مقدمه مترجم

با سلام به خواننده محترم: همانطور که در وبلاگ مخصوص ترجمه فارسی این راهنما نوشتم:

من در ترجمه این راهنما برای درج اشارات تلویحی و افزودن قابلیت تشخیص بیشتر به خط فرمانها، یک رنگ آمیزی خاص به شرح زیر را به کار برده‌ام:

برای فرمانها(اعم از دستورات داخلی، خارجی و برنامه‌های کاربردی) از ا ین ر نگ مانند cd

برای کلید واژه‌های bash از این رنگ مانند]] یا do

برای کاراکترهای ویژه از **این رنگ** مانند && اما نه در تمام موارد

برای متغیرها از این رنگ مانند PATH

برای گزینه‌های دستور از **این رنگ**مانند f-

برای توابع از **این رنگ** مانند function()

برای داده‌ها از این رنگ مانند story

خروجی در ترمینال و برای لینک‌های خارج از راهنماااز این رنگ مانندBashGuide

برای لینک به بخش دیگری از این مجموعه از این رنگ مانند نقل‌قول‌ها که در حال حاضر تا تکمیل شدن ترجمه کار نمی‌کنند.

برای خطا در خط فرمان از این رنگ

و از *این رنگ* برای شناسه‌ها و بعضاً نمایان کردن برخی از قسمت‌ها و همچنین تاکید در متن استفاده شده است، سعی کرده‌ام تا آنجا که ممکن است آن را رعایت نمایم ولی امکان دارد که گاهی از نظر دور مانده باشد.

همچنین تلاش بسیاری شده که از بهم ریختگی و دگرگون شدن خط دستورها که در میان متن فارسی(متن چپ به راست در میان متن راست به چپ) درج شده‌اند و در نوشته‌های وبلاگها بسیار با آن برخورد نموده‌ام به طور جدی پرهیز نمایم. لطفاً اگر با موردی در این راهنما یا ترجمه‌های قبلی برخورد نمودید، اطلاع بدهید. ممنون خواهم شد.

در این راهنما پیوندهای بسیاری وجود داشت که من در قالب pdf نیز آنها را حفظ نموده‌ام. بعضی از آنها ارجاع به مبحثی در محل دیگری از این راهنما هستند که مشکلی ندارند، و برخی از آنها پیوند به صفحه‌ای در وب می‌باشند، اما اکثراً پیوند به صفحه‌ای در همان وبلاگ فارسی مخصوص این راهنما می‌باشند.

چون وبلاگ فوق در حال حاضر تکمیل نگردیده است، لذا در حال حاضر(زمان انتشار این فایل) اکثر پیوندها به آن صفحات کار نمی‌کند.

من در حال ترجمه صفحات هستم، و به تدریج به وبلاگ افزوده می‌شوند. امیدوارم تا اواخر بهار آینده(همانطور که در آنجا اشاره نموده‌ام تعداد صفحات بسیار زیاد است) به تکمیل نهایی برسد.

پس از آن پیوندها در حالت متصل به شبکه می‌توانند استفاده شوند. اما احتمال دارد به صورت جداگانه، یا اگر مجال باشد همگی باهم به صورت یک مجموعه منتشر گردند.

محمود پهلوانی

مقدمه

ازشما درخواست می‌شود ، اضافه کنید ، اصلاح کنید ، به شرطی که آن‌ها را مصون از خطا نگاه دارید.لطفاً هر نمونه کدی که می‌نویسید ، آزمایش کنید .

تمام اطلاعاتی که در اینجا ارائه گردیده، بدون ضمانت و تضمین درستی می‌باشد. با مسئولیت خودتان به کاربرید. هرگاه مردد هستید، لطفاً صفحه‌های man یا info گنو را به عنوان مراجع معتبر کنکاش کنید.

در باره این راهنما

هدف این راهنما کمک به افرادی است که علاقمند به یادگیری کار با BASH می‌باشند. آرزومند است تکنیک‌های تمرینی خوبی، برای کار با BASH و نوشتن اسکریپت‌های ساده را تعلیم دهد.

این راهنما کاربران مبتدی را هدف قرار داده است. فرض بر عدم وجود دانش پیشرفته می‌نمایدـــ فقط توانایی متصل شدن به یک سیستم مبتنی بر یونیکس وبازکردن رابط خط فرمان(ترمینال) را انتظار دارد. اگر چگونگی استفاده از یک ویرایشگر متن را بدانید، کمک خواهد نمود، ما ویرایشگرها را پوشش نمی‌دهیم، انتخاب ویرایشگر خاصی را توصیه نمی‌کنیم . آشنایی با مبانی مجموعه ابزار یونیکس،یا سایر زبان‌های برنامه نویسی یا مفاهیم برنامه نویسی ضروری نیست، اماآنهایی که از این قبیل آگاهی‌ها دارند، ممکن است برخی مثال‌ها را سریع‌تر درک کنند.

اگر مطلبی برای شما مبهم می‌باشد، ازشما میخواهیم آن را گزارش کنید(از BashGuideFeedback ، یا کانال #bash در irc.freenode.org) تا امکان پالایش آن برای خوانندگان آتی فراهم گردد .

همچنین از شما دعوت می‌گردد، با توسعه و گسترش این راهنما، یا تصحیح اطلاعات نامعتبر یا ناقص آن در بهبود بخشی به آن همکاری نمایید.

نگاه دارنده اصلی این سند:

- Lhunath -- (مولف اصلی)
- GreyCat --
- ـــ مترجم: محمود پهلوانی

این راهنما همچنین به صورت PDF format در دسترس می‌باشد.به طور جایگزین، پس از رفتن به FullBashGuide موفق به چاپ هم می‌شوید. وتضمین می‌شود که همواره آخرین نگارش این سند را چاپ می‌گیرید.

یک تعریف

BASH کوته‌نوشتی برای *Bourne Again Shell* است. بر مبنای پوسته *Bourne* ، و تا حد ممکن سازگار با ویژگی‌های آن می‌باشد.

شل‌ها مفسرهای فرمان هستند. آنها برنامه‌های کاربردی می‌باشند که قابلیت صدور دستورات محاوره‌ای کاربران به سیستم عامل، یا اجرای سریع پردازشهای دسته‌ای را میسر می‌سازند. به هیچ وجه برای اجرای برنامه‌ها لازم نمی‌باشند، آنها صرفاً یک لایه بین فراخوان‌های سیستم و کاربر هستند.

شل را به مثابه روشی برای صحبت کردن با سیستم خود در نظر آورید. سیستم شما برای اکثر کارهایش احتیاجی به آن ندارد، اما شل یک رابط بسیار خوب مابین شما و آنچه سیستم می‌تواند ارائه کند، می‌باشد. شل به شما اجازه می‌دهد که حساب کنید، بررسی‌های اصلی را انجام دهید و برنامه‌های کاربردی را اجرا نمایید. و از آن مهمتر به شما اجازه می‌دهد این اعمال را با یکدیگر ترکیب کنید و برنامه‌ها را به هم متصل نموده و عملیات پیچیده را انجام دهید و وظایف را خودکار سازید.

BASH سیستم عامل شما نیست. مدیر پنجره شما نیست. ترمینال شما نیست(اما اغلب *داخل* ترمینال شما اجرا می‌شود). صفحه‌کلید یا موشواره شما را کنترل نمی‌کند. سیستم شما را پیکربندی نمی‌کند، محافظ نمایشگر را فعال نمی‌سازد، فایل‌ها را با دوبار کلیک نمودن روی آنها برایتان باز نمی‌کند. به طور کلی Bash درگیر فعال کردن برنامه‌ها از طریق مدیر پنجره یا محیط رومیزی شما نمی‌شود. این مهم است که بدانید Bash فقط رابط شما برای اجرای فرمان‌ها(با استفاده از دستور زبان آن) هم در اعلان محاوره‌ای آن و هم در اسکریپت‌های Bash می‌باشد.

در مستندات گنو: Introduction

پوسته یا شل: یک مفسر فرمان(احتمالاً محاوره‌ای)، عمل کننده به عنوان یک لایه بین کاربر و سیستم است.
BASH: یا Bourne Again Shell، یک پوسته سازگار با *Bourne* است.

کاربرد Bash

اکثر کاربران تصور می‌کنند که BASH یک اعلان و خط‌فرمان است. این BASH در *حالت محاوره‌ای* است. BASH همچنین می‌تواند در *وضعیت غیر محاوره‌ای* اجرا گردد، مانند وقتی‌که اسکریپت‌ها اجرا می‌شوند. می‌توانیم از اسکریپت‌ها برای خودکار سازی برخی وظایف استفاده کنیم. اسکریپت‌ها اساساً لیستی از فرمان‌ها می‌باشند(دقیقاً مانند همان فرمان‌هایی که می‌توانید در خط‌فرمان تایپ نمایید)، اما در یک فایل ذخیره شده‌اند. موقعی که اسکریپت اجرا می‌شود، تمام این دستورات(به طور معمول) به ترتیب یکی پس از دیگری اجرا می‌شوند.

ما با دستورات اساسی در *شل محاوره‌ای* شروع می‌کنیم. موقعی که شما با آنها مأنوس شوید، می‌توانید آنها را با هم در اسکریپت‌ها قرار بدهید.

مهم!
شما باید خودتان را با فرمان‌های `man` و `apropos` در شل آشنا و مأنوس نمایید. آنها برای خودآموزی ضروری می‌باشند.

```
$ man  man
$ man apropos
```

در این راهنما علامت $ در ابتدای یک سطر بیانگر اعلان BASH می‌باشد. به طور سنتی، یک اعلان شل به $ یا % یا # ختم می‌گردد. اگر به $ ختم شود، به آن معناست که یک شل سازگار با پوسته Bourne است(ازقبیل یک پوسته POSIX ، یا یک شل Korn ، یا BASH). اگر به علامت % ختم شود بیانگر یک *شل C* (csh یا tcsh) است، این راهنما C shell را پوشش نمی‌دهد. اگر به کاراکتر # ختم شود،

نشان دهنده آنست که شل با حساب کاربر ارشد(**root**) اجرا می‌گردد، و بایستی بسیار محتاط و دقیق باشید.

اعلان واقعی شما احتمال دارد خیلی طولانی‌تر از $ باشد. اعلان فرمان‌ها اغلب خیلی شخصی‌سازی شده هستند.

دستور `man` از "manual" می‌آید، این دستور مستندات(صفحات man نامیده شده‌اند) را در مباحث گوناگون باز می‌کند. شما آن را به صورت اجرای دستور `man` [*topic*] در اعلان BASH استفاده کنید، که [*topic*] در آن نام صفحه‌ایست که می‌خواهید آن را بخوانید. توجه داشته باشید که بسیاری از این صفحه‌ها(علیرغم ظاهر نامشان) به طور قابل ملاحظه‌ای طولانی‌تر از یک صفحه چاپ شدنی می‌باشند. هر فرمان(یا برنامه کاربردی) در سیستم شما احتمال دارد یک صفحه man داشته باشد. همچنین برای سایر مواردی از قبیل فراخوان‌های سیستمی یا فایل‌های پیکربندی خاص نیز صفحاتی موجود است. در این راهنما، ما فقط دستورات را پوشش خواهیم داد.

توجه نمایید که اگر در جستجوی اطلاعاتی در باره دستورات داخلی BASH(دستورات فراهم شده توسط خود BASH، و نه برنامه‌های خارجی) می‌باشید، باید به جای آن دستور `man` ***bash*** را به کار ببرید. مستندات BASH گسترده و مفصل است. یک مرجع ممتاز و بسیار خوب است ولو اینکه تکنیکی‌تر از این راهنمامی‌باشند.

BASH همچنین یک دستور `help` که خلاصه کوتاهی از دستورات داخلی را شامل می‌شود ارائه می‌نماید(که در آینده در مورد آنها صحبت خواهیم نمود) .

```
$ help
$ help read
```

در پرسش و پاسخ‌های متداول:
آیا لیستی از ویژگی‌های اضافه شده به یک نگارش معین Bash وجود دارد؟

حالت محاوره‌ای: حالتی از عملکرد که در آن یک اعلان در هر زمان یک دستور را از شما می‌پذیرد.

اسکریپت: یک فایل محتوی یک سلسله دستورات برای اجرا یکی پس از دیگری.

فهرست مطالب

این راهنما به بخش‌هایی تقسیم شده، که مقصود خوانده شدن آن تقریباً به همان ترتیب ارائه شده است. اگر شما با پرش از بخش معینی به جلو بروید، ممکن است با کمبود اطلاعات پیش‌زمینه‌ای بخش‌های قبلی مواجه شوید. (لینک‌هایی جهت مراجعه به بخش‌های مناسب در مباحث قبلی در همه موارد ارائه نگردیده.)

- متغیرها، پارامترهای ویژه، انواع پارامترها، بسط پارامتر.

- الگوها
  - جانشینی‌ها(Globs)، انطباق نام فایل،جانشینی‌های توسعه‌یافته، عبارت‌های منظم.
- بررسی‌ها و شرطی‌ها
  - وضعیت خروج، && و ||، test ,if و [[، while, until و for، case و select.
- آرایه‌ها
  - آرایه‌ها، آرایه‌های انجمنی.
- ورودی و خروجی
  - تغییر مسیر، here document ها, here string ها، لوله‌ها، جایگزینی پردازش.
- دستورات مرکب
  - زیرپوسته‌ها، گروه‌بندی دستورات،ارزیابی حسابی،توابع، مستعارها.
- Sourcing
  - خواندن دستورات از دیگر فایل‌ها.
- کنترل Job
- تکنیک‌ها
  - انتخاب پوسته،نقل‌قولی کردن، خوانایی، اشکالزدایی.

دستورات و شناسه‌ها

فهرست مطالب

دستورات و شناسه‌ها

BASH دستورات را از ورودی(که می‌تواند ترمینال یا یک فایل باشد) می‌خواند. با هر سطری که می‌خواند به عنوان یک *دستور* رفتار می‌کند ــ یک دستورالعمل که باید انجام بشود(چند حالت پیشرفته‌تری از قبیل دستوراتی که سطرهای چندتایی را دربرمی‌گیرند، موجود است، اما در حال حاضر نگران آنها نمی‌باشیم.)

BASH هر سطر را از محل هرکاراکتر فضای سفید(فاصله و Tab) به *کلمه‌ها* تقسیم می‌کند. اولین کلمه‌ای که پیدا می‌کند نام دستوری است که باید اجرا شود. تمام کلمات باقی‌مانده *شناسه‌ها* برای دستور تلقی می‌گردند(گزینه‌ها، نام فایل‌ها و غیره).

فرض کنیم شما در یک دایرکتوری خالی هستید. (اگر می‌خواهید این کد را امتحان کنید می‌توانید دایرکتوری به نام test را با اجرای: `mkdir test ; cd test` ایجاد کنید و داخل آن بروید).

```
$ ls            #    لیست فایل‌ها در دایرکتوری جاری( بدون خروجی: فایلی موجود نیست).
$ touch a b c   #    فایل‌های 'a' و 'b' و 'c' را ایجاد می‌کند.
$ ls            #    دوباره لیست فایل‌ها، این دفعه در خروجی سه فایل ایجاد شده نمایش داده می‌شود.
  a  b  c
```

touch یک برنامه کاربردی است که زمان *آخرین ویرایش* یک فایل معین را به زمان حال تغییر می‌دهد. اگر فایل نامبرده در حال حاضر موجود نباشد، به سادگی آن را، به عنوان یک فایل جدید خالی ایجاد می‌کند . در این مثال، ما سه شناسه به آن داده‌ایم. touch برای هر یک از آنها یک فایل ایجاد می‌کند. فرمان ls سه فایل ایجاد شده را به ما نشان می‌دهد.

```
$ rm *          #    تمام فایل‌های دایرکتوری فعلی را حذف می‌کند
$ ls
$               #    لیست فایل‌های دایرکتوری جاری(فایل موجود نیست)
$ touch a  b  c #    فایل‌های a و b و c را ایجاد می‌کند
```

```
$ ls                    # دوباره لیست فایل‌ها، این دفعه فایل‌های a و b و c در خروجی نمایش داده می‌شود.
a  b  c
```

rm ابزاری برای حذف فایل‌هایی می‌باشد که به عنوان شناسه به آن داده شده‌اند. * یک جانشین (glob) می‌باشد. و در اینجا در اصل به معنای *تمام فایل‌های موجود در دایرکتوری جاری* می‌باشد. بعداً بیشتر درباره جانشین‌ها (globs) خواهید خواند.

حالا توجه نمودید که بین *a* و *b* و همچنین مابین *b* و *c* چند تا فاصله وجود داشت؟ همین‌طور نیز متوجه شدید که نتیجه، یعنی فایل‌های ایجاد شده توسط touch با دفعه قبل که فقط یک فاصله بین آنها بود تفاوتی ندارد. حال آگاه شدید که تعداد فضاهای سفید بین شناسه‌ها اهمیت ندارد. دانستن این مطلب با اهمیت است. برای مثال:

```
$ echo This is a test.
This is a test.
$ echo This    is    a    test.
This is a test.
```

echo دستوری است که شناسه‌هایش را در *خروجی استاندارد* (که در این حالت ترمینال است) می‌نویسد. در این مثال، ما دستور echo را با چهار شناسه به کار برده‌ایم: 'This'و 'is'و 'a' و 'test.' که فرمان echo این شناسه‌ها را می‌گیرد، وآنها را یک به یک با یک فاصله مابین آنها در خروجی چاپ می‌کند. در حالت دوم دقیقاً همان اتفاق رخ می‌دهد. فاصله‌های اضافی تفاوتی ایجاد نمی‌کنند. اگر واقعاً فاصله‌های اضافی را خواسته باشیم، لازم است جمله را به صورت یک شناسه منفرد به آن بدهیم. می‌توانیم این کار را با استفاده از نقل‌قول‌ها انجام بدهیم:

```
$ echo "This    is    a    test."
This    is    a    test.
```

نقل‌قول‌ها هر چیز در داخلشان را به صورت یک شناسه منفرد بسته‌بندی می‌کنند. این شناسه منفرد 'This is a test.' می‌باشد، با فاصله‌های صحیح. توجه داشته باشید که نقل‌قول‌ها بخشی از شناسه نیستند، BASH قبل از تحویل شناسه به فرمان echo آنها را حذف می‌کند. echoاین شناسه منفرد را درخروجی چاپ می‌کند، درست مانند آنچه همیشه انجام می‌دهد

در اجتناب از موارد ذیل دقیق باشید:

```
$ ls                                    # لیست فایل‌های دایرکتوری جاری.
The secret voice in your head.mp3  secret
$ rm The secret voice in your head.mp3  # فرمان rm را با 6 شناسه اجرا می‌کند، نه با یکی
rm: cannot remove `The': No such file or directory
rm: cannot remove `voice': No such file or directory
rm: cannot remove `in': No such file or directory
rm: cannot remove `your': No such file or directory
```

```
rm: cannot remove `head.mp3': No such file or directory
$ ls                                    # لیست فایل‌های دایرکتوری جاری: فایل مورد نظر هنوز آنجاست
The secret voice in your head.mp3       # اما فایل secret شما حالا دیگر نیست
```

شما باید در محصور کردن صحیح نام فایل با علائم نقل‌قول دقیق باشید. اگر محتاط نباشید، فایل‌های اشتباهی را حذف خواهید نمود! فرمان rm نام فایل‌ها را به عنوان شناسه می‌پذیرد. اگر نام‌هایی که در آنها کاراکتر فاصله وجود دارد را نقل‌قولی نکنید، rm فکر می‌کند که هر شناسه یک نام فایل است. نظر به اینکه BASH شناسه‌ها را از محل فضای سفید تفکیک می‌کند، rm سعی خواهد نمود هر کلمه را به طور جداگانه حذف کند. در مثال فوق، به جای نام فایل صوتی، سعی کرده هر یک از کلمات موجود در نام فایل مورد نظر را حذف نماید. این موجب گردیده فایل *secret* ما حذف گردد، و فایل صوتی را پشت سر باقی بگذارد!

شناسه‌ها از نام دستور و از یکدیگر نسبت به فضای سفید(فاصله، Tab، و سطر جدید) جدا می‌شوند. به خاطر سپردن این مطلب اهمیت دارد، برای مثال، مورد زیر اشتباه است:

```
$ [-f file]
```

شما می‌خواهید دستور] از شناسه‌های **-f** *file* و [جدا شود. اگر شما] و **-f** را با فاصله از یکدیگر جدا نکنید، BASH تصور می‌کند شما می‌خواهید دستوری به نام [-f] را اجرا کنید، و در PATH شما برای برنامه‌ای با نام [-f] جستجو می‌کند. علاوه بر این، شناسه *file* و [نیز لازم است با فاصله جدا شوند. فرمان] انتظار دارد آخرین شناسه‌اش، [باشد. در دستور صحیح همه شناسه‌ها با فاصله از یکدیگر جدا می‌شوند:

```
$ [ -f file ]
```

(ما فرمان] را در آینده مفصل‌تر شرح خواهیم داد. افراد زیادی را دیده‌ایم که با آن سر در گم شده‌اند، و گمان برده‌اند که می‌توان از فضای سفید میان آن و شناسه‌هایش صرفنظر نمود، بنابراین، **لازم** دانستیم خیلی زود در این مثال خاص آن را مطرح کنیم.)

و البته اگر نام فایل شما شامل فضای سفید و یا دیگر کاراکترهای خاص باشد، باید نقل‌قولی بشود:

```
$ [ -f "my file" ]
```

توجه: لطفاً اگر هنوز مطلب برای شما خیلی واضح نیست، از شناسه‌ها، نقل‌قول‌ها، تفکیک کلمه و http://wiki.bash-hackers.org/syntax/words دید خوبی در این موارد حاصل نمایید. این مهم است که شما قبل از ادامه این راهنما، درک مناسبی از اینکه شل دستوراتی را که به آن می‌دهید چگونه تفسیر می‌کند، داشته باشید.

تکرار مفید:

شما باید ***همواره*** جمله‌ها و رشته‌هایی که مرتبط با یکدیگر هستند، حتی اگر به طور قطعی ضرورت ندارد، را نقل‌قولی کنید. این شما را گوش به زنگ نگاه می‌دارد و ریسک خطای انسانی در اسکریپت‌ها را کاهش می‌دهد.

برای مثال، همیشه باید شناسه‌های فرمان echo را نقل‌قولی کنید.

در پرسش و پاسخ‌های رایج:
من سعی می‌کنم یک فرمان را در متغیری قرار دهم، اما همیشه در حالت‌های پیچیده‌تر شکست می‌خورم!
چگونه می‌توانم شناسه‌های(گزینه‌های) خط‌فرمان را به آسانی مدیریت کنم؟

شناسه‌ها: اینها کلمات اختیاری اضافه‌ای هستند که می‌توانید موقع اجرای فرمان‌ها تعیین کنید. آنها پس از نام فرمان‌ها آورده می‌شوند('ls -l *foo*' فرمان `ls` را با دو شناسه اجرا می‌کند).

نقل‌قول‌ها: دو نوع نقل‌قول(' و ") برای محافظت از کاراکترهای خاص معین در داخل عبارت نقل‌قولی، از تفسیر شدن به عنوان کاراکتر خاص توسط BASH، استفاده می‌کنیم. تفاوت بین ' و " را بعداً صحبت خواهیم نمود.

انواع دستورات

BASH چند نوع مختلف از دستورات را شناسایی می‌کند: مستعارها، توابع، دستورات داخلی، کلمات کلیدی، و اجرایی‌ها.

- مستعارها: مستعارها روشی برای کوتاه نمودن دستورات است. اینها فقط در پوسته‌های محاوره‌ای به کار می‌روند، نه در اسکریپت‌ها. (این یکی از **معدود** تفاوتهای یک اسکریپت و یک پوسته محاوره‌ای می‌باشد.) مستعار یک *نام* کوتاه شده برای یک *رشته* معین است. موقعی که آن *نام* به عنوان یک دستور استفاده شود، قبل از اینکه فرمان اجرا گردد، نام با *رشته* جایگزین می‌گردد. بنابراین به جای آن اجرا می‌گردد:

```
$ nmap -P0 -A --osscan_limit 192.168.0.1
```

می‌توانید از یک مستعار به این شکل استفاده کنید:

```
$ alias nmapp='nmap -P0 -A --osscan_limit'
$ nmapp 192.168.0.1
```

قدرت مستعارها محدود است، جایگزینی فقط در اولین کلمه رخ می‌دهد. اگر قابلیت انعطاف بیشتری می‌خواهید، از یک تابع استفاده کنید. مستعارها فقط به عنوان میانبرهای متنی مفید می‌باشند.

- **توابع**: توابع در BASH چیزی مشابه مستعارها هستند، اما قدرتمندتر. بر خلاف مستعارها، توابع می‌توانند در اسکریپت‌ها به کار بروند. یک تابع محتوی فرمان‌های پوسته است، بسیار زیاد مانند یک اسکریپت کوچک، حتی می‌توانند شناسه‌ها را قبول کنند و متغیرهای محلی ایجاد کنند. وقتی یک تابع فراخوانده می‌شود، دستورات داخل آن اجرا می‌شوند. توابع به طور کامل‌تری بعداً در این راهنما پوشش داده می‌شوند.

- دستورات داخلی: BASH تعدادی دستور اصلی در ساختمان خودش دارد، از قبیل `cd` (تغییر دایرکتوری)، `echo` (نوشتن خروجی)، وغیره. می‌توانید آنها را به عنوان توابعی که از قبل فراهم شده‌اند در نظر بگیرید.

- کلمات کلیدی: کلمه‌های کلیدی کاملاً مانند دستورات داخلی می‌باشند، فقط اختلاف اصلی آنها درقواعد تجزیه ویژه در نظر گرفته شده برای آنها می‌باشد. برای مثال، `[` یک دستور داخلی است، در حالیکه `[[` یک کلمه کلیدی است. هر دو برای بررسی موضوعات به کار می‌روند، اما چون `[[` یک کلمه کلیدی است تا فرمان داخلی، این کلمه کلیدی از چند قاعده ویژه تجزیه سود می‌برد که کاربرد

آن را آسانتر می‌سازد:

```
$ [ a < b ]
-bash: b: No such file or directory
$ [[ a < b ]]
```

مثال اول یک خطا باز می‌گرداند، زیرا bash سعی می‌کند فایل *b* را به دستور `[ a ]` تغییر مسیر بدهد(بخش تغییر مسیر فایل را ملاحظه کنید) مثال دوم به طور واقعی آنچه شما انتظار دارید انجام می‌دهد. کاراکتر `<` وقتی که با یک فرمان `[[` به کار می‌رود، دیگر معنی خاص عملگرتغییر مسیر را ندارد.

- اجرایی‌ها: آخرین نوع دستوری که می‌تواند توسط bash اجرا بشود، یک دستور *اجرایی* است، که همچنین یک *دستور خارجی* یا *برنامه‌کاربردی* نیز نامیده می‌شود. اجرایی‌ها با استفاده از نام مسیر فراخوانی می‌شوند. اگر فایل اجرایی در دایرکتوری جاری است، به صورت `./myprogram` به کار ببرید. اگر در دایرکتوری /usr/local/bin است، از */usr/local/bin/*`myprogram` استفاده کنید.

برای اندکی آسان‌تر نمودن زندگی شما، به‌هرحال، BASH از متغیری استفاده می‌کند، که بگوید در مواقعی که شما فقط نام برنامه کاربردی را می‌دانید و نه نام مسیرکامل آنرا، در کجا برنامه را پیدا کند. این متغیر PATH نام دارد، ومجموعه‌ای از نام مسیرهای دایرکتوری است که با کاراکتر کولن از هم جدا شده‌اند-- به عنوان مثال /bin:/usr/bin موقعی که یک فرمان بدون نام مسیر در BASH مشخص می‌شود (مثل `myprgram`، یا `ls`)، و یک مستعار، تابع، کلمه کلیدی یا دستور داخلی نباشد BASH در میان دایرکتوری‌های PATH به ترتیب از چپ به راست آنرا جستجو می‌کند، که ببیند آیا آنها دارای فایل اجرایی با نامی که تایپ کرده‌اید، هستند.

نکته:
می‌توانید از فرمان `type` برای پی‌بردن به نوع یک فرمان استفاده کنید.
به عنوان مثال:

```
$ type rm
rm is hashed (/bin/rm)
$ type cd
cd is a shell builtin
```

در مستندات گنو: دستورات ساده

در پرسش و پاسخ‌های رایج:
تفاوت بین `test` و `]` و `]]` چیست؟
چگونه می‌توانم مستعاری ایجاد کنم که یک شناسه دریافت کند؟

مستعارها: یک نام که به یک رشته نسبت داده شده است. موقعی که نام به عنوان یک فرمان به کار برود، با رشته نسبت داده شده

تعویض می‌گردد.

تابع: یک نام که به یک مجموعه از دستورات نسبت داده شده است . وقتی که نام به عنوان یک دستور استفاده شود، تابع با شناسه‌های به کار رفته در خط‌فرمان ، فراخوانی می‌شود. توابع روش اصلی برای ایجاد فرمان‌های جدید هستند.

دستورات داخلی: فرمان‌های معینی که در داخل BASH ساخته شده‌اند. اینها موقعی که در خط‌فرمان اجرا شوند به طور داخلی اجرا می‌شوند(و پردازش جدیدی ایجاد نمی‌کنند).

برنامه‌های کاربردی: یک برنامه که می‌تواند با ارجاع به نام مسیر آن اجرا شود(/bin/ls)، و یا اگر محل استقرار آن در متغیر PATH شما وجود دارد، با نام آن اجرا گردد.

اسکریپت ها

یک اسکریپت اساساً یک سلسله دستورات در داخل یک فایل است. BASH فایل را می‌خواند و دستورات را به ترتیب پردازش می‌کند. فقط وقتی به دستور بعدی می‌رود که اجرای دستور فعلی به پایان رسیده باشد، مگر اینکه دستور جاری به طور غیر همزمان اجرا گردیده باشد(در پس‌زمینه). در مورد حالت اخیر خیلی نگران نباشید، در آینده بیشتر در باره آن خواهید آموخت.

در واقع هر مثالی که در این راهنما ملاحظه می‌کنید، دقیقاً می‌تواند به همان خوبی خط‌فرمان در اسکریپت‌ها به کار برده شود.

ساخت یک اسکریپت آسان است. شما فقط یک فایل جدید بسازید و جمله زیر را در ابتدای آن قرار دهید:

```
#!/usr/bin/env bash
```

این سرآیند اطمینان ایجاد می‌کند که هرگاه اسکریپت اجرا شود، BASH به عنوان مفسر آن به کار خواهد رفت. (روشهای معتبر دیگری نیز برای انجام این امر وجود دارد.) کاراکترهای #! باید در ابتدایی‌ترین نقطه فایل، بدون هر گونه فاصله یا سطری قبل از آنها قرار گیرند. دستورات اسکریپت شما همگی در سطرهای زیر این سطر ظاهر می‌شوند.

لطفاً با مثال‌های موجود در اینترنب که از /bin/sh به عنوان مفسر استفاده می‌کنند فریب نخورید. sh همان bash نیست. هرچند که، دستور زبان sh و bash خیلی همسان به نظر برسند، و ولواینکه اکثر اسکریپت‌های bash در پوسته sh اجرا خواهند شد، مقدار زیادی از مثال‌های این راهنما فقط برای bash تهیه شده‌اند و در sh در می‌مانند و یا رفتار غیر منتظره‌ای خواهند داشت.

همچنین، خواهشمند است از دادن پسوند گمراه کننده sh. به اسکریپت‌هایتان خودداری نمایید. چون مقصود را برآورده نمی‌کند و کاملاً گمراه کننده است(چون قرار است اسکریپت bash باشد، نه یک اسکریپت sh).

و در ضمن اگر از *Windows* برای نوشتن اسکریپت‌هایتان استفاده می‌کنید، در هرحال از کاربرد *Notepad* برای نوشتن اسکریپت اجتناب نمایید. *Microsoft Notepad* فقط می‌تواند فایل‌های با انتهای سطر به روش DOS ایجاد نماید. یعنی هر سطری که در notepad ایجاد می‌کنید با دو کاراکتر خاتمه می‌یابد: یک کاراکتر *رفتن سر سطر* و یک کاراکتر *سطرجدید*. BASH فقط سطرهایی که به یک کاراکتر *سطرجدید* خاتمه می‌یابند را می‌خواند. در نتیجه، کاراکتر *رفتن سرسطر Carriage Return* وقتی که نمی‌دانید آنجا هست باعث دردسر باور نکردنی شما خواهد شد(پیغام خطاهای خیلی مرموز). در هر حالت از *ویرایشگر محبوبی* مانند Vim و Emacs و kate و GEdit و GVIM یا xemacs استفاده کنید. اگر چنین نکنید، بعد قبل از اجرای اسکریپت لازم است تمام کاراکترهای CR(رفتن سرسطر) را پاک کنید.

وقتی که اسکریپت شما آماده شد، می‌توانید آن را به صورت زیر اجرا کنید:

```
$ bash myscript
```

در این مثال، ما BASH را اجرا کرده و به او می‌گوییم اسکریپت ما را بخواند، سطر #! فقط یک توضیح است. BASH به هیچ وجه کاری با آن نمی‌کند.

به طور جایگزین می‌توانید به اسکریپت خود مجوزهای اجرا بدهید. موقعی که این کار را انجام دهید، به جای فراخوانی BASH به طور دستی، می‌توانید اسکریپت را به طور واقعی به عنوان یک برنامه کاربردی اجرا نمایید:

```
$ chmod +x myscript
$ ./myscript
```

موقعی که اسکریپت به این روش اجرا می‌شود، سطر #! به سیستم عامل (OS) می‌گوید، که از چه مفسری استفاده کند. سیستم عامل /usr/bin/env را برای اجرای bash به کار می‌برد، و آن هم اسکریپت ما را می‌خواند. BASH خودش از سطر سرآیند # چشم‌پوشی می‌کند.

بعضی‌ها مایل هستند اسکریپت‌های خود را در یک دایرکتوری شخصی نگهداری نمایند. دیگرانی مایل می‌باشند اسکریپت‌هایشان را در محلی که در متغیر PATH ذکر شده نگهداری کنند، اکثراًهر دو را با هم انجام می‌دهند. آنچه من به شما پیشنهاد می‌دهم چنین است:

```
$ mkdir -p "$HOME/bin"
$ echo 'PATH="$HOME/bin:$PATH"' >> "$HOME/.bashrc"
$ exec bash
```

دستور اول، دایرکتوری با نام bin را در دایرکتوری خانگی شما ایجاد می‌کند. یک سنت است که دایرکتوری‌های شامل دستورات bin نامیده شوند، حتی وقتی‌که آن دستورات اسکریپت هستند نه برنامه‌های ترجمه شده به زبان ماشین ("*باینری*"). با دستور دوم یک سطر به فایل .bashrc شما اضافه می‌شود، که دایرکتوری ایجاد شده را به ابتدای متغیر PATH اضافه می‌کند. حالا هر اجرای جدیدی از BASH برای یافتن اسکریپت‌های اجرایی، دایرکتوری bin شما را بررسی می‌کند. سرانجام، سطر سوم اجرای فعلی BASH ما را با یک نمونه جدید تعویض می‌کند، که فایل bashrc. جدید را می‌خواند.

تغییرات در DotFiles (از قبیل bashrc.) هرگز بلافاصله تأثیر نمی‌کنند.باید برخی اقدام‌ها را برای خواندن مجدد فایل انجام دهید. در مثال فوق، از exec *bash* برای تعویض پوسته در حال اجرا استفاده نمودیم. اگر خواسته باشید، می‌توانید ترمینال باز را بسته وترمینال جدیدی باز کنید. در این حالت BASH دوباره با خواندن فایل bashrc.(و احتمالاً سایر فایل‌ها) خودش را مقدار دهی مجدد می‌کند. یا، می‌توانید فقط دستور ذکر شده را در خط‌فرمان به کار ببرید (PATH="$HOME/bin:$PATH") و یا فایل bashrc. به طور دستی در پوسته جاری با استفاده از source "$HOME/.bashrc" اجرا نمایید.

