Bölüm 5. Kendi Kabuğunuzu Yazmak

Bir şeyi programlayana kadar onu gerçekten anlarsınız.
--GRR

Giriş

Geçen bölümde UNIX komutlarını kullanarak bir kabuk programının nasıl kullanılacağı anlatılmıştı. Kabuk, bir terminal aracılığıyla kullanıcıyla etkileşime giren veya bir dosyadan girdi alan ve İşletim Sistemine iletilen bir dizi komutu yürüten bir programdır. Bu bölümde kendi kabuk programınızı nasıl yazacağınızı öğreneceksiniz.

Kabuk Programları

Kabuk programı, bilgisayarla etkileşime girmeyi sağlayan bir uygulamadır. Bir kabukta kullanıcı programları çalıştırabilir ve ayrıca bir dosyadan gelecek girdiyi ve bir dosyadan gelecek çıktıyı yönlendirebilir. Kabuklar ayrıca if, for, while, fonksiyonlar, değişkenler vb. gibi programlama yapıları sağlar. Ek olarak, kabuk programları satır düzenleme, geçmiş, dosya tamamlama, joker karakterler, ortam değişkeni genişletme ve programlama yapıları gibi özellikler sunar. İşte UNIX'teki en popüler kabuk programlarının bir listesi:

sh	Kabuk Programı. UNIX'teki orijinal kabuk programı.
csh	C Kabuğu. sh'nin geliştirilmiş bir versiyonu.
tcsh	Csh'ın satır düzenleme özelliğine sahip bir sürümü.
ksh	Korn Kabuğu. Tüm gelişmiş kabukların babası.
bash	GNU kabuğu. Tüm kabuk programlarının en iyisini alır. Şu anda en yaygın kabuk programıdır.

Komut satırı kabuklarına ek olarak, çoğu kullanıcı için bilgisayar kullanımını basitleştiren Windows Masaüstü, MacOS Finder veya Linux Gnome ve KDE gibi Grafik Kabuklar da vardır. Bununla birlikte, bu grafik kabuklar, karmaşık komut dizilerini tekrar tekrar veya dostça, ancak sınırlı grafik iletişim kutuları ve kontrollerde bulunmayan parametrelerle yürütmek isteyen güçlü kullanıcılar için komut satırı kabuklarının yerini tutmaz.

Bir Kabuk Programının Parçaları

Kabuk uygulaması üç bölüme ayrılmıştır: **Ayrıştırıcı**, **Yürütücü** ve **Kabuk Alt Sistemleri.**

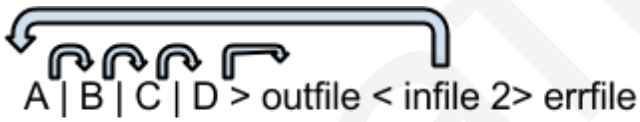
Ayrıştırıcı

Ayrıştırıcı, "ls al" gibi komut satırını okuyan ve çalıştırılacak komutları saklayacak ***Komut Tablosu*** adı verilen bir veri yapısına yerleştiren yazılım bileşenidir.

İnfaz Memuru

Yürütücü, ayrıştırıcı tarafından oluşturulan komut tablosunu alacak ve dizideki her SimpleCommand için yeni bir süreç oluşturacaktır. Gerekirse, bir sürecin çıktısını bir sonrakinin girdisine iletmek için borular da oluşturacaktır. Ayrıca, herhangi bir yönlendirme varsa standart girdi, standart çıktı ve standart hatayı yeniden yönlendirecektir.

Aşağıdaki şekilde "A | B | C | D" komut satırı gösterilmektedir. Ayrıştırıcı tarafından algılanan **"< *infile"*** gibi bir yönlendirme varsa, ilk SimpleCommand A'nın girdisi ***infile'***dan yönlendirilir. **"> *outfile***" gibi bir çıktı yönlendirmesi varsa, son SimpleCommand'ın (D) çıktısını ***outfile'a yönlendirir***.



">& errfile" gibi **errfile'a** bir yönlendirme varsa, tüm SimpleCommand süreçlerinin stderr'ı **errfile'a yönlendirilecektir**.

Kabuk Alt Sistemleri

Kabuğunuzu tamamlayan diğer alt sistemler şunlardır:

- Ortam Değişkenleri: ${VAR} biçimindeki ifadeler ilgili ortam değişkeni ile genişletilir. Ayrıca kabuk ortam değişkenlerini ayarlayabilmeli, genişletebilmeli ve yazdırabilmelidir.
- Joker karakterler: a*a biçimindeki bağımsız değişkenler, yerel dizinde ve birden çok dizinde kendileriyle eşleşen tüm dosyalara genişletilir .
- Alt kabuklar: `` (backticks) arasındaki argümanlar çalıştırılır ve çıktı kabuğa girdi olarak gönderilir.

