flex & bison

Zorica Suvajdžin Rakić
Predrag Rakić

septembar 2013

Sadržaj

Predgovor

Ova zbirka zadataka je napisana za potrebe kursa Programski prevodioci na Fakultetu tehničkih nauka, Univerziteta u Novom Sadu. Namera zbirke je da prenese konkretne tehnike i alate koji se koriste za implementaciju kompajlera.

Flex i *bison* su alati namenjeni za pisanje kompajlera i interpretera, iako su korisni i za mnoge aplikacije koje će zanimati i one koji se ne bave kompajlerima. Svaka aplikacija koja traži obrasce (*pattern*) u svom ulazu ili ima ulazni ili komandni jezik je dobar kandidat za *flex* i *bison*. Osim toga, oni omogućavaju brzu izradu prototipa aplikacije, lake modifikacije i jednostavno održavanje programa.

Kompletni primeri su numerisani, sa naznačenim imenom datoteke u kojoj se nalaze. `Neproporcionalnim fontom`, radi bolje preglednosti, navedeni su primeri, pokretanja programa i gramatika mC jezika.

Svako poglavlje, na kraju, sadrži vežbe, koje nude dodatni rad za one koji to žele.

Pregled

Poglavlje 1, *Uvod*, pruža istorijski pregled alata *lex* i *yacc*, kao i njihovih potomaka *flex* i *bison*, i daje pregled kako i u kojim situacijama se koriste *flex* i *bison*.

Poglavlje2, *Flex*, detaljno opisuje način upotrebe *flex-a* kroz nekoliko primera.

Poglavlje3, *Bison*, daje potpune primere koji koriste i *flex* i *bison*.

Poglavlje 4, *Konflikti i njihovo razrešavanje*, objašnjava dvosmislenosti i konflikte u *bison-u*, i probleme u gramatici koji onemogućuju *bison* da napravi parser. Predstavljene su metode za lociranje i razrešavanje konflikata.

Poglavlje 5, *Rukovanje greškama i oporavak*, opisuje tehnike za lociranje, prepoznavanje i prijavljivanje grešaka u ulaznoj datoteci.

Rečnik sadrži tehničke pojmove iz teorije kompajlera.

Pretpostavlja se da je čitalac familijaran sa programskim jezikom C, jer su svi primeri na C-u, *flex-u* i *bison-u*.

Kako preuzeti *flex* i *bison*

Flex i *bison* su savremene zamene za klasične alate *lex* i *yacc*.

GNU Project od strane Free Software Foundation zajednice distribuira *bison*, i on je uključen u sve uobičajene distribucije Linux-*a*, ali ako želite aktuelnu (poslednju) verziju, možete je naći na *web* stranici:

`http://www.gnu.org/software/bison/`

BSD i GNU Project takođe distribuiraju *flex* (*Fast Lexical Analyzer Generator*), u uobičajenim distribucijama Linux-a, ali ako želite poslednju verziju, možete je naći na *web* stranici:

`http://flex.sourceforge.net/`

Zahvalnica

Autori žele da se zahvale studentima koji su imali u rukama *draft* verziju zbirke zadataka na kursu Programski prevodioci i koji su našli brojne slovne i druge greške u prvim verzijama materijala, i učinili ovaj materijal čitljivijim i zanimljivijim.

Autori

Poglavlje 1

Uvod

Flex i *bison* su alati za pravljenje programa koji procesiraju strukturiran ulaz. Izvorno, to su alati za pravljenje kompajlera, ali su se dokazali kao veoma korisni i u mnogim drugim oblastima. Evo nekoliko primera za šta se mogu koristiti *flex* i *bison* (ili njihovi preci *lex* i *yacc*), da razviju [5]:

- *desktop* kalkulator
- domenske jezike za određenu aplikaciju
- alate za preprocesiranje raznih dokumenata
- programske kompajlere, npr. C kompajler
- sam *flex*

U ovom poglavlju ćemo se upoznati sa istorijatom ovih alata i sa osnovnim teorijskim postavkama koje stoje iza ovih alata.

1.1 Istorijat

Bison je potomak programa *yacc* [3], generatora parsera. Napravio ga je Stiven C. Johnson iz Bell Labs, u periodu između 1975 i 1978. Kao što samo ime *yacc* kaže (skraćeno od *yet another compiler compiler* - još jedan kompajler kompajlera), u to vreme su mnogi ljudi pisali generatore parsera. Johnson je svoj alat zasnovao na čvrstoj teorijskoj osnovi i na radu D. E. Knuth-a, koja je proizvela izuzetno pouzdane parsere. Ova pouzdanost je povećala *bison-ovu* popularnost među korisnicima UNIX sistema, iako je restriktivna licenca, pod kojom je Unix distribuiran u to vreme, ograničavala njegovo korišćenje izvan univerziteta i Bell laboratorija.

Oko 1990, Robert Corbett, postdiplomac na Univerzitetu u Kaliforniji, na Berkliju, je reimplementirao *yacc* koristeći donekle naprednije algoritme i razvio Berkli *yacc* (*Byacc*, *Berkeley yacc*)[2]. Pošto je njegova verzija bila brža od Bell-ovog *yacc*-a, a i deljen je pod fleksibilnijom Berkli licencom, brzo je postao najpopularnija verzija *yacc*-a. Richard Stalman, iz FSF (Free Software Foundation), je adaptirao Corbett-ov rad za korišćenje u GNU projektu. Alat je proširen mnogobrojnim novim karakteristikama i evoluirao je u trenutnu verziju *bison*-a. *Bison* je sada projekat FSF-a i distribuira se pod GNU javnom licencom (*GNU Public Licence*) [7].

1975 godine, Majk Lesk i pripravnik Eric Schmidt su napisali *lex*, generator leksičkih analizatora [4]. Videli su ga i kao samostalni alat i kao pratilac *yacc*-a. *Lex* je takođe postao veoma popularan, uprkos tome što je relativno spor i bagovit.

Oko 1987, Vern Paxson iz Lawrence Berkeley Lab je uzeo verziju *lex*-a pisanog na proširenom Fortran jeziku (popularnom u to vreme), preveo ga na C i nazvao ga *flex* (skraćeno od *fast lexical analyzer generator*) [6]. Pošto je bio brži i pouzdaniji od AT&T verzije *lex*-a i dostupan pod Berkli licencom, potpuno je zamenio originalni *lex*. *Flex* je sada *SourceForge* projekat, još uvek pod Berkli licencom.

1.2 Leksička analiza i sintaksna analiza

Najraniji kompajleri, tokom 1950-tih godina, koristili su potpuno *ad hoc* tehnike za analizu sintakse izvornog koda. Tokom 1960-ih, ovo polje je dobilo mnogo pažnje akademske zajednice, pa je ranih 1970-ih sintaksna analiza postala već dobro istraženo polje.

Jedan od ključnih uvida bio je da se posao prevođenja podeli na dva dela: leksička analiza (koja se naziva još i skeniranje) i sintaksna analiza (ili parsiranje).

Grubo govoreći, skeniranje deli ulaz na atomske delove, koji se zovu još i tokeni, a parsiranje analizira kako su ti tokeni međusobno povezani. Na primer, sledeći deo C koda:

```
a = b + c;
```

skener deli u tokene: `a`, `znak jednakosti`, `b`, `znak plus`, `c` i `znak tačka-zarez`. Zatim parser utvrđuje da je `b + c` izraz, a da se izraz dodeljuje promenljivoj `a`.

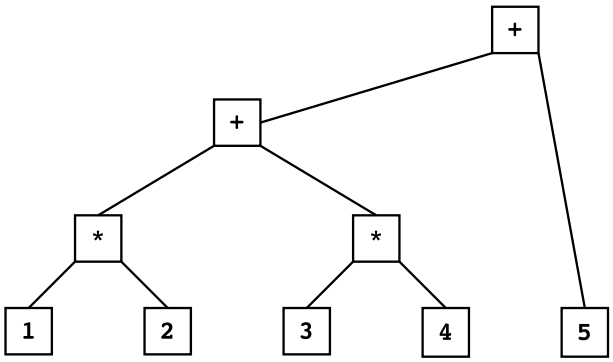
1.2.1 Skeniranje i regularni izrazi

Skeneri rade tako što traže šablone karaktera u ulazu. Na primer, u C programu, celobrojna konstanta je niz od jedne ili više cifara, ime promenljive je slovo iza kog sledi nula ili više slova ili cifara, dok su razni operatori pojedinačni karakter ili parovi karaktera. Ovi šabloni se mogu na jednostavan način opisati regularnim izrazima (*regular expressions*), koji se često skraćeno zovu *regex* ili *regexp*.

Flex program (*flex* specifikacija) se u osnovi sastoji od liste regularnih izraza. Uz svaki regualrni izraz stoji uputstvo o tome šta skener treba da radi kada ulazni string odgovara tom regularnom izrazu. Ova uputstva se zovu akcije. Skener koji se izgeneriše iz *flex-a* čita ulazni tekst s leva na desno i pokušava da prepozna neki od regularnih izraza, a u slučaju prepoznavanja vrši odgovarajuću akciju. *Flex* sve regularne izraze prevodi u efikasnu unutrašnju formu[1] koja omogućava da se istovremeno proverava slaganje ulaza sa svim obrascima. Zato je *flex* jednako brz za 100 obrazaca kao i za jedan.

1.2.2 Parsiranje i gramatike

Zadatak parsera je da shvati povezanost između ulaznih tokena [5, 1]. Uobičajen način da se takva veza prikaže je stablo parsiranja (*parse tree*). Na primer, uz uobičajena airtmetička pravila, aritmetički izraz `1 * 2 + 3 * 4 + 5` bi imao stablo parsiranja kao na slici 1.1.



Slika 1.1: Stablo parisiranja za izraz `1*2+3*4+5`

[1] Unutrašnja forma je deterministički konačni automat (*deterministic finite automation*, DFA)

Množenje ima veći prioritet od sabiranja, pa su prva dva izraza `1 * 2` i `3 * 4` listovi tog stabla. Onda se ta dva izraza sabiraju, pa se na tu sumu dodaje `5`. Svaka grana stabla pokazuje odnos između tokena ili podstabla ispod nje. Struktura ovog konkretnog stabla je vrlo jednostavna. *Bison* pravi stablo parsiranja dok parsira ulazni tekst. Ovo stablo je samo implicitno, u sekvenci operacija koje parser vrši, ali sam korisnik može eksplicitno kreirati stablo, kao strukturu podataka u memoriji.

1.2.2.1 BNF gramatike

Da bi se napisao parser, potrebno je na neki način opisati pravila po kojima će parser niz tokena pretvoriti u stablo parsiranja. Uobičajena vrsta jezika koju koriste parseri je kontekstno nezavisna gramatika (*context-free grammar*, CFG). Gramatika je skup pravila za pisanje programa na nekom jeziku. Gramatika opisuje sintaksu jezika, ali ne i semantiku. Gramatika se sastoji od pravila, simbola (*terminals*) i pojmova (*nonterminals*). Simbole prepoznaje skener, dok pojmove i ispravnost redosleda pojmova prepoznaje parser. Standardna forma u kojoj se pišu CFG gramatike je Bakus-Naurova forma (*Backus-Naur Form*, BNF). U nastavku je dat BNF zapis aritmetičkih izraza koji su korišteni u primeru, na slici 1.1:

```
      exp ::= factor
            | exp + factor

   factor ::= NUMBER
            | factor * NUMBER
```

Svaka linija predstavlja pravilo koja opisuje kako se može napraviti jedna grana stabla parsiranja. U BNF formi, `::=` se čita kao "je" ili "postaje", a uspravna crta "|" se čita kao "ili". Ime sa leve strane pravila (sa leve strane znaka `::=`) je pojam. Po konvenciji, svi tokeni se smatraju simbolima. BNF pravilo može biti rekurzivno, što znači da se ime sa leve strane pravila definiše pomoću samog sebe (pa se isto ime pojavljuje i na desnoj strani pravila). U primeru su oba pravila, i `exp` i `factor`, definisana rekurzivno.

Flex i *bison* nasuprot ručno pisanim skenerima i parserima

Tehnika prepoznavanja obrazaca (šablona, paterna) koju koristi *flex* je veoma brza i obično je iste brzine kao ručno napisan skener. Za kompleksnije

skenere, sa puno obrazaca, *flex* skener je verovatno brži, jer ručno napisani skener verovarno ima puno poređenja karaktera, dok *flex* uvek radi jedno. *Flex* verzija skenera je sigurno mnogo kraća nego ekvivalentni C skener, što olakšava pronalaženje grešaka. Uopšteno, ako se pravila za podelu ulaznog teksta u tokene može opisati regularnim izrazima, *flex* je pravi izbor.

Takođe, *bison* parser je mnogo kraći i lakše se otkrivaju greške nego u ekvivalentnom ručno pisanom parseru, naročito zato što *bison* radi verifikaciju nedvosmilenosti gramatike.

Poglavlje 2

Flex

Leksički analizator (skener) je program namenjen za leksičku analizu (skeniranje) teksta. Skener radi tako što čita tekst, karakter po karakter, i pokušava da prepozna neku od zadatih reči. Kada prepozna reč (sekvencu karaktera) izvrši zadatu akciju.

Skener se može napraviti (isprogramirati) ručno ili korišćenjem alata za generisanje skenera. Jedan od često korišćenih alata za generisanje skenera je program *flex*. *Flex* generiše skener na osnovu pravila zadatih u *flex* specifikaciji. Specifikaciju kreira korisnik u posebnoj datoteci koja po konvenciji ima ekstenziju `.l`. Ova datoteka sadrži pravila koja opisuju reči koje skener treba da prepozna. Pravila se zadaju u obliku regularnih izraza. Svakom regularnom izrazu moguće je pridružiti akciju (u obliku `C` koda) koja će se izvršiti kada skener prepozna dati regularni izraz. Uobičajena akcija je vraćanje oznake simbola, odnosno tokena za taj simbol. Token je numerička oznaka grupe (vrste) simbola. Na primer: token `NUMBER` označava bilo koji broj (i `2` i `359`), dok token `IF` označava jedino ključnu reč `if`.

