

```
$ sh cx cx
```

دقیقاً همانگونه که شل به طور خودکار انجام می‌داد، اگر `CX` قابل اجرا بوده و ما تایپ کردیم

```
$ cx cx
```

اگر شما بخواهید با بیش از یک آرگومان کار کنید، چه باید کرد؟ برای مثال ایجاد برنامه‌ای مانند `cx` که در یک زمان با چندین فایل کار می‌کند. اولین برش ناقص، قراردادن `۹` آرگومان در برنامه شل می‌باشد، مانند

```
Chomd+x$1 $2 $3 $4 $5 $6 $7 $8 $9
```

(این فرمان فقط تا `$9` کار می‌کند، چون رشته `$10` به صورت اولین آرگومان `$1` و `0` تجزیه می‌شود!). اگر کار بر این فایل شل، کمتر از نه آرگومان را فراهم کند، تعداد آرگومانهای مفقوده، رشته‌های تهی هستند؛ تأثیر آن، این است که فقط آرگومانهایی که حقیقتاً تهیه

شده‌اند توسط زیر شل از `chomd` عبور می‌کنند. بنابراین این اجرا کار می‌کند، اما به طور بدیهی کثیف است و خراب می‌شود، اگر بیش از نه آرگومان تهیه شود.

با پیش‌بینی این برنامه، یک صورت مختصر از `* $` را فراهم می‌کند که به معنای «همه آرگومانها» می‌باشد. روش صحیح برای تعریف `cx` به صورت `* $ + chomd x` می‌باشد که بدون توجه به اینکه چند تا آرگومان تهیه می‌شوند، کار می‌کند. با افزودن `* $` به مجموعه عملکردهای خود، شما می‌توانید فایل‌های شل مناسبی را تهیه کنید مانند `lc` یا `m`:

```
$ cd/usr/you/bin
```

```
$ cat lc
```

تعداد سطرها را در فایلها بشمارید

```
1 # c:
```

```
wc -l * $
```

```
$ cat m
```

یک روش مختصر برای تایپ پست

```
: #m
```

```
mail $*
```

```
$
```

هر دو می‌توانند به طور مناسب بدون آرگومانها استفاده شوند. اگر هیچ آرگومانی وجود نداشته باشد، `* $` تهی خواهد بود و اصلاً هیچ آرگومانی از `wc` یا `mail` عبور نخواهد کرد. با وجود آرگومانها یا بدون آنها، فرمان به طور صحیح راه‌اندازی می‌شود:

```
$ ls /usr/you/bin/*
```

```
usr/you/bin/cx 1
```

```
/usr/you/bin/lc 2
```

```
usr/you/bin/m 2
```

```
/usr/you/bin/nul
```

```
2/usr/you/bin/what
```

```
1/usr/you/bin/where
```

```
9 total
```

```
$ ls /usr/you/bin | lc
```

```
4
```

```
$
```

این فرمانها و سایر فرمانها موجود در این فصل، مثالهایی از برنامه‌های شخصی، نوع چیزهایی که شما برای خود می‌نویسید و در `bin` خود قرار می‌دهید، می‌باشند، اما بعید به نظر می‌رسد که طور عمومی در دسترس باشند، چون آنها بسیار وابسته به سلیقه شخصی می‌باشند. در فصل ۵، ما موضوعات مربوط به نوشتن برنامه‌های شل را که برای استفاده عمومی مناسب می‌باشند، مورد بحث قرار می‌دهیم.

آرگومانها برای یک فایل شل نباید اسامی فایلها باشند. برای مثال، جستجوی یک فهرست راهنمای تلفن شخصی را در نظر بگیرید. اگر شما دارای فایلی باشید که به صورت `/usr/you/bib/phone-book` نامگذاری شده باشد و شامل سطرهایی مانند زیر باشد

```
dial - a - joke 2 - 976 - 3838
```

```
ial - a - prayer 212 - 246 - 4200
```

```
ial santa 212 - 976 - 3636
```

212 - 976 - dow Jones report 4141

در نتیجه فرمان `grep` می تواند برای جستجوی آن استفاده شود. (فهرست `lib` شما یک مکان خوب برای ذخیره بانک اطلاعاتی شخصی می باشد). چون `grep` به فرصت اطلاعات اهمیتی نمی دهد، شما می توانید به جستجوی اسامی، آدرس ها، رمزهای پستی و یا هر چیزی که دوست دارید، بپردازید. می خواهیم یک برنامه فهرست مشارکت بسازیم و آن را به افتخار شماره مشارکت فهرست تلفن جایی که در آن زندگی می کنیم ۴۱۱ بخوانیم:

```
411$ <' echo 'grep $* /usr/you/lib/phone - book
411$ cx
joke 411 $
3838 212 _ 976 _ dial - a - joke
Dial 411 $
3838 212 - 976 _ dial - a - joke
4200 212 _ 246 _ dial - a - prayer
3636 212 _ 976 _ dial santa
dow jones' 411' $
```

Grep : can't open jones

در اینجا چیزی اشتباه است

نمی تواند `jones` را باز کند

\$

مثال آخر شامل نشان دادن یک مشکل پتانسیل می باشد: اگر چه `jones dow` برای ۴۱۱ به عنوان یک آرگومان مفرد نشان داده می شود، اما شامل یک فضای خالی است و در نقل قولها بزرگتر نمی باشد، بنابراین زیرشل، با تفسیر فرمان ۴۱۱، آن را به دو آرگومان برای `grep` تبدیل می کند: این فرمان به گونه ای است که شما تایپ کردید

```
$ grep dow jones /usr/you/lib/phone - book
```

و چنین چیزی به طور بدیهی اشتباه است.

یک راه علاج، اعتماد به روشی است که شل بر طبق آن، نقل قولهای دوگانه را بررسی می کند. اگر چه هر چیزی که با `'...'` نقل قول می شود، تخطی ناپذیر است، اما شل به درون `"..."` برای `$, $, 's` و `'s` نگاه می کند. بنابراین اگر شما ۴۱۱ را به صورت زیر تجدید نظر کنید

```
Grep "$*/usr/you/lib/phone -book
```

\$ توسط آرگونها جایگزین خواهد شد، اما به عنوان یک آرگومان مفرد از `grep` عبور خواهد کرد، حتی اگر دارای فاصله های خالی باشد.

```
dow jones 411 $
dow jones report 212-976-4141
```

به راستی، شما می توانید `grep` (و بنابراین ۴۱۱) را مستقل از مورد و با انتخاب `y` بسازید:

```
$ grep -y pattern...
```

با `-y`، حروف موردی پایین در طرح نیز با حروف موردی بالایی در ورودی، متناسب خواهند بود. (این انتخاب در هفتمین ویرایش `grep` وجود دارد، اما در سایر سیستم ها وجود ندارد).

چهار نکته در مورد آرگومانهای فرمان وجود دارند که ما تا فصل ۵ آنها را نادیده می‌گیریم، اما یکی از آنها، چیز با ارزشی در اینجا می‌باشد. آرگومان \$0، نام برنامه‌ای است که اجرا می‌شود - در \$0، cx به صورت « cx » می‌باشد. یک استفاده جدید از \$0 در پیاده

سازی برنامه‌های ۲ و ۳ و ۴ ... می‌باشد که ورودی خود را در ستونهای زیادی پرینت می‌کنند: \$ | 2 who

```
drh    ttyo    sep    28    21:23    cvw    tty 5    28    21:09
dmr    4Tty    sep    28    22:10    scj    tty 7    28    22:11
You    ttyg    sep    28    23:00    J1b    b tty    28    19:58
```

\$

پیاده سازیهای 2 و 3 و ... مشابه هستند؛ و در حقیقت آنها مرتبط به همان فایل هستند:

ln \$ 3 2 : ln 24 : ln 25 : ln 26

-li - \$ 1s]9-1 [

```
sep    51    you    5    -rwxrwxrwx    16722    28    23:21    2
sep    51    you    5    -rwxrwxrwx    16722    28    23:21    3
sep    51    you    5    -rwxrwxrwx    16722    28    23:21    4
sep    51    you    5    -rwxrwxrwx    16722    28    23:21    5
sep    51    you    5    -rwxrwxrwx    16722    28    23:21    6
```

ls/usr/you/bin \$ | 5

```
2          3          4          5
          lc          cx          6          m    nu
```

where what

cat 5 \$

در n ستون پرینت کنید : ... 3 و 2 و #

-Pr-\$o-t 11 \$*

\$

انتخاب -t، عنوان را در بالای صفحه خارج می‌کند و انتخاب -ln، طول صفحه را تا n سطر تعیین می‌کند. نام برنامه، تعداد ستونهای آرگومان برای pr می‌شود، در نتیجه خروجی، به صورت یک ردیف در یک زمان در تعداد تعداد ستونهای مشخص شده توسط \$0 پرینت می‌شود.

۳.۵ خروجی برنامه به صورت آرگومانها

اکنون می‌خواهیم از آرگومانهای فرمان در یک فایل شل به تولید آرگومانها برگردیم. مطمئناً گسترش اسم فایل از فراکارتهایی شبیه * عمومی‌ترین روش برای تولید آرگونها می‌باشد (غیر از تهیه کردن آنها به صورت آشکارا)، اما روش خوب دیگر از طریق اجرا کردن برنامه می‌باشد. خروجی هر برنامه، می‌تواند با ضمیمه کردن تقاضا در نقل قولهای گذشته '1000'، در یک سطر فرمان قرار گیرد:

\$ 'echo at the tone the time will be 'date

19830 EDT 02:15 : 00 29 At the tone the time will be thu sep

یک تغییر کوچک شرح می‌دهد که '1000' درون نقل قول‌های دوگانه « ... » تفسیر می‌شود:

\$ echo "At the tone

\$ ' the time will be 'date "

1983 EDT 03:07 : 00 29 The time will be thu sep

\$

به عنوان مثالی دیگر، فرض کنید شما می‌خواهید پست الکترونیکی به یک لیست از افرادی بفرستید که اسامی login آنها در فهرست نامه‌رسانی فایل وجود دارد. یک روش نامناسب برای انجام چنین کاری، ویرایش فهرست نامه‌رسانی در یک فرمان پستی مناسب و نشان دادن آن به شل می‌باشد، اما بسیار راحت‌تر این است که بگوییم.

```
$ mail 'cat mailing list' < letter
```

چنین فرمانی cat را برای تولید فهرستی از اسامی کاربرها اجرا می‌کند و این اسامی آرگومانهایی برای پست می‌شوند. (زمانی که خروجی درنقل قولهای گذشته به صورت آرگومان تفسیر می‌شود، شل سطر جدید را به عنوان جداساهای کلمه و نه پایان نماهای سطر فرمان، بررسی می‌کند؛ و این موضوع به طور کامل در فصل ۵ مورد بحث و بررسی قرار می‌گیرد). نقل قولهای گذشته به اندازه کافی برای استفاده آسان می‌باشند به گونه‌ای که حقیقتاً نیازی به یک انتخاب فهرست نامه‌رسانی مجزا به فرمان پستی نمی‌باشد.

یک روش اندکی متفاوت، تبدیل فهرست نامه‌رسانی از یک فهرست از اسامی به برنامه‌ای می‌باشد که فهرست اسامی را پرینت می‌کند:

```
$ cat mailing list
```

```
echo don whr ejs mb
```

```
$ cx mailing list
```

```
$ mailing list
```

```
don whr ejs mb
```

\$

اکنون به افرادی که در فهرست هستند نامه ارسال کنید

```
$ mail / 'mailing list' < letter
```

با افزودن یک برنامه دیگر، حتی تغییر فهرست کاربر به صورت محاوره‌ای امکان‌پذیر می‌باشد. برنامه pick نامیده می‌شود:

```
$ pick arguments...
```

آرگومانها را در هر زمان یکی یکی نشان می‌دهد و پس از ارائه هر کدام منتظر جواب می‌ماند. خروجی pick، آرگومانهایی می‌باشد که توسط پاسخهای y (برای «بله») انتخاب می‌شوند؛ و هر پاسخ دیگری باعث می‌شود که آرگومان حذف شود. برای مثال،

```
$ pr 'pick*. C' | pr
```

هر اسم فایلی را نشان می‌دهد که با C، پایان می‌پذیرد؛ و اسامی که انتخاب می‌شوند با pr و 1pr پرینت می‌شوند (pick بخشی از هفتمین ویرایش نمی‌باشد، اما آنقدر آسان و مفید است که ما نسخه‌های مربوط به آن را در فصل ۵ و ۶ گنجانده‌ایم.

فرض کنید شما دومین نسخه از فهرست نامه‌رسانی را دارید. سپس

```
$ mail 'pick\'mailing list\' "<letter
```

```
don?y
```

```
?whr
```

```
?ejs
```

```
mb?y
```

\$

نامه را به don و mb می‌فرستد. توجه داشته باشید که نقل قولهای گذشته تو در تو وجود دارند؛ پس کج خطها از تفسیر داخل «...»

در طول تجزیه مورد بیرونی، جلوگیری می کنند.

تمرین ۳.۱۰. اگر پس کج خطها در فرمان \$ echo 1 'echo'\date ..\

تمرین ۳.۱۱. \$ 'date' را بررسی کنید و نتیجه را شرح دهید.

تمرین ۳.۱۲

\$ grep -l pattern filenames

اسامی فایلهایی را لیست می کند که در آنها یک تطبیق از طرح وجود دارد، اما خروجی دیگری را تولید نمی کند. برخی از تغییرات را بر

روی

\$ 'command'grep -l pattern filenames

بررسی کنید.

۳.۶ متغیرهای شل

شل، همانند اکثر زبانهای برنامه نویسی دارای متغیرهایی می باشد که در زبان حرفه ای شل، گاهی اوقات پارامتر نامیده می شوند. رشته هایی مانند \$1، پارامترهای مکانی هستند - متغیرهایی که آرگومانها را برای یک فایل شل نگه می دارند. رقم موقعیت را در سطر فرمان نشان می دهد. ماسایر متغیرهای شل را نیز دیده ایم: PATH، فهرست جهت ها برای جستجوی فرمانها می باشد، Home، فهرست login شما می باشد و به همین ترتیب تا آخر. بر خلاف متغیرهای موجود در یک زبان منظم، متغیرهای آرگومان نمی توانند تغییر کنند؛ اگر چه PATH متغیری است که ارزش آن \$ PATH می باشد، اما متغیر 1 که ارزش آن \$1 باشد وجود ندارد. \$1، چیزی بیشتر از یک نماد گذاری فشرده برای اولین آرگومان نمی باشد.

با کنار گذاشتن پارامترهای مکانی، متغیرهای شل می توانند بوجود آیند، وارد شوند و تغییر کنند. برای مثال،

\$ PATH = : /bin:/usr/bin

یک تخصیص می باشد که مسیر جستجو را تغییر می دهد. نباید فضاهای خالی اطراف علامت مساوی وجود داشته باشد و ارزش تخصیص یافته باید یک کلمه مفرد باشد که به این معنا است که باید نقل قول شود اگر شامل فرا کاراکترهایی از شل می باشد که نباید تفسیر شوند. ارزش یک متغیر با مقدم واقع شدن یک نام توسط علامت دلار بدست می آید:

\$ PATH = \$ PATH:/usr/games

\$ echo \$ PATH

:/usr/you/bin:/bin:/usr/bin:/usr/games

\$ PATH=:/usr/you/bin:/bin/usr/bin

\$

همه متغیرها برای شل خاص نیستند. شما می توانید متغیرهای جدید را با دادن ارزش به آنها ایجاد کنید؛ و به صورت سنتی، متغیرها با معنی خاص در جعبه بالایی تشکیل می شوند و در نتیجه اسامی معمولی در جعبه پائینی قرار دارند. یکی از استفاده های عمومی از متغیرها، به خاطر آوردن رشته های بلند مانند اسامی مسیرها می باشد:

pwd \$

usr/you/bin/

dir = pwd \$

جایی را که ما بودیم به خاطر آورد

cd/usr/mary/bin \$	به جايي ديگر برويد
ln \$dir/cx \$	از متغير در يك اسم فايل استفاده كنيد
...\$	براي مدتي كار كنيد
cd \$dir \$	برگرديد

```
pwd $
usr/you/bin/
$
```

مجموعه فرمان توکار شل، ارزشهای همه متغیرهای تعریف شده شما را نشان می دهد. برای دیدن فقط یک یا دو متغیر، پژواک مناسب تر است.

```
$ set
Home=/usr/you
IFS=
PATH=./usr/you/bin:/bin:/usr/bin
PS1=$
PS2=>
Dir=/usr/uou/bin
$ echo $dir
/usr/you/bin
$
```

ارزش یک متغیر، وابسته به شلی می باشد که آن را بوجود می آورد و به طور خودکار از اولاد شل عبور نمی کند.

x = Hello \$	X را بوجود آورید
sh \$	مثل جدید
\$ echo \$ x	

فقط سطر جدید : X بدون تعریف در زیر شل

ctl-d \$	این شل را ترک کنید
\$	به شل اولیه برگردید
\$ echo \$ x	
Hello	X تعریف می شود
\$	

این فرمان به این معناست که فایل شل نمی تواند ارزش یک متغیر را تغییر دهد، چون فایل شل توسط یک زیر شل اجرا می شود:

"echo "X="GoodBye \$	یک فایل دوسطری ایجاد کنید...
> echo \$ X" > set X برای تعیین و پرینت X
cat setX \$	
"X="GoodBye	
Echo \$X	
echo \$ X \$	

```

Hello
sh setx $
Good Bye
echo $ X $

```

x هر شل اولیه، سلام می باشد

x در زیر شل خدا حافظ می باشد...

Hello

اما هنوز سلام در این شل می باشد ...

\$
اما زمانهایی وجود دارند که استفاده از یک فایل شل برای تغییر متغیرهای شل مفید می باشد. یک مثال بدیهی این است که یک فایل، یک فهرست جدید به PATH شما اضافه می کند. بنابراین شکل یکفرمان « . » (نقطه) می سازد که فرمانها را در یک فایل در شل جاری اجرا می کند، به جای اینکه در زیر شل اجرا کند. چنین چیزی در ابتدا اختراع شد، در نتیجه افراد می توانستند به طور مناسب فایلهای profile خود را بدون ورود مجدد به سیستم، مجدداً اجرا کنند، اما این اختراع استفاده های دیگری نیز دارد:

```

$ cat /usr/bin/games
PATH=$PATH:/usr/games
$ echo $PATH
:/usr/bin:/usr/bin
$ . games
$ echo $PATH
:/usr/bin:/usr/bin:/usr/games

```

فایل برای فرمان « . » با مکانیسم PATH جستجو می شود، در نتیجه می تواند در فهرست bin شما جایگزین شود. زمانی که یک فایل با « . » اجرا می شود، فقط به صورت ظاهری شبیه اجرای یک فایل شل می باشد. فایل در مفهوم معمول کلمه اجرا نمی شود. در عوض، فرمانهای موجود در آن، دقیقاً تفسیر می شوند، گویا اینکه شما آنها را به صورت محاوره ای تایپ کرده اید - ورودی استاندارد شل موقتاً برای خارج شدن از فایل تغییر مسیر می دهد. چون فایل خوانده می شود اما اجرا نمی شود، نباید مجوزهای اجرا داشته باشد. تفاوت دیگر این است که فایل آرگومانهای سطر فرمان را دریافت نمی کند؛ در عوض، \$1 و \$2 و مابقی، خالی هستند. اگر آرگومانها عبور می کردند خوب بود، اما آنها عبور نمی کنند.

روش دیگر برای تعیین ارزش یک متغیر در یک زیر شل، نسبت دادن آن به طور آشکارا به سطر فرمان، قبل از خود فرمان می باشد:

```

echo echo $ x' > echo x $
CX echo x $
echo $ x $

```

همانند قبل

Hello

در زیر شل تعیین نمی شود x

echox \$

x = Hi echo x \$

ارزش x از زیر شل عبور کرد

Hi

\$

(در اصل، نسبت دهی ها در هر جایی از سطر فرمان وارد فرمان شدند و اما این عبور با (1) dd تداخل پیدا کرد).

مکانیسم « . » باید برای تغییر ارزش یک متغیر به طور دائم استفاده شود، در حالیکه نسبت دهی های خطی باید برای تغییرات موقتی استفاده شوند. به عنوان یک مثال، دوباره جستجوی /usr/games/ را برای فرمانها با فهرستی که در PATH شما وجود ندارد، در نظر

بگیرید:

```
$ /S /usr / games : grep fort
```

```
fortune
```

 فرمان بیسکویت شانسی

```
$ fortune
```

```
fortune :
```

 یافت نمی شود

```
$ echo $ PATH
```

```
:/usr / you / bin: / bin: /usr / bin
```

 PATH در /usr/games وجود ندارد .

```
$ PATH = /usr / games / fortune
```

```
شمع را خاموش کنید ؛ کتاب را ببندید ؛ زنگ را به صدا درآورید
```

```
$ echo $ PATH
```

```
:/usr / you / bin: / bin: /usr / bin
```

```
$ cat /usr / you / bin / games
```

```
PATH = $ PATH :/usr / games
```

```
$ . games
```

```
$ fortune
```

Khuth - بهینه سازی پیش از موقع، ریشه همه مضرات است

```
$ echo $ PATH
```

```
:/usr / you / bin: / bin: /usr / bin: /usr / games
```

 در این زمان تغییر می کند PATH

```
$
```

استفاده کردن از هر دو مکانیسم در یک فایل شل مفرد، امکان پذیر است. یک فرمان games اندکی متفاوت می تواند برای اجرای یک game تنها بدون تغییر PATH استفاده شود و یا می تواند PATH را به طور دائم برای قرار دادن /usr/games/ تعیین کند:

```
$ cat /usr / you / bin / games
```

```
* $ PATH = $ PATH : /usr / games
```

 توجه کنید *\$ به

```
$ CX /usr / you / bin / games
```

```
$ echo $ PATH
```

```
:/usr / you / bin:/ bin: /usr / bin
```

 ندارد /usr/games

```
$ games fortune
```

```
I'd give my right arm to be ambidextrous.
```

```
$ echo $ PATH
```

```
:/usr / you / bin:/ bin: /usr / bin
```

 هنوز وجود ندارد

```
$ . games
```

```
$ echo $ PATH
```

```
:/usr / you / bin:/ bin: /usr / bin: /usr / games
```

 اکنون وجود دارد

```
$ for tune
```

```
فردی که تردید داشت، گاهی اوقات ذخیره می شود
```

```
$
```

اولین فراخوانی برای games، فایل شل را در یک زیر شل اجرا کرد، جایی که PATH به طور موقتی برای ایجاد /usr/games/ تغییر کرد. در عوض دومین مثال، فایل را در شل جاری با *\$ رشته خالی تغییر کرد، بنابراین فرمانی بر روی سطر وجود نداشت و PATH

تغییر کرد. استفاده از `games` به این دو روش دشوار است، اما منجر به یک مسیر ساده می شود که برای استفاده، مناسب و طبیعی است. زمانی که شما می خواهید ارزش یک متغیر را در زیر شل ها، قابل دستیابی کنید، از فرمان `export` شل باید استفاده شود. (شما ممکن است فکر کنید که چرا راهی برای صدور ارزش یک متغیر از یک زیرشل به سقف آن وجود ندارد). در اینجا یکی از مثالهای قبلی ما وجود دارد، اکنون با متغیر صادر شده:

```
$ x = Hello
$ export x
$ sh
    شل جدید
$ echo $ x
    x معروف در زیر شل
Hello
$ x = 'Good Bye '
    ارزش آن تغییر دهید
$ echo $ x
GoodBye
$ ctl - d
    از شل خارج شوید
$
    به شل اولیه برگردید
$ echo $ x
Hello
    x still Hello
$
```

`export` دارای معنای دقیقی می باشد، اما حداقل برای اهداف روزمره، یک دستور ورق زدن، کافی می باشد؛ مجموعه متغیرهای موقت را برای راحتی کوتاه مدت صادر نکنید، اما همیشه متغیرهایی را صادر کنید که می خواهید در تمام شل ها و زیر شل های خود ست کنید. (برای مثال شامل شلهایی که با فرمان `sed` آغاز می شوند). بنابراین، متغیرهایی که برای شل، خاص می باشند، مانند `PATH` و `HOME`، باید صادر شوند. تمرین ۳.۱۳ چرا ما همیشه فهرست جاری را در `PATH` قرار می دهیم؟ این فهرست در چه جایی باید قرار گیرد؟

۳.۷ مطالبی بیشتر در خصوص جهت دهی مجدد `1/0`

خطای استاندارد اختراع شد، در نتیجه پیغام های خطا همیشه بر روی رایانه ظاهر می شوند:

```
diff file 1 file 2 > diff. Out $
: diff : file 2
$
```

مطمئناً مطلوب است که پیغام های خطا به این روش کار کنند - بسیار ناخوشایند می باشد اگر آنها در `diff . out` ناپدید شوند و شما با این تصور باشید که فرمان نادرست `diff` به درستی کار کرده است.

هر برنامه دارای سه فایل پیش فرض می باشد که زمانی برقرار می شوند که برنامه آغاز می شود، این سه فایل پیش فرض با اعداد صحیح کوچک شماره می گیرند و توصیف گران می شود، این سه فایل پیش فرض با اعداد صحیح کوچک شماره می گیرند و توصیف گران فایل نامیده می شوند (ما در فصل ۷ به آنها مراجعه خواهیم کرد). ورودی استاندارد «0» و خروجی استاندارد «1» که ماقبلاً با آنها

آشنا شدیم، اغلب از فایلها و لوله‌ها و یا به طرف فایلها و لوله‌ها تغییر جهت می‌دهند. خروجی با شماره ۲، یک خروجی خطای استاندارد می‌باشد و در حالت عادی راه خود را به سمت پایانه شما پیدا می‌کند.

گاهی اوقات، برنامه‌ها، خروجی را بر روی خطای استاندارد تولید می‌کنند حتی زمانی که آنها به درستی کار می‌کنند. یک مثال عمومی، زمان برنامه می‌باشد که یک فرمان را اجرا می‌کند و سپس به خطای استاندارد گزارش می‌دهد که چقدر زمان می‌گیرد.

```
$ time wc ch 1 . 3
22691      4288      931      ch 1 . 3
real      0/1
rse      4/0
sys      4/0
```

```
$ thime wc ch 1 . 3 >wc .out
real      2/0
user      0/4
Sys      0/3
$ time wc ch 3.1 > wc . out 2 > time . out
$ cat time - out
```

```
real      1/0
user      4/0
sys      0/3
$
```

ساخت `filename >2` (هیچ فاصله‌ای بین 2 و > نباید باشد) خروجی خطای استاندارد را به درون فایل هدایت می‌کند؛ این ساختار از نظر نحوی نامناسب است اما کارش را انجام می‌دهد. (زمانهای ایجادشده توسط `time` برای آزمایش کوچکی مانند این نمونه صحیح نمی‌باشند. اما یک توالی از آزمایشات بلندتر، اعداد مفید و به طور معقول قابل اعتماد هستند و شما ممکن است به خوبی بتوانید آنها را برای تحلیل بیشتر ذخیره کنید؛ برای مثال به جدول ۸.۱ مراجعه کنید).

همچنین این امکان وجود دارد که دو مسیل خروجی را در هم ادغام کنیم:

```
$ time wc ch 1 . 3 >wc .out 2 >$ 1
$ cat wc out
22691      4288      931      ch 1 . 3
real      1/0
user      0/4
sys      0/3
$
```

نمادگذاری `$1 > 2` به شل می‌گوید که خطای استاندارد را بر روی همان مسیل خروجی استاندارد قرار دهد. ارزش قابل یادآوری برای آمپرساند وجود ندارد؛ شیوه‌ای است که به سهولت باید آموخته شود. شما نیز می‌توانید از `$2 > 1` برای افزودن خروجی استاندارد به خطای استاندارد استفاده کنید:

```
Echo ... 1 > $2
```

بر روی خطای استاندارد پرینت می‌کند. در فایل‌های شل، این فرمان از ناپدید شدن پیغامها درون یک لوله یا درون فایل به طور تصادفی، جلوگیری می‌کند.

شل مکانیسمی را بوجود می‌آورد که بواسطه آن شما می‌توانید ورودی استاندارد را برای یک فرمان، در طول دستور قرار دهید، به جای اینکه بر روی یک فایل جداگانه بگذارید. در نتیجه فایل شل می‌تواند به طور کامل همه چیز را در خود داشته باشد. برنامه اطلاعات فهرست ما، ۴۱۱، می‌تواند به این صورت نوشته شود:

```
$ cat 411
grep "$*" << End
dial - a - joke 3838 -212 _ 976 _
dial - a - prayer 4200 -212 _ 246 _
dial santa 3636 212 _ 976 _
dow jones report 4141-121 _ 976 _
End
$
```

جارگن شل برای این ساختار، یک سند در اینجا، می‌باشد؛ به این معنا که ورودی در اینجا صحیح است به جای اینکه در یک فایل در جای دیگر باشد. << ، به ساختار، علامت می‌دهند؛ کلمه‌ای که دنبال می‌شود (کلمه End در مثال ما)، برای مجزا ساختن ورودی استفاده می‌شود، که به عنوان هر چیزی برای رخداد این کلمه روی یک سطر توسط خودش، پذیرفته می‌شود. شل برای \$ ، '...' و 1، در یک سند در اینجا، جایگزین می‌کند، مگر اینکه بخشی از کلمه با نقل قولها یا یک پس کج خط ، نقل قول شود؛ در این مورد، کل سند، به صورت حرفی می‌شود.

ما به موضوع اسناد در اینجا، در پایان فصل، با یک مثال جالب تر مراجعه می‌کنیم.

جدول ۳.۲. جهت دهی های مجدد و متعدد ورودی - خروجی را که شل آنها را درک می‌کند، لیست می‌کند.
تمرین ۳.۱۴. نسخه سند در اینجا ۴۱۱ را با نسخه اصلی مقایسه کنید. کدام یک راحتتر قابل نگهداری است؟ کدام یک مبنای بهتری برای یک سرویس کلی می‌باشد؟

جدول ۳.۲: جهت دهی های مجدد I/O در شل

file<	خروجی استاندارد را به سمت فایل هدایت می‌کند
file <<	خروجی استاندارد را از فایل می‌گیرد
file>	خروجی استاندارد را به سمت پیوست می‌کند
2P. P	خروجی استاندارد برنامه 1P را به ورودی 2P متصل می‌کند
^	مترادف را برای خارج می‌کند
n>file	خروجی را از توصیفگر فایل n به فایل هدایت می‌کند
n>>file	خروجی را از توصیفگر فایل n به فایل پیوست می‌کند
n>\$ m	خروجی توصیفگر فایل n را در توصیفگر فایل m ادغام می‌کند

<code>n<\$m</code>	ورودی توصیفگر فایل n را در توصیفگر فایل m ادغام می کند
<code>S>></code>	سند در اینجا: ورودی استاندارد را می گیرد، تا زمان S بعدی در آغاز یک سطر؛ آن را جایگزین \$، '...' و 1 می کند
<code>>> S1</code>	سند در اینجا بدون جایگزینی
<code>>> 'S'</code>	سند در اینجا بدون جایگزینی

۳.۸ حلقه سازی در برنامه های شل

شل، حقیقتاً یک زبان برنامه نویسی می باشد: شل دارای متغیرها، حلقه ها، تصمیم گیری و مواردی از این قبیل می باشد. ما در اینجا به بحث و بررسی در خصوص حلقه سازی اساسی می پردازیم و در خصوص روند کنترل در فصل ۵ صحبت می کنیم. حلقه ای کردن یک مجموعه از اسامی فایلها، بسیار عمومی است و بیان for شل، تنها بیان روند کنترل شل می باشد که عموماً باید در پایانه تایپ شود به جای اینکه در یک فایل یا در اجرای بعدی قرار گیرد. نمو عبارت است از:

```
for var in listof wrds
do
    commands
done
```

برای مثال، یک بیان for برای پژواک اسامی فایل ها در هر سطر به این صورت است.

```
$ for I in *
> do
>     echo $ i
> done
```

«i» می تواند هر متغیری از شل باشد، اگرچه i، قدیمی است. توجه داشته باشید که ارزش متغیر، توسط \$i ارزیابی می شود، اما ارزش حلقه، به متغیره صورت i، نسبت داده می شود. ما از * برای جمع کردن همه فایلها در فهرست جاری استفاد کردیم، اما از هر فهرست دیگری از آرگومانها می توان استفاده کرد. در حالت عادی، شما می خواهید چیزی جالبتر از صرفاً پرینت کردن اسامی فایلها انجام دهید. چیزی که ما غالباً انجام می دهیم، مقایسه کردن یک مجموعه از فایلها با نسخه های قبلی می باشد برای مثال، برای مقایسه نسخه قبلی فصل ۲ (حفظ شده در فهرست قبلی) با نسخه فعلی:

```
$ 1S ch2.*|5
ch1 . 2      ch2 . 2      ch3 . 2      ch4 . 2      ch5 . 2
ch6. 2      ch 7. 2
$ for i in ch2.*
> do
>     echo $ i:
>     diff -b old / $ i $ i
>     echo      یک سطر خالی برای قابلیت خواندن اضافه کنید
```

```
> don | pr - h "diff 'pwd' / old 'pwd' " | lpr $
3712 process -id
$
```

ما خروجی را به `pr` و `lpr` از طریق لوله متصل کردیم برای اینکه شرح دهیم این امکان وجود دارد که: خروجی استاندارد برنامه‌ها در یک `for`، وارد خروجی استاندارد خود `for` شود. ما یک عنوان خیالی را با انتخاب `h` - از `pr`، بر روی خروجی قرار می‌دهیم و از دو فراخوانی توکار `pwd` استفاده می‌کنیم. و ما کل توالی را طوری ست می‌کنیم که به طور ناهمگام (\$) اجرا شود، در نتیجه ما نباید برای اجرای آن صبر کنیم؛ \$ برای تمام حلقه و خط لوله‌ای بکار می‌رود.

ما ترجیح می‌دهیم که یک بیان `for` رافرمت کنیم، همانگونه که نشان داده می‌شود، اما شما می‌توانید از اندازه‌ای آن را متراکم کنید. محدودیت‌های مهم، `do` و `done` می‌باشند که فقط به عنوان لغات کلیدی شناخته می‌شوند، زمانی که دقیقاً پس از یک سطر جدید یا سمی کالن ظاهر می‌شوند. براساس اندازه `for` گاهی اوقات بهتر است که آن را تماماً روی یک سطر بنویسیم:

```
for i in list ; do commands ; done
```

شما باید از حلقه `for` برای فرمانهای متعدد و یا در جایی استفاده کنید که پردازش آرگومان توکار در فرمانهای منفرد، مناسب نمی‌باشد. اما از آن، زمانی که فرمان منفرد، اسامی فایلها را حلقه سازی می‌کند، استفاده نکنید:

```
# poor idea:
for I in $ *
do
    < homd +× $I
done
```

این فرمان پائین تر از فرمان زیر می‌باشد:

```
* $ × + Chomd
```

چون حلقه `for` یک `chomd` مجزرا را برای هر فایل اجرا می‌کند، که در دستگاههای کامپیوتر پرهزینه‌تر می‌باشد. (اما اطمینان داشته باشید که شما تفاوت بین

```
* for i in
```

را که همه اسامی فایلها را در یک فهرست جاری حلقه بندی می‌کند و

```
* $ for i in
```

را که همه آرگومانها را برای فایل شل حلقه بندی می‌کند، درک می‌کنید).

فهرست آرگومان برای یک `for`، اغلب از طرحی می‌آید که منطبق با اسامی فایلها می‌باشد اما می‌تواند از هر جای دیگری نیز بیاید. این فهرست می‌تو

```
$ for i in 'cat .... '
```

باشد و یا آرگومانها نمی‌توانند تایپ شوند. برای مثال، قبلا در این فصل ما یک گروه از برنامه‌ها را برای پرینت چند ستونی، با عناوین ۲. و ۳. از این قبیل بوجود آوردیم. این موارد خطوط پیوند به یک فایل تنها می‌باشند که می‌تواند ساخته شود، زمانی که فایل ۲ به این صورت نوشته می‌شود:

```
Done ; $I 2 in do ; 6 5 4 3 $ for i in
$
```

به عنوان یک استفاده تاحدودی جالب تر از `for` ما می‌توانیم از `pick` استفاده کنیم برای اینکه انتخاب کنیم کدام فایلها را با سایر فایلها موجود در فهرست پشتیبان مقایسه کنیم:

```
$ for i in 'pick ch2.*'
```

```

> do
>     echo $i:
>     diff old / $i $i
> done | pr | lpr
chr. 1 ? y
ch2 . 2  ؟
ch 3 . 2  ؟
ch 4 . 2  ؟ y
ch 5 . 2  ؟ y
ch 6 . 2  ؟
ch7 . 2  ؟
$

```

بدیهی است که این حلقه باید برای تایپ زمان بعدی در یک فایل شل قرار داده شود: اگر شما چیزی را برای دوبار انجام داده‌اید، این احتمال وجود دارد که شما دوباره آن را انجام دهید.

تمرین ۳.۱۵ اگر حلقه `diff` در یک فایل شل قرار داده شود، آیا شما `pick` را در فایل شل قرار می‌دهید؟ چرا بله و چرا نه؟

تمرین ۳.۱۶ چه اتفاقی می‌افتد اگر سطر آخر حلقه بالا به این صورت باشد.

```
> done | pr | 'pr $
```

یعنی اینکه با یک آپرساند پایان پذیرد؟ ببینید آیا می‌توانید از آن سردر بیاورید، سپس آن را بررسی کنید.

۳.۹ بسته : یکی کردن همه آنها

برای اینکه بدانیم چگونه فایل‌های شل گسترش می‌یابند، با یک مثال بزرگتر کار می‌کنیم. تصور کنید که پستی را از یک دوست از دستگاہی دیگر دریافت کرده‌اید، که می‌گوید `somewhere!bob` و یعنی اینکه او می‌خواهد کپی‌هایی از فایل‌های شل در `bin` شما را داشته باشد. ساده‌ترین راه برای فرستادن آنها، برگرداندن پست است، بنابراین شما با تایپ زیر آغاز کنید:

```

$ cd /usr/you/bin
$ for i in 'pick*'
> do
>     echo ======$i هست فایل =====
>     cat $i
> done | mail / somewhere ! bob
$

```

اما از نقطه نظر `bob ! somewhere` به آن نگاه کنید: او قصد دارد که یک پیغام پستی با همه فایل‌هایی که به وضوح مرزنامه‌ی می‌شوند، دریافت کند، اما او باید از یک ویراستار برای تبدیل کردن آنها به فایل‌های جزء استفاده کند. علامت دریافت این است که یک پیغام پستی به طور صحیح ساخته شده بتواند به طور اتوماتیک خود را باز کند، در نتیجه گیرنده نباید کاری انجام دهد. چنین چیزی دال بر این است که یک فایل شل باید حاوی هر دو فایل و دستورالعمل‌هایی برای بازکردن آن باشد.

دومین دریافت این است که اسناد در اینجای شل، یک روش مناسب برای ترکیب یک درخواست فرمان و داده‌های فرمان باشند. مابقی کار، فقط گرفتن نقل قولها به طور صحیح می‌باشد. در اینجا یک برنامه کار با عنوان `bundle` وجود دارد که فایلها را در یک فایل شل خود توضیح بر روی خروجی استاندارد آن، گروه‌بندی می‌کند:

```
$ cat bundle
# bundle : group files into distribution package
echo '# To unbundle , sh this file'
for i
do
    echo "echo $i 1>$2"
    echo "cat >>i <<'End of $ i'"
    cat $i
    echo "End of $i"
done
$
```

نقل قول کردن «End of \$i» اطمینان می‌دهد که همه فراکاراکترهای شل در فایل‌های نادیده گرفته می‌شوند.

در حالت طبیعی، شما باید قبل از وارد کردن آن بر روی `somewhere 1bob`، آن را بررسی کنید:

```
$ bundle CX 1C > junk
$ cat junk
# To unbundle , sh this file
echo CX 1>$2
cat SCX <<'End of CX'
chomd +x $*
End of CX
Echo 1C 1>$2
Cat >1C <<'End of 1C'
# 1C : count number of lines in files
wc 1_ $*
End of 1C
$ mkdir test
$ cd test
$ sh ../junk          Try it out
cx
1c
$ 1S
cx
1c

$ cat CX
chomd +x $*
$ cat 1C
# 1C : count number of lines infiles
wc 1_ $*

$ cd ..                looks good
```

```
$ rm junk test /*; rmdir test          clean up
$ pwd
/usr/you/bin
$ bundle 'pick *' | mail / somewhere !bob  send the files
```

اگر یکی از فایل‌هایی که شما می‌فرستید برحسب اتفاق دارای یک سطر به شکل زیر باشد:

End of filename

مشکل وجود دارد، اما این یک اتفاق با احتمال بسیار پائین می‌باشد. برای ایجاد یک بسته صرفاً امن، ما نیاز به یک یا دو مورد از فصل‌های بعدی داریم، اما چنین چیزی به طور قابل توجهی، قابل استفاده و مناسب می‌باشد، همانگونه که نشان می‌دهد.

bundle، تغییرپذیری محیط یونیکس را شرح می‌دهد نه **bundle** از حلقه‌های شل، جهت دهی مجدد I/O، اسناد موجود در اینجا و فایل‌های شل استفاده می‌کند. **bundle** مستقیماً با پست (mail) ارتباط برقرار می‌کند و شاید به طور جالب‌تر، برنامه‌ای است که یک برنامه را بوجود می‌آورد. **bundle**، یکی از بهترین برنامه‌های شل می‌باشد که ما می‌شناسیم - سطرهای اندکی از رمز که هرچیزی را ساده، مفید و دقیق انجام می‌دهند.

تمرین ۳.۱۷. شما چگونه از **bundle** برای فرستادن همه فایلها در یک فهرست و زیرفهرست‌های آن استفاده می‌کنید؟ توجه: فایل‌های شل می‌توانند بازگشتی باشند.

تمرین ۳.۱۸. **bundle** را به گونه‌ای تغییر دهید که با هر فایل حاوی اطلاعات ذخیره شده از S1_1، بویژه موارد مجاز و زمان آخرین تغییر باشد. تسهیلات **bundle** را با برنامه بایگانی **ar** مقایسه کنید.

۳.۱۰ چرا یک شل قابل برنامه ریزی

شل یونیکس، نمونه مفسران فرمان نمی‌باشد: اگرچه این امکان را به شما می‌دهد که فرمانها را در روش معمول اجرا کنید اما چون یک زبان برنامه‌نویسی است، می‌تواند کارهای بیشتری انجام دهد. ارزش آن تا حدودی به چیزی برمی‌گردد که ما در آن مشاهده کرده‌ایم، تا حدودی به این دلیل که مطالب زیادی در این فصل وجود دارد، اما بیشتر به خاطر اینکه ما قول دادیم که درخصوص «ویژگیهای عموماً استفاده شده» صحبت کنیم و سپس حدود ۳۰ صفحه در خصوص مثالاتی مربوط به برنامه ریزی شل نوشتیم.

اما زمانی که شما از شل استفاده می‌کنید، شما در تمام مدت، مطالب اندکی درخصوص برنامه‌های یک سطری می‌نویسید: یک خط لوله‌ای، یک برنامه است همانند این مثال که «چای حاضر است». شل به این صورت کار می‌کند: شما به طور مداوم به آن برنامه می‌دهید، اما بسیار آسان و طبیعی است (همانگونه که شما با آن آشنا هستید) بطوری که شما نمی‌توانید به عنوان یک زبان برنامه‌نویسی به آن فکر کنید.

شل بعضی از کارها را انجام می‌دهد، مانند حلقه سازی، جهت دهی مجدد I/O با دو < > و < * > گسترش اسم فایل با *، بنابراین هیچ برنامه‌ای نگران این کارها نمی‌باشد و به طور مهمتر، کاربرد این تسهیلات درمیان همه برنامه‌ها یکسان است.

سایر ویژگی‌ها، مانند فایلها و لوله‌های شل، حقیقتاً توسط کرنل تهیه می‌شوند، اما شل برای ایجاد آنها دارای نمو طبیعی می‌باشد. آنها بیشتر از چیزی که مناسب است، توانایی‌های سیستم را افزایش می‌دهند.

قسمت اعظم قدرت و سهولت شل، از کرنل یونیکسی می‌باشد که در زیر آن قرار دارد: برای مثال، اگرچه شل، لوله‌ها را نصب می‌کند

اما کرنل حقیقتاً داده‌ها را به درون آنها وارد می‌کند. روشی که بر طبق آن، سیستم فایل‌های قابل اجرا را بوجود می‌آورد، امکان نوشتن فایل‌های شل را بوجود می‌آورد و در نتیجه آنها دقیقاً شبیه برنامه‌های کامپایل شده اجرا می‌شوند. کاربر نباید مطلع باشد که آنها فایل‌های فرمان هستند - به آنها با یک فرمان خاص مانند RUN استفاده نمی‌شود. همچنین شل خودش یک برنامه است و بخشی از کرنل نمی‌باشد، در نتیجه می‌تواند تنظیم شود، گسترش یابد و شبیه هر برنامه دیگری استفاده شود. این عقیده برای سیستم یونیکس بی‌نظیر نمی‌باشد، اما بهتر از هر جای دیگری استفاده شده است.

در فصل ۵ ما به موضوع برنامه نویسی شل باز می‌گردیم، اما شما باید به خاطر داشته باشید که هرکاری که شما با شل انجام می‌دهید: شما آن را برنامه‌نویسی می‌کنید - به همین دلیل است که به خوبی کار می‌کند.

تاریخچه و نکات کتاب شناسی

شل، از همان زمان‌های اولیه، قابل برنامه نویسی بوده است. در اصل، فرمان‌های مجزایی برای `goto`، `if` و `labels` وجود داشتند و فرمان `goto` که از طریق مرور فایل ورودی عمل می‌کند، از ابتدا به دنبال `lable` صحیح می‌باشد. (چون این امکان وجود ندارد که یک لوله مجدداً خوانده شود، این امکان وجود ندارد که درون یک فایل شلی را لوله‌گذاری کنیم که دارای جریان کنترل بوده است). هفتمین ویرایش شل، در ابتدا توسط استیو برون با کمک و عقاید جون ماشری همچنانکه در فصل ۵ خواهیم دید. به علاوه، ورودی و خروجی توجیه می‌شوند: این امکان وجود دارد که `I/O` را در داخل و خارج از برنامه‌های شل بدون محدودیت، تغییر جهت دهیم. تجزیه فراکاراکترهای اسم فایل نیز برای این شل درونی می‌باشد؛ تجزیه یک برنامه مجزا در نسخه‌های اولیه بوده است که باید بر روی ماشین‌های خیلی کوچک قرار گیرد.

یک شل مهم دیگر، که ممکن است شما در `csh` اجرا کنید (حتی ممکن است شما آن را ترجیحاً استفاده کنید) شل `C` می‌باشد که دربرکلی توسط بیل جوی با ایجاد ششمین ویرایش شل، توسعه یافت. شل `C`، بیشتر از شل بورن در جهت واکنش کمکی متقابل استفاده شده است - بویژه این شل، یک مکانیسم تاریخی را فراهم می‌کند امکان تکرار خلاصه نویسی فرمان‌هایی را می‌دهد که قبلاً صادر شده‌اند (شاید با اندکی ویرایش). نمو، نیز تا حدودی متفاوت می‌باشد. اما چون براساس شل قبلی می‌باشد، ازسهولت برنامه‌نویسی کمتری برخوردار است؛ و نمو، بیشتر یک مفسر فرمان محاوره‌ای می‌باشد تا یک زبا برنامه‌نویسی. بویژه، این امکان وجود ندارد که در داخل یا خارج از ساختهای جریان کنترل را لوله‌گذاری کنیم.

`Pick` توسط تام داف اختراع شد و `bundle` به طور مستقل توسط آلن هیرت و جیمز گاسلینگ اختراع شد.

فصل 4 : فیلترها

یک خانواده بزرگ از برنامه‌های یونیکس وجود دارد که ورودی را می‌خوانند، یک تبدیل ساده را بر روی آن انجام می‌دهند و خروجی را می‌نویسند. مثلهایی از این قبیل شامل `grep` و `tail` می‌باشند که بخشی از ورودی را انتخاب می‌کند، `sort` که آن را ترتیب‌بندی می‌کند، `Wc` که آن را می‌شمرد و مواردی از این قبیل. چنین برنامه‌هایی، فیلتر نامیده می‌شوند.

این فصل، فیلترهایی را که غالباً مورد استفاده قرار می‌گیرند مورد بحث و بررسی قرار می‌دهد. ما با `grep` آغاز می‌کنیم و بر طرحهایی متمرکز می‌شویم که پیچیده‌تر از طرحهای شرح داده شده در فصل ۱ می‌باشند. ما همچنین به دو عضو دیگر از خانواده `grep` یعنی `egrep` و `fgrep` می‌پردازیم.

بخش بعد به طور خلاصه تعدادی دیگر از فیلترهای مفید را توصیف می‌کند که شامل `tr` برای حرف‌نگاری کاراکتر، `dd` برای پرداختن به داده‌هایی از سایر سیستم‌ها و `unig` برای آشکارسازی سطرهای تکرار شده متن می‌باشند. `Sort` نیز به طور مفصل‌تر از فصل ۱ ارائه می‌شود.

مابقی فصل، به دو هدف کلی مبدل‌های داده‌ها یا فیلترهای برنامه‌پذیر، اختصاص دارد. آنها به این دلیل برنامه‌پذیر نامیده می‌شوند که تبدیل ویژه به عنوان یک برنامه در یک زبان ساده برنامه‌نویسی، بیان می‌شود. برنامه‌های متفاوت می‌توانند تبدیلهای بسیار متفاوت را بوجود آورند.

برنامه‌ها عبارت از `sed`، که برای ویراستار جریان می‌باشد و `awk` که پس از نویسندگانش، نامگذاری می‌شود می‌باشند. هر دو برنامه از یک تعمیم از `grep` مشتق می‌شوند :

`$ program pattern – action filenames ...`

فایلها را به طور متوالی پویش می‌کند و به دنبال سطرهایی می‌باشد که یک طرح را تطبیق می‌دهند؛ زمانی که کشف می‌شود عملکرد مربوطه انجام می‌شود. برای `grep`، طرح یک بیان منظم مانند `ed` می‌باشد و عملکرد پیش فرض، پرینت کردن هر سطر می‌باشد که طرح را تطبیق می‌دهد.

`sed` و `awk`، هم طرحها و هم عملکردها را تعمیم می‌کنند. `Sed`، یک مشتق از `ed` می‌باشد و یک برنامه از فرمانهای ویراستار و داده‌های جریانی می‌گیرد که از طریق فایلها از آنها عبور می‌کنند و فرمانهای برنامه را روی هر سطر انجام می‌دهند. `awk` برای جایگزینی متن به سهولت `sed` نمی‌باشد، اما شامل حساب، متغیرها، تابعهای توکار و یک زبان برنامه‌نویسی می‌باشد که اندکی شبیه به زبان `C` است. این فصل دارای یک داستان کامل در مورد هر برنامه نمی‌باشد و جلد ۲۰ از کتاب راهنمای برنامه‌نویس یونیکس دارای برنامه‌های آموزشی درخصوص هر دو برنامه می‌باشد.

۴.۱ `grep`

ما در فصل ۱ به طور خلاصه در مورد `grep` ذکر کردیم و از آن موقع از آن در مثلهایی استفاده کرده‌ایم.

`$ grep pattern filp-nomes...`

به جستجوی فایل‌های نامگذاری شده یا ورودی استاندارد می‌پردازد و هر سطر را پرینت می‌کند که شامل یک مورد از طرح باشد. `grep` برای پیدا کردن رخداد‌های متغیرها در برنامه‌ها و یا کلمات در اسناد و یا برای پیدا کردن قسمتهایی از خروجی یک برنامه، ارزشی ندارد :

متغیر را در منبع C قرار دهید

```
$ grep -n variable *.[ch]
```

```
$ grep from $ mail
```

```
$ grep from $ mail | grep -v mary
```

```
$ grep -y mary $ Home/lib/phone-book
```

```
$ who | grep mary
```

```
$ Is | grep -v temp
```

انتخاب -n، تعداد سطرها را پرینت می‌کند، انتخاب -v، حس امتحان را تبدیل می‌کند و -y حروف جعبه پائینی را در حروف تطبیق طرح همان جعبه در فایل می‌سازد (جعبه بالایی نیز فقط جعبه بالایی را تطبیق می‌دهد).

در همه مثالهایی که تاکنون مشاهده کرده‌ایم، grep به دنبال رشته‌های معمولی حروف و اعداد بوده است. اما grep می‌تواند حقیقتاً در جستجوی طرحهای پیچیده‌تر باشد: grep، بیانها را در یک زبان ساده برای توصیف رشته‌ها تفسیر می‌کند.

از نظر تکنیکی، طرحها اندکی یک شکل محدود شده از مشخص کننده‌های رشته می‌باشند و عبارت منظم نامیده می‌شوند. Grep همان عبارت منظم مانند ed را تفسیر می‌کند؛ در حقیقت grep در ابتدا (در یک بعدازظهر) از طریق عمل مستقیم بر روی ed بوجود آمد.

عبارتهای منظم با دادن معنی ویژه به کاراکترهای خاص، مشخص می‌شوند، درست شبیه * و غیره که توسط شل استفاده می‌شوند. فرا کاراکترهای بیشتری وجود دارند و متاسفانه دارای تفاوتهایی در معانی خود می‌باشند. جدول ۴.۱ همه فراکاراکترهای عبارت منظم را نشان می‌دهد اما به طور خلاصه در اینجا به مرور آنها می‌پردازیم.

فرا کارکترهای ^ و \$ طرح را برای شروع (^) یا پایان (\$) سطر تثبیت می‌کنند. برای مثال،

```
$ grep From $ MAIL
```

فایلهایی را که شامل From می‌باشند در صندوق پستی شما قرار می‌دهد اما

```
$ grep '^From' $ MAIL
```

سطرهایی را پرینت می‌کند که با Form آغاز می‌شوند، که به احتمال قوی سطرهایی با عنوان پیغام می‌باشند. فراکاراکترهای عبارت منظم، با فراکاراکترهای شل هم‌پوشانی می‌کنند، بنابراین همیشه یک عقیده خوب برای ضمیمه کردن طرحهای grep در نقل‌های منفرد وجود دارد.

grep، طبقه‌هایی از کاراکترها را حمایت می‌کند که بسیار شبیه به کارکترهای موجود در شل می‌باشند، بنابراین [a-z]، هر گونه حرف در جعبه پائینی را تطبیق می‌کند. اما تفاوتهایی وجود دارند: اگر یک طبقه از کاراکتر grep با یک سیرکومفلکس ^، آغاز شود طرح هر کاراکتری را تطبیق می‌کند، به جز کاراکترهایی را که در طبقه وجود دارند.

بنابراین، [^0-9] هر گونه کاراکتر غیر رقمی را تطبیق می‌کند. همچنین در شل یک پس کج خط از 1 و -، در یک طبقه از کاراکتر محافظت می‌کند، اما grep و ed مستلزم این هستند که کاراکترها درجایی آشکار شوند که معنی آنها مبهم نباشد. برای مثال، [sil] [-]]، یک گروه مربع بسته یا باز و یا یک علامت منها را تطبیق می‌کند.

یک دوره «.» معادل؟ شل می‌باشد: هر گونه کاراکتری را تطبیق می‌کند. (دوره احتمالاً کاراکتر با متفاوت‌ترین معنی برای برنامه‌های

متفاوت یونیکس می باشد) در اینجا دو مثال وجود دارد :

فهرست اسامی راهنمای فرعی `Is - 1 | grep '^d' $`

فهرست فایل‌هایی که می‌توانند خوانده و نوشته شوند `Is - 1 | grep '^.....rw' $`

« ^ » و هفت د و ره، هر هفت کاراکتر را در آغاز سطر تطبیق می‌دهند، زمانی که از خروجی Is-1 به معنای هر رشته مجاز، استفاده می‌کنند.

عملگر ستبار * برای کاراکتر یا فراکاراکتر قبلی (شامل یک طبقه از کاراکتر) در عبارت بکار می‌رود و آنها جمعاً تعداد تطبیق‌های موفق کاراکتر یا فراکاراکتر را تطبیق می‌کنند. برای مثال، *x یک توالی از x's را تا جایی که امکان دارد تطبیق می‌کند، [a-ZA-Z]* ، یک رشته الفبایی را تطبیق می‌کند، * هر چیزی را تا سطر جدید تطبیق می‌کند و *x ، هر چیزی را تا زمان داشتن آخرین x بر روی سطر تطبیق می‌کند.

دو چیز مهم قابل توجه در مورد ستبارها وجود دارند. روی اینکه، ستبار فقط برای یک کاراکتر بکار می‌رود، بنابراین xy * یک x را که با y's دنبال می‌شود تطبیق می‌کند نه یک توالی مانند xy xy xy . دوم اینکه، هر عددی شامل صفر می‌باشد، بنابراین اگر شما بخواهید که حداقل یک کاراکتر را تطبیق کنید، شما باید آن را دو نسخه‌ای کنید. برای مثال، برای تطبیق یک رشته از حروف، عبارت صحیح [a-ZA-Z]* [a-ZA-Z] می‌باشد [یک حرف یا چند حرفی که با Z دنبال می‌شوند]. اسم فایل شل * که کاراکتر را تطبیق می‌کند، شبیه به عبارت منظم * می‌باشد.

هیچ گونه عبارت منظم grep، یک سطر جدید را تطبیق نمی‌کند؛ عبارت‌ها برای هر سطر به طور جداگانه بکار می‌روند.

با عبارت منظم، grep یک زبان برنامه‌نویسی ساده می‌باشد. برای مثال، به خاطر بیاورید که دومین میدان فایل کلمه رمز، رمزی کردن کلمه رمز می‌باشد. این فرمان بدون کلمات رمز برای کاربرها به جستجو می‌پردازد:

`$ grep '^[:^:] * : : ' / etc / passwd`

این طرح به این صورت است : آغاز سطر، هر عدد بدون دو نقطه، دو نقطه دویل grep، در حقیقت، قدیمی‌ترین خانواده از برنامه‌ها می‌باشد، سایر اعضاء آن Fgrep و egrep نامیده می‌شوند. رفتار اصلی آنها شبیه به هم است، اما fgrep به جستجوی بسیاری از رشته‌های حرفی به طور همزمان می‌پردازد، در حالیکه egrep عبارت‌های منظم درست را تفسیر می‌کند - همانند grep ، اما با یک عملگر or و پرانتزها برای گروه بندی کردن عبارت‌هایی که در زیر شرح داده می‌شوند.

هم fgrep و هم egrep ، هر دو برای مشخص کردن یک فایل که از آن طرح را بخوانند انتخاب f- را می‌پذیرند. در فایل، سطرهای جدید، طرح‌هایی را که باید مورد جستجو قرار بگیرند به طور موازی از هم جدا می‌کنند. اگر کلماتی باشند که شما بر حسب عادت به طور تلفظ می‌کنید، شما می‌توانید اسناد خود را برای چنین رخدادی، با حفظ آنها در یک فایل، در هر سطر و با استفاد از fgrep کنترل کنید :

`$ fgrep -f common - errors document`

عبارت‌های منظم تفسیر شده توسط egrep (که در جدول ۴.۱ نیز فهرست وار وجود دارند) شبیه عبارات موجود grep ، با دو مورد اضافه می‌باشند. پرانتزها می‌توانند برای گروه‌بندی کردن استفاده شوند، بنابراین (x y) * هر رشته خالی را تطبیق می‌کند، xy ، xy xy ، xy ، xy xy و به همین ترتیب. نوار عمودی | یک عملگر "or" می‌باشد؛ today | tommorow - امروز یا فردا را تطبیق می‌کند همانند (to (day | morrow) . در آخر اینکه دو عملگر ستبار دیگر در egrep

وجود دارند. + و ؟. طرح x + يك يا چند x's را تطبیق می‌کند و x ? صفر یا يك x را تطبیق می‌کند و نه بیشتر.

، در بازی کلمات عالی است و شامل جستجوی فرهنگ لغت برای کلمات با ویژگیهای خاص می‌باشد. فرهنگ لغت ما دومین و بستر بین المللی می‌باشد و به عنوان فهرستی از لغات، در هر سطر و بدون تعریف ترتیب بندی می‌شود. سیستم شما ممکن است دارای /usr/dict/words باشد، يك فرهنگ لغت کوچکتر که به کنترل تلفظ می‌پردازد: برای کنترل فرمت به آن نگاه کنید. در اینجا يك طرح برای پیدا کردن لغاتی وجود دارد که حاوی 5 و اول در ترتیب الفبایی می‌باشند :

```
$ cat alphvowels
^[^ aeio] * a [^ a eiou] * e [^ aeiou] * i [^ a eiou] * o [^ a eiou] * a
[^ aeio] * $
$ egrep -f alphrowels /usr/dict/web 2 | 3
abstemious          abstemiously        abstentious
acheilous           acheirous            acleistous
affectious          annelidiou           arsenious
arterious           bacterious           caesoious
facetious           facetiously          fracedinous
majectious
$
```

طرح، در واژه‌های الفبایی فایل، ضمیمه نقل قولها نمی‌شود. زمانی که نقل قولها برای ضمیمه کردن طرحهای egrep استفاده می‌شوند، مثل فرمانها را از تفسیر شدن حفظ می‌کند اما نقل قولها را قطع می‌کند؛ egrep هرگز آنها را نمی‌بیند. چون فایل توسط شل بررسی نمی‌شود، نقل قولها در اطراف محتواهای آن استفاده نمی‌شوند. ما می‌توانستیم از grep برای این مثال استفاده کنیم، اما به خاطر روشی که بر طبق آن egrep کار می‌کند، زمانی که به جستجوی طرحهایی پردازیم که شامل بستر می‌باشند، این کار سریعتر انجام می‌شود، بویژه زمانی که فایل‌های بزرگ را پویش می‌کنیم.

به عنوان مثال دیگر، پیدا کردن همه کلمات شش حرفی یا بیشتر می‌باشد که در ترتیب الفبایی دارای حرف می‌باشند :

```
$ cat monotonic
^a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?$
$ egrep -f monotonic /usr/dict/web 2 | grep '.....' | 5
abdest      acknow      adipsy      agnosy      almost
befist      behint      beknow      bijoux      biopsy
chinte      debors      dehort      deinos      dimpsy
egilops     ghosty
$
```

(egilops، یک نوع بیماری است که به گندم حمله می‌کند). توجه داشته باشید که استفاده از grep برای فیلتر کردن خروجی grep می‌باشد.

چرا سه برنامه grep وجود دارد؟ fgrep، هیچ فرا کاراکتری را تفسیر نمی‌کند اما می‌تواند به طور موثر به دنبال هزاران لغت به صورت موازی باشد (زمانی که آغاز می‌شود. زمان اجرای آن مستقل از تعداد کلمات می‌باشد) و بنابراین در ابتدا برای وظایفی مانند جستجوهای کتاب‌شناسی استفاده می‌شود. اندازه طرحهای اصلی fgrep، مافوق ظرفیت الگوریتم‌های استفاده شده در grep و egrep می‌باشد. تمایز بین grep و egrep به

سختی قابل توجیه است. grep، اندکی زودتر می‌آید، عبارات منظم آشنا را از ed می‌گیرد و بر روی عبارات منظم و یک مجموعه وسیعتری از انتخابها علامت زده است. egrep عبارات کلیدی را تفسیر می‌کند (به جز برای علامت زدن) و به طور حائز اهمیتی سریعتر اجرا می‌کند (با سرعتی مستقل از طرح)، اما نسخه استاندارد، زمان بیشتری را برای شروع می‌گیرد زمانی که عبارت پیچیده است. یک نسخه جدیدتر وجود دارد که سریعاً آغاز می‌شود، بنابراین egrep و grep می‌توانند اکنون به یک طرح واحد تبدیل شوند که برنامه را تطبیق می‌کند.

جدول ۴.۱: عبارات منظم grep و egrep

(به ترتیب تقدم)

c	هر کاراکتر غیر ویژه c، خودش را تطبیق می‌کند
c \	هر معنی ویژه‌ای از کاراکتر c را خاموش می‌کند
^	آغاز سطر
\$	پایان سطر
0	هر کاراکتر مفرد
[...]	هر یک از کاراکترها در ... قرار دارند؛ مراتبی مانند a-z مجاز هستند
[... ^]	هر کاراکتری که در ... قرار ندارد؛ مراتب مجاز هستند.
\ n	چیزی که n'th (... \) تطبیق می‌شود (فقط grep)
* r	صفر یا چند رخداد از r
+ r	یک یا چند رخداد از r (فقط egrep)
? r	صفر یا یک رخداد از r (فقط egrep)
r ₁ r ₂	r ₁ با r ₂ دنبال می‌شود
r ₁ r ₂	r ₁ یا r ₂ (فقط egrep)
(r) \	عبارت منظم و دارای علامت r (فقط grep)؛ می‌تواند تودر تو ساخته شود
(r)	عبارت منظم r (فقط egrep)؛ می‌توند تو در تو ساخته شود

هیچ عبارت منظمی یک سطر جدید را تطبیق نمی‌کند.

تمرین ۴-۱. عبارات منظم دارای علامت (\ (and) |) را در ضمیمه ۱ یا (۱) ed پیدا کنید و از grep برای جستجوی جناسهای قلبی استفاده کنید - کلمات به صورت عقب رو و جلورو تلفظ می‌شوند. توجه: یک طرح متفاوت برای هر طول کلمه بنویسید.

تمرین ۴.۲. ساختار grep برای خواندن یک سطر مفرد می‌باشد، آن را برای تطبیق کنترل کنید، سپس آن را حلقه سازی کنید. grep چگونه تحت تأثیر قرار می‌گیرد اگر عبارات منظم بتوانند سطرهای جدید را تطبیق دهند؟

۴.۲ سایر فیلترها

هدف این بخش، آگاه کردن شما در خصوص وجود و توانایی‌های یک مجموعه غنی از فیلترهای کوچکی می‌باشد که توسط سیستم

تهیه می‌شوند و ارائه مثالهایی درخصوص استفاده از آنها می‌باشد. این فهرست به هیچ وجه کامل نمی‌باشد - تعداد زیاد دیگری نیز هستند که بخشی از هفتمین ویرایش بودند و هر نصب، بخشی از خودش را بوجود می‌آورد. همه فیلترهای استاندارد در بخش اول کتاب راهنما توصیف می‌شوند.

ما با sort آغاز می‌کنیم که احتمالاً مفیدترین آنها می‌باشد. اصول اولیه sort در فصل اول، ارائه شدند: sort، ورودی خود را با یک سطر در ترتیب ASCII ترتیب بندی می‌کند. اگر چه این یک مورد بدیهی است که با پیش فرض انجام می‌شود، اما راههای بسیار دیگری نیز وجود دارند که فرد می‌خواهد از آن طریق داده‌ها را ترتیب بندی کند و sort درصدد فراهم کردن این راهها از طریق ارائه انتخابهای متفاوت برای آنها می‌باشد. برای مثال، انتخاب -f، باعث می‌شود که جعبه بالایی و پائینی تا شوند و در نتیجه تفاوتهای جعبه حذف می‌شوند. انتخاب -d (به ترتیب فرهنگ لغت) همه کاراکترها به جز حروف، ارقام و فاصله در مقایسه‌ها را نادیده می‌گیرد. اگر چه مقایسه‌های الفبایی بسیار عمومی هستند، اما گاهی اوقات یک مقایسه عددی لازم است. انتخاب -n توسط ارزش عددی ترتیب بندی می‌کند و انتخاب -I حس هر گونه مقایسه‌ای را تغییر می‌دهد. بنابراین،

اسامی فایلها را به ترتیب الفبایی ترتیب بندی می‌کند \$ sort -f / s
 با کوچکترین فایلها مرتب می‌کند \$ sort -n / s
 با بزرگترین فایلها مرتب می‌کند \$ sort -nr / s - s

sort، در حالت عادی بر روی یک سطر کامل ترتیب بندی می‌کند، اما می‌توان گفت که توجه خود را فقط به زمینه‌های خاص متمرکز می‌کند. نماد +m به معنای این است که مقایسه از اولین زمینه‌های m جست می‌زند؛ +o آغاز سطر است. بنابراین، برای مثال،

با شمارش بایت، از بزرگترین مرتب می‌شود \$ sort + 3 nr / s - 1
 با زمان login، از قدیمی‌ترین مرتب می‌شود \$ who | sort + 4n

سایر انتخابهای مفید sort شامل o - می‌باشند که یک اسم فایل را برای خروجی مشخص می‌کند (این انتخاب می‌تواند یکی از فایلها را ورودی باشد) و u - که همه، به جز یکی از گروههای سطرها را که در زمینه‌های sort مشابه هستند، حذف می‌کند.

کلیدهای متعدد sort می‌توانند استفاده شوند، همچنانکه با این مثال رمزی از sort صفحه کتاب راهنما شرح داده می‌شود:

\$ sort + of + o - u filenames

+ of، سطر را مرتب می‌کند، جعبه بالایی و پائینی را با هم تا می‌کند، اما سطرهایی که مشابه هستند، نمی‌توانند مجاور هم باشند. بنابراین + o دومین کلید است که سطرهای یکسان از اولین sort را در ترتیب عادی ASCII مرتب می‌کند. در آخر، -u نسخه‌های مجاور را از هم جدا می‌کند. بنابراین با توجه به فهرست کلمات و هر سطر، فرمان، لغات بی نظیری را پرینت می‌کند. شاخص برای این کتاب با یک فرمان sort مشابه تهیه شد، که حتی بیشتر، از توانایی‌های sort استفاده می‌کند. به sortcu مراجعه کنید.

فرمان uniq، الهام برای نشانه -u از sort می‌باشد: این فرمان، همه، به جز یک گروه از سطرهای نسخه‌ای مجاور را از هم جدا می‌کند. داشتن یک برنامه مجزا برای این عملکرد منجر به انجام وظایف بدون ارتباط با ترتیب بندی می‌شود. برای مثال، uniq، سطرهای فاصله متعدد را حذف می‌کند، حال چه ورودی آن مرتب شود و چه مرتب نشود.

انتخابها به روشهای خاصی برای پردازش نسخه‌ها استفاده می‌کند: `uniq -d` فقط سطرهایی را پرینت می‌کند که دو نسخه‌ای می‌شوند؛ `uniq-u` فقط سطرهایی را پرینت می‌کند که بی‌نظیر می‌باشند. (یعنی دو نسخه‌ای نمی‌باشند) و `uniq-c` تعداد رخدادهای هر سطر را می‌شمارد.

ما به یک مثال به طور خلاصه مراجعه خواهیم کرد.

فرمان `comm` یک برنامه مقایسه فایل می‌باشد. با توجه به دو فایل ورودی مرتب شده f_1 و f_2 `comm`، سه ستون از خروجی را پرینت می‌کند:

سطرهایی که فقط در f_1 رخ می‌دهند، سطرهایی که فقط در f_2 رخ می‌دهند و سطرهایی که در هر دو فایل رخ می‌دهند. هر کدام از این ستونها می‌توانند توسط یک انتخاب حذف شوند:

```
$ comm -12 f1 f2
```

فقط سطرهایی را پرینت می‌کند که در هر دو فایل قرار دارند و

```
$ comm -23 f1 f2
```

سطرهایی را پرینت می‌کند که در اولین فایل وجود دارند، اما در دومین فایل نمی‌باشند. چنین چیزی برای مقایسه فهرست‌ها را برای مقایسه یک فهرست کلمه با یک فرهنگ لغت، مفید است.

فرمان `tr`، کاراکترها را در ورودی خودش ترجمه می‌کند. تاکنون عمومی‌ترین استفاده از `tr`، تبدیل مورد می‌باشد:

```
tr a-Z A-Z $
```

از مورد پائین به مورد بالایی می‌نگارد

```
tr AZ a-Z $
```

از مورد بالایی به مورد پائینی می‌نگارد

فرمان `dd` تا حدودی متفاوت از سایر فرمانهایی می‌باشد که ما دیده‌ایم. این فرمان در ابتدا فقط قصد بررسی داده‌های نوار را از سیستم‌های دیگر دارد. نام آن یک باقیمانده از زبان کنترل کار `os/360` می‌باشد. `dd`، تبدیل مورد را انجام می‌دهد (با یک نحو خیلی متفاوت از `tr`)؛ `dd`، از `ASCII` به `EBCDIC` و بالعکس تبدیل می‌کند؛ و داده‌ها را در ثبیتی با اندازه ثابت، با سبب دهی فاصله‌ای که سیستم‌های غیر یونیکس را توصیف می‌کنند، می‌خواند یا می‌نویسد. در عمل، `dd` اغلب برای پرداختن به داده‌های خام و فرمت نشده استفاده می‌شود، با هر منبعی که باشند؛ `dd`، یک مجموعه از امکانات را برای پرداختن به داده‌های دوگانه، آشکار می‌سازد.

برای شرح چیزی که می‌توان با ترکیب فیلترها انجام داد، خط لوله‌ای زیر را در نظر بگیرید، که غالباً 10 کلمه را در ورودی خود پرینت می‌کند:

```
cat $ * |
```

```
tr -sc A-Za-z '\012'|
```

اجراهای غیر حرفی را در سطر جدید متراکم کنید

```
sort |
```

```
uniq -c |
```

```
sort -n |
```

```
tail |
```


cat، فایلها را جمع آوری می کند، چون tr، فقط ورودی استاندارد خود را می خواند، فرمان tr، از کتاب راهنما می باشد: این فرمان موارد غیر حرفی مجاور را در سطرهای جدید متراکم می کند، بنابراین، ورودی را به یک کلمه در هر سطر تبدیل می کند. کلمات در نتیجه مرتب می شوند و uniq-c هر گروه از کلمات مشابه را در یک سطر متراکم می کند که دارای پیشوند تعداد می باشد و برای sort -n، به زمینه ترتیب بندی تبدیل می شود. (این ترکیب از دو ترتیب بندی در اطراف یک uniq که اغلب رخ می دهد، یک شیوه نامیده می شود). نتیجه، کلمات بی نظیر در سند می شود که در تکرار در حال افزایش مرتب می شوند. ۱۰، tail مورد از عمومی ترین کلمات را انتخاب می کند (انتهای فهرست مرتب شده) و ۵ آنها را در ۵ ستون پرینت می کند.

در ضمن، توجه داشته باشید که اتمام یک سطر با |، یک روش معتبر برای ادامه آن می باشد.

تمرین ۴.۳. در این بخش از ابزار برای نوشتن یک کنترل کننده ساده تلفظ، با استفاده از

usr/ dict / words / استفاده کنید. کمبودهای آن چیست و شما چگونه آنها را مورد خطاب قرار می دهید؟

تمرین ۴.۴. یک برنامه شمارش کلمه به زبان برنامه نویسی مورد علاقه خود بنویسید و اندازه، سرعت و قابلیت حفظ آن را با خط لوله ای شمارش کلمه مقایسه کنید. چقدر آسان شما می توانید آن را به یک کنترل کننده تلفظ تبدیل کنید؟

۴.۳ ویراستار جریان sed

اکنون به سراغ sed می رویم. چون sed مستقیماً از ed مشتق می شود، یادگیری آن باید آسان باشد و sed آگاهی شما را در مورد ed، تقویت می کند. ایده اصلی در خصوص sed ساده است:

```
$ sed 'list of ed commands' filenames ...
```

سطرها را از فایل های ورودی می خواند البته در هر زمان یک سطر؛ فرمانهایی را از فهرست برای هر سطر بکار می برد و شکل ویرایش شده آن را بر روی خروجی استاندارد می نویسد. بنابراین، برای مثال، شما می توانید یونیکس را به یونیکس (tm) تبدیل کنید. در هر جایی که در یک مجموعه از فایلها با فرمان زیر رخ می دهد:

```
$ sed 's / unix / unix (Tm) / g' filenames ... > output
```

چیزی را که در اینجا رخ می دهد به غلط تفسیر نکنید. Sed، محتوای فایل های ورودی خود را تغییر نمی دهد. Sed بر روی خروجی استاندارد می نویسد، در نتیجه، فایل های اصلی تغییر نمی کنند. اکنون شما تجربه کافی در خصوص مثل دارید برای اینکه پی ببرید که

```
$ sed '...' file > file
```

یک ایده خوبی نیست: برای جایگزین کردن محتوای فایلها، شما باید از یک فایل موقت یا برنامه ای دیگر استفاده کنید. (ما بعداً در مورد یک برنامه برای آشکار کردن دیده مربوط به روی هم نوشتن یک فایل موجود صحبت خواهیم کرد؛ به روی هم نویسی در فصل ۵ مراجعه کنید).

Sed هر سطر به صورت اتوماتیک خارج می شود، در نتیجه به-p پس از جایگزینی فرمان بالا نیازی نیست؛ حقیقتاً اگر یک-p وجود داشت، هر سطر اصلاح شده، دو بار پرینت می شد. نقل قولها تقریباً همیشه لازم هستند، چون بسیاری از فراکاراکترهای sed برای شل دارای معنی می باشند. برای مثال، استفاده از du - a را برای ایجاد یک فهرست از اسامی فایلها در نظر بگیرید. در حالت عادی، da - اندازه و اسم فایل را پرینت می کند:

```
$ du - a ch 4 . *
```

```

18      ch 4. 1
13      ch 4. 2
14      ch 3. 4
17      ch 4. 4
2       ch 4. 9
$

```

شما می‌توانید از `sed` برای کنار گذاشتن (حذف) بخش اندازه استفاده کنید، اما فرمان ویرایش نیازمند نقل قولها برای محافظت از یک * و یک جدول از تفسیر شدن توسط شل می‌باشد :

```

$ du -a ch 4. * | sed 's / . * → // '
ch 4.1
ch 4.2
ch 4.3
ch 4.4
ch 4.9
$

```

جایگزینی همه کاراکترهای (* 0) را حذف می‌کند و شامل راست‌ترین سمت جدول‌بندی می‌باشد (که در نمونه به صورت ® نشان داده می‌شود).