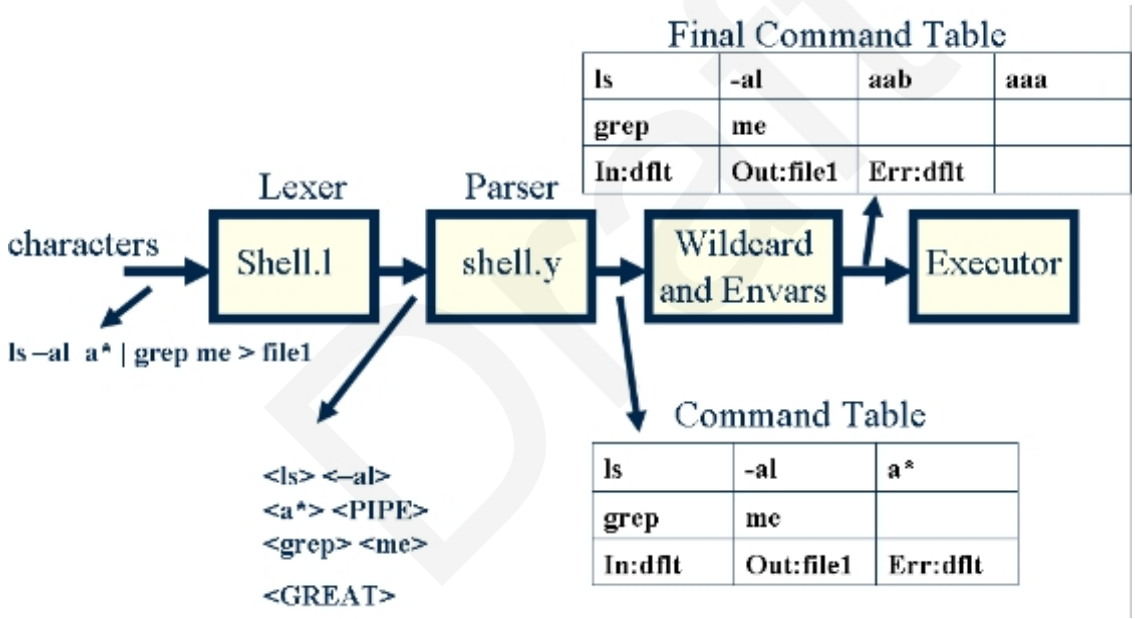
Bu bölümdeki adımları izleyerek kendi kabuğunuzu oluşturmanızı şiddetle tavsiye ederiz. Kendi kabuğunuzu uygulamak, kabuk yorumlayıcı uygulamalarının ve işletim sisteminin nasıl etkileşime girdiğini çok iyi anlamanızı sağlayacaktır. Ayrıca, iş görüşmeniz sırasında gelecekteki işverenlere göstermek için iyi bir proje olacaktır.

Ayrıştırıcıyı uygulamak için Lex ve Yacc kullanma

Ayrıştırıcınızı uygulamak için iki UNIX aracı kullanacaksınız: Lex ve Yacc. Bu araçlar derleyicileri, yorumlayıcıları ve önişlemcileri uygulamak için kullanılır. Bu araçları kullanmak için derleyici teorisini bilmenize gerek yoktur. Bu araçlar hakkında bilmeniz gereken her şey bu bölümde açıklanacaktır.

Bir ayrıştırıcı iki bölüme ayrılır: bir **Sözcüksel Çözümleyici** veya **Lexer** girdi karakterlerini alır ve karakterleri **belirteç** adı verilen sözcükler halinde bir araya getirir ve belirteçleri bir dilbilgisine göre işleyen ve komut tablosunu oluşturan bir **Ayrıştırıcı**.

Burada Lexer, Parser ve diğer bileşenlerle birlikte Shell'in bir diyagramı bulunmaktadır.



Belirteçler düzenli ifadeler kullanılarak **shell.l** dosyasında tanımlanır. **shell.l** dosyası, sözcüksel analizörü oluşturan lex adlı bir programla işlenir.

Ayrıştırıcı tarafından kullanılan d i l b i l g i s i kuralları, aşağıda açıkladığımız sözdizimi ifadeleri kullanılarak **shell.y** adlı bir dosyada tanımlanır. **shell.y, bir ayrıştırıcı** programı oluşturan **yacc adlı bir** programla işlenir. Hem lex hem de yacc UNIX'in standart komutlarıdır. Bu komutlar çok karmaşık derleyicileri gerçeklemek için kullanılabilir. Kabuk için, kabuğun ihtiyaç duyduğu komut tablosunu oluşturmak için Lex ve Yacc'nin bir alt kümesini kullanacağız.

Ayrıştırıcımızın komut satırlarını yorumlamasını ve çalıştırıcımıza doğru bilgileri sağlamasını sağlamak için ***shell.l*** ve ***shell.y'de*** aşağıdaki grameri uygulamanız gerekir.

```
cmd [arg]* [ | cmd [arg]* ]*
    [ [> dosya adı] [< dosya adı] [ >& dosya adı] [>> dosya adı] [>>& dosya adı] ]* [&]
```

Şekil 4: BackusNaur Formunda Kabuk Grameri

Bu gramer *"Backus-Naur Formu"* adı verilen bir formatta yazılır. Örneğin **cmd [arg]*** bir komut, ***cmd,*** ardından 0 veya daha fazla ***argüman, arg*** anlamına gelir. **cmd [arg]*]*** ifadesi, 0 veya daha fazla sayıda olabilen isteğe bağlı boru alt komutlarını temsil eder. **>filename]** ifadesi, 0 veya 1 **>filename** yönlendirmesi olabileceği anlamına gelir. Sondaki ***[&], &*** karakterinin isteğe bağlı olduğu anlamına gelir.

Bu gramer tarafından kabul edilen komut örnekleri şunlardır:

ls -al
ls -al > out
ls -al | sort >& out
awk -f x.awk | sort -u < dosya içi > dosya dışı &

Komut Tablosu

Komut Tablosu, ***SimpleCommand*** struct'larından oluşan bir dizidir. Bir ***SimpleCommand*** *struct,* boru hattındaki tek bir girişin komut ve argümanları için üyeler içerir. Ayrıştırıcı, komut satırına da bakacak ve komutta bulunan sembollere (yani < infile veya > outfile) dayalı olarak herhangi bir girdi veya çıktı yönlendirmesi olup olmadığını belirleyecektir.