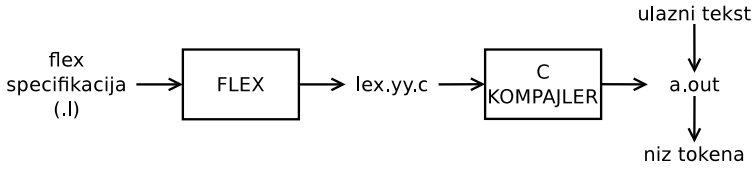
`.l` datoteka se prosleđuje *flex*-u, koji generiše skener na jeziku C, u globalnoj funkciji `yylex()` u datoteci `lex.yy.c`. Akcije, pridružene regularnim izrazima u `.l` datoteci, su delovi `C` koda koji se direktno prenose (kopiraju) u `lex.yy.c`. Ovako generisan C kod može da se prevede i C i C++ kompajlerom (tj. može da se uključi u C ili u C++ program). Ako se za generisanje skenera, umesto programa *flex* pozove program *flex++*[1], skener biva generisan u C++ funkciji `yylex()`, članici klase `yyFlexLexer` ili članici

[1] U pitanju je isti program, samo se drugačije ponaša kada mu se ime završava znakom '+'.

klase proizvoljnog imena koja nasleđuje klasu `yyFlexLexer`. Ovako generisan skener može da se uključi samo u C++ program.

Ovakav program predstavlja leksički analizator koji transformiše ulazni tekst u niz tokena (u skladu sa pravilima zadatim u specifikaciji).

Korišćenje programa *flex* je prikazano na slici 2.1:



Slika 2.1: Korišćenje *flex*-a

Prilikom izvršavanja, skener će tražiti pojavu stringa u ulaznom tekstu koji odgovara nekom regularnom izrazu. Ako ga pronađe, izvršiće akciju (kod) koju je korisnik pridružio tom regularnom izrazu. U suprotnom, podrazumevana akcija za tekst koji ne odgovara ni jednom regularnom izrazu je kopiranje teksta na standardni izlaz.

2.1 Format *flex* specifikacije

Specifikacioni *flex* datoteka se sastoji od tri dela, međusobno razdvojena oznakom `%%`:

1. definicije
2. pravila i
3. korisnički kod.

2.1.1 Definicije

Prvi deo se obično koristi za uključivanje zaglavlja i definicije promenljivih (pri čemu ovakav kod mora biti okružen specijalnim zagradama `%{` i `%}`). U ovom delu se navode još i regularne definicije, *flex* makroi, početni uslovi, kao i opcije za upravljanje radom generatora. Ovaj deo *flex* specifikacije nije obavezan.

flex nudi razne opcije za pravljenje skenera. Većina se može navesti u oliku

`%option name` u prvom delu specifikacije, ili kao

`--name` u komandnoj liniji prilikom poziva *flex*-a.

Da bi se neka opcija isključila, pre imena opcije treba dodati string "`no`", kao na primer:

`%option noyywrap` ili

`--noyywrap`

Opcije koje ćemo koristiti su:

`%option noyywrap` (–`noyywrap`) Ovom opcijom korisnik kaže *flex*-u da ne želi da definiše funkciju `yywrap()`. Ovu funkciju poziva skener kada dođe do kraja ulazne datoteke. U njoj korisnik može da opiše ponašanje skenera posle čitanja datoteke (npr. da nastavi čitanje neke druge datoteke). Ako funkcija `yywrap()` nije definisana, podrazumeva se da će skener preuzeti i čitati samo jednu ulaznu datoteku i da će nakon čitanja kraja datoteke vratiti pozivaocu vrednost `0`.

`%option yylineno (--yylineno)` Zgodno je znati broj skenirane linije u ulaznoj datoteci kada, na primer, treba prijaviti leksičku grešku. *flex* definiše promenljivu `yylineno` koja sadrži broj trenutne linije i automatski je ažurira svaki put kada pročita karakter `\n`. Skener ne inicijalizuje ovu promenljivu, pa joj zato korisnik sam mora dodeliti vrednost `1` svaki put kada započinje čitanje ulazne datoteke.

2.1.2 Pravila

Drugi deo specifikacije sadrži niz pravila koja opisuju ponašanje skenera. Na osnovu ovih pravila *flex* generiše kod skenera. Svako pravilo se zadaje u obliku:

```
obrazac   [akcija]
```

Obrasci se zadaju korišćenjem proširenog skupa regularnih izraza (datog u nastavku), a akcije blokom C (ili C++) koda koji će se izvršiti kada skener prepozna obrazac. Obrazac mora da počne u prvoj koloni, a akcija ne mora da postoji. Ako akcija postoji, njena definicija mora da počne u istom redu u kojem je definisan obrazac. Ovaj deo *flex* specifikacije je obavezan.

2.1.2.1 *flex* regularni izrazi

U tabeli 2.1 su dati operatori koji se koriste za pisanje regularnih izraza u *flex*-u.

Da bi karakter koji predstavlja *flex* metasimbol mogao da ima i svoje prirodno značenje, ispred njega treba staviti kosu crtu (\) ili ga navesti pod navodnicima (“”). Tako, na primer, regularni izraz `a\.b` opisuje string `”a.b”` (tačka se tretira kao običan karakter), dok regularni izraz `a.b` opisuje string `”a”` iza kojeg sledi bilo koji karakter osim `newline` iza kojeg sledi karakter “`b`” (tačka se tretira kao operator).

REGULARNI IZRAZ	ZNAČENJE
`x`	karakter `x`
`"x"`	`x`, čak i ako je `x` operator
`\x`	`x`, čak i ako je `x` operator
`x?`	opciono `x` (0 ili 1 instanca)
`x*`	0 ili više instanci `x`
`x+`	1 ili više instanci `x`
`x|y`	`x` ili `y`
`[xy]`	`x` ili `y`
`[x-z]`	karakteri `x`, `y` ili `z`
`[^x]`	bilo koji karakter osim `x`
`.`	bilo koji karakter osim `newline`
`^x`	`x` na početku linije
`x$`	`x` na kraju linije
`(x)`	`x`
`x/y`	`x` ali ako i samo ako iza njega sledi `y`
`{xx}`	pravilo za `xx` iz prvog dela specifikacije
`x{m,n}`	`m` do `n` pojava `x`

Tabela 2.1: Regularni izrazi u *flex*-u

2.1.3 Korisnički kod

U ovom opcionom delu korisnik može da definiše proizvoljan blok koda (funkcija `main()`, druge funkcije, globalne promenljive, ...) koji će biti bez izmena prekopiran na kraj generisanog skenera.

Ovaj deo datoteke omogućava da se ceo program smesti u samo jednu (`.l`) datoteku. Kompleksni programi se svakako (zbog čitljivosti) raspoređuju u

više izvornih datoteka, pa tada ovaj deo, uglavnom, i ne postoji.

2.1.4 Komentari

U *flex*-u se koriste komentari iz C programskog jezika:

`//` jednolinijski komentar

`/*...*/` višelinijski komentar

Komentari se u *flex*-u pišu uvučeni (indentovani) bar za jedno prazno mesto.

2.2 *flex* skeneri

U ovom delu su prikazane osnovne karakteristike programa *flex* kroz nekoliko jednostavnih primera skenera. Primeri su implementirani na jeziku C.

2.2.1 Bez ijednog pravila

Najjednostavnija *flex* specifikacija (ovde nazvana `zero.l`) je prikazana u listingu 2.1:

Listing 2.1: `zero.l`

```
%%
```

Ova datoteka sadrži samo oznaku `%%`. Skener izgenerisan iz ove specifikacije (koja ne sadrži nijedno pravilo) prihvata karaktere sa standardnog ulaza i kopira ih na standardni izlaz (karakter po karakter). Iz ovog jednostavnog primera se vidi da će svi karakteri iz ulaznog teksta, koji ne budu prepoznati (koji se ne uklapaju ni u jedan od zadatih obrazaca), biti ispisani na standardnom izlazu.

Skener se generiše, kompajlira i pokreće naredbama:

```
$ flex --noyywrap zero.l
$ gcc -o zero lex.yy.c -l l
$ ./zero
```

U prvoj liniji pokrećemo *flex* koji preuzima specifikaciju iz datoteke `zero.l` i generiše skener u datoteci `lex.yy.c`.

Svič `--noyywrap`, prosleđen *flex*-u, kaže da korisnik ne želi da definiše funkciju `yywrap()`.

U drugoj liniji pozivamo `gcc` kompajler i prosleđujemo mu `lex.yy.c` datoteku sa skenerom u funkciji `yylex()`.

Svič `--o` prosleđen `gcc`-u omogućava korisniku da iza sviča navede ime programa koji će biti izgenerisan. Ukoliko se ne upotrebi ovaj svič, kompajler će napraviti izvršnu datoteku sa imenom `a.out`.

Uz program *flex* dolazi i biblioteka `libl.a`. U ovoj biblioteci je definisana funkcija `main()`, koja jedino poziva skener, odnosno funkciju `yylex()`:

```
int main() {
   while (yylex() != 0) ;
   return 0;
}
```

Ako u programu nije definisana funkcija `main()` koristi se ona iz biblioteke. Svič `-l` govori linkeru da pogleda u biblioteku čije ime je navedeno iza sviča. Tom prilikom se iz imena biblioteke izostavljaju prva tri slova: '`lib`' (jer se ona podrazumevaju[2]) i ekstenzija. Otuda, "`-l l`" u pozivu `gcc`-a.

U trećoj liniji naredbom `./zero` se izgenerisani skener pokreće u interaktivnom režimu, što znači da se ulaz očekuje preko tastature, a da se program završava kada se na ulazu pavi kombinacija `^D`.

Ukoliko želimo skeneru da prosledimo datoteku (`test.txt`) na skeniranje, upotrebićemo redirekciju standardnog ulaza:

```
$ ./zero <test.txt
```

A ako pokrenemo skener u interaktivnom režimu, to može da izgleda ovako:

```
$ ./zero
tekst
tekst
se prekopira
se prekopira
na standardni izlaz
na standardni izlaz
^D
$
```

[2]Konvencija je da nazivi svih biblioteka počinju slovima '`lib`'.

2.2.2 Obrasci

Obrasci opisuju sekvence (nizove) karaktera koje skener treba da prepozna. Kada je potrebno prepoznati samo jednu sekvencu karaktera (kao na primer ključnu reč `if` ili kao u listingu 2.2 reč **“njam”**), tu sekvencu je moguće eksplicitno zadati.

Ako se prethodno prikazan *flex* datoteka `zero.l` preimenuje u `njam.l` i proširi jednim pravilom koje prepoznaje reč **“njam”**, dobije se *flex* datoteka prikazan na listingu 2.2. Kada se prepozna, ova reč će biti ignorisana.

Listing 2.2: `njam.l`

```
%%

njam
```

Ako se zatim, ponovo izgeneriše, kompajlira i pokrene, program `njam` se ponaša sličano programu `zero` - većinu karaktera koje primi na ulazu on prosledi na izlaz. Ako se na ulazu pojavi niz karaktera **“njam”** oni neće biti prosleđeni na izlaz, već će nestati (“poješće” ih skener - otuda ‘njam‘). Jedna interaktivna sesija programa `njam` izgleda ovako:

```
$ ./njam
Ovo prolazi bez izmena.
Ovo prolazi bez izmena.
njam

Cunjam ovuda.
Cu ovuda.
```

Kada je potrebno prepoznati sve sekvence karaktera iz neke klase (kao na primer: “broj, kao niz od jedne ili više cifara”), tada se za specifikaciju obrasca koriste složeniji izrazi, iz proširenog skupa regularnih izraza. Sledi primer (listing 2.3) specifikacije skenera koji prepoznaje prirodne brojeve.

Listing 2.3: `num.l`

```
%%

[0-9]+
```

Uglaste zagrade opisuju klasu karaktera – bilo koji karakter u rasponu od ASCII karaktera 0 do ASCII karaktera 9 (tj. bilo koja cifra), a operator “+” kaže da se cifra može ponavljati jednom ili više puta (bar jedna mora postojati).

Izlaz ovog programa može da izgleda ovako:

```
$ ./num
1 i jedan jesu 2.
 i jedan jesu .
9 + 11 = 20
 +  =
```

2.2.3 Akcije

Akcija je `C` (ili `C++`) kod koji skener izvršava kada prepozna string koji odgovaraju obrascu.

Na listingu 2.4 je prikazan skener sa samo jednim pravilom koji prepoznaje beline (proizvoljno dugačke sekvence praznih mesta i tabova) i zamenjuje ih samo jednim praznim mestom.

Listing 2.4: `ws.l`

```
%%

[ \t]+      { putchar('␣'); }
```

Obrazac je regularni izraz koji opisuje: "ili razmak ili tab jednom ili više puta".

Akcija je poziv C funkcije koja ispisuje jedan karakter - jedno prazno mesto na standardni izlaz. Kako se ova akcija izvršava svaki put kada skener prepozna niz razmaka i/ili tabova, dobija se efekat zamene belina jednim praznim mestom.

Upotreba ovog programa:

```
$ ./ws
Neko pise   reci odvojene  jednim razmakom   a neko      ne.
Neko pise reci odvojene jednim razmakom a neko ne.
^D
$
```

Na listingu 2.5 je prikazan skener sa samo jednim pravilom, koji prepoznaje karaktere za novi red (`newline`) i broji ih.

Listing 2.5: `lines.l`

```
%{
   int lines = 0;
%}

%%
```

```
\n      { lines++; }

%%

main() {
   yylex();
   printf("\nlines: %d.\n", lines);
}
```

U prvom delu specifikacije je definisana brojačka promenljiva `lines` koja će čuvati broj linija i inicijalizovana je na vrednost `0`. Promenljiva `lines` se inkrementira unutar akcije, tj. svaki put kada se prepozna karakter za novi red, pa se dobija efekat brojanja linija.

U trećem delu je navedena definicija funkcije `main`, koja sada ne poziva samo skener, tj. funkciju `yylex()` (kao što to radi bibliotečka implementacija funkcije `main`()), već nakon toga još i ispiše broj redova u pročitanom tekstu. Ako ovom programu prosledimo ulaznu datoteku `test.txt`, sa sadržajem:

Listing 2.6: `test.txt`

```
If I can dream it,
I can live it.
```

izlaz će biti:

```
$ ./lines <test.txt
If I can dream it,I can live it.
lines: 2.
$
```

Prethodni primer se može proširiti pravilom koji prepoznaje reči i ispisuje one reči koje počinju velikim slovom (listing 2.7).

Listing 2.7: `words.l`

```
[a-zA-Z]+   { if(isupper(yytext[0]))
                 printf("%s\n", yytext); }
.           { /*do nothing*/ }
```

Obrazac za reč opisuje stringove "ili malo ili veliko slovo jednom ili više puta". Svaki put kada se prepozna reč, izvršava se akcija koja je vezana za taj obrazac. Provera da li je prvo slovo prosleđenog stringa veliko se vrši pomoću funkcije `isupper()`. String prepoznate reči se nalazi u *flex*-ovoj globalnoj promenljivoj `yytext`. Ova promenljiva je tipa `char*` i u svakom trenutku sadrži string poslednjeg prepoznatog obrasca.