در هر حال، اکنون می‌توانیم اسکریپت خودمان را در دایرکتوری bin خود قرار داده و به صورت یک دستور معمولی اجرایش کنیم(حالا دیگر احتیاجی نیست که در ابتدای نام اسکریپت بخش ./ را مانند مثال قبلی اضافه کنیم):

```
$ mv myscript "$HOME/bin"
```

```
$ myscript
```

نکته:
وقتی شما در سرآیند اسکریپت نوع مفسر را تعیین می‌کنید، همچنین می‌توانید زمان کوتاهی را برای شرح دادن عملکرد اسکریپت و شناسه‌های مورد نیاز آن صرف کنید:

```
#!/usr/bin/env bash
#
#   شناسه ها  نام اسکریپت
#
# .توضیح کوتاهی در تشریح هدف اسکریپت
#
# Copyright [date], [name]
```

نکته:
می‌توانید سرآیند را با یک کلمه از شناسه‌های اختیاری که می‌خواهید به مفسر ارسال کنید به کار ببرید. به عنوان مثال، شناسه‌های زیر اشکالزدایی تفصیلی را فعال می‌کنند:

```
#!/bin/bash -xv
```

لیکن دانستن آنکه Bash کجا نصب گردیده، لازم است، و همیشه هم در /bin نیست. متأسفانه نمی‌توانید از این یکی استفاده کنید:

```
#!/usr/bin/env bash -xv
```

به علت آنکه برای آن، دو کلمه در سرآیند لازم است. یونیکس آن را اجازه نمی‌دهد.می‌توانید به جای آن این‌طور بنویسید:

```
#!/usr/bin/env bash
set -xv
```

برای جزئیات بیشتر بخش اشکال‌زدایی را ملاحظه کنید.

سرآیند: سرآیند یک اسکریپت برنامه کاربردی که به عنوان مفسر عمل خواهد نمود، را تعیین می‌کند(مانند bash و sh و perl و ...). به طور مصطلح یک *shebang* نیز نامیده می‌شود-- یک عبارت عامیانه که از ترکیب *hash* (#) و *bang*(!) می‌آید. شاید ببینید که کلمه *shebang* خیلی بیشتر از *header* استفاده می‌شود، به‌ویژه چون *سرآیند* چند معنی دیگر در زمینه‌های متفاوت

دارد، حال آنکه *shebang* فقط یک معنی دارد.

کاراکترهای خاص

تعدادی کاراکتر ویژه، که معنای غیر لفظی دارند، در BASH وجود دارد، موقعی که این کاراکترها را استفاده می‌کنیم، BASH این کاراکترها و معنی خاص آنها را ارزیابی می‌کند، اما به طور معمول، آنها را به دستورات عبور نمی‌دهد. اینها را *فوق کاراکترها* نیز نامیده‌اند.

در اینجا تعدادی از این کاراکترها و عملی که انجام می‌دهند آمده است:

- [*فضای سفید*]: فضای سفید(فاصله‌ها، Tabها، و سطرجدید). BASH از فضای سفید برای اینکه یک کلمه از کجا شروع می‌شود و در کجا تمام می‌شود استفاده می‌کند. اولین کلمه از هر فرمان به عنوان نام دستور تلقی می‌گردد، و هر کلمه اضافه دیگر، به منزله شناسه برای آن دستور می‌باشد.
- $: کاراکتر بسط. این کاراکتر در اکثر جایگزینی‌ها از جمله بسط پارامتر(جایگزینی متغیر) استفاده می‌شود. بعداً بیشتر در باره آن خواهیم گفت.
- 'متن': نقل‌قول تکی متن درونش را از هرگونه بسط توسط شل محافظت نموده و از شکسته شدن آن به کلمات یا شناسه‌های چندگانه ممانعت می‌نماید. این نقل‌قول‌ها همچنین از معانی خاص کاراکترهای ویژه داخل نقل‌قول پیش‌گیری می‌کنند.
- "متن": نقل‌قول دوگانه، متن درونش را از شکسته شدن به کلمات و شناسه‌های چندگانه محافظت می‌کنند، اما انجام جایگزینی را اجازه می‌دهند. آنها از معنای ویژه *اکثر* کاراکترهای خاص در درون نقل‌قول،-- اساساً همه غیر ازکاراکتر $-- جلوگیری می‌کنند.
- #: کاراکتر توضیح. هر کلمه شروع شده با کاراکتر # شروع یک *توضیح* خواهد بود که تا سطر جدید ادامه خواهد یافت. توضیحات توسط پوسته پردازش نمی‌شوند.
- ;: جدا کننده دستور. سمی‌کالن برای جدا کردن دستورات چندتایی از یکدیگر، در صورت انتخاب کاربر برای در یک سطر قرار دادن آنها، به کار می‌رود. *اساساً* چیزی مانند سطر جدید است.
- \: کاراکتر گریز. ممیز برعکس(backslash) از اینکه کاراکتر پس از آن به هر طریق ویژه‌ای به کار رود پیش‌گیری می‌کند. این خاصیت در نقل‌قول دوگانه عمل می‌کند اما در نقل‌قول تکی خیر.
- ~: علامت مد یک میانبر برای دایرکتوری خانگی شما می‌باشد. خودش به تنهایی، یا وقتی که با یک / دنبال می‌شود، این کاراکتر معادل$HOME می‌باشد. موقعی که با یک نام کاربری دنبال شود، به معنای دایرکتوری خانگی *آن کاربر* خواهد بود. مثالها:

```
cd ~john/bin; cp coolscript ~/bin
```

- < یا >: کاراکترهای تغییر مسیر. این کاراکترها برای اصلاح(تغییر جهت) ورودی و یا خروجی یک فرمان به کار می‌روند. تغییر مسیرها بعداًپوشش داده می‌شوند.
- |: خط‌لوله‌ها ارسال خروجی یک دستور به عنوان ورودی به یک دستور دیگر را فراهم می‌کنند.
- [[عبارت]]: بررسی عبارت. یک عبارت شرطی را به صورت منطقی برای تعیین آنکه آیا صحیح است یا غلط ارزیابی می‌کند.
- { commands; } : گروه‌بندی فرمانها. با دستورات داخل ابروها مانند آنکه یک دستور واحد باشند رفتارمی‌شود. برای جاهایی که دستور زبان BASH فقط یک دستور را لازم دارد، و شما حس نمی‌کنید که یک تابع آن را برآورده سازد، ارائه گردیده.

- `دستور` و $(command) : جایگزینی دستور (حالت دوم به مراتب ارجحتر می‌باشد.) جایگزینی فرمان اول دستور داخل علامت‌ها را اجرا می‌کند، و سپس تمامی `...` یا $(...) را با خروجی استاندارد دستور تعویض می‌نماید.
- (دستور) : اجرای زیر پوسته. این به جای پوسته فعلی دستور را در یک پوسته bash جدید اجرا می‌کند. اگر این دستور موجب اثرات جانبی(مانند تغییر متغیرها) بشود، این تغییرات در پوسته جاری تأثیر نمی‌کنند.
- ((*عبارت*)): دستور محاسبات. عملگرهای داخل پرانتزها از قبیل + و - و * و / به عنوان عملگرهای ریاضی در نظر گرفته می‌شوند. این ساختار می‌تواند برای تخصیص‌هایی مثل ((*a*=$*b*+7)) و نیز بررسی‌هایی مثل ((*$a* < *$b*)) به کار برود. بعداً if بیشتر توصیح می‌دهیم.
- $((*expression*)): جایگزینی حسابی. نظیر مورد فوق، اما در اینجا نتایج ارزیابی حسابی جایگزین می‌گردد. مثال:

```
echo "The average is $(( ($a+$b)/2 ))".
```

چند مثال:

```
$ echo "I am $LOGNAME"
I am lhunath
$ echo 'I am $LOGNAME'
I am $LOGNAME
$ # boo
$ echo An open\ \ \ space
An open   space
$ echo "My computer is $(hostname)"
My computer is Lyndir
$ echo boo > file
$ echo $(( 5 + 5 ))
10
$ (( 5 > 0 )) && echo "Five is bigger than zero."
Five is bigger than zero.
```

در مستندات گنو: Shell Syntax

کاراکترهای ویژه: کاراکترهایی که برای BASH معنی خاصی دارند. به طور معمول معنی آنها تفسیر می‌شود و بعد قبل از اجرای فرمان از آن حذف می‌گردند.

پارامترها

فهرست مطالب

پارامترها

پارامترها محلهای نام‌گذاری شده در حافظه هستند که شما می‌توانید داده‌ها را در آنجا ذخیره کنید، آنها به طور معمول داده رشته‌ای را ذخیره می‌کنند، اما می‌توانند برای ذخیره آرایه‌ها یا اعداد صحیح نیز استفاده شوند.

پارامترها دو دسته هستند: *متغیرها* و *پارامترهای خاص*. پارامترهای خاص فقط‌خواندنی هستند، توسط پوسته مقداردهی اولیه می‌شوند، و برای تعامل با برخی وضعیت‌های داخلی استفاده می‌شوند. متغیرها پارامترهایی هستند، که خودتان می‌توانید آنها را ایجاد و به‌هنگام سازی نمایید. نام متغیرها مطابق قواعد زیر محدود می‌گردد:

> *نام*: یک کلمه فقط متشکل ازحروف، ارقام، و خطزیر(runderscore) است، که با یک حرف یا یک خطزیر شروع بشود. همچنین به عنوان یک *شناسه* به آن رجوع می‌شود.

برای ذخیره داده در یک متغیر، از ترکیب دستوری *تخصیص* به شکل زیر استفاده می‌کنیم:

```
$ varname=vardata
```

این دستور مقدار vardata را به متغیری به نام varname اختصاص می‌دهد.

لطفاً توجه نمایید که *نمی‌توانید* از فاصله در اطراف علامت تخصیص = استفاده کنید. اگر این را بنویسید:

```
# This is wrong!
$ varname = vardata
```

BASH متوجه نخواهد شد که سعی می‌کنید یک تخصیص انجام دهید. تفکیک کننده varname را بدون = می‌بیند و با آن همچون نام یک فرمان رفتار می‌کند، و بعد هم = و vardata را به عنوان شناسه‌های آن عبور می‌دهد.

برای دستیابی به محتوای متغیرها، از بسط پارامتر استفاده می کنیم. یعنی *جایگزینی* پارامتر ذکر شده با مقدار آن، ترکیب به کار رفته به bash می‌گوید که شما می‌خواهید محتویات متغیر را به کار ببرید. پس از آن، BASH *می‌تواند دستکاری‌های اضافه را روی نتایج انجام بدهد.*

درک این مفهوم به طورصحیح، بسیار با اهمیت می‌باشد، زیرا خیلی متفاوت با رفتار سایر زبانهای برنامه‌نویسی با متغیرها است!

برای تشریح آنکه بسط پارامتر چیست، بیایید از مثال استفاده کنیم:

```
$ foo=bar
$ echo "Foo is $foo"
```

وقتی Bash می‌خواهد کد شما را اجرا کند، اول دستور شما را با گرفتن نتیجه بسط پارامتر($foo)، و تعویض آن با محتوای foo، که bar است، تغییر می‌دهد. دستور اینطور می‌شود:

```
$ echo "Foo is bar"
```

اکنون Bash آماده اجرای فرمان است. اجرای آن یک جمله ساده در صفحه نمایش نشان می‌دهد.

```
Foo is bar
```

اهمیت دارد بدانیم که بسط پارامتر موجب می‌شود که $parameter با محتوای آن تعویض گردد، به علت حالت زیر که به استناد فهم مبحث تفکیک شناسه در فصل قبل می‌باشد :

```
$ song="My song.mp3"
$ rm $song
rm: My: No such file or directory
rm: song.mp3: No such file or directory
```

چرا این عمل نمی‌کند؟ به علت آنکه Bash، جمله $song شما را با محتوای متغیر تعویض نموده، که My song.mp3 شده است، سپس تفکیک کلمه را انجام داده، و فقط پس از آن دستور را اجرا کرده است. و این مانند آن است که تایپ کرده باشید:

```
$ rm My song.mp3
```

و بر اساس قواعد تفکیک کلمه، Bash گمان می‌برد که منظور شما *My* و *song.mp3* به معنی دو فایل مختلف است، زیرا بین آنها فضای سفید وجود دارد و نقل‌قولی هم نشده است. چطور آن را رفع کنیم؟ به خاطر بسپاریم که در اطراف هر بسط پارامتر، نقل‌قول دوگانه را قرار بدهیم!

```
$ rm "$song"
```

پارامترها: پارامترها داده‌هایی که می‌توانند به واسطه یک نشانه یا نام بازیابی بشوند را ذخیره می‌کنند.

پارامترهای خاص و متغیرها

اجازه دهید قبل از اقدام واقعی واژگان به کار رفته را مرتب کنیم. *پارامترها* و *متغیرها* را داریم. متغیرها در واقع فقط نوعی از پارامترها می‌باشند: پارامترهایی که با یک نام مشخص می‌شوند. آن پارامترهایی که متغیر نیستند، *پارامترهای خاص* نامیده شده‌اند. اطمینان دارم که با چند مثال بهتر متوجه خواهید شد:

```
$ # Some parameters that aren't variables:
$ echo "My shell is $0, and has these options set: $-"
My shell is -bash, and has these options set: himB
$ # Some parameters that ARE variables:
$ echo "I am $LOGNAME, and I live at $HOME."
I am lhunath, and I live at /home/lhunath.
```

لطفاً توجه نمایید: برخلاف PHP و Perl ...پارامترها با علامت $ شروع نمی‌شوند.علامت $ که شما در مثال مشاهده می‌کنید، صرفاً موجب می‌شود که پارامتر ذکر شده بعد از آن، *بسط* داده شود. بسط اساساً به معنای آنست که پوسته پارامتر را با محتوای آن تعویض می‌کند. به این ترتیب، LOGNAME پارامتری(متغیری) است که محتوای آن نام کاربری شماست. $LOGNAME عبارتی است که با محتوای آن متغیر تعویض خواهد شد، که در این حالت lhunath است.

گمان می‌کنم حالا، مقصود را دریافته‌اید. در اینجا خلاصه‌ای از اکثر *پارامترهای ویژه*:

- **0:** محتوی نام یا مسیر اسکریپت است(این در همه حال صدق نمی‌کند.)
- **پارامترهای موضعی(مکانی):** 1, 2, 3 ...، اینها محتوی شناسه‌هایی می‌باشند که ما به اسکریپت یا تابع جاری می‌دهیم.
- *: به همه کلمات تمام پارامترهای موضعی بسط می‌یابد. اگر نقل‌قول دوگانه بشود، به یک رشته منفرد شامل تمام پارامترهای موضعی بسط می‌یابد، که با اولین کاراکتر متغیر IFS (که بعد در باره‌اش صحبت می‌کنیم)، از یکدیگر جدا شده‌اند.
- @: به تمام کلمات پارامترهای موضعی بسط می‌یابد، اگر نقل‌قول دوگانه بشود، به لیستی از تمام کلمات پارامترهای موضعی به صورت کلمه‌های منفرد، بسط می‌یابد.
- #: به عدد معادل تمام پارامترهای موضعی(مکانی) ارائه شده فعلی، بسط می‌یابد.
- ?: به کد خروج آخرین فرمان تکمیل شده در پیش‌زمینه، بسط می‌یابد.
- $: به PID (شماره ID پردازش) پوسته جاری، بسط می‌یابد.
- !: به PID آخرین دستور اجرا شده در پس‌زمینه، بسط می‌یابد.
- _: به آخرین شناسه آخرین فرمانی که اجرا شده است، بسط داده می‌شود.

و در اینجا چند مثال از *متغیرهایی* که پوسته برای شما فراهم می‌کند:

- BASH_VERSION: محتوی رشته‌ایست که شماره نگارش BASH را شرح می‌دهد.

- HOSTNAME: شامل نام میزبان کامپیوتر شما می‌باشد، به شکل کوتاه یا بلند، بستگی به چگونگی تنظیمات کامپیوتر شمادارد.
- PPID: محتوی شماره شناسایی پردازش(PID) پردازش والد پوسته جاری است.
- PWD: محتوی دایرکتوری کاری جاری است.
- RANDOM: هرگاه این متغیر را بسط بدهید، یک عدد تصادفی(ساختگی) بین 0 تا 32767 تولید می‌شود.
- UID:شماره شناسایی(ID) کاربر فعلی . نامعتبر برای مقاصد امنیتی یا تأییدی، افسوس.
- COLUMNS: تعداد کاراکترهایی که می‌تواند یک سطر از ترمینال شما را پر کند(عرض ترمینال شما بر حسب کاراکتر).
- LINES: تعداد سطرهایی که ترمینال جاری شما را پر می‌کند(ارتفاع ترمینال بر حسب سطر).
- HOME: دایرکتوری خانگی کاربر فعلی.
- PATH: یک لیست از مسیرها، که با کاراکتر کولن ازیکدیگر جدا شده‌اند، و در صورتیکه یک فرمان، مستعار ، تابع، دستور داخلی، و کلمه کلیدی نباشد، و مسیر آن نیز تعیین نشده باشد، برای یافتن آن دستور جستجو می‌شوند.
- PS1: شامل رشته‌ایست که اعلان پوسته شما را تشریح می‌کند.
- TMPDIR: شامل دایرکتوری مورد استفاده پوسته برای نگهداری فایل‌های موقتی آن، می‌باشد.

(خیلی بیشتر از اینها وجود دارد-- برای لیست جامع، مستندات را ملاحظه نمایید.) البته، شما محدود به این متغیرها نمی‌باشید. هرطور که مایلید متغیرهای خودتان را تعریف کنید:

```
$ country=Canada
$ echo "I am $LOGNAME and I currently live in $country."
I am lhunath and I currently live in Canada.
```

توجه داشته باشید که ما کمیت Canada را به متغیر country اختصاص دادیم. به یاد داشته باشید که شما مجاز به قرار دادن فاصله، قبل و بعد از علامت تساوی نمی‌باشید!

```
$ language = PHP
-bash: language: command not found
$ language=PHP
$ echo "I'm far too used to $language."
I'm far too used to PHP.
```

به خاطر بیاورید که BASH پرل یا PHP نیست. شما باید خیلی مراقب چگونگی کارکرد *بسط*، جهت پرهیز از دردسر بزرگ، باشید. اگر این طور عمل نکنید، موقعیت‌های خیلی خطرناکی در اسکریپت‌هایتان خلق می‌کنید، به‌خصوص موقعی که این اشتباه با دستور rm همراه باشد:

```
$ ls
```

```
no secret  secret
$ file='no secret'
$ rm $file
rm: cannot remove `no': No such file or directory
```

فرض کنید ما دو فایل no secret و secret داریم. اولی محتوی چیز ارزشمندی نیست، اما دومی محتوی اطلاعات سری است، که جهان را از سرنوشت بد(نابودی) رهایی می‌بخشد. ملاحظه کنید که اگر نقل‌قولی کردن بسط پارامتر file را فراموش کنید. BASH پارامتر را بسط می‌دهد ونتیجه آن *no secret* rm می‌شود. BASH مطابق معمول شناسه‌ها را نسبت به فضای سفید آنها تفکیک می‌کند، و به فرمان rm دو شناسه تحویل می‌دهد: *'no'* و *'secret'*. درنتیجه موفق به یافتن فایل no نمی‌شود و اما فایل secret را حذف می‌کند. *فایل secret از دست می‌رود!*

تکرار مفید:
همواره باید بسط‌های پارامتر را به طور صحیح نقل‌قولی کنید. این امر از تفسیر فضای سفید یا جانشین‌های احتمالی در داخل آنها و دادن موی خاکستری به شما یا پاک کردن غیر منتظره فایلها از کامپیوترتان، پیش‌گیری می‌کند. تنها PE(بسط پارامتر) خوب، PE نقل‌قولی شده است.

در مستندات گنو: Shell Parameters , Shell Variables

در پرسش و پاسخ‌های رایج:
چگونه می‌توانم دو متغیر را به هم الحاق کنم؟ چگونه یک رشته را در یک متغیر اضافه(پیوست) کنم؟
چطور می‌توانم پارامترهای موضعی(مکانی) بعد از $9 را دستیابی کنم

متغیر: یک متغیر نوعی از پارامتر است که می‌توانید مستقیماً آن را ایجاد و ویرایش کنید. با یک نام مشخص می‌شود، که باید با یک حرف یاخط زیر(_)شروع بشود، باید فقط متشکل از حرف، رقم، و خط زیر باشد. نام متغیرها حساس به نوع حروف می‌باشد.

بسط: بسط موقعی رخ می‌دهد که یک پارامتربا علامت دلار قبل از نامش همراه باشد. BASH مقدار پارامتر گرفته و قبل از اجرای دستور، بسط پارامتر را با آن جایگزین می‌نماید. این عمل *جایگزینی* هم نامیده می‌شود.

انواع متغیر

اگر چه BASH یک زبان تیپیک نیست، چند نوع متغیر متفاوت دارد. این گروه‌ها، نوع کمیتی که می‌توانند داشته باشند را مشخص می‌کنند. اطلاعات نوع متغیر به طور داخلی توسط Bash نگهداری می‌شوند.

- **آرایه**: دستور declare -a variable: یک متغیر که آرایه‌ای از رشته‌هاست، تعریف می‌کند.
- **آرایه انجمنی(شرکت‌پذیر)**: دستور declare -A variable: یک متغیر آرایه‌ای شرکت پذیر از رشته‌ها تعریف می‌کند(bash نگارش 4.0 یا بالاتر).

- **عدد صحیح**: دستور `declare -i variable`: یک متغیر نگاهدارنده عدد صحیح تعریف می‌کند. تخصیص مقدار به این متغیر، به طور خودکار *ارزیابی حسابی* را فعال می‌کند.
- **فقط خواندنی**: دستور `declare -r variable`: متغیری تعریف می‌کند که نمی‌تواند اصلاح یا حذف شود.
- Export: دستور `declare -x variable`: متغیر را به صورت صادر نمودنی تعریف می‌کند، یعنی می‌تواند به زیرپوسته‌ها یا پردازش فرزند به ارث برسد.

آرایه‌ها اساساً لیست شاخص‌گذاری شده‌ای از رشته‌ها هستند. اینها به جهت قابلیت نگهداری چندین رشته همراه یکدیگر، بدون استناد به *جداکننده* برای تفکیک آنها از هم(که انجام صحیح آن کسل‌کننده و درغیر آن‌صورت متمایل به خطا هستند)، خیلی مناسب می‌باشند.

تعریف متغیرها به عنوان عدد صحیح، این مزیت را دارد، که موقع تخصیص و اصلاح آنها می‌توانید از برخی ترکیب‌های دستوری(syntax)صرفنظر کنید:

```
$ a=5; a+=2; echo $a; unset a
52
$ a=5; let a+=2; echo $a; unset a
7
$ declare -i a=5; a+=2; echo $a; unset a
7
$ a=5+2; echo $a; unset a
5+2
$ declare -i a=5+2; echo $a; unset a
7
```

هرچندکه، در عمل استفاده از `declare -i` به اندازه زیادی کمیاب است. به طور عمده، این به علت ایجاد رفتاری است، که برای شخصی که از اسکریپت نگهداری می کند و به وجود دستور `declare` اشراف ندارد ، شگفت‌آور خواهد بود. اکثر اسکریپت نویسان ورزیده، ترجیح می‌دهند، وقتی می‌خواهند محاسبات انجام دهند، از دستورات صریح حسابی `let` یا `((...))`، استفاده کنند.

همچنین یک تعریف صریح از یک آرایه با استفاده از `declare -a` به ندرت می‌بینید. نوشتن `array=(...)` کافی است و Bash آگاه خواهد شد که اکنون متغیر یک آرایه است. به استثنای آرایه شرکت‌پذیر، که *باید* به طور صریح تعریف شود، یعنی: `declare -A myarray`.

رشته: یک رشته، توالی از کاراکترها می‌باشد.

آرایه: یک آرایه لیستی از رشته هامی‌باشد، که با اعداد شاخص‌گذاری شده‌اند.

عدد صحیح: یک عدد کامل مثبت، منفی، یا صفر می‌باشد.

فقط خواندنی: پارامترهایی که فقط خواندنی هستند نمی‌توانند حذف یا اصلاح شوند.

Export: متغیرهایی که به عنوان صادر کردنی علامت خورده‌اند به هر زیرپوسته یا پردازش فرزند به ارث می‌رسند.

در پرسش و پاسخ‌های رایج:
چگونه می توانم از متغیرهای آرایه‌ای استفاده کنم؟

بسط پارامتر

بسط پارامتر اصطلاحی است که به هر عملی اشاره می‌کند که موجب بسط یافتن(تعویض با محتوا) یک پارامتر گردد. در اساسی‌ترین شکل، بسط پارامتر با پیشوند کردن پارامتر توسط یک علامت $ به پارامتر به دست می‌آید. در بعضی موقعیت‌های خاص جفت ابروی اضافی در اطراف نام پارامتر لازم است:

```
$ echo "'$USER', '$USERs', '${USER}s'"
'lhunath', '', 'lhunaths'
```

این مثال شرح می‌دهد که اساس بسط پارامتر(PE) چگونه است. دومین PE به یک رشته تهی منجر می‌شود. به علت آنکه پارامتر USERs تهی است. هدف ما این نبود که s بخشی از نام پارامتر باشد. چون به این طریق BASH نمی‌تواند بداند که شما می‌خواهید یک s لفظی به محتوای پارامتر پیوست کنید، لازم است از ابروها برای علامت گذاری ابتدا و انتهای نام پارامتر استفاده کنید. آنچه که ما در سومین PE در مثال فوق انجام داده‌ایم.

بسط پارامتر همچنین ترفندهایی برای ویرایش رشته‌ای که بسط خواهد یافت، به ما می‌دهد. این عملیات می‌توانند بسیار مناسب باشند:

```
$ for file in *.JPG *.jpeg
> do mv "$file" "${file%.*}.jpg"
> done
```

کد بالا می‌تواند برای تغییر نام همه فایل‌های JPEG با پسوند *.JPG* یا *.jpeg* به فایل با پسوندمعمول *.jpg* استفاده شود. عبارت ${file%.*}$ قسمتی ازانتهایی‌ترین بخش نام فایل که با نقطه(.) شروع شده باشد، را جدا می‌کند. سپس در همان نقل‌قول، پسوند جدید به انتهای نتیجه حاصل از بسط افزوده می‌شود.

در اینجا خلاصه‌ای ازاکثر ترفندهای PE که معتبر هستند آمده است:

- **${parameter:-word}**: مقادیر پیش‌فرض استفاده می‌شود. اگر parameter موجود نباشد یا تهی باشد، بسط word جایگزین می‌گردد. در غیر آن‌صورت، مقدار parameter جایگزین می‌گردد.
- **${parameter:=word}**: تخصیص مقدار پیش‌فرض. اگر parameter موجود نباشد یا تهی باشد، بسط word به parameter تخصیص داده می‌شود. سپس مقدار parameterجایگزین می‌شود.
- **${parameter:+word}**: استفاده از مقدار جایگزین. اگر parameter موجود نباشد یا تهی باشد، چیزی جایگزین نمی‌شود، در غیر آنصورت بسط word جایگزین می‌گردد.
- **${parameter:offset:length}** بسط زیر رشته(قسمتی از رشته). به تعداد length کاراکتر از parameter با

شروع ازکاراکتری که محل آن توسط offset تعیین شده(شمارش از صفر است)، بسط می‌یابد. اگر length: از قلم افتاده باشد، تا انتها را شامل می‌شود. اگر offset منفی باشد(از پرانتز استفاده شود!)، به جای شمارش از ابتدای parameter به جلو، از انتها به طرف عقب انجام می‌شود.

- **${#parameter}**: تعداد کاراکترهایی به اندازه مقدار parameter جایگزین می‌گردد.
- **${parameter#pattern}**: الگوی pattern از ابتدای parameter مطابقت داده می‌شود. نتیجه بسط، جایگزینی باقیمانده parameter پس ار حذف کوتاهترین انطباق با الگو خواهد بود.
- **${parameter##pattern}**: مانند مورد فوق، اما با حذف *بلندترین* مورد انطباق.
- **${parameter%pattern}**: الگوی pattern با انتهای ، parameter مطابقت داده می‌شود. نتیجه، زیر رشته‌ای از parameter است که پس از حذف کوتاه‌ترین انطباق حاصل می‌شود.
- **${parameter%%pattern}**: مانند مورد فوق، اما *بلندترین* انطباق حذف می‌گردد.
- **${parameter/pattern/string}**: از چپ به راست parameter را جستجو نموده و اولین انطباق pattern را با رشته string تعویض می‌کند.
- **${parameter//pattern/string}**: مانند مورد فوق، اماهمه موارد انطباق pattern، تعویض می‌شوند.

با تمرین، همه موارد فوق را یاد می‌گیرید. اینها، اغلب خیلی بیش از آنکه فکر می‌کنید، سودمند خواهند بود. در اینجا چند مثال برای شروع اولیه شما:

```
$ file="$HOME/.secrets/007"; \
> echo "File location: $file"; \
> echo "Filename: ${file##*/}"; \
> echo "Directory of file: ${file%/*}"; \
> echo "Non-secret file: ${file/secrets/not_secret}"; \
> echo; \
> echo "Other file location: ${other:-There is no other
file}"; \
> echo "Using file if there is no other file:
${other:=$file}"; \
> echo "Other filename: ${other##*/}"; \
> echo "Other file location length: ${#other}"
File location: /home/lhunath/.secrets/007
Filename: 007
Directory of file: /home/lhunath/.secrets
Non-secret file: /home/lhunath/.not_secret/007
```

```
Other file location: There is no other file
Using file if there is no other file: /home/lhunath/.secrets/007
Other filename: 007
Other file location length: 26
```

تفاوت میان ${v#p} و ${v##p} را به خاطر بسپارید. دوتایی شدن کاراکتر #به آن معنی است که، الگوها حریص(پر خور) می‌شوند. همین مطلب برای % نیز صادق است:

```
$ version=1.5.9; echo "MAJOR: ${version%%.*}, MINOR:
${version#*.}."
MAJOR: 1, MINOR: 5.9.
$ echo "Dash: ${version/./-}, Dashes: ${version//./-}."
Dash: 1-5.9, Dashes: 1-5-9.
```

توجه: نمی‌توانید *PE*های چندگانه را با هم به کار ببرید. اگر نیاز به اجرای *PE*چندگانه روی یک پارامتر دارید، باید از چندین جمله استفاده کنید:

مترجم: *PE*=Parameter Expansion(بسط پارامتر)

```
$ file=$HOME/image.jpg; file=${file##*/}; echo "${file%.*}"
image
```

تکرارمفید:

ممکن است برای ویرایش رشته‌ها، به استفاده از برنامه‌های خارجی از قبیل sed و awk و cut و perl یا سایر برنامه‌ها، وسوسه شوید. آگاه باشید که برای اجرای هر یک از این برنامه‌ها، یک پردازش اضافه‌ای باید شروع شود، که در بعضی موارد می‌تواند موجب کندی اجرا بشود. بسط پارامترها یک جایگزین بدون نقص است.

در مستندات گنو: Shell Parameter Expansion

در پرسش و پاسخ‌های رایج:

در bash چگونه می‌توانم رشته‌ها را دستکاری کنم؟

چگونه می‌توانم تمام فایل‌های *.foo را به *.bar تبدیل نمایم، یا فاصله‌ها را به خط زیر تبدیل کنم، و یا حروف بزرگ نام فایلها را به حروف کوچک تبدیل کنم؟

چگونه می‌توانم از بسط پارامتر استفاده کنم؟ چطور زیر رشته‌ها را ایجاد کنم؟ چگونه می‌توانم نام فایل را بدون پسوند کنم، یا فقط پسوند فایل‌ها را به دست آورم؟

بسط پارامتر: هر گونه بسط (تعریف پیشتر را ملاحظه کنید) یک پارامتر. در حین انجام این بسط، عملیات معینی ممکن است روی کمیتی که بسط داده خواهد شد، صورت گیرد.

الگوها

BASH سه نوع مختلف از *انطباق الگو* را ارائه می‌کند. انطباق الگو در پوسته، دونقش ایفا می‌کند: انتخاب نام فایل‌ها درون یک دایرکتوری ، یا تعیین آنکه آیا یک رشته با یک قالب دلخواه مطابقت می‌نماید.

در خط فرمان شما غالباً از *جانشین‌ها*(*globs*) استفاده می‌کنید. جانشین‌ها به طور مساعدی شکل ساده الگوها هستند، که می‌توانند به آسانی برای انطباق با گروهی از فایل‌ها به کار بروند، یا متغیرها را در برابر قواعد ساده بررسی کنند.

دومین نوع انطباق الگوها، *globهای توسعه یافته* را در بر می‌گیرند، که نسبت به جانشین‌های معمولی، کاربرد عبارتهای پیچیده‌تری را اجازه می‌دهند.

پس از نگارش 3.0، BASH از الگوهای*عبارت‌های منظم* نیز پشتیبانی می‌کند. اینها در اسکریپ‌ها برای بررسی ورودی کاربر و یا تفکیک داده‌ها مناسب هستند.

الگو: الگو یک رشته طراحی شده با یک ساختار ویژه برای انطباق با نام فایلها، یا کنترل، دسته‌بندی، یا معتبرسازی رشته‌ها است.

الگوهای جانشین(Glob Patterns)

جانشین‌ها(globs) اگر فقط برای راحتی باور نکردنی‌شان باشد هم، مفهوم بسیار مهمی در BASH می‌باشند. درک صحیح globها به طُرق بسیاری برای شما مفید خواهد بود. جانشین‌ها اساساً الگوهایی می‌باشند که می‌توانند برای انطباق با نام فایلها یا سایر رشته‌ها به کار بروند.

جانشین‌ها مرکب از کاراکترهای معمولی و فوق کاراکترها هستند. فوق کاراکترها، آن کاراکترهایی هستند که معنی ویژه‌ای دارند. فوق کاراکترهای اصلی عباتند از:

- *: بر هر رشته‌ای ازجمله رشته تهی منطبق می‌گردد.
- ?: بر یک کاراکتر منفرد منطبق می‌شود.
- [...]: بر هر یک از کاراکترهای محصور در کروشه‌ها منطبق می‌شود.

جانشین‌ها به طور صریح از هر دو طرف *مهار* می‌گردند. این به آن معناست که یک جانشین بایستی بر *تمام* رشته(نام فایل یا رشته داده‌ای) منطبق شود. a* با رشته cat منطبق نیست، به علت آنکه فقط بر at، منطبق می‌شود، نه بر تمام رشته. در حالیکه، یک جانشین ca*، با رشته cat منطبق می‌گردد.

در اینجا مثالی در مورد اینکه چگونه می‌توانیم از الگوهای جانشین برای بسط نام فایلها استفاده کنیم:

```
$ ls
a  abc  b  c
$ echo *
```

```
a abc b c
$ echo a*
a abc
```

BASH جانشین را می‌بیند، به عنوان مثال a* را، واین جانشین را از طریق نگاه کردن به دایرکتوری جاری و مطابقت glob با تمام فایلهای موجود در آن، بسط می‌دهد. *هر نام فایلی* که با الگوی جانشین مطابقت داشته باشد، به شمار آمده و به جای جانشین به کار می‌رود. در نتیجه جمله echo a* با جمله echo a abc تعویض شده و بعد اجرا گردیده است.

