Construcția compilatoarelor folosind Flex și Bison

Anthony A. Aaby
Walla Walla College
cs.wwc.edu
aabyan@wwc.edu

Popa Dan
Universitatea Bacău
www.ub.ro
popavdan@yahoo.com, dpopa@ub.ro

Versiunea din 19 septembrie 2005

Walla Walla College
204 S. College Ave.
College Place, WA 99324
E-mail: aabyan@wwc.edu

Versiunea în limba româna este tradusă și adaptată de Dan Popa , popavdan@yahoo.com, dpopa@ub.ro. Am făcut de asemenea o serie de rectificări și corecturi. Am unificat notațiile pentru codul generat și instrucți-unile mașinii virtuale. Toate fișierele sursă au fost rescrise la calculator și toate programele au fost testate. Datorită adaptării, o serie de porțiuni din text au fost rescrise iar paragrafe întregi cu explicații au fost adăugate. Marcarea lor folosind caractere italice ar fi încărcat inutil textul dar cititorul interesat poate compara cele două ediții, engleză și română pentru a vedea ce modificări și adăugiri s-au făcut cu ocazia traducerii și adaptării. O parte din anexele versiunii în limba engleză nu au fost incluse în această versiune.

Cuprins

Titlul original: Compiler Construction using Flex and Bison

Contribuția autorilor:
Anthony A. Aaby: întregul volum cu excepția paragrafelor care nu se găsesc în ediția originală, sursele programelor cu comentarii în limba engleză, sursele din anexă
Popa Dan: paragrafele explicative care lipsesc din ediția în limba engleză, sursele programelor comentate în limba română, eratele cuprinse în text, notațiile unificate, rularea și verificarea programelor, corecturile surselor din anexă, corectura finală, traducerea și adaptarea

Popa Dan
Panselelor 10, Sc A, Et 1, Apt 5, Bacău, România
tel: 0234 562266
e-mail: popavdan@yahoo.com, dpopa@ub.ro

Popa Dan – Cartea prin e-mail
Acest volum poate fi solicitat gratuit, în format .pdf, la adresa publică de e-mail a autorului:
e-mail: popavdan@yahoo.com

Descrierea CIP propusă Bibliotecii Naționale a României

Anthony A. Aaby, Dan Popa

Construcția compilatoarelor folosind Flex și Bison / Anthony A. Aaby, Dan Popa - Bacău: 2005 , 78 pg

Bibliogr.

ISBN 973-0-04013-3

I. Aaby,Anthony A.

II. Popa, Dan

În loc de motivație

De ce să scriu o carte despre compilatoare, interpretoare ,Flex și Bison, în limba română, în anii primului cincinal al mileniului al doilea ? Sunt de actualitate ? Folosesc ele cuiva ? Sau rămân un simplu exercițiu academic ori tehnic, destinat profesorilor de teoria compilatoarelor și hackerilor (adevăraților hackeri, acei experți autentici în compilatoare care sunt în stare să-ți inventeze un limbaj de pe azi pe mâine sau să-ți ascundă un virus într-un compilator astfel ca el să genereze programe de login "sparte", chiar dacă le compilezi din nou) ? De ce o asemenea carte este necesară oportună și cui ar folosi ? Ce motive sunt pentru scrierea ei ?

Iată-le, într-o ordine oarecare, așa cum mi-au venit "la prima strigare":

1) Linux-ul se răspândeste rapid, numărul de utilizatori crescând vertiginos cu cifre între 6% și 36% pe an (numărul depinde de categoria de utilizatori, utilizatori casnici sau manageri de servere). Pentru acei utilizatori de Linux care, uneori în ciuda vârstei, mai au dorința de a învăța, de a se bucura ca niște copii care descoperă Google-ul, pentru ei, cei care uneori habar n-au ce servicii le pot aduce Flex-ul și Bisonul, merită să traduc, să scriu, să completez această carte. Au o șansă în plus să afle ce mai conține distribuția lor favorită de Linux. Nu scriu care distribuție, deoarece majoritatea distribuți-ilor de Linux pe care le-am consultat (RedHat 4.2, 5.2, 6.0, 6.1, 6.2, 7.0, 7.2 , Corel Linux , Mandrake Linux 8.0, 8.1, 8.2, 9, 10 și altele) includ pachetele Flex și Bison.

2) Flex și Bison nu sunt numai instrumente de scriere a compilatoarelor. Flex-ul e un generator de analizoare lexicale deci o unealtă pentru împărțit texte în "cuvinte" (sau litere sau alte grupări de simboluri cu structură specifică) și de executat acțiuni la întâlnirea lor. Bisonul este chiar mai puternic: poate prelucra date imbricate, având structuri și substructuri, specificate printr-o gramatică. Este un generator de parsere (analizoare sintactice) care pot fi dotate și cu reguli semantice, ceea ce îl face un veritabil generator de compilatoare.

3) Flex-ul și în mai mică măsură Bison-ul pot fi de folos și altor categorii de personal.
Într-un număr din revista PC Report, un alt generator de analizoare lexicale era prezentat printr-un exemplu care rezolva o problemă spinoasă tehnoredactorilor: conversia textelor de la un font la altul sau la standardul UTF. Cei care s-au lovit de problema diacriticelor limbii române, problemă pentru care există câte "bordeie" atâtea "obiceie de scriere" - a se citi fonturi sau standarde, și au primit nu o dată mail-ul unui șef supărat că textul nu se vede cu diacritice pe computerul său, vor înțelege actualitatea problemei.

4) Ați putea fi de asemenea cititori ai unei asemenea cărți dacă doriți să manipulați date structurate ori vreți să obțineți viteză la prelucrări ale unor anumite baze de date realizate cu XML , fără a mai învăța XML și în condițiile în care aplicația va fi realizată cu un

compilator de limbaj C.

5) Alte probleme banale pe fond dar relativ dificil de abordat cu mijloace clasice și totodată adesea întâlnite la concursurile pentru posturi de programatori se rezolvă ușor cu Flex și Bison. Dintre ele amintim crearea unui calculator evaluator de expresii numerice cu paranteze , (un asemenea modul software ar ferici orice utilizator de produs de contabilitate, dar admit că poate fi realizat și altfel decât cu un generator de compilatoare) sau clasica problemă de traducere care constă în a genera pentru un număr în cifre transcrierea lui în litere. Aceste probleme au și alte soluții dar dacă realizați o aplicație contabilă cu baze de date MySQL pe platformă Linux poate v-ar interesa și problemele de mai sus, mai ales dacă programați în C sau C++.

6) Există o mulțime de generatoare de compilatoare mai sofisticate decăt perechea Flex, Bison, unele folosite la Academie sau în medii universitare. Există și metode de specificare mai evoluate (de exemplu Viena Definition Method). Din nefericire ori softul este inaccesibil majorității cititorilor, ori are licență proprietară, ori rulează pe cine știe ce computer Sun Sparc Station pe care nu-l avem la dispoziție noi, utilizatoriui de PC-uri. Ori pur și simplu nu putem găsi în România software-ul ori documentația. Combinația Linux plus Flex plus Bison plus GCC (compilatorul GNU, fost GNU C Compiler) rămâne deocamdată generatorul de compilatoare cu răspândirea cea mai largă și deși există voci care afirmă că rezolvă doar 80% din problemă, este totuși îndrăgită de programatorii care adoră să se afirme scriind ei cât mai mult cod.

7) Dacă sunteți student la un curs de compilatoare sau doctorand la un îndrumător care se ocupă de construcția limbajelor, dacă sunteți frustrat că la finalul cursului n-ați făcut încă un compilator, fie și unul cât de mic, pentru un limbaj oricât de simplu și știți să dați câteva comenzi unui sistem Linux atunci această carte e pentru dumneavoastră cu siguranță. Nu pot să vă descriu emoția scrierii primului progrămel care va rula pe interpretorul dumneavoastră sau va fi compilat cu compilatorul dumneavoastră. Și mai ales satisfacți-a care o veți simți văzând că rulează sau se compilează și se execută fără erori. V-o spune un fost absolvent al primei serii de la o facultate de Informatică, serie născută prin transformarea uneia din cele de la secțiile Facultații de Matematică. Ca student reușisem performanța de a termina facultatea de Informatică fără să fac practic nici un compilator !!! Motiv pentru care dedic această carte și foștilor colegi de an și foștilor mei profesori, începătorii de altă dată (preparatori, asistenți etc), specialiștii de astăzi.

Capitolul 1

Introducere

Un compilator este în esență un program traducător ce transformă programe scrise într-un limbaj sursă în echivalentele lor scrise într-un limbaj obiect. Limbajul sursă este de obicei un limbaj de programare de nivel înalt iar limbajul obiect este de obicei limbajul mașină al computerului. Din punct de vedere pragmatic putem afirma că prin translator (compilator) se definește o anume semantică a limbajului de programare. Translatorul transformă operațiile date în mod sintactic în operații ale modelului de calcul al unei mașini virtuale sau reale. Acest capitol arată cum sunt folosite gramaticile independente de context la construcția translatoarelor (compilatoarelor). Deoarece traducerea este ghidată de sintaxa limbajului sursă, traducerea aceasta este numită "syntax-directed translation" adică "traducere bazată pe sintaxă" sau, mai bine zis "traducere dirijată de sintaxă".

De ce este nevoie de tot acest ansamblu de tehnici ? V-ați întrebat vreodată de fac compilatoarele analiză sintactică pe baza unei gramatici ? Doar de dragul verificării sintaxei ? Nicidecum ! Un limbaj de programare este echivalent cu o mulțime potențial infinită de programe. Sunt programele care se scriu în el. Dacă am încerca să definim direct cum s-ar traduce ele, luîndu-le rând pe rând, nu ne-ar ajunge veșnicia. Deci trebuie găsite alte tehnici. Totuși asemenea mulțimi infinite pot fi definite inductiv și stăpânite cu tehnici similare banalei inducții matematice de la liceu.

Ce este o definiție inductivă ? Dau un exemplu arhicunoscut: numerele naturale nenule se pot defini cu doar două elemente: 1 și functia succesor s(x) care ni-l dă pe succesorul lui x adică pe x plus 1. Atunci numerele naturale nenule sunt: 1, s(1), s(s(1)), șamd. V-ați prins ? Se începe cu ceea ce matematicienii numesc baza inducției și se continuă după o regulă inductivă care definește ceea ce urmează funcție de ceea ce aveam definit mai înainte. Astfel putem stăpâni numerele naturale, în infinitatea lor, lucrând cu doar două elemente , 0 și funcția s(x), cea care în fapt descrie cum se obține un număr mai complicat din precedentul.

Dorim să transcriem aceste numere ca sume de unități ? Dat fiind că numerele naturale nenule sunt definite doar cu ajutorul a două elemente, acum este simplu. Trebuie să definim cum se transcriu ca sume de unități cele două elemente : 1 și s(**ceva**) unde **ceva** la rândul său s-ar putea scrie tot ca o sumă de unități. Iată cele două reguli:

1 se transcrie prin 1, "suma" cu un termen
s(x) se transcrie prin: 1 plus transcrierea lui x

Adică T(1) este 1
iar T(s(x)) este 1 plus T(x)

Cât este, de exemplu, T(s(s(1)) ?
Este 1 plus T(s(1)) adică 1plus 1 plus T(1) adică 1 plus 1 plus 1.

Concluzia: Putem defini o traducere pe o mulțime infinită de elemente definite structural inductiv dacă știm cum se traduc: elementele cele mai simple (la un limbaj de programare structurat printre ele sunt instrucțiunile) și cum se traduc structurile compuse. Pentru structuri sunt date niște reguli care ne spun de obicei că traducerea structurii compuse se obține imediat din traducerile structurilor componente. Acesta este de altfel celebrul principiu compozițional din semantică.

Deci de ce avem nevoie de sintaxa limbajului, de structura lui sintactică sau schema, arborele sintactic al acestuia ? (Care nu-i mai complicat decât ceea ce se făcea la școala generală, la gramatică - acele desene arborescente!) Deoarece nu-l putem traduce altfel, fiind infinită mulțimea programelor posibile. Un mod de a stăpâni infinitatea aceasta este inducția matematică, definirea inductivă iar pentru inducție avem nevoie de baza inducției (elemente de limbaj despre care știm direct cum se traduc) și regulile inductive (structurale) care spun cum se traduce o structură compusă, traducându-i întâi componentele. Având traducerea definită astfel, inductiv, pe structuri și substructuri, e clar că primul pas al compilării e să găsesc, aceste structuri. Exact așa ceva face analiza sintactică ! Iar proiectantul unui limbaj și al unui compilator va începe prin a scrie gramatica limbajului, acea descriere inductivă, structurată a sintaxei limbajului însuși. Ei îi va asocia pe urmă regulile semantice destinate a conduce traducerea.

Definiție: Un *compilator* este un translator al cărui limbaj sursă este un limbaj de nivel înalt (apropiat omului, cu cuvinte din limbile umane) și al cărui limbaj obiect este apropiat de limbajul mașină al computerului concret disponibil. Compilatorul tipic parcurge următoarele faze și fiecare fază prelucrează "output-ul", ieșirea care se obține din faza precedentă:

> Faza de *analiză lexicală* ("scanarea" textului în căutarea cuvintelor și altor entități) grupează caracterele în unități lexicale, numite atomi lexicali (în engleză "tokens"). La intrarea analizorului lexical se primește programul ca un șir amorf de caractere, litere spații și simboluri. La ieșire rezultă un șir de atomi lexicali. Tradițional, a șazisele *"expresiile regulate"* sunt folosite pentru a defini șabloane de recunoaștere a atomilor de către scaner (analizor lexical). Acesta este implementat ca un automat finit determinist, o mașină cu un număr de stări finit care-și schimbă starea la primirea câte unui caracter. (Notă: Sunt posibile și alte tehnici, de exemplu s-a inventat analizorul lexical adaptiv care recunoște și învață singur diferitele categorii de atomi lexicali și ne scutește să mai scriem noi specificațiile. Tot ce ne rămâne e să spunem care dintre clasele de atomi descoperite vor fi identificatori, care numere etc. Nu va fi însă subiect al acestei cărți.)

Lex-ul și Flex-ul sunt instrumentele cele mai populare de generare a analizoarelor lexicale , acele programe care recunosc singure șabloane lexicale în text. Flex este o variantă mai rapidă a Lex-ului, compatibilă cu acesta la nivel de specificații. În acest capitol, prin Lex / Flex vom numi simultan ambele generatoare. În anexa despre Lex /Flex (din versiunea engleză a cărții) sunt preluate condensat elemente din manualul "flexdoc" scris de Vern

Paxon.

> *Analizorul sintactic* (parser-ul) grupează atomii lexicali (tokens) în unități (structuri) sintactice. Ca rezultat obține o reprezentare arborescentă: Arborele sintactic al programului, abreviat ca ST - *syntax tree*, în literatura de limbă engleză. O gramatică independentă de context furnizată parserului (gramatica limbajului) îi spune acestuia ce structuri sintactice să recunoască. Parserul este cel mai adesea implementat ca un automat push-down. (Adică o mașină cu număr finit de stări înzestrată și cu o stivă. Intuiți probabil deja că stiva îi este necesară pentru a urmări imbricarea structurilor sintactice, închiderea parantezelor deschise în expresii șamd.)

Yacc și Bison sunt instrumente pentru generarea de analizoare sintactice (parsere - programe care recunosc structura sintactică a unui program). Bison este o versiune mai rapidă de Yacc. În acest capitol, prin Yacc / Bison ne vom referi la ambele instrumente. Secțiunea privind Yacc / Bison este o condensare și apoi extindere a documentului "BISON the Yacc-compatible Parser Generator" scris de Charles Donelly împreună cu celebrul Richard Stallman.

> Faza de *analiză semantică* prelucrează arborele sintactic în căutarea acelor informații dependente de context numite în engleză "static semantics" (s-ar traduce prin "semantică statică"). La terminarea ei se obține un arbore sintactic (AST) adnotat. (De exemplu stabilirea tipurilor variabilelor din expresii, pe baza declarați ilor precedente se face în această fază.) *Gramaticile cu atribute* sunt suportul teoretic al acestei etape. Cu ajutorul lor se descrie semantica statică a unui program.

Această fază se realizează de obicei concomitent cu analiza sintactică. Informațiile colectate în această fază privind variabilele și alte entități (definite de programator în program) sunt culese și stocate în *tabela de simboluri* (în engleză - *symbol table*). Aceste informații sunt folosite la verificările dependente de context, acele verificări în care se confirmă că o entitate a fost folosită conform cu specificațiile existente despre ea.

> *Optimizatorul* (arborelui) supune arborele sintactic adnotat (AST) unor transformări care păstrează sensul de ansamblu al programului dar simplifică structura arborelui și facilitează generarea unui cod mai eficient.
>
> *Generatorul de cod* transformă arborele adnotat și simplificat în cod obiect, folosind reguli care denotă semantica limbajului de programare. (Vedeți și materiale despre *denotational semantics*.) Generatorul de cod poate fi integrat analizorului sintactic, astfel ca el să genereze cod imediat ce recunoaște o structură sintactică.
>
> Optimizatorul de cod generat (în engleză "peep-hole optimizer) examinează codul obiect generat și, constatând uneori că un grup de instrucțiuni ale procesorului se

poate înlocui cu una singura sau cu un grup mai mic, mai eficient, ori mai rapid, realizează îmbunătățiri ale codului dependente de mașină.

În contrast cu compilatorul, *interpretorul* este un program care simulează execuția programului scris în limbajul sursă. Interpretoarele pot "înțelege" și executa fie direct instrucțiuni ale unui limbaj de programare de nivel înalt, fie pot compila programul și executa iterpretativ codul rezultat, acest cod fiind de obicei generat ca pentru o mașină virtuală, adică un procesor de care nu dispunem în formă hardware. În această situație, compilatorul inclus generează un cod pentru acea mașină virtuală, cod care corespunde programului inițial din limbajul sursă.