İşte bir komut örneği ve oluşturduğu ***Komut Tablosu:***

komuta

ls al | grep me > dosya1

Komut Tablosu

SimpleCommmand dizisi:

0:	**ls**	**al**	**NULL**
1:	**grep**	**Ben**	**NULL**

IO Yeniden Yönlendirme:

içinde: varsayılan	**çıkış: dosya1**	**err: varsayılan**

Komut tablosunu temsil etmek için aşağıdaki sınıfları kullanacağız: **Komut** ve **SimpleCommand**.

```
// Komut Veri Yapısı

// Basit bir komutu ve argümanları tanımlar
struct SimpleCommand {
        // Şu anda önceden ayrılmış argümanlar için
        kullanılabilir alan int _numberOfAvailableArguments.

        // Argüman sayısı int
        _numberOfArguments.

        // Argüman dizisi char
        ** _arguments.

        SimpleCommand().
        void insertArgument( char * argument ).
};

// Varsa çoklu borularla birlikte tam bir komutu açıklar
// ve varsa giriş/çıkış yönlendirmesi. struct
Command {
        int _numberOfAvailableSimpleCommands;
        int _numberOfSimpleCommands;
        SimpleCommand ** _simpleCommands;
        char * _outFile.
        char * _inputFile;
        char * _errFile;
        int _background.

        void prompt();
        void print();
        void execute();
        void clear().

        Command().
        void insertSimpleCommand( SimpleCommand * simpleCommand ).

        static Command _currentCommand.
        static SimpleCommand *_currentSimpleCommand.
};
```

SimpleCommand::SimpleCommand yapıcısı basit ve boş bir komut oluşturur. SimpleCommand**::insertArgument(char * argument)** yöntemi SimpleCommand'a yeni bir argüman ekler ve gerekirse _arguments dizisini genişletir. Ayrıca exec() sistem çağrısı için gerekli olduğundan son elemanın NULL olduğundan emin olur.

Command::Command() yapıcısı, Command::**`insertSimpleCommand( SimpleCommand * simpleCommand)`** yöntemiyle doldurulacak boş bir komut oluşturur. insertSimpleCommand, gerekirse _simpleCommands dizisini de genişletir. Yönlendirme yapılmadıysa _outFile, _inputFile, _errFile değişkenleri NULL veya yönlendirildikleri dosyanın adı olacaktır.

`_currentCommand` ve **`_currentCommand`** değişkenleri statik değişkenlerdir, yani tüm sınıf için yalnızca bir tane vardır. Bu değişkenler, komutun ayrıştırılması sırasında Command ve Simple komutunu oluşturmak için kullanılır.

Command ve SimpleCommand sınıfları kabukta kullanacağımız ana veri yapısını gerçekleştirir.

Sözcüksel Çözümleyicinin Uygulanması

Sözcüksel çözümleyici girdiyi belirteçlere ayırır. Standart girdiden karakterleri tek tek okuyacak ve ayrıştırıcıya aktarılacak bir belirteç oluşturacaktır. Sözcüksel çözümleyici, belirteçlerin her birini tanımlayan düzenli ifadeler içeren bir ***shell.l*** dosyası kullanır. Sözcük çözümleyici girdiyi karakter karakter okuyacak ve girdiyi düzenli ifadelerin her biriyle eşleştirmeye çalışacaktır. Girdideki bir dize düzenli ifadelerden biriyle eşleştiğinde, düzenli ifadenin sağındaki {...} kodunu çalıştıracaktır. Aşağıda shell.l'nin kabuğunuzun kullanacağı basitleştirilmiş bir versiyonu verilmiştir:

```
/*
 * shell.l: kabuk için basit sözlüksel analizör.
 */

%{

#include <string.h>
#include "y.tab.h"

%}

%%

\n      {
                    NEWLINE'ı
                    döndür.
```

```
        }

[ \t]   {
                /* BoŞluklarI ve sekmeleri atIn */
        }

">"     {
                BÜYÜK dönüş.
        }

"<"     {
                DAHA AZ döndür.
        }

">>"     {
                return GREATGREAT.
        }

">&"     {
                GREATAMPERSAND'a geri dönün.
        }

"|"    {
                dönüş PIPE.
        }

"&"    {
                AMPERSAND'ı döndürün.
        }


[^ \t\n][^ \t\n]* {
                /* Dosya adlarında yalnızca alfa karakterleri olduğunu
                varsayın */ yylval.string_val = strdup(yytext).
                WORD'ü döndür.
        }

/* Buraya daha fazla belirteÇ ekleyin */

.       {
                /* Girdide geçersiz karakter */
                return NOTOKEN.
        }

%%
```

shell.l dosyası ***lex.yy.c adında*** bir C dosyası oluşturmak için ***lex'ten*** geçirilir. Bu dosya, ayrıştırıcının karakterleri belirteçlere çevirmek için kullanacağı tarayıcıyı uygular.

İşte lex'i çalıştırmak için kullanılan komut.

```
bash% lex shell.l
bash% ls
lex.yy.c
```

lex***.yy.c*** dosyası, aşağıdaki paragrafta açıklanan belirteçleri ayırmak için lexer'ı uygulayan bir C dosyasıdır
Shell.l.

Shell.l'de iki kısım vardır. Üst kısım şuna benzer:

```
%{
#include <string.h>
#include "y.tab.h"
%}
```

Bu, tarayıcıda kullanacağınız başlık dosyalarını ve değişken tanımlarını içeren ***lex.yy.c*** dosyasının en üstüne değişiklik yapılmadan doğrudan eklenecek bir kısımdır. Lexer'ınızda kullanacağınız değişkenleri burada bildirebilirsiniz.

ile sınırlandırılmış ikinci kısım şu şekilde görünür:

```
%%
\n      {
            NEWLINE'ı döndür.
        }
  [ \t] {
            /* Boşlukları ve sekmeleri atın */
        }
">"     {
            BÜYÜK dönüş.
        }
[^ \t\n][^ \t\n]* {
            /* Dosya adlarında yalnızca alfa karakterleri olduğunu
            varsayın */ yylval.string_val = strdup(yytext).
            WORD'ü döndür.
        }
%%
```

Bu kısım, standart girdiden karakterler alınarak oluşturulan belirteçleri tanımlayan düzenli ifadeleri içerir. Bir belirteç oluşturulduktan sonra geri döndürülür veya bazı durumlarda atılır. Bir belirteci tanımlayan her kuralın da iki bölümü vardır:

düzenli ifade {
eylem
}

Örneğin.