BASH بسط نام فایل را *بعد از* تفکیک کلمه‌ای، که قبلاً انجام داده است، اجرا می‌نماید، بنابراین، نام فایل‌های ایجاد شده توسط جانشین، همیشه به طور صحیح به کار خواهد رفت. برای مثال:

```
$ touch "a b.txt"
$ ls
a b.txt
$ rm *
$ ls
$
```

در اینجا، * به نام یک فایل منفرد "a b.txt" بسط یافته. این نام فایل به عنوان یک شناسه منفرد به فرمان rm تحویل می‌گردد. مهم است که بدانیم کاربرد *جانشین‌ها* برای به شمار آوردن نام فایلها همواره از ایده به کارگیری دستور `ls` برای این منظور، بهتر هستند. در اینجا مثالی می‌آوریم با ترکیب پیچیده‌تری که بعداً آنرا پوشش خواهیم داد، اما دلیل مطلب فوق را خیلی خوب تشریح می‌کند:

```
$ ls
a b.txt
$ for file in `ls`; do rm "$file"; done
rm: cannot remove `a': No such file or directory
rm: cannot remove `b.txt': No such file or directory
$ for file in *; do rm "$file"; done
$ ls
$
```

در اینجا از فرمان for برای پوشش دادن تمام خروجی دستور ls استفاده کرده‌ایم. دستور ls رشته a b.txt را به خروجی می‌دهد. دستور for آن رشته را به کلمات تفکیک می‌کند و به تعداد آن کلمات تکرار را انجام می‌دهد. در نتیجه، for ابتدا برای a، و بعد هم برای b.txt تکرار می‌شود. بدیهی است، این، آنچه ما می‌خواهیم نیست. در حالیکه، glob، به شکل صحیح بسط می‌یابد. و فایل "a b.txt" را نتیجه می‌دهد، که فرمان for آن را به عنوان یک شناسه منفرد دریافت می‌کند.

BASH همچنین از یک ویژگی به نام *جانشین‌های توسعه یافته* پشتیبانی می‌کند. این *جانشین‌ها* در ماهیت قدرتمندتر هستند، از لحاظ فنی، آنها معادل عبارتهای معمولی هستند، اگر چه ساختار آنها به ظاهر متفاوت با آنچه اکثریت مردم به کار می‌برند، باشد. این ویژگی به طور پیش‌فرض غیر فعال است، لیکن می‌تواند با دستور shopt، که برای تغییر وضعیت گزینه‌های پوسته به کار می‌رود، فعال شود. این دستور

کوته‌نوشتی از عبارت **sh**ell **opt**ions می‌باشد:

```
$ shopt -s extglob
```

- **?(list)**: صفر یا یک مورد تطابق با الگوی داده شده.
- ***(list)**: هر یا هیچ مورد انطباق با الگوی مورد اشاره.
- **+(list)**: یک مورد انطباق با الگو یا بیشتر.
- **@(list)**: انطباق با یکی از نمونه‌های داده شده.
- **!(list)**: با هر چیزی غیر از موارد ذکر شده انطباق می‌یابد.

کلمه list داخل پرانتزها لیستی از جانشین‌های معمولی یا توسعه یافته می‌باشد که با کاراکتر | از یکدیگر جدا شده‌اند. این هم یک مثال:

```
$ ls
names.txt  tokyo.jpg  california.bmp
$ echo !(*jpg|*bmp)
names.txt
```

در اینجا الگوی جانشین(list) به هر چیزی که بر *jpg یا *bmp منطبق نمی‌شود بسط داده می‌شود. فقط فایلهای متن همان طور که بسط یافته‌اند به دستور تحویل شده‌اند.

علاوه بر بسط نام فایل، از جانشین‌ها می‌توان برای بررسی انطباق داده‌ها با یک قالب مشخص شده نیز استفاده نمود. برای مثال، ممکن است نام فایلی را داده باشیم، و انتظار عملیات متفاوت بر اساس پسوند فایل داشته باشیم:

```
$ filename="somefile.jpg"
$ if [[ $filename = *.jpg ]]; then
> echo "$filename is a jpeg"
> fi
somefile.jpg is a jpeg
```

کلمه کلیدی [[و دستور داخلی case (که بعداً با تفصیل بیشتری شرح داده می‌شوند) هر دو فرصت بررسی یک رشته در برابر جانشین معمولی ـــو یا جانشین توسعه یافته در صورتی‌که فعال شده باشدـــ را فراهم می‌کنند.

سپس، *بسط ابرو* را داریم. از نظر تکنیکی بسط ابرو در زمره جانشین‌ها نمی‌باشد، اما مشابه آن است. جانشین‌ها فقط به نام فایلهای حقیقی بسط می‌یابند، در جایی که بسط ابرو به هر جایگردی از الگو بسط خواهد یافت. در اینجا چگونگی کارکرد آن:

```
$ echo th{e,a}n
```

```
then than
$ echo {/home/*,/root}/.*profile
/home/axxo/.bash_profile /home/lhunath/.profile
/root/.bash_profile /root/.profile
$ echo {1..9}
1 2 3 4 5 6 7 8 9
$ echo {0,1}{0..9}
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
```

تکرار سودمند:

برای به شمار آوردن فایلها همواره باید از جانشین‌ها به جای `ls`(یا مشابه آن) استفاده کنید. جانشین‌ها همیشه به طور ایمن و با حداقل ریسک ایجاد باگ بسط می‌یابند.

گاهی ممکن است با نام فایلهای خیلی عجیب روبرو شوید. اکثر اسکریپت‌ها برای هر یک از مواردی که در نتیجه استفاده از آنها ممکن است حاصل شود، بررسی نمی‌شوند. اجازه ندهید اسکریپت شما نیز یکی از آنها باشد!

در مستندات گنو: Pattern Matching

در پرسش و پاسخ‌های رایج:
چگونه می‌توانم از AND/OR/NOT منطقی در الگو(جانشین) پوسته استفاده کنم؟

جانشین(*glob*): یک جانشین رشته‌ایست که می‌تواند با نام فایلها یا رشته‌های معینی منطبق گردد.

عبارت‌های منظم

عبارت‌های منظم(**regex**) مشابه *الگوهای جانشین* هستند، اما در BASH نمی توانند برای انطباق با نام فایل به کار بروند. از نگارش 3.0، BASH عملگر **=~** در کلمه کلیدی [[ ]] را پشتیبانی می‌کند. این عملگر رشته‌ای را که قبل از آن می‌آید با الگوی **regex** که بعد از آن می‌آید، مطابقت می‌دهد. موقعی که رشته با الگو منطبق گردد، کلمه‌کلیدی [[ ]] یک کد خروج **0** (true) بر می‌گرداند. اگر رشته با الگو مطابقت نداشته باشد، یک کد خروج **1** (false) باز گردانده می‌شود. در صورتیکه ترکیب دستوری الگو معتبر نباشد، [[ ]] از عملیات صرفنظر نموده و یک کد خروج **2** صادر می‌کند.

BASH از *عبارت منظم توسعه یافته* (**ERE**) نیز استفاده می‌کند. ما در این راهنما regexها را به طور گسترده پوشش نمی‌دهیم، اما اگر این مفهوم برای شما جالب است، لطفاً به عبارت منظم، یا Extended Regular Expressions مراجعه نمایید.

الگوهای *عبارت منظم* که برای گرفتن گروه‌ها(پرانتزها)به کار می‌روند، رشته‌های گرفته شده‌شان را برای بازیابی بعدی، به متغیر BASH_REMATCH، تخصیص خواهند داد.

اجازه دهید، تشریح کنم که *regex* در BASH چگونه کار می‌کند:

```
$ langRegex='(..)_(..)'
$ if [[ $LANG =~ $langRegex ]]
> then
> echo "Your country code (ISO 3166-1-alpha-2) is  ${BASH_REMATCH[2]}."
> echo "Your language code (ISO 639-1) is ${BASH_REMATCH[1]}."
> else
> echo "Your locale was not recognised"
> fi
```

آگاه باشید که تفکیک کلمه regex در BASH از نگارش **3.1** به **3.2** تغییر کرده است. قبل از نگارش **3.2** محصور نمودن الگوی regex در نقل‌قول، صحیح بود، که این در نگارش **3.2** تغییر کرده است. پس بنابراین، regex همیشه باید غیر نقل‌قولی باشد. شما باید هر کاراکتر ویژه را با کاربرد کاراکتر \ محافظت کنید. بهترین روش برای سازگاری همیشگی، قرار دادن regex در یک متغیر و بسط آن متغیر در [[بدون استفاده از نقل‌قول‌ها می‌باشد.

تکرار مفید:

از آن جهت که روش regex مورد استفاده در 3.2 در نگارش 3.1 نیز معتبر می‌باشد، ما *قویاً* پیشنهاد می‌کنیم هرگز عبارت منظم خودتان را نقل‌قولی نکنید. به خاطر داشته باشید که کاراکترهای ویژه را به طور صحیح با کاراکتر گریز پوشش دهید!

برای سازگاری سراسری (اجتناب از الزام به پوشش کاراکترهای خاص) از یک متغیر برای ذخیره عبارت منظم خود استفاده کنید، مانند. `re='^\*( >| *Applying|.*\.diff|.*\.patch)'; [[ $var =~ $re ]]` این خیلی آسانتر از آن است که شما فقط ترکیب دستوری ERE را بنویسید و از لزوم پوشش، به همان خوبی اجتناب کنید که با تمام نگارش‌های 3.xاز BASH سازگار باشد.

همچنین، بخش E14 از Chet Ramey's Bash FAQ، را ملاحظه نمایید.

در مستندات گنو: Regex(3)

در پرسش و پاسخ‌های رایج:

من می‌خواهم بدون تکرار n مرتبه $var، عبارت ... $var == more یا $var == bar یا $var == foo [[if را بررسی کنم.

عبارت منظم: یک عبارت منظم، الگوی پیچیده‌تری است که می‌تواند برای انطباق با رشته‌های معین به کار برود(اما بر خلاف جانشین‌ها نمی‌تواند به نام فایل‌ها بسط داده شود).

شرط‌ها و بررسی‌ها

فهرست مطالب

اجرای ترتیبی فرمانها به جای خود، اما برای دستیابی به منطق پیشرفته در اسکریپت‌هایتان یا در خط فرمان یک جمله‌ای، به شرط‌ها و بررسی‌ها نیاز دارید. بررسی‌ها تعیین می‌کنند که یک مطلبی صحیح است یا غلط. شرط‌ها برای تصمیم‌سازی در مورد انجام فرامینی در اسکریپت به کار می‌روند.

1. وضعیت خروج

از هر دستور موقعی‌که خاتمه می‌یابد یک کد خروج حاصل می‌شود. این کد خروج توسط هر برنامه‌ای که آن دستور را اجرا نموده برای تعیین آنکه مقصودش به درستی انجام شده یا نه استفاده می‌شود. این کد خروج مشابه مقدار برگشتی از توابع می‌باشد. این کد یک عدد صحیح از صفر تا ۲۵۵ می‌باشد. مطابق قرارداد از صفر برای مشخص نمودن موفقیت استفاده می‌کنیم، وهر عدد دیگر بیانگر نوعی شکست می‌باشد. هر برنامه معینی، عدد خاصی را برای اشاره به آنکه دقیقاً چه اشتباهی رخ داده به کار می‌برد.

به عنوان مثال، دستور `ping` بسته‌های ICMP را در شبکه برای یک میزبان معین ارسال می‌کند. به طور معمول آن میزبان، با برگشت دادن دقیق همان بسته پاسخ می‌دهد. به این طریق می‌توانیم کنترل کنیم که آیا می‌توانیم یک ارتباط با میزبان راه دور برقرار کنیم. دستور`ping` دامنه‌ای از کدهای خروج دارد که اگر مشکلی باشد، می‌تواند به ما بگوید، چه چیز نادرست است:

از مستندات `ping` لینوکس:

> اگر ping هیچ بسته بازگشتی دریافت نکند، با کد 1 خارج خواهد شد. اگر یک شماره بسته و یک محدوده زمانی تعیین شده باشد، و شمارش بسته‌های دریافتی در زمان تعیین شده با عدد کمتری اعلام شود نیز با کد 1 خارج می‌شود. در سایر موارد خطا با کد 2 خارج می‌شود. در غیر اینصورت با کد صفر خارج می‌شود. و استفاده از کد خروج امکان آن را فراهم می‌کند که ببینیم میزبان فعال می‌باشد یا خیر.

پارامتر ویژه `?` کد خروج آخرین پردازش پیش‌زمینه خاتمه یافته را به ما می‌دهد. اجازه دهید برای دیدن کدهای خروج فرمان `ping` مثال‌هایی بزنیم:

```
$ ping God
```

```
ping: unknown host God
$ echo $?
2
$ ping -c 1 -W 1 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
--- 1.1.1.1 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
$ echo $?
1
```

تکرارمفید:

همواره باید مطمئن شوید که اسکریپت شما در صورت وقوع رخداد ناخواسته در جریان اجرایش، کد خروج غیر صفر برمی‌گرداند. می‌توانید با استفاده از دستور داخلی `exit` این کار را عملی کنید:

```
rm file || { echo 'Could not delete file!' >&2; exit 1; }
```

در مستندات گنو: Exit Status

کد خروج / وضعیت خروج: هنگامی که یک دستور خاتمه می‌یابد به والدش(در موقعیت ما همیشه پوسته‌ای می‌شود که شروع کرده‌ایم) ، وضعیت خروج خود را گزارش می‌کند. این وضعیت با یک عدد از صفر تا ۲۵۵ نمایانده می‌شود. این کد اشاره‌ای به موفقیت اجرای دستور است.

2. عملگرهای کنترلی (&& و ||)

حال که می‌دانیم کدهای خروج چیستند، و یک کد خروج صفر به معنای اجرای موفق یک دستور می‌باشد، استفاده از این اطلاعات را خواهیم آموخت. ساده‌ترین روش انجام یک عمل معین بر اساس موفقیت دستور قبلی از راه به کارگیری ***عملگرهای کنترلی*** می‌باشد. این عملگرها `&&` و `||` می‌باشند، که به ترتیب یک AND و یک OR منطقی را نمایندگی می‌کنند. این عملگرها بین دو دستور به کار می‌روند، و برای کنترل آنکه آیا دستور دوم بر مبنای موفقیت دستور اول اجرا بشود، استفاده می‌شوند. این مفهوم اجرای شرطی نامیده می‌شود.

بیایید این مطلب را در عمل به کار ببریم:

```
$ mkdir d && cd d
```

این مثال ساده دو دستور دارد، `mkdir d` و `cd d`. می‌توانستید از یک سمی‌کالن در آنجا برای جدا کردن دستورهاو اجرای ترتیبی آنها استفاده کنید، اما ما چیزی بیش از آن می‌خواهیم. در مثال فوق، BASH فرمان `mkdir d` را اجرا می‌کند، سپس `&&` نتیجه برنامه `mkdir` پس از اتمامش را بررسی می‌کند. اگر برنامه `mkdir` موفق بود(کد خروج صفر)، بعد Bash دستور بعدی `cd d` را اجرا می‌کند.

اگر `mkdir d` ناموفق باشد، و یک کد خروج غیر صفر برگرداند، Bash از اجرای دستور بعدی صرفنظر می‌کند، و در دایرکتوری جاری خواهد ماند.

مثالی دیگر:

```
$ rm /etc/some_file.conf || echo "I couldn't remove the file"
rm: cannot remove `/etc/some_file.conf': No such file or directory
I couldn't remove the file
```

`||` خیلی مشابه `&&` می‌باشد، امادقیقاً مخالف آن عمل می‌کند. فقط موقعی دستور بعدی اجرا می‌شود که دستور اول ناموفق شود. به این ترتیب، پیغام فقط در صورتی که فرمان `rm` ناموفق باشد، نمایش داده می‌شود.

به طور کلی، متصل کردن چند دستور کنترلی در یک جمله منفرد ایده خوبی *نیست*(ما این مطلب را در بخش بعدی باز خواهیم کرد). `&&` و `||` در حالت‌های ساده کاملاً سودمند می‌باشند، اما در وضعیت‌های پیچیده اینطور نیست. در چند بخش بعدی برخی ابزارها که می‌توانید در تصمیم سازی به کار ببرید رانشان خواهیم داد.

تکرارمفید:

وقتی با عبارت‌های شرطی سر و کار دارید خیلی هواخواه این عملگرها نباشید. اینها میتوانند درک اسکریپت شما را دشوار سازند، به ویژه برای کسی که به نگهداری آن منصوب شده و خودش اسکریپت را ننوشته است.

در مستندات گنو: Lists of Commands

عملگرهای کنترل: این عملگرها برای پیوند زدن دستورها با یکدیگر استفاده می‌شوند. آنها کد خروج دستور قبلی را برای تعیین اجرا یا عدم اجرای دستور بعدی بررسی می‌کنند.

---

3. گروه‌بندی دستورات

استفاده از عملگرهای شرطی ساده و موجز می‌باشد، به شرطی که بخواهیم کنترل خطای ساده‌ای انجام دهیم. گرچه، موقعی که بخواهیم در صورت صحیح بودن یک شرط، جملات چندگانه‌ای را اجرا کنیم، یا نیاز به بررسی شرط‌های چندگانه داشته باشیم، مسائل قدری خطرناک‌تر می‌شوند.

فرض کنید می‌خواهید یک فایل را در صورت وجود کلمه معین "good" در آن و نیز عدم وجود کلمه مشخص "bad" در آن حذف کنید. با استفاده از `grep` (فرمانی که ورودی‌اش را برای الگوهای تعیین شده بررسی می‌کند)، این شرایط را به این صورت ترجمه می‌کنیم:

```
grep -q goodword "$file"      #  exit status 0 (success) if "$file"
                                 contains 'goodword'
! grep -q "badword" "$file"   #  exit status 0 (success) if "$file" does
                                 not contain 'badword'
```

ما از گزینه **-q** (quiet) با فرمان grep استفاده کردیم چون نمی‌خواهیم موارد انطباق را به خروجی ارسال کند، فقط می‌خواهیم کد خروج را تنظیم کند.

علامت **!** در جلوی دستور موجب می‌شود Bash وضعیت خروج فرمان را **نفی** کند. اگر دستور صفر(موفقیت) را برگرداند، کاراکتر **!** آن را به عدم موفقیت تبدیل می‌کند، و برعکس، اگر کد غیر صفر(عدم موفقیت) برگرداند، کاراکتر **!** آن را به موفقیت تبدیل نماید.

حال برای متصل کردن این شرطها به یکدیگر و ربط دادن حذف فایل به موفقیت هر دو، می‌توانستیم از عملگرهای شرطی استفاده کنیم:

```
$ grep -q goodword "$file" && ! grep -q badword "$file" && rm "$file"
```

این بخوبی کار می‌کند(در حقیقت می‌توانیم هر تعداد از **&&**ها که می‌خواهیم بدون مشکل با هم زنجیر کنیم). حالا فرض کنید می‌خواهیم در حالتی که حذف فایل ناموفق باشد، یک پیغام خطا نمایش بدهیم:

```
$ grep -q goodword "$file" && ! grep -q badword "$file" && rm "$file"
|| echo "Couldn't delete: $file" >&2
```

این هم ظاهراً در نگاه اول صحیح است. اگر کد خروج rm برابر **0**(موفقیت) نباشد،سپس عملگر **||** ماشه اجرای دستور بعدی را می کشد و echo پیغام خطا را نمایش می‌دهد(**>&2** در خروجی استاندارد خطا).

اما مشکلی وجود دارد. موقعی که ما یک توالی از دستوراتی که با **عملگرهای شرطی** از یکدیگر جدا شده‌اند، داریم Bash به ترتیب از چپ به راست به هر یک از آنها نگاه می‌کند. وضعیت خروج آخرین دستور اجرا شده ثابت می‌ماند و پرش از روی دستورات بعدی آن را تغییر نمی‌دهد.

همچنین تصور کنید اولین grep ناموفق است(کد وضعیت یک می‌شود) . Bash حالا **&&** بعدی را می‌بیند، بنابراین به طور کلی دومین grep را نادیده می‌گیرد. بعد یک **&&** دیگر می‌بیند، بنابراین از دستور rm که بعد از آن است نیز عبور می‌کند. عاقبت یک عملگر **||** می‌بیند. آها! وضعیت خروج ناموفق است، و ما یک عملگر **||** داریم، پس Bash دستور echo را اجرا می‌کند، و به ما می‌گوید که نمی‌تواند فایل را حذف کند ولواینکه هرگز *اقدام* به این عمل نکرده است! و این چیزی نیست که ما می‌خواهیم.

وقتی فقط پیغام خطای اشتباهی دریافت می‌کنید انعکاس خیلی بدی ندارد، اما اگر دقیق نباشید، سرانجام در کدهای خطرناک‌تر، این اتفاق خواهد افتاد. شما که نمی‌خواهید به طور تصادفی در اثر نارسایی منطق برنامه خود فایلی را حذف یا رونویسی نمایید!

نقص منطق ما در این واقعیت است که ما می‌خواهیم فرمان‌های rm و echo وابسته به یکدیگر باشند. دستور echo مربوط به rm می‌باشد، نه مربوط به grepها. بنابراین آنچه ما لازم داریم، *گروه‌بندی* آنها است. گروه‌بندی با استفاده از ابروها انجام می‌گردد:

```
$ grep -q goodword "$file" && ! grep -q badword "$file" && { rm "$file"
```

```
|| echo "Couldn't delete: $file" >&2; }
```

(**توجه:** فراموش نکنید که قبل ازبستن ابرو یک سمی‌کالن یا سطر جدید لازم است!)

حالا دستورات rm و echo را با هم گروه‌بندی نموده‌ایم. این به طورمؤثر و کارامدی به معنای آنست که گروه به عنوان یک جمله در نظر گرفته می‌شود، نه چند دستور. برگردیم به موقعیتی که اولین دستور grep ما ناموفق بود، حالا BASH به جای اینکه به جمله "$file" && rm && { ... } را بررسی می‌کند. چون یک عملگر && مقدم بر این جمله است و آخرین دستور اجرا شده ناموفق بوده(دستور ناموفق grep)، از روی این گروه عبور کرده به پیش می‌رود.

گروه‌بندی دستورات برای موارد بیشتری غیر از *عملگرهای شرطی* نیز می‌تواند به کار رود. ممکن است بخواهیم دستورات را گروه‌بندی کنیم تا یک ورودی را به این گروه تغییر مسیر بدهیم، نه فقط به یکی از دستورات:

```
{
    read firstLine
    read secondLine
    while read otherLine; do
        something
    done
} < file
```

در اینجا ما file را به ورودی یک گروه از دستورات تغییر مسیر داده‌ایم. وقتی اجرای گروه دستورات شروع می‌گردد، فایل باز می‌شود و درتمام مدت اجرا باز می‌ماند و پس از اتمام اجرای گروه بسته می‌شود. به این طریق، می‌توانیم به طور متوالی توسط دستورات چندگانه سطرهای آن را بخوانیم.

یک مورد استفاده رایج دیگر از گروه‌بندی، مدیریت خطای ساده است:

```
cd "$appdir" || { echo "Please create the appdir and try again" >&2;
exit 1; }
```

4. بلوک‌های شرطی(if و test و [[ )

if یک کلمه‌کلیدی پوسته است که یک دستور(یا یک مجموعه دستور) را اجرا می‌کند، وکد خروج آن دستور را بررسی می‌کند که ببیند آیا موفق شده است. بر مبنای کد خروج، دستور if یک بلوک مشخص، متفاوت، از دستورات را اجرا می‌کند.

```
$ if  true
> then echo "It was true."
> else echo "It was false."
```

```
> fi
It was true.
```

در اینجا یک نمای کلی اساسی از *if -statement* ملاحظه می‌کنید.با فراخوانی دستور if با true شروع کرده‌ایم. true یک دستور داخلی است که همیشه به طور موفق خاتمه می‌یابد. if این دستور داخلی را اجرا می‌کند، و موقعیکه دستور اجرا شد، if کد خروج آن را بررسی می‌کند. چون true همواره به طور موفق خارج می‌شود، if با بلوک then ادامه می‌دهد، و کد را اجرا می‌کند. اگر به فرض دستور true به طریقی ناموفق می‌شد، و یک کد خروج عدم موفقیت صادر می‌کرد، دستور if از روی کد then عبور کرده و در عوض، کد بلوک else را اجرا می‌نمود.

افراد مختلف شیوه‌های متفاوتی از نوشتن جملات if را ترجیح می‌دهند. در اینجا برخی شیوه‌های رایج را می‌آوریم:

```
if  commands
then other commands
fi
--------------------
if  commands
then
    other commands
fi
--------------------
if  commands; then
    other commands
fi
```

چند دستور وجود دارد که به طور ویژه برای *بررسی* موارد و بازگرداندن وضعیت خروج نسبت به آنچه تشخیص می‌دهند، طراحی گردیده‌اند. اولین دستور از این قبیل test می‌باشد(که] نیز شناخته می‌شود.) یک نگارش پیشرفته‌تر آن]] نامیده می‌شود(*مترجم: برای راحت خواندن این قسمت] را تست و]] را تست جدید بخوانید*).] یا test یک دستور معمولی است که شناسه‌هایش را می‌خواند و برخی کنترل‌ها را با آنها انجام می‌دهد.]] خیلی مشابه] است، اما این استثنایی(یک کلمه‌کلیدی پوسته) می‌باشد که تنوع و تطبیق‌پذیری بیشتری ارائه می‌کند.بیایید به کار ببریم:

```
$ if [ a = b ]
> then echo "a is the same as b."
> else echo "a is not the same as b."
> fi
a is not the same as b.
```

if دستور] را(به خاطر داشته باشید، که نیازی به یک if برای اجرای دستور] ندارید!)با شناسه‌های *a* و = و *b* و] اجرا می‌کند. دستور] این شناسه‌ها را برای تعیین آنچه باید بررسی شود، به کار می‌برد. در این حالت، بررسی می‌کند که آیا رشته a (شناسه اول) مساوی(شناسه دوم) است با رشته b (شناسه سوم)، و اگر چنین باشد، به طور موفق خارج می‌شود. اگرچه، ما می‌دانیم که اینطور نیست،] به طور موفق

خارج نمی‌شود(کد خروج آن 1 خواهد بود). if می‌بیند که دستور] به طور ناموفق خاتمه یافته است پس کد بلوک else را اجرا می‌کند.

حال ببینیم که چرا [[خیلی بیش از] جالب و مورد اعتماد است، اجازه دهید برخی مسائل محتمل با] را مشخص نماییم:

```
$ myname='Greg Wooledge' yourname='Someone Else'
$ [ $myname = $yourname ]
-bash: [: too many arguments
```

می‌توانید حدس بزنید چه مشکلی موجب بروز خطا شده؟

دستور] با شناسه‌های Greg و Wooledge و = و Someone و Else و] اجرا گردیده است. اینها ۶ شناسه هستند نه ۴ تا! دستور] نمی‌فهمد که اجرای چه آزمونی مورد انتظار است، زیرا انتظار دارد یکی از دو شناسه اول یا دوم، یک عملگر باشد. در وضعیت ما، عملگر سومین شناسه است. باز هم دلیل دیگری برای چرایی اهمیت شگرف نقل‌قولی نمودن. هنگامی که در Bash فضای سفیدی تایپ می‌کنیم که متعلق به کلمات قبل یا بعد آن می‌باشد، **لازم است آن را نقل‌قولی کنیم**، و همین طور هم برای بسط پارامترها:

```
$ [ "$myname" = "$yourname" ]
```

در این حالت] دومین شناسه را یک عملگر(=) می‌بیند و می‌تواند به کارش ادامه دهد.

برای کمی مساعدت با ما، پوسته *Korn* یک سبک جدید بررسی شرطی را معرفی نموده(و BASH نیز آن را اخذ کرده). مؤلف اصل اینها که [[نامیده می‌شوند، پوسته کورن است. در [[چند ویژگی بسیار جالب گنجانیده شده است که در] غایب بودند.

یکی از ویژگیهای [[انطباق الگو است:

```
$ [[ $filename = *.png ]] && echo "$filename looks like a PNG file"
```

ویژگی دیگر [[کمک به ما در ارتباط با بسط پارامترها می‌باشد:

```
$ [[ $me = $you ]]          # Fine.
$ [[ I am $me = I am $you ]] # Not fine!
-bash: conditional binary operator expected
    -bash: syntax error near `am'
```

در این حالت، نیازی به نقل‌قولی کردن $me و $you نیست. چون [[یک دستور معمولی نیست(آن طور که[هست)، بلکه یک *کلمه کلیدی shell* می‌باشد، و قدرت جادویی مخصوصی دارد. این کلمه کلیدی شناسه‌هایش را قبل از اینکه آنها توسط Bash بسط داده شوند، تفکیک می‌کند و خودش بسط را انجام می‌دهد و نتیجه را به عنوان یک شناسه منفرد می‌گیرد، حتی اگر این نتیجه شامل فضای سفید هم باشد. (به بیان دیگر [[تفکیک کلمه روی شناسه‌هایش را اجازه نمی‌دهد.) *به هرحال*، هنوز هم مراقب باشید که رشته‌های ساده به طور صحیحی نقل‌قولی بشوند. زیرا [[نمی‌تواند تشخیص بدهد که آیا فضای سفید درجمله تعمدی است یا خیر، بنابراین آنها را مطابق روشی که BASH به طورمعمول انجام می‌دهد، تفکیک می‌کند. اجازه بدهید مثال را تصحیح کنیم:

```
$ [[  "I am $me" = "I am $you" ]]
```

همچنین، تفاوت ظریف زیرکانه‌ای بین نقل‌قولی کردن و نکردن سمت راست مقایسه در [[وجود دارد. عملگر = وقتی *طرف راست* نقل‌قولی نشده باشد، به طور پیش‌فرض، انطباق الگو را انجام می‌دهد:

```
$ foo=[a-z]* name=lhunath
$ [[ $name = $foo ]] && echo "Name $name matches pattern $foo"
Name lhunath matches pattern [a-z]*
$ [[ $name = "$foo" ]] || echo "Name $name is not equal to the
string $foo"
Name lhunath is not equal to the string [a-z]*
```

بررسی اول کنترل می‌کند که آیا $name با *الگوی* محتوای $foo مطابقت دارد. دومین بررسی کنترل می‌کند که آیا $name مساوی *رشته* محتوای $foo می‌باشد. نقل‌قول‌ها به طور واقعی اختلاف آنها را خیلی زیاد نموده‌اند--سزاوار زیرکی نیست.

یادآوری: اگر اطمینان ندارید، همواره نقل‌قولی کنید. اگر foo واقعاً به جای یک رشته شامل یک الگو باشد(موردی که خواستن آن نادر است -- به طور معمول الگوی شما به طور لفظی نوشته می‌شود: [[$name = [a-z]*]])، در اینجا شما یک خطای بی‌خطر دریافت می‌کنید و می‌توانید بیایید و آن را تصحیح کنید. اگر در نقل‌قول کردن سهل‌انگاری کنید، پیدا کردن باگ‌هامی‌تواند بسیار مشکل بشود، چون کد معیوب می‌تواند به طور فوری خرابی را بروز ندهد.

می‌توانید چندین دستور if را هم با استفاده از elif به جای else در یک جمله ترکیب کنید، که در آن هر بررسی نشانگر یک احتمال دیگر باشد:

```
$ name=lhunath
$ if [[ $name = "George" ]]
> then echo "Bonjour, $name"
> elif [[ $name = "Hans" ]]
> then echo "Goeie dag, $name"
> elif [[ $name = "Jack" ]]
> then echo "Good day, $name"
> else
> echo "You're not George, Hans or Jack.  Who the hell are you,
$name?"
> fi
```

حال که درک مناسبی از مسائلی که با نقل‌قولها ممکن است ایجاد شود به دست آورده‌اید، بیایید به سایر ویژگی‌هایی که [و [[با آنها پر بار شده‌اند، نگاه کنیم:

- بررسی‌هایی که با] (که به عنوان test نیزشناخته می‌شود) پشتیبانی می‌شود:
  - -e FILE : اگر فایل موجود باشد صحیح است.
  - -f FILE : اگر فایل موجود معمولی باشد صحیح است.
  - -d FILE : اگر فایل یک دایرکتوری باشد صحیح است.
  - -h FILE : اگر فایل یک پیوند نمادین باشدصحیح است.
  - -r FILE : اگر فایل برای شما قابل خواندن باشد صحیح است.
  - -s FILE : اگر فایل موجود باشد وتهی نباشد صحیح است.
  - -t FD : اگر FD(توصیف‌گر فایل) در یک ترمینال باز شده باشد صحیح است.
  - -w FILE : اگر فایل برای شما قابل نوشتن باشد صحیح است.
  - -x FILE : اگر فایل برای شما قابل اجرا باشد صحیح است.
  - -O FILE : اگر فایل به طور مؤثر در مالکیت شما باشد صحیح است.
  - -G FILE : اگر فایل به طور مؤثر در مالکیت گروه شما باشد صحیح است.
  - FILE -nt FILE: اگر فایل اول جدیدتر از فایل دوم باشد صحیح است.
  - FILE -ot FILE: اگر فایل اول قدیمی‌تر از فایل دوم باشد صحیح است.
  - -z STRING : اگر رشته تهی باشد(طول آن صفر باشد) صحیح است.
  - -n STRING : اگر رشته تهی نباشد(طول آن صفر نباشد) صحیح است.
  - STRING = STRING: اگر رشته اول از هر نظر مانند دومی باشد صحیح است.
  - STRING != STRING: اگر رشته اول دقیقاً مانند رشته دوم نباشد صحیح است.
  - STRING < STRING:اگر در مرتب‌سازی رشته اول قبل از دومی قرار می‌گیرد صحیح است.
  - STRING > STRING: اگر رشته اول در مرتب‌سازی بعد از رشته دوم قرارمی‌گیرد صحیح است.
  - EXPR -a EXPR: اگر هر دوعبارت صحیح باشندصحیح است(and منطقی).
  - EXPR -o EXPR: اگر هر یک از دو عبارت صحیح باشد صحیح است(or منطقی).
  - EXPR !: نتیجه عبارت را معکوس می‌کند(NOTمنطقی).
  - INT -eq INT: اگر هر دو عدد صحیح دقیقاً برابر باشند صحیح است.
  - INT -ne INT: اگر هر دو عدد صحیح دقیقاً برابر نباشند، صحیح است.
  - INT -lt INT: اگر عدد صحیح اولی کوچکتر از دومی باشد صحیح است.

- INT -gt INT: اگر عدد صحیح اولی از دومی بزرگتر باشد صحیح است.
- INT -le INT: اگر عدد صحیح اولی کوچکتر یا مساوی دومی باشد صحیح است.
- INT -ge INT: اگر عدد صحیح اولی بزرگتر یا مساوی دومی باشد صحیح است.

- بررسی‌های اضافی که فقط توسط [[پشتیبانی می‌شوند :
  - STRING = (or ==) PATTERN: مانند [(یا test) مقایسه نمی‌کند، بلکه *انطباق الگو* انجام می‌شود. اگر رشته با الگوی جانشین منطبق گردد، صحیح است.
  - STRING =~ REGEX: اگر رشته با الگوی regex(عبارت منظم)تطبیق کند، صحیح است.
  - (EXPR): پرانتزها می‌توانند برای تغییر اولویت ارزیابی‌ها به کار بروند.
  - EXPR && EXPR: خیلی مشابه عملگر -a در test می‌باشد، اما اگر نتیجه عبارت اول صحیح نباشد، عبارت دوم ارزیابی نمی‌شود.
  - EXPR || EXPR: خیلی مشابه عملگر -o در test می‌باشد، اما اگر نتیجه عبارت اول صحیح باشد، عبارت دوم ارزیابی نمی‌شود.