Există o largă paletă de translatoare care se folosesc împreună cu compilatoarele în procesul de realizare a programelor. Un *asamblor* este acel translator (compilator dacă vreți) al unui *limbaj de asamblare.* Instrucțiunile acestui limbaj corespund una câte una unei instrucțiuni din codul obiect, adică din limbajul procesorului. (Cu alte cuvinte limbajul de asamblare nu este un limbaj structurat, în care să puteți scrie instrucțiuni compuse, imbricate, ci doar cel mult expresii , iar acestea sunt prelucrate doar de unele asambloare mai sofisticate). Există compilatoare care generează codul în limbaj de asamblare iar acesta va fi transformat în limbaj mașină de către un asamblor. Un încărcător (în engleză *loader*) e un translator pentru care atât limbajul de la intrare cât și cel de la ieșire sunt limbaj obiect. La intrare primește și o tabelă care spune unde trebuie modificat programul în limbaj mașină pentru a fi corect executat când este plasat în memorie la o adresă anume. Un *link-editor* e tot un fel de translator care primește o colecț ie de module de program executabile și le leagă împreună formând un singur program executabil destinat apoi execuției. Un *preprocesor* este un translator al cărui limbaj sursă este un fel de limbaj de nivel superior extins cu alte construcții, iar al cărui limbaj obiect este acel limbaj de nivel superior, neextins.

Ca exemplu, în cele ce urmează vom construi un compilator pentru un limbaj de programare imperativ - deja clasic în cursurile de compilatoare - numit **Simple**. Gramatica limbajului **Simple,** așa cum ne-o propune domnul Anthony A. Aaby, este :

```
program         ::=     LET [ declarations ] IN [command sequence] END

declarations    ::=     INTEGER [ id seq ] IDENTIFIER .

id seq          ::=     [id seq ...] IDENTIFIER ,

command sequence ::= command... command

command         ::=      SKIP ;
                 |       IDENTIFIER := expression ;
                 |       IF exp THEN [command sequence] ELSE [command sequence] FI ;
                 |       WHILE exp DO [command sequence] END ;
                 |       READ IDENTIFIER ;
                 |       WRITE expression ;
```

```
expression      ::=     NUMBER | IDENTIFIER | '(' expression ' )'
                | expression + expression     |       expression - expression
                | expression * expression     |       expression / expression
                | expression ^ expression
                | expression = expression
                | expression < expression
                | expression > expression
```

Neterminalii sunt scriși cu minuscule, terminalii (cuvintele limbajului) s-au scris cu majuscule sau cu simboluri și acolo unde ar fi fost pricină de confuzie cu metasimboluri din EBNF, s-au folosit și ghilimele simple. Simbolul de start este **program,** neterminalul corespunzător categoriei gramaticale a programelor complet scrise. Deși am folosit majuscule la scrierea cuvintelor limbajului (simbolurilor terminale), la implementare vom lucra cu minuscule.

Limbajul va avea două reguli, restricții, dependente de context. Se cere ca variabilele să fie declarate înainte de a fi folosite și ca variabilele să fie declarate exact o dată.

Pentru necunoscătorii notației folosite (de inspirație EBNF), comentăm sintaxa de mai sus, regulă cu regulă:

```
program         ::=     LET [ declarations ] IN [command sequence] END
```

Programul e format din cuvântul obligatoriu LET, niște declarații care pot lipsi (atunci când nu folosim variabile), cuvântul obligatoriu IN , secvența de instrucțiuni de executat, care poate fi și ea vidă, și, în cele din urmă se termină cu END.

```
declarations    ::=     INTEGER [ id seq ] IDENTIFIER .
```

Declarațiile încep cu cuvântul INTEGER și se termină cu un identificator (oarecare) și un punct. Poate fi după INTEGER o secvență mai lungă de alți identificatori declarați în aceeași declarație.

```
id seq          ::=     [ id seq ...] IDENTIFIER ,
```

Această secvență se termină totdeauna cu virgulă pusă după ultimul identificator și începe cu o secvență (care notăm noi poate fi și vidă. Practic secvența, așa cum va apare în declarație, este formată din identificatori urmați de virgulă, cu excepția ultimului (din declarație) despre care am văzut din regula anterioară că era urmat de punct.

```
command sequence ::= command... command
```

Secvența de comenzi este formată din comenzi puse una după alta, pur și simplu. Vom vedea din regula următoare că fiecare comandă include un simbol "punct și virgulă". Am scris peste tot [command sequence] cu paranteze drepte deoarece ea putea fi și vidă, adică putea lipsi. În notația de specialitate paranteza pătrată marchează un element op-

țional al construcției sintactice respective.

```
command     ::=     SKIP ;
                |   IDENTIFIER := expression ;
                |   IF exp THEN [command sequence] ELSE [command sequence] FI ;
                |   WHILE exp DO [command sequence] END ;
                |   READ IDENTIFIER ;
                |   WRITE expression ;
```

Instrucțiunile posibile sunt instrucțiunea vidă SKIP, atribuirea, IF-THEN-ELSE terminat cu FI, WHILE-DO terminat cu END precum și două instrucțiuni de intrare ieșire, absolut necesare ca să comunicăm cu calculatorul în Simple: citirea valorii unui identificator și scrierea valorii unei expresii.
Notați că IDENTIFICATOR este scris cu majuscule ca un terminal (iar el va fi furnizat ca atare după analiza lexicală, ca un singur atom lexical oricum s-ar numi el iar numele îi va deveni un simplu atribut). În timp ce "expression" , expresia, este scrisă cu minuscule, ca un neterminal deoarece e numele unei categorii gramaticale din program. Expresiile au structură, nu-s simboli terminali, și vor trebui analizate sintactic mai departe. Felul cum sunt structurate expresiile e descris de regula următoare:

```
expression  ::=     NUMBER |    IDENTIFIER |   '(' expression ' )'
                | expression + expression   |    expression - expression
                | expression * expression   |    expression / expression
                | expression ^ expression
                | expression = expression
                | expression < expression
                | expression > expression
```

Expresia poate fi un simplu număr, un simplu identificator sau o expresie pusă în paranteză (ce se comportă ca valoarea ei). Mai poate fi o sumă, o diferență, un produs sau un cât de expresii precum și un “adevărat” (1) sau “fals” (0) rezultat în urma unei comparații, a unei verificări dacă alte două expresii sunt într-o relație de egalitate , inegalitate ori comparație de un fel sau altul (< , >).

Observație: acele categorii gramaticale care corespund unor neterminali din care poate deriva un șir vid (deci acelea care pot lipsi) au fost, nu uitați, puse în paranteză. Vom vedea în capitolul următor o altă notație pentru gramatica limbajului Simple, evidențiind ni ște reguli (sau mai corect alternative ale unor reguli) cu partea dreaptă vidă. Dar aceasta la momentul potrivit. Deocamdată, în acest capitol, am vrut doar să va familiarizăm cu limbajul ce urmează a fi implementat.

Capitolul 2

Analizorul sintactic (Parser-ul)

Un parser este un program care determină dacă textul primit de el este sintactic corect și în plus, îi determină chiar structura. Altfel spus nu spune doar "Corect !" sau "Incorect !". Parserele pot fi scrise de mână sau pot fi automat generate de un generator de analizoare sintactice, pornind de la descrierea unor structuri sintactice corecte (ce vor fi recunoscute, validate, de viitorul parser). Aceste descrierei de sintaxă iau forma unor gramatici independente de context. Generatoarele de analizoare sintactice pot fi folosite la dezvoltarea de parsere pentru o gamă largă de limbaje, de la limbajul expresiilor cu paranteze pe care îl folosește un calculator de birou până la cele mai complexe limbaje de programare.

Yacc / Bison este un asemenea generator care primind la intrare o descriere a unei gramatici independente de context (GIC) construiește un program în C (iar versiunile noi de Bison pot produce și programe în C++ !!) care va analiza sintactic un text conform cu regulile gramaticii date. Yacc a fost dezvoltat de S. C. Johnson și de colegii săi de la AT&T Bell Laboratories. Atât Yacc cât și Bison permit să manipulați împreună cu stiva analizorului sintactic o a doua stivă (în care se operează sincron cu prima la fiecare începere/terminare de structură sintactică), această a doua stivă fiind destinată manipulărilor făcute de rutinele semantice. (Nu uitați că vom face traducere "syntax-driven", deci condusă de sintaxă).

Fișierul de intrare pentru Yacc/Bison are următoarea structură:

```
declarații C și declarații parser
%%
Regulile gramaticii și acțiunile semantice atașate (care vor opera cu a doua stivă)
%%
Funcții C (adesea aici se include și funcția Main, care nu e generată automat)
```

Pe scurt:

```
declarații C și  parser
%%
Reguli
%%
Funcții C
```

Prima secțiune conține și *lista atomilor lexicali* (diferiți de simple caractere) care sunt a șteptați de către parser deoarece fac parte din limbaj. Tot aici se include și declarația *simbolului inițial* din gramatică, acel neterminal care corespunde noțiunii de program complet. Această secțiune poate conține și specificații privind *ordinea operațiilor* și *asociativitatea operatorilor.* Ca efect, utilizatorul are o mai mare libertate în proiectarea limbajului, în algerea gramaticii sale. Adunarea și scăderea vor fi declarate asociative la

stânga și cu cea mai mică precedență în timp ce ridicarea la putere este declarată ca fiind asociativă la dreapta și având cea mai mare precedență, cea mai mare prioritate.

```
%start program
%token LET INTEGER IN
%token SKIP IF THEN ELSE END WHILE DO READ WRITE FI

%token NUMBER
%token IDENTIFIER ASSGNOP
%left  '-'  '+'
%left  '*'  '/'
%right '^'
%%
Reguli ale gramaticii cu acțiuni
%%
Subrutine C
```

A doua secțiune a fișierului cuprinde gramatica independentă de context a limbajului de analizat. Regulile (numite producții) sunt aici separate prin ";" (punct și virgulă), simbolul ' ::= ' este înlocuit de ':' , o eventuală parte dreaptă vidă este scrisă ca atare , neterminalii sunt scriși cu minuscule iar terminalii din mai multe simboluri (cum este semnul atribuirii) cu majuscule. Remarcați o anumită simplificare a gramaticii datorită separării informațiilor despre prioritatea operatorilor de restul gramaticii. (Constructorii de compilatoare cu experiență știu că luând în considerare prioritatea operatorilor, gramatica expresiilor aritmetice se scrie cu mai multe reguli decât în exemplul nostru.)

```
Declaratiile anterioare
%%
program : LET declarations IN commands END ;
declarations : /* empty */
        | INTEGER id seq IDENTIFIER '.'
;
id seq : /* empty */
        | id seq IDENTIFIER ','
;
commands : /* empty */
        | commands command ';'
;
command : SKIP
        | READ IDENTIFIER
        | WRITE exp
        | IDENTIFIER ASSGNOP exp
        | IF exp THEN commands ELSE commands FI
        | WHILE exp DO commands END
;
exp : NUMBER
        | IDENTIFIER
        | exp '<' exp
        | exp '=' exp
        | exp '>' exp
        | exp '+' exp
```

```
        | exp '-' exp
        | exp '*' exp
        | exp '/' exp
        | exp '^' exp
        | '(' exp ')'
;
%%
Subrutine C
```

A treia secțiune din fișierul cu specificații Yacc constă într-o colecție de funcții scrise în limbajul C. Printre ele se recomandă să existe funcția **main()** care va apela funcția **yyparse().** Funcția **yyparse()** este de fapt subrutina produsă de generator (pe baza specificațiilor de mai înainte) și este cea care va conduce analiza sintactică. În caz de eroare ea va încerca să apeleze funcția **yyerror()** pentru a semnala și prelucra eroarea. Această funcție yyparse() e lăsată să fie scrisă de programator, ceea ce conferă flexibilitate. Iată niște exemple simple cum ar putea arăta funcțiile main() și yyparse():

```
declarații C și declarații parser
%%
Regulile gramaticii
%%
main( int argc, char *argv[] )
{ extern FILE *yyin;
        ++argv; --argc;
        yyin = fopen( argv[0], 'r' );
        yydebug = 1;
        /* errors = 0; */
        yyparse ();
}

yyerror (char *s) /* Called by yyparse on error */
{       /* errors ++; */
        printf ("%s\n", s);
}
```

Notă: Variabila **errors** va numără erorile din timpul analizei sintactice. Puteți renunța la ea dacă nu doriți acest lucru. Dacă totuși doriți s-o folosiți, eliminați comentariile de pe rândurile unde se fac referințe la ea și nu uitați s-o declarați ca variabilă globală a programului în secțiunea declarațiilor C. (**int errors;**)
Dar pentru a nu complica acest exemplu minimal v-aș recomanda ca în prima fază să renunțați la ea.

Parserul, așa cum este generat pe baza specificațiilor de până acum, nu generează nimic la ieșire, nici măcar arborele sintactic. Acesta este însă implicit construit în timpul analizei. Pe măsură ce analiza progresează, parserul crează niște structuri interne care corespund sintaxei programului analizat. Reprezentarea aceasta internă se bazează pe părțile drepte

ale regulilor gramaticii. Când o parte dreaptă a unei producții (reguli) este recunoscută ea e înlocuită cu partea stângă corespunzătoare. Operația se numește *reducere.* Vom spune că partea dreaptă a regulii a fost *redusă* la neterminalul din stânga regulii. Acesta la rândul său este candidat la a figura în partea dreaptă a unei alte reguli și așa mai departe până când întregul program este redus la simbolul de start al gramaticii, semn că analiza a reușit cu succes.

Compilarea fișierului sursă de mai sus, **simple.y** se face cu Yacc, tastând comanda:

```
yacc -vd simple.y
```

sau pe un sistem Linux pe care este instalat Bison, cu comanda:

```
bison -vd simple.y
```

sau

```
bison -dv simple.y
```

...unde puteți înlocui pe simple.y cu numele fișierului dumneavoastră.
În urma prelucrării se obține un fișier cu extensia dublă .tab.c și același nume înaintea extensiei ca fișierul inițial. În exemplul nostru se va numi simple.tab.c deoarece specificați iile se numeau simple.y. Extensia tradițională a fișierelor cu specificații Yacc/Bison este y.

Fișierul .tab.c astfel obținut poate fi compilat cu compilatorul de C de pe sistemul dumneavoastră Unix / Linux. El conține și funcțiile scrise de dumneavoastră și funcția generată yyparse() care face analiza. Sistemul de Operare Linux vine cu compilatorul GCC, iar linia de comandă necesară pentru a compila sursa .c va fi:

```
gcc -c simple.tab.c
```

... ceea ce duce la obținerea modulului obiect cu numele simple.tab.o .

Notați și faptul că în urma folosirii Yacc / Bison se mai generează un fișier, cu extensia . tab.h . Acesta conține definiții de constante care asigură transferul corect al informațiilor între analizorul sintactic (parserul) generat de Yacc /Bison și analizorul lexical generat de Flex / Lex . Ideea este că fiecare tip de atom lexical va avea un număr, iar aceste numere definite și așteptate de parserul produs de Yacc / Bison trebui să-i vină de la analizorul lexical. Acesta trebuie deci și el să le cunoască, ceea ce va face incluzând în sursa sa fi șierul cu extensia .tab.h generat de Yacc / Bison.

Nu căutați Yacc pe sistemele Linux. Yacc este distribuit împreună cu sistemele Unix. Sistemele Linux, oferite sub licența GNU GPL de către Free Software Foundation Inc., au în distribuție Bison în loc de Yacc. Bison este un produs al Free Software Foundation.

Pentru mai multe informații despre folosirea Yacc / Bison vă rugăm fie să căutați în anexele cărții de față, fie să consultați pagina de manual pentru **bison** folosind comanda:

```
man bison
```

Mai puteți citi (dacă le găsiți !) lucrările: "Programming Utilities and Libraries LR Parsing" de A.V.Aho și S.C.Johnson publicată în Computing Surveys în iunie 1974 și documentul "BISON the Yacc-compatible Parser Generator", de Charles Donnelly și Richard Stallman.

În urma completării tuturor celor trei secțiuni ale fișierului Yacc, acesta va arăta ca mai jos:

```
/* Declaratii */
%start program
%token LET INTEGER IN
%token SKIP IF THEN ELSE END WHILE DO READ WRITE FI
%token NUMBER
%token IDENTIFIER ASSGNOP
%left '-' '+'
%left '*' '/'
%right '^'
%%
program : LET declarations IN commands END ;
declarations : /* empty */
  | INTEGER id_seq IDENTIFIER '.'
;
id_seq : /* empty */
  | id_seq IDENTIFIER ','

commands : /* empty */
  | commands command ';'
;
command : SKIP
  | READ IDENTIFIER
  | WRITE exp
  | IDENTIFIER ASSGNOP exp
  | IF exp THEN commands ELSE commands FI
  | WHILE exp DO commands END
;
exp : NUMBER
  | IDENTIFIER
  | exp '<' exp
  | exp '=' exp
  | exp '>' exp
  | exp '+' exp
  | exp '-' exp
  | exp '*' exp
  | exp '/' exp
  | exp '^' exp
  | '(' exp ')'
;
%%
/* Subrutine  */
int main (int argc, char * argv[] )
{extern FILE *yyin;
   ++argv; --argc;
   yyin = fopen( argv[0], "r" );
```

```
   yydebug = 1 ;
   yyparse();
}
yyerror (char *s)  /* Called by yyparse on error */
{
  print("%s\n", s);
}
```

Pentru a-l procesa urmați indicațiile din paginile precedente sau exemplul de sesiune de lucru la terminal dat în anexă.

Capitolul 3

Scaner-ul (Analizorul lexical)

Un analizor lexical (engl. scanner) este un program capabil să descopere și să clasifice porțiuni de text ("cuvinte") conforme cu niște șabloane date. Analizoarele lexicale pot fi ori scrise de mână, ori generate automat de un generator de analizoare lexicale, pe baza listei de șabloane pe care le au de recunoscut. Descrierea acestor șabloane se face sub formă de *expresii regulate*.

Lex este un generator de analizoare lexicale dezvoltat de M.E.Lesk și E.Schmidt de la AT&T Bell Laboratories. Lex primește la intrare un fișier conținând descrierile atomilor de recunoscut, fiecare clasă de asemenea atomi (ex. numere, identificatori etc) fiind caracterizată, descrisă, de o expresie regulată. Lex produce la ieșire un întreg modul de program, scris în C, care poate fi compilat și link-editat împreună cu alte module. (Printre aceste module este adesea și cel generat de Yacc /Bison - analizorul sintactic - precum și alte componente ale unui compilator, sau interpretor.) Fișierul de intrare pentru Lex /Flex este organizat similar cu cel pentru Yacc /Bison , tot în trei secțiuni separate prin %%. Prima conține declarații pentru scanner și alte declarații C , a doua conține în loc de gramatică descrierile atomilor lexicali (tokens) sub formă de expresii regulate și acțiunile pe care le va face analizorul întâlnindu-le (de obicei returnează parser-ului codul atomului respectiv, eventual și alte informații) iar a treia conține restul de subrutine (funcții) scrise în C.