در یک روش مشابه ، شما می‌توانید اسامی کاربرها و زمانهای `login` را از خروجی `who` انتخاب کنید :

```

$ who
lr      tty1      sep 29   07 : 14
ron     tty3      sep 29   10 : 31
you     tty4      sep 29   08 : 36
td      tty5      sep 29   08 : 47
$ who | sed 's / . * // '
lr      07 : 14
ron     10 : 31
you     08 : 36
td      08 : 47
$

```

فرمان `s` یک جای خالی را جایگزین می‌کند و هر چیزی را که تا فاصله بعدی توسط یک فاصله تنها به دنبال آن می‌آید (تا جایی که امکان‌پذیر است شامل اکثر فاصله‌ها می‌باشد). دوباره نقل قولها لازم هستند.

تقریباً همان فرمان `sed` می‌تواند برای ایجاد یک برنامه `getname` استفاده شود که نام کاربر شما را بر می‌گرداند :

```

$ cat getname
who am : | sed 's / . * // '
$ getname
you
$

```

توالی دیگر `sed` که غالباً استفاده می‌شود این است که ما آن را در یک فایل شل با عنوان `ind` ساخته‌ایم. فرمان `ind`، در ورودی خود یک

ایست جدول بندی را فاصله گذاری می کند؛ این فرمان برای حرکت دادن چیزی به منظور ایجاد تناسب بهتر بر روی صفحه چاپگر سطر مفید و مناسب است.

اجرای `ind` آسان است - یک جدول بندی در جلوی هر سطر بچسبانید :

نسخه 1 از ind * \$ sed 's/^/→/'

این نسخه یک جدول بندی را بر روی هر سطر خالی نیز قرار می دهد که به نظر غیر ضروری می رسد. یک نسخه بهتر از توانایی `sed` برای انتخاب سطرهایی استفاده می کند که باید تغییر داده شوند. اگر شمایک طرح برای فرمان به صورت پیشوند قرار دهید، فقط سطرهایی که طرح را تطبیق می کنند تحت تاثیر قرار می گیرند :

نسخه 2 از ind * \$ sed '/s/^/→/'

طرح `/s/^/`، هر سطر را تطبیق می دهد که حداقل دارای یک کاراکتر بر روی خود غیر از یک سطر جدید می باشد؛ فرمان `s` برای آن سطرها انجام می شود اما برای سطرهای خالی اجرا نمی شود. به خاطر داشته باشید که `sed`، همه سطرها را بدون توجه به اینکه آیا آنها تغییر کرده اند یا نه، خارج می کند، در نتیجه سطرهای خالی به همان صورتی به وجود می آیند که باید باشند.

هنوز یک راه دیگر برای نوشتن `ind` وجود دارد. این امکان وجود دارد که فرمانها فقط بر روی سطرهایی انجام شوند که طرح منتخب را تطبیق نمی کنند، با قرار دادن یک علامت تعجب ! قبل از فرمان. در

نسخه 3 از ind * \$ sed '/^\$/!s/^/→/'

طرح `/^$/`، سطرهای خالی را تطبیق می دهد (انتهای سطر سریعاً با آغاز سطر دنبال می شود)، بنابراین `!/^$/`، فرمان را بر روی سطرهای خالی اجرا نمی کند.

همانگونه که در بالا گفتیم، `sed`، هر سطر را به طور اتوماتیک پرینت می کند، بدون توجه به اینکه چه چیزی درخصوص آن انجام شده است (مگر اینکه حذف شود). به علاوه، اکثر فرمانهای `ed` می توانند استفاده شوند. بنابراین، نوشتن یک برنامه `sed` که سر سطر اول ورودی خود (`say`) را پرینت می کند، آسان است، سپس از سیستم خارج شوید :

`sed 3q`

اگر چه `3q` یک فرمان `ed` مجاز نمی باشد، اما در `sed` حس بوجود می آورد :

سطرهای کپی، در نتیجه پس از سومین فرمان از سیستم خارج می شوند.

شما می خواهید پردازش دیگری را برای داده ها انجام دهید، مانند گذاشتن فاصله (تورفتگی) برای آن. یک راه برای انجام این کار، اجرای خروجی از `sed` از طریق `ind` می باشد، اما چون `sed` فرمانهای متعدد را می پذیرد، چنین چیزی را می توان با یک درخواست تنها (که تا حدودی بعید به نظر می رسد) از `sed` انجام داد :

`sed 's/^/→/'`

`3q'`

توجه به جایی که نقل قولها و سطر جدید قرار دارند، داشته باشید : فرمانها باید بر روی سطرهای مجزا باشند. اما `sed` فاصله های اصلی و جدول بندی ها را نادیده می گیرد.

با این عقاید، نوشتن یک برنامه با عنوان `head`، برای پرینت سطرهای اولیه از هر آرگومان اسم فایل، محسوس به نظر می رسد. اما `sed` `3q` (با `10q`) برای تایپ کردن آنقدر آسان است که ما هرگز نیاز آن را حس نکرده ایم. اما، ما یک `ind` اجرا می کنیم، چون فرمان `sed` معادل آن برای تایپ، دشوارتر است. (در فرآیند نوشتن این کتاب ما برنامه 30 سطر `c` را با دومین نسخه از اجراهای یک سطر که

قبلاً نشان داده شده است، جایگزین کردیم). ملاک آشکاری در این خصوص وجود ندارد که چه موقع ایجاد یک فرمان مجزا از یک سطر فرمان پیچیده با ارزش است؛ بهترین قاعده‌ای که ما پیدا کرده‌ایم، قرار دادن آن در `bin` خود و مشاهده استفاده واقعی از آن می‌باشد.

همچنین این امکان وجود دارد که فرمانهای `sed` را در یک فایل قرار دهیم و آنها را از آنجا با فرمان

```
$ sed -f cmd file ...
```

اجرا کنیم.

شما می‌توانید از گزینش گران سطر غیر از اعدادی مانند ۳ استفاده کنید :

```
$ sed ' / pattern / q '
```

ورودی خود را پرینت می‌کند و دارای طرح تطبیق‌کننده اولین سطر می‌باشد و

```
$ sed ' / pattern / d '
```

هر سطر را که شامل `pattern` باشد، حذف می‌کند؛ حذف قبل از اینکه سطر به طور اتوماتیک پرینت شود رخ می‌دهد، در نتیجه سطرهای حذف شده، جدا می‌شوند.

اگر چه پرینت خودکار معمولاً مناسب می‌باشد، اما گاهی اوقات نیاز به روش دارد. پرینت خودکار می‌تواند با انتخاب `c-n`، خاموش شود؛ در این حالت، فقط سطرهایی که آشکارا با یک فرمان `P` پرینت می‌شوند در خروجی ظاهر می‌شوند.

برای مثال ،

```
$ sed -n ' / pattern / p '
```

کاري را انجام مي‌دهد که `grep` انجام مي‌دهد. چون شرایط تطبیق می‌تواند با دنبال شدن آن با ! معکوس شود، در نتیجه

```
sed -n ' / pattern / ! p ' $
```

 يك `grep -v` می‌باشد (و در نتیجه فرمان `sed ' / pattern / d ' می‌باشد).`

چرا ما هر دوی `sed` و `grep` را داریم؟ بعد از همه این موارد، `grep` فقط یک مورد خاص از `sed` می‌باشد. بخشی از علت آن به خاطر تاریخچه آن می‌باشد - `grep` قبل از `sed` آمد. اما `grep` باقی می‌ماند و به راستی توسعه می‌یابد، چون برای کار ویژه‌ای که هر دوی آنها انجام می‌دهند، اساساً استفاده از `sed` آسانتر است : `grep`، مورد عمومی تا جایی که امکان دارد به طور مختصر انجام می‌دهد. (`grep`)، همچنین کارهای دیگری را که `sed` انجام نمی‌دهد. انجام می‌دهد : برای نمونه به انتخاب `b`- نگاه کنید). اما برنامه‌ها از بین می‌روند. زمانی برنامه‌ای وجود داشت که `grep` نامیده می‌شد و جایگزینی ساده را انجام می‌داد، اما تقریباً خیلی سریع منقضی شد، زمانی که `sed` بوجود آمد.

سطرهای جدید می‌توانند با `sed` و با استفاده از همان نحو `ed` درج شوند:

```
$ sed 's / $ / \
```

```
> /'
```

يك سطر جديد را به انتهاي هر سطر اضافه مي‌کند، در نتیجه فاصله دابل به ورودی آن

```
\ / * [sed 's / [ → ] [→ $
```

اضافه می‌شود و

```
g' / <
```

هر رشته از فاصله‌های خالی یا جدول‌بندی‌ها را با يك سطر جديد جایگزین می‌کند و در

نتیجه ورودی خود را در یک کلمه در هر سطر، تقسیم می‌کند. (عبارت منظم '[→]' ، ' ، یک فاصله یا جدول‌بندی را تطبیق می‌کند ؛ '[→]' * ' تعداد صفر یا چند تا از این فاصله‌ها یا جدول‌بندی‌ها را تطبیق می‌کند، بنابراین کل طرح یک یا چند فاصله و یا چند جدول‌بندی را تطبیق می‌کند.

همچنین شما می‌توانید جفت‌هایی از عبارات منظم یا شماره‌های سطر را برای انتخاب یک دامنه از سطرهایی بکار ببرید که بر روی آنها یکی از فرمانها عمل می‌کنند.

فقط از ۲۰ تا ۳۰ سطر را پرینت کنید
`sed -n '20,30p' $`
 سطرهای ۱ تا ۱۰ را حذف کنید (+11 = tail)
`$ sed '1,10d' tail`
 تا جایی حذف کنید که فقط یک سطر خالی باشد
`sed '1,/^$/d' $`
 هر گروه از سطرها را از یک سطر خالی
`sed -n '/^$/ / ^end/p' $`
 تا سطری که با end آغاز می‌شود پرینت کنید
 آخرین سطر را حذف کنید.
`sed '$d' $`

شماره‌های سطر از آغاز ورودی می‌آیند؛ آنها در آغاز یک سطر جدید رسیت نمی‌شوند. اما یک محدودیت مهم از sed وجود دارد که مشترک با ed نمی‌باشد: شماره‌های نسبی سطر حفظ نمی‌شوند. بویژه، + و ... در عبارتهای شماره سطر درک نمی‌شوند، بنابراین، رسیدن به عقب‌روها در ورودی امکان‌پذیر نمی‌باشد.

نمی‌تواند به عقب رو استناد شود: غیرمجاز
`sed '$ - 1 d' $`

d فرمان شناسایی نشده: \$ - 1

\$

زمانی که یک سطر خوانده می‌شود، سطر قبلی برای همیشه پاک می‌شود: هیچ راهی برای شناسایی سطر بعد از قبلی وجود ندارد، یعنی چیزی که این فرمان نیاز دارد. (به طور عادلانه، یک راه برای کارکردن آن با sed وجود دارد، اما این راه خیلی پیشرفته است. به فرمان «hold» در کتاب راهنما مراجعه کنید). همچنین راهی برای انجام نشانی دهی نسبی به طرف جلو وجود ندارد: نمی‌تواند به جلورو استناد شود: غیر مجاز

\$ sed '/ thing / + d'

sed، توانایی خواندن بر روی فایل‌های خروجی متعدد را فراهم می‌کند. برای مثال،

\$ sed -n '/ pat / w file /
 / pat / ! w file r' filenames ...

سطرهایی را می‌نویسد که pat روی فایل 1 را تطبیق می‌دهند و سطرهایی که pat روی فایل ۲ را تطبیق نمی‌دهند. یا برای مشاهده مجدد اولین مثال ما،

```
$ sed 's / unix / unix (Tm) / gw u.04 t ' filenames ... > output
```

کل خروجی را برای خروجی فایل همانند قبل می نویسد، اما همچنین فقط سطرهای تغییر یافته برای فایل u.out را نیز می نویسد. گاهی اوقات، همکاری با شل برای قراردادن آرگومانهای فایل شل در وسط یک فرمان sed لازم است. یک مثال در این خصوص، برنامه newer می باشد که همه فایلهایی را در یک فهرست لیست می کند که جدیدتر از فایلهای خاص می باشند.

```
$ cat newer
# newer f: list files newer than f
Is -t: sed '/^' '$|'$ /q'
$
```

نقل قولها، از کاراکترهای خاص و متعدد هدف دار در sed محافظت می کنند؛ زمانی که در معرض حذف \$ 1 قرار می گیرند، در نتیجه شل آن را با اسم فایل جایگزین می کند. یک روش دیگر برای نوشتن آرگومان به این شکل می باشد :

```
"/^$|\$q"
```

چون \$ / با آرگومان جایگزین می شود زمانی که \ \$ فقط به \$ تبدیل می شود. به همین روش، ما می توانیم older را بنویسیم که همه فایلهای قدیمی تر از فایلهای نامگذاری شده را لیست می کند :

```
$ cat older
# older f: list files older than f
Is -tr | sed '/^' '$|'$ /q'
```

تنها تفاوت انتخاب -r بر روی IS برای معکوس کردن ترتیب، می باشد.

جدول ۴.۲: خلاصه فرمانهای sed

سطرها را تا جایی به خروجی پیوست می دهد که هیچ سطر با \ به پایان نرسد

a\	انتقال به فرمان : abe//
/ b /abe	سطرها را به متن بعدی به صورت a تغییر می دهد
\c	سطر را حذف می کند؛ سطر ورودی بعدی را می خواند
d	متن بعدی را قبل از خروجی بعدی اینسرت می کند.
i /	سطر را فهرست بندی می کند، همه کاراکترهای غیر پرینتر را مرئی می کند
l	سطر را پرینت می کند
p	از سیستم خارج می شود
q	فایل را می خواند، محتواهای آن را برای خروجی کپی می کند
r file	New را با old جایگزین می کند. اگر f = g، همه
s / old / new / f	رخدادها را جایگزین می کند ؛ f = p . پرینت می کند ؛ f = wfile ،
	فایل را می خواند.
	امتحان : انقال به /abp/ انجام می شود اگر جایگزینی برای سطر خللی انجام شود.
/t /abe	سطر را برای فایل می نویسد
wfile	هر کاراکتر از str1 را با کاراکتر مطابق از str2 جایگزین
/y / str1 / str 2	می کند (مراتب مجاز نیستند)
=	شماره سطر ورودی فعلی را پرینت می کند
cmd !	sed cmd را انجام می دهد فقط اگر سطر انتخاب نشود
/ abe / :	/ abe / برای فرمانهای t و b تعیین می کند

فرمانها را تا تطبیق { به عنوان يك گروه بررسی می‌کند }

اگرچه sed، خیلی بیشتر از چیزهایی که شرح دادیم انجام می‌دهد، شامل بررسی شرایط، حلقه‌سازی و منشعب کردن، به خاطر آوردن سطرهای قبلی و البته بسیاری از فرمانهای ed که در ضمیمه ۱ توصیف می‌شوند - اما قسمت اعظم استفاده sed، شبیه به چیزی است که ما در این جا نشان داده‌ایم - یک یا دو فرمان ساده ویرایش - و چیزی غیر از توالی‌های بلند یا پیچیده می‌باشد. جدول ۴.۲، برخی از توانایی‌های sed را به طور خلاصه بیان می‌کند، اگر چه، عملکردهای چند سطری در این جدول حذف است.

Sed مناسب و آسان است چون به صورت اختیاری می‌تواند با ورودی‌های بلند کار کند، چون سریع است و چون شبیه به ed با عبارتهای منظم و پردازش یک سطر در یک زمان آن می‌باشد. اما روی دیگر سکه، sed - یک شکل نسبتاً محدود از حافظه را فراهم می‌کند (به خاطر آوردن متن از یک سطر تا سطر دیگر دشوار است)، sed فقط یک عبور از داده‌ها را امکان‌پذیر می‌سازد، برگشتن به عقب امکان‌پذیر نمی‌باشد، راهی برای انجام ارجاعات به سمت جلو مانند +1 / ... /، وجود ندارد و sed تسهیلاتی را برای کار کردن با شماره‌ها فراهم نمی‌کند - sed صرفاً یک ویراستار متن می‌باشد.

تمرین ۵. ۴. Older و Newer را به گونه‌ای تغییر دهید که دیگر دارای فایل آرگومان در خروجی خود نباشند. آنها را به گونه‌ای تغییر دهید که فایلها در یک ترتیب مخالف فهرست‌بندی شوند.

تمرین ۴.۶. از sed برای استحکام bundle استفاده کنید. توجه: در اسناد موجود در اینجا، کلمه نشان پایان، فقط زمانی شناسایی می‌شود که سطر را به طور دقیق تطبیق کند.

۴.۴ پوشش طرح awk و زبان پردازش

برخی از محدودیت‌های sed، توسط awk برطرف می‌شوند. طرح مربوط به awk، تا حدود زیادی شبیه به sed می‌باشد، اما جزئیات آن بیشتر بر اساس زبان برنامه‌نویسی c می‌باشند تا يك ویراستار متن. استفاده از awk درست شبیه به sed می‌باشد: \$ awk 'program' filenames ...

اما برنامه متفاوت است:

```
pattern { action }
pattern { action }
...
```

awk، ورودی را در اسامی فایل، در هر زمان یک سطر، می‌خواند. هر فایل با هر طرح به این ترتیب مقایسه می‌شود؛ برای هر طرحی که سطر را تطبیق می‌کند، عملکرد مطابق با آن انجام می‌شود. awk همانند sed، فایل‌های ورودی خود را تغییر نمی‌دهد. طرحها می‌توانند عبارتهای منظم، دقیقاً مانند عبارتهای egrep باشند، یا می‌توانند دارای شرایط پیچیده‌تر و یادآور c باشند. به عنوان یک مثال ساده، اگر چه

```
$ awk '/regular expressino / { print }' filenames ...
```

اما کاری را انجام می‌دهد که egrep انجام می‌دهد: این عبارت هر سطری را پرینت می‌کند که عبارت منظم را تطبیق می‌کند.

طرح یا عملکرد انتخابی می‌باشند. اگر عملکرد حذف شود، عملکرد پیش فرض سطرهای تطبیق شده را پرینت می‌کند، بنابراین

```
$ awk '/regular expression /' filenames ...
```

همان کاری را انجام می دهد که مثال قبلی انجام می دهد. به طور معکوس، اگر طرح حذف شود. در نتیجه بخش عملکرد برای هر سطر ورودی انجام می شود. بنابراین

```
$ awk '{ print }' filenames ...
```

چیزی را انجام می دهد که `cat` انجام می دهد، ولو اینکه کندتر انجام می دهد.

و اما نکته نهایی، قبل از اینکه به مثالهای جالب پردازیم. همانند `sed`، این امکان وجود دارد که برنامه را برای `awk` از یک فایل ارائه دهیم:

```
$ awk -f cmd file filenames ....
```

میدانها

`awk` به طور خودکار هر سطر ورودی را به میدانها تقسیم می کند، یعنی رشته هایی از کاراکترهای بدون فاصله که توسط فاصله ها یا جدول بندی ها از هم جدا می شوند. با این تعریف، خروجی `who` دارای پنج میدان می باشد:

```
$ who
  you      tty 2      sep 29   11 : 35
  jim      tty 4      sep 29   11 : 27
$
```

`awk` میدانهای `$1`، `$2` و `...` `$NF` را می خواند، `NF` متغیری است که ارزش آن برای تعداد میدانها معین می شود. در این مورد، `NF` برای هر دو سطر 5 می باشد. (به تفاوت بین `NF`، عداد میدانها و `$NF`، یعنی آخرین میدان بر روی سطر توجه کنید. در `awk` بر خلاف شل، فقط میدانها با یک `$` آغاز می شوند؛ متغیرها ساده هستند). برای مثال، برای جدا کردن اندازه های فایل ایجاد شده توسط `du-a`

```
$ du -a | awk '{ print $2 }'
```

و برای پرینت اسامی افراد وارد شده به سیستم و زمان `login`، در یک سطر:

```
$ who | awk '{ print $1 , $5 }'
  you      11 : 53
  jim      11 : 27
$
```

برای پرینت نام و زمان `login` ترتیب بندی شده توسط زمان:

```
$ who | awk '{ print $5 , $1 }' | sort
  11 : 27 jim
  11 : 53 you
$
```

راهکارهای دیگری برای نسخه های `sed` وجود دارند که قبلاً در این فصل ارائه شدند. اگر چه `awk` آسانتر از `sed` برای عملکردهایی مانند اینها، قابل استفاده می باشد، اما معمولاً کندتر است و هر دو آغاز به کار و اجرا می کنند زمانی که ورودی زیادی وجود دارد. `awk` در حالت عادی به صورت فضای سفید (هر تعداد از فاصله ها یا جدول بندی ها)، میدانهای مجزا، تصور می شود، اما جداساز می تواند برای هر کاراکتر مفرد تغییر کند. یک روش برای تغییر، انتخاب سطر فرمان `-F` (مورد بالایی) می باشد. برای مثال، میدانهای موجود در

فایل کلمه رمز / passwd / etc ، توسط «دو نقطه‌ها» از هم جدا می‌شوند.

```
$ sed 3 g / etc / passwd
root : 30 . FHR 5 KOB. 3 s : 0 : 1 : S . user : / :
ken : y - 68 wd 10 ijayz : 6 : 1 : k . Thompson : / usr / ken :
dmr ; z 4 a 3 d j w bg v w c k : v : 1 : D-M. Ritchie : / usr / dmr :
$
```

برای پرینت کردن اسامی کاربرها، که از اولین میدان می‌آیند ،

```
$ sed 3 q / etc / passwd | awk - f : ' { print $ 1 } '
root
ken
dmr
$
```

استفاده از فاصله‌ها و جدول‌بندی‌ها، تعمداً خاص می‌باشد. توسط پیش فرض، هم فاصله‌ها و هم جدول‌بندی‌ها، جداساز هستند و جداسازهای اصلی جدا می‌شوند. اگر جداساز، برای هر چیزی غیر از فاصله تنظیم شود، در نتیجه، جداسازی‌های اصلی در تعیین میدانها، به حساب آورده می‌شوند. در اصل، اگر جداساز، یک جدول بندی باشد، در نتیجه فاصله‌ها، کاراکترهای جداساز نمی‌باشند. فاصله‌های اصلی بخشی از میدان می‌باشند و هر جدول بندی، یک میدان را تعریف می‌کند.

چاپ

awk ، رد کمیتهای جالب را در کنار تعداد میدانهای ورودی حفظ می‌کند. متغیر توکار NR، تعداد رکورد ورودی جاری یا سطر می‌باشد. در نتیجه برای افزودن تعداد سطرها به یک جریان ورودی ، از عبارت زیر استفاده کنید :

```
$ awk ' { print NR , $ o } '
```

میدان \$0 ، کل سطر ورودی ، بدون تغییر می‌باشد. در یک بیان print، ارقام که توسط کاماها از هم جدا می‌شوند، به طور مجزا توسط جداساز میدان خروجی چاپ می‌شوند که توسط پیش فرض، یک فاصله خالی می‌باشد.

فرمت کردن که print آن را انجام می‌دهد، اغلب قابل قبول می‌باشد، اما اگر قابل قبول نباشد، شما می‌توانید از یک بیان با عنوان print f برای کنترل کامل خروجی خود استفاده کنید. برای مثال، برای چاپ تعداد سطرها در یک میدان با چهار رقم پهنا، شما می‌توانید از عبارت زیر استفاده کنید:

```
$ awk ' { print f " % 4 d % s \ n " , nR , $ o } '
```

% 4d یک عدد صحیح دهدهی (NR) را در یک میدان با چهار رقم پهنا، مشخص می‌کند، % S یک رشته از کاراکترها (\$0) را و \n یک کاراکتر از سطر جدید را مشخص می‌کنند، چون print f، هیچ فاصله یا سطر جدیدی را به طور خودکار چاپ نمی‌کند. بیان print f در awk شبیه عملکرد c می‌باشد؛ به (print f) مراجعه کنید.

ما توانستیم اولین نسخه ind (از ابتدا در این فصل) را به صورت زیر بنویسیم

```
awk ' { print f " \ t % s \ n " , $ o } ' $ *
```

که یک جدول‌بندی (\t) و رکورد ورودی را چاپ می‌کند.

طرح‌ها

فرض کنید شما می‌خواهید به عبارت `etc / passwd` در جستجوی افرادی باشید که دارای هیچ گونه کلمه رمزی نمی‌باشند. کلمه رمز رمزی شده، دومین میدان می‌باشد، بنابراین برنامه، فقط یک طرح می‌باشد :

```
$ awk -f : '$2 == ""' /etc/passwd
```

طرح درخواست می‌کند که آیا دومین میدان، یک رشته خالی است (`' == ''` ، عملگر تست تساوی می‌باشد) شما می‌توانید این طرح را به روشهای متعدد بنویسید :

```
" " == 2 $
```

```
/ $ ^ / ~ 2 $
```

```
/ . / ~ ! 2 $
```

```
== (0length $2
```

دومین میدان خالی است

دومین میدان، رشته خالی را تطبیق می‌کند

دومین میدان هیچ کاراکتری را تطبیق نمی‌کند

طول دومین میدان صفر است

علامت `~` ، تطبیق عبارت منظم را نشان می‌دهد و به ! به معنای « تطبیق نمی‌کند » می‌باشد. عبارت منظم خودش ضمیمه اسلش‌ها می‌شود.

`length` ، یک عملکرد توکار از `awk` می‌باشد که طول یک رشته از کاراکترها را بوجود می‌آورد. یک طرح می‌تواند به دنبال ! برای منفی کردن آن بیاید، مانند

```
! ($2 == "")
```

! ' یک عملگر شبیه `c` می‌باشد، اما برخلاف `sed` می‌باشد، چون در آنجا ! پس از طرح می‌آید.

یکی استفاده عمومی از طرحها در `awk`، برای وظایف مربوط به معتبرسازی داده‌های ساده می‌باشد. بسیاری از این وظایف، اندکی بیشتر از جستجوی سطرهایی می‌باشند که معیار خود را از دست می‌دهند؛ اگر هیچ گونه خروجی وجود نداشته باشد. داده‌ها قابل قبول هستند (هیچ خبری، خبر خوبی نیست). برای مثال، طرح زیر ، اطمینان می‌دهد که هر رکورد ورودی که دارای تعدادی از میدانها می‌باشد، از عملگر % برای محاسبه باقیمانده، استفاده می‌کند:

```
NF % 2 != 0 # print if old number of fields
```

دیگری، سطرهای بیش از حد طولانی را، با استفاده از عملکرد `length` توکار چاپ می‌کند :

```
/ length ($) > v2 # print if too long
```

`awk`، از همان تبدیل کامنت، همانند شل استفاده می‌کند : یک `#` . ابتدای کامنت را علامت‌گذاری می‌کند.

شما می‌توانید خروجی را تا حدودی آگاهی دهنده‌تر، از طریق چاپ یک اخطار و بخشی از سطر بی‌نهایت و طولانی و با استفاده از یک عملکرد توکار دیگر ، `substr` ، بسازید :

```
length ($) > v2 {print "line " , NR , "too long : " , substr
```

```
($0 , 1 , 4)
```

`substr (s , m , n)` ، رشته فرعی `s` را می‌سازد که در موقعیت `m` آغاز می‌شود و `n` کاراکتر طولانی می‌باشد. (رشته در موقعیت 1 آغاز می‌شود). اگر `n` حذف می‌شود، رشته فرعی از `m` تا پایان، استفاده می‌شود. `Substr`، نیز می‌تواند برای استخراج میدانهای دارای موقعیت ثابت استفاده شود ، برای مثال، انتخاب ساعت و دقیقه از خروجی `date` .

```
$ date
```

```
Thu sep 29 12 : 17 : 01 EDT 1983
```

```
$ date | awk ' { print substr ( $ 4 , 1 , 5 ) } '
12 : 17
$
```

تمرین ۴.۷. چند برنامه awk می‌توانید بنویسید که ورودی را برای خروجی همانند کاری که cat انجام می‌دهد، کپی کنند؟ کدام یک کوتاه‌ترین می‌باشد؟

طرح‌های END , BEGIN

awk، دو طرح خاص را بوجود می‌آورد، BEGIN و END. عملکردهای BEGIN قبل اینکه سطر ورودی خوانده شود، اجرا می‌شوند؛ شما می‌توانید از طرح BEGIN برای آغاز متغیرها، برای چاپ عنوان‌ها و یا برای تعیین جداساز میدان با نسبت دادن آن به متغیر FS استفاده کنید:

```
$ awk ' BEGIN { FS = " : " }
>          $2 == " " ' / etc/passwd
```

ما از همه رمزهای عبور استفاده می‌کنیم: خروجی ندارد

عملکردهای

END، پس از پردازش آخرین سطر ورودی، انجام می‌شوند:

```
$ awk ' END { print NR } ' ...
```

تعداد سطرهای ورودی را پرینت می‌کند.

حساب و متغیرها

مثالهایی که تاکنون بیان شده‌اند، فقط به استفاده در متن ساده پرداخته‌اند. قدرت واقعی awk، مربوط به توانایی آن برای انجام محاسبات بر روی داده‌های ورودی می‌باشد؛ شمردن چیزها، محاسبه مجموعه‌ها و میانگین‌ها و مواردی از این قبیل آسان است. یک استفاده عمومی از awk، جمع کردن ستون اعداد می‌باشد. برای مثال، برای جمع کردن همه اعداد در اولین ستون:

```
END          { s = s + $ 1 }
              { prints }
```

چون تعداد ارزش‌ها در متغیر NR در دسترس می‌باشد. در نتیجه آخرین سطر را به شکل زیر تغییر می‌دهیم

```
END          { print s , s / NR }
```

که هم مجموع و هم میانگین را چاپ می‌کند.

این مثال، استفاده از متغیرها در awk را نیز شرح می‌دهد. S، یک متغیر توکار نمی‌باشد. اما از طریق استفاده شدن، تعریف می‌شود. متغیرها بر طبق پیش فرض با صفر شروع می‌شوند، در نتیجه شما معمولاً نباید نگران آغاز باشید.

awk نیز همان عملگرهای مختصرنویسی حساب را شبیه به C فراهم می‌کند، بنابراین مثال آن در حالت عادی به این صورت نوشته

می شود

```
{ s += $1 }
END { prints }
```

همانند $S + S1 = S$ می باشد، اما از نظر نمادی، فشرده تر می باشد. شما می توانید مثالی را که به شمارش سطرهای ورودی می پردازد مانند زیر تعمیم دهید:

```
{ nc += length ($o) + 1      # number of chars 1 form \ n
  nw += NF                    # number of words
}
END { print NR , nw , nc }
```

این عبارت سطرها، کلمات و کاراکترها را در ورودی خود می شمارد، در نتیجه کار `wc` را انجام می دهد (اگر چه کل ها را توسط فایل تجزیه نمی کند).

به عنوان مثال دیگری از حساب، این برنامه، تعداد صفحات 66 سطر را که با اجرای یک مجموعه از فایلها در `pr` بوجود می آید، محاسبه می کند. چنین چیزی می تواند در یک فرمان با عنوان `prpages`، قرار گیرد:

```
$ cat prpages
# prpages : compute number of pages that pr will
print wc $ * |
awk ! / total $ / { n += int (( $1 + 55) / 56) }
END { print n }
$
```

۵۶، `pr` سطر از متن را در روی هر صفحه قرار می دهد (حقیقتی که از نظر تجربی مشخص شد). تعداد صفحات گرد می شود، سپس با یک عملکرد توکار `int`، به یک عدد صحیح، برای هر سطر از خروجی `wc` که `total` را در انتهای یک سطر تطبیق نمی دهد. سر راست می شود.

```
$ wc ch 4. *
  753   3090   18129   ch 1 . 4
   612   2124   13242   ch 2 . 4
   637   2462   13455   ch 3 . 4
   802   2986   16904   ch 4 . 4
    50    213    117     ch 9 . 4
  2854  11172  62847     کل
$ prpages ch 4. *
53
$
```

برای این نتیجه، `pr` را در `awk` مستقیماً اجرا کنید:

```
$ pr ch 4.* | awk 'End { print NR / 66 }'
53
$
```

متغیرها در `awk` نیز رشته های کاراکترها را ذخیره می کنند. اینکه آیا یک متغیر به عنوان یک

عدد رفتار کند و یا به عنوان یک رشته از کاراکترها، به متن بستگی دارد برای صحبت کردن به صورت تضمینی، در یک عبارت حسابی مانند $S += \$1$ ، ارزش عددی، استفاده می‌شود؛ در یک متن رشته مانند $x = "abc"$ ، ارزش رشته، استفاده می‌شود؛ و در یک مورد مبهم مانند $x > y$ ، ارزش رشته، استفاده می‌شود مگر اینکه عملوندها به وضوح عددی باشند. (قواعد به طور دقیق در کتاب راهنمای awk بیان می‌شوند). متغیرهای رشته، برای رشته خالی، آغاز می‌شوند. آمدن بخش‌ها، منجر به استفاده خوبی از رشته‌ها می‌شود.

Awk خودش شامل تعدادی از متغیرهای توکار از هر دو نوع می‌باشد، مانند NR و FS.

جدول ۴.۳، یک فهرست کامل را ارائه می‌دهد. جدول ۴.۴، عملگرها را فهرست می‌کند.

تمرین ۴.۸. بررسی ما از prpages، اجراهای متناوب را بیان می‌کند. بررسی کنید که کدام یک سریعترین می‌باشد.

روند کنترل

به طور قابل توجهی (با صحبت از تجربه)، ایجاد کلمات همجوار دو نسخه‌ای به طور تصادفی آسان است زمانی که یک سند بزرگ ویرایش می‌شود و بدیهی است که چنین چیزی هرگز به طور عمودی اتفاق نمی‌افتد. برای پیشگیری از چنین مشکلاتی، یکی از اجزاء سازنده خانواده و رک بنچ (workbench) از برنامه‌های نویسنده با عنوان double، به دنبال جفت‌هایی از کلمات مشابه همجوار می‌باشد. در اینجا یک اجرا از double در awk وجود دارد:

```
$ cat double
awk '
FILENAME != prevfile { # New file
    NR=1                # vesttine number
    Prefile = FILENAME
}
NF> 0 {
    If ($) == last word)
        Printf " double %s , file %s , line %d \n", $1, FILENAMENR
    For (i = 2 , i <= NF, i + t)
        If ( $ I == $ ( i -1))
            Printf " double %s , file % s , line %d \n " $I FLENAME, NR
    If ( N F > 0 )
        Lastword = $ NF
} '$ *
$
```

عملگر ++، عملوند خود را افزایش می‌دهد و عملگر --، عملوند خود را کاهش می‌دهد.

متغیر توکار FILENAME، شامل نام فایل ورودی فصلی می‌باشد. چون NR، سطرها را از آغاز ورودی می‌شمارد، در نتیجه ما آن را در هر زمانی که اسم فایل تغییر می‌کند، دوباره تنظیم می‌کنیم، بنابراین، به این ترتیب یک سطر مزاحم به درستی شناسایی می‌شود.

بیان if، دقیقاً شبیه به بیان آن در C می‌باشد:

if (condition)

```
statement 1
else
statement 2
```

اگر `condilion` دست باشد، در نتیجه `statement 1` اجرا می شود؛ و اگر اشتباه باشد، و اگر بخش `else` وجود داشته باشد، در نتیجه `statement 2` اجرا می شود، بخش `else` اختیاری است.

بیان `for`، یک حلقه شبیه به حلقه موجود در `C` می باشد، اما متفاوت از حلقه `شل` می باشد:

```
fo ( expression1 , condition , expression 2)
    statement
```

`for`، شبیه بیان `while` زیر می باشد، که در `awk` نیز معتبر است:

```
expression 1
while ( condition) {
    statement
    expression 2
}
```

برای مثال ،

```
(+ + for ( I = 2 , i <= NF, i
```

حلقه را با مجموعه `i` به ترتیب ۲ و ۳... تا تعداد میدانها، `NF` اجرا می کند.

بیان `break`، منجر به یک خروج سریع از `for` یا `while` ضمیمه می شود؛ و بیان `continue`، منجر به تکرار بعدی برای شروع می شود همانند `condition` موجود در `while` و `expression 2` موجود در `for` (بیان `Next`، منجر به خوانده شدن سطر ورودی بعدی و تطبیق طرح به منظور ایجاد خلاصه در آغاز برنامه `awk` می شود. بیان `exit` منجر به یک انتقال سریع به طرح `END` می شود.

جدول ۴.۳ : متغیرهای توکار `awk`

FILENAME	نام فایل ورودی فعلی
FS	کاراکتر جداساز میدان (پیش فرض فاصله و جدول بندی)
NF	تعداد میدانها در رکورد ورودی
NR	تعداد رکورد ورودی
OFMT	فرمت خروجی برای اعداد (پیش فرض <code>g %</code> و به <code>printf (۳)</code> مراجعه کنید
OFS	رشته جداساز میدان خروجی (پیش فرض فاصله)
ORS	رشته جداساز ثبت خروجی (پیش فرض سطر جدید)
RS	کاراکتر جداساز ثبت ورودی (پیش فرض سطر جدید)

جدول ۴.۴ : علمگردهای `awk` (به ترتیب افزایش تقدم)

```
= + = - = * = / = % = ( (expr
```

تخصیص؛ `expr = Vop` , `V = Vop` می باشد)

```
| |
```

هر کدام باشد صحیح است `expr1 || expr 2` or :

	expr 2 ارزیابی نمی شود اگر expr صحیح باشد
\$ \$	expr 1 \$ \$ AND : اگر هر دو باشند، صحیح است
	expr 2 ارزیابی نمی شود اگر expr 1 اشتباه باشد
!	ارزش عبارت را منفی می کند
< < = > > = = ≈ ! ≈	عملگرهای رابطه ای ؛ « و ، ! »
	تطبیق هستند و پیوند دهی رشته را تطبیق نمی کنند.
+ -	به اضافه ، منها
* \ %	ضرب ، تقسیم ، باقیمانده
+ + - -	افزایش ، کاهش (پیشوند یا پسوند)

آرایه ها

awk، همانند اکثر زبانهای برنامه نویسی، آرایه ها را فراهم می کند. به عنوان یک مثال جزئی، این برنامه awk، هر سطر از ورودی را در یک عنصر آرایه مجزا جمع آوری می کند و توسط شماره سطر شاخص بندی می کند، سپس آنها را در یک ترکیب معکوس پرینت می کند:

```
$ cat back words
# backwords : print in put in backward line order
awk ' { line [ NR ] = $0 }
END { for ( i = NR ; i > 0 ; i -- ) print line [ i ] ' $ *
```

توجه داشته باشید که همانند متغیرها، آرایه ها نباید اعلان شوند؛ اندازه یک آرایه فقط به حافظه موجود در ماشین شما محدود می شود. البته، اگر یک فایل خیلی بزرگ در آرایه خوانده شود. در نهایت ممکن است از حافظه خارج شود. برای پرینت کردن انتهای یک فایل بزرگ در یک ترتیب معکوس، همکاری با **tail** لازم است :

```
$ tail - 5 / usr / dict / web 2 | backwards
zymurgy
zumotically
zymotic
zymothenic
zymosis
$
```

tail، از مزیت یک عملکرد از سیستم فایل با عنوان **seeking**، برای پیشرفت به سمت پایان یک فایل بدون خواندن داده های مزاحم استفاده می کند. به بحث و بررسی **seek 1** در فصل ۷ مراجعه کنید. (نسخه محلی ما از **tail** دارای یک انتخاب **-r** می باشد که سطرها را در یک ترتیب معکوس پرینت می کند، به طوریکه عقب رها را جایگزین می کند).

پردازش ورودی عادی، هر سطر ورودی را در میدانها تقسیم می کند. این امکان وجود دارد که همین عملکرد تقسیم میدان را روی هر رشته ای با عملکرد توکار Split انجام دهیم:

```
n=split(s,arr,sep)
```

رشته n را درون میرانهایی تقسیم می کند که در عناصر 1 تا n از آرایه arr ترتیب بندی می شوند. اگر یک کاراکتر جدا ساز sep فراهم شود، استفاده می شود؛ در غیر این صورت ارزش جاری Fs استفاده می شود. برای مثال، (" : " و split (, a 0\$ سطر ورودی را بر روی «دو نقطه ها» تقسیم می کند که برای پردازش etc/passwd / مناسب می باشد و (" / " و date (" 9/29/ 83 " split ، یک date را بر روی اسلش ها، تقسیم می کند.

```
$ sed 1q/etc / passwd: awk ' [split($0,a, ":",print a [1] ]' root
```

```
$ echo 9/29/83 : awk ' [split($0,date," / " ) , print date [3]]'
```

```
83
```

```
$
```

جدول ۴.۵، عملکردهای توکار awk را فهرست بندی می کند.

جدول ۴.۵: عملکردهای توکار awk

(cos(expr

کسینوس expr

(exp (pexpr

تابع نمایی expr: e^{expr}

() get line

سطر ورودی بعدی را می خواند : صفر را برمی گرداند اگر پایان فایل باشد

اگر پایان فایل نباشد، 1 را بر می گرداند

موقعیت رشته S2 در S1: اگر آشکار نشود صفر را بر می گرداند index(s1,s2) بخش صحیح expr ؛ به طرف صفر سر راست

(int(expr

می کند

(length (s

طول رشته S

(log (expr

لگاریتم طبیعی expr

(sin (expr

سینوس expr

تقسیم S بر [1] ... [n a روی کاراکتر c ؛ split n) s,a,c را بر می گرداند .

(..., sprint f(fmt

فرمت ... بر طبق ویژگی fmt

آرایه های انجمنی

یک مشکل استاندارد در پردازش داده ها، جمع آوری ارزشها برای یک مجموعه از جفت های نام / ارزش می باشد. چنین چیزی از ورودی به صورت زیر است.

```
susie      400
John      100
Mary      200
Mary      300
John      100
Susie     100
Mary      100
```


ما می خواهیم ارزش کل را برای هر نام محاسبه کنیم:

John	200
Mary	600
Susie	500

awk، یک روش موثر را برای انجام این کار فراهم می کند، آرایه انجمنی. اگر چه فرد در حالت عادی زیرنویسهای آرایه را به عنوان اعداد صحیح تصور می کند، اما در awk هر ارزش می تواند به عنوان یک زیرنویس استفاده می شود. بنابراین

```
[sum[$1]+=$2]
```

```
{[END {for (name in sum) print name , sum [name
```

یک برنامه کامل برای جمع کردن و چاپ کردن مجموع جفت های نام - ارزش همانند موارد مذکور در بالا می باشد، حال چه آنها ترتیب بندی شوند و چه نشوند.

هر نام (\$1) به عنوان یک زیرنویس در مجموع استفاده می شود؛ در پایان، یک شکل خاص از بیان for، برای تکرار شدن در تمام عناصر مجموع به کار می رود و آنها را پرینت می کند. از نظر نحوی، این متغییر از بیان for، به این صورت می باشد

```
For (var in array)
statement
```

اگرچه، همانند حلقه for در شل به نظر می رسد، اما نامربوط می باشد. بیان for بر روی زیرنویس های آرایه و نه بر روی عناصر، حلقه سازی می کند و Var را برای هر زیرنویس به ترتیب معین می کند. زیرنویسها، در یک ترتیب غیرقابل پیش بینی بوجود می آیند، بنابراین ترتیب بندی آنها لازم است.

در مثال بالا، خروجی می تواند درون Sort برای فهرست کردن افراد با بزرگترین ارزشها در بالا، لوله گذاری شود.

```
$ awk '...'| sort + lny
```

تحقق حافظه انجمنی، از یک برنامه هش زنی استفاده می کند برای اینکه اطمینان دهد که به هر عنصری با میزان زمان یکسان با عنصری دیگر، دستیابی می کند و اینکه (حداقل برای اندازه های متوسط آرایه) زمان به تعداد عناصر موجود در آرایه، بستگی ندارد. حافظه انجمنی برای وظایفی مانند شمارش همه کلمات در ورودی مؤثر است:

```
$ cat word freq
```

```
awk' {for (I=1,i <=NF, I++)num[$I]++}
END {for (word in num)print word , num[word]}
' $*
```

```
$ word freq ch40* | sort +1 -nr | sed 20q | 4
the 372 .cw 345 of 22 is 185
to 175 a 167 in 109 and 100
.p1 94 .p2 94 pp9 $87
awk87 sed 83 tha 76 for 75
The 63 are 61 line 55 print 52
$
```

اولین حلقه for، به هر کفه در سطر ورودی نگاه می کند و عنصر آرایه زیرنویس شده num را توسط کلمه، افزایش می دهد (i\$) از awk میدان i از سطر ورودی را با هر کدام از متغیرهای شل اشتباه نگیرید). پس از اینکه فایل خوانده شد، دومین حلقه for، در یک ترتیب اختیاری، کلمات و شماره های آنها را پرینت می کند.

تمرین ۴.۹، خروجی از `word freg` شامل فرمانهای فرمت کننده متن مانند `cw` می باشد که برای پرینت کلمات در این فونت استفاده می شود. چگونه شما از مواردی که کلمه نمی باشند خلاص می شوید؟ چگونه شما از `tr` برای انجام کار `wordfreg` به طور صحیح بدون توجه به مورد ورودی آن، استفاده می کنید؟ تحقق و عملکرد `word freg` را با خط لوله ای از بخش ۴.۲ و با این مورد مقایسه کنید:

```
Sed' s / [→] [→]*
/g' $*:fort | uniq -c | sort -nr
```

رشته ها

اگر چه `sed` و `awk` هر دو برای کارهای جزئی مانند انتخاب یک میدان تنها استفاده می شوند، اما `awk` فقط برای وظایفی که حقیقتاً نیاز به برنامه نویسی دارند، استفاده می شود و تا هر میزانی قابل استفاده است. یک مثال در این خصوص، برنامه ای است که سطرهای طولانی را تا ۸۰ ستون تا می کند. هر سطر که بیش از ۸۰ کاراکتر داشته باشد پس از کاراکتر ۸۰، شکسته می شود؛ یک \ به عنوان یک اخطار ضمیمه می شود و مابقی آن ادامه می یابد. بخش نهایی یک سطر تا شده، هم ستون شده از راست می باشد نه هم از ستون شده از چپ، چون هم ستون شده از راست خروجی مناسب تری را برای صورتهای برنامه تهیه می کند، که این چیزی است که ما اغلب از `fold for` استفاده می کنیم. به عنوان یک مثال، از سطرهای ۲۰ کاراکتری بجای ۸۰ کاراکتری استفاده می کنیم و

```
$ cat test
A short lihe
A somewhat longer line
This line is quite a bit longer than the last one.
$ fold test
A short line
A somewhat longer li\
ne.
This line is quite a\
Bit longer than the\
Last one.
```

\$
عجیب است که هفتمین ویرایش، برنامه ای را برای افزودن یا کم کردن جدول بندی ها، فراهم نمی کند، اگر `pr` در سیستم `v`، هر دو را انجام می دهد. اجرای `fold`، از `sed` برای تبدیل جدول بندی ها به فاصله ها استفاده می کند، در نتیجه شمار کاراکتر `awk`، صحیح می باشد، چنین چیزی به طور صحیح برای جدول بندی های اصلی کار می کند (دوباره، نمونه مبدأ برنامه) اما ستونها را برای جدول بندی ها در وسط سطر، حفظ نمی کند.

```
# fold: fold long lines
sed ' /→/ /g' $* | # converttabsto 8 spaces]
awk'
BEGIN{
N=80 #folds at column 80
For (I=1, <=N,I+t) # make astring of blanks
```

```
Blanks = blanks " "
}
{ if (n = length ($0) <=n)
print
else{
for ( I=1, n>N,n-=N)}
printf " % s \\\n" , substr($0, i,N)
i t=N,
}
printf " s%s \n", substr (blanks , 1 ,N -n), substr($0,i)
}
}'
```

در `awk` عملگر پیوند دهی رشته به صورت آشکار وجود ندارد؛ رشته‌ها به هم محلق می‌شوند، زمانی که آنها هم جوار هستند. در ابتدا، `blanks` یک رشته تهی می‌باشد. حلقه در بخش `BEGIN`، یک رشته بلند از `blank` ها را از طریق پیوند دهی بوجود می‌آورد: هر حرکت در اطراف حلقه، یک `blank` دیگر به انتهای `blank` ها اضافه می‌کند. دومین حلقه سطر ورودی را در `chunk` ها پردازش می‌کند تا جایی که بخش باقیمانده به میزان کافی کوتاه می‌باشد. همانند `C`، بیان تخصیص می‌تواند به عنوان یک عبارت استفاده شود، بنابراین ساخت

`If ((n = length($0)) <=N)...`

طول سطر ورودی به `n` اختصاص می‌دهد قبل از اینکه ارزش را آزمایش کند، به پرانتزها توجه داشته باشید.

تمرین ۴.۱۰، `fold` را به گونه‌ای تغییر دهید که سطرها را در فاصله‌ها یا جدول‌بندی‌ها تا کند به جای اینکه کلمه را تقسیم کند. آن را برای کلمات طولانی‌تر، به طور مطلوب بسازید.

برهم کنش با شل

فرض کنید شما می‌خواهید یک برنامه `fieldn` را به گونه‌ای بنویسید که میدان `n` ام از هر سطر ورودی را پرینت کند، بنابراین شما برای مثال می‌گوئید،

```
who | field $
```

برای اینکه فقط اسامی `login` را پرینت کنید. `awk` به وضوح، توانایی انتخاب میدان را فراهم می‌کند؛ مشکل عمده عبور تعداد `n` میدان در یک برنامه `awk` می‌باشد. در اینجا یک تحقق وجود دارد:

```
awk ' {print $' $1' } '
```

`$1`، عرضه می‌شود (درون نقل قولها نمی‌باشد) و بنابراین عدد میدانی می‌شود که توسط `awk` مشاهده می‌شود. روش دیگر، استفاده از نقل قولهای دوگانه می‌باشد:

```
awk " {print 1 $ $ 1 } "
```

در این مورد، آرگومان، توسط شل تفسیر می‌شود، بنابراین، `$1` به `$` تبدیل می‌شود و `$1`، توسط ارزش `n` جایگزین می‌شود. ما روش نقل قول منفرد را ترجیح می‌دهیم چون بسیاری از `s,1` اضافی با روش نقل قول دوگانه در یک برنامه نوعی `awk` لازم هستند. دومین مثال، `addup n` می‌باشد که شماره‌ها را به `n` امین میدان اضافه می‌کند:

```
awk' { s += '$ $ 1'}
END {print s}'
```

سومین مثال، مجموع‌های مجزایی را از هر کدام از ستونهای n - همراه با مجموع کل تشکیل می‌دهد:

```
awk'
BEGIN{n = ' $ 1'}
{
    for (i=1 , i < n, I+t)
        sum [i] t = $I
}
END {
    for (i=1 , i <= n, i+t){
        Printf " % 4g " , sum[i]
    }
    printf " , total = % 4g \ n " , total
}'
```

ما از BEGIN برای درج ارزش n در یک متغیر استفاده می‌کنیم، به جای اینکه مابقی برنامه را با نقل قولها دسته‌بندی کنیم. مشکل عمده با همه این مثالها، نگهداشتن رد چیزی نیست که در داخل یا خارج از این نقل قولها می‌باشد (اگر چه، یک دردسراست)، بلکه همانگونه که اخیراً نوشته شده است، این برنامه‌ها می‌توانند فقط وردی استاندارد خود را بخوانند؛ راهی برای عبور آنها، هم پارامتر n و هم یک فهرست باند اختیاری از اسامی فایلها وجود ندارد. چنین چیزی نیازمند برنامه‌نویسی شل می‌باشد که ما در فصل بعد به آن می‌پردازیم.

یک سرویس تقویم بر اساس *awk*

مثال اخیر ما از آرایه‌های انجمنی استفاده می‌کند؛ همچنین یک شرح از چگونگی بر هم کنش با شل وجود دارد و تا حدودی در مورد ارزیابی برنامه، شرح می‌دهد.

وظیفه، این است که سیستم هر روز صبح پستی را برای شما بفرستد که شامل یک تقویم از وقایع آینده باشد. ممکن است چنین سرویس تقدیمی وجود داشته باشد؛ به 1 (calender) مراجعه کنید. این بخش یک روش دیگر را نشان می‌دهد.

سرویس اصلی باید به شما وقایعی را بگوید که امروز اتفاق می‌افتد؛ دومین مرحله، دادن یک هشدار روزانه می‌باشد - وقایع فردا همانند وقایع امروز، استفاده صحیح از پایان هفته‌ها و تعطیلات به عنوان یک تمرین باقی می‌ماند.

اولین شرط لازم مکانی است که تقویم را نگه دارد. برای چنین کاری یک فایل با عنوان `calender in/ usr / you`، آسانترین راه می‌باشد.

```
$ cat calender
    sep 30 mother's birthday
    oct 1  lunch with joe.noon
    oct 1  metting 4 pm
$
```

دوم، شما نیاز به یک روش برای پویش کردن تقویم برای یک تاریخ دارید. در اینجا انتخابهای زیادی وجود دارند؛ ما از *awk* استفاده می‌کنیم چون در انجام محاسبات لازم برای رفتن از امروز به فردا، بهترین می‌باشد، اما شاید برنامه‌ها مانند *sed* یا *egrep* نیز می‌توانند موثر باشند. سطرهای انتخاب شده از تقویم، توسط پست الکترونیکی، ارائه می‌شوند.

سوم اینکه، شما نیاز به یک روش برای داشتن تقویم پوشش شده به صورت قابل اعتماد و خودکار در هر روز و احتمالاً در صبح، دارید. چنین چیزی می‌تواند با `at` انجام شود، چیزی که ما به طور خلاصه در فصل اول ذکر کردیم.

اگر ما فرصت تقویم را محدود کنیم در نتیجه هر سطر با یک نام ماه و روز به عنوان تاریخ آغاز می‌شود، اولین نمونه برنامه تقویم آسان است:

```
$ date
Thu sep 29 15:23 : 12 EDT 1983
$ cat bin / calendar
# calendar:Version 1 - - today only
awk < $ HOME / calendar
      BEGIN { split (" " \ date \ " " , date)}
$ 1== date [2] $ $ 2== date [3]
  | mail $NAME
$
```

بلوک `BEGIN`، تاریخ ایجاد شده توسط `date` را در یک آرایه تقسیم می‌کند؛ دومین و سومین عنصر آرایه، ماه و روز می‌باشند. ما فرض می‌کنیم که متغیر `Name` از `shell`، شامل اسم `login` شما می‌باشد.

توالی قابل توجه کاراکترهای نقل قول، نیازمند داشتن تاریخ در یک رشته در وسط برنامه `awk` می‌باشند. یک روش دیگر که آسانتر می‌باشد این است که تاریخ را به عنوان اولین سطر ورودی بپذیریم:

```
$ cat bin / calendar
# calendar : version 2 - - today only .no quotes
(date , cat $HOME / calendar) |
awk '
NR == 1 {mon = #2 , day = $3} # set the date
NR > 1 $ $ 1== mon $ $ 2 == day # print calendar lines
  | mail $ NAME
```

مرحله بعدی، مرتب کردن تقویم برای دیدن وقایع فردا همانند امروز می‌باشد. اکثر اوقات، همه آن چیزی که مورد نیاز است، گرفتن تاریخ امروز و اضافه کردن ۱ به روز است. اما در پایان ماه، ما باید ماه بعد را وارد کنیم و روز را دوباره به یک برگردانیم. و البته هر ماه دارای یک تعداد متفاوت از روزها می‌باشد.

اینجا جایی است که آرایه انجمنی سودمند می‌باشد. دو آرایه، `days`، `Nextman`، که زیرنویس‌های آنها اسامی ماه هستند، تعداد روزها در یک ماه و نام ماه بعدی را با خود دارند. بنابراین 31. `days[" Jan"]` و `days[" Feb,next mon"]` (فوریه) می‌باشد. به جای ایجاد یک توالی کامل از بیاناتی مانند

```
dsys [" Jan" ] = 31 , next mon ["Jan" ] = " Feb"
days [" Feb" ] = 28 , nextmon [" Feb" ] = " mar"
```

ما از `split` برای تبدیل یک ساختار مناسب از داده‌ها به یک ساختار حقیقتاً مورد نیاز استفاده می‌کنیم:

```
$ cat calendar
# calendar: version 3 - - today and tomorroew
awk <$ HOME / calendar
BEGIN {
x = " Jan 31 Feb 28 Mar 31 Apr 30 may 31 "
  " Jan 31 Aug 31 sep30 oct31 nov30 Dec31 Jan 31"
split (x , data)
```

```

for (i=1 , I (24 , I t=2))
days [data[i] ]=data[i+1]
next mon [dafa[i]]=data[I+2]
}
split (" , " \ date ' " , " , date)
mon /= date [2] , day /= date [3]
mon2= mon/ , day 2 = day 1+1
if(day) > = days [mon1] )}
day 2 =1
mon 2=next mon [ mon1]
}
$ 1 == mon 1 $ $ $2 == day || $1 == mon 2 $ $ $2 == day 2
| mail $ NAME
$

```

توجه داشته باشید که Jan در یک داده دوباره ظاهر می شود ؛ یک ارزش داده « اشاره نما » مانند این، پردازش را برای دسامبر آسان می کند.

مرحله نهایی، مرتب کردن برنامه تقویم به گونه ای می باشد که هر روز اجرا شود. چیزی که شما می خواهید این است که هر روز صبح حدود ساعت ۵ صبح از خواب بیدار شوید و `calendar` را اجرا کنید. شما می توانید این کار را با به خاطر آوردن برای `say` خودتان انجام دهید (هر روز !)

```

$ at 5am
calendar
ct1-d
$

```

اما چنین چیزی دقیقاً قابل اعتماد یا خودکار نمی باشد. ترفند این است که `at` رانه فقط برای اجرای تقویم، بلکه همچنین برای برنامه ریزی اجرای بعدی نیز بیان کنیم.

```

$ cat early . morning
calendar
echo early . morning | at 5 am
$

```

دومین سطر، فرمان `at` دیگری را برای روز بعد، برنامه ریزی می کند، در نتیجه زمانی که شروع می شود، این توالی، به صورت دائمی می باشد. فرمان `at PATH` ، شما را و نیز فهرست جاری و سایر پارامترها را برای فرمانهایی که پردازش می کند، تعیین می کند، بنابراین شما نباید کار خاصی انجام دهید.

تمرین ۱۱-۴ . تقویم را به گونه ای تغییر دهید که در خصوص آخر هفته ها نیز بداند. در جمعه، «فردا» شامل شنبه، یکشنبه و دوشنبه می باشد. تقویم را به گونه ای تغییر دهید که به صورت پرشی از سال استفاده کند. آیا تقویم در خصوص تعطیلاتها باید چیزی می داند؟ چگونه آن را مرتب می کنید؟

تمرین ۱۲-۴ . آیا تقویم در خصوص تاریخهای موجود در یک سطر باید بداند، نه فقط در آغاز سطر؟ در خصوص تاریخهای ارائه شده در سایر فرمتها مانند ۱۰/۱/۸۳ چگونه؟

تمرین ۱۳-۴ . چرا تقویم به جای `$NAME` از `getname` استفاده نمی کند؟

تمرین ۱۴-۴. یک نسخه شخصی از `im` بنویسید که فایلها را وارد یک فهرست موقتی کند به جای اینکه آنها را حذف کند، این کار را با یک فرمان `at` برای پاک کردن فهرست انجام دهید زمانی که شما خواب هستید.

پایانهای آزاد

`awk` یک زبان بد ترکیب می باشد، و نشان دادن همه تواناییهای آن در یک فصل با اندازه معقول، غیرممکن می باشد. در اینجا چیزهای دیگری وجود دارند که در کتاب راهنما به آنها می پردازیم:

• جهت دهی خروجی `print` در فایلها و لوله ها: هر بیان `print` یا `printf`، می تواند با یک `>` و یک اسم فایل دنبال شود (به عنوان یک رشته نقل قول شده یا در یک متغیر)؛ خروجی به همان فایل فرستاده می شود. همانند شل «`<<`» به جای روی هم نویسی، ضمیمه می شود. پرینت کردن درون یک لوله به جای `>` نیاز به `|` دارد.

• رکوردهای چند سطری: اگر جداساز `RS` رکورد، برای سطر جدید تعیین شود، در نتیجه رکوردهای ورودی توسط یک سطر خالی از هم جدا می شوند. در این روش، سطرهای ورودی متعددی می توانند به عنوان یک رکورد منفرد رفتار کنند.

• «طرح، طرح» به عنوان یک گزینش گر: همانند `sed`، `ed` یک دامنه از سطرها می تواند توسط یک جفت طرح مشخص شوند. چنین چیزی سطرها را از یک رخداد از اولین طرح تا رخداد بعدی دومین طرح، تطبیق می دهد. یک مثال ساده عبارت است از

```
NR == 10 , NR == 20
```

که سطرهای ۱۰ تا ۲۰ شامل را تطبیق می کند.

۴.۵ فایلها و فیلترهای خوب

اگر چه مثالهای اخیر در مورد `awk`، فرمانهای خود شامل هستند، اما اکثر استفاده های `awk`، برنامه های ساده یک سطری یا دو سطری برای انجام پای ییدن به عنوان بخشی از یک خط لوله بزرگتر می باشند. چنین چیزی در مورد اکثر فیلترها صحیح می باشد- گاهی اوقات مشکل موجود می تواند با به کار بردن یک فیلتر تنها حل شود. اما عموماً، این مشکل به مشکلات فرعی قابل حل توسط فیلترهای متصل شده به هم در یک خط لوله، تجزیه می شود. این استفاده از ابزار اغلب به عنوان قلب محیط برنامه نویسی یونیکس ذکر می شود. این نظریه، کاملاً محدود کننده می باشد؛ با این وجود، استفاده از فیلترها، سیستم را گسترش می دهد و مشاهده چگونگی عملکرد آن، ارزشمند است.

خروجی تهیه شده توسط برنامه های یونیکس، در یک فرمت می باشد و به عنوان ورودی توسط سایر برنامه ها قابل درک است. فایل های قابل فیلتر، شامل سطرهایی از متن، فاقد عنوانهای تزئینی، دنباله روها یا سطرهای فاصله می باشند. هر سطر یک هدف از منفعت می باشد - یک اسم فایل، یک کلمه، یک توصیف از یک فرآیند اجرا - بنابراین برنامه هایی مانند `wc` و `grep` می توانند اقلام جالب را بشمارند و یا از طریق نام در جستجوی آنها باشند. زمانی که اطلاعات بیشتری برای هر هدف ارائه می شود، فایل هنوز سطر به سطر است، اما درون زمینه های جدا شده توسط فاصله ها یا جدول بندی ها ستون بندی می شود، مانند خروجی `ls-l`. با توجه به داده های تقسیم شده به چنین زمینه هایی، برنامه هایی مانند `awk` می توانند به آسانی انتخاب شوند، توسعه یابند و یا دوباره اطلاعات را مرتب کنند.

فیلترها، دارای یک طرح مشترک می‌باشند. هر کدام از آنها بر روی خروجی استاندارد خود، نتیجه پردازش فایل‌های آرگومان را می‌نویسد و یا بر روی ورودی استاندارد، اگر هیچ آرگومانی ارائه نشود، آرگومان‌ها فقط ورودی‌ها را مشخص می‌کند و هرگز خروجی را مشخص نمی‌کند. - در نتیجه خروجی یک فرمان می‌تواند همیشه درون یک خط لوله‌ای پیش برود. آرگومانهای انتخابی- (یا آرگومان‌هایی که اسم فایل نمی‌باشد مانند طرح `grep`)، قبل از اسامی فایلها قرار می‌گیرند. در آخر، پیغامهای اشتباه بر روی خطای استاندارد نوشته می‌شوند، بنابراین، آنها درون یک لوله ناپدید نمی‌شوند.

این تبدیل‌ها، دارای اثرات اندکی بر روی فرمانهای فردی می‌باشند، اما زمانی که به طور یکنواخت برای همه برنامه‌ها بکار می‌روند، منجر به یک سهولت در امر ارتباط می‌شوند، چیزی که توسط مثالهای زیادی در سراسر این کتاب شرح داده می‌شود، اما شاید به طور چشمگیری، توسط مثال شمارش کلمه در پایان بخش ۴.۲ شرح داده شوند. اگر همه برنامه‌ها خواهان یک ورودی نامگذاری شده یا فایل خروجی باشند، برای تشخیص پارامترها نیازمند ارتباط باشند، و یا عنوان‌ها و دنباله‌ها را بوجود آورند، خط لوله‌ای کار نمی‌کند - و البته، اگر سیستم یونیکس لوله‌ها را تهیه نکرده باشد، فرد باید یک برنامه تبدیلی برای انجام این کار بنویسد. اما لوله‌ها وجود دارند و خط لوله‌ای کار می‌کند و حتی نوشتن آسان است اگر شما با ابزارها آشنا باشید.

تمرین ۴.۱۵ . ps ، یک عنوان توضیحی را پرینت می‌کند و ls-1 تعداد کل بلوکها را در فایلها اعلان می‌کند. شرح دهید.

تاریخچه و نکات کتاب‌شناسی

یک بررسی خوب از طرح تطبیق کننده الگوریتم ها، می‌تواند در مقاله « طرح در رشته‌ها تطبیق می‌کند » نوشته ال آهو نویسنده `grep` یافت شود. (دادرسی‌های سمپوزیوم در خصوص تئوری زبان رسمی به سانتا باربرا ۱۹۷۹) .

`Sed` طراحی شد و توسط ال آهو، پیتر و ینبرگر و بر این کرنیگهان، از طریق یک فرآیند اندکی دقیق‌تر، بکار گرفته شد. نام‌گذاری یک زبان پس از نویسندگان آن نیز یک کمبود خاص از تصور را نشان می‌دهد. یک مقاله توسط محققان `Awk` - یک پویش دهی طرح و زبان پردازش؛ نرم افزار- تجربه و عمل، جولای ۱۹۷۸، به بحث و بررسی در خصوص طرح می‌پردازد.

`awk`، در حیطه‌های متعدد دارای منابع خودش می‌باشد، اما مطمئناً عقاید خوبی را از `SNOBOL 4`، از `sed`، از یک زبان معتبر طراحی شده توسط مارک روچکاینند، از ابزار زبان `lex`، `yacc` و البته از زبان `C`، به سرقت برده است. حقیقتاً، شباهت بین `awk` و `C`، منبع مشکلات می‌باشد- زبان شبیه `C` است اما زبان `C` نیست. برخی از ساختارها مفقود هستند؛ سایر ساختارها در روشهای دقیق، متفاوت هستند.

یک بند توسط داگ کامر، با عنوان « سیستم فایل ثابت FFC » می‌باشد :

یک سیستم پایگاه داده‌ها شامل اصول اولیه (نرم‌افزار - عمل و تجربه، نوامبر ۱۹۸۲)، به بحث و بررسی در خصوص استفاده از شل و `awk` برای ایجاد یک سیستم پایگاه داده‌ها می‌پردازد.

فصل پنجم: برنامه ریزی پوسته

با وجود آنکه اکثر کاربران پوسته را یک عنصر فرمان تبدلی می‌شمارند، اما در واقع نوعی زبان برنامه‌نویس است که در آن هر عبارت فرمانی را اجرا می‌کند. از آنجا که باید هر دو زمینه تبدلی و برنامه‌ریزی اجرای دستور را، تأمین کند، یک زبان غیرعادی است که بیشتر با تاریخچه شکل گرفته تا با طرح. محدوده کاربرد آن، حجم نابسامانی از جزئیات در زبان برنامه‌نویسی را به همراه دارد. این فصل، مبانی برنامه‌ریزی پوسته را به همراه نشان دادن سیر تکامل برخی برنامه‌های پوسته، تشریح خواهد کرد، اما باید توجه کرد که راهنمایی برای پوسته نیست، چنین چیزی در صفحه راهنمای (1)sh از راهنمای برنامه نویسان یونیکس، که باید همواره در حین خواندن کتاب، آنرا در دسترس داشته باشید، یافت می‌شود.

همچون اکثر دستورات، در پوسته نیز یافتن جزئیات رفتار برنامه، به سرعت و باتجربه، تحقق می‌یابد. راهنمای استفاده می‌تواند پیچیده باشد، اما هیچ چیز برای درک مسایل، بهتر از یک مثال مناسب نیست. بدین منظور، این فصل بیشتر حول مثالها متمرکز شده تا ویژگیهای پوسته و راهنمایی است جهت استفاده از پوسته برای برنامه‌نویسی، نه فقط دایره‌المعارفی از توانائی‌های آن. ما تنها در مورد آنچه پوسته می‌تواند انجام دهد، صحبت نمی‌کنیم، بلکه در مورد توسعه و نوشتن برنامه‌های پوسته، با تأکید بر آزمایش و تبادل نظرات آزمایشی نیز سخن به میان می‌آوریم.

هنگامی که شما یک برنامه را نوشته‌اید، در پوسته یا در هر زبان دیگر، ممکن است استفاده از آن توسط سایر افراد به قدر کافی مفید باشد، اما استانداردهایی که سایرین از یک برنامه انتظار دارند، معمولاً موشکافانه تر از استانداردهای شخص است که در مورد خود به کار می‌گیرد. نمای کلی در برنامه‌نویسی پوسته نیرومند کردن برنامه‌ها به گونه ایست که بتوانند ورودی غیرصحیح را منتقل کرده و به هنگام غلط بودن مواردی، اطلاعات مفیدی ارائه دهند.

۵.۱. سفارشی کردن دستور cal

یکی از استفاده‌های معمول برنامه پوسته، اصلاح یا بهبود ارتباط کاربر با برنامه است. به عنوان مثالی از یک برنامه که بهبود را نشان دهد، به فرمان (1)cal را توجه کنید:

```
$ cal
usage: cal [month]year          Good so far
$ cal october 1983
bad argument                    Not so good
$ cal 10 1983
  october 1983
S  M  Tu  W  Th  F  S
                1
2  3  4  5  6  7  8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$
```

اعمال ماهها بصورت عدد، ایجاد مزاحمت می کند و هنگامی که برگردانیده شود، 10 cal برای کل سال، ۱۰ را چاپ می کند، بنابراین باید همواره برای مشاهده تقویم برای یک ماه، سال را مشخص کنید.

نکته مهم در اینجا این است که فرقی نمی کند که فرمان cal چه ارتباطی را تأمین می کند، می توانید بدون تغییر خود cal، آنرا تغییر دهید. می توانید دستوری را در دایرکتوری bin شخصی خود، که یک متن مناسبتر را به آنچه cal واقعی نیاز دارد تبدیل می کند، قرار دهید. حتی می توانید cal نسخه خود را بازخوانی کنید که به معنای یک چیز کمتر برای یادآوری است.

اولین قضیه، طراحی است: cal چه باید بکند؟ اصولاً cal را برای منطقی بودن نیاز داریم و باید هر ماه را به اسم بشناسد. به دو دلیل، باید همانند cal قدیمی عمل کند، به جز در مورد برگرداندن نام ماهها به اعداد بنا به آرگومان یک، cal باید ماه یا تقویم سال را به همان درستی چاپ کند و مطابق آرگومان صفر، باید تقویم جاری را چاپ کند، چرا که این امر، مطمئناً عمده ترین استفاده فرمان است. پس مسئله این است که تشخیص داده شود که چند آرگومان موجود است و سپس آنها را آنگونه که cal استاندارد نیاز دارد، مطرح کنیم. پوسته، یک عبارت با عنوان case را نیز که مناسب اتخاذ چنین تصمیمهاییست را نیز پیش رو می گذارد:

Case word in

pattern) commands ;;

pattern) commands ;;

...

esac

عبارت word,case را با نمونه (pattern) ها، از بالا تا پایین مقایسه می کند و باعث می شود که فرامین، با اولین و تنها اولین نمونه ای که مطابقت کند، همراه شوند. نمونه ها با استفاده از قوانین تطابق نمونه پوسته نوشته می شوند، که تاحدی تعمیم یافته آنچه می باشند که برای انطباق نام فایلها موجود می باشد. هر عمل با دو ; خاتمه یافته است. (; ممکن است در انتهای آخرین عبارت جایگذاری شده باشد، اما برای ویرایش ساده تر، معمولاً آنرا داخل می گذاریم.)

نسخه cal ما تصمیم می گیرد که چند آرگومان وجود دارد، به پردازش الفبایی نام ماهها مبادرت می کند، سپس cal واقعی را فراخوانی می کند. متغیر پوسته \$#، تعداد آرگومانهایی را که یک فایل پوسته با آنها فراخوانی می کند، در خود نگهداری می کند، سایر متغیرهای خاص پوسته، در جدول ۱-۵ فهرست شده اند.

\$ cat cal

\$ cal: nicer interface to /use/bin/cal

case \$# in

0) ;; set `date` ; m=\$2; y=\$6 # هیچ آرگومانی موجود نیست: از امروز استفاده می شود

1) ;; m=\$1; set `date`; y=\$6 # یک آرگومان: از امسال استفاده می شود

*) m=\$1; y=\$2 ;; # دو آرگومان: ماه و سال

esac

case \$m in

jan*|Jan*) m=1 ;;

feb*|Feb*) m=2 ;;

mar * | Mar *) m=3 ;;

apr * | Apr *) m=4 ;;

may * | May*) m=5 ;;

```

jun* | Jun*)           m=6 ;;
jul* | Jul*)           m=7 ;;
aug* | Aug*)           m=8 ;;
sep* | sep*)           m=9 ;;
oct* | oct*)           m=10 ;;
nov* | Nov*)           m=11 ;;
dec* | Dec*)           m=12 ;;
[1-9] | 10 | 11 | 12) ;;      # ماه به عدد
*)           y=$m; m=" " ;;   # سال عادی
esac
/usr/bin/cal $m $y          # تقویم واقعی اجرا می شود

```

اولین عبارت تعداد آرگومانی را بررسی می کند، # \$، و عملکرد مناسب را انتخاب می کند. نمونه * انتهای در case اول catch-all : اگر شماره آرگومان نه صفر باشد و نه یک، آخر عبارت اجرا خواهد شد. (از آنجایی که نمونه ها به ترتیب جاروب می شوند، catch-all باید آخرین باشد.) این امر y,m را برای ماه و سال انتخاب می کند - دو آرگومان داده شده و cal ما همانند نمونه اصلی عمل خواهد کرد.

اولین عبارت case، دارای یک جفت خط پیچیده شامل

Set `date`

است. با وجود اینکه از ظاهر آن مشخص نیست، اما می توان به سادگی با آزمایش کردن آن دریافت که این عبارت چه عمل انجام می دهد:

جدول ۵.۱: متغیرهای درونی پوسته

\$#	تعداد آرگومانها
\$*	تمامی آرگومانهایی که وارد پوسته می شوند
\$@	مشابه \$*؛ بخش ۵-۷ را ببینید
\$ -	گزینه هایی که برای پوسته فراهم شده اند.
\$?	مقدار بازگشتی آخرین فرمان اجرا شده
\$\$	شناسایی فرآیند پوسته
\$!	شناسایی فرآیند آخرین فرمانی که با & آغاز شده
\$Home	آرگومان پیش فرض برای فرمان cd
\$IFS	فهرست کاراکترهایی که کلمات را در آرگومانها از هم جدا می کنند
\$MHJL	فایلی که هنگام تغییر یافتن، پیغام "you have mail" را فعال می کند.
\$PATtl	فهرست دایرکتوریهای جستجوی فرامین
\$PS1	رشته فوری، پائین فرض '\$'
\$PS2	رشته فوری برای خط فرمان ادامه دار، پائین فرض '>'

```
$ date
sat oct 1 06 : 05 : 18 EDT 1983
$ set `date `
$ echo $1
sat
$ echo $4
06 : 05 : 20
$
```

set یک دستور درونی پوسته است که کارهای بسیاری انجام می دهد. بدون هیچ آرگومانی، مقدار متغیرها را در محیط نشان می دهد، همانگونه که در فصل ۳ دیدیم. آرگومانهای متعارف، مقادیر \$1، \$2 و غیره را صفر می کند. از این رو '\$ `date `set' را برای روز هفته انتخاب می کند، \$2 را برای نام ماه و همینطور بقیه. بنابراین اولین case در cal، ماه و سال را از تاریخ جاری تنظیم می کند، در صورتیکه آرگومانی موجود نباشد؛ اگر یک آرگومان داشته باشیم به عنوان ماه در نظر گرفته می شود و سال از تاریخ جاری استخراج می شود.

Set همچنین چندین گزینه را که اغلب آنها -x و -V می باشند، شناسایی می کند که فرمانهای بازتابی را به موازات پیشبرد توسط پوسته، به کار می اندازند که برای عیب یابی برنامه های پیچیده پوسته، حیاتی می باشند.

مسئله ای که باقی می ماند، برگرداندن ماه است به عدد، اگر به فرم متنی باشد. این امر بوسیله عبارت case دوم انجام می گیرد که باید کاملاً واضح و بدیهی باشد. تنها دوگانگی این است که کاراکتر | در عبارت case، مانند egrep، دلالت بر یک انتخاب دارد: | big small منطبق با بر big است یا small. البته این حالتها، باید بصورت * [jJ] an یا مشابه آن نوشته شوند. برنامه اسامی ماهها را هم از تمامی قضایای پایتتر دریافت می کند، چرا که بیشتر فرامین، ورودی وضعیت پایین را قبول می کنند، و یا با اولین حرف بزرگ، چرا که date این ساختار را چاپ می کند. قاعده های تطبیق نمونه های پوسته در جدول ۲-۵ آورده شده است.