U prethodnim primerima smo videli da se svi neprepoznati karakteri sa ulaza prosleđuju na izlaz. Ukoliko ne želimo da se, pored reči sa prvim velikim slovom, ispisuju i ostali (neprepoznati) simboli, potrebno je dodati pravilo koje prepoznaje bilo koji karakter (operator `.`) i nema nikakvu akciju.

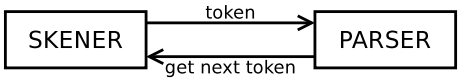
Ukoliko se ovakav program pokrene sa ulaznom datotekom `test.txt` iz prethodnog primera, dobiće se izlaz:

```
$ ./words <test.txt
If
I
I
lines: 2.
$
```

2.2.4 Upotreba skenera sa parserom

Skener, kreiran pomoću *flex*-a, u kombinaciji sa parserom se ponaša na sledeći način: kada parser aktivira leksičku analizu, skener počinje sa čitanjem preostalog ulaznog teksta, jedan po jedan karakter, dok ne nađe najduži niz karaktera koji odgovara nekom obrascu. Tada, izvršava akciju koja je pridružena prepoznatom obrascu. Akcija može da vrati kontrolu parseru ukoliko se u njoj izvrši naredba `return`. Ako do toga ne dođe, leksički analizator nastavlja da čita ostatak ulaznog teksta, dok akcija ne izazove vraćanje kontrole parseru. Ponavljanje traženja stringova dok se kontrola eksplicitno ne vrati parseru, olakšava skeneru obradu praznina i komentara.

Skener parseru ne prosleđuje npr. string prepoznatog simbola, već token za taj simbol (slika 2.2). Token je numerička oznaka kategorije (vrste) simbola (npr. token `IF` opisuje ključnu reč `if`, token `NUM` opisuje brojeve, token `ID` opisuje identifikatore odnosno imena).



Slika 2.2: Komunikacija između skenera i parsera

Na listingu 2.8 sledi primer prepoznavanja ključne reči `if`, prirodnih brojeva i reči. Tokeni su definisani u enumeraciji u prvom delu *flex* specifikacije.

Listing 2.8: `return.l`

```
%{
   enum { IF = 1, NUM, WORD };
```

```
%}

%%

if          { return IF; }
[0-9]+      { return NUM; }
[a-zA-Z]+   { return WORD; }

%%

int main() {
    int tok;
    while(tok = yylex()) {
        switch(tok) {
            case IF   : printf("IF"); break;
            case NUM  : printf("NUM"); break;
            case WORD : printf("WORD"); break;
        }
    }
}
```

Svaka od akcija vraća odgovarajući token prepoznatog simbola. U praksi, funkciju skenera najčešće poziva parser, dok u ovom primeru to radi funkcija `main()`. Kada joj skener isporuči token, ona ga ispiše na ekran.

Primer pokretanja ovog programa:

```
$ ./return
if 123 abc 5abc !
IF NUM WORD NUMWORD !
$
```

Za svaki pojedinačni string skener vraća odgovarajući token (za string `if` vraća token `IF`, za string `123` vraća `NUM`, za string `abc` token `WORD`). Za string `5abc`, vidimo da skener vraća 2 tokena: prvo za string `5` vraća token `NUM`, a zatim za string `abc` vraća token `WORD`. Skener je, krenuvši od karaktera `5`, pokušao da pronađe najduži string koji se poklapa sa nekim obrascem. Našao je samo jedan obrazac koji započinje brojem (`[0-9]+`). Preuzeo je sledeći karakter `a`, ali je zaključio da se on ne uklapa u obrazac za broj, pa ga je vratio nazad na standardni ulaz - odakle ga je i preuzeo (da bi se taj karakter obradio u okviru sledećeg simbola). Skener je uspeo da prepozna broj koji se sastoji samo od 1 cifre i za njega vraća token `NUM`. Dalje nastavlja skeniranje tamo gde je stao i preuzima ponovo karakter `a`, zatim `b` i onda `c`, prepoznaje to kao reč i vraća token `WORD`. Svi neprepoznati karakteri (npr: `!`) se samo prosleđuju na standardni izlaz.

Skener parseru, osim tokena, može da vrati i neku vrednost koja preciznije

opisuje svojstva tokena. U slučajevima kada skener prepozna jednostavne reči, kao što su ključne reči (npr. `if`, `while`, ...) parseru je dovoljno proslediti (vratiti) token te ključne reči. U slučaju kada skener prepozna identifikator ili broj, parseru je, osim tokena, potrebno proslediti i konkretan string imena odnosno vrednost konkretnog broja. Ova vrednost se prosleđuje preko globalne promenljive `yylval`. Podrazumevani tip ove promenljive je `int`, a uobičajeno je da se ova promenljiva definiše kao unija (`union`), jer se za različite simbole vezuje različita vrednost (drugačijeg tipa). Unija je promenljiva koja može čuvati objekte različitih tipova i veličina, odnosno to je jedna promenljiva koja može sadržati vrednost jednog od nekoliko tipova (a programer je dužan da vodi računa o tome koji tip vrednosti se trenutno čuva u uniji).

Za prethodni primer, potrebno je u prvom delu specifikacije, definisati uniju.

Listing 2.9: `union.l` - unija

```
union {
   int n;
   char *s;
} yylval;
```

Uz token `IF` se ne prosleđuje dodatna vrednost simbola, jer se ključna reč `if` može napisati na jedan jedini način.

Uz token `NUM` se prosleđuje konkretna vrednost broja (C funkcija `atoi` prihvata string i konvertuje ga u `int`). Ova vrednost se smeša u polje `n` unije jer je ono tipa `int`.

Listing 2.10: `union.l` - pravila

```
if          { return IF; }
[0-9]+      { yylval.n = atoi(yytext); return NUM; }
[a-zA-Z]+   { yylval.s = yytext; return WORD; }
```

Uz token `WORD` se prosleđuje konkretni string reči, jer se reči mogu različito napisati. Ovaj string se smeša u polje `s` promenljive `yylval`.

Izmena funkcije `main`() bi bila u tome da ispisuje na ekran sve vrednosti koje dobije od skenera: tokene i vrednosti koje su prosleđene uz token (listing 2.11).

Listing 2.11: `union.l` - `main()`

```
        case IF   : printf("IF"); break;
        case NUM  : printf("NUM:␣%d", yylval.n); break;
        case WORD : printf("WORD:␣%s", yylval.s); break;
```

2.3 Primeri

2.3.1 Broj reči, karaktera i redova u tekstu

Napraviti skener koji preuzme sve karaktere sa ulaza i ispiše:

- koliko je bilo reči
- koliko je bilo karaktera
- koliko je bilo redova.

Listing 2.12: `wclc.l`

```
%option noyywrap

%{
   int chars = 0;
   int words = 0;
   int lines = 0;
%}

%%

[a-zA-Z]+   { words++; chars += strlen(yytext); }
\n          { chars++; lines++; }
.           { chars++; }

%%

main() {
   yylex();
   printf("chars: %d, words: %d, lines: %d.\n",
           chars, words, lines);
}
```

Prvi deo specifikacije sadrži opciju za isključivanje funkcije `yywrap`. Isti efekat smo u ranijim primerima postizali navođenjem sviča `--noyywrap` u komandnoj liniji prilikom poziva *flex*-a.

U prvom delu specifikacije se nalaze i definicije 3 promenljive koje će čuvati broj karaktera, reči i linija.

Prvi obrazac `[a-zA-Z]+` opisuje reč. Karakteri u zagradama opisuju jedno malo ili veliko slovo, a karakter `+` opisuje jednu ili više pojava slova. Znači, ceo obrazac opisuje niz slova, tj. reč. Akcija inkrementira broj viđenih reči i karaktera. Promenljiva `yytext` pokazuje na string iz ulaznog teksta koji je prepoznat na osnovu tog obrasca. U ovom slučaju, ne zanima nas koji je to string, već koliko karaktera ima u njemu.

Drugi obrazac `\n` opisuje karakter za novi red, a pridružena akcija inkrementira broj karaktera i broj redova.

Treći obrazac je regularni izraz koji opisuje bilo koji karakter osim karaktera za novi red. Akcija inkrementira broj karaktera.

`main` funkcija poziva skener, tj. funkciju `yylex()` i ispisuje vrednosti 3 brojača. Ako skeneru ne damo ulaznu datoteku, on će ulazni tekst čitati sa tastature:

```
$ flex wclc.l
$ gcc lex.yy.c
$ ./a.out
Eci peci pec
ti si mali zec
^D
chars: 28, words: 7, lines: 2.
$
```

Prvo kažemo *flex*-u da prevede skener na `C`, zatim kompajliramo `lex.yy.c` (`C` program koji je izgenerisan), pokrenemo program i otkucamo mali ulaz za skener. Izgleda da radi.

Neki ozbiljniji program, koji broji reči i karaktere, bi imao malo drugačiju definiciju reči: niz karaktera koji nisu belina, pa bi *flex* pravilo moglo da izgleda ovako:

```
[^ \t\n\r\f\v]+   { words++; chars += strlen(yytext); }
```

Operator `^` na početku zagrada znači "prepoznaj bilo koji karakter osim onih koji su navedeni u zagradama". Ovo pokazuje snagu i fleksibilnost *flex*-a: jednostavno napravite izmene u obrascu, a zatim pustite *flex* da brine o izmenama u izgenerisanom kodu skenera.

2.3.2 Ispravan kraj rečenice

Napraviti skener koji prepravlja ulazni tekst tako da ne dozvoljava beline pre tačke, a ubacuje samo jedan razmak posle tačke.

Listing 2.13: `wsdot.l`

```
%%

[ \t\n]*"."[ \t\n]*   { printf(".␣"); }
.                     { ECHO; }
```

Ovaj skener treba da reaguje na beline pre i posle tačke, a sve ostale karaktere da "propušta" na izlaz. Zato pravimo dva pravila. Prvo pravilo opisuje prepoznavanje 0 ili više belina pre tačke, iza kojih sledi tačka, iza koje sledi niz od 0 ili više belina. Akcija koja se izvršava u slučaju da se pronađe tačka okružena belinama je da se ceo string zameni stringom ". ", čime zadovoljavamo zahtev zadatka da posle tačke treba da stoji samo jedno prazno mesto.

Drugo pravilo kaže da svi karakteri, koji ne odgovaraju prvom pravilu, treba da prođu na izlaz neizmenjeni. To se dobije akcijom koja radi eho ulaznog stringa na izlaz (`ECHO`). Ovaj program bi se ponašao isto i da smo izostavili drugo pravilo, jer *flex* ima predefinisano ponašanje, da svaki karakter, koji se ne uklapa ni u jedno pravilo, ispiše na izlaz.

Primer pokretanja ovog programa:

```
$ ./wsdot
Prva recenica .Druga recenica.   Treca recenica.Cetvrta    .
Prva recenica. Druga recenica. Treca recenica. Cetvrta.
^D
$
```

2.3.3 Izostavljanje komentara iz programa

Napraviti skener koji briše komentare iz C i C++ programa.

Listing 2.14: `comments.l`

```
%x COMMENT

%%

    /* Blok komentari */
"/*"            { BEGIN COMMENT;
                  /* predji u stanje COMMENT */ }

<COMMENT>.|\n   { /* preskoci tekst komentara */ }

<COMMENT>"*/"   { BEGIN INITIAL;
                  /* vrati se u normalno stanje */ }

    /* Linijski komentari */
"//".*
```

U prvom delu se nalazi i definicija (ekskluzivnog) stanja `%x COMMENT`. To znači da kada je to stanje aktivno, samo obrasci koji su vezani za to stanje se mogu prepoznavati.

Prva tri pravila služe za prepoznavanje blok komentara. Prvo pravilo ulazi u stanje `COMMENT` kada prepozna `/*`, a treće pravilo se vraća nazad u normalno `INITIAL` stanje kada prepozna `*/`. Drugo pravilo prepoznaje sve što se nalazi između, odnosno tekst komentara.

Ako bismo dodali još i pravilo

```
<COMMENT><<EOF>> {
   printf("line␣%d:␣Unterminated␣comment\n", yylineno);
   return 0;
}
```

mogli bismo da detektujemo i prijavimo nezavršene komentare.

Poslednje pravilo opisuje jednolinijske komentare (dva `slash` karaktera iza kojih može da sledi proizvoljan broj karaktera).

Za ulaznu datoteku `test-comment.c`, koja sadrži jednu C funkciju:

Listing 2.15: `test-comment.c`

```
/* vraca vrednost 1 ako se vrednost prvog parametra
   nalazi u navedenim granicama. Inace vraca 0. */
int interval(int x, int lower, int upper) {
   if(x >= lower && x <= upper)
      return 1;   // vrati bool vrednost true
   else
      return 0;   // vrati bool vrednost false
}
```

program daje izlaz:

Listing 2.16: `out.c`

```
int interval(int x, int lower, int upper) {
   if(x >= lower && x <= upper)
      return 1;
   else
      return 0;
}
```

2.3.4 Pretvaranje binarnih u dekadne brojeve

Napraviti skener koji u ulaznom tekstu pronalazi binarne brojeve i prevodi ih u dekadne brojeve. Primer binarnog broja je `0B0110`.

Listing 2.17: `2to10.l`

```
%{
   unsigned val = 0;
   char *i;
%}

%%

0[bB][01]{1,32}   {  for(i = yytext+2; *i!=0; ++i) {
                        val <<= 1;
                        val += *i - '0';
                     }
                     printf("%d", val);
                     val = 0;
                  }
.                 {  ECHO;  }
```

Prvi deo specifikacije sadrži definicije dve promenljive: celobrojnu neoznačenu `val`, koja će služiti za konverziju vrednosti, i pokazivačku promenljivu `i` koja će služiti kao iterator `for` petlje.

Prvo pravilo prepoznaje binarne brojeve kao karakter `'0'` zatim malo ili veliko slovo `b`, a zatim minimalno jedna, a maksimalno 32 binarne cifre. Akcija koja prati ovo pravilo radi konverziju binarnog u dekadni broj. Iterator `for` petlje počinje konverziju od trećeg karaktera ulaznog stringa, jer preskače `"0b"`. Promenljiva `val` se resetuje nakon ispisa, da bi bila spremna za ispravnu konverziju sledećeg broja.

Drugo pravilo preusmerava sve ostale karaktere na izlaz (poziva makro `ECHO`).

Primer pokretanja bi mogao da izgleda:

```
$ ./2to10
123
123
0b01
1
0B0110
6
^D
$
```

Prvi broj `123` se ispisuje neizmenjen jer nije binarni broj.