Ca urmare a prelucrării intrării, Lex produce un fișier cu extensia .c, în care e definită funcț ia **yylex(),** funcție ce va returna un întreg corespunzător fiecărui atom găsit. Această funcț ie va fi apelată de funcția **main()** scrisă de programator în ultima secțiune, dacă facem doar recunoașterea și prelucrarea atomilor lexicali sau de funcția din modulul de analiză sintactică, dacă construim un compilator sau un interpretor. Într-o asemenea situație funcț ia yylex() oferă servicii (valori) funcției yyparse() din modulul generat de Yacc/Bison.

Codurile diversilor atomi lexicali sunt la fel înțelese de ambele funcții yylex() și yyparse() deoarece fișierul generat de Lex va cuprinde și directiva de includere a fișierului header generat de Yacc, cu extensia .tab.h. Includerea pică în sarcina programatorului.

Așa arată fișierul de intrare pentru Lex:

```
declarații C și declarații pentru scanner
%%
Descrieri de atomi lexicali sub forma expresilor regulate și acțiuni
%%
Funcții C , subrutine
```

În exemplul nostru, cea dintâi declarație care se va scrie în prima secțiune a fișierului

pentru Lex este cea de includere a fișierului simple.tab.h. , generat anterior de Yacc / Bison. El conține definții privind atomii multi-caracter. De asemenea, prima secțiune cuprinde o serie de definiții ajutătoare pentru scrierea expresiilor regulate din secțiunea a doua. În exemplul nostru vom defini aici pe DIGIT ca o secvență de una sau mai multe cifre de la 0 la 9 iar pe ID (de la identificator) ca fiind o literă urmată de una sau mai multe litere și/sau cifre.

```
%{
        #include "simple.tab.h"    /* The tokens */
%}
        DIGIT [0-9]
        ID [a-z][a-z0-9]*
%%
Descrieri de atomi lexicali sub forma expresilor regulate și acțiuni
%%
Funcții C , subrutine
```

Porțiunea dintre %{ și %} va fi transcrisă fără modificări în textul generat. Observați că am folosit directiva #include pentru a include fișierul simple.tab.h . 0-9 semnifică o cifră din mulțimea de cifre de la 0 la 9. a-z înseamnă o literă din mulțimea de litere de la a la z iar a-z0-9 un simbol din reuniunea celor două mulțimi. Observați că se scriu cu paranteză pătrată nu cu acoladă. Steluța de după paranteza pătrată înseamnă că pot fi oricâte elemente din acea mulțime în succesiune imediată, inclusiv nici unul. În cele din urmă facem observația că DIGIT reprezintă orice cifră iar ID orice identificator.

A doua secțiune din fișierul Lex (acestea au extensia .lex) este *seria de perechi* **șablon** (dat de expresia regulată) **- acțiune** (dată ca un corp de funcție C, cuprins în acolade). Fiecărui fel de atom lexical (token) descoperit îi corespunde o acțiune bine precizată. Șirurile de unu sau mai mulți digiți sunt recunoascute ca numere întregi și ca urmare, valoarea constantei NUMBER va fi returnată parserului. Cuvintele rezervate ale limbajului sunt secvențe de litere minuscule. (Se poate lucra și cu majuscule dar ele va fi necesar să fie tratate altfel !!) Spațiile albe, blanc-uri, tab-uri vor fi ignorate, nu vor produce nici o acț iune. În dreptul lor în loc de acțiune este un comentariu. Toate celelalte caractere (ceea ce în Lex se notează cu simbolul punct “**.**”) vor fi pur și simplu returnate așa cum sunt. Scanerul plasează textul de parcurs în variabila string **yytext** și întotdeauna yytext[0] este caracterul curent de prelucrat.

```
declarații C și declarații pentru scanner
%%
":="            { return(ASSGNOP); }
{DIGIT}+        { return(NUMBER); }
do              { return(DO); }
else            { return(ELSE); }
end             { return(END); }
fi              { return(FI); }
if              { return(IF); }
in              { return(IN); }
integer                 { return(INTEGER); }
```

```
let             { return(LET); }
read            { return(READ); }
skip            { return(SKIP); }
then            { return(THEN); }
while           { return(WHILE); }
write           { return(WRITE); }
{ID}            { return(IDENTIFIER); }
[ \t\n]+                /* blank, tab, new line: eat up whitespace */
.               { return(yytext[0]); }
%%
Subrutine C
```

Observați că fiecărui tip de atom lexical i se asociază de regulă acțiunea de a întoarce o valoare întreagă (constantă) care semnifică felul de atom lexical descoperit.

Lista următoare cuprinde forma câtorva feluri uzuale de expresii regulate care vă pot fi de folos la a specifica atomii lexicali de recunoscut. Notați și faptul că există o variabilă globală **yylval ,** accesibilă atât scanerului generat de Lex cât și parserului generat de Yacc /Bison. Ea poate fi folosită pentru a transmite alte informații legate de atomul al cărui cod tocmai a fost returnat.

. orice caracter cu excepția returului de car, ENTER

x șablonul care se potrivește cu caracterul 'x'

rs concatenarea (succesiunea) a ceva care se potrivește cu r și ceva ce se potrivește cu s

r|s reuniunea sau alternativa; se potrivesc cu ea atît secvențele de cractere corespunzătoare expresiei r cît și cele corespunzătoare expresiei s

(r) la fel ca r ; parentezele se folosesc la grupare

r* zero sau mai multe secvențe de caractere ce se potrivese cu r,
unde r poate fi orice expresie regulată

r+ una sau mai multe secvențe de caractere ce se potrivese cu r,
unde r poate fi orice expresie regulată

[xyz] o clasă (mulțime) de caractere, în acest exemplu se vor potrivi cu șablonul doar caracterele 'x' , 'y', 'z'

[abj-oZ] o clasă (mulțime) de caractere, în acest exemplu se vor potrivi cu șablonul doar caracterele 'a' sau 'b' ,orice de la 'j' la 'o' inclusiv, sau 'Z'

{NAME} expandarea unei (macro)definiții anterioare, cu numele NAME

\X dacă X este un ul dintre caracterele 'a', 'b', 'f', 'n', 'r', 't' sau 'v' , atunci \X se interpretează conform standardului ANSI-C

"[+xyz]+\"+foo" stringul: [xyz]"foo

A treia secțiune a fișierului va fi lăsată necompletată în acest exemplu. Dar în cazul în care acțiunile analizorului lexical presupuneau operații complexe și apeluri de funcții utilizator, aceste funcții puteau fi adăugate aici, în a treia secțiune. Astfel scrierea acțiunilor devine mai clară iar unele detalii nesemnificative sunt ascunse în funcțiile (subrutinele) din a treia secțiune.

Compilarea fișierului Lex se face cu una din comenzile:

lex *fișier.lex*

respectiv

flex *fișier.lex*

și se va obține un fișier cu denumirea lex.yy.c în care e generată funcția **yylex().** La fiecare apel al funcției yylex(), intrarea va fi scanată (parcursă) în căutarea următorului atom lexical și se va executa acțiunea programată care-i corespunde. În cazul din exemplul nostru se va returna codul atomului respectiv.

Veți lucra cu Lex dacă aveți un sistem Unix de firmă, deoarece Lex este distribuit împreună cu sistemele Unix sau cu "clona" acestuia, Flex, dacă folosiți un sistem Linux. Flex este un produs al Free Software Foundation, Inc., distribuit sub licență gnerală publică GNU GPL.

O sesiune tipică de lucru cu o mașină Linux, folosind Flex, Bison și compilatorul GCC pentru a obține programul binar, executabil, (in sursele C nerate de Flex și Bison) se desfășoară dând comenzile din lista de mai jos. Am presupus că cele două fișiere cu specificații se numesc *simple.y* și *simple.lex*. Promptul sistemului este notat cu "$" de obicei și nu face parte din comandă:

```
$ bison -dv simple.y
$ gcc -c simple.tab.c
$ flex simple.lex
$ gcc -c lex.yy.c
$ gcc -o simple simple.tab.o lex.yy.o -lm -lfl
```

Prima comandă invocă Bison pentru a genera din sursa *simple.y* fișierul *simple.tab.c* și fi șierul *simple.tab.h* care figurează ca inclus în sursa ce VA FI generată de Flex /Lex.
A doua invocă compilatorul gcc pentru a compila *simple.tab.c* și a obține modulul obiect *simple.tab.o.*
A treia generează din specificația Lex *simple.lex* sursa C cu nume standard lex.yy.c.
A patra este comanda de compilare a acestei surse și rezultă modulul obiect lex.yy.o.
În sfârșit, cu ultima comandă se leagă modulele obiect *simple.tab.o* și lex.yy.o împreună cu o serie de funcții de bibliotecă în executabilul binar *simple*. Atenție la opțiunile din finalul liniei de comandă. Pe sistemul Red Hat Linux 5.2 cu care am rulat prima oară aceste exemple trebuie scrise neaparat opțiunile -LM -LFL dar cu minuscule !!! Pe alte versiuni de sisteme Linux este suficientă doar prima opțiune -LM scrisă tot cu minuscule!
Pentru mai multe informații despre comenzile și opțiunile Lex / Flex consultați paginile de

manual **lex** , **flex** de pe sistemul dumneavoastră Unix / Linux, documentația **flexdoc** și citiț i lucrarea "Lex - Lexical Analyzer Generator de M. E. Lesk și E. Schmidt, în cazul când o puteți găsi.

În final, întregul fișier *simple*.lex, pentru limbajul Simple, arată așa:

```
%{
#include "simple.tab.h"  /* The tokens */
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
":="            { return(ASSGNOP);   }
{DIGIT}+        { return(NUMBER);    }
do              { return(DO);        }
else            { return(ELSE);      }
end             { return(END);       }
fi              { return(FI);        }
if              { return(IF);        }
in              { return(IN);        }
integer         { return(INTEGER);   }
let             { return(LET);       }
read            { return(READ);      }
skip            { return(SKIP);      }
then            { return(THEN);      }
while           { return(WHILE);     }
write           { return(WRITE);     }
{ID}            { return(IDENTIFIER); }
[ \n\t]+        /* blank, tab, new line: eat up whitespace */
.               { return(yytext[0]);   }
%%
```

Comentariile le puteți scrie, bineînțeles și în limba română. Primul **/* The tokens */** s-ar putea traduce prin **/*Atomii lexicali*/** deși textual tokens înseamnă “jetoanele”.
Al doilea /* blank, tab, new line: eat up whitespace */ s-ar traduce mai bine prin
/* succesiunile formate cu spațiu, tab sau enter – toate sunt consumate */.

Capitolul 4

Contextul

Specificațiile din fișierele Lex și Yacc pot fi extinse pentru a manipula informații dependente de context. De exemplu, să presupunem că în limbajul Simple impunem ca variabilele să fie declarate înainte de a fi folosite. Din acest motiv parserul (analizorul sintactic) trebuie să fie capabil să compare referințele la variabile cu declarațiile acestea, care sunt memorate în vederea verificărilor.

Un mod de a rezolva această problemă este de a construi încă din timpul parcurgerii secț iunii de declarații a programului un fel de listă de variabile și de a verifica apoi fiecare variabilă întâlnită, comparând-o cu datele de pe listă. O asemenea listă este numită *tabelă de simboluri*. Tabelele de simboluri pot fi implementate folosind liste, arbori, (arbori de căutare) sau tabele-hash.

Vom modifica fișierul Lex pentru a atribui variabilei globale **yylval** un string cu informații de identificare de fiecare dată când întâlnim o entitate (atom) a cărei set de caracteristici se vor păstra în context. Asemenea informații vor fi, în teorie, atribute prelucrate de *gramatica cu atributre*.

Modulul Tabelă de Simboluri (ST.h)

Pentru a păstra informațiile cerute de prelucrările ce le va face gramatica cu atribute, vom construi o tabelă de simboluri. Tabela de simboluri cuprinde informații de context privind atributele diverselor construcții (sintactice) scrise în limbajul de programare.În particular, pentru variabile, cuprinde informații despre tipul (eng. *type*) și domeniul de vizibilitate (eng. *scope*) al variabilei.

Tabela de simboluri pe care o vom dezvolta va fi un modul sursă C care se va include de către sursa generată (de către Yacc / Bison), cu ocazia compilării.

Cititorul ambițios este invitat ca, după parcurgerea integrală a acestui volum, să inventeze un limbaj de specificare a tabelelor de simboluri și un generator de surse C (similar Yacc-ului și Bison-ului) pentru a genera module asemănătoare celui de mai jos pornind de la specificații. Instrumentele cu care va lucra vor fi evident Yacc / Bison pentru analiza sintactică și generarea outputului prin acțiuni semantice și Lex /Flex pentru analiza lexicală).

Tabela de simboluri pentru limbajul Simple constă într-o listă simplu înlănțuită de identificatori, inițial vidă. Iată declarația unui nod (element) al listei

```
struct symrec
{char *name;                    /* name of symbol  - numele simbolului*/
```

```
struct symrec *next;   /* link field - pointerul la alt element*/
};
```

apoi definirea tipului *symrec* și inițializarea listei vide cu un pointer nul

```
typedef struct symrec symrec;
symrec *sym_table = (symrec *)0;
symrec *putsym ();
symrec *getsym ();
```

și cele două operații. Prima, *putsym* e cea care pune un identificator în tabelă

```
symrec *
putsym ( char *sym_name )
{symrec *ptr;
        ptr = (symrec *) malloc (sizeof(symrec));
        ptr -> name = (char *) malloc (strlen(sym_name)+1);
        strcpy (ptr -> name, sym_name);
        ptr -> next = (struct symrec *)sym_table;
        sym_table = ptr;
return ptr;
}
```

Comentez puțin codul funcției *putsym* pentru eventualii cititori mai puțin familiarizați cu limbajul C. Dacă stăpâniți bine C-ul, treceți direct la pagina următoare, la funcția *getsym*. Primul rând declară tipul rezultatului dat de funcție, care e pointer la un record (element de listă) *symrec*, deci e și pointer la o listă de asemenea elemente.

```
symrec *
```

Urmează numele funcției și paranteza cu parametrul formal unic de tip pointer la caracter (sinonim în C cu string), cu numele *sym_name*:

```
putsym ( char *sym_name )
```

Corpul funcției începe cu declararea variabilei locale *ptr*, pointer la tipul *symrec*:

```
{symrec *ptr;
```

Se alocă spațiu pentru structura (record-ul) propriu-zis și pentru numele *sym_name* ce e destinat a fi stocat în acea intrare din tabelă.

```
        ptr = (symrec *) malloc (sizeof(symrec));
        ptr -> name = (char *) malloc (strlen(sym_name)+1);
```

Se copie apoi numele dat ca parametru în spațiul alocat deja, indicat de pointerul *ptr->name*, stocând astfel copia numelui în tabelă. (Funcția C strcpy() este o veritabilă atribuire pentru stringuri, capabilă să copie al doilea string de la adresa sa, la adresa

primului string, dată ca prim parametru.)

```
        strcpy (ptr -> name, sym_name);
```

Elementul nou creat, aflat la adresa ptr devine primul element din listă iar capul de listă *sym_table* va arata acum spre el. Tot această adresă a elementului nou și simultan a listei este returnată de funcția al cărei corp se încheie cu obișnuita acoladă.

```
        ptr -> next = (struct symrec *)sym_table;
        sym_table = ptr;
return ptr;
}
```

Iar *getsym* returnează un pointer către intrarea din tabela de simboluri corespunzând unui identificator dat sau pointerul nul, zero, dacă nu l-a găsit.

```
symrec *
getsym ( char *sym_name )
{symrec *ptr;
        for (ptr = sym_table ; ptr != (symrec *) 0 ; ptr = (symrec *)ptr- >next)
                if (strcmp (ptr ->name,sym name) == 0)
                return ptr;
return 0 ;    /* sau eventual return (symrec *) 0; */
}
```

În final, modulul ST.h ar putea arăta așa:

```
/* ST.h the symbol table module */
/* De Dan Popa, dupa Anthony A Aaby , "Compiler Construction using Flex and Bison" */
/* Cap 4 pg 14 */

struct symrec
{
char *name;             /* name of symbol - numele variabilei*/
struct symrec *next;  /* link field - legatura la elementul urmator */
};

typedef struct symrec symrec;
symrec *sym_table = (symrec *)0;
symrec *putsym ();
symrec *getsym ();

/* Doua operatii:  punerea unui identificator in tabela ...*/

symrec *
putsym ( char *sym_name )
```

```
{
symrec *ptr;
ptr = (symrec *) malloc (sizeof(symrec));
ptr->name = (char *) malloc (strlen(sym_name)+1);
strcpy(ptr->name, sym_name);
ptr->next = (struct symrec *) sym_table;
sym_table = ptr;
return ptr;
}

/* si a doua operatie:  aflarea pointerul intrarii din tabel corespunzator  unui identificator. */

symrec*
getsym (char *sym_name)
{
symrec *ptr;
for (ptr = sym_table; ptr != (symrec *)0; ptr = (symrec *) ptr->next)
if (strcmp (ptr->name, sym_name) == 0 )
return ptr;
return 0;
}
```

Observați că funcția caută în lista *sym_table* parcurgând-o cu pointerul (local) *ptr* câtă vreme mai sunt elemente în listă deci pointerul e nenul. Dacă stringul dat *sym_name* coincide cu string-ul din elementul de listă curent, adresa acestuia este returnată. Altfel se returnează un 0 (care e practic un pointer nul). Funcția *strcmp()* compară stringurile de la cele două adrese.