جدول ۲-۵: قوانین تطبیق نمونه های پوسته

*	هر رشته ای را مطابقت می دهد، از جمله رشته خنثی
?	هر رشته منفرد را انطباق می دهد
[CCC]	هر کدام از کاراکترها را د ccc مطابقت می دهد.
?\$	[a-do-3] معادل است با [abcdo123]
"..."	دقیقاً ... را مطابقت می دهد. کوتیشن ها. کاراکترهای سویژ-مسل-نگه می دارند، همچنین '...'
\c	C را جزء به جز تطبیق می دهد.
a b	تنها در عبارات a, case یا b را تطبیق می دهد
/	در نام فایلها، آنهایی را که فقط با یک / در عبارات، تطبیق داده شده اند؛ در case هر کدام که مانند سایر کاراکترها مطابقت دارند.
0	به عنوان اولین کارکتر نام یک فایل، فقط با یک 0 در عبارت تطابق دارد.

دو عبارت آخر در case دوم، دلالت بر یک آرگومان ساده دارند که می تواند یک سال باشد، به خاطر بیاورید که اولین عبارت case، آنرا یک ماه فرض کرده بود. اگر شماره ای باشد که بتواند نشان دهنده ماه باشد، تنها رها می شود، در غیراین صورت به عنوان سال تلقی

می شود.

سرانجام، آخرین خط، cal (usr/bin/cal واقعی) را با آرگومانهای تغییر یافته، فراخوانی می کند. نسخه cal ما، به گونه ای عمل می کند که یک تازه وارد انتظار دارد:

```
$ date
sat oct 1 06 : 09 : 55 EDT 1983
$ cal
  october 1983
S  M  Tu  W  Th  F  S
                1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$ cal dec
  December 1983
S  M  Tu  W  Th  F  S
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 34
25 26 27 28 29 30 31
$
```

و ۱۹۸۴ cal، تقویم را برای کل ۱۹۸۴ چاپ می کند.

برنامه cal ارتقا یافته ما، همان کار را مانند نمونه اصلی انجام می دهد اما با یک روش ساده تر و قابل حفظ تر. از این رو آنرا بیشتر cal می خوانیم تا calendar (که یک دستورات) یا چیزی که کمتر به یاد می ماند مانند ncal. تنها نهادن این نام، این خوبی را دارد که کاربران نیاز به توسعه و افزایش بازتابی را جهت چاپ تقویم، ندارند.

پیش از آنکه از case بگذریم، بهتر است به خلاصه ای از این موضوع پردازیم که چرا قوانین تطبیق نمونه ها در پوسته متفاوت از آنچه در ed و مشتقات آن می باشند. با این همه، دو نوع نمونه، به معنای دو مجموعه قوانین برای یادگیری و دو بخش شامل کد جهت پردازش آنها می باشد. برخی از تفاوتها، گزینه های بعدی می باشند که هرگز اصلاح نشدند، برای مثال هیچ دلیلی غیر از سازگاری برای این که برای تطبیق هر کاراکتر، ed از نقطه استفاده می کند و پوسته از ؟، نیست. اما گاهی نمونه ها، کارهای متفاوتی انجام می دهند عبارات با قاعده در ویرایشگر، به دنبال رشته ای می گردند که ممکن است هر جایی خط واقع شود؛ کارکترهای ویژه ^ و \$، جهت اتصال جستجو به ابتدا و انتهای خط می باشند. به هر حال، برای نام فایلها، باید جستجو را به صورت پیش فرض، تثبیت کنیم، چرا که بسیار پیش می آید که مجبور باشیم بنویسیم:

```
$ 1S ^?*. C$
```

به جای

\$ 1S *.C

که بسیار ایجاد مزاحمت می کند.

تمرین ۵-۱. اگر کاربران نسخه cal شما را ترجیح دهند، برای استفاده جهانی از آن چه تبدیری می اندیشید؟ برای گذاشتن آن در `usr/bin/` چه باید کرد؟

تمرین ۵-۲. آیا تنظیم `cal` به گونه ای که `cal ۸۳` تقویم ۱۹۸۳ را چاپ کند، ارزشمند می باشد؟ اگر چنین است، تقویم ۱۹۸۳ را چگونه چاپ می کنید؟

تمرین ۵-۳. `cal` را بگونه ای تعریف کنید که بیشتر از یک ماه را قبول کند، همانند:

\$ cal oct nou

یا در صورت امکان تعدادی از ماهها را:

\$ cal oct - dec

اگر اکنون دسامبر باشد و شما `cal jan` را بخواهید، ژانویه سال جاری را خواهید دید یا سال آینده را؟ چه زمانی باید اضافه کردن ویژگیها را به `cal` متوقف کنید. □

۲-۵. *which* کدام دستور است؟

در ساختن نسخه های خصوصی از دستورهایی مانند `cal`، مشکلاتی وجود دارد. واضح ترین مشکل، این است که اگر با `Mary` کار می کنید و در حالی `cal` را تایپ کنید که با عنوان `mary` وارد سیستم شده باشید، `cal` استاندارد را به جای نمونه جدید خواهید دید، البته مگر اینکه `Mary cal` جدید را با دایرکتوری `bin` خود مرتبط ساخته باشد. این امر می تواند گمراه کننده باشد. به خاطر بیاورید که پیغامهای خطا از `cal` اصلی، چندان مفید نمی باشند. اما تنها می تواند مثالی از یک مسئله کلی باشد. از آنجایی که پوسته در یک مجموعه از دایرکتوریها که بوسیله `PATH` مشخص شده اند، به دنبال دستورات می گردد، همیشه امکان این وجود دارد که به نسخه ای از یک دستور، غیر از آنچه انتظار دارید، برخورد کنید. برای مثال، اگر دستوری را تایپ کنید، فرضاً `echo`، نام مسیر فایلی که واقعاً اجرا می شود، می تواند `usr/bin/echo/0` یا چیز دیگری باشد، که بستگی به اجزاء `PATH` و محل قرارگیری فایلها دارد. وقوع یک فایل اجرایی با نام صحیح اما رفتار اشتباه غیر از آنچه انتظار دارید در جستجو، می تواند گمراه کننده باشد. احتمالاً معمول ترین نمونه، دستور `test` می باشد که بعداً به آن خواهیم پرداخت: نام آن برای یک نسخه موقت برنامه، چنان واضح است که برنامه `test` اشتباه، می تواند بطور آزار دهنده ای فراخوانی شود. دستوری که گزارش دهد کدام نسخه برنامه اجرا خواهد شد، بسیار مفید و مورد استفاده خواهد بود.

یک شیوه اجرا، حلقه زدن زیاد دایرکتوری هایبست که در `PATH` نام برده شده و جستجوی هرکدام برای یک فایل قابل اجرا است که نام آن موجود می باشد. در فصل ۳، از `FOR` برای حلقه زدن نام فایلها و آرگومانها استفاده کردیم. در اینجا، به حلقه ای نیاز داریم که بگوید:

For i در هر جز از `PATH`

do

اگر نام ارائه شده در دایرکتوری، `i` موجود است.

نام کامل سیر آن را چاپ کن

done

از آنجا که می‌توانیم هر دستوری را درون '...' اجرا کنیم، راه حل واضح، اجرا کردن sed، پیش از PATH \$ است و تبدیل در نقطه فاصله. می‌توانیم این موضوع را با دوست قدیممان echo امتحان کنیم.

```
$ echo $ PATH
```

```
: usr/you/bin : .bin : /usr/bin
```

جزء ۴

```
$ echo $ PATH | sed 's/:/ /g'
```

```
/usr / you / bin / bin / usr / bin
```

!تنها ۳ چاپ شده

```
$ echo `echo $ PATH | sed 's/:/ g' `
```

```
/usr / you / bin / bin / usr / bin
```

هنوز فقط 3

```
$
```

به وضوح یک مسئله وجود دارد. یک رشته خنثی در PATH، مترادف است با «.». تعویض دو نقطه با فاصله، در PATH، به قدر کافی مناسب نیست - اطلاعات مربوط به اجزاء خنثی. از بین می‌روند.

بعداً خواهیم دید که چگونه این مشکل را در فایل‌های پوسته، در جایی که معمولاً از test استفاده می‌شود. از بین ببریم.

برای ایجاد فهرست صحیح دایرکتوریها، باید یک جزء بی‌اثر از PATH را به نقطه تبدیل کنیم. جزء بی‌اثر، می‌تواند هم در وسط و هم در آخر رشته باشد، از این رو، دستیابی به تمام حالات، کار زیادی نمی‌برد:

```
$ echo $ PATH | sed 's/^ : / : /g'
> s/^ : : / : : /g
> s/^ : $ / : /g
> s/^ : / /g'
. /usr / you / bin / bin / usr / bin
$
```

می‌توانستیم آنرا بصورت چهار sed جدا از هم نوشته باشیم، اما از آنجایی که sed، جابه‌جایی را به ترتیب انجام می‌دهد، یک تقاضا می‌تواند تمامی آنها را انجام دهد.

هنگامی که اجزاء دایرکتوری-PATH را داشته باشیم، دستور-test(1) که به آن اشاره کردیم. می‌تواند اعلام کند که فایلی در هر دایرکتوری وجود دارد. دستور test در واقع یکی از برنامه‌های خام تز یونیکس می‌باشد. برای مثال، test-r file امتحان می‌کند که آیا file وجود دارد و قابل خواندن است و test-w file قابل نوشتن بودن را بررسی می‌کند، اما ویرایش هفتم، هیچگونه test-x را تأمین نمی‌کند (باوجود اینکه سیستم V و سایر ویرایش‌ها تأمین می‌کنند). ما به test-f که امتحان می‌کند که فایل موجود است و دایرکتوری نیست، به عبارت دیگر، یک فایل با قاعده است. به هر حال هنگامی که ویرایش‌های گوناگونی موجود باشند، برای test باید به راهنما مراجعه کرد.

هر دستور، یک وضعیت خروج را برمی‌گرداند - یک مقدار برای اینکه نشان دهد چه رخ داده، به پوسته باز می‌گردد. وضعیت خروج، یک رقم کوچک است که طبق قرارداد، Q به معنای «درست» است (دستور به درستی اجرا شده) و غیر صفر به معنای «نادرست» است

(دستور با موفقیت اجرا نشده). توجه داشته باشید که این مقادیر، برعکس مقادیر درست و نادرست در C می‌باشند. از آنجاییکه بسیاری مقادیر، می‌توانند دلالت بر «نادرست» داشته باشند، علت نقص، اغلب در وضعیت خروج «نادرست» رمز می‌شود، برای مثال، در صورت وجود تطابق، `grep` مقدار صفر را برمی‌گرداند و اگر تطبیقی پیش نیاید، یک و اگر اشتباهی در نام فایل یا نمونه باشد، ۲. هر برنامه، وضعیتی را برمی‌گرداند، هرچند معمولاً مقدار آن جالب توجه ما نیست. `test` در این مورد، غیر معمول عمل می‌کند چرا که وظیفه اصلی آن، برگرداندن وضعیت خروج است و هیچ خروجی ایجاد نکرده و تغییری در هیچ فایلی نمی‌دهد. پوسته وضعیت خروج آخرین برنامه را در متغیر `?` نگهداری می‌کند:

```
$ cmp/usr/you/.profile /usr/you/.profile
$      هیچ خروجی : مشابهند
$ echo $ ?
0
Zero implies ran O.K. : files identical
$ cmp /usr/you/.profile /usr/mary/.profile
/usr/you/.profile /usr/mary/.profile differ : char 6, line 3
$ echo $ ?
1
غیر صفر به معنای تفاوت فالتیست
$
```

تعدادی از دستورات مانند `cmp` و `grep`، دارای یک گزینه `S` - می‌باشند که باعث خروج آنها با یک وضعیت صحیح می‌شود اما تمامی خروجی‌ها متوقف می‌کند.

عبارت `if` پوسته، فرامینی را اجرا می‌کند که براساس وضعیت خروج یک دستور می‌باشند، مانند

```
if
    دستور
then
    دستور می‌دهد، اگر شرط درست باشد
else
    دستور می‌دهد اگر شرط نادرست باشد
fi
```

موقعیت خطوط جدید، دارای اهمیت می‌باشد: `then`، `fi` و `else` تنها پس از یک خط جدید یا نقطه ویرگول شناخته می‌شوند. `else` اختیاری می‌باشد.

جمله `if`، همواره دستوری را اجرا می‌کند - شرطی را - در حالی که جمله `case`، تطبیق نمونه را مستقیماً در پوسته به عهده دارد. در برخی ویرایشهای یونیکس، که شامل `V System` نیز می‌باشند، `test` تابعی درونی از پوسته است از اینرو، یک `if` و یک `test`، به سرعت یک `case` عمل می‌کنند. اگر `test` درونی نباشد، جملات `case` کارآمدتر از جملات `if` می‌باشند و باید برای هر نوع تطبیق نمونه‌ای مورد استفاده قرار گیرند:

```
Case "$/" in
```

```
hello ) command
      esac
Will be faster than
```

```

if tset "$1" = hello
  کندتر مگر آنکه test دستوری درونی در پوسته باشد
  then
    command
  fi

```

یک دلیل برای اینکه گاهی از جملات case، برای امتحان کارهایی که در پوسته بوسیله یک جمله if در اغلب زبانهای برنامه نویسی انجام می‌گیرد، وجود دارد. از طرف دیگر، یک جمله case، نمی‌تواند به سادگی تشخیص دهد که فایلی، مقادیر مجاز را خوانده باشد؛ که بهتر است بوسیله یک test و if انجام گیرد.

از این رو همه چیز برای اولین نسخه از فرمان which، سر جای خودش است؛ برای نشان دادن اینکه کدام فایل به یک دستور پاسخ می‌دهد:

```

$ cat which
# Which cmd : Which cmd in PATH is executed , version 1

case $# in
0) echo 'usage : Which command' 1>&2 ; exit 2
  esac
  for i in ` echo $ PATH | 's/^:/:./
            s/::/:./g
            s/:$/:./
            s/:/ /g' `
  do
    if test -f $i/$1      # use test -x if you can
    then
      echo $i/$1
      exit 0              # found it
    fi
  done
  exit 1                  # not found
$

```

آنرا امتحان می‌کنیم:

```

$ cx which          آنرا قابل اجرا می‌کند
$ Which Which
./ Which
$ which ed
/bin / ed
$ mv Which /usr / you / bin
$ Which Which
/usr / you / bin / Which
$

```

جمله case مبنای تنها بررسی خطا را به عهده دارد. به جهت یابی دوباره 1>82 در echo توجه فرمایید، که پیغام خطا، مسیر را محو

نمی‌کند، دستور درونی پوستهٔ `exit`، می‌تواند به منظور برگرداندن یک وضعیت خروج مورد استفاده قرار گیرد. ما `exit 2` را برای برگرداندن یک وضعیت خطا در صورتی که دستور کار نمی‌کند، `exit 1` اگر نمی‌توانست فایل را پیدا کند و `exit 0` اگر یکی را پیدا می‌کرد. اگر هیچ جملهٔ `exit` صریحی وجود نداشته باشد، وضعیت خروج از فایل پوسته، وضعیت آخرین دستور اجرا شده است. چه اتفاقی رخ می‌دهد اگر برنامه‌ای داشته باشید که نام آن در دایرکتوری جاری `test` باشد؟ (فرض می‌کنیم که `test` دستور درونی از پوسته نیست.)
یک `Test` حلقه ایجاد می‌کند. آنرا قابل اجرا می‌کند.

```
$ echo 'echo hello' >test
$ cx test                آنرا قابل اجرا می کند
Which Which $         را امتحان می کند Which حالا
hello                  مردود می شود !
./Which
$
```

بررسی خطای بیشتری فراخوانی می‌شود. می‌توانستید `which` را به منظور یافتن نام کامل مسیر برای `test` اجرا کنید (اگر `test` در دایرکتوری جاری نبود)، و آنرا صراحتاً مشخص کنید. اما قانع کننده نیست: `test` ممکن است در دایرکتوری‌های مختلف در سیستمهای مختلفی باشد و `which` نیز بستگی به `echo` و `sed` دارد، از این رو باید نام مسیرهای آنها را نیز مشخص کنیم. یک راه حل ساده‌تر وجود دارد: `PATH` را در فایل پوسته تنظیم کنید، تا فقط در `/bin/` و `/usr/bin/` برای دستورات باشد. البته، تنها برای دستور `which`، مجبور به ذخیرهٔ `PATH` پیشین برای تعیین توالی دایرکتوری‌هایی که باید جستجو شوند می‌باشید.

```
$ cat which
# Which cmd : Which cmd in PATH is executed , final version
opath = $ PATH
PATH = / bin : / usr / bin
case $ # in
0 )    echo ' Usage : Which command ' 1 > & 2 ; exit 2
esac
for i in ` echo $ opath | sed 's/^ : / . : /
          s / : : / : : / g
          s / : $ / : . /
          s / : / / g' `
do
    if test - f $ i / $1          # this is / bin / test
    then                          # or / usr / bin / test only
        echo $i / $1
        exit 0                    # found it
    fi
done
exit 1                            # not found
$
```

اکنون which حتی اگر یک Test (یا sed یا echo) جعلی نیز در خلال جستجو وجود داشته باشد، عمل می کند.

```
$ 1S - 1 test
rwxrwxrwx 1 you 11 oct 1 06:55 test still there
$ which which
/usr / you / bin / which
$ which test
./ test
$ rm test
$ which test
/bin / test
$
```

پوسته دو عملگر دیگر را نیز برای ترکیب فرامین، تأمین می کند، || و &&، که اغلب فشرده تر و راحت تر از جمله if می باشند. برای مثال || می تواند برخی جملات if را جابجا می کند:

dose not exist نام فایل echo file || نام فایل test - f معادل است با:

! شرط را منفی می کند. نام فایل if test ! - f

```
then
    echo file filename dose not exist
fi
```

عملگر ||، علی رقم ظاهر آن، هیچ ربطی به لوله ها ندارد - عملگری است شرطی به معنای OR .

دستور سمت چپ || به اجرا درخواهد آمد. اگر وضعیت خروج آن، صفر باشد، (موفقیت)، دستور سمت راست || نادیده گرفته می شود. اگر سمت چپ، مقداری غیر صفر را برگرداند (عیب) سمت راست اجرا شده و مقدار کل عبارت، وضعیت خروج سمت راستی خواهد بود. به عبارت دیگر، || یک عملگر OR شرطی است که دستور سمت راست خود را در صورت موفقیت سمت چپ، اجرا نمی کند. && شرطی پاسخ، AND می باشد؛ دستور سمت راست خود را خود را تنها هنگامی اجرا می کند که سمت چپی موفقیت آمیز باشد.

تمرین ۴-۵. چرا PATH Which را به Opath، قبل از خروج مقدار دهی اولیه نمی کند؟

تمرین ۵-۵. از آنجائیکه پوسته از esac برای پایان دادن به یک case استفاده می کند و از fi برای اتمام یک if، چرا از done برای اتمام do استفاده می کند؟

تمرین ۶-۵. یک گزینه a - را به which اضافه کنید. بگونه ای که تمامی فایلها را در PATH چاپ کند به جای اینکه بعد از اولین فایل، خارج شود. راهنمایی: match='exit O' .

تمرین ۷-۵. Which را بگونه ای تعریف کنید که فرامین درون پوسته مانند exit را بشناسد.

تمرین ۸-۵. Which را بگونه ای تعریف کنید که مجاز بودن اجرا را روی فایلها بررسی کند. آنرا جهت چاپ یک پیغام خطا هنگام نیافتن یک فایل، تغییر دهید.

۵-۳ حلقه‌های *while* و *Loop* : مراقبت از اشیاء

در فصل سوم، از حلقه *for* برای برخی برنامه‌های تبدلی ساده استفاده می‌شود. معمولاً *For* حول مجموعه‌ای از نام فایلها حلقه می‌زند، مانند '*For i in *.c*' یا هر برهانی در برنامه پوسته همانند '*i in For \$**'. اما حلقه‌ای پوسته کلی تر از این تعابیر هستند؛ حلقه *for* را در *Which* در نظر آورید.

سه نوع حلق وجود دارد: *for* , *while* , *until* . *for* دارای بیشترین استفاده است و مجموعه ای از دستورات را اجرا می‌کند - بدنه حلقه - یکبار برای هر عضوی از مجموعه کلمات که اغلب آنها نام فایلهاست.

Until - while وضعیت خروج را از یک دستور، برای کنترل اجرای دستورات بدنه حلقه به کار می‌برد. بدنه حلقه تا هنگامی اجرا می‌شود که دستور شرط، یک مقدار غیر صفر را (برای *while*) یا صفر (برای *unit*) باز گرداند. *While* و *unit* ، مگر برای تفسیر وضعیت خروج دستور، قابل تشخیص می‌باشند.

در اینجا فرم پایه هر حلقه آورده شده است:

```
for I in          فهرست کلمات
do
```

تنظیم از روی عناصر متوالی فهرست \$i ، بدنه حلقه

```
done
```

(فهرست ، تمامی آرگومانهای فایل پوسته است، برای مثال ، \$ *) *for i* (

```
do
```

بدنه حلقه ، \$ i به آرگومانهای متوالی تنظیم می‌شود.

```
Done
```

```
While          دستور
```

```
do
```

بدنه حلقه‌ها را که دستور مقدار درست را بر می‌گرداند اجرا می‌شود.

```
Done
```

```
until          دستور
```

```
do
```

بدنه حلقه تا جایی که دستور مقدار نادرست را بر می‌گرداند، اجرا می‌شود

```
Done
```

دومین فرم *for* که در آن یک فهرست خالی، دلالت بر *\$** می‌کند، خلاصه نویسی است برای بسیاری استفاده‌های معمول. دستور شرطی که *while* یا *until* را کنترل می‌کند، می‌تواند هر دستوری باشد. یک مثال جزیی، حلقه *Loop* است که منتظر می‌ماند تا کسی وارد سیستم شود (*Mary*):

```
While sleep 60
```

```
do
```

```
    Who | grep mary
```

```
done
```

Sleep که توقفی 60 ثانیه ایجاد می‌کند، در حالت عادی همواره اجرا می‌شود (مگر اینکه

قطع گردد) و از این رو «موفقیت» را بر می‌گرداند، سپس حلقه هر یک دقیقه یک بار بررسی می‌کند که آیا Mary وارد شده یا نه.

این نسخه دارای این نقص می‌باشد که اگر Mary در حال حاضر وارد شده باشد، باید 60 ثانیه صبر کنید تا متوجه شوید.

همچنین اگر mary در این حال بماند، هر یک دقیقه یکبار از وضعیت آن، مطلع خواهید شد. محتویات حلقه را می‌توان بیرون آورده و با `until` نوشت تا اطلاعات را یکبار و بدون تأخیر تأمین کرد، اگر mary اکنون وارد شده باشد:

```
until who | grep mary
do
    sleep 60
done
```

این شرط، جالب‌تر می‌باشد، اگر Mary وارد شده باشد، `who | grep mary` مشخصات ورود او را در فهرست بندی `who` چاپ کرده و درستاً را بر می‌گرداند، چراکه `grep` یک وضعیت را جهت نشان دادن این که چیزی را یافته، بر می‌گرداند. سرانجام، این دستور را پوشش داده، به آن اسمی می‌دهیم و آنرا نصب می‌کنیم.

```
$ cat watchfor
# watchfor : watch for someone to log in
PATH = / bin : / usr / bin
case $ # in
0 ) echo 'Usage : watchfor person' 1 > &2 ; exit 1
esac
until who \ egrep " $ 1 "
do
    sleep 60
done
$ cx watchfor
$ watchfor you
you ttyo oct 1 08:01 کار می کند
$ mv watchfor / usr / you / bin آنرا نصب می کند
$
```

`grep` را با `egrep` تعویض کردیم، پس می‌توان نوشت: `$ 'watch for 'joe | mary` تا ورود بیشتر از یک نفر را کنترل کند. به عنوان یک مثال پیچیده‌تر، ورود و خروج تمامی افراد را زیر نظر می‌گیریم و هر گاه افرادی وارد یا خارج شوند، گزارشی ارائه می‌دهیم - نوعی `who` افزایشی، ساختار پایه آن، ساده است: هر یک دقیقه، `who` اجرا می‌شود، خروجی آن با یک دقیقه قبل مقایسه می‌شود و هر گونه اختلافی، گزارش می‌شود. خروجی `Who`، در یک فایل نگهداری می‌شود تا بتوان آنرا در دایرکتوری `tmp` / ذخیره کرد. برای تشخیص فایل‌های خود از فایل‌های مربوط به سایر پردازشها، متغیر پوسته `$$` (شناسه پردازش دستور پوسته) با نام فایل ترکیب

می شود که قراردادی متداول است. رمز کردن نام دستور در فایل های موقتی، اکثراً برای مدیر سیستم انجام می پذیرد. دستورات (شامل این نسخه از watch who) اغلب فایلها را در /tmp/ رها می کند و جالب است بدانید که کدام دستور این کار را انجام می دهد.

```
$ cat watch who
# watchwho : watch who logs in and out

PATH = / bin : / usr / bin
new = / tmp / wwwho 1 . $$
old = / tmp / wwwho 2 . $$
> $ old # create an ampty file
while : # lopp forever
do
    who > $ new
    diff $old $new
    mv $new $old
    sleep 60
done | awk '/>/ { $1 = "in : "; print }
        /</ { $1 = "out : "; print } '
$
```

" : یک دستور درون پوسته است که کاری به جز نشان دادن برهانهای خود و برگرداندن " درست " انجام نمی دهد. در عوض، می توانستیم از دستور true که خود یک وضعیت خروج صحیح را بر می گرداند استفاده کنیم، (همچنین یک دستور false نیز وجود دارد). اما " : مؤثرتر از true است چرا که دستور را از سیستم فایل اجرا نمی کند.

خروجی diff، از < و > برای تشخیص داده ها از دو فایل استفاده می کند. برنامه awk این مورد را برای گزارش تغییرات بصورتی بسیار ساده برای یادگیری، پردازش می کند. توجه کنید که کل حلقه while، به awk مرتبط است، به جای اینکه یک awk تازه را هر از یک دقیقه اجرا کند. Sed برای این پردازش مناسب نمی باشد، چرا که خروجی آن همواره عقب تر از ورودی آن در یک خط است: همواره یک خط ورودی وجود دارد که پردازش می شود اما چاپ نمی شود، و این امر باعث تأخیری ناخواسته می گردد.

به دلیل اینکه old خالی ایجاد شده است، اولین خروجی از Watchwho، فهرستی از تمامی کار برانی است که در حال حاضر وارد شده اند. تغییر دادن دستوری که old را می سازد، به who > \$old، باعث می شود. Watchwho تنها تغییرات را چاپ کند؛ که امری است اختیاری.

یک برنامه حلقه دیگر، برنامه ایست که متناوباً به mailbox شما نگاه می کند؛ هر گاه تیر کند، برنامه " You have mail " را چاپ می کند که جایگزینی است مناسب برای مکانیزم لرونی پوسته که از متغیر MAIL استفاده می کند. ما آنرا با متغیرهای پوسته به جای فایلها به کار گرفتیم، تا راهی متفاوت برای انجام کارها را نشان دهیم.

```
$ cat checkmail
# checkmail : watch mailbox for growth
PATH = / bin : / usr / bin
MAIL = / uar / spool / mail / `getname` # system dependent
t = $ { 1 - 60 }
x = "`ls -l $MAIL`"
while :
do
```

```

y = "` 1s -1 $ MSIL`"
echo $ x $ y
x = "$y"
sleep $t
done | awk '$4 < $1 { print "you have mail " }'
$

```

دوباره از `awk`، اینبار برای اطمینان از اینکه پیغام تنها هنگامی که `mail box` زیاد شود چاپ شود و نه هنگامی که تنها تغییر کند، استفاده کردیم. در غیر این صورت، دقیقاً بعد از حذف یک `mail`، پیغامی مشاهده خواهید نمود. (ویرایش درونی پوسته از این نقطه، دارای ضعف است).

ساعت درونی، به طور معمول روی 60 ثانیه تنظیم شده، اما اگر پارامتری در خط دستور باشد، مانند:

```
cat checkmail 30 $
```

این زمان به جای قبلی استفاده می شود. متغیر پوسته `t`، در صورتی زمانی اعمال شود، مقدار آنرا می گیرد و اگر مقداری داده نشود، همان 60 را از خط :

```
{ t = $ { 1 - 60
```

می گیرد. این موضوع، ویژگی دیگری از پوسته را معرفی می کند.

`{var}` \$ معادل است با `$ var` و می تواند جهت صرف نظر کردن از مسائل ناشی از متغیرهای درون رشته های شامل حروف یا اعداد، به کار رود:

```

$ var = hello
$ varx=goodbye
$ echo $ var
hello
$ echo $ varx
goodbye
$ echo $ {var }x
hellox
$

```

کاراکترهای خاص درون براکتها، پردازش ویژه متغیرها را مشخص می کند. اگر متغیر تعریف نشده باشد و در پی نام آن علامت سؤال آمده باشد، رشته بعد از ؟ چاپ می شود و پوسته خارج می شود (مگر اینکه تباد لی باشد). اگر پیغام خاصی مهیا نشده باشد، استاندارد آن بصورت زیر چاپ می شود:

```

$ echo $ {var ?}
hello
O.K.;var is set
$ echo $ {junk?}
junk : parameter not set
پیغام پیش فرض
$ echo $ {junk?error! }
junk : error !
پیغام اعمال شده
$

```

توجه داشته باشید که پیغامی که توسط پوسته ساخته می شود، همواره شامل نام متغیر تعریف نشده است.

یک فرم دیگر، `{Var-thing}` می باشد که آنرا با `$ Var` ارزیابی می کند، اگر تعریف شده باشد و با `thing`، اگر تعریف شده نباشد. `{Var = thing}` نیز مشابه است، اما `thing` را نیز با `$ Var` مقدار دهی می کند:

```
$ echo $ {junk- 'Hi there ' }
Hi echo $ {junk?}
junk : parameter not set          junk تأثیر نپذیرفته است
$ echo $ {junk= 'Hi  there ' }
Hi there
$ echo $ {junk? }
Hi there          junk , Hi مقدار دهی شده
$
```

قوانین ارزیابی متغیرها، در جدول ۳-۵ آمده است.

بر می گردیم به مثال ساده خودمان، `{ 1-60 }` `t = $`

`t` را از روی `$1` ، یا اگر هیچ آگومان اعمال نشده باشد ، `60` مقدار دهی می کند.

	جدول ۳-۵: ارزیابی متغیرهای پوسته
<code>\$ Var</code>	مقدار <code>Var</code> ؛ اگر <code>Var</code> تعریف نشده باشد، هیچ مقداری
<code>{ Var }</code>	مشابه است؛ اگر بعد از متغیر نام ، حروف و ارقام آورده شود.
<code>{ Var = thing }</code>	اگر تعریف شده باشد، مقدار <code>Var</code> و در غیر این صورت ، <code>\$ Var</code> <code>thing</code> تغییر نمی یابد.
<code>{var?message}</code>	اگر تعریف شده باشد، مقدار <code>Var</code> و در غیر این صورت ، <code>\$ Var</code> <code>thing</code> مقدار <code>thing</code> نمی یابد.
<code>{Var+thing}\$</code>	اگر تعریف شده باشد، <code>\$ Var</code> . در غیر این صورت، <code>message</code> را چاپ کرده و از پوسته خارج می شود. اگر <code>message</code> خالی باشد، چاپ می کند: <code>Var: parameter not set</code>
	اگر <code>\$ Var</code> تعریف شده باشد، <code>thing</code> و در غیر این صورت هیچ چیز

تمرین ۹-۵. به کاربرد `true` و `False` در `bin /` یا `user /bin /` توجه کنید. چگونه می توانید متوجه شوید چگونه کار می کنند؟

تمرین ۱۰-۵. `Water for` را بگونه ای تغییر دهید که برهانهای گوناگونی به عنوان افراد تلقی گردند، به جای اینکه نیاز باشد کاربر `Joe` `mary` را تایپ کند.

تمرین ۱۱-۵. نسخه ای از `watchwho` بنویسید که از `Comm` به جای `awk` برای مقایسه داده های جدید و قدیمی استفاده کند.

تمرین ۱۲-۵. نسخه ای `Watchwho` بنویسید که خروجی `who` را به جای فایلها، در متغیرهای پوسته ذخیره کند. کدام نسخه را ترجیح

می‌دهید؟ کدامیک سریعتر اجرا می‌شود؟ آیا Watchwho و Checkmail ، باید & را به صورت خودکار انجام دهند؟ تمرین ۱۳-۵. چه تفاوتی میان دستور do-nothing و کارکتر # در پوسته وجود دارد؟ آیا هر دو مورد نیاز هستند؟

۴-۵-22126. Trap ها، وقفه‌های جاذب

اگر DEL را فشار دهید یا هنگام اجرای watchwho، گوشی تلفن را بگذارید، یک یا دو فایل موقت، در tmp / باقی می‌ماند. Watchwho باید آنها را قبل از خروج حذف کند و باید راهی برای رویایی چنین حالاتی و بازیافت آنها، بیابیم. هنگامی که DEL را تایپ می‌کنید، یک سیگنال وقفه به تمام پردازشهای در حال اجرا در ترمینال، ارسال می‌شود. مشابهاً، هنگامی که گوشی را می‌گذارید یک سیگنال hang up فرستاده می‌شود. سیگنالهای دیگری نیز وجود دارند. اگر برنامه‌ای، عملی واضح را در مورد سیگنالها به کار نیندد، سیگنال آنها به اتمام می‌رساند. پوسته برنامه‌هایی که با & اجرا می‌شوند را از وقفه‌ها محافظت می‌کند اما از تعلیق نه.

فصل هفتم سیگنالها را مفصل به بحث می‌گذارد، اما نیازی به شناختن زیاد آنها برای به کار بستنشان در پوسته نیست.

فرمان درونی پوسته Trap ، رشته‌ای از دستورها را هنگامی که سیگنال واقع می‌شود، جهت اجرا می‌چیند:

فهرستی از شماره‌های سیگنال رشته‌ای از دستورات trap

رشته‌ای از دستورات، آرگومان‌ی منفرد است، از این رو باید تقریباً همیشه داخل کوتیشن قرار گیرد. شماره‌های سیگنال، ارقام کوچکی هستند که سیگنال را معرفی می‌کنند. برای مثال، ۲ سیگنالی است که بواسطه فشردن کلید DEL بوجود می‌آید. و ۱ بواسطه گذاشتن گوشی، بیشتر شماره‌های سیگنال مفید برای برنامه‌نویسان پوسته در جدول ۴-۵ آمده‌اند.

جدول ۴-۵ شماره‌های سیگنال پوسته

0	خروج از پوسته (به هر دلیلی، از جمله پایان فایل)
1	
2	
3	معلق شدن
9	وقفه (کلید DEL)
19	
	خروج (Ctl-1 باعث می‌شود برنامه انباری موقتی در هسته ایجاد کند)
	ازبین بردن (قابل صرف نظر پا گرفتن نیست)
	اتمام ، پیغام پیش فرضی که توسط (1) Kill ایجاد می‌شود.

از این رو جهت پاک کردن فایل‌های موقتی در watchwho، یک فراخوانی trap باید دقیقاً پیش از حلقه انجام گیرد، تا تعلیق، وقفه و اتمام را به دست گرفت:

```
...
trap 'rm -f $new $old ; exit 1' 1 2 15
while :
```

رشته دستوری که اولین آرگومان را برای Trap شکل می دهد، مانند فراخوانی یک زیر روال است که بلافاصله هنگامی که سیگنال واقع می شود، رخ می دهد. هنگامی که به اتمام رسید، برنامه ای که در حال اجرا بوده، همانجایی که بوده را از سر می گیرد، مگر این که سیگنال آنرا از بین ببرد. بنابراین، رشته دستوری trap، باید صراحتاً exit را احضار کند، یا برنامه پوسته اجرا را پس از وقفه، ادامه خواهد داد. همچنین رشته دستوری، دوباره خوانده خواهد شد: یکبار هنگام مقدار دهی trap و یکبار هنگام احضار آن. از این رو رشته دستوری به بهترین نحو با کوتیشن محافظت شده، متغیرها فقط هنگام اجرای روال های Trap ارزیابی می شوند که در این حالت هیچ تفاوتی نمی کند اما به یک مورد بر خواهیم خورد که تفاوت خواهد داشت. به هر حال، گزینه f - به rm می گوید که سؤالی نپرسد. Trap گاهی از نظر تبادلی مفید می باشد، اغلب جهت جلوگیری از بین رفتن یک برنامه بوسیله سیگنال تعلیق ایجاد شده توسط یک تلفن شکسته:

```
$ ( trap ' ' 1; long-running- command ) &
2134
$
```

رشته دستوری خنثی به معنای « صرف نظر از وقفه ها » در این فرآیند می باشد. پراترها باعث اجرای همزمان trap و دستور در یک زیر پوسته زمینه می شود؛ بدون آنها trap برای پوسته login، همانند long-ranning-comman به کار بسته خواهد شد. فرمان (1) nohup برنامه ای کوتاه از پوسته است جهت جلوگیری از این عمل. در اینجا نسخه ویرایش هفتم آمده شده است:

```
$ cat `which nohup`
trap " " 1 15
if test -t 2 >&1
then
    echo " sending output to ' nohup . out ' "
    exec nice -5 $ * >>nohup.out 2>&1
else
    exec nice -5 $ * 2>&1
fi
$
```

Test-t امتحان می کند که خروجی استاندارد، یک ترمینال است، تا آنرا ذخیره کند. برنامه زمینه، بوسیله nice اجرا می شود تا اولویتی پایینتر از برنامه های تبادلی به آن دهد. (توجه داشته باشید که nohup به PATH مقدار دهی نمی کند. آیا باید بدهد؟) exec فقط برای کارایی است؛ دستور به تنهایی و بدون آن نیز به خوبی اجرا می شود. exec دستوری درونی از پوسته است که فرآیند در حال اجرا در این پوسته با برنامه ای که نام آن آورده می شود، عوض می کند، بدین وسیله یک فرآیند را ذخیره می نماید - پوسته ای که در حالت عادی منتظر می ماند تا برنامه تکمیل شود. می توانستیم از exec در جاهای مختلفی استفاده کنیم، مانند انتهای برنامه cal ارتقاء یافته، هنگامی که /usr/bin/bin را احضار می کند.

سیگنال ۹، سیگنالی است که نمی توان آن را صرف نظر کرد یا بدست گرفت: همواره از بین می رود، از پوسته، بدین گونه ارسال

می شود:

... عنوان فرآیند \$ Kill - 9

kill -9 is not the default because a process killed that way is given no chance to put its affairs in order before dying.

پیش فرض نیست Kill-a

تمرین ۵-۱۴ . نسخه بالایی nohup ، خطای استاندارد دستور را با خروجی استاندارد ترکیب می کند. آیا این طرح، مناسب است؟ اگر نه، چگونه آن را به خوبی از هم جدا می کنید؟

تمرین ۵-۱۵ . دستور درون پوسته times را پیدا کنید و به profile خود خطی اضافه کنید به گونه ای که هر گاه از پوسته خارج شوید، مدت زمان استفاده شما از CPU چاپ شود.

تمرین ۵-۱۶ . برنامه ای بنویسید که مشخصات کاربر قابل دسترس بعدی را در etc/passwd پیدا کند. اگر مشتاق هستید (و اجازه دارید)، آنرا به دستوری اعمال کنید که کاربری جدید را به سیستم اضافه کند. چه مجوزهایی نیاز دارد؟ چگونه باید با وقفه ها برخورد کند؟

۵-۵. جایگزین کردن یک فایل Over write

دستور command ، دارای یک گزینه 0 - است جهت بازنویسی يك فایل

```
sort file1 - 0 file2 $
```

معادل است با

```
sort file > file2 $
```

اگر file1 و file2 معادل باشند، دوباره جهت دهی بوسیله > فایل ورودی را قبل از مرتب کردن، ناقص می کند. اما گزینه 0 - ، درست کار می کند، چرا که ورودی در یک فایل موقت، مرتب و ذخیره شده است، پیش از اینکه فایل خروجی بوجود آید.

بسیاری دستورات دیگر قادر به استفاده از گزینه 0 - می باشند. برای مثال، sed می تواند فایلی را در جای خود ویرایش کند:

```
$ sed 'S/UNIX/UNIX(TM)/g' Chv-O Chv عمل نمی کند !
```

تعریف چنین دستورهایی جهت افزودن گزینه، غیر عملی خواهد بود. علاوه بر اینکه، طرحی بد خواهد بود: بهتر است که توابع را هم مرکز کنیم، همانگونه که پوسته با عملگر > انجام می دهد. برای این کار، Overwrite را برای برنامه به کار می بریم. اولین طرح مشابه زیر است:

```
sed 's / UNIX/UNIX(TM) / g' ch2 | overwrite ch2 $
```

این اشاره ابتدایی، قابل فهم است - فقط ورودی را تا انتهای فایل جایی ذخیره کنید و سپس داده ها را روی فایل آرگومان ذخیره کنید:

```
# overwrite : copy standard input to output after EOF
```

```
# version 1. BUG here
```

```
PATH= /bin:/usr/bin
```

```
case $# in
```

```
1) ;;
```

```
• ) echo 'Usage : overwrite file' 1>&2; exit 2
```

```
esac
```

```
new= /tmp/overwr . $$
```

```
trap 'rm -f $new ; exit 1' 1 2 15
```

```
cat > $new          # collect the input
cp $new $1          # overwrite the input file
rm -f $new
```

Cp، به جای mv استفاده می‌شود تا مجوزها و مالک‌های فایل خروجی، در صورت وجود آن، تغییر نکنند. همانگونه که به سادگی از این نسخه بر می‌آید، دارای یک نقص مضر است: اگر کاربر DEL را در حین CP تایپ کند، فایل ورودی اصلی، از بین خواهد رفت. باید از وقوع وقفه حاصل از متوقف ساختن بازنویسی فایل ورودی، جلوگیری کنیم:

```
# overwrite : copy standard input to output sfdter EOF
# version 2.  BUG here too
PATH=/bin : /usr/bin
case $# in
1)      ;;
. )     echo `Usage : overwrite file ' 1>&2; exit 2
esac
new=/tmp/overwr 1.$$
old=/tmp/overwr2.$$
trap 'rm -f $new $old; exit 1' 1 2 15
cat >$new          #collect the input
cp $1 $old         # save original file
trap ' ' 1 2 15   # we are committed; ignore signals
cp $new $1         # overwrite the input file
rm -f $new $old
```

اگر پیش از دستیابی به فایل اصلی، یک DEL رخ می‌دهد، فایل‌های موقتی حذف شده و فایل تنها رها می‌شود. پس از ساختن پشتیبان، سیگنالها صرف نظر می‌شوند بطوریکه CP با وقفه، متوقف نمی‌گردد - هنگامی که CP شروع شود، Overwrite مجبور به تغییر فایل اصلی می‌شود.

هنوز یک مسئله ریز وجود دارد. توجه کنید:

```
$ sed 's/UNIX/UNIX(TM)g' precious | overwrite precious
command garbled : s/UNIX/UNIX(TM) g
$ ls -l precious
-rw-rw-rw- 1you          0 oct 1 09:02 precious      # $%@*!
$
```

اگر برنامه‌ای که ورودی را برای Overwrite مهیا می‌کند، دچار اشتباه شود، خروجی آن خالی خواهد بود و Overwrite بنا به وظیفه و به لحاظ اطمینان، فایل آرگومان را از بین می‌برد.

چند راه حل ممکن به نظر می‌رسد. Overwrite پیش از جابجایی فایل، منتظر تأیید می‌تواند باشد، اما تبدالی کردن آن، شایستگی آن را می‌کاهد. Overwrite می‌تواند بررسی کند که ورودی فایل خالی نباشد (بوسیله Test-z) اما این کار نیز صحیح نیست: چند خروجی ممکن است ایجاد شود پیش از آنکه خطا ردیابی شود.

بهترین راه حل، اجرای برنامه تولید داده، تحت کنترل Overwrite است تا حالت خروج آن قابل بررسی باشد، که خلاف روال مرسوم

است، `Overwrite` هیچ چیزی در خروجی خود، ایجاد نمی‌کند، با این حال، هیچ کلیاتی از بین نمی‌رود و متن آن، بی سابقه نیست: `time`، `nice` و `nohup` همگی دستوراتی هستند که دستور دیگری را به عنوان آرگومان می‌پذیرند. نسخه صحیح در اینجا آمده است:

```
# Overwrite : copy standard input to output after EOF
# final version
opath=$ PATH
PATH=/bin:/usr/bin
case $# in
0 | 1 ) echo `Usage: overwrite file cmd [args]' 1>&2; exit 2
esac
file=$1; shift
new=/tmp/overwr 1 . $$; old=/tmp/overwr2.$$
trap 'rm -f $new $old; $old; exit 1' 1 2 15 # clean up files
if PATH=$opath "$@" >$new # collect input
then
    cp $file $old # save original file
    trap ' ' 1 2 15 # we are committed; ignore signals
    cp $ new $file
else
    echo "overwite : $1 failed , $ file unchanged" 1>&2
    exit 1
fi
rm -f $new $old
```

فرمان درون پوسته‌ای `Shift`، کل فهرست آرگومان را یکی به چپ منتقل می‌کند: `$2` می‌شود `$1`، `$3` می‌شود `$2` و غیره. `@ $` "تمامی آرگومانها را (پس از انتقال)، مانند `$*`، تأمین می‌کند، اما وقفه نمی‌پذیرد. در بخش 5-7 دوباره به آن خواهیم پرداخت. توجه داشته باشید که `PATH` برای اجرای دستورات کاربر بازخوانی می‌شود؛ اگر چنین نبود، دستوراتی که در `bin/` یا `usr/bin/` نبودند، برای `Overwrite` غیر قابل دسترس می‌بودند.

`Overwrite` now works (if somewhat clumsily):

اکنون عمل می‌کند

```
$ cat notice
UNIX is a Trademark of Bell laboratories
$ overwrite notice sed 's/UNIXUNIX(TM)/g' notice
command garbled: s/ UNIXUNIX(TM)/g
overwrite : sed failed, notice unchanged
$ cat notice
UNIX is a Trademark of Bell Laboratories          Unchanged
$ overwrit notice sed 's /UNIX/UNIX(TM)/g' notice
$ cat notice
UNIX (TM) is a Trademark of Bell Laboratorise
$
```

استفاده از Sed برای جابجایی تمام رخدادهای یک کلمه، با یکی دیگر، عملکردی معمولی است. با داشتن Overwrite، خودکار کردن این وظیفه آسان است:

```
$ cat replace
# replace:  replace str1 in files with str2, in place
PATH = /bin:/usr/bin
case $# in
0 | 1 | 2 ) echo ' Usage: replace str1 str2 files ' 1>&2; exit 1
esac
left = " $1 " ; right = "$2"; shift; shift
for i
do
    overwrite $i sed "s@$ Left@$right@g" $i
done
$ cat footnote
UNIX is not an acronym
$ replace UNIX Unix footnote
$ cat footnote
Unix is not an acronym
$
```

(به خاطر بیاورید که اگر فهرست در یک عبارت for خالی باشد، پیش فرض آن *\$ می شود.) ما از @ به جای / برای محدود کردن دستور جابجایی استفاده کردیم، چرا که @ کمتر با یک رشته ورودی اشتباه گرفته می شود. 22126

PATH، replace را از /bin:/usr/bin مقدار دهی می کند.

این بدان معنی است که Overwrite باید برای عمل کردن replace، در /usr/bin باشد. این فرض را برای ساده شدن کار در نظر گرفتیم. اگر نمی توانید overwrite را در /usr/bin نصب کنید، مجبور به گذاشتن \$HOME/bin در PATH درون replace هستید یا نام مسیر overwrite را صراحتاً وارد کنید. از این به بعد فرض می کنیم.

تمرین ۱۷-۵. چرا overwrite، از کد سیگنال O در trap استفاده نمی کند که فایلها هنگام خروج، حذف شوند؟ راهنمایی: هنگام اجرای برنامه زیر، DEL را تایپ کنید:

```
trap " echo exiting; exit 1" 0 2
Sleep 10
```

۱۸-۵. یک گزینه -V را به replace اضافه کنید تا تمامی خطوط تغییر یافته در /dev/tty را چاپ کند. راهنمایی: S/\$/sedt/\$right/gsublog

۱۹-۵. Replace را به گونه ای تنظیم کنید که بدون توجه به کاراکترها در جابجایی رشته ها عمل کند.

۲۰-۵. آیا replace را می توان برای عوض کردن متغیر i با index در هر کجای برنامه استفاده کرد؟ برای انجام چنین موردی، چه چیزهایی را می توانید تغییر دهید؟

۲۱-۵. آیا replace به مقدار کافی توانمند می باشد که به /usr/bin تعلق داشته باشد. آیا وارد کردن دستورات sed، هنگام نیاز، ساده

می‌باشد؟

۵-۲۲. (مشکل) \$ overwrite file 'who | sort'

عمل نمی‌کند. توضیح دهید چرا و آنها را تصحیح کنید. راهنمایی: eval(1) را در (1) sh ببینید. راه حل شما، چگونه بر تفسیر متاکاراکترها در دستور تأثیر می‌گذارد.

۵.۶. Zap : از بین بردن فرآیند ، از روی نام

دستور kil فقط فرآیندهایی را به پایان می‌رساند که بوسیله شناسه فرآیند، معرفی شده باشند. هنگامی که یک فرآیند زمینه خاص، لازم به از بین رفتن است، معمولاً باید PS را برای یافتن شناسه فرآیند اجرا کرد و سپس به دشواری آنرا مانند یک آرگومان دوباره وارد کرد تا از بین روند. اما وارد کردن برنامه برای چاپ عددی که فوراً آنرا بطور دستی رونویسی می‌کنید، چندان عاقلانه نیست. چرا برنامه‌ای ننویسیم، مثلاً با say که این کار را خودکار انجام دهد؟

یک دلیل این است که فرآیندهای از بین برنده، خطرناک می‌باشند و باید مراقب بود که فرآیند صحیح از بین برود. یک روش سالم و مناسب، اجرا کردن ZAP است بصورت تبادلی، و استفاده از Pick برای انتخاب قربانی‌ها.

یک یادآوری سریع درباره Pick: هر کدام از آرگومانهای خود را به نوبت چاپ کرده و از کاربر پاسخ می‌خواهد؛ اگر پاسخ y باشد، آرگومان چاپ می‌گردد. (Pick موضوع بخش بعدی است)

ZAP از Pick استفاده می‌کنند تا صحت فرآیندهایی را که از روی نامشان توسط کاربر، برای از بین رفتن انتخاب شده‌اند را بررسی کند.

```
$ cat zap
# zap pattern: kill all ptoesses matching pattern
# BUG in this vetsion
PATH=/bin : /usr/bin
cas $# in
0)      ecvgo 'Usage: zap pattern' 1>&2; exit 1
esac
kill `pick \ `ps - sg | frep " $* "\ ` | awk ' {print $!} ' `
$
```

1. به علامتهای ` که توسط backslash ها احاطه شده اند، توجه کنید. برنامه awk، شناسه فرآیند را از خروجی PS که بوسیله Pick انتخاب شده، بر می‌گزیند:

```
4 sleep 1000 &
22126
$ ps - ag
PID TTY TIME CMD
...
22126 0 0:00 slepp 1000
...
$ zap sleep
22126?
```


0 ? q چه اتفاقی می افتد؟

مسئله این است که خروجی PS به کلماتی بخش بخش شده که بوسیله Pick به عنوان آرگومانهای مجزا دیده می شوند، به جای اینکه کل خط یک باره پردازش شود.

رفتار عادی پوسته، شکستن رشته‌ها به آرگومانهایی است فاصله دار و بدون فاصله، مانند:

```
For i in 1 2 3 4 5
```

در این برنامه باید تقسیمات پوسته را بر روی رشته‌ها، به آرگومانها، کنترل کنیم، تا فقط خطوط جدید، کلمات کنار هم را از هم جدا کنند.

متغیر پوسته IFS (جدا کننده میدان داخلی)، رشته‌ایست از کاراکترها که کلمات را در فهرست آرگومانها، از هم جدا می کند، مانند ` و جملات for. به طور معمول، IFS، شامل یک جای خالی، یک پرش و یک خط جدید می باشد اما می توانیم آنرا به هر چیز مفیدی تبدیل کنیم، مانند یک خط جدید:

```
$ echo 'echo $#' >nargs
```

```
$ cx naegs
```

```
$ who
```

```
you      tty0      Oct  1 05:59
```

```
pjw      tty2      oct   1 11:26
```

```
$ nargs `who`
```

```
10
```

میدانهای مجزای ده خط و یک خط جدید

```
$ IFS = '
```

```
 فقط یک خط جدید
```

```
$ nargs `who`
```

```
2
```

دو خط جدید. دو میدان

```
$
```

به خوبی کار می کند zap، توسط خط جدید IFS با مقدار دهی

```
$ cat zap
```

```
# zap pat: kill all procses matching pat
```

```
# final version
```

```
22126
```

```
PATH=/bin :/usr/bin
```

```
IFS='
```

```
' # just a newline
```

```
case $1 in
```

```
  ") echo 'Usage: zap [-2] pattern' 1>&2; exit 1 ;;
```

```
  - * ) SIG=$1 ; shift
```

```
  esac
```

```
  esac ' PID TTY TIME CMD'
```

```
  kill $$SIG `pick` ps -ag | egrep "$ * "` | awk '{print $1}' `
```

```
$ ps -ag
```

```
 PID TTY TIME CMD
```

```
...
22126 0 0:00 sleep 1000
```

```

...
$ zap sleep
  PID TTY TIME CMD
 22126 0   0:00 sleep 1000? y
 23104 0   0:02  egrep sleep? n
$

```

ما دو چین خوردگی را وارد کردیم: يك آرگومان اختیاری برای مشخص کردن سیگنال (توجه کنید که SIG معرفی نشده میماند و از این رو، به عنوان يك رشته بی تأثیر تلقی خواهد شد، اگر آرگومان به کار گرفته نشود) و استفاده از egrep به جای grep ، برای مجاز کردن نمونه‌های پیچیده‌تر مانند ' Sleep | date ' . يك echo ابتدایی، سرصفحه‌های ستونی را برای خروجی Ps به چاپ می‌رساند.

حتماً تعجب خواهید کرد چرا این فرمان به جای فقط kill ، zap خوانده می‌شود. دلیل اصلی این است که برخلاف مثال ما در مورد cal ، در حقیقت فرمان Kill جدیدی ایجاد نمی‌کنیم: ZAP الزاماً تبادلی است، به یک دلیل - و ما می‌خواهیم Kill را به همین دلیل نگهداریم. Zap همچنین به طرز آزار دهنده‌ای کند است - هزینه تمام برنامه‌های اضافی، قابل قبول است، با وجود آنکه Ps (که باید به هر صورتی اجرا شود) گرانترین است.

در فصل بعد، کارکردی مؤثر تر از ارائه خواهیم داد.

تمرین ۲۳-۵ . ZAP را بگونه‌ای تعریف کنید که سرصفحه PS را از pipeline چاپ کند بطوریکه به تغییرات در ساختار خروجی Ps ، حساس نباشد. این تغییر ، تا چه حد برنامه را پیچیده می‌کند؟

۵-۷ - دستور Pick : جاهای خالی آرگومانها

ما تقریباً با هر آنچه برای نوشتن دستور pick در پوسته نیاز داریم، بر خورد داشته‌ایم. تنها چیز جدیدی که نیاز است، مکانیزمی است برای خواندن ورودی کاربر. دستور داخلی پوسته read ، یک خط از متن را از ورودی استاندارد خوانده و متن را به عنوان مقدار متغیر نام برده می‌خواند (بدون خط جدید).

```

$ read greeting
hello , world
$ echo $ greeting
hello , world
$

```

معمول‌ترین استفاده read در profile است جهت تنظیم محیط، هنگام ورود به سیستم تا متغیرهای پوسته مقدار دهی اولیه شوند، مانند TERM.

Read فقط می‌تواند از ورودی استاندارد بخواند؛ حتی قابل جهت دهی مجدد نیز نیست. هیچ یک از دستورات درون پوسته (برخلاف اصول اولیه کنترل جریان مانند for) قابل جهت دهی مجدد با > یا < نیستند.

```

$ read greeting </etc/passwd

```

```

goodbye          باید مقداری وارد شود
illegal io       اکنون پوسته گزارش خطا می دهد
$ echo $greeting
goodbye         Greeting را وارد کرده ، نه از فایل
$

```

این امر را می توان به عنوان نقصی در پوسته قلمداد کرد، اما واقعیت دارد. خوشبختانه می توان با دوباره جهت دادن حلقه محاط بر for این موضوع را نیز حل کرد. این کلید مشکل ما در به کار بردن دستور pick است:

```

# pick: select arguments
PATH= /bin:/usr/bin
for i             # for each argument
do
    echo -n "$i? " >/dev/tty
    read response
    case $response in
    y * )         echo $i ;;
    q * )         break
    esac
done </dev/tty

```

echo-n آخرین خط جدید را متوقف می کند، تا پاسخ در آن، به صورت یک پیغام فوری، قابل تایپ شدن باشد. و البته پیغامهای فوری در /dev/tty/ چاپ می شوند، چرا که خروجی استاندارد اغلب مطمئناً ترمینال نیست.

جمله break از : C قرض گرفته شده است: حلقه درونی احاطه شده را پایان می دهد. در این حالت، حلقه For را هنگامی که q تایپ شود، قطع می کند. اجازه می دهیم q انتخاب را به پایان برساند چرا که انجام آن آسان است، ذاتاً بی زحمت و با سایر برنامه ها نیز سازگار است.

برای اجرای Pick ، استفاده از جاهای خالی در آرگومانها، جالب به نظر می رسد:

```

$ pick '1 2' 3
1 2?
3?

```

اگر مایلید بدانید Pick چگونه آرگومانهای خود را انتخاب می کند، آنرا اجرا کرده و فقط پس از هر پیغام فوری، Return را فشار دهید. همانگونه که انتظار می رود، درست عمل می کند: For i آرگومانها را درست جابجا می کند. می توانستیم حلقه را به روشهای دیگری بنویسیم:

```

$ grep for pick          see what this version does
for i in $ *
$ pick '1 2' 3
1 ?
2 ?
3 ?

```

\$

این فرم جواب نمی‌دهد، زیرا عملگرهای حلقه دوباره جاروب می‌شوند و جاهای خالی در اولین آرگومان، باعث می‌شود به دو آرگومان تبدیل شوند. این بار با گذاشتن " برای * \$ امتحان کنید.

\$ grep for pick

نسخه ای دیگر را امتحان کنید

for i in "\$*"

\$ pick ' 1 2 ' 3

1 2 3 ?

\$

این یکی نیز کار نمی‌کند، چرا که " * \$ " کلمه‌ایست مفرد و متشکل از تمامی آرگومانهایی که به یکدیگر پیوسته اند و با جای خالی از هم جدا شده‌اند.

البته راه حلی وجود دارد، اما تقریباً شعبده بازی است: رشته " @ \$ "، توسط پوسته، به طور ویژه‌ای مورد توجه قرار می‌گیرد و دقیقاً به آرگومانهای فایل پوسته تبدیل می‌شود:

\$ grep for pick

نسخه سوم را امتحان کنید

for i in "\$@"

\$ pick ' 1 2 ' 3

1 2 ?

3 ?

\$

اگر @ \$ داخل کوتیشن قرار نگرفته باشد، همانند * \$ تلقی می‌گردد؛ رفتار فقط هنگامی مخصوص خواهد بود که داخل " قرار گیرد، از آن در Overwrite برای نگه‌داشتن آرگومانها برای دستور کاربر استفاده کردیم. به طور خلاصه، قوانینی در اینجا ذکر می‌گردند:

* \$ و @ \$، به آرگومانها توسعه داده و دوباره جاروب می‌شوند؛ جاهای خالی در آرگومانها، آرگومانهای متعددی را نتیجه خواهد داد.

* "\$*" کلمه‌ایست مفرد، مرکب از تمامی آرگومانهای موجود در پوسته که با فاصله‌هایی به یکدیگر متصل شده‌اند.

* "\$@" مشابه آرگومانهایی است که بوسیله فایل پوسته دریافت می‌گردند: جای خالی در آرگومانها نادیده گرفته می‌شوند و نتیجه، فهرستی است از کلماتی که یکسان با آرگومانهای اصلی هستند.

اگر Pick آرگومانی نداشته باشد، باید ورودی استاندارد خود را بخواند، از این رو می‌توان از:

\$ Pick < mailing list >

به جای

\$ Pick `Cat mailing list`

استفاده کرد. اما ما این نسخه از Pick را مورد بررسی قرار نمی‌دهیم: با پیچیدگی‌های ناهنجاری همراه است و بسیار مشکل‌تر از برنامه‌های مشابهی است که با C نوشته می‌شود، که در فصل بعد آنرا بحث خواهیم کرد.

دو تمرین اول پیش رو مشکل می‌باشند، اما برای برنامه‌نویسان پیشرفته پوسته، آموزنده‌اند.

ما دو چین خوردگی را وارد کردیم: یک آرگومان اختیاری برای مشخص کردن سیگنال (توجه کنید که SIG معرفی نشده می ماند و از این رو، به عنوان یک رشته بی تأثیر تلقی خواهد شد، اگر آرگومان به کار گرفته نشود) و استفاه از-egrep به جای-grep، برای مجاز کردن نمونه های پیچیده تر مانند 'sleep | date'. یک echo ابتدایی، سرصفحه های ستونی را برای خروجی ps به چاپ می رساند.

تمرین ۵-۲۴. سعی کنید با pick برنامه ای بنویسید که آرگومانهایش را از ورودی استاندارد بخواند، اگر هیچ کدام در خط فرمان عرضه نشده باشند. باید جاهای خالی را به خوبی کنترل کند. آیا q کار می کند؟ اگر نه، تمرین بعدی را امتحان کنید.

تمرین ۵-۲۵. با وجود اینکه دستورات درونی پوسته مانند read و set قابل جهت دهی مجدد نیستند، خود پوسته، موقتاً قابل جهت دهی است. بخش مربوط به (1)sh را که enec را توصیف کرده و چگونگی خواندن از dev/tty، را بدون فراخوانی یک زیر پوسته، کامل تشریح می کند. (ممکن است خواندن فصل ۷ بتواند کمک کند.)

تمرین ۵-۲۶. (بسیار ساده تر) read را در protile. خود بخوانید و TERM و هرآنچه به آن بستگی دارد را بنویسید.

۵-۸. دستور news: پیغامهای خدمات اجتماعی

در فصل اول، اشاره کردیم که سیستم شما ممکن است دارای یک فرمان news جهت گزارش پیغامهای عمومی جامعه کاربر باشد. باوجود اینکه نام و جزئیات دستور تفاوت دارند، اکثر سیستمها، دارای یک سرویس خبری است. دلیل ما برای ارائه فرمان news جایگزین کردن دستور محلی شما نیست بلکه برای اینست که نشان دهیم به چه سادگی چنین برنامه ای را در پوسته، می توان نوشت. مقایسه دستور news ما با ویرایش محلی شما، می تواند جالب باشد.

ایده ابتدایی چنین برنامه ای، این است که مقالات خبری، هرکدام در یک فایل، در یک دایرکتوری ویژه مانند /usr/news/ ذخیره می شوند. news (برنامه news) از مقایسه زمانهای تعریف فالها در /usr/news/ با همان نمونه ها در دایرکتوری شما (news-time). عمل می کند.

می توانیم از « . » به عنوان دایرکتوری برای هم فایلهای خبری و هم news-time. استفاده کنیم؛ که هنگام آماده بودن برای اشکال زدایی، برنامه جهت استفاده عمومی، قابل تغییر به /usr/news/ می باشد.

```
$ cat news
# news: print news files, vrsion 1
HOME=. # debugging only
cd . # place holder for /usr/news
for i in `ls -t * $HOME/. news - time`
do
    case $i in
        . / .news - time) break ;;
        . ) echo news: $i
        . 22126
    esac
done
touch $HOME/. news - time
$ touch x
$ touch y
```

```
$ news
news : y
news : x
$
```

touch زمان آخرین تغییر فایل آرگومان خود را به زمان حال تغییر می دهد، بدون تغییری در فایل، برای عیب یابی، فقط اسامی فایل های خبری را بازگشت می دهیم، به جای اینکه آنها را چاپ کنیم.

حلقه وقتی پایان می یابد که news-time . را بیابد، از آن به بعد، تنها آنهایی را فهرست می کند که جدیدتر باشند. دقت کنید که * در جملات case، می تواند با یک / مطابقت داشته باشد.

چه اتفاقی می افتد اگر newd-time . وجود نداشته باشد؟

```
$ rm .news - time
4 news
$
```

این سکوت، غیر منتظره است و اشتباه، اما رخ می دهد چرا که اگر LS نتواند فایلی را بیابد، در خروجی استاندارد خود، پیش از چاپ هر اطلاعاتی درباره فایل های موجود، وجود مشکلی را گزارش می دهد. این یک عیب بوده اما ما می توانیم آنرا با تشخیص مشکل در حلقه، حل کنیم و خطای استاندارد را به خروجی استاندارد دوباره جهت دهیم. (این شکل در نسخه های جدیدتر سیستم حل شده بود.)

```
$ cat news
# news: print news files, version 2
HOME= . # debugging only
cd . # place holder for / usr/news
IFS = '
' # just a newline
for i in `ls -t * $HOME/ .news - time 2>&1`
do
    case $i in
        . 'not found ' ) ;;
        / .news - time ) break ;;
        . ) echo news : $i ;;
    esac
done
touch $HOME/ .news - time
$ rm .news - time
$ news
news : news
news: y
news: x
$
```

باید IFS را از خط جدید مقدار دهی کنیم تا پیغام

```
./ .news-time no found
```

به سه کلمه تجزیه نشود.

در قدم بعدی، news باید فایل‌های خبری را چاپ کند، به جای آنکه نام آنها را برگرداند. این کار از این بابت مفید است که می‌توان فهمید چه کسی و چه زمانی را ارسال نموده، و از این رو از دستور set و LS-1 برای چاپ یک سرصفحه قبل از خود پیغام، استفاده می‌کنیم:

```
$ ls -l news
-rwxrwxrwx 1 you 208 oct 1 12:05 news
```

```
- $ set `ls -l news`
-rwxrwxrwx: bad option(S)      اشتباهی رخ داده است!
```

```
$
```

اینجا، مثالی است از جایی که قابلیت جابجایی برنامه‌ها و داده‌ها در پوسته، مورد استفاده قرار می‌گیرد.

Set اعتراض می‌کند، چرا که آرگومان آن ("rwxrwxrwx-") باعلامت منفي شروع شده و مانند يك گزینه بیه نظر می‌آید. يك راه حل ساده (اگر باهوش باشید)، افزودن يك پیشوند است به آرگومان مانند يك کارکتر عادي:

```
$ set X `ls -l news`
$ echo "news: ($3) $5 $6 $7 "
news: (you) oct 1 12:05
$
```

این یک غالب قابل قبول است و نویسنده و تاریخ پیغام را همراه با نام فایل نشان می‌دهد.

در اینجا آخرین ویرایش دستور news آمده است:

```
# news: print news files , final version
PATH= /bin:/usr/bin
IFS= '
'          # just a newline
cd/ usr/news
for i in `ls -t * $HOME/ .news -time 2>&1`
do
    IFS= ' '
    case $i in
        . 'not found' ) ;;
        /.news -time) break ;;
        ) set X `ls -l $i`
            echo "
$i : ($3) $5 $6 $7
            cat $i
    esac
done
touch $HOME/ . news - time
```

خطوط جدید اضافی در سرصفحه، مقالات خبری را در حین چاپ از هم جدا می‌کنند. اولین مقدار IFS، فقط یک خط جدید است، از این رو پیغام not found (اگر باشد) از اولین LS، به عنوان یک آرگومان منفرد تلقی می‌گردد. دومین وظیفه IFS، آنرا دوباره جای خالی مقدار دهی می‌کند، از این رو خروجی دومین LS، به آرگومان‌هایی تجزیه می‌کند.

تمرین ۲۷-۵. یک گزینه n- را به news اضافه کنید تا مقالات خبری را گزارش دهد اما چاپ نکند و به news - time دست نیابد. ممکن است در profile. شما جای گیرد.

۲۸-۵. طرح، و کارکرد news در دستورات مشابه روی سیستم خود را با یکدیگر مقایسه کنید.

۹-۵. get و put: ردیابی تغییرات فایل

در این بخش، تا آخرین بخش یک فصل طولانی، یک مثال بزرگتر و پیچیده‌تر را به بحث می‌گذاریم که همکاری پوسته را با sed، awk، تشریح می‌کند.

یک برنامه، با درست کردن معایب و اضافه کردن ویژگیها، تکامل می‌یابد. گاهی ردیابی این ویرایشها، به ویژه هنگامی که افراد برنامه را برای سایر ماشینها به کار می‌برند، ساده می‌نماید - برمی‌گردند سؤال می‌کنند «از وقتی که ویرایش ما نصب شده، چه تغییری حاصل شده؟» یا «این عیب یا آن مشکل را چگونه برطرف کردید؟» همچنین، همواره نگهداشتن نسخه‌های پشتیبان، تجربه کردن ایده‌ها را بی‌آسیب تر می‌کند: اگر چیزی کار نمی‌کند، بازگشتن به برنامه اصلی، بدون دردسر خواهد بود. یک راه حل، نگه داشتن کپی‌های تمام ویرایشهاست، اما این کار با زحمت و هزینه زیادی همراه است. در عوض، بر روی نسخه‌هایی متمرکز می‌شویم که دارای بسیاری قسمتهای مشترک بوده و یکبار نیاز به ذخیره شدن دارند.

دستور new - e diff \$

فهرستی از دستورات ed را که old را به new تبدیل می‌کنند، ایجاد می‌کند، از این رو نگهداشتن تمامی نسخه‌های یک فایل در یک فایل جداگانه (متفاوت) با نگهداری یک نسخه کامل و مجموعه‌ای از دستورات ویرایشی برای تبدیل آن به هر نسخه، میسر می‌شود.

دو نوع سازماندهی وجود دارد: جدیدترین ویرایش را در دسترس نگه داشته و دستورات ویرایشی را عقب برد و یا قدیمی‌ترین ویرایش‌ها را نگه داشت و دستورات ویرایشی را جلو برد. با وجود اینکه دومی برای برنامه نویسی ساده‌تر است، اولی سریعتر است اگر ویرایشهای زیادی در دسترس باشد. ما سازماندهی اول را انتخاب می‌کنیم. در یک فایل منفرد که آنرا فایل تاریخچه نام گذاری می‌کنیم، ویرایش جاری و در پی آن مجموعه‌ای از دستورات ویرایشی که هر ویرایشی را به ویرایش قبلی برمی‌گرداند، موجود می‌باشند.

هرمجموعه از دستورات ویرایشی، با خطی شروع می‌شوند مشابه

خلاصه تاریخ شخص @@@

خلاصه، خطی است منفرد، که بوسیله شخص تأمین شده و تغییر را توصیف می‌نماید. برای نگهداری از نسخه‌ها، دو دستور وجود دارد: get ویرایشی را از فایل تاریخچه گرفته و put نسخه‌ای جدید را به آن وارد می‌نماید، پس از اینکه یک خط خلاصه تغییرات درخواست شود.

پیش از نشان دادن کاربرد، مثالی در اینجا برای بیان چگونگی عمل get و put و فایل تاریخچه چگونه نگهداری می‌شود، آورده شده:


```

$ echo a line of text >junk
$ put junk
Summary: make a new file          توصیفات را وارد نمایید
get: no file junk . H            تاریخچه موجود نمی باشد .....
put: creating junk . H          put ..... آنرا ایجاد می کند
$ cat junk . H
a line of text
@@@ you sat oct 1 13:31:03 EDT 1983 make a new file
$ echo another line >> junk
$ put junk
summary: one line added
$ line of text
another line
@@@ you sat oct 1 13:32:28 EDT 1983 one line added
2d
@@@ you sat oct 1 13:31:03 EDT 1983 make a new file
$

```

« دستورات ویرایشی » شامل خط منفرد 2d که خط دوم فایل را از بین می‌برند، ویرایش جدید را به نسخه اصلی برمی‌گرداند.

```

$ rm junk
$ get junk                       جدیدترین ویرایش
$ cat junk
a line of text
another line
$ get -1 junk
$ cat junk                       ویرایش یکی مانده به آخر
a link of text
$ get junk                       جدیدترین ویرایش دوباره
$ reolace antother 'a different' junk
$ put junk
summary: second line changed
$ cat junk . H
a line of text
a different line
@@@ you sat oct 1 13:34:07 EDT 1983 second line changed
2c
another line
.
@@@ you sat oct 1 13:32:28 EDT 1983 one line added
2d
@@@ you sat oct 1 13:31:03 EDT 1983 make a new file
$

```

دستورات ویرایشی، از بالا تا پایین سراسر فایل تاریخچه، اجرا شده تا ویرایش دلخواه را استخراج کنند: سری اول، جدیدترین را به دومین نسخه جدید تبدیل می‌کند، بعدی آن را به سومین نسخه جدید برمی‌گرداند و ... از این رو، درواقع فایل جدید را هر بار با اجرای ed یکی، به نسخه قدیمی برمی‌گردانیم.

اگر فایلی را که با @@@ آغاز شده باشد را مورد اصلاح قرار دهیم، قطعاً به مشکل برخورد و بخش عیب‌ها از diff(1)، درباره خطوطی که شامل فقط یک دوره می‌شوند، اخطار خواهد داد. از-@@@ برای علامت گذاری دستورات ویرایشی استفاده می‌کنیم، چرا که رشته ایست ناخواسته برای متنی معمولی.

باوجود آنکه نشان دادن چگونگی تکامل دستورات get و put، می‌تواند مفید واقع شود، اما طولانی بوده و نشان دادن فرم‌های گوناگون آنها، مستلزم بحث زیادی است. از این جهت تنها فرم‌های نهایی آنها را به شما نشان می‌دهیم. Put ساده‌تر است:

```
# put: install file in to history
PATH=/bin: /usr / bin
case $# in
    1 )    HIST=$1 . H ;;
    2 )    echo 'Usage: put fill ' 1>&2; exit 1 ;;
esac
if test ! -r $1
then
    echo "put: can ' t open $1" 1>&2
    exit 1
fi
trap 'rm -f /tmp/put . [ab]$$; exit 1' 1 2 15
echo -n ' Summary: '
read Summary
if get -o /tmp/put.a$$ $1          # previous versoin
then
    # merge pieces
    cp $1 /tmp/put.b$$             # current versin
    echo "@@@ `getname ` `date` $Summary " >>/tmp/put.b$$
    diff -e $1 /tmp/put.a$$ >>/tmp/put.b$$ # Latest diffs
    sed -n '/^@@@/, $p' <$HIST >>/tmp/put.b$$ # old diffs
    overwrite $HIST cat /tmp/put.b$$ # put it back
else
    # mske a new one
    echo "put: creating $HIST"
    cp $1 $HIST
    echo "@@@ `getname ` `date` $Summary" >> $HIST
fi
rm -f /tmp/put.[ab]$$
```

بعد از خواندن خلاصه یک خطی، put ger را جهت استخراج ویرایش قبلی فایل از فایل تاریخچه، فراخوانی می‌کند. گزینه 0-، یک فایل خروجی متبادل را به get اختصاص می‌دهد. اگر get نتوانست فایل تاریخچه را بیابد، یک پیغام خطا برگردانده و put یک فایل تاریخچه جدید ایجاد می‌کند. اگر فایل تاریخچه وجود داشته باشد، عبارت then، تاریخچه جدید را در یک فایل موقت، به ترتیب از آخرین ویرایش، خط @@@ دستورات ویرایشگر، برای تبدیل از جدیدترین ویرایش قبلی، ایجاد می‌کند. سرانجام، فایل موقت روی

فایل، با استفاده از `overwrite` کپی می‌شود.

`get` پیچیده‌تر از `put` است که بیشتر به خاطر داشتن گزینه‌هاست.

```
# get: extract file from history
PATH=/bin:/usr/bin
VERSION=0
while test "$1" != " "
do
    case "$1" in
        -i) INPUT=$2; shift ;;
        -o) OUTPUT=$2; shift ;;
        -[0-9]) VERSION = $1 ;;
        - *) echo "get: Unknown srgument '$1' 1>&2; exit 1 ;;
        . ) case "$OUTPUT" in
            " ") OUTPUT = $1 ;;
            . ) INPUT = $1.H ;;
        esac
    esac
    shift
done
OUTPUT= $ {OUTPUT? "Usage: get [- o outfile] [-i file . H] file "}
INPUT = $ { INPUT - $OUTPUT.H }
test -r $ INPUT || { echo "get: no file $ INPUT" 1>&2; exit 1; }
trap 'rm -f /tmp/get.[ab] $$; exit 1' 1 2 15
# split into current version and editing commands
sed < $ INPUT -n ' 1 ,/^@@@/w /tmp/get.a'$$
    /^@@@/ ,sw /tmp/get.b'$$
# perform the edits
awk </tmp/get.b$$ '
    /^@@@/ { count ++ }
    !/^@@@/ && count > 0 && count < = '$VERSION'
    END { print "$d"; print "w", "' $OUTPUT ' " }
' | ed - /tmp/get.a$$
rm -f /tmp/get.[ab] $$
```

گزینه‌ها، کاملاً عادی و معمولی می‌باشند. `i` - و `O` - ، ورودی و خروجی را مشخص می‌کنند. `[0-9]` - ، ویرایشی خاص را انتخاب می‌نماید: `O` جدیدترین ویرایش است (پیش فرض) ، `i` - یکی مانده به جدیدترین و ... حلقه شامل آرگومانها، `while` است با یک `test` و `shift` به جای `for` می‌باشد، چرا که برخی گزینه‌ها (`O` - و `i` -) از یک آرگومان دیگر نیز استفاده کرده و باید آنرا `shift` داد و اینکه حلقه‌های `for` و `shift`ها، اگر `shift` داخل حلقه `for` باشد، با هم به درستی کار نمی‌کنند. گزینه « - » ، شمارش کارکتر را که معمولاً به همراه خواندن یا نوشتن یک فایل اجرا می‌شود را متوقف می‌کند.