\n {
NEWLINE'ı döndür.
}

İlk kısım, eşleşmesini beklediğimiz belirteci tanımlayan düzenli bir ifadedir. Eylem, programcının eklediği ve belirteç düzenli ifadeyle eşleştiğinde çalıştırılan bir C kodu parçasıdır. Yukarıdaki örnekte, newline karakteri bulunduğunda, lex ***NEWLINE*** sabitini döndürecektir. ***NEWLINE sabitlerinin*** nerede tanımlandığını daha sonra açıklayacağız.

İşte bir ***WORD'ü*** tanımlayan daha karmaşık bir belirteç. Bir ***WORD, bir*** komut için bir argüman veya komutun kendisi olabilir.

```
[^ \t\n][^ \t\n]* {
                /* Dosya adlarında yalnızca alfa karakterleri
                olduğunu varsayın */ yylval.string_val =
                strdup(yytext).
                WORD'ü döndür.
}
```

...] içindeki ifade, parantezlerin içindeki herhangi bir karakterle eşleşir. ***^...]*** ifadesi, parantezlerin içinde olmayan herhangi bir karakterle eşleşir. Bu nedenle, ***[^ \t\n][^ \t\n]**** boşluk, sekme ve ya satırsonu olmayan bir karakterle başlayan ve boşluk, sekme veya satırsonu olmayan sıfır veya daha fazla karakter tarafından takip edilen bir belirteci tanımlar. Eşleştirilen ***belirteç yytext*** adlı bir değişkendedir. Bir sözcük eşleştirildiğinde, eşleştirilen belirtecin bir kopyası aşağıdaki deyimde ***yylval.string_val öğesine*** atanır:

yylval.string_val = strdup(yytext).

belirtecin değeri ayrıştırıcıya bu şekilde aktarılır. Son olarak, sabit ***WORD ayrıştırıcıya*** döndürülür.

shell.l dosyasına yeni belirteçler ekleme

Yukarıda açıklanan shell***.l*** şu anda daha az sayıda belirteci desteklemektedir. Kabuğunuzu geliştirmenin ilk adımı olarak, yeni dilbilgisine şu anda shell***.l'***de olmayan daha fazla belirteç eklemeniz gerekecektir. Hangi belirteçlerin eksik olduğunu ve shell***.l'***ye eklenmesi gerektiğini görmek için ***Şekil 4'teki gramere*** bakın:

```
">>" { return GREATGREAT; }
"|" { return PIPE;}
"&" { return AMPERSAND}
Vb.
```

Yeni belirteçleri shell.y dosyasına ekleme

Bir önceki adımda oluşturduğunuz token isimlerini ***shell.y içerisine*** %token kısmına ekleyeceksiniz:

```
%token NOTOKEN, GREAT, NEWLINE, WORD, GREATGREAT, PIPE,
AMPERSAND vb.
```

Dilbilgisinin tamamlanması

Kabuğun gramerini tamamlamak için shell.y dosyasına daha fazla kural eklemeniz gerekir. Aşağıdaki şekil, kabuğun sözdizimini grameri oluşturmak için kullanılacak farklı parçalara ayırmaktadır.

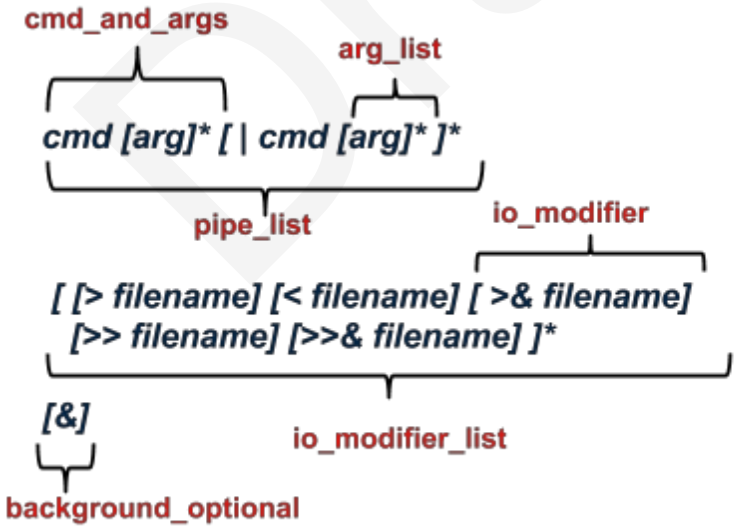


Figure 3. Shell Grammar labeld with the different parts.

İşte yukarıda tanımlanan etiketleme kullanılarak oluşturulan gramer:

```
hedef: command_list.
```

```
arg_listesi:
        arg_list WORD
        | /*boş*/
                ;
 cmd_and_args:
        WORD arg_list
                ;
pipe_list:
        pipe_list PIPE cmd_and_args
        | cmd_and_args
        ;

io_modifier:
        HARİKA Kelime
         | BÜYÜK Kelime
         | BÜYÜK KAMPANYALAR VE Kelime
         | BÜYÜKAMPERSAND Kelime
         | LESS Word
        ;
 io_modifier_list:
         io_modifier_list io_modifier
                | /*boş*/
                ;
background_optional:
                AMPERSAND
                | /*boş*/
                ;
command_line:
         pipe_list io_modifier_list background_opt NEWLINE
        | NEWLINE /*boş cmd satırını kabul et*/
        | hata NEWLINE{yyerrok;}
                 /*hata kurtarma*/

command_list :
                 command_list command_line
                ;/* komut döngüsü*/
```

Yukarıdaki gramer, komut döngüsünü gramerin kendi içinde uygular.

Hata belirteci, hata kurtarma için kullanılan özel bir belirteçtir. error, <NEWLINE> gibi bilinen bir belirteç bulunana kadar tüm belirteçleri ayrıştıracaktır. ***yerrok,*** ayrıştırıcıya hatanın kurtarıldığını söyler.