Varijacije primera: prevesti heksa broj u dekadni.

2.3.5 Interpreter jednostavnih komandi

Napraviti skener koji čita `float` vrednosti sa ulaza, dok se na ulazu ne pojavi neka od komandi:

- reč "sum" - vraća pozivaocu sumu unetih vrednosti
- reč "mean" - vraća pozivaocu srednju vrednost unetih vrednost
- reč "quit" - dovodi do završetka programa.

Listing 2.18: `cmd.l`

```
%{
   double sum = 0;
   int count = 0;
%}

%%

[0-9]+\.[0-9]*  { sum += atof(yytext);
                  count++; }

"sum"           { printf("Sum: %f", sum);
                  sum = 0;
                  count = 0; }

"mean"          { printf("Mean: %f", sum/count);
                  sum = 0;
                  count = 0; }

"quit"          { return 0; }

.               { }
```

Prvo pravilo prepoznaje `float` brojeve kao jednu ili više cifara, iza čega sledi tačka, iza čega opciono sledi 0 ili više cifara. To znači da su tačka i cifra ispred tačke obavezni, a cifre iza tačke nisu. Akcija, dodeljena ovom pravilu, sabira vrednost broja u promenljivoj `sum`.

Drugo i treće pravilo opisuju prepoznavanje stringa komande i izvršavaju akciju: ispisuju rezultat komande i resetuju promenljive.

Četvrto pravilo vraća vredost `0`, što dovodi do kraja skeniranja i time realizuje komandu `"quit"`.

Poslednje pravilo sve ostale karaktere prosleđuje na izlaz.

Primer pokretanja programa:

```
$ ./cmd
```

```
1.1 2.2 sum
Sum: 3.300000
5.5 3.5 6. mean
Mean: 5.000000
quit
```

2.3.6 Izmena formata datuma

Napraviti skener koji procesira tekst i ako naiđe na datum u obliku `dd/mm/yyyy`, menja ga u oblik `dd-mmm-yyyy`.

Listing 2.19: `date.l`

```
%option noyywrap yylineno

%{
   char *months[] = { "", "jan", "feb", "mar", "apr", "maj",
      "jun", "jul", "avg", "sep", "okt", "nov", "dec" };
%}
%%

[0-9]{2}\/[0-9]{2}\/[0-9]{4}    { printf("%d-%s-%d",
   atoi(yytext), months[atoi(yytext+3)], atoi(yytext+6)); }
```

U prvom delu specifikacije definisan je niz stringova `months`, koji označavaju mesece u godini, tako da redni broj meseca u godini odgovara njegovom indeksu u nizu.

Pravilo za datum prepoznaje po 2 ili 4 cifre razdvojene *slash* karakterom `/`. Pošto je *slash* karakter jedan of *flex* operatora, u pravilu se piše `\/`. Kada se prepozna datum u prvobitnom obliku, ispisuje se datum u novom obliku, sa crticama. Promenljiva `yytext` sadrži ceo prepoznat string, pa se broju dana pristupa od karaktera na indeksu 0, mesecu na indeksu 3 i godini na indeksu 6. Funkcija `atoi()` vrši konverziju stringa u celobrojnu vrednost.

Evo jednog izvršavanja ovog programa:

```
$ ./date
Danas je utorak 06/11/2012 godine.
Danas je utorak 6-nov-2012 godine.
^D
$
```

2.3.7 Skeniranje teksta

Tekst se sastoji od 0 ili više rečenica. Rečenica započinje rečju na čijem početku je veliko slovo. Zatim mogu da slede reči koje započinju bilo velikim bilo malim slovom. Na kraju stoji tačka.

Skener za ovaj zadatak, znači, treba da prepozna:

- reči koje počinju velikim slovom: prvo jedno veliko slovo, a zatim 0 ili više malih slova: `[A-Z][a-z]*`
- reči koje počinju malim slovom: 1 ili više malih slova: `[a-z]+` .

U nastavku je data implementacija ovog skenera:

Listing 2.20: `text.l`

```
%option noyywrap yylineno
%{
   char* yylval;
   #define _DOT             1
   #define _CAPITAL_WORD    2
   #define _WORD            3

   static char* tokens[] = { "",
                             "_DOT␣␣␣␣␣␣␣␣␣",
                             "_CAPITAL_WORD",
                             "_WORD␣␣␣␣␣␣␣␣" };
%}

%%

[ \t\n]+        { /* skip */ }

"."             { return _DOT; }
[A-Z][a-z]*     { yylval = yytext; return _CAPITAL_WORD; }
[a-z]+          { yylval = yytext; return _WORD; }

.               { printf("\nline␣%d:␣LEXICAL␣ERROR␣on␣char␣%c",
                         yylineno, yytext[0]); }

%%

main() {
   int token = -1;
   while((token = yylex()) != 0) {
      printf("\n%16s␣TOKEN:␣%s", yytext, tokens[token]);
      if(token == _CAPITAL_WORD || token == _WORD) {
         printf("␣value:␣%s", yylval);
      }
   }
```

```
}
```

Prvi deo specifikacije sadrži opcije za isključivanje funkcije `yywrap` i za uključivanje brojanja linija `yylineno`. U nastavku su definisani tokeni sa vrednostima 1, 2 i 3. Dalje su definisani stringovi koji odgovaraju imenima tokena, i služe za štampanje izlaza programa. Vrednosti tokena odgovaraju indeksima njihovih stringova.

Prvo pravilo preskače beline. Drugo pravilo prepoznaje tačku i vraća samo token.

Treće i četvrto pravilo prepoznaju reči i vraćaju i token i string simbola kroz promenljivu `yylval`. Zato je promenljiva `yylval` u prvom delu specifikacije definisana kao `char*` .

Poslednje pravilo služi za prijavljivanje leksičke greške: ako je neki karakter dospeo u ovo pravilo, znači da nije odgovarao nijednom prethodnom pravilu, a to znači da njegova pojava nije ni predviđena.

Funkcija `main` prihvata tokene i vrednosti simbola i ispisuje ih.

Jedan primer izvršavanja ovog programa može biti:

```
$ ./a.out
Ovo je tekst. Za skeniranje.

             Ovo TOKEN: _CAPITAL_WORD value: Ovo
              je TOKEN: _WORD          value: je
           tekst TOKEN: _WORD          value: tekst
               . TOKEN: _DOT
              Za TOKEN: _CAPITAL_WORD value: Za
      skeniranje TOKEN: _WORD          value: skeniranje
               . TOKEN: _DOT
^D
$
```

2.4 Vežbe

1. Proširiti prvi primer (sa listinga 2.7), tako da posebno ispiše broj reči koje počinju velikim slovom i broj reči koje počinju malim slovom.
2. Napraviti skener (sličan primeru 2) koji prepravi ulazni tekst tako da se iza tačke uvek nalazi veliko slovo. Ispravka treba da se primeni samo u slučaju da rečenica počinje rečju, a ne, recimo, brojem. Na primer, ulaz: `Petak. to je divan dan.` treba da postane `Petak. To je divan dan.`.

3. Prepraviti primer 3, tako da menja datume iz oblika `dd/mm/yyyy` u `mmmm dd, yyyy`. Na primer, ulaz `31/12/2012` treba da se izmeni u `Decembar 31, 2012`.
4. Napraviti skener koji u ulaznom tekstu prepravlja vreme iz oblika `HH:MM` (gde su sati predstavljeni brojevima 0-24) u oblik `HH:MM AM` ili `HH:MM PM` (gde su sati predstavljeni brojevima 0-12). Na primer, ulazni tekst `16:55` treba da se izmeni u `4:55 PM`, dok ulaz `2:55` treba da se izmeni u `2:55 AM`.
5. Napraviti skener, koji prima tekst napisan na američkom engleskom jeziku, i menja neke reči na britanski engleski jezik. Reči koje treba zameniti su:

am	br
color	colour
fall	autumn
cookie	biscuit
fries	chips
freeway	motorway

Poglavlje 3

Bison

Sintaksa jezika opisuje pravila po kojima se kombinuju simboli jezika (npr. u `while` iskazu se prvo navede ključna reč `while`, zatim se u malim zagradama navodi logički izraz, a iza zagrada slede naredbe koje čine telo `while` iskaza). Sintaksa se opisuje gramatikom, obično pomoću BNF notacije. Svako pravilo se sastoji od leve i desne strane.

Sintaksna analiza ima zadatak da proveri da li je ulazni tekst sintaksno ispravan. Ona to čini tako što preuzima niz tokena od skenera i proverava da li su tokeni navedeni u ispravnom redosledu. Ako jesu, to znači da je ulazni tekst napisan u skladu sa pravilima gramatike korišćenog jezika, tj. da je sintaksno ispravan. U suprotnom, sintaksna analiza treba da prijavi sintaksnu grešku i da nastavi analizu. Opisani proces se vrši u delu kompajlera koji se zove parser i naziva se parsiranje. U toku parsiranja gradi se stablo parsiranja.

3.1 LR parseri

LR parseri spadaju u grupu *bottom-up* parsera, što znači da grade stablo parsiranja od listova ka korenu stabla. U toku parsiranja se prvo preuzimaju simboli (tokeni) sa ulaza i parser pokušava da prepozna desnu stranu nekog pravila i da ga zameni levom stranom pravila. Cilj je da se stigne do korena stabla, tj. do polaznog pojma gramatike. To je trenutak kada je parsiranje uspešno završeno.

"L" ("left") označava da se ulaz čita s **l**eva na desno, a "R" ("right") označava da se koristi **d**esno izvođenje na gore.

LR parseri koriste dve vrste akcija koje se odnose na ulazni tekst i njihov stek:

shift: simbol se pročita iz ulaznog teksta i stavi se na stek

reduce: gornjih N elemenata steka čuvaju simbole identične sa N simbola koji se nalaze sa desne strane određenog pravila. Reduce akcijom se ovih N simbola zamenjuje pojmom koji se nalazi sa leve strane tog pravila.

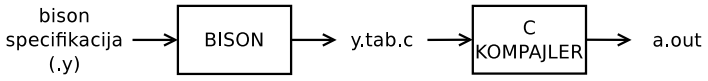
Zbog ovih akcija LR parseri se nazivaju još i *shift-reduce* parseri. Osim ove dve akcije postoje još dve koje se koriste u procesu LR parsiranja:

accept: ova akcija se izvršava kada je pročitan ceo ulazni tekst, a na steku se nalazi samo jedan, i to početni, pojam gramatike

error: ako ulazni tekst nije u skladu sa gramatikom, parser neće moći da primeni ni *shift* ni *reduce* akciju, pa će se zaustaviti i prijaviti grešku.

3.1.1 Generatori LR parsera

Jedan od najčešće korišćenih alata za generisanje parsera je *bison*. Korišćenje *bison*-a je prikazano na slici 3.1.



Slika 3.1: Korišćenje *bison*-a

Prvi korak u generisanju parsera je priprema njegove specifikacije. Specifikaciju kreira korisnik u posebnoj datoteci koja po konvenciji ima ekstenziju `.y` . Ova datoteka sadrži gramatiku koju parser treba da prepozna. Pravila se zadaju u BNF obliku. Svakom pravilu je moguće pridružiti akciju (u obliku `C` koda) koja će se izvršiti kada parser prepozna dato pravilo.

`.y` datoteka se prosleđuje *bison*`-u`, koji kao izlaz, generiše `C` program `y.tab.c`. Kod koji je izgenerisao *bison*, sadrži tabelarnu reprezentaciju dijagrama dobijenih iz opisa pravila. Akcije pridružene pravilima u `.y` datoteci su delovi `C` koda koji se direktno prenose (kopiraju) u `y.tab.c`. Na kraju, `y.tab.c` se

prosleđuje `C` kompajleru da bi se kao izlaz dobio program `a.out`. Ovaj program predstavlja parser koji proverava sintaksnu ispravnost ulaznog teksta (na osnovu pravila gramatike).

Prilikom izvršavanja, parser će od skenera tražiti naredni token iz ulaznog teksta i proveriti da li je njegova pojava na datom mestu dozvoljena (da li je u skladu sa gramatikom). Ako jeste nastaviće parsiranje, a ukoliko nije, prijaviće grešku, pokušaće da se oporavi od greške i da nastavi parsiranje. U momentu kada parser prepozna celo pravilo, parser će izvršiti akciju koja je pridružena tom pravilu.

3.2 Format *bison* specifikacije

Bison specifikacija se sastoji od tri dela, međusobno razdvojena oznakom `%%`:

1. definicije
2. pravila i
3. korisnički kod.

3.2.1 Definicije

Prvi deo se obično koristi za uključivanje zaglavlja i definicije promenljivih (pri čemu ovakav kod mora biti okružen specijalnim zagradama `%{` i `%}`). U ovom delu se definišu i tokeni tako što se ime tokena navede iza ključne reči `%token`. Uopšteno, u ovom delu se navode opcije za upravljanje radom generatora. Ovaj deo datoteke nije obavezan.

3.2.2 Pravila

Drugi deo specifikacije sadrži gramatiku, odnosno pravila koja opisuju sintaksu jezika. Gramatika svakog jezika obuhvata: simbole, pojmove, pravila i polazni pojam. Na osnovu ovih pravila *bison* generiše kod parsera. Svako pravilo se zadaje u obliku:

```
pravilo   [akcija]
```

Pravila se pišu na formalan način u BNF obliku, a akcije blokom C (ili C++) koda koji će se izvršiti kada skener prepozna pravilo. Obrazac mora da počne u prvoj koloni, a akcija ne mora da postoji. Ako akcija postoji,

njena definicija mora da počne u istom redu u kojem je definisan obrazac. Ovaj deo opisa *bison* programa je obavezan.

Pravila (*production*) određuju dozvoljene načineređanja/pisanja pojmova i simbola:

```
pojam ::= pojmovi i/ili simboli
```

Leva strana pravila (pre znaka `::=`) sadrži pojam koji može biti zamenjen sekvencom pojmova i/ili simbola koje sadrži desna strana pravila (iza znaka `::=`). To znači da se leva strana pravila može zameniti desnom stranom pravila, i obrnuto. Jedan od pojmova predstavlja polazni pojam i on se navodi kao prvo pravilo.

Na primer, `IF-ELSE` iskaz u C programskom jeziku ima formu:

```
IF ( expr ) stmt ELSE stmt
```

Drugim rečima, to je konkatenacija: ključne reči `IF`, otvorene zagrade `(`, izraza `expr`, zatvorene zagrade `)`, iskaza `stmt`, ključne reči `ELSE`, i na kraju, još jednog iskaza `stmt`.