چند مثال؟ حتماً:

```
$ test -e /etc/X11/xorg.conf && echo 'Your Xorg is configured!'
Your Xorg is configured!
$ test -n "$HOME" && echo 'Your homedir is set!'
Your homedir is set!
$ [[  boar != bear ]] && echo "Boars aren't bears."
Boars aren't bears!
$ [[  boar != b?ar ]] && echo "Boars don't look like bears."
$
$ [[  $DISPLAY ]] && echo "Your DISPLAY variable is not empty, you probably have Xorg running."
Your DISPLAY variable is not empty, you probably have Xorg running.
$ [[  ! $DISPLAY ]] && echo "Your DISPLAY variable is not not empty, you probably don't have Xorg running."
$
```

تکرارمفید:
هنگامی که یک اسکریپت BASH ایجاد می‌کنید، همیشه باید از [[به جای [استفاده کنید.
وقتی یک اسکریپت پوسته می‌نویسید، که پس از اتمام ممکن است در محیطی که BASH در دسترس نباشد، به کار برود، باید از [

استفاده کنید، به دلیل آنکه به مراتب قابل حمل‌تر می‌باشد. (در حالیکه در BASH و برخی پوسته‌های دیگر، [یک دستور داخلی است، به صورت یک برنامه خارجی نیز به خوبی در دسترس می‌باشد، یعنی به عنوان شناسه مثلاً exec و xargs کار خواهد کرد.) هرگز از **a-** یا **o-** در بررسی‌های فرمان [استفاده نکنید. به جای آن از فرمان‌های چندگانه [(یا اگر می‌توانید از [[ ) استفاده کنید. استاندارد POSIX رفتار [در مجموعه بررسی‌های پیچیده را تعریف نکرده، بنابراین هرگز نمی‌دانید چه رفتاری حاصل می‌شود.

```
if  [ "$food" = apple ] && [ "$drink" = tea ]; then
  echo "The meal is acceptable."
fi
```

در مستندات گنو: Conditional Constructs

در پرسش و پاسخ‌های رایج:

چگونه می‌توانم عبارت‌ها را گروه‌بندی کنم، مثل (a **AND** b) **OR** c؟

تفاوت دستور قدیمی و جدید تست ([و [[ ) چیست؟

چطور می‌توانم تعیین نمایم که آیا یک متغیر شامل یک زیر رشته هست؟

چگونه می‌توانم بگویم که یک متغیر آیا محتوی یک عدد معتبر هست؟

---

if (*کلمه‌کلیدی*): لیستی از دستورات را اجرا می‌کند و سپس نسبت به کد خروج آنها، کد بلوک then (بخش اختیاری else) را اجرا می‌نماید.

5. حلقه های شرطی(while و until و for)

تا اینجا آموخته‌اید چگونه برخی تصمیم‌گیری های اساسی در اسکریپت‌هایتان را بسازید. اگر چه، برای انجام همه انواع وظایفی که ممکن است از اسکریپت بخواهیم کافی نمی‌باشد. گاهی اوقات نیاز به تکرار برخی کارها داریم. برای همین، کاربرد یک ***حلقه*** لازم است. دو نوع اصلی ازحلقه(به اضافه نوع دیگری) وجود دارد، و استفاده از نوع صحیح حلقه به شما در نگهداری خوانایی و قابلیت پشتیبانی اسکریپت‌هایتان کمک می‌کند.

دو نوع اساسی حلقه‌ها while و for می‌باشند. حلقه while نوع دیگری دارد که until نامیده شده، که به سادگی بررسی را برعکس انجام می‌دهد، و حلقه for در دو قالب متفاوت ظاهرمی‌شود. در اینجا خلاصه‌ای از آنها:

- while ***command***: تا وقتی که *command* به طور موفقی اجرا می‌شود(کد خروج صفر است)، تکرار می‌شود.
- until ***command***: مادامی‌که *command* به طور ناموفق اجرا گردد(کد خروج صفر نباشد)، تکرار می‌شود.
- for ***variable*** in ***words***: حلقه برای هر یک از *words* که به نوبت در متغیر *variable* قرار می‌گیرند، تکرار می‌شود.

- for ((***expression***; ***expression***; ***expression***)): با اجرای اولین عبارت حسابی شروع می‌کند، تا موقعیکه ارزیابی دومین عبارت حسابی موفق است حلقه تکرار می‌شود، و در پایان هر حلقه عبارت حسابی سوم انجام می‌شود.

هر شکل از حلقه‌ها با کلمه‌کلیدی do دنبال می‌شود، پس از آن یک یا چند فرمان در *بدنه*، بعد هم کلمه‌کلیدی done. کلمه‌کلیدی های do و done مشابه then و fi (و elif یا else احتمالی) در ساختار دستور if که قبلاً دیدیم، می‌باشند. کار آنها این است که به ما بگویند حلقه از کجا شروع و به کجا ختم می‌شود.

در عمل، حلقه‌ها برای انواع مختلفی از وظایف به کار می‌روند. حلقه for (شکل اول) برای موقعی مناسب است که لیستی داریم، و می‌خواهیم عناصر آن را به طور ترتیبی به کار ببریم. حلقه while مناسب وقتی است که به طور دقیق نمی‌دانیم چند مرتبه عملی باید تکرار شود، و می‌خواهیم تا رسیدن به موردی که در انتظار آن هستیم تکرار انجام شود.

در اینجا چند مثال برای تشریح تفاوتها و همچنین شباهت‌های حلقه‌ها می‌آوریم. (یادآوری: در اکثر سیستم‌عامل‌ها، برای کشتن برنامه‌ای که در ترمینال در حال اجرا است از ترکیب کلیدی **Ctrl-C** استفاده می‌شود.)

```
$ while true
> do echo "Infinite loop"
> done
```

```
$ while ! ping -c 1 -W 1 1.1.1.1; do
> echo "still waiting for 1.1.1.1"
> sleep 1
> done
```

```
$ (( i=10 )); while (( i > 0 ))
> do echo "$i empty cans of beer."
> (( i-- ))
> done
$ for (( i=10; i > 0; i-- ))
> do echo "$i empty cans of beer."
> done
$ for i in {10..1}
> do echo "$i empty cans of beer."
> done
```

سه حلقه آخری با ترکیب متفاوت، دقیقاً به نتیجه یکسانی می‌رسند. در تجربه اسکریپت‌نویسی شل خود بارها با این مورد مواجه می‌شوید. تقریباً همیشه راهکارهای چندگانه‌ای برای حل یک مسئله موجود است. به زودی تشخیص مهارت شما در حل مسئله نخواهد بود، آنقدر که درچگونگی **بهترین** روش حل آن خواهد بود. شما باید بیاموزید که بهترین زاویه رویکرد حل مسئله را انتخاب کنید. به طور معمول، سادگی و قابلیت انعطاف از عوامل اصلی کد نویسی خوب محسوب می‌شوند. مورد مطلوب شخصی من، آخرین مثال است. در آن نمونه من *بسط ابرو* را برای تولید کلمات به کار برده‌ام، اما روشهای دیگری نیز وجود دارد.

بیایید نگاه نزدیک‌تری به آن مثال آخری داشته باشیم، زیرا اگر چه از دو حلقه for به نظر آسان‌تر می‌آید، اما، اگر به طور دقیق ندانید که چگونه عمل می‌کند، غالباً می‌تواند فریب دهنده باشد.

به طوری که قبلاً اشاره کرده‌ام: for لیستی از کلمات را گرفته و هر یک از آنها را در متغیر شاخص حلقه می‌گذارد، هر دفعه یکی، و سپس بدنه حلقه را با آن اجرا می‌کند. قسمت فریب دهنده آن است که BASH چطور تصمیم می‌گیرد که کلمات کدام هستند. به من اجازه دهید خودم با بسط ابروها در مثال قبلی شرح بدهم:

```
$ for i in 10 9 8 7 6 5 4 3 2 1
> do echo "$i empty cans of beer."
> done
```

BASH کاراکترهای بین کلمه‌کلیدی in و انتهای سطر را می‌گیرد، و آنها را به کلمات تفکیک می‌نماید. این تفکیک نسبت به فاصله و tabها انجام می‌شود، درست مانند تفکیک شناسه‌ها. اما اگر هر جایگزینی نقل‌قولی نشده‌ای آنجا باشد، آن هم به کلمات تفکیک می‌گردد(با استفاده از محتوای متغیر IFS). تمام این کلمات تفکیک شده، عناصر تکرار می‌شوند.

در نتیجه، **خیلی** مراقب باشید که اشتباه زیر را مرتکب نشوید:

```
$ ls
The best song in the world.mp3
$ for file in $(ls *.mp3)
> do rm "$file"
> done
rm: cannot remove `The': No such file or directory
    rm: cannot remove `best': No such file or directory
    rm: cannot remove `song': No such file or directory
    rm: cannot remove `in': No such file or directory
    rm: cannot remove `the': No such file or directory
    rm: cannot remove `world.mp3': No such file or directory
```

شما از قبل نسبت به نقل‌قولی کردن $file در دستور rm آگاه بودید، اما در اینجا چه چیزی اشتباه است؟ BASH جایگزینی دستور ($(ls *.mp3)) را بسط می‌دهد، آن را با خروجی دستور تعویض می‌کند، و *بعد* تفکیک کلمه را روی آن انجام می‌دهد(به علت آنکه نقل‌قولی نیست). در واقع Bash این عبارت را اجرا می‌کند .for file in *The best song in the world.mp3*
Boom, مات شدید.

خواهید گفت، آن را نقل‌قولی می‌کنم؟ اجازه دهید فایل دیگری اضافه کنم:

```
$ ls
 The best song in the world.mp3  The worst song in the world.mp3
$ for file in "$(ls *.mp3)"
```

```
> do rm "$file"
> done
rm: cannot remove `The best song in the world.mp3  The worst song in the
world.mp3': No such file or directory
```

نقل‌قول‌ها به راستی از فضای سفید در نام فایل‌های شما محافظت می‌کنند، اما چیزی بیش از آن انجام می‌دهند. نقل‌قول‌ها از تمام فضاهای سفید خروجی فرمان ls محافظت خواهند کرد. راهی وجود ندارد که BASH بتواند تشخیص بدهد کدام بخشهای خروجی فرمان ls نام فایل‌ها را نمایندگی می‌کنند. خروجی فرمان ls یک رشته ساده است، و BASH با آن به همین عنوان رفتار می‌کند. بعد for تمام خروجی نقل‌قولی شده را در متغیر i قرار می‌دهد و دستور rm را با آن اجرا می‌کند. *لعنت، دوباره مات شدید.*

بنابراین چه کار بکنیم؟ به طوری که قبلاً پیشنهاد نمودم، جانشین‌ها بهترین دوست شما هستند:

```
$ for file in *.mp3
> do rm "$file"
> done
```

حالا، BASH می‌داند که با نام فایل‌ها سروکار دارد، و نام فایها را می‌شناسد، و بنابراین به طور مطلوبی آنها راتفکیک می‌کند. نتیجه بسط جابشین چنین است: for file in *"The best song in the world.mp3" "The worst song in the world.mp3"*. مشکل رفع شد!

حالا بیایید به حلقه while بپردازیم. حلقه while به خاطر ظرفیتش در اجرای دستورات تا موقع وقوع موضوع مورد نظر، خیلی جالب است. در اینجا چند مثال که نشان می‌دهد چگونه حلقه while غالباً به کار می‌رود :

```
$ # ماشین نوشیدنی، نوشیدنی‌ها را در ازای بهای ۲۰ سنت تحویل می‌دهد
$ while read -p $'The sweet machine.\nInsert 20c and enter your name: '
name
> do echo "The machine spits out three lollipops at $name."
> done
```

```
$ # هر پنج دقیقه یکبار ایمیل شما را بررسی می‌کند
$ while sleep 300
> do kmail --check
> done
```

```
$ # برای برخط(آنلاین) شدن مجدد میزبان منتظر می‌ماند
$ while ! ping -c 1 -W 1 "$host"
> do echo "$host is still unavailable."
```

```
> done; echo -e "$host is available again.\a"
```

حلقه until خیلی به ندرت استفاده می‌شود، فقط به علت آنکه تا حد بسیار زیادی مشابه حلقه while می‌باشد. می‌توانستیم آخرین مثال را با حلقه until به صورت زیربنویسیم:

```
$ # برای برگشت میزبان به حالت آماده برقراری ارتباط منتظر می‌ماند
$ until ping -c 1 -W 1 "$host"
> do echo "$host is still unavailable."
> done; echo -e "$host is available again.\a"
```

در عمل، اکثر مردم حقیقتاً از حلقه while به جای آن استفاده می‌کنند.

بالإخره، از دستور داخلی `continue` برای پرش به جلو بدون اجرای بقیه بدنه و اجرای دور بعدی تکرار حلقه، و دستور داخلی `break` برای پریدن به خارج از حلقه و ادامه دستورات پس از حلقه در اسکریپت، می‌توانید استفاده کنید. این دستورات با هر دو حلقه for و while کار می‌کنند.

در مستندات گنو: Looping Constructs

در پرسش و پاسخ‌های رایج:

چگونه می‌توانم یک دستور را با تمام فایل‌های دارای پسوند .gz اجرا کنم؟

چگونه می‌توانم از اعدادی که با صفر شروع می‌شوند مثل 01 و 02 در یک حلقه استفاده کنم؟

چطو می‌توانم نام فایل‌های شامل کاراکتر فاصله یا سطر جدید یا هر دو را پیدا کرده وبا آنها کار کنم؟

چطور می‌توانم تمام فایل‌های foo.* را به bar.* تغییر نام بدهم، یا فاصله را به خط زیر تبدیل نمایم، یاحروف بزرگ در نام فایل‌ها را به حروف کوچک تبدیل کنم؟

آیا می‌توانم در Bash یک چرخنده اجرا کنم؟

می‌خواهم بررسی کنم که آیایک کلمه در یک لیست وجود دارد(یا یک عنصر عضوی از یک مجموعه هست).

حلقه: یک حلقه ساختاری است که برای تکرار کد تا موقع تحقق یک وضعیت معین، طراحی می‌شود. در آن نقطه حلقه متوقف می‌شود و کد بعد از آن اجرا می‌گردد.

for (*کلمه‌کلیدی*): یک حلقه for نوعی حلقه است که یک متغیر را به ترتیب، معادل عناصر لیستی از کمیت‌ها قرار می‌دهد، و بدنه را با آن متغیر اجرا می‌کند، و تا تمام شدن لیست تکرار می‌کند.

while (*کلمه‌کلیدی*): یک حلقه while نوعی حلقه است که اجرای کد را تا موقعی که یک دستور معین(قبل از هر تکرار اجرامی‌شود) به طور موفق اجرا می‌شود، ادامه می‌دهد.

until (*کلمه‌کلیدی*): یک حلقه until نوعی حلقه است که اجرای کد را تا موقعی که یک دستور معین(قبل از هر تکرار اجرامی‌شود) به طور ناموفق اجرا می‌شود، ادامه می‌دهد..

6. انتخاب‌ها(case و select)

گاهی اوقات می‌خواهید برنامه‌ای منطقی بر مبنای محتوای یک متغیر بسازید. این می‌توانست با گرفتن انشعاب‌های مختلف یک جمله if بر اساس نتایج حاصل از بررسی یک glob پیاده‌سازی شود:

```
shopt -s extglob

if  [[  $LANG = en* ]]; then
    echo 'Hello!'
elif [[  $LANG = fr* ]]; then
    echo 'Salut!'
elif [[  $LANG = de* ]]; then
    echo 'Guten Tag!'
elif [[  $LANG = nl* ]]; then
    echo 'Hallo!'
elif [[  $LANG = it* ]]; then
    echo 'Ciao!'
elif [[  $LANG = es* ]]; then
    echo 'Hola!'
elif [[  $LANG = @(C|POSIX) ]]; then
    echo 'hello world'
else
    echo 'I do not speak your language.'
fi
```

اما این همه مقایسه یک مقدار زائد است. BASH یک کلمه‌کلیدی به نام case دقیقاً برای چنین وضعیت‌هایی فراهم نموده است. یک جمله case اساساً چندین احتمال *الگوهای جانشین* را به شمار می‌آورد و محتوای پارامتر شما را نسبت به آنها بررسی می‌کند:

```
case $LANG in
    en*) echo 'Hello!' ;;
    fr*) echo 'Salut!' ;;
    de*) echo 'Guten Tag!' ;;
    nl*) echo 'Hallo!' ;;
```

```
    it*) echo 'Ciao!' ;;
    es*) echo 'Hola!' ;;
    C|POSIX) echo 'hello world' ;;
    *)   echo 'I do not speak your language.' ;;
esac
```

هر انتخاب در جمله case شامل یک الگو(یا لیستی از الگوها با کاراکتر | بین آنها)، یک پرانتز سمت راست، یک بلوک کد که در صورت انطباق رشته با یکی از نمونه‌ها باید اجرا بشود، و دو کاراکتر سمی‌کالن برای علامت‌گذاری انتهای کد(چون ممکن است لازم شود کد را در چند سطر بنویسید). case انطباق نمونه‌ها را موقعی‌که یک مورد موفق است، متوقف می‌کند. سپس می‌توانیم ازالگوی * در انتها برای انطباق با هرحالت دیگری که موارد انتخاب ذکر شده را در بر نمی‌گیرد، به کار ببریم. و در پایان نیز کلمه‌کلیدی esac.

یک ساختار دیگر برای انتخاب، ساختار select می‌باشد. که مشابه یک حلقه است، این جمله‌ای برای سهولت ایجاد منویی از انتخابها می‌باشد، که کاربر می‌تواند از میان آنها گزینش کند .

به کاربر انتخابهایی ارائه می‌شود و از او درخواست می‌شود یک عدد منعکس کننده انتخابش را وارد کند کند. سپس کد داخل بلوک select با متغیری که با انتخاب کاربر تنظیم می‌شود اجرا می‌شود. اگر انتخاب کاربر معتبر نباشد، محتوای متغیر تهی می‌شود:

```
$ echo "Which of these does not belong in the group?"; \
> select choice in Apples Pears Crisps Lemons Kiwis; do
> if  [[  $choice = Crisps ]]
> then echo "Correct!  Crisps are not fruit."; break; fi
> echo "Errr... no.  Try again."
> done
```

تا موقعی که دستور break اجرا نشده است، منو باز تولید می‌شود. در این مثال دستور break فقط وقتی اجرا می‌شود که کاربر مورد صحیح را انتخاب نماید.

همچنین می‌توانیم از متغیر PS3 برای تعیین اعلانی که کاربر به آن پاسخ می‌دهد استفاده کنیم. به جای نشان دادن پرسش قبل از اجرای جمله select، می‌توانیم تنظیم سؤال را به عنوان اعلان انتخاب کنیم:

```
$ PS3="Which of these does not belong in the group (#)? " \
> select choice in Apples Pears Crisps Lemons Kiwis; do
> if  [[  $choice = Crisps ]]
> then echo "Correct!  Crisps are not fruit."; break; fi
> echo "Errr... no.  Try again."
> done
```

تمام این ساختارهای شرطی(if و for و while و case) می‌توانند *تو در تو* بشوند. این به آن معناست که می‌توانید یک حلقه for با یک حلقه while در داخل آن داشته باشید، یا هر ترکیب دیگر با هرچقدر تودرتویی که مسئله شما را حل کند.

```
# یک منوی ساده
while true; do
      echo "Welcome to the Menu"
      echo "  1. Say hello"
      echo "  2. Say good-bye"

      read -p "-> " response
      case $response in
          1) echo 'Hello there!' ;;
          2) echo 'See you later!'; break ;;
          *) echo 'What was that?' ;;
      esac
done
```

تکرارمفید:

جمله select ساختن منوی ساده را آسان می‌کند، اما انعطاف‌پذیری بیشتری ارائه نمی‌کند. اگر چیزی استادانه‌تر خواسته باشید، شایدترجیح بدهید منوی خودتان را با کاربرد یک حلقه while بنویسید، با چند دستور `echo` یا `printf` ، و یک دستور `read` .

در مستندات گنو: Conditional Constructs

در پرسش و پاسخ‌های رایج:

می‌خواهم بدون n مرتبه تکرار $var، عبارت ... $var == more یا $var == bar یا $var == foo [[if را بررسی کنم.

چگونه می‌توانم شناسه‌های(گزینه‌های) خط‌فرمان را به آسانی مدیریت کنم؟

---

case(*کلمه‌کلیدی*): جمله case مقدار یک پارامتر را نسبت به چند الگوی داده شده(انتخاب‌ها) ارزیابی می‌کند.

select(*کلمه‌کلیدی*): جمله select انتخاب چند گزینه را به کاربر پیشنهاد می‌کند و بلوک کد مربوط با انتخاب کاربر در یک پارامتر اجرا می‌کند. منو تا وقتی که یک دستور `break` اجرا گردد تکرار می‌شود.

آرایه ها

به طوری که قبلاً اشاره شد، BASH سه نوع پارامتر ارائه می‌کند: رشته‌ها، اعداد صحیح، و آرایه‌ها.

بدون تردید رشته‌ها پر استفاده‌ترین نوع پارامترها می‌باشند. اما آنها همچنین، بد رفتارترین نوع پارامترها هستند. اهمیت دارد که به خاطر بسپاریم، یک رشته فقط یک عنصر را نگاه می‌دارد. به عنوان نمونه، گرفتن خروجی یک فرمان، و قرار دادن آن در پارامتر رشته‌ای بدین معنا می‌باشد که پارامتر فقط یک رشته از کاراکترها می‌باشد، صرفنظر از اینکه آیا آن رشته نام بیست فایل، بیست عدد، یا نام بیست نفر را نمایندگی می‌کند.

و همینطور است که همیشه وقتی اقلام چندگانه را در یک رشته منفرد قرار می‌دهید، باید این اقلام را به طریقی از یکدیگر جدا کنید. ما، به عنوان انسان معمولاً می‌توانیم وقتی به یک رشته نگاه می‌کنیم نام فایل‌های مختلف را کشف کنیم. ما فرض می‌کنیم که، شاید، هر سطر در یک رشته نام یک فایل را نشان می‌دهد، یا هر کلمه نام یک فایل را نمایندگی می‌کند. در حالیکه این پنداشت قابل درک است، همچنین به طور ذاتی معیوب است. هر نام فایل منفرد می‌تواند شامل هر کاراکتری باشد که ممکن است شما بخواهید برای جداکردن نام فایل‌ها در یک رشته استفاده کنید. به این معنی که از نظر تکنیکی گفته نمی‌شود که نام اولین فایل در کجای یک رشته به پایان می‌رسد، زیراکاراکتری وجود ندارد که بتواند بگوید: «من به پایان نام فایل اشاره می‌کنم» چون آن کاراکتر خودش می‌تواند بخشی از نام فایل باشد.

غالباً، اشخاص این اشتباه را مرتکب می‌گردند:

```
# این در حالت کلی کار نمی‌کند
$ files=$(ls ~/*.jpg); cp $files /backups/
```

در حالیکه احتمالاً این می‌تواند ایده بهتری باشد(استفاده از آرایه‌ها که در بخش بعد شرح داده می‌شوند):

```
# این در حالت کلی کار می‌کند
$ files=(~/*.jpg); cp "${files[@]}" /backups/
```

تلاش اولی در پشتیبان‌گیری از فایل‌های دایرکتوری جاری معیوب است. ما خروجی دستور `ls` را در یک رشته به نام files قرار داده‌ایم و سپس از بسط پارامتر **$files** به صورت غیرنقل‌قولی برای بریدن آن رشته به شناسه‌ها(بر مبنای *تفکیک کلمه*) استفاده کرده‌ایم. به طوری که قبلاً اشاره شد، تفکیک کلمه و شناسه، یک رشته را از جایی که فضای سفید وجود دارد به قطعاتی برش می‌دهد. استناد به بسط فوق به این معنی است که فرض کرده‌ایم در نام فایل‌های ما هیچ فضای سفیدی نیست، اگر باشد نام فایل به دو نیمه یا بیشتر بریده می‌شود. فرجام: نامساعد.

تنها روش مطمئن نشان دادن عناصر چندگانه رشته در Bash از طریق استفاده از آرایه‌ها می‌باشد. آرایه نوعی متغیر است که رشته‌ها را با اعداد ترسیم می‌کند. این اساساً به معنای آن است که یک لیست شماره گذاری شده از رشته‌ها را نگهداری می‌کند. چون هر یک از این رشته‌ها یک هویت(عنصر) جداگانه است، می‌تواند بدون خطر هر کاراکتری، حتی فضای سفید را در خود داشته باشد.

برای بهترین نتیجه و کمترین دردسر، به خاطر بسپارید که اگر لیستی ازعناصر دارید، همیشه باید آنها را در یک آرایه قرار دهید.

بر خلاف برخی زبانهای برنامه‌نویسی، Bash لیست‌ها، رکوردها و غیره را ارائه نکرده است. فقط آرایه‌ها و آرایه‌های انجمنی(که در نگارش

4 از Bash جدید است).

آرایه‌ها: یک آرایه لیست شماره گذاری شده رشته‌ها است: رشته‌ها را با اعداد صحیح مرتبط می‌کند .

ایجاد آرایه‌ها

چند روش موجود است که می‌توانید آرایه‌ها را ایجاد نموده یا با داده‌ها پر کنید. یک روش صحیح منفرد وجود ندارد: روشی که شما نیاز خواهید داشت بستگی به آن دارد که داده‌ها کدامند و از کجا می‌آیند.

ساده‌ترین راه برای ایجاد یک آرایه ساده با داده، استفاده از ترکیب **=()** می‌باشد:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
```

این ترکیب دستوری(syntax) برای ایجاد آرایه‌هایی با داده‌های ایستا یا مجموعه‌ای از پارامترهای رشته‌ای معلوم، عالی است، اما قابلیت انعطاف بسیار کمی برای افزودن مقادیر زیاد عناصر آرایه، در اختیار می‌گذارد. اگر انعطاف پذیری بیشتری می‌خواهید، می‌توانید از شاخص‌های صریح استفاده کنید:

```
$ names=([0]="Bob" [1]="Peter" [20]="$USER" [21]="Big Bad John")
# or...
$ names[0]="Bob"
```

توجه نمایید که بین شاخص 1و 20 در این مثال یک شکاف وجود دارد. یک آرایه باحفره‌هایی در آن *آرایه پراکنده* نامیده می‌شود. Bash این امر را اجازه می‌دهد واغلب می‌تواند کاملاً سودمند باشد.

اگر می‌خواهید یک آرایه را با نام فایل‌ها پر کنید، ممکن است احتمالاً بخواهید از *Globs* استفاده کنید:

```
$ photos=(~/"My Photos"/*.jpg)
```

توجه نمایید که در اینجا بخش My Photos را نقل‌قول کرده‌ایم زیرا شامل یک فاصله است. اگر این کار را نمی‌کردیم، Bash آن را به صورت photos=('~/My' 'Photos/' *.jpg) تفکیک می‌نمود، که به وضوح آنچه ما می‌خواهیم **نبود**. همچنین توجه نمایید که ما **فقط** بخش شامل فاصله را نقل‌قولی کردیم. به این دلیل چنین است که ما نمی‌توانیم ~ یا * را نقل‌قولی کنیم، اگر چنین کنیم، آنها کاراکترهای لفظی می‌شوند و Bash دیگر با آنها همچون کاراکترهای خاص رفتار نمی‌کند.

متأسفانه، به راستی ایجاد آرایه‌های ابهام آمیز با یک گروه نام فایل که به روش زیر ایجاد می‌شوند، آسان است:

```
$ files=$(ls)    # BAD, BAD, BAD!
$ files=($(ls))  # STILL BAD!
```

به یاد داشته باشید همیشه از کاربرد `ls` به این شکل پرهیز کنید، اولی یک رشته با خروجی فرمان `ls` ایجاد می‌کند. آن رشته احتمالاً به دلیلی که در مقدمه آرایه‌ها اشاره شد نمی‌تواند به طور بی خطر به کار برود. دومی نزدیک‌تر است، اما هنوز نام فایل‌ها را با فضای سفید تفکیک می‌کند.

روش صحیح انجام آن این است:

```
$ files=(*)      # Good!
```

این جمله یک آرایه به ما می‌دهد که در آن هر نام فایل یک عنصر جداگانه است. کامل!

این بخش که در اینجا مطرح می‌کنیم شامل برخی مفاهیم پیشرفته است. اگر هنوز آماده نیستید، شاید بخواهید پس از اینکه تمام این راهنما را خواندید به اینجا بازگردید. اگر می‌خواهید با موارد ساده ادامه دهید، می‌توانید بااستفاده از آرایه‌ها به پیش بروید.

گاهی اوقات می‌خواهیم یک آرایه از یک رشته یا خروجی یک فرمان تشکیل بدهیم. خروجی فرمانها رشته هستند: برای نمونه، اجرای یک فرمان `find` نام فایل‌ها را به شمار می‌آورد و آنها را با یک کاراکتر سطر جدید(قرار دادن هر نام فایل در یک سطر جداگانه) از هم جدا می‌کند. بنابراین برای تفکیک یک رشته بزرگ به داخل یک آرایه، لازم است به Bash بگوییم هر عضو کجا به انتها می‌رسد. (تذکر، این یک مثال بد است، چون نام فایل می‌تواند شامل یک سطر جدید باشد،بنابراین جدا کردن آنها با سطر جدید نمی‌تواند ایمن باشد! اما مثال زیر را نگاه کنید.)

آنچه برای شکستن یک رشته به کار می‌رود محتوای متغیر IFS می‌باشد:

```
$ IFS=. read -a ip_elements <<< "127.0.0.1"
```

در اینجا از متغیر IFS با محتوای . برای بریدن آدرس IP داده شده به عناصر آرایه از جایی که . وجود دارد، نتیجه یک آرایه با عناصر، 127 و 0 و 0 و 1 است.

(دستور داخلی `read` و عملگر `<<<` به طورمفصل‌تری درفصل ورودی و خروجی. پوشش داده می‌شود)

می‌توانستیم همین کار را با دستور `find` انجام بدهیم، در صورتی‌که متغیر IFS را به کاراکتر سطر جدید تنظیم می‌کردیم. اما موقعی که شخصی فایلی دارای کاراکتر سطر جدید ایجاد نماید(به طور اتفاقی یا بدخواهانه)، اسکریپت ما کار نخواهد کرد.

بنابراین، آیا روشی برای دریافت لیستی از عناصر از یک برنامه خارجی (مانند `find`) در یک آرایه Bash وجود دارد؟ به طور کلی، پاسخ بلی است، به شرط آنکه راه قابل اطمینانی برای جداسازی عناصر موجود باشد.

در یک حالت خاص از نام فایلها، پاسخ این مشکل، بایت‌های تهی(NUL) است. یک بایت تهی، بایتی است که همه بیت‌های آن صفر است: *00000000*. رشته‌های Bash نمی‌توانند شامل بایت‌های تهی باشند، به عنوان یک محصول زبان برنامه‌نویسی "C" : در زبان C بایت تهی برای علامت گذاری انتهای رشته به کاررفته است. از این جهت Bash که به زبان C نوشته شده و از رشته‌های بومی C استفاده می‌کند، این رفتار را به ارث می‌برد.

یک جریان داده(مانند خروجی یک فرمان، یا یک فایل) می‌تواند شامل بایت تهی باشد. جریانها مانند رشته‌ها هستند، با سه تفاوت عمده: آنها به صورت ترتیبی خوانده می‌شوند(به طور معمول نمی‌توانید با پرش از روی آنها عبور کنید)، آنها یک سویه می‌باشند(شما می‌توانید از آنها بخوانید یا در آنها بنویسید، اما به طور نوعی هر دو با هم میسر نیست)، و آنها می‌توانند شامل بایت‌های تهی باشند.

نه *نام‌های* فایل می‌توانند شامل بایت تهی باشند(چون آنها توسط یونیکس همانند رشته‌های C تکمیل شده‌اند)، و نه اکثریت وسیع اقلام قابل خواندن برای انسان که شاید ما بخواهیم در یک اسکریپت ذخیره کنیم(ازقبیل نام افراد، آدرس‌های IP، و غیره). این موضوع NUL را یک نامزد عالی برای جداسازی عناصر در یک جریان ، می‌سازد. به طور کلی اغلب، دستوری که می‌خواهید خروجی آن را بخوانید، یک گزینه‌ای خواهد داشت، که خروجی‌اش را به صورت جدا شده با بایت تهی، به جای سطر جدید یا کاراکتر دیگری، ایجاد می‌کند.

دستور `find` (در GNU و BSD, در هر حال) گزینه **-print0** را دارد، که ما در این مثال استفاده خواهیم نمود:

```
files=()
while read -r -d $'\0'; do
    files+=("$REPLY")
done < <(find /foo -print0)
```

این یک روش مطمئن تفکیک خروجی یک فرمان به رشته‌ها می‌باشد. به طور قابل فهمی، ابتدا کمی به هم پیچیده و گیج کننده به نظر می‌رسد. لذا، بیایید کمی آن را باز کنیم:

سطر اول **files=()** یک آرایه خالی به نام files ایجاد می‌کند.

ما از یک حلقه while استفاده می‌کنیم که هر مرتبه یک دستور `read` را اجرا می‌کند. فرمان `read` از گزینه **-d $'\0'** استفاده می‌کند، به آن معنا که به جای خواندن یک سطر در هر دفعه(تا رسیدن به یک کاراکتر سطر جدید)، تا رسیدن به بایت NUL می‌خوانیم (**\0**). همچنین از گزینه **-r** برای جلوگیری از رفتار ویژه با کاراکتر**** استفاده می‌کند.

وقتی `read` مقداری از داده ها را می‌خواند و با یک بایت تهی مواجه می‌شود، بدنه حلقه while اجرا می‌گردد. ما آنچه را خوانده‌ایم(که در متغیر REPLY قرار دارد) در آرایه قرار می‌دهیم.

برای انجام این کار، ما از ترکیب **+=()** استفاده می‌کنیم. این ترکیب دستوری یک یا چند عنصر را به انتهای آرایه ما اضافه می‌کند.

و سرانجام، ترکیب دستوری **<(..)<** که ترکیبی از یک *تغیرمسیر فایل* (**<**) و *جایگزینی پردازش* (**(..)<**) می‌باشد. در حال حاضر صرف نظر از جزئیات تکنیکی، به سادگی می‌گوییم این چگونگی ارسال خروجی فرمان `find` به درون حلقه while ما می‌باشد.

همان طور که قبلاً بیان گردید، فرمان `find` خود با یک گزینه **-print0** به کار رفته، که به او بگوید،نام فایلهایی که می‌یابد را، با یک بایت تهی تفکیک کند.

تکرار مفید:

آرایه‌ها یک لیست مطمئن از رشته‌ها هستند. آنها برای نگهداری چندین نام فایل، بدون عیب می‌باشند.

اگر شما باید یک جریان داده را به اجزاء تشکیل دهنده عناصر تفکیک نمایید، باید طریقه‌ای برای گفتن آنکه هر عنصر از کجا شروع و به کجا ختم می‌گردد، وجود داشته باشد. بسیاری اوقات، بایت تهی بهترین انتخاب برای این کار می‌باشد.

اگر لیستی از اقلام دارید، تا آنجا که ممکن است آن را به صورت یک لیست حفظ کنید. تا موقعی که واقعاً لازم نیست، آن را در یک رشته یا فایل تخریب نکنید. اگر باید به تفصیل آن را در یک فایل بنویسید و بعداً آنرا بخوانید، مشکل جداکننده را که در بالاتوضیح داده شد، به خاطر داشته باشید.

در مستندات گنو: Arrays

در پرسش و پاسخ‌های رایج:
چگونه می‌توانم از متغیرهای آرایه‌ای استفاده کنم؟
چطور می‌توانم از متغیرهای متغیر(متغیرهای غیرمستقیم، اشاره‌گرها، مرجع‌ها) یا آرایه‌های انجمنی استفاده کنم؟
چگونه می‌توانم نام فایلهای شامل کاراکتر سطرجدید، فاصله، یا هردو را پیدا کرده و با آنها کار کنم؟
من متغیرهایی را در یک حلقه مقرر می‌کنم. چرا آنها پس از اتمام حلقه، ناگهان ناپدید می‌گردند؟ یا، چرا نمی‌توانم داده‌ها را برای خواندن لوله‌کشی نمایم؟

استفاده از آرایه‌ها

استفاده از مزیت عناصر آرایه‌ها به راستی آسان است. به علت آنکه یک آرایه وسیله مطمئن ذخیره است، ما به سادگی می‌توانیم یک حلقه for را برای تکرار روی عناصر آن، به کار ببریم:

```
$ for file in "$>{myfiles[@]}"; do
>     cp "$file" /backups/
> done
```

به ترکیب دستوری استفاده شده برای بسط آرایه در اینجا توجه نمایید. ما شکل نقل‌قولی به کار برده‌ایم: "${myfiles[@]}". سپس Bash این ترکیب را با هر عنصر منفرد در آرایه، تعویض می‌نماید، نقل‌قولی صحیح.

دو مثال زیر نتیجه یکسان دارند:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
$ for name in "${names[@]}"; do echo "$name"; done
```

```
$ for name in "Bob" "Peter" "$USER" "Big Bad John"; do echo "$name";
done
```

مثال اول یک آرایه به نام names ایجاد می‌کند، که با چند عضو پر شده است. سپس آرایه به این عناصر بسط می‌یابد، که بعد توسط حلقه for به کار می‌روند. در مثال دوم، از آرایه صرف نظر کرده‌ایم و لیست اقلام را به طور مستقیم به حلقه for داده‌ایم.