Ayrıştırıcı, sözcüksel çözümleyici tarafından oluşturulan belirteçleri alır ve bunların ***shell.y'***deki dilbilgisi kuralları tarafından tanımlanan sözdizimine uyup uymadığını kontrol eder. Girdi komut satırının sözdizimine uyup uymadığını kontrol ederken, ayrıştırıcı, dilbilgisi kuralları arasına ekleyeceğiniz eylemleri veya C kodu parçalarını yürütür. Bu kod parçaları eylem olarak adlandırılır ve küme parantezleri { action; } ile sınırlandırılır.

Komut tablosunu doldurmak için dilbilgisine {...} eylemlerini eklemeniz

gerekir. Örnek:

```
arg_listesi:
    arg_list WORD { currSimpleCmd>insertArg($2); }
    | /*boş*/
       ;
```

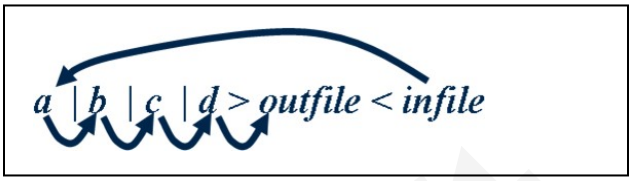
Kabuğunuzda Süreçler Oluşturma

Boru hattındaki her komut için yeni bir süreç oluşturarak ve ana sürecin son komutu beklemesini sağlayarak başlayın. Bu, "ls al" gibi basit komutların çalıştırılmasına izin verecektir.

```
Command::execute()
{
  int ret.
  for ( int i = 0; i < _numberOfSimpleCommands; i++ ) {
     ret = fork().
     eğer (ret == 0) {
       //Çocuk
       execvp(sCom[i]>_args[0], sCom[i]>_args);
       perror("execvp").
       _exit(1).
     }
     else if (ret < 0) {
       perror("fork");
       return.
     }
     // Ana kabuk devam
  } // iÇin
  if (!background) {
     // son işlemi bekle waitpid(ret,
     NULL).
  }
}// execute
```

Kabuğunuzda Boru ve Giriş/Çıkış Yönlendirme

Kabuğunuz için strateji, süreçleri çatallandırmadan önce tüm boru tesisatını ve yeniden yönlendirmeyi ana sürecin yapmasıdır. Bu şekilde çocuklar yeniden yönlendirmeyi devralacaktır. Ebeveynin girdi/çıktıyı kaydetmesi ve sonunda geri yüklemesi gerekir. Stderr tüm süreçler için aynıdır



Bu şekilde ***a*** işlemi çıkışını ***boru 1'e*** gönderir. Sonra ***b*** girdisini boru ***1'***den okur ve çıktısını ***boru 2'***ye gönderir ve bu böyle devam eder. Son komut ***d*** girdisini ***boru 3***'ten okur ve çıktısını ***outfile'a gönderir***. a'nın girdisi ***infile'dan*** gelir.

Aşağıdaki kod bu yönlendirmenin nasıl uygulanacağını göstermektedir. Basitlik için bazı hata kontrolleri elenmiştir.

```
 1 void Command::execute(){
 2     //giriş/çıkış kaydet
 3     int tmpin=dup(0).
 4     int tmpout=dup(1).
 5 
 6     //başlangıç girdisini ayarla
 7     int fdin.
 8     if (infile) {
 9       fdin = open(infile,O_READ).
10     }
11     else {
12       // Varsayılan girişi kullan
13       fdin=dup(tmpin).
14     }
15 
16     int ret.
17     int fdout.
18     for(i=0;i<numsimplecommands; i++)   {
19       //yönlendirme girişi
20       dup2(fdin, 0).
21       close(fdin).
22       //kurulum çıktısı
23       if (i == numsimplecommands1){
24         // Son basit komut
25         if(outfile){
```

```
26            fdout=open(outfile,¦â€¦).
27          }
28          else {
29            // Varsayılan çıktıyı
30            kullan fdout=dup(tmpout).
31          }
32        }
33
34         else {
35            // Son değil
36            //basit komut
37            //create pipe
38            int fdpipe[2];
39            pipe(fdpipe);
40            fdout=fdpipe[1];
41            fdin=fdpipe[0].
42         }// if/else
43
44         // Çıktıyı yeniden
45         yönlendir dup2(fdout,1);
46         close(fdout).
47
48         // Çocuk süreç oluştur
49         ret=fork().
50         if(ret==0) {
51           execvp(scmd[i].args[0], scmd[i].args);
52           perror(â€œexecvpâ€).
53           _exit(1).
54         }
55       } // için
56
57       //giriş/çıkış
58       varsayılanlarını geri
59       yükle dup2(tmpin,0);
60       dup2(tmpout,1);
61       close(tmpin);
62       close(tmpout).
63
64       if (!background) {
65         // Son komut için bekle
66         waitpid(ret, NULL).
67       }
68
     } // yürütmek
```

execute() yöntemi kabuğun belkemiğidir. Basit komutları her komut için ayrı bir süreçte çalıştırır ve yeniden yönlendirmeyi gerçekleştirir.