Modificările analizorului sintactic (parserului)

Este necesar să modificăm specificațiile oferite Yacc-ului / Bison-ului, astfel încât să includem tabela de simboluri și rutinele necesare introducerii și căutării unui identificator în vederea verificării folosirii sale corecte, conforme contextului. Am adăugat și variabila pentru numărarea erorilor, **errors**, nedeclarată în capitolul anterior. Adăugirile se fac în secțiunea de specificații pe care Yacc / Bison le va copia direct în sursa C generată, adică vor fi scrise la început, între %{ și %}.

```
%{
#include <stdlib.h>          /* Include malloc, necesar pt. symbol table */
#include <string.h>          /* Include strcmp, necesar pt. symbol table */
#include <stdio.h>           /* Functii pentru afisarea de  mesaje de eroare */
#include "ST.h"                      /* Modulul "Symbol Table"  */
#define YYDEBUG 1            /* Pentru debugging setez variabila Yacc la valoarea 1*/

int errors;                  /* Contor de erori */

install ( char *sym_name )
{ symrec *s;
```

```
s = getsym (sym_name);
if (s == 0)
s = putsym (sym_name);
else { errors++;
        printf( "%s is already defined\n", sym_name );
        }
}

context check( char *sym_name )
{ if ( getsym( sym_name ) == 0 )
printf( "%s is an undeclared identifier\n", sym_name );
}

%}
Declaratii pentru Parser
%%
Gramatica: reguli plus actiuni
%%
Subrutine C
```

Deoarece analizorul lexical (scanerul generat de Lex / Flex) va returna doar valorile unor identificatori (constante întregi) și doar atât, o structură de record semantic (semantică statică) este necesară pentru a transmite denumirea fiecărui identificator de variabilă. Așa putem deosebi variabilele întâlnite deoarece altfel analizorul lexical returnează doar o aceeași valoare a constantei IDENTIFIER pentru orice variabilă.
Simple.lex va fi modificat astfel:

```
Declaratii C
%union {                /* RECORD SEMANTIC */
char *id;               /* Pentru transferul numelor identificatorilor*/
}
%start program
%token INT SKIP IF THEN ELSE FI WHILE DO END
%token <id> IDENTIFIER /* Simple identifier */
%left  '-'  '+'
%left  '*'  '/'
%right '^'
%%
Gramatica: reguli plus actiuni
%%
Subrutine C
```

În continuare, gramatica independentă de context și acțiunile ei semantice sunt modificate pentru a include apeluri la funcțiile *install()* și *context_check()* . Notați că fiecare $n este o variabilă Yacc care face referire la al n-lea record semantic, cel corespunzător celui de-al n-lea element din partea dreaptă a unei producții (reguli a gramaticii). Astfel vom putea prelucra, stoca, testa, denumirile diverșilor identificatori de variabile care apar în regulile gramaticii, în diverse poziții (date de ordinea sintactică). Metoda se aplică la orice fel de element sintactic (nu numai identificator de variabilă) a cărui proprietăți, atribute, urmează să le prelucrăm. Remarcați că poziția cuvântului IDENTIFIER în regula gramaticii

corespunde ca număr cu variabila $n implicată în acți-unea semantică, (unde n este acela și număr !).

```
Declaratiii pt. C si parser
%%
...
declarations : /* empty */
        | INTEGER id_seq IDENTIFIER '.' { install( $3 ); }
;
id_seq : /* empty */
        | id_seq  IDENTIFIER ',' { install( $2 ); }
;
command : SKIP
        | READ IDENTIFIER { context_check( $2 ); }
        | IDENTIFIER ASSGNOP exp { context_check( $1 ); }
...
exp : INT
        | IDENTIFIER { context_check( $2 ); } ...
%%
Subrutine C
```

După modificare și salvarea fișierului cu numele *simple.y*, Bison (pe un sistem Linux) va fi invocat cu una din comenzile:

```
bison -vd simple.y
bison -d simple.y
```

În această implementare arborele sintactic este creat implicit și adnotat implicit cu informații privind situația unei variabile: dacă ea este sau nu declarată (deci poate sau nu furniza o valoare) în momentul când ea este referită dintr-o expresie. Informațiile care ar trebui să adnoteze arborele sintactic sunt însă păstrate direct în tabela de simboluri.

Cititorii interesați de o implementare care crează efectiv un arbore sintactic adnotat pot consulta lucrarea "A Compact Guide To Lex & Yacc" de Thomas Niemann, disponibilă gratuit pe Internet, pe site-ul epapers.com. Notăm totuși că exemplele de acolo nu pot fi folosite pe un sistem Linux fără unele modificări. Vom încerca să publicăm în anexa acestui volum sau în partea a doua și unele surse corectate pentru Flex / Bison (deci destinate platformei Linux) pe care le-am rescris documentându-ne din lucrarea citată.

Modificările Scanner-ului (analizorului lexical)

Și scanerul (analizorul lexical) trebuie modificat pentru a returna denumirea (sub formă de string a ...) fiecărui identificator, ea fiind practic valoarea semantică (d.p.d.v. al semanticii statice) asociată fiecărui identificator. Această valoare semantică va fi returnată printr-o variabilă globală numită **yylval.** Tipul acestei variabile este un tip reuniune (*union* ca în C) creat cu directiva *%union* plasată în fișierul cu descrierea parserului. După caz și tipul

valorii semantice de comunicat de la scaner la parser, aceasta va fi plasată într-un membru corespunzător ca tip al reuniunii. Dacă declarația "union"-ului va fi

```
%union { char *id;
}
```

valoarea semantică va trebui copiată din variabila globală **yytext** (care conține întotdeauna textul utimului atom lexical) în **yylval.id** . Deoarece exemplul de mai jos folosește funcția **strdup** din biblioteca string.h, această bibliotecă trebuie să fie și ea inclusă. Fișierul cu descrierea scanerului, care va fi prelucrat de Lex / Flex , va conține:

```
%{
#include <string.h>            /* fiind necesar strdup */
#include "simple.tab.h"                 /* cu definitiile atomilor si cu yylval */
%}
DIGIT [0-9]
ID [a-z][a-z0-9]*
%%
":="            { return(ASSGNOP); }
{DIGIT}+        { return(NUMBER); }
do              { return(DO); }
else            { return(ELSE); }
end             { return(END); }
fi              { return(FI); }
if              { return(IF); }
in              { return(IN); }
integer                 { return(INTEGER); }
let             { return(LET); }
read            { return(READ); }
skip            { return(SKIP); }
then            { return(THEN); }
while           { return(WHILE); }
write           { return(WRITE); }
{ID}            { yylval.id = (char *) strdup(yytext);
                  return(IDENTIFIER);}
[ \t\n]+                /* eat up whitespace - pentru a consuma albitura, blancurile, tab-
urile */
.               { return(yytext[0]);}
%%
```

Reprezentarea intermediară

Multe compilatoare convertesc codul sursă într-o reprezentare intermediară în cursul acestei faze de traducere și verificare a restricțiilor contextuale. Pe această reprezentare intermediară operează gramatica cu atribute și optimizatorul arborelui (sau al reprezentării intermediare). În exemplul nostru reprezentarea intermediară implicită este chiar arborele sintactic. Practic el este construit ramură cu ramură pe stivă, pe măsura parcurgerii. Deși deocamdată el este doar implicit, cu mici modificări de program el poate fi construit explicit, apelând pentru fiecare structură sintactică funcții care creează nodul respectiv

sau modifică reprezentarea.

Alte soluții răspândite pentru realizarea reprezentării intermediare includ arborii de sintaxă abstractă, cod cu trei adrese, cunoscutele quadruple și scrierea postfix (operatorii se scriu după operanzi). Sunt folosite în diverse etape intermediare ale compilării.

În acest exemplu de implementare a limbajului **simple** vom sări peste etapa generării reprezentării intermediare a programului (deci și peste optimizarea ei) și vom trece direct la generarea de cod. De altfel principiile ilustrate mai departe la generarea de cod se pot aplica foarte bine și la generarea de reprezentări intermediare (de exemplu la generarea de quadruple).

În urma modificărilor indicate în acest capitol, fișierele dumneavoastră ar putea arăta așa:

Simple.lex

```
%{
#include <string.h>     /* Aici este functia "strdup"      */
#include "simple.tab.h" /* Definitiile atomilor si yylval  */
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
":="            { return(ASSGNOP);   }
{DIGIT}+        { return(NUMBER);    }
do              { return(DO);        }
else            { return(ELSE);              }
end             { return(END);               }
fi              { return(FI);        }
if              { return(IF);        }
in              { return(IN);        }
integer         { return(INTEGER);   }
let             { return(LET);       }
read            { return(READ);              }
skip            { return(SKIP);              }
then            { return(THEN);              }
while           { return(WHILE);     }
write           { return(WRITE);     }
{ID}            { yylval.id = (char *) strdup (yytext);
                  return (IDENTIFIER); }
[ \n\t]+                /* eat up whitespace - pentru a consuma albitura, blancurile, tab-urile */
.               { return(yytext[0]);    }
%%
```

Simple.y

```
/* Yacc/bison file for Simple */
/* Dan Popa, dupa Anthony A Aabey, Comp. Constr with Flex and  Bison, cap 4 pg 14     */
/* C and parser declarations */
```

```
%{
#include <stdlib.h> /*Contine malloc folosit de tab. de simboluri ST.h*/
#include <string.h> /*Contine strcmp folosit de tab. de simboluri ST.h*/
#include <stdio.h>  /* Pentru mesajele de eroare ?  */
#include "ST.h"     /* Modulul cu Tabela de simboluri */
#define YYDEBUG 1   /* For debugging */

int errors;
install (char *sym_name )
{ symrec *s;
s = getsym (sym_name);
if (s == 0)
  s = putsym (sym_name);
else
  {  errors++;
    printf( "%s is already defined \n", sym_name );
  }
}


context_check(char * sym_name )
{ if ( getsym(sym_name) == 0 )
printf ("%s is an undeclared identifier \n", sym_name);
}
    /* Sectiunea continua cu: Parser declarations */
    /* Deoarece scanerul generat de Lex va returna identificatori,   */
    /* iar noi vom utiliza o semantica statica, o inregistrare       */
    /* numita  "semantic record" e necesara pentru a stoca valoarea  */
    /* iar IDENT este asociat cu acest "semantic record".    */
%}

%start program
%union {          /* SEMANTIC RECORD */
        char *id; /*for returning identifiers */
       }
%token LET INTEGER IN
%token SKIP IF THEN ELSE END WHILE DO READ WRITE FI
%token ASSGNOP NUMBER
%token <id> IDENTIFIER   /* Simple identifier */
%left '-' '+'
%left '*' '/'
%right '^'
%%
program : LET declarations IN commands END ;
declarations : /* empty */
  | INTEGER id_seq IDENTIFIER '.' { install($3); }
;
id_seq : /* empty */
  | id_seq IDENTIFIER ','  {install($2); }
;
commands : /* empty */
  | commands command ';'
;
```

```
command : SKIP
  | READ IDENTIFIER                        { context_check ($2); }
  | WRITE exp
  | IDENTIFIER ASSGNOP exp  { context_check ($1); }
  | IF exp THEN commands ELSE commands FI
  | WHILE exp DO commands END
;
exp : NUMBER
  | IDENTIFIER                      { context_check ($1); }
  | exp '<' exp
  | exp '=' exp
  | exp '>' exp
  | exp '+' exp
  | exp '-' exp
  | exp '*' exp
  | exp '/' exp
  | exp '^' exp
  | '(' exp ')'
;
%%
/* C subroutines */
int main (int argc, char * argv[] )
{extern FILE *yyin;
   ++argv; --argc;
   yyin = fopen( argv[0], "r" );
   yydebug = 1 ;
   /* errors = 0 ; */
   yyparse();
}
yyerror (char *s)  /* Called by yyparse on error */
{
  print("%s\n", s);
}
```

Capitolul 5

Optimizarea

Este teoretic posibil să transformăm arborele de derivare pentru a-i reduce dimensiunile sau pentru a oferi generatorului de cod oportunitatea de a produce un cod mai eficient. Întrucât compilatorul din exemplul nostru nu produce (decât implicit) arborele, nu vom ilustra aceste transformări de arbore. În schimb vom prezenta asemenea transformări sub forma codului sursă corespunzător instrucțiunilor neoptimizate și respectiv optimizate. Această prezentare ignoră deliberat faptul că unele compilatoare fac două feluri de optimizări, unele asupra arboreluli altele asupra codului generat. Aceste două feluri de optimizări, nu se pot substitui una pe alta.

Evaluarea constantelor din timpul compilării își propune să precalculeze tot ce se poate pe baza valorilor constante cunoscute la momentul compilării, care apar în expresii. Iată cum s-ar transforma ele:

```
j := 5 + K + 6 ;                --->     j := 11 + K;
m := 4 ; n := m + 3;  --->     m := 4; n := 7;
```

Eliminarea calculelor repetate din bucle, scoaterea lor înafara buclei, reprezintă o altă optimizare. Remarcați paranteza din ciclu ce va trebui evaluată inutil și repetat.

```
while ( index < maxim ) do
        input sales;
        value := salex * ( mark_up + tax );
        output := value;
        index := index + 1;
end;
```

Codul ar putea fi optimizat în felul următor:

```
temp := ( mark_up + tax );
while ( index < maxim ) do
        input sales;
        value := salex * temp;
        output := value;
        index := index + 1;
end;
```

Eliminarea variabilei contor: Cea mai mare parte din timpul de calcul al programelor este ocupat cu execuția calculelor din bucle. Optimizarea buclelor este deci o îmbunătățire semnificativă. Uzual, variabila contor a buclei este folosită doar în interiorul buclei. În acest caz este mai bine ca ea să poată fi stocată într-un registru în loc să fie stocată în memorie. În plus, dacă ea este folosită doar ca indice pentru un vector (ceea ce implică o înmulțire și o adunare la fiecare rotire a buclei) optimizarea constă în a iniția-liza variabila cu adresa de început a vectorului și a o "incrementa" cu lungimea unui element de vector.

O buclă:

```
For I := 1 to 10 do
   A[I] := A[I] + E
```

Devine :

```
For I := adresa primului element din A
         to adresa ultimului element din A
         by dimensiunea unui element din A do
     A[I] := A[I] + E
```

Eliminarea subexpresiilor comune dintr-un șir de expresii prin calcularea valorii lor doar o singură dată:

```
A := 6 * (B+C);
D := 3 + 7 * (B+C);
E := A * (B+C);
```

Devine:

```
TEMP := B + C;
A := 6 * TEMP;
D := 3 * 7 * TEMP;
E := A * TEMP;
```

Înlocuirea unor înmulțiri cu operații mia rapide:

```
2*x --> x + x
2*x --> shift left x
```

Utilizarea identităților matematice:

```
a*b + a*c --> a*(b+c)
a - b --> a + ( - b )
```

Nu vom înzestra implementarea noastră de Simple cu un optimizator de cod.

Capitolul 6

Mașini Virtuale

Un calculator construit efectiv din componente electronice este considerat o mașină de calcul reală. El există fizic. Din punct de vedere al programatorul, setul de instrucțiuni ale acelui hardware definește clar mașina de calcul. Un sistem de operare este un program construit peste resursele oferite de mașina reală, pentru a fi gestionarul acestora și pentru a oferi o serie de alte servicii, de exemplu apelurile de sistem. Serviciile oferite de sistemul de operare, practic instrucțiuni aparte ce pot figura într-un program sau altul, definesc de data aceasta o mașină virtuală. Ea nu există fizic. Nu există circuite electronice specializate care să ofere acele servicii.

Un limbaj de programare pune la dispoziție un set de date și un set de instrucțiuni exprimabile printr-o anume sintaxă. Un hardware capabil să execute direct acele instrucț iuni asupra datelor ar fi un computer real. În teorie este posibil să vorbim de o mașnă Pascal, o mașină Scheme sau una Forth. Cea din urmă e chiar implementată
într-un cip specializat. Practic, pentru progarmator nu contează dacă mașina este reală sau virtuală. Cei care au trecut de la C la Java au constatat că trecerea de la programarea unei mașini reale la programarea uneia virtuale nu-i deranja cu nimic. Iar pentru programator mașina e sinonimă cu limbajul computerului. Limbajul de programare definește practic un computer imaginar, virtual sau araeori real, cu care interacționezi programându-l. Orice mașină virtuală, programabilă, capabilă să manipuleze date, trebuie să aibă o modalitate de a stoca atât datele cât și programul. Dacă o implementezi în alt limbaj trebuie să specifici în acest alt limbaj cum se execută instrucțiunile mașini virtuale asupra datelor ei.

De obicei (obicei care a fost preluat de limbaje interpretate ca Basic, Java) între programator, care dialoghează cu o mașină virtuală bazată pe un limbaj de nivel înalt și mașina virtuală care e sistemul de operare se interpune o altă mașină virtuală. Compilatorul limbajului poate genera altfel de cod (bytecode) și pe acesta trebuie să-l ruleze cineva. Această mașină virtuală este așa-zisul "run-time engine" al limbajului. Complexitatea acesteia poate varia de la nimic (cazul Fortranului și al acelor compilatoare care generează cod direct pentru procesor sau pentru mașina virtuală care este sistemul de operare) la acele sisteme extrem de sofisticate, având manager de memorie și mecanisme de intercomunicare între procese cum e cazul unor limbaje de programare concurente, de exemplu SR.

Run-time system-ul pentru limbajul Simple va include o componentă de procesare capabilă să execute cod, implementată ca un mic interpretor interior și o zonă de date în care valorile atribuite variabilelor sunt accesate printr-un deplasament raportat la începutul zonei. E practic un vector.

Notă: unele programe utilizator pot fi considerate de asemenea ca o clasă aparte de mașini virtuale.

The S(tack) Machine - O mașină virtuală cu stivă

(adaptare după Niklaus Wirth, *Algorithms + Data Structure = Programs*
Prentice-Hall, Englewood Clifs, N.J., 1976.)

S-mașina este o mașină cu stivă proiectată pentru a simplifica implementarea limbajelor cu structură de bloc. Ea oferă o alcare dinamică sub forma unei stive de înregistrări de activare. (Care corespund, nota bene, blocurilor.) Aceste înregistrări de activare sunt legate între ele pentru a permite tipizarea statică și conțin și informațiile de context necesare procedurilor.

Organizarea mașinii: S-mașina constă în două zone de stocare, una dedicată programului și numită **C** (organizată ca un vector read-only) și o zonă de date **S** (organizată ca o stivă). Există patru regiștri; un registru de instrucțiuni **IR** care conține instrucțiunea ce trebuie interpretată, un registru "stack-pointer" **T** care conține adresa ultimului element al stivei, **PC,** acel "program-counter" care conține adresa următoarei instrucțiuni de adus în vederea interpretării și registrul înregistrării de activare curente, **AR,** cel care conține adresa bazei înregistrării de activare aferente procedurii care e în curs de interpretare. Fiecare locație din **C** este capabilă să stocheze o instrucțiune. Fiecare locație din **S** este capabilă să stocheze o adresă sau un întreg. Fiecare instruc-
țiune constă în trei informații, trei câmpuri: un cod al operației și doi parametri.

Setul de instrucțiuni: *S-codurile* formează limbajul mașină al S-mașinii. S-codurile ocupă fiecare câte patru bytes. Primul byte este codul operației (op-code). Există nouă astfel de instrucțiuni (S-coduri) de bază, fiecare cu un cod de operație diferit. Al doilea byte al S-codului conține ori un zero, ori un fel de offset lexical, ori un cod de condiție pentru instrucț iunile de salt condiționat. Ultimii doi bytes luați împreună formează un întreg pe 16 biți cu rolul de operand. El poate fi o constantă ("literal value"), un offset al poziției unei variabile pe stivă pornind de la baza, adresa locației altui S-cod, un număr de operație (!?!), ori un alt număr special al cărui sens e valabil doar pentru un S-cod cu anumit cod de operație.