به خاطر داشته باشید، بسط ${arrayname[@]} را به طور صحیح نقل‌قولی نمایید. در غیر اینصورت، تمام مزایای استفاده از آرایه را از دست می‌دهید: رها کردن شناسه‌های غیر نقل‌قولی، به معنی آنست که به Bash برای تفکیک آنها به قطعات و جداسازی دوباره آنها تأییدیه می‌دهید.

مثال فوق آرایه را در یک ساختار حلقه for بسط می‌دهد. اما می‌توانید آرایه را در هرجایی که بخواهیدعناصر آن را به عنوان شناسه قرار دهید، بسط بدهید، در یک فرمان cp :

```
myfiles=(db.sql home.tbz2 etc.tbz2)
cp "${myfiles[@]}" /backups/
```

این مثال، دستور cp را، با تعویض عبارت "${myfiles[@]}" با همه نام فایل‌های موجود در آرایه myfiles اجرا می‌نماید، نقل‌قول شده صحیح. پس از انجام بسط، Bash به طور مؤثر دستور زیر را اجرا می‌کند:

```
cp "db.sql" "home.tbz2" "etc.tbz2" /backups/
```

فرمان cp فایلها را به دایرکتوری */backups/* شما کپی خواهد نمود.

همچنین می‌توانیدعناصر منفرد آرایه را با ارجاع به شماره عضویت آنها(که index یا شاخص نام دارد)، بسط بدهید. به خاطر داشته باشید، که به طور پیش‌فرض،آرایه‌ها *zero-based* می‌باشند، یعنی شماره شاخص اولین عضو آنها صفر می‌باشد:

```
$ echo "The first name is: ${names[0]}"
$ echo "The second name is: ${names[1]}"
```

(می‌توانید آرایه‌ای بدون عضو شماره صفر ایجاد کنید. آنچه قبلاً در مورد *آرایه‌های پراکنده* گفتیم را به خاطر بیاورید ـــشما می‌توانید بین شاخص‌ها حفره داشته باشیدـــ، و این مطلب در ابتدای آرایه نیز به همان خوبی صدق می‌کند. این وظیفه شما به عنوان برنامه‌نویس است که بدانیدکدام‌یک از آرایه‌های شما به طور بالقوه پراکنده است، و کدام‌یک اینطور نیست.)

روش دیگری نیز برای بسط تمام عناصر آرایه وجود دارد، که به شکل "${arrayname[*]}" می‌باشد. این شکل **فقط** برای تبدیل آرایه به یک رشته منفردکه تمام عناصر آرایه در آن باهم متصل گردیده‌اند، مفید می‌باشد. مقصود اصلی در این روش ارائه خروجی آرایه به اشخاص می‌باشد:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
$ echo "Today's contestants are: ${names[*]}"
Today's contestants are: Bob Peter lhunath Big Bad John
```

توجه نمایید که در رشته حاصل شده، راهی برای گفتن آنکه نامها، کجا شروع و کجا ختم گردیده‌اند، وجود ندارد! این است چرایی آنکه، هر چیزی را تا آنجا که ممکن است، جدا نگاه می‌داریم.

به خاطر داشته باشید، هنوز هم به دقت نقل‌قولی نمایید! اگر ${arrayname[*]} را نقل‌قولی نکنید، یکبار دیگر *تفکیک کلمه* Bash موجب بریدن آن به تکه‌ها می‌گردد.

می‌توانید متغیر IFS را با ${arrayname[*]} ترکیب کنید، که نشان بدهید از چه کاراکتری برای جدا کردن عناصر آرایه از یکدیگر، موقعی که آنها را در یک رشته منفرد ادغام می‌کنید، استفاده شود. برای مثال، وقتی می‌خواهید نامها با کاراکتر **کاما** از هم جدا شوند، به راحتی به صورت زیر انجام می‌گردد:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
```

```
$ ( IFS=,; echo "Today's contestants are: ${names[*]}" )
Today's contestants are: Bob,Peter,lhunath,Big Bad John
```

توجه نمایید که در این مثال، چگونه جمله IFS=,; echo ... را با قرار دادن بین (و) در یک Subshell یا پوسته فرعی اجرا نمودیم. چنین کردیم زیرا نمی‌خواهیم مقدار پیش‌فرض متغیر IFS در پوسته اصلی را تغییر بدهیم. موقعی که پوسته فرعی خارج می‌شود، متغیر IFS بازهم مقدار پیش‌فرض را دارد، و دیگر کاما نمی‌باشد. این اهمیت دارد، به دلیل آنکه متغیر IFS برای موارد بسیاری استفاده می‌شود، و تغییر مقدار آن به چیزی غیر از مقدار پیش‌فرض، رفتار غریبی را که انتظار آن را ندارید، موجب خواهد شد!

افسوس، بسط "${array[*]}" فقط کاراکتر *اول* از متغیر IFS را برای بهم پیوستن عناصر با یکدیگر به کار می‌گیرد. اگر در مثال قبل، می‌خواستیم نامها را با یک کاما و یک فاصله از یکدیگر جدا کنیم، می‌باید برخی تکنیک‌های دیگر را به کار می‌بردیم(به عنوان مثال، یک حلقه for).

فرمان printf در اینجا سزاوار یک یادآوری می‌باشد، زیرا روش فوق العاده برازنده نسخه برداری از یک آرایه است:

```
$ names=("Bob" "Peter" "$USER" "Big Bad John")
$ printf "%s\n" "${names[@]}"
Bob
Peter
lhunath
Big Bad John
```

البته یک حلقه for نهایت انعطاف‌پذیری را ارائه می‌نماید، اما printf و حلقه ضمنی آن روی شناسه‌ها، می‌تواند بسیاری موارد ساده‌تر را پوشش بدهد. حتی می‌تواند جریانهای جدا شده با بایت تهی را، برای بازیابی بدون نقص بعدی تولید کند:

```
$ printf "%s\0" "${myarray[@]}" > myfile
```

یک نکته پایانی: شما می‌توانید تعداد عناصر یک آرایه را با استفاده از ${#array[@]} به دست آورید.

```
$ array=(a b c)
$ echo ${#array[@]}
3
```

تکرار مفید:

همیشه بسط آرایه‌ها را به طور صحیح نقل‌قولی کنید، درست همانطور که بسط پارامترهای معمولی را نقل‌قولی می‌کنید .

از ${#myarray[@]} برای بسط تمام عناصر آرایه استفاده کنید و ${#myarray[*]} را فقط موقعی که می‌خواهید همه عناصر آرایه را در یک رشته منفرد با یکدیگر متصل کنید، به کار ببرید.

آرایه‌های انجمنی

تا همین اواخر، BASH فقط از اعداد(به طور دقیقتر، اعداد صحیح مثبت) می‌توانست برای شاخص آرایه‌ها استفاده کند. به این معنی که نمی‌توانستید یک رشته را با دیگری ترجمه یا ترسیم کنید . این به عنوان یک کمبود احساس می‌شد. اشخاصی به منظور آدرس‌دهی به یک موضوع، سوءمصرف از متغیرهای غیرمستقیم را آغاز کردند .

پس از انتشار BASHنگارش 4، دیگر بهانه‌ای برای استفاده از متغیر غیر مستقیم(یا بدتر از آن، `eval`) برای این منظور نیست. اکنون شما می‌توانید آرایه‌های انجمنی خوش‌ساخت را به کار ببرید.

برای ایجاد یک آرایه انجمنی، باید آرایه به صورت(`declare -A`) تعریف شود. این برای هماهنگی با تعریف آرایه‌های استاندارد شاخص گذاری شده است. در اینجا چگونگی انجام آن، آمده است:

```
$ declare -A fullNames
$ fullNames=( ["lhunath"]="Maarten Billemont" ["greycat"]="Greg Wooledge" )
$ echo "Current user is: $USER.  Full name: ${fullNames[$USER]}."
Current user is: lhunath.  Full name: Maarten Billemont.
```

با همان دستور زبانی که برای آرایه‌های شاخص‌دار استفاده می‌شد، می‌توانید تکرار روی کلیدهای آرایه‌های انجمنی را انجام دهید:

```
$ for user in "${!fullNames[@]}"
> do echo "User: $user, full name: ${fullNames[$user]}."; done
User: lhunath, full name: Maarten Billemont.
User: greycat, full name: Greg Wooledge.
```

در اینجا دو مورد یادآوری: اول، ترتیب بازیابی کلیدها از یک آرایه انجمنی، با کاربرد ترکیب دستوری **${!array[@]}** غیرقابل پیش‌بینی است، و لزوماً به همان ترتیب که شما اعضاء را اختصاص داده‌اید، یا هر نوع ذخیره مرتب دیگر نمی‌باشد.

دوم، وقتی از پارامترها به عنوان کلید آرایه انجمنی استفاده می‌کنید، نمی‌توانید از علامت **$** صرف‌نظر کنید. با آرایه‌ها شاخص‌دار معمولی،قسمت **[...]** در حقیقت یک مفهوم محاسباتی است(در آنجا به راستی، می‌توانید بدون یک علامت‌گذاری صریح **$((...))** محاسبه انجام دهید(. در یک زمینه محاسباتی، یک *نام* به هیچ وجه نمی‌تواند عدد معتبری باشد، و بنابراین BASH فرض می‌کند، آن یک پارامتر است که شما می‌خواهید از محتوای آن استفاده کنید. این مورد در آرایه‌های انجمنی صدق نمی‌کند، چون در اینجا یک *نام* نیز به خوبی می‌تواند یک کلید معتبر آرایه انجمنی باشد.

اجازه دهید با مثال تشریح کنیم:

```
$ indexedArray=( "one" "two" )
$ declare -A associativeArray=( ["foo"]="bar" ["alpha"]="omega" )
$ index=0 key="foo"
```

```
$ echo "${indexedArray[$index]}"
one
$ echo "${indexedArray[index]}"
one
$ echo "${indexedArray[index + 1]}"
two
$ echo "${associativeArray[$key]}"
bar
$ echo "${associativeArray[key]}"
$
$ echo "${associativeArray[key + 1]}"
$
```

به طوری که می‌توانید ملاحظه کنید، هم $index و هم index به خوبی با آرایه‌های معمولی کار می‌کنند. هر دو، عدد *0* ارزیابی می‌شوند. حتی می‌توانید برای اضافه کردن *1* به آن و به دست آوردن مقدار دوم، با آن محاسبه کنید. آنچه که با آرایه‌های انجمنی نمی‌تواند انجام شود. در اینجا، لازم است $key به کار برود، آن دیگری کار نمی‌کند.

ورودی و خروجی

فهرست مطالب

ورودی و خروجی در اسکریپت‌های Bash مبحث پیچیده‌ایست، زیرا انعطاف‌پذیری بسیار زیادی در چگونگی انجام آن، وجود دارد. این فصل فقط یک ارائه سطحی از آنچه ممکن است، می‌باشد.

ورودی به هر اطلاعاتی که برنامه شما دریافت می‌کند(یا می‌خواند) اشاره می‌نماید. در یک اسکریپت Bash ورودی از چند محل مختلف می‌تواند برسد:

- شناسه‌های خط فرمان(که در پارامترهای مکانی قرارگرفته‌اند)
- متغیرهای محیطی، موروثی از هر پردازشی که اسکریپت را آغاز نموده
- فایلها
- هر چیز دیگری که یک *توصیف‌گر فایل* می‌تواند به آن اشاره کند(لوله‌ها،ترمینال‌ها، سوکت‌ها، و غیره). این موارد جلوتر بحث خواهند شد.

خروجی به هر اطلاعاتی که برنامه شما ارائه می‌کند(یا می‌نویسد) اشاره می‌کند. خروجی یک اسکریپت Bash نیز می‌تواند به چندین محل مختلف برود:

- فایلها
- هر چیز دیگر که یک توصیف‌گر فایل می‌تواند به آن اشاره کند
- شناسه‌های خط فرمانی برای سایر برنامه‌ها
- متغیرهای محیطی که به سایر برنامه‌ها ارسال می‌شوند

ورودی و خروجی، نان و پنیر اسکریپت‌نویسی پوسته هستند. معین کردن اینکه ورودی شما ازکجا می‌آید، چگونه به نظر می‌رسد، و شما برای بدست آوردن خروجی مطلوب خود، چه کاری باید روی آن انجام بدهید، هسته مرکزی احتیاجات تقریباً تمام اسکریپت‌ها می‌باشند.

1. شناسه‌های خط فرمان

برای بسیاری از اسکریپت‌ها، اولین(یا تنها) ورودی که به آن توجه می‌نماییم، شناسه‌هایی می‌باشند که اسکریپت در خط فرمان دریافت نموده است. به طوری که در فصل پارامترها دیدیم، تعدادی *پارامتر ویژه* معتبر برای هر اسکریپت وجود دارد که، محتوی این شناسه‌ها هستند. اینها *پارامترهای مکانی* نام دارند. این پارامترها یک آرایه خیلی ساده از رشته‌ها می‌باشند که با اعداد شاخص‌گذاری شده‌اند(در حقیقت، در شل POSIX ، *تنها* آرایه موجود در شل هستند). به اولین پارامتر مکانی با $1 رجوع می‌شود، به دومین، با $2، و به همین ترتیب، پس از نهمین پارامتر، باید از ابروها برای رجوع به آنها استفاده گردد: ${10}$, ${11}$، و غیره. اما در عمل، خیلی به ندرت ممکن است مجبور به استفاده از این ترکیب بشوید، به علت آنکه، روشهای بهتری برای کار با آنها به عنوان یک گروه، موجود است.

علاوه بر ارجاع یک به یک ، همچنین می‌توانید به مجموعه کامل پارامترهای مکانی، با جایگزینی "$@" رجوع نمایید. نقل‌قول دوگانه در اینجا بینهایت مهم است. اگر از نقل‌قول دوگانه استفاده نکنید، هر کدام از پارامترهای مکانی دستخوش تفکیک و جانشینی می‌گردند. شما آن را نمی‌خواهید. با استفاده از نقل‌قول‌ها، به Bash می‌گویید که می‌خواهید هر پارامتر به صورت یک کلمه جداگانه حفظ گردد.

یک روش دیگر کارکردن با پارامترهای مکانی، دور انداختن هر یک پس از استفاده است. یک دستور داخلی ویژه‌ای به نام `shift` وجود دارد، که برای این منظور به کار می‌رود. موقعی که شما فرمان `shift` را صادر می‌کنید، اولین پارامتر مکانی ($1) از بین می‌رود. دومین پارامتر می‌شود $1، سومی می‌شود $2، و به همیت ترتیب تا پایان خط. بنابراین، اگر مایل باشید، می‌توانید حلقه‌ای بنویسید که استفاده از $1 را چندین بار ادامه دهد.

در اسکریپت‌های حقیقی، ترکیبی از این تکنیک‌ها به کار می‌رود. یک حلقه برای پردازش $1 مادامی‌که با یک علامت - شروع می‌شود آن را به عنوان گزینه تعبیر می‌کند و سپس وقتی همه گزینه‌ها به کنار رفتند، هر چیز باقیمانده (در "$@") نام فایلی است که می‌خواهیم پردازش کنیم.

برای اختصار، در اینجا مثالهایی از پردازش شناسه‌ها نمی‌آوریم. به جای آن، به FAQ جایی که مثالهای آنها قبلاً نوشته شده ارجاع می‌دهیم.

تکرار مفید:

قبل از شروع به نوشتن، شناسایی کنید که ورودی برنامه شما از کجا می‌آید. اگر می‌خواهید اطلاعاتی را به اسکریپت خود ارسال کنید، روشی برای معنی کردن نوع اطلاعاتی که با آن سر و کار دارید، انتخاب نمایید. اگر احتیاج به ارسال نام فایلها دارید، ارسال آنها به صورت شناسه‌ها، یک رویکرد عالی است، زیرا هر یک از آنها به صورت یک کلمه بسته‌بندی می‌گردد.

در پرسش و پاسخ‌های رایج:

چگونه می‌توانم شناسه‌های(گزینه‌های) خط‌فرمان را به آسانی مدیریت کنم؟

---

2. محیط

هر برنامه‌ای اطلاعات، منابع، امتیازها و محدودیت‌هایی از پردازش والد خود به ارث می‌برد. (برای بحث پیشرفته‌تر در این موضوع، بخش مدیریت پردازش را ملاحظه کنید.)یکی از آن منابع، مجموعه‌ای از متغیرها به نام *متغیرهای محیط* می‌باشند.

در Bash, متغیرهای محیط تا اندازه بسیار زیادی مشابه متغیرهای معمولی مورد استفاده ما، کار می‌کنند. تنها تفاوت واقعی، آن است که آنها قبلاً، موقعی که اسکریپت شروع به اجرا می‌کند، مقرر شده‌اند، ما خودمان نباید به آنها مقداردهی کنیم.

به طور سنتی، متغیرهای محیط نامهایی تماماً با حروف بزرگ دارند، از قبیل PATH یا HOME. این مطلب به شما کمک می‌کند، از ایجاد متغیرهایی که موجب تصادم با آنها گردد، اجتناب نمایید، شما نباید نگرانی در مورد تصادم اتفاقی با محیط داشته باشید، مشروط به آن که متغیرهای شما، حداقل یک حرف کوچک در نام خود داشته باشند . (متغیرهای ویژه Bash نیز با حروف بزرگ هستند، از قبیل PIPESTATUS. این مورد نیز دقیقاً به همان علت می‌باشد ـــ به طوری که شما می‌توانید از لگدمال شدن متغیرهایتان توسط Bash، پرهیز کنید.)

ارسال اطلاعات به برنامه‌ها از طریق متغیرهای محیط، در بسیاری موقعیت‌ها سودمند است. یکی از آنها اولویت‌های کاربر است. در سیستم‌های یونیکسی، همه کاربران علایق و بی‌علاقگی‌های یکسان در برنامه‌های کاربردی ندارند، و در برخی موارد، ممکن است همه به یک زبان صحبت نکنند. بنابراین، برای کاربران سودمند خواهد بود که به هر برنامه بگویند کدام ویرایشگر مورد علاقه آنها برای اجرا می‌باشد(متغیر محیطی EDITOR)، یا به چه زبانی صحبت می‌کنند(متغیرهای محیطی گوناگون که منطقه کاربر را معین می‌کنند). متغیرهای محیطی می‌توانند در فایلهای نقطه‌ای هر کاربرتنظیم شوند، و سپس اینها به طور خودکار به هر برنامه‌ای که کاربر از یک نشست کاری اجرا می‌کند، ارسال می‌شوند.

متغیرهای محیط همچنین می‌توانند به طور بینهایت آسانی در حین کار تنظیم گردند(آسان‌تر از آنکه اگر همان اطلاعات در یک فایل ذخیره شده باشند). موقعی که در Bash دستوری را اجرا می‌کنید، گزینه‌ای دارید، برای تعیین یک تغییر موقتی محیط که فقط در طول مدت اجرای آن فرمان مؤثر است. این با قرار دادن عبارت VAR=value جلوی آن فرمان انجام می‌شود. در اینجا یک مثال آورده‌ایم:

```
$ ls /tpm
ls: no se puede acceder a /tpm: No existe el fichero o el
directorio
$ LANG=C ls /tpm
ls: cannot access /tpm: No such file or directory
```

محیط موقتی LANG=C باعث نمی‌شود که منطقه کاربر برای مورد دیگری غیر از آن فرمانی که بعد از آن تایپ گردیده است، تغییر نماید.

اگر می‌دانید که برخی اطلاعات در یک متغیر محیط ذخیره شده است، در اسکریپت، می‌توانید درست مانند سایر متغیرها از آن استفاده کنید:

```
if [[ $DISPLAY ]]; then
    xterm -e top
else
    top
fi
```

این مثال، در صورتی که متغیر محیطی DISPLAY تنظیم شده باشد(و تهی نباشد) دستور xterm -e top را اجرا می‌کند، در غیر آن صورت، دستور top را اجرامی‌نماید.

اگر می‌خواهید اطلاعاتی را در متغیر محیطی قرار دهید که به پردازش‌های فرزند به ارث برسد، فرمان export را به کار ببرید:

```
export MYVAR=something
```

بخش دشوار مطلب در اینجا آنست که تغییرات محیط شما فقط برای فرزندان موروثی خواهد بود. نمی‌توانید محیط یک برنامه را که از قبل در حال اجراست، یا شما آن را شروع نکرده‌اید، تغییر دهید.

تغییر محیط و سپس اجرا، برای برخی برنامه‌ها به شدت رایج می‌باشد. اسکریپتی که این کار را به عنوان وظیفه اصلی‌اش انجام می‌دهد یک WrapperScript می‌نامند.

تکرار مفید:

در اسکریپت‌های خود از نامهای تماماً با حروف بزرگ برای متغیرها استفاده نکنید. برای پرهیز از تصادمات، حروف کوچک یا ترکیبی از کوچک و بزرگ به کار ببرید .

در پرسش و پاسخ‌های رایج:

من سعی می‌کنم اسکریپتی بنویسم که دایرکتوری جاری را تغییر دهد(یا یک متغیر را تنظیم کند)، اما بعد از به پایان رسیدن اسکریپت، در همان جایی هستم که از آنجا شروع کرده بودم(یا متغیر من موجود نیست)!

---

3. توصیف‌گرهای فایل

توصیف‌گرهای فایل (به طور کوتاه: FDها) روشی برای ارجاع برنامه‌ها به فایلها، یا منابع دیگری که همانند فایلها کارمی‌کنند(از قبیل لوله‌ها،دستگاهها، سوکت‌ها، یا ترمینال‌ها) می‌باشند. FDها نوع مشابه اشاره‌گرها به منابع داده، یا محل‌هایی که اطلاعات می‌توانند نوشته شوند، هستند. موقعی که چیزی از آن FD, خوانده یا در آن نوشته می‌شود، داده در حال نوشته شدن در منبع FD یا خوانده شدن از آن می‌باشد.

به طور پیش‌فرض، هر فرایند جدیدی با سه FD آغاز می‌شود. به این توصیف‌گرهای فایل با نامهای *ورودی استاندارد*، *خروجی استاندارد* و *خطای استاندارد* رجوع می‌شود.در حالت کوتاه شده به ترتیب stdin و stdout و stderr نامیده می‌شوند. در یک پوسته محاوره‌ای، یا اسکریپت درحال اجرا در ترمینال، *ورودی استاندارد* طریقی است که bash کاراکترهای تایپ شده توسط شما در صفحه کلیدتان را می‌بیند. *خروجی استاندارد* جایی است که برنامه اکثر اطلاعات معمولی‌اش را به طوری که کاربر بتواند آن را ببیند، ارسال می‌کند، و *خطای استاندارد* جایی است که برنامه پیغام‌های خطایش را می‌فرستد. آگاه باشید که برنامه‌های کاربردی رابط گرافیکی هنوز به همین روش عمل می‌کنند، اما رابط گرافیکی واقعی کاربر از طریق این FDها کار نمی‌کند.برنامه‌های رابط گرافیکی هنوز می‌توانند از FDهای استاندارد بخوانند یا در آنها بنویسند، اما به طور معمول چنین نمی‌کنند. معمولاً، آنها تمام ارتباط متقابل با کاربر را از راه آن GUI انجام می‌دهند، که کنترل آن برای BASH را دشوار می‌سازد. در نتیجه، ما در برنامه‌های کاربردی ساده ترمینال خواهیم ماند. برنامه‌هایی، که به آسانی داده‌ها را به ورودی استانداردشان بخورانیم، و داده‌ها را از خروجی استاندارد و خطای استانداردشان بخوانیم.

اجازه بدهید کمی این تعاریف را محسوس‌تر کنیم. در اینجا یک نمایش تجربی از چگونگی کار *ورودی استاندارد* و *خروجی استاندارد* می‌آوریم:

```
$ read -p "What is your name? " name; echo "Good day, $name.  Would you like some tea?"
What is your name? lhunath
Good day, lhunath.  Would you like some tea?
```

read دستوری است که اطلاعات را از stdin می‌خواند و در یک متغیر ذخیره می‌کند. ما تعیین نموده‌ایم که name آن متغیر باشد. وقتی read یک سطر اطلاعات را از stdin بخواند، خاتمه یافته و echo اجازه می‌یابد یک پیغام نمایش دهد. echo از stdout برای ارسال خروجی‌اش استفاده می‌کند. stdin به دستگاه ورودی پایانه شما که احتمالاً صفحه کلید شما می‌باشد، متصل گردیده است. stdout به دستگاه خروجی پایانه شما متصل گردیده است، که ما فرض می‌کنیم نمایشگر کامپیوتر شما است. در نتیجه، شما می‌توانید نام خود را تایپ نموده و با یک پیغام خوش‌آمد گویی دوستانه در نمایشگر خود به یک فنجان چای دعوت شوید.

پس stderr چیست؟ اجازه دهید نمایش بدهیم:

```
$ rm secrets
rm: cannot remove `secrets': No such file or directory
```

بدون داشتن فایلی به نام secrets در دایرکتوری جاری خودتان، آن دستور rm با شکست مواجه خواهد شد و یک پیغام خطا در تشریح آنچه اشتباه است، نمایش می‌دهد. چنین پیغام خطاهایی بر حسب قرارداد در stderr نمایش داده می‌شوند. stderr نیز به دستگاه خروجی پایانه شما متصل گردیده است، درست مانند stdout. در نتیجه، پیغام‌های خطا در نمایشگر شما نشان داده می‌شوند درست مانند پیغام‌ها در stdout. به هرحال، این افتراق، جدانگاه داشتن خطاها از پیغام‌های معمول برنامه‌ها را تسهیل می‌کند. بعضی افراد مایل هستند تمام خروجی در stderr را به رنگ قرمز بسته‌بندی کنند، به طوری که بتوانند پیغام‌های خطا را واضح‌تر ببینند. این به طور کلی قابل توصیه نیست، اما یک مثال ساده ازامکانات بسیاری است که این جدایی برای ما فراهم می‌کند. تکنیک رایج دیگر، نوشتن stderr در یک فایل ثبت وقایع(log file) خاص است.

تکرار مفید:

به خاطر داشته باشید موقعی که اسکریپت‌ها را ایجاد می‌کنید،شما باید پیغام خطاهای سفارشی خود را به توصیف‌گر stderr ارسال نمایید. این یک قرارداد است و پیروی برنامه‌های کاربردی از قرارداد بسیار مناسب است. به همین ترتیب, به طوری‌که به زودی در باره تغییر مسیر خواهید آموخت، اما اجازه دهید من به سرعت اشاره‌ای به آن داشته باشم:

```
echo "Uh oh.  Something went really bad.." >&2
```

توصیف‌گر فایل: یک شاخص عددی ارجاع به یکی از فرآیندهای فایل باز است. هر دستوری حداقل سه توصیف‌گر اصلی دارد: FD شماره 0، stdin است، FD شماره 1، stdout است و FD شماره 2، stderr می‌باشد.

---

4. تغییر مسیر

اساسی ترین شکل دستکاری ورودی و خروجی در BASH *تغییر مسیر* است. *تغییر مسیر* برای تغییر منبع داده یا مقصد توصیف‌گرهای فایل یک برنامه کاربردی به کار می‌رود. به طریقی، که می‌توانیدخروجی برنامه را به جای ترمینال به یک فایل ارسال کنید، یا برنامه‌ای داشته باشید که به جای خواندن از صفحه کلید از یک فایل بخواند.

همچنین، تغییر مسیر به اشکال مختلف می‌باشد. که عبارتند از *تغییر مسیر فایل*، *دستکاری توصیف‌گرفایل*، *Heredocها* و *Herestringها*.

در مستندات گنو: Redirections

تغییر مسیر: شیوه تغییر یک توصیف‌گرفایل معین برای خواندن وردی‌اش از جای دیگر یا ارسال خروجی‌اش به جایی دیگر می‌باشد

4.1. تغییر مسیر فایل

تغییر مسیر فایل تغییر یک توصیف‌گر فایل منفرد برای اشاره به یک فایل را شامل می‌گردد. بیایید با یک تغییر مسیر خروجی شروع کنیم:

```
$ echo "It was a dark and stormy night.  Too dark to write." > story
$ cat story
It was a dark and stormy night.  Too dark to write.
```

عملگر > یک *تغییر مسیر خروجی* را شروع می‌کند. تغییر مسیر فقط بر یک فرمان اِعمال می‌گردد(در این حالت، یک دستور echo). این عملگر به Bash می‌گوید که وقتی آن فرمان را اجرا می‌کند، stdout در عوض جایی که قبلاً اشاره می‌کرد، باید به یک فایل اشاره کند.

در نتیجه، فرمان echo خروجی‌اش را به ترمینال ارسال نمی‌کند، به جای آن، تغییر مسیر > story ، مقصد توصیف‌گر stdout را تغییر می‌دهد به طوری که، حالا به فایلی به نام story اشاره می‌کند. هشیار باشید که این تغییر مسیر قبل از اینکه فرمان echo اجرا بشود، صورت می‌گیرد. به طور پیش‌فرض، BASH اول بررسی نمی‌کند که آیا آن فایل story وجود دارد یا خیر، فقط فایل را باز می‌کند، و اگر از قبل فایلی با آن نام وجود داشته باشد، محتویات قبلی آن از بین می‌رود. اگر فایل وجود نداشته باشد، به صورت یک فایل تهی ایجاد می‌گردد، طوری که آن FDبتواند به آن اشاره کند. این رفتار می‌تواند با *گزینه‌های شل* تغییر وضعیت یابد(بعداً می‌بینید).

بایستی توجه گردد که این تغییر مسیر فقط برای دستور منفرد echo که برایش به کار برده شده، مؤثر است . دستورات دیگری که بعد اجرا می‌شوند، ارسال خروجی خود به محل stdout اسکریپت را ادامه می‌دهند.

سپس ما از برنامه cat برای چاپ محتویات آن فایل استفاده کردیم. cat برنامه کاربردی است که محتویات تمام فایلهایی که به عنوان شناسه به آن داده‌ایم را می‌خواند. سپس هر فایل را یکی پس از دیگری به stdout ارسال می‌کند. در اساس، این برنامه محتویات همه فایلهایی که به عنوان شناسه به آن داده‌اید را به یکدیگر *con**cat**enate*(الحاق) می‌کند.

هشدار:مثالهای بسیار زیادی از کد ها و خودآموزهای شل در اینترنت به شما می‌گویند، هر جا احتیاج به خواندن محتویات یک فایل دارید از cat استفاده کنید. این کار ضرورت ندارد! cat در الحاق چند فایل به یکدیگر خوب خدمت می‌کند، یا به عنوان یک ابزار سریع در اعلان شل برای دیدن آنچه داخل یک فایل است. شما نباید از cat در اسکریپت‌های خود برای لوله‌کشی فایلها به دستورات، استفاده کنید. تقریباً همیشه راههای بهتری برای انجام این کار وجود دارد. لطفاً این هشدار را به خاطر بسپارید. استفاده بی‌مورد از cat صرفاً به ایجاد یک پردازش اضافی منجر خواهد شد، و غالباً سرعت خواندن ضعیف‌تری را نتیجه می‌دهد، زیرا cat نمی‌تواند مفهوم آنچه می‌خواند و هدفی که داده، برای آن خوانده می‌شود را تشخیص دهد.

موقعی که ما برنامه cat را بدون هرگونه شناسه‌ای به کار می‌بریم، به طور آشکاری نمی‌داند که کدام فایل را بخواند. در این وضعیت، cat به جای خواندن از یک فایل، فقط از stdin خواهد خواند(بیشتر همانند read). از آنجاییکه stdin یک فایل عادی نیست، شروع cat بدون هرگونه شناسه، به نظر می‌رسد که کاری انجام نمی‌دهد:

```
$ cat
```

حتی اعلان فرمان را به شما باز نمی‌گرداند! چه اتفاقی رخ می‌دهد؟ cat هنوز در حال خواندن از stdin می‌باشد، که صفحه کلید شماست. حالا هر چیزی که شما تایپ کنید به cat فرستاده خواهد شد. به محض اینکه شما کلید *Enter* را بزنید، cat همان کاری را خواهد کرد، که به طور معمول انجام می‌دهد: آنچه را خوانده در خروجی استاندارد نمایش خواهد داد، دقیقاً به همان طریقی که فایل story ما را در stdout نمایش داد:

```
$ cat
test?
test?
```

حالا چرا test? را دو مرتبه نمایش می‌دهد؟ خوب، همانطور که شما تایپ می‌کنید، ترمینال شما تمام کاراکترهایی را که به stdin می‌فرستید، قبل از ارسال آنها به آنجا، به شما نمایش می‌دهد. نتیجه آن اولین test? است که شما می‌بینید. به محض اینکه کلید *Enter* را می‌زنید، cat یک سطر از stdin می‌خواند، وآن را در stdout نمایش می‌دهد، که آن نیز ترمینال شماست، از این‌جهت، سطر دوم : test?. می‌توانید با فشردن *Ctrl+D* کاراکتر *انتهای فایل* را به ترمینال ارسال کنید. که باعث می‌گردد، cat گمان کند stdin بسته شده است. خواندن را متوقف خواهد نمود و اعلان فرمان را به شما باز می‌گرداند.

بیایید یک *تغییر مسیر ورودی* را برای پیوست یک فایل به stdin به کار ببریم، به طوری که stdin دیگر از صفحه کلید ما نخواند، بلکه به جای آن، اکنون از فایل بخواند:

```
$ cat < story
The story of William Tell.

It was a cold december night.  Too cold to write.
```

نتیجه این دقیقاً همانند نتیجه cat story قبلی می‌باشد،غیر از اینکه این دفعه، روش عمل آن کمی متفاوت است. در مثال اول، cat یک FD برای فایل story باز کرد و محتویات فایل را از طریق آن FDخواند. در دومین مثال، cat واقعاً از stdin می‌خواند، دقیقاً مانند موقعی که از صفحه کلید ما خواند. به هرحال، این دفعه، عملگر story <، ورودی استاندارد را تغییر داده است، به طوری که منبع اطلاعات، به جای صفحه کلید ما فایل story می‌باشد.