Sintaksa `IF-ELSE` iskaza napisana BNF notacijom ima izgled (imena pojomova su napisana malim slovima, a simbola velikim):

```
stmt ::= IF ( expr ) stmt ELSE stmt
```

ili

```
stmt -> IF ( expr ) stmt ELSE stmt
```

Bison ovakvu gramatiku prihvata u malo modifikovanom obliku BNF notacije: umesto oznake `::=` ili strelice `->` piše se dvotačka i na kraju pravila se piše karakter tačka-zarez:

```
stmt : IF ( expr ) stmt ELSE stmt ;
```

Dvotačka se čita "može imati oblik". Više pravila, koja imaju istu levu stranu, na primer:

```
list : + digit ;
list : - digit ;
list : digit   ;
```

mogu se grupisati zajedno, tako što se razdvoje vertikalnom linijom koja ima značenje "ili":

```
list : + digit | - digit | digit ;
```

Ako se ovo napiše malo drugačije, postaje čitljivije:

```
list : + digit
     | - digit
     | digit
     ;
```

Ovo je uobičajeni način pisanja grupe pravila u *bison* specifikaciji.

3.2.3 Korisnički kod

U ovom opcionom delu korisnik može da definiše proizvoljan blok koda (funkcija `main()`, druge funkcije, globalne promenljive, ...) koji će, bez izmena, biti prekopiran na kraj generisanog parsera.

3.3 Parsiranje

Bison preuzima gramatiku iz specifikacije i pravi parser koji prepoznaje "rečenice", odnosno iskaze te gramatike.

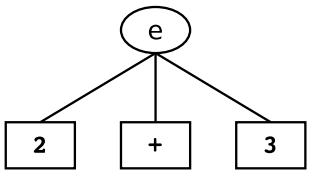
Programi mogu biti sintaksno ispravni, ali semantički neispravni (na primer, C program koji `string` dodeljuje `int` promenljivoj). *Bison* rukuje samo sintaksnom, a sve ostale validacije su ostavljene korisniku.

Postupak parsiranja će biti opisan na primeru gramatike za jednostavni kalkulator:

```
e : NUMBER PLUS NUMBER
  | NUMBER MINUS NUMBER
  ;
```

gde pojam `e` opisuje izraz (*expression*). `NUMBER`, `PLUS` i `MINUS` su simboli. Simbolu `NUMBER` odgovara broj (niz cifara), simbolu `PLUS` karakter `+`, a simbolu `MINUS` karakter `-`. Vertikalna crta (`|`) znači da za isti pojam postoje dve različite mogućnosti i opisuje se rečju ILI. U ovom slučaju, izraz može biti ili sabiranje ili oduzimanje. Svi pojmovi koji se koriste u gramatici moraju biti definisani, odnosno mora postojati bar jedno pravilo u kom se taj pojam nalazi na levoj strani. Na levoj strani pravila se ne sme naći token (to je greška).

Uobičajeni način da se predstavi isparsiran tekst je stablo. Na primer, ako se parsira ulaz: `2+3` ovom gramatikom, stablo izgleda kao na slici 3.2.



Slika 3.2: Stablo parsiranja za izraz `2+3`

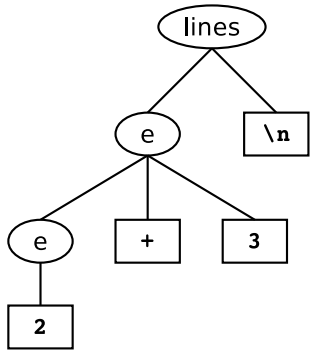
Parser ne kreira automatski ovakvo stablo kao strukturu podataka, iako je relativno jednostavno da korisnik to uradi.

Pravila mogu da pozivaju, direktno ili indirektno, sama sebe, pa se zovu rekurzivna pravila. Time je omogućeno parsiranje proizvoljno dugačkih ulaznih nizova. Ako prethodnu gramatiku malo izmenimo, dobićemo:

```
lines :
      | lines e NEWLINE
      ;

e     : e PLUS NUMBER
      | e MINUS NUMBER
      | NUMBER
      ;
```

gde `e` opisuje izraz, a `lines` opisuje (prazne redove ili) redove koji sadrže po jedan izraz (i karakter za novi red). `NUMBER`, `NEWLINE`, `PLUS` i `MINUS` su simboli, a `e` i `lines` su pojmovi. Simbolu `NEWLINE` odgovara karakter za novi red `\n`. Ako se ovom gramatikom parsira ulaz `2+3\n`, stablo izgleda kao na slici 3.3.

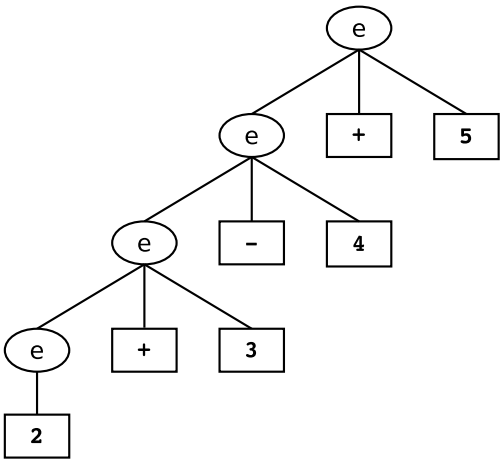


Slika 3.3: Stablo parsiranja za ulaz `2+3\n`

U ovom primeru, `2+3` je izraz, a `e NEWLINE` (ili `e\n`) je iskaz. Svaka gramatika ima početni pojam, onaj koji mora da bude koren stabla parsiranja. U ovoj gramatici početni pojam je `lines`. Da bi neki pojam bio proglašen početnim, treba da bude naveden kao prvo pravilo u *bison* specifikaciji ili da se ime početnog pojma navede iza ključne reči `%start`, u prvom delu *bison* specifikacije, na primer:

```
%start lines
```

U našem primeru pravilo `e` je sada rekurzivno definisano, i to levo rekurzivno. To znači da se ime pravila nalazi na početku (na levom kraju) desne strane pravila. Zbog ove osobine, moguće je parsirati izraz `2+3-4+5` ponovljenom primenom pravila `e` (deo stabla parsiranja koji se odnosi na ovaj ulazni izraz prikazano je na slici 3.4).



Slika 3.4: Deo stabla parsiranja za izraz `2+3-4+5`

Listing 3.1 sadrži *flex*, a listing 3.2 *bison* implementaciju prethodne gramatike: potreban je skener za prepoznavanje simbola i parser za prepoznavanje iskaza.

Listing 3.1: `calc1.l`

```
%{
   #include "calc1.tab.h"
%}

%%

[ \t]+
[0-9]+     { yylval = atoi(yytext); return NUMBER; }
```

```
"+"        { return PLUS; }
"-"        { return MINUS; }
\n         { return NEWLINE; }
.          { printf("Unknown␣char␣%c\n", *yytext); }
```

U prvom delu specifikacije potrebno je uključiti datoteku `calc1.tab.h` koju proizvodi *bison*, jer se u njoj nalaze izgenerisani tokeni koji se koriste u skeneru.

U pravilima se prepoznaje broj kao niz cifara i uz token se prosleđuje konkretna vrednost broja (podrazumevanti tip promenljive `yylval` je `int`). Za simbole `PLUS, MINUS i NEWLINE` se prosleđuje samo token. Za sve ostale karaktere prijavljuje se greška, jer gramatikom nije predviđena njihova pojava u ulaznom tekstu.

Listing 3.2: `calc1.y`

```
%{
   #include <ctype.h>
   #include <stdio.h>
   int yyparse(void);
   int yylex(void);
%}

%token NUMBER
%token PLUS
%token MINUS
%token NEWLINE

%%

lines
   :
   |  lines NEWLINE
   |  lines e NEWLINE
   ;

e
   :  e PLUS NUMBER
   |  e MINUS NUMBER
   |  NUMBER
   ;

%%

int main() {
   return yyparse();
}
```

```
int yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
```

U prvom delu su definisana 4 tokena za 4 simbola gramatike. Pravilo `lines` opisuje prazan red ili red u kojem se nalazi izraz iza kog sledi karakter za novi red. Izraz je rekurzivno definisan kao jedan broj, ili sabiranje ili oduzimanje dva broja.

`main()` funkcija jedino poziva parser. Ukoliko dođe do sintaksne greške, pozvaće se funkcija `yyerror`() koja ispisuje poruku o greški.

Generisanje skenera i parsera i pokretanje programa se odvija ovako:

```
$ bison -d calc1.y
$ flex --noyywrap calc1.l
$ gcc -o calc1 calc1.tab.c lex.yy.c
$ ./calc1
2+3
5-2
^D
$
```

Prvo se pokrene *bison* sa opcijom `-d` (za generisanje `.h` datoteke sa **d**efinicijama) koji će napraviti datoteke `calc1.tab.c` i `calc1.tab.h`. Zatim se pokrene *flex* koji napravi `lex.yy.c`. Na kraju se pozove `gcc` kompajler da napravi izvršni program `calc1`. Pokretanjem programa vidimo da kalkulator prihvata izraze, ali ne izvršava sabiranje i oduzimanje, jer pravilima još nismo pridružili akcije, tj. nismo mu rekli šta da radi kada prepozna izraz i liniju.

3.3.1 Akcije

Hajde da pravilima u prethodnoj *bison* specifikaciji dodamo akcije (listing 3.3):

Listing 3.3: `calc2.y`

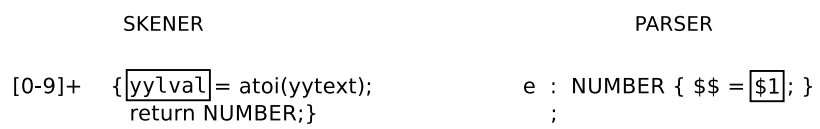
```
lines
   :
   |  lines NEWLINE
   |  lines e NEWLINE   { printf("%d\n", $2); }
   ;

e
   :  e PLUS NUMBER     { $$ = $1 + $3; }
```

```
    |   e MINUS NUMBER     { $$ = $1 - $3; }
    |   NUMBER             { $$ = $1; }
    ;
```

U akcijama se koriste meta-promenljive, čija imena počinju karakterom `'$'`, a iza nje sledi broj. Ovaj broj označava redni broj simbola ili pojma na desnoj strani pravila. Tako se meta-promenljiva `$1` odnosi na vrednost pojma/simbola koji se nalazi prvi naveden na desnoj strani pravila. Vrednost simbola je u parser "stigla" preko promenljive `yylval`, a poslao ju je skener. Vrednost pojma se definiše u parseru, u akciji pravila za dotični pojam. Meta-promenljiva `$$` se odnosi na pojam sa leve strane pravila, odnosno sadrži njegovu vrednost. To znači da, kada želimo da definišemo vrednost nekog pojma, u njegovom pravilu ćemo imati akciju `{ $$ = ... ; }`.

Krenućemo od poslednjeg pravila za izraz `e : NUMBER` kojem smo dodali akciju `{ $$ = $1; }`. Ova akcija kaže da vrednost pojma `e` dobija istu onu vrednost koju ima simbol `NUMBER`, a to je konkretna vrednost prepoznatog broja, jer smo u skeneru, uz token `NUMBER` preko `yylval`, prosledili i vrednost broja (slika 3.5).



Slika 3.5: Prenošenje vrednosti simbola iz skenera u parser

U pravilu za oduzimanje dodali smo akciju `{ $$ = $1 - $3; }`, gde ćemo oduzeti 2 broja. `$1` sadrži vrednost prvog broja (to je vrednost pojma `e`), a `$3` sadrži vrednost drugog broja (koji se nalazi na trećoj poziciji desne strane pravila, pa se njegovoj vrednosti pristupa preko meta-promenljive `$3`). Vrednosti pojma `e` (kojoj se pristupa preko meta-promenljive `$$`) se dodeljuje rezultat oduzimanja. Slično se dešava i u pravilu za sabiranje.

U pravilu za `lines` dodali smo akciju `{ printf("%d\n", $2); }` koja, nakon prepoznavanja jedne linije sa izrazom, ispisuje vrednost izraza. Ovu akciju ne treba dodati u ostala dva pravila za `lines`, jer su to pravila za prepoznavanje praznih linija, bez izraza, pa nema ni izračunavanja ni ispisivanja.

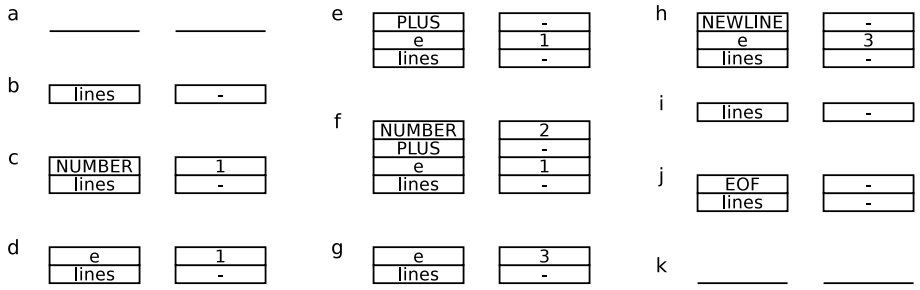
3.3.2 *Shift-reduce* parsiranje

Bison radi tako što proverava da li postoji pravilo (desna strana pravila) koje odgovara tokenima koje je do sada primio. Dok *bison* pravi parser,

on kreira skup stanja, od kojih svako odražava poziciju u pravilu koje se parsira. Dok parser čita tokene, svaki put kada pročita token kojim se još nije kompletiralo pravilo, on ga stavi na interni stek i pređe u novo stanje. Ova akcija se zove *shift*. Kada parser pronađe sve simbole koji čine desnu stranu pravila, on ih sve ukloni sa steka i umesto njih na stek stavi pojam sa leve strane pravila, a zatim pređe u novo stanje. Znači, zamenio je desnu stranu pravila na steku levom. Ova akcija se zove *reduce*, jer smanjuje broj elemenata na steku. Kad god *bison* prepozna pravilo (*reduce*), on izvrši akciju koja je pridružena tom pravilu.

Da vidimo kako će parser parsirati ulaz `1+2\n`. Parsiranje ćemo pratiti kroz izmene na steku stanja i steku vrednosti (vrhovi oba steka su gore). Na početku parsiranja, oba steka su prazna (slika 3.6.a). Pre preuzimanja prvog karaktera, dešava se redukcija po prvom pravilu, koja prazan red pretvara u `lines` (slika 3.6.b). Pošto ne može da izvrši više ni jednu redukciju, parser poziva skener i od njega dobija prvi simbol iz ulaznog teksta, a to je `NUMBER`, sa vrednošću `1`. Parser smešta simbol `NUMBER` (šiftuje ga) na stek stanja, a na stek vrednosti stavlja vrednost `1` (stek sada izgleda kao na slici 3.6.c).