Comentarii: E normal ca diversele feluri de instrucțiuni (S-coduri) să se distingă prin codul operației. O adunare e ceva diferit de un salt, de exemplu. La rândul lor salturile condiț ionate pot fi condiționate de faptul că o valoare este zero, sau nu este zero deci oricum avem mai multe categorii de instrucțiuni de salt. Al doilea octet diferențiază tocmai instrucț iunile din aceeași familie, cu același cod de operație. Instrucțiunile pot să manipuleze și adrese sau numere de 16 biți ori de 2 bytes deci mai poate fi nevoie de încă doi bytes suplimentari pentru corpul instrucțiunii. Practica arată că la un asemenea limbaj simplu nu e nevoie de mai mulți bytes pentru un S-code. Toate instrucțiunile, indiferent de codul de operație din primul octet n-au nevoie de mai mult de 3 octeți pentru a avea în continuarea op-code-ului informațiile necesare.

Semantica, acțiunea fiecăreia dintre instrucțiuni este descrisă mai jos, folosind un amestec de limbaj natural și notație formală. Această notație inspirată din limbajele de

programare de nivel înalt este folosită pentru a preciza schimbările care au loc în memoria de date și în registrele mașinii cu stivă. Observați că instrucțiunile de acces la date (practic la variabilele locale ale procedurilor) au nevoie de un deplasament raportat la înregistrarea de activare curentă (arată a câta variabilă localâ e implicată) și de diferența de nivel dintre nivelul referinței și nivelul declarației (era posibil de exemplu ca variabila să fi fost declarată într-o procedură exterioară, dar contează exact în a câta deoarece aceasta determină poziția ei pe stivă în înregistrarea de activare corespunzătoare). Apelurile de procedură au nevoie de adresa codului apelat și de asemenea de diferența dintre nivelul referinței și cel al declarației.

Instruction	**Operands**	**Comments**
		I/O operations
READ	0,N	Input integer in to location N: S(N) := Input
WRITE		Output top of stack: Output := S(T); T:= T-1
OPR		*Arithmetic and logical operations*
	0,0	process and function, return operation T:= AR-1; AR:= S(T+2); P:= S(T+3)
ADD		ADD: S(T-1) := S(T-1) + S(T); T := T-1
SUB		SUBTRACT: S(T-1) := S(T-1) - S(T); T := T-1
MULT		MULTIPLY: S(T-1) := S(T-1) * S(T); T := T-1
DIV		DIVIDE: S(T-1) := S(T-1) / S(T); T := T-1
MOD		MOD: S(T-1) := S(T-1) mod S(T); T := T-1
EQ		TEST EQUAL: S(T-1) := if S(T-1) = S(T) then 1 else 0; T:= T-1
LT		TEST LESS THAN: S(T-1) := if S(T-1) < S(T) then 1 else 0; T:= T-1
GT		TEST GREATER THAN: S(T-1) := if S(T-1) > S(T) then 1 else 0; T:= T-1
LD_INT	0,N	LOAD literal value onto stack: T:= T+1; S(T):= N
LD_VAR	L,N	LOAD value of variable at level offset L, base offset N in stack onto top of stack T:= T + 1; S(T):= S(f(L,AR)+N)+3
STORE	L,N	store value on top of stack into variable location at level offset L, base offset N in stack S(f(ld,AR)+os+3):= S(T); T:= T - 1
CAL	L,N	call PROC or FUNC at S-code location N declared at level offset L S(T+1):= f(ld,AR); { Static Link } S(T+2):= AR; { Dynamic Link } S(T+3):= P; { Return Address } AR:= T+1; { Activation Record } P:= cos; { Program Counter } T:= T+3 { Stack Top }
CAL	255,0	call procedure address in memory: POP address, PUSH return address, JUMP to address
DATA	0,NN	ADD NN to stack pointer T := T+NN
GOTO	0,N	JUMP: P := N
JMP_FALSE	C,N	JUMP: if S(T) = C then P:= N; T:= T-1

Aici diferența de nivel static între procedura curentă și cea apelată este notată cu **ld**.
Cu **os** este notat offsetul (deplasamentu) în cadrul înregistrării de activare curente și tot **ld** este diferența de nivel static între înregistrarea de activare curentă și înregistrarea de activare în care va fi stocată valoarea. Funcția f(ld,a) este implementarea următoarei expresii condiționale. O puteți scrie ca o funcție cu acoladă daca doriți.

f(i,a) = if i=0 then a else f(i-1,S(a))

Imaginați-vă că fiecare înregistrare de activare are la începutul ei, la adresa a, un pointer către precedenta înregistrare de activare. Practic S(a) va fi adresa înregistrării de activare precedente.
Se merge pe acest șir de înregistrări de activare înapoi, pas cu pas, pînă ajungem la prima. Adresa ei este la baza stivei.
Funcționarea: Registrii mașinii cu stivă sunt înițializați după cum urmează:

```
P = 0. { Program Counter  - Indica instructiunea curenta}
AR = 0; { Activation Record – Indica inregistrarea de activare curenta}
T = 2; { Stack Top – Virful stivei}
S[0] := 0; { Static Link – Legatura statica}
S[1] := 0; { Static Dynamic Link – Legatura dinamica}
S[2] := 0; { Return Address – Adresa de reintoarcere }
```

Mașina aduce periodic în registrul IR instrucțiunea indicată de registrul PC, incrementează registrul PC și execută instrucțiunea. Totul se repetă pînă când PC-ul ajunge să conțină zero sau este întâlnită și executată o instrucțiune HALT. Într-un pseudocod inspirat de Pascal,(parte a unei masini virtuale) ciclul s-ar scrie:

```
REPEAT
        IR:= C(P);          (* Fetch *)
        PC:= PC+1;
        interpret(IR);      (* Execute*)
UNTIL   IR = "Halt" or PC = 0
```

Această buclă numită și "fetch-execute loop" are de făcut deci două lucruri înainte de a executa (interpreta) o instrucțiune: să aducă op-codul ei în registrul de instrucțiuni și să poziționeze contorul de program PC astfel ca el să indice octetul următor. Aici va găsi subrutina "interpret" byte-ul al doilea. Recuperarea acestor al doilea și la nevoie, al treilea și al patrulea byte din S-codul instrucțiunii curente rămân în sarcina subrutinei "interpret". Ea poate după caz să aducă spre utilizare unu, doi sau trei dintre acești bytes sau chiar pe niciunul, dar oricum va face va trebui să incrementeze registrul PC astfel încât el să indice op-cod-ul următor. Practic după terminarea subrutinei "interpret" PC-ul va arăta o adresă cu 3 bytes mai departe, ea fiind adresa următoarei instrucțiuni. Op-codul de aici va fi încărcat în IR la următorul "fetch" atunci când se reia bucla. Și tot așa mai departe.

Modulul "Mașina Virtuală cu Stivă", SM.h

Implementarea mașinii cu stivă de care avem nevoie e dată în continuare. Setul de instrucțiuni implementate și structura unei instrucțiuni sunt definite după cum urmează:

```
/* OPERATIONS: Internal Representation */

enum code_ops { HALT, STORE, JMP_FALSE, GOTO,

                DATA, LD_INT, LD_VAR,

                READ_INT, WRITE_INT,

                LT, EQ, GT, ADD, SUB, MULT, DIV, PWR };

/* OPERATIONS: External Representation */
char *op_name[] = {"halt", "store", "jmp_false", "goto",
                   "data", "ld_int", "ld_var",
                   "in_int", "out_int",
                   "lt", "eq", "gt", "add", "sub", "mult", "div", "pwr" };

struct instruction
               {
                enum code_ops op;
                int arg;
               };
```

Memoria este separată în două segmente, un segment destinat codului și altul destinat deopotrivă stivei și variabilelor locale create dinamic în cursul execuției.

```
struct instruction code[999];
int stack[999];
```

Urmează definițiile regiștrilor; contorul de instrucțiuni (eng. "program counter") **pc**, registrul de instrucțiuni **ir,** registrul indicator al înregistrării de activare (eng. "activation record") curente numit **ar** și registrul care arată spre vârful stivei, numit **top**. În paginile anterioare fusese numit T dar este vorba de același registru. Este uneori numit și stack pointer - indicator de stivă, din această cauză fiind notat SP la unele procesoare. Registrul **ar** indică începutul porțiunii din stivă care va corespunde ultimei proceduri active. De variabila ch vom avea nevoie mai târziu dar o declarăm deja aici.

```
int                 pc = 0 ;
struct instruction  ir ;
int                 ar = 0 ;
int                 top = 0 ;
char                ch ;
```

Ciclul "fetch-execute" se repetă până la întâlnirea unei instrucțiuni speciale, numită HALT, instrucțiune care există și la procesoarele reale.

```
void fetch_execute_cycle()
{
do {    /* Fetch */
        ir = code[pc++];
```

```
        /* Execute */
        switch (ir.op) {
          case HALT       : printf( "halt\n" );                    break;
          case READ_INT   : printf( "Input: " );
                            scanf( "%ld", &stack[ar+ir.arg] );     break;
          case WRITE_INT  : printf( "Output: %d\n", stack[top--] );      break;
          case STORE      : stack[ir.arg] = stack[top--];                break;
          case JMP_FALSE  : if ( stack[top--] == 0 )
                              pc = ir.arg;                         break;
          case GOTO       : pc = ir.arg;                           break;
          case DATA       : top = top + ir.arg;                    break;
          case LD_INT     : stack[++top] = ir.arg;                       break;
          case LD_VAR             : stack[++top] = stack[ar+ir.arg];              break;


          case LT         : if ( stack[top-1] < stack[top] )
                              stack[--top] = 1;
                            else stack[--top] = 0;                         break;
          case EQ         : if ( stack[top-1] == stack[top] )
                              stack[--top] = 1;
                            else stack[--top] = 0;                         break;
          case GT         : if ( stack[top-1] > stack[top] )
                              stack[--top] = 1;
                            else stack[--top] = 0;
                              top--;                               break;
          case ADD        : stack[top-1] = stack[top-1] + stack[top];
                            top--;
                            break;
          case SUB        : stack[top-1] = stack[top-1] - stack[top];
                            top--;
                            break;
          case MULT       : stack[top-1] = stack[top-1] * stack[top];
                            top--;
                            break;
          case DIV        : stack[top-1] = stack[top-1] / stack[top];
                            top--;
                            break;
          case PWR        : stack[top-1] = stack[top-1] * stack[top];
                            /* Veti corecta aici! */
                            top--;
                            break;
          default         : printf( "%sInternal Error: Memory Dump\n" );
                            break;
        }
    }
while (ir.op != HALT);
}
```

Mașina virtuală de mai sus are nevoie de o corectură deoarece tentativa ei de a executa instrucțiunea PWR duce la calculul unui produs nu la cel al unei ridicări la putere. Realizarea acestei corecturi este lăsată pe seama cititorului, ca exercițiu.

Pentru depanarea buclei "fetch-execute" puteți scrie și următoarele linii de program în locul comentariului /* Fetch */ sau imediat după acesta.

```
printf ( "PC = %23d IR.arg =%8d AR = %23d Top = %3d , %8d \n",
        pc,  ir.arg, ar, top , stack[top] );
```

Modificări ale parserului

Sunt necesare modificări în fișierul anterior cu specificații Yacc/Bison astfel ca programul principal să apeleze mașina virtuală pentru a rula programul care a fost analizat ("parsat") cu succes. Adăugați în secțiunea MAIN a parserului Simple.y linile:

```
printf ( "Parse Completed\n" );
if ( errors == 0 )
{ print_code ();
  fetch_execute_cycle();
}
```

Întreaga funcție main din fișierul Simple.y va arăta așa:

```
/*=============================================================
MAIN
=============================================================*/
main( int argc, char *argv[] )
{ extern FILE *yyin;
   ++argv; --argc;
   yyin = fopen( argv[0], "r" );
   /*yydebug = 1;*/
   errors = 0;
   yyparse ();
   printf ( "Parse Completed\n" );
   if ( errors == 0 )
   { print_code ();
     fetch_execute_cycle();
   }
}
```

Aceste modificări vor permite să fie executat pe mașina virtuală codul generat. De generarea codului se ocupă capitolul următor.

Capitolul 7

Generarea codului

Pe măsură ce programul sursă este parcurs (de către parser) el este tradus într-o formă internă. Frecvent folosită este forma de arbore. Acesta poate fi creat explicit sau parcurs implict, ca o secvență de apeluri de subprograme. În exemplul nostru arborele va fi implicit. Pot exista și alte forme interne. La unele compilatoare, înainte de codul obiect generat, există o variantă a lui într-un limbaj care seamănă cu limbajul de asamblare. Codul în formă internă este în final transformat în cod obiect de către generatorul de cod. Tipic, codul obiect este un program pentru o minusculă mașină virtuală. Alteori el este chiar cod pentru procesorul hardware al sistemului. Pentru execuția programelor scrise în Simple vom folosi mașina virtuală descrisă anterior, formată din trei segmente (de memorie); un **segment de date**, un **segment de cod** și o **stivă**, aceasta fiindu-ne utilă pentru evaluarea expresiilor. La implementări de limbaje care folosesc subprograme stiva servește și pentru controlul execuției acestora și stocarea valorilor locale.

Segmentul de date este cel care stochează valorile variabilelor.Fiecare variabilă are asociată o locație în care se va stoca valoarea ei. Deci o parte din activitatea generatorului de cod va consta în atribuirea de adrese pentru fiecare variabilă.

Segmentul de cod conține secvența tuturor instrucțiunilor programului. Constantele din program sunt stocate tot în segmentul de cod, lucru posibil deoarece valorile lor nu se schimbă iar segmentul de cod este din punct de vedere al mașinii virtuale "read-only".

Stiva expresiilor este o stivă folosită în special pentru a păstra valorile intermediare care apar în timpul evaluării expresiilor. Datorită prezenței ei, mașina virtuală a limbajului Simple este numită o mașină cu stivă sau S-mașină (eng. Stack-machine).

Traducerea declarațiilor

Declarațiile definesc contextul evaluării expresiilor. Pentru a rezerva spațiul pe stivă necesar variabilelor vom folosi instrucțiunea DATA a mașinii virtuale, creată chiar în acest scop. Rolul ei este să "împingă" capul stivei mai departe lăsând loc pe stivă pentru numărul de variabile cerut. În Simple toate variabilele fiind de același tip, spațiul ocupat pe stivă este proporțional cu numărul lor și e suficient să dăm acest număr ca parametru al instrucțiunii DATA. El va fi păstrat în segmentul de cod ca argument al instrucțiunii DATA.
Dar variabilele vor fi stocate în segmentul de date. Acesta fiind practic un vector declarat în C, numărătoarea variabilelor va începe cu 0, prima variabilă ocupând poziția 0, a doua poziția 1, a treia poziția 2 ș.a.m.d. Deci se va compila:

integer x,y,z. ---> DATA 2 iar integer n,x. ---> DATA 1

Traducerea instrucțiunilor

Instrucțiunile de atribuire, if, while, read și write sunt traduse după regulile de mai jos:

```
x := expr               cod pentru expr
                        STORE X

if cond then            cod pentru cond
   S1                   JMP_FALSE L1
else                    cod pentru S1
  S2                    BR L2
end                L1:cod pentru S2
                   L2:

while cond do      L1: cod pentru cond
  S                    JMP_FALSE L2
end                    cod pentru S
                       GOTO L1
                   L2:

read X                 IN_INT X

write expr             cod pentru expr
                       OUT_INT
```

Observați că anumite adrese cum sunt L1 și L2 pentru if sau L2 pentru while nu sunt imediat disponibile, valoarea lor fiind determinată de lungimea unor segvențe de cod care n-au fost încă generate. Ele vor fi completate atunci când valoarea lor exactă devine cunoscută. Operația se numește "back-patching" și va fi executată de o subrutină care se apelează în finalul compilării structurii respective. În exemplul nostru această subrutină se va numi chiar *back_patch*.