خط

```
test -r $INPUT || { echo " get: no file $INPUT" 1>&2; exit 1; }
```

معادل است با

```

if test ! -r $INPUT
then
    echo "get: no file $ INPUT" 1>&2
    exit 1
fi

```

(که همان فرمی است که برای put استفاده نمودیم) اما کوتاهتر و برای برنامه نویسانی که با عملگر -| آشنا هستند، واضح تر است. دستورهای بین {,} ، در پوسته جاری اجرا می شوند، در یک زیر پوسته؛ این امر الزامی است چرا که exit از get خارج می شود و نه فقط از یک زیر پوسته. کارکترهای {,} مانند do و done می باشند - اگر بعد از آنها، یک « ; » یا خط جدید یا هر پایان دهنده دستور، دارای معنی خاصی است.

سرانجام سراغ کد واقع در get که این کار را انجام می دهد می آییم. در ابتدا، sed فایل تاریخچه را به دو قسمت تقسیم می کند: آخرین نسخه و مجموعه ویرایش ها. سپس برنامه awk فرامین ویرایشی را پردازش می کند. خطوط @@@ ، شمرده (و نه چاپ) می شوند و آنجایی که شمارش، بیشتر از ویرایش مورد نظر نباشد، دستورهای ویرایشی، عبور می کنند (به خاطر بیاورید که عملکرد پیش فرض awk، چاپ کردن خط ورودی است). دو دستور ed پس از آن از فایل تاریخچه اضافه می شوند: \$d ، خط @@@ را sed ، آنرا در ویرایش جدید، رها می کند، حذف می نماید و یک دستور -W ، فایل را در موقعیت نهایی خود، می نویسد. Overwrite در اینجا غیرالزامی می نماید، چرا که get تنها ویرایش فایل را تغییر می دهد و نه فایل ارزشمند تاریخچه را.

تمرین ۵-۲۹ . یک version دستوری بنویسید که دو کار انجام دهد:

```
version -5 file $
```

خلاصه، تاریخ تغییر و شخصی که تغییر ویرایش انتخاب شده را در فایل تاریخچه انجام دهد:

```
version sep 20 file $
```

گزارش دهد که کدام شماره ویرایش در ۲۰ سپتامبر جاری بوده، که نوعا در

```
`get `version sep 20 file $
```

مورد استفاده قرار می گیرد. (version می تواند نام فایل تاریخچه را برای سادگی کار برگرداند.)

تمرین ۵-۳۰ . get و put را طوری تعریف کنید که فایل تاریخچه را در یک دایرکتوری جداگانه، به خوبی به کار بندد. به جای اینکه دایرکتوری در حال کار را با فایل های H. شلوغ کند.

۵-۳۱ . تمامی ویرایش های یک فایل، پس از سامان گرفتن همه چیز، ارزش حفظ کردن ندارند. چگونه می توانید ترتیبی اتخاذ کنید که ویرایشهایی را از میان فایل تاریخچه حذف کنید.

۱۰-۵ نگاهی به آنچه گذشت

هنگامی که بانوشن یک برنامه مواجه می شوید، تمایلی طبیعی به شروع به فکر کردن به این موضوع وجود دارد که آنرا به زبان مورد علاقه خودتان بنویسید. در این مورد، آن زبان، پوسته است.

باوجود آنکه گاهی رسم الخطی نامأنوس دارد، پوسته زبان برنامه نویسی مناسبی می باشد. مطمئناً سطح آن بالاست؛ عملگرهای آن برنامه هایی هستند کامل. از آنجاییکه تبدلی می باشد، برنامه ها می توانند بصورت تبدلی توسعه پیدا کنند و در چند قدم و در حین کار

قابل پالایش شدن هستند. بعد از آن، اگر جهت کارهایی غیرشخصی نیز نیاز باشند، با پرداخته و محکم کاری شدن، می‌توانند جهت استفاده‌های متنوع‌تر آماده شوند. در آن حالات غیرمعمولی که پوسته باید بسیار کارآمد باشد، برخی، یا تمام برنامه‌های آن می‌توانند با C نوشته‌شوند، اما باز هم با طرح دستنویس. (این روش را در فصل بعد به بحث خواهیم گذاشت).

در این فصل بسیاری مثالها را که با برنامه‌ها و پوسته‌های موجود راحت قابل انجام هستند، آورده شده‌اند. گاهی دوباره مرتب کردن آرگومانها، کافی می‌باشد؛ در حالتی که با cal کار می‌کنیم. گاهی پوسته حلقه‌ای بر یک مجموعه از نام فایلها یا یک توالی از اجرای دستورات، ایجاد می‌کند، مانند watchfor , checkmail . مثالهای پیچیده‌تر، نسبت به C، کمتر کار می‌برند؛ برای مثال ویرایش ۲۰ خطی ما از news، جایگزین ویرایش ۳۵۰ خطی C شده است.

اما اینها برای داشتن یک زبان دستورات قابل برنامه‌ریزی، کافی نیست. آنچه اهمیت دارد، این است که تمامی اجزاء با هم کار کنند. هرکدام برای تمرکز بر روی یک کار و انجام بهینه آن طراحی شده‌اند، سپس پوسته آنها را هرگاه که ایده‌ای جدید دارید، با هم پیوند می‌دهد، به راحتی و با کارایی بالا، همین همکاری باعث می‌شود که محیط برنامه‌نویسی UNIX بسیار کارآمد باشد.

نکاتی از تاریخچه و شکل‌گیری

ایده get , put ، از سیستم کنترل که مرجع (scs) که بوسیله Marc Rochkind ایجاد شده، می‌آید. (" IEEE ، sccs ترجمه‌ای بر مهندسی نرم‌افزار ، 1975) . sccs بسیار توانمند و انعطاف پذیرتر از برنامه‌های ساده ماست؛ جهت نگهداری برنامه‌های عظیم در یک محیط تولیدی کاربرد دارد. مبنای sccs مشابه برنامه diff است.

فصل ششم: مقدمه‌ای بر زبان C

این بخش شامل مقدمه مختصر زبان C می باشد. قصد داشته به عنوان خود آموز زبان بوده، و با این هدف که خواننده ای که C را تازه شروع کرده به سرعت ممکن برسد. مطمئنا بعنوان جانشینی برای کتابهای درسی بیشتر C نیست. بهترین راه آموزش یک زبان جدید انسانی صحبت کردن آن درست از آغاز است. گوش دادن و تکرار پیچیدگی های دستور زبان را از بین می برد. همینطور برای زبان های کامپیوتری- یادگیری C- بکار می رود و ما باید نوشتن برنامه های C را با سرعت ممکن آغاز کنیم.

یک کتاب درسی عالی C نوشته شده توسط دو نویسنده شناخته شده و بسیار بزرگ:

زبان برنامه نویسی ANSI C-C

بریان W.C کرینان و دنیس M. ریتچی

فرنیس هال ۱۹۸۸

دنیس ریتچی اولین مترجم C را بر روی PDP-11 طراحی و پیاده سازی کرد (ماشین ماقبل تاریخ با استانداردهای امروزی، که تاکنون تاثیر زیادی بر محاسبات علمی مدرن داشته است). زبان C بر مبنای دو زبان (اکنون از بین رفته) بوده: BCPL (بی سی پی ال)؛ مارتین ریچاردز، و B (بی) نوشته کن تامپسون در سال ۱۹۷۰ برای اولین سیستم یونیکس بر روی PDP_7. زبان اصلی و رسمی R:C و K بود. که برگرفته از نام دو نویسنده زبان برنامه نویسی C اصلی بوده است. در سال ۱۹۸۸ موسسه استانداردهای ملی آمریکا (ANSI) نسخه جدید و اصلاح شده C را که امروزه با عنوان (ANSI C) شناخته شده را پذیرفت. این نسخه در ویرایش جاری زبان برنامه نویسی ANSI C-C توصیف شده. نسخه ANSI شامل اصلاحیه های زیاد نحوی و کارهای درونی زبان می باشد. موارد مهم اصلاح شده نحو فراخوانی برای روالها و استاندارد سازی بیشتر کتابخانه های سیستم است (ولی، متاسفانه نه تمام آنها)

یک برنامه مقدماتی

با نام خدا شروع می کنیم

برنامه زیر را در ویرایشگر مورد علاقه خود تایپ کنید.

```
# include <stdio .h>
void main ( )
{
print f (“\ nHello world \n”);
}
```

کد را در فایل hello.c ذخیره، و سپس با تایپ gcc hello.c آن را ترجمه (کامپایل) کنید.

این یک فایل قابل اجرای a.out را بعد از اجرا بسادگی با تایپ نام آن ایجاد می کند.

در نتیجه کاراکترهای Hello world با یک خط خالی چاپ می شود. یک برنامه C شامل توابع و متغیر هاست. توابع کارهایی که توسط برنامه اجرا می شوند را مشخص می کنند.

تابع «main» (اصلی) در بالای برنامه قرار می گیرد. بطور معمول کوتاه بوده و توابع مختلف را برای اجرای زیر – وظایف فراخوانی می کند. همه کدهای C باید دارای تابع main باشند.

کد `hello.c`, `printf` را که یک تابع خروجی از کتابخانه I/O (ورودی /خروجی) که در فایل `stdio.h` تعریف شده) را فراخوانی می کند. زبان اصلی C به هیچوجه دارای حکم های I/O توکار (درونی) نمی باشد. توابع ریاضی زیادی نیز ندارد. زبان اصلی واقعا به منظور محاسبات علمی یا تکنیکی نبوده ... این توابع اکنون توسط کتابخانه استاندارد اجرا می شوند، که اکنون بخشی از ANSI C هستند. کتاب K & R محتوای اینها و کتابخانه های استاندارد دیگر در ضمیمه (پیوست) را فهرست می کند.

خط `printf` پیام `hello word` را بر `stdout` (مسیر خروجی متناظر با پنجره ترمینال یا پایانه x در آنچه شما کد را اجرا می کنید) چاپ می کند.

“\n” یک خط جدید چاپ نموده، که کرسور را به خط بعدی می آورد. توسط `printf`، این خط هرگز ایجاد نمی شود. برنامه زیر همین نسخه را تولید می کند:

```
<include <stdio.h#
```

```
void main()
{
    printf("\n");
    printf("Hello World");
    printf("\n");
}
```

سعی کنید “\n” را خارج کرده و نتیجه را ببینید.

اولین حکم `#include<stdio.h>` شامل مشخصه کتابخانه I/O زبان C است. همه متغیرهای C باید صریحا قبل از استفاده تعریف شوند. فایل های “h” فایل های سر آیند بوده که شامل تعریف متغیرها و توابع لازم برای کار کردن برنامه ها چه در بخش نوشته شده توسط کاربر از کد باشند، یا بعنوان کتابخانه های استاندارد C باشد، می باشد. نماد `<...>` مترجم را برای جستجوی فایل در کتابخانه های استاندارد سیستم راهنمایی می کند.

کلمه `void` قبل از `main` مشخص می کند که تابع `main` از نوع `void` است. یعنی هیچ نوع وابسته به آن ندارد. به این معنا که نمی تواند نتیجه ای را هنگام اجرا برگرداند.

کاراکتر-“” انتهای دستور را مشخص می کند. بلوکهای دستورات در آکولاد {...} قرار می گیرند، همانند تعریف توابع. تمام دستورات C در فرمت آزاد تعریف می شوند. بدون طرح بندی مشخص شده یا انتساب ستون. فضای سفید (تباها یا فاصله ها) معنی ندارند. بجز درون کوتیشن ها (نقل قول) بعنوان بخشی از رشته کاراکتری. برنامه زیر دقیقا همین نتیجه را برای مثال ما می دهد:

```
#include <stdio.h>
void main(){printf("\nHello World\n");}
```

دلایل مرتب کردن برنامه در خطوط و نمایش ساختار واضح است.

محاسبه

برنامه زیر، sine.c جدولی از توابع sine را برای زاویه های بین ۰ تا ۳۶۰ درجه محاسبه می کند.

```

/*****
/* Table of      */
/* Sine Function */
*****/

/* Michel Vallieres */
/* Written: Winter 1995 */

#include <stdio.h>
#include <math.h>

void main()
{
    int  angle_degree;
    double angle_radian, pi, value;

        /* Print a header */
    printf ("\nCompute a table of the sine function\n\n");

        /* obtain pi once for all */
        /* or just use pi = M_PI, where
        M_PI is defined in math.h*/
    pi = 4.0*atan(1.0);
    printf ( " Value of PI = %f\n\n", pi );

    printf ( " angle   Sine\n" );

    angle_degree=0;          /* initial angle value      */
                            /* scan over angle          */

    while ( angle_degree <= 360 ) /* loop until angle_degree > 360 */
    {
        angle_radian = pi * angle_degree/180.0 ;
        value = sin(angle_radian);
        printf ( " %3d   %f\n ", angle_degree, value );

        angle_degree = angle_degree + 10; /* increment the loop index */
    }
}

```


کد با توضیحاتی که هدف آن را مشخص می کند شروع می شود.

این یک سبک خوب برنامه نویسی است که کار شما را مشخص و مستند می کند.

توضیحات می توانند هر جایی از کد نوشته شوند. هر کاراکتر بین `*`، `\`، `*` توسط مترجم نادیده گرفته می شود و فقط برای کد استفاده می شود. استفاده از نامهای متغیر با معنی در متن مساله نیز نظر خوبی است.

دستور `# include` اکنون شامل فایل سر آیند برای کتابخانه ریاضی استاندارد `math.h` می باشد. این دستور نیاز به تعریف فراخوانی های توابع مثلثاتی `atsn` و `sin` دارد. (توجه کنید که `compilation` همچنین باید شامل کتابخانه ریاضیات دقیقاً با تایپ `gcc sine.c` -`lm` باشد).

نام های متغیر دلخواه است. (توسط بعضی از مترجم ها طول ماکزیمم نوعاً `۳.۲` کاراکتر تعریف شده) `c` از انواع متغیرهای استاندارد زیر استفاده می کند.

`int`: متغیر صحیح

`short`: صحیح کوتاه

`long`: صحیح طولانی

`float`: متغیر حقیقی

`double`: متغیر حقیقی (متغیر شناور) دقت مضاعف

`char`: متغیر کاراکتری (یک بایت)

مترجم ها سازگاری انواع همه متغیرهای استفاده شده در کد را بررسی می کنند. این ویژگیها باعث جلوگیری از اشتباه می شود. و خصوصاً در خطای نوع نام های متغیر. محاسبات انجام شده در روالها در کتابخانه ریاضی معمولاً در دقت مضاعف محاسبه شده. (۶۴ بیت در بیشتر ایستگاههای کاری) تعداد واقعی بایتها در حافظه درونی این انواع داده وابسته به ماشینی که استفاده شده بکار رفته است.

تابع `printf`، می تواند برای چاپ مقادیر صحیح، حقیقی و رشته ها بکار رود

نحو کلی این است:

`printf("format", variable);`

که `format` مشخصه تبدیل و `variable` فهرست مقادیر چاپ است. بعضی فرمتهای مورد استفاده عبارتند از:

`%nd`: صحیح (`n` اختیاری = تعداد ستونها؛ اگر `0` باشد، با صفر پر می شود)

`%mnf`: متغیر حقیقی ساده یا مضاعف (`m` اختیاری = تعداد ستونها و `n` تعداد ارقام اعشار)

`%ns`: رشته (`n` اختیاری = تعداد ستونها)

`%c`: کاراکتر

`n\t`: برای ایجاد خط یا تب جدید

`\g`: صدای زنگ ("بیب") بر روی پایانه (ترمینال)

۳- حلقه ها:

اکثر برنامه های واقعی شامل ساختاری است که در برنامه حلقه می زند. اجرای عملهای مکرر در یک جریان داده یا ناحیه ای از حافظه. چندین روش برای حلقه در C وجود دارد. دو حلقه بسیار رایج یکی while است.

while (عبارت)

```
.....}
{
```

و دیگر حلقه for

for (عبارت ۳؛ عبارت ۲؛ عبارت ۱)

```
.....}
{
```

حلقه while به حلقه ادامه می دهد تا زمانیکه عبارت شرطی نادرست شود. شرط هنگام ورود به حلقه تست می شود. هر ساختار منطقی (فهرست زیر را ببینید) می تواند در این زمینه استفاده شود. حلقه for یک نوع خاص است و معادل است با حلقه while زیر.

؛ عبارت ۱

while (عبارت ۲)

```
.....}
{
```

؛ عبارت ۳

برای نمونه، ساختار زیر اغلب مشاهده می شود:

$i = i$ مقدار اولیه ;

$(i \leq i)$ while (ماکزیمم)

```
...}
{
```

$i = i + i$ نمو ;

این ساختار با نحو آسانتر حلقه for بازنویسی می شود:

(i) for = مقدار اولیه ; $> i =$ حداکثر ; $i + i =$ نمو

```
.....}
{
```

حلقه بی نهایت نیز ممکن است (برای مثال (; ; for)). ولی برای هزینه کامپیوتر خیلی خوب نیست!

C به شما اجازه نوشتن حلقه بی نهایت را می دهد و دستور break را برای شکستن و خروج از حلقه فراهم کرده است. برای مثال، به مثال زیر که بازنویسی شده حلقه قبلی است توجه کنید:

```
angle_degree = 0;

for ( ; ; )
{
..... بلوک دستورات .....
angle_degree = angle_degree + 10;
if (angle_degree == 360) break;
}
```

شرط if به طور ساده می پرسد آیا degree _ angle (درجه زاویه) مساوی ۳۶۰ هست یا نه؛ اگر هست، حلقه تمام شود (بایستد)

ثوابت نهادی

شما می توانید ثابت های هر نوع را با استفاده از راهنمای مترجم # define تعریف کنید، نحو آن ساده است . برای نمونه

```
,#define ANGLE_MIN 0
#define ANGLE_MAX 360
```

ANGLE_MAX,ANGLE_MIN را به ترتیب مقادیر ۰ و ۳۶۰ درجه تعریف می کند. C بین حروف کوچک و بزرگ در نام متغیرها تفاوت قائل می شود. معمول است که از حروف بزرگ در تعریف ثابتهای سراسری استفاده شود.

شرطها

شرط ها با ساختارهای if و while استفاده می شوند.

```
(if شرط )
{
..... بلوک دستورات اجرایی اگر شرط ۱ درست باشد .....
}
else if ( شرط ۲ )
{
..... بلوک دستورات اجرایی اگر شرط ۲ درست باشد .....
}
else
{
..... بلوک دستورات اجرایی در غیر اینصورت .....
}
```

و مشتقات متفاوت آن. هم با حذف شاخه ها یا شامل شرایط تو در تو. شرط ها عملگرهای منطقی شامل مقایسه مقادیر (از یک نوع) هستند که از عملوندهای شرطی استفاده می کنند.

< کوچکتر از

= < کوچکتر از یا مساوی با

= = مساوی با

= ! مساوی نیست با

= > بزرگتر از یا مساوی با

> بزرگتر از

و عملوند های بولی

&& و

|| یا

not نه

کاربرد دیگر شرط در ساختار switch است.

switch (عبارت)

}

case عبارت ثابت ۱ ;

}

..... بلوک دستورات

; break

{

case عبارت ثابت ۲ ;

}

..... بلوک دستورات

; break

{

: default

}

..... بلوک دستورات

{

}

بلوک مناسب دستورات مطابق مقدار عبارت، مقایسه شده توسط عبارت های ثابت در دستور case اجرامی شود. دستور break اطمینان می دهد که case هایی که در ادامه case انتخاب شده آمده اند اجرا نخواهند شد.

اگر شما می خواهید این دستورات را اجرا کنید. باید دستور break را رها کنید. این ساختار بویژه در بررسی متغیر ها ورودی کاربرد دارد.

اشاره گرها

زبان C به برنامه نویس اجازه خواندن و نوشتن در محل‌های حافظه را بطور مستقیم می‌دهد. این قابلیت انعطاف بالا و قدرت زبان را نشان می‌دهد. البته یکی از موانع بزرگی است که برنامه نویس مبتدی باید در استفاده از زبان چیره شود. همه متغیرهای یک برنامه در حافظه مقیم هستند. دستورات

```
; float X
x = 6.5
```

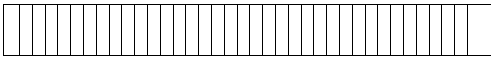
نیاز دارد که مترجم ۴ بایت از حافظه را در یک کامپیوتر ۳۲ بیتی برای متغیرهای ممیز شناور x ذخیره کند و سپس مقدار 6.5 را در آن قرار دهد.

گاهی اوقات ما می‌خواهیم بدانیم یک متغیر در کجای حافظه مقیم است. آدرس (مکان در حافظه) هر متغیر با قرار دادن عملوند "&" قبل از نام آن بدست می‌آید. بنابراین &px آدرس x است.

C به ما امکان رفتن به مرحله بالاتر و تعریف متغیر را می‌دهد، که اشاره گر نامیده می‌شود. شامل آدرس (یعنی "points to") (اشاره به) متغیرهای دیگر است. برای مثال

```
float x;
float* px;
x = 6.5;
px = &x;
```

px را به عنوان اشاره گر به اشیاء نوع شناور تعریف می‌کند، و آن را برابر با آدرس x قرار می‌دهد



اشاره گر را برای متغیر استفاده می‌کند.

محتوای مکان حافظه رجوع شده توسط اشاره گر با استفاده از عملوند "*" بدست می‌آید. (این اشاره گر، اشاره گر مرجع نامیده می‌شود)

بنابراین، *px به مقدار x مراجعه می‌کند.

C به ما اجازه اجرای عملگرهای حسابی را با استفاده از اشاره گرها می‌دهد. البته آگاه باشید که واحد (UNIT) در اشاره گر حسابی اندازه (بر حسب بایت) شی که اشاره گر، به آن اشاره می‌کند است. برای مثال، اگر px یک اشاره گر به متغیر x از نوع حقیقی باشد، عبارت px + 1 به بیت یا بایت بعدی حافظه مراجعه نمی‌کند بلکه به مکان نوع حقیقی بعد از x (چهار بایت بعد در بیشتر سیستم‌های کاری) مراجعه می‌کند.

اگر x از نوع مضاعف باشد px + 1 به مکان ۸ بایت (اندازه مضاعف) بعد مراجعه می‌کند. و مانند آن. فقط اگر x از نوع کاراکنتر باشد px + 1 واقعا به بایت بعدی در حافظه مراجعه می‌کند.

بنابراین، در

```
char* pc;
float* px;
float x;

x = 6.5;
px = &x;
```

`pc = (char*) px;`
 ((`*chart`)) در خط آخر یک "طرح" است که یک نوع داده را به نوع دیگر تبدیل می کند (`px` و `pc` هر دو به یک مکان در حافظه اشاره می کنند. یعنی آدرس `x`. البته `px+1` و `pc+1` به مکانهای متفاوتی از حافظه اشاره می کنند.
 به کد ساده زیر توجه کنید:

```
void main()
{
    float x, y;           /* x and y are of float type */
    float *fp, *fp2;     /* fp and fp2 are pointers to float */

    x = 6.5;             /* x now contains the value 6.5 */

                        /* print contents and address of x */
    printf("Value of x is %f, address of x %ld\n", x, &x);

    fp = &x;            /* fp now points to location of x */

                        /* print the contents of fp */
    printf("Value in memory location fp is %f\n", *fp);

                        /* change content of memory location */
    *fp = 9.2;
    printf("New value of x is %f = %f\n", *fp, x);

                        /* perform arithmetic */
    *fp = *fp + 1.5;
    printf("Final value of x is %f = %f\n", *fp, x);

                        /* transfer values */
    y = *fp;
    fp2 = fp;
    printf("Transferred value into y = %f and fp2 = %f\n", y, *fp2);
}
```

این کد را اجرا کنید و نتایج این عملوند های مختلف را ببینید. دقت کنید که، زمانیکه مقدار اشاره گر (اگر شما آن را با `printf` چاپ کنید) نوعا یک متغیر بزرگ است؛ مکان خاصی از حافظه در کامپیوتر، را مشخص می کند. اشاره گرها مقادیر صحیح نیستند - آنها یک نوع داده کاملا متفاوت هستند.

۷-آرایه ها:

آرایه ها از هر نوع می توانند در `C` شکل بگیرند. نحو آن ساده است:

[بعد] نام آرایه

در `c`، آرایه ها از 0 شروع می شوند. عناصر آرایه مکانهای مجاور در حافظه را اشغال می کنند. `c` با نام آرایه مانند اینکه آنها یک اشاره گر به اولین عنصر باشد رفتار می کنند. این در فهمیدن اینکه چگونه محاسبات توسط آرایه ها انجام گیرد مهم است. بنابراین، اگر `v` یک آرایه باشد `*V` یک چیزی است نظیر `v[0]`، `*(v+1)` چیزی نظیر `v[1]` و به همین ترتیب.

اشاره گر برای یک آرایه استفاده می شود.

به کد زیر توجه کنید، که کاربرد اشاره گرها را با مثال توضیح می دهد.

```
#define SIZE 3

void main()
{
    float x[SIZE];
    float *fp;
    int i;

    /* initialize the array x */
    /* use a "cast" to force i */
    /* into the equivalent float */
    for (i = 0; i < SIZE; i++)
        x[i] = 0.5*(float)i;

    /* print x */
    for (i = 0; i < SIZE; i++)
        printf(" %d %f\n", i, x[i]);

    /* make fp point to array x */
    fp = x;

    /* print via pointer arithmetic */
    /* members of x are adjacent to */
    /* each other in memory */
    /* *(fp+i) refers to content of */
    /* memory location (fp+i) or x[i] */
    for (i = 0; i < SIZE; i++)
        printf(" %d %f\n", i, *(fp+i));
}
```

(عبارت `"i++"` مختصر نویسی `c` برای `"i = i+1"` است). `x[i]` یعنی `i` اولین عنصر آرایه `x`، `fp=x`، به شروع `x` اشاره می کند، سپس `*(fp+i)` محتوای آدرس `i` مکانهای بعد از `fp` است. که `x[i]` می باشد.

۸- آرایه های کاراکتری

یک ثابت رشته ای، نظیر "I am a string"

یک آرایه از کاراکترها می باشد. که به طور داخلی در C توسط کاراکترهای اسکی (ASCII) در رشته موجود است.

یعنی "i"، فاصله و "a" و "m" و برای رشته بالا، وبا کاراکتر ویژه /0 پایان یافته بنابراین برنامه می تواند انتهای رشته را پیدا کند.

ثابت های رشته ای اغلب در ایجاد خروجی کد قابل فهم با استفاده از printf بکار می روند.

```
printf("Hello, world\n");
printf("The value of a is: %f\n", a);
```

ثابت های رشته می توانند با متغیرها در ارتباط باشند. C متغیر نوع char را فراهم کرده. که می تواند شامل یک کاراکتر باشد - ۱ بایت - در یک زمان. یک رشته کاراکتر در یک حافظه نوع کاراکتری ذخیره می شود، یک کاراکتر اسکی در مکان هرگز فراموش نکنید که از آنجا که رشته ها بطور قرار دادی با کاراکتر پوچ "\0" خاتمه می یابند، ما نیاز به یک مکان ذخیره اضافه در آرایه داریم!

کدهای C هیچ عملوندی را برای دستکاری یکباره رشته های درست فراهم نکرده. رشته ها یا توسط اشاره گرها یا از طریق روال های خاص قابل دسترسی از کتابخانه استاندارد رشته string دستکاری می شوند. استفاده از اشاره گر کاراکتری نسبتا ساده است چون نام آرایه فقط اشاره گر به اولین عنصر آن می باشد.

به کد زیر توجه کنید:

```
void main()
{
    char text_1[100], text_2[100], text_3[100];
    char *ta, *tb;
    int i;

    /* set message to be an array */
    /* of characters; initialize it */
    /* to the constant string "... " */
    /* let the compiler decide on */
    /* its size by using [] */
    char message[] = "Hello, I am a string; what are you?";

    printf("Original message: %s\n", message);

    /* copy the message to text_1 */
    /* the hard way */
    i=0;
    while ( (text_1[i] = message[i]) != '\0' )
        i++;
    printf("Text_1: %s\n", text_1);

    /* use explicit pointer arithmetic */
    ta=message;
    tb=text_2;
```



```
while ( ( *tb++ = *ta++ ) != '\0' )
;
printf("Text_2: %s\n", text_2);
```

{
کتابخانه استاندارد رشته شامل توابع مفید زیادی است که رشته‌ها را دستکاری می‌کنند. توصیف این کتابخانه در پیوست کتاب *k & R* یافت می‌شود. بعضی از توابع مفیدتر عبارتند از:

strcpy (c, cf) : *char** ct را به داخل *s* کپی می‌کند. شامل *"\0"*؛ *s* را برمی‌گرداند.

strncpy(c, cf, n) : *char ** n کاراکتر *ct* را به داخل *s* کپی می‌کند، *s* را بر می‌گرداند.

strncat (c, ct) : *char ** ct را به انتهای *s* الحاق می‌کند؛ *s* را بر می‌گرداند.

strncat (c, ct, n) : *char** n کاراکتر *ct* را به انتهای *s* الحاق می‌کند، با *"\n"* خاتمه یافته *s* را بر می‌گرداند.

strcmp (cs, ct) : *int* *cs* و *ct* را مقایسه می‌کند؛ اگر *cs=ct* باشد 0، اگر *cs<ct* مقدار منفی و اگر *cs>ct* مقدار مثبت را برمی‌گرداند.

strchr (cs, c) : اشاره گر به اولین رخداد *c* در *cs* را بر می‌گرداند یا مقدار *NULL* را بر می‌گرداند.

strlen (cs) : طول *cs* را بر می‌گرداند.

(*s* و *t* از نوع *char ** هستند. *cs* و *ct* ثابت *char**، *c* یک کاراکتر از نوع *char* تبدیل شده به صحیح و *n* یک نوع صحیح است.)
به کد زیر که از بعضی از این توابع استفاده می‌کند. توجه کنید.

```
#include < string.h>
```

```
void main()
{
    char line[100], *sub_text;
        /* initialize string */
    strcpy(line, "hello, I am a string.");
    printf("Line: %s\n", line);
        /* add to end of string */
    strcat(line, " what are you?");
    printf("Line: %s\n", line);
        /* find length of string */
        /* strlen brings back */
        /* length as type size_t */

    printf("Length of line: %d\n", (int)strlen(line));

        /* find occurrence of substrings */
    if ( (sub_text = strchr ( line, 'W' ) )!= NULL )
        printf("String starting with \"W\" ->%s\n", sub_text);

    if ( ( sub_text = strchr ( line, 'w' ) )!= NULL )
        printf("String starting with \"w\" ->%s\n", sub_text);
```

```

if ( ( sub_text = strchr ( sub_text, 'u' ) ) != NULL )
    printf("String starting with \"w\" ->%s\n", sub_text);
}

```

قابلیت های ورودی / خروجی (I/O)

سطح کاراکتر I/O

C (از طریق کتابخانه هایش) روتین های گوناگون I/O را فراهم کرده است. در سطح کاراکتر () `getchar` کاراکتر را در یک زمان از `stdin` می خواند. هنگامیکه () `putchar` یک کاراکتر را در یک زمان روی `stdout` می نویسد. برای مثال، توجه کنید:

```

include < stdio.h>#

void main()
{
    int i, nc;

    nc = 0;
    i = getchar();
    while ( i != EOF ) {
        nc = nc + 1;
        i = getchar();
    }
    printf("Number of characters in file = %d\n", nc);
}

```

این برنامه تعداد کاراکترها را در جریان ورودی می شمارد. (مثلا در یک فایل لوله شده در زمان اجرا) کد کاراکترها را (هر چقدر ممکن است باشند) از `stdin` (صفحه کلید) خوانده، `stdout` را (پایانه x که شما از آن اجرا می کنید) برای خروجی استفاده نموده و پیام خطایی برای `stderr` (معمولا پایانه x شما) می نویسد. این جریانات همیشه در زمان اجرا تعریف می شوند. EOF (پایان فایل) مقدار مشخصی را بر می گرداند. که در `stdio.h` تعریف شده و توسط () `getchar` هنگامیکه در زمان خواندن با نشانه `end-of-file` مواجه شد، بر گردانده می شود. مقدار آن به کامپیوتر بستگی دارد. البته کامپایلر C این واقعیت را از کاربر با تعریف متغیر EOF پنهان می کند. بنابراین برنامه کاراکترها را از `stdin` خوانده و مجموع آن را در شمارنده `nc` نگه می دارد. تا زمانی که با "end of file" (انتهای فایل) روبرو شود.

یک برنامه نویس با تجربه C احتمالاً این برنامه را به صورت زیر کد می کند:

```

<include < stdio.h>#

void main()
{
    int c, nc = 0;

```

```
while ( ( c = getchar() ) != EOF ) nc++;

printf("Number of characters in file = %d\n", nc);
}
```

c امکان عبارات مختصر زیادی را می دهد. معمولا با قابلیت خوانائی!

() در دستورات (c=getchar()) اجرای فراخوانی () getchar را بیان می کند. و نتیجه را c قبل از مقایسه آن با EOF انتساب می کند. (قرار می دهد)

کروشه ها اینجا لازم است. بیاد داشته باشید نماد گذاری nc++ (و در حقیقت، ++nc) راه دیگری برای نوشتن nc=nc+1 است. (تفاوت بین نمادگذاری پیشوند و پسوند است که در ++nc، nc قبل از اینکه استفاده شود افزایش یافته. در حالیکه در nc++، nc قبل از اینکه افزایش یابد استفاده می شود.

در این مثال خاص، هر دو انجام شده (این نماد گذاری فشرده تر است و اغلب بطور مؤثرتری توسط مترجم کد می شود. فرمان wc یونیکس کاراکترها و کلمات و خطوط را در یک فایل می شمارد. برنامه بالا می تواند بعنوان wc شما مورد توجه قرار گیرد. اجازه دهید شمارنده ای را برای خطوط اضافه کنم.

```
<#include <stdio.h>
```

```
void main()
{
    int c, nc = 0, nl = 0;

    while ( ( c = getchar() ) != EOF )
    {
        nc++;
        if (c == '\n') nl++;
    }

    printf("Number of characters = %d, number of lines = %d\n",
        nc, nl);
}
```

شما می توانید در مورد راههای شمارش تعداد کلمات در فایل فکر کنید؟

قابلیت های سطح بالای I/O

هنوز می بینیم که printf خروجی فرمت بندی شده با stdout را بکار می بردودر مقابل دستور خواندن از stdin، scanf است. نحو آن

```
scanf ("format string", variables);
```

مانند printf است. فرمت رشته ممکن است شامل فاصله ها یا تب ها باشد. (چشم پوشی می شود) کاراکترهای اسکی معمولی، که باید آنها را با stdin مطابقت داد، و تبدیل مشخصات مانند printf دستورات معادل برای خواندن یا نوشتن رشته های کاراکتری موجود است. آنها عبارتند از:

```
sprintf(string, "format string", variables);
scanf(string, "format string", variables);
```

این کد را ترجمه و اجرا کنید؛ سپس ویرایشگری را برای خواندن فایل `foo.dat` بکار ببرید.

توابع

توابع برای استفاده آسانند؛ آنها اجازه می دهند که برنامه های پیچیده به بلوک های کوچک تری تجزیه شوند، هر کدام از آنها برای نوشتن، خواندن و نگهداری آسانترند. ما تا کنون با تابع `main` بر خورد داشته و روال های ریاضی از کتابخانه های استاندارد استفاده کرده ایم. اکنون اجازه دهید به برخی دیگر از توابع کتابخانه ای و چگونگی نوشتن و استفاده از آنها نگاهی بیندازیم.

فراخوانی یک تابع

فراخوانی یک تابع در C بسادگی شامل مراجعه به نام آن توسط نشانندهای مناسب می باشد. مترجم مطابقت بین آرگومانها در دنباله فراخوانی و تعریف تابع را بررسی می کند.

توابع کتابخانه ای عموماً در فرم منبع در دسترس ما نیستند. بررسی نوع نشانوند از طریق استفاده از فایل های سرآیند (مانند `stdio.h`) انجام می شود که شامل همه اطلاعات لازم می باشد. برای مثال، همانطور که بزودی خواهیم دید، بمنظور استفاده از کتابخانه ریاضی استاندارد شما باید `math.h` را از طریق دستور

```
#include <math.h>
```

در بالای فایل محتوی کدتان اضافه کنید. فایل های سرآیند رایجتر عبارتند از:

`<stdio.h>`: تعریف روالهای I/O

`<ctype.h>`: تعریف روالهای دستکاری کارکترها

`<string.h>`: تعریف روالهای دستکاری رشته

`<math.h>`: تعریف روالهای ریاضی

`<stdlib.h>`: تعریف تبدیل عدد تخصیص حافظه و وظایف (کارهای) مشابه

`<time.h>`: تعریف روالهای زمان دستکاری

به علاوه، فایل های سرآیند زیر هم موجود است:

`<assert.h>`: تعریف روالهای تشخیص

`<setjmp.h>`: تعریف فراخوانی های تابع غیر محلی

`<signal.h>`: تعریف گرداننده های سیگنال

`<limits.h>`: تعریف ثابت هایی از نوع صحیح

`<float.h>`: تعریف ثابتهایی از نوع شناور

پیوست B در کتاب K & R این کتابخانه ها را با جزئیات بیشتر توضیح می دهد.

نوشتن تابع های خودتان

یک تابع دارای طرح بندی زیر می باشد:

```
(فهرست نشانوند اگر لازم باشد) نام تابع (نوع بازگشتی )
```

```
{
..... اعلان محلی .....
```

```
..... دستورات .....
```

```
return مقدار بازگشتی ;
```

```
}
```

اگر نوع بازگشتی حذف شده باشد، C بطور پیش فرض از `int` استفاده می کند. مقدار بازگشتی باید از نوع اعلان شده باشد.

یک تابع ممکن است بطور ساده یک وظیفه را بدون بازگشت هیچ مقداری اجرا کند. در این حالت دارای طرح بندی زیر می باشد:

```
void (فهرست نشانوند اگر لازم باشد) نام تابع
```

```
{
```

```
..... اعلان های محلی .....
```

```
..... دستورات .....
```

```
}
```

یک مثال از فراخوانی تابع X به کد زیر توجه کنید:

```
/* include headers of library */
```

```
/* defined for all routines */
```

```
/* in the file */
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
/* prototyping of functions */
```

```
/* to allow type checks by */
```

```
/* the compiler */
```

```
void main()
```

```
{
```

```
int n;
```

```
char string[50];
```

```
/* strcpy(a,b) copies string b into a */
```

```
/* defined via the stdio.h header */
```

```
strcpy(string, "Hello World");
```

```
/* call own function */
```

```
n = n_char(string);
```

```
printf("Length of string = %d\n", n);
```

```
}
```

```
/* definition of local function n_char */
```

```

int n_char(char string[])
{
    /* local variable in this function */
    int n;
    /* strlen(a) returns the length of */
    /* string a */
    /* defined via the string.h header */
    n = strlen(string);
    if (n > 50)
        printf("String is longer than 50 characters\n");

    /* return the value of integer n */
    return n;
}

```

نشانوند ها همیشه توسط مقدار در فراخوانی های تابع رد می شوند. یعنی کپی های محلی مقادیر نشانوند ها از روالها عبور می کنند. هر تغییر در نشانوند ها بطور درونی در تابع فقط در کپی های محلی نشانوند ها ایجاد می شود.

بمنظور تغییر (یا تعریف) نشانوند در فهرست نشانوند، این نشانوند باید بعنوان یک آدرس عبور کند، (در نتیجه الزام C برای تغییر نشانوند واقعی در روال فراخوانی).

بعنوان یک مثال به تعویض دو عدد بین متغیرها توجه کنید، ابتدا اجازه دهید آنچه اتفاق می افتد را اگر متغیرها توسط مقدار عبور کنند را با مثال نشان دهیم

```

#include <stdio.h>

void exchange(int a, int b);

void main()
{
    /* WRONG CODE */
    int a, b;

    a = 5;
    b = 7;
    printf("From main: a = %d, b = %d\n", a, b);

    exchange(a, b);
    printf("Back in main: ");
    printf("a = %d, b = %d\n", a, b);
}

void exchange(int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}

```

```
printf(" From function exchange: ");
printf("a = %d, b = %d\n", a, b);
}
```

این کد را اجرا کنید و ملاحظه کنید که **a** و **b** عوض نشده اند: فقط کپی های نشانوند ها تعویض شده اند:
البته روش صحیح انجام این کار استفاده از اشاره گر هاست:

```
#include <stdio.h>

void exchange ( int *a, int *b );

void main()
{
    /* RIGHT CODE */
    int a, b;

    a = 5;
    b = 7;
    printf("From main: a = %d, b = %d\n", a, b);

    exchange(&a, &b);
    printf("Back in main: ");
    printf("a = %d, b = %d\n", a, b);
}

void exchange ( int *a, int *b )
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
    printf(" From function exchange: ");
    printf("a = %d, b = %d\n", *a, *b);
}
```

قواعد ملموس اینجا عبارتند از :

- شما متغیرهای منظم (باقاعده) را استفاده می کنید اگر تابع مقادیر آن نشانوند ها را متغیر ندهد.
- شما باید از اشاره گر ها استفاده کنید اگر تابع مقادیر آن نشانوندها را تغییر دهد.

نشانوندهای خط فرمان

این یک تمرین استاندارد یونیکس برای اطلاعات در مورد گذر از خط فرمان به برنامه به طور مستقیم از طریق استفاده از یک یا بیشتر از نشانوندهای خط فرمان است. (switches) سوئیچ ها نوعاً برای تغییر رفتار برنامه، یا تنظیم مقادیر برخی پارامترهای ورودی استفاده می

شود. شما تا کنون با چند مورد از اینها روبرو شده اید. برای مثال، فرمان «ls» فایلها را در فهرست جاری شما لیست می کند. ولی هنگامیکه سوئیچ ۱- اضافه شود، «ls -1» یک فهرست طولانی را در عوض تولید می کند. به طور مشابه «ls -1 -a» یک فهرست طولانی، شامل فایل‌های پنهان را تولید می کند. فرمان «tail -20» و ۲۰ خط آخر فایل را (به جای مقدار پیش فرض ۱۰) چاپ می کند، و مانند آن .

به طور فرضی، سوئیچ‌ها بسیار شبیه نشاننده‌های توابع در C رفتار می کنند. و آنها از برنامه C از سیستم عامل عبور می کنند. دقیقا با همان روشی که نشاوندها بین توابع عبور می کنند. تا کنون، دستورات () main در برنامه های ما هیچ چیزی بین پرانتزها نداشته. بنابراین، یونیکس واقعا برای برنامه ها (هنگامیکه برنامه نویس انتخاب می کند که از اطلاعات استفاده کند یا نه) دو نشانوند برای main رافراهم می سازد.

یک آرایه از رشته های کاراکتری، که بطور مرسوم argv نامیده می شود و یک صحیح، که معمولا argv نام دارد. که تعداد رشته های آرایه را مشخص می کند. دستور کامل خط اول برنامه عبارتست از:

```
main ( integer ,char** argv)
```

(نحو char**argv مشخص می کند argv باید اشاره گری به اشاره گر کاراکتر باشد، که آن، اشاره گری است به آرایه کاراکتر (یک رشته کارکتری) از لحاظ کلمات، یک آرایه از رشته های کاراکتری. شما همچنین می توانید آن را به صورت (chat*argv بنویسید به هر حال خیلی در مورد جزئیات نحو آن نگران نباشید. کاربرد آرایه در زیر واضح تر خواهد شد.)

هنگامیکه شما یک برنامه را اجرا می کنید، آرایه argv شامل همه اطلاعات خط فرمان، هنگامیکه شما فرمانی را وارد کردید می شود. * رشته ها با فضای سفید مشخص می شوند) شامل خود فرمان صحیح argv تعداد کل رشته ها را می دهد، بنابراین برابر است با تعداد نشانوندها به اضافه یک. برای مثال اگر شما q 3 -x -g 2 -i a.out را تایپ کنید برنامه مقادیر زیر را دریافت می کند.

```
argc = 7
argv[0] = "a.out"
argv[1] = "-i"
argv[2] = "2"
argv[3] = "-g"
argv[4] = "-x"
argv[5] = "3"
argv[6] = "4"
```

دقت کنید که نشانوندها، حتی مقادیر عددی، در این مورد رشته هستند. این وظیفه برنامه نویس است که آنها را کد گشایی کرده و تصمیم بگیرد با آنها چه بکند.

برنامه زیر بطور ساده نام و نشانوندهایش را چاپ می کند.

```
#include <stdio.h>
```

```
main(int argc, char** argv)
{
    int i;

    printf("argc = %d\n", argc);
```



```

for (i = 0; i < argc; i++)
    printf("argv[%d] = \"%s\\n\"", i, argv[i]);
}

```

برنامه نویسان یونیکس قرار دادهای مشخصی در مورد چگونگی تفسیر فهرست نشانوندها دارند. آنها اجباری نیستند بلکه برنامه شما را بای استفاده و فهم دیگران آسانتر می سازند. اول اینکه اصطلاحات سوئیچ ها و کلیدها با کاراکتر « - » شروع می شوند. این باعث می شود که به آسانی آنها را تشخیص دهید. سپس ۷ بسته به سوئیچ، نشانوندها ممکن است شامل اطلاعاتی که بعنوان صحیح، شناور یا فقط بعنوان یک رشته کاراکتری نگه داشته شده اند تفسیر شوند. با این قرار دادها، رایجترین راه تجزیه فهرست نشانوند بوسیله حلقه For و یک دستور switch می باشد.

مانند زیر:

```

#include <stdio.h>
#include <stdlib.h>

main(int argc, char** argv)
{
    /* Set defaults for all parameters: */

    int a_value = 0;
    float b_value = 0.0;
    char* c_value = NULL;
    int d1_value = 0, d2_value = 0;

    int i;

    /* Start at i = 1 to skip the command name. */

    for (i = 1; i < argc; i++) {

        /* Check for a switch (leading "-"). */

        if (argv[i][0] == '-') {

            /* Use the next character to decide what to do. */

            switch (argv[i][1]) {

                case 'a':    a_value = atoi(argv[++i]);
                            break;

                case 'b':    b_value = atof(argv[++i]);
                            break;

                case 'c':    c_value = argv[++i];
                            break;

```

```

        case 'd':    d1_value = atoi(argv[++i]);
                   d2_value = atoi(argv[++i]);
                   break;
    }
}

printf("a = %d\n", a_value);
printf("b = %f\n", b_value);
if (c_value != NULL) printf("c = \"%s\"\n", c_value);
printf("d1 = %d, d2 = %d\n", d1_value, d2_value);
}

```

دقت کنید که `argv [i] [j]` یعنی `j` امین کاراکتر `i` امین کاراکتری دستور `if` راهنمای `if` (کاراکتر 0) را بررسی می کند. سپس دستور `switch` امکان تغییرهای گوناگون عملیاتی که بسته به کاراکتر بعدی در رشته گرفته می شوند را می دهد. (اینجا کاراکتر)

دقت کنید که استفاده از `argv[++i]` برای افزایش `i` قبل از استفاده، اجازه می دهد به رشته بعدی در یک دستور فشرده تکی دستیابی داشته باشیم. توابع `atoi` و `atof` در `stdlib.h` تعریف شده اند.

آنها از رشته های کاراکتری به ترتیب به صحیح (`ints`) و مضاعف (`doubles`) تبدیل می کنند.

یک خط فرمان تایپ شده ممکن است چنین باشد:

```
a.out -a 3 -b 5.6 -c "I am a string" -d 222 111
```

(کاربرد دبل کوتیشن (")) با `-c` در اینجا اطمینان می دهد که پوسته با رشته کامل سرو کار داشته، شامل فاصله ها، بعنوان یک شی منفرد)

با قرار دادی خطوط پیچیده فرمان می توانند در این روش بکار روند. سر انجام، در اینجا یک برنامه ساده چگونگی جای دادن دستورات تجزیه شده در تابع مجزا که هدف آن خط فرمان و تنظیم مقادیر نشانوندهایش می باشد را نشان می دهد.

```

/*****/
/*                               */
/*  Getting arguments from      */
/*                               */
/*  the Command Line           */
/*                               */
/*****/

/* Steve McMillan    */
/* Written: Winter 1995 */

```

```

#include <stdio.h>
#include <stdlib.h>

```

```
void get_args(int argc, char** argv, int* a_value, float* b_value)
```

```

{
    int i;

    /* Start at i = 1 to skip the command name. */

    for (i = 1; i < argc; i++) {

        /* Check for a switch (leading "-"). */

        if (argv[i][0] == '-') {

            /* Use the next character to decide what to do. */

            switch (argv[i][1]) {

                case 'a':    *_a_value = atoi(argv[++i]);
                            break;

                case 'b':    *_b_value = atof(argv[++i]);
                            break;

                default:     fprintf(stderr,
                                "Unknown switch %s\n", argv[i]);
            }
        }
    }
}

main(int argc, char** argv)
{
    /* Set defaults for all parameters: */

    int a = 0;
    float b = 0.0;

    get_args(argc, argv, &a, &b);

    printf("a = %d\n", a);
    printf("b = %f\n", b);
}

```

۱۲- واسط های گرافیکی: جعبه های محاوره

فرض کنید شما نمی خواهید با تفسیر خط فرمان مواجه شوید، ولی هنوز می خواهید برنامه شما قادر به تغییر مقادیر متغیرهای مشخص

در یک روش محاوره ای باشد. شما می توانید برنامه را در یک سری خطوط `printf/scanf` برای امتحان کاربرد در مورد اولویت هایشان ساده کنید.

```
printf("Please enter the value of n: ");
scanf("%d", &n);
```

```
printf("Please enter the value of x: ");
scanf("%f", &x);
```

و مانند آن، البته این کار خوبی نخواهد بود اگر برنامه شما به عنوان بخشی از خط لوله (pipeline) یا پایپ لاین استفاده می شود. برای مثال استفاده از برنامه گرافیکی `plot - data`، چون پرسشها و پاسخها در جریان داده ها مخلوط می شوند. رفتار درست استفاده از یک واسط گرافیکی ساده است که یک جعبه محاوره تولید کند، اختیار پارمترهای گوناگون کلید در برنامه را به شما می دهد. بسته های گرافیکی ما تعدادی از ابزارهای آسان برای استفاده از ساختارها و جعبه های مشابه را فراهم می کند. ساده ترین راه برای ست کردن متغیر صحیح `n` و متغیر شناور `x` (یعنی برای اجرای یک اثر مانند خطوط کد بالا) استفاده از جعبه محاوره مانند زیر می باشد:

```
/* Simple program to illustrate use of a dialog box */

main()
{
    /* Define default values: */

    int n = 0;
    float x = 0.0;

    /* Define contents of dialog window */

    create_int_dialog_entry("n", &n);
    create_float_dialog_entry("x", &x);

    /* Create window with name "Setup" and top-left corner at (0,0) */

    set_up_dialog("Setup", 0, 0);

    /* Display the window and read the results */

    read_dialog_window();
```

```
/* Print out the new values */  
  
printf("n = %d, x = %f\n", n, x);  
}
```

این برنامه را با استفاده از نام مستعار `cgfx` برای پیوند در همه کتابخانه های لازم ترجمه کنید. دو خط `create` مدخلهای جعبه و متغیرهای وابسته به آنها را تعریف می کند. `set-up – dialog` جعبه ها را نام گذاری کرده و مکان آنها را تعریف می کند. سر انجام، `read-dialog-window` پنجره را بالا پرانده (باز می کند) و به شما اجازه تغییر مقادیر متغیرها را می دهد. هنگامیکه برنامه اجرا می شود، شما جعبه ای مشابه زیر را خواهید دید.

شکل

اعداد نشان داده شده را تغییر داده، `OK` را فشار دهید. (یا فقط کلید `Enter` را بزنید.) تغییرات انجام می شود. همه آنها به همین صورت است. بیشترین فایده این مورد اینست که آن مستقل از روند داده ها از طریق `stdin/stdout` عمل می کند. بطور کلی، شما می توانید حتی عملیات هر مرحله را در پایپ لاین بسیاری از فرمانهای زنجیره ای با استفاده از یک جعبه محاوره برای هر کدام کنترل کنید.

فصل هفتم: ورودی خروجی استاندارد

تاکنون ما از ابزار موجود برای ساخت ابزار جدید استفاده کردیم، اما محدود به این هستیم که به طور معقول با `sed` و `awk` چه می توان انجام داد. در این فصل ما قصد داریم، برخی از برنامه های ساده را به زبان برنامه نویسی `C` بنویسیم. فلسفه اصلی ساختن چیزها این است که همکاری تا نفوذ بر بحث و بررسی و طرح برنامه ها ادامه یابد - ما می خواهیم ابزاری را بوجود آوریم که دیگران بتوانند استفاده کنند و به آنها اعتماد داشته باشند. در هر مورد، ما سعی می کنیم که یک استراتژی تحقق مناسب را نیز انجام دهیم: آغاز با حداقل میزان، که کار مفیدی را انجام دهد، سپس افزودن ویژگی ها و انتخابها (فقط) اگر مورد نیاز باشند.

دلایل خوبی برای نوشتن برنامه های جدید از حافظه وجود دارند. می تواند به این دلیل باشد که مشکل موجود نمی تواند فقط با برنامه های موجود حل شود. این مورد اغلب زمانی صحیح است که برنامه باید به فایل های غیر مبتنی پردازد. برای مثال - اکثر برنامه هایی که ما تاکنون نشان داده ایم. حقیقتاً فقط بر روی اطلاعات متنی به خوبی کار می کنند. و یا ممکن است دستیابی به توان یا سودمندی مناسب با `shell` و سایر ابزارهای دارای هدف کلی، بسیار دشوار باشد. در چنین مواردی، یک نسخه `shell` می تواند برای اختصاص به تعریف و واسط کاربر از یک برنامه خوب باشد `0` و اگر به خوبی کار کند، نیازی به انجام مجدد آن نیست). برنامه `zap` از فصل آخر. مثال خوبی است: این برنامه فقط به چند دقیقه زمان برای نوشتن اولین نسخه در `shell` نیاز دارد و نسخه نهایی دارای یک واسط کاربر مناسب می باشد، اما بسیار کند می باشد.

ما به زبان `C` خواهیم نوشت، چون `C` یک زبان استاندارد از سیستم های یونیکس می باشد - کرنل و همه برنامه های کاربر به زبان `C` نوشته می شوند - و حقیقتاً از هیچ زبان دیگری، تقریباً به این خوبی حمایت نمی شود. ما فرض می کنیم که شما زبان `C` را می شناسید، حداقل تا اندازه ای که آن را بخوانید. اگر این اینگونه نیست، زبان برنامه نویسی `C` را که نوشته `W.B.` کرنیگمان و `M.D.` ریتچای می باشد. مطالعه کنید (پرنتیک - مال 1978).

ما همچنین از کتابخانه استاندارد `I/O` استفاده خواهیم کرد، یک مجموعه از زیربرنامه هایی که خدمات موثر و قابل انتقال سیستم و `I/O` را برای برنامه های `C` فراهم می کند.

کتابخانه استاندارد `I/O` در همه سیستم های غیر یونیکس که از `C` حمایت می کنند، در دسترس می باشد، بنابراین، برنامه هایی که برهم کنش های سیستم خود را برای تسهیلات آن محدود می کنند، می توانند به سهولت انتقال یابند.

مثالهایی که ما برای این فصل انتخاب کرده ایم، دارای یک ویژگی مشترک می باشند: آنها ابزار کوچکی هستند که ما از آنها به طور منظم استفاده می کنیم، اما بخشی از ویرایش هفتم نبودند. اگر سیستم شما دارای برنامه های مشابه باشد، شما می توانید از آن به عنوان یک آگاه کننده برای مقایسه طرحها استفاده کنید. و اگر آنها برای شما جدید باشند، شما آنها را به عنوان ابزاری مفید می یابید. در هر حال، آنها باید در روشن کردن این نکته که هیچ سیستمی کامل نیست و اینکه اغلب توسعه دادن چیزها و فائق آمدن بر کمبودها با تلاش نه چندان زیاد آسان است، به شما کمک کنند.

1-7 - ورودی و خروجی استاندارد: `vis`

بسیاری از برنامه ها، فقط یک ورودی را می خوانند و یک خروجی را می نویسند: برای چنین برنامه هایی، `I/O` که فقط از ورودی استاندارد و خروجی استاندارد استفاده می کند، می تواند کاملاً مناسب باشد و تقریباً همیشه برای شروع کار، کافی است.

اکنون این مورد را با یک برنامه با عنوان `vis` شرح می دهیم که ورودی استاندارد خود را برای خروجی استاندارد خود کپی می کند، به

استثنای اینکه این برنامه ، همه کاراکترهای غیرچاپی را با چاپ کردن آنها به صورت 1 nnn مرئی می سازد، nnn، ارزش هشت هشتی کاراکتر می باشد. Vis برای آشکارسازی کاراکترهای عجیب یا کاراکترهای ناخواسته که ممکن است درون فایلها نفوذ کرده باشند، بی ارزش می باشد. برای نمونه، vis ، هر پس برد را به صورت 1010 پرینت می کند که ارزش هشت هشتی کاراکتر پس برد می باشد :

```
cat x $
```

```
a b c
$ vis sx
a b c \ 0101010101010 ---
$
```

برای پویش فایلهای متعدد با این نسخه ابتدایی از vis ، شما می توانید از cat برای جمع کردن فایلها استفاده کنید :

```
$ cat file1 file 2 ... ?| vis
```

```
.....
```

```
$ cat file1 file 2 ... | vis | grep ' \\'
```

و در نتیجه از یادگیری چگونگی دستیابی به فایلها از یک برنامه اجتناب کنید.

در ضمن ، باید مشاهده شود که شما می توانید این کار را با sed انجام دهید. چون فرمان 'l' ، کاراکترهای غیر قابلچاپ را در یک شکل قابل فهم آشکار می سازد :

```
$ sed - n 1 x
a b c <<<---
$
```

خروجی sed. احتمالاً واضح تر از vis می باشد. اما هرگز برای فایلهای غیرمتنی معنی نمی دهد:

```
$ sed - n | /usr/you/bin
$ Nothing at all !
```

(این برنامه بر روی یک PDP – 11 قرار دارد ؛ روی یک سیستم VAX و sed خاموش می شود، احتمالاً به دلیل اینکه ورودی شبیه یک سطر خیلی طولانی از متن به نظر می رسد).

بنابراین sed نامناسب است و ما مجبور هستیم که یک برنامه جدید بنویسیم. ساده ترین برنامه های کار ورودی و خروجی، getchar و putchar نامیده می شوند. هر فراخوانی برای getchar، کاراکتر بعدی را از ورودی استاندارد می گیرد، که ممکن است یک فایل یا یک لوله و یا یک پایانه (پیش فرض) باشد - برنامه، نمی داند که کدام یک می باشد - متشابهاً C (putchar) ، کاراکتر C را روی خروجی استاندارد قرار می دهد که بر طبق پیش فرض، نیز یک پایانه می باشد.

تابع (3 print f) ، تبدیل فرمت خروجی را انجام می دهد. فراخوانی های printf و putchar می توانند در هر ترتیبی ، یکی در میان شوند؛ خروجی، در ترتیب فراخوانی ها آشکار می شود. یک تابع دیگر با عنوان (3 scan f) برای تبدیل فرمت ورودی وجود دارد؛ این تابع ورودی استاندارد را می خواند و آنها را به رشته ها، اعداد و غیره در جایی که لازم می باشد ، تجزیه می کند. فراخوانی ها برای scanf و getchar نیز می توانند، با هم مخلوط شوند.

در این جا اولین نسخه vis عبارت است از :

```
/ * vis : make funny characters visible (version \) * /
# include <stdio .hs
# include <stype.h

main ( )
{
```

```

int c ;
while (( = getchar ( ) ) != EOF)
    if (isascii ( c ) $$
        ( isprint ( c ) || c == '\n' || c == '\t' || ' ' ))
        put char ( c ) ;
    else
        print f ( " \\% ' 30 " , c ) ;
exit ( 0 ) ;
}

```

get char ، بایت بعدی یا ارزش EOF را از ورودی، بر می گرداند، زمانی که به پایان فایل (یا به یک اشتباه) برخورد می کند. در ضمن، EOF یک بایت از فایل نمی باشد؛ بحث و بررسی مربوط به پایان فایل را در فصل 2 به خاطر بیاورید. ارزش EOF به گونه ای متفاوت از هر ارزشی که در یک بایت تنها رخ می دهد، تضمین می شود، بنابراین ، میتواند متمایز از داده های واقعی باشد ؛ C به صورت int اعلان می شود و نه char. بنابراین، به اندازه کافی برای حفظ ارزش EOF بزرگ می باشد . سطر

```
# include < stdio.h >
```

باید در آغاز هر فایل مبدأ ظاهر شود. این سطر باعث می شود که کامپایلر، یک فایل عنوان (usr / include / stdio.h /) از برنامه های کاری استاندارد و سمبل هایی را بخواند که شامل تعریف EOF می باشند. ما از > stdio.h به عنوان یک صورت مختصر برای اسم کامل فایل در متن استفاده می کنیم.

فایل < ctype . h > . فایل عنوان دیگری در / usr / include می باشد که آزمونهای وابسته به ماشین را برای تعیین ویژگیهای کاراکترها تعریف می کن. ما در اینجا از isascii و isprint ، برای تعیین اینکه آیا کاراکتر ورودی ، Ascll (یعنی دارای ارزش کمتر از 0200) و قابل چاپ می باشد یا نه، استفاده کردیم؛ سایر آزمونها در جدول 1 - 6 فهرست وار بیان می شوند. توجه داشته باشید که سطر جدید جدول بندی و فاصله ، توسط تعاریف موجود در < ctype . b > قابل چاپ نیستند.

فراخوانی exit در انتهای vis، برای انجام کار برنامه به طور صحیح، ضروری نیست، اما اطمینان می دهد که هر شماره گیرنده از برنامه با یک وضعیت خروج نرمال (به صورت قراردادی صفر) از برنامه مواجه می شود زمانی که کامل می شود یک روش دیگر برای بازگرداندن وضعیت، خارج شدن از main با return 0 می باشد ارزش بازگشت از main، وضعیت خروج برنامه می باشد. اگر return یا exit به صورت آشکار وجود نداشته باشد، وضعیت خروج ، غیر قابل پیش بینی است.

برای کامپایل کردن یک برنامه C ، مبدأ را در فایلی قرار دهید که نام آن با C. پایان می یابد، مانند vix.c ، آن را با cc کامپایل کنید، سپس نتیجه را اجرا کنید که به این صورت کامپایلر در یک فایل با عنوان a.out باقی می ماند. ('a' برای اسمبلر می باشد) :

```

$ cc vis . c
$ a .out
hello world ctl-g
helo world \ 007
ctl - d
$

```

در حالت عادی شما باید a.out را مجدداً نامگذاری کنید. زمانی که کار می کند و یا از انتخاب -o از cc برای انجام آن به طور مستقیم استفاده کنید :


```
$ cc -o vis vis.c
```

```
output in vis not a.out
```

خروجی در vis، a.out نمی باشد.

تمرین 1 - 6. ما تصمیم گرفتیم که جدول بندی ها باید کنار گذاشته شوند، به جای اینکه به صورت \t یا \r یا \011 آشکار شوند، چون استفاده مهم ما از vis، پیدا کردن کاراکترهای حقیقتاً نامبهم می باشد. جدول بندیها، غیرتصویریها، فاصلهها در انتهای سطرها و غیره. Vis را به گونه ای تغییر دهید که کاراکترهایی شبیه جدول بندی، پس کج خط، پس برد، کاغذ - خورد و غیره، در اجراهای قراردادی c خود \f، \b، \، \، t و غیره. پرینت شوند و در نتیجه فاصله های خالی در انتهای سطرها علامت گذاری شوند. آیا شما میتوانید چنین کاری را بدون ابهام انجام دهید؟ طرح خود را با \$ sed -n 1 مقایسه کنید.

تمرین 2 - 6. Vis را به گونه ای تغییر دهید که سطرهای بلند را در طول معقول تا کند. این کار چگونه با خروجی غیر مبهم مورد نیاز در تمرین قبلی برهم کنش می کند؟

2 - 6. آرگومانهای برنامه : نسخه 2 vis

زمانی که یک برنامه c اجرا می شود. آرگومانهای سطر فرمان، به عنوان یک شماره argc و یا آرایه argv از اشاره گرها، برای رشته های کاراکتری که شامل آرگومانها می باشند در دسترس تابع main قرار می گیرند. بر طبق قرارداد، [0] argv خودش اسم فرمان است، بنابراین، argc همیشه بزرگتر از صفر است؛ آرگومانهای مفید [1] argv ... [argc - 1] هستند. به خاطر داشته باشید که جهت دهی مجدد با < و >، توسط شل انجام می شود و نه توسط برنامه های منفرد، بنابراین جهت دهی مجدد تأثیری بر تعداد آرگومانهای مشاهده شده توسط برنامه ندارد.

برای شروع عملکرد یا آرگومان، با افزودن یک آرگومان انتخابی vis را تغییر می دهیم : vis-s، همه کاراکترهای غیر چاپی را خارج می کند به جای اینکه آنها را به طور مداوم نمایش دهد. این انتخاب، برای پاک کردن فایلها از سایر سیستم ها مناسب است، برای مثال، سیستم هایی که از CRLF (سطر خورد و سطر جدید) به جای یک سطر جدید به سطرها پایان می دهند.

```
/* کاراکترهای جالب را آشکار می سازد (نسخه 2 : vis) */
```

```
# include <stdio.h>
```

```
# include <ctype.h>
```

```
main (argc , argv)
    int argc ;
    char * argv [ ] ;
```

```
{
int c , strip = 0 ;
```

```
if argc > 1 & $ strcmp (argv [ 1 ] , "- s") == 0
```

```
strip = 1 ;
```

```
while (( c = getchar (1) ) != EOF )
```

```

if (isascii(c) &&
    (isprint(c) || c == '\n' || c == '\t' || c == ' '))
    putchar (c);
else if (!strip)
    printf ("\\% 30", c);
exit (0);
}

```

جدول 6 - 1 - درشت - دستورالعملهای آزمون کاراکتر < ctype.h >

(isalpha (c	الفبایی : A-Z ، a-z
(isupper (c	مورد بالای : A - Z
(islower (c	مورد پائین : a - z
(isdigit (c	رقم : 0 - 9
(isxdigit (c	رقم شانزده شانزدهی : 0 - 9 a - f A - F
(isalnum (c	الفبایی یا رقم
(isspace (c	فاصله، جدول بندی، سطر جدید، جدول بندی عمودی، کاغذ - خورد ، بازگشت
(ispunct (c	غیر الفبایی - عددی، یا کنترل یا فاصله
(isprint (c	قابل چاپ : هر تصویری
(iscntrl (c	کاراکتر کنترل : 0 <= c <= 0177
(isascii (c	کاراکتر : Ascii : 0 <= c <= 0177

argv ، یک اشاره گر به آرایه ای است که عناصر منفرد آن اشاره گرهایی به آرایه های کاراکترها می باشند؛ هر آرایه ، یک کاراکتر NUL (از ' '10 Ascii) پایان می پذیرد ، بنابراین می تواند به صورت یک رشته عمل کند. این نسخه از vis ، با کنترل اینکه آیا یک آرگومان وجود دارد یا نه و آیا s - می باشد یا نه ، آغاز می کند.

(آرگومانهای غیر معتبر نادیده گرفته می شوند). تابع (3) strcmp ، دو رشته را مقایسه می کند، و اگر آنها شبیه به هم باشند، صفر را بر می گرداند.

جدول 2 - 6 . یک مجموعه از تابعها را با استفاده کلی و به کارگیری رشته، فهرست بندی می کند، که یکی از آنها strcmp می باد. معمولاً استفاده از این تابعها، به جای نوشتن تابعهای خود، بهترین راه می باشد، چون آنها استاندارد هستند، آنها بدون خطا هستند و آنها اغلب سریعتر از چیزی هستند که شما خودتان می توانید بنویسید، چون آنها برای ماشین های خاص ، مطلوب شده اند (گاهی اوقات ، با نوشته شدن در زمان اسمبلی مطلوب می باشند).

تمرین 3 - 6. آرگومان s را به گونه‌ای تغییر دهید که $vis - sn$ فقط رشته‌های n یا کاراکترهای پیاپی قابل چاپ را پرینت کند، کاراکترهای غیرچاپی را حذف کند و توالیهای کاراکترهای قابل چاپ را کوتاه کند. چنین چیزی برای جدا کردن بخشهای بعدی فایلها غیر متنی مانند برنامه‌های قابل اجرا ارزشمند می‌باشد. بر فراز نسخه‌های سیستم، یک برنامه `string` را تهیه می‌کنند که چنین کاری را انجام می‌دهد. آیا داشتن یک برنامه مجزا یا یک آرگومان برای `vis` بهتر است.

تمرین 4 - 6. در دسترس بودن کد مبدأ c یکی از نقاط قوت سیستم یونیکس می‌باشد - کد، راه‌حلهای ظریف را برای بسیاری از مشکلات برنامه‌نویسی شرح می‌دهد. موازنه بین قابلیت خوانده شدن مبدأ c و بهینه سازی اتفاقی دست آمده از نوشتن مجدد به زبان اسمبلی را شرح دهید.

جدول 2 - 6 - تابعهای استاندارد رشته

<code>(stract (s , t</code>	رشته t را به رشته s پیوست می‌دهد؛ s را بر می‌گرداند
<code>(strnact (s , t , n</code>	اکثر کاراکترهای n از t را به s ضمیمه می‌کند
<code>(strcpy (s , t</code>	t را برای s کپی می‌کند؛ s را بر می‌گرداند
<code>(strncpy (s , t , n</code>	دقیقاً کاراکترهای n را کپی می‌کند؛ اگر لازم باشد <code>pad</code> را خالی می‌کند
<code>(strcmp (s , t</code>	s و t را مقایسه می‌کند، < 0 ، $= 0$ و > 0 را برای دو $> = =$ ، برمی‌گرداند
<code>(strncmp (s , t , n</code>	اکثر کاراکترهای n را مقایسه می‌کند
<code>(strlen (s</code>	طول s را بر می‌گرداند
<code>(strchr (s , c</code>	اشاره‌گر را به اولین c در s بر می‌گرداند، اگر <code>Null</code> نباشد
<code>(strrchr (s , c</code>	اشاره‌گر را به آخرین c در s بر می‌گرداند، اگر <code>null</code> نباشد
<code>(atoi (s</code>	اینها، <code>index</code> و <code>rindex</code> بر روی سیستم‌های قبلی هستند
<code>(atof (s</code>	ارزش عدد صحیح s را بر می‌گرداند
<code>(malloc (n</code>	ارزش مییز شناور s را بر می‌گرداند؛
<code>(calloc (n,m</code>	مستلزم اعلان <code>(atof (s</code> دوگانه می‌باشد.
<code>(free (p</code>	اشاره‌گر را به بایتهای n از حافظه برمی‌گرداند، اگر <code>null</code> نتواند
	اشاره‌گر را به بایتهای $n \times m$ برمی‌گرداند، برای صفر تنظیم می‌کند
	اگر <code>null</code> نتواند. <code>Mallo</code> و <code>char</code> ، <code>* calloc</code> را بر می‌گرداند
	حافظه آزاد تخصیص یافته توسط <code>malloc</code> یا <code>calloc</code>

3 - 6 - دستیابی به فایل : نسخه 3 vis

دو نسخه اول `vis`، ورودی استاندارد را می‌خوانند و خروجی استاندارد را می‌نویسند، و هر دو از شل منتج می‌شوند. مرحله بعدی، تغییر `vis` برای دستیابی به فایلها از طریق اسامی آنها است. بنابراین

\$ file 1 file 2 vis ...

فایل‌های نامگذاری شده را به جای ورودی استاندارد، پویش می‌کند. اگر چه آرگومانهای اسم فایل شاید وجود نداشته باشند اما ما می‌خواهیم vis، ورودی استاندارد خود را بخواند. سوال این است که چگونه فایلها را برای خوانده شدن مرتب کنیم - یعنی اینکه، چگونه اسامی فایلها را به بیان‌های I/O متصل کنیم که دقیقاً داده‌ها را بخوانند.

قواعد ساده هستند. قبل از اینکه یک فایل بتواند خوانده یا نوشته شود، باید توسط تابع کتابخانه استاندارد یعنی Fopen باز شود. Fopen، یک اسم فایل را می‌گیرد (همانند temp یا passwd / etc)، سیستم‌داری و انتقال را با کرنل انجام می‌دهد و یک اسم درونی را که باید در عملکردهای بعدی روی فایل استفاده شود، بر می‌گرداند.

این نام درونی، دقیقاً یک اشاره‌گر می‌باشد که اشاره‌گر فایل نام دارد، برای ساختاری که شامل اطلاعاتی در مورد فایل می‌باشد، مانند مکان یک میانگیر، موقعیت کاراکتر فعلی در میانگیر، و اینکه آیا یک فایل خوانده یا نوشته می‌شود و مواردی از این قبیل. یکی از تعاریف بدست آمده از طریق <stdio.h>، برای یک ساختار می‌باشد که FILE نام دارد. اعلان برای یک اشاره‌گر فایل عبارت است از

```
FILE * fp ;
```

این اعلان بیان می‌کند که fp یک اشاره‌گر به یک FILE می‌باشد. Fopen یک اشاره‌گر را به یک FILE بر می‌گرداند؛ یک نوع اعلان برای fopen در <stdio.h> وجود دارد.

فراخوانی واقعی برای fopen در یک برنامه عبارت است از

```
chr * name , * mode ;
```

اولین آرگومان Fopen، نام فایل، به عنوان یک رشته کاراکتر می‌باشد. دومین آرگومان، نیز یک رشته کاراکتر می‌باشد و نشان می‌دهد که شما چگونه از فایل استفاده می‌کنید و روشهای مجاز، خواندن ("r")، نوشتن ("w") یا ضمیمه کردن ("a") می‌باشند.

اگر فایلی که شما برای نوشتن یا ضمیمه کردن باز می‌کنید، وجود نداشته باشد، اگر امکان‌پذیر باشد، بوجود می‌آید. باز کردن یک فایل موجود برای نوشتن باعث می‌شود که محتوای قبلی، حذف شوند. تلاش برای خواندن فایلی که وجود ندارد، یک خطا است و همانند تلاش برای خواندن یا نوشتن یک فایل می‌باشد زمانی که شما اجازه ندارید. اگر خطایی وجود داشته باشد، fopen، ارزش اشاره‌گر بی‌اعتبار را به صورت null (تهی) بر می‌گرداند. (که معمولاً به صورت (* char) در <stdio.h> تعریف می‌شود).

مورد بعدی مورد نیاز، روش خواندن یا نوشتن فایل می‌باشد زمانی که باز می‌شود امکانات متعددی برای این مورد وجود دارند، که از آنها getc و putc، ساده‌ترین هستند. کاراکتر بعدی را از فایل می‌گیرد.

```
(C = getc (fp
```

کاراکتری بعدی از فایل را که به عنوان fp به آن استناد می‌شود در c قرار می‌دهد؛ و EOF را برمی‌گرداند، زمانی که به انتهای فایل می‌رسد. Putc، شبیه به getc می‌باشد :

```
(putc (c , fp
```

کاراکتر c را بر روی فایل fp قرار می‌دهد و c را برمی‌گرداند. getc و EOF، putc را برمی‌گرداند، اگر یک اشتباه رخ دهد. زمانی که یک برنامه آغاز می‌شود، سه فایل باز می‌شوند و اشاره‌گرهای فایل برای آنها تهیه می‌شوند. این فایلها، ورودی استاندارد، خروجی استاندارد و خروجی خطای استاندارد هستند؛ اشاره‌گرهای فایل مربوطه، stdout، stdin و stderr نامیده می‌شوند. این اشاره‌گرهای فایل در

<stdio.h>، اعلان می‌شوند؛ آنها می‌توانند در هر جایی استفاده شوند که به همان منظور تایپ FILE*، می‌تواند استفاده شود. اما

آنها ثابت‌ها هستند و نه متغیرها، بنابراین شما نمی‌توانید، به آنها استناد کنید.

getchar () ، همانند (stdin) getc می‌باشد و putchar (c) همانند (stdout) putc می‌باشد. در حقیقت، هر چهار تابع، به عنوان

<stdio.h> تعریف می‌شوند، چون آنها با اجتناب از

یک یک تابع برای هر کاراکتر، سریعتر اجرا می‌کنند. به جدول 3 - 6 برای سایر تعاریف در

<stdio.h> مراجعه کنید.