Zatim, parser proverava da li neka sekvenca na vrhu steka odgovara desnoj strani nekog pravila. Konstatuje da `NUMBER` čini desnu stranu pravila za `e`, pa izvršava redukciju. Sa steka stanja uklanja `NUMBER` i menja ga pojmom `e`, a sa steka vrednosti uklanja vrednost `1` i stavlja vrednost pojma `e`. Vrednost pojma se računa u akciji koja je pridružena tom pravilu i koja se izvršava u toku redukcije. U toj akciji piše da vrednost pojma `e` (`$$`) dobija vrednost simbola `NUMBER` (`$1`), a to je vrednost `1` (vidi 3.6.d).



Slika 3.6: Primer rada parsera za ulaz `1+2\n`

Pošto ne može da izvrši više nijednu redukciju, parser se ponovo obraća skeneru za token. Dobija token `PLUS` kog šiftuje (smešta) na stek stanja (vidi 3.6.e). Uz token `PLUS` iz skenera nije poslata vrednost. U ovom trenutku elementi na vrhu steka ne formiraju nijedno pravilo, pa parser poziva skener

i dobija token `NUMBER` i vrednost `2` (vidi 3.6.f).

Sada poslednja 3 elementa na vrhu steka čine desnu stranu pravila `e` za sabiranje izraza. Parser može da uradi redukciju po tom pravilu, što znači da uklanja 3 elementa sa steka stanja i stavlja pojam `e`, a sa steka vrednosti uklanja 3 elementa i stavlja novu vrednost `3`. Ovu, novu, vrednost pojma `e` dobio je izvršavanjem akcije uz ovo pravilo, koja opisuje da se vrednost pojma `e` dobija kao zbir prvog operanda (`$1` - u našem slučaju vrednost `1`) i drugog operanda (`$3` - u našem slučaju vrednost `2`). Vrednosti simbola `NUMBER` se pristupa preko meta-promenljive `$3`, jer se simbol nalazi na trećem mestu na desnoj strani pravila, a istovremeno je treći element koji je skinut sa steka (vidi 3.6.g).

Parser dalje šiftuje simbol `NEWLINE` (vidi 3.6.h) i sada je u mogućnosti da uradi redukciju po pravilu `lines : lines e NEWLINE`. U procesu redukcije uklanja po 3 elementa sa svakog steka i stavlja pojam `lines` na steku stanja (vidi 3.6.i).

Nakon toga, šiftuje `EOF` karakter, jer je stigao do kraja datoteke i u prilici je da uradi *accept* akciju. Ova akcija je moguća u trenutku kada se na steku nalazi samo polazni pojam, a iza njega `EOF` karakter (3.6.j, 3.6.k). U ovom slučaju parser objavljuje uspešno parsiranje.

3.4 Leva i desna rekurzija

Kada se piše rekurzivno pravilo, rekurzivna referenca se može staviti na levi kraj, pa se zove leva rekurzija

```
exp_list : exp_list ',' exp ;
```

ili se može staviti na desni kraj pravila, pa se zove desna rekurzija

```
exp_list : exp ',' exp_list ;
```

U većini slučajeva, gramatika se može pisati na oba načina. *Bison* rukuje levom rekurzijom mnogo efikasnije nego desnom. To je zato što na svom steku čuva sve simbole koji su dotad viđeni za sva delimično isparsirana pravila. Ako se koristi desno rekurzivna verzija pravila `exp_list` i na ulazu se pojavi lista od 10 izraza, kad se bude pročitao 10-ti izraz, na steku će biti već 20 elemenata: izraz i zarez za svih 10 izraza. Kada se lista završi, svi ugnježdeni `exp_list` pojmovi će biti redukovani, s desna na levo. S druge strane, ako se koristi levo rekurzivna varijanta pravila `exp_list`, pravilo `exp_list` će biti redukovano posle svakog `exp`, tako da lista nikad neće imati više od tri elementa na steku.

3.5 Primeri

3.5.1 Broj reči i rečenica u ulaznom tekstu

Data je gramatika teksta: Tekst se sastoji od 0 ili više rečenica. Rečenica započinje rečju na čijem početku je veliko slovo. Zatim mogu da slede reči koje započinju bilo velikim bilo malim slovom. Na kraju stoji tačka. Napraviti parser koji će izbrojati reči i rečenice koje se pojave u ulaznom tekstu.

Tekst gramatika u *bison*-BNF obliku izgleda ovako:

Listing 3.4: Gramatika teksta

```
text
   :  /* empty text */
   |  text sentence
   ;

sentence
   :  _CAPITAL_WORD words _DOT
   ;

words
   :  /* empty */
   |  words _WORD
   |  words _CAPITAL_WORD
   ;
```

Skener, koji predstavlja deo rešenja ovog primera, je već dat na listingu 2.20 (glava 2.3.7). Razlika je u tome, što je prethodni skener napravljen kao samostalni program, pa sadrži svoju `main()` funkciju i sopstvenu definiciju tokena. Skener za ovaj primer se koristi u kombinaciji sa parserom i prikazan je listingu 3.5.

Listing 3.5: `count1.l`

```
%option noyywrap yylineno
%{
   #define YYSTYPE char*
   #include "count1.tab.h"
%}

%%

[ \t\n]+     { /* skip */ }

"."          { return _DOT; }
[A-Z][a-z]* { yylval = yytext; return _CAPITAL_WORD;}
```

```
[a-z]+      { yylval = yytext; return _WORD; }

.           { printf("\nline␣%d:␣LEXICAL␣ERROR␣on␣%c",
                     yylineno, *yytext); }
```

Parser treba da prebroji reči i rečenice, pa su zato potrebne 2 brojačke promenljive: `word_counter` i `sentence_counter` (listing 3.6). Brojač rečenica se inkrementira svaki put kada se prepozna cela rečenica, a to se dešava u akciji koja sledi iza pravila `sentence`. Brojač reči se inkrementira svaki put kada parser dobije od skenera token za neku reč: kada dobije prvu reč rečenice `_CAPITAL_WORD` (pravilo `sentence`), kada dobije `_WORD` u sredini rečenice (pravilo `word`) i kada dobije `_CAPITAL_WORD` u sredini rečenice (pravilo `word`).

Listing 3.6: `count1.y`

```
%{
   #include <stdio.h>
   #define YYSTYPE char*
   int yylex(void);
   int yyparse(void);

   int word_counter = 0;
   int sentence_counter = 0;
   extern int yylineno;
%}

%token _DOT
%token _CAPITAL_WORD
%token _WORD

%%

text
   :  /* empty text */
   |  text sentence
   ;

sentence
   :  _CAPITAL_WORD words _DOT
         {  word_counter++;
            sentence_counter++; }
   ;

words
   :  /* empty */
   |  words _WORD
         {  word_counter++; }
```

```
    |  words _CAPITAL_WORD
          {  word_counter++; }
    ;

%%

main() {
    yyparse();
    printf("Total␣words:␣%d.\n", word_counter);
    printf("Total␣sentences:␣%d.\n", sentence_counter);
}

yyerror(char *s) {
    fprintf(stderr, "line␣%d:␣SYNTAX␣ERROR␣%s\n", yylineno, s);
}
```

Kada se ovaj program pokrene, njegov izlaz će izgledati ovako:

```
$ ./count1
Marko se igra. Mile spava.
^D
Total words: 5.
Total sentences: 2.
$
```

3.5.2 Reči iz ulaznog teksta

Proširiti prethodni primer tako da parser, osim što prebroji reči i rečenice, još i ispiše sve reči iz ulaznog teksta.

Za rešenje ovog zadatka, parseru su potrebni stringovi reči koje treba da ispiše. Ove stringove prepoznaje skener, pa ih je na neki način potrebno proslediti parseru. Zato je potrebno malo modifikovati skener iz prethodnog primera, tako što će se sačuvati string reči koju skener prepozna (listing 3.7). Za tokene `_WORD` i `_CAPITAL_WORD` se, pored tokena, parseru prosleđuju i stringovi reči, kao vrednosti simbola, preko globalne *bison* promenljive `yylval`. String se kopira pomoću funkcije `strdup()` koja radi i alokaciju memorije i kopiranje. Zato je potrebno da se u parseru uradi dealokacija dotične memorije, u trenutku kada ti stringovi više ne trebaju.

Listing 3.7: `count2.l` - modifikacija

```
[A-Z][a-z]* { yylval = strdup(yytext); return _CAPITAL_WORD;}
[a-z]+      { yylval = strdup(yytext); return _WORD; }
```

Ispis reči se vrši svaki put kada u parser stigne neka reč. Nakon svakog ispisa reči sledi oslobađanje memorije koju je u skeneru zauzela funkcija `strdup`().

Listing 3.8: `count2.y` - pravila

```
text
   :  /* empty text */
   |  text sentence
   ;

sentence
   :  _CAPITAL_WORD words _DOT
         {  word_counter++;
            sentence_counter++;
            printf("%s\n", $1);
            free($1); }
   ;

words
   :  /* empty */
   |  words _WORD
         {  word_counter++;
            printf("%s\n", $2);
            free($2); }

   |  words _CAPITAL_WORD
         {  word_counter++;
            printf("%s\n", $2);
            free($2); }
   ;
```

Kada se ovaj program pokrene, njegov izlaz će izgledati ovako:

```
$ ./count2
Danas je predivan dan.
je
predivan
dan
Danas
^D
Total words: 4.
Total sentences: 1.
$
```

Osim prve reči, sve reči su ispisane u redosledu u kom su se pojavile u ulaznom tekstu. Prva reč je ispisana na kraju. Ovo se dešava zato što se kod, koji vrši ispis prve reči u rečenici, nalazi u akciji koja se izvršava tek kada se završi parsiranje cele rečenice (pa se i ispis prve reči vrši kada je prepoznata cela rečenica).

U tekstu zadatka se ne traži da se reči ispišu u redosledu u kom su se pojavile u ulaznom tekstu, pa se ovo rešenje može smatrati zadovoljavajućim. Sledeći primer rešava pitanje redosleda ispisivanja reči.

3.5.2.1 Ispravan redosled reči iz ulaznog teksta

Jedina izmena je u pravilu `sentence`. Akciju, koja ispisuje prvu reč je potrebno prebaciti odmah nakon preuzimanja tokena `_CAPITAL_WORD`. U ovom slučaju, prva reč u rečenici će biti ispisana čim dospe u parser, a ne, kao u prethodnom primeru, kada se prepozna cela rečenica.

Listing 3.9: `count3.y` - pravilo `sentence`

```
sentence
   :  _CAPITAL_WORD
         {  word_counter++;
            printf("%s\n", $1);
            free($1); }
      words _DOT
         {  sentence_counter++;  }
   ;
```

Kada se ovaj program pokrene, njegov izlaz će izgledati ovako:

```
$ ./count3
Danas je lep dan.
Danas
je
lep
dan
^D
Total words: 4.
Total sentences: 1.
$
```

3.5.2.2 Tekst u zagradama

Proširiti prethodnu tekst gramatiku tako da omogući da se jedna ili više reči piše u malim zagradama. Ispred prve reči ne sme da se pojavi zagrada. Prazne zagrade su dozvoljene. Zagrade moraju biti u paru. Mogu da postoje ugnježdene zagrade (zagrade u zagradama).

Za rešenje možemo preuzeti skener iz prethodnog primera i dodati mu pravila za prepoznavanje otvorene i zatvorene male zagrade (listing 3.10).

Listing 3.10: `count4.l` - male zagrade

```
"("         { return _LPAREN; }
")"         { return _RPAREN; }
```

Možemo preuzeti i parser iz prethodnog primera pa mu dodati definicije tokena (listing 3.11).

Listing 3.11: `count4.y` - tokeni

```
%token   _LPAREN
%token   _RPAREN
```

i proširiti pravilo `words` novim pravilom koje sadrži zagrade (listing 3.12).

Listing 3.12: `count4.y` - pravilo `words`

```
words
   :  /* empty */
   |  words _WORD
         {  word_counter++;
            printf("%s\n", $2);
            free($2); }

   |  words _CAPITAL_WORD
         {  word_counter++;
            printf("%s\n", $2);
            free($2); }

   |  words _LPAREN words _RPAREN
   ;
```

Novo rekurzivno pravilo opisuje da se par zagrada može naći oko bilo koje grupe reči. Pošto `words` može biti i prazan, znači da i zagrade mogu biti prazne. `words` može biti samo jedna reč, pa je moguće da se između zagrada nađe samo jedna reč, a opet, `words` se može sastojati od više reči, pa smo time omogućili da se između zagrada nađe i više od jedne reči.

Kako pojam `words` ne opisuje prvu reč u rečenici, obezbedili smo da se zagrade ne mogu naći ispred prve reči u rečenici (prvu reč u rečenici opisuje token `_CAPITAL_WORD` u pravilu `sentence`).

Time što je par zagrada definisan u pravilu `words`, omogućena je pojava ugnježdenih zagrada.

Kada se ovaj program pokrene, izlaz će izgledati kao na sledećem listingu:

```
$ ./count4
Danas (utorak) je lep dan (nije hladno (i vedro je)) ().
Danas
utorak
je
lep
dan
```

```
nije
hladno
i
vedro
je

Nezavrsene (zagrade u tekstu.
Nezavrsene
zagrade
u
tekstu
line 2: SYNTAX ERROR syntax error
Total words: 14.
Total sentences: 1.
$
```

Prva rečenica je sintaksno ispravna. Testirali smo varijante: jedna reč u zagradama, više reči u zagradama, ugnježdene zagrade i prazne zagrade. Druga rečenica je sintaksno neispravna, jer ne sadrži zatvorenu zagradu. Parser je preuzimao simbole sa ulaza dok nije stigao do kraja datoteke, a zatim je ispisao poruku da je detektovao sintaksnu grešku (jer nije pronašao zatvorenu zagradu), i završio svoju aktivnost.

3.5.3 Izmena formata datuma

Proširiti osnovnu tekst gramatiku (iz primera 3.5.1, na listingu 3.4) datumima u obliku `dd/mm/yyyy`. Napraviti parser koji procesira ulazni tekst i menja formu datuma u `dd-mmm-yyyy`. Ostatak teksta se ne menja (ispisuje se na izlazu u neizmenjenom obliku).