Traducerea expresiilor

Expresiile vor fi evaluate folosind stiva. La traducerea lor se vor genera instrucțiuni ale ma șinii virtuale cum sunt LD și LD_INT. Codul pentru un operator va fi generat după codurile pentru operanzi. Practic traducerea transformă expresia obișnuită în forma ei poloneză postfixată (cu operatorul după operanzi) iar evaluarea se va face de către ma-șina virtuală care implementează regulile: 1) constanta sau variabila se pune pe stivă și 2) operatorul scoate din stivă câți operanzi are nevoie, face calculul și pune rezultatul tot în stivă. Cu aceste reguli simpla parcurgere a unei expresii duce la plasarea valorii ei pe stivă. Ca o consecință, pentru mașina virtuală, simpla execuție a codului asociat expresiei va duce la punerea rezultatului în stivă. Traducerea se fece după regulile:

```
constanta      LD_INT constanta
variabila      LD_VAR  variabila
```

```
e1 op e2        cod pentru e1
                cod pentru e2
                cod pentru op
```

Modulul generator de cod : CG.h

Segmentul de date începe cu locația 0 . Rezervarea spațiului în segmentul de date se va face apelând funcția *data_location* care va returna adresa locației nou rezervate. Variabila *data_offset* arată totdeauna următoarea locație liberă în segmentul de date. Întrucât datele ocupă exact o locație în segment, tot ce are de făcut *data_location* este să returneze adresa locației curente libere și apoi să incrementeze variabila *data_offset.*

```
int data_offset = 0;
int data_location() { return data_offset++; }
```

Segemntul de cod are și el locațiile numerotate începând cu zero. Aici însă alocarea de spațiu se va face apelând subrutina *reserve_loc* care va returna adresa locației găsite. (Prima locație liberă.) Dacă nu mai vrem să alocăm dar ne interesează la ce adresă va începe secvența de cod următoare (ca în cazul în care acolo ar fi o etichetă L2) vom apela funcția *gen_label* care nu face decât să returneze valoarea adresei următoarei locații libere,dar fără să aloce spațiul.

```
int code_offset = 0;
int reserve_loc()
{
   return code_offset++;
}

int gen_label()
{
   return code_offset;
}
```

Funcțiile *reserve_loc* și *gen_label* ne vor servi la back-patching-ul codului.
Funcțiile *gen_code* și *back_patch* sunt folosite în procesul generării codului. *gen_code* generează codul unei instrucțiuni în poziția curentă (code_offset) în segmentul de cod. În schimb *back_patch* este folosită la a genera completări ale codului la câte-o adresă rezervată dinainte, adresa fiindu-i dată ca prim parametru.

```
void gen_code( enum code_ops operation, int arg )
{ code[code_offset].op = operation;
  code[code_offset++].arg = arg;
}

void back_patch( int addr, enum code_ops operation, int arg )
{
  code[addr].op = operation;
  code[addr].arg = arg;
}
```

Modificări ale modulului Tabela de Simboluri: ST.h

Formatul articolelor din tabela de simboluri trebuie extins. El va conține pentru fiecare variabilă din tabelă și offset-ul (poziția) acesteia în segmentul de date. Aceasta e adresa zonei în care variabila își păstrează valoarea. Iar funcția *putsym* de adăugare a simbolului în tabelă va fi și ea extinsă pentru a îndeplini sarcina de plasare a offset-ului în articolul din tabelă corespunzător variabilei.

```
struct symrec
{
   char *name;                          /* numele simbolului - identificatorul variabilei */
   int offset;                          /* offset in segmentul de date */
   struct symrec *next;                 /* legatura cu urmatorul din lista */
};
...
symrec * putsym (char *sym_name)
{
  symrec *ptr;
  ptr = (symrec *) malloc (sizeof(symrec));
  ptr->name = (char *) malloc (strlen(sym_name)+1);
  strcpy (ptr->name,sym_name);
  ptr->offset = data_location();
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}

...
```

Modificările parserului (analizorului sintactic): Simple.y

Completăm exemplul nostru anterior cu generarea de cod. Vom extinde definițiile din fișierele Lex și Yacc ale limbajului Simple astfel ca să generăm cod pentru S-mașină. Pentru început vom modifica fițierele Yacc și Lex astfel ca valorile constantelor să fie transmise de la scanner (analizorul lexical) la parser (analizorul sintactic). El e acel care, prin reguli semantice atașate regulilor sintactice, generează cod.
Vom modifica definiția acelui "semantic-record" din fișierul Yacc astfel ca valoarea constantelor să fie returnată ca parte a acestui "semantic-record". Corespunzătoare celor două etichete care apar în codurile instrucțiunilor if și while va exista o structură cu două câmpuri.
Se va defini și un tip special de atomi ("token") numit <lbls> corespunzător lui IF și WHILE. El oferă și spațiu de stocare pentru adresele etichetelor ce-s adăugate în procesul de backpatching. *newlblrec* e funcția care alocă spațiul necesar pentru a stoca valorile etichetelor necesare la backpatching.
Funcția *context_check* va prelua generarea de cod aferentă operației efectuate asupra unei variabile (dar va avea nevoie și de codul operației de implementat !). Codul e generat doar dacă identificatorul există în tabelă.

```
%{#include <stdio.h>                          /* Pentru  I/O */
#include <stdlib.h>                          /* Pentru malloc folosit aici si in ST.h  */
#include <string.h>                          /* Pentru strcmp folosuit in tabela de simboluri St.h
*/
#include "ST.h"                              /* Tabela de Simboluri */
#include "SM.h"                              /* S-Masina */
#include "CG.h"                              /* Generator de cod */
#define YYDEBUG 1                            /* Pentru depanare */
int errors;                                  /* Contor de erori incrementat de CG.h  */

struct lbs                                   /* Pentru etichetele de la if si while */
{
  int for_goto;
  int for_jmp_false;
};
struct lbs * newlblrec()                             /* Alloca spatiu pentru etichete */
{
  return (struct lbs *) malloc(sizeof(struct lbs));
}
install ( char *sym_name )
{
   symrec *s;
   s = getsym (sym_name);
   if (s == 0)
   s = putsym (sym_name);
   else { errors++;
          printf( "%s is already defined\n", sym_name );
   }
}
context_check( enum code_ops operation, char *sym_name )
{ symrec *identifier;
 identifier = getsym( sym_name );
 if ( identifier == 0 )
 { errors++;
   printf( "%s", sym_name );
   printf( "%s\n", " is an undeclared identifier" );
 }
 else gen_code( operation, identifier->offset );
}
%}
%union semrec                                        /* Record-ul semantic */
{
  int intval;                                /* Valoarea cinstantei intregi */
  char *id;                                  /* Identificatorul */
  struct lbs *lbls                           /* Adrese etichetate  pentru  backpatching*/
}
%start program                               /*  Neterminalul  de  la  care  incepe  analiza  e
"program" */
%token <intval>      NUMBER        /* Constante intregi */
%token <id>          IDENTIFIER    /* Identificatori */
%token <lbls>                IF WHILE      /* Etichete pentru backpatching */
```

```
%token SKIP THEN ELSE FI DO END
%token INTEGER READ WRITE LET IN
%token ASSGNOP
%left '-'  '+'                  /* Operatorii asociaza la stinga si au prioritate minima */
%left '*'  '/'                  /* Operatorii asociaza la stinga */
%right '^'                      /* Operatorii asociaza la dreapta si au prioritatea
maxima*/
%%
/* Regulile gramaticii si Actiunile semantice */
%%
/* Subrutine scrise in C*/
```

Analizorul sintactic, parserul, este acum extins cu reguli de generare și îmbinare a porț iunilor de cod. În final, codul generat pentru instrucțiunile if și while trebuie să conțină adresele corecte pentru salturi. Acetsea sunt marcate cu etichete, în regulile de traducere. Deoarece destinațiile exacte ale salturilor nu sunt cunoscute decât după generarea codului întregii structuri, e nevoie de această adăugare a lor, numită *back-patching* (ceea ce s-ar putea traduce prin "peticire înapoi"). Valoarea exactă a adresei destinație pentru salturi va fi adăugată (ca un petec) în cod, după ce, prin compilarea întregii instrucțiuni structurate se va ști precis ce adresă e în punctul vizat, corespunzător etichetei (fie ea L1 sau L2).

Yacc permite pentru fiecare regulă sintactică să adăugăm acțiuni semantice care vor genera cod. Mai mult decăt atât, pentru fiecare element (terminal sau neterminal din regulă) Yacc asociază o variabilă specială al cărui nume începe cu $ și continuă cu numărul elementului din regulă. Capul regulii beneficiază și el de o asemenea variabilă, $$, fără număr. Consultați documentația Yacc/Bison de pe sistemul dumneavoastră, pentru a afla amănunte. Puteți folosi aceste variabile locale procedurilor de analiză sintactică pentru a stoca informații semantice (atribute) despre elementul sintactic respectiv. Dacă aveți de stocat mai multe informații puteți să folosiți variabila ca pe adresa unei structuri de date ce va conține informațiile necesare. Structura de date o veți creea însă dumneavoastră prin alocare dinamică.

Chiar și declarațiile de variabile (statice) vor genera cod: codul generat de o declarație nu face decât să rezerve loc în segmentul de date pentru variabila respectivă.

```
/* Declaratiile pentru C si Parser */
%%
program :       LET
                  declarations
                IN              { gen_code( DATA, data_location()-1 );        }
                  commands
                END             { gen_code( HALT, 0 ); YYACCEPT;    }
;
declarations : /* empty */
                | INTEGER id_seq IDENTIFIER '.' { install( $3 ); }
;
id_seq : /* empty */
```

```
                | id_seq IDENTIFIER ',' { install( $2 ); }
;
```

Instrucțiunile IF și WHILE vor recurge la backpatching.

```
commands : /* empty */
                | commands command ';'
;
command : SKIP
                | READ IDENTIFIER               { context_check( READ_INT, $2 );     }
                | WRITE exp                     { gen_code( WRITE_INT, 0 );          }
                | IDENTIFIER ASSGNOP exp        { context_check( STORE, $1 );                }
                | IF exp                                { $1 = (struct lbs *) newlblrec();
                                                  $1->for_jmp_false = reserve_loc();         }
                  THEN commands                 { $1->for_goto = reserve_loc(); }
                  ELSE                          { back_patch(    $1->for_jmp_false,
                                                                 JMP_FALSE,
                                                                 gen_label() );          }
                        commands
                FI                              { back_patch( $1->for_goto, GOTO, gen_label() ); }
                | WHILE                         { $1 = (struct lbs *) newlblrec();
                                                  $1->for_goto = gen_label();
        }
                        exp                     { $1->for_jmp_false = reserve_loc();
        }
                DO
                        commands
                END                             { gen_code( GOTO, $1->for_goto );
                                                  back_patch(  $1->for_jmp_false,
                                                                 JMP_FALSE,
                                                                 gen_label() );                }
;
```

Codul corespunzător expresiilor e generat de regulile de mai jos:

```
exp : NUMBER                                    { gen_code( LD_INT, $1 ); }
        | IDENTIFIER                            { context_check( LD_VAR, $1 ); }
        | exp '<'  exp                          { gen_code( LT, 0 ); }
        | exp '=' exp                           { gen_code( EQ, 0 ); }
        | exp '>' exp                           { gen_code( GT, 0 ); }
        | exp '+' exp                           { gen_code( ADD, 0 ); }
        | exp '-'  exp                          { gen_code( SUB, 0 ); }
        | exp '*'  exp                          { gen_code( MULT, 0 ); }
        | exp '/'  exp                          { gen_code( DIV, 0 ); }
        | exp '^'  exp                          { gen_code( PWR, 0 ); }
        | '(' exp ')'
;
%%
/* Subprograme C */
```

Modificările Analizorului Lexical, (Scanerului)

O întrebare pe care v-o puneți citind regulile de compilare a expresiilor și semnăturile

funcțiilor implicate este de ce pentru NUMBER valoarea variabilei Yacc $1 este chiar valoarea numerică a constantei iar în cazul unui identificator (IDENTIFIER) este chiar stringul format de literele identificatorului. La urma urmei de ce n-ar fi și la număr la fel ca la identificator ? Să se transmită șirul cifrelor, de ce nu ? Ceea ce nu am specificat este că acest comportament va trebui precizat express în specificațiile scanerului.
Valoarea respectivă va fi stocată în record-ul semantic.

```
%{
#include <string.h>                    /* for strdup */
#include "simple.tab.h"                /* for token definitions and yylval */
%}
DIGIT           [0-9]
ID              [a-z][a-z0-9]*
%%
{DIGIT}+        { yylval.intval = atoi( yytext );
                  return(INT); }
...
{ID}            { yylval.id = (char *) strdup(yytext);
                  return(IDENT); }
[ \t\n]+                /* eat up whitespace */
.               { return(yytext[0]);}
%%
```

O funcție pentru afișarea codului

Pentru a afișa codul generat puteți adăuga la finalul modulului CG.h următoarea funcție. Apelați-o după generarea de cod, înainte de a-l executa, pentru a verifica codul generat. O serie de exemple de mici programe în Simple și codul generat la compilarea lor intenționăm să le anexăm acestui material.

```
void _print_code();
{
  int i = 0;
  while  (i < code_offset) {
     printf("%3ld: %-10s%4ld\n",i,op_name[(int) code[i].op], code[i].arg );
     i++
  }
}
```

Un exemplu

Pentru a ilustra felul cum funcționează generatorul de cod al compilatorului nostru prezentăm un program în Simple și codul generat

```
let
   integer n,x.
in
   read n;
   if n < 10 then x := 1; else skip; fi;
   while n < 10 do x := 5*x; n := n+1; end;
   skip;
   write n;
   write x;
end
```

Figura 7.1: Program Simple

Un exemplu de cod produs în urma compilării:

```
0: data        1
1: in_int      0
2: ld_var      0
3: ld_int      10
4: lt          0
5: jmp_false   9
6: ld_int      1
7: store       1
8: goto        9
9: ld_var      0
10: ld_int     10
11: lt         0
12: jmp_false  22
13: ld_int     5
14: ld_var     1
15: mult       0
16: store      1
17: ld_var     0
18: ld_int     1
19: add        0
20: store      0
21: goto       9
22: ld_var     0
23: out_int    0
24: ld_var     1
25: out_int    0
26: halt       0
```

Figura 7.2: Cod generat

Capitolul 8

Optimizarea de cod după generare

Chiar și după generarea codului există optimizări care se mai pot face. Nu uitați că porțiunile de cod generat s-au alăturat așa cum impunea sursa programul de compilat dar nimic nu garantează că îmbinarea lor este optimă. De asemenea anumite constante care apar în instrucțiunile generate pot sugerea înlocuirea întregii instrucțiuni cu alta mai scurtă sau mai rapidă. Unele înmulțiri sau împărțiri, de exemplu cele cu 2 pot fi înlocuite cu alte operații, cum sunt cele de deplasare a biților. Instrucțiuni de adunare cu zero , cele de înmulțire cu 1 și alte operații devenite inutile din cauza anumitor valori ale constantelor pot fi înlocuite ori cu NOP (no operation) sau chiar eliminate. Adunarea sau scăderea cu 1 beneficiază și ele de instrucțiuni speciale mai rapide, pe care majoritatea procesoarelor hardware le oferă. Iar anumite operații cu variabile pot fi înlocuite cu operații realizate cu ajutorul regiștrilor procesorului, mărindu-se astfel viteza codului generat.

Pentru a-l optimiza, întregul cod generat va fi parcurs având la fiecare moment în vizor câteva instrucțiuni vecine (ca și cum ați vedea o cameră privind pe gaura cheii). Dacă ele se potrivesc cu anumite șabloane cunoscute de optimizator (procesul e numit chiar "peephole optimization" - "optimizare tip gaura cheii") atunci acesta le va înlocui cu alte secvențe de cod mai rapide (sau mai eficiente din alt punct de vedere, de exemplu mai scurte). Un exemplu tipic este eliminarea unor secvențe PUSH – POP care nu fac decât să pună și să scoată aceeași valoare din stivă sau inlocuirea acestora cu operații de transfer între registri (dacă sunt mai mulți cu funcții asemănătoare) sau între regiștri și memorie.

Nu vom ilustra în exemplul nostru de compilare a limbajului Simple asemenea tehnici de optimizare. De altfel limbajul mașină extrem de simplu al S-mașinii noastre nu oferă prea multe posibilitați de a face asemenea optimizări, fiind un limbaj minimalist. Cu totul alta este situația atunci când se generează cod pentru un procesor hardware real, care este oferit de fabricant cu un set mult mai vast de instrucțiuni, dintre care unele pot înlocui uneori cu suces pe altele, oferind și avantaje.

Capitolul 9

Lecturi suplimentare

În ultimele versiuni (15 sept. 2003 și 25 feb. 2004) ale documentului care a stat la baza acestui volum, document oferit de prof. Anthony A. Aaby prin Internet capitolul 9 nu cuprindea decât două rânduri care recomandă alte materiale - nenumite - privind Lex și Yacc precum și lucrările - tot necitate - scrise de un autor pe nume Pratt, privitoare la ma șinile virtuale. Vom suplini această omisiune indicând alte materiale accesibile în România sau pe Internet pe care le-am selectat în procesul elaborării acestui volum derivat din lucrarea domnului Anthony A. Aaby.

Despre S.O. UNIX și componentele sale, incluzând Yacc și Lex puteți consulta capitolul 11 al volumui de mai jos, scris de Valentin Cristea și colectivul său de coautori. Explica-
ții suplimentare despre folosirea variabilelor $$ și $1,$2 în regulile semantice se găsesc la pagina 254.

1) Valentin Cristea, Alexandru Pănoiu, Eugenia Kalisz, Irina Athanasiu, Lorina Negreanu, Silviu Călinoiu, Florin Baboescu - *UNIX*, București 1993, Ed.Teora , ISBN 973-601-105-4

O carte accesibilă, disponibilă și prin Internet după ce în prealabil fusese publicată de International Thomson în 1996 aparține profesorului P.D.Terry de la Rhodes University. În anexă este un întreg compilator pentru limbajul Clang, scris în C. Extrem de instructivă. Oferă un bun suport teoretic dar recomandă alt generator de compilatoare, Coco/R. Se poate găsi căutând numele profesorului Terry cu un motor de căutare. În prefața cărții sunt indicate site-urile: http://www.scifac.ru.ac.za/compilers/
și http://cs.ru.ac.za/home/cspt/compbook.htm.

2) P.D.Terry, *Compilers and Compiler Generators*, Rhodes University, 1996, Published by International Thomson

Urmează un concis dar frumos manual de folosire pentru Lex și Yacc, utilizabil cu puțin effort și pentru clonele lor de pe platforma Linux, Flex-ul și Bison-ul. Era disponibil în format pdf pe site-ul epapers.com. Compilatorul prezentat acolo foloseste un arbore creat explicit nu implicit ca în exemplul din acest volum. Acest arbore poate fi exploatat de un evaluator, interpretor-ul de arbori, obținîndu-se un interpretor pentru limbajul propus. Altă variantă este generarea de cod pe baza arborelui. Un exercițiu interesant este implementarea unui compilator ca cel din lucrarea amintită (mai jos) folosind o distribuție Linux care include programele Flex și Bison.

3) Thomas Niemann, *A Compact Guide to Lex & Yacc*, epapers.com

Pentru cunoscătorii de Pascal interesați să scrie fără generator un mic compilator putem

recomanda un volum practic, de tipul "learn-by-doing" scris de un specialist în fizică dar pasionat de compilatoare, domnul Jack W.Crenshaw. Există diverse versiuni pe Internet ale serialului său despre scrieirea practică a unui compilator, fie ca arhive zip cu plain-text sau ca fișier pdf. Am avut la dispoziție versiunea 1.8 în format pdf. E un mod interesant de a face cunoștință cu un compilator și cu problemele realizării acestuia. Punctul de vedere este cel al practicianului care nu-și pierde mult timp cu teoria. Ceea ce face ca lectura unui volum de teorie înpreună cu acesta să fie cu atât mai interesantă.

4) Jack W. Crenshaw, *Compiler Building Tutorial*, ver. 1.8, Aprill 11,2001.

În cele din urmă, dar poate cel mai citat, celebrul volum scris de Alfred Aho, Ravi Sethi și Jeffrey Ullmann în 1986 și republicat apoi în 1988. Se poate găsi la unele biblioteci din România.

5) Alfred Aho, Ravi Sethi și Jeffrey Ullmann, Compilers, Principles, Techniques and Tools, 1986, Addison-Wesley, Reading, Massachusetts.

Capitolul 10

Exerciții

Exercițiile care urmează sunt variate ca dificultate. Pentru fiecare se cere să determinați ce modificări trebuie făcute gramaticii, tabelei de simboluri și mașinii cu stivă. Sunt câteva dintre exercițiile oferite de prof. A.A.Aaby elevilor săi.