عملگرهای تغییر مسیر میتوانند با تقدم یک عدد همراه گردند. آن عدد توصیف‌گرفایلی که تغییر می‌کند را مشخص می‌نماید.

اجازه دهید با چند مثال خلاصه کنیم:

- command > file: خروجی استاندارد فرمان را به file ارسال می‌کند.
- command 1> file: خروجی استاندارد فرمان را به file می‌فرستد. چون خروجی استاندارد در FD شماره 1 باز شده، عدد آن رابر عملگر تغییر مسیر مقدم نموده‌ایم. توجه،تأثیر این درست مانند مثال قبل است، زیرا FD شماره 1 پیش‌فرض عملگر > می‌باشد.
- command < file: موقعی که فرمان از stdin می‌خواند، محتویات file را به کار می‌برد.

- `command 0< file`: موقعی که فرمان از stdin می‌خواند، محتویات file را استفاده می‌کند، دقیقاً مانند مثال قبل، چون FD شماره 0 (stdin) پیش‌فرض عملگر < می‌باشد.

عدد مربوط به FD (توصیف‌گرفایل) stderr شماره 2 می‌باشد. بنابراین بیایید خروجی stderr را به یک فایل ارسال کنیم:

```
$ for homedir in /home/*
> do rm "$homedir/secret"
> done 2> errors
```

در این مثال، روی هر دایرکتوری(یا فایل) در شاخه*/home* حلقه ایجاد می‌کنیم. سپس ما سعی در حذف فایل secret در هر یک از آنها می‌نماییم. برخی *دایرکتوری‌های خانگی* ممکن است فایل secret نداشته باشند، یا ممکن است اجازه دسترسی برای حذف را نداشته باشیم. در نتیجه، عمل rm ناموفق خواهد بود و پیغام خطا به stderr ارسال خواهد شد.

ممکن است توجه نموده باشید که عملگرتغییر مسیر روی فرمان rm نمی‌باشد، بلکه روی done است. چرا اینطور است؟ خوب، به این ترتیب، تغییر مسیر برای تمام خروجیِ ایجاد شده در حلقه جهت stderr، می‌باشد. به طور تکنیکی، آنچه اتفاق می‌افتد آنست که، Bash فایلی به نام errors را باز می‌کند و stderr را قبل از اینکه حلقه شروع شود، به آن اشاره می‌دهد، سپس وقتی حلقه خاتمه می‌یابد، آن را می‌بندد. هر دستور اجرایی در داخل حلقه(از قبیل rm) از FD باز شده Bash *ارث می‌برد*.

اجازه بدهید ببینیم نتیجه حلقه ما چه بوده:

```
$ cat errors
rm: cannot remove `/home/axxo/secret': No such file or directory
rm: cannot remove `/home/lhunath/secret': No such file or
directory
```

دو پیغام خطا در فایل ثبت خطا. دو نفر که فایل secret در دایرکتوری خانگی‌اشان ندارند.

اگر یک اسکریپت می‌نویسید، و پیش‌بینی می‌کنید که یک دستور معین تحت شرایطی ممکن است ناموفق باشد، اما نمی‌خواهید کاربر اسکریپت، با خطاهایی که ممکن است دستور ایجاد کند پریشان گردد، می‌توانید FD آن را ساکت کنید. ساکت کردن آن به آسانی یک *تغییر مسیر فایل* معمولی است. فقط تمام خروجی برای آن FD را به سیاهچاله سیستم ارسال می‌کنیم:

```
$ for homedir in /home/*
> do rm "$homedir/secret"
> done 2> /dev/null
```

فایل */dev/null* همیشه خالی است، مسئله‌ای نیست که در آن چه می‌نویسید یا از آن می‌خوانید. همینطور، موقعی که ما پیغام خطاها را در آن می‌نویسیم، آنها ناپدید می‌شوند. فایل */dev/null* همچنان مانند قبل خالی می‌ماند. چنین است زیرا یک فایل معمولی نمی‌باشد، یک دستگاه *مجازی* است. بعضی‌ها */dev/null* را bit bucket[1] می‌نامند.

یک مطلب نهایی هست که باید در باره *تغییر مسیر فایل* بدانید. جالب است که شما می‌توانید یک فایل ثبت وقایع برای نگهداری پیغام‌های

خطایتان ایجاد کنید، اما همانطور که قبلاً اشاره کردم، BASH موقعی که به یک فایل تغییر مسیر داده می‌شود، محتویات موجود آن فایل را ازبین می‌برد. در نتیجه، هر دفعه که ما آن حلقه را برای حذف کردن فایلهای secret اجرا کنیم، فایل ثبت وقایع ماقبل از اینکه دوباره با پیغام‌های جدید پر شود،از سر کوتاه و خالی می‌شود. چه کار باید بکنیم، اگر می‌خواهیم رکوردی از هر پیغام خطای تولید شده در حلقه را حفظ کنیم؟ چه کار کنیم که با هر بار اجرای حلقه[زیرنویس مترجم 2] فایل از سر کوتاه نشود؟ چاره کار با دوگانه کردن عملگر تغییر مسیر به دست می‌آید. > می‌شود >>. عملگر >> فایل را خالی نمی‌کند بلکه فقط اطلاعات جدید را در انتهای فایل اضافه می‌کند!

```
$ for homedir in /home/*
> do rm "$homedir/secret"
> done 2>> errors
```

Hooray!

درضمن، فاصله بین عملگر تغییر مسیر و نام فایل، اختیاری است. بعضی افراد می‌نویسند file > و برخی می‌نویسند file>. هر دو روش صحیح است.

تکرار مفید:
وقتی که یک برنامه به فایل داده‌ای نیاز دارد، و برای خواندن داده از stdin ساخته می‌شود، تغییر مسیر ایده خوبی است. مقدار بسیار زیادی نمونه‌های بد در اینترنت هست، که به شما می‌گویند خروجی cat را به پردازش‌ها لوله‌کشی(بعد می‌خوانید) کنید، اما این چیزی بیش از یک ایده *نامساعد* نمی‌باشد.
موقعی که برنامه ای طراحی می‌نمایید که می‌تواند داده‌ها را از منابع مختلفی تغذیه کند، واقعاً اغلب بهترین راه آنست که برنامه شما از stdin بخواند، به این ترتیب، کاربر می‌تواند از تغییر مسیر برای تغذیه برنامه از هر آنچه مایل است، استفاده کند. یک برنامه کاربردی که ورودی استاندارد را از یک مسیر کلی می‌خواند *filter* نامیده می‌شود.

در مستندات گنو: Redirecting Input, Redirecting Output, Appending Redirected Output, Redirecting Standard Output and Standard Error

---

4.2. دستکاری توصیف‌گر فایل

حال که دانستید چگونه پردازش ورودی و خروجی را با فرستادن به فایلها یا دریافت از آنها اداره کنید، اجازه دهید باز هم کمی آن را جذاب‌تر نماییم.

همانطور که می‌دانید، تغییر منبع یا مقصد توصیف‌گرهای‌فایل برای اشاره به فایلها یا از آنها، امکان پذیر است. همچنین کپی کردن یک FD به دیگری ممکن است. اجازه دهید بستر آزمایش ساده‌ای فراهم کنیم:

```
$ echo "I am a proud sentence." > file
```

فایلی به نام file ساخته‌ایم، و یک جمله در آن نوشته‌ایم.

برنامه کاربردی به نام grep وجود دارد که به طور مختصر در فصل قبل دیده‌ایم. grep شبیه یک نوار لوله‌ای است: می‌توانید تقریباً در هر پروژه‌ای آن را به کار ببرید(خواه ایده خوبی باشد یا نباشد). اساساً یک *الگوی جستجو* را به عنوان یک شناسه دریافت می‌کند و می‌تواند چند نام فایل نیز به عنوان شناسه اضافی دریافت نماید. درست مانند cat، برنامه grep نیز اگر نام فایلی را به عنوان شناسه تعیین نکنید از stdin استفاده می‌کند. grep فایلها رامی‌خواند(یا stdin را در صورتکه فایلی مشخص نشده باشد) و *الگوی جستجویی* که به آن داده‌اید را جستجو می‌کند. اکثر نگارش‌های grep حتی از یک گزینه -r نیز پشتیبانی می‌کنند، که موجب می‌گردد دایرکتوریها را نیز مانند فایلها به عنوان شناسه اضافی قبول کنند، و سپس تمام فایلها و دایرکتوری‌های داخل آن شاخه‌ها را که به آن داده‌اید، جستجو می‌کند. در اینجا مثالی برای آن که grep چگونه می‌تواند عمل کند می‌آوریم:

```
$ ls house/
drawer  closet  dustbin  sofa
$ grep -r socks house/
house/sofa:socks
```

در این مثال تخیلی، دایرکتوری به نام *house* با چند قطعه اثاثیه منزل در آن به عنوان فایل داریم. اگر ما در جستجوی *socks*(*ساک‌ها*)ی خود در هر یک از آن فایلها باشیم، grep را به جستجو در دایرکتوری *house/* می‌فرستیم. grep هر چیزی در آنجا را جستجو می‌کند، هر فایل را باز می‌کند و محتویات آن را نگاه می‌کند. در مثال ما، grep ساک‌ها(*socks*) را در فایل *house/sofa* پیدا می‌کند،احتمالاً پنهان شده زیر یک بالش. مثال واقع‌بینانه‌تری می‌خواهید؟ حتماً:

```
$ grep "$HOSTNAME" /etc/*
/etc/hosts:127.0.0.1       localhost Lyndir
```

اینجا به grep دستور داده‌ایم هر آنچه $HOSTNAME به آن بسط می‌یابد را در هر فایلی که */etc/** به آن بسط می‌یابد، جستجو نماید. نام میزبانم را که Lyndir است در فایل */etc/hosts* پیدا می‌کند، و سطری از آن فایل را که شامل *الگوی جستجو* است، به من نشان می‌دهد.

خوب، حال که grep را دریافتید, اجازه بدهید با *دستکاری توصیف‌گرفایل* ادامه دهیم. به خاطر می‌آورید که فایلی به نام file ایجاد کردیم، و یک جمله شامل proud در آن نوشتیم؟ اینک اجازه بدهید grep را برای یافتن جمله proud به کار ببریم:

```
$ grep proud *
file:I am a proud sentence.
```

صحیح! grep جمله ما را در file یافته است. نتیجه عمل خود را در stdout که در ترمینال ما نمایش داده می‌شود، می‌نویسد. حالا بیایید ببینیم آیا می‌توانیم با grep یک پیغام خطا نیز ارسال کنیم:

```
$ grep proud file 'not a file'
file:I am a proud sentence.
grep: not a file: No such file or directory
```

این دفعه، به grep دستور داده‌ایم رشته *proud* را درفایلهای 'file' و '*not a file*' جستجو کند. file وجود دارد، و جمله در آن هست، بنابراین grep با خوشحالی نتیجه را در stdout می‌نویسد. برای بررسی سراغ فایل بعدی که '*not a file*' است، می‌رود. grep نمی‌تواند چنین فایلی را برای خواندن باز کند، به دلیل آنکه وجود ندارد. در نتیجه، grep یک پیغام خطا به stderr که بازهم به ترمینال ما متصل است، ارسال می‌کند.

حال، چطور می‌توان این جمله grep را به طور کامل خاموش کنیم؟ می‌توانیم تمام خروجی که روی ترمینال ظاهر می‌شود را به جای آن به یک فایل بفرستیم، اجازه بدهید آن را proud.log بنامیم:

```
# Not quite right....
$ grep proud file 'not a file' > proud.log 2> proud.log
```

صحیح به نظر می‌آید؟ ابتدا از > برای ارسال stdout به فایل proud.log استفاده کرده‌ایم، و سپس از 2> هم برای ارسال stderr به فایل proud.log. تقریبا درست، اما نه کاملاً. اگر این دستور را اجرا نمایید وسپس به فایل proud.log نگاه کنید، (لااقل در برخی سیستم‌ها) فقط یک پیغام خطا خواهید دید، نه خروجی مربوط به stdout را. در اینجا ما شرایط خیلی نامناسبی ایجاد نموده‌ایم. دو FD ایجاد کرده‌ایم که هر دو به طور مستقل از یکدیگر به یک فایل اشاره می‌کنند. نتایج آن به طور کامل معلوم نیست. بر اساس چگونگی مدیریت توصیف‌گرهای فایل توسط سیستم‌عامل شما، ممکن است بعضاً اطلاعات نوشته شده از طریق یک FD اطلاعات نوشته شده از طریق FD دیگر را پاک کنند.

```
$ echo "I am a very proud sentence with a lot of words in it, all for
you." > file2
$ grep proud file2 'not a file' > proud.log 2> proud.log
$ cat proud.log
grep: not a file: No such file or directory
    of words in it, all for you.
```

در اینجا چه اتفاقی رخ داده؟ grep اول فایل file2 را باز کرده، آنچه را به او گفته‌ایم جستجو کند، یافته و سپس جمله very proud ما را در stdout(یعنیFD شماره 1) نوشته است. FD شماره 1 به فایل proud.log اشاره می‌کند، بنابراین اطلاعات در این فایل نوشته شده است. اما، توصیف‌گرفایل دیگری هم (FD شماره 2) داریم که به همین فایل و بویژه به *ابتدای* این فایل اشاره می‌کند. وقتی grep سعی می‌کند فایل '*not a file*' را برای خواندن باز کند، نمی‌تواند. پس، یک پیغام خطا در stderr یعنی(FD شماره 2) می‌نویسد، که به ابتدای فایل proud.log اشاره می‌کرد. در نتیجه، دومین عمل نوشتن، اطلاعات اولی را رونویسی می‌کند!

لازم است از داشتن دو توصیف‌گرفایل مستقل از یکدیگر که روی یک منبع یا مقصد کار کنند، پیش‌گیری نماییم. می‌توانیم با *دونسخه‌ای نمودن* توصیف‌گرفایل، این کار را انجام دهیم:

```
$ grep proud file 'not a file' > proud.log 2>&1
```

برای درک این مطالب لازم است توجه داشته باشید که: تغییر مسیر فایل از چپ به راست خوانده می‌شود.این ترتیبی است که BASH آنها را پردازش می‌کند. اول، stdout طوری تغییر داده می‌شود که به فایل proud.log اشاره کند. بعد، ما از ترکیب >& برای دو نسخه‌ای نمودن FD شماره 1 و قرار دادن آن نسخه دوم در FD شماره 2 استفاده کرده‌ایم.

یک FD دونسخه‌ای با دو FD مستقل از یکدیگر که به یک محل اشاره می‌کنند، متفاوت عمل می‌نماید. عملیات نوشتن در هر دونسخه دقیقاً همانند است. آشفتگی مربوط به یک FD اشاره کننده به ابتدای فایل در حالیکه دیگری قبلاً آن را ادامه داده وجود نخواهد داشت.

مراقب باشید ترتیب آنها را اشتباه نکنید:

```
$ grep proud file 'not a file' 2>&1 > proud.log
```

این مثال stderr را نسبت به جایی که stdout اشاره می‌کند(که ترمینال است) دونسخه‌ای می‌کند، و بعد stdout به فایل proud.log تغییر مسیر داده خواهد شد. در نتیجه پیغام‌های stdout ثبت می‌شوند، اما پیغام خطاها بازهم به ترمینال می‌روند. *ای وای.*

نکته: برای راحتی، BASH همچنین شکل دیگری از تغییر مسیر را در دسترس شما قرار می‌دهد. عملگر تغییر مسیر &> در حقیقت شکل کوتاه شده‌ای از آنچه ما در اینجا دیدیم، است، تغییر مسیر stdout و stderr هر دو به یک فایل:

```
$ grep proud file 'not a file' &> proud.log
```

این نیز همان 2>&1 > proud.log می‌باشد، اما قابل حمل به پوسته BourneShell نیست. این تکنیک پیشنهاد شده‌ای نیست، اما اگر ببینید که شخص دیگری از آن در اسکریپت خود استفاده کرده، باید آن را تشخیص بدهید.

در مستندات گنو: Duplicating File Descriptors, Moving File Descriptors, Opening File Descriptors for Reading and Writing

---

در پرسش و پاسخ‌های رایج:

چطور می‌توانم چند دستور را در یک مرحله تغییر مسیر بدهم؟

چطور می‌توانم خروجی 'time' را به یک متغیر یا فایل تغییر مسیر بدهم؟

چگونه می‌توانم از dialog برای دریافت ورودی کاربر استفاده کنم؟

چطور می‌توانم stderr را به یک لوله تغییر مسیر بدهم؟

به طور کلی در باره 2>&1 بگویید ـــ تفاوت بین foo> 1>&2 و foo> 2>&1 چیست، و چه‌وقت می‌توانم از هرکدام استفاده کنم؟

4.3. Heredocها و Herestringها

گاهی نگهداری داده‌ها در یک فایل زائد است. ممکن است فقط مقدار بسیارکمی داشته باشیم ـــ مناسب برای به راحتی گنجاندن آن در خود اسکریپت .یا ممکن است خواسته باشیم محتوای متغیری را ، بدون آنکه اول آن را در یک فایل بنویسیم، به یک دستور تغییر مسیر بدهیم.

```
$ grep proud <<END
```

```
> I am a proud sentence.
> END
I am a proud sentence.
```

این یک *Heredoc* (یا *سند اینجا*) می‌باشد. وقتی می‌خواهید یک قطعه کوچک چند سطری داده‌ها را در اسکریپت خود تعبیه کنید *Heredoc*ها سودمند هستند. (تعبیه قطعات بزرگ تکنیک نامناسبی است. شما باید منطق خود(کد خودتان) و ورودی خود(داده‌هایتان) را جداگانه و ترجیحاً در فایلهای مختلف نگاه دارید، مگر اینکه داده‌ها جزئی باشند.)

در یک *Heredoc*، کلمه‌ای برای ایفای نقش نگهبان انتخاب می‌کنیم. هر کلمه‌ای می‌تواند باشد، ما از END در این مثال استفاده کرده‌ایم. کلمه‌ای انتخاب کنید، که درمجموعه داده‌ای شما ظاهر نمی‌شود. تمام سطرهایی که پس از *اولین* نمونه نگهبان تا رسیدن به *دومین* نمونه آمده‌اند، برای دستور stdin می‌شوند. دومین نمونه نگهبان، باید خودش یک سطر کامل باشد.

چند مورد متمایز برای *Heredoc*ها وجود دارد. به طور معمول، نمی‌توانید آنها را دندانه‌دار کنید، همه فاصله‌هایی که در اینجا برای دندانه‌دار کردن اسکریپت به کار ببرید در stdin ظاهر خواهند شد. رشته پایان دهنده(در مثال ما END) باید در ابتدای سطر باشد.

```
echo "Let's test abc:"
if [[ abc = a* ]]; then
    cat <<END
        abc seems to start with an a!
END
fi
```

چنین نتیجه خواهد داد:

```
Let's test abc:
      abc seems to start with an a!
```

می‌توانید با حذف موقتی توگذاری برای سطرهای *Heredoc*های خود، از این مطلب اجتناب کنید. به هر حال، موجب بدشکل شدن تورفتگی شکیل و زیبای شما می‌گردد. یک جایگزین وجود دارد. اگر از *END*-<< به جای *END*<< به عنوان عملگر *Heredoc* خود استفاده کنید، BASH همه کاراکترهای tab در ابتدای هر سطر از محتوای *Heredoc* شما را قبل از ارسال به دستور حذف می‌کند. به این ترتیب بازهم می‌توانید از tabها (اما از فاصله خیر) برای توگذاری محتوای *Heredoc* خود با بقیه کد استفاده کنید. آن tabها برای فرمانی که *Heredoc* شما را دریافت می‌کند، ارسال نخواهند شد. همچنین می‌توانید tabها را برای توگذاری رشته نگهبان هم به کار ببرید.

به طور پیش‌فرض، جایگزینی‌های BASH در محتوای *Heredoc* انجام می‌شود. هر چند، اگر کلمه‌ای که برای جداکردن *Heredoc* خود به کار می‌برید را نقل‌قولی نمایید، BASH هیچگونه جایگزینی در محتویات انجام نخواهد داد. برای دیدن تفاوت، این مثال را با و بدون کاراکترهای نقل‌قول امتحان کنید:

```
$ cat <<'XYZ'
> My home directory is $HOME
```

```
> XYZ
My home directory is $HOME
```

رایج‌ترین کاربرد *Heredoc*ها، در ارائه مستندات به کاربر است:

```
usage() {
cat <<EOF
usage: foobar [-x] [-v] [-z] [file ...]
A short explanation of the operation goes here.
It might be a few lines long, but shouldn't be excessive.
EOF
}
```

حالا اجازه بدهید *Herestring* خیلی مشابه اما فشرده‌تر را آزمایش کنیم:

```
$ grep proud <<<"I am a proud sentence"
I am a proud sentence.
```

این‌دفعه، stdin مستقیماً اطلاعاتش را از رشته‌ای که پس از عملگر <<< قرار داده‌اید، می‌خواند. این برای ارسال اطلاعات داخل متغیرها، به پردازش‌ها خیلی مناسب است:

```
$ grep proud <<<"$USER sits proudly on his throne in $HOSTNAME."
lhunath sits proudly on his throne in Lyndir.
```

*Herestring*ها کوتاه‌تر، کمتر مزاحم، و به طور کلی مناسب‌تر از *Heredoc*های حجیم همتای خود می‌باشند. گرچه آنها قابل حمل به پوسته Bourne نیستند.

بعداً، شما در باره لوله‌ها و اینکه چطور می‌توانند برای ارسال خروجی یک دستور به stdin دستور دیگر به کار بروند، خواهید آموخت. افراد بسیاری برای ارسال خروجی یک متغیر به عنوان stdin برای یک فرمان، از لوله‌ها استفاده می‌کنند. گرچه، برای این مقصود *Herestring*ها باید ترجیح داده شوند. آنها پوسته فرعی ایجاد نمی‌کنند و برای پوسته و هم برای سبک نگارش اسکریپت پوسته شما روشن‌تر هستند:

```
$ echo 'Wrap this silly sentence.' | fmt -t -w 20
Wrap this silly
   sentence.
$ fmt -t -w 20 <<< 'Wrap this silly sentence.'
Wrap this silly
```

```
sentence.
```

تکرار مفید:

*heredoc*های بلند به طور معمول ایده نامناسبی هستند، زیرا اسکریپت‌ها باید محتوی منطق باشند، نه داده‌ها. اگر اسکریپت به سند حجیمی نیاز دارد، باید آن را در یک فایل جداگانه با اسکریپت همراه کنید. با این وجود *Herestring*ها، اغلب کاملاً سودمند هستند بویژه برای ارسال محتوای یک متغیر(به جای فایلها) به فیلترهایی مانند `grep` یا `sed`.

در مستندات گنو: Here Documents, Here Strings

5. لوله‌ها(Pipes)

حال که می‌توانید بدون دردسر *توصیف‌گرهای فایل* را برای هدایت انواع معینی از خروجی‌ها به فایلهای معین، مدیریت نمایید، وقت آنست که رموز مبتکرانه‌تری که برای تغییرمسیر ورودی و خروجی در دسترس می‌باشد را بیاموزید.

می‌توانید از *تغییرمسیرفایل* برای نوشتن خروجی در فایلها یا خواندن ورودی از فایلها، استفاده کنید. اما اگر بخواهید خروجی یک برنامه را به طور مستقیم به ورودی برنامه دیگر مربوط کنید چطور؟ از آن طریق می‌توانید زنجیر پیوسته‌ای از پردازش خروجی ایجاد کنید. اگر از قبل درباره FIFOها، آگاه بودید، می‌توانستید با استفاده از چیزی مانند این مثال آن را کامل کنید:

```
$ ls
$ mkfifo myfifo; ls
myfifo
$ grep bea myfifo &
[1] 32635
$ echo "rat
> cow
> deer
> bear
> snake" > myfifo
bear
```

از دستور `mkfifo` برای ایجاد یک فایل جدید به نام 'myfifo' در دایرکتوری جاری استفاده کردیم. این یک فایل معمولی نیست، بلکه یک FIFO است(که از عبارت *First In, First Out* اخذ گردیده). فایلهای FIFO فایلهای ویژه‌ای هستند که جهت داده‌های مبتنی بر *First In, First Out* به کار می‌روند. وقتی از یک FIFO می‌خوانید، فقط داده‌ها را به محض اینکه پردازش دیگری در آن فایل می‌نویسد، دریافت می‌کنید. همینطور یک FIFO هرگز به طورواقعی محتوی هیچ داده‌ای نیست. تا وقتی که پردازشی در آن نمی‌نویسد، هر گونه عملیات خواندن از FIFO در انتظار رسیدن داده‌های معتبر متوقف می‌گردد. همین کار برای نوشتن در FIFO انجام می‌شود ــ نوشتن نیز تا وقتی پردازش دیگری در حال خواندن از FIFO است، متوقف می‌شود.

در مثال ما، FIFO با نام myfifo توسط grep خوانده می‌شود. پردازش grep منتظر می‌ماند تا اطلاعات در FIFO قابل دسترس بشوند . اینکه چرا ما عملگر & را به دستور grep پیوست می‌کنیم، برای آنست که آن را در پس‌زمینه قرار دهیم. به این ترتیب، ما می‌توانیم در حالیکه grep در انتظار داده می‌ماند، به تایپ و اجرای دستورات ادامه بدهیم. دستور echo اطلاعات را به FIFO تغذیه می‌کند. به مجرد اینکه این داده قابل دسترس می‌شود، دستور در حال اجرای grep آن را می‌خواند و پردازش می‌کند. نتیجه نمایش داده می‌شود. ما به طور موفقیت‌آمیزی داده را از دستور echo به فرمان grep فرستاده‌ایم.

اما این فایلهای موقتی یک اذیت واقعی هستند. شاید شما مجوز نوشتن نداشته‌باشید. لازم است به خاطر داشته باشید که فایلهای موقتی که ایجاد نموده‌اید را پاک کنید. لازم است مطمئن شوید که داده‌ها وارد و خارج می‌شوند، یا شاید FIFO به دلیل نامعلومی مسدودسازی را خاتمه دهد.

به خاطر این مسائل، ویژگی دیگری در دسترس قرارگرفته است: ***لوله‌ها***. در اصل، لوله فقط stdout یک فرایند را به stdin فرایند دیگر ارتباط می‌دهد، یعنی ***لوله‌کشی*** مؤثر داده‌ها از یک پردازش به دیگری. مجموعه کامل دستوراتی را که به این صورت با هم متصل شده‌اند، ***خط لوله*** می‌نامند. بیایید مثال فوق را دوباره امتحان کنیم، اما با استفاده از لوله‌ها:

```
$ echo "rat
> cow
> deer
> bear
> snake" | grep bea
bear
```

لوله با استفاده از عملگر | بین دو دستور که با هم متصل می‌گردند، ایجاد می‌شود. stdout دستور قبلی به stdin دستور بعدی متصل می‌گردد. در نتیجه، grep می‌تواند خروجی echo را بخواند و نتیجه عملیات خود را که bear است نمایش بدهد.

لوله‌ها به طور وسیعی به عنوان پس‌پردازش خروجی برنامه‌ها به کار می‌روند. به FIFOها، نیز در حقیقت به عنوان ***لوله‌های دارای‌نام*** رجوع می‌شود. آنها همان نتایج عملگر pipe را حاصل می‌کنند، لیکن از طریق یک نام فایل.

توجه:
عملگر لوله برای هر دستور یک محیط پوسته فرعی ایجاد می‌کند. آگاهی از این مطلب اهمیت دارد، به علت آنکه، هر متغیری که در دستور دوم تعیین یا ویرایش کنید، در خارج از آن به صورت ویرایش نشده قبلی ظاهر می‌گردد. اجازه بدهید تشریح نمایم:

```
$ message=Test
$ echo 'Salut, le monde!' | read message
$ echo "The message is: $message"
The message is: Test
$ echo 'Salut, le monde!' | { read message; echo "The message is:
$message"; }
The message is: Salut, le monde!
$ echo "The message is: $message"
```

```
The message is: Test
```

وقتی خط لوله به پایان می‌رسد، پوسته‌های فرعی که برای آن ایجاد شده‌اند نیز خاتمه می‌یابند. همراه با پوسته‌های فرعی، هر تغییر و تبدیل انجام شده در آنها نیز از بین می‌رود. بنابراین مراقب باشید!

تکرار مفید:
لوله‌ها به عنوان پس‌پردازشگر خروجی برنامه‌ها، خیلی جالب هستند. به هرحال، شما بایدمراقب باشید در کاربرد آنها زیاده‌روی نکنید. اگر شما خط‌لوله‌ای که شامل سه برنامه یا بیشتر باشد را به انتها برسانید، وقت آنست که از خود بپرسید آیا در حال انجام امور به روش هوشمندانه‌ای هستید؟ امکان دارد قادر باشید ویژگی‌های بیشتری از برنامه را نسبت به آنچه در یک پس‌پردازش در لوله به کار گرفته‌اید، استفاده کنید. هر دستور جدید در یک خط‌لوله موجب یک پوسته فرعی جدید می‌گردد و یک برنامه جدید باید بارگزاری شود. همچنین این مطلب دنبال کردن منطق اسکریپت شما را دشوار می‌سازد!

در مستندات گنو: Pipelines

در پرسش و پاسخ‌های رایج:

من متغیرهایی را در یک حلقه مقرر می‌کنم. چرا آنها پس از اتمام حلقه، ناگهان ناپدید می‌گردند؟ یا، چرا نمی‌توانم داده‌ها را برای خواندن لوله‌کشی نمایم؟

چطور می‌توان دو پردازش را با استفاده از لوله‌های بانام (fifoها) به یکدیگر ارتباط داد؟

چطور می‌توانم stderr را به یک لوله تغییر مسیر بدهم؟

چطور می‌توانم یک فایل را سطر به سطر بخوانم؟

به طور کلی در باره 2>&1 بگویید ـــ تفاوت بین >foo 2>&1 و 2>&1 >foo چیست، و چه‌وقت می‌توانم از هرکدام استفاده کنم؟

---

6. عملگرهای گوناگون

در کنار عملگرهای ورودی و خروجی استاندارد، bash همچنین چند عملگر پیشرفته‌تر نیز که کار با پوسته را دلپسندتر می‌نمایند، فراهم نموده است.

6.1. جایگزینی پردازش

پسرعموی لوله، عملگر جایگزینی پردازش است، که به دو شکل ظاهر می‌گردد: <() و >(). این روش مناسبی برای به کار بردن لوله‌های با نام، بدون لزوم ایجاد فایلهای موقتی می‌باشد. هنگامی که تصور می‌کنید برای انجام مواردی نیاز به فایل موقتی دارید، ممکن است جایگزینی پردازش روش بهتری برای انجام آنها باشد.

کاری که انجام می‌دهد، اساساً اجرای دستور داخل پرانتزها می‌باشد. با عملگر <() ، خروجی دستور در یک لوله با نام(یا چیزی مشابه آن)

که توسط bash ایجاد شده است، قرار داده می‌شود. خود عملگر در دستور شما با نام آن فایل تعویض می‌گردد. پس از اینکه دستور به پایان رسید، فایل پاک می‌شود.

در اینجا چگونگی انجام آن را در عمل می‌بینیم: موقعیتی را در نظر بگیرید که می‌خواهید تفاوت میان خروجی دو دستور را ببینید. به طور معمول، باید دو خروجی را در دو فایل قرار بدهید و با برنامه `diff` آنها را مقایسه کنید:

```
$ head -n 1 .dictionary > file1
$ tail -n 1 .dictionary > file2
$ diff -y file1 file2
Aachen                                                  | zymurgy
$ rm file1 file2
```

می‌توانیم با به کاربردن عملگر *جایگزینی پردازش*، تمام آن را در یک سطر انجام بدهیم، و نیازی به پاکسازی دستی هم نیست:

```
$ diff -y <(head -n 1 .dictionary) <(tail -n 1 .dictionary)
Aachen                                                  | zymurgy
```

قسمت `<(..)` با فایل FIFO موقتی ایجاد شده توسط bash، تعویض می‌گردد، بنابراین `diff` در حقیقت، چیزی به این شکل را می‌بیند:

```
$ diff -y /dev/fd/63 /dev/fd/62
```

در اینجا می‌بینیم که وقتی ما از جایگزینی پردازش استفاده می‌کنیم، bash چگونه `diff` را اجرا می‌کند. دستورهای `head` و `tail` را اجرا می‌کند، خروجی‌های آنها را به ترتیب به فایلهای */dev/fd/63* و */dev/fd/62* تغییرمسیر می‌دهد. سپس فرمان `diff` را با قبول کردن آن نام فایلها در جایی که ما عملگرهای جایگزینی پردازش را قرار داده بودیم اجرا می‌کند.

پیاده‌سازی واقعی فایلهای موقتی ازیک سیستم تا سیستم دیگر فرق می‌کند. در حقیقت، با قرار دادن یک دستور `echo` در ابتدای دستور یک سطری خود، می‌توانید به طور واقعی مشاهده کنید آنچه در بالا گفته شد، به نظر دستور `diff` خواهد آمد:

```
$ echo diff -y <(head -n 1 .dictionary) <(tail -n 1 .dictionary)
diff -y /dev/fd/63 /dev/fd/62
```

عملگر `>(..)` بسیار همانند عملگر `<(..)` می‌باشد، اما به جای تغییرمسیر خروجی فرمان به یک فایل، یک فایل را به ورودی فرمان تغییر مسیر می‌دهد. این برای مواردی به کار می‌رود که شما دستوری را اجرا می‌نمایید که در یک فایل می‌نویسد، اما شما می‌خواهید به جای آن در دستور دیگر بنویسد:

```
$ tar -cf >(ssh host tar xf -) .
```

تکرار مفید:

جایگزینی پردازش روش فشرده‌ای برای ایجاد خودکار فایلهای موقتی FIFO در اختیار قرار می‌دهد. آنها نسبت به زمانی که شما خودتان به طور دستی pipeهای با نام را ایجاد می‌کنید، کمتر انعطاف‌پذیر هستند، اما برای دستورات کوتاه متداول مانند `diff` که نیازمند نام‌فایلها برای منابع ورودی‌اشان می‌باشند، بدون نقص هستند.

---

1. **مترجم:** bit bucket در اصل برای مخزنی موهوم جهت گرفتن بیت‌هایی که با دستورالعمل shift اسمبلی به انتهای یک ثَبّات(register) عقب رانده می‌شوند به کار رفته است (1)

2. **مترجم:** منظور از هر بار اجرای حلقه در این قسمت آن است که اگر به فرض این حلقه در یک اسکریپت باشد، با هر بار اجرای آن اسکریپت، فایل خالی و بازنویسی می شود. نه تکرار حلقه که به ازای هر مقدار homdir انجام می‌شود. چون همانطور که قبلاًتوضیح داده شده برای جلوگیری از خالی شدن فایل به ازای مقادیر homdir، در هر بار تکرار حلقه، عملگر تغییر مسیر بعد از کلمه‌کلیدی done قرار داده شده است. (2)

دستورات مرکب

فهرست مطالب

BASH روشهای بسیاری برای ترکیب فرمانها جهت رسیدن به هدفهایمان ارائه می‌کند. قبلاً برخی از آنها را در عمل دیده‌ایم، اما اکنون بیایید کمی بیشتر به آنها بپردازیم.