Listing 3.13: `date.l`

```
%option noyywrap yylineno
%{
   #include "date.tab.h"
%}

%%

[ \t\n]+       { /* skip */ }

"/"            { return _SEPARATOR; }
[0-9]{2}       { yylval.i = atoi(yytext); return _2D;}
[0-9]{4}       { yylval.i = atoi(yytext); return _4D; }

"."            { return _DOT; }
[A-Z][a-z]*    { yylval.s = yytext; return _CAPITAL_WORD; }
```

```
[a-z]+        { yylval.s = yytext; return _WORD; }
```

U skeneru se kao deo datuma prepoznaju simboli od 2 i od 4 cifre. Za njih su vezani tokeni `_2D` i `_4D`, a vrednosti ovih simbola su konkretne vrednosti brojeva, koje se smeštaju u polje `i` unije `yylval`. Kao separator ovih simbola definisan je karakter "`/`" i za njega je vezan token `_SEPARATOR`. Ostali simboli su, kao u prethodnim primerima, simboli tekst gramatike (jedina razlika je što se ovde stringovi reči prenose preko polja `s` unije `yylval`).

Unija u ovom parseru treba da ima jedan `integer` (jer se uz tokene `_2D` i `_4D` šalju konkretni brojevi) i jedan pokazivač na `string` (jer se uz tokene `_CAPITAL_WORD` i `_WORD` šalju stringovi). Kada se promenljiva `yylval` definiše kao unija, za svaki token koji ima vrednosti, se mora reći kog tipa je ta vrednost. Zato uz definiciju tokena `_2D` i `_4D` stoji oznaka `<i>`, a uz definiciju tokena `_CAPITAL_WORD` i `_WORD` stoji oznaka `<s>`.

Listing 3.14: `date.y`

```
    int yylex(void);
    int yyparse(void);
    extern int yylineno;
    char *months[] = { "", "jan", "feb", "mar", "apr", "maj",
       "jun", "jul", "avg", "sep", "okt", "nov", "dec" };
%}

%union {
    int i;
    char *s;
}

%token    _SEPARATOR
%token    <i> _2D
%token    <i> _4D
%token    _DOT
%token    <s> _CAPITAL_WORD
%token    <s> _WORD

%%

text
    :  /* empty text */
    |  text sentence
    ;

sentence
    :  _CAPITAL_WORD
          {  printf("%s", $1); }
       words _DOT
```

```
          {  printf(". "); }
   ;

words
   :  /* empty */

   |  words _WORD
          {  printf(" %s", $2); }

   |  words _CAPITAL_WORD
          {  printf(" %s", $2); }

   |  words date
   ;

date
   :  _2D
          {  printf(" %d-", $1); }
      _SEPARATOR _2D
          {  printf("%s-", months[$4]); }
      _SEPARATOR _4D
          {  printf("%d", $7); }
   ;

%%

int main() {
   yyparse();
   printf("\n");
}

int yyerror(char *s) {
   fprintf(stderr, "line %d: %s\n", yylineno, s);
}
```

Parser je proširen u pravilu `words`, tako da prima i datume, bilo gde u sredini rečenice. Čim se prepoznaju dve cifre koje označavaju dan, izvršava se akcija koja ispisuje taj broj na ekran (vrednost meta-promenljive `$1`), a iza njega novi separator. Zatim, kada se prepoznaju sledeće dve cifre, koje označavaju mesec, pomoću niza stringova `months`, definisanog u prvom delu specifikacije, se ispisuje tekstualna forma meseca i novi separator. Nakon što se prepoznaju 4 cifre za godinu, one se ispišu na ekran (vrednost meta-promenljive `$7`). Sve reči (i tačka), koje se prepoznaju u tekstu, se neizmenjene ispišu na izlaz.

Primer pokretanja ovog programa je prikazan na sledećem listingu:

```
$ ./date
```

```
Danas je 06/11/2012 godine, a za dva meseca ce biti 06/01/2013.
Danas je 6-nov-2012 godine, a za dva meseca ce biti 6-jan-2013.
^D
$
```

3.5.4 Strukturirana datoteka sa meteorološkim podacima

Napraviti program koji analizira ulaznu datoteku sa klimatskim podacima za jedan mesec i ispisuje koliko je bilo dana sa temperaturom iznad 12 stepeni Celzijusa. Datoteka se sastoji od više slogova, a slog ima sledeću strukturu:

```
temperatura:  13
pritisak:     1004.9
pravac vetra: E
brzina vetra: 3
vlaznost:     67
```

Listing 3.15 sadrži skener a listing 3.16 parser ovog primera.

Listing 3.15: `meteo.l`

```
%option noyywrap yylineno
%{
   #include "meteo.tab.h"
%}

%%

[ \t\n]+    { /* skip */ }

"temperatura"    { return _TEMPERATURA; }
"pritisak"       { return _PRITISAK; }
"pravac vetra"   { return _PRAVAC_VETRA; }
"brzina vetra"   { return _BRZINA_VETRA; }
"vlaznost"       { return _VLAGA; }
":"              { return _DVOTACKA; }
[0-9]+           { yylval.i = atoi(yytext); return _INT;}
[0-9]+\.[0-9]*   { yylval.f = atoi(yytext); return _FLOAT; }
E|NE|SE|W|NW|SW  { return _VETAR; }
```

Skener prepoznaje sve ključne reči iz sloga, kao i dvotačku, i za njih šalje samo token. Prilikom prepoznavanja celog i razlomljenog broja, on šalje odgovarajući token (`_INT` ili `_FLOAT`) i vrednost broja. Unija sadrži tipove `int` i `float`, da bi mogle da se prenesu konkretne vrednosti ovih brojeva. Tokenu `_INT` je dodeljen tip `<i>`, a tokenu `_FLOAT` tip `<f>`. Token `_VETAR` se šalje kada skener prepozna neku kombinaciju karaktera, koja označava smer vetra (`E`, `NE`, `SE`, `W`, `NW`, `SW`).

Primenjena gramatika opisuje da se jedan fajl sastoji od jednog ili više slogova, a da jedan slog redom sadrži: opis temperature, pritiska, pravca vetra, brzine vetra i vlažnosti. Deo sloga koji opisuje temperaturu se sastoji prvo od ključne reči "`temperatura`", zatim sledi dvotačka, pa zatim ceo broj koji sadrži vrednost temperature (listing 3.16).

Listing 3.16: `meteo.y`

```
%{
   #include <stdio.h>
   int yylex(void);
   int yyparse(void);
   extern int yylineno;
   int temp = 0;
%}

%union {
   int i;
   float f;
}

%token   _TEMPERATURA
%token   _PRITISAK
%token   _PRAVAC_VETRA
%token   _BRZINA_VETRA
%token   _VLAGA
%token   _DVOTACKA

%token   <i> _INT
%token   <f> _FLOAT
%token   _VETAR

%%

fajl
   :  slog
   |  fajl slog
   ;

slog
   :  temperatura pritisak pravac_vetra brzina_vetra vlaga
   ;

temperatura
   :  _TEMPERATURA _DVOTACKA _INT
         {  if($3 > 12) temp++;  }
   ;

pritisak
```

```
    :  _PRITISAK _DVOTACKA _FLOAT
    ;

pravac_vetra
    :  _PRAVAC_VETRA _DVOTACKA _VETAR
    ;

brzina_vetra
    :  _BRZINA_VETRA _DVOTACKA _INT
    ;

vlaga
    :  _VLAGA _DVOTACKA _INT
    ;

%%

int main() {
    yyparse();
    printf("U␣ovom␣mesecu␣bilo␣je␣%d␣dana␣sa␣t␣>␣12C.\n", temp);
}

int yyerror(char *s) {
    fprintf(stderr, "line␣%d:␣%s\n", yylineno, s);
}
```

Da bi izračunali koliko je dana bilo sa temperaturom preko 12 stepeni, treba nam promenljiva u kojoj ćemo brojati takve dane (promenljiva `temp`). Nakon prepoznavanja tokena `_INT` (što je ujedno i kraj pravila `temperatura`), smeštamo akciju koja inkremetira promenljivu `temp`, ako je vrednost temperature `> 12`. Vrednosti broja pristupamo preko meta-promenljive `$3` (jer želimo da pročitamo vrednost uz token `_INT` koji se nalazi na 3-ćoj poziciji na desnoj strani pravila). Ispis broja dana se vrši iz `main` funkcije, nakon parsiranja svih slogova.

3.6 Vežbe

1. Proširiti kalkulator tako da prihvata i linijske komentare.
2. Proširiti kalkulator tako da prihvata i heksa i decimalne brojeve. Uputstvo: u skeneru dodati obrazac `0x[a-f0-9]+` za prepoznavanje heksa cifara, a u akciji koristiti funkciju `strtol` za konverziju stringa u broj koji se prenosi preko `yylval`, i vratiti token `NUMBER`. Prilagoditi poziv funkcije `printf` tako da može da ispiše rezultat i kao decimalni i kao heksa broj.

3. Proširiti kalkulator operatorima na nivou bita, kao što su `AND` i `OR`.
4. Proširiti primer 3 tako da izračuna prosečnu temperaturu.
5. Napisati gramatike za svaki od ovih jezika:
   (a) Svi nizovi reči "da" i "ne" koji sadrže isti broj reči "da" i reči "ne" (u bilo kom redosledu).
   (b) Svi nizovi reči "da" i "ne" koji sadrže duplo više reči "da" od "ne".

Poglavlje 4

Konflikti i njihovo razrešavanje

Za sve, osim najjednostavnijih gramatika, korisnik generatora LR parsera treba da očekuje prijavu konflikata, kada prvi put propusti gramatiku kroz generator. Ovi konflikti mogu biti posledica dvosmislenosti gramatike ili ograničenja metode parsiranja. U oba slučaja, konflikti se mogu eliminisati prepravljanjem gramatike ili deklarisanjem prednosti operatora. *Bison* pruža mogućnost da se prednost operatora opiše odvojeno od pravila, što čini gramatiku i parser manjim i jednostavnijim za održavanje.

Većina generatora može pružiti informacije pomoću kojih se može locirati mesto u gramatici na kom se nalazi problem. Pokretanjem *bison*-a sa opcijom (svičem) `-v` (*verbose*), generiše se datoteka `name.output`. Ova datoteka sadži spisak konflikata, kao i opis u kom stanju parsera se desio konflikt.

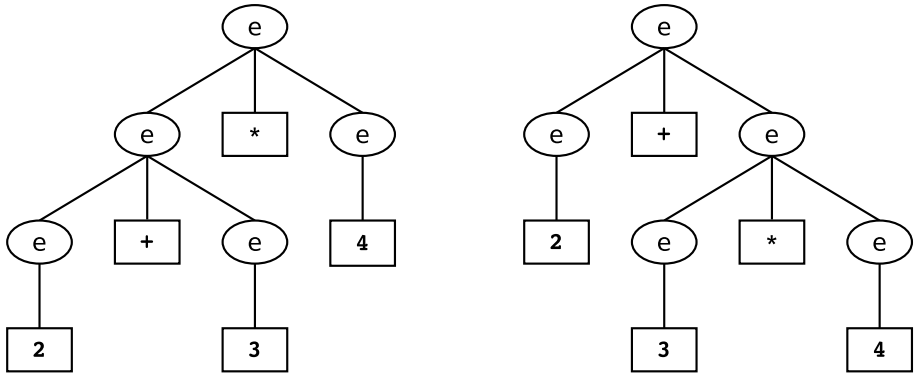
4.1 Primer konflikta

Ako prethodnu gramatiku za izraze proširimo operacijama množenja, deljenja i unarnim minusom:

```
e : e "+" e
  | e "-" e
  | e "*" e
  | e "/" e
  | "-" e
  | NUMBER
  ;
```

dobićemo dvosmislenu gramatiku. Na primer, ulaz `2+3*4` može značiti `(2+3)*4` ili `2+(3*4)`, a ulaz `3-4-5-6` može značiti `3-(4-(5-6))` ili `(3-4)-(5-6)`

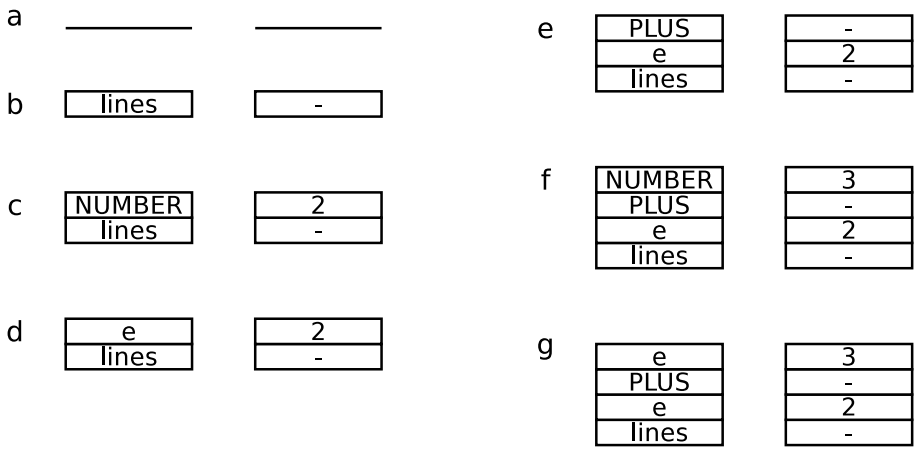
i još mnogo sličnih mogućnosti. Slika 4.1 pokazuje dva moguća stabla parsiranja za primer `2+3*4`.



Slika 4.1: Dva moguća stabla parsiranja za primer `2+3*4`

4.2 Razrešenje konflikta

Ako ovakvu gramatiku prosledimo *bison*-u, on će nam saopštiti da postoji više *shift/reduce* konflikata. To su stanja gde on može da uradi i *shift* i *reduce* akciju (da šiftuje token ili da uradi redukciju po nekom pravilu), a ne zna za šta da se odluči. Kada parser parsira prethodni primer `2+3*4` on prolazi kroz korake prikazane na slici 4.2.



Slika 4.2: Primer konflikta za izraz `2+3*4`

Nakon koraka g, parser vidi * i može da uradi redukciju poslednja 3 elementa (2+3) po pravilu e : e '+' e, a može da uradi i šiftovanje operatora *, očekujući da će moći kasnije da uradi redukciju po pravilu e : e '*' e.