1. Re-implementați tabela de simboluri sub formă de arbore binar de căutare.

2. Re-implementați tabela de simboluri ca tabelă hash.

3. Re-implementați tabela de simboluri, generatorul de cod și mașina cu stivă sub formă de clase C++**.**(Noile versiuni de Flex și Bison pot produce și ele cod sub formă de clase C++.)

4. Extindeți compilatorul cu extensiile date mai jos. Extensiile cer modificarea scanerului pentru a manipula noii atomi și modificarea parserului pentru a gestiona gramatica extinsă.

(a) Declarații: Schimbați procesarea semanticii identificatorilor astfel încât aceștia să (nu) necesite declarare anterioară.

(b) Constante și variabile reale: Extindeți tabela de simboluri astfel încât subrutinele care-o actualizează să stocheze și atributul de tip pentru fiecare identificator. Extindeți rutinele semantice ale generatorului de cod astfel ca acesta să genereze codul corespunzător tipului de constante și variabile pe care le întâlnește, variabile pe care aceste rutine le primesc ca parametri.

(c) Înmulțire, împărțire (și ridicare la putere !). Faceți schimbările necesare în rutinele semantice pentru a se ocupa corect de generarea de cod.

(d) Instrucțiunile **if** și **while:** Rutinele semantice trebuie să genereze propriile teste și salturi.

(e) Proceduri fără parametri: Tabela de simboluri trebuie extinsă pentru a manipula declarații imbricate iar rutinele semantice trebuie extinse pentru a genera cod pentru asigurarea transferului controlului la fiecare punct de apel și de asemenea la începutul și sfârșitul corpului procedurii. Indicație: Se va folosi registrul AR al mașinii virtuale și instrucț iunile care operează cu el.

Opțional puteți adăuga:

(a) Un interpretor pentru codul produs de compilator.

(b) Înlocuiți parserul ghidat de tabele, cu un parser descendent recursiv.

5. Extindeți compilatorul. O descriere este (sau a fost) inclusă în dosarul cs360/compiler tools de pe site-ul cs.wwc.edu. Pe scurt, sunt cerute următoarele extensii:

(a) Extinderea scanerului pentru a manipula noii atomi. Folosiți un generator pentru a produce noile tabele.

(b) Declarații pentru variabile întregi și reale.

(c) Constante întregi, expresii cu întregi, operații de I/O pentru întregi și de ieșire pentru stringuri.

(d) Instrucțiunea **loop** cu **exit** și adăugarea lui **else** și **elsif** la instrucțiunea **if.**

(e) Proceduri recursive cu parametri.

(f) Declararea recordurilor și utilizarea câmpurilor.

(g) Declararea vectorilor și utilizarea elementelor din vectori.

(h) Utilizarea modulelor și folosirea identificatorilor cu calificare.

6. Compilatorul următor trebuie să-l scrieți complet, de la zero. Lista de mai jos enumeră caracteristicile limbajului, prezentând întâi un subset de bază absolut necesar. Celelalte caracteristici care urmează sunt opționale.

Setul de bază:

(a) Tipurile întreg, real, boolean.

(b) Expresii simple cu întregi reali și valori booleene folosind operatorii: **+,- ,*, /, not, and, or, abs, mod, **, <,<=,>,>=,=, /=**

(c) Operații de intrare pentru valori întregi, reale și booleene

(d) Operații de ieșire pentru stringuri și expresii întregi, reali, valori booleene. Fără formatare.

(e) Structuri de bloc, incluzând declarații de variabile locale și de constante

(f) Atribuirea

(g) Instrucțiunile **if**, **loop** și **exit**

(h) Proceduri și funcții fără argumente, cu rezultat scalar, inclusiv imbricate și cu variabile non-locale.

Setul opțional:

Nu este tradus aici. Îl puteți găsi în lucrarea originală a prof. A.A.Aaby. Reamintim că un compilator scris de la zero dar care generează cod pentru un procesor real Motorola găsiți în volumul lui **Jack W. Crenshaw, *Compiler Building Tutorial*, ver. 1.8, Aprill 11,2001,** disponibil pe Internet.

Anexa A

Limbajul Simple - Implementarea completă

A1. Parserul Simple.y

Dacă ați realizat fișierul cu specificații Yacc/Bison, conform cu indicațiile din capitolele precedente, ar trebui să obțineți un text similar cu cel de mai jos, cu excepția comentariilor. Decomentând rândul `/*yydebug = 1;*/` din funcția *main* veți obține la execuție mult mai multe detali despre funcționarea parserului.

```
%{/*************************************************************************
                        Compiler for the Simple language
*************************************************************************/
/*=========================================================================
            C Libraries, Symbol Table, Code Generator & other C code
=========================================================================*/
#include <stdio.h>             /* For I/O                                 */
#include <stdlib.h>            /* For malloc here and in symbol table     */
#include <string.h>            /* For strcmp in symbol table        */
#include "ST.h"                /* Symbol Table                            */
#include "SM.h"                /* Stack Machine                           */
#include "CG.h"                /* Code Generator                          */
#define YYDEBUG 1              /* For Debugging                           */
int errors;                    /* Error Count                             */
/*-------------------------------------------------------------------------
                       The following support backpatching
-------------------------------------------------------------------------*/
struct lbs                     /* Labels for data, if and while           */
 {
  int for_goto;
  int for_jmp_false;
 };
struct lbs * newlblrec()       /* Allocate space for the labels           */
{
  return (struct lbs *) malloc(sizeof(struct lbs));
}
/*-------------------------------------------------------------------------
               Install identifier & check if previously defined.
-------------------------------------------------------------------------*/
install ( char *sym_name )
{
  symrec *s;
  s = getsym (sym_name);
  if (s == 0)
    s = putsym (sym_name);
  else { errors++;
    printf( "%s is already defined\n", sym_name );
  }
}
/*-------------------------------------------------------------------------
                    If identifier is defined, generate code
-------------------------------------------------------------------------*/
```

```
context_check( enum code_ops operation, char *sym_name )
{ symrec *identifier;
  identifier = getsym( sym_name );
  if ( identifier == 0 )
  { errors++;
    printf( "%s", sym_name );
    printf( "%s\n", " is an undeclared identifier" );
  }
  else gen_code( operation, identifier->offset );
}
/*=============================================================================
                                SEMANTIC RECORDS
=============================================================================*/
%}
%union semrec                   /* The Semantic Records              */
{
  int intval;                   /* Integer values                         */
  char *id;                     /* Identifiers                            */
  struct lbs *lbls;             /* For backpatching                       */
}
/*=============================================================================
                                    TOKENS
=============================================================================*/
%start program
%token <intval>   NUMBER      /* Simple integer                         */
%token <id>       IDENTIFIER  /* Simple identifier                      */
%token <lbls>     IF WHILE    /* For backpatching labels                */
%token SKIP THEN ELSE FI DO END
%token INTEGER READ WRITE LET IN
%token ASSGNOP
/*=============================================================================
                               OPERATOR PRECEDENCE
=============================================================================*/
%left '-' '+'
%left '*' '/'
%right '^'
/*=============================================================================
                     GRAMMAR RULES for the Simple language
=============================================================================*/
%%
program :   LET
              declarations
            IN { gen_code( DATA, data_location() - 1 ); }
              commands
            END { gen_code( HALT, 0 ); YYACCEPT; }
;
declarations : /* empty */
            | INTEGER id_seq IDENTIFIER '.' { install( $3 ); }
;
id_seq : /* empty */
            | id_seq IDENTIFIER ',' { install( $2 ); }
;
commands : /* empty */
            | commands command ';'
```

```
;
command : SKIP
      | READ IDENTIFIER         { context_check( READ_INT, $2 );           }
      | WRITE exp               { gen_code( WRITE_INT, 0 );                }
      | IDENTIFIER ASSGNOP exp { context_check( STORE, $1 );               }
      | IF exp                       { $1 = (struct lbs *) newlblrec();
                                       $1->for_jmp_false = reserve_loc(); }
        THEN commands           { $1->for_goto = reserve_loc();            }
        ELSE                    { back_patch(    $1->for_jmp_false,
                                                 JMP_FALSE,
                                                 gen_label() );             }

        commands
        FI                { back_patch( $1->for_goto, GOTO, gen_label() ); }
      | WHILE             { $1 = (struct lbs *) newlblrec();
                            $1->for_goto = gen_label();              }
          exp             { $1->for_jmp_false = reserve_loc();             }
        DO
          commands
        END               { gen_code( GOTO, $1->for_goto );
                            back_patch( $1->for_jmp_false,
                                        JMP_FALSE,
                                        gen_label() );                      }
;
exp : NUMBER              { gen_code( LD_INT, $1 );                          }
| IDENTIFIER              { context_check( LD_VAR, $1 );                     }
| exp '<' exp             { gen_code( LT, 0 ); }
| exp '=' exp             { gen_code( EQ, 0 ); }
| exp '>' exp             { gen_code( GT, 0 ); }
| exp '+' exp             { gen_code( ADD, 0 ); }
| exp '-' exp             { gen_code( SUB, 0 ); }
| exp '*' exp             { gen_code( MULT, 0 ); }
| exp '/' exp             { gen_code( DIV, 0 ); }
| exp '^' exp             { gen_code( PWR, 0 ); }
| '(' exp ')'
;
%%
/*=============================================================================
                                    MAIN
=============================================================================*/
main( int argc, char *argv[] )
{ extern FILE *yyin;
  ++argv; --argc;
  yyin = fopen( argv[0], "r" );
  /*yydebug = 1;*/
  errors = 0;
  yyparse ();
  printf ( "Parse Completed\n" );
  if ( errors == 0 )
  { print_code ();
    fetch_execute_cycle();
  }
}
/*=============================================================================
```

```
                                    YYERROR
============================================================================*/
yyerror ( char *s ) /* Called by yyparse on error */
{
  errors++;
  printf ("%s\n", s);
}
/************************** End Grammar File ************************/
```

A.2. Indicații de lucru

Am transcris mai jos comenzile pe care va trebui să le dați sistemului dumneavoastră Linux. Am testat această secvență pe un RedHat Linux 5.2 și nu am găsit deosebiri față de comenzile propuse de dl. A.A.Aaby. Singurele deosebiri au fost că promptul sistemului meu Linux nu este ">" ci "$" (deși el se poate modifica folosind comanad prompt) iar versiunea de compilator gcc folosită mai avea nevoie de o opțiune în linia de comandă (-lfl).

```
> bison -d Simple.y
     sau
> bison -dv Simple.y
Simple.y contains 39 shift/reduce conflicts.
> gcc -c Simple.tab.c
> flex Simple.lex
> gcc -c lex.yy.c
> gcc -o Simple Simple.tab.o  lex.yy.o  -lm  -lfl
> Simple test_simple
Parse Completed
0: data         1
1: in_int       0
2: ld_var       0
3: ld_int       10
4: lt           0
5: jmp_false    9
6: ld_int       1
7: store        1
8: goto         9
9: ld_var       0
10: ld_int      10
11: lt          0
12: jmp_false   22
13: ld_int      5
14: ld_var      1
15: mult        0
16: store       1
17: ld_var      0
18: ld_int      1
19: add         0
20: store       0
21: goto        9
22: ld_var      0
23: out_int     0
24: ld_var      1
25: out_int     0
26: halt        0
```

A.3. Scanerul: Simple.lex

```
/*************************************************************************
                         Scanner for the Simple language
*************************************************************************/
%{
/*=======================================================================
                       C-libraries and Token definitions
=======================================================================*/
#include <string.h>            /* for strdup */
/*#include <stdlib.h> */       /* for atoi */
#include "Simple.tab.h"  /* for token definitions and yylval */
%}
/*=======================================================================
                                TOKEN Definitions
=======================================================================*/
DIGIT [0-9]
ID [a-z][a-z0-9]*
/*=======================================================================
        REGULAR EXPRESSIONS defining the tokens for the Simple language
=======================================================================*/
%%
":="         { return(ASSGNOP);       }
{DIGIT}+     { yylval.intval = atoi( yytext );
               return(NUMBER);        }
do           { return(DO);            }
else         { return(ELSE);          }
end          { return(END);           }
fi           { return(FI);            }
if           { return(IF);            }
in           { return(IN);            }
integer      { return(INTEGER);       }
let          { return(LET);           }
read         { return(READ);          }
skip         { return(SKIP);          }
then         { return(THEN);          }
while        { return(WHILE);         }
write        { return(WRITE);         }
{ID}         { yylval.id = (char *) strdup(yytext);
               return(IDENTIFIER);    }
[ \t\n]+     /* eat up whitespace */
.            { return(yytext[0]);     }
%%
int yywrap(void){}
/************************* End Scanner File ****************************/
```

A.4 Tabela de simboluri: St.h

```
/*************************************************************************
                              Symbol Table Module
*************************************************************************/
/*=======================================================================
                                 DECLARATIONS
=======================================================================*/
/*-----------------------------------------------------------------------
                              SYMBOL TABLE RECORD
-----------------------------------------------------------------------*/
struct symrec
{
  char *name;                  /* name of symbol       */
  int offset;                  /* data offset          */
  struct symrec *next;         /* link field           */
};
typedef struct symrec symrec;
/*-----------------------------------------------------------------------
                              SYMBOL TABLE ENTRY
-----------------------------------------------------------------------*/
symrec *identifier;
/*-----------------------------------------------------------------------
                                SYMBOL TABLE
                      Implementation: a chain of records.
-----------------------------------------------------------------------*/
symrec *sym_table = (symrec *)0; /* The pointer to the Symbol Table */
/*=======================================================================
                           Operations: Putsym, Getsym
=======================================================================*/
symrec * putsym ();
symrec * getsym ();
symrec * putsym (char *sym_name)
{
  symrec *ptr;
  ptr = (symrec *) malloc (sizeof(symrec));
  ptr->name = (char *) malloc (strlen(sym_name)+1);
  strcpy (ptr->name,sym_name);
  ptr->offset = data_location();
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}
symrec * getsym (char *sym_name)
{
  symrec *ptr;
  for ( ptr = sym_table;
        ptr != (symrec *) 0;
        ptr = (symrec *)ptr->next )
    if (strcmp (ptr->name,sym_name) == 0)
      return ptr;
  return 0;
}
/*********************** End Symbol Table ************************/
```

A.5 Generatorul de cod: CG.h

```
/*************************************************************************
                                Code Generator
*************************************************************************/
/*-----------------------------------------------------------------------
                                Data Segment
-----------------------------------------------------------------------*/
int data_offset = 0;           /* Initial offset                         */
int data_location()            /* Reserves a data location               */
{
  return data_offset++;
}
/*-----------------------------------------------------------------------
                                Code Segment
-----------------------------------------------------------------------*/
int code_offset = 0;           /* Initial offset                         */
int reserve_loc()              /* Reserves a code location               */
{
  return code_offset++;
}
int gen_label()                /* Returns current offset                 */
{
  return code_offset;
}

                               /* Generates code at current location     */
void gen_code( enum code_ops operation, int arg )
{
  code[code_offset].op = operation;
  code[code_offset++].arg = arg;
}
                               /* Generates code at a reserved location  */
void back_patch( int addr, enum code_ops operation, int arg )
{
  code[addr].op = operation;
  code[addr].arg = arg;
}
/*-----------------------------------------------------------------------
                              Print Code to stdio
-----------------------------------------------------------------------*/
void print_code()
{
  int i = 0;
  while (i < code_offset) {
    printf("%3ld: %-10s%4ld\n",i,op_name[(int) code[i].op], code[i].arg );
    i++;
  }
}
/************************* End Code Generator *************************/
```

A.6 Mașina cu stivă : SM.h

```
/*************************************************************************
                                Stack Machine
*************************************************************************/
/*=======================================================================
                                DECLARATIONS
=======================================================================*/
/* OPERATIONS: Internal Representation */

enum code_ops {    HALT, STORE, JMP_FALSE, GOTO,
                   DATA, LD_INT, LD_VAR,
                   READ_INT, WRITE_INT,
                   LT, EQ, GT, ADD, SUB, MULT, DIV, PWR };

/* OPERATIONS: External Representation */

char *op_name[] = {      "halt", "store", "jmp_false", "goto",
                         "data", "ld_int", "ld_var",
                         "in_int", "out_int",
                         "lt", "eq", "gt", "add", "sub", "mult", "div", "pwr" };
struct instruction
  {
    enum code_ops op;
    int arg;
  };

/* CODE Array */

struct instruction code[999];

/* RUN-TIME Stack */

int stack[999];

/*-----------------------------------------------------------------------
                                Registers
-----------------------------------------------------------------------*/
int                     pc = 0;
struct instruction      ir;
int                     ar = 0;
int                     top = 0;
char ch;
/*=======================================================================
                             Fetch Execute Cycle
=======================================================================*/
void fetch_execute_cycle()
{   do { /*printf( "PC = %3d IR.arg = %8d AR = %3d Top = %3d,%8d\n",
                   pc, ir.arg, ar, top, stack[top]); */
         /* Fetch */
         ir = code[pc++];
         /* Execute */
         switch (ir.op) {
```

```
            case HALT      :  printf( "halt\n" ); break;
            case READ_INT  :  printf( "Input: " );
                              scanf( "%ld", &stack[ar+ir.arg] ); break;
            case WRITE_INT :  printf( "Output: %d\n", stack[top--] ); break;
            case STORE     :  stack[ir.arg] = stack[top--]; break;
            case JMP_FALSE :  if ( stack[top--] == 0 )
                              pc = ir.arg;
                              break;
            case GOTO      :  pc = ir.arg; break;
            case DATA      :  top = top + ir.arg; break;
            case LD_INT    :  stack[++top] = ir.arg; break;
            case LD_VAR    :  stack[++top] = stack[ar+ir.arg]; break;
            case LT        :  if ( stack[top-1] < stack[top] )
                                stack[--top] = 1;
                              else stack[--top] = 0;
                              break;
            case EQ        :  if ( stack[top-1] == stack[top] )
                                stack[--top] = 1;
                              else stack[--top] = 0;
                                break;
            case GT        :  if ( stack[top-1] > stack[top] )
                                stack[--top] = 1;
                              else stack[--top] = 0;
                              break;
            case ADD       :  stack[top-1] = stack[top-1] + stack[top];
                                    top--;
                                    break;
            case SUB       :  stack[top-1] = stack[top-1] - stack[top];
                                    top--;
                                    break;
            case MULT      :  stack[top-1] = stack[top-1] * stack[top];
                                    top--;
                                    break;
            case DIV       :  stack[top-1] = stack[top-1] / stack[top];
                                    top--;
                                    break;
            case PWR       :  stack[top-1] = stack[top-1] * stack[top];
                                    top--;
                                    break;
            default : printf( "%sInternal Error: Memory Dump\n" );
                                    break;
          }
      }
  while (ir.op != HALT);
}
/************************** End Stack Machine *************************/
```

Așa cum am precizat în capitolele precedente, instrucțiunea PWR a mașinii virtuale trebuie să facă ridicare la putere nu înmulțire. Porțiunea din text care trebuie înlocuită este evidențiată.