با برخی از مقدمات خارج از روش، اکنون ما می‌توانیم سومین نسخه از vic را بنویسیم. اگر آرگومانهای سطر فرمان وجود داشته

باشند، آنها به ترتیب پردازش می‌شوند. اگر آرگومانها وجود نداشته باشند، ورودی استاندارد پردازش می‌شود.

```
/* vis : make funny characters visible (version 3) */
```

```
# include <stdio.h>
# include <type.h>
int strip = 0; /* 1 => discard special characters */
main (argc, argv)
    int argc;
    char * argv[];
{
    int i;
    FILE * fp;
    while (argc > 1 && argv[1][0] == '-') {
        switch (argv[1][1]) {
            case 's': /* -s; strip funny chars */
                strip = 1;
                break;
            default:
                fprintf(stderr, "%s: unknown arg %s\n",
                    argv[0], argv[1]);
                exit(1);
        }
        argc--;
        argv++;
    }
    if (argc == 1)
        vis(stdin);
    else
        for (i = 1; i < argc; i++)
            if ((fp = fopen(argv[i], "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n",
                    argv[0], argv[i]);
                exit(1);
            } else {
                vis(fp);
                fclose(fp);
            }
}
```

```
exit (0) ;
}
```

این رمز ، بر این قرارداد تکیه دارد که آرگومانهای اختیاری در ابتدا می آیند. پس از اینکه هر آرگومان اختیاری، پردازش می شود، `argc` و `argv` تنظیم می شوند، در نتیجه مابقی برنامه، وابسته به حضور آن آرگومان نمی باشد. اگر چه `vis`، فقط انتخاب تنها را تشخیص می دهد، اما ما رمز را به صورت یک حلقه برای نشان دادن راه برای تشخیص پردازش آرگومان نوشتیم. در فصل 1 ما روش نامقطعی را که بر طبق آن برنامه های یونیکس از آرگومانهای انتخابی استفاده می کردند توضیح دادیم. جدای از یک میل برای بی نظمی، یکی از دلایل آن، این است که نوشتن رمز برای استفاده از بخش آرگومان برای هر تغییر، به طور بدیهی آسان نمی باشد. تابع `get opt (3)` که بر روی برخی از سیستم ها یافت می شود. یک تلاش برای توجیه کردن موقعیت می باشد؛ شما باید قبل از نوشتن برنامه خود، به تحقق در مورد آن پردازید.

برنامه کار `ris`، یک فایل منفرد را پرینت می کند :

```
vis (fp) /* make chars visible in FILE * fp */
FILE * Fp ;
{
    int c ;
    while (( c = get c (fp)  != EOF)
           if (isacii(c) $$
               ( isprint (c) ||  c == '\n ||  c == '\t' / |  c == ' ' ))
               put cahr ( c ) ;
           elw if ( ! strip)
               printf ( " \\ % 30 " , c ) ;
}
```

تابع `fprintf` مشابه با `printf` می باشد، به جز برای یک آرگومان نشان گر فایل، که فایلی را که باید نوشته شود مشخص می کند. تابع `fclose`، ارتباط بین اشاره گر فایل و نام خروجی را که توسط `Fopen` تعیین شد، از بین می برد و اشاره گر فایل را برای فایل دیگر آزاد می کند. چون یک محدودیت بر روی تعداد فایل هایی (حدود 20) وجود دارد که یک برنامه می تواند به طور همزمان باز کند، در نتیجه، بهترین کار، آزاد کردن فایلها در زمانی است که دیگر به آنها نیازی نیست. در حالت عادی، خروجی تهیه شده با هر کدام از تابعهای کتابخانه استاندارد، مانند `putc`، `printf` و غیره، میانگیر می سازد، و در نتیجه می تواند به منظور موثر بودن، در `chunk` های بزرگ نوشته شود. (در اینجا استثنا، خروجی برای یک پایانه می باشد، که معمولاً در همان زمانی که تولید می شود، نوشته می شود و یا حداقل، زمانی که سطر جدید پرینت می شود). فراخوانی `fclose` بر روی یک فایل خروجی نیز هرگونه خروجی میانگیر شده را خارج می کند. `Fclose` نیز به طور خودکار برای هر فایل باز، فراخوان می شود زمانی که یک برنامه، `exit` را فرا می خواند و یا از `main` برمی گردد.

`Stderr`، به کل برنامه تخصیص داده می شود به همان روشی که `stdin` و `stdout` تخصیص داده می شوند. خروجی نوشته شده بر روی `stderr` بر روی پایانه کاربر ظاهر می شود، حتی اگر خروجی استاندارد مجدداً جهت یابی شود. `Vis`، تشخیص خود را بر روی `stderr` به جای `stdout` می نویسد، بنابراین اگر یکی از فایلها نتواند به دلیلی دستیابی شود، پیغام راه خود را به سمت پایانه کاربر به جای ناپدید شدن درون خط لوله ای یا درون یک فایل خروجی، پیدا می کند. (خطای استاندارد، اندکی پس از لوله ها، اختراع شد، پس از اینکه

پیغامهای خطا، شروع به ناپدید شدن درون خطوط لوله‌ای کردند).

تا حدوی اختیاری، ما تصمیم گرفتیم که `vis` از سیستم خارج میشود اگر نتواند یک فایل ورودی را باز کند؛ چنین چیزی برای برنامه‌ای که اغلب به صورت برهم‌کنشی استفاده می‌شود و با یک فایل ورودی تنها می‌باشد، معقول است. شما می‌توانید برای طرح دیگر به بحث و بررسی بپردازد.

تمرین 5 - 6. یک برنامه `printable` بنویسید که نام هر فایل آرگومان را که شامل فقط کاراکترهای قابل چاپ می‌باشد، پرینت کند؛ اگر فایل شامل کاراکتر غیر قابل چاپ باشد، اسم آن پرینت نشود. `Printable` در موقعیتهایی مانند این، مفید است:

```
$ pr 'printable *'|pr
```

انتخاب 7 - را برای معکوس کردن حس آزمون، همانند `grep` وارد کنید.

`Printable` چه باید انجام دهد اگر آرگومانهای اسم فایل وجود نداشته باشند؟ `printable` به چه وضعیتی باید برگردد؟

جدول 3 - 6 - برخی از تعاریف `<stdio.h>`

<code>stdin</code>	ورودی استاندارد
<code>stdout</code>	خروجی استاندارد
<code>stderr</code>	خطای استاندارد
<code>EOF</code>	انتهای فایل : در حالت عادی 1 -
<code>NULL</code>	نشانگر غیرمعتبر ؛ در حالت عادی
<code>FILE</code>	استفاده شده برای اعلان اشاره‌گرهای فایل
<code>BUFSIZ</code>	اندازه نرمال میانگرم I / O (اغلب 512 یا 1024)
<code>(getc (fp</code>	یک کاراکتر از جریان FP باز می‌گرداند
<code>(getc (stdin</code>	<code>getchar ()</code>
<code>(putc (c , fp</code>	کاراکتر <code>c</code> را بر روی رشته <code>fp</code> قرار می‌دهد
<code>(putc (c , stdout</code>	<code>put char (c)</code>
<code>(feof (fp</code>	صفر نمی‌باشد زمانی که انتهای فایل بر روی رشته <code>fp</code> قرار دارد
<code>(ferror (fp</code>	صفر نمی‌باشد زمانی که هر گونه خطا بر روی رشته <code>fp</code> قرار دارد
<code>(fileno (fp</code>	توصیف‌گر فایل برای رشته <code>fp</code> ؛ به فصل 7 مراجعه کنید

4 - 6 - یک چاپگر نمایشی در یک زمان : P

تاکنون ما از `cat` برای بررسی فایلها استفاده کردیم. اما اگر یک فایل بلند باشد، و اگر شما توسط یک اتصال دارای سرعت بالا به سیستم خود متصل شده باشید، `cat`، خروجی را به گونه‌ای تولید می‌کند که بی‌نهایت سریع خوانده شود، حتی اگر شما با `ctl-s` و `ctl-q` سریع باشید.

به طور آشکارا، باید یک برنامه برای پرینت یک فایل در `chunk`های کوچک قابل کنترل وجود داشته باشد، اما یک برنامه استاندارد وجود ندارد، احتمالاً به دلیل اینکه سیستم اولیه یونیکس، در روزهای پایانه‌های نسخه سخت- (کاغذ) و سطرهای ارتباطی کند نوشته

شده است. بنابراین، مثال بعدی ما برنامه‌ای با عنوان P می‌باشد که یک فایل پرده پر را در یک زمان پرینت می‌کند و منتظر جواب از کاربر، پس از هر پرده نمایش و قبل از رفتن به پرینت بعدی، می‌ماند. ("P"، یک اسم کوتاه و مناسب برای برنامه‌ای است که ما آن را بسیار زیاد استفاده می‌کنیم).

همانند سایر برنامه‌ها، p از فایل‌های نامگذاری شده به عنوان آرگومانها و یا از ورودی استاندارد خود می‌خواند :

```
$ p vis . c
...
$ grep '#define' * . [ ch ] | p
...
$
```

این برنامه ، به بهترین نحو در زبان C نوشته می‌شود، چون این برنامه در زبان C آسان است و در زبانهای دیگر دشوار است؛ ابزار استاندارد، در ترکیب کردن ورودی از یک فایل یا لوله با ورودی پایانه، مناسب نیستند.

طرح اساسی و بی‌پیرایه ، چاپ کردن ورودی در قطعه‌های کوچک می‌باشد. یک اندازه مناسب برای قطعه، 22 سطر می‌باشد: که این اندازه، اندکی کمتر از پرده نمایش 24 سطری اکثر پایانه‌های تصویری می‌باشد و $\frac{1}{3}$ از یک صفحه استاندارد 66 سطری می‌باشد. یک روش ساده برای اینکه p به کاربر پیامواره بدهد، چاپ کردن آخرین سطر جدید از هر قطعه 22 سطری می‌باشد. بنابراین، مکان نما در انتهای راست هر سطر به جای حاشیه چپ، مکث می‌کند. زمانی که کاربر کلید RETURN را فشار می‌دهد. سطر جدید مفقود را ذخیره می‌کند و بنابراین باعث می‌شود که سطر بعدی در مکان صحیح خود آشکار شود. اگر کاربر، ctl-d یا g را در انتهای یک پرده نمایش تایپ کند، p بوجود می‌آید.

ما عملکردهای خاصی را برای سطرهای طولانی انجام نمی‌دهیم ما همچنین درخصوص فایل‌های متعدد نگران نیستیم: ما صرفاً بدون توضیح از یک فایل به فایل دیگر جست می‌زند، به این صورت رفتار

```
... p filenames $
```

شبيه به رفتار \$ | p ... filenames t خواهد بود.

اگر نیاز به اسامی فایلها باشد، آنها می‌توانند با یک حلقه For به صورت زیر اضافه شوند :

```
$ for i in filenames ...
> do
>     echo $ i :
>     cat $ I
> done | P
```

حقیقتاً، ویژگیهای بسیار زیادی وجود دارند که ما می‌توانیم به این برنامه اضافه کنیم. بهتر است که یک نسخه جدا شده بسازیم، سپس بگذاریم گسترش یابد، به همان صورتی که تجربه پذیرفته میشود. به این صورت، ویژگیها از نوعی هستند که مردم حقیقتاً طالب آنها هستند و نه ویژگیهایی که ما فکر می‌کنیم آنها می‌خواهند.

ساختار اصلی P، همانند vis می‌باشد: زیر برنامه اصلی، درون فایلها تکرار می‌شود و زیر برنامه print را که بر روی هر کدام از آنها

کار می کند، فرا می خواند.

```

/ * p : print input in chunks (version \) * /
# include <stdio . h>
# define PAGESIZE 22
char * programe ; / * program name for error message * /
main (argc , argv)
    int argc ;
    char * argv [ ] ;
{
    int i ;
    FILE * fp , * fopen < 1 ;

Programme = argv [0] ;
if ( argc == 1 )
    print (stdin , PAGESIZE) ;
else
    for ( i = 1 ; i < argc ; i ++ ) {
        fp = < fopen (argv [i] , "r") ;
        print (fp , PAGESIZE) ;
        fclose (fp) ;
    }
    exit (0) ;
}

```

زیر برنامه `efopen`، یک عملکرد بسیار عمومی را وارد می کند. سعی کنید یک فایل را باز کنید؛ اگر امکان پذیر نیست، یک پیغام خطا را پرینت کنید و خارج شوید. برای اینکه پیغامهای خطا را تشویق کنید که برنامه مزاحم (یا مزاحم شده) را شناسایی کنند، `efopen` به یک برنامه رشته خارجی استناد داده می شود که شامل نام برنامه می باشد و در `main` تنظیم می شود.

```

FILE * fopen (file , mode) / * fopen file , die if (an.'t * /
Char * file , * mode ;
{
    FILE * fp , * fopen ( ) ;
    Extern char * programe ;

    if (( fp = fopen (file , mode)) != NULL)
        return fp ;
    fprintf (stderr , " % s : can't open file % s mode % s \ n " ,
        programe , file , mode ) ;
    exit (1) ;
}

```

ما یک جفت از طرحهای دیگر را برای `efopen`، قبل از فرو نشانیدن بر روی این برنامه، بررسی کردیم. یکی از آنها، قبل از چاپ کردن پیام، با یک اشاره گر تهی که خرابی و اشکال را نشان می دهد، آن را باز می گرداند. چنین چیزی به شماره گیرنده انتخاب ادامه یا خروج را می دهد. طرحی دیگر، `efopen` را با سه آرگومان تهیه می کند و این آرگومانها مشخص می کنند که آیا باید پس از شکست در باز کردن فایل بازگشت یا خیر. اما در اکثر مثالهای ما، اشاره ای برای ادامه وجود ندارد، اگر نتوان به یک فایل دست یافت، بنابراین نسخه

فعلی از `efopen` ، بهترین نسخه برای استفاده می باشد .

کار واقعی فرمان `p` ، در `print` انجام می شود.

```
Print (fp , pagesize) / * print fp in pagesize chunks */
FILE * fp ;
Int pagesize ;
{
    static int lines = 0 ; /* number of lines sofar */
    char buf [ BUFSIZ ] ;
    while (fgets (buf , size of buf , fp) != NULL)
        if ( ++ lines < page size)
            fputs (buf , stdout) ;

    else {
        buf [ strlen (buf) -1 ] '\0 ' ;
        fputs (buf , stdout) ;
        fflush (stdout) ;
        ttyin ( ) ;
        lines = 0 ;
    }
}
```

ما از `BUFSIZ` استفاده کردیم، که در `<stdio.h>` ، به عنوان اندازه میانگیر ورودی تعریف می شود. (`fgets (buf , size , fp)` ، سطر بعدی ورودی را از `fp` تا یک سطر جدید واکنشی می کند و سطر جدید را درون `buf` قرار می دهد و یک `0\` پایان دهنده اضافه می کند؛ بیشتر ، کاراکترهای اندازه 1 ، کپی می شوند. `BUFSIZ` , `NULL` را در انتهای فایل باز می گرداند. (`fgets` می "وند بهتر طراحی شود : `fgets` به جای یک تعداد کاراکتر، `buf` را باز می گرداند؛ به علاوه، هیچ اختطاری نمی دهد، اگر خط ورودی خیلی طولانی باشد. هیچ کدام از کاراکترها مفقود نمی شوند، اما شما باید برای مشاهده اینکه واقعا چه اتفاقی می افتد به `buf` نگاه کنید.) تابع `strlen`، طول یک رشته را باز می گرداند؛ ما از آن برای این منظور استفاده می کنیم که سطر جدید پسین، آخرین سطر ورودی را حذف کنیم. (`fputs (buf , fp)` ، رشته `buf` را روی فایل `fp` می نویسد. فراخوانی `fflush` در پایان صفحه، هر گونه خروجی میانگیر شده را خارج می کند.

وظیفه خواندن پاسخ از کاربر، پس اینکه هر صفحه پرینت شد، به یک زیربرنامه با عنوان `ttyin` فرستاده می شود. `Ttyin` نمی تواند ورودی استاندارد را بخواند، چون `p` باید کار کند حتی زمانی که ورودی آن از یک فایل یا از یک لوله می آید. برای بکارگیری آن، برنامه، فایل `/dev/tty` را باز می کند، که پایانه کاربر، بدون توجه به هر گونه جهت دهی مجدد ورودی استاندارد، می باشد. ما `ttyin` را برای بازگرداندن اولین کاراکتر از پاسخ نوشتیم، اما از آن ویژگی در اینجا استفاده نمی شود.

```
ttyin ( ) /* process response from /dev/tty (version 1) */
{
    char buf [ BUFSIZE ] ;
    FILE * efopen ( ) ;
    Static FILE * tty = NULL

    if (tty == NULL)
```

```

    tty = efopen ( " / dev / tty " , " r " );
if ( fgets ( buf , BUFSIZE , tty ) == NULL || buf [0] == ' q ' )
    exit ( 0 ) ;
else /* ordinary line */
    return buf [0] ;
}

```

اشاره گر فایل `dev/tty`، به صورت `static` اظهار می شود، در نتیجه ارزش خود را از یک فراخوان از `ttyin` به فراخوان بعدی نگه می دارد؛ فایل `dev/tty`، فقط در اولین فراخوان باز می شود.

به طور بدیهی ویژگیهای دیگری وجود دارند که باید بدون کار زیاد، به `p` اضافه شوند، اما ارزشی ندارد که نسخه اول ما از این برنامه، کاری را انجام دهد که در اینجا توصیف می شود: 22 سطر را پرینت کنید و منتظر بمانید. قبل از اینکه سایر چیزها اضافه شوند، زمان طولانی بود و تا امروز فقط افراد کمی از این ویژگیها استفاده می کنند.

یک ویژگی اضافی آسان، تعیین تعداد سطرها در هر صفحه می باشد، یک متغیر `pagesize` که می تواند از سطر فرمان تنظیم شود:

```
... p - n $
```

قطعات را با `n` سطر، پرینت می کند. چنین چیزی فقط مستلزم اضافه کردن یک رمز آشنا در آغاز `main` می باشد:

```

/* p : print input in chunks (version 2) */
...
int i , pagesize = pAGESIZE ;
programe = argv [0] ;
if ( argc > 1 $ $ argv [1] [0] == ' - ' ) {
    pagesize = atoi ( $ argv [1] [1] );
    argc -- ;
    argv ++ ;
}
...

```

تابع `atoi` یک رشته کاراکتر را به یک عدد صحیح تبدیل می کند (به (3) `atoi` مراجعه کنید).

افزایش دیگر به `p`، توانایی گریز به طور موقت، در پایان هر صفحه برای انجام فرمانی دیگر می باشد. در مقایسه با `ed` و بسیاری دیگر از برنامهها، اگر کاربر، سطری را تایپ کند که با یک علامت تعجب آغاز می شود، مابقی سطر به عنوان یک فرمان تلقی می شود و برای اجرا از شل عبور می کند. این ویژگی نیز بی اهمیت است، چون تابعی با عنوان (3) `system` وجود دارد که این کار را انجام می دهد، اما توضیح زیر را بخوانید. نسخه اصلاح شده `ttyin`، به شرح زیر است:

```

ttyin ( ) /* process response from / dev / tty (version 2) */
{
    char buf [BUFSIZ] ;
    FILE * efopen ( ) ;
    Static FILE * tty = NNUL ;
    if ( tty == NULL )
        tty = efopen ( " / dev / tty " , " r " ) ;
}

```

```

for ( ;; ) {
    if (fgets (buf , BUFSIZE , tty) == NULL || buf [0] == 'q'
        exit (0) ;
    else if (buf [0] == '!') {
        system (buf + 1) ;      /* BUG here */
        printf ("!\n") ;
    }
    else /* ordinary line */
        return buf [0] ;
}
}

```

متأسفانه ، این نسخه از `ttyin` دارای یک خطای نامحسوس و خطرناک می باشد. فرمان، توسط `system` اجرا می شوند و ورودی استاندارد را از `p` می گیرد ، بنابراین اگر `p` از لوله یا یک فایل خوانده شود، فرمان می تواند با ورودی خودش، تداخل کند :

```

$ cat / etc / passwd | p - 1
root : 3 D : f HR 5 KoB. 3 s : o : 1 : s . usr : / : ! ed      ed از p می گیرد
                        ؟                               ed / ... passwd / etc را می خواند
                        !                               ... غلط است و از سیستم خارج می شود

```

راه حل ، مستلزم آگاهی در این خصوص می باشد که چگونه فرآیندهای یونیکس کنترل می شوند و ما آن را در بخش 4 - 7 ارائه می دهیم. در حال حاضر، آگاه باشید که سیستم استاندارد در کتابخانه می تواند مشکل آفرین باشد، اما `ttyin` به درستی کار می کند اگر با نسخه `system` در فصل 7 کامپایل شود.

ما اکنون دو برنامه نوشته ایم ، `vis` و `p` که باید متغیرهای `cat`، با اندکی شاخ و برگ تلقی شوند. بنابراین ، آیا آنها باید همگی بخشی از `cat` و قابل دستیابی توسط آرگومانهای اختیاری مانند `p -` و `v -` باشند؟ پرسش مربوط به اینکه آیا یک برنامه جدید بنویسیم یا ویژگیها را به برنامه قبلی اضافه کنیم، مکرراً مطرح می شود، مادامی که افراد عقاید جدید دارند. ما یک پاسخ قطعی نداریم، اما اصولی وجود دارند که به تصمیم گیری در این زمینه کمک می کنند.

اصل مهم، این است که برنامه باید فقط یک کار اصلی را انجام دهد - اگر کارهای زیادی را انجام دهد، بزرگتر، کندتر و سختتر قابل نگهداری و سخت تر قابل استفاده می باشد. به راستی ، ویژگیها، اغلب بدون استفاده می مانند، چون مردم نمی توانند انتخابها را به خاطر بسپارند.

این موضوع بیان می کند که `cat` و `vis` نباید ترکیب شوند. `Cat` فقط ورودی خودش را کپی می کند ، بدون اینکه آن را تغییر دهد، در حالیکه `vis` آن را منتقل می کند. ترکیب آنها، برنامه ای می سازد که دو چیز متفاوت را انجام می دهند. این حالت تقریباً با `cat` و `p` نیز آشکار می باشد. `Cat` برای کپی کردن سریع و موثر معنی می دهد، در حالیکه `p` برای گرفتن اطلاعات معنی می دهد. و `p` ورودی خودش را منتقل می کند : هر سطر جدید بیست و دوم، حذف می شود. سه برنامه مجزا به نظر طرح صحیحی می آیند.

تمرین 6 - 6 - آیا `p` به طور معقول عمل می کند اگر `pagesize` مثبت نباشد؟

تمرین 7 - 6 - چه چیز دیگری می توان با **p** انجام داد؟ توانایی برای پرینت مجدد بخش های ورودی قبلی را (اگر مناسب است) ارزیابی و اجرا کنید. (این یک ویژگی اضافی است که ما از آن استفاده می کنیم). یک مسیر ساده را برای مجاز کردن پرینت کمتر از یک پرده پر از ورودی پس از هر وقفه، اضافه کنید. یک مسیر ساده را برای پوشش به طرف جلو یا به طرف عقب برای یک سطر مشخص شده توسط عدد یا محتوا، اضافه کنید.

تمرین 8 - 6 - از توانایی های کار - گردانی فایل **exes** توکار شل استفاده کنید (برای تثبیت فراخوان **ttyin** با سیستم به **sh (1)** مراجعه کنید).

تمرین 9 - 6 - اگر شما فراموش کنید که یک ورودی را برای **p** مشخص کنید ، **p** به آرامی منتظر ورودی از پایانه باقی می ماند. آیا کشف این خطای احتمالی، ارزشمند است؟ اگر ارزشمند است؟ چگونه ؟ توجه : **(isutty 3)**

5 - 6 - يك مثال : pick

نسخه **Pick** در فصل 5، به وضوح توانایی های شل را گسترش می دهد. نسخه **c** که اکنون بیان می شود، تا حدودی متفاوت از نسخه **c** موجود در فصل 5 می باشد. اگر این نسخه دارای آرگومانها باشد، آنها همانند قبل پردازش می شوند. اما اگر تنها آرگومان **'_'** مشخص شود. **Pick**، ورودی استاندارد آن را پردازش می کند.

اگر هیچ آرگومانی وجود نداشته باشد، چرا ورودی استاندارد خوانده نمی شود؟ نسخه دوم فرمان **zap** را در بخش 6 - 5 در نظر بگیرید :

```
kill $ SIG 'pick ]' ps - ag | egrep "$*" '\|awk' {print $1}'
```

چه اتفاقی می افتد اگر طرح **egrep** چیزی را تطبیق نکند؟ در این حالت **pick** هیچ آرگومانی ندارد و شروع به خواندن ورودی استاندارد خود می کند؛ فرمان **zap**، در یک روش گیج کننده، خراب می شود. نیاز به یک آرگومان آشکار، یک روش آسان برای غیر مبهم کردن چنین موقعیتهایی می باشد و قرارداد **'_'** از **cat** و سایر برنامه ها، نشان می دهد که چگونه آن را تشخیص دهیم.

```
/* pick : offer choice on each argument */
#include <stdio.h>
char * programe; /* program name for error message */

main (argc , argv)
int argc ;
char * argv [ ] ;
{
int i ;
char buf [ BUFSIZ ] ;
programe = argv [0] ;
if (argc == 2 $ $ strcmp (argv [1] , "_") == 0) /* pick _ */
while (fgets (buf , sizeof buf , stdin) i = NULL) {
```

```

        buf [ strlen (buf) -1 ] = '\0'; /* drop newline */
        pick (buf);
    }
else
    for ( i = 1; i < argc; i++)
        pick (argv [i] );
    exit ( 0 );
}
pick (s) /* offer choice of s */
char * s;
{
    fprintf (stderr , "% s?" , s );
    if (ttyin ( ) == 'y')
        print f ("% s \n" , s );
}

```

`pick` برای انتخاب آرگومانها به طور برهم کنشی یک مسیر ساده را در یک برنامه متمرکز می کند. چنین چیزی نه تنها یک کار مفید را فراهم می کند، بلکه همچنین نیاز به انتخابهای برهم کنشی را بر روی سایر فرمانها کاهش می دهد.

تمرین 10 - 6 - با توجه به `pick`، آیا نیاز به `rm - i` وجود دارد؟

6 - 6 - روی خطاها و خطازدایی

اگر شما قبلاً یک برنامه نوشته باشید، با تصور یک خطا آشنا هستید. راه حل خوبی برای نوشتن رمز بدون خطا وجود ندارد به جز اینکه مراقب باشیم که یک طرح ساده و تمیز به وجود آورید و آن را به دقت اجرا کنید و آن را تمیز نگه دارید، همچنانکه آن را تغییر می دهید.

ابزار اندکی از یونیکس وجود دارند که به شما در پیدا کردن خطاها کمک می کنند، اگر چه هیچ کدام از آنها واقعاً عالی نمی باشند. اما برای شرح آنها، ما به یک خطا نیاز داریم و همه برنامه های موجود در این کتاب کامل هستند. بنابراین، ما یک خطای نمونه ایجاد می کنیم.

تابع ارائه شده `pick` در بالا، را در نظر بگیرید. در این جا نیز وجود دارد، اما اکنون دارای یک خطا می باشد. (نیازی نیست که دوباره آن را از اول تکرار کنیم).

```

pick (s) /* offer choice of s */
Char * s;
{
    fprintf ( "% s ?" , s );
    if (ttyin ( ) == 'y')
        print f ("% s \n" , s );
}

```

اگر ما آن را کامپایل و سپس اجرا کنیم ، چه اتفاقی می افتد؟

\$ (pick . c - o pick)

\$ pick * . c

آن را بررسی کنید

ناپدید می شود !

(memory fault (ore dumped

“memory fault” به این معناست که برنامه شما سعی می کند به بخشی از حافظه مراجعه کند

که مجاز نمی باشد. معمولاً به این معناست که یک اشاره گر، به جایی نامعقول اشاره می کند. “Bus error”، تشخیصی دیگر با همین معنی می باشد و اغلب با پویش کردن یک رشته پایان نیافته، بوجود می آید.

“cor dumped”، به این معناست که کرنل، وضعیت برنامه اجرا کننده شما را در یک فایل با نام core در فهرست ذخیره می کند. شما همچنین می توانید یک برنامه را مجبور کنید که با تایپ `1 - ctl`، حافظه اصلی را خالی کند، البته اگر در پیش زمینه اجرا می شود و یا با نرمال `kill-3`، حافظه اصلی را خالی کنید، البته اگر در پیش زمینه می باشد.

دو برنامه برای نوشتن در حافظه وجود دارد، `adb` و `sdb`. همانند اکثر خطا زدها، آنها مرموز، پیچیده و ضروری هستند. `Adb` در هفتمین ویرایش وجود دارد؛ `sdb` در اکثر نسخه های اخیر سیستم در دسترس می باشد. یکی از این دو برنامه، مطمئناً در سیستم وجود دارند.

ما در اینجا فقط به حداقل استفاده قطعی از هر کدام از آنها می پردازیم : پرینت کردن یک

`stack trace`، که تابعی است که زمانی اجرا شد که برنامه به پایان رسید، تابعی که آن را فراخواند و مواردی از این قبیل. اولین تابع نامگذاری شده در `stack trace`، در جایی است که برنامه وجود داشت زمانی که خاموش شد.

برای اجرای یک `stack trace` با `adb`، فرمان `$ c` می باشد :

`adb pick core $`

به `adb` استناد می کند

`c $`

`stack trace` را درخواست می کند

(`strout -و 0 و 011 و 011200 01155772`)

`adjust :` °

`fillch :` 060542

`doprint (011200 01155722 و 011 و 0 و)`

`f print f(0177345 011200 و)`

`iop :` 011200

`fmt :` 0177345

`args :` °

`pick (0177345)`

`s :` 0177345

`main (0177234 035 و)`

`argc :` 035

`argv :` 0177234

`i :` 01

`buf :` °

`ctl - d`

از سیستم خارج شوید

\$

این برنامه ، بیان می کند که pick ، main را فراخواند، که در نتیجه fprintf را فراخواند. doprint ، strout - ، را فراخواند. چون - dopmt در هیچ جایی در pick.c ذکر نمی شود، در نتیجه مشکلات ما باید جایی در fprintf یا بالاتر باشند. (سطرها پس از هر زیربرنامه در- traceback، ارزش متغیرهای محلی را نشان می دهند. \$c این اطلاعات را حذف می کند، همان کاری که \$c خودش در برخی از نسخه های adb انجام می دهد)

قبل همه این موارد ، همین مورد را با sdb بررسی می کنیم :

\$ sdb pick core

warning : ' a . out ' not compiled with - g

iseek : address oxa 64 زیر برنامه در جایی که برنامه به پایان می رسد

* t

stack trace را درخواست می کند

isseek ()

fprintf (54 91 47 47 21 540 61)

pick (54 91 47 47 21)

main (12 91 47 47 21 30 و 21 47 47 89 88)

* q

از سیستم خارج شوید

\$

اطلاعات به صورت متفاوت فرمت می شوند، اما یک موضوع عمومی وجود دارد:

fprintf متفاوت است چون بر روی یک ماشین متفاوت اجرا شد - VAX_11/750 - که دارای یک پیاده سازی

متفاوت از کتابخانه استاندارد I/O می باشد). و مطمئناً، اگر ما در نسخه خراب pick به درخواست fprintf نگاه کنیم، اشتباه است :

f print f (" % s " , s) ;

stderr وجود ندارد، بنابراین رشته فرمت " % s ? " ، به عنوان يك اشاره گر FILE استفاده

می شود و البته بی نظمی رخ می دهد.

ما این خطا را پاک کردیم، چون عمومی است، یک نتیجه از بی توجهی به جای طرح بد. همچنین این امکان وجود دارد که خطاها را به

این صورت پیدا کنیم، که در آن یک تابع با آرگومانهای اشتباه و از طریق استفاده از lint (1) ، درستی سنجی c ، فراخوان می شود.

Lint به بررسی برنامه های c به منظور یافتن خطاهای بالقوه، مشکلات قابل حمل و ساختارهای مشکوک می پردازد. اگر ما lint را روی

کل فایل pick.c اجرا کنیم، خطا مشخص می شود :

\$ lint pick . c

.....

fprintf , arg . 1 used in consistently " llib - 1 c " (69) : : " pick.c " (28)

در ترجمه ، این برنامه بیان می کند که اولین آرگومان fprintf در تعریف کتابخانه استاندارد، از استفاده آن در 28 سطر از برنامه ها،

متفاوت می باشد. این یک تذکر قوی درباره چیزی می باشد که اشتباه است.

Lint ، یک موفقیت مرکب است. Lint دقیقاً بیان می کند که چه چیزی در این برنامه اشتباه است، اما همچنین تعداد زیادی از پیامهای

نامربوط را تولید می کند که ما در بالا حذف کردیم و مستلزم مقداری تجربه در این خصوص می باشد که بدانیم به چه چیزی توجه شود

و چه چیزی نادیده گرفته شود. اگر چه تلاش ارزشمندی است ، چون lint برخی از خطاهایی را پیدا می کند که تقریباً غیرممکن است

افراد آنها را ببینند. اجرای lint پس از یک زمانی طولانی از ویرایش ، ارزشمند است و اطمینان می دهد که شما هر خطاری را که می دهد ، درک می کنید.

7 - 6 - يك مثال : zap

zap ، که به صورت انتخابی، فرآیندها را حذف می کند، برنامه دیگری است که ما آن را به عنوان یک فایل شل در فصل 5 ، ارائه دادیم. مشکل عمده با این نسخه، سرعت است : zap فرآیندهای زیادی را بوجود می آورد که به کندی آنها را اجرا می کند؛ و بویژه برای برنامه ای که فرآیندهای خاطی را حذف می کند. نوشتن مجدد zap در C ، آن را سریعتر می کن. اما ما قصد نداریم که کل کار را در اینجا انجام هیم : ما هنوز از ps برای پیدا کردن اطلاعات فرآیند استفاده می کنیم. Ps خیلی آسانتر از خارج کردن اطلاعات از کرنل می باشد و قابل انتقال نیز می باشد. zap ، یک لوله را با ps بر روی انتهای ورودی باز می کند و از روی آن می خواند به جای اینکه از یک فایل بخواند. تابع (3) popen ، مشابه به fopen می باشد. به جز اینکه اولین آرگومان ، یک فرمان است به جای اینکه یک اسم فایل باشد. همچنین یک pclose وجود دارد که ما در اینجا به آن نیازی نداریم.

```

/ * zap : interactive process killer */
# include <stdio.h>
# include <signal.h>
char * progame ;      / * program name for error message */
char * ps = "ps - ag" ; / * system dependent */
main ( argc , argv )
    int argc ;
    char * argv [ ] ;
{
    FILE * fin , * popen ( ) ;
    Char buf [ BUFSIZ ] ;
    int pid ;

    progame = argv [0] ;
    if (( fin = popen ( ps , "r" )) == NULL) {
        fprintf (stderr , "%s : can't run %s \n" , progame , ps) ;
        exit (1) ;
    }
    fgets (buf , size of buf , fin) ;      / * get header line */
    fprintf (stderr , "%s" , buf) ;
    while (fgets (buf , size of buf , fin) != NULL )
        if (argc == 1 || strindex (buf , argv [1]) >= 0 ) {
            buf [ strlen (buf) - 1 ] = '\0' ; / * suppress \n */
            fprintf (stderr , "%s" , buf) ;
            if (ttyin () = 'y') {
                sscanf (buf , "%d" , $ pid0;
                kill (pid , SIGKLL) ;
            }
        }
}

```

```
exit ( 0 ) ;
}
```

ما برنامه را برای استفاده از `ps - ag` نوشتیم (انتخاب وابسته به سیستم می باشد)، اما شما می توانید فقط فرآیندهای خود را حذف کنید مگر اینکه شما ابرکاربر باشید. اولین فراخوان برای `fgets`، سطر عنوان را از `ps`، پاک می کند؛ این یک عملکرد جالب برای استنتاج چیزی است که اتفاق می افتد، اگر شما سعی کنید فرآیندی را که مطابق با آن سطر عنوان می باشد، حذف کنید.

تابع `sscanf`، عضوی از خانواده `(3 scanf)` برای انجام تبدیل فرمت ورودی می باشد. این تابع، از یک رشته تبدیل می شود به جای اینکه از یک فایل تبدیل شود.

فراخوان `kill` از سیستم، علامت خاصی را به فرآیند می فرستد؛ علامت `SI GKI LL` که در

`< signal. h >` تعریف می شود و نمی تواند متوقف شود و یا نادیده گرفته شود. شما ممکن است از فصل 5 به خاطر آوردید که ارزش عددی آن 9 است. اما عملکرد بهتر، استفاده از ثابت های نمادی از فایل های عنوان می باشد، به جای اینکه برنامه های خود را با اعداد جادویی، بیان کنید.

اگر هیچ آرگومانی وجود نداشته باشد، `zap` هر سطر از ورودی `Ps` را برای انتخاب ممکن، ارائه می دهد. اگر یک آرگومان وجود داشته باشد. در نتیجه `zap`، فقط سطرهایی از خروجی `ps` را ارائه می دهد که آن را تطبیق کنند. تابع `(strindex (s1, s2))`، بررسی می کند که آیا آرگومان، بخشی از یک سطر از خروجی `ps` را تطبیق می کند یا نه و این کار را با استفاده از `st rn cmp` انجام می دهد (جدول 2 - 6). `Strindex`، موقعیت در `s1` را به جایی باز می گرداند که `s2` رخ می دهد و یا به `-1` باز می گرداند، اگر `s2` وجود نداشته باشد.

```
strindex (s , t ) /* return index of tins , - 1 if none */
char * s , * t ;
{
    int i , n ;
    n = strlen (t) ;
    for ( i = 0 ; s [i] != '\0' ; i ++ )
        for (strncmp (sti , t , n) == 0 )
            return i ;
    return - 1 ;
}
```

جدول 4 - 6 - تابعهایی را که عموماً استفاده می شوند، از کتابخانه استاندارد `I/O` را به طور خلاصه بیان می کند.

تمرین 11 - 6 - `zap` را به گونه ای تغییر دهید که هر تعداد از آرگومانها بتوانند ذخیره شوند. همانگونه که نوشته شد، `zap` در حالت عادی، سطر را بر طبق خودش به عنوان یکی از انتخابها، پژواک می کند. آیا باید اینگونه باشد؟ اگر نه، برنامه را متعاقباً تغییر دهید: نکته `(get pid (3 :`

تمرین 12 - 6 - یک تابع 1) fgrep در اطراف strindex بسازید. زمانهای اجرا برای جستجوهای پیچیده را با هم مقایسه کنید، ده کلمه را در یک سند بیان کنید. چرا fgrep سریعتر عمل می کند؟

8 - 6 - يك برنامه مقایسه فایل برهم کنش : idiff

یک مشکل عمومی ، داشتن دو نسخه از یک فایل می باشد، که تا حدودی متفاوت هستند و هر کدام شامل بخشی از یک فایل مطلوب می باشند؛ چنین چیزی اغلب زمانی بوجود می آید که تغییرات به طور مستقل توسط دو فرد متفاوت انجام شوند. Diff به شما می گوید که چگونه فایلها با هم تفاوت دارند، اما diff به طور مستقیم به شما کمک نمی کند، اگر شما بخواهید بخشی از قسمتهای اولین فایل و برخی از قسمتهای دومین فایل را انتخاب کنید.

در این بخش ، ما یک برنامه diff (idiff برهم کنشی) می نویسیم که هر قطعه از خروجی diff را ارائه می دهد و اختیار انتخاب بخش « from » انتخاب بخش « to » یا ویرایش بخش را پیشنهاد می کند.

idiff ، قطعات انتخاب شده را در یک ترتیب صحیح. در یک فایل با عنوان out . idiff قرار می دهد. که با توجه به این دو فایل می باشد :

file 1 :	file 2:
this is	this is
a tegt	hot a test
of	of
your	our
skill	ability
and comprehension	
diff produces	

جدول 4 - 6 - تابعهای مفید و استاندارد I/O

(fp = fopen (s , mode

فایل < را باز می کند؛ وضعیت "a" ، "w" ، "r" برای

خواندن، نوشتن ، ضمیمه کردن (NULL) را برای خطا باز می گرداند)

(c = getc (ff

کاراکتر را می گیرد؛ get char (st din) ، () می باشد

(putc (c , fp

کاراکتر را قرار می دهد؛ put char (c , std out) ، می باشد

(ungetc (c , fp

کاراکتر را بر روی فایل ورودی fp قرار می دهد؛ حداکثر یک char

می تواند در یک زمان به عقب برگردد.

(... , scanf (fmt , al

کاراکترها (از stdin (, ... , a) ، بر طبق fmt

می خواند. هر ai باید یک اشاره گر باشد.

EOF های بازگشت یا تعداد میدانها معکوس می شوند.

(... , f scanf (fp

از فایل fp می خواند

(... , ss can f (s

از فایل s می خواند

(... , printf (fmt , al	al , ... را بر طبق fmt فرمت می کند و بر روی stout پرینت می کند
(... , fprintf (fp	... را بر روی فایل fp پرینت می کند
(... , sprintf (s	... را درون رشته s، پرینت می کند
(fgets (s , n , fp	حداکثر n کاراکتر را از fp ، درون s می خواند NULL را در انتهای فایل باز می گرداند.
(fputs (s , fp	رشته s را بر روی فایل fp پرینت می کند
(fflush (fp	هر گونه خروجی میانگیر شده، بر روی فایل fp را پاک می کند
(fclose (fp	فایل fp را می بندد
(fp = fopen (s , mode	لوله را برای فرمان s باز می کند. Fopen را مشاهده کنید
(pclose (fp	لوله fp را می بندد
(system (s	فرمان s را اجرا می کند و منتظر کامل شدن آن باقی می ماند

```
$ diff file 1 file 2
2 c 2
< a test
.....
> not a test
4 , 6 c 4 , 5
< your
< skill
< and comprehension
.....
> our
> ability
$
```

: مانند زیر می باشد idiff یک مکالمه با

```
$ idiff file 1 file 2
2 c 2
< a test
.....
> not a test
? >
4 , 6 c 4 , 5
< your
< skill
< and comprehension
.....
> our
```

اولین تفاوت

را انتخاب می کند (<) کاربر نسخه دوم

دومین تفاوت

```
> ability
```

```
? <
```

را انتخاب می کند (>) کاربر اولین نسخه

```
idiff out put in file idiff . out
```

```
$ cat idiff . out
```

خروجی در این فایل قرار می گیرد

```
this is
```

```
not a test
```

```
of
```

```
your
```

```
skill
```

```
and comprehension
```

```
$
```

اگر بجای < یا > پاسخ e داده شود، در نتیجه ، ed ، idiff را به دو گروه از سطرهایی که خوانده می شوند، استناد می کند. اگر دومین پاسخ e بوده باشد، در نتیجه، میانگیر ویراستار به این صورت به نظر می رسد :

```
your
```

```
skill
```

```
and comprehension
```

```
.....
```

```
our
```

```
ability
```

هر چیزی که درون فایل توسط ed نوشته می شود، چیزی است که وارد خروجی نهایی می شود.

در آخر، هر فرمانی می تواند از طریق گریز با ! cmd در idiff اجرا شود. از نظر تکنیکی، سخت ترین قسمت کار diff می باشد و تاکنون برای ما انجام شده است. بنابراین، کار واقعی idiff، تجزیه کردن خروجی diff و باز کردن ، بستن، خواندن و نوشتن فایل های صحیح در زمان درست آن می باشد. زیر برنامه اصلی idiff، فایلها را تنظیم می کند و فرآیند diff را اجرا می کند :

```
/ * idiff : interactive diff */
```

```
# include <stdio . h >
```

```
# include <ctype . h >
```

```
char * programe ;
```

```
# define HUGE 10000 /* large number of lines */
```

```
main (argc , argv )
```

```
int argc ;
```

```
char * argv [ ] ;
```

```
{
```

```
FILE * fin , * fout , * f1 , * f2 , * fopen ( ) ;
```

```
char buf [ BUFSIZ] , * mktemp ( ) ;
```

```
char * diffout = " idiff. x x x x x " ;
```

```
programe = argv [ 0 ] ;
```

```
if (argc != 3) {
```

```
printf (stderr , " usage ; idiff file 1 file 2 \n " ) ;
```

```
exit (1) ;
```

```
}
```

```

f1 = fopen (argv [1] , " r " ) ;
f2 = fopen (argv [2] , " r " ) ;
fout = fopen ( " idiff . out " , " w " ) ;
mktemp (diffout) ;
sprintf (buf , " diff % s > % s " , argv [1] , argv [2] , diffout) ;
system (buf) ;
fin = fopen (diffout , " r " ) ;
idiff (f1 , f2 , fin , fout) ;
unlike (diffout) ;
printf ( " % s output in file idiff . out \ n " , progname ) ;
exit < 01 ;
}

```

تابع (3) `mktemp` ، فایل را بوجود می آورد که نامش، متفاوت از هرگونه فایل موجود می باشد. `mktemp` ، آرگومان خود را روی هم می نویسد: شش-`x's` ، توسط `process-id` از فرآیند-`idiff` و یک حرف، جایگزین می شوند. فراخوان سیستم-`(2) unlike` ، فایل نامگذاری شده را از سیستم فایل پاک می کند. کار حلقه سازی درون تغییرات گزارش شده توسط `diff` ، توسط یک تابع با عنوان `idiff` انجام می شود. عقیده اصلی ساده است: یک قطعه از خروجی `diff` را پرینت کنید، از روی داده های ناخواسته در یک فایل بگذرید، سپس نسخه مطلوب از فایل دیگر را کپی کنید. جزئیات یکنواخت بسیار زیادی وجود دارد. بنابراین رمز بزرگتر از چیزی است که ما می خواهیم، اما درک بخشهای آن، بسیار آسان است.

```

idiff (f1 , f2 , fin , fout) /* process diffs */
FILE * f1 , * f2 , * fin , * fout ;
{
char * tempfile = " idiff - x x x x x x " ;
char buf [ BUFSIZ ] , buf2 [ BUFSIZ ] , * mktemp ( ) ;
FILE * ft , * fopen ( ) ;
int cmd , n , from1 , to1 , from2 , to2 , nf1 , nf2 ;

mktemp (tempfile) ;
nf1 = nf2 = 0 ;
while (fgets (buf , size of buf , fin) != NULL) {
parse (buf , $ from1 , $ from 1 , $ to 1 , $ cmd , $ from 2 , $ to 2) ;
n = to1 - from1 + to2 - from 2 + 1 ; /* # lines from diff */
if (cmd == ' c ' )
n += 2 ;
else if (cmd == ' a ' )
from 1 ++ ;
else if (cmd == ' d ' )
from 2 ++ ;
printf ( " % s " , buf) ;
while (n -- > 0) {
fgets (buf , size of buf , fin) ;

```

```

        print f ( " % s " , buf ) ;
    }
do {
    print f ( " ? " ) ;
    fflush ( stdout ) ;
    fgets ( buf , sizeof buf , stdin ) ;
    switch ( buf [ 0 ] ) {
    case ' > ' :
        nskip ( f1 , to 1 - nf1 ) ;
        ncopy ( f2 , to2 - nf2 , fout ) ;
        break ;
    case ' < ' :
        nskip ( f2 , to2 - nf2 ) ;
        ncopy ( f1 , to1 - nf1 , fout ) ;
        break ;
    case ' e ' :
        ncopy ( f1 , from 1 - 1 - nf1 , fout ) ;
        nskip ( f2 , from 2 - 1 - nf2 ) ;
        ft = fopen ( tempfile , " w " ) ;
        ncopy ( f1 , to 1 + 1 - from1 , ft ) ;
        fprintf ( ft , " - - \n " ) ;
        ncopy ( f2 , to 2 + 1 - from 2 , ft ) ;
        fclose ( ft ) ;
        sprintf ( buf2 , " ed % s " , temp file ) ;
        system ( buf2 ) ;
        ft = fopen ( tempfile , " ed % s " , temp file ) ;
        system ( buf2 ) ;
        ft = fopen ( tempfile , " r " ) ;
        ncopy ( ft , MUGE , fout ) ;
        fclose ( ft ) ;
        break ;
    case ' ! ' :
        system ( buf + 1 ) ;
        printf ( " ! \n " ) ;
        break ;
    default :
        print f ( " < or > ore or ! \n " ) ;
        break ;
    }
} while ( buf [ 0 ] != ' < ' $ $ buf [ 0 ] = ' > ' $ $ buf [ 0 ] != ' e ' ) ;
nf 1 = to 1 ;
nf 2 = to 2 ;
}
ncopy ( f1 , HUGE , fout ) ; /* can fail on very long files */
unlike ( tempfile ) ;

```

}
تابع parse، فرمان را انجام می دهد اما کار دشوار تجزیه سطرها که توسط diff انجام می شود، چهار شماره سطر و فرمان را استخراج می کند (یکی از a، b، c یا d).

parse، اندکی پیچیده است چون diff، می تواند یک یا دو شماره سطر را در هر طرف از طرف فرمان بوجود آورد.

```
parse (s , pfrom 1 , pto 1 , pcmd , pfrom 2 , pto2)
char * s ;
int * pcmd , * pfrom1 , * pto1 , * pfrom2 , * pto2 ;
{
# define a2i (p) while (is digit (*s)) p = 10 * (p1 + * s ++ - '0'
* pfrom1 = * pto1 = * pfrom 2 = * pto 2 = 0 ;
a2i (* pfrom1 );
if (* s == ' , ' ) {
s ++ ;
a2i (* pto1 );
} else
* pto 1 = * pfrom 1 ;
* pcmd = * s ++ ;
a2i (* pfrom 2 );
if (* s == ' , ' ) {
s ++
a2i (* pto2 );
} else
* pto 2 = * pfrom 2 ;
}
```

درشت دستورالعمل a2i، تبدیل ویژه ما از Ascii به عدد صحیح را در چهار مکانی که رخ می دهد، انجام می دهد.

nskip و ncopy، از تعداد خاصی از سطرهای یک فایل عبور می کنند و یا آنها را کپی می کنند :

```
nskip (fin , n ) /* skip n lines of file fin */
FILE * fin ;
{
char buf [BUFSIZ] ;

while (n -- > 0)
fgets (buf , sizeof buf , fin) ;
}
ncopy (fin , n , fout) /* copy n lines from fin to fout */
FILE * fin , * fout
{
char buf [BUFSIZ] ;
while ( n -- > 0) {
if (fgets (buf , sizeof buf , fin) == null)
return ;
fputs (buf , fout) ;
}
}
```


همانطور که نشان می‌دهد، `idiff`، نمی‌تواند با ملایمت از سیستم خارج شود، اگر دچار وقفه شود، چون `idiff`، فایل‌های متعددی را که در `tmp/` قرار دارند، رها می‌کند. در فصل بعد، ما نشان می‌دهیم که چگونه برای حذف فایل‌های موقتی، مانند فایل‌هایی که در اینجا استفاده می‌شوند، از وقفه‌ها استفاده کنیم.

مشاهده‌دشوار در `zap` و `idiff`، این است که قسمت اعظم کار سخت، توسط فردی دیگر انجام شده است. این برنامه‌ها، صرفاً یک واسطه مناسب را روی برنامه‌ای دیگر قرار می‌دهند که اطلاعات درست را محاسبه می‌کند. به دنبال فرصتی بودن برای ساخت بر روی کار فردی دیگر، ارزشمندتر این است که خودتان آن را انجام دهید - این یک روش ارزان قیمت و سودمندتر است.

تمرین 13 - 6 - یک فرمان `q` را به `idiff` اضافه کنید: پاسخ `q`، همه باقیمانده انتخاب‌های `>` ' را به طور خودکار می‌گیرد؛ `q` < همه باقیمانده انتخاب‌های `<` ' را می‌گیرد.

تمرین 14 - 6 - `idiff` را به گونه‌ای تغییر دهید که همه آرگومان‌های `diff` از `diff` عبور کنند؛ `-b` و `-h`، احتمالاً منتخب‌ها هستند. `Idiff` را به گونه‌ای تغییر دهید که یک ویراستار متفاوت بتواند شناسایی شود، مانند

```
$ idiff - eanother - editor file 1 file 2
```

چگونه این دو تغییر برهم کنش دارند؟

تمرین 15 - 6 - برای استفاده از `popen` و `pclose` به جای یک فایل موقتی برای خروجی `idiff`، `diff` را تغییر دهید. چه تفاوتی در سرعت و پیچیدگی برنامه بوجود می‌آید؟

تمرین 16 - 6 - `diff` دارای این ویژگی می‌باشد که اگر یکی از آرگومان‌های آن، یک فهرست راهنما باشد، به جستجوی آن فهرست برای یک فایل با نامی شبیه آرگومانی دیگر می‌پردازد. اما اگر شما همین کار را با `idiff` انجام دهید، به شکلی عجیب خراب می‌شود. شرح دهید چه اتفاقی می‌افتد و سپس آن را ثابت کنید.

9-6 - دستیابی به محیط

دستیابی به متغیرهای محیط شل از یک برنامه `C` آسان است و چنین چیزی گاهی اوقات می‌تواند برای ساختن برنامه‌هایی موافق با محیط‌شان و بدون نیاز به کاربرهای آنها، استفاده شود. برای مثال، فرض کنید که شما از یک پایانه استفاده می‌کنید که در آن اندازه پرده نمایش بزرگتر از 24 سطر نرمال می‌باشد. اگر شما بخواهید از `p` استفاده کنید و از توانایی‌های پایانه خود حداکثر استفاده را ببرید، چه انتخابی برای شما وجود دارد؟ مشخص کردن اندازه پرده نمایش در هر زمانی که شما از `p` استفاده می‌کنید، دشوار است:

```
$ p - 36 ...
```

: خود قرار دهید `bin` شما باید همیشه یک نام شل را در

```
$ cat /usr / you / bin / p
```

```
exec /usr / bin / p - 36 $ *
```

```
$
```

سومین راه حل، تغییر `P` برای استفاده از یک متغیر محیط می‌باشد که ویژگی‌های پایانه شما را تعریف می‌کند. فرض کنید که شما متغیر

PAGESIZE را در profile خود تعریف می کنید :

```
PAGESIZE = 36
Export PAGESIZE
```

زیر برنامه (" var " getenv)، به جستجوی محیطی برای متغیر var شل می پردازد و ارزش خود را به صورت یک رشته از کاراکترها باز می گرداند و یا به صورت NULL، اگر متغیر تعریف نشود. با توجه به getenv، تغییر p، آسان است. همه آن چیزی که مورد نیاز است، اضافه کردن یک جفت از اعلانها و یک فراخوانی برای getenv در آغاز زیر برنامه اصلی می باشد.

```
/* p : print input in chunks (version 3) */
```

```
.....
```

```
char * p , * getenv ( ) ;
```

```
progname = argv [0] ;
```

```
if (( p = getenv ( " PAGESIZE" )) != NULL
```

```
    pagesize = atoi ( p ) ;
```

```
if (argc > 1 $ $1 argv [1] [0] == ' _ $ ) {
```

```
    pagesize = atoi ( $ arg v [1] [1] );
```

```
    argc -- ;
```

```
    argv ++ ;
```

```
}
```

```
.....
```

آرگومانهای انتخابی، پس از متغیر محیط پردازش می شوند. بنابراین هر اندازه آشکار از صفحه، یک اندازه ناآشکار از صفحه را نمی پذیرند.

تمرین 17 - 6 - idiff را به گونه ای تغییر دهید که به جستجوی محیط برای نام ویراستاری پردازد که استفاده می شود. برای استفاده از PAGESIZE، 2 و 3، غیره را تغییر دهید.

تاریخچه و نکات کتابشناسی

کتابخانه استاندارد I/O، پس از کتابخانه قابل انتقال از میک لاسک، توسط دنیس ریتچای، طراحی شد. هدف هر دو بسته نرم افزاری، فراهم کردن روش های ساده استاندارد است که برنامه ها بتوانند توسط آنها، بدون تغییر از سیستم های یونیکس به سیستم های غیر یونیکس حرکت کنند.

طرح p بر اساس یک برنامه از هنری اسپنسر می باشد.

adb توسط استیو بورن، sdb توسط هوارد کاتسف و lint توسط استیو جانسون نوشته شد.

Idiff، کما بیش بر اساس یک برنامه می باشد که در اصل توسط جومارائانو نوشته شد. Diff خودش توسط داگ میکلوری نوشته شد و بر اساس الگوریتمی می باشد که مستقلاً توسط هارولد استون و توسط واینی هانت و تام سیزمانسکی اختراع شد. (به کتاب « یک الگوریتم سریع برای محاسبه طولانی ترین نتایج عمومی » نوشته [w. هانت و G.T سیزمانسکی، CACM، ماه می 1977 مراجعه کنید). الگوریتم diff، در کتاب D.M میلکوری و W.J هانت با عنوان « یک الگوریتم برای مقایسه فایل متفاوت»، 41 گزارش علمی

و تکنیکی بل لابر، 1976، توصیف می شود. به نقل قول از میکلوری « من حداقل سه الگوریتم کاملاً متفاوت را قبل از الگوریتم نهایی بررسی کرده ام. diff یک مورد ناب و اصیل نه فقط برای تعیین صلاحیت - محض در یک برنامه می باشد، بلکه همچنین آن را مجدداً بررسی میکنید تا جایی که صحیح باشد.»

فصل هشتم فراخوانیهای سیستمی

این فصل بر پائین ترین سطح برهمکنش با سیستم عامل یونیکس تاکید دارد - فراخوانیهای سیستم. این فراخوانیها، ورودیهای کنترل هستند آنها مسیرهای ساده ای هستند که سیستم عامل فراهم می کند و هر چیز دیگری بر بالای آنها ساخته می شود. ما بخش های متعدد مهمی را تحت پوشش قرار می دهیم. اولین بخش، سیستم I/O است، زیر بنایی تحت زیر برنامه های کتابخانه مانند pntc, fopen. ما در خصوص سیستم فایل، بویژه فهرست ها و winod بیشتر صحبت خواهیم کرد. بعدا به بحث و بررسی در خصوص فرآیندها می پردازیم. چگونه برنامه ها از درون یک برنامه اجرا می شوند. پس از آن، ما در خصوص علامت ها و وقفه ها صحبت می کنیم:

چه اتفاقی می افتد زمانی که شما کلید Delete را فشار می دهید و چگونه از آن به طور معقول در یک برنامه استفاده می کنیم. همانند فصل 6 بسیاری از مثالهای ما، برنامه های مفیدی هستند که بخشی از ویرایش هفتم نمی باشند. حتی اگر آنها مستقیما برای شما مفید نباشند، شما باید چیزی از خواندن آنها یاد بگیرید و آنها باید ابزار مشابهی را بیان کنند که شما می توانید برای سیستم خود بسازید.

جزئیات کامل در مورد فراخوانیهای سیستم در بخش 2 از کتاب راهنمای برنامه نویس یونیکس وجود دارند و این فصل، مهمترین بخشها را توصیف می کند، اما در مورد تمامیت آن چیزی ارائه نمی دهد.

7-1 I/O دارای سطح پائین:

پائین ترین سطح I/O، ورود مستقیم به سیستم عامل می باشد. برنامه شما فایلها را در قطعاتی با اندازه مناسب می خواند و می نویسد. کرنل، داده های شما را در قطعاتی، میانگیر می ند که طرحهای پیرامونی و عملیات های برنامه ها را بر روی طرحها، به منظور بهینه کردن عملکرد آزاد برای همه کاربرها تطبیق دهند.

توصیف گران فایل

همه ورودی و خروجی از طریق خواندن و نوشتن فایلها انجام می شود، چون همه طرحها پیرامونی، حق پایانه شما، فایلهایی در سیستم فایل می باشند. مفهوم این عبارت این است که یک اتصال منفرد همه ارتباط بین یک برنامه و طرحهای پیرامونی را انجام می دهد.

در عمومی ترین مورد، قبل از خواندن یا نوشتن یک فایل، لازم است که سیستم خود را برای انجام آن، مطلع سازید، فرآیندی که باز کردن فایل نامیده می شود. اگر شما قصد نوشتن بر روی یک فایل را دارید. بوجود آوردن آن فایل نیز می تواند لازم باشد. سیستم صحت انجام کار شما را کنترل می کند (آیا فایل خارج می شود؟ آیا شما مجاز به دستیابی به آن هستید؟) و اگر همه چیز درست باشد. یک عدد صحیح مثبت را که توصیف گر فایل نامیده می شود، باز می گرداند. هر موقع که I/O بر روی فایل انجام شود، توصیف گر فایل برای شناسایی فایل، به جای اسم استفاده می شود. همه اطلاعات در مورد یک فایل باز. توسط سیستم حفظ می

شود و برنامه شما ، فقط توسط یک توصیف گر فایل ، به فایل رجوع می کند . یک اشاره گر FILE به همان صورتی که در فصل 6 توصیف شد . به ساختاری اشاره می کند که در میان سایر چیزها دارای توصیف گر شد . به ساختاری اشاره می کند که در میان سایر چیزها ، دارای توصیف گر فایل می باشد . `fileno(ff) macro` ، که در `<stdio.h>` تعریف می شود . توصیف گر فایل را باز می گرداند .

آرایش های خاصی برای مناسب ساختن ورودی و خروجی پایانه وجود دارند . زمانی که یک برنامه توسط شل شروع می شود ، سه فایل باز را با توصیف گران فایل 0,1 و 2 بدست آورد که ورودی استاندارد ، خروجی استاندارد و خطای استاندارد نامیده می شوند . هر سه مورد ، توسط پیش فرض به پایانه متصل می شود بنابراین اگر یک برنامه فقط توصیف گر فایل 0 را بخواند و توصیف گران فایل 1 و 2 را بخواند ، می تواند 1/0 را بدون باز کردن فایلها بخواند . اگر برنامه فایلهای دیگر را باز کند ، آنها دارای توصیف گران فایل 3 و 4 و غیره خواهند بود .

اگر 1/0 به سمت فایلها یا لوله ها تغییر جهت دهد و یا از آنها خارج شود ، شل ، تخصیص های پیش فرض را برای توصیف گران فایل 0 و 1 از پایانه به سمت فایلهای نام گذاری شده تغییر می دهد. در حالت عادی ، توصیف گر فایل 2 ، متصل به پایانه باقی می ماند ، در نتیجه پیامهای خطا می توانند به آنها بروند . عملکردهای شل ، مانند `filename 2 & 1` و 2 ، منجر به آرایشهای پیش فرضها می شوند ، اما تخصیص های فایل توسط شل تغییر می کنند و نه توسط برنامه . (برنامه خودش می تواند این تخصیص ها را مجددا مرتب کند ، اگر بخواهیم ، اما این مورد نادر است .

I/O فایل – خواندن و نوشتن

تمام ورودی و خروجی توسط دوفراخوانی سیستم انجام می شود ، `read` ، `write` ، که از `c` توسط عملیات هایی با همین نام ، دستیابی می شوند . برای هر دو ، اولین آرگومان ، توصیف گر فایل است . دومین آرگومان ، یک آرایه از بایتهایی می باشد که به عنوان منبع یا مقصد داده ها ارائه می شود . سومین آرگومان ، تعداد بایتهایی است که باید منتقل شوند .

```
Int fd,n,read , written;
Char buf [SIZE];
Nread = read (fd,buf,n);
Nwritten=write (fd , buf,n);
```

هر فراخوانی ، یک شما را از تعداد بایتهای انتقال یافته را باز می گرداند . در خواندن ، تعداد بایتهای بازگردانده شد ، ممکن است کمتر از تعداد بایتهای درخواست شده باشد ، چون ، کمتر از `n` بایت برای خواندن باقی می ماند . (زمانی که فایل یک پایانه است ، `red` در حالت عادی ، فقط تا سطر بعدی را می خواند که معمولا کمتر از چیزی است که درخواست می شود.) ارزش بازگشت صفر ، در انتهای فایل ایجاب می کند و 1- ، یک خطا را نشان می دهد . برای نوشتن ، ارزش بازگردانده شده ، تعداد بایتهایی است که حقیقتا نوشته می شوند و یک خطا رخ می دهد اگر این تعداد ، برابر با تعداد بایتهای در نظر گرفته شده برای نوشتن نباشند .

زمانی که تعداد بایتهایی که باید خوانده یا نوشته شوند ، محدود نمی شود ، عمومی ترین ارزشها 1 ، که به معنای یک کاراکتر در یک زمان می باشد (میانگیر نشده) و اندازه یک بلوک بر روی یک دیسک که اغلب دارای 512 یا 1024 بایت می باشد ، (پارامتر `BUFSIZ` در `<stdio.h>` دارای این ارزش می باشد) هستند .

برای شرح ، در اینجا برنامه ای وجود دارد که ورودی خود را برای خروجی خود کپی می کند . چون ، ورودی و خروجی می توانند

برای هر فایل یا طرحی مجددا تغییر جهت دهند، این برنامه ، حقیقتا هر چیزی را برای چیزی کپی می کند و این برنامه یک تحقق چارچوب اصلی cat می باشد

```
/*cat:minimal version*/
#define SIZE 512/*arbitrary*/
main[1
.
char buf[SIZE]
int n;
while ((n=read (0,buf , size of buf))>0)
write (1,buf,n);
exit (0) ;
```

اگر اندازه فایل ، یک مضرب از SIZE نباشد ، برخی از read ، تعداد کمتری از بایتهایی را که باید توسط write نوشته شوند ، باز می گرداند و فراخوانی بعدی برای read که پس از آن ، صفر بر می گردد .

خواندن و نوشتن در قطعاتی که دیسک را تطبیق می کنند ، بسیار موثر می باشد، اما حتی I/O به صورت یک کاراکتر در یک زمان ، برای میزان ناچیزی از مقادیر داده ها ، امکان پذیر است . چون کرنل ، داده های شما را میانگیر می کند و ارزش اصلی ، فراخوانی های سیستم می باشد . برای مثال ed از خواندن های یک بایتی برای بازیابی ورودی استاندارد آن ، استفاده می کند . ما این نسخه از cat را روی یک فایل با 54000 بایت زمان دادیم ، برای 6 ارزش از SIZE :

Tim (user + system, sec)

SIZE	Pdp= 11.70	Vax-11.750
1	0/271	8/188
10	9/29	3/19
100	8/3	6/2
512	3/1	0/1
1024	2/1	6/0
5120	0/1	6/0

اندازه بلوک دیسک ، 512 بایت بر روی سیستم PDP-11 و 1024 بایت بر روی VAX می باشد . تقریبا برای فرآیندهای متعدد مجاز می باشد که در یک زمان به یک فایل دست یابند و حقیقتا ، یک فرآیند می تواند نوشته شود ، زمانی که فرآیندی دیگر خوانده می شود اگر این چیزی نباشد که شما می خواهید می توان نگران کننده باشد ، اما گاهی اوقات مفید است . اگر چه ، یک فراخوانی برای read ، صفر را باز می گرداند و در نتیجه علامت های انتهای فایل را باز می گرداند ، اما اگر داده های بیشتری بر روی آن فایل نوشته شود ، read بعدی ،بایتهای بیشتری را در دسترس می یابد . این مشاهده ، مبنای برنامه ای است که redslow نامیده می شود برنامه ای که خواندن ورودی خود را، بدون توجه به این که آیا به انتهای فایل می رسد یا نه ، ادامه می دهد . redslow برای تماشای پیشرفت یک برنامه مناسب است .

```
#slowprog > tem
5213 process - id
# redslow <tem: grep sometting
```

به عبارت دیر کی برنامه کند که خروجی را در یک فایل تولید می کند ؛ redslow و شاید در همکاری با برنامه ای دیگر ، انباشته

شدن داده ها را مشاهده کند .

از نظر ساختاری، `redslow` مشابه با `cat` می باشد، به استثنای اینکه، به جای خارج شدن از سیستم، حلقه سازی می کند، زمانی که با انتهای جریان ورودی مواجه می شود. `redslow` باید از `I/O` دارای سطح پائین استفاده کند، چون زیر برنامه های کتابخانه استاندارد، پس از اولین پایان فایل، `EOF` را گزارش می دهند .

```
/*redslow:keep reading, waiting for more*/
# define SIZE 512/*arbitrary */
main[]
char buf[SIZE]
int n ;
for(; ;){
while ((n=read (0,buf , sizeof buf)) >0)
write (1,buf , n);
sleep (10);
}
```

تابع `sleep` باعث می شود که برنامه برای چند ثانیه خاص، مسکوت باقی بماند و این تابع در `sleep (3)` توصیف می شود. ما نمی خواهیم `redslow`، در فایل، به خاطر جستجوی مداوم برای داده های بیشتر، بسته شود، چون چنین چیزی در زمان `cpu` بسیار پرهزینه خواهد بود. بنابراین این نسخه `redslow`، ورودی خود را تا انتهای فایل کپی می کند، اندکی می خوابد و سپس دوباره تلاش می کند. اگر داده های بیشتری برسند، زمانی که این نسخه خواب است. توسط `read` بعدی خوانده خواهد شد .

تمرین 1-7. یک آرگومان `n` به `redslow` اضافه کنید، سپس زمان خواب پیش فرض می تواند تا `n` ثانیه تغییر کند. برخی از سیستم ها یک انتخاب `f-` (برای همیشه) برای `tail` تهیه می کنند که تابعهای `tail` را با تابعهای `redslow` ترکیب می کند. در خصوص این طرح توضیح دهید .

تمرین 2-7. چه اتفاقی برای `redslow` می افتد اگر فایلی که خوانده می شود کوتاه شود؟ چگونه شما آن را ثابت می کنید؟ نکته در خصوص `fstat` در بخش 3-7 مطالعه کنید .

ایجاد فایل – بازکردن، ایجاد کردن، بستن، مواردی متفاوت

غیر از فایل های پیش فرض ورودی، خروجی و خطای استاندارد، شما باید آشکارا، فایلها را به منظور خواندن یا نوشتن آنها باز کنید. دو فراخوانی سیستم برای این کار وجود دارند، `open`، `creat` .

`Open` تا حدودی شبیه به `fopen` در فصل قبل می باشد، به استثنای اینکه به جای بازگرداندن یک اشاره گر فایل، یک توصیف گر فایل را بر می گرداند که یک `int` می باشد .

```
Char*name ;
Int fd , rwmode;
Fd=open (name , rwmode);
```

همانند `fopen`، آرگومان `name`، یک رشته کاراکتر و شامل نام فایل می باشد. اما دستیابی به آرگومان `mode` متفاوت است: `rw` `mode` برای خواندن صفر، برای نوشتن 1 و برای بازکردن یک فایل چه برای خواندن و چه برای نوشتن 2 می باشد. `1-`، `open` را باز می گرداند، اگر خطایی رخ دهد و در غیر این صورت یک توصیف گر بهتر فایل را باز می گرداند .

تلاش برای بازکردن فایلی که وجود ندارد یک خطاست. فراخوانی سیستم `creat`، برای ایجاد فایل های جدید و یا خواندن مجدد

فایل‌های متنی قبلی، تهیه می‌شود.

Int perms ;

Fd = creat (name , perms);

Creat، یک توصیف گر فایل را بر می‌گرداند اگر قادر به ایجاد فایل متنی با عنوان name باشد و اگر قادر به چنین کاری نباشد، -1 را برمی‌گرداند. اگر فایل وجود نداشته باشد Creat آن را با اجازه های مشخص شده توسط آرگومان perms بوجود می‌آورد. اگر فایل از قبل وجود داشته باشد. Creat طول آن را تا صفر کوتاه می‌کند؛ این کار یک خطا برای ایجاد فایلی که از قبل وجود دارد، نمی‌باشد (اجازه ما، تغییر نخواهد کرد) بدون توجه به perms یک فایل ایجاد شده، برای نوشتن باز می‌باشد.

همانگونه که در فصل 2 توصیف شد. نه بیت از اطلاعات پشتیبانی همراه با یک فایل وجود دارند که خواندن، نوشتن و اجازه اجرا را کنترل می‌کنند، بنابراین یک عدد هشت هشتی سه رقمی برای مشخص کردن آنها مناسب است. برای مثال 0755 خواندن. نوشتن و اجازه اجرا را برای مالک مشخص می‌کند و نیز اجازه خواندن و اجرا کردن را برای گروه و هر کسی دیگری مشخص می‌کند. عدد صفر اصلی را فراموش نکنید، که چگونگی مشخص شدن اعداد هشت هشتی در C می‌باشد.

برای شرح در اینجا یک نسخه ساده شده از cp وجود دارد. سادگی اصلی آن، این است که نسخه‌ها فقط یک فایل را کپی می‌کند و به دومین آرگومان اجازه نمی‌دهد که یک فهرست باشد. عیب دیگر این است که نسخه‌ها، جوازهای فایل مبدا را حفظ نمی‌کند؛ ما نشان می‌دهیم که چگونه این مورد را چاره کنیم.

```

/*cp:minimal version*/
#include cstdio-hs
# define PERMS 0644 /*RW for owner , R for group , others*/
char * progname ;
main(arg,argv)/*cp:copy fito f2 */
int argc;
char * grgv [ ];
int f1, f2 , ;
char buf [BUFSIZ];
programe = qrgv [0]
if (argc !=3)
error (usage:%s from to , progname);
if ((f 1= open (argv [1] , 0))=-1
error (cant create %s , argv[2]);
while ((n=read (f,buf , Bufsiz))>0)
if (write error , (char *)0);
exit (0);
}

```

ما error را در بخش فرعی بعد مورد بحث و بررسی قرار می‌دهیم.

در اینجا یک محدودیت (اساساً حدود 20 به مشاهده NOFILE در <sys/param.hs> پردازید)، در تعداد فایل‌هایی وجود دارد که یک برنامه می‌تواند به طور همزمان باز کند. بر همین اساس، هر برنامه‌ای که قصد دارد فایل‌های زیادی را پردازش کند، باید برای استفاده مجدد از توصیف گران فایل آماده باشد. فراخوانی سیستم close، ارتباط بین یک اسم فایل و یک توصیف گر فایل را از بین می‌برد و توصیف گر فایل را به منظور استفاده توسط دیگری، آزاد می‌کند. اتمام یک برنامه از طریق exit یا بازگشت از برنامه اصلی، همه فایل‌های باز را می‌بندد.

فراخوانی سیستم `unlike` یک فایل را از سیستم فایل پاک می کند .

پردازش خطا – `error`

فراخوانی های سیستم مورد بحث واقع شده در این بخش ، و در حقیقت همه فراخوانیهای سیستم می توانند منجر به خطا شوند .

معمولا آنها ، یک خطا را با برگرداندن یک ارزش 1- نشان می دهند . گاهی اوقات فهمیدن اینکه چه خطای ویژه ای رخ داده است .

خوب است برای این منظور همه فراخوانیهای سیستم ، زمانی که مناسب است یک عددخطا را در یک عدد صحیح خارجی جا می

گذارند که `error` نامیده می شود .

(معنی اعداد متعدد خطا، در مقدمه بخش 2 از کتاب راهنمای برنامه نویسی یونیکس، فهرست وار بیان می شود) با استفاده از `error`

برنامه شما برای مثال می تواند ، تعیین کند که آیا تلاش برای بازکردن یک فایل با شکست مواجه شده است ، به خاطر اینکه فایل

وجود نداشته است و یا به خاطر اینکه شما مجاز به خواندن آن نبوده اید .

همچنین یک آرایه از رشته های کاراکتر `sys-errlist` وجود دارد که توسط `error` مشخص می شود و اعداد را به یک رشته معنی دار

ترجمه می کند . نسخه `error` ما ، از این ساختار داده ها استفاده می کند :

```
Error (s1 , s2) /*print error message and die */
Char * s1,*s2 ;
{
extern int error , sys-nerr ;
extern char *sys- errlist [ ] ,*programe
if (programe)
fprintf(stderr , %s , programe);
fprintf (stderr,s1,s2);
if (errno>&&errno<sys-nerr)
fprintf (stderr,(% s ) , sys-errlist[errno]);
fprintf f(stderr,in);
exit (1);
}
```

`errno` ، در ابتدا صفر است و باید همیشه کمتر از `sys-nerr` باشد . `errno` برای صفر ریست نمی شود . زمانیکه همه چیز خوب است

. اما شما باید آن را پس از هر `error` ریست کنید اگر می خواهید برنامه خود را ادامه دهید .

در اینجا ، چگونگی آشکار پیامهای خطا با این نسخه از `cp` وجود دارد :

```
&cp foo bar
cp:cant open foo (No such file or directory)
& date>foo ;chmod o foo
& cp foobar
cpi : cant open foo (permission denied)
```

&

دستیابی تصادفی – lseek

I/O فایل در حالت عادی تربیتی است : هر خواندن یا نوشتن در داخل فایل درست سپس از مورد قبلی رخ می دهد . اما زمانی که لازم است ، یک فایل می تواند در یک ترتیب قراردادی خوانده یا نوشته شود . فراخوانی سیستم `lseek` ، راهی را برای وارد شدن در یک فایل بودن خواندن یا نوشتن واقعی فراهم می کند .

```
Int fd , origin ;
Long offset , pos, lseek();
Pos=lseek (fd , offset , origin);
```

موقعیت فعلی در فایل را که توصیف گر آن `fd` می باشد مجبور به حرکت به سمت موقعیت `offset` می کند ، که متناسب با موقعیت مشخص شده توسط `origin` می باشد . خواندن یا نوشتن بعدی در آن موقعیت شروع می شود . `origin` می تواند ، 0 و 1 یا 2 باشد برای اینکه مشخص کند `offset` از آغاز ، از موقعیت قطعی جدید یا -1 برای یک خطا می باشد . برای مثال برای فیحصه کردن به یک فایل ، قبل از نوشتن در جستجوی انتهای فایل باشید :

```
Lseek (fd , ol,2);
```

برای بازگشتن به آغاز (باز پیچش) و

```
Lseek (fd , ol ,0) ;
```

برای تعیین موقعیت فعلی :

```
Pos = lseek (fd,ol,1);
```

توجه داشته باشید به آرگومان `ol` : `offset` به یک عدد صحیح طولانی می باشد . (1 در `lseek` به منظور تشخیص آن از ششمین ویرایش سیستم فراخوانی `seek` می باشد که از اعداد صحیح کوچک استفاده می کند .) با `seek` : و این امکان وجود دارد که با فایلها کم و بیش شبیه آرایه های بزرگ به قیمت دستیابی کندتر، برخورد کنیم . برای مثال ، تابع زیر هر عدد از بایتها را از سرمکانی در یک فایل می خواند .

```
Get (fd , pos , buf , n)/*read n bytes from position pos*/
```

```
Int fd,n;
```

```
Long pos
```

```
Char*buf;
```

```
If (lseek(fd ,pos,0)=-1) /*yet to pos*/
```

```
Return -1;
```

```
Else
```

```
Return read (fd,buf ,n);
```

```
}
```

تمرین 3-7. برای استفاده از یک آرگومان فایل اگر وجود دارد ، `readslow` را تغییر دهید انتخاب `e` را اضافه کنید :

&e-readslow

منجر می شود که `readslow` در جستجوی انتهای ورودی قبل از شروع به خواندن باشد . `lseek` بر روی یک لوله چه کاری انجام می

دهد ؟

تمرین 4-7. efopen از فصل 6 برای فراخوانی error مجددا بنویسید .