Problem je nastao zbog toga što *bison*-u nismo ništa rekli o prioritetu i asocijativnosti operatora. Prioritet upravlja redosledom izvršavanja izraza. Množenje i deljenje imaju prednost nad sabiranjem i oduzimanjem, tako da izraz a+b*c ima značenje a+(b*c), a d/e-f znači (d/e)-f. U bilo kojoj gramatici izraza, operatori su grupisani u nivoe prioriteta od najnižeg ka najvišem. Postoje dva načna da se definišu prioritet i asocijativnost u gramatici: implicitno i eksplicitno. Implicitna definicija prioriteta podrazumeva definiciju zasebnih pojmova za svaki nivo prioriteta. Naša gramatika bi u ovom slučaju izgledala:

```
mul_exp : num_exp "*" num_exp
        | num_exp "/" num_exp
        ;

num_exp : exp "+" exp
        | exp "-" exp
        ;

exp     : NUMBER
        ;
```

Međutim, *bison* pruža mogućnost i eksplicitnog definisanja prioriteta operatora. Dodavanjem linija sa listinga 4.1 u prvi deo *bison* specifikacije, saopštavamo parseru kako da razreši konflikte.

Listing 4.1: calc4.y - prioritet operatora

```
%left  PLUS MINUS
%left  MULTIPLY DIVIDE
%right UMINUS
```

Svaka od ovih deklaracija definiše nivo prioriteta, a redosled navođenja %left, %right, i %nonassoc declaracija definiše redosled prioriteta od najnižeg ka najvišem. One kažu *bison*-u da su + i - levo asocijativni i sa najnižim prioritetom, da su * i / levo asocijativni i sa višim prioritetom i da je UNIMUS, pseudotoken za unarni minus, desno asocijativan i ima najviši prioritet.

Bison dodeljuje svakom pravilu prioritet krajnje desnog tokena na desnoj strani pravila. Ako taj token nema pridružen prioritet, ni pravilo nema svoj prioritet. Kada *bison* naiđe na *shift/reduce* konflikt, konsultuje tabelu prioriteta i, ako sva pravila koja učestvuju u konfliku imaju dodeljen prioritet, koristi taj prioritet za razrešavanje konflikta.

U našoj gramatici, svi konflikti se pojavljuju u pravilima oblika `exp OP exp`, tako da definisanje prioriteta ova 4 operatora omogućuje razrešavanje svih konflikata. Parser koji koristi eksplicitne definicije prioritera operatora je malo manji i brži od onog koji sadrži dodatna pravila sa implicitnom definicijom prioriteta, jer ima manje pravila za redukciju.

Listing 4.2: `calc4.y` - prioritet pravila

```
e
    :   e PLUS e                { $$ = $1 + $3; }
    |   e MINUS e               { $$ = $1 - $3; }
    |   e MULTIPLY e            { $$ = $1 * $3; }
    |   e DIVIDE e              { $$ = $1 / $3; }
    |   MINUS e %prec UMINUS    { $$ = -$2; }
    |   NUMBER                  { $$ = $1; }
    ;
```

Na listingu 4.2 pravilo za negaciju sadrži `%prec UMINUS`. Jedini operator u ovom pravilu je - (minus), koji ima nizak prioritet. Međutim kada ga koristimo kao unarni minus, želimo da ima viši prioritet od množenja i deljenja. Oznaka `%prec` kaže *bison*-u da korisiti prioritet `UMINUS` za ovo pravilo.

Prioritet operatora treba koristiti u gramatikama koje opisuju izraze. Inače, treba popraviti gramatiku tako da konflikt nestane. Postojanje konflikta znači da *bison* ne može da napravi parser za gramatiku jer je ona dvosmislena.

Poglavlje 5

Rukovanje greškama i oporavak

5.1 Rukovanje greškama

Kada bi kompajler trebao da obrađuje samo ispravne programe, njegovo osmišljavanje i implementacija bi bili znatno pojednostavljeni. Međutim, programeri često pišu neispravne programe, pa dobri kompajleri treba da pomognu programeru u identifikovanju i lociranju grešaka. Programi mogu sadržati greške na različitim nivoima. Na primer, greške mogu biti:

- leksičke (pogrešno napisano ime, ključna reč ili operator)
- sintaksne (logički izraz sa nepotpunim parom zagrada)
- semantičke (operator primenjen na nekompatibilni operand)
- logičke (beskonačan rekurzivni poziv).

Rukovaoc greškama u parseru ima nekoliko ciljeva:

- da saopšti prisustvo grešaka jasno i ispravno
- da se oporavi od greške dovoljno brzo da bi mogao da detektuje naredne greške
- da ne usporava bitno obradu ispravnih programa.

U svakoj fazi kompajliranja se može pojaviti greška, pa zato svaka faza mora postupiti sa greškom tako, da se proces kompajliranja može nastaviti, sa ciljem detekcije još novih grešaka. Najvećim brojem grešaka (koje kompajler može da otkrije) rukuju sintaksna i semantička analiza.

5.2 Oporavak od greške

Kada *bison*-ov parser detektuje grešku, ispiše string "`syntax error`" i završi parsiranje (završi program). Ukoliko korisniku to nije dovoljno, *bison* nudi mogućnost oporavka od greške pomoću tokena `error`.

Specijalni pseudo-token `error` označava mesto oporavka od greške. Kada parser detektuje grešku, on počinje da odbacuje simbole sa steka sve dok ne dostigne mesto gde će `error` token biti validan. Parser odbacuje ulazne tokene sve dok ne pronađe jedan koji može šiftovati u trenutnom stanju i tada nastavlja parsiranje od te tačke. Ako parsiranje ponovo detektuje grešku, odbacuje se još simbola sa steka i još ulaznih tokena, sve dok ne postane moguć nastavak parsiranja ili dok se stek ne isprazni. Da bi se izbeglo puno suvišnih (neadekvatnih) poruka o greškama, parser, posle prve greške, prestane da prijavljuje poruke sve dok ne uspe da šiftuje tri tokena jedan za drugim.

U nastavku je dat primer kalkulatora koji detektuje grešku unutar izraza, oporavlja se od greške i nastavlja parsiranje (listing 5.1). Ovaj primer nastao je proširenjem prethodnog primera novim pravilom za pojam `lines`.

Listing 5.1: `calc3.y`

```
lines
    :
    |  lines NEWLINE
    |  lines e NEWLINE       { printf("%d\n", $2); }
    |  lines error NEWLINE   { yyerror("reenter␣last␣line:\n");
                               yyerrok; }
    ;
```

Dodali smo još jedno pravilo sa tokenom `error`. Pozicija ovog tokena opisuje pojavu bilo kakve sintaksne greške unutar linije, pre karaktera za novi red. Ukoliko se na ulazu pojavi greška, parser će uraditi redukciju po ovom pravilu, i u našem slučaju, ispisati poruku korisniku da ponovo unese izraz. Nakon novog unosa, parser nastavlja sa parsiranjem. Za ispis greške koristi se funkcija `yerror()`.

Makro `yyerrok` u akciji kaže parseru da je oporavak završen i da se naredne poruke o greškama mogu prijavljivati.

U cilju detekcije što više grešaka moguće je dodati puno pravila koja opisuju greške, ali u praksi retko postoji više od nakoliko pravila sa greškama.

Listinzi

Slike

Indeks

Dodatak A

Rečnik

akcija

C ili C++ kod pridružen obrascu u *flex* specifikaciji ili pravilu u *bison* specifikaciji. Kada se u ulaznom nizu prepozna obrazac ili pravilo, izvršava se kod pridružene akcije.

ASCII

American Standard Code for Information Interchange; skup od 128 simbola koji predstavljaju simbole Američkog alfabeta: mala slova, velika slova, cifre, znaci interpunkcije i dodatni karakteri za formatiranje.

bison

Program koji prevodi gramatiku, napisanu u posebnom obliku BNF notacije, u LALR(1) parsere. Vidi LALR(1).

BNF

Backus-Naur Form; način predstavljanja kontekstno nezavisnih gramatika. Obično se koriste za specifikaciju formalnih gramatika programskih jezika. Ulazna sintaksa za *bison* je pojednostavljena verzija BNF-a.

dvosmislenost

Dvosmislena gramatika je ona koja sadrži više od jednog pravila ili skupa pravila kojima se može prepoznati isti ulazni string. U *bison-ovoj* gramatici, dvosmilsena pravila dovode do pojave *shift/reduce* ili *reduce/reduce* konflikta. Mehanizam parsiranja, koji *bison* uobičajeno koristi, ne može da razreši dvosmislene gramatike. Programer može da definiše prioritet operatora i prioritet pravila da bi razrešio konflikte, ili da preformuliše gramatiku.

epsilon

Specijalan slučaj ulaznog stringa sa nula simbola, tj. prazan string.

flex

Program koji generiše leksičke analizatore (skenere), koji prepoznaju obrasce (definisane regularnim izrazima) u ulaznom tekstu.

gramatika

Skup pravila koji zajedno definišu jezik.

interpreter

Program koji čita program napisan na jeziku visokog nivoa i izvršava ga. Uporedi sa kompajlerom.

jezik

Formalno: definisan skup stringova iz nekog alfabeta, a neformalno: skup instrukcija za opis zadataka koji se mogu izvršiti na računaru.

kompajler

Program koji prevodi program, napisan na jeziku visokog nivoa, na ekvivalentan program na jeziku niskog nivoa. Obično, izlaz iz kompajlera je na mašinskom jeziku koji se može direktno izvršavati na računaru. Uporedi sa interpreterom.

konačni automat

Finite automaton; apstraktna mašina koja se sastoji od konačnog broja instrukcija (prelaza). Konačni automati su korisni za modelovanje uobičajenih računarskih procesa a imaju i korisna matematička svojstva. *Flex* i *bison* prave skenere i parsere na osnovu konačnih automata.

konflikt

Greška u *bison* gramatici koja se pojavljuje u situaciji kada se, za isti ulazni niz tokena, mogu izvršiti dve ili više akcija parsiranja. Postoji dve vrste konflikata: *shift/reduce* i *reduce/reduce*. Vidi dvosmislenost.

kontekstno nezavisna gramatika

Context-free grammar; gramatika u kojoj svako pravilo ima samo jedan pojam sa leve strane pravila.

LALR(1)

Look Ahead Left to Right; tehnika parsiranja koju *bison* obično koristi. (1) označava da je *lookahead* limitiran na 1 token. Vidi *lookahead*.

leksički analizator

Program koji konvertuje niz karaktera u niz tokena. Zove se još i skener ili lekser.

lex

Program koji generiše leksičke analizatore (skenere). Savremeni naslednik mu je *flex*. Vidi leksički analizator.

lookahead

Ulaz koji je pročitan od strane parsera ili skenera, ali još nije prepoznat kao deo obrasca ili pravila. *Bison-ovi* parseri imaju jedan *lookahead* token, dok *flex-ovi* skeneri mogu imati neodređeno mnogo ovakvih tokena.

obrazac

U *flex-ovom* skeneru, obrazac se opisuje pomoću regularnih izraza. Skener pretražuje ulazni tekst da pronađe string koji odgovara datom regularnom izrazu, odnosno obrascu.

parsiranje

Proces preuzimanja niza tokena i provere ispravnosti njihovog redosleda.

pojam

Pojmovi (*nonterminals*) su elementi gramatike koji se ne pojavljuju u ulaznom tekstu nego se definišu pravilima. Vidi i simbol.

početni pojam

Jedan pojam koji predstavlja "cilj" parsiranja ulaznog teksta. Pravilo koje na levoj strani ima početni pojam se zove početno pravilo.

pravilo

U *bison-u*, pravila su apstraktni opis gramatike. Pravilo ima levu i desnu stranu, razdvojenu znakom ":", dok na kraju pravila stoji znak ";". Na levoj strani se nalazi samo jedan pojam (koji se definiše) a na desnoj strani niz simbola i pojmova (koji ga definišu).

prioritet

Redosled u kom se neke određene operacije vrše. Na primer, kada se interpretiraju matematičke naredbe, množenje i deljenje imaju viši prioritet od sabiranja i oduzimanja. Tako izraz `2+3*4` rezultira vrednošću `14` a ne `20`.

program

Skup instrukcija koje izvršavaju određeni zadatak.

reduce

Kada se u ulaznom tekstu prepozna niz simbola koji čini desnu stranu nekog pravila, *bison-ov* parser vrši *reduce* akciju, odnosno uklanja elemente koji obrazuju desnu stranu pravila sa steka i zamenjuje ih pojmom sa leve strane pravila.

***reduce/reduce* konflikt**

Situacija u kojoj istom nizu tokena odgovaraju dva ili više pravila. *Bison* razrešava konflikt tako što vrši *reduce* akciju po pravilu koje je ranije navedeno u gramatici.

regularni izraz

Jezik za opisivanje obrazaca koji služe za prepoznavanje niza karaktera.

shift

Akcija u toku parsiranja u kojoj *bison-ov* parser preuzima ulazni simbol i stavlja ga na stek.

***shift/reduce* konflikt**

Situacija u kojoj *bison-ov* parser može da uradi i *shift* i *reduce* akciju. *Shift/reduce* konflikti se javljaju ili zbog dvosmislenosti gramatike ili je parseru potrebno više od jednog *lookahead* tokena da bi odlučio koju akciju da primeni. *Bison*, u ovakvoj sitaciji, podrazumevano vrši *shift* akciju.

simbol

Simboli (*terminals*) se prepoznaju u skeneru i šalju se parseru. Vidi pojam (*nonterminal*).

specifikacija

Flex specifikacija je skup obrazaca po kojima se pretražuje ulazni tekst. Ovakvu specifikaciju *flex* pretvara u skener.

stek parsera

U *bison-ovom* parseru, simboli i pojmovi delimično prepoznatog pravila se smeštaju na interni stek. Simboli i pojmovi se dodaju na stek kada parser vrši *shift* akciju i uklanjaju sa njega kada parser vrši *reduce* akciju.

tabela simbola

Struktura podataka koja sadrži informacije o imenima koja se pojavljuju u programu (ulaznom tekstu) tako da sve reference na isto ime mogu biti vezane za isti objekat.

tokenizacija

Proces konvertovanja niza karaktera u niz tokena. Skener tokenizuje ulazni tekst.

ulaz

Podaci koje čita program. Na primer, ulaz za *flex-ov* skener je niz bajtova, dok je ulaz za *bison-ov* parser niz tokena (dobijenih od skenera).

vrednost

Skener parseru šalje token i, opciono, neku semantičku vrednost simbola.

yacc

Yet Another Compiler Compiler; predak *bison-a*, program koji generiše parser iz gramatike napisane u (modifikovanom) BNF formatu.

Bibliografija

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1986.

[2] Robert Corbett. Byacc. Technical report, 1990.

[3] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[4] M. E. Lesk and E. Schmidt. Unix vol. ii. chapter Lex - a lexical analyzer generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[5] J. Levine. *flex & bison*. Oreilly Series. O'Reilly Media, 2009.

[6] Vern Paxson. Flex, http://flex.sourceforge.net/. Technical report, 1987.

[7] Richard Stallman. Gnu bison, http://www.gnu.org/software/bison/. Technical report.