A.7 Un exemplu de program: test_simple

```
let
   integer n,x.
in
   read n;
   if n < 10 then x := 1; else skip; fi;
   while n < 10 do x := 5*x; n := n+1; end;
   skip;
   write n;
   write x;
end
```

Și codul produs, de data aceasta alăturate:

```
0: data        1
1: in_int      0
2: ld_var      0
3: ld_int      10
4: lt          0
5: jmp_false   9
6: ld_int      1
7: store       1
8: goto        9
9: ld_var      0
10: ld_int     10
11: lt         0
12: jmp_false  22
13: ld_int     5
14: ld_var     1
15: mult       0
16: store      1
17: ld_var     0
18: ld_int     1
19: add        0
20: store      0
21: goto       9
22: ld_var     0
23: out_int    0
24: ld_var     1
25: out_int    0
26: halt               0
```

Figura 7.2: Codul generat

Anexa D

Instalarea programelor Flex și Bison

Pentru a putea experimenta folosirea Flex-ului și Bison-ului trebuie să instalați aceste programe pe computerul dumneavoastră dotat cu Linux. Mai ales dacă nu sunteți familiarizat cu instalarea programelor sub Linux sau dacă Linux-ul dumneavoastră a venit de la magazin gata instalat dar fără Flex și Bison (ceea ce se întâmplă în alte țări ale Europei și în SUA), acest capitol vă poate fi de folos.

Anexa descrie felul cum puteți instala Flex-ul și Bison-ul pe un computer dotat cu o distribuție Linux care folosește pachete RPM. (Red Hat Packages).
Asemenea distribuții Linux sunt: Red Hat Linux, Mandriva Linux (fostă Mandrake Linux), PC Linux OS, Openna Linux, Lycoris, Corel Linux (o veche distribuție care nu mai este dezvoltată) și multe multe altele.

Prima metodă:

Este suficient să aveți acces la o consolă și privilegii de root (manager de sistem) pentru a putea instala pachetele cu comanda **rpm,** dată la promptul **#:**

```
# rpm -Uvh <fișierul-pachet-dorit>
```

unde <fișierul-pachet-dorit> va fi numele pachetului pe care doriți să-l instalați.
Se pot instala mai multe pachete cu aceeași linie de comandă, lucru util mai ales dacă depind unele de altul (ceea ce nu este cazul aici.).
Ați putea uneori dori să folosiți alte combinații de switch-uri în comandă, cum ar fi **-ivh** sau să adăugați **--force**. Consultați manualul cu comanda **man rpm** sau help-ul comenzii rpm tastând **rpm –help | more** pentru a afla detalii despre folosirea rpm.
Din cauza numerelor de versiune numele pachetelor rpm par complicate și lungi.
Truc util: Unii utilizatori scriu doar începutul numelui pachetului și apasă tasta TAB pentru ca sistemul să scrie continuarea automat.
Dacă sunt mai multe pachete cu nume similare în același director puteți obține lista lor cu comanda ls, dată în directorul unde sunt pachetele:

```
# ls <prefix-pachet>*
```

Încercați:

```
# ls flex*
```

Sau:

```
# ls bison*
```

Cele de mai sus combinate cu ideea de substituție ne permit să scriem o comandă de instalare capabilă să instaleze toate pachetele al căror nume încep cu un prefix dat. Folosiți comenzi ca:

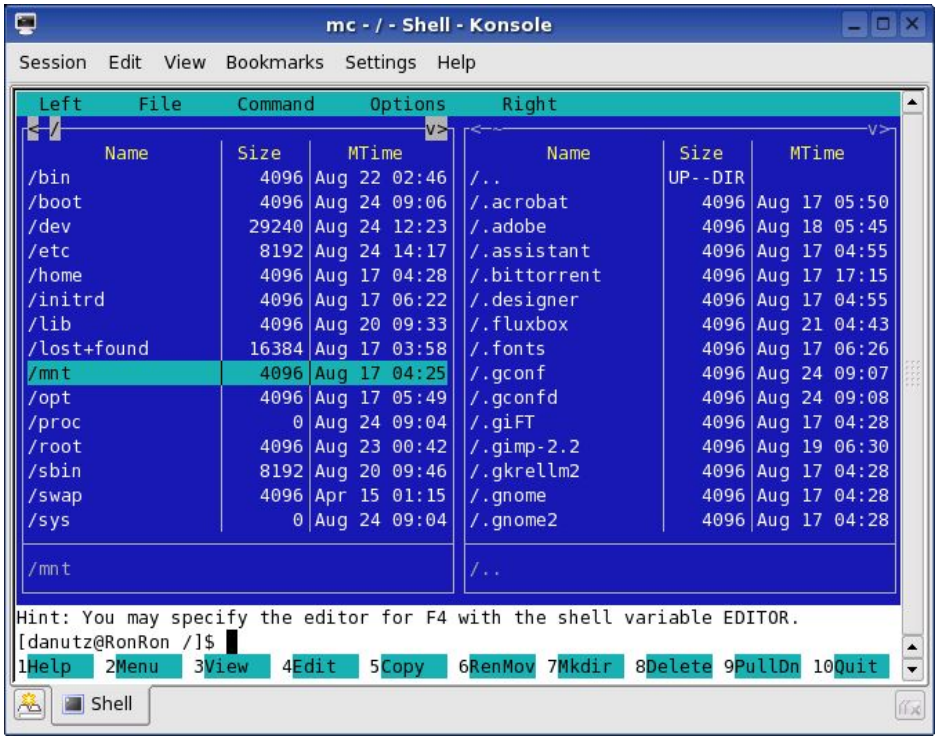
```
# rpm -Uvh `ls flex*`
# rpm -Uvh `ls bison*`
```

Bineînțeles și în aceste linii de comandă se pot adăuga alte switch-uri decât -Uvh , așa cum este **--force** citat mai sus. El vă ajută să reușiți o instalare care altfel ar fi eșuat, de exemplu din lipsa sincronizării dintre lista pachetelor din sistem și fișierele efectiv existente.
Acest truc este util atunci cînd aveți de instalat un grup mai mare de pachete cu nume asemănătoare, prefixate la fel. Dar atenție: riscați să instalați mai multe pachete decât ați fi dorit dacă prefixul este prea scurt.

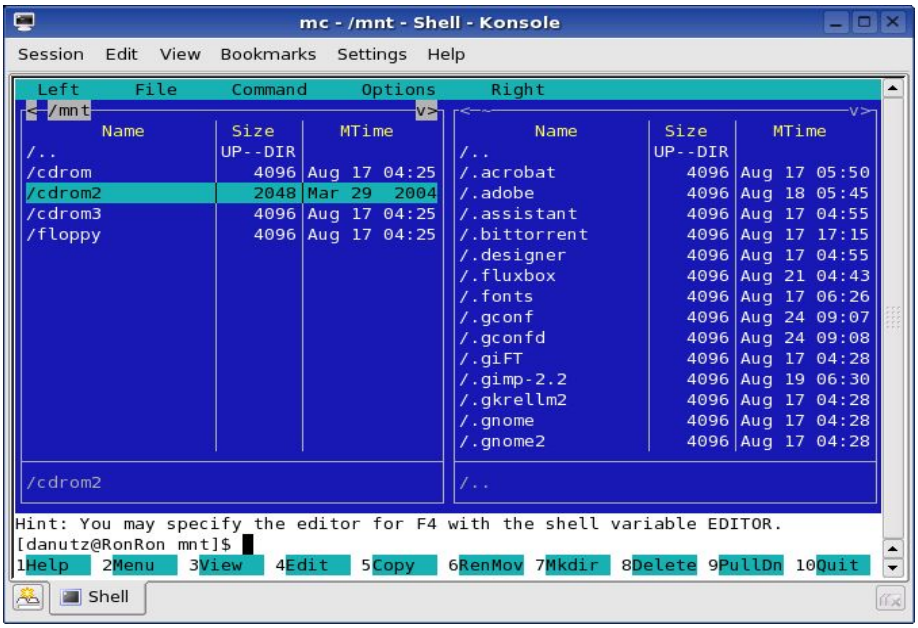
Dar există și alte instrumente capabile de a instala pachete rpm. Unul din ele este binecunoscuta clonă de Norton Commander numită Midnight Commander (mc).

Instalarea pachetelor cu mc decurge astfel:

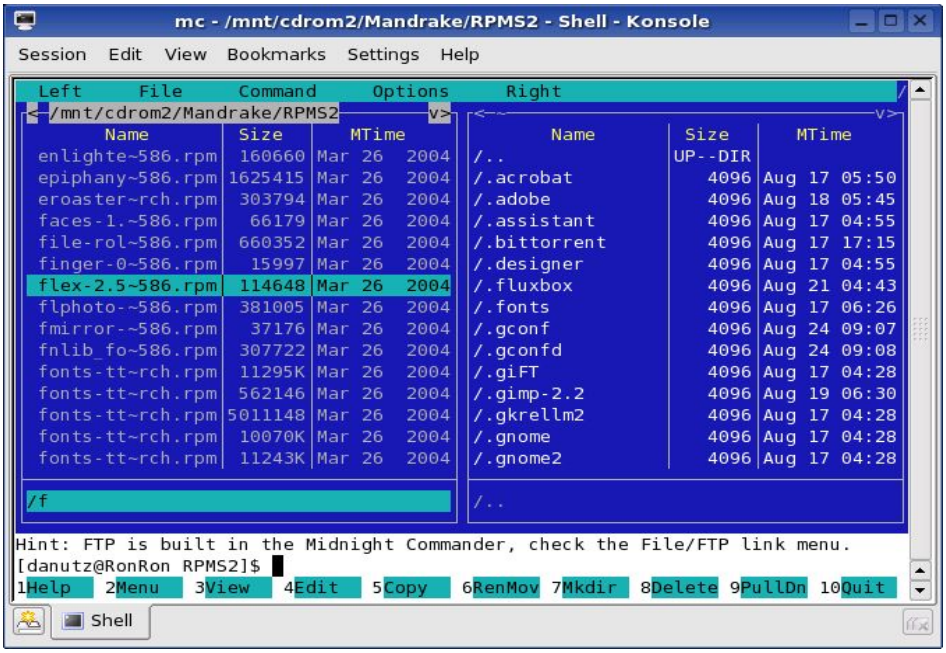
Programul mc este capabil să intre în pachetele rpm așa cum intră și în directoare. Folosiț i tastele cu săgeți pentru a alege pachetul dorit și tastați Enter în dreptul unui pachet pentru a-i vedea conținutul. Se intră cu Enter în directoarele din pachet așa cum se intră în directoarele obișnuite.



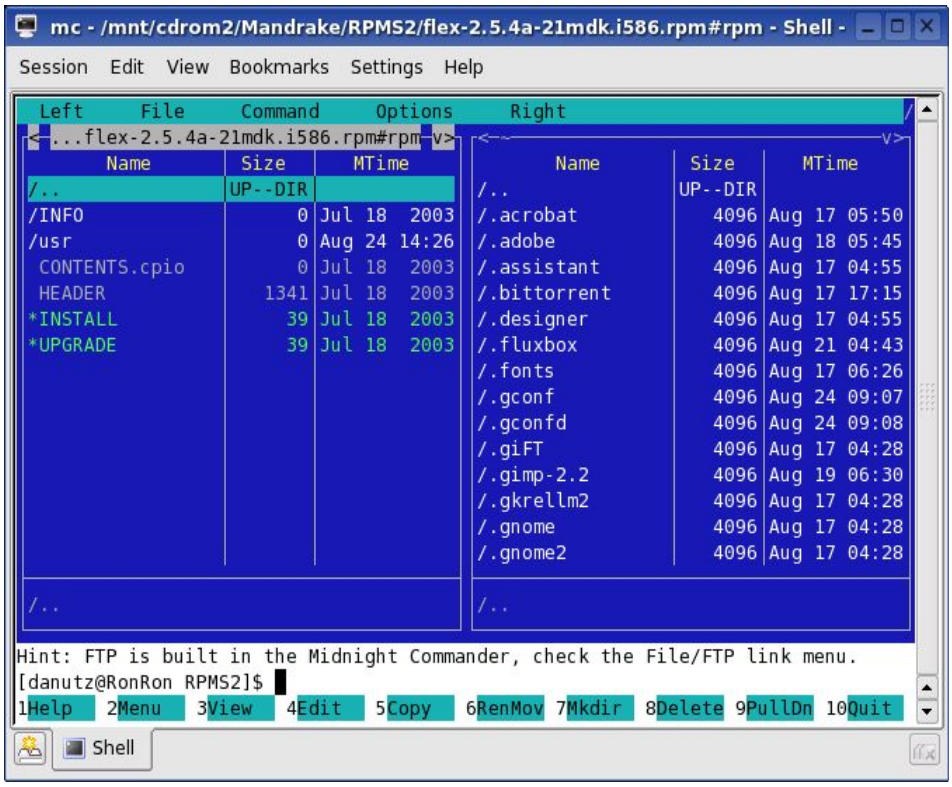
Exemplu: În cazul din figură, tastând Enter se intră în directorul /mnt, locul unde sunt de obicei montate mediile amovibile, printre care și CD-ROM-ul.
Intrați la fel în directorul /cdrom și de acolo în directorul cu rpm-uri.



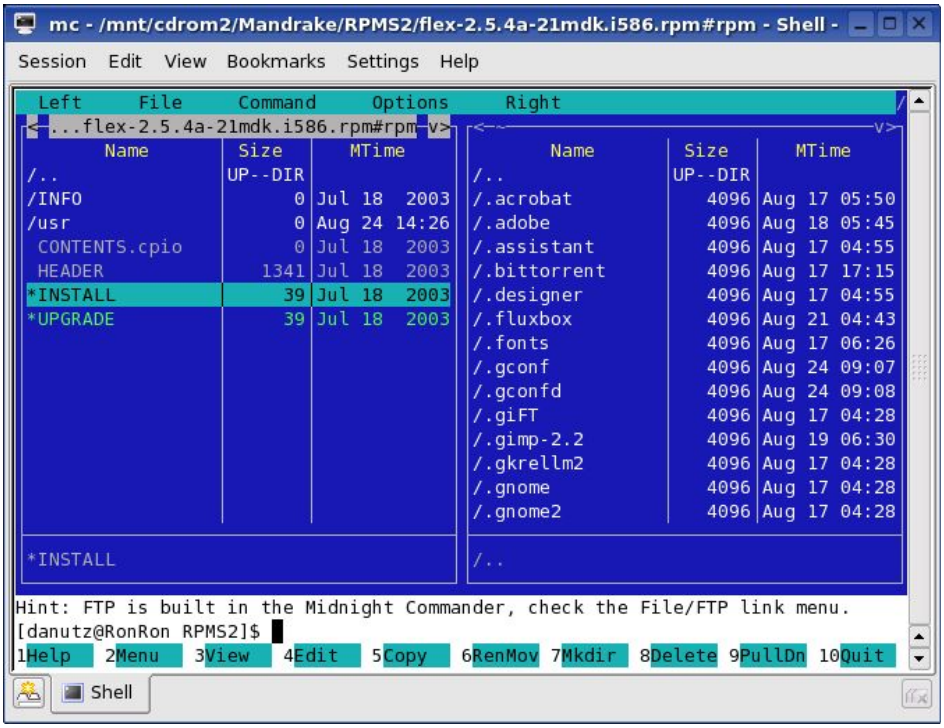
Aici puteți să parcurgeți lista de pachete cu tastele cu săgeți sau să căutați un pachet cu CTRL-s, scriind începutul numelui. Vă opriți pe rândul unde este fișierul dorit.



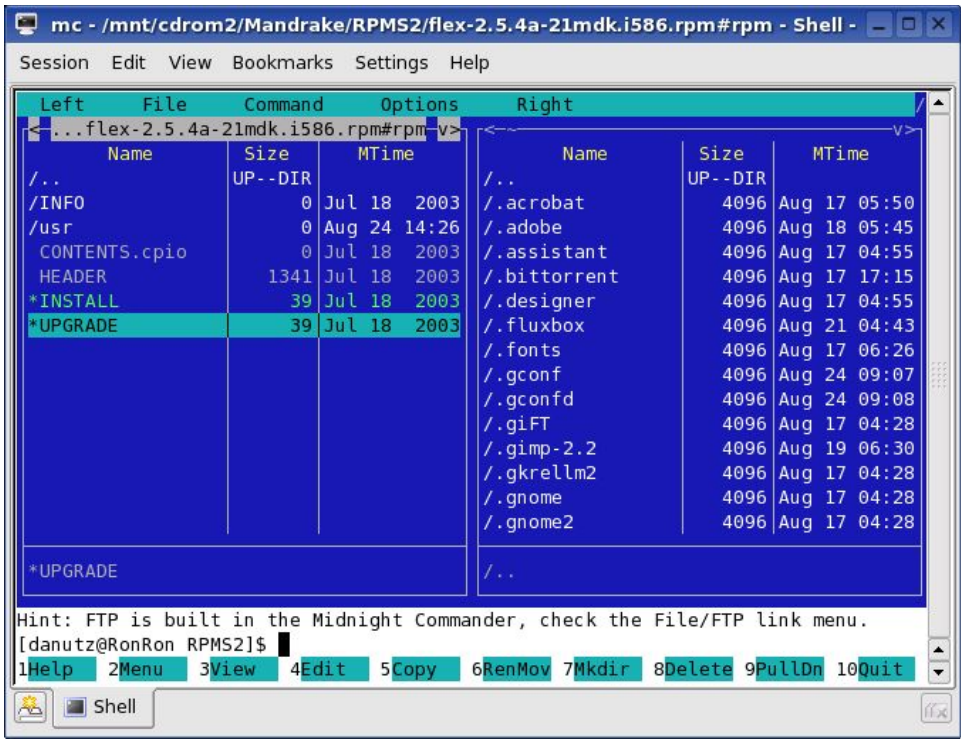
Tastând încă o dată Enter conținutul pachetului devine vizibil. Arată ca în imaginea următoare. Acest conținut vă este prezentat ca și cum ar fi un director .