2-7 سیستم فایل : فهرست های راهنما

موضوع بعدی ، چگونگی حرکت از طریق سلسله مراتب فهرست راهنما می باشد . برای چنین چیزی در حقیقت از فراخوانیهای جدید سیستم استفاده نمی شود ، فقط برخی از فراخوانیهای قبلی در یک متن جدید قرار می گیرند . ما با نوشتن یک تابع باعنوان `spname` شرح می دهیم که چگونه بر اسامی فایل‌های دارای تلفظ غلط غلبه کنیم . تابع

```
N=spname (name ,new name);
```

به جستجوی یک فایل بایک نام-« به میزان کافی نزدیک» به منظور نامگذاری آن می پردازد . اگر یک فایل یافت شود. درون `newname` کپی می شود . ارزش `n` بازگردانده شده توسط `1-` ، `spname` می باشد اگر هیچ فایلی که به میزان کافی نزدیک باشد ، یافت نشود . ارزش `0 n` می باشد اگر یک تطبیق واقعی وجود داشته باشد و `1` است اگر یک تصحیح ساخته شود .
`Spname` یک افزایش مناسب به فرمان `P` می باشد . اگر شما سعی کنید که یک فایل را تایپ کنید اما اسم آن (غلط تلفظ کنید، `p` می تواند از شما سؤال کند اگر شما واقعا منظورتان چیز دیگری است :

```
&p/urs/.YX/comd/p/spnam.c
```

```
/usy/syc/cmd/p/spname.c:y
```

```
/*spname: return correctly spelled filename*/
```

همانگونه که ما نوشتیم ، `spname` در هر جزء از اسم فایل ، سعی می کند اشتباهاتی را که در آنها یک حرف تنها افتاده است یا اضافه شده است یا تنها یک حرف غلط است و یا یک جفت از حروف با هم جابجا شده اند تصحیح کند و همه این موارد در فرمان بالا شرح داده می شوند . این یک لطف برای تایپیست های شلخته است .

قبل از نوشتن رمز، یک بررسی کوتاه از ساختار سیستم فایل ، اشکالی ندارد . یک فهرست راهنما، فایلی است که شامل یک فهرست از اسامی فایلها و یک نشانه از جایی است که آنها قرار دارند «مکان» حقیقتا یک شاخص در جدولی دیگر می باشد که جدول `inode` نام دارد . `inocle` برای یک فایل جایی است که همه اطلاعات در خصوص فایل به جز اسم آن ، حفظ می شود . یک مدخل فهرست راهنما، متشکل از فقط دو مورد می باشد . یک عدد `inode` و یک اسم فایل . تشخیص دقیق را می توان در فایل `<sys>dir.h` پیدا کرد :

```
& cat/usr/include / sys/dir.h
```

```
#define DIRIZ 14 /*max length of file name*/
```

```
struct direct /*structure of directory entry */
```

```
{
```

```
ino-td- ino; /*inode number */
```

```
char d-name [DIRSIZ] ; /*file name */
```

```
};
```

```
&
```

تایپ `ino-t` یک `typedef` می باشد که شاخص را درون جدول `inode` توصیف می کند . `ino-t` ، به صورت یک `short` بدون علامت بر روی نسخه های `PDP-11` و `VAX` از سیستم می باشد ، اما چنین چیزی قطعا گونه ای از اطلاعات واقع شده در یک برنامه نمی باشد : بلکه بر روی یک ماشین متفاوت ، متفاوت می باشد . از این رو ، `typedef` ، یک مجموعه کامل از تایپ های سیستم در >

<sys/types.b> یافت می شود ، جایی که باید قبل از <sys/dir.b> باشد .

عملکرد `spname` ، در مسیر مستقیم است ، اگر چه ، موقعیت هایی زیادی برای رفتن به سمت راست وجود دارد . فرض کنید اسم فایل `d1/ d2 / f/` می باشد . عقیده اصلی ، جدا کردن اولین جزء (/) می باشد، سپس جستجوی فهرست راهنما، برای اسمی است که نزدیک به جزء بعدی (`d1`) می باشد، سپس جستجوی فهرست راهنما، برای چیزی نزدیک `d2` می باشد و به همین ترتیب ، تا جایی که یک تطبیق برای جزء یافت شود . اگر در هر مرحله به یک داوطلب قابل قبول ، در فهرست راهنما نباشد ، جستجو متوقف می شود . ما این کار را به سه تابع تقسیم کرده ایم `spname` خودش اجزاء مسیر را از هم جدا می کند و آنها را در یک اسم فایل « تا اندازه ای دارای بهترین تطبیق می سازد . جزء جدید `mindist` را فرامی خواند که به جستجوی یک فهرست راهنمای مشخص برای فایل می پردازد که به مدل های فعلی ها نزدیک می باشد و با استفاده از یک تابع سوم `spdist` فاصله بین دو اسم را محاسبه می کند .

```

/*spname: return correctly spelled filename*/
/*
*spname (oldname , newname)char *oldname ,*newname;
*returns -1 if no reasonable match to oldname,
* 0 if exact match,
* 1 if corrected .
* stores corrected name in newname
*/
# include <sys/ types.b>
# include <sys /dir.b>
spname (oldname,newname)
char*oldname,*newname;
{
char *p,guess [DIRSIZ+], best [DIRSIZ+1];
char * new = newname , *old = oldname;
for( ; ; ){
while (*old==/)/ *skip slashes*/
new++=old ++;
*new =\0
if (*old=10) /*exact or corrected */
return strcmp (oldname,newname) !=0;
p=guess ;/*copy next component into guess*/
for ( ; *old!=/ &&*old !=10,old ++)
if (p<guess +DIRSIZ)
*P++=*old ;
*p= IO;
if (mindist (newname , guess,best)>3)
return-1 , /*hopless */
for (p=best ; new =p++;)/ *add to
/*of newname*/
}
mindist(dir,guess,best)/ *search dir for guess*/
char dir,guess,*best;
{
/*set best , return distance 0003*/

```

```

int d , nd , fd;
struct{
ino_tino ;
char name [DIRSIZ+1] ;/*1 more than in dir.b*/
}nbuf ;
nbuf . name [DIRSIZ]=IO ;/*+1 for terminal io */
if (dir [0] == 0) /*current directory */
dir =0;
d=3; /*minimum distance*/
if ((fd = open(dir , 0)) == -1)
return di
while (read (fd,(char *)&nbuf , sizeof(structdirect))>0)
if(nbuf.ino){
nd = spdist (nbuf name,guess);
if(nd<=d&nd !=3){
strcpy(best , nbuf . name);
d=nd;
if (d==0) /*exct match*/
break ;
}
}
close (fd) ;
returnd ;
}

```

اگر نام فهرست راهنمای ارائه شده برای mindist خالی باشد ، 0 جستجو می شود mindist یک ورودی فهرست راهنما را در هر زمان می خواند . توجه داشته باشید که میانگیر برای read یک ساختار می باشد نه یک آرایه از کاراکترها ما از sizeof برای محاسبه تعداد بایتها و حرکت آدرس به سمت یک اشاره گر کاراکتر ، استفاده می کنیم .

اگر یک شکاف در یک فهرست راهنما ، اخیرا مورد استفاده نباشد (چون یک فایل حذف شده است) در نتیجه مدخل cnode صفر است و از این موقعیت به صورت جهشی عبور می شود . تست فاصله عبارت است از

If (nd <=d...)

به جای

if (nd <d...)

بنابراین ، هر کاراکتر مفرد دیگری یک تطبیق بهتر از 0 می باشد که همیشه اولین ورودی در یک فهرست راهنما است .

/*spdist : return distabce between two names*/

/*

*very rough spelling metric:

*0 if the string are identical

* 1 if the string are identical

*2 if one char wrong,added or deleted

*3 otherwise

*/

define E& (s,t)(strcmp (s,t)==0)

spdist (s,t)

char*s,*t;

```

{
while (*s++==*t)
if (*t++== io)
return 0 ; /*exact match */
if (*--s){
if (*t){
if (s[1]&&t [1]&&*s == t [1]
&&*t == s[1]&&E* (S+2 , t+2))
return 1; /*transposition */
if (Ea (s+1 , t+1)
return 2 ; /*1 char mismatch */
}
if (Ea(s+1, t))
return 2; / *extra character */
}
if (*t&& E&(S,t+1))
return 2; /*missing character */
return 2 ;
}

```

زمانی که ما spname را داریم ، ادغام تصحیح تلفظ در p آسان است :

```

/*p:print input in chunks (version 4 )*/
# include cstdio.bs
# define PAEGSIZE 22
char *programe ; /*programname for error message */
main (argc , argv)
int argc ;
char*argv [ ] ;
{
FILE *FP,*efopen ( ) ;
Int ; pagesize= PAGESIZE ;
Char *p,get v ( ) , buf [BUFSIZ];
Programe = argv[0] ;
If ((p=getenv (PAGESIZE)) !=NULL)
Pagesize = atoi(p);
If (argc>1&&argv [1] [ 0] == - {
Pugesize = atoi(&argv [ 1] [0] );
Argc --;
Argv ++;
}
if (argc==1)
print (stdin ,pagesize);
else
for (I=1 , I<argc;I ++)
switch (spname (argv [I] ) , buf)){
case -1 ; /*nomatchpossible */
fp = efopen cargv [I] , r);
break;

```

```

casel: /*corrected */
fprintf(stderr, 1 %s \, buf);
if (ttyin () ==n)
break ;
argv [I] = buf ;
/*fall through...*/
case 0: /*exact match */
fp= fopen argv [I] , r);
print (fp,pagesize);
fclose (fp);
}
exit (0);
}

```

تصحیح تلفظ ، چیزی نیست که به صورت کورکورانه برای هر برنامه ای که از اسامی فایلها استفاده می کند ، بکار رود ، تصحیح تلفظ ، با p به خوبی کار می کند ، چون p برهم کنش می باشد ، اما برای برنامه هایی که برهم کنش نمی باشند ، مناسب نیست .

تمرین 5-7 . شما چقدر می توانید برای انتخاب بهترین تطبیق `sname` ، بر روی روشهای اکتشافی ، پیشرفت کنید ؟ برای مثال ، احمقانه است که با یک فایل منظم به گونه ای برخورد کنیم که گویا یک فهرست راهنما می باشد و چنین چیزی می تواند با نسخه فعلی رخ دهد .

تمرین 6-7 . نام `tx` ، هر گونه `tc` را که در انتهای فهرست راهنما رخ می دهد ، برای هر کاراکتر `c` ، تطبیق می کند . آیا شما می توانید یک ارزیابی بهتر از فاصله را اختراع کنید ؟ آن را اجرا کنید و مشاهده کنید که چگونه با کار برهای واقعی کار می کند .

تمرین 7-7 . `mindist` در هر زمان یک مدخل از فهرست راهنما را می خواند . آیا `p` به صورت هوشمندانه سریعتر اجرا می شود ، اگر خواندن فهرست راهنما، در مقامات بزرگتر انجام شود:

تمرین 8-7 . `sname` را به گونه ای تغییر دهید که نامی را بازگرداند که یک پیشوند از یک نام دلخواه باشد ، اگر تطبیق نزدیکتری یافت نشود . چگونه پیوندها باید شکسته شوند، اگر اسامی متعددی وجود داشته باشند که همگی پیشوند را تطبیق کنند ؟

تمرین 9-7 . چه برنامه های دیگری ، می توانند از `sname` بهره مند شوند به یک برنامه خود اتکا را طراحی کنید که تصحیح را برای آرگونهای خود بکار برد ، قبل از اینکه آنها را در طول برنامه عبور دهند ، مانند

`...fix progfilenames&`
 آیا شما می توانید یک نسخه از `cd` را بنویسید که از `sname` استفاده کند که چگونه آن را نصب می کنید ؟

3-7 سیستم فایل : *inodes*

در این بخش به ما بحث در خصوص فراخوانیهایی از سیستم می پردازیم که به سیستم فایل و بویژه با اطلاعاتی در خصوص فایلها ، مانند اندازه ، تاریخ ها ، اجازه ها و مواردی از قبیل می پردازند . این فراخوانیهای سیستم به شما این امکان را می دهند که همه اطلاعاتی را بدست آورید که ما در خصوص آنها در فصل 2 صحبت کردیم .

می خواهیم وارد خود `inodes` شویم . بخشی از `inode` توسط یک ساختار با عنوان `stat` توصیف می شود که در `<sys /stat .b>` تعریف می شود :

Struct stat /*structure returned by stat */

}

Dev-t	St-dev;	/*device of inode */
Ino-t	St-ino;	/*inode number*/
short	St-mod;	/*mode bits*/
short	St-nlink;	/*number of links tofile*/
short	St-uid ;	/*owners u serid*/
short	St-gid ;	/*owners groug id */
Dev-t	St - rder ;	/*for special files*/
Off-t	St - size ;	/*file size in characters*/
Tim-t	St-atime;	/*time file last read */
Time-t	St-mtime;	/*time file last wnttenor created*/
Time-t	St-time;	/*time file or inode last changed*/

}

اکثر فایلها توسط کافت ها ، توضیح داده می شوند . تایپهایی مانند dev-t , ino-t در <sys/stat.b> تعریف می شوند ، همانگونه که بالا توصیف شد . ورودی st-mode شامل یک مجموعه از پرچم هایی است که فایل را توصیف می کنند و برای سهولت ، تعاریف پرچم نیز بخشی از فایل <sys/stat.b> می باشند :

# define	s-IFMI	0170000	/*type of file */
# define	S-IFDIR	0040000	/*directory */
# define	S-IFCHR	0020000	/*characterspecial*/
#define	S-IFDLK	0060000	/*block special*/
#define	S-IFREG	0100000	/*regular*/
#define	S-ISUID	0004000	/*set user id on execution*/
#define	S-ISGID	0002000	/*set group id on execution */
#define	S-ISVTX	0001000	/*save swapped text even after use*/
#define	S-IREAD	0000400	/*read permission , owner */
#define	S-IWRITE	0000200	/*write permission,owner*/
#define	S-IEXEC	0000100	/*execute/search permission,owner*/

Inode برای یک فایل ، توسط یک جفت از فراخوانیهای سیستم با نامهای stat , fstat پردازش می شود . stat یک اسم فایل را می گیرد و اطلاعات inode را برای آن فایل باز می گرداند (یا 1- را باز می گرداند اگر یک فقط خطا وجود داشته باشد) .

Fstat ، همان کار را از یک توصیف گر فایل برای یک فایل باز انجام می دهد (که از یک اشاره گر file)، که به این صورت است :

```
Char *name;
Int fd;
Struct stat stbuf ;
Stat (name ,& st buf);
Fstat (fd , & st buf );
```

ساختار stbuf را با اطلاعات inode برای اسم فایل و توصیف گر فایل fd پر می کند .

با وجود همه این حقایق ، ما می توانیم نوشتن برخی از کدهای مفید را آغاز کنیم . اکنون با یک نسخه c از checkmail آغاز می کنیم ، برنامه ای که صندوق پستی شما را نگاه می کند . اگر فایل بزرگتر شود ، checkmail ، عبارت « شما پست الکترونیکی داریو » را پرینت می کند و زنگ را به صدا در می آورد . (اگر فایل کوتاه تر شود ، از قرار معلوم ، به خاطر این می باشد که شما نامه پستی را خوانده و حذف کرده اید و پیغامی درخواست نمی شود) . چنین چیزی کاملاً به عنوان قدم اول مناسب است و شما می توانید ممتاز تر باشید زمانی که این برنامه کار می کند .

```
/* checkmail :watch users mailbox*/
# include <stdio .bs>
# include <sys / types.bs>
# include <sys/ stat-h>
char * progname ;
char *maildir = /usr/spool/mail , /*sys tem dependent*/
main (argc,argv)
int argc;
char *argv [ ] ;
{
struct stat buf ;
char *name , * getlogin ( ) ;
int lastsize=0
progname = argv [0] ;
if ((name = getlogin ( )) == null)
error (cant get name , (char*)0);
if (chdir (maildir)== -1)
```

```

error (cant cd to % s , muildir );
for ( ; ; ) {
if (stat (name , & buf) == -1)/k no mailbox */
buf.st -size =0 ;
if (buf .st-size >last size)
fprintf (stderr,lnyou have mail loov ln);
lastsize = buf.st - size;
sleep (40) ;
}
}

```

تابع (3 getlogin) اسم Login شما را باز می گرداند و یا اگر نتواند null را باز می گرداند . chdir با فراخوانی سیستم chdir ، به فهرست پستی تغییر می کند ، بنابراین ، فراخوانیهای بعدی stat ، نباید هر فهرستی از ریشه را برای فهرست پستی جستجو کنند . شما باید maildir را به گونه ای تغییر دهید که بر روی سیستم شما تصحیح شود . ما check mail را نوشتیم برای اینکه بررسی کنیم آیا صندوق پستی وجود ندارد ، چون اکثر نسخه های mail ، صندوق پستی را حذف می کنند اگر خال باشد . ما این برنامه را در فصل 5 نوشتیم برای اینکه تا اندازه ای ، حلقه های شل را شرح دهیم .

این نسخه فرآیندهای متعددی را بوجود می آورد ، در هر زمان که به صندوق پستی نگاه می کند ، بنابراین می تواند بیشتر از بار سیستمی باشد که شما می خواهید . نسخه C یک فرآیند منفرد می باشد که یک stat را بر روی فایل در هر دقیقه انجام می دهد . چه قدر این فرآیند برای اجرای checkmail در زمینه درتمام زمان ، ارزش دارد؟ ما آن را به خوبی در یک ثانیه در هر ساعت اندازه گیری کردیم ، و میزان آن ، آنقدر پائین است که اهمیتی ندارد .

SV : یک شرح از کنترل خطا

ما بعدا قصد داریم برنامه ای را با عنوان SV بنویسیم که شبیه CP می باشد و یک مجموعه از فایلها را برای یک فهرست راهنما کپی می کند ، اما هر فایل مقصد را فقط زمانی تغییر می دهد که فایل وجود ندارد و یا قدیمی تر از مبدا می باشد . SV برای ذخیره کردن می باشد . ایده در اینجا این است که SV چیزی را که بیشتر جدید به نظر می رسد، روی هم کپی نمی کند . SV اکثر اطلاعات را در inode به جای checkmail استفاده می کند . طرحی که ما برای SV استفاده می کنیم عبارت است از :

```
& sv file 1 file 2... dir
```

فایل 1 را برای dir/file1 و فایل 2 را برای dir/file 2 و غیره کپی می کند ، به استثنای زمانی که یک فایل مقصد، جدیدتر از فایل مبدا آن می باشد ، هیچ گونه کپی انجام نمی شود و یک اخطار پرینت می شود . برای اجتناب از ایجاد کپی های متعدد از فایلها مرتبط ، SV در هیچ کدام از اسامی فایلها مبدا به S/ اجازه نمی دهد .

```

/*sv: save new files */
# include <stdio .h>
# include <sys/types.h>
# include <sys/stat .h>
# include <sys/stat .h>
char*programe ;
main(argc,argv)

```

```

int argc;
char *argv [ ] ;
{
int I ;
struct stat stbuf ;
char *dir = argv [argc-1]
progrname = argv [0] ;
if (argc<=2)
error (usage :%s files ...dir , progrname);
if (stat (dir,&stbuf)== -1
error (cant access directory % s, dir);
if ((st buf.st-mode &s-ifmt)! =s-ifdir)
error (%s is not a directory , dir);
for (I=1,I<argc-1 , I++)
sv(argv [I] , dir) ;
exit (0) ;
}

```

اول ژانویه ، 1970) ، بنابراین فایل‌های قدیمی تر GMT از مدتها قبل برحسب ثانیه می باشند (inode 0:00 زمانهای موجود در

خود می باشند st-mtime دارای ارزشهای کمتری در زمینه .

```

SV(file , dir) /*save file in dir*/
Char *file , * dir
{
struct stat sti , sto ;
int fin , fout , n;
char target [BUFSIZE] , buf [BUFSIZ], *index2
sprintf (target , %s/%s , dir , file) ;
if (index (file , /)! =Null)/*strchrin some systems */
error (wont handle/s in %s , file);
if (stat (file, &sti )=-1
error(cant't stat %s , file) ;
if (stat(target , &sto )=-1) /*target not present*/
sto.st - mtime =0 ; /* somake it look old */
if (sti.st-mtime< sto .st-mtime) /*target is newer */
fprintf (stderr , % s : % s not copied\n ,
progrname ,file ) ;
else if ((fin=open (file ,0) == -1 )
error (can't open file % s , file );
else if ((fout = creat (target , stist - mode)) == -1
error (can't create % s , target);
else
while ((n=read(fin , buf , sizeof buf ))>0)
if (write (fout , buf , n)! = n)
error (error writing % s , target);
close (fin) ;
close (fout ) ;
}

```

ما به جای تابعهای استاندارد I/O از creat استفاده کردیم ، در نتیجه sv می تواند از وضعیت فایل ورودی محافظت کند . (توجه داشته باشید که index , strchr نامهای متفاوت برای یک زیر برنامه می باشند و کتاب راهنمای خود را تحت 3 string) کنترل کنید برای اینکه ببینید سیستم شما از چه نامی استفاده می کند .)

اگر چه برنامه sv ، تا حدودی مشخص می باشد ، اما برخی از عقاید مهم را نشان می دهد . بسیاری از برنامه ها ، «برنامه سیستم» نمی باشند ، اما می توانند از اطلاعات ذکر شده توسط سیستم عامل استفاده کنند و به فراخوانی های سیستم دست یابند . برای چنین برنامه هایی لازم است که ارائه اطلاعات ، فقط از فایل های عنوان استاندارد مانند <dir . b> و <stat.h> ، ظاهر شود و این برنامه ها شامل آن فایلها می باشند ، به جای اینکه اعلان های واقعی را در خودشان قرار دهند . چنین رمزی ، به احتمال قوی ، قابل انتقال از یک سیستم به سیستمی دیگر می باشد .

همچنین ارزشمند است که حداقل در سوم رمز در sv ، کنترل خطا باشد . در مراحل اولیه نوشتن یک برنامه ، صرفه جویی در استفاده از خطا ، وسوسه انگیز می باشد ، چون یک انحراف از وظیفه اصلی می باشد . و زمانی که برنامه کار می کند ، اشتیاق در مورد برگشتن برای انتخاب بازبینی هایی که یک برنامه کار می کند ، اشتیاق در مورد برگشتن برای انتخاب بازبینی هایی که یک برنامه خصوصی را به برنامه ای تبدیل می کنند که بدون توجه به این که چه اتفاقی می افتد کار می کند ، دشوار است .

Sv ، یک شاهد در مقابل همه اشتباهات ممکن نمی باشد - sv در زمانهای نامناسب به وقفه های نمی پردازد - اما دقیق تر از اکثر برنامه ها می باشد . برای تمرکز بر روی فقط یک نکته در یک لحظه ، بیان write نهایی را در نظر بگیرید . خراب شدن write به ندرت رخ می دهد . بنابراین اکثر برنامه ها این احتمال را نادیده می گیرند . اما دیسکها از فضا خارج می شوند و کاربرها ، پافراتر از نقل قولها می گذارند ؛ خطوط ارتباطات شکسته می شود . همه این موارد می توانند منجر به خطاهای write شوند و شما بهتر عمل خواهید کرد اگر در خصوص آنها مطالبی بشنوید به جای اینکه برنامه به آرامی وانمودند که همه چیز خوب است .

درست این است که کنترل خطا خسته کننده اما مهم است . ما در اکثر برنامه های این کتاب به خاطر محدودیتهای مکانی و تاکید بر موضوعات جالب تر سرفراز بوده ایم . اما ، برای تولید واقعی برنامه ها ، شما نمی توانید خطاها را نادیده بگیرید .

تمرین 7-10 . برای تشخیص فرستنده پست ، به عنوان بخشی از پیام «شما نامه پستی دارید» chekemail را تغییر دهید . نکته

lseek-ss canf

تمرین 7-11 . chekmail را به گونه ای تغییر دهید که قبل از اینکه وارد حلقه خود شود ، برای فهرست پستی تغییری نکند . آیا این کار دارای اثر قابل اندازه گیری بر عملکرد خود می باشد ؟ (سخت تر) ، آیا شما می توانید یک نسخه از checkmail را به گونه ای بنویسید که فقط نیاز به یک فرآیند برای اطلاع به همه کاربرها باشد؟

تمرین 7-12 . یک برنامه watchfile به گونه ای بنویسید که یک فایل را بررسی کند و فایل را از آغاز هر زمانی که تغییر می کند ، پرینت کند . چه موقع شما می توانید از آن استفاده کنید ؟

تمرین 7-13 . sv در استفاده از خطای خود ، تقریباً سختگیر می باشد . آن را به گونه ای تغییر دهید که ادامه یابد ، اگر نمی تواند فایلی را پردازش کند .

تمرین 7-14 . sv را بازگشتی بسازید : اگر یکی از فایل های مبدا ، یک فهرست راهنما باشد ، آن فهرست و فایل هایش به یک روش پردازش می شوند . cp را بازگشتی بسازید . بحث کنید که آیا sv,cp باید یک نوع برنامه باشند ، در نتیجه cp-v کپی را انجام نمی دهد ، اگر فایل مقصد جدیدتر باشد .

تمرین 7-15 . برنامه random را بنویسید :

& random filename

یک خط منتخب به صورت تصادفی از فایل تولید می کند . با دادن اسامی افراد به فایل ، random می تواند در یک برنامه با عنوان scapeyoat استفاده شود ، برنامه ای که برای اختصاص به اشتباه ارزشمند می باشد :

& cat scapegoat

echo it s all random peoples foul !

&scape goat

its all kens a fault!

&

اطمینان حاصل کنید که random ، بدون توجه به توزیع طول سطرها ، درست می باشد .

تمرین 7-16 . اطلاعات دیگری در inode نیز وجود دارد ، بویژه دیسک به جای نشان می دهد که بلوکهای فایل قرار دارند . فایل > sys / ino.b را بررسی کنید ، سپس برنامه icat که فایلهای مشخص شده توسط عدد inode و طرح دیسک را می خواند ، را بنویسید . (این برنامه ، فقط زمانی کار می کند که دیسک قابل خواندن باشد) . تحت چه شرایطی ، icat مفید است ؟

4-7 فرآیندها

این بخش ، توصیف می کند که چگونه یک برنامه را از داخل برنامه ای دیگر ، اجرا کنیم . آسانترین راه برای انجام آن ، با زیر برنامه کتابخانه استاندارد . system ، می باشد . که در فصل 6 ذکر شد اما سانسور شد . system یک آرگومان را می گیرد ، یک سطر فرمان دقیقاً به همان صورتی که در پایانه تایپ می شود (به جز برای سطر جدید در انتها) و آن را در یک زیر شل ، اجرا می کند . اگر سطر فرمان باید از قطعات ساخته شود ، توانایی های فرمت درون حافظه sprintf می تواند مفید باشد . در پایان این بخش ، ما یک نسخه ایمن تر از system را برای استفاده توسط برنامه های برهم کنش نشان می دهیم ، اما در ابتدا ما باید قطعات را از چیزی که ساخته می شود بررسی کنیم .

ایجاد فرآیند دارای سطح پائین – execup , execlp

مهمترین عملکرد ، اجرای برنامه ای دیگر بدون بازگشت و با استفاده از فراخوانی سیستم execlp می باشد . برای مثال . برای پرینت تاریخ به عنوان آخرین اقدام یک برنامه اجرا از برنامه زیر استفاده کنید :

execlp (date , date , (char *)0) ;

اولین آرگومان برای execlp اسم فایل فرمان می باشد ؛ execlp ، مسیر جستجو را از محیط شما (یعنی PATH و ..) استخراج می کند و جستجو را همانند شل انجام می دهد . آرگومانهای دوم و بعدی اسم فرمان و آرگومانهایی برای فرمان می باشند و این آرگومانها ، آرابه argv برای برنامه جدید هستند . انتهای فهرست توسط یک آرگومان صفر علامت گذاری می شود . (برای آگاهی در خصوص طرح execlp , exec(2) را بخوانید) .

فراخوانی execlp به برنامه موجود را با برنامه جدید می پوشاند ، آن را اجرا می کند و سپس خارج می شود . برنامه اصلی کنترل را فقط زمانی بر می گرداند که یک خطا وجود داشته باشد . برای مثال ، اگر فایل نتواند یافت شود و یا قابل اجرا نباشد .

execlp (date, date , (char*)0)

fprintf (stderr , couldn't execute date \n;

exit (1);

یک گونه از execlp با عنوان execup زمانی مفید است که شما به طور پیشرفته نمی دانید که چند آرگومان ، باید وجود داشته باشند . فراخوان عبارت از :

Execvp (filename , argv);

در اینجا ، argp ، یک آرایه از اشاره گرها برای آرگومانها می باشد (مانند argv) ؛ آخرین اشاره گر در آرایه ، باید Null باشد ، بنابراین execlp می تواند به ما بگوید که فهرست در کجا به پایان می رسد . همانند execlp اسم فایل ، فایلی است که در آن برنامه یافت می شود و argp ، آرایه argv برای برنامه جدید می باشد ؛ [0] argp ، اسم برنامه است .

هیچ کدام از این زیربرنامه ها ، منجر به گسترش فراکاراکترهایی مانند < , > * و نقل قولها و غیره در فهرست آرگومان نمی شوند . اگر شما چنین چیزهایی را می خواهید ، برای راه اندازی برنامه bin/sh/ در شل ، از execlp استفاده کنید ، که همه کار را انجام می دهد . یک سطر فرمان رشته ای بسازید که شامل فرمان کامل باشد ، همچنانکه در پایانه تایپ شده است ، سپس بگوئید :

(char*) 0 , commadline , -c , sh , /bin/sh) Execlp ;

آرگومان -c به عنوان سطر فرمان کامل با آرگومان بعدی رفتار می کند ، نه به عنوان یک آرگومان تنها .

به عنوان یک شرح از exec و برنامه waitfile را در نظر بگیرید . فرمان & wait file filename [command

به طور متناوب ، فایل نامگذاری شده را کنترل می کند . اگر این فایل تا آخرین زمان بدون تغییر باقی بماند ، فرمان اجرا می شود . اگر هیچ فرمانی مشخص نشود ، فایل برای خروجی استاندارد کپی می شود . ما از waitfile برای کنترل پیشرفت troff استفاده می کنیم ، مانند

& wait file troff . out echo troff done &

اجرای wait file ، از fstat برای استخراج زمانی که فایل برای آخرین بار تغییر کرده است ، استفاده می کند .

```
/* wait file : wait until file stops changing*/
```

```
# include <stdio.h>
# include <sys/types.h>
# include <sys/stat.h>
char * progname ;
```

```
main(argc,argv)
int argv ;
char *argv [ ] ;
{
int fd ;
struct stat stbuf ;
time-t old - time = 0;
```

```
progname = argv [ 0 ] ;
if (argc<2)
error (dsage : % s file name [(md] ,progname) ;
if (( fd=open (argv [1] , 0)) == -1)
error (cant open % s , argv [1]);
fstat (fd , &stbuf ) ;
while (stbuf . st -mtime != old - time){
old - time = stbuf.st - mtime ;
sleep (6- ) ;
```

```
fstat (fd , & stbuf ) ;
}
if (argc == 2) { /*copy file */
execvp (cat , cat , argv [1] , (char *)0) ;
error (can't execute cat % s , argv [1 ] );
} else { /*run process */
cexecvp cargv [2] , & argv [2] ;
error (can't execute % s , argv [2] );
}
exit (0) ;
}
```

چنین برنامه ای هم `execvp` و هم `execclp` را شرح می دهد .

ما این طرح را انتخاب کردیم ، چون مفید است ، اما سایر گونه ها قابل قبول هستند . برای مثال ، `wail file` می تواند ، بازگشت را به سهولت امکان پذیر کند ، پس از اینکه فایل ، تغییر را متوقف کرده است .
تمرین 7-17 . `watek file` رابه گونه ای تغییر دهید (تمرین 7-12) که دارای همان ویژگی `wait file` باشد : اگر فرمانی وجود ندارد ، `watch file` ، فایل را کپی می کند و در غیر این صورت فرمان را اجرا می کند . آیا `wait file` ، `watch file` می توانند دارای یک رمز مبدا باشند؟ توجه : `argv[0]`

کنترل فرآیندها – `wait , fork`

مرحله بعدی بدست آوردن مجدد کنترل پس از اجرای یک برنامه با `execclp` یا `execbp` می باشد . چون این زیر برنامه ها ، به سهولت ، برنامه جدید را بر روی قبلی می پوشانند ، برای ذخیره برنامه قبلی لازم است که این برنامه در ابتدا به دو کپی تقسیم شود یکی از آنها می تواند پوشانده شود ، در حالیکه کپی دیگر ، منتظر برنامه جدید باقی می ماند و برنامه را تا پایان می پوشاند . تقسیم توسط یک سیستم فراخوان با عنوان `fork` انجام می شود :

```
Proc-id=fork( ) ;
```

برنامه را به دو کپی تقسیم می کند و هر دوی آنها اجرا می شوند . تنها تفاوت بین این کپی ، مقدار بازگردانده شده توسط `fork` می باشد ، `process-id` . در یکی از این فرآیندها ، `proc - id (child)` صفر می باشد . در فرآیند دیگر `(parent)` ، `proc - id` مقداری غیر از صفر می باشد این فرآیند `process - id` بچه می باشد . بنابراین روش اصلی برای فراخوانی هر بازگشت از برنامه ای دیگر عبارت است از :

```
If (fork ( ) == 0 )
```

```
Execclp (/bin /sh, sh , c , command line , (char *)o);
```

درحقیقت به جز برای کارکردن با خطاها ، این روش ، مناسب می باشد . `fork` روکپی از برنامه می سازد . در بچه ، مقدار بازگردانده شده توسط `fork` صفر است بنابراین `execclp` را فرا می خواند که سطر فرمان را انجام می دهد و سپس حذف می شود .
در والد ، `fork` مقدار غیر از صفر را باز می گرداند ، در نتیجه از `execclp` پرش می کند (اگر خطایی وجود نداشته باشد ، -1 ، `fork` را بر می گرداند) .

اغلب ، والد ، منتظر می ماند تا بچه به پایان برسد ، قبل از اینکه خودش ادامه دهد چنین چیزی با فراخوانی سیستم `wait` انجام می شود

:

```

Int status ;
If (fork ( ) == 0)
Execlp (000);      /*child */
Wait (&status );  /*purent*/

```

چنین چیزی، هیچ کدام از موقعیتهای غیر عادی مانند خرابی `execlp` یا `fork` را بکار نمی برد و احتمالا ممکن است بیش از یک بچه به طور همزمان اجرا شود. (`wait`, `process-id`) از بچه پایان یافته را بر می گرداند. اگر شما بخواهید، آن را توسط `fork` در مقابل مقدار بازگردانده شده، کنترل کنید. در آخر، این بخش به هیچ کدام از رفتارهای عجیب از طرف بچه نمی پردازد. هنوز این سه سطر، قلب تابع استاندارد `system` می باشند.

`Stats` بازگردانده شده توسط `wait` تصور سیستم را در خصوص وضعیت خروجی بچه به 8 بیت پائین مرتبه آن، رمز گذاری می کند و این میزان برای اتمام عادی صفر و برای نشان دادن انواع متعدد از مشکلات صفر می باشد. 8 بیت بزرگتر بعدی، از آرگومان فراخوان برای `exit` گرفته می شوند و یا از `main` که منجر به اتمام فرآیند بچه می شود، باز می گردند.

زمانی که یک برنامه توسط شل فراخوانده می شود، سه توصیف گر فایل صفر و یک و دو، با اشاره به فایل های صحیح تنظیم می شوند و سایر توصیف گران فایل در دسترس برای استفاده هستند زمانی که این برنامه برنامه ای دیگر را فرامی خواند، تشریفات صحیح، اطمینان می دهد که همان شرایط حفظ می شوند. نه `fork` و نه `exec`، به هیچ وجه بر فایل های باز تاثیر نمی گذارند و هم والد و هم بچه دارای فایل های باز یکسان می باشند. اگر والد، خروجی ایی را میانگیر کند، که باید قبل از خروجی بچه، خارج شود، والد باید میانگیرهای خود را قبل از `execlp`، خارج کند به طور معکوس، اگر واد، یک جریان ورودی را میانگیر کند، بچه، هر گونه اطلاعاتی را که توسط والد خوانده شده است، رها می کند. خروجی می تواند خارج شود، اما ورودی نمی تواند به تعویق بیفتد. هر دوی این بررسی ها مطرح می شوند، اگر ورودی یا خروجی با کتابخانه استاندارد I/O که در فصل 6 توصیف شد، انجام شود، چون کتابخانه استاندارد I/O به طور عادی هم خروجی و هم ورودی را میانگیر می کند.

این خصوصیت توصیف گران فایل در میان یکد `execip` می باشد که `system` را بشکنند: اگر برنامه فراخوان دارای ورودی یا خروجی استاندارد متصل به پایانه نباشد، هیچ کدام فرمان را با عنوان `system` نمی خواهند. این ممکن است چیزی باشد که خواسته شود و در یک متن `ed` برای مثال از متن بیاید. حتی `ed` باید ورودی خود را به صورت یک کاراکتر در یک زمان بخواند، برای اینکه از مشکلات میانگیر ساز ورودی جلوگیری کند.

اما برای برنامه های برهم کنشی مانند `system.p` باید مجددا ورودی و خروجی استاندارد را به پایانه متصل کند. یک راه برای این کار، متصل کردن آنها به `dev/tty` می باشد.

فراخوان سیستم (`dup(f)`) و توصیف گر فایل `fd` را بر روی توصیف گر فایل دارای پائین ترین شماره و تخصیص نیافته، کپی می گیرد یک توصیف گر جدید را که به همان فایل باز استناد می شود، باز می گرداند. این رمز، ورودی استاندارد یک برنامه را به فایل متصل می کند:

```

In fd;
Fd = open (file , 0) ;
Close (0) ;
Clup (fd) ;
Close (fd) ;

```

(`Close (0)`)، توصیف گر فایل صفر و ورودی استاندارد را آزاد می سازد، اما در حالت معمول، بر والد تاثیر نمی گذارد.

در این جا نسخه ما از سیستم (system) برای برنامه های برهم کنشی وجود دارد ؛ این نسخه از progname برای پیامدهای خطا استفاده می کند . شما باید بخشهایی از تابع را که به علامت ها می پردازند . نادیده بگیرند ما در بخش بعد به آنها می پردازیم .

```

/*
*safer vesion of system for interactive programs
*/
# include <sigul.h>
# include <stdio.h>
system (s) /*run command lines */
char*s ;
{
int status >,pid ,w ,tty ;
int (*istat)( ) , (*qstat )( ) ;
extern char * progname ;
fflush (stdout);
tty= open (/dev/tty , 2);
if (tty== -1){
fprintf (stderr,%s : can't open /dev/tty\n , progname);
return-1;
}
if ((pid = fork (1) == 0){
close (0) ; dup (tty) ;
close (1); dup (tty) ;
close (2) ; dup (tty) ;
close (tty) ;
execlp (sh , sh ,-c , s , (char *)
exit (127) ;
}
close (tty) ;
istat = signal (SIGINT , SIG -1 GN) ;
qstat = signal(SIGQUIT , SIG- IGN);
while ((w=wait (&status))! = pid &&w! = -1)
}
close (tty) ;
istat = signal (SIGINT , SIG-!GN);
q stat = signal (SIGQUIT , SIG - ICN);
while(( w= wait (cstatus)) ! = pid&& w! = -1)
;
if (w == -1)
status = -1 ;
signal (SIGINT , istat);
signal (SIGQUIT ,gstat);
return stats ;
}

```

توجه داشته باشید که `dev / tty/` با حالت 2 باز می شود . خواندن و نوشتن - و سپس با `dup ed` برای تشکیل خروجی و ورودی استاندارد باز می شود . این دقیقا روشی است که سیستم خروجی ، ورودی و خطای استاندارد را مونتاژ می کند ، زمانی که شما دارد آن

می شود . بنابراین ، خروجی استاندارد شما ، قابل نوشتن می باشد :

```
& echohello 1>&0
hello
&
```

مفهوم آن این است که ما می توانیم توصیف گر 2 از فایل dup'ed را برای اتصال مجدد ورودی و خروجی استاندارد داشته باشیم ، اما بازکردن dev/tty/ تمیز تر و ایمن تر است . حتی این system دارای مشکلات بالقوه می باشد: فایل های باز در شماره گیرنده مانند tty در زیر برنامه ttyin در p از فرآیند بچه عبور خواهند کرد .

درس در اینجا این نیست که شما باید از نسخه system ما برای همه برنامه های خود استفاده کنید- برای مثال به این نسخه ed غیر برهم کنش را می شکنند — اما درس این است که شما درک کنید چگونه فرآیندها کنترل می شوند و از موارد اولیه به درستی استفاده می کنند و معنی «به درستی» با کاربرد فرق می کند و ممکن است موافق با اجرای استاندارد system نباشد .

5-7 . علائم و وقفه ها

این بخش در رابطه با چگونگی پرداختن به علائم (مانند وقفه ها) به طور دقیق ، از دنیای خارج و با اشکالات برنامه می باشد . اشکالات برنامه اساسا از مراجع غیر مجاز حافظه ، اجرای ساختارهای ویژه یا خطاهای ممیز شناور بوجود می آیند . عمومی ترین علائم دنیای خارج ، وقفه می باشد که زمان فرستاده می شود که کاراکتر DEL تایپ می باشد و خارج شدن از برنامه که توسط کاراکتر (FS) -1 ctl بوجود می آید ؛ گیرماندگی که با گیرماندن تلفن بوجود می آید و پایان دادن که با فرمان kill بوجود می آید . زمانی که یکی از این وقایع رخ می دهد ، علامت به همه فرآیندهایی فرستاده می شود که از همان پایانه آغاز شدند و مگر اینکه سایر آرایش ها ، ساخته شده باشند ، علامت به فرآیند خاتمه می دهد . برای اکثر علائم ، یک فایل تصویر حافظه اصلی ، برای خطا زدایی بالقوه نوشته می شود (adb(1) sdb(1) را مشاهده کنید) .

علامت فراخوانی سیستم ، عملکرد پیش فرض را تغییر می دهد . این علامت دارای دو آرگومان می باشد . اولین آرگومان ، عددی است که علامت را مشخص می کند . دومین آرگومان ، آدرس یک تابع و یا رمزی می باشد که درخواست می کند ، علامت نادیده گرفته شود یا به عملکرد پیش فرض ارائه شود . فایل <signal.h> شامل تعاریفی برای آرگومانهای متعدد می باشد . بنابراین :

```
# include <signal.h>
```

```
.....
Signal (SIGINT , SIG-IGN) ;
```

منجر به نادیده گرفته شدن وقفه ها می شود در حالیکه

```
Signal (SIGINT , SIG - DEL) ;
```

عملکرد پیش فرض خاتمه فرآیند را مجددا ذخیره می کند . در همه موارد ، signal ، ارزش قبلی علامت را باز می گرداند . اگر دومین آرگومان برای signal ، نام یک تابع باشد ، (که باید دقیقا در همان فایل مبدا اعلان شده باشد) . تابع فراخوان می شود ، زمانی که signal رخ می دهد . عموما این روش به این منظور استفاده می شود که برنامه کار ناتمام را قبل از اتمام پاک کند ، برای مثال یک فایل موقت را حذف کند .

```
# include < signal-h>
```

```
char * tempfile = temp .xxxxxy;
main(1
```

```

{
extern onintr ( ) ;
if (signal (SIGINT , SIG-IGN) != SIG-IGN)
signal (SIGINT , onitr) ;
mktemp(tempfile);
/*process...*/
exit (0) ;
}
onitr ( ) /*clean up if interrupted */
{
unlike (tempfile) ;
exit (1) ;
}

```

چرا فراخوانی آزمون `double test` برای `signal` در `pmain` به خاطر بیاورید که علائم ، به همه فرآیندهای آغاز شده از یک برنامه ، به صورت غیر برهم کنشی (با & آغاز می شود) اجرا می شود ، شل به گونه ای مرتب می شود که برنامه ، وقفه ها را نادیده بگیرد ، بنابراین شل توسط وقفه های مورد نظر برای فرآیندهای پیش زمینه متوقف نمی شود . اگر این برنامه با بیان این موضوع آغاز شود که همه وقفه ها بدون توجه به اینکه تلاش شل را برای حمایت از آن ، زمانی که در زمینه اجرا می شود بی اثر می سازند ، به زیر برنامه `onitr` فرستاده خواهد شد .

راه حل نشان داده شده در بالا ، آزمایش الت استفاده از وقفه و تداوم آن برای نادیده گرفتن وقفه ها می باشد ، اگر آنها نادیده گرفته می شوند . رمز همانگونه که نوشته می شود ، به این واقعیت بستگی دارد که `signal` حالت قبلی یک علامت خاص را باز می گرداند . اگر علامتها ، نادیده گرفته شوند ، فرآیند باید برای نادیده گرفتن آنها ادامه یابد در غیر این صورت ، آیا باید متوقف شوند . یک برنامه پیشرفته تر ممکن است بخواهد یک وقفه را نگاه دارد و آن را به عنوان یک درخواست برای متوقف کردن چیزی بکار برد که انجام می شود و آن را به حلقه پردازش فرمان خود بازگرداند . به یک ویراستار متن فکر کنید : متوقف کردن یک خروجی چاپی بلند نباید منجر به خروج آن شود و کاری را که قبلا انجام شده است ، از راست بدهد . رمز برای این مورد ، می تواند به این صورت نوشته شود :

```

#include <signal.h>
#include <sp-timp.h>
jmp -buf sjbuf ;
main()
{
int onitr ( ) ;
if (signal (SIGINT , SIG-IGN)!=SIG-IGN)
signal /(SIGINT , onitr);
setjmp (sjbuf); /* save cvrrent stack position */
for (;) {
/* main processing loop */
}
.....
}
onitr ( ) /*reset ifinterrupted */
{

```

```

signal /(SIGINT , onitr ); /*reset for next interrupt */
printf (\nInterrupt\n);
long jmp (sjbuf , 0) ; /*return to saved state */
}

```

فایل <set jmp.h> تایپ jmp-buf را به عنوان یک هدف بیان می کند که در آن موقعیت stack می تواند ذخیره شود و sjbuf ، به عنوان یک چنین هدفی بیان می شود . تابع (set jmp 3) یک رکورد از جایی را ذخیره می کند که برنامه در آن اجرا می شود . ارزشهای متغیرها ، ذخیره نمی شوند . زمانی که یک وقفه رخ می دهد ، یک فراخوان به سمت زیر برنامه onitr رانده می شود ، که می تواند یک پیام را پرینت کند ، پرچم ها را تنظیم کند و یا هر چیز دیگری انجام دهد . longjmp به عنوان یک آرگومان یک هدف ذخیره شده توسط setjmp را می گیرد و کنترل را برای موقعیت پس از فراخوانی setjmp مجدداً ذخیره می کند . بنابراین کنترل (و سطح stack) ، به جایی در زیر برنامه اصلی ، بر می گردند ، جایی که حلقه اصلی وارد می شود .

توجه داشته باشید که signal دوباره در onitr تنظیم می شود ، پس از اینکه یک وقفه رخ می دهد چنین چیزی لازم است : علائم به طور خودکار برای عملکرد پیش فرض خود ریست می شوند زمانی که رخ می دهند .

برخی از برنامه هایی که می خواهند ، علائم را به سهولت آشکار سازند ، نمی توانند در یک نقطه قراردادی ، برای مثال در وسط روز آمدسازی یک ساختار پیچیده از داده ها متوقف شوند . راه حل ، داشتن یک زیربرنامه وقفه برای تنظیم یک پرچم و بازگشت به جای فراخوانی exit یا longjmp می باشد . اجرا ، در نقطه ای ادامه می یابد که دقیقاً متوقف شده است و پرچم وقفه می تواند بعداً آزمایش شود .

یک پیچیدگی همراه با این روش وجود دارد . فرض کنید برنامه پایانه را می خواند ، زمانی که وقفه فرستاده می شود . زیر برنامه مشخص شده ، به موقع فراخوان می شوند این زیربرنامه ، پرچم خود را تنظیم می کند و باز می گردد . اگر حقیقتاً درست باشد ، همان گونه که ما در بالا عنوان کردیم که این اجرا ، در نقطه ای که دقیقاً قطع شده دوباره از سرگرفته شود برنامه خواندن پایانه را تا جایی ادامه می دهد که کاربر سطر دیگری راتایپ کند . این رفتار گیج کننده است ، چون کاربر نمی داند که می خواند و ظاهراً ترجیح می دهد که علامتی داشته باشد که به طور مداوم موثر باشد . برای حل این مشکل ، سیستم read را پایان می دهد ، اما با یک حالت خطا ، که نشان می دهد چه اتفاقی افتاده است : errno ، برای EINTR تنظیم می شود و در <errno.h> برای نشان دادن یک فراخوانی سیستم متوقف شده ، تعریف می شود .

بنابراین ، برنامه هایی که متوقف می شوند ، دوباره اجرا را از سر می گیرند ، پس از اینکه علائم باید برای خطاهای ایجاد شده توسط فراخوانیهای سیستم متوقف شده آماده شوند . (سیستم برای تماشای read ها از یک پایانه ، wait و pause ، فراخوان می شود) . چنین برنامه ای می تواند از رمزی مانند رمز زیر استفاده کند ، زمانی که ورودی استاندارد را می خواند :

```

#include <errno .h>
extern int errno ;
.....
if (read (0,& c,1)<=0) /*EOF or interrupted */
if (errno == EINTR ) { /*EoF caused by interrupt */
errno = 0 ; /*reset for next time */
.....
}else { /*true end of file */
.....

```

یک ظرافت نهایی برای به خاطر سپردن وجود دارد ، زمانی که گیرنده علامت با اجرای سایر برنامه ها ، ترکیب می شود . برنامه ای را

در نظر بگیرید که وقفه ها را می گیرد و نیز شامل یک شیوه (مانند «!» در ed) می باشد و به موجب آن سایر برنامه ها می توانند اجرا شوند. سپس رمز، می تواند چیزی شبیه به رمز زیر به نظر برسد.

```
If (fork ( ) == 0)
Execlp (...);
Signal (SIGINT , SIG-IGN); /*parent ignores interrupts */
Wait (& status ); /*until child is doen */
Signal (SIGINT , onitr); /* restore interrupts */
```

چرا اینگونه است؟ علائم به همه فرآیندهای شما فرستاده می شود. فرض کنید برنامه ای که شما فرامی خوانید، وقفه های خود را می گیرد، همانند کاری که ویراستار انجام می دهد. اگر شما برنامه فرعی را متوقف کنید، علامت را می گیرد و به حلقه اصلی خود باز می گرداند و احتمالاً پایانه شما را می خواند. اما برنامه فراخوان نیز برای برنامه فرعی و خواندن پایانه شما از wait خود خارج می شود. داشتن دو فرآیندی که پایانه شما را می خوانند، بسیار گیج کننده است، چون در واقع، سیستم یک سکه را پرتاب می کند برای اینکه تصمیم بگیرد چه کسی باید وارد هر سطر از ورودی شود. راه حل این است که برنامه والد، وقفه را نادیده بگیرد تا جایی که بچه انجام می شود. این استدلال در بکارگیری علامت در system، منعکس می شود:

```
# include <signal -h>
sgstem (s) /*run command line */
char * s ;
{
int status , pid .w, tty ;
int (*istat) ( ), (*qstat) ( ) ;
.....
If ((pid = fork ( )) == 0 ) {
....
Execlp (sh , sh , -c , s (char *)0) ;
Exit (127) ;
}
.....
istat = signal (SIGINT , SIG-IGN);
qstat = signal (SIGQBIT , SIGIGN);
while ((w = wait (&status )) != pid&& w != -1
;
if (w == -1)
status = -1 ;
signal (SIGINT , istat);
signal (SIGQVIT , qstat);
return status ;
}
```

جدای از این اظهارات، تابع signal به طور بدیهی دارای یک آرگومان ثانویه عجیب می باشد. این آرگومان در حقیقت یک اشاره گر به تابعی است که یک عدد صحیح را دریافت می کند و همچنین تایپ خود زیر برنامه signal می باشد. دو ارزش SIG-DFL، SIG-LGN، دارای تایپ صحیح می باشند اما انتخاب می شوند و در نتیجه با هیچ کدام یک از تابعهای واقعی ممکن تلاق ندارند. برای افراد شایق در اینجا چگونگی تعریف آنها برای PDP-11، VAX وجود دارد؛ تعاریف باید تا جایی نگران کننده باشند که ما را تشویق به استفاده از <signal.h> کنند.

```
#define SIG-DEL (int(*) ( 1)0
# define SIG -IGN (int (*) (1)1
```

هشدارها

فراخوانی سیستم (alarm(n باعث می شود که یک علامت SIGALRM به فرآیند شما پس از چند ثانیه فرستاده می شود . علامت alarm (هشدار) می تواند برای اطمینان یافتن از اینکه چیزی درمیزان زمان رخ می دهد ، استفاده شود ؛ اگر چیزی اتفاق بیفتد ، علامت alarm می تواند خاموش شود ، اما اگر خاموش نشود ، فرآیند می تواند کنترل را مجدداً با گرفتن علامت alarm بدست آورد . برای شرح ، در این جا یک برنامه با عنوان time out وجود دارد که فرمان دیگر را اجرا می کند ؛ اگر آن فرمان ، توسط زمان مشخص شده ، پایان نپذیرد ، لغو می شود . زمانی که alarm قطع می شود . برای مثال – فرمان watch for از فصل 1 را به خاطر آورید . به جای اینکه این فرمان به طور نامحدود اجرا شود، شما باید یک محدودیت زمانی را تنظیم کنید :

```
& time out - 3600 wathcfor dmg &
```

رمز در time out تقریباً هر چیزی را که ما در خصوص آن در دو بخش گذشته صحبت کردیم شرح می دهد . بچه ایجاد می شود والد یک alarm را تنظیم می کند و سپس منتظر به پایان رسیدن بچه باقی می ماند . اگر alarm در ابتدا ظاهر شود ، بچه حذف می شود . یک تلاش برای بازگرداندن وضعیت خروجی بچه انجام می شود .

```
/* time out : set time limit on a process */
# include <stdio .h>
# include <signal .h>
intpid ; /* child process id */
char * progname;
main (argc , argv)
intargc ;
char* argv [ ] ;
{
int sec = 10 , status , on alarm ( ) ;
progname = argv [0] ;
if (argc>1&& argv[ 1] [0]= = -){
sec = atoi (&&argv [1] [1] ;
argc -- ;
argv ++;
}
if (argc <2)
error (usage . % s [ 10]command , progname ) ;
if ((pid = fork (1) = = 0 ) {
execvp cargv [1] ,& argv [1] ;
error (couldn't start % s ,argv [1] ;
}
signal (SIGALRM , onalarm);
alarm (sec);
if (wait(status) = = -1 : : (statis & 0177 )!= 0)
error (% s killed , argv[ 1] ;
exit ((status ))8) & 0377);
```

```

}
onalarm () /* kill child when alarm arrives */
{
kill cpid , SIC KILL );
}

```

تمرین 7-18 . آیا شما می توانید استنباط کنید که چگونه SLEEP اجرا می شوید ؟ توجه 2 (PAUSE) تحت چه شرایطی اگر شرایطی وجود ندارد آیا sleep , alarm می توانند با یکدیگر تداخل شوند ؟

تاریخچه و نکات کتاب شناسی

توصیف کاملی در خصوص اجرای سیستم یونیکس وجود ندارد ، تا حدودی به خاطر اینکه ، رمز ، باز می باشد. مقاله اجرای یونیکس از کن تامپسون ، (1978 ، جولای BSTJ) موضوعات مهم را شرح می دهد . سایر مقالاتی که موضوعات مربوط را شرح می دهند ، « مقاله سیستم یونیکس - یک مقاله بازنگرانه در همان موضوع [BST] و ارزیابی سیستم اشتراک زمانی یونیکس (سمپوزیوم در خصوص متولوژی برنامه نویسی و طرح زبان ، نکات مربوط به سخنرانی اسپرینگر - وولاگ در خصوص علم کامپیوتر 79 ، 1979) می باشند که هر دو نوشته دنیس ریتچای هستند .

برنامه readslow ، توسط پیتر مرینبرگ ، به عنوان یک روش اضافی پائین برای ماشاگران به منظور تماشای پیشرفت ماشین شطرنج بل ، کن تامپسون و جوکاندون در طول مسابقات شطرنج ، اختراع شد .

بل ، وضعیت بازی خود را در یک فایل ثبت کرد و تماشاگران فایل را با readslow بدست آوردند ، به گونه ای که بسیاری از چرخه های قبلی را از طرح بل نگیرد . (جدیدترین نسخه سخت اقرار بل ، محاسبه اندکی را بر روی ماشین میزبان خود انجام می دهد ، بنابراین مشکل بطرف شده است) .

فکر بکر b برای spname از تام داف می آید . یک مقاله توسط ایوردارهام ، دیوید لامب و جیمز ساکس که تصحیح تلفظ را در واسطه های کاربر مجاز کرد ، CACM ، اکتبر 1983 ، تا حدودی طرح متفاوتی را برای تصحیح تلفظ ، در متن یک برنامه پستی ارائه می دهد .

فصل 9- توسعه برنامه

سیستم UNIX در واقع محیطی جهت طراحی و توسعه برنامه است. در این فصل در مورد ابزارهایی که بالاخص برای طراحی و توسعه برنامه مناسبند صحبت می کنیم. ابزار ما یک برنامه باارزش و **متسری** برای زبان برنامه نویسی در حد توان Basic می باشد. از آنجا که یک زبان نماینده ای از مشکلاتی است که در برنامه های بزرگ پیش می آید، قصد داریم مراحل توسعه یک زبان را مطرح کنیم. علاوه براین می توان به بسیاری از برنامه ها به عنوان زبانهای نگاه کرد که یک ورودی سیستماتیک داخلی را به یکسری عملیات و خروجیهای پشت سرهم تبدیل می کند، بنابراین قصد ابزارهای توسعه زبان را بیان کنیم.

در این فصل دروس خاصی راجع به مطالب زیر را مطرح خواهیم کرد:

-Yacc: یک مولد تجزیه گر (parser)، برنامه ای که با یک بیان گرامری زبان جداکننده تولید می کند.

-Make: برنامه ای برای تعیین و کنترل فرآیندهایست که یک برنامه پیچیده با آنها کامپایل می شود.

-Lex: برنامه ای شبیه yacc، برای ساخت تحلیلگرهای واژه ای.

در ضمن مواردی نظیر چگونگی مواجهه با یک پروژه، اهمیت شروع یک برنامه، توسعه تدریجی زبان و استفاده از ابزارهای مختلف را مطرح می کنیم.

زبان را در شش مرحله توسعه خواهیم داد. حتی اگر تا انتهای شش مرحله نیز پیش نروید، هر یک از مراحل به تنهایی آموزنده و مفید خواهد بود. توسعه یک برنامه دقیقاً به **ترتیب** این شش مرحله خواهد بود. این مراحل عبارتند از:

• ماشین حساب چهار عمل اصلی، شامل +، -، ×، / و پرانتز، که بر روی اعداد اعشاری عمل می کند. در هر خط یک عبارت تایپ می شود و ارزش آن فوراً چاپ می شو.

• متغیرها با اسامی a تا z، این مرحله علامت منفی از عبارت و حساسیت به خطاها را نیز در برمی گیرد.

• نامهای متغیرها با طول دلخواه، توابع داخلی exp, sin و ...، ثوابت ثابتی نظیر $1/T$ (عدد n به دلیل محدودیتهای تایپی به صورت $1T$ نمایش داده شده است). و یک اپراتور نمایی.

• تغییر در توابع داخلی. برای هر دستور (statement) کدی تولید می شود و سپس به جای برآورد سریع تفسیر می شود. هیچ ویژگی feature جدیدی اضافه نمی شود اما نهایتاً به مرحله (5) می انجامد.

• جریان کنترل: uelse و while، عبارات هم گروه با {and} و اپراتورهای رابطه ای نظیر <, >, ...

• توابع و عملیات برگشتی به همراه **آرلگانهایشان**. در این مرحله دستوری نیز برای ورودی و خروجی ---- و اعداد اضافه کرده ایم.

زبان حاصل از این 6 مرحله در دو فصل 9 توضیح داده شده است. در این فصل از این زبان به عنوان مثال اصلی در ارائه نرم افزار تهیه

راهنمای UNIX استفاده شده است . پیوست (2) راهنمای مرجع است .

از آنجایی که در نوشتن صحیح يك برنامه مفادین جزئیات زیادی باید مدنظر قرار گیرد ، این فصل بسیار طولانی است . فرض ما بر این است که خواننده زبان C را درک می کند و نسخه ای از جلد دوم راهنمای برنامه نویسی UNIX را در دست دارد ، چرا که در این جا مجالی برای توضیح تمامی جزئیات نیست . توجه کنید و خودتان را آماده کنید که دوباره این فصل را مطالعه کنید . تمامی کدهای مورد نیاز برای نسخه پایانی را در پیوست 3 آورده ایم ، بنابراین به راحتی مشاهده خواهید بود که اجزاء چگونه در تناسب با یکدیگر قرار گرفته اند .

زمان زیادی را صرف کردیم تا نام مناسبی برای این زبان بیابیم اما هرگز به نتیجه مطلوب دست نیافتیم . نهایتاً hoc را برگزیدیم که از high order calculator ” ” گرفته شده است . بنابراین نسخه ها hoc1, hoc2 و ... می باشند .

8-1- مرحله 1 : ماشین حساب چهار عمل اصلی :

این بخش تولید hoc1 را توضیح می دهد و این برنامه ، برنامه ای است که تواناییهای معادل يك ماشین حساب جیبی با امکانات محدود را فراهم می کند و البته به راحتی آن حمل نمی شود . این برنامه فقط چهار تابع + ، - ، × ، / را دارد (به علاوه پرانتز را هم که می تواند به دلخواه وارد عبارت شود شامل می شود) که ماشین حسابهای جیبی با قابلیت های محدود ارائه می دهند . اگر پس از يك عبارت RETURN تایپ کنید ، جواب حاصل در خط بعد چاپ خواهد شد .

```
$ hoc1
4*3*2
24
(1+2)*(3+4)
21
1/2
0.5
355/113
3.1415929
-3-4
hoc1: syntax error near line 4
$
```

It doesn't have unary minus yet

- قواعد (grammars)

از زمانیکه فرم **Backus- Naur** برای Algol به وجود آمد، زبانها با قواعد منطقی بیان شده اند. قواعد *hoc1* در نمایش اختصاری شان کوچک و ساده اند.

```
list :   expr \n list
        expr \n
expr:   NUMBER
        expr + expr
        expr - expr
        expr * expr
        expr / expr
        (expr)
```

چنانکه از مجموعه عبارات فوق برمی آید، *list* تناوبی از عباراتی است که در خطوط مجزا به دنبال هم می آیند. هر عبارت شامل یک عدد یا یک جفت عبارت که توسط یک اپراتور به هم مرتبطند و یا کی عبارت داخل پرانتز می شود.

این برنامه کامل نیست. روال اولویت را در میان برنامه های مختلف و نیز ارتباط اپراتورها را مشخص نمی کند. در ضمن هیچ معنی را به مفاهیم (*constructs*) نمی بخشد. یا اینکه *list* با استفاده از *expr* و *expr* با استفاده از *NUMBER* بیان می شود، اما خود *NUMBER* هیچ جا بیان نمی شود.

این جزئیات باید وارد شوند تا از یک طرح اولیه زبان به یک برنامه کاری برسیم.

بازنگری yacc:

yacc یک مولد جداکننده است. به این معنی که *yacc* برنامه ای برای تبدیل بیان گرامری (قاعده ای)، نظیر آنچه در برنامه بالا آمده، به یک جداکننده که عبارات داخل زبان را جدا می کند می باشد. *yacc* روشی برای ارتباط معانی با اجزاء گرامری فراهم می کند که همانطور که عملیات تجزیه رخ می دهد، معنی هم برآورد شود. مراحل استفاده از *yacc* به ترتیب زیر است:

ابتدا، قاعده ای مشابه به آنچه در بالای صفحه آمده است اما دقیق تر نوشته می شود. این قاعده ترکیب (*syntax*) زبان را مشخص می کند. *yacc* می تواند در این مرحله جهت هشدار برای خطاها و بروز شک در گرامر به کار رود.

در مرحله دوم، هر گرامر یا محصول آن می تواند با یک عمل (*action*) توسعه یابد. عمل عبارتی است که بیان می کند زمانی که یک فرم گرامری خاص در برنامه ای که در حال تجزیه است یافت شد چه عمل خاصی انجام شود. این عمل خاص به زبان *c* نوشته می شود و با تبدیلاتی گرامر را به زبان *c* ارتباط می دهد. این مرحله (*semantic*) معنای زبان را تعیین می کند.

در مرحله سوم ، يك تحليل گر واژه مورد نیاز است . این تحلیل گر باید ورود جايي را که در حال تجزیه هستند بخواند و آنها را به قطعات (**chunks**) معني داري براي تجزیه گر بشکند . يك NUMBER مثالي از يك قطعه واژه اي به طول چند کاراکتر است . اپراتورهاي تك کاراکتری نظیر + ، * نیز قطعه هستند . يك قطعه واژه اي يك نشانه (to ken) نامیده مي شود .

نهایتاً ، يك برنامه مستقل (routine) کنترل کننده جهت فراخواني تجزیه گر که توسط yacc ساخته مي شود مورد نیاز است . yacc⁴ گرامر عملیات معنایي را به يك تابع پردازنده تبدیل مي کند . این تابع yyparse نامیده مي شود و به صورت فایل c نوشته مي شود . اگر yacc هیچ خطايي پیدا نکند ، پردازنده ، تحلیل گر واژه و برنامه مستقل کنترل کننده مي توانند کامپایل و احتمالاً در ارتباط با سایر برنامه هاي مستقل c و اجرا شوند . اجرای این برنامه عبارتست از فراخواني مکرر تحت تحلیل گر واژه اي براي نشانه ها ، درك ساختار گرامري در ورودي و اجرای عملیات معنایي به موازات اینکه هر قانون گرامري درك مي شود . ورودي به تحلیل گر باید yylex نام بگیرد زیرا هر بار yyparse این تابع را فرا مي خواند ، نشانه جدیدي مي خواهد . (اسامي- که در yacc استفاده مي شود با- y شروع مي شود . به بیان دقیق تر ورودي yacc فرم زیر را مي گیرد .

```
% {
C statements like #include, declarations , etc. This section is optional.
%}
yacc declaration: lexical tokens ,grammar variables,
precedence and associativity information
%%
grammar rules and actions
%%
more C statements (optional):
main () { . . . ; yyparse () ; . . . }
yylex () { . . . }
```

این فرم با yacc اجرا می شود و نتیجه در فایلی به نام y.tab.c با چیدمان زیر نوشته می شود .

```
C statements form between %{and%}, if any
C statements from after second %% , if any:
Main() { ... ; yyparse () ; ... }
Yylex () { ... }
```

```
Yyparse () { parser ,Which calls yylex () }
```

این مورد که yacc يك فایل C به جاي فایل کامپایل شده (0) - مي سازد، طريقه معمول

⁴ . yacc از **yet another comiler** ----- گرفته شده است . این مورد ، توصیفی توسط نویسنده اش Steve Johnson ، بر روی تعدادی از برنامه هایی که در زمان تولید yacc وجود داشت (حدود 1972) می باشد .

برخورد یونیکس است. این شیوه که در آن تولید شده قابل حمل و انتقال به سایر فرآیندهاست منعطف ترین شیوه ممکن است. خود yacc ابزاری قدرتمند است. لذا هر چند ممکن است آموختن yacc تلاش زیادی را بطلبد اما نتیجه مثبت این تلاش به دفعات دیده می شود. تجزیه گرهایی که با yacc تولید می شوند، کوچک، اکاراً و صحیح هستند. (هرچند صحت عملیات معنایی برعهده خود شماست.) با بسیاری از مشکلات واضح تجزیه اتوماتیک برخورد می شود. برنامه های شناخت زبان براحتی ساخته می شوند و (مهمتر اینکه) به موازات پیشرفت بیان زبان به کرات قابل اصلاحند.

برنامه مرحله 1)

کد مرجع برای hoc1 شامل گرامری به همراه عملیات، یک برنامه مستقل واژه ای و یک main می باشد که همگی در فایل hoc.y قرار دارند. (اسامی - فایل های yacc به y ختم می شوند. اما این نحوه نامگذاری مرسوم، برخلاف cc و c با خود yacc **تحصیل** نمی شود.) بخش گرامری، نیمه اول hoc.y است:

```
$ cat hoc.y
%{
#define YYSTYPE double %}ft data type of yacc stack *1
%token NUMBER
%left
%left '+' '-' '*' left associative, same precedence *1 1* left
%%
'I' assoc., higher precedence *1
list:
    1* nothing ..I
    list '\n'
    list expr '\n' { printf("\t%.8g\n", $2); }

expr:
    NUMBER { $$ ::=$1; }
    expr '+' expr { $$ ::=$1 $2 $3; }
    expr '-' expr { $$ ::=$1 - $3; }
    expr 'ft' expr { $$ ::=$1 .. $3; }
    expr 'I' expr { $$ ::=$1 I $3; }
    '(' expr ')' { $$::=$2; }

%%
1* end of grammar ftl
```

اطلاعات جدید زیادی در این خطوط وارد شده است. قصد نداریم که هر جزئیات این بخش را توضیح دهیم و همچنین قصد نداریم نحوه عمل تجزیه گر را بیان کنیم. برای کسب اطلاعاتی در این زمینه می توانید به راهنمای yacc مراجعه کنید.

قوانین یک درمیان با I جدا می شوند. هر قانون گرامری می تواند عمل مربوط به خود را داشته باشد. این قانون زمانی اجرا می شود که نمونه ای از آن در ورودی شناسایی شود. یک عمل، مجموعه ای از عبارات c است که در براکت قرار گرفته اند. {and}. با یک عمل، S_n (مانند S_1 ، S_2 ، ...) دال بر مقداری است که به وسیله n امین جزء

برگردانده می شود و $\$ \$$ مقداری است که به عنوان مقدار کل قانون برگردانده می شود . بنابراین به عنوان مثال ، در قانون

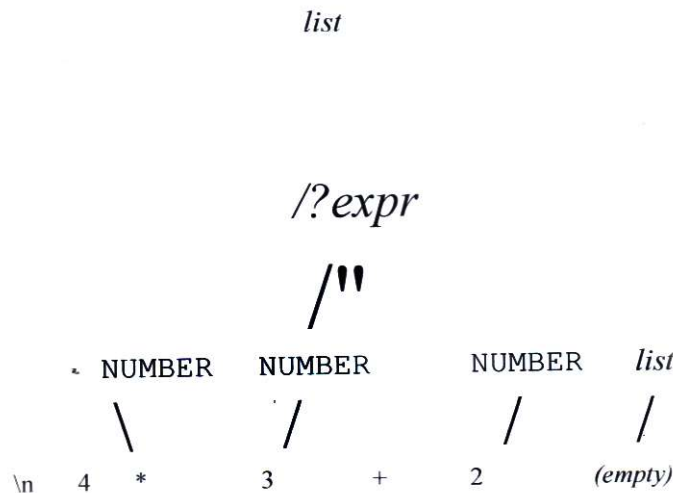
$\text{expr} : \text{NUMBER } \{ \$ \$:: \$ 1 ; \}$

$\$ 1$ مقداری است که با شناسایی NUMBER برمی گردد و این مقدار به عنوان مقدار expr برمی گردد . رابطه خاص $\$ s :: \$ 1$ قابل حذف است زیرا $\$ \$$ همیشه برابر $\$$ فرض می شود مگر اینکه صراحتاً مقداری دیگری به آن نسبت داده شود .
در سطح بعد ، زمانی که قانون

$\text{expr} : \text{expr } ' + ' \text{expr } \{ \$ \$:: \$ 1 + \$ 3 ; \}$

است ، مقدار expr ، مجموع مقادیر دو جزء expr است . توجه کنید که $\$ 2, ' + '$ است . هر جزء شماره گذاری می شود .

در سطح بالاتر ، سطر جدید $\{ n \}$ به عبارت افزوده می شود ، به عنوان یک list شناسایی شده و مقدار آن چاپ می شود . اگر آخر ورودی ساختار اینچنینی داشته باشد ، فرآیند تجزیه به وضوح متوقف می شود . یک list ممکن است یک رشته (string) خالی باشد . به این ترتیب به خطوط ورودی خالی نیز عمل می شود .
ورودی yacc فرم خاصی ندارد . فرمی که ما استفاده می کنیم ، فرمت توصیه شده استاندارد است . در این چنین صورت ، عمل شناسایی با تجزیه ورودی ، برآورد فوری عبارت را نیز به همراه دارد . در موارد پیچیده تر (شامل $hoc4$ و نسخه های پس از آن) ، فرآیند تجزیه برای اجزای بعدی که تولید می کند . بهتر است که تجزیه را به صورت یک درخت تجزیه (parse tree) مشابه شکل 8-1 تصویر کنیم و مقادیر را به صورتی که محاسبه می شوند و از برگهای این درخت به سمت ریشه رشد می کنند ببینیم .



شکل 1-8 : درخت تجزیه گر برای $2+3*4$

مقادیر قوانینی که کامل شناسایی نشده اند روی یک پشته (stack) ذخیره می شوند . به این ترتیب مقادیر از یک قانون به دیگری انتقال می یابند . نوع داده های این پشته معمولاً یک nt/ است اما از آنجا که بر اعداد اعشاری عمل می کنیم باید پیش فرض را در نظر بگیریم

بیان

```
#define YYSTYPE double
```

نوع پشته را به صورت نوع مضاف (double) تنظیم می کند .

-گروههای ترکیبی که با تحلیل گر واژه ای شناسایی می شوند باید

مشخص شوند مگر اینکه تک کاراکتری مثل $\hat{+}$ یا $\hat{-}$ باشند . تعریف %token

یک یا چند مورد از این گروهها را تشریح می کند . ارتباط چپ یا راست در صورت لزوم یا استفاده از left% یا right% به جای %token مشخص می شود و ارتباط چپ به این معنی است که a-b-c به صورت (a-b)-c تجزیه می شود نه به صورت a-(b-c) . اولویت به وسیله مرتبه ظاهری تعیین می شود .

- نشانه ها در تعریف یکسان بالاترین اولویت را دارند . نشانه هایی که بعداً معرفی می شوند اولویت بالاتری دارند

. در این روش انتخاب گرامری ، مبهم است (یعنی ، روشهای مختلفی برای تجزیه بعضی ورودیها وجود دارد .) ،

اما اطلاعات اضافی در تعریفها ، این ابهام را دفع می کند . بقیه کد ، برنامه های مستقلی است که در نیمه دوم فایل

hocy می آیند :

```

Continuing hoc. y
#include <stdio.h>
#include <ctype.h>
char *programe; int      1* for error messages *1
lineno = 1;

main (argc, argv)      /* hoc1 */
{
    char *argv [ ] ;

    programe = argv[0]; yyparse
    ();
}

```

main, Yyparse را جهت تجزیه ورودی فرا می خواند . کل حلقه از یک عبارت به عبارت بعد با استفاده از گرامر و به کمک مجموعه ای از محصولات list انجام می گیرد .

قرار دادن یک حلقه اطراف فراخوانی yyparse در main و داشتن عملی برای list که مقدار را چاپ کند و فوراً برگرداند نیز به همین میزان قابل قبول خواهد بود .

در عوض yyparse, yylex را به کرات برای ورودی نشانه ها فرا می خواند . yylex ما ساده است . این فایل blank ها ، tab ها را رد می کند و رشته های اعداد را به یک مقدار عددی تبدیل می کند ، خطوط ورودی را برای گزارش خطا شمارش می کند و سایر کارکترها را به صورت خودشان برمی گرداند . از آنجا که گرامر تنها انتظار دارد که (n\, /, ,, * , - , + را ببیند ، سایر کارکترها سبب می شود که yyparse پیغام خطایی بدهد . بازگرداندن صفر ، سیگنال **end – file** را به yyparse می فرستد.

```

Continuing hoc. y 1*
yylex () {
    hoc1 *1

    int c;

    while ((c=getchar() == ' ' , :: c == '\t')

    if (c == EOF)
        return 0;
    if (c == '.' II isdigit(c))      /* number */
        { ungetc(c, stdin); scanf
          ("%lf", &yyl val); return
          NUMBER;
        }

    if (c == '\n')
        lineno++

    return c;
}

```

متغیر yyl val برای ارتباط بین تجزیه گر و تحلیل گر واژه ای استفاده می شود . این متغیر به وسیله yyparse مشخص می شود و نوعی مشابه پشته yacc دارد . yylex نوع یک نشانه را به عنوان مقدار تابعی اش برمی گرداند و yyllbval را به مقدار نشانه نسبت می دهد ، (در صورت وجود) . به عنوان مثال نوع یک عدد اعشاری NUMBER و مقدار آن مثلاً 12/34 است . برای بعضی نشانه ها

بخصوص تک کاراکترهایی نظیر '\n', '+' گرامر، مقدار را استفاده نمی کند و تنها نوع را استفاده می کند. در این صورت yyval نیازی به تنظیم و مقدار گذاری ندارد.