Satır 3 ve 4, dup() fonksiyonunu kullanarak mevcut stdin ve stdout'u iki yeni dosya tanımlayıcısına kaydeder. Bu, execute() işlevinin sonunda stdin ve stdout'un eski haline dönmesini sağlayacaktır

execute() işlevinin başlangıcında. Bunun nedeni, stdin ve stdout'un (dosya tanımlayıcıları 0 ve 1) basit komutların yürütülmesi sırasında ebeveynde değiştirilecek olmasıdır.

```
 3    int tmpin=dup(0);
 4    int tmpout=dup(1).
```

6'dan 14'e kadar olan satırlar, "command < infile" biçimindeki komut tablosunda girdi yönlendirme dosyası olup olmadığını kontrol eder. Eğer girdi yönlendirmesi varsa, dosyayı ***infile*** içinde açacak ve ***fdin içine*** kaydedecektir. Aksi takdirde, girdi yönlendirmesi yoksa, varsayılan girdiye atıfta bulunan bir dosya tanımlayıcısı oluşturacaktır. Bu talimat bloğunun sonunda ***fdin,*** komut satırının girdisini içeren ve ana kabuk programını etkilemeden kapatılabilen bir dosya tanımlayıcısı olacaktır.

```
 6    //başlangıç girdisini
 7    ayarla int fdin.
 8    if (infile) {
 9      fdin = open(infile,O_READ).
10    }
11    else {
12      // Varsayılan girişi kullan
13      fdin=dup(tmpin).
14    }
```

Satır 18, komut tablosundaki tüm basit komutları yineleyen ***for döngüsüdür.*** Bu ***for döngüsü*** her basit komut için bir işlem oluşturacak ve boru bağlantılarını gerçekleştirecektir.

20. satır standart girdiyi ***fdin'***den gelecek şekilde yeniden yönlendirir. Bundan sonra stdin'den herhangi bir okuma ***fdin*** tarafından işaret edilen dosyadan gelecektir. İlk yinelemede, ilk basit komutun girdisi ***fdin'd***en gelecektir. ***fdin*** döngünün ilerleyen kısımlarında bir girdi borusuna yeniden atanacaktır. Satır 21, dosya tanımlayıcısına artık ihtiyaç duyulmayacağı için ***fdin'i*** kapatacaktır. Genel olarak, her işlem için yalnızca birkaç tane mevcut olduğundan (normalde varsayılan olarak 256), dosya tanımlayıcılarını ihtiyaç duyulmadıkları sürece kapatmak iyi bir uygulamadır.

```
16    int ret;
17    int fdout.
18    for(i=0;i<numsimplecommands; i++) {
19      //girişi yönlendir
20      dup2(fdin, 0);
21      close(fdin).
```

Satır 23, bu yinelemenin son basit komuta karşılık gelip gelmediğini kontrol eder. Eğer durum buysa, 25. satırda "command > outfile" şeklinde bir çıktı dosyası yönlendirmesi olup olmadığını test edecek ve ***outfile dosyasını*** açıp ***fdout***'a atayacaktır. Aksi takdirde, 30. satırda varsayılan girdiyi işaret eden yeni bir dosya tanımlayıcı oluşturacaktır. 23'ten 32'ye kadar olan satırlar, ***fdout'un*** son yinelemedeki çıktı için bir dosya tanımlayıcısı olduğundan emin olur.

```
23      //kurulum Çıktısı
23      if (i == numsimplecommands1){
24        // Son basit komut
25        if(outfile){
26          fdout=open(outfile,¦â€¦).
27        }
28        else {
29          // Varsayılan çıktıyı
30          kullan fdout=dup(tmpout).
31        }
32      }
33
34      else {...
```

34'ten 42'ye kadar olan satırlar sonuncusu olmayan basit komutlar için çalıştırılır. Bu basit komutlar için çıktı bir dosya değil bir boru olacaktır. 38. ve 39. satırlar yeni bir boru oluşturur. Yeni boru. Bir boru, bir tampon aracılığıyla iletilen bir çift dosya tanımlayıcısıdır. Dosya tanımlayıcısı fdpipe[1]'e yazılan her şey fdpipe[0]'dan okunabilir. 41 ve 42. satırlarda fdpipe[1] fdout'a ve fdpipe[0] fdin'e atanmıştır.

Satır 41 `fdin=fdpipe[0], bir` sonraki yinelemedeki bir sonraki basit komutun girdi fdin'inin mevcut basit komutun fdpipe[0]'ından gelmesini sağladığı için boruların uygulanmasının çekirdeği olabilir.

```
34       else {
35          // Son değil
36          //basit komut
37          //create pipe
38          int fdpipe[2];
39          pipe(fdpipe);
40          fdout=fdpipe[1];
41          fdin=fdpipe[0].
42       }// if/else
43
```

45. satır stdout'u fdout tarafından işaret edilen dosya nesnesine gitmesi için yönlendirir. Bu satırdan sonra, stdin ve stdout ya bir dosyaya ya da bir boruya yönlendirilmiştir. 46. satır artık ihtiyaç duyulmayan fdout'u kapatır.

```
44        // Çıktıyı yeniden
45        yönlendir
46        dup2(fdout,1);
          close(fdout).
```

Kabuk programı 48. satırdayken, geçerli basit komut için girdi ve çıktı yönlendirmeleri zaten ayarlanmıştır. Satır 49, stdin, stdout ve stderr'e karşılık gelen ve terminale, bir dosyaya ya da bir boruya yönlendirilen 0,1 ve 2 dosya tanımlayıcılarını devralacak yeni bir çocuk süreç oluşturur.

Süreç oluşturmada bir hata yoksa, 51. satır bu basit komut için çalıştırılabilir dosyayı yükleyen execvp() sistem çağrısını çağırır. Eğer execvp başarılı olursa geri dönmeyecektir. Bunun nedeni, mevcut sürece yeni bir çalıştırılabilir imaj yüklenmiş ve belleğin üzerine yazılmış olmasıdır, dolayısıyla geri dönülecek bir şey yoktur.

```
48        // Çocuk süreç oluştur
49        ret=fork().
50        if(ret==0) {
51          execvp(scmd[i].args[0], scmd[i].args);
52          perror("execvp").
53          _exit(1).
54        }
55      } // iÇin
```

55. satır, tüm basit komutları yineleyen for döngüsünün sonudur.