BASH ساختارهایی دارد که *دستورات مرکب* نامیده می‌شوند، عبارتی چندپهلو، که با مفاهیم متفاوتی مرتبط می‌باشد. ما قبلاً بعضی از دستورات مرکب ارائه شده در BASH را دیده‌ایم ــ جمله‌های if و حلقه‌های for و حلقه‌های while و کلمه‌کلیدیهای [[و case و select. دوباره آن مطالب را در اینجا تکرار نخواهیم نمود. به جای آن، دستورات مرکب دیگری را که هنوز ندیده‌ایم، بررسی خواهیم کرد: پوسته‌های فرعی، گروه‌بندی دستورات، و ارزیابی محاسباتی.

علاوه براین، به *توابع* و *مستعارها* که دستورات مرکب نیستند، اما به روش مشابهی عمل می‌کنند، نیز خواهیم پرداخت.

در مستندات گنو: Compound Commands

پوسته های فرعی

یک پوسته فرعی مانند یک پردازش فرزند است، غیر از اینکه اطلاعات بیشتری ارث می‌برد. در یک خط لوله، پوسته‌های فرعی به طور ضمنی برای هر فرمان ایجاد می‌گردند. همچنین به طور صریح نیز با به کار بردن پرانتزها در اطراف دستور، ایجاد می‌شوند:

```
$ (cd /tmp || exit 1; date > timestamp)
$ pwd
/home/lhunath
```

موقعی که پوسته فرعی خاتمه می‌یابد، اثر دستور cd از بین رفته است ــ ما همانجایی هستیم که از آنجا شروع کرده بودیم. همچنین، هر متغیری که در پوسته فرعی تنظیم گردیده، قابل یادآوری نیست. می‌توانید پوسته فرعی را به صورت پوسته موقتی در نظر بگیرید. برای جزئیات بیشتر پوسته فرعی را ملاحظه کنید.

توجه داشته باشید که در این مثال، اگر فرمان cd ناموفق باشد، دستور exit 1 به پوسته فرعی خاتمه می‌دهد، اما به شل محاوره‌ای ما خیر. به طوری که می‌توانید حدس بزنید، این مورد در اسکریپت‌های حقیقی کاملاً سودمند است.

در مستندات گنو: Command Grouping

گروه بندی دستورات

قبلاً با این موضوع در گروه بندی جملات برخورد کرده‌ایم، با وجود آن در ضمن این فصل تکرار می‌شود.

با استفاده از ابروها، دستورات را می‌توان گروه‌بندی نمود. این تا حدودی مانند پوسته‌های فرعی به نظر می‌آید، اما اینطور نیست. دستورات گروهی مانند هر چیز دیگر در همان پوسته اجرا می‌شوند نه در یک شل جدید.

دستورات گروهی می‌توانند برای اجرای دستورات چندگانه به کار رفته و یک تغییر مسیر منفردِ مؤثر بر تمام آنها داشته باشند:

```
$ { echo "Starting at $(date)"; rsync -av . /backup; echo "Finishing at
$(date)"; } > backup.log 2>&1
```

یک زیر پوسته در این وضعیت زیاده‌روی خواهد بود، به دلیل آنکه نیازی به یک محیط موقتی نداریم. هر چند که، یک پوسته فرعی نیز کار خواهد کرد.

گروه‌های دستورات همچنین برای کوتاه کردن وظایف معین متداول، سودمند می‌باشند:

```
$ [[ -f $CONFIGFILE ]] || { echo "Config file $CONFIGFILE not found"
>&2; exit 1; }
```

این را با روایت رسمی آن مقایسه کنید:

```
$ if [[ ! -f $CONFIGFILE ]]; then
> echo "Config file $CONFIGFILE not found" >&2
> exit 1
> fi
```

پوسته فرعی در اینجا کار نخوهد کرد، زیرا دستور exit 1 موجود در گروه دستورات، به طور کلی پوسته راخاتمه می‌دهد، که آنچه ما در اینجا می‌خواهیم، نیست.

گروه دستورات، همچنین برای تنظیم متغیرها در حالت‌های غیر معمول، به کار می‌رود:

```
$ echo "cat
```

```
> mouse
> dog" > inputfile
$ { read a; read b; read c; } < inputfile
$ echo "$b"
mouse
```

خواندن دومین سطر از یک فایل، بدون یک گروه دستورات، که خواندن چندگانه دستورات read از یک توصیف‌گر فایل باز، بدون بازگشت هر دفعه به ابتدای فایل را مجاز می‌کند، بینهایت دشوار خواهد بود. با این مورد مقایسه کنید:

```
$ read a < inputfile
$ read b < inputfile
$ read c < inputfile
$ echo "$b"
cat
```

ابداً آنچه ما می‌خواستیم، نیست!

اگر آنگونه که ما در اینجا نشان دادیم، گروه دستورات در یک سطر باشد، سپس *باید* یک سمی‌کالن قبل از بستن گروه با } قرار داده شود، در غیر اینصورت، BASH گمان خواهد نمود } یک شناسه برای آخرین دستور گروه است. اگر گروه دستورات در چندین سطر گسترده شده باشند، آنوقت سمی‌کالن می‌تواند با کاراکتر سطر جدید جایگزین شود:

```
$ {
>  echo "Starting at $(date)"
>  rsync -av . /backup
>  echo "Finishing at $(date)"
> } > backup.log 2>&1
```

در مستندات گنو: Command Grouping

ارزیابی محاسباتی

BASH دارای چند روش مختلف است، برای آنکه به او بگوییم، به جای عملیات روی رشته‌ها، می‌خواهیم ارزیابی محاسباتی انجام بدهیم. اجازه دهید یکی یکی به آنها بپردازیم.

اولین روش دستور let می‌باشد:

```
$ unset a; a=4+5
```

```
$ echo $a
4+5
$ let a=4+5
$ echo $a
9
```

اگر عبارت را نقل‌قولی کنید، می‌توانید از فاصله‌ها، پرانتزها و غیره استفاده کنید:

```
$ let a='(5+2)*3'
```

برای لیست کامل عملگرهای معتبر، دستور `help let` یا مستندات را ببینید.

بعد دستور مرکب *ارزیابی محاسباتی* حقیقی است:

```
$ ((a=(5+2)*3))
```

این معادل دستور `let` می‌باشد، اما می‌توانیم آن را به عنوان یک *فرمان* نیز به کار ببریم، به عنوان مثال در یک جمله `if` به این صورت:

```
$ if (($a == 21)); then echo 'Blackjack!'; fi
```

عملگرهایی مانند `==` و `>` , `<` و غیره موجب انجام یک مقایسه در درون ارزیابی محاسباتی می‌شوند. اگر مقایسه صحیح باشد(به طور نمونه $2 < 10$ در محاسبه صحیح است ــ اما در رشته‌ها خیر!) ، بعد دستور مرکب با کد وضعیت 0 خارج می‌شود. اگر مقایسه غلط باشد با کد وضعیت 1 خارج می‌گردد. این امر آن را برای انجام بررسی‌ها در یک اسکریپت سودمند می‌سازد.

هرچند اگر یک دستور مرکب نباشد، یک ترکیب دستوری معتبر *جایگزینی حسابی* (یا یک *عبارت حسابی*) نیز می‌باشد:

```
$ echo "There are $(($rows * $columns)) cells"
```

بخش درونی `$((...))` یک *مفهوم محاسباتی* است، درست مانند `((...))`، این به معنای آنست که به جای دستکاری رشته‌ها(الحاق *$row*، فاصله، ستاره، فاصله، و *$columns*)، می‌خواهیم محاسبه انجام بدهیم(عملیات چندگانه). ترکیب `$((...))` همچنین قابل حمل به شل POSIX می‌باشد، در حالیکه `((...))` نیست.

خوانندگانی که با زبان برنامه‌نویسی C آشنا هستند، ممکن است مایل باشند بدانند که `((...))` ویژگی‌های C ـ شکل بسیاری دارد. از میان آنها یکی عملگر سه‌گانه است:

```
$ ((abs = (a >= 0) ? a : -a))
```

و یکی هم استفاده از مقدار صحیح به عنوان مقدار درست:

```
$ if ((flag)); then echo "uh oh, our flag is up"; fi
```

توجه نمایید که ما متغیرها را در داخل ((...)) بدون پیشوند کردن علامت $ به آنها، استفاده کرده‌ایم. این یک ترکیب خاص کوتاه شده است که BASH در درون ارزیابی محاسباتی و عبارتهای محاسباتی اجازه می‌دهد. همچنین می‌توانیم آنرا درون یک $((...)) در BASH به کار ببریم، اما در شل POSIX خیر.

یک مطلب نهایی وجود دارد ، که باید در باره ((flag)) متذکر شویم. چون درون عبارت ((...)) قالب C-مانند دارد، یک متغیر(یا عبارت) که *صفر* ارزیابی گردد، به واسطه مفاهیم ارزیابی محاسباتی به عنوان *false* در نظر گرفته می‌شود. سپس، چون ارزیابی، غلط است، با کد وضعیت 1 *خارج* می‌شود. علاوه بر این، اگر عبارت درون ((...)) *غیر صفر* باشد، به عنوان *صحیح* در نظر گرفته می‌شود، و نظر به اینکه، ارزیابی صحیح است با کد وضعیت *صفر خارج* خواهد شد. این امر به طور بالقوه *بسیار* گیج‌کننده است، حتی برای اهل فن، به طوری که وقت زیادی برای پرداختن به آن از شما خواهد گرفت. با وجود این، موقعی که مسائل به طریقی که برای آنها در نظر گرفته شده‌اند به کار بروند، در نهایت قابل درک خواهند بود:

```
$ flag=0      # no error
$ while read line; do
>   if [[ $line = *err* ]]; then flag=1; fi
> done < inputfile
$ if ((flag)); then echo "oh no"; fi
```

در مستندات گنو: Arithmetic Expansion, Shell Arithmetic

توابع

توابع در اسکریپت‌های bash خیلی جذاب هستند. بلوک‌هایی از فرمانها می‌باشند، خیلی مشابه اسکریپت‌های عادی، که شاید شما بنویسید، به جز آنکه به صورت فایلهای جداگانه نیستند. هرچند که، آنها درست مانند اسکریپت‌ها شناسه‌ها را می‌پذیرند -- و بر خلاف اسکریپت‌ها، اگر شما بخواهید، می‌توانند بر متغیرهای داخل اسکریپت‌های شما تأثیرگذار باشند. برای مثال این مورد را ملاحظه کنید:

```
$ sum() {
>   echo "$1 + $2 = $(($1 + $2))"
> }
```

این نمونه وقتی اجرا شود، مطلقاً کاری انجام نمی‌دهد. این به آن دلیل است که، فقط درحافظه ذخیره می‌شود، خیلی همانند یک متغیر، اما تا موقعی که هنوز فراخوانی نشده است. برای اجرای تابع، به این طریق باید عمل کنید:

```
$ sum 1 4
1 + 4 = 5
```

شگفتا! اکنون یک ماشین حساب ابتدایی داریم، و جایگزین بالقوه مقرون به صرفه‌تری برای یک کودک پنج‌ساله.

اگر بخواهید توابع را در داخل اسکریپت‌ها تعبیه کنید، که بسیار مناسب‌تر خواهد بود، آنوقت لازم است متوجه باشید، پارامترهایی که با اسکریپت به کار می‌برید، لزوماً نبایستی همان پارامترهای داده شده به تابع باشند. برای پوشاندن این تابع در داخل یک اسکریپت، باید چنین فایلی بنویسیم:

نمایش یا عدم نمایش شماره سطرها

```
1 #!/bin/bash
2 sum() {
3         echo "$1 + $2 = $(($1 + $2))"
4 }
5 sum "$1" "$2"
```

به طوری که می‌توانید ملاحظه کنید، ما دو پارامتر اسکریپت را به تابع داخل آن داده‌ایم، اما می‌توانستیم هر پارامتر دیگری که مایل باشیم به تابع بدهیم، (اگرچه ، این حالت فقط موجب سردرگمی کاربران استفاده کننده از آن خواهد شد).

توابع در اسکریپت چند هدف را برآورده می‌سازند. اول آنکه یک بلوک از کدها که وظیفه معینی انجام می‌دهند را مجزا می‌کنند، به طوری که مانع درهم ریختگی سایر کدها می‌گردد. این مطلب تا وقتی اعتدال را رعایت می‌کنید، به شما کمک می‌کند، کدهای خواناتر بنویسید. (پرش در طول اسکریپت برای پیگردی 7تابع جهت کشف آنکه یک دستور منفرد چه کار می‌کند، تأثیر منفی خواهد داشت، بنابراین درنظر داشته باشید کاری که انجام می‌دهید، قابل فهم باشد.) دوم، میسر نمودن استفاده مجدد از کد با تغییر جزئی شناسه‌ها است.

این یک مثال کمتر فاقد کیفیت:

نمایش یا عدم نمایش شماره سطرها

```
1 #!/bin/bash
2 open() {
3     case "$1" in
4         *.mp3|*.ogg|*.wav|*.flac|*.wma) xmms "$1";;
5         *.jpg|*.gif|*.png|*.bmp)        display "$1";;
6         *.avi|*.mpg|*.mp4|*.wmv)        mplayer "$1";;
7     esac
8 }
9 for file; do
10     open "$file"
11 done
```

در اینجا *تابعی* به نام open تعریف نموده‌ایم. این تابع قطعه‌ای کُد است که یک شناسه منفرد را دریافت می‌کند، و بر اساس *الگوی* آن شناسه، برنامه xmms یا display یا mplayer را با آن شناسه اجرا خواهد نمود. سپس، یک حلقه for با تمام پارامترهای مکانی

اسکریپت تکرار می‌شود. (به خاطر داشته باشید که for file معادل for file in "$@" می‌باشد و هر یک از آنها تکرار روی مجموعه کامل پارامترهای موقعیتی را انجام می‌دهند.) حلقه for برای هر یک از پارامترها، تابع **open** را فراخوانی می‌کند.

به طوری که شاید ملاحظه نموده باشید، پارامترهای تابع غیر از پارامترهای اسکریپت هستند.

همچنین، تابع ممکن است *متغیرهای محلی* داشته باشد، که با دستور داخلی `local` یا `declare` تعریف شده باشند. این امر شما را قادر می‌سازد بدون احتمال رونویسی متغیرهای مهم، توابع را فراخوانی کنید. برای نمونه:

```
count() {
    local i
    for ((i=1; i<=$1; i++)); do echo $i; done
    echo 'Ah, ah, ah!'
}
for ((i=1; i<=3; i++)); do count $i; done
```

متغیر iمحلی به طور متمایز از متغیر i خارج از تابع اسکریپت، ذخیره می‌گردد. این باعث می‌شود هر یک از دو حلقه،بدون تداخل با شمارش‌گر دیگری عمل کند.

توابع می‌توانند خودشان را نیز به طور *بازگشتی* فراخوانی نمایند، اما در اینجا به آن نمی‌پردازیم. *شاید بعداً!*

در مستندات گنو: Shell Functions

مستعارها

مستعارها در نگاه اول ظاهراً مشابه توابع هستند، اما در بررسی عمیق‌تر، آنها رفتار کاملا متفاوتی دارند.

- مستعارها به هیچ وجه در اسکریپت‌ها عمل نمی‌کنند. آنها فقط در پوسته محاوره‌ای کار میکنند.
- مستعارها نمی‌توانند شناسه قبول کنند.
- مستعارها خودشان را به طور بازگشتی احضار نمی‌کنند.
- مستعارها نمی‌توانند متغیرهای محلی داشته باشند.

اساساً مستعارها، میانبرهایی برای استفاده درفایلهای .bashrc جهت آسانتر نمودن روند کارهای شما می‌باشند. آنها به طور معمول به این شکل ظاهر می‌شوند:

```
$ alias ls='ls --color=auto'
```

BASH اولین کلمه هر دستور ساده را بررسی می‌کند، که ببیند آیا یک *مستعار* است، و اگر چنین باشد، یک جایگزینی ساده متن را انجام می‌دهد. بنابراین، اگر شما تایپ کنید

```
$ ls /tmp
```

BASH چنان عمل می‌کند، که گویی تایپ نموده‌اید

```
$ ls --color=auto /tmp
```

اگر خواسته باشید این توانایی را در یک تابع ایجاد کنید، به این شکل خواهد شد:

```
$ unalias ls
$ ls() { command ls --color=auto "$@"; }
```

همانند یک *گروه دستور*، اگر بخواهیم تمام آن را در یک سطر بنویسیم، لازم است یک سمی‌کالن (;) قبل از بستن تابع با } به کار ببریم. دستور داخلی ویژه command به تابع ما می‌گوید خودش را به طور بازگشتی فراخوانی نکند، به جای آن می‌خواهیم آن فرمان ls را فراخوانی کند، که در صورت عدم وجود تابع همنام خودش آن را احضار می‌کرد.

مستعارها تا زمانی که شما از آنها نخواهید همچون توابع کار کنند، مناسب هستند. اگر رفتار پیچیده‌ای مورد انتظار شماست، به جای آن از تابع استفاده کنید.

انهدام ساختارها

برای از بین بردن یک تابع یا متغیراز محیط جاری پوسته خود، دستور unset را به کار ببرید.

```
$ unset myfunction
```

برای از بین بردن مستعار، فرمان unalias را به کار ببرید.

```
$ unalias rm
```

در مستندات گنو: Bourne Shell Builtins

در پرسش و پاسخ‌های رایج

منبع یابی

وقتی اسکریپتی را از داخل اسکریپت دیگری احضار می‌کنید، اسکریپت جدید محیط اسکریپت اصلی را به ارث می‌برد، درست مانند هر برنامه دیگر اجرا شده در UNIX. تشریح آنکه این به چه معناست، خارج از حوزه این راهنما می‌باشد، اما بیش از همه، محیط را می‌توانید دایرکتوری کاری فعلی، توصیف‌گرفایل‌های باز، ، و متغیرهای محیط که می‌توانید آنها را با دستور `export` ببینید، در نظر بگیرید.

وقتی اسکریپتی که شما اجرا کرده‌اید(یا هر برنامه دیگری برای آن منظور) اجرایش به پایان می‌رسد محیطش دور انداخته می‌شود. محیط اسکریپت اول همان خواهد بود که قبل از فراخوانی اسکریپت دوم بود، اگرچه برخی پارامترهای ویژه bash(از قبیل مقدار متغیر `$?`، مقداربرگشتی آخرین فرمان اجرا شده) شاید تغییر کرده باشند. این به معنای آنست که به عنوان مثال، شما در واقع نمی‌توانید اسکریپتی را برای تغییر دایرکتوری جاری خود، اجرا نمایید.

در پرسش و پاسخ‌های رایج:
من سعی می‌کنم اسکریپتی بنویسم که دایرکتوری جاری را تغییر دهد(یا یک متغیر را تنظیم کند)، اما بعد از به پایان رسیدن اسکریپت، در همان جایی هستم که از آنجا شروع کرده بودم(یا متغیر من نیست)!

کاری که می‌توانید انجام دهید منبع نمودن اسکریپت درعوض اجرای آن به عنوان پردازش فرزند می‌باشد. این کار را با استفاده از فرمان `.` (نقطه) انجام بدهید:

```
. ./myscript   # فرمانها را در این محیط از فایل myscript اجرا می‌کند
```

این کار غالباً *نقطه‌ای نمودن*(dotting in) اسکریپت نامیده می‌شود. این `.` به BASH می‌گوید دستورات *./myscript*. را بخواند و آنها را در محیط پوسته جاری اجرا کند. نظر به اینکه دستورات در پوسته جاری اجرا می‌شوند، نمی‌توانند متغیرهای پوسته جاری، دایرکتوری‌های کاری، توصیف‌گرفایلهای باز، توابع و غیره را تغییر بدهند.

توجه کنید که Bashنام دومی برای این دستور دارد، `source`، اما نظر به اینکه، این عیناً مانند دستور `.` کار می‌کند، احتمالاً آسانتر است آنرا فراموش نموده و از دستور `.` ، که در همه جا کار خواهد کرد، استفاده کنیم.

کنترل Job

اگرچه در اسکریپت‌ها معمول نیست، اما کنترلjob در پوسته‌های محاوره‌ای بسیار با اهمیت است. کنترل Job شما را قادر می‌سازد با کارهای در حال اجرای پس‌زمینه، ارتباط متقابل داشته باشید، و jobهای در حال اجرا را معوق کنید، وغیره.

اصول نظری

در سیستم‌های Posix، jobها به صورت «گروههای پردازش»، با یک پردازش سرگروه، پیاده‌سازی شده‌اند. هر tty (ترمینال) یک «گروه پردازش» منفرد در پیش‌زمینه دارد، که ارتباط محاوره‌ای با ترمینال را ممکن می‌سازد. تمام گروههای پردازشی کنترلی دیگرِ همان tty به صورت jobهای پس‌زمینه می‌باشند، و می‌توانند در حال اجرا یا تعلیق باشند.

یک job موقعی تعلیق می‌شود که پردازش سر گروه آن یکی از سیگنال‌های SIGSTOP یا SIGTSTP یا SIGTTIN یا SIGTTOU را دریافت کند. سیگنالهای SIGTTIN و SIGTTOU هر وقت که یک job پس‌زمینه سعی کند از ترمینال بخواند یا در آن بنویسد، به طور خودکار ارسال می‌شوند___ این است چرای آنکه `cat &` به جای آنکه در پس‌زمینه اجرا گردد، فوراً به حالت تعلیق می‌رود.

فشردن برخی کلیدها در ترمینال باعث ارسال سیگنالها به تمام پردازشها در گروه پردازشی پیش‌زمینه می‌گردند. این کلیدها می‌توانند با فرمان `stty` پیکربندی شوند، اما معمولاً به طور پیش‌فرض تنظیم گردیده‌اند:

- Ctrl-Z سیگنال SIGTSTP را به job پیش‌زمینه ارسال می‌کند(به طور معمول آنرا به تعلیق در می‌آورد)
- Ctrl-C سیگنال SIGINT را به job پیش‌زمینه ارسال می‌کند(به طور معمول آن را خاتمه می‌دهد)
- Ctrl-\ سیگنال SIGQUIT را به job پیش‌زمینه ارسال می‌کند(به طور معمول موجب ایجاد یک نسخه فایل core و سپس انصراف از job می‌گردد)

شیوه عمل

کنترل Job به طور پیش‌فرض در پوسته‌های محاوره‌ای عمل می‌کنند. امکان آن هست که برای اسکریپت‌ها با دستور `set -m` یا `set -o monitor` فعال گردد.

پردازش job پیش‌زمینه با فشردن Ctrl-Z می تواند به حالت تعلیق درآید. در bash هیچ راهی برای ارجاع به job پیش‌زمینه وجود ندارد: اگر job پیش‌زمینه‌ای غیر از bash وجود داشته باشد، bash در انتظار می‌ماند تا آن job خاتمه یابد، و ازاینرو نمی‌تواند هیچ کدی را اجرا نماید(حتی trapها تا خاتمه یافتن jobپیش‌زمینه به تعویق می‌افتند). بنابراین، دستورات زیر فقط برای jobهای پس‌زمینه(و تعلیق شده) کار می‌کنند.

کنترل Job فرمانهای زیر را فعال می‌کند:

- `fg` [*job مشخصه*]: یک job پس‌زمینه را به پیش‌زمینه می‌آورد.
- `bg` [*job مشخصه* ...] : یک job به تعلیق درآمده را در پس‌زمینه اجرا می‌کند.

- `suspend`:پوسته را به حالت تعلیق می‌برد(اکثراً برای موقعی مناسب است که پردازش والد، یک پوستهِ با کنترل job باشد).

دستورات دیگر برای محاوره با jobها عبارتند از:

- `jobs` [*options*] [*jobspec* ...] : این jobهای تعلیقی و پس‌زمینه را لیست می‌کند. گزینه‌ها شامل `-p` (فقط شماره شناسایی پردازش را لیست می‌کند)، `-s` (فقط jobهای تعلیقی را لیست می‌کند)، و `-r` (فقط jobهای در حال اجرا در پس‌زمینه را لیست می‌کند). اگر یک یا چند مشخصه(jobspec) معین شده باشد، تمام jobهای دیگر صرفنظر می‌شوند.
- `kill` می‌تواند به جای شماره شناسایی، یک مشخصه job قبول کند.
- `disown` به *bash* می‌گوید یک job موجود را فراموش کند. این دستور bash را از ارسال خودکار سیگنال SIGHUP به پردازشهای آن job مانع می‌شود، اما همچنین به معنای آنست که دیگر نمی‌توان با jobspec به آن رجوع نمود.

بنابراین، تمام اینها به چه معناست؟ کنترل Job به شما اجازه می‌دهد در یک نشست منفرد ترمینال، امور چندگانه‌ای در حال اجرا داشته باشید. (در روزگار گذشته، زمانی که شما *فقط یک ترمینال* روی میز داشتید، و روشی برای ایجاد ترمینالهای مجازی جهت اضافه کردن آن نبود)، بسیار زیاد اهمیت داشت. در سیستمها و سخت افزار مدرن، شما انتخاب‌های بیشتری در دسترس دارید ــ برای مثال می‌توانید `screen` یا `tmux` را برای به دست آوردن ترمینالهای مجازی اجرا نمایید. یا داخل یک نشست X، می‌توانید `xterm` یا شبیه‌سازهای ترمینال بیشتری باز کنید(و می‌توانید این دو را به خوبی باهم ترکیب کنید).

اما گاهی اوقات، یک کنترل job ساده(تعلیق یا پس‌زمینه) سودمند واقع می‌شود. شاید یک پشتیبان‌گیری را شروع کرده باشید و بیش از آنچه انتظار دارید طول بکشد. می‌توانید با Ctrl-Z آن را به تعلیق درآورید و سپس آن را با دستور `bg` در پس‌زمینه قرار بدهید، واعلان فرمان پوسته را بازپس بگیرید،به طوری که در ضمن اینکه پشتیبان‌گیری در پس‌زمینه انجام می‌شود، بتوانید کار دیگری در همان نشست ترمینال انجام بدهید.

مشخصات Job

مشخصه job یا "jobspec" روشی برای ارجاع به پردازشی است که job را می‌سازد. یک مشخصه می‌تواند چنین باشد:

- `%`*n* جهت ارجاع به job شماره *n*.
- `%`*str* برای ارجاع به آن job که با دستوری که رشته *str* در ابتدای آنست، شروع شده باشد. اگر بیش از یک job اینگونه موجود باشد، یک خطا رخ می‌دهد.
- `%?`*str* برای ارجاع به یک job که با دستوری شامل رشته *str* شروع شده باشد. اگر بیش از یک job از این قبیل باشد، یک خطا رخ می‌دهد.
- `%%` یا `%+` برای ارجاع به job اخیر: آنکه آخرین job شروع شده در پس‌زمینه، یا تعلیق شده در پیش‌زمینه باشد. `fg` و `bg` در صورتی که jobspec داده نشده باشد، روی این job عمل می‌کنند.
- `%-` برای job قبلی(که قبل از job فعلی `%%` بود).

امکان آن هست که یک دستور اختیاری را بایک jobspec به کار برد، این ساختار: `jobs -x "cmd` *args...*`"`. در اینصورت argهایی که مشخصه job به نظر می‌آیند را با شماره شناسایی‌های سرگروه پردازشهای هر job متناظر تعویض می‌کند، سپس دستور را اجرا می‌کند. برای مثال، `jobs -x strace -p %%` فرمان `strace` را به job اخیر پیوست خواهد نمود(اگر به جای حالت تعلیق در حال اجرا

در پس‌زمینه باشد سودمندتر است).

مطلب انتهایی، یک مشخصه job عریان می‌تواند به عنوان دستور به کار رود: `%1` معادل است با `fg %1`, در حالیکه `%1 &` معادل با `bg %1`می‌باشد.

See Also

مدیریت پردازش تمرین‌هایی برای کار با پردازشهای چندتایی را بحث می‌کند. یک مثال از کاربرد کنترل job در داخل اسکریپت نیز دارد.

راه و روش‌ها

فهرست مطالب

1. انتخاب پوسته شما

اولین کاری که باید قبل از شروع به نوشتن یک اسکریپت پوسته یا هر نوع اسکریپت یا برنامه‌ای مشابه آن، انجام بدهید، برشمردن احتیاجات و اهداف آن اسکریپت است. سپس ارزیابی آنچه، بهترین ابزار برای انجام آن اهداف است.

BASH ممکن است برای یادگیری و نوشتن در آن آسان باشد، اما همیشه مناسب انجام کار نیست.

در مجموعه ابزارهای اساسی، تعداد بسیاری ابزار موجود است که می‌تواند به شما کمک کند. اگر شما فقط به AWK نیاز داشته باشید، نباید یک اسکریپت شل ایجاد کنید که آن را فراخوانی کند. فقط یک اسکریپت AWK ایجاد کنید. اگر به بازیابی داده از یک فایل HTMLیا XML به یک روش معتبر نیاز دارید، نیز Bash ابزار اشتباهی برای انجام آن کار است. باید به جای آن XPath/XSLT را به کار ببرید، یا یک زبانی که کتابخانه معتبری برای تجزیه XML یا HTML دارد.

اگر تصمیم گرفتید که اسکریپت پوسته آن ابزاری است که شما می‌خواهید، اول این سؤال‌ها را از خود بپرسید:

- در یک آینده قابل پیش‌بینی، آیا ممکن است اسکریپت شما در محیطی که Bash *به طور پیش‌فرض* در دسترس نیست، مورد احتیاج باشد؟
  - اگر اینطور است، پس به جای آن sh را در نظر بگیرید. `sh` یک شل POSIX است و ویژگی‌هایش در هر پوسته موافق با استاندارد POSIX، در دسترس می‌باشد. به این واقعیت تکیه کنید که هر سیستم POSIX قادر به اجرای اسکریپت شما خواهد بود. شما باید توازنی بین لزوم قابلیت حمل و عدم استفاده از قابلیت‌های ویژه Bash برقرار کنید.
  - *به خاطر داشته باشید که این راهنما شامل sh نمی‌شود!* صفحه bashism پیشنهادهایی دارد، اما کامل نیست.
- آیا اطمینان دارید که در تمام محیط‌هایی که اسکریپت را اجرا می‌کنید یا ممکن است در آینده بخواهید اجرا کنید، Bash 3.x (یا 4.x) در دسترس شما خواهد بود؟
  - اگر نه، باید خود را فقط به ویژگی‌های Bash 2.x محدود نمایید.

اگر سؤلات فوق انتخاب شما را محدود نمی‌کند، از تمام ویژگی‌های Bash که لازم دارید، استفاده کنید، توجه کنید که کدام نگارش Bash

برای اجرای اسکریپت شما لازم است.

استفاده از Bash نگارش 3 یا بالاتر به معنای آنست که می‌توانید از شیوه‌های اسکریپت‌نویسی کهنه و قدیمی که به *دلایل بسیار خوبی* با موارد خیلی بهتری جایگزین شده‌اند، اجتناب نمایید.

- از اسکریپت‌های نمونه‌ای که در Web می‌بینید، بدون درک کامل عملکرد آنها دوری کنید. اسکریپت‌هایی که در Web پیدا می‌کنید اکثراً به نوعی ناقص هستند. از آنها کپی paste نکنید.
- همواره از شبانگ صحیح استفاده کنید. اگر در حال نوشتن اسکریپت هستید، قرار دادن **#!/usr/bin/env bash** در بالای اسکریپت لازم است. از قلم انداختن این سرآیند یا استفاده از سرآیند **#!/bin/sh** اشتباه است. در حالت اخیر، شما دیگر قادر به استفاده از ویژگیهای Bash نیستید. محدود به اسکریپت‌نویسی استاندارد POSIX می‌شوید(حتی اگر */bin/sh* شما یک پیوند به Bash باشد).
- موقع نوشتن اسکریپت‌های پوسته، از دستور [استفاده نکنید. Bash جایگزین بسیار بهتری دارد: [[. کلمه‌کلیدی [[در Bash از بسیاری جهات قابل اعتمادتر است و به هیچ وجه مزیتی برای چسبیدن به نوع عتیقه آن وجود ندارد. همچنین مقدار زیادی از ویژگی‌های فراهم شده برای [[که در [وجود ندارد(مانند انطباق الگو) را از دست می‌دهید.
- وقت آنست که `...` نیز به فراموشی سپرده شود. این مورد با ساختار *بسط* سازگار نیست. به جای آن از **$(...)** استفاده کنید.
- و به واسطه قدرت شگرف، "کاربرد بیشتر نقل‌قولها!" رشته‌ها و بسط پارامترهای خود را از تفکیک کلمات محافظت کنید. اگر به طور صحیح نقل‌قولی نکنید، تفکیک کلمات نوزادان شما را می‌خورد.
- به جای استفاده از sed یا cut برای کار با رشته‌های ساده در Bash، استفاده از بسط پارامترها را بیاموزید. اگر می‌خواهید پسوند نام فایل را حذف کنید، به جای **`echo "$filename" | sed 's/\.[^.]*$//'`** یا بعضی دایناسورهای دیگر، از **${filename%.*}** استفاده کنید.
- به جای استفاده از expr برای انجام محاسبات ساده، از حساب داخلی استفاده کنید، *مخصوصاً* وقتی‌که فقط مقدار متغیری افزایش می‌یابد. اگر اسکریپتی می‌خوانید که **x=`expr $x + 1`** را به کار برده، این چیزی نیست که شما از آن تقلید کنید.

2. نقل‌قولی کردن

تفکیک کلمه اهریمن درون BASH است که با جدیت تلاش می‌کند تازه‌واردها یا حتی کهنه سربازانی که سپر محافظ خود را زمین می‌گذارند، را غافلگیر کند.

اگر درک نکنید که *تفکیک کلمه* چطور کار می‌کند یا چه وقت اِعمال می‌شود، در استفاده از رشته‌ها و *بسط پارامترها* باید بسیار مراقب باشید. پیشنهاد می‌کنم اگر در مورد آگاهی خود تردید دارید،بیشتر در باره تفکیک کلمه مطالعه کنید.

بهترین روش محافظت خود از این جانور، نقل‌قولی کردن تمام رشته‌هایتان است. نقل‌قول‌ها، رشته‌های شما را به صورت یکپارچه نگاه می‌دارند *تفکیک کلمه* را از گسستن آنها منع می‌کنند. اجازه بدهید تشریح کنم:

```
$ echo Push that word           away from me.
Push that word away from me.
```

```
$ echo "Push that word             away from me."
Push that word             away from me.
```

حال، تصور نکنید که *تفکیک کلمه* برای فرو ریختن فاصله‌هاست. آنچه به طور واقعی در این مثال اتفاق می‌افتد، آنست که در مثال اول، هر یک از کلمات جمله ما به عنوان یک شناسه(argument) جداگانه به echo تحویل داده می‌شود. BASH جمله ما را به کلمات تجزیه می‌کند، از فضای سفید برای تعیین آن که هر شناسه از کجا شروع و به کجا ختم می‌گردد، استفاده می‌نماید. در مثال دوم BASH وادار شده تمام رشته نقل‌قولی شده را با هم نگاه دارد. به این معنا که به شناسه‌ها تفکیک نمی‌شوند و تمام رشته به عنوان یک شناسه به echo تحویل می‌شود. دستور echo همه شناسه‌هایی که به آن داده شود را با یک فاصله مابین آنها در خروجی چاپ می‌کند. اکنون باید اساس *تجزیه کلمه* را متوجه شده باشید.

اینجاست که خطرناک می‌شود: *تفکیک کلمه* فقط در رشته‌های لفظی اتفاق نمی‌افتد. این مطلب بعد از *بسط پارامتر* نیز رخ می‌دهد! در نتیجه، در یک شرایط رکود و خستگی، شاید برای انجام این خطا، به اندازه کافی کند ذهن شده باشید:

```
$ sentence="Push that word             away from me."
$ echo $sentence
Push that word away from me.
$ echo "$sentence"
Push that word             away from me.
```

به طوری که ملاحظه می‌کنید، در دستور echo اول، سهل‌انگاری کرده و نقل‌قول‌ها را از قلم انداخته‌ایم. یک اشتباه بود. BASH جمله ما را بسط داده و سپس از *تفکیک کلمه* برای تجزیه نتیجه بسط به شناسه‌ها جهت تحویل به echo استفاده نموده. در دومین مثال، نقل‌قول‌ها در اطراف *بسط پارامتر* جمله، اطمینان ایجاد می‌کند که BASH آن را به چندین شناسه پیرامون فضاهای سفید تجزیه نمی‌کند.