عبارت `yacc%token NUMBER` به یک دستور مشخص در فایل خروجی `yacc` به نام `y.tab.c` تبدیل می شود. بنابراین `NUMBER` همه جا می تواند به عنوان یک ثابت در برنامه `c` استفاده شود. `yacc` مقادیری را که با کاراکترهای `ASCII` مشابه نیستند انتخاب می کند.

اگر یک خطای ترکیبی وجود داشته باشد، `yyerror`, `yyparse` را با یک رشته که شامل یک پیغام رمزی به صورت `syntax error` است فرا می خواند. انتظار می رود که استفاده کننده `yacc` یک `yyerror` ایجاد کند. استفاده کننده `yacc` ما فقط رشته را به تابع دیگری وتابع اخطار (`warning`) منتقل می کند که اطلاعات بیشتری را چاپ کند. نسخه های بعدی `hoc` از اخطار مستقیماً استفاده می کنند.

```
yyerror(s) /* called for yacc syntax error */
char *s;
{
    warning(s, (char *) 0);
}

warning(s, t) /* print warning message */
char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, n "%s", t); fprintf(stderr,
        " near line %d\n", lineno);
}
```

به این ترتیب پایان برنامه های مستقل در `hoc.y` تعیین می شود.

کامپایل یک برنامه `yacc` یک فرآیند دو مرحله ای است:

```
$ yacc hoc.y $ ccy.
```

```
Tab. C $ hoc 1 -0 hoc 1 leaves output in Y . tab.c Leaves executable
Program in hoc 1
```

```
2/3 0.0000007
```

```
-3-4
```

```
hoc 1 : syntax error near line 1
```

```
$
```

تمرین 8-1 (ساختار فایل `y.tab.c` را امتحان کنید. (این فایل حدود 300 خط برای `hoc1` می باشد.)

ایجاد تغییرات منفی قبل از عبارت

قبلاً ادعا کردیم که استفاده از yacc ایجاد تغییر در یک زبان را سهولت می بخشد. به عنوان مثال، اجازه بدهید علامت منفی را به *hoc1* اضافه کنیم، به گونه ای که عباراتی نظیر 3-4 برآورد شوند و به عنوان خطاهای ترکیبی تلقی نشوند. دقیقاً دو خط باید به *hoc.y* اضافه شود. یک UNARY MINUS token جدید به آخر بخش ابتدایی اضافه می شود تا اینکه به علامت منفی بیشترین اولویت را بدهد.

```
% left '+' '-'
%left '*' '/'
%left UNARYMINUS /* new*/
```

به گرامر نیز یک محصول باری *expr* اضافه می شود:

```
expr : NUMBER {$ $=-$1;}
      : '-' expr %prec UNARYMINUS {$ $=-$2;} 1*new*1
```

%prec بیان می کند که علامت منفی (قبل از عبارت) اولویت UNARYMINUS (بالا) دارد. در اینجا عمل، تغییر علامت است. یک علامت منفی بین دو عبارت اولویت پیش فرض را می گیرد.

تمرین 2-8) اپراتورها % (قدرمطلق یا باقیمانده) و + قبل از عبارت را به *hoc1* اضافه کنید. راهنمایی: به *frexp(3).0* مراجعه کنید.

-انحرافی از make

تایپ دو دستور جهت کامپایل یک نسخه جدید *hoc1* مشکل ساز است. هر چند ساخت یک فایل پوسته ای (shell) برای انجام کار راحت است اما روش بهتری نیز وجود دارد. روشی که در آن *wiII* به طور کلی تعیین می کند که چه زمانی بیش از یک فایل منبع در برنامه وجود دارد. برنامه مشخصه *make* نحوه ارتباط اجزاء به یکدیگر را فرا می خواند. این برنامه زمانهایی را که در آنها اجزاء مختلف آخرین بار اصلاح شده اند چک می کند، مقدار مینیمم کامپایل مجدد لازم برای ساختن یک نسخه جدید پایا (consistent) را درک می کند و سپس فرآیندها را اجرا می کند. *make* همچنین پیچیدگی های فرآیندهای چند مرحله ای مثل *yacc* را در می یابد، بنابراین این موارد می توانند بدون اینکه تک تک مراحل خوانده شوند وارد مشخصه *make* بشوند.

در ضمن، زمانی که برنامه در حال تولید به اندازه کافی بزرگ است که روی فایل های منبع مختلفی کشیده شود، *make* مفید واقع می شود. هر چند این برنامه حتی برای فایل هایی به کوچکی *hoc1* نیز مناسب است. در اینجا مشخصه *make* برای *hoc1* آورده شده است. در اینجا *make* در فایلی به نام *makefile* وارد شده است.

```
$ cat makefile
hoc1 : hoc.o
      choc.o -o hoc1
$
```

این خطوط حاکی از آن است که `hoc1` به `hoc.o` وابسته است و `hoc.o` با اجرای کامپایلر `cc C` و قرار دادن خروجی در `hoc1` ، به `hoc1` تبدیل می شود . `make` از قبل می داند که چگونه فایل منبع `yacc` در `hoc.y` را به یک فایل شیمی (object) `hoc.o` تبدیل کند .

```
$ make                          Make the first thing in makefile , hoc 1
Yacc hoc . Y
CC - C Y.tab.C
Rm Y.tab.c
Mv y.tab.o hoc.o
Cc hoc.o -o hoc 1
$ make                          Do it again
`hoc 1 ` is up to date . $      make realizes it's unnecessary
```

8-2 (مرحله 2: متغیرها و بازگشت از خطا) *error recovery*

مرحله بعد (یک مرحله کوچک) افزودن حافظه به `hoc1` برای ساخت `hoc2` است . حافظه 26 متغیر است که از `a` تا `z` نامگذاری می شوند . این مرحله خیلی رسمی نیست اما یک مرحله میانی ساده و مفید می باشد . همچنین چند مورد بررسی خطا خواهیم افزود . اگر `hoc1` را امتحان کنید، متوجه می شوید که برخورد آن با خطاهای ترکیبی چاپ کردن پیغام و متوقف شدن است و اصلاح خطاهای حسابی مثل تقسیم بر صفر امکانپذیر می باشد .

```
$ hoc 1
1/0
Floating $ exception - core dumped
```

تغییراتی که برای این موارد لازم است ساده بوده و حدود 35 خط کد می باشد تحلیل گر واژه ای `yylex` ، باید حروف را به عنوان متغیرها شناسایی کند . گرامر باید محصولاتی به فرم زیر را در بر بگیرد:

```
expr:  VAR
      Var ' = ' expr
```

یک عبارت می تواند رابطه ای را شامل شود که چند جایگزینی همزمان نظیر

$X=Y=Z=0$

را امکانپذیر می کند .

روش ساده تر جهت ذخیره مقادیر متغیرها استفاده از یک آرایه-26- المانی است. نام متغیر تک حرفی می تواند برای ایندکس گذاری آرایه استفاده شود. اما اگر بنا باشد گرامر هم اسامی متغیر و هم مقادیر آنرا در یک پشته ذخیره کند ، بایستی به `ya.cc` گفته شود که پشته آن شامل یک واحد دوگانه و یک `int` است نه فقط یک واحد دوگانه . این مورد بلیک واحد معرفی- ----- نزدیک سطح بالا انجام می گیرد . یک `define#` بلیک `typedef` برای تنظیم پشته به یک نوع پایه ای مثل نوع مضاعف مناسب است . اما مکانیسم واحد برای انواع واحد لازم است ، زیرا `yacc` در عباراتی نظیر `$$=$2` -- برای تعیین پایایی چک می کند . در این قسمت بخش گرامری `hoc.y` برای `hoc.2` آورده شده است :

```
$ cat hoc.y
%{
double mem[26]; %}      /* memory for variables  'a'..'z' *1
%union
    {
        /* stack type *1
        double  val;    /* actual value */
        int     index; /* index into mem[] *1
    }
%token <val> NUMBER
%token <index> VAR
%type <val> expr
%right
%left '+' '-'
%left '*' '/'
%left UNARYMINUS
%%
list:      /* nothing *1 list
'\n'
list expr '\n' list      { printf("\t%.8g\n", $2); }
error '\n'              { yyerror; }

expr:      NUMBER
VAR        { $$ = mem[$1]; }
VAR '=' expr { $$ = mem [ $ 1 ] - $ 3; }
expr '+' expr { $$ = $1 + $3; } expr
 '-' expr { $$ = $1 - $3; } expr '*'
expr { $$ = $1 * $3; } expr ' / ' expr
{
    if ($3 == 0.0)
        execerror ("division by zero". "");
    $$ = $1 / $3; } : '(' expr ')' { $$ = $2; } : '-'
expr %prec UNARYMINUS { $$ = -$2; }

%%

/* end of grammar *1
```

`Union declaration%` بیان می کند که المانهای پشته یک مضاعف (معمولاً یک عدد) و یا یک `int` را نگه می دارند که ایندکسی برای حافظه آرایه است . به `token declaration%` با یک نشانه گر نوع اضافه شده است .

`type declaration%` تعیین می کند که `expr` عضو `<val>` واحد است ، به این معنی که یک مضاعف اطلاعات نوع به `yacc` این

امکان را می دهد که مراجعی برای تصحیح اعضای واحد تولید کند. همچنین توجه کنید که = ارتباط دهنده از راست است در حالیکه سایر اپراتورها ارتباط دهنده از چپ می باشند .

بررسی خطا در بخشهای مختلف می آید . يك مورد واضح **تستی** است که برای تقسیم بر صفر انجام می گیرد . اگر این مورد رخ دهد يك برنامه **مستق** خطا `execerror` نامیده می شود .

تست دوم گرفتن سیگنال `floating point excaption` (استثناء عدد اعشاری) است که زمانی رخ می دهد که یک عدد اعشاری سرریز کند . (overflow) این سیگنال در `main` تنظیم شده است .

بخش نهایی بازگشت از خطا ، افزودن يك محصول برای خطاست .خطا يك کلمه ذخیره شده در يك گرامر `yacc` است . این کلمه راهی برای فهمیدن و برگشتن از يك خطای ترکیبی فراهم می کند . اگر خطایی رخ دهد . `yacc` نهایتاً از يك محصول استفاده می کند ،خطا را به عنوان **مرودی** که از نظر گرامری درست است تلقی می کند و برمی گردد . عمل `yyerror` نشانه (flay) را در تجزیه گر تنظیم می کند که به آن اجازه بازگشت به يك حالت تجزیه منطقی می دهد . بازگشت از خطا در هر تجزیه گری مشکل است . باید توجه کرد که در اینجا تنها مراحل ابتدایی را در نظر گرفته ایم و نیز به سرعت از روی قابلیت های `yacc` گذاشته ایم . عملیات گرامری `hoc2` - خیلی تغییر نمی کند . در اینجا `main` است که به آن `setjmp` را افزوده ایم تا يك حالت کاملاً مناسب برای تکرار بعد از يك خطا را ذخیره کند . `execerror` ، انطباق `longjmp` را انجام می دهد . (به بخش 5-7 برای توضیح `longjmp, setjmp` مراجعه کنید .)

```

jmp_buf begin;

main(argc, argv)          /* hoc2 */
{
    char *argv[];

    int fpecatch();

    progname = argv[0];
    setjmp(begin) ;
    signal (SIGFPE, fpecatch);
    yyparse ( ) ;
}

execerror(s, t) 1* recover from run-time error *1
char *s, *t;
{
    warning ( s, t);
    longjmp(begin, 0);
}

fpecatch ( ) {
    1* catch floating point exceptions *1
    execerror("floating point exception", (char *) 0);
}

```

برای اشکال زدایی ، لغو فراخوانی `execerror` مناسب است . (به `abort(3)` مراجعه کنید) که به يك کپی هسته ای منجر می شود که می تواند با `adb` یا `sdb` خوانده شود . زمانی که برنامه نسبتاً قویتر است ، لغو یا `longjmp` جایگزین می شود . خط جدید تحلیل گر واژه ای اختلاف کوچکی در `hoc2` است . يك تست اضافی برای حروف کوچک وجود دارد و از

آنجا که `yylval` يك واحد است، عضو مناسب باید قبل از اینکه `yylex` تنظیم شود، برگردد. در اینجا بخشهایی که باید تغییر کنند آورده شده اند:

`yylex()`

```

if(c == '.' || isdigit(c)) { ungetc ( c , stdin);
    scanf("%lf", &yylval.val); return
    NUMBER;
}
if (islower(c)) {
    yyl val. index return  = c - 'a'; /* ASCII only */
    VAR;
}

```

مجدداً توجه کنید که چگونه نوع نشانه (به عنوان مثال NUMBER) متمایز از مقدار آن است. (مثلاً 3/1416)

در اینجا متغیر و بازگشت از خطا را که موارد جدیدی در `hoc2` هستند بیان کنیم:

```

$ hoc2
x :: 355

          355
y :: 113
          113
p :: x/Z          z is undefined and thus zero
hoc2: division by zero near line 4 Error recovery
x/y
3.1415929
1e30 ... 1e30          Overflow
hoc2: floating point exception near line 5

```

در واقع `ppp-II` ساختار ویژه ای برای تشخیص سرریز عدد اعشاری دارد اما روی بیشتر ماشینهای دیگر `hoc2` همانطور که نشان داده شده عمل می کند. تمرین 3-8) قابلیت برای حفظ جدیدترین مقدار محاسبه شده اضافه کنید به گونه ای که نیازی به تایپ مجدد در یکسری از محاسبات مرتبط نباشد. یک راه حل این است که از طریق یکی از متغیرهای `make` عمل کنیم به عنوان مثال 'p' برای 'o-previous'. تمرین 4-8) `hoc` را به گونه ای اصلاح کنید که یک سمی کال بتواند به عنوان یک خاتمه گر عبارت عمل کند. معادل یک `newline-o`.

3-8 مرحله 3: اسامی متغیر دلخواه، توابع داخلی

در این نسخه، نسخه `hoc3`، تعداد زیادی قابلیت جدید و مقدری مرتبط با کد اضافی، افزوده می شود. ویژگی جدید اصلی دسترسی به توابع داخلی زیر است:

Sin cos dtam exp log logio sqrt int abs

يك اپراتور توان نیز اضافه کرده ایم. این اپراتور بالاترین اولویت را دارد و ارتباط دهنده از راست است. از آنجا که تحلیل گر واژه ای باید با **اسهای** داخلی بلندتر از يك کاراکتر مواجه شود، سعی در جهت افزایش طول اسامی متغیرها کار بیهوده ای نیست. جداول علایم پیچیده تري برای ذخیره سازی مسیر این متغیرها خواهیم داشت. زمانی که این جدول را داشته باشیم می توانیم آنرا با نامها و مقادیری برای بعضی ارتباطات مفید از پیش فراخوانی کنیم.

PI	E	3.14159265358979323846	π
GAMMA		2.71828182845904523536	Base of natural logarithms
DEG		0.57721566490153286060	Euler-Mascheroni constant
PHI		57.29577951308232087680	Degrees per radian Golden ratio
		1.61803398874989484820	

نتیجه حاصل، یک ماشین حساب مفید است:

```
$ hoc3
1.5^2.3
2.5410306
exp(2.3*log(1.5))
2.5410306
sin (PI/2)
1
atan (1)*DEG
45
```

رفتار را نیز تا حدی سازماندهی کرده ایم. در `hoc2`، رابطه `x=expr` نه تنها رابطه را ایجاد می کند، بلکه مقدار را هم چاپ می کند زیرا همه عبارات چاپ می شوند.

```
$ hoc2
x=2*3.14159
6.28318
```

Value printed for assignment to variable

در `hoc3`، تمایزی بین رابطه ها و عبارات ساخته می شود، مقادیر تنها برای عبارت چاپ می شود.

```
$ hoc3
x=2*3.14159
x
```

Assignment : no value is printed

```
6.28318
```

: Expression

حدود 250 خط) که بهتر است به فایل های جداگانه ای

Value is printed

برنامه ای که بعد

برای ویرایش ساده تر و کامپایل سریعتر شکسته شوند. اکنون 5 فایل به جای یک فایل وجود دارد:

hoc.y	Grammar, main, yylex (as before)
hoc.h	Global data structures for inclusion
symbol.c	Symbol table routines: lookup, install
init.c	Built-ins and constants; init
math.c	Interfaces to math routines: Sqrt, Log, etc.

برای این جداسازی باید بیشتر راجع به اینکه چگونه یک برنامه چند فایل C را سازماندهی کنیم و بیشتر راجع به `make` بدانیم برای اینکه بخشی از کار را برای ما انجام دهد.

به زودی به `make` برمی گردیم. در ابتدا اجازه دهید که به کد جدول علامت نگاهی بیندازیم. یک علامت، نام، نوع (VAR یا BLTIN) و یک مقدار دارد. اگر علامت Var، باشد، مقدار یک مضاعف است. اگر BLTIN باشد مقدار یک اشاره گر به تابعی است که یک مضاعف برمی گرداند. این اطلاعات در `hoc.y`، `symbol.c`، `init.c` مورد نیاز است.

می توانستیم تنها سه کپی بسازیم اما زمانی که یک تغییر ساخته می شود احتمال اینکه اشتباه کنیم یا اینکه فراموش کنیم که یک کپی را جدید کنیم بسیار زیاد است. در عوض اطلاعات معمول را در یک فایل سرآمد `hoc.h` قرار می دهیم. هر فایلی که به این فایل نیاز داشته باشد آنرا وارد می کند. (پسوند `h` مرسوم است اما با هیچ برنامه ای تحصیل نمی شود.) همچنین به فال `make` این حقیقت را که این فایلها به `hoc.h` بستگی دارند اضافه می کنیم به نحوی که زمانی که تغییری رخ می دهد کامپایل مجدداً انجام شود.

```
$ cat hoc.h
typedef struct
  char
  short
  Symbol { 1* symbol table entry *1 *name;
          type;
          1* VAR, BLTIN, UNDEF *1
  union {
    double val;          1* if VAR *1 1* if
    double (*ptr)();     BLTIN *1
  } U;
  struct Symbol *next; 1* to link to another *1
} Symbol;
Symbol *install(), *lookup(); $
```

نوع VAR, UNDEF ای است که هنوز یک مقدار به آن نسبت داده نشده است. علایمی که با هم در یک `list` مرتبط می شوند، دو علامت قسمت `next` را استفاده می کنند.

خود `list` برای `symbol.c` محلی است. تنها دسترسی به آن از طریق جستجو و نصب توابع است. این روش، تغییر به ساختار جدول علایم را در صورت لزوم آسان می سازد. (یکبار آنرا انجام داده ایم.) `lookup`، `list` را برای دستیابی به یک `na!Tie` ویژه جستجو می کند اگر اسم علامت را پیدا کند به آن یک اشاره گر برمی گرداند و در غیر اینصورت به آن صفر برمی گرداند. جدول علامت از جستجوی خطی استفاده می کند، که از آنجایی که متغیرها تنها در طی تجزیه جستجو می شوند و نه در حین اجرا کاملاً برای ماشین حساب واکنشی ما مناسب است.

Install يك مقدار را با نوع و مقدار مرتبطش در بالای list قرار می دهد .
 malloc , emalloc و تخصیص دهنده ذخیره استاندارد را فرامی خواند (<<malloc3) و
 نتیجه را چک می کند . این سه برنامه مستقبل محتوای symbol.c هستند . فایل y.tab.h
 با اجرای yacc.d تولید می شود . این فایل دستور #define ای را که yacc برای نشانه
 هایی نظیر VAR , NUMBER , BLTIN ، ... تولید کرده است ، شامل می شود .

```

$ cat symbol.c #include
"hoc.h" #include
"y.tab.h"

static Symbol *symlist = 0;      /* symbol table: linked list */

Symbol *lookup(s)                /* find s in symbol table */
{
    char *s;

    Symbol *sp;

    for (~Jp = symlist; sp != (Symbol *) 0; sp = if sp->next)
        (strcmp(sp->name, s) == 0)
            return sp;
    return 0;                    /* 0 ==> not found */
}

Symbol *install(s, t, d)         /* install s in symbol table */
char *s; int t
double d;

{
    Symbol *sp; char
    *emalloc ();

    sp = (Symbol *) emalloc(sizeof(Symbol));
    sp->name = emalloc(strlen(s)+1); /* +1 for '\0' */
    strcpy(sp->name, s);
    sp->type = t;
    sp->u.val = d;
    sp->next = symlist; /* put at front of list */ symlist = sp;
    return sp;
}

char *emalloc(n)                /* check return from malloc */
unsigned n;
{
    char *p, *malloc();

    p = malloc(n);
    if (p == 0)
        execerror ("out of memory", (char *) 0);
    return p;
}

```

فایل init.c شامل تعاریفی برای ثوابت (PI ، ...) و اشاره گرهای تابعی برای توابع داخلی است . این موارد در جدول علامت به وسیله

تابع `init` که با `main` فراخوانی می شود، نصب می گردند.

```

$ cat init.c #include
"hoc.h" #include "y.tab.h"
#include <math.h>

extern double      Log(), Log10(), Exp(), Sqrt(), integer();
static struct {
    char          *name;
    double cval;
} consts[] = {
    "PI",      3.14159265358979323846,      "E",
    2.71828182845904523536,      "GAMMA",
    0.57721566490153286060,      "DEG",
    57.29577951308232087680,      "PHI",
    1.61803398874989484820, 0, 0
};

static struct {
    char double *name; (*func)
} builtins[] = {
    "sin", sin, "cos",
    cos,
    "atanlt, atan,
    "log", Log,
    "exp", Exp,
    "sqrt", Sqrt,
    "abs", f abs,
    0, 0
};

init() {
    int i; Symbol *S;

    for (i = 0; consts[i].name; i++)
        install(consts[i].name, VAR, consts[i].cval);
    for (i = 0; builtins[i].name; i++) {
        S = install(builtins[i].name, BLTIN, 0.0);
        S->u.ptr = builtins[i].func;
    }
}

```

داده ها به جای اینکه وارد کد شوند در جداول ذخیره می شوند. زیرا فراخوانی و ایجاد تغییر در جداول ساده تر است. از آنجا که جداول تنها در این فایل قابل مشاهده اند نه در کل برنامه، جداول استاتیک نامیده می شوند. بزودی به برنامه های مستقل ریاضی

sqrt,log برمی گردیم . با ساخت در محل ، می توان تغییراتی در گرامری که از این توابع داخلی استفاده می کند ، ایجاد کرد

```

$ cat hoc.y
"{
#include "hoc.h" extern
double Pow(); "
"union
{
    double val;    /* actual value */
    Symbol "sym;   /* symbol table pointer */
}
"token <val>    NUMBER
"token <sym>    VAR BLTIN UNDEF expr
"token <_val>   asgn
"type
"right '+' '-'
"left '*' '/'
"left UNARYMINUS
"left 'A,      /* exponentiation */
"right
""          /* nothing */ :list
list:      '\n'
           list asgn '\n'
           list expr '\n' :list      { printf("\t%.8g\n", $2); }
           error '\n'                { yyerror; }

asgn:
    VAR '=' expr { $$=$1->u.val=$3; $1->type =VAR; }

expr:
    NUMBER
    : VAR { if ($1->type ==UNDEF)
              execerror("undefined          variable", $1->name);
              U=$1->u.val; }
    : asgn
    I BLTJCN '(' expr ')' { $$ = (*($1->u.ptr))($3); }
    I expr '+' expr { $$ = $1 + $3; } '-'
    I expr expr { $$ = $1 - $3; } , ..'
    I expr expr { $$ = $ 1 * $ 3 ; } , '/' expr
    I expr {
    I     if ($3 == 0.0)
    I         execerror( "division by zero", "");
    I     $$ = $1 / $3; }
    : expr 'A, expr { $$ = POW($1, $3); }
    : '(' expr ')' {U = $ 2 ; }
    : '-' expr %prec UNARYMINUS { $$ = -$2; }
""

1* end of grammar */

```

اکنون گرامر برای جایگزینی و ایجاد رابطه علاوه بر `expr` , `asgn` دارد ؛ یک خط ورودی که فقط شامل

VAR expr

باشد یک جایگزین (رابطه) است و بنابراین هیچ تعدادی چاپ نمی شود . توجه کنید که افزودن توان به گرامر حاوی ارتباط دهنده راست آن ساده است .

پشته `yacc` یک واحد متفاوت دارد . به جای مراجعه به یک متغیر با ایندکس آن در یک جدول 26 المانه ، یک اشاره گر به یک عنصر نوع علامتی وجود دارد . فایل سرآمد `(hoc.c) header` ، تعریف این نوع را در برمی گیرد .

تحلیل گر واژه ای اسامی متغیر را می شناسد ، آنها را در جدول علامت جستجو می کند و تصمیم می گیرد که آیا متغیر (VAR) است یا داخلی (BLTIN) . نوعی که با `yylex` برمی گردد یکی از اینهاست . هم متغیرهای تعریف شده توسط کاربر و هم متغیرهای از پیش تعریف شده مثل `PI` از نوع `VAR` هستند . یکی از خواص یک متغیر این است که آیا به آن یک مقدار نسبت داده شده است یا خیر ، بطوریکه استفاده از یک متغیر نامشخص می تواند سبب ایجاد یک پیغام خطا توسط `yyparse` شود . تستی که تعیین می کند آیا یک متغیر مشخص شده است یا خیر باید در گرامر باشد نه در تحلیل گر واژه ای . زمانی که یک `VAR` ، واژه ای تشخیص داده شود ، فضای آن هنوز مشخص نشده است . نمی خواهیم با این اعتراض مواجه شویم که علیرغم اینکه فضای `X` کاملاً مجاز است اما هنوز مشخص نشده است . (مثل سمت چپ رابطه یا نظیر `X==1`)

در اینجا بخش تصحیح شده `yylex` آورده شده است :

```

yylex ( )          /* hoc3 */

if (isalpha(c) {
    Symbol *s;
    char sbuf[100], *p == do {sbuf;
        *p++ == c;
    } while «c:::getchar(» 1= EOF && isalnum(c); ungetc(c,
    stdin);
    *p == '\0 ' .
    if «s==lookup(sbuf) == 0)
        s == install(sbuf, UNDEF, 0.0); yylval.sym :::
    a;
    return s-' .type == UNDEF ? VAR : a->type;
}

```

`Main` یک خط اضافی دارد که برنامه مستقل ابتدای `init` را می خواند تا توابع داخلی اسامی از پیش تعیین شده مثل `PI` را در جدول علامت نصب کنند .

```

main(argc, argv)          /* hoc3 */
  char *argv[];
{
    int fpecatch();

    progname = argv[0]; init( );
    setjmp(begin) ;
    signal (SIGFPE, fpecatch);
    yyparse ( ) ;

```

تنها فایل باقیمانده `math.c` است. برخی از توابع ریاضی استاندارد رابطی برای چک کردن خطا برای پیغام و بازگشت از آن نیاز دارند. به عنوان مثال تابع `sqrt` در صورتیکه آرگانش منفی باشد به سادگی، صفر برمیگرداند. کد در `math.c` تستهای خطای جلد 2 راهنمای برنامه نویسی UNIX را استفاده می کند. برای این منظور به فصل 7 مراجعه کنید. این روش قابل اعتماد تر و اجرایی تر از این است که خودمان تستها را بنویسیم. زیرا محدودیت های خاص برنامه منطبقاً در کد `afficial` (اصلی) به بهترین وجه منعکس می شوند. فایل سرآمد `<math.h>` تعاریف نوع برای توابع استاندارد ریاضی را شامل می شود. `<error.h>` اسامی خطاهایی را که یافت می شوند را در برمی گیرد.

```

$ cat: mat:h.c
#include <math.h> #include
<errno.h>
extern int      errno;
double errcheck();

double Log(x)
{
    double X;

    return errcheck(log(x), "log");
} double Log10(x)
    double X;

{
    return errcheck(log10(x), "log10");
} double Exp(x)
    double x;

{
    return errcheck(exp(x),      "exp");
} double Sqrt(x)
    double x;

{
    return errcheck(sqrt(x), "sqrt");
}

double Pow (x, y)
double x, y;
{
    return errcheck(pow(x,y), "exponentiation");
}
double integer(x)
    double x;
{
    return (double) (long)x;
}
double errcheck(d, s)      /* check result of library call */
double d;
char *s;
{
    if (errno == EDOM) {
        errno = 0;
        execerror(s, "argument out of domain");
    } else if (errno == ERANGE) {
        errno = 0;
        execerror(s, "result out of range");
    }
    return d;
}
$

```

زمانی که yacc را روی گرامر جدید اجرا می کنیم یک تشخیص غیرگرامری جالب رخ می دهد:

```
$ yacc hoc.y
```

```
Conflicts: 1 shift/reduce $
```

پیغام shift /reduce به این معنی است که گرامر hoc3 ، مبهم است : تک خط ورودی

```
X=1
```

به دور روش می تواند تجزیه شود :

$$\frac{\quad}{+} \quad \backslash n$$

list asgn

$$\overset{n}{(empty) x = '1}$$

تجزیه گر یا باید تصمیم بگیرد که *asgn* به یک *expr* و سپس به یک *list*، در قسمت چپ درخت تجزیه گر، کاهش یابد یا اینکه تصمیم بگیرد *n/* بعدی *shift* و انتقال را فوراً استفاده کند و همه را به یک *list* بدون قانونی میانی در سمت راست درخت تبدیل کند. با این ابهامی که پیش می آید، *yacc* انتقال را انتخاب می کند. زیرا این کار تقریباً همیشه کار درستی در برخورد با گرامرهاست. باید تلاش کنید این پیغامها را درک کنید تا مطمئن شوید که *yacc* تصمیم درستی گرفته است. اجرای *yacc* با گزینه *-V*، یک فایل پر حجم به نام *y.out put* تولید می کند که به علت مشکل اشاره می کند.

تمرین 5-8- همانطور که *hoc3* بیان می کند، $PI=3$ ، قانونی است. آیا این ایده، ایده خوبی است؟ چگونه *hoc3* را به منظور پیشگیری از نسبت دادن مقادیر به ثوابت اصلاح می کنید؟

تمرین 6-8- تابع داخلی $a \tan^2(y.x)$ را که زاویه ای را که \tan آن $\frac{y}{x}$ است اضافه کنید. تابع داخلی $\text{rand}()$ را که یک متغیر تصادفی اعشاری براساس توزیع یکنواخت روی فاصله (اوه) می دهد را اضافه کنید. چگونه باید گرامر را تغییر دهید که توابع داخلی با مقدار متفاوتی از آرگومانها اجازه عمل دهید؟

تمرین 7-8- چگونه یک قابلیت به منظور اجرای دستورات *hoc* مشابه با ویژگی سایر برنامه های UNIX اضافه می کنید؟

تمرین 8-8- کد را در *math.c* تصحیح کنید بطوریکه از جدول به جای یکسری از توابع یکسانی که تولید کردیم استفاده کند.

انحراف دیگری بر *make*:

از آنجا که برنامه *hoc3* اکنون به 5 فایل وابسته است نه تنها یک فایل، فایل *make* پیچیده تر است. پیغام *vedvce/veduce conflict* در *yacc* حاکی از آن است که به جای یک ابهام داخلی معمولاً نشانه ای از یک خطای واضح گرامری دیده می شود.

```

$ cat makefile
YFLAGS = -d      # force creation of y.tab.h
OBJS = hoc.o init.o math.o symbol.o # abbreviation

hoc3: $(OBJS)
      cc $(OBJS) -lm -O hoc3

hoc.o: hoc.h

init.o symbol.o:      hoc.h y.tab.h

pr:
      @pr hoc.y hoc.h init.c math.c symbol.c makefile

clean:
      rm -f $(OBJS) y.tab.[ch]
$

```

خط `YFLAGS=-d` گزینه `-d` را به خط دستور `yacc` که به `make` تولید شده می افزاید ، این گزینه به `yacc` می گوید که فایل `y.tab.h` دستورات `Iddefine` را تولید کند . خط `OBJS=` اختصاری برای مفهومی که ساخته می شود و به کرات استفاده خواهد شد ، تعیین می کند . ترکیب ، مشابه متغیرهای پوسته ای نیست؛ پرانتزها اجباری اند . `flag-lm` کتابخانه ریاضی را نتیجه می دهد که برای تابعهای ریاضی مورد جستجو قرار می گیرد .

اکنون `hoc3` به فایلهاي `four.o` بستگی دارد . بعضي از `o` فایلهاي به `h` فایلها بستگی دارند . با وجود این وابستگی ها ، `make` می تواند نتیجه بگیرد که چه کامپایل مجددی بعد از اینکه تغییرات در هر يك از فایلهاي درگیر صورت گرفت مورد نیاز است . اگر می خواهید ببینید که `make` بدون اینکه فرآیندها را اجرا کند چه انجام خواهد داد ،

`$make-n`

را امتحان کنید .

از طرف دیگر ، اگر بخواهید زمانهای فایل را به یک حالت پایا تحمیل کنید ، گزینه `touch` ("t"- آنها را بدون انجام هیچ مرحله کامپایل ، جدید (update) می کند .

توجه کنید که ما نه تنها یکسری وابستگی به فایلهاي منبع را اضافه کرده ایم ، بلکه برنامه های خدماتی متنوعی هم اضافه کرده ایم . تمامی اینها در یک مکان به ترتیب قرار گرفته اند . `make` با پیش فرضش ، اولین چیزی را که در فایل لیست شده است می سازد . اما اگر یک مورد را به گونه ای نامگذاری کنید که وابستگی خاصی را نشان دهد ، مثل `symbol.o` یا `make,pr` اول آنرا می سازد . یک تابعیت خالی به این معنی است که آن مورد هیچ گاه جدید نمی شود و تغییر تنها در صورتی رخ می دهد که صریحاً خواسته شود .

بنابراین

`make prflpr $`

نوع `list` را که شما روی یک چاپ خطی خواسته اید تولید می کند . (علامت `@` در `"pr@"` از انعکاس دستوری که توسط `make` در حال اجراست جلوگیری می کند .

\$ make clean

فایل‌های خروجی yacc و فایل‌های صفر(0) را حذف می‌کند .

این مکانیسم تابعیتهای خالی در فایل make نسبت به فایل پوسته ای به عنوان راهی برای نگهداری مقامی ارتباطات در یک تک فایل ارجح است . در ضمن make به طراحی برنامه محدود نمی‌شود. این مورد جهت بسته بندی هر سری از اپراتورها که تابعیتهای زمانی دارند مناسب است .

- انحرافی از lex

برنامه lex تحلیل گر واژه ای را که در یک رفتار مشابه با روشی که yacc تجزیه گر را تولید می‌کند ، ایجاد می‌کند . یک مشخصه قوانین واژه ای زبان‌تان رامی نویسد و از عبارات منظم و قطعات (c fragments) که زمانی که یک رشته منطبق شونده پیدا می‌شود ، اجرا می‌شوند ، استفاده می‌کنید . lex آنرا به یک تشخیص دهنده ترجمه می‌کند . yacc,lex با مکانیسم تحلیل گر واژه ای که قبلاً نوشته ایم همکاری می‌کنند . در اینجا در مورد جزئیات ریز lex توضیح نمی‌دهیم . بحث بعدی برای تشویق شما به آموختن بیشتر است . برای این منظور به راهنمای lex در جلد 2B راهنمای برنامه نویسی UNIX مراجعه کنید .

در ابتدا در این قسمت برنامه lex ای از فایل lex.1 - آورده شده است . این برنامه تابع yylex را که تا بحال استفاده کرده ایم جایگزین می‌کند .

```
$ cat lex . 1
%{
#include "hoc.h"
#include "y.tab.h"
extern int lineno;
%}
%
%

[t] {} /* skip blanks and tabs */
[0-9]+\.[0-9]*\.[0-9]+ {
    sscanf(yytext, "%lf", &yyval.val); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* {
\n    Symbol *s;
    if «s=lookup(yytext) == 0)
$        s = install (yytext, UNDEF, 0.0); yyval.sym = s;
        return s->type == UNDEF ? VAR : s->type; }
Each "rule" is a regular expression like those in egrep or awk, except that
lex recognizes C-style escapes like \t and \n. The action is enclosed in
{ lineno++; return '\t'; } /* everything else */
{ return yytext[0]; }
```

هر قانون يك عبارت منظم شبیه آنهایی است که در- egrep یا- awk وجود دارد . با این تفاوت که- escape,lex (فرار- خروج) فضای- c شامل مثل -n,t را درک می‌کند . عمل در براکت قرار- ---- می‌گیرد . قوانین به ترتیب امتحان می‌شوند و مفاهیمی نظیر- *و+ تا

وقتی که ممکن باشد بر يك - رشته بند منطبق می شوند . اگر قانون با بخش بعدی ورودی منطبق شود ، عمل اجرا می شود . رشته ورودی که انطباق نشان داده ، يك رشته lex به نام yacc قابل دسترسی است .
 Makefile جهت استفاده از lex باید تغییر کند :

```
$ cat makefile
YFLAGS = -d
OBJS = hoc.o lex.o init.o math.o symbol.o

hoc3:    $(OBJS)
        cc $(OBJS) -lm -ll -O hoc3

hoc.o:   hoc.h

lex.o init.o symbol.o: hoc.h y.tab.h

$
```

مجدداً ، make می داند که چگونه يك فایل 1 به فایل صفر دسترسی پیدا کند . تمام آنچه که از ما می خواهد اطلاعات وابستگی است . (همچنین باید کتابخانه lex11 را به لیستی که با CC جستجو می شود اضافه کنیم . زیرا شناساگر تولید شده کامل نیست .) خروجی جالب و کاملاً اتوماتیک است :

```
$ make yacc -d
hoc.y

conflicts: 1 shift/reduce
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
lex lex.1
cc -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc -c init.c
cc -c math.c
cc -c symbol.c
cc hoc.o lex.o init.o math.o symbol.o -lm -ll -O hoc3 $
```

اگر تک فایل عوض شود ، make تک دستور برای ساخت نسخه جدید (up to data) کافی است .

```

$ touch lex.1          Change modified-time of lex. 1
$ make
lex lex. 1
cc -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc hoc.o lex.o init.o $ math.o symbol.o -ll -lm -0 hoc3

```

بحث کرده ایم که آیا با lex به عنوان یک مورد جانبی عمل کنیم که خیلی مختصر نشان داده شده و سپس حذف شود. یا اینکه به عنوان یک ابزار اولیه برای تحلیل واژه ای زمانی که زبان پیچیده میشود. در هر دو مورد بحثهایی وجود دارد. مشکل اصلی ما Lex (گذشته از اینکه نیاز دارد کاربر زبان دیگری را نیز بیاموزد) این است که سرعت اجرای آن و تولید شناساگرهای بزرگتر و کوچکتر آن نسبت به نسخه های مشابه C کمتر است. همچنین در صورتی که یک مکانیسم ورودی به آن عملی غیرمعمول برای lex، نظیر بازگشت از خطا یا حتی ورودی از فایلها را دربرگیرد، تطبیق این مکانیسم برای lex نسبتاً مشکلتر است. هیچ یک از این موارد در فضای hoc جدیدی نیست. مشکل اصلی محدودیت در فضا است. توضیح نسخه lex صفحات بیشتری نیاز دارد، بنابراین (متأسفانه) ما به C برای تحلیلهای واژه ای بعدی برمی گردیم.

تمرین 8-9-8- سایزهای دو نسخه hoc3 را مقایسه کنید. راهنمایی: $size(1).0$ مراجعه کنید.

8-4- مرحله 4) کامپایل به یک ماشین

Hoes را که مفسری برای یک زبان با جریان کنترل است، در پیش داریم. hoc4 یک مرحله میانی است که توابعی مشابه hoc3 فراهم می کند. با این تفاوت که با چارچوب مفسر hoes عمل می کند. به این دلیل hoc4 را به این گونه نوشته ایم که دو برنامه یکسان رفتار کنند. این مسأله برای اشکال زدایی با ارزش است. همانطوری که ورودی تجزیه می شود، hoc4 کدی را برای یک کامپیوتر ساده به جای پاسخهای محاسبه شده فوری، تولید می کند. هرگاه به آخر یک دستور می رسد، کد تولید شده تفسیر می شود تا نتیجه مطلوب را محاسبه کند. این کامپیوتر ساده یک ماشین پشته است. زمانی که به یک کمیت تحت عمل برخورد می شود، این کمیت به داخل یک پشته وارد (push) می شود. (صحیحتر این است که بگوییم کدی برای راندن این کمیت به داخل پشته تولید می شود)؛ بیشتر از اپراتورها روی مواردی که در بالای پشته هستند عمل می کنند. به عنوان مثال برای بررسی رابطه

$$X=2*y$$

کد زیر تولید میشود.

```

constpush
  2
varpush
  y
eval mul
varpush
  x assign
pop
STOP

```

یک ثابت را به داخل پشته وارد کن

... ثابت 2

اشاره گر جدول علامت را به داخل پشته وارد کن

برای متغیر y

برآورد: اشاره گر را با مقدار جایگزین کن

دو مورد بالا را در هم ضرب کن، نتیجه آنها را جایگزین کن

اشاره گر جدول علامت را به داخل پشته وارد کن

... برای متغیر x

مقدار را در متغیر ذخیره کن، اشاره گر را خارج کن

مقدار بالا را از پشته پاک کن

پایان دستورات

زمانی که این کد اجرا می شود، عبارت برآورد میشود و همانطور که در دستورات نشان داده شده، نتیجه در x ذخیره می شود. Pop نهایی مقدار روی پشته را که دیگر مورد نیاز نیست از آن خارج می کند. ماشینهای پشته معمولاً مفسرهای ساده ای را نتیجه می دهند ماشین ما هم استثناء نیست - آن تنها یک آرایه شامل اپراتورها و کمیتهای تحت عمل آنهاست. اپراتورها دستورات ماشین هستند. هر یک فراخوان تابعی با آرگومانهایش (در صورت وجود) اند که دستورات را دنبال می کنند. باقی کمیتهای تحت عمل ممکن است همانند مثال بالا از قبل روی پشته باشند.

کد جدول علامت برای hoc4 با کد جدول علامت hoc3 یکسان است. همچنین شروع در init.c و توابع ریاضی در math.c یکسان است. گرامر، مشابه hoc3 است اما عملیات کاملاً متفاوت است. هر عمل، دستور ماشین و آرگانهای مرتبط با آنها را تولید می کند. به عنوان مثال سه مورد برای VAR در یک عبارت تولید می شوند: یک دستور Varpush، اشاره گر جدول علامت برای متغیر و یک دستور eval که اشاره گر جدول علامت را بعد از اجرا، با مقادیرش جایگزین می کند. کد برای '*' تنها mul است، چرا که کمیتهای آن از قبل روی پشته خواهند بود.

```

Scat hoc.y
"{
#include "hoc.h" #define
code2(c1,c2)          code(c1); code(c2) code(c1);
#define cOde3(c1,c2,c3)  code(c2); code(c3)
,,}
"union {
    Symbol *sym; /* symbol table pointer *I */
    Inst *inst;  machine instruction *I
}
<sym>  NUMBER VAR BLTIN UNDEF
"token ' '
"right '+' '-'
"left  '*' 'I'
"left  UNARYMINUS
"left  'A' /* exponentiation */
"right
"" /* nothing *I list
list:  '\n'
list asgn '\n' list code2 (pop, STOP); return 1; }
expr '\n' list error code2(print, STOP); return 1; }
'\n' { yyerror; }

asgn:   VAR '=' expr { code3(varpush, (Inst)$1, assign); }
expr:   NUMBER { cOde2(constpush, (Inst)$1); }
        VAR { code3(varpush, (Inst)$1, eval); }
        | asgn
        | BLTIN '(' expr ')' { cOde2(bltin, '(' expr
                                (Inst)$1->u.ptr); }
        | ')'
        | expr '+' expr { code( add); }
        | expr '-' expr { code ( sub); }
        | expr '*' expr code(mul); }
        | { expr '/' expr cOde(div); }
        | { expr 'A, expr { code (po\\,er); }
        | '-' expr %prec UNARYMINUS {code(negate); }

%%
/* end of grammar *I

```

Inst، نوع داده ای دستوری از ماشین است (اشاره گری به یک تابع که یک int برمی گرداند.) که به زودی به آن برمی گردیم. توجه کنید که آرگانها، اسامی توابعند، یعنی اشاره گرها به توابع یا مقادیر دیگری که به اشاره گرهای تابع نسبت داده شده اند. Main را تا حدی تغییر داده ایم. اکنون تجزیه گر پس از هر دستور یا عبارت برمی گردد. کدی را که تولید کرده اجرا می شود. Yyparse در پایان فایل صفر را برمی گرداند.

```

main(argc, argv)          /* hoc4 */
{
    char *argv[];

    int fpecatch();

    progname = argv[0];
    inite) ;
    set jmp (begin) ;
    signal (SIGFPE, fpecatch); for

```

تحلیل گر واژه ای تفاوت کوچکی دارد . تفاوت اصلی این است که اعداد باید از قبل ذخیره شوند نه فوراً روش ساده تری در این مورد، نصب کردن آنها (اعداد) در جدول در کنار متغیرهاست . در اینجا بخش تغییر یافته yylex آورده شده است :

```

yylex ( )                1* hoc4 *1

    if (c == '.' : I isdigit{c} { double|d;number *1
        ungetc(c, stdin); scanf("%lf"
            &d); yylval.sym= install("",
            return NUMBER;

                                NUMBER, d);

```

هر المان روی پشته ، مفسر یا یک مقدار اعشاری و یا یک اشاره گر به یک ورودی جدول علامت است . نوع داده های پشته یکی از اینهاست . خود ماشین آرایه ای از اشاره گرهاست که به برنامه های مستقلی نظیر mul که یک عملیات را انجام می دهد و یا به داده ها در جدول علامت اشاره می کند . فایل سرآمد hoc.h باید به گونه ای افزایش یابد که این ساختارهای داده ای و تعاریف تابعی را برای مفسر دربرگیرد . آنها در طول برنامه در جایی که مورد نیازند شناخته می شوند . (ما تصمیم گرفتیم که تمام این اطلاعات را به جای دو فایل در یک فایل قرار دهیم . در یک برنامه بزرگتر ، ممکن است بهتر باشد که اطلاعات سرآمد را به فایل های مختلفی تقسیم کنیم ، به گونه ای که هر یک تنها در جایی که مورد نیازند وارد شوند .)

```

$ cat hoc.h typedef
struct      Symbol { ///- symbol table entry *l *name;
    char    type;
    short   /* VAR, BLTIN, UNDEF */
    union {
        double ,val;           /* if VAR *l l* if
        double (*ptr) ();      BLTIN */
    } u;
    struct Symbol *next; ///- to link to another *l
} Symbol;
Symbol *install(), *lookup();

typedef union Datum /* interpreter stack type *l
{double val;
  Symbol * sym;
} Datum;
extern Datum pope);

typedef int (*Inst)(); l* machine instruction *l
#define STOP      (Inst) 0

extern Inst prog [ J ;
extern eval(), add(), sub(), mule), dive), negate(), power(); assign(),
extern $ bltin(), varpush(), constpush(), print());

```

برنامه های مستقلی که دستورات ماشین را اجرا و پشته را اداره می کنند ، در فایل جدیدی به نام code.c نگهداری می شوند . از آنجا که این فایل حدود 150 خط است ، آنرا در قطعات مختلف می آوریم :

show it in pieces.

```

$ cat code.c
#include "hoc.h" #include
"y.tab.h"

#define NSTACK 256 stack[NSTACK];
static Datum static *stackp; /* the stack */
Datum /* next free spot on stack */

#define NPROG 2000 prog
Inst Inst [NPROG] *progp; /* the machine */
Inst *pc; /* next free spot for code generation */ /* program counter
during execution */

ini tcode () { /* initialize for code generation */
    stackp = stack; progp =
    prog;
}

```

The stack is manipulated by calls to `push` and `pop`:

```

pushed) /* push d onto stack */ Datum d;
{
    if (stackp >= &stack[NSTACK])
        execerror("stack overflow", (char *) 0);
}

```

ماشین در طی تجزیه با فراخوانیهایی به کد تابع، تولید می شود و به سادگی یک دستور را به موضع آزاد بعدی در آرایه `prog` وارد می کند. این ماشین مکان دستوری (را که در `hoc4` استفاده نمی شود) برمی گرداند.

```

Inst *code(f) /* install one instruction or operand */
Inst f;
{
    Inst *oprogp = progp;
    if (progp >= &prog[NPROG])
        execerror("program too big", (char *) 0);
    *oprogp++ = f;
    return oprog;
}

```

اجرا کردن ماشین ساده است. به دلیل کوچکی برنامه مستقلی که ماشین را یکبار پس از نصب آن، اجرا می کند اجرا نسبتاً ساده است.

```

execute(p)      1* run the machine *1
Inst *P;
{
    for (pc = p; *pc != STOP; )
        ;
}

```

در هر سیکل شمارنده برنامه (PC) به دستور و دستور به تابع اشاره می کند و آنرا اجرا می کند . بدین ترتیب PC افزایش می یابد تا اینکه آماده دستور بعدی شود . دستوری با STOP opcode را خاتمه می دهد . بعضی از دستورات ، مثل constpush, varpush, pc را افزایش می دهند تا به آرگانهای که دستور را دنبال می کنند وارد شوند .

```

constpush( ) {      1* push constant onto stack *1
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d) ;
}

varpush( ) {        1* push variable onto stack *1
    Datum d;
    d.sym = (Symbol *)(*pc++);
    push( d);
}

```

باقی ماشین ساده است . برای مثال ، عملیات حسابی در اصل مشابهند و با ویرایش یک نمونه تولید می شوند . در اینجا add آورده شده است .

```

add (
{
    /* add top two elems on stack */
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push( d1 );
}

```

بای برنامه ها هم به همین میزان ساده اند .


```

eval() {
    /* evaluate variable on stack */

    Datum d;
    d = pop ( ) ;
    if (d.sym->type == UNDEF)
        execerror("undefined
                    variable". d.sym->name);
    d.val = d.sym->u.val;
    pUshed) ;
}

assign( )
{
    /* assign top value to next value */

    Datum d 1. d2;
    d 1 = pop ( ) ;
    d2 = pop ( ) ;

    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable".
                    d 1. sym->name ) ;

    d1.sym->u.val = d2.val;
    d1.sym->type ::= VAR;
    push( d2) ;
}

print ( ) {
    /* pop top value from stack. print it */

    Datum d; d =
    pop ( ) ;
    printf("\t%.8g\n". d.val);
}

bltin( )
{
    /* evaluate built-in on top of stack */

    Datum d;
    d = pop ( ) ;
}

```

مشکل ترین بخش طراحی bltin است که بیان می کند که *pc باید به صورت اشاره گره به تابعی که یک مضاعف برمی گرداند و تابعی که با d.val به عنوان آرگان اجرا می شود ریخته شود .

تشخیص ما در assign,eval ، در صورتیکه همه چیز به درستی کار کند ، نباید رخ دهد . خطاهای برنامه ای را که سبب می شوند پشته ، بسته شود، توضیح نداده ایم . اگر تغییری بدون دقت کافی ، در برنامه ایجاد کنیم (که امری معمول است) ، بالاسری در زمان و فضا در مقایسه با فایده شناسایی خطا کوچک است . قابلیت C در اداره اشاره گرها به توابع ، سبب ایجاد یک **کاد** -کارآ و فشرده می شود . یک روش جایگزین برای ساخت اپراتورها ، ثوابت و ترکیب توابع معنایی به یک دستور گزینه ای (switch) بزرگ در

حین اجزا ، تفسیر وجود دارد که روش ساده ای است و به عنوان تمرین پیشنهاد می شود .

- سومین انحراف از make

به موازات توسعه کد منبع برای hoc ، ردیابی مکانیکی اینکه چه چیزی تغییر کرده و چه چیزی به آن بستگی دارد ، بیشتر و بیشتر با ارزش می شود . زیبایی make در این است که آن دسته از کارهایی که بدون دستیابی به آن مجبور بودیم دستی انجام دهیم (وگاهی اشتباه کنیم) یا با تولید يك فایل پوسته ای مخصوص انجام دهیم ، به صورت اتوماتیک انجام می دهد .

دو پیشرفت برای فایل make ساخته شده است . اولی بر این مبناست که اگرچه فایل‌های مختلف به ثوابت معرفی شده توسط yacc در y.tab.h بستگی دارد ، اما هیچ نیازی به کامپایل مجدد آنها نیست مگر اینکه ثوابت تغییر کنند : تغییرات کد C در hoc.y هیچ چیز دیگری را تحت تأثیر قرار نمی دهد . در فایل make جدید ، فایل‌های صفر به يك فایل جدید x.tab.h بستگی دارد که تنها زمانی که ثوابت y.tab.h تغییر می کنند ، جدید (update) می شود . دومین پیشرفت ساختن قانون برای pr (چاپ فایل‌های مرجع) است که به فایل‌های مرجع وابسته اند به طوریکه تنها تغییراتی که در فایل‌ها رخ داده چاپ شود . اولین تغییری که ایجاد می شود صرفه جویی در زمان در برنامه های بزرگتر ، زمانی که گرامر استاتیک است اما معناها استاتیک نیستند (موارد معمول) ، است و دومین تغییر صرفه جویی در تعداد صفحات .

در اینجا فایل make جدید برای hoc4 آورده شده است :

```

YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o

hoc4: $(OBJS)
    cc $(OBJS) -lm -O hoc4

hoc.o code.o init.o symbol.o: hoc.h

code.o init.o symbol.o: x.tab.h

x.tab.h: y.tab.h
    -cmp -s x.tab.h y.tab.h:: cp y.tab.h x.tab.h

pr: hoc.y hoc.h code.c init.c math.c symbol.c @pr $?
    @touch pr

clean:
    rm -f $(OBJS) [xy].tab.[ch]

```

'-' قبل از cmp به make می گوید که عمل کند حتی اگر cmp ، انجام نگیرد. این مورد به فرآیند اجازه عمل می دهد ،حتی اگر x.tab.h وجود نداشته باشد. (گزینه -s سبب می شود cmp هیچ خروجی ایجاد نکند بلکه موقعیت خروج را تنظیم کند) علامت \$? از قانون به list مواردی که جدید نیستند وارد می شود . متأسفانه ارتباط تبدیلات نمادین make و تبدیلات پوسته ای بسیار ضعیف است . برای نشان دادن اینکه این مورد چگونه عمل می کند ،فرض کنید همه چیز جدید (uptodata) شده است . بنابراین

\$ touch hoc.y \$ make
 To illustrate how these operate, suppose that everything is up to date.
 Then

```
conflicts: 1 shift/reduce cc -e y.tab. e
rm y.tab.c
mv y.tab.o hoe.o
cmp -s x.tab.h y.tab.h cc hoe.o code.o
init.o
$ make -n pr :: ep y.tab.h x.tab.h math.o symbol.o -lm -0
pr hoe.y hoe4
touch pr
$
```

Print changed files

توجه کنید که هیچ چیز به جزء hoc.y مجدداً کامپایل نشده است . زیرا فایل y.tab.h مشابه قبلی است .

تمرین 8-10) ساینهای پشته و prag را دینامیک کنید به طوریکه امکان دستیابی به حافظه با فراخوانی malloe.o وجود داشته باشد و hoc4 هیچ گاه خارج از فضا اجرا نشود

تمرین 8-11) hoc4 را به گونه ای اصلاح کنید که به جای توابع فراخوانی از یک switch روی نوع عمل استفاده کند . نسخه ها چگونه خطوط ، اكد مرجع و سرعت اجرا را مقایسه می کنند ؟ سادگی ، نگهداری و رشد را چگونه مقایسه می کنند ؟

8-5- مرحله 5: جریان کنترل و اپراتورهای رابطه ای :

این نسخه ، |hoc5 ، نتیجه تلاشی را که در ساختن یک مفسر به خرج داده ایم به کار می گیرد . این نسخه عبارات while,y-else را شبیه آنچه در C هست ، عبارات همگروه با {and} ، و یک عبارت چاپ فراهم می کند . یک سری کامل از اپراتورهای رابطه ای وارد شده اند . (<,<=,>,>=,...) در ضمن اپراتورهای OR,AND به صورت SS ، :: می باشند . (دو مورد آخر برآورد چپ به راست را که در C یک خصیصه است ، ضمانت نمی کنند ؛ آنها هر دو شرط را برآورد می کنند حتی اگر لازم نباشد .)

به گرامر نشانه های غیرپایانه ای و محصولات **while , up , for** ، برکت و اپراتورهای رابطه ای اضافه شده است . این موارد سبب طولانی تر شدن آن می شوند . (ممکن است **wile, up** طولانی نکنند) اما به پیچیدگی بیشتر برنامه نمی انجامند .

```

$ cat hoc.y
"{
#include "hoc.h" #define
code2(c1,c2) #define      code(c1); code(c2) code(c1);
code3(c1,c2,c3) " }      code(c2); code(c3)
"union
{
    Symbol *sym;    I* symbol table pointer *I I* machine
    Inst *inst;    instruction *I
}
"token <sym>  NUMBER PRINT VAR BLTIN UNDEF WHILE IF ELSE stmt asgn
"type <_inst> expr stmtlist cond while if end
"right
"left OR
"left AND
"left GT GE LT LE EQ NE
"left '+' '-'
"left '*' 'I'
"left UNARYMINUS NOT
"right ' '

""
list:      I* nothing *I
list '\n'
list asgn '\n' { code2(pop, STOP); return 1; }
list stmt '\n' { code(STOP); return 1; }
list expr '\n' { code2(print, STOP); return 1; }
list error '\n' { yyerrok; }

asgn:      V AR '=' expr      { $$=$3; code3(varpush, (Inst)$1, assign); }

stmt:
    expr :      { code ( pop); }
PRINT :  expr { code(preexpr); $$ = $2; }
while   cond stmt end {
    ($1)[1] = (Inst)$3;      I* body of loop *I
    ($1)[2] = (Inst)$4;} /* end, if cond fails */
: if cond stmt end { /* else-less if *I
    ($1)[1] = (Inst)$3;      /* thenpart */
    ($1)[3] = (Inst)$4;} /* end, if cond fails */
: if cond stmt end ELSE stmt end { I* if with else */

```

گرامر پنج مورد shift/reduce را مشابه آنچه در hoc3 مطرح شد، دارد .

توجه کنید که اکنون دستورات stop در مکانهای مختلفی برای پایان دادن به تناوب تولید می شوند . مانند قبل ، pragp مکان دستور بعد از تولید WiJI است . WiJI زمانی که این دستورات stop اجرا شوند ، حلقه در حال اجرا را خاتمه خواهد داد . با کمک يك subroutine (زیرروالده) که از مکانهای مختلفی فراخوانی می شود ، محصولی برای end ایجاد می شود . این subroutine يك Stop تولید می کند و مکان دستوراتی که آنرا دنبال می کنند ، برمی گرداند .

کدی که برای **while** ، **up** تولید شده به مطالعه خاصی نیاز دارد . زمانی که به لغت while برخورد می شود ، عملیات whilecode تولید می شود و موقعیت آن در ماشین به

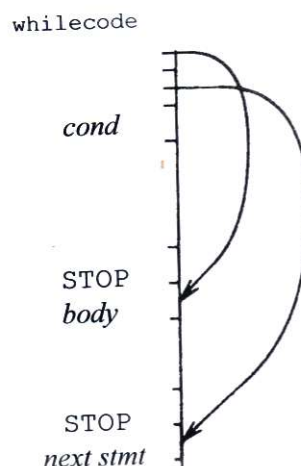
عنوان مقدار محصول

While:WHILE

برگردانده می شود . همزمان ، دو موقعیت بعدی هم در ماشین ذخیره می شوند تا بعداً پر شوند . کد تولیدی بعدی عبارتی است که بخش شرط while را می سازد . مقداری که توسط کد برمی گردد ، شروع که برای شرط است . بعد از اینکه تمام دستور while شناخته نشد ، دو موقعیت اضافی ای که بعد از دستور while ذخیره شده اند با مکانهای بدنه حلقه و دستوری که به دنبال حلقه می آید ، پر می شوند . (کد آن دستور بعداً تولید می شود .)

```
while cond stmt end {
    ($1)[1] = (Inst)$3;          /* body of loop */
    ($1)[2] = (Inst)$4;        /* end, if cond fails */
}
```

\$1 مکانی در ماشین است که while در آن ذخیره می شود . بنابراین (\$1)[1] ، (\$2)[2] دو موقعیت بعدی هستند . ممکن است تصویر این موضوع را بهتر نشان دهد :



موقعیت if هم مشابه while است . با این تفاوت که در if سه موضع else,then و دستوری که پس از if می آید ، حضور دارند . به زودی این مورد را توضیح خواهیم داد . این بار آنالیز واژه ای کمی طولانی تر است ، بخصوص برای اینکه اپراتورهای اضافی را در برمی گیرد .

```

yylex( )
/* hoc5 */

switch ( c ) { case
'>':          return follow('=', GE, GT); return follow('=',
case '<':      LE, LT);
case '=':     return follow( '=', EQ, '='); return follow('=', NE,
case '!':     NOT); return follow(':', OR, '!'); return
case '|':     follow( '&', AND, '&'); lineno++; return '\n';
case ' ', '&.' : return c;
case
default: } '\n' :
}

```

Follow ، يك کاراکتر را جستجو مي کند و اگر آنچه را که یافته چیزی نباشد که به دنبالش بوده، آنرا با ungetc روي ورودی عقب مي راند .

```

follow(expect, if yes , ifno)
{
/* look ahead for >=, etc. */

int c = getchar();

if (c == expect)
return if yes; ungetc(c,
stdin); return ifno;
}

```

تعاریف تابعی بیشتری در hoc.h وجود دارد . - به عنوان مثال ، تمامی رابطه ها - به جز این مورد کاملاً مشابه hoc4 است . در اینجا خطوط پایانی آورده شده است :

```

$ lines:hoc.h

typedef int (*Inst)(); /* machine instruction */
#define STOP (Inst) 0

extern Inst prog[], *progp, *code();
extern eval(), add(), sub(), mule(), div(), negate ( ), power();
extern assign(), bltin(), varpush(), constpush(), print();
extern prexpr ( ) ;
extern
$ gt(), lt(), eq(), gee(), le(), ne(), and(), or(), not();

```

بیشتر کد C نیز مشابه است ، هر چند تعداد زیادی از برنامه های مستقل جدید برای اجرای اپراتورهای رابطه ای وجود دارد . تابع Ie (کوچکتر مساوی) يك مثال نوعی

است:

```

Ie
( )
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double) (d1.val <= d2.val);
    push(d1);
}

```

دو برنامه مستقل که خیلی واضح نیستند، کدهای `if`، `while` می باشند. نکته کلیدی در فهم این موارد، درک جریانهای اجرایی در طی تناوبی از دستورات تا رسیدن به `stop` است، که با رسیدن به آن برمی گردند (از حلقه خارج می شوند) تولید کد در طی فرآیند تجزیه با دقت طراحی شده است. به طوریکه `stop`، تناوبی از دستوراتی را که باید با `تک` فراخوانی بررسی و اجرا شوند، خاتمه دهد. بدنه `یک` `while` و شرط آن و نیز اجزاء `else,then` `یک` دستور `if` یا فراخوانیهای بازگشتی اجرا می شوند تا آنچه را که پس از تکمیل عملیاتشان به سطح برنامه کلید (parent) برمی گردانند، اجرا شود. کنترل این عملیات بازگشتی به وسیله کدی در `if`، `while` انجام می گیرد که مستقیماً با دستورات `if`، `while` در ارتباط است.

```

while code ( ) {
    Datum d;
    Inst *savepc = pc;          /* loop body */
    execute(savepc+2);          /* condition */
    d = pop();
    while (d.val) {
        execute(*«Inst **)(savepc); /* body */
        execute(savepc+2);
        d = pop();
    }
    pc = *«Inst **)(savepc+1); /* next statement */
}

```

مبحث قبلی را یادآوری می کنیم. به دنبال عمل `while` `یک` اشاره گر به بدنه حلقه، `یک` اشاره گر به دستور بعدی، و سپس بخش آغازین شرط می آیند. زمانی که `while` خوانده می شود، `pc` قبلاً افزایش یافته است، بنابراین به اشاره گر بدنه حلقه اشاره می کند. لذا `pc+1` به دستور `pc+2` به شرط اشاره می کند.

کد if بسیار شبیه است. در این مورد pc به بخش then اشاره می کند. ، pc+1 به pc+2، else، به دستور و pc+3 به شرط اشاره می کند.

```
$ cat init.c
static struct          1* Keywords *1
{ char *name;
  int kval;
} keywords[]          {
    "if" ,             IF, ELSE,
    "else" ,           WHILE,
    "while" ,          PRINT,
    "print" ,          0,
    0,
}
;
```

همچنین يك حلقه بیشتر در init برای نصب واژه های کلیدی نیاز داریم :

```
for (i = 0; keywords[i].name; i++)
    install(keywords[i].name, keywords[i].kval, 0.0);
```

تغییرات 0~ برای اداره جدول علامت لازم است؛ کد C که هنگامی که

```
prexpr ( ) {          1* print numeric value *1
    Datum d;
    d = pop ( ) ; printf("%.8g\n",
    d.val);
}
```

این تابع چاپ نیست که اتوماتیک برای چاپ نتیجه نهایی يك برآورد، فراخوانده می شود. بلکه آن تابع، تابعی است که پشته را خالی می کند و يك tab به خروجی می افزاید. hoes تا کنون يك ماشین حساب سرویس دهنده است، هر چند برای برنامه نویسی بزرگ و پیچیده قابلیت های بیشتری لازم است تمرین های بعدی به بعضی از این قابلیت ها اشاره می کنند.

تمرین 8-12) hoes را به گونه ای اصلاح کنید که ماشینی را که در يك فرم قابل خواندن جهت اشکال زدایی تولید می کند، چاپ کند.

تمرین 8-13) اپراتورهای رابطه ای مثل +=* ، ... و اپراتورهای افزایش و کاهش ++ ، - - را اضافه کنید. SS ، : را به گونه ای اصلاح کنید که برآورد چپ به راست و پایان زرد

هنگام را مشابه آنچه در C وجود دارد ، تضمین کنند.

تمرین 8-14) يك عبارت for مشابه آنچه در C هست به hoes بیفزایید . break و Continue را نیز اضافه کنید .

تمرین 8-15) چگونه گرامر یا تحلیل گر واژه اي (یا هردو) hoes را اصلاح مي کنید تا اینکه در مورد مکان خطوط جدید کمتر سخت گيري کند ؟ چگونه سمي کالن را به عنوان معادلي براي خطوط جدید اضافه مي کنید ؟ چگونه يك تبدیل دستوري اضافه مي کنید ؟ چه ترکیبي استفاده مي کنید ؟

تمرین 8-16) يك ابزار انقطاع به hoes بیفزایید به طوریکه محاسبه گریز (runaway) بدون اینکه حالت متغیرهایی که قبلاً محاسبه شده اند را از دست بدهد ، متوقف شود .
تمرین 8-17) لزوم تولید در يك برنامه ، اجرای آن و سپس ویرایش فایل در جهت ایجاد يك تغییر جزئي مشکل ساز است . چگونه hoes را به گونه اي اصلاح مي کنید که يك دستور ویرایشی فراهم کند که شما را در يك ویرایشگر با کپی اي از برنامه hoe تان که قبلاً خوانده شده قرار دهد ؟ راهنمایی : يك opcode متني را در نظر بگیرید .

6-8 مرحله 6: توابع و عملکردها، ورودی و خروجی

مرحله آخر توسعه hoc ، حداقل در این کتاب ، افزایش قابلیت جدید ، توابع و عملکردها ، می باشد . همچنین قابلیت چاپ رشته های کاراکتری (علاوه بر اعداد) و خواندن مقادیر از ورودی استاندارد را نیز افزوده ایم . همچنین hoc6 آرلگانهای نام فایل را که شامل نام " - " برای ورودی استاندارد است قبول می کند . این تغییرات کلاً 235 خط به کد می افزاید و کل برنامه را به 810 خط می رساند و اثری که دارد این است که hoc را از يك ماشین حساب اولیه به يك زبان برنامه نویسی تبدیل می کند . همه خطوط برنامه را در اینجا نشان نمی دهیم . پیوست 3 کل برنامه را در برمی گیرد به طوریکه مشاهده خواهید کرد که قطعات مختلف این برنامه چگونه در تناسب با یکدیگر قرار گرفته اند .
در گرامر فراخوانیهای تابعی عباراتند و فراخوانیهای عملکردی دستور . جزئیات هر دو در پیوست 2 توضیح داده می شود . مثالهای بیشتری نیز در این پیوست آمده است . به عنوان مثلاً ، بیان و استفاده يك عملکرد برای چاپ تمام اعداد فیبوناتچی از آرلگانش به صورت زیر است :

```

$ cat fib proc
fib() {
    a = 0
    b = 1
    while (b < $1)
        { print b
          c = b
          b = a+b a =
          c
        }
    print "\n"
}

$ hoc6 fib
fib(1000)
11235813          21 34 55 89 144 233 377 610 987

```

در ضمن این مورد استفاده از فایلها را نیز نشان می دهد. نام فایل - " " ورودی استاندارد راست . در اینجا یک تابع فاکتوریل آمده است .

```

$ cat fac func
fac() {
    if ($1 <= 0) return 1 else return $1 * fac($1-1)
}

$ hoc6 fac
fac          1
(0)
fac(7)
          5040
fac(10)
          3628800

```

به ارلگانها با یک تابع عملکرد مثل \$1 و ... مراجعه می شود . همانند آنچه در پوسته وجود دارد . در ضمن نسبت دادن به آنها هم مجاز است . توابع و عملکردها بازگشتی هستند اما تنها ارلگانها --- محلی اند . سایر متغیرها کلید (global) می باشند به این معنی که در کل برنامه قابل دسترسند .

Hoe توابع را از عملکردها متمایز -- زیرا با این عمل به ایجاد سطح چک کننده با ارزشی در تولید پشته منجر می شود . فراموش کردن یک بازگشت یا افزودن عبارت اضافی و بی نظمی در پشته به سادگی ایجاد می شود .

با تغییرات گرامری hoes به hoc6 تبدیل می شود . اما این تغییرات محلی اند . اکنون نشانه ها و غیرپایانه ها مورد نیازند و % تعریف واحد عضو جدیدی جهت نگهداری شماره آرگومانها دارد :

```

$ cat hoc.y

%union {
    Symbol *sym;      1* symbol table pointer *1 1* machine
    Inst int*inst;    instruction *1 1* number of arguments
                    narg;      *1
}
%token <sym> NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE READ
%token <sym> FUNCTION PROCEDURE RETURN FUNC PROC ARG
%token <narg> expr stmt asgn prlist stmtlist cond
%type <inst> while if begin end procname
%type <inst> arglist
%type <sym>
%type <narg>

list:      1* nothing *1
list '\n'
list defn '\n'
list asgn '\n' { code2(pop, STOP); return 1; }
list stmt '\n' { code(STOP); return 1; }
list expr '\n' { code2(print, STOP); return 1; }
list error '\n' { yyerrok; }

asgn:      VAR '=' expr { code3(varpush, (Inst)$1, assign); $$=$3; }
: ARG '=' expr
  { defnonly("$"); code2(argassign, (Inst)$1); $$=$3; }

stmt:      expr {code(pop)j }
: RETURN {defnonly("return"); code(procret); }
: RETURN expr
  { defnonly("return"); $$=$2; code(funcrct); }
: PROCEDURE begin '(' arglist ')'
  { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
: PRINT prlist { $$ = $2; }

expr:      NUMBER { : $$ = code2(constpush, (Inst)$1); }
VAR {      $$ = code3(varpush, (Inst)$1, eval); }
: ARG {    defnonly("$"); $$ = code2(arg, (Inst)$1); }
| asgn
| FUNCTION begin '(' arglist ')'
  { $$ = $2; code3(call, (Inst)$1, (Inst)$4); } : READ
 '(' VAR ')' { $$ = code2(varread, (Inst)$3)j }

begin:     1* nothing *1          { $$ = progp; }

```

محصولات برای arglist آرلگانها را می شمارند . در نگاه اول ممکن است به نظر بیاید که لازم است آرلگانها متصل شوند ، اما در واقع لازم نیست ، زیرا هر expr در لیست یک آرلگان ، مقدارش را روی پشته در جایی که دقیقاً خواسته شده قرار می دهد . تنها دانستن اینکه چه مقدار روی پشته وجود دارد مورد نیاز است . قوانین برای defin یک ویژگی جدید yacc را که عبارت از یک عمل ادغامی است ، معرفی می کنند . قرار دادن یک عمل در یک قانون امکانپذیر است ، به طوریکه این عمل در طی شناسایی قانون اجرا شود . آن ویژگی را در ثبت این حقیقت که در یک بیان عملکردی یا تابعی هستیم استفاده می کنیم . در روش دیگر تولید یک شبه علامت جدید برای شروع است که در زمان مناسب شناسایی شود . (در صورتیکه یک مفهوم خارج از بیان یک تابع یا عملکرد رخ دهد ، در حالیکه نایستی حادث می شده ، تابع defonly یک پیغامل خطا چاپ می کند . معمولاً انتخابی برای اینکه آیا خطاها ترکیبی (قاعده ای) یافته شوند یا معنایی وجود دارد . با یکی از این موارد قبلاً در اجزای متغیرهای تعیین نشده مواجه شدیم . تابع defonly مثالی خوب برای مواردی است که **چک معایبی** ساده تر از چک ترکیبی (قاعده ای) است .

```
defnonly(s) /... warn if illegal definition .../
char *s;
{
    if (!indef)
        execerror(s, "used outside definition");
}
```

متغیر indef در hoc.y تعریف و توسط عملیاتی برای defn تنظیم می شود . تحلیل گر واژه ای با تستهایی برای آرلگانها - یک \$ که یک عدد بدنبالش می آید - و برای رشته های نقل شده ، افزایش می یابد . خطوط حاوی Backslash مثل n/ با یک تابع backslash به مفسرها تفسیر می شوند .

```

yylex ()          /* hoc6 */

    if(c == '$') { /* argument? */
        int n = 0;
        while (isdigit(c=getc(fin))
            n ::= 10 * n + C - '0';
        ungetc(c, fin);
        if (n == 0)
            execerror("strange $...", (char *)0); yylval.narg --n;
        return ARG;
    }
    if(c == '"') { /* quoted string */
        char sbuf[100], *p, *emalloc();
        for (p = sbuf; (c==getc(fin) l= ""; p++) {
            if (c ::= '\n' :: c ::= EOF)
                execerror("missing quote", "h");
            if (p >= sbuf + sizeof{sbuf} - 1) {
                *p = '\0';
                execerror("string too long", sbuf);
            }
            *p ::= backslash(c);
        }
        *p = 0;
        yylval.sym ::= (Symbol *)emalloc(strlen(sbuf)+1); strcpy(yylval.sym, sbuf);
        return STRING;
    }

backslash(c)     /* get next char with \s interpreted */
{
    int c;

    char *index(); /* 'strchr()' in some systems */ static char transtab [] ::=
    "b\b f\fn\nr\rt\t";
    if (c l= '\')
        return c;
    c ::= getc(fin);
    if (islower(c) && index(transtab, c))
        return index(transtab, c)[1];
    return c;
}

```

يك تحليل گر واژه اي مثالي از يك ماشين محدود است چه در C نوشته شده باشد و چه با يك مولد برنامه مثل lex. نسخه adhoc ما نسبتاً پيچيده شده است. براي بالاي اين سطح، احتمالاً lex هم در ساير كد منبع و هم جهت سادگي تغيير بهتر است. ساير تغييرات اغلب در كد C ايجاد مي شوند و با تعدادي افزايش در اسامي توابع به hoch برده مي شوند. ماشين مانند قبل است، به جز اينكه به آن، يك پشته ثانويه جهت ردیابی تابع عملکردتودرتو اضافه شده است. (استفاده از پشته ثانويه نسبت به

انباشتن انبوهی از اطلاعات در یک پشته ساده تر است (. در اینجا شروع کد آورده شده است :

```
$ cat code.c #define
NPROG 2000 Inst
prog(NPROG]; Inst      1* the machine */
*progp;                1* next free spot for code generation */ 1*
Inst *pc;              program counter during execution */
Inst *progbase         = prog; 1* start of current subprogram *1
int returning;        1* 1 if return stmt seen */

typedef struct Frame { *sp proc/func call stack frame *1 1* symbol
    Symbol *retpc; table entry */
    Inst *argn; 1* where to resume after return *1 1* n-th
    Datum nargs; argument on stack */
    int      1* number of arguments */
} Frame;
#define NFRAME 100 Frame
frame(NFRAME]; Frame *fp;
                                1* frame pointer */

ini tcode () {
    progp = progbase;
    stackp ...stack; fp =
    frame; returning = 0;
}

$
```

از آنجا که اکنون ، جدول علامت اشاره گرهایی به عملکردها و توابع و رشته هایی برای چاپ را نگه می دارد ، بخش به نوع واحد در hoc.h اضافه می شود :

```
$ cat hoc.h typedef
struct Symbol { 1* symbol table entry */
    char *name;
    short type;
    union {
        double val; /* VAR *1
        double (*ptr) (); /* BLTIN */ /*
        int (*defn) (); FUNCTION, /* PROCEDURE *1
        char *str; STRING */
    } u;
    struct Symbol *next; /* to link to another */
} Symbol;

$
```

در طی کامپایل ، یک تابع به داخل جدول علامت با استفاده از define وارد می شود

define اصلش را در جدول ذخیره می کند و مکان آزاد بعدی را پس از کد تولید شده در صورت موفقیت آمیز بودن کامپایل جدید (update) می کند .

```
define(sp)      1* put func/proc in symbol table *1
{
    Symbol *sp;
    sp->u.defn = (Inst)progbase; 1* start of code *1
    progbase = prog;          1* next code starts here *1
}
```

زمانیکه يك تابع یا عملکرد در طی اجرا فراخوانی می شود تمام آرگانها از قبل محاسبه شده و به داخل پشته وارد شده اند . (اولین آرگان داخلی ترین است .) پس از opcode برای فراخوانی با اشاره گر جدول علامت و تعداد آرگانها می آیند . چارچوبی که پشته می شود تمام اطلاعات جالب راجع به برنامه مستقل را در برمی گیرد. این اطلاعات شامل ورودی آن در جدول علامت، جایی که پس از فراخوانی برمی گردد ، جایی که آرگانها روی پشته عبارت هستند و تعداد آرگانهایی که تابع را خوانده اند ، می باشد . چارچوب با فراخوانی ای ایجاد شده است که نهایتاً برنامه مستقل را اجرا می کند .

```
call() Symbol *sp = (Symbol)pc[0]; 1* symbol table entry *1
{
    1* for function *1
    if (fp++ >= &frame[NFRAME-1])
        execerror(sp->name, "call nested too deeply");
    fp->sp = sp;
    fp->nargs = (int)pc[1];
    fp->retpc = pc + 2;
    fp->argn = stackn - 1; 1* last argument *1
}
```

این ساختار در شکل 8-2 آمده است .