For döngüsü yürütüldükten sonra, tüm basit komutlar kendi süreçlerinde çalışır ve borular kullanarak iletişim kurarlar. Ana sürecin stdin ve stdout'u yeniden yönlendirme sırasında değiştirildiğinden, 58. ve 59. satırlar dup2'yi çağırarak stdin ve stdout'u tmpin ve tmpout'a kaydedilen aynı dosya nesnesine geri yükler. Aksi takdirde, kabuk girdiyi girdinin yönlendirildiği son dosyadan alır. Son olarak, 60 ve 61. satırlar ana kabuk sürecinin stdin ve stdout'unu kaydetmek için kullanılan geçici dosya tanımlayıcılarını kapatır.

```
57      //giriş/çıkış
58      varsayılanlarını geri
59      yükle dup2(tmpin,0);
60      dup2(tmpout,1);
61      close(tmpin);
        close(tmpout).
```

Komut satırında "&" arka plan karakteri ayarlanmamışsa, kabuk üst sürecinin kabuk istemini yazdırmadan önce komuttaki son alt sürecin bitmesini beklemesi gerektiği anlamına gelir. Eğer "&" arka plan karakteri ayarlanmışsa, bu komut satırının

kabukla eşzamansız olarak çalışır, böylece ana kabuk işlemi komutun bitmesini beklemez ve komut istemini hemen yazdırır. Bundan sonra komutun yürütülmesi tamamlanır.

```
63      if (!background) {
64        // Son komut için
          bekleyin
65        waitpid(ret, NULL).
66      }
67
68  } // yürütmek
```

Yukarıdaki örnek standart hata yönlendirmesi yapmaz(dosya tanımlayıcı 2). Bu kabuğun semantiği, tüm basit komutların stderr'yi aynı yere göndereceği şeklinde olmalıdır. Yukarıda verilen örnek stderr yönlendirmesini destekleyecek şekilde değiştirilebilir.

Yerleşik Fonksiyonlar

printenv dışındaki tüm yerleşik işlevler üst süreç tarafından yürütülür. Bunun nedeni, setenv, cd vb. işlevlerin ebeveynin durumunu değiştirmesini istememizdir. Eğer bunlar alt süreç tarafından çalıştırılırsa, alt süreç çıktığında değişiklikler kaybolacaktır. Bu yerleşik işlevler için, yeni bir süreci çatallamak yerine işlevi execute içinde çağırın.

Kabukta Joker Karakterlerin Uygulanması

Hiçbir kabuk joker karakterler olmadan tamamlanmış sayılmaz. Joker karakterler, joker karakterle eşleşen birden fazla dosya üzerinde tek bir komutun gerçekleştirilmesini sağlayan bir kabuk özelliğidir.

Bir joker karakter, joker karakterle eşleşen dosya adlarını tanımlar. Joker karakter, geçerli dizindeki veya joker karakterde tanımlanan dizindeki tüm dosyalar üzerinde yineleme yaparak ve ardından joker karakterle eşleşen dosya adlarını komuta argüman olarak vererek çalışır.

Genel olarak "*" karakteri herhangi bir türden 0 veya daha fazla karakterle eşleşir. "?" karakteri herhangi bir türden bir karakterle eşleşir.

Bir joker karakteri uygulamak için, öncelikle joker karakteri bir düzenli ifade kütüphanesinin değerlendirebileceği bir düzenli ifadeye çevirmelisiniz.

İlk olarak geçerli dizindeki joker karakterleri genişlettiğiniz basit durumu uygulamanızı öneririz. Argümanların tabloya eklendiği shell.y'de genişletme işlemini yapın.

Shell.Y:

Daha önce:

```
argüman: WORD { Command::_currentSimpleCommand>insertArgument($1).
} ;
```

Sonra:

```
argüman: WORD {
  expandWildcardsIfNecessary($1).
} ;
```

expandWildcardsIfNecessary() fonksiyonu daha sonra verilmiştir. 4 ila 7. satırlar arg argümanında "*" veya "?" yoksa argümanı ekleyecek ve hemen geri dönecektir. Ancak, bu karakterler varsa, joker karakteri düzenli bir ifadeye çevirir.

```
1  void expandWildcardsIfNecessary(char * arg)
2  {
3    // arg "*" veya "?" içermiyorsa döndür
4    if (arg ne "*" ne de "?" içeriyorsa (strchr kullanın) ) {
5          Command::_currentSimpleCommand>insertArgument(arg);
6          return.
7    }
8
9    // 1. Joker karakteri düzenli ifadeye dönüştürün
10   // Convert "*" > ".*"
11   // "?" > "."
12   // "." > "\." ve ihtiyacınız olan diğerleri
13   // Ayrıca eşleştirmek için başına ^ ve sonuna $ ekleyin
14   // kelimenin başı ve sonu.
15   // Düzenli ifade için yeterli alan ayırın char *
16   reg = (char*)malloc(2*strlen(arg)+10); char * a
17   = arg.
18   char * r = reg.
19   *r = '^'; r++; // satır başıyla eşleşirken
20   while (*a) {
21         if (*a == '*') { *r='.'; r++; *r='*'; r++; }
22         else if (*a == '?') { *r='.' r++;}
23         else if (*a == '.') { *r='\\'; r++; *r='.'; r++;}
24         else { *r=*a; r++;}
25         a++;
26   }
27   *r='$'; r++; *r=0;// satır sonuyla eşleştir ve boş
28   karakter ekle
29   // 2. Düzenli ifadeyi derleyin. Bkz. lab3src/regular.cc
30   char * expbuf = regcomp( reg, â€¦ ).
31   if (expbuf==NULL) {
32         perror("regcomp");
           return.
```

```
33      }
34      // 3. Dizini listeleyin ve girişleri argüman olarak
        ekleyin
35      // düzenli ifade ile eşleşen
36      DIR * dir = opendir(".").
37      if (dir == NULL) {
38            perror("oopendir").
39            Geri dön.
40      }
41      struct dirent * ent.
42      while ( (ent = readdir(dir))!= NULL) {
43            // İsmin eşleşip eşleşmediğini kontrol et
44             if (regexec(ent>d_name, re ) ==0 ) {
45            // Argüman ekle
46            Command::_currentSimpleCommand>
47            insertArgument(strdup(ent>d_name)).
48            }
49      }
50      closedir(dir).
51  }
52
```

Bir joker karakterden düzenli ifadeye yapılacak temel çeviriler aşağıdaki tabloda yer almaktadır.

Joker Karakter	***Düzenli İfade***
""*	*".*"*
"?"	*"."*
"."	*"\."*
Joker karakterin başlangıcı	"^"
Joker Kartın Sonu	"$"

16. satırda düzenli ifade için yeterli bellek ayrılır. 19. satır. Tüm dosya adının eşleşmesini zorlamak istediğimizden, düzenli ifadenin başlangıcını dosya adının başlangıcıyla eşleştirmek için "^" ekleyin. Satır 20 ila 26, yukarıdaki tabloda yer alan joker karakterleri düzenli ifadenin karşılık gelen eşdeğerlerine dönüştürür. Satır 27, düzenli ifadenin sonunu dosya adının sonuyla eşleştiren "$" ekler.

29'dan 33'e kadar olan satırlar düzenli ifadeyi değerlendirilebilecek daha verimli bir temsile derler ve *expbuf* içinde saklar. 41. satır geçerli dizini açar ve 42 ila 48. satırlar

Geçerli dizindeki tüm dosya adları üzerinde yineleme yapar. Satır 44, dosya adının düzenli ifadeyle eşleşip eşleşmediğini kontrol eder ve doğruysa, dosya adının bir kopyası argümanlar listesine eklenir. Tüm bunlar, düzenli ifadeyle eşleşen dosya adlarını bağımsız değişkenler listesine ekleyecektir.

Dizin Girişlerini Sıralama

Bash gibi kabuklar bir joker karakterle eşleşen girdileri sıralar. Örneğin "echo *" geçerli dizindeki tüm girdileri sıralı olarak listeleyecektir. Aynı davranışı elde etmek için joker karakter eşleştirmesini aşağıdaki gibi değiştirmeniz gerekecektir:

Satır 5, joker karakterle eşleşen dosya adlarını tutacak geçici bir dizi oluşturur. Dizinin başlangıç boyutu maxentries=20'dir. Satır 7'deki while döngüsü tüm dizin girişlerini yineler. Eğer eşleşirlerse onları geçici diziye ekleyecektir. Girdi sayısı maksimum sınıra ulaştıysa, satır 10 ila 14 dizinin boyutunu iki katına çıkarır. 20. satır, seçtiğiniz sıralama işlevini kullanarak girdileri sıralar. Son olarak, 23 ila 26. satırlar dizideki sıralanmış girişler üzerinde yineleme yapar ve bunları sıralanmış sırada argüman olarak ekler.

```
 1
 2  struct dirent * ent;
 3  int maxEntries = 20;
 4  int nEntries = 0.
 5  char ** array = (char**) malloc(maxEntries*sizeof(char*)).
 6
 7  while ( (ent = readdir(dir))!= NULL) {
 8    // İsmin eşleşip eşleşmediğini kontrol et
 9    if (regexec(ent>d_name, expbuf) ) {
10          if (nEntries == maxEntries) {
11             maxEntries *=2.
12             dizi = realloc(dizi, maxEntries*sizeof(char*));
13             assert(dizi!=NULL).
14          }
15          array[nEntries]= strdup(ent>d_name);
16          nEntries++.
17    }
18  }
19  closedir(dir).
20  sortArrayStrings(array, nEntries); // Herhangi bir sıralama
21  işlevi kullanın
22
23  // Argüman ekle
24  for (int i = 0; i < nEntries; i++) {
25    Command::_currentSimpleCommand>
26    insertArgument(array[i])).
27  }
28
    free(dizi).
```

Joker Karakterler ve Gizli Dosyalar

Bash gibi kabukların bir başka özelliği de joker karakterlerin varsayılan olarak "." karakteriyle başlayan gizli dosyalarla eşleşmemesidir. UNIX'te gizli dosyalar .login, .bashrc vb. gibi "." ile başlar.
"." ile başlayan dosyalar bir joker karakter ile eşleştirilmemelidir. Örneğin "echo *" "." ve "... "yi göstermeyecektir.

Bunu yapmak için, kabuk "." ile başlayan bir dosya adını yalnızca j o k e r k a r a k t e r i n başında da bir "." varsa ekleyecektir. Bunu yapmak için, match if deyiminin aşağıdaki şekilde değiştirilmesi gerekir:. Dosya adı joker karakterle eşleşiyorsa, yalnızca dosya adı '.' ile başlıyorsa ve joker karakter '.' ile başlıyorsa dosya adını bağımsız değişken olarak ekleyin. Aksi takdirde, dosya adı "." ile başlamıyorsa, dosya adını argüman listesine ekleyin.

```
if (regexec (...) ) {
   if (ent>d_name[0] == '.') {
      if (arg[0] == '.')
         dosya adını argümanlara ekleyin.
      }
    }
    else {
       argümanlara ent>d_name ekleyin
    }
}
```

Alt Dizin Joker Karakterleri

Joker karakterler bir yol içindeki dizinlerle de eşleşebilir:

Örneğin, "echo /p*/*a/b*/aa*" yalnızca dosya adlarıyla değil, yoldaki alt dizinlerle de eşleşecektir.

Alt dizinleri eşleştirmek için bileşen bileşen eşleştirmeniz gerekir

Joker Karakter Stratejisini aşağıdaki şekilde uygulayabilirsiniz.

expandWildcard(prefix, suffix) işlevini yazın, burada

prefix Zaten genişletilmiş olan yol. `Joker karakter` içermemelidir. suffix - Yolun henüz genişletilmemiş kalan kısmı. Joker karakter içerebilir.

Sonek boş olduğunda önek bir argüman olarak eklenecektir. expandWildcard(prefix, suffix) ilk olarak boş bir önek ve sonekteki joker karakter ile çağrılır. expandWildcard, yolda eşleşen öğeler için özyinelemeli olarak çağrılacaktır.