فقط فاصله‌ها نیستند که باید محافظت شوند. *تفکیک کلمه* در فاصله‌ها، tabها، سطر جدید، یا هر کاراکتر دیگری که در متغیر IFS باشد، صورت می‌گیرد. در اینجا مثال دیگری هست، که به شما نشان می‌دهد، چطور سهل‌انگاری در استفاده از نقل‌قول‌ها، موجب تجزیه نامناسبی می‌شود:

```
$ echo "$(ls -al)"
total 8
drwxr-xr-x   4 lhunath users 1 2007-06-28 13:13 "."/
drwxr-xr-x 102 lhunath users 9 2007-06-28 13:13 ".."/
-rw-r--r--   1 lhunath users 0 2007-06-28 13:13 "a"
-rw-r--r--   1 lhunath users 0 2007-06-28 13:13 "b"
-rw-r--r--   1 lhunath users 0 2007-06-28 13:13 "c"
drwxr-xr-x   2 lhunath users 1 2007-06-28 13:13 "d"/
drwxr-xr-x   2 lhunath users 1 2007-06-28 13:13 "e"/
$ echo $(ls -al)
total 8 drwxr-xr-x 4 lhunath users 1 2007-06-28 13:13 "."/
```

```
drwxr-xr-x 102 lhunath users 9 2007-06-28 13:13 "../" -rw-r--r--
1 lhunath users 0 2007-06-28 13:13 "a" -rw-r--r-- 1 lhunath
users 0 2007-06-28 13:13 "b" -rw-r--r-- 1 lhunath users 0
2007-06-28 13:13 "c" drwxr-xr-x 2 lhunath users 1 2007-06-28
13:13 "d"/ drwxr-xr-x 2 lhunath users 1 2007-06-28 13:13 "e"/
```

در موقعیت‌های بسیار نادری ممکن است، صرفنظر از نقل‌قول‌ها مطلوب باشد. مواردی که شما نیاز به انجام *تفکیک کلمه* داشته باشید:

```
$ friends="Marcus JJ Thomas Michelangelo"
$ for friend in $friends
> do echo "$friend is my friend!"; done
Marcus is my friend!
JJ is my friend!
Thomas is my friend!
Michelangelo is my friend!
```

اما، صادقانه؟ برای چنین حالاتی باید از آرایه‌ها استفاده کنید. آرایه‌ها این فایده را دارند که بدون نیاز به جداکننده صریح، رشته‌ها را جدا می‌کنند. این به آن معنا می‌باشد، که رشته‌های شما می‌توانند شامل هر کاراکتر معتبر(غیرتهی) باشند، بدون آنکه نگران جداکننده بودن کاراکتر باشید(مانند فاصله در مثال فوق). به کار بردن آرایه در مثال فوق ما را قادر می‌کند نام فامیل دوستان را نیز اضافه کنیم:

```
$ friends=( "Marcus The Rich" "JJ The Short" "Timid Thomas" "Michelangelo
The Mobster" )
$ for friend in "${friends[@]}"
> do echo "$friend is my friend!"; done
```

توجه نمایید که در حلقه for قبلی، ما از $friends به شکل *غیر نقل‌قولی* استفاده کردیم. این کار BASH را برای جداکردن رشته friends به کلمات، مجاز می‌سازد. در مثال اخیر، *بسط پارامتر* ${friends[@]}$ را نقل‌قولی کرده‌ایم . نقل‌قولی کردن یک آرایه با شاخص سراسری @ موجب می‌شود BASH آن آرایه را به صورت یک توالی از تمام عناصر که در آن هر یک در علامت نقل‌قول پیچیده شده، بسط بدهد.

3. خوانایی

تقریباً خوانایی کد شما به همان اندازه نتایج آن اهمیت دارد.

احتمال نمی‌رود که اسکریپتی را فقط برای یکبار بنویسید و بعد فراموشش کنید. اگر چنین باشد، باید آنرا اجرا نموده و سپس حذف کنید. اگر خیال دارید استفاده از آن را ادامه دهید، باید طرحی برای نگهداری از آن داشته باشید. برخلاف اطاقتان، کدهای شما نمی‌توانند زمان زیادی کثیف باشند، اما شما به طور دائم رویکردها و شیوه‌های نوینی فرا می‌گیرید. همچنین بینش جدیدی در خصوص چگونگی کاربرد اسکریپت خود به دست می‌آورید. تمام اطلاعات جدیدی که حین تکمیل کد ابتدایی خود به دست می‌آورید، باید به طریقی در حفظ و نگهداری کد شمابه کار گرفته شود، که به طور مداوم آن را بهبود بدهد. کد شما بایستی در جهت کاربر محوری و پایداری بیشتر رشد کند.

به من اعتماد کنید وقتی می‌گویم، هیچ بخشی از کد، هرگز 100% تکمیل نیست، به استثنای، برخی کدهای خیلی کوتاه و عاری از فایده .

برای اینکه سالم نگهداری کدهایتان را آسان‌تر نمایید و به طور مرتب آنها را اصلاح کنید، باید نگاهتان را متوجه خوانایی آنچه می‌نویسید، بنمایید. وقتی پس از مدت طولانی، به اسکریپتی که از آخرین بازبینی آن یکسال گذشته، باز می‌گردید و می‌خواهید آن را اصلاح کنید، یک ویژگی جدید اضافه کنید، یا اشکالی را در آن رفع نمایید، مرا یاد کنید که می‌گویم ترجیحاً این مورد را ببینید:

نمایش یا عدم نمایش شماره سطرها

```
1  friends=( "Marcus The Rich" "JJ The Short" "Timid Thomas"
"Michelangelo The Mobster" )
2
3  # مطالب مهمی در باره دوستانم می‌گوید
4  for name in "${friends[@]}"; do
5
6      #(در لیست) اولین دوست من
7      if [[ $name = ${friends[0]} ]]; then
8          echo $name was my first friend.
9
10     # دوستان من که نامشان با M شروع می‌شود
11     elif [[ $name = M* ]]; then
12         echo "$name starts with an M"
13
14     # دوستان کوتاه من
15     elif [[ " $name " = *" Short "* ]]; then
16         echo "$name is a shorty."
17
18     # دوستانی که زحمت بخاطر سپردن آنها را نمی‌کشم
19     else
20         echo "I kind of forgot what $name is like."
21
22     fi
23 done
```

تا اینکه با موردی مشابه این روبرو شوید:

نمایش یا عدم نمایش شماره سطرها

```
1  x=(       Marcus\ The\ Rich JJ\ The\ Short
```

```
2       Timid\ Thomas Michelangelo\ The\ Mobster)
3     for name in "${x[@]}"
4       do if [ "$name" = "$x" ]; then echo $name was my first friend.
5      elif
6        echo $name    |   \
7       grep -qw Short
8         then echo $name is a shorty.
9      elif [ "x${name:0:1}" = "xM" ]
10         then echo $name starts   with an M; else
11     echo I kind of forgot what $name \
12      is like.; fi; done
```

و بله، می‌دانم که این مثال کمی اغراق‌آمیز است، اما من بعضی کدهای معتبری را دیده‌ام که واقعاً شباهت بسیاری به مثال اخیر دارند.

رای سلامت خودتان این چند نکته را به یاد داشته باشید:

- ***فضای سفید مناسب به شما فضای تنفس می‌دهد.*** کدهایتان را به طور صحیح و نامتناقض دندانه‌دار نمایید. از سطرهای خالی برای جداکردن پاراگراف‌ها یا بلوک‌های منطقی استفاده کنید.
- ***از پوشش با کاراکتر \ اجتناب کنید.*** استفاده زیاد از این کاراکترِ گریز، موجب حواس‌پرتی و آشفتگی ذهنی می‌گردد. حتی در مثالهای کوچک، تقلای ذهنی بیشتری برای فهمیدن a\ b\ c نسبت به 'a b c' صرف می‌شود.
- ***روش تفکرتان را، یادداشت کنید، قبل از اینکه آن را فراموش کنید.*** ممکن است دریابید، کدی که کاملاً متعارف حس می‌شود، می‌تواند موضوع *"چه جهنمی فکر می‌کردم، وقتی این را نوشتم؟"* یا *"تصور انجام چه کاری از این داشتم؟"*.
- ***سازگاری از ناراحتی ذهن پیشگیری می‌کند.*** در شیوه نام‌گذاری استوار باشید. در استفاده از حروف بزرگ سازگار باشید. در استفاده خود از ویژگی‌های پوسته پایدار باشید. در کدنویسی، برخلاف اطاق‌خواب، **خوبست** ساده و قابل پیش بینی باشید.

4. بررسی‌های Bash

فرمان `test` که به عنوان `[` نیز شناخته شده، یک برنامه کاربردی است که به طور معمول جایی در */usr/bin* یا */bin* استقرار می‌یابد و توسط برنامه‌نویس پوسته برای اجرای آزمایش‌های معینی با متغیرها و فایلها، خیلی زیاد به کار می‌رود. در تعدادی از پوسته‌ها، از جمله Bash, دستور `test` به صورت دستور داخلی پوسته نیز پیاده‌سازی گردیده است.

این مورد می‌تواند نتایج شگفت‌انگیزی فراهم نماید، به ویژه برای آنان که شروع به اسکریپت‌نویسی پوسته می‌نمایند و تصور می‌کنند `[ ]` بخشی از دستور زبان پوسته است.

اگر از پوسته `sh` استفاده می‌کنید، انتخاب کمی دارید و استفاده از `test` تنها راه انجام اکثر بررسی‌هایتان می‌باشد.

گرچه اگر از Bash در اسکریپت‌نویسی استفاده می‌کنید(و من فرض می‌کنم چنین است، چون در حال خواندن این راهنما هستید)، پس می‌توانید از کلید واژه `[[` نیز استفاده کنید. هر چند بازهم از خیلی جهات همچون یک فرمان رفتار می‌کند، چندین مزیت نیز نسبت به فرمان سنتی `test` ارائه می‌کند.

اجازه بدهید تشریح کنم که چگونه [[می‌تواند با فرمان test تعویض شود، وچطور می‌تواند به شما کمک کند از برخی اشتباهات متداول در کاربرد test پرهیز نمایید:

```
$ var=''
$ [ $var = '' ] && echo True
-bash: [: =: unary operator expected
$ [ "$var" = '' ] && echo True
True
$ [[ $var = '' ]] && echo True
True
```

قسمت ['' = $var] به ['' =] بسط داده می‌شود. اولین کاری که دستور test انجام می‌دهد، شمارش شناسه‌هایش می‌باشد. چون [را به کار برده‌ایم، باید شناسه الزامی] در انتها را کنار بگذاریم. در مثال اول، test دو شناسه می‌بیند: = و ''. حالا می‌داند که دو شناسه دارد، اولی باید *unary operator* (یک عملگر که یک عملوند می‌گیرد). اما = عملگر یگانی(unary operator) نیست(یک عملگر binary است که دو عملوند نیاز دارد)، بنابراین، test نمی‌تواند کار کند.

بله، test متغیر تهی $var را نمی‌بیند، زیرا BASH قبل از اینکه test حتی بتواند آن را ببیند، به هیچ بسطش داده است. نتیجه اخلاقی؟ استفاده بیشتر از نقل‌قول‌ها! کاربرد نقل‌قول‌ها در قسمت، ['' = "$var"] موجب بسط آن به ['' = ""] می‌گردد و test مشکلی ندارد.

حال آنکه، [[می‌تواند تمام دستور را قبل از اینکه بسط داده شود، ببیند. می‌تواند $var را ببیند، و نه بسط $var را. در نتیجه، نیازی به نقل‌قولها نمی‌باشد! [[مطمئن‌تر است.

```
$ var=
$ [ "$var" < a ] && echo True
-bash: a: No such file or directory
$ [ "$var" \< a ] && echo True
True
$ [[ $var < a ]] && echo True
True
```

در این مثال سعی نموده‌ایم یک مقایسه رشته‌ای بین یک متغیر تهی و 'a' انجام بدهیم. شگفت‌زده می‌شویم با دیدن آنکه از اولین تلاش ما True حاصل نمی‌گردد، ولواینکه تصور می‌کردیم، می‌شود. درعوض، با خطای عجیبی که دلالت بر تلاش BASH برای باز کردن فایلی به نام 'a' می‌نماید، مواجه می‌شویم.

ما توسط *تغییر مسیر فایل* گزیده شده‌ایم. چون test دقیقاً یک برنامه کاربردی است، کاراکتر < در دستور ما به جای عملگر مقایسه رشته‌ای برای test به عنوان عملگر *تغییر مسیر فایل* تفسیر شده است(همانطور که باید می‌شد). BASH دستور باز کردن فایل 'a' و اتصال آن به stdin برای خواندن را دریافت نموده. برای ممانعت از این مورد، لازم است, < را با کاراکتر گریز پوشش دهیم، به طوری که به جای BASH برنامه test عملگر را دریافت کند. این دومین تلاش ما را تشکیل داد.

با استفاده از [[می‌توانیم روی‌هم‌رفته از نابسامانی اجتناب نماییم. [[عملگر < را قبل از آنکه BASH آنرا برای *تغییر مسیر* دریافت کند، می‌بیند ــ مشکل رفع می‌شود. یکبار دیگر [[مطمئن‌تر است.

حتی خطرناک‌تر، استفاده از عملگر < به جای عملگر > مثال قبلی است. چون > ماشهٔ *تغییر مسیر* خروجی را می‌کشد، فایلی به نام'a' ایجاد خواهد نمود. در نتیجه، هیچ پیغام خطای هشداردهنده‌ای برای ما صادر نمی‌شود که بدانیم مرتکب اشتباه شده‌ایم! به جای آن، فقط اسکریپت ما خراب می‌شود. حتی وخیم‌تر، شاید فایل مهمی را رونویسی کنیم! برای ما *حدس زدن* آنکه مشکل کجاست، سخت است:

```
$ var=a
$ [ "$var" > b ] && echo True || echo False
True
$ [[ "$var" > b ]] && echo True || echo False
False
```

دو نتیجه متفاوت، شگرف. به من اعتماد کنید، وقتی می‌گویم، همیشه می‌توانید به [[بیشتر از [اطمینان کنید. ["$var" > b] به ["a"] بسط یافته و خروجی به یک فایل جدید به نام 'b' تغییر مسیر داده می‌شود. چون ["a"] در واقع همان [-n "a"] می‌باشد و اساساً بررسی می‌شود که آیا رشته "a" غیرتهی است، نتیجه بررسی موفق است و echo True اجرا می‌شود.

با کاربرد [[انتظار ما که مقایسه "a" در برابر "b" است، برآورده می‌شود، و نظر به اینکه همه می‌دانیم "a" قبل از "b" مرتب می‌شود، ماشه اجرای دستور echo False کشیده می‌شود. و این چگونگی آنست که اسکریپت شما می‌تواند بدون پی‌بردن شما ناموفق بشود. هر چند که، شما یک فایل شبهه برانگیزی به نام 'b' نیز در دایرکتوری جاری خواهید داشت.

بنابراین به من باور داشته باشید، وقتی می‌گویم، [[مطمئن‌تر از [است. زیرا هر کسی به ناچار خطاهای برنامه‌نویسی را ایجاد می‌کند. افراد به طور معمول قصد ندارند، باگهایی در کدهایشان ارائه کنند. اما اتفاق می‌افتد. بنابراین مدعی نشوید که از [استفاده می‌کنید و *"مراقب خواهید بود که چنین اشتباهاتی مرتکب نگردید"*، زیرا می‌توانم شما را مجاب کنم که مرتکب خواهید شد.

گذشته ازاین، [[ویژگیهای زیر را علاوه بر [ارائه می‌کند:

- [[می تواند مطابقت الگوی جانشین(glob) را انجام دهد:

  [[abc = a*]]

- [[می‌تواند انطباق الگوی عبارت باقاعده(regex) را (از Bash نگارش 3.1 به بعد) انجام دهد:

  [[abb =~ ab+]]

تنها برتری test قابلیت حمل آن است.

5. هرگز این موارد را انجام ندهید

پوسته Bash امکان انجام کارهای بسیاری برای شما فراهم می‌کند، ارائه قابلیت انعطاف‌پذیری قابل ملاحظه به شما. متأسفانه، خیلی کم شما را از سوءمصرف و دیگر رفتارهای نامطلوب، بر حذر می‌دارد. امید می‌رود، اشخاص خودشان دریابند که از برخی مسائل معین باید به هر قیمتی پرهیز نمایند.

متأسفانه بسیاری اشخاص به اندازه کافی دقیق و مراقب نیستند که بخواهند خودشان موشکافی کنند. آنها بدون اندیشیدن در مورد مسائل، می‌نویسند و بسیاری از اسکریپتهای خطرناک و مهیب به محیطهای تولید و یا توزیع‌های لینوکس ختم می‌شود. نتیجه اینها، و حتی اسکریپتهای خیلی شخصی شما در یک شرایط اهمال اغلب می‌تواند مصیبت‌آمیز بشود.

برای پاکیزگی اسکریپت‌هایتان، و به خاطر تمام افراد بشر، هرگز هیچ موردی از سطور زیر را انجام ندهید:

- `ls -l | awk '{ print $8 }'`

هرگز تجزیه خروجی فرمان `ls` را انجام ندهید! خروجی فرمان `ls` به چند دلیل نمی‌تواند قابل اعتماد باشد.

1. اول، اگر نام فایلها شامل کاراکترهای پشتیبانی نشده زبان محلی شما باشد، `ls` نامها را خُرد خواهد نمود. در نتیجه، خروجی حاصل از تجزیه نام فایلها توسط `ls`، هرگز تضمین نمی‌شود که واقعاً همان نامهایی که شما قادر به یافتن آنها می‌باشید را به شما بدهد. `ls` ممکن است بعضی کاراکترها در نام فایل را با کاراکتر علامت سؤال تعویض نماید.

2. دوم، `ls` سطرهای داده‌ها را بر اساس کاراکتر سطر جدید تفکیک می‌کند. به این طریق، هر تکه از اطلاعات یک فایل در یک سطر است. متأسفانه، نام فایلها نیز خودشان می‌توانند شامل سطر جدید باشند. این به معنی آنست که اگر شما فایلی در دایرکتوری جاری با نام شامل کاراکتر سطر جدید داشته باشید، کاملاً نتیجه تجزیه شما را درهم می‌ریزد و اسکریپت شکست می‌خورد!

3. آخر از همه، اما نه کم اهمیت‌تر، قالب خروجی فرمان `ls -l` تضمین نمی‌شود که در تمام پلاتفرمها همسان باشد. برخی سیستمها به طور پیش‌فرض شماره شناسایی گروه را از قلم می‌اندازند، و اثر گزینه `-g` را معکوس می‌نمایند. بعضی سیستمها از دو فیلد برای زمان ویرایش و برخی از سه فیلد برای آن استفاده می‌کنند. در سیستمهایی که از سه فیلد استفاده می‌کنند، فیلد سوم می‌تواند سال یا یک HH:MM الحاقی، نسبت به سن فایل، باشد.

در موقعیت‌های بسیاری جایگزین‌هایی برای ls وجود دارد. اگر لازم است که شما با زمان ویرایش فایل کار کنید، به طور نمونه می‌توانید از بررسی‌های Bash استفاده کنید. اگر هیچ یک از آنها میسر نباشد، *پیشنهاد می‌کنم زبان متفاوتی، همچون پرل یا python انتخاب کنید.*

- `if echo "$file" | fgrep .txt; then`
  `ls *.txt | grep story`

هرگز نام فایل ها را با `grep` بررسی یا فیلتر نکنید! غیر از آنکه الگوی `grep` شما واقعاً هوشمند باشد، این کار احتمالاً قابل اطمینان نخواهد بود.

در نمونه اول مثال فوق، بررسی با هر دو مورد story.txt و story.txt.exe منطبق می‌گردد. اگر الگوهایی از `grep` ایجاد کنید که به اندازه کافی هوشمند باشند، احتمالاً آنها به قدری زشت، حجیم و ناخوانا می‌شوند، که با این وجود نباید از آنها استفاده کنید.

جایگزین آن globbing نامیده می‌شود(**مترجم:** من در این ترجمه گاهی به جای آن کلمه«جانشینی» را به کاربرده‌ام). Bash یک ویژگی به نام *بسط نام‌مسیر* دارد. این ویژگی به شما کمک می‌کند، که تمام فایلهایی که با یک الگوی معین مطابقت دارند را به شمار آورید. همچنین، می‌توانید از globها جهت بررسی آنکه یک نام فایل، آیا با یک الگوی معین مطابقت می‌کند، (در یک دستور case یا [[ ) استفاده نمایید .

- `cat file | grep pattern`

برنامه `cat` را برای خوراندن محتویات یک فایل منفرد به یک فیلتر به کار نبرید. `cat` یک ابزار مورد استفاده برای الحاق محتویات چند فایل با یکدیگر است.

برای تغذیه محتویات فایلی به یک پردازش، احتمالاً می‌توانید نام فایل را به عنوان شناسه تحویل برنامه مورد نظر(مانند `grep 'pattern' /my/file` یا `sed 'expression' /my/file` و غیره) بدهید.

اگر مستندات برنامه هیچ راهی برای انجام این کار تعیین نکرده است، باید از تغییر مسیر استفاده کنید (`read column1 column2 < /my/file` یا `tr ' ' '\n' < /my/file` و غیره).

- `for line in $(<file); do`

از حلقه for برای خواندن سطرهای یک فایل استفاده نکنیم. به جای آن حلقه while read را به کار ببریم.

- `for number in $(seq 1 10); do`

به خاطر خدا و به خاطر تمام مقدسات، از برنامه `seq` برای شمارش استفاده نکنید.

Bash به اندازه کافی در انجام شمارش توانمند است. نیازی به یک برنامه خارجی(مخصوصاً یک برنامه تک سکویی) برای انجام محاسبه و ارسال آن به خروجی Bash جهت تفکیک کلمه، ندارید. ترکیب دستوری for را قبلاً آموخته‌اید!

باید در Bash نگارش 3 به بعد، از این: `for number in {1..10}`، یا در Bash 2 از این: `for ((i=1; i<=10; i++))` استفاده کنید.

اگر شما عملاً یک جریانی از اعداد که با کاراکتر سطر جدید از هم جدا شده‌اند، هنگام بررسی ورودی می‌خواهید، این مورد را در نظر بگیرید: `printf '%d\n' {1..10}`

- `` i=`expr $i + 1` ``

`expr` یک عتیقه رُم باستان است. آن را به کار نبرید.

این برنامه در اسکریپت‌های نوشته شده برای پوسته‌هایی با امکانات بسیار محدود، به کار می‌رفت. اساساً با استفاده از آن در حال ایجاد یک پردازش جدید هستید که برنامه C دیگری برای انجام برخی محاسبات را برایتان فراخوانی نماید و نتایج را به صورت رشته به bashتحویل بدهد. Bash تمام اینها را خودش می‌تواند خیلی سریعتر، و به طور قابل اعتمادتر (بدون تبدیل عدد به ->رشته -> به عدد) و در همه حال بهتر، انجام بدهد.

شما در Bash باید از این استفاده کنید: `((i++))` یا `let i++`

حتی پوسته POSIX بورن می‌تواند محاسبات را انجام بدهد: `i=$((i+1))`. فقط فاقد عملگر `++` و دستور `((...))` می‌باشد(فقط عبارت جایگزینی `$((...))` را دارد).

6. اشکالزدایی

خیلی وقتها، خودتان را مستأصل می‌بینید که چرا، اسکریپت شما آنگونه عمل نمی‌کند، که شما می‌خواهید. حل این مسئله همواره، موضوع

درک عمومی و شیوه‌های اشکال‌یابی است.

تشخیص مشکل

بدون آنکه دقیقاً بدانید مشکل چیست، به احتمال بسیار زیاد، خیلی زود نمی‌توانید ***چاره‌سازی*** نمایید. بنابراین مطمئن شوید، که به طور دقیق می‌دانید چه چیز اشتباه است. علائم و پیغام‌های خطا را بررسی و ارزیابی کنید.

سعی کنید مشکل را به صورت یک جمله با قاعده بیان کنید. چون اگر بخواهید از دیگران در حل مشکل کمک بگیرید نیز، این کار خیلی ضروری می‌باشد. شما که نمی‌خواهید آنها تمام اسکریپت شما را بازنویسی نمایند، همینطور هم نمی‌خواهید آنها سرتاسر اسکریپت شما را بازبینی یا آنرا اجرا کنند تاببینند که چه مشکلی پیش می‌اید. نه، **شما** لازم است مسئله را کاملاً برای خودتان و هر کسی که بخواهد کمکتان کند، روشن سازید. ***وگرنه لازمه‌اش، منتظرماندن تا زمانی است که نوع بشر وسایل تله‌پاتی را اختراع کند.***

حداقل‌سازی کد اصلی

اگر اشکال‌یابی اسکریپت را شروع می‌کنید، به خودتان الهام خدایی اهدا نکنید، مورد دیگری که باید انجام دهید، کوشش جهت حداقل‌سازی کد اصلی برای مجزا نمودن مسئله می‌باشد.

نگران حفظ توانایی اسکریپت خود نباشید. تنها چیزی که باید باقی نگاه دارید، منطق قطعه کد اصلی است، که به نظر مشکل آفرین می‌باشد.

اغلب، بهترین روش آنست که اسکریپت خود را در یک فایل جدید کپی نموده و شروع به حذف نمودن هر آن چیزی که به نظر می‌رسد نامربوط است، بنمایید. به طور جایگزین، می‌توانید یک اسکریپت جدید که کاری مشابه همان کد انجام می‌دهد، بسازید، و ساختار را تا ایجاد دوباره مشکل ادامه دهید.

به مجرد اینکه، موردی که مشکل ایجاد نموده را حذف کردید، دست بکشید(یا موردی که اضافه نمودنش دوباره آن مشکل را ظاهر می‌کند)، شما کشف کرده‌اید که مشکل در کجا قرار دارد. حتی اگر به طور دقیق به آن نرسیده‌اید، حداقل دیگر به یک اسکریپت حجیم خیره نمی‌شوید، بلکه امیدوارانه، با کوتوله‌ای نه بیش از 3 تا 7 سطر، مواجه هستید.

برای مثال، اگر اسکریپتی دارید که باز کردن فایلهای تصویری موجود در شاخه image را برحسب تاریخ برای شما انجام می‌دهد، و بنا به دلایلی، تکرار روی فایلهای دایرکتوری را نمی‌توانید به طور صحیح پیش ببرید، کافی است اسکریپت را تا اندازه این قطعه کُد کاهش بدهید:

```
for image in $(ls -R "$imgFolder"); do
    echo "$image"
done
```

اسکریپت واقعی شما به مراتب پیچیده‌تر از این خواهد بود، و درون حلقه for آن نیز طولانی‌تر خواهد بود. اما اصل مشکل این کُد است. وقتی مشکل را به این اندازه محدود کنید، ممکن است دیدن آنکه با چه مشکلی مواجه هستید، آسانتر باشد. دستور echo اجزاء نام فایلها را بیرون می‌دهد، به نظر می‌رسد تمام فضاهای سفید با کاراکتر سطرجدید تعویض گردیده‌اند. باید اینطور باشد، زیرا echo برای هرقسمت از نام فایل که منتهی به فضای سفید است، اجرا می‌شود، نه برای هر نام تصویر(در نتیجه، به نظر می‌رسد، خروجی نام فایلهایی که دارای فضای سفید می‌باشند، تفکیک شده‌اند). با این کُد کاهش یافته، دیدن آنکه مسبب واقعی، جمله for است که خروجی ls را به کلمات تفکیک نموده، آسانتر است. هرگز ls را در اسکریپتها به کار نبرید، مگر آنکه بخواهید خروجی آنرا به کاربر نمایش بدهید.

ما نمی‌توانیم glob بازگشتی به کار ببریم(مگر در bash نگارش 4)، بنابراین باید دستور find را برای به دست آوردن نام فایلها به کار بگیریم. یک راه اصلاح آن، چنین خواهد بود:

```
find "$imgFolder" -print0 | while IFS= read -r -d '' image; do
    echo "$image"
done
```

اکنون که مشکل را در این مثال کوچک برطرف نموده‌اید، برگشتن و ترکیب کردن آن با اسکریپ اصلی آسان است.

	فعال نمودن وضعیت اشکال‌یابی BASH	

اگر بازهم خطای روش‌هایتان را نمی‌بینید، شاید وضعیت اشکال‌یابی BASH برای دیدن مشکل در میان کُد به شما کمک نماید.

موقعی که BASH با گزینه -x اجرا می‌شود، این وضعیت فعال می‌گردد، هر دستوری را قبل از اجرا در خروجی چاپ می‌کند. همینطورهم ، بعد از هر بسط و گسترشی که انجام شده است. در نتیجه، به طور دقیق می‌توانید ببینید با اجرای هر سطر کُد، چه اتفاقی رخ می‌دهد. به نقل‌قول‌های استفاده شده خیلی دقیق توجه نمایید. BASH نقل‌قولها را برای نشان دادن آنکه دقیقاً کدام رشته به عنوان یک شناسه منفرد عبور داده شده، به کار می‌برد.

سه روش برای فعال کردن این وضعیت موجود است.

- اجرای اسکریپت به صورت bash -x:

```
$ bash -x ./mybrokenscript
```

- ویرایش سرآیند اسکریپت:

```
#!/bin/bash -x
[.. script ..]
```

یا:

```
#!/usr/bin/env bash
set -x
```

- یا اضافه نمودن set -x در جایی از کُد برای فعال کردن این حالت، منحصراً برای قطعه معینی از کُد:

```
[..کدهای بی ارتباط..]
set -x
```

```
    [..قطعه کد مرتبط..]
    set +x
    [..کدهای بی ارتباط..]
```

- اگر `set -x` شما مقدار زیادی خروجی دارد، ویژگی دلپذیر bash نگارش 4.1 و بالاتر، متغیر BASH_XTRACEFD است. این متغیر امکان تعیین یک توصیف‌گر فایل برای هدایت خروجی اشکال‌های `set -x` به آن را فراهم می‌کند. در نسخه‌های قدیمی‌تر bash، همواره این خروجی به stderr می‌رفت، و اگر جدا کردن آن از خروجی معمولی ناممکن نبود، ولی دشوار بود. این هم یک روش دلپسند برای کاربرد آن:

```
# بخش set -x را در یک فایل کپی می‌کند
# با یک نام فایل به عنوان $1 آنرا فعال می‌کند
# اگر پارامتری وجود نداشته باشد آن را غیر فعال می‌کند
# شماره fd 4 نباید در جای دیگری از اسکریپت استفاده شده باشد
setx_output()
{
    if [[ $1 ]]; then
        exec 4>>"$1"
        BASH_XTRACEFD=4
        set -x
    else
        set +x
        exec 4>&-
    fi
}
```

اگر اسکریپت‌های پیچیده و آشفته‌ای دارید، شاید تغییر محتوی متغیر PS4 قبل از برقراری اشکال‌یابی با `-x` را سودمند بیابید:

```
export PS4='+$BASH_SOURCE:$LINENO:$FUNCNAME: '
```

	Step your code	

اگر خروجی اشکال‌یابی از نظر شما خیلی سریع عبور می‌کند، می‌توانید کُد-مرحله‌ای را فعال کنید. کُد زیر از *DEBUG* دستور `trap` برای اطلاع به کاربر در باره دستوری که اجرا خواهد شد و انتظار برای تایید پیشرفت، استفاده می‌کند. این کُد را در محلی از اسکریپت خود که می‌خواهید مرحله‌ای بشود، قرار دهید:

```
trap '(read -p "[$BASH_SOURCE:$LINENO] $BASH_COMMAND?")'
```

```
DEBUG
```

اشکالزدای BASH

پروژه اشکالزدای Bash یک اشکال‌یاب gdb-مانند، در آدرس http://bashdb.sourceforge.net/ است.

اشکال‌یاب فوق به شما کمک می‌کند در سرتاسر اسکریپت حرکت نموده و اشکالهای آن را پیگردی و پیدا کنید.

بازخوانی مستندات

اگر هنوز به نظر می‌رسد اسکریپت برایتان قابل قبول نیست، شاید ادراک شما از روش انجام کارها اشتباه است. به مستندات(یا این راهنما) رجوع کنید، برای ارزیابی آن که آیا فرمانها درست همانگونه که شما در مورد آنها می‌اندیشید کارمی‌کنند، یا دستور زبان کاربرد آنها همانطور است که شما فکر می‌کنید. بسیاری اوقات، اشخاص در باره چگونگی کارکرد for ، عملکرد *تفکیک کلمه*، یا اینکه چگونه باید از نقل‌قولها استفاده کنند، درک ناقصی دارند.

نکته‌ها را حفظ کنید و تکرارهای راهنمایی این آموزش را خوب به خاطر بسپارید. اینها غالباً برای پرهیز از مشکلات در اسکریپت‌ها به شما کمک می‌کنند.

من این مطلب را در بخش *اسکریپتهای* این راهنما نیز اشاره کرده‌ام، اما تکرار آن در اینجا هم با ارزش است. اول از همه، اطمینان حاصل کنید که سرآیند اسکریپت شما به راستی `#!` */bin/bash* است. اگر از قلم افتاده یا اگر موردی مانند این `#! /bin/sh` است، پس شما سزاوار مشکلاتی که دارید، هستید. چون به آن معنی است که احتمالاً حتی از BASH برای اجرای اسکریپت خود استفاده نمی‌کنید. به طور وضوح علت مشکل همانست. همچنین، اطمینان حاصل کنید که کاراکترهای *رفتن سر سطر*(*CF*) در انتهای سطرها ندارید. این به سبب اسکریپتهایی است که در *ویندوز* نوشته شده‌اند. می‌توانید به آسانی به طور مساعدی اینها را به این صورت پاک کنید:

```
$ tr -d '\r' < myscript > tmp && mv tmp myscript
```

پرسش و پاسخها و Pitfallها را بخوانید

صفحه‌های پرسش و پاسخهای رایج و دامهای Bash تصورات غلط معمول و مشکلاتی که دیگران در اسکریپت‌های BASH با آنها روبرو شده‌اند را شرح می‌دهند. احتمال بسیار دارد، مشکل شما به شکلی در آنجا تشریح شده باشد.

برای اینکه قادر به یافتن مشکل خود در آنجا باشید، باید مسئله را به طور کاملاً واضح *شناسایی* کرده باشید. شما باید بدانید که در جستجوی چه چیزی می‌باشید.

از ما در IRC بپرسید

در 24 ساعت هفت روز هفته، اکثراً افرادی در کانال **#bash** حضور دارند. این کانال در شبکه IRC *freenode* مستقر است. برای رسیدن به ما، لازم است یک سرویس‌گیرنده IRC داشته باشید. از طریق آن به irc.freenode.net و /join #bash ارتباط برقرار نمایید.

مطمئن شوید که می‌دانید مشکل واقعی چیست و آنرا به صورت مرحله‌ای روی کاغذ بیاورید، به طوری که خوب بتوانید آنرا شرح بدهید. ما دوست نداریم در مورد مسائل حدس بزنیم. با توضیح آنکه اسکریپت شما چه کاری *باید* انجام بدهد شروع کنید.

نکته دیگر، لطفاً قبل از ورود به #bash صفحه : XyProblem را ملاحظه نمایید.