POGLAVLJE 31

BOURNE SHELL

prof. dr Samir Ribić, dipl.ing.el., Dejan Popović, Ensar Kapur

31.1 Osnovne osobine jezika

Bourn Shell program koji radi na operativnim sistemima Unix® i LINUX® je skriptni programski jezik i pruža interfejs za izvršavanje programa na sistemu. Bourn Shell je interaktivni tumač naredbi i programski jezik naredbi. Često se naziva interfejs naredbenog retka ili interpreter naredbi, jer korisniku ne pruža nikakav grafičko-korisnički interfejs. Shell je naredba koja čita linije iz datoteke ili terminala, interpretira ih i općenito izvršava druge naredbe. To je program koji se pokreće kada se korisnik prijavi u sistem. Ljuska implementira jezik koji ima konstrukcije kontrole toka, makro mogućnost koja pruža niz funkcija pored skladištenja podataka, zajedno sa ugrađenom historijom i mogućnosti uređivanja linija. Sintaksa mu podsjeća na ALGOL 68. Veliki broj Unix aplikacija je razvijen u ovom jeziku. Odmah nakon što je Unix izumljen, Stephen Bourne se dao na zadatak i smislio ono što je nazvao ljuskom: mali kompajler u hodu koji je mogao preuzeti jednu po jednu naredbu, prevesti je u slijed razumljivih bitova i dati izvršiti tu naredbu. Ovu vrstu programa sada nazivamo interpretatorom, ali u to je vrijeme izraz "ljuska" bio mnogo češći (budući da je to bila ljuska iznad temeljnog sistema za korisnika). Stephenova ljuska bila je tanka, brza i premda s vremena na vrijeme pomalo nezgrapna, njezinoj snazi i danas zavide mnoga trenutna sučelja naredbenog retka operativnog sustava. Budući da ju je dizajnirao Stephen Bourne, ova školjka se zove Bourne Shell. Izvršni se jednostavno zove sh i korištenje ove ljuske u skriptiranju je još uvijek toliko sveprisutno da ne postoji sistem baziran na Unixu koji ne nudi ljusku čiju se izvršnu datoteku može pristupiti pod imenom sh.

31.1.1 Vrsta i namjena

Bourn Shell omogućuje pisanje i izvršavanje skripti ljuske, koje pružaju osnovni tok kontrole programa, kontrolu nad deskriptorima ulazno/izlaznih (I/O) datoteka i sve ključne značajke potrebne za stvaranje skripti ili strukturiranih programa za ljusku. Skriptni jezici se koriste najčešće za pisanje malih programa (skripti) koji se brzo pišu i služe za obavljanje malih poslova. Oni se koriste, jer je razvoj programa znatno jednostavniji i nije potrebno prevoditi skripte u mašinski jezik. Bourneova ljuska nije bila samo važan korak naprijed nego i sidro za brojne izvedenice, od kojih se mnoge danas koriste u tipičnim Linux sustavima. Iako se danas koristi kao interaktivni interpreter naredbi, izvorno je zamišljen kao skriptni jezik i sadrži niz funkcija za koje se obično pretpostavlja da proizvode strukturirane programe. Bourne shell je poznat

po tome što je uveo mnogo suštinskih ideja, kao što je, na primer, izlazni status izvršenih komandi, koji je praktino omoguio pisanje shell script programa.

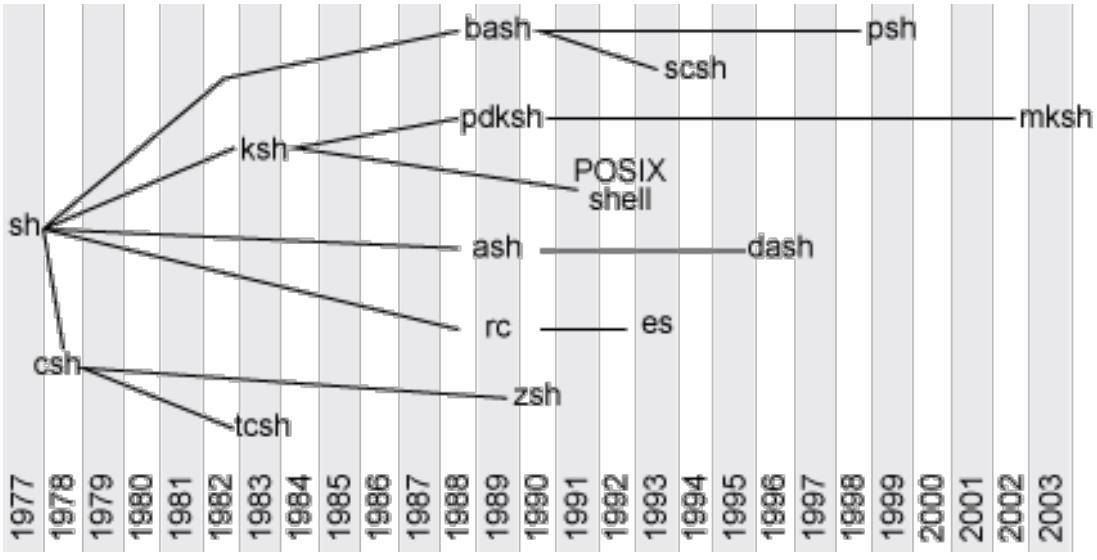
Neki od primarnih ciljeva školjke bili su:

• Dopustiti da se skripte ljuske koriste kao filteri
• Pružiti programabilnosti uključujući kontrolni tijek i varijable
• Kontrola nad svim deskriptorima ulaznih/izlaznih datoteka
• Kontrola nad upravljanjem signalom unutar skripti
• Nema ograničenja u duljini nizova pri tumačenju skripti ljuske
• Racionalizirajte i generalizirajte mehanizam navođenja stringova
• Mehanizam okruženja. To je omogućilo uspostavljanje konteksta pri pokretanju i omogućilo je način da skripte ljuske proslijede kontekst podskriptama (procesima) bez potrebe za korištenjem eksplicitnih pozicijskih parametara

31.1.2 Historijat

Nezavisnost ljuske od UNIX operativnog sistema sama po sebi dovela je do razvoja desetaka ljuski tokom historije UNIX-a—iako je samo nekoliko njih postiglo široku upotrebu. Prva velika ljuska (školjka) bila je Bourn shall, nazvana po svom kreatoru, Stevenu Bourneu, a razvijena u Bell Labs kompaniji, 1979. godine. Razvijena je kao zamjena za Thompson shell. Uključena je u prvu popularnu verziju UNIX-a, verziju 7. Na stil kodiranja Stephena Bournea utjecalo je njegovo iskustvo s kompajlerom ALGOL 68C na kojem je radio na Cambridge Univerzitetu. Osim stila u kojem je program napisan, Bourne je ponovno koristio dijelove ALGOL-a 68. Počevši od 1979. Bourn shell je u sistemu poznat kao sh. Iako je UNIX prošao kroz mnoge promjene, Bourn shell je još uvijek popularna i u biti nepromijenjena. Iako se koristi kao interaktivni tumač komandi, također je zamišljen kao skriptni jezik i sadrži većinu karakteristika za koje se obično smatra da proizvode strukturirane programe.

Bourn shell je doveo do razvoja KornShell (ksh), Almquist školjke (pepeo) i popularne Bourne-again shell (ili Bash).). C ljuska (csh) bila je u razvoju u vrijeme kada je Bourneova ljuska objavljena. Popularnost je stekao objavljivanjem The Unix Programming Environment Briana Kernighana i Roba Pikea—prve komercijalno objavljene knjige koja je predstavila Bourn shell kao programski jezik u obliku tutorijala.