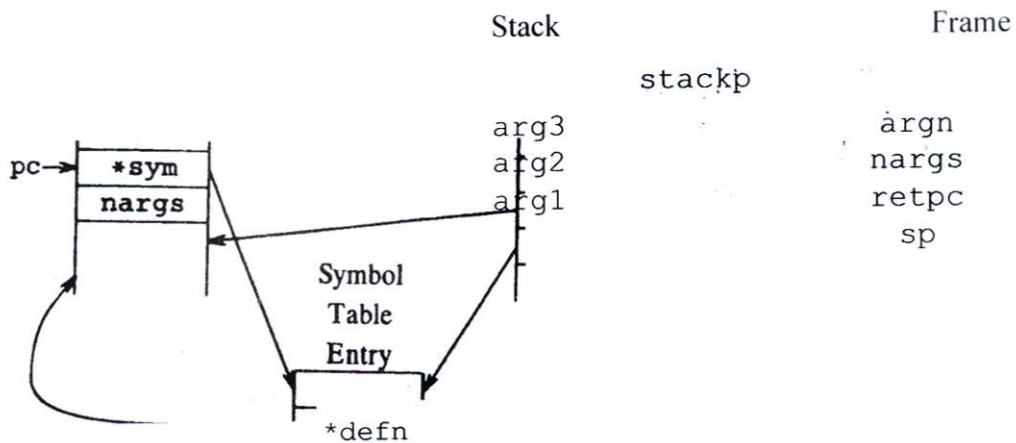
سرانجام برنامه فراخوانی شده با اجزای يك procret یا يك funcrret باز خواهد گشت .


```

a funcret:
funcret ( ) {
    1* return from a function *1

    Datum d;
    if (fp->sp->type == PROCEDURE)
        execerror(fp->sp->name, "(proc) returns value");
    d = pope);          1* preserve function return value *1
    ret ( ) ;
}
pushed) ;

```



شکل 2-8) ساختار داده ها برای فراخوانی عملکرد

```

procret ( ) {
    1* return from a procedure */

    if (fp->sp->type ..... FUNCTION)
        execerror(fp->sp->name,
            "(func) returns no value");
    ret();
}

```

تابع `ret` آرگانها را از پشته خارج می کند، اشاره گر چارچوب `fp` را مجدداً ذخیره می کند و شمارنده برنامه را تنظیم می کند .

```

ret ( )
{
    /* common return from func or proc */
    int i;
    for (i = 0; i < fp->nargs; i++)
        pope); /* pop arguments */ pc =
    (Inst *)fp->retpc;
    --fp;
    returning' " l;
}

```

برنامه های مفسر مختلف ، به کمترین جزئیات نیاز دارند تا بتوانند در زمانی که در آن بازگشت در يك عمل تودرتورخ مي دهد موقعیت را پیش ببرند . این عمل ، خیلی رسمی نیست اما به طور مناسب با نشانه اي به نام بازگشت انجام مي گیرد که زمانی که عبارت بازگشتي صحيح است ، دیده مي شود . اگر بازگشت تنظیم شود ، ، while code ، if code و اجراي زودهنگام خاتمه مي یابند . call آنرا به صفر برمي گرداند .

```

ifcode ( ) {
    Datum d;
    Inst *savepc = pc;          /* then part */

    execute (savepc,..J) ;
    d = pop ( ) ;
    if (d. val)
        execute(*«Inst **)(savepc»);
    else if (*«Inst **)(savepc+1») /* else part? */
        execute(*«Inst **)(savepc+1»);
    if (Ireturning)
        pc = *«Inst **)(savepc+2»); /* next stmt */
}

whilecode(
)
{
    Datum d;
    Inst *savepc = pc;

    execute(savepc+2);
    d = pop ( ) ;
    while (d. val) {
        execute(*«Inst **)(savepc»); if /* body */
            (returning)
                break;
        execute (savepc+2);
        d = pop ( ) ;
    }
    if (I returning)
        pc = *«Inst **)(savepc+1»); /* next stmt */
}

execute(p)
{
    Inst *p;

    for (pc = p; *pc != STOP && I returning; )
        (* (*pc++) ) ();
}

```

Arguments are fetched for use or assignment by getarg, which does the correct arithmetic on the stack:

```

double *getarg ( ) {          /* return pointer to argument */
    int nargs = (int) *pc++;
    if (nargs > fp->nargs)
        exec error (fp->sp->name, "not enough arguments"); return
        &fp->argn[nargs - fp->nargs].val;
}

```

چاپ رشته ها و اعداد با prexpr,prstr انجام می گیرد .

```
arg ( ) /* push argument onto stack */
{
    Datum d;
    d.val = *getarg();
    pushed) ;
}

argassign( ) { /* store top of stack in argument */
```

متغیرها یا تابعی به نام Varread خوانده می شوند . اگر **end ---file** برسد ، این تابع صفر را برمی گرداند . در غیر اینصورت 1 را برمی گرداند و متغیر ویژه را تنظیم می کند .

```
varread ( ) { /* read into variable */
    Datum d;
    extern FILE *fin;
    Symbol *var :: (Symbol *) *pc++;
    Again:
    switch (fscanf(fin, "%If". &'var->u.val»
    { case EOF:
        if (more input ( »
            goto Again; d.val ::
            var->u.val :: 0.0; break;
    case 0:
        execerrar("non-number read into". var->name);
        break;
    default:
        d.val :: 1.0;
        break;
    }
    var->type :: VAR;
    pushed);
```

اگر **end - file** در فایل های ورودی موجود رخ دهد varread,moreinput را می خواند که فایل آرلگان بعدی را در صورت وجود باز می کند . ذکر آنچه **---morein** در مورد فرآیند ورودی نشان می دهد در اینجا مناسب بنظر نمی رسد . جزئیات تکامل در پیوست 3 آمده است . این بخش ما را به انتهای طراحی hoc می رساند برای مقایسه در اینجا تعدادی خط غیرخالی در هر نسخه آورده شده است :

hoc1	59	
hoc2	94	
hoc3	248	(lex version 229)
hoc4	396	
hoc5	574	
hoc6	809	

تعداد با برنامه ها محاسبه شده اند :

زبان به هیچ وجه خاتمه نمی یابد ، حداقل همیشه فکر کردن به توسعه های مفید ساده است ، اما ما در اینجا بیشتر از این ادامه نمی دهیم . تمرینهای بعدی ، بر بعضی مواردی که با ارزش به نظر می رسند تاکید می کنند .

تمرین 8-18) hoc6 را به گونه ای اصلاح کنید که پارامترهای رسمی نام گرفتند در subroutine ها را به عنوان یک جایگزین \$ تبدیل کند .

تمرین 8-19) همانطور که بیان شد ، همه متغیرها به جز پارامترها کلی اند . بیشتر مکانیسم لازم برای افزودن متغیرهای محلی روی پشته از قبل وجود دارد . یک رویکرد ، داشتن تعریف اتوماتیکی است که فضایی روی پشته برای متغیرهای لیست شده می سازد . متغیرهایی که اینگونه نامگذاری شوند کلی فرض می شوند . در ضمن جدول wiu باید توسعه یابد بگونه ای که یک جستجو در ابتدا برای متغیرهای محلی و سپس برای متغیرهای کلی صورت گیرد . این شیوه چگونه با آرگانها برخورد می کند ؟

تمرین 8-20) چگونه به hoc آرایه می افزائید ؟ این آرایه ها چگونه به توابع و عملکردها فرستاده می شوند ؟ آنها چگونه باز می گردند ؟

تمرین 8-21) بررسی و اجرای رشته را توسعه دهید ، به گونه ای که متغیرها بتوانند رشته ها را به جای اعداد نگه دارند . چه نوع اپراتورهایی مورد نیازند؟ بخش شکل این عمل مدیریت ذخیره سازی است : اطمینان از اینکه رشته ها به گونه ای ذخیره می شوند که زمانی که مورد نیاز نیستند آزاد شوند ، به گونه ای که ذخیره مورد نیاز خارج نشود . به عنوان یک مرحله میانی ، قابلیت‌های بهتری برای فرمت خروجی اضافه کنید ، نظیر دستیابی به برخی فرمهای عبارت printf در C .

8-7) بر آورد عملکرد

Hoc را با برخی از دیگر برنامه های ماشین حساب UNIX مقایسه کردیم تا یک ایده اولیه راجع به اینکه WEB چگونه عمل می کند بدست آوریم . جدول زیر باید به عنوان یک

تخمین گرفته شود، اما می تواند نشان دهد که روش ساخت ما، یک روش منطقی است بقای زمانها بر حسب زمان کاربرد در روی یک PDP-11170 محاسبه می شوند. دو عمل وجود داشت اولی محاسبه تابع $ack(3,3)$ آکرمن است. این عمل تست خوبی از تابع CaB می باشد. این تست به 2432 خط نیاز دارد که برخی بسیار تودرتو هستند. تست دوم، محاسبه اعداد فیبوناتچی با مقادیر کمتر از 1000 به تعداد صد دفعه است. این عمل بیشتر عملیات حسابی را با یک فراخوان تابعی گاهی دربرمی گیرد.

```

    == 0) return $2+1
    computing Ackermann's function ack(3,3) «This is a good test of the
    ack($1-1, ack($1, $2-1)»
}
function-caB mechanism; it requires 2432 calls, some nested quite deeply.
ack(3,3)
func ack() {
    if ($1 == 0) return $2+1
    if ($1 == 1) return $2
    return ack($1-1, ack($1, $2-1)
}
The second test is computing the Fibonacci numbers with values less than 1000
($2)
a total of one hundred times; this involves mostly arithmetic with an occasional
return
proc fib () {
function call
    a ... 0
    b ... 1
    while (b < $1) { c
        = b b =
        a+b a = c
    }
}
i ... 1
while (i < 100)
{ fib(1000)
  i ... i + 1
}

```

چهار زبان عبارت بودند از hac , bc(1) , bas , (که یک لهجه BASIC قدیمی است که تنها روی pop-II اجرا می کند) و C (که برای همه متغیرها از مضاعف استفاده می کند) اعداد جدول 1-8 مجموع زمان CPU سیستم و کاربر می باشند .

در ضمن فراهم آوردن امکان زمان سنجی برای یک برنامه C جهت تعیین اینکه هر تابع چه مقدار از آن زمان را استفاده می کند ، مقدور است . برنامه باید مجدداً با بروز نیمرخ ، یا افزودن گزینه -p به هر کامپایل و فراخوانی C کامپایل شود . اگر makefile اصلاح کنیم تا عبارت زیر را بخوانیم :

```
make clean; make CFLAGS=-p $
```

به طوریکه دستور C متغیر CFLAG5 را استفاده کند و سپس بگوییم .

```
hoc6 $(OBJS)
```

```
CC $(CFLAGS) $(OBJS)-1 m-0 hoc6
```

برنامه حاصله کد نیمرخ را در برخواهد گرفت. زمانی که برنامه اجرا می شود ، فایل به نام main خارج از داده ها را فراهم می کند که توسط برنامه prof تفسیر می شود .

برای نمایش این نمادها، به اختصار تستی روی hoc6 با برنامه فیوناتچی بالا انجام دادیم:

```
$ hoc6 <fibtest                                Run the test
$ profhoc6 . sed 15q                            Analyze
```

name	%time	cumsecs	#call	ms/call
_pop	15.6	0.85	32182	0.03 0.02
_push	14.3	1.63	32182	
mcount	11.3	2.25		
csv	10.1	2.80		
cret	8.8	3.28		
_assign	8.2	3.73	5050	0.09
eval	8.2	4.18	8218	0.05
_execute	6.0	4.51	3567	0.09
_varpush	5.9	4.83	13268	0.02
It	2.7	4.98	1783	0.08
_constpu	2.0	5.09	497	0.22
_add	1.7	5.18	1683	0.05
_getarg	1.5	5.26	1683	0.05
_yyparse	0.6	5.30	3	11.11

اندازه گیریهایی که از نیمرخ بدست می آید ، تنها در حد نوسانات زمانی است . بنابراین باید با آنها به عنوان نشانگر استفاده شود نه داده های حقیقی ، اعداد در اینجا پیشنهاد می کنند که چگونه hoc را در صورت نیاز سریعتر بسازیم ، حدود یک سوم زمان اجرا ، صرف وارد کردن و خارج کردن از پشت می شود . در صورتیکه زمانهای لازم برای توابع اتصال Subroutine C مثل CSU , Cret را هم در نظر بگیریم ، زمان سرآمد بیشتر خواهد بود . و me aunt قطعه ای از کد نیمرخ است که با CC-P کامپایل می شود . (جایگزین کردن فراخوانهای تابع با ماکروها باید تفاوت چشمگیری ایجاد کند . جهت بررسی این موضوع کد C را اصلاح کردیم . به این صورت که فراخوانها با ورود و خروج (popm , push) را با ماکروها برای اداره پشت می جایگزین کردیم .

```
#define push(d) *stackp++ = (d)      1* function still needed *1
#define popm() *--stackn
```

(تابع pop به عنوان یک opcode در ماشین مورد نیاز stm است . بنابراین نمی توانیم تمامی popها را جایگزین کنیم .) نسخه جدید حدود 35% سریعتر اجرا می شود . زمانها در جدول 1-8 از 5/5 به 3/7 ثانیه و از 5 به 3 ثانیه کاهش می یابند .
 تمرین-8-22) ماکروهای push , pop , خطاها را چک نمی کنند . دستوری جهت این عمل تولید کنید چگونه می توانید چک کردن خطا با نسخه های تابعی را با سرعت ماکروها تلفیق کنید ؟

8-8- نگاهی به عقب

دروس مهمی در این فصل وجود دارند . اول اینکه ابزارهای طراحی زبان امکانات مهمی هستند . این ابزارها ، امکان تمرکز روی بخش جذاب کار – طراحی زبان – را فراهم می کنند . زیرا تجربه این کار ساده است . همچنین استفاده از یک گرامر ، ساختار سازمان دهنده ای را برای پیاده سازی فراهم می کند . این ساختار برنامه های مستقلی است که به وسیله گرامر به هم مرتبط می شوند و به موازات پیشروی تجزیه در زمانهای مناسب فراخوانی می شوند .

نکته دوم که منطقی تر هم است ، ارزش فکر کردن به کاری که در دست داریم بیشتر به عنوان طراحی زبان است تا به عنوان نوشتن برنامه . سازماندهی یک برنامه به عنوان یک زبان منظم ، ترکیبی - (که رابط کاربر است) را ایجاد می کند و ساخت مرا ساخت را ساختار می بخشد . همچنین به حصول اطمینان از اینکه ویژگی جدید تا حدی با ویژگیهای قبلی انطباق نشان می دهد ، کمک می کند . زبانها مطمئناً به زبانهای برنامه - نویسی - مرسوم - محدود - نمی - شوند - نمونه - هایی - از - تجربه - خودمان lex,yacc,pic,eqn و make می باشند .

همچنین آموزشهایی جهت نحوه استفاده از ابزارهای موجود وجود دارد . به عنوان مثال make, خیلی کارآ نیست . این برنامه خطاهایی را که به سبب فراموشی در کامپایل مجدد بعضی برنامه های مستقل به وجود می آید ، حذف می کند . این مقوله (کامپایل مجدد) اطمینان می دهد که هیچ کار اضافی انجام نشده است و روش مناسبی جهت گروهبندی اپراتورهای مرتبط و شاید وابسته در یک تک فایل فراهم می آورد .

فایل‌های سرآمد روش خوبی برای اداره تعریف داده‌هایی که باید در بیش از یک فایل حضور یابند، هستند. با مرکزیت بخشیدن به اطلاعات، آنها خطاهایی را که با سخنهای ناپایا ایجاد می‌شوند، بخصوص زمانی که با `make` کوپل می‌شوند، حذف می‌کنند. همچنین سازماندهی داده‌ها و برنامه‌های مستقل به داخل فایل‌ها مهم است به گونه‌ای که زمانی که وجود آنها ضروری نیست قابل دیدن نباشند.

دو موضوع وجود دارد که به دلیل کمبود فضا بر آنها تمرکز نکردیم. یکی میزان استفاده از سایر ابزارهای UNIX در طی طراحی است که در خانواده `hoc` انجام دادیم. هر نسخه برنامه، در یک دایرکتوری جداگانه به همراه فایل‌های یکه‌ای که به یکدیگر متصل شده‌اند می‌باشد. `Dr, Is` به کرات جهت ردیابی استفاده شدند. بسیاری از پرسش‌های دیگر با برنامه‌ها پاسخ داده می‌شوند. به عنوان مثال اینکه یک متغیر خاص کجا معرفی می‌شود. از `grep` استفاده کنید. در این نسخه چه تغییری ایجاد کردیم؟ `diff` را به کار ببرید. چگونه تغییرات را در یک نسخه جمع کردیم؟ از `idiff` استفاده کنید. اندازه فایل چقدر است؟ از `we` استفاده کنید. آیا زمان برای ساخت یک کپی مناسب است؟ از `ep` استفاده کنید. چگونه می‌توان تنها فایل‌هایی را که نسبت به کپی قبلی تغییر کرده‌اند، کپی کرد؟ `make` را به کار ببرید. این روش کلی، روش نوعی توسعه روزافزون برنامه روی سیستم UNIX است. میزبانی برای ابزارهای کوچک که به صورت مجزا و یا در صورت نیاز تلفیقی استفاده می‌شوند، به مکانیزه کردن کاری کمک می‌کند که در غیاب این میزبان مجبور بودیم با دست انجام دهیم.

تاریخچه و معرفی کتاب

`Yacc` توسط Steve Johuson طراحی شد. از نظر فنی، گروهی از زبان‌هایی که `yacc` برای آنها می‌تواند تجزیه‌گر ایجاد کند (LALR(1 نامیده می‌شوند): که یک تجزیه‌چپ به راست بوده و به جستجوی حداکثر یک نشانه در ورودی می‌پردازد. مفاد یک بیان مجزا جهت رفع اولویت و ابهام در گرامر برای `yacc` جدید است. به "تجزیه‌تعیینی گرامرهای مبهم" نوشته [A.V.Aho,CACM,J.D.U//man,S.C,Johnson](#)، آگوست 1975 مراجعه کنید. همچنین برای تولید و ذخیره‌سازی جداول تجزیه‌تعدادی الگوریتم و ساختارهای داده‌ای جدید وجود دارد.

بیان خوبی از تئوری پایه‌ای قابل توسعه `yacc` و سایر مولدان تجزیه‌ممکن است. در اصول طراحی کامپایل نوشته ([Addsiam-wesly\) J.D.Ullman,A.V.Aho 1997](#)) یافت شود. خود `yacc` در جلد 2 راهنمای برنامه‌نویس UNIX توضیح داده شده است. این جلد

ماشین حسابی قابل مقایسه با *hoc2* را نیز ارائه می دهد . ممکن است این مقایسه آموزنده به نظر بیاید .

Lex در اصل توسط mike lesk نوشته شد . تئوری lex نیز توسط Aho , Ullman توضیح داده شده و خود زبان lex در راهنمای برنامه نویسی UNIX آمده است . yacc و تا حد کمتری lex جهت ساخت بسیاری از پردازشگرهای زبان مثل کامپایلر C قابل انتقال ، پاسگال ، فورترن VV,Ratfor,awk,be,eqn,pic استفاده شده اند .

Make توسط stu Feldman نوشته شد به MAKE : برنامه ای برای حفظ برنامه های کامپیوتری ، نرم افزار - تمرین و تجزیه ، آوریل 1979 مراجعه کنید . برنامه های کارآیی نوشتاری نوشته (John Bentley,1982) (prentice-Hall) تکنیکهایی را برای سرعت بخشیدن به برنامه ها توضیح می دهند . اولین تاکید بر یافتن الگوریتم راست و سپس یافتن مجدد که در صورت لزوم است .

فصل دهم آماده سازی مستندات

یکی از کاربردهای سیستم UNIX ویراستن و فرمت کردن فایل هاست. در واقع شرکت Bellabs که با ضمانت سیستم آماده سازی فایل سخت افزار PDP-11 را خریداری کند خوشبختانه آنها بیش از آنچه انتظار می رفت سود کردند.

اولین برنامه فرمت کننده troff نام داشت که بسیار کوچک بود و سریع و آسان نیز مورد استفاده رار می گرفت. فرمت کننده بعدی که nroff نام داشت توسط joeossanna پایه ریزی شد. که ازان پس بسیاری از برنامه های فرمت کننده زبان برنامه پذیر nroff مورد استفاده قرار گرفت، به جای فراهم سازی هر سبک فایلی که توسط کاربر درخواست می شد.

وقتی در سال 1973 یک حرفچین کوچک حق نشر گرفت، nroff برای استفاده از در سایزها و فونت ها و کاراکترهای مختلفی، که این حرفچین محیا می کرد، گسترش یافت. برنامه جدید troff نام گرفت. nroff و troff اساساً برنامه های مشابهی هستند و زبان ورودی مشابهی را می پذیرند. ما عمدتاً در مورد troff بحث می کنیم. اما بسیاری از نظریه های در مورد nroff نیز صدق می کند که این به علت محدودیت های دستگاه های خروجی است. یکی از مزیت های troff انعطاف پذیری زبان اصلی و برنامه پذیر بودن آن است. troff را می توان وادار کرد که بسیاری از برنامه های فرمت کننده را اجرا کند. اما انعطاف پذیری آن افزایش قیمتش را موجب می شود. استفاده از troff کمی مشکل است، و البته سخت افزار آماده سازی فایل UNIX برای پوشاندن بسیاری از قسمت های آشکار و عریان troff طراحی ریزی شده است.

برای مثال یک صفحه بندی - سبک عمومی فایل و اینکه عنوان بندی و پاراگراف بندی چگونه است. شماره صفحه کجای صفحه قرار گیرد بزرگی صفحه چقدر باشد و غیره از پیش ساخته نیست. بلکه باید برنامه ریزی شود به جی اینکه کاربر را وادار کنیم تا این جزئیات را در فایل مشخص کند می توان یک بسته فرمان فرمت کننده استاندارد را برای این کار فراهم ساخت. دیگر، اربسته بیان نمی کند که خط بعدی در وسط قرار گیرد با حروف بزرگ و فونت درشت بلکه می گوید خط بعدی یک عنوان است. پس تعریف بسته بندی شده سبک عنوان مورد استفاده قرار می گیرد. از این پس کاربر مولفه های منطقی و اساسی یک فایل را بیان می کند - نظیر عنوان پارگراف پانویس و غیره... به جای سایز (اندازه) فونت و موقعیت.

متاسفانه، چیزی که به عنوان یک بسته استاندارد فرمان فرمت کننده پیش می رود برای مدت زمان زیادی استاندارد باقی نمی ماند. بسته های زیادی با کاربردهای وسیعی وجود دارد. ما در اینجا در مورد اهداف بسته های کلی صحبت می کنیم. اولی ms که استاندارد اصلی می باشد و دومی - mm که یک طرح جدیدیست که در سیستم ۷ استاندارد است. همچنین در مورد بسته man برای صفحه های راهنما و چاپ کننده توضیح خواهیم داد.

ما بر روی ms بیشتر تاکید می کنیم. زیرا در man Edition ۳ استاندارد است و نمونه است از تمامی اینگونه بسته ها و برای انجام کاری به اندازه کافی قدرتمند است و ما برای تایپ این کتاب از آن استفاده کرده ایم ولی می بایست آن را کمی بسط می دادیم. I. برای مثال با اضافه کردن یک فرمان برای به کار بستن واژه های in this font در متن.

این نمونه ای از یک تجربه است بسته های ماکروی بزرگ بسیاری از فرمان های فرمت شده کفایت البته گاهی اوقات بازگشت به فرمان های اساسی troff ضروریست و در اینجا قسمت های کوچکی از troff را توضیح خواهیم داد.

گرچه troff به کلی توانایی کنترل فرمت خروجی را فراهم می سازد. اما استفاده از اطلاعات پیچیده ای شبیه ریاضیات جدول های و ارقام بسی مشکل است. هر یک از این موارد به اندازه صفحه بندی سخت و دشوار است. حل چنین مشکلی اشکال متفاوتی را طلب می کند. به جای بسته های فرمان های فرمت کننده زبان هایی با اهداف مشکلی برای ریاضیات جدول ها و ارقام وجود دارد که توضیح اینکه چه

چیزی خواسته شده است را آسان می کند. هر کدام از اینها بایک برنامه مجزا استفاده می شود که زبانش را به فرمان `troff` بر می گرداند ترجمه می کند. برنامه ها و توسط مسیرهای اطلاعاتی (`pipes`) منتقل می شوند.

این پیش پردازنده ها نمونه خوبی از رهیافت `UNIX` به عرصه کار است. نسبت به ساخت `troff` حتی بزرگتر و پیچیده تر از چیزی هست. برنامه های مجزا نیز با آن همکاری می کنند. (البته ابزارهای توسعه زبان که در فصل 8 توضیح داده شده برای کمک به تحقیق سازی مورد استفاده قرار گرفته است).

ما دو برنامه را برایتان شرح خواهیم داد. اول `tbl` که جدولها را فرمت می کند و دوم `eqn` که عبارات ریاضی را فرمت می کند.

ما سعی می کنیم در مورد آماده سازی فایلها و تامین ابزارها توصیه هایی داشته باشیم نمونه هایی که در سراسر این بخش بیان شد فایلی ست که زبان `noc` و صفحات راهنمای `noc` را توضیح خواهد داد. این فایل در صفحه 2 چاپ شده است.

9-1 بسته های ماکروی `ms`:

یک فایل توسط مآژه های قسمت های اصلی در بسته های ماکرو توضیح داده خواهد شد. (عنوان) سر بخشها، پاراگرافها) و نه با جزئیاتی فاصله گذاری، فونت و سایز در یک برنامه. این روش شما را از انجام کاری سخت نجات می بخشد و از فایل شما در برابر جزئیات نامربوطه محافظت می کند. در حقیقت با استفاده از تعاریف مختلف دستگاه ماکرو - با اسامی منطقی مشابه - شما می توانید فایل را با اندکی تفاوت جلوه گر سازید. برای مثال یک فایل ممکن است مراحل یک گزارش تکنیکی کنفرانس نشریه و یا کتاب را به وسیله فرمان های فرمت کننده بگذرانند، که این عملیات با چهار بسته ماکروی مختلف انجام می شود.

ورود به `troff` آیا یک بسته ماکرو را شامل می شود یا نه، یک متن معمولی ست که با فرمان های فرمت شده پدید می آید. دو نوع فرمان وجود دارد. اولی یک دوره ای را در ابتدای خط شامل می شود، که با یک یا دو حرف و یا رقم همراه است و شاید هم یک پارامتر همانگونه که در زیر نشان داده شده است.

`pp`

`ft B this is a little font paragraph`

`troff` که در فرمان به صورت پیش ساخته وجود دارد، با حروف کوچک نامگذاری می شود. در صورتیکه فرمانها در بسته های ماکرو با حروف بزرگ نام برده می شوند. برای مثال `pp` یک فرمان `ms` برای پاراگراف و `ft B` یک فرمان `troff` است که در فونت درشت تغییر ایجاد می کند (فونتها با حروف بزرگ نامگذاری می شوند و ممکن است در حروف چین های مختلف، متفاوت باشند)

دومین فرم فرمان `troff` توالی کارکترهاست، که با یک اسلش شروع می شود و ممکن است در هر جایی از ورودی ظاهر شود. برای مثال `FB` تغییری در فونت درشت به وجود می آورد به این شکل از فرمان یک `troff` کامل می گویند.

شما می توانید با اندکی توجه فرمان `pp` را قبل از هر پاراگراف فرمت کنید و برای بسیاری از فایلها می توانید فرمانهای `ms` متفاوت 12 تایی را به خوبی به انجام برسانید. برای مثال ضمیمه 2 را `noc` را توضیح داده دارای یک عنوان اسامی نویسندگان، چکیده نامه، عنوان هایی که شماره بندی شده و پاگراف می باشد و تنها دارای 14 فرمان مجزا است که تعدادی از آنها به صورت جفت آمده است. متن این فرم های معمولی را از `ms` برداشت می کنیم.

`TL`

`title of document (one or mor lines)`

`au`

Author names one per line

AB

Abstract, terminated by AE

NH

Numbered heading cauto matic numbering)

pp

paragraph

pp

another PARAGRAPH

Sh

sub -heading (not numbered)

pp

فرمان‌های فرمت کننده باید در ابتدا خط رخ دهد. ورودی بین فرمان‌ها خالیست. محل قرارگیری سطرهای جدید در ورودی مهم نیست زیرا troff کلمه‌های را از سطری به سطر دیگر انتقال می‌دهد. تا سطرها را به میزان کافی طولانی سازد (فرایندی که پر کردن (filling) نام دارد. و فاصله مناسب و یکنواختی بین واژه‌ها ترتیب می‌دهد تا حاشیه‌ها را همتراز می‌کند. این تمرین خوبیست گرچه شروع کردن هر جمله در یک سطر جدید ویرایش بعدی را آسانتر ممکن می‌سازد. (فرمان صفحه 292)

فرمان آ نشانه‌اش را به صورت یک شناسه در troff نشان داده می‌شود.

troff - ms - hoc .ms

کاراکترها بعد از m بسته ماکرو را تعیین می‌کنند. وقتی که توسط ms فرمت شده متن noc به شکل زیر می‌باشد.

HOC یک زبان محاوره‌ای برای شناور شدن در کانون علم حساب.

Brian kernighan

Rob pike

چکیده: HOC یک تقسیم کننده برنامه پذیر ساده است برای فرو رفتن در عمق عبارات که دارای روند کنترل سبک -C تعریف کارکردها عملکردهای پیش ساخته عددی سینوس و الگاریتم است.

عبارات: HOC زبان عبارت است شباهت زیادی به C دارد، گرچه تعداد زیادی عبارت روند کنترل در آن وجود دارد بسیاری از عبارات مانند نسبت دهی‌ها مورد بی توجهی قرار گرفته است.

نمایشگر:

گرچه اینکه troff رفتن را پر کند و یا همستون سازد بسیار مطلوب است اما بعضی اوقات نیز ناخوشایند به نظر می‌رسد. برای مثال برنامه‌ها نباید حاشیه خود را همستون سازند. یک چنین موارد فرمت نشده را متن نمایشگر می‌خوانند. فرمان‌های DE (display) , DC (display start) ms , End) از اینکه متن مطابق با ظاهرش چاپ شود جلوگیری می‌کنند و آنم را به صورت مرتب حاشیه بندی می‌کند. در اینجا قسمت بعدی یک راهنمای noc نشان داده شده که شامل یک نمایشگر کوچک است.

(فرمان صفحه 293)

noc یک زبان گویاست مانند c گرچه تعداد زیادی عبارات روند کنترل وجود دارد. بسیاری از عبارات مانند نسبت دهی مورد بی توجهی قرار می گیرند برای مثال اپراتور نسبت دهی =میزان عملوند راست را به عملوند چپ نسبت می دهد. و میزانش را به دست می آورد. نسبت دهی های چندگانه بدین ترتیب کار می کنند. گرامر عبارت این چنین است:

```
en pr
number
variable
enpr
enpr binop enpr
unop enpr
function arguments
```

داخل متن یک نمایشگر به صورت نرمال پر شده و یا همستون شده نیست. علاوه بر این اگر در متن موجود جای خالی کافی وجود ندارد مطالب نمایش داده شده به صفحه بعدی انتقال می یابد. DS انتخابها بسیاری را شامل می شود. مثلاً برای همستونی سمت چپ، C که هر سطر را به طور انحصاری در مرکز قرار می دهد، و B که کل نمایشگر را در مرکز قرار می دهد. آیتمها در نمایشگر بالا با تبها جدا می شوند. تبهای troff در نیم اینچی یکدیگر قرار می گیرند. و نه در فاصله هشت تایی که رایج بوده حتی اگر تب متوقف شود هر فاصله هشت تایی وجود دارد گرچه کارکترها عرضهای متفاوتی دارند. تبهای پردازش شده توسط troff آنگونه که انتظار می رود ظاهر نمی شود.

تغییرات فونت

ماکروهایی ms فرمان برای تغییر فونت فراهم می کنند. R فونتها را به لاتین تغییر می دهد که یک فونت معمولی و رایج است. آن را به ایتالیک تغییر می دهد و B آن را سیاه می کند هر فرمان فونت را برای متن بعدی گزینش می کند.

this text is roman, but

I

this text is italic

R

This is roman again, and

B

this is bold face

I و B یک نشانه انتخاب کرده اند که در این حالت تغییر فونت فقط در مورد نشانهها اعمال می شود. در troff نشانهها شامل فضاهای خالیست که نقل قول شده اند گرچه کارکترهای نقل قول کننده نقل قول مضاعف هستند.

در نهایت دومین نشانه برای I و B که به صورت لاتین چاپ می شود بدون فضای خالی به نشانه اول ضمیمه می شود. این صورت به میزان و افری برای نقطه گذاری در فونت سمت راست استفاده می شود. این جمله را (parenthetical. I italic words) که به عنوان parenthetical italic words نادرست چاپ شده با این جمله (parenthetical i italic word) که به عنوان parenthetical italic words درست چاپ شده مقایسه کنید.

تفکیک فونت‌ها توسط troff تشخیص داده می‌شود ولی نتیجه مطلوبی ندارد. بر روی کارکترهای ایتالیک تاکید زیادی شده و کارکترهایی با فونت درشت وجود ندارد اگر چه بعضی شکل‌های _nroff حروف درشت را توسط روی هم چاپ کردن کارکترها شبیه سازی می‌کند.

فرمان‌های متفرقه:

پا نوشت توسط FS معرفی شده و با FE پایان می‌یابد. شما مسئول هر علامت تشخیص هویتی شبیه یک نماد ستاره و یا نماد خنجر در پا نوشت اینگونه ساخته می‌شود.

identifying nerl like an asterisk or a dagger.

fs

dg like this one.

FE

this foot note was creat ea with

پاراگراف‌های فاصله دار با یک شماره و یا علامت دیگری در حاشیه با فرمان IP ساخته می‌شود. برای ساختن آن اولین پاراگراف کوچک. 2 - دومین پاراگراف که ما آن را طولانی تر ساختیم برای اینکه نشان دهیم که بر روی سطر دوم به مانند سطر اول فاصله دار می‌شود.

IP

first little paragtaph

IP

SECOND PARAGRAPH...

IP,(LEFT-INSTIFIED PARYRAPH,.PP خاتمه می‌دهند. نشانه

IP می‌تواند هر رشته - متنی (String) باشد. در صورت نیاز از گیومه برای حفاظت از فضاهای خالی استفاده کنید. نشانه دومی می‌تواند برای تعیین کردن بسیاری از فواصل استفاده می‌شود.

فرمان‌های KS و KE با یکدیگر نگه داشته می‌شوند. متن که بین این دو فرمان محصور شده اگر در صفحه موجود جانگیرد به صفحه اول جا دهیم می‌توانیم از -KF به جای KS استفاده کرده تا متن مورد نظر در بالای صفحه بعدی قرار گیرد. ما از -KF برای تمامی جدول‌های این کتاب استفاده کرده‌ایم.

شما می‌توانید مقادیر پیش ساخته ms را تغییر دهید با قرار دادن ثبات اعداد که متغیرهای troff هستند و مورد استفاده ms بوده است. و شاید اینکه ثبات‌ها اندازه‌های متن و فواصل بین سطرها را کنترل کند رایج تر باشد. اندازه متن نرمال 10 پونت است که هر پونت تقریباً 1/72 اینچ است. سطرها به طور نرمال در مکان‌های -12 پونتی قرار می‌گیرند برای تغییر آن برای مثال به -9 و 11 ثبات اعداد را در ps,vs قرار بده.

ثبات اعداد ما شامل LL برای طول سطرست، PI برای فاصله پاراگراف و PD برای تفکیک بین سطرها، که تمامی این عملیات در PP و یا LP بعدی تاثیر گذار است.

جدول 9-1: فرمان‌های فرمت کننده ms

--

بسته‌های ماکروی MM:

ما در اینجا به جزئیات بسته‌های ماکروی mm نخواهیم پرداخت که البته در برخی جزئیات شبیه ms می باشد و همانند ms بر روی پارامترها کنترل دارد البته با توانایی بیشتر و پیتام خطانمای بهتر. جدول 9-2 فرمان‌های mm را در مقابل با فرمان‌های ms در جدول 9-1 نشان می دهد.

سطح troff

در یک عمر حقیقی هر چیزی باید به مهارت‌های ms و mm و یا وابسته‌های دیگر واقف باشد. وبه قابلیت‌های تروف (troff) خالی دست یابد، که اجرا شدنش شبیه به برنامه ریزی در زبان همگذاری اسمبلی است گرچه به صورت هشدار دهنده اجرا می شود. سه موقعیت روی می دهد. دستیابی به کارکترهای ویژه سایز خطی و تغییرات فونت و عملکردهای فرمت کننده اصلی.

نام‌های کارکترها:

کارکترهای ناآشنا - حروف یونانی مثل گرافیکی و سطرها و مکان‌های متفاوت و متنوع آسان هستند، اما خیلی هم روشن مند و منظم نیستند. یک چنین کارکترهایی دارای نامی هستند مثل C در جایی که C یک کارکتر تنهاست و یا cd موقعی که cd یک کارکتر جفتی است. troff یک علامت منهای _ ascll را به عنوان خط تیرم (hyphan) به جای اینکه یکگ منها چاپ می کند. یک علامت منهای درست اینگونه تایپ می شود. و علامت دش باید اینگونه چاپ شود. به جای کارکتر آمده است. جدول 9-3 برخی از کارکتر ویژه رایج و معمولی را لیست کرده است، که در راهنمای troff بیش از این لیست کارکتر وجود دارد که این لیست با سیستم شما ممکن است متفاوت باشد. مواقعی برای تغییر رایج ترین کارکتر خود کار ماشینی است. توالی و ترتیب برای چاپ یک اسلش تضمین شده است و در خروجی برای یک اسلش استفاده می شود. به عبارت دیگر فضایی با عرض صفر است. بیشترین کاربرد آن جلوگیری troff از تفسیر تناوب‌ها در آغاز سطر است.

ما در این سطر از زیاد استفاده کرده ایم. برای مثال کل ms را در آغاز این فصل اینگونه تایپ شده.

```
tl
title if document
Au
Author name
AB
```

البته قسمت بالا نیز اینگونه تایپ شده.

```
tl
i title of document
au
```

و شما می توانید تصور کنید که چگونه به ترتیب و به نوبت تایپ شده است.

جدول 9-2

فرمان‌های فرمت کننده mm
 چکیده را آغاز کنید که با ae پایان می‌پذیرد. AS
 اسم نویسنده به عنوان اولین نشانه AU
 متن را با حروف پر رنگ آغاز کنیو یا نشانه‌ها را پر رنگ کنید B
 متن را کنار هم بگذارید و در صورت لزوم در صفحه بعدی شناور کنید DF
 متن نمایشگر را آغاز کنید که با DE پایان می‌گیرد. DS
 معادله را آغاز کنید و رودی eqn که با EN پایان می‌گیرد. EQ
 پا نوشت را آغاز کنید که با FE خاتمه می‌یابد FS
 متن ایتالیک را شروع کنید و یا نشانه‌ها را ایتالیک کنید I
 سطح th-n عنوان شماره گذاری شده H
 عنوان شماره گذاری نشده HU
 پاراگراف از 1 pt برای پاراگراف‌های فاصله دار استفاده می‌شود P
 برگشت به فونت لاتین R
 عنوان به دنبال فرمان mm بعدی می‌آید TL
 جدول را شروع کنید (ورودی (tb) که با TE خاتمه می‌پذیرد. TS

کارکتر خاص دیگری که گاهی رخ می‌دهد، فاصله غیر قابل بسط است یک به دنبال فاصله و یا جای خالی می‌آید. نرمال troff یک فاصله را برای همستون سازی حاشیه‌ها بسط می‌دهد. اما یک فاصله غیر قابل بسط هرگز همستون نمی‌شود. این شبیه هر کارکتر دیگریست که عرض مشخصی دارد و می‌تواند برای پذیرفتن واژه‌های چند گانه به عنوان یک نشانه تنها به کار رود.

i title /of/docn ment

جدول 9-3

troff برخی مراتب کارکتر خاص
 -- خط تیره
 \hy خط تیره، مانند بالا
 -\em dash\cem
 \& حفاظت از تناوب پیشین
 \blank فاصله غیر قابل بسط
 \اکارکتر نموی خروجی
 \e گلوله
 \bn\خنجر
 \bg
 \(*a
 \fX* تغییر به فونت
 \fxx تغییر به فونت
 \sn n=0 قبلی، n تغییر به سایز پونت
 \s+-n تغییر اندازه پونت نسبی

تغییرات فونت و سایز:

بسیاری از تغییرات فرمت و فونت می تواند در ماکروهای ابتدای خط ایجاد شود. شبیه I. ولی بعضی اوقات تغییرات به صورت خطی و ردیفی ساخته می شوند. کارکتر خط جدید یک تفکیک کننده کلمه است. چنانچه یک فونت در وسط کلمه تغییر می یابد ماکروها غیر قابل استفاده هستند. زیر بخش ها در این مورد بحث می کند که چگونه troff بر این مشکل غلبه می کند. توجه داشته باشید که این troff است که از مهارت ها و قابلیت ها حمایت می کند و نه بسته های ماکروی ms.

troff از کارکترهای یک اسلش برای معرفی فرمان های خطی استفاده می کند. f فرمانی برای تغییر فونت و s فرمانی برای تغییر اندازه پونت بسیار رایج است.

فونت توسط f با یک کارکتر به سرعت بعد از f مشخص می شود.

```
a\fbf riv\fiolous\fr\fivar\fbiety\fr
of\ffonta\fp
```

متغیر فونت \fp به فونت قبلی باز می گردد- به هر صورتی که فونت قبل از آخرین تغییر بوده است- بسیاری از فونت ها دارای 2 نام کارکتر هستند، که توسط فرمت f(xx\ جایابی که xx نم فونت می باشد مشخص می شوند. برای مثال فونتی که روی حروفچین ما جایی که برنامه های این کتاب چاپ شده cw خوانده می شود بنابراین keyword اینگونه نوشته می شود.

\f(cw keyword)\fp

که البته تایپ کردن آن بسیار ملال آور است. بنابراین ماکرو یکی از مصداق های ms_omm است که ما دیگر نباید back slash را بخوانیم و یا تایپ کنیم. ما برای حروفچینی واژه های خطی از این روش استفاده می کنیم مانند troff برای مثال:

the
cw troff
for matter

تصمیمات فرمت کننده که توسط ماکروهای تعریف می شود برای تغییرات بعدی بسیار ساده است. تغییر سایز توسط مراتب \sn معرفی می شود جایی که n یک یا دو ورقعی است و سایز جدید را مشخص می کند. \s8 به هشت نوع پونت تغییر می بابد. تغییرات نسبی ممکن است توسط علامت منفی و یا مثبت در مقابل سایز ساخته شود. برای مثال کلمه می تواند به شکل حروف کوچک چاپ شود.

\s-25mall caps\so

so باعث می شود که سایز میزان قبلی خود باز گردد. این قایسی ست از \fp البته به رسم troff به صورت \sp بیان نمی شود. مصداق ms در نظر ما یک ماکروی با حروف بزرگ (uc(upper case)) برای کار.

فرمان های تروف troff اصلی:

با یک بسته ماکروی خوب شما باید مقداری از فرمان های troff را بدانید برای کنترل فضای خالی و یا پر کردن و قرار دادن تب و غیره فرمان br باعث یک انقطاع شکست می شود که ورودی بعدی به همراه br. بر روی خط خروجی جدید خواهد شد. که این می توانست برای مثال برای دو قسمتی کردن عنوان های بلند بر روی مکانی خاص استفاده شود. فرمان nf پر کردن سطرهای خروجی را قطع می کند. هر سطر ورودی مستقیم به سوی یک از سطرهای خروجی می رود. فرمان fi پر کردن از عقب را قطع می کند. فرمان ce سطر بعدی را در مرکز قرار می دهد. فرمان bp یک صفحه جدید را آغاز می کند. فرمان sp باعث می شود که تنها یک سطر خالی در خروجی ظاهر شود. یک فرمان sp می تواند با یک نشانه همراه باشد تا مشخص کند چند سطر خالی و چند فضای خالی نیاز است.

sp 3 leave 3 blank lines
sp 05 leave dank hal-line
sp 1.5i leave 1.5 in
sp 3p leave 3 points
sp 3.1c leave 3.1 centimeters

فضایی وسیع در انتهای صفحه می شود. بنابراین sp. با bp برابر می کند. فرمان ta تب را در جایگاه هایش قرار می دهد (که در هر اینچ ایتالیک شده است)

رمان بالا جایگاه های تب را در فواصل معینی از حاشیه چپ مشخص می کند با \sp هر شماره ای در هر اینچ قرار دارد اگر با همراه باشد جایگاه تب که توسط r افزوده شده متن را در جایگاه تب بعدی همستون می کند.

c یک تب مرکزی را تشکیل می دهد فرمان

ps اندازه های پونت را به n تنظیم می کند و فرمان $\text{ft } x \setminus$ فونت را به x. قواعد در مرود سایزهای مفوی و بازگشت به میزان قبلی در $\text{f} \setminus$ مشابه است.

ماکروهای تعریف کننده:

ماکروهای تعریف کننده ما را بیشتر به سوی پیچیدگی های troff می برد. اما ما می توانیم بسیاری از موارد اصلی را نشان دهیم. برای مثال در اینجا تعریفی از وجود دارد.

de cw sart a defin tion

$\&\f(cw\ \$\fp\ \2 font chang arund first argument

End of definition

وقتی که ماکرو راه اندازی می شود $n \setminus$ میزان نشانه $n = tn$ را تهیه می کند و اگر نشانه $n - tn$ فراهم نشد خالی می ماند. یک اسلش دو تایی در ارزیابی $n \setminus$ در مدت تعریف ماکرو تاخیر ایجاد می کند. از اینکه نشانه به عنوان یک فرمان troff تفسیر شود جلوگیری می کند پیش پردازنده های tbl, eqn

troff یک برنامه پیچیده و بزرگ است چه در ورودی و چه در خروجی بنابراین تغییر دادن آن برای قبول یک کار جدید به سادگی صورت نمی گیرد. طبق توسعه برنامه ها برای ریاضیات و جدول ها که رهیافت متفاوتی دارند و طرح ریزی زبان های مجزا که توسط برنامه های مجزا تحقیق می یابد، به عنوان یک پیش پردازنده برای troff عمل می کند در حقیقت troff یک زبان همگذاری اسمبلی است برای یک ماشین حروفچینی و به آن برگردانده مترجمه می شوند.

اول eqn به وجود آمد. آن اولین کاربرد yacc برای یک زبان برنامه ریزی نشد بود. tbl بعد از آن آمد که شبیه به eqn بود، البته با یک نحو نا مربوطه و غیر وابسته.

از yacc استفاده نمی کند زیرا گرامرش بسیار ساده است.

مسیر برنامه های مجزائی تقسیم می شود گذشته از فاکتورگیری کار به چند بخش مسیرهای اطلاعاتی ارتباط بین بخش ها و برنامه ها را کاهش می دهند. این پونت آخری مهم است و نیازی به دستیابی به کد منبع برای ساخت پیش پردازنده ها ندارد. علاوه بر این با وجود مسیرها هیچ فایل بزرگی که ایجاد ناراحتی کند وجود ندارد و گرنه موارد تشکیل دهنده عامل ها برای خط زدائی به صورت مجزا اجرا می شوند.

وقتی که برنامه های مجزا با مسیرها ارتباط برقرار کنند مشکلی به وجود می آید و تا وقتی که ورودی و خروجی های زیادی وجود دارد سرعت کمی دچار اختلال می شود. tbl و eqn یک بسطدهی 8 به 1 از ورودی به خروجی ایجاد می کنند. از همه مهمتر اطلاعات تنها یک مسیر را دنبال می کنند. برای مثال، eqn می تواند سایز پونت موجود را تعیین کند. که منجر به زشتی زبان می شود. در نهایت گزارش دادن خطا در اینجا سخت می شود. استفاده از تفکسکها خطاها را سنگین تر نشان می دهد. بنابراین بیشتر پیش پردازنده ها دیگر که نوشته شده اند دارای طرح و مدل یکسانی هستند.

بگذارید بحث مختصری در مورد tbl داشته باشیم. اولین چیزی که ما می خواهیم نشان دهیم جدول اپراتورها از فایل hoc می باشد. tbl فایل ورودی خود را و یا ورودی استاندارد را می خواند و متن ها بین فایل های $(ts(table start)$ و $(TE(Table end)$ را تبدیل می کند.

به فرمان troff که جدول را چاپ می کند. ستون را مرتب می کند و از تمامی جزئیات چاپی مراقبت می کند. سطرهای TS.TE کپی شده هستند بنابراین یک بسته کاکرو می تواند تعاریف مناسبی برای آن ها فراهم کند. برای مثال ادامه جدول روی یک صفحه و تنظیم آن در محیط متن.

گرچه نیازی است که به یک راهنمای tbl برای ساختن جدول های پیچیده نگاهی بیندازید. یک مثال برای نشان دادن شکل های معمولی و رایج آن کافیهست. در اینجا مثالی از فایل noc وجود دارد.

(فرمان صفحه 303)

که این برنامه جدول زیر را تشکیل می دهد.

جدول 1- اپراتورها ترتیب کاهش تقدم
 (**FORTRAN) به توان رسانی
 منفی سازی منطقی و حسابی
 تقسیم، ضرب
 تفریق، جمع
 بزرگتر یا مساوی، بزرگتر، اپراتور نسبی
 کمتر یا مساوی، کمتر
 مساوی، نامساوی منطقی (همه عملوندهایی که ارزیابی شده اند)
 منطقی و یا (همه عملوند هایی که
 ارزیابی شده اند)
 نسبت دهی متناظر

کلماتی که قبل از سعی کالن (center,box) قرار دارند خواص جدول را توضیح می دهند یعنی آن را در وسط جدول به طور افقی قرار می دهد و یک مستطیل دور آن می کشد. دیگر امکانات آن شامل all box doublebox (که هر کدام موارد داخل مستطیل است) و expand جدول را به عرض صفحه بسط می دهد می شود.

سطرهای بعدی فرمت های بخش بعدی جدول را توضیح می دهد که در این حالت سطر عنوان و قسمت اصلی جدول قرار دارد. اولین تشریح برای سطر اول جدول است. دومین تشریح برای سطر دوم کاربرد دارد. و آخرین کاربرد برای سطرهای باقی مانده است در جدول 1 تنها و سطر تشریح وجود دارد بنابراین دومین تشریح برای هر سطر جدول به جز سطر اول کاربرد دارد. کارکتر فرمت برای مواردی که در وسط ستون قرار دارد c است. f و b برای همستون بازی است راست و چپ و n برای همستون سازی عددی بر روی پونت ده دهی. S یک ستون گسترده شده را مشخص می کند. در این حالت CS یعنی قرار دادن در وسط جدول توسط گسترده کردن ستون دوم

و همچنین ستون اول یک فونت می تواند برای یک ستون تعریف شده باشد. تشریح `tbl` برای `cifce` یک ستون از چپ مرتب شده در فونت `cw` چاپ می کند.

متن جدول به همراه اطلاعات فرمت کننده می آید. کارکتر تب ستون ها را مجزا می کند و برخی از فرمان های `troff` مانند `sp`. در داخل جدول ابل درک هستند. به علامت هایی - و `==` توجه کنید که در ستون به `tbl` می گوید سطرهایی را در عرض جدول با این پونت بکش. `tbl` جدول های متنوع و عریض تری را نسبت به مثال ساده ای که بیان شد تشکیل می دهد و حتی می تواند متن جدا داخل مستطیل را پر کند. عناوین ستون را مرتب کند و غیره ساده ترین راه برای استفاده از آن در جدول های پیچیده جستجوی مثال مشابهی ست در برنامه های یونیکس `volume 2Aunix` و تطبیق فرمان هاست.

عبارات ریاضی

دومین پیش پردازنده `troff`، `eqn` است که یک زبان توضیحی دهنده عبارات ریاضی را به فرمان های `troff` برای چاپ آنها تبدیل می کند. به صورت خودکار تغییرات فونت و سایز را دست کاری کرده و برای کارکترهای ریاضی استاندارد نامی فراهم می کند. ورودی `eqn` معمولاً بین سطرهای `EQ` و `EN` ظاهر می شود و `TS` و `TE` را قیاس می کند.

EQ
IX SUB
EN

برای مثال فرمان بالا `xi` را می سازد اگر بسته ماکروی `ms` مورد استفاده قرار گرفته باشد. معادله به عنوان یک نمایشگر چاپ می شود و یک نشانه گزینه ای برای `EQ` یک شماره معادله مشخص می کند برای مثال انتگرال زیر:

اینگونه نوشته می شود.

زبان `eqn` بر پایه روشی قرار دارد که با صدای بلند ریاضی صحبت می کند. تفاوت بین ریاضی صحبت شده و ورودی `eqn` آکولاد است که برای `eqn` پرانتز محسوب می شود - آنها قواعد تقدم پیش فرض زبان را قطع می کنند - البته پرانتزها اهمیت خاصی ندارند. جای خالی ها عموماً مهمترند توجه کنید که اولین `zeta` توسط جای خالی ها در مثال بالا احاطه شده و ازه های کلیدی مثل `zeta` و `over` تنها وقتی تشخیص داده می شوند که توسط فضاهای خالی و یا آکولادها احاطه شده باشند و نه هر چیزی که در خروجی باشد. برای وارد کردن فضای خالی به خروجی از یک کارکتر که به صورت استفاده کنید. برای تشکیل آکولاد اینگونه استفاده کنید.

{and}

دسته بندی متعددی از واژه های کلیدی `eqn` وجود دارد. حروف هایب یونانی با حروف کوچک یا بزرگ استفاده می شود. مانند `grad,infinty,int,sum` و `LAMBDA` کارکترهای ریاضی دیگر دارای نام هستند مانند `lambda`

اپراتورهایی موقعیتی مانند `over,to,from,sup,sub`

صورت بالا اینگونه تفسیر می شود.

اپراتورهایی شبیه `eqrt` و پرانتزهای قابل گسترش، آکولادها و... وجود دارد. `eqn` ستون‌ها و ماتریس‌های موضوع را خواهد ساخت. فرمان‌هایی برای کنترل فونت و سایز و موقعیت وجود دارد برای وقتیکه پیش فرض‌ها صحیح نباشد.

قرار دادن عبارات ریاضی کوچک مانند `log(n)` در اصل متن رایجتر است تا در نمایشگرها. واژه‌های کلیدی `eqn` برای `delim` یک جفت کارکتر مشخص می‌کند به منظور در پرانتز قرار دادن عبارات خطی. این کارکترها که به عنوان فاصله‌های چپ و راست استفاده می‌شوند معمولاً شبیه به هم هستند. علامت دلار (\$) اغلب مورد استفاده قرار می‌گیرد. البته `hoc` از (\$) برای نشانه‌های استفاده می‌کند ما از در مثال هایمان استفاده می‌کنیم. نیز برای جداسازی مناسب است بسیاری از کارکترها دارای موجودی ویژه‌ای در برنامه‌های متنوع خود هستند که می‌توانند رفتارهای غیرعادی تماشایی را به دست دهد.

بنابراین بعد از بیان:

EQ
delim
EN

عباراتی خطی نظیر می‌تواند اینگونه چاپ شود.

فرمان (صفحه 305)

این جدول همچنین نشان می‌دهد که چگونه `tbl` پونت‌های ده دهی را در ستون‌های عددی به خط می‌کند. خروجی جدول 3 را به دست می‌دهد.

در نهایت تا زمانیکه `eqn` هر ردیف از حروفی را که مشخص نیست ایتالیک می‌کند ایتالسیک کرده‌واژه‌های که از `eqn` استفاده می‌کند روشی رایج و معمولی است.

برای مثال به صورت `word` چاپ می‌شود. پس مراقب باشید `eqn` بسیاری از واژه‌های معمولی را تشخیص می‌دهد مانند `to fro` و با آن‌ها رفتار خاصی دارد، و فضاهایی خالی را حذف می‌کند. بنابراین در حین استفاده از این راه کار دقت کنید.

خروجی:

شما باید تمامی پیش پردازنده‌ها و `troff` را برای به دست آوردن خروجی، ردیف و به خط کنید. و باد فایل‌تان را آماده سازید. `tbl` درخواست فرمان هاست. سپس `eqn` و پس از آن `troff` است. اگر تنها از `troff` استفاده می‌کنید تایپ کنید.

(troff-ms filenames (or-mm\$

در غیر این صورت شما باید نام فایل‌های نشانه را برای فرمان اول در مسیر اطلاعاتی مشخص کنید و اجازه دهید دیگران ورودی استاندارد خود را بخوانند.

eqn filenames :troff -ms

or

tbl file names: eqn:troff-ms

ما دریافتیم که نوشتن برنامه‌ای به نام `doctype` مفید است که مراتب فرمان را استنباط می‌کند. (فرمان صفحه 306)

`doctype` توسط ابزارهایی که در فصل 4 بحث شد تحقیق می‌یابد. به خصوص یک برنامه `awk` مراتب فرمان را جستجو می‌کند. که این مراتب مورد استفاده پیش پردازنده‌ها ست و سطر فرمان را به منظور راه اندازی این نیازها و فرمت کردن فایل چاپ می‌کند. همچنین فرمان

`pp` را که توسط بسته‌های ام اس (ms) درخواست‌های فرمت کنه مورد استفاده قرار می‌گیرد راجستجو می‌کند.

(گزینه `n`- برای `egrep` باعث می‌شود که عنوان گذاری برای هر نام فایل بر روی هر خطی متوقف شود. متأسفانه این گزینه برای همه طرح‌های سیستم وجود ندارد) ورودی اسکن می‌شود مجموعه اطلاعات بر روی جزئیات مورد استفاده قرار می‌گیرد. بعد از اینکه کل ورودی امتحان شد در تربیت راست برای چاپ خروجی پردازش می‌شود. جزئیات توسط پردازنده‌های استاندارد برای فایل‌های تروف `t` `roff` فرمت کننده مشخص می‌شوند. ولی به طور کلی بگذارید که دستگاه مراقب جزئیات باشد.

`doctype` مانند `bundle` مثالی است که از یک برنامه‌ای را می‌سازد. گرچه این برنامه نوشته شده است اما نیازمند است که یک کاربر سطر را در شل دوباره تایپ کند.

وقتی که فرمان `troff` اجرا می‌شود شما باید در نظر داشته باشید که رفتار `roff` به صورت یک سیستم وابسته است. در بسیاری از تاسیسات خود سیستم حروفچین را مستقیماً به کار می‌اندازد. درحالیکه در سیستم‌های دیگر اطلاعات بر روی خروجی استاندارد آن ساخته می‌شود که باید توسط یک برنامه مجزا بر روی حروفچین قرار گیرد.

به هر حال اولین شکل از این برنامه از `egrep` و `sort` استفاده نکرد. `awk` به تنهایی تمامی ورودی را اسکن کرد. و آن را برای یک فایل بزرگ آرام و آهسته کرد. بنابراین `egrep` را برای یک تحقیق سریع اجرا کردیم و `sort` را برای خلاص شدن از کپی‌ها. برای فرمان‌های عادی دو فرایند ساخته شده اضافی برای غربال کردن داده‌ها کمتر است از اجرای `awk` بر روی بسیاری از ورودی‌ها. در اینجا مقایسه‌ای بین `doctype` و یک طرح که تنها `awk` را اجرا می‌کند نشان داده شده و در محتوای این فصل نیز استفاده شده است.

از قرار معلوم این مقایسه برای طراحی که از سه فرایند استفاده می‌کرده مطلوب بوده است. و این عملیات در ماشین توسط یک کاربر انجام شده است توجه کنید که ما ابتدا به یک طرح ساده‌دارای کارایی دست یافته‌ایم قبل از آنکه بهینه سازی را آغاز کنیم.

4-9 صفحه راهنما

مهمترین مستندات برای یک فرمان معمولاً صفحه راهنما می‌باشد توصیف یک صفحه‌ای در راهنمای برنامه ریز `usr\man` برنام ریزی (`UNIX` صفحه راهنما در دایرکتوری استاندارد ذخیره می‌شود. معمولاً `usr\man` در یک دایرکتوری فرعی طبق بخش‌های راهنما شماره گذاری می‌شود. صفحه راهنمای `noc` چون که فرمان کاربر را توضیح می‌دهد به این صورت نگه داری می‌شود.

`usr\man\man1\noc.1\`

صفحه راهنما، فرمان `man(1)` را چاپ می‌کند. یک فایل، شل `man-nroff` را اجرا می‌کند. بنابراین `man noc` راهنمای `noc` را چاپ می‌کند اگر برای یک بخش بیش از یک نام آورده شد به همان صورت برای `man` نیز می‌شود (بخش 1 فرمان را توضیح می‌دهد در حالی که بخش 7 ماکروها را توضیح می‌دهد) بخش می‌تواند `man` را مشخص کند.

man 7 man

فرمان بالا را تنها توصیف ماکروها را چاپ می‌کند اقدامات پیش ساخته باید تمامی صفحه را بانام‌های مشخص شده چاپ کنند با

استفاده `man-troff` اما `troff` استفاده می کند تولید می کند.

نویسنده صفحات راهنما روی دایرکتوری فرعی مناسب `\usr\man` یک فایل می سازد. فرمان `man`، `troff` و یا `nroff` را با یک بسته ماکرو برای چاپ صفحه فرا می خواند.

فرمان صفحه 309 نتیجه کار چنین است:

اختلاف در سر و کار داشتن با گزینه هاست: `troff` و `nroff`

که آیا `eqn` یا غیره اجرا می شود یا نه. ماکروهای راهنما که با `troff-man` راه اندازی می شوند فرمان `troff` را که به سبک یک راهنما فرمت شده است را تعریف می کنند. آنها اساس "شبهه ماکروهای `ms` هستند. اما تفاوت هایی نیز وجود دارد به خصوص در قرار گرفتن عنوان و در فرمان های تغییر فونت ماکروهای مستند سازی می شوند مختصر¹ در `(man)1` اما اساس آن به راحتی در خاطر می ماند. صفحه بندی صفحه راهنما به شکل زیر است:

(فرمان صفحه 309)

اگر هر بخشی خالی باشد عنوان آن حذف می شود. سطر `TH` و بخش های `DES CRIPNAME` دارای احکام و دستوراتی هستند.

TH COMMAND SECTION-NUMBER

سطر بالا یک فرمان نامیده می شود و شماره بخش را مشخص کند. سطرهای متنوع `SH`. قسمت های صفحه راهنما را مشخص می کند. قسمت `NAME` و `SYNOPSIS` ویژه و خاص هستند و بقیه شامل یک نثر معمولی هستند. قسمت `NAME,COMMAND` (فرمان) خوانده می شود و یک توصیف یک خطی از آن فراهم می سازد. قسمت `option,synopsis` (گزینه) خواهد می شود که آن ها را توضیح نمی دهد. چنانچه در هر-قسمتی ورودی خالی باشد تغییرات فونت می تواند توسط ماکروهای `B.I.R` مشخص شود. در قسمت `synopsis` نام و گزینه با حروف درشت هستند. و بقیه اطلاعات به صورت لاتین است. قسمت های `NAME(1)` و `synopsis` برای مثال اینگونه هستند.

```
sll name
ed\teat editor
sh synopsis
Bed
I
B-X
NAMEI
NAME
ed-text editor
synopsis
```

به کاربرد نسبت به شکل ساده دقت کنید.

قسمت `DESCRIPTION` فرمان و گزینه اش را تعریف می کند. در بسیاری از موارد توصیفی از فرمان است نه زبانی که فرمان را توضیح می دهد و صفحه راهنمای

`(1)cc` زبان `C` را تعریف نمی کند. او بیان می کند که چگونه برای ترجمه برنامه های `C` فرمان `cc` را می توان اجرا کرد و چگونه می توان بهینه سازی را راه اندازی کرد در جایی که خروجی در چاپ است و زبان در راهنمای مرجع مشخص شده و در قسمت `seealso` که مربوط به `(1)cc` می شود ذکر شده است. به عبارت دیگر مقوله های مطلق نیستند `(7)man` توصیفی از یک زبان راهنمای ماکروهاست. به

صورت قراردادی در قسمت DESCRIPTION نام‌های فرمان و بر چسب‌های گزینه‌های به صورت ایتالیک چاپ می‌شود ماکروهای I که اول نشانه‌ها را به ایتالیک تایپ می‌کنند و RI که اول نشانه‌ها را به ایتالیک و سپس به لاتین تایپ می‌کند این عمل را ممکن می‌سازد ماکروی RI وجود دارد به خاطر اینکه ماکروی I در بسته‌های man با آن سهیم نمی‌شود.

قسمت files فهر فایلی را که به طور صفتی توسط فرمان استفاده می‌شود را ذکر می‌کند. اگر خروجی غیر عادی که توسط فرمان ساخته شده است وجود داشته باشد نیاز است که مضمول آن شود که این می‌تواند یک پیام عیب‌شناسی باشد.

قسمت BWGS ندکی ناشناخته است. عیب و نقص‌های گزارش شده زیاد هم دارای خ‌خطا و اشتباه نیستند.

و به عنوان یک خطای جزئی و ساده می‌بایست قبل از اینکه فرمان نصب گردد بهبود یابد برای اینکه بدانید چه چیزی وارد قسمت BWGS,DIAGNOSTICS م می‌شود باید در راهنمای استاندارد جستجو کنید. یک مثال توضیح دهد که چگونه می‌توان یک صفحه راهنما نوشت. برنامه‌ای برای (1\hoc\hoc\man1\man\uis) در شکل 1-9 و 2-9 نشان داده شده است.

نام
hoc: زبان پونت شناور کننده واکنش
خلاصه:
hoc[file]
توصیف:
noc یک زبان ساده را برای یک پونت شناور کننده حسابی تفسیر می‌کند، که در مورد سطح بیسیک است در نحو و عملکرد و روش شبیه به C می‌باشد. و دارای نشانه و خاصیت بازگشتی است.
noc و رودی استاندارد را تفسیر می‌کند.
ورودی noc شامل عبارات و بیاناتی ست. عبارات ارزیابی شده است و نتایج آن چاپ شده عبارات به ویژه نسبت دهی‌ها عملکردها و یا توضیح روش‌ها تا زمانی صریحاً چاپ نشود خروجی را تشکیل نمی‌دهد.
SEE Also
noc: زبان واکنشی برای پونت‌های شناور کننده حسابی توسط Rob pike, Brian Kerrigan
خطاها:
بازسازی در میان عملکرد و توصیف روش‌ها عملی است ناقص.

5-9 دیگر ابزار آماده سازی فایل: برنامه‌های دیگری برای آماده سازی فایل وجود دارد. فرمان توسط واژه‌های کلیدی به مرجع نگاه می‌کند و دست آخر استنادهای خطی و قسمت مرجع را به فایل شما نصب می‌کند.

برای توصیف ماکروهای مناسب شما می‌توانید تربیتی دهید که refer مرجع‌ها را به همان سبک مورد نظر چاپ کند.

توضیحاتی برای تنوع ژورنال‌های علمی کامپیوتر وجود دارد refer قسمتی از است و برای بسیاری از طرح‌ها انتخاب شده است.

بر روی تصاویر همان کاری را می‌کند که معادل‌ها انجام می‌کند تصاویر نسبت به معادله‌ها بسیار پیچیده‌تر هستند حداقل در حد حروفچینی و هیچ‌چ روش شفاهی چگونگی صحبت در مورد تصاویر وجود ندارد. بنابراین زبان‌ها برای یادگیری و استفاده از آن

کارهایی انجام داده‌اند در این جا یک تصویر ساده و عبارات مربوط به آن در وجود دارد.

تصویرهای این کتاب همگی توسط `pic,deapic` انجام شده که جزئی از نمی‌باشد اما قابل دسترسی هستند.

HOC یک زبان واکنشی برای شناور شدن پونت‌های حسابی

Brian kerni ghan
rob pike

چکیده: HOC یک مفسر برنامه‌پذیر ساده است که عبارات پونت را شناور می‌کند که دارای روند کنترل به سبک C تعریف و عملکردها و عملکردهای پیش ساخته عددی عادی است مانند کوسینوس و لگاریتم.

1 - عبارات: HOC زبان عبارات است مانند که در آن چندین عبارت کنترل روند و بیاناتی شبیه نسبت دهی که عباراتی هستند که به میزان آن‌ها توجه تری شده است برای مثال اپراتور سینوس دهی میزان عملوند راستش را به عملوند چپش نسبت می‌دهد و میزانش محصول و بهره می‌دهد. گرامر عبارات اینچنین است:

number
variable
(eapr)
enpr binop enpr
unop enpr
function

اعداد پونت را شناور می‌کند. فرمت ورودی که توسط تشخیص داده می‌شود ارقام پونت دهی اعداد توان علامت دار حداقل یک رقم و یا یک پونت دهی وجود داشته باشد بقیه عامل‌ها گزینشی هستند.

اسامی متغیر فرم‌های فرمت شده‌ای هستند که یک حرف توسط یک ردیف حرف و یا رقم دنبال می‌شود. `binop` به اپراتورهای دهی دهی مانند مجموع و یا مقایسه منطقی رجوع می‌کند. `unop` به دو اپراتور منفی سازی رجوع می‌کند. منفی سازی منطقی `not` و منفی سازی حسابی تغییر علامت جدول 1 اپراتورها را لیست کرده است.

جدول 1: اپراتورها
^ به توان رسانی (**FORTRAN)
منفی سازی حسابی و منطقی !
تقسیم ضرب *
تفریق، جمع + -
اپراتورهای نسبی بزرگتر و بزرگتر مساوی <=>
کوچکتر و کوچکتر مساوی >=>
مساوی. نامساوی !=
منطقیو (همه عملوندها ارزیابی شده اند) &&
منطقی یا همه عملوندها ارزیابی شده اند
نسبت دهی متناظر =

عملکردها همان گونه که قبلاً توضیح داده شد ممکن است توسط کاربر تعریف شود نشانه های عملکرد عباراتی هستند که توسط کاما (،) جدا می شوند. تعداد زیادی عملکردهای از پیش ساخته وجود دارد که همه آنها تنها دارای یک نشانه اند که در جدول 2 توضیح داده شده است.

جدول 2 عملکردهای از پیش ساخته
قدر مطلق
xatan(x) آرک تانژانت
xcos(x) کوسینوس
xexp(x) به توان رسانی
xint(x) قسمت صحیح
log(x) بر پایه x لگاریتم
log10(x) بر پایه 10 x لگاریتم
xsin(x) سینوس
xsqrt(x) رادیکال

عبارات منطقی دارای میزانی برابر 1/0 صحیح و 0.0 غلط است همچنین در c هر میزان صفحه ی که برداشته می شد صحیح بود. همچنین HOC دارای مقداری محتوای از پیش ساخته است که در جدول 3 نشان داده شده است.

3 - عبارات و روند کنترل:

عبارات HOC یک گرامر دنبال کننده دارد:

```

expr
variable expr
pyocedur (arglist)
while
if(expr)stmt else stmt
stmtlist
print expr list
return optional-expr
nothing
stmlist stnt

```

یک نسبت دهی توسط پیش ساخته‌ها به عنوان عبارتی تجزیه و تحلیل می‌شوند تا یک عبارت. بنابراین نسبت دهی ای که تایپ شده میزان خود را تایپ نمی‌کند.

توجه داشته باشید که سمی کامن در تنها در hoc وجود ندارد. عبارات با سطر جدید پایان می‌پذیرند که موجب اجرای رفتار خاصی می‌شود.

بیانات عبارات if است که مجاز نیز می‌باشد.

```

if (x<0)print(>) else print(z)
if (x<0)
else
print(z)

```

در مثل آکولادها دارای احکام و دستوراتی هستند. خط جدید بعد از if عبارت را پایین می‌دهد و یک خطای نحوی می‌سازد که پرانتز را حذف می‌کند.

نحو و معنانشناسی مهارت‌های روند کنترل hih اساساً شبیه به c است. while و if نیز در وجود دارد با این تفاوت که انقطاع و یا دنباله کلام در آن وجود ندارد.

3 - ورودی و خروجی: خواندن و چاپ کردن

عملکرد ورودی خواندن مانند بقیه پیش ساخته‌ها تنها یک نشانه بر می‌گزینند. بر عکس بقیه پیش پردازنده‌ها نشانه‌ها یک عبارت نیستند بلکه نام متغیرهاست. شماره بعدی همانطور که در بالا تعریف شده از ورودی استاندارد خوانده می‌شود و به متغیرهای نام گذاری شده نسبت داده می‌شود. میزان بازگشت صحیح است اگر میزان آن خوانده شده باشد و غلط است اگر به پایان فایل و یا یک خطا مواجه شده باشد.

خروجی با عبارت تولید می‌شود نشانه‌های کاما جدا شده در لیست عبارات و یک گیومه دو تایی مانند سطر جدید باید فراهم شود. آنها هیچگونه به طور خودکار فراهم نمی‌شود.

4 - روش ها و عملکردها

روش ها و عملکردها در hoc متمایز هستند گرچه آنها با یک مکانیک مشابهی تعریف می شوند. این تمایز برای چک کردن خطای زمان اجراست. این خخطا برای روش بازگشت به میزان است و برای عملکرد بازگشتی نیست. نحو تمایز این گونه است:

```
funcron func name ()stmt
procedur proce mame()stmt
```

اسم ممکن است اسم هر متغیر عملکرد از پیش ساخته باشد که مستثنی شده است. بر عکس اصل عملکرد و یا روش کار می توان هر بیانی باشد و نه الزاماً یک بیان مرکب و مختلط تا زمانیکه سعی در مفهومی ندارد تنه اصلی روش خالی با یک جفت آکولاد خالی فرمت می شود.

عملکردها و رویه ها ممکن است نشانه ای گزینش کنند که وقتی راه اندازه می شوند توسط کاماها مجزا می شوند. نشانه ها به عنوان یک شل رجوع داده می شوند. \$3 به سومین نشانه رجوع می کند. آنها توسط میزان گذارنده می شوند عملکردها نیز با متغیرها همتراز هستند. رجوع به یک نشانه شماره گذاری شده بزرگتر از شماره نشانه هایی که توسط روال کار گذارنده می شود اشتباه و خطاست. چک کردن خطاها به صورت پویا انجام می پذیرد گرچه ممکن است روال کاری شماره های متغیر نشانه ها را داشته باشد اگر نشانه اصلی ونخستین بر روی شماره شناسه ها به منظور رجوع شدن تاثیر بگذارد.

عملکردها و رویه ها ممکن است باز گردند. ام پشته ها stack از نظر عمق محدود شده اند. در زیر تعریف hoc در عملکرد Ackermann نشان داده شده است.

```
hoc
func ack(){
if ($ 1==0)return$s
if($2=0) return ack
return ack($1-1,ack($1,$2-1))
ack(3,2)
```

```
29
ack(3,3)
61
ack(3,4)
hoc:sta ek too deep hear line 8
.....
```

5 - مثال ها

فرمول استرلینگ

$n!2n$

```
hoc
func stirl (){
return sqrt(2*$1*pi)
}
stirl(20)
2.4328818e+18
```

عملکرد فاکتوریل : !n

```
funcfac()is($1<=0)return 1 else return$ 1*fac($1-1)
```

نسبت به فاکتوریل به تقریب استرلینگ

```
i=9
while(li=i+1)<=20){
print,fac(i)stirl(i)in
}
```

```
10 1.0000318
11 1.0000205
12 1.0000224
13 1.0000166
14 1.0000146
16 1.0000128
17 1.0000114
18 1.0000102
19 1.000092
20 1.000083
```