Slika 31.1: Historijski razvoj Shell-a

31.1.3 Način implementacije

Skripte (programe) napravljene u Shell-u ne treba kompajlirati. Shell ih tumači redak po redak, jednu po jednu. Sekvencijalno. Čak i ako druga linija skripta ima grešku, shell interpreter će izvršiti prvu liniju. Zato su one poznate ili su nazvane Shells Scripts i mogu se kretati od jednostavnih naredbi do složenih serija uputstava za pokretanje samog OS-a. Imaju prilično čistu (očiglednu) sintaksu (konstrukcija, redoslijed), što ih čini dobrom polaznom tačkom za početak u svijetu programiranja.

Bourneova ljuska imala je dva primarna cilja: poslužiti kao tumač naredbi za interaktivno izvršavanje naredbi za operativni sistem i za skriptiranje (pisanje skripti za višekratnu upotrebu koje se mogu pozvati kroz ljusku). Osim što je zamijenila Thompsonovu školjku, Bourneova školjka je ponudila nekoliko prednosti u odnosu na svoje prethodnike. Bourne je uveo kontrolne tokove, petlje i varijable u skripte, pružajući funkcionalniji jezik za interakciju s operativnim sustavom (interaktivno i neinteraktivno). Shell je, također, dopuštao korištenje shell skripti kao filtera, pružajući integriranu podršku za rukovanje signalima, ali nije imala mogućnost definiranja funkcija. Konačno, uključio je niz značajki koje danas koristimo, uključujući zamjenu naredbi (koristeći povratne navodnike) i dokumente HERE za ugrađivanje sačuvanih literala niza unutar skripte. Bourneova ljuska nije bila samo važan korak naprijed nego i sidro za brojne izvedenice, od kojih se mnoge danas koriste u tipičnim Linux sustavima.

31.1.4 Format pisanja programa

Kako napisati shell skriptu u Linuxu?

1. Napraviti datoteku pomoću vi editora (ili bilo kojeg drugog uređivača).
   Datoteka skripte naziva s ekstenzijom . sh.
2. Započeti skriptu s #! /bin/sh.
3. Napisati neki kod.
4. Spremite datoteku skripte kao ime datoteke.sh.
5. Za izvršavanje skripte upisati bash ime datoteke.sh.

Prije svega, šta je školjka (shell)? Pod Unixom, ljuska (školjka) je tumač naredbi. Čita naredbe s tipkovnice i izvršava ih. Nadalje, naredbe se mogu staviti u datoteku i izvršiti ih sve odjednom. Ovo je poznato kao skripta. Evo jednog jednostavnog:

```
#!/bin/sh
# Rotate procmail log files
cd /homes/arensb/Mail
rm procmail.log.6       # This is redundant
mv procmail.log.5 procmail.log.6
mv procmail.log.4 procmail.log.5
mv procmail.log.3 procmail.log.4
mv procmail.log.2 procmail.log.3
mv procmail.log.1 procmail.log.2
mv procmail.log.0 procmail.log.1
mv procmail.log procmail.log.0
```

Ovdje treba napomenuti nekoliko stvari: prije svega, komentari počinju s hashom (#) i nastavljaju se do kraja retka (prvi redak je poseban). Drugo, sama skripta je samo niz naredbi.

Kada Unix pokuša izvršiti skriptu, prepoznaje prva dva znaka (#!) i zna da je to skripta. Zatim čita ostatak retka kako bi saznao koji program treba izvršiti skriptu. Shell zatim analizira ove tokene u naredbe i druge konstrukcije, uklanja posebno značenje određenih riječi ili znakova, proširuje druge, preusmjerava ulaz i izlaz prema potrebi, izvršava navedenu naredbu,

čeka izlazni status naredbe i taj izlazni status čini dostupnim radi daljnjeg pregleda ili obrade.

Shebang

Za Bourne shell skriptu /bin/sh prvi red skripte mora biti:

```
#!/bin/sh
```

Prilikom pisanja skripte, u prvu liniju moramo navesti koju ljusku ćemo koristiti. Ta početna oznaka mora biti u prvoj liniji skripte inače bi se znak '#' ponašao kao da želi označiti komentar. Ovo označavanje se još prepoznaje kao '**shebang**'. Potiče se pisanje komentara u skriptu kako bi imali više detalja čemu skripta služi te šta pojedini dio radi. Nakon što Unix otkrije koji će program djelovati kao tumač za skriptu, pokreće taj program i prosljeđuje mu ime skripte u naredbenom retku. Redak "shbang" je prvi redak skripte i omogućuje kernelu do znanja koja će ljuska tumačiti retke u skripti. shbang linija se sastoji od #! praćena punim nazivom putanje do ljuske, a mogu biti praćene opcijama za kontrolu ponašanja ljuske.

Comments

Komentari su opisni materijal kojem prethodi znak #. Na snazi su do kraja linije i mogu se započeti bilo gdje na liniji.

```
# this text is not
# interpreted by the shell
```

Bourneova ljuska će izvršiti svaki redak koji upišete, sve dok se ne pronađe kraj datoteke.

31.1.5 Tipičan primjer /bin/sh skripte

```
#!/bin/sh
#
# $FreeBSD: src/etc/rc.d/cron,v 1.7.10.1.4.1 2009/04/15 03:14:26 kensmith Exp $
#

# PROVIDE: cron
# REQUIRE: LOGIN cleanvar
# BEFORE: securelevel
# KEYWORD: shutdown

. /etc/rc.subr

name="cron"
rcvar="`set_rcvar`"
command="/usr/sbin/${name}"
pidfile="/var/run/${name}.pid"

load_rc_config $name
if checkyesno cron_dst
then
        cron_flags="$cron_flags -s"
fi
run_rc_command "$1"
```

31.2 Sintaksa i semantika

Sintaksa jezika Bourn shell je opisana u proširenom BNF Bachus-Naur obliku (EBNF). Proširena Backus-Naurova forma ima istu izražajnu moć kao BNF, s tim što su u njoj izvršene izmene koje doprinose čitljivosti zapisu pravila. U proširenoj verziji BNF (Extended Backus-Naur form) ili skraćeno EBNF, uvedeni su dodatni elementi u sintaksi pravila u svrhu pojednostavljenja opisa nekih gramatika. Ti elementi su opcioni dijelovi gramatika, alternative između desnih strana, ponavljanja izraza.
Kada se opisuje jezik, Backus-Naurov oblik (BNF) je formalna notacija za kodiranje gramatika namijenjenih korištenju. Mnogi programski jezici, protokoli ili formati imaju BNF opis u svojoj specifikaciji za opis sintakse. Prvi ju je predstavio John Backus za opis Algol 58 programskog jezika. Kroz BNF se definiše klasa jezika kontekstno nezavisne gramatike U BNF uočavaju se neterminalni simboli (pojmovi), terminalni simboli, pravila i startni simbol. Pojmovi se koriste da predstave klasu sintaksnih struktura. Oni se ponašaju kao sintaksne varijable, (poznati i kao neterminalni simboli). Neterminalni simboli se često uokviravaju u oštrougle zagrade. Ili se pišu kosim slovima. Terminalni simboli su leksemi ili tokeni , i pišu se podebljanim slovima ili pod navodnicima. Pravila imaju lijevu stranu (LHS), koja je jedan neterminalni simbol i desnu stranu (RHS) koja je niz terminalnih i/ili neterminalnih simbola.

Konvencije za EBNF su:

- Pomoćni simboli se zapisuju velikim početnim slovom
- Završni simboli se zapisuju pod jednostrukim navodnicima ako se sastoje od jednog karaktera, a crnim slogom (engl. bold) ako su višeslovni.
- Oble zagrade (i) se koriste za grupisanje
- Vitičaste zagrade i ograđuju dio koji se ponavlja 0 ili više puta
- Uglaste zagrade [i] opisuju opcionu konstrukciju

• Značenje ostalih oznaka je isto kao kod BNF.

```
#!/bin/sh
   # usage: fsplit file1 file2
   total=0; lost=0
   while read next
   do
   total=`expr $total + 1`
   case "$next" in
   *[A-Za-z]*)  echo "$next" >> $1 ;;
   *[0-9]*)     echo "$next" >> $2 ;;
   *)           lost=`expr $lost + 1`
   esac
   done
   echo "$total lines read, $lost thrown away"
```

31.3 Imena, vezanja i opsezi

31.3.1 Imena

Ime varijable je kombinacija niza slova, brojeva i donjih crta koji počinje donjom crtom ili slovom. Unutar naziva varijable nisu dopušteni zarezi ili praznine. Prvi znak imena varijable mora biti abeceda ili donja crta. Imena trebaju biti razumne dužine. Komande i varijable su case sensitive. Sistem datoteka je case sensitive, osjetljiv na velika i mala slova. To znači da mogu postojati datoteke pod nazivom file, File i FILE u istom folderu. Svaki fajl bi imao različit sadržaj, jer se tretiraju velika slova i mala slova kao različiti znakovi. Da bi se koristila vrijednost koja se dodijeli varijabli, dodaje se $ (znak dolara) na početak njenog imena. Dakle, varijabla $Name daje vrijednost specificiranu promjenljivom String. Može se staviti više od jednog dodjeljivanja na komandnu liniju, ali zapamtite da ljuska izvršava dodjele s desna na lijevo.

31.3.2 Rezervisane i ključne riječi

Rezervirane riječi su riječi koje imaju posebno značenje za ljusku i prepoznati su na početku reda i nakon kontrolnog operatera. Sljedeće rezervirane riječi za Bourneovu ljusku prepoznaju se samo kada se pojavljuju bez navodnika kao prva riječ naredbe.

case, do, done, elif, else, esac, fi, for, if, then, until, while, {, }, !

Ovo prepoznavanje će se dogoditi samo kada nijedan od znakova nije naveden i kada se riječ koristi kao:

- Prva riječ naredbe
- Prva riječ iza jedne od rezerviranih riječi osim **case, for** ili **in**
- Treća riječ u naredbi **case** (samo in vrijedi u ovom slučaju
- Treća riječ u naredbi **for** (samo **in** i **do** su validni u ovom slučaju)

Sljedeće riječi mogu se prepoznati kao rezervirane riječi u nekim implementacijama (kada nijedan od znakova nije naveden):

[[ ]] function select

Sve ključne riječi su ujedno i rezervisane.

31.3.3 Vezanja i opsezi

Ako pokušamo čitati nedeklarisanu varijablu dobit ćemo prazan string. Varijable nisu deklarirane i ne postoji disciplina statičnog tipa. U principu, vrijednosti su samo nizovi. Opseg varijabli u Bourne shell-u po defaultu je globalni. Varijable imaju dinamički opseg. Funkcije mogu pristupiti nelokalnim varijablama, a to je učinjeno prema hronološkom redoslijedu varijabli na izvršnom stacku - dinamički opseg, a ne prema sintaktičkom redoslijedu u skripti tekst (leksički opseg). Konstante se predstavljaju preko readonly varijabli.

31.3.4 Variable

Varijable u skriptama ljuske sadrže vrijednosti. Nazivi varijabli počinju abecednim ili podvlačnim znakom (_), a slijedi ih nula ili više alfanumeričkih ili podvlačnih znakova. Nazivi varijabli razlikuju velika i mala slova. Dakle, mačka i Mačka su različite varijable. Varijablama se dodjeljuju vrijednosti pomoću = operatora. Vrijednost varijabli može se prikazati korištenjem naredbe echo. $ prije naziva varijable govori ljusci da koristi vrijednost varijable, a ne njezino ime.

Bourneova ljuska ima slijedeće vrste varijabli
• Korisnički definirane varijable
• Varijable okruženja
• Pozicione varijable ili argumenti ljuske
• Unaprijed definirane ili posebne varijable

31.3.4.1 Korisnički definirane varijable

Kao što naziv implicira, korisnički definirane varijable su kakve god želimo da budu. Nazivi varijabli sastoje se od alfanumeričkih znakova i znaka podvlačenja, uz uslov da nazivi varijabli ne počinju s jednom od brojeva od 0 do 9. Kao i svi UNIX nazivi, varijable su osjetljive na velika i mala slova. Nazivi varijabli poprimaju vrijednosti kada se pojave u naredbeni redak lijevo od znaka jednakosti (=). Na primjer, u sljedećim redovima za naredbe, COUNT poprima vrijednost 1, a NAME poprima vrijednost Stephanie.

```
1 $ COUNT=1
2 $ NAME=Stephanie
```

Školjka prepoznaje alfanumeričke varijable kojima se mogu dodijeliti vrijednosti niza. Da bi se imenu dodijelila vrijednost niza, unosi se:

Ime = String

Ako varijablu String priložite s ” ili ’ (dvostruki ili jednostruki navodnici), ljuska ne tretira praznine, tabulatore, točke-zareze i znakove novog retka unutar niza kao graničnike riječi, već ih ugrađuje doslovno u niz.
Ako varijablu String priložite s ” (dvostruki navodnici), ljuska i dalje prepoznaje nazive varijabli u nizu i izvodi zamjenu varijabli; to jest, zamjenjuje reference na pozicijske parametre i druge nazive varijabli kojima je predgovor $ (znak dolara) s njihovim odgovarajućim vrijednostima,

ako ih ima.Ljuska također izvodi zamjenu naredbi unutar nizova koji su zatvoreni u dvostrukim navodnicima.
Ako varijablu String priložite s ' (jednostruki navodnici), ljuska ne zamjenjuje varijable ili naredbe unutar niza. Sljedeći slijed ilustrira ovu razliku:

You:
num=875
number1="Add $num"
number2='Add $num'
echo $number1
System:
Add 875
You:
echo $number2
System:
Add $num

Školjka ne interpretira prazna mjesta u dodjeli nakon zamjene varijable. Dakle, sljedeće dodjele rezultiraju da $first i $second imaju istu vrijednost:

```
first='a string with embedded blanks'
second=$first
```

Kada se referencira varijabla, može se staviti ime varijable (ili cifru koja označava pozicijski parametar) u (zagrade) da se razgraniči ime varijable od bilo kojeg niza koji slijedi. Konkretno, ako je znak odmah iza imena slovo, cifra ili donja crta, a varijabla nije pozicijski parametar, tada su zagrade potrebne:

You:

a='This is a'
echo "$an example"

System:

This is an example

You:

echo "$a test"

System:

This is a test

31.3.4.2 Varijable okruženja

Globalne varijable okoline vidljive su iz sesije ljuske i bilo kojeg podređenog procesa kojeg ljuska stvara. Lokalne varijable dostupne su samo u ljusci koja ih stvara. To čini globalne varijable

okruženja korisnim u aplikacijama koje pokreću podređene procese koji zahtijevaju informacije od nadređenog procesa. Linux sustav postavlja nekoliko globalnih varijabli okruženja kada pokrenete svoju bash sesiju. Varijable okoline sustava uvijek koriste sva velika slova kako bi se razlikovale od normalnih varijabli korisničkog okruženja.

Varijable okruženja su varijable koje imaju posebno značenje za ljusku. Oni se trebaju koristiti za prilagodbu okruženja za određenog korisnika. Te bi varijable trebale biti definirane u posebnom programu koji se izvršava tokom prijave pod nazivom datoteka .profile (profil tačke). Sve definirane varijable tada će biti postavljene do kraja sesije osim ako se eksplicitno ne ponište ili ponovno definiraju. Tablica 2.3 sadrži abecedni popis varijabli okoline Bourneove ljuske, kratak opis za što se svaka koristi i zadanu postavku.

Nazivi varijabli i njihove definicije:

CDPATH - Popis direktorija odvojenih dvotočkama koji se koriste kao put za pretraživanje za ugrađenu naredbu cd.

HOME - Početni direktorij trenutnog korisnika; zadana vrijednost za ugrađeni cd. Varijabla definira kamo ide cd kada se izvršava bez ikakvih argumenata. HOME enviroment varijabla je postavljena procesom prijave.

IFS - Popis znakova koji odvajaju polja; koristi se kada ljuska dijeli riječi kao dio proširenja.

MAIL - Ako je ovaj parametar postavljen na ime fajla i MAILPATH varijabla nije postavljena, Bash informiše korisnika o mailu u specifičnom fajlu.

MAILPATH - Popis naziva datoteka odvojenih dvotočkama koje ljuska povremeno provjerava ima li nove pošte.

OPTARG - Vrijednost argumenta zadnje opcije procesirane ugrađenim getopts.

OPTIND - Indeks argumenta zadnje opcije procesirane ugrađenim getopts.

PATH - Lista direktorija odvojenih dvotočkama u kojima ljuska traži naredbe.

PS1 - Primarni prompt string. Zadana vrijednost je ”\s-\v$ ”.

PS2- Sekundarni prompt string. Zadana vrijednost je ”> ”.

31.3.4.3 Unaprijed definirane ili posebne varijable

Nekoliko varijabli ima posebna značenja. Sljedeće varijable postavlja samo ljuska.

$@ Proširuje pozicione parametre, počevši od $1. Svaki parametar je odvojen razmakom. Ako se stave ” (dvostruki navodnici) oko $@, ljuska svaki pozicioni parametar smatra zasebnim stringom. Ako ne postoje pozicioni parametri, Bourne shell proširuje izraz na null string bez navodnika.

$* Proširuje pozicione parametre, počevši od $1. Shell svaki parametar odvaja prvim znakom vrijednosti IFS varijable. Ako se stavi ” (dvostruki navodnici) oko $*, ljuska uključuje vrijednosti pozicijskih parametara, u dvostrukim navodnicima. Svaka vrijednost je odvojena pr-

vim znakom IFS varijable.

$# Određuje broj pozicionih parametara proslijeđenih shell-u, ne računajući ime same shell procedure. Varijabla $# tako daje broj pozicionog parametra s najvećim brojem koji je postavljen. Jedna od primarnih upotreba ove varijable je provjera prisutnosti potrebnog broja argumenata. Samo pozicioni parametri od $0 do $9 su dostupni kroz shell.

$? Određuje izlaznu vrijednost posljednje izvršene naredbe. Njegova vrijednost je decimalni niz. Većina naredbi vraća vrijednost od 0 koja označava uspješan završetak. Sama ljuska vraća trenutnu vrijednost $? varijabla kao njegova izlazna vrijednost.

$$ Identificira broj procesa trenutnog procesa. Budući da su brojevi procesa jedinstveni među svim postojećim procesima, ovaj niz se često koristi za generiranje jedinstvenih imena za privremene datoteke.

$! Određuje broj procesa posljednjeg procesa koji je pokrenut u pozadini pomoću terminatora & (ampersand).

$- Niz koji se sastoji od imena izvršnih zastavica trenutno postavljenih u ljusci.

31.4 Tipovi

U Bourne Shell-u ne postoje tipovi podataka. Bourne shell ne pruža nikakve alate za rukovanje numeričkom analizom. Bourne shell ne pruža nikakve kompleksne strukture podataka. Zapravo ljuska sve vidi kao string, pa za svaki programski zadatak koji zahtijeva nizove ili strukture, zapisi su nemogući. Shell nije ”dobra” za velike programske zadatke. Shell funkcije ne daju pravu modulacija i opet nedostaju tipovi podataka.

Sve varijable su tipa string. Ljuska ne mari za vrste varijabli; mogu pohraniti stringove, integere brojeve, stvarne brojeve - sve što želite. Stringovi, integeri, realni brojevi, itd. se čuvaju kao stringovi i nema sintaksne razlike (npr. x=hello, y=9, z=9.63), ali će rutine koje očekuju broj raditi ispravno jedino ako su im argumenti stvarno brojevi npr. ako varijabli dodjelimo neki string i probamo dodati broj 1 sabiranjem, dobit ćemo error “expr: non-numeric argument”. To je zato što eksterni program expr očekuje brojeve kao argumente. Bourne shell nema ugrađeni tip podataka kao što je niz ili array, ali se on može simulirati komandom “set” i čuvanjem varijabli kao 1,2, itd. Niz se također može simulirati spašavanjem varijabli npr. sa imenima array1, array2, itd. i korištenjem komande “eval”. Regularni izrazi (regex) nisu ugrađeni u Bourne Shell, ali se mogu simulirati korištenjem “grep” komande.

Bourne shell ne pruža nikakve alate za rukovanje numeričkom analizom . Bourne shell ne pruža nikakve kompleksne strukture podataka. Zapravo ljuska sve vidi kao niz, pa za svaki programski zadatak koji zahtijeva nizove ili strukture, zapisi su nemogući. Shell nije ”dobra” za velike programske zadatke. Shell funkcije ne daju pravu modulaciju, jer nedostaju tipovi podataka.

31.4.1 Osnovni tipovi podataka

Bourne shell ima jedan osnovni tip podataka a to je string.

31.5 Aritmetički izrazi

Bourneova ljuska izvorno nije imala nikakav mehanizam za izvođenje jednostavnih aritmetičkih operacija, ali koristi vanjske programe, bilo awk ili expr.
Aritmetika nije ugrađena u Bourne shell. Ako se trebaju izvesti jednostavne integer aritmetičke izračune, u skriptama Bourne shell najčešće se koristi naredba UNIX expr. Za aritmetiku s pomičnim zarezom mogu se koristiti programi awk ili bc. Budući da aritmetika nije ugrađena, performanse ljuske se pogoršavaju kada se ponavlja kroz petlje više puta. Svaki put kada se brojač povećava ili smanjuje u mehanizmu petlje, potrebno je račvati drugi proces za rukovanje aritmetikom. Značajke artimetičkog izraza su u Bourne shell jedino dostupne putem eksterne komande expr Stoga je to još jedan primjer poželjne značajke koju pruža vanjska naredba koja je bolje integrirana u ljusku. [[...]] i getopts također su primjeri ovog načina dizajna.

Sljedeći primjer pokazuje kako sabrati dva broja:

```
#!/bin/sh

val='expr 2 + 2'
echo "Total value : $val"
```

Ova skripta će generirati sljedeći rezultat:

```
Total value : 4
```

Prilikom sabiranja treba uzeti u obzir sljedeće:
- Mora postojati razmak između operatora i izraza.
Na primjer, 2+2 nije tačno; trebalo bi biti napisano kao 2 +; 2.
- Potpuni izraz trebao bi biti zatvoren između ‘ ‘, koji se zove obrnuti zarez

31.5.1 Operatori

Postoje različiti operatori koje podržava svaki shell. Kada je riječ o Bourne shell podržani su slijedeći operatori:

- Aritmetički operatori
- Relacijski operatori
- Booleovi operatori
- String operatori
- File test operatori

Character operator nije u Bourne shell, a umjesto njega je operator koji se zove string operator.

31.5.1.1 Aritmetički operatori

Bourne Shell podržava sljedeće aritmetičke operatore. Pretpostavimo da varijabla a drži 10, a varijabla b drži 20.

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

Vrlo je važno razumjeti da svi uvjetni izrazi trebaju biti unutar uglastih zagrada s razmacima oko njih, na primjer **[$a == $b]** je tačan, dok je **[$a==$b]** netačan. Svi aritmetički izračuni se rade pomoću dugih integera.

Slijedi primjer koji koristi sve artimetičke operatore:

```
#!/bin/sh

a=10
b=20

val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
    echo "a is equal to b"
fi

if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```

Gornja skripta generiše slijedeći rezultat:

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

31.5.1.2 Relacioni operatori

Bourne Shell podržava slijedeće relacijske operatore koji su specifični za numeričke vrijednosti. Ovi operateri ne rade za vrijednosti stringa osim ako je njihova vrijednost numerička. Na primjer, sljedeći operateri će raditi na provjeravanju odnosa između 10 i 20, kao i između ”10” i ”20” ali ne između ”deset” i ”dvadeset”. Pretpostavimo da varijabla a drži 10, a varijabla b drži 20 tada Bourne Shell podržava sljedeće relacijske operatore koji su specifični za numeričke vrijednosti. Ovi operatori ne rade za vrijednosti niza osim ako njihova vrijednost nije numerička.

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

Vrlo je važno razumjeti da se svi uvjetni izrazi trebaju staviti unutar uglatih zagrada s razmacima oko njih. Na primjer, **[$a <= $b]** je tačno dok je **[$a <= $b]** netačno.

```
#!/bin/sh

a=10
b=20

if [ $a -eq $b ]
then
    echo "$a -eq $b : a is equal to b"
else
    echo "$a -eq $b : a is not equal to b"
fi

if [ $a -ne $b ]
then
    echo "$a -ne $b : a is not equal to b"
else
    echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
    echo "$a -gt $b : a is greater than b"
else
    echo "$a -gt $b : a is not greater than b"
fi

if [ $a -lt $b ]
then
    echo "$a -lt $b : a is less than b"
```

Gornja skripta generiše sljedeći rezultat:

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not grater than b
10 -lt 20: a is less than b
```

31.5.1.3 Boolean operatori

Sljedeće Booleove operatore podržava Bourne Shell.

Pretpostavimo da varijabla a drži 10, a varijabla b drži 20 tada

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

```
#!/bin/sh

a=10
b=20

if [ $a != $b ]
then
    echo "$a != $b : a is not equal to b"
else
    echo "$a != $b : a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
    echo "$a -lt 100 -a $b -gt 15 : returns true"
else
    echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
    echo "$a -lt 100 -o $b -gt 100 : returns true"
else
    echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
    echo "$a -lt 5 -o $b -gt 100 : returns true"
else
    echo "$a -lt 5 -o $b -gt 100 : returns false"
fi
```

Gornja skripta će generirati sljedeći rezultat:

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

31.5.1.4 String operatori

Bourne Shell podržava sljedeće string operatore. Pretpostavimo da varijabla a drži ”abc”, a varijabla b drži ”efg”, tada:

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

Postoji nekoliko operatora koji se mogu koristiti za testiranje različitih svojstava povezanih s Unix datotekom.

```
#!/bin/sh

a="abc"
b="efg"

if [ $a = $b ]
then
    echo "$a = $b : a is equal to b"
else
    echo "$a = $b : a is not equal to b"
fi

if [ $a != $b ]
then
    echo "$a != $b : a is not equal to b"
else
    echo "$a != $b : a is equal to b"
fi

if [ -z $a ]
then
    echo "-z $a : string length is zero"
else
    echo "-z $a : string length is not zero"
fi
```

Gornja skripta će generirati sljedeći rezultat:

```
abc = efg : a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

31.5.1.5 Pattern matching

```
Podudaranje sa 0 ili vise karaktera: *
Podudaranje sa 1 karakterom: ?
Podudaranje sa bilo kojim karakterom
iz liste: [AaBbCc]
Podudaranje sa bilo kojim karakterom
koji nije naveden u listi: [^RGB]
Podudaranje sa karakterom iz opsega: [a-g]
```

31.6 Osnovne programske strukture

Bourne shell podržava sljedeće programske strukture za kontrolu toka:

- if iskaz,
- case iskaz,
- for,
- while petlju,
- until petlju

IF naredba se koristi da bi se na osnovu nekog uslova izvršio određeni set komandi. If je ključna reč i koristi uslovni iskaz da bi odredila da li da izvrši ili ne određenu akciju.

Može se primjeniti u slijedećim oblicima:

- **If-then**

Naredba if procjenjuje izraz i zatim vraća kontrolu na temelju ovog statusa. Naredba fi označava kraj if, (fi je if napisano unatrag). Naredba if izvršava naredbe neposredno nakon nje ako izraz vraća status true. Ako je povratni status false, kontrola će se prenijeti na izjavu nakon fi.

```
Command Format:  if expression
                    then  commands
                 fi
```

- **If-then-else**

Naredba if procjenjuje izraz i zatim vraća kontrolu na temelju ovog statusa. Naredba fi označava kraj if. Naredba if izvršava naredbe neposredno nakon nje ako izraz vraća status true. Ako je status povratka false, kontrola će se prenijeti na izjavu nakon fi.

```
Command Format:  if expression
                    then commands
                    else commands
                 fi
```

- **If-then-elif**

Konstrukcija elif kombinira naredbe else i if i dopušta da se konstruiše ugniježđeni skup if then else struktura.

```
Command Format:  if expression
                       then commands
                    elif expression
                       then commands
                       else commands
                  fi
```

CASE

Naredba **case** može se koristiti za izvršavanje naredbi na temelju određene postavke druge varijable. Case struktura omogućava mehanizam odlučivanja s više grana. Put koji se uzima zavisi o podudarnosti između test-stringa i jedan od uzoraka. Slična je if naredbi, ali se grana na više slučajeva, pa se često koristi da zamjeni višestruke if-then-else izraze. Međutim, ponekad je potreban fleksibilniji test višestrukih uslova. Tu case stupa na scenu. Šalje se varijabla ovoj instrukciji, a onda se specificira seriju akcija koje će biti preduzete u zavisnosti od vrednosti varijable. U takvim slučajevima case je bolja alternativa zbog čitljivosti i fleksibilnosti. Karakter "|" se koristi da razdvoji uzorke, a ")" da označi kraj jedne liste uzoraka. Broj case slučajeva je proizvoljan, a svaki završava sa ";;".

Sintaksa je:

```
Command Format:  case test-string in
                       pattern-1 ) commands-1 ;;
                       pattern-2 ) commands-2 ;;
                       pattern-3 ) commands-3 ;;
                       .
                       .
                       .
                       *)          commands   ;;
                  esac
```

FOR

Ova struktura će dodijeliti vrijednost prve stavke u listu argumenata u indeks petlje i izvršava naredbe između komandi do i done. Naredbe do i done pokazuju početak i kraj for petlje. Nakon što struktura preda kontrolu done iskazu, dodjeljuje vrijednost druge stavke na listi argumenata indeks petlje i ponavlja naredbe. Struktura će se ponoviti naredbe između naredbi do i done jednom za svaku argument na listi argumenata. Kada je lista argumenata iscrpljena, kontrola prelazi na naredbu koja slijedi done.

Osnovna sintaksa petlje je:

```
Command Format:  for loop-index in argument-list
                       do
                       commands
                       done
```

WHILE

While petlja se izvršava sve dok je određeni uslov ispunjen. Sve dok izraz vraća istinit izlazni status, struktura nastavlja izvršavati naredbe između do i done izjava. Prije svake petlje kroz naredbe, struktura izvršava izraz. Kada je izlazni status izraza netačan (ne-nula), kontrola se prosljeđuje sljedećem iskazu nakon done. Naredbe koje se trebaju izvršiti moraju promijeniti test izraza ili to može rezultirati beskonačnoma petljom.

Sintaksa je:

```
Command Format:  while expression
                 do
                 commands
                 done
```

UNTIL

Pored WHILE petlje postoji i **until** petlja. Strukture WHILE i UNTIL su vrlo slične. Jedinina razlika je u tome što je test na vrhu petlje. UNTIL struktura će nastaviti petlju sve dok izraz ne vrati true ili stanje bez greške. Izvršava se sve dok uslov nije ispunjen, što je obrnuto od WHILE petlje. Until se rijetko koristi.

Sintaksa je:

```
Command Format:  until expression
                 do
                 commands
                 done
```

31.7 Potprogrami

Kako programi ljuske rastu u veličini, postaje potrebno nametnuti neku globalnu strukturu pomoću potprograma. Potrebno je podijelite program u module koji se mogu izvoditi kao zasebni potprogrami i pozivati koliko god često želite. Rad u Bourne Shell-u se u tom pogledu ne razlikuje od rada na bilo kojem drugom jeziku. Mogu im se proslijediti argumenti i pristupiti im na isti način kao i glavnoj skripti. Iz jednog shell programa možemo pozivati druge shell ili bilo koje izvršive ili uslužne programe. Programe koje pozovemo iz jednog okruženja se izvršavaju unutar novog procesa i u svom vlastitom okruženju. Između okruženja se dijele jedino environment varijable. Varijable se također mogu eksportovati iz jednog okruženja u drugo, korištenjem naredbe “export”. Potprogram je definiran konstrukcijom:

```
subprogram_name() {
 ...
 return
}
```

Mogu postojati i druge povratne izjave. Pozivaju ga:

```
subprogram_name arg_list
```

Jednostavni primjer:

```
#!/bin/bash

mysub() {
  echo "I got passed: $1"
  return
}

mysub abc
mysub def

exit 0
```

Nijedan proceduralni programski jezik ne bi bio potpun bez nekog pojma potprograma, funkcija ili drugih sličnih konstrukcija. Bourne shell nije iznimka.
U Bourne shell postoje dva osnovna načina pristupa potprogramima. Prvi je pomoću izvršavanja vanjskih alata (što može uključivati skriptu koja se sama izvršava rekurzivno). Također možete učiniti da izvršenje jedne naredbe bude uvjetovano kodom rezultata koji je vratila druga naredba.
Drugi način pristupa potprogramima (i onaj koji općenito rezultira boljom izvedbom) je korištenje stvarnih potprograma. Također, mogu se napisati kratke, jednostavni potprogrami u liniji. Potprogrami se izvode unutar iste instance ljuske kao i glavna skripta ljuske. Kao rezultat toga, sve varijable ljuske prema zadanim se postavkama dijele između potprograma i glavnog tijela programa. To stvara mali problem pri pisanju rekurzivnog koda. Na sreću, varijable ne moraju ostati globalne. Pravila opsega za potprograme ljuske razlikuju se od pravila opsega za većinu drugih programskih jezika.
Može biti korisno uključiti jednu cijelu skriptu ljuske u drugu. Također se mogu izvršiti vanjske skripte u pozadini i provjeriti njihov status kasnije.
Potprogrami osnove
Potprogrami u Bourne shell izgledaju sličvo C funkcijama bez liste argumenata. Ovi potprogrami pozivaju baš kao što se pokreće program, a potprogrami se mogu koristiti bilo gdje gdje možete koristiti izvršna datoteku.

```
#!/bin/sh

mysub()
{
        echo "Arg 1: $1"
}

mysub "This is an arg"
```

Baš kao što su argumenti shell skripte pohranjeni u shell varijablama pod nazivom $1, $2 i tako dalje, tako su i argumenti za potprograme ljuske. Zapravo, na većinu načina, shell potpro-

grami se ponašaju tačno kao izvršavanje vanjske skripte. Jedno mjesto na kojem se ponašaju drugačije je u opsegu varijabli. Pogledajte Opseg varijable za više informacija. Općenito, potprogram može učiniti sve što može učiniti skripta ljuske. Bourne shell omogućuje da se grupira više od jedne naredbe zajedno i da ih obje tretiraju kao zasebnu naredbu. Zapravo, stvara se anonimni potprogram inline.

31.7.1 Opseg varijable

Pravila opsega za potprograme ljuske razlikuju se od pravila opsega za većinu drugih programskih jezika. Potprogrami se izvode unutar iste instance ljuske kao i glavna skripta ljuske. Kao rezultat toga, sve varijable ljuske prema zadanim se postavkama dijele između potprograma i glavnog tijela programa. To stvara mali problem pri pisanju rekurzivnog koda. Na sreću, varijable ne moraju ostati globalne.

31.7.2 Deklarisanje lokalne varijable

Za deklariranje varijable lokalno za dati potprogramu, koristi se lokalni izraz.

```
#!/bin/sh

mysub()
{
        local MYVAR
        MYVAR=3
        echo "SUBROUTINE: MYVAR IS $MYVAR";
}

MYVAR=4
echo "MYVAR INITIALLY $MYVAR"
mysub "This is an arg"
echo "MYVAR STILL $MYVAR"
```

31.7.3 Korištenje globalnih varijabli u potprogramima

Općenito, slobodno se mogu čitati i mijenjati globalne varijable unutar bilo kojeg potprograma. Međutim, postoje dvije situacije u kojima to nije slučaj:

- Promjene varijabli koje su prethodno deklarirane kao lokalne u trenutnom stacku poziva.
- Promjene napravljene u potprogramima pozvanim putem inline izvršenja. Ako pozovete potprogram korištenjem inline izvršenja, taj potprogram dobiva lokalnu kopiju svih varijabli ljuske. Promjene napravljene na tim varijablama ne prenose se natrag u kontekst glavne skripte, jer se potprogram izvršava u zasebnoj ljusci.

Uključujući jednu shell skriptu unutar druge (sourcing)
Kao i kod svakog programskog jezika koji uključuje potprograme, često je korisno izgraditi biblioteku uobičajenih potprograma koje vaše skripte mogu koristiti. Kako bi se izbjeglo dupliciranje ovog sadržaja, skriptni jezik Bourne shell pruža mehanizam za uključivanje jedne skripte ljuske unutar druge referencom. Ovaj proces se obično naziva sourcing. Za izvor jedne skripte iz druge, koristit se . builtin. Stvoriti datoteku koja sadrži potprogram mysub, nazivamo ga mysub.sh Da bi se taj potprogram koristio u drugoj skripti, treba uraditi sljedeće:

```
#!/bin/sh
MYVAR=4

# The next line sources the external script.
. /path/to/mysub.sh

echo "MYVAR INITIALLY $MYVAR"
mysub "This is an arg"
echo "MYVAR STILL $MYVAR"
```

31.8 Bourne shell built-in komande

Svaka Unix ljuska ima barem neke ugrađene naredbe. Ove ugrađene naredbe dio su ljuske i implementirane su kao dio izvornog koda ljuske. Školjka prepoznaje da je naredba koju je od nje zatraženo da izvrši jedna od njezinih ugrađenih komponenti i izvodi tu akciju samostalno, bez pozivanja zasebne izvršne datoteke.

Bourne shell ima sljedeće built-in komande (komanda - opis):

: - Vraća nultu izlaznu vrijednost
. - Čita i izvršava naredbe iz parametra datoteke i zatim vraća.
break - Izlazi iz okvira for, while ili until petlje naredbe, ako ih ima.
cd - Mijenja trenutni direktorij u navedeni direktorij.
continue - Nastavlja sljedeću iteraciju okruženja for, while ili until petlje naredbe.
echo - Zapisuje znakovne nizove u standardni izlaz
eval - Čita argumente kao ulaz u ljusku i izvršava rezultirajuću naredbu ili naredbe.
exec - Izvršava naredbu specificiranu parametrom Argument, umjesto ove ljuske, bez stvaranja novog procesa.
exit - Izlazi iz ljuske čiji je izlazni status određen parametrom n.
export - Označava nazive za automatski izvoz u okruženje naknadno izvršenih naredbi.
hash - Pronalazi i pamti lokaciju na stazi pretraživanja navedenih naredbi.
pwd - Prikazuje trenutni imenik
read - Čita jedan redak iz standardnog unosa
readonly - Označava ime specificirano u Ime parametra kao read-only (samo za čitanje)
return - Uzrokuje izlazak funkcije s navedenom povratnom vrijednošću
set - Upravlja prikazom raznih parametara na standardni izlaz
shift - Pomiče argumente naredbenog retka ulijevo
test - Procjenjuje uvjetne izraze
times - Prikazuje akumulirana vremena korisnika i sustava za procese pokrenute iz ljuske
trap - Pokreće određenu naredbu kada ljuska primi određeni signal ili signale
type - Tumači kako bi ljuska interpretirala navedeno ime kao ime naredbe
ulimit - Prikazuje ili prilagođava dodijeljene resurse ljuske
umask - Određuje dopuštenja datoteke
unset - Uklanja varijablu ili funkciju koja odgovara navedenom imenu
wait - Čeka da se navedeni podređeni proces završi i izvješćuje o statusu

završetka

31.9 Quoting

Quoting (citiranje) se koristi za uklanjanje posebnog značenja određenih znakova ili riječi iz ljuske. Quoting se može koristiti za očuvanje doslovnog značenja posebnih znakova u slijedećem odlomku, sprječavanje prepoznavanja rezerviranih riječi kao takvih i sprječavanje proširenja parametara i zamjene naredbi unutar obrade Here dokumenta.
Bourne shell podržava nekoliko vrsta navodnika. Koji će se koristiti ovisi o tome što se želi učiniti.

31.9.1 Obrnuta kosa crta

Baš kao u C stringovima, obrnuta kosa crta ("\") uklanja svako posebno značenje iz znaka koji slijedi. Ako znak nakon obrnute kose crte nije poseban za početak, obrnuta kosa crta nema učinka. Obrnuta kosa crta je sama po sebi posebna, pa da bi je izbjegli, samo se udvostruči: \\.

31.9.2 Pojedinačni navodnici

Jednostruki navodnici, kao npr:

```
'foo'
```

radi otprilike onako kako bi se očekivalo - sve što je unutar njih (osim samih navodnika) citira se.
Može se reći

```
echo '* MAKE   $$$   FAST *'
```

sačuvaće razmake i većinu posebnih znakova. Međutim, varijable i izrazi s navodnicima prošireni su i zamijenjeni svojom vrijednošću.

31.9.3 Obrnuti navodnici

Ako je izraz unutar obrnutih navodnika (poznati kao obrnuti navodnici ili pozadinske oznake), npr.

```
`cmd`
```

izraz se evaluira kao naredba i zamjenjuje onim što izraz ispisuje na svoj standardni izlaz. Tako,

```
echo You are `whoami`
```

printa

```
You are arensb
```

31.9.4 Dvostruki navodnici

```
"foo"
```

Čuva razmake i većinu posebnih znakova. Međutim, varijable i izrazi s navodnicima prošireni

su i zamijenjeni svojom vrijednošću.

31.10 Apstraktni tipovi podataka, enkapsulacija

31.10.1 Apstrakcija

Apstrakcija podataka jedna je od najvažnijih značajki objektno orijentiranog programiranja. Apstrakcija znači prikazivanje samo bitnih informacija i skrivanje detalja. Apstrakcija podataka odnosi se na pružanje samo bitnih informacija o podacima vanjskom svijetu, skrivajući pozadinske detalje ili implementaciju.
Apstraktni tipovi podataka su korisnički definisani tipovi podataka koji zadovoljavaju slijedeća dva uslova:
- Predstavljanje i operacija nad objektima su definisani u jednoj sintaksnoj jedinici
- Interno predstavljanje objekata datog tipa je skriveno od programskih jedinica koje koriste ove objekte, pa su jedine operacije moguće one koje su date u definiciji tipa.

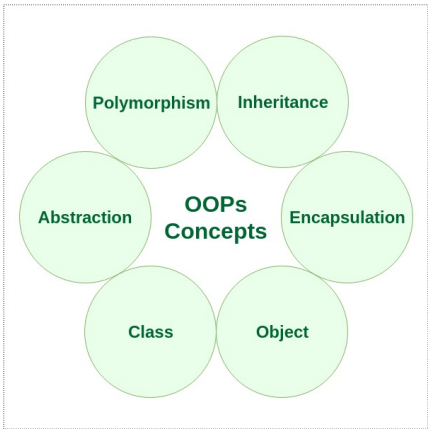
Bourne shell je skriptni programski jezik koji nije namjenjen za neke stvari kao što su neki opšti jezici kao što su C, C++, Java, itd. Bourne shell nema podršku za apstraktne tipove podataka.

31.10.2 Enkapsulacija

U objektno orijentiranom programiranju, enkapsulacija ili čahurenje označava grupisanje podataka sa metodama koje rade s tim podacima, ili restrikciju direktnog pristupa nekim komponentama objekta. Enkapsulacija se koristi za skrivanje vrijednosti ili stanja struktuiranog podatkovnog objekta unutar klase, sprečavajući direktan pristup njima od strane klijenta u smislu otkrivanja detalja skrivene implementacije ili narušavanja stanja nepromjenjivosti koju održavaju druge metode. Glavna razlika između apstrakcija podataka i encapsulation je da apstrakcija podataka skriva detalje implementacije i prikazuje samo funkcionalnost korisniku kako bi se smanjila kompleksnost koda dok enkapsulacija veže ili oblaže podatke i metode zajedno u jednu jedinicu i skriva detalje za zaštitu podataka.
Bourne shell nema podršku za enkapsulaciju.

31.11 Podrška objektno-orijentisanom programiranju

Objektno orijentirano programiranje podržava značajke kao što su nasljeđivanje, inkapsulacija, polimorfizam, apstrakcija.



Bourne shell nije objektno orijentirani jezik, jer uopće nema podršku za objektno orijentirano programiranje, enkapsulaciju, nasljeđivanje i polimorfizam. Budući da je Bourneova ljuska (i svi derivati) 1984. godine dobila podršku za korisnički definirane funkcije ljuske, može se smatrati proceduralnim jezikom. Shell se pokreće u naredbenom retku tako da se ni on ne bi pokrenuo imalo smisla u ovom jeziku.

Polimoformizam
Riječ polimorfizam znači imati mnogo oblika. Jednostavnim riječima, polimorfizam možemo definirati kao sposobnost da se poruka prikaže u više oblika. Operacija može pokazati različita ponašanja u različitim slučajevima. Ponašanje ovisi o vrstama podataka koji se koriste u operaciji.

31.11.1 Funkcionalni programski jezici

Dizajn imperativnih jezika je baziran na von Neumann arhitekturi. Za većinu programera, imperativni jezici su zadovoljavajući i napredovali su kroz duži period. Primarno pitanje imperativnih jezika je efikasnost.
Dizajn funkcionalnih jezika je baziran na matematičkim funkcijama. Solidna teoretska baza bliža korisnicima, ali nevezana za arhitekturu na kojim programi rade.
Osnovni proces računa je fundamentalno drugačiji u funkcionalnim programskim jezicima u odnosu na imperativne jezike. U imperativnom jeziku obave se operacije, a rezultat se smjesti u varijablama za kasniju upotrebu.
Jedan od načina računa parametara u nestriktnim jezicima je lijena evaluacija. To znači da se računaju samo vrijednosti koje su potrebne i u trenutku kada postanu potrebne.
Kada je u pitanju Born Shell, on ne spade u funkcionalne jezike, ne podržava prave lambda funkcije kako ih poznajemo u drguim programskim jezicima. Bourne shell nije imperativni jezik. Bourne shell je jezik interfejsa naredbenog retka.
Kada je riječ o lijenoj evaluaciji, tako nešto nije podržano ni u jednom Shell jeziku.

31.11.2 Podrška za logičko i deklarativno programiranje

U Bourne Shellu nije moguće pisati programe koji se sastoje isključivo od operacija, kao ni u jednom drugom Shell programskom jeziku.

31.11.3 Domensko specifični jezici

Interpreteri prevode i izvršavaju program liniju po liniju. Oni Proizvode izlaz prije nego se vidi cijeli program pa su korisni su ako je program dugačak ili ako korisnik želi da vidi rezultate izvršenja prije pisanja novog dijela.
Komandni jezici poput Unix shell, scripting languages su primjeri domenskih jezika koji se realizuju interpreterski.

31.12 Zaključak

Bourneova ljuska je mali program koji radi na operativnim sustavima Unix® i LINUX® i pruža interfejs za izvršavanje programa na sistemu. Često se naziva interfejs naredbenog retka ili interpreter naredbi. Naredbe i svi potrebni parametri koji se trebaju izvršiti upisuju se u ljusku. Bourne shell je skriptni jezik koji korisnicima omogućuje stvaranje i izvršavanje skriptnih datoteka koje mogu obraditi podatke kroz više programa putem jedne naredbe. Na suistemima sličnim Unix®, program je jednostavno poznat kao ”sh.”

Bourne shell je bio planiran kao skriptni jezik od samog početka. Iako je to još uvijek bilo sučelje naredbenog retka za Unix® verziju sedam, također je otvorilo mogućnost korisnicima da razviju shell skripte koje bi povezivale naredbe zajedno u svrhu obrade podataka. Korištenjem ovog programiranja, korisnik može uspostaviti varijable za hvatanje poznatih ili nepoznatih podataka s ulaza ili izlaza i manipulirati obradom tih podataka korištenjem uvjetnih iskaza u skripti putem tehnike koja se naziva kontrolni tok.

Ovo je bila prva ljuska koja je implementirala značajku poznatu kao rukovanje signalom. Putem Bourneove ljuske, korisnik može poslati određenu vrstu signala procesu koji je već pokrenut na računalu, nalažući tom procesu da učini nešto drugo.

Bourne shell bila je prva sa mogućnošću direktne kontrole deskriptora. Na sistamu sličnom Unixu, svaki pokrenuti program ima tablelu koja navodi deskriptore datoteka za svaku otvorenu datoteku. Da bi korisnik imao kontrolu nad deskriptorima datoteka na sistemu, omogućio je neviđenu kontrolu nad ulazom i izlazom za gotovo sve na računalu.

Iako je Bourne shell korisnicima nudila takvu dodatnu funkcionalnost, nedostajale su joj značajke poput mogućnosti interaktivne kontrole procesa, uspostavljanja aliasa naredbi i zadržavanja historije. Međutim, kasnije se počelo pojavljivati niz potomaka koji su uzeli najkorisnije značajke ljuske koje su osmišljene tokom godina i umotale ih u nove školjke. Jedan uobičajen primjer je Bourne-again shell, ili Bash, koji je uobičajen na mnogim LINUX® sistemima. Kao rezultat toga, mnogi od ovih potomaka su u potpunosti sposobni izvršavati redovne Bourneove skripte ljuske, dajući svakom sistemu sličnom Unix®-u neku implementaciju izvorne Bourneove ljuske na ovaj ili onaj način. Na mnogim LINUX® sisitemima, ovo je jednostavno veza od ”sh” do ”bash” ili nekog drugog sposobnog potomka.

BIBLIOGRAFIJA

[1] IBM®
https://www.ibm.com/docs/en/aix/7.1?topic=shells-bourne-shell

[2] Bourne Shell Tutorial - The Grymoire!.
https://www.grymoire.com/Unix/Bourne.html

[3] Bourne Shell Builtins
https://www.gnu.org/software/bash/manual/html_node/Bourne-Shell-Builtins.html

[4] The Shell Scripting Tutorial
https://www.shellscript.sh/

[5] Bourne Shell Programming
https://www.ooblick.com/text/sh/

[6] The Bourne Shell Syntax and Constructs
https://se.ifmo.ru/ ad/Documentation/Shells_by_Example/ch02lev1sec4.html

[7] Bourne Shell Reference - LinuxReviews
https://linuxreviews.org/Bourne_Shell_Reference

[8] Variable and File Name Substitution in the Bourne Shell
https://sites.ualberta.ca/dept/chemeng/AIX-43/share/man/info/C/a_doc_-
lib/aixuser/usrosdev/var_file_name_subst_bourne.htm

[9] Subroutines, Scoping, and Sourcing
https://developer.apple.com/library/archive/documentation/OpenSource/Conceptual/ShellScripting/Sub

[10] Bourne Shell Programing
http://ppmps.zesoi.fer.hr/literatura/Bourne_Shell_Programming/Bourne_Shell_-
Programming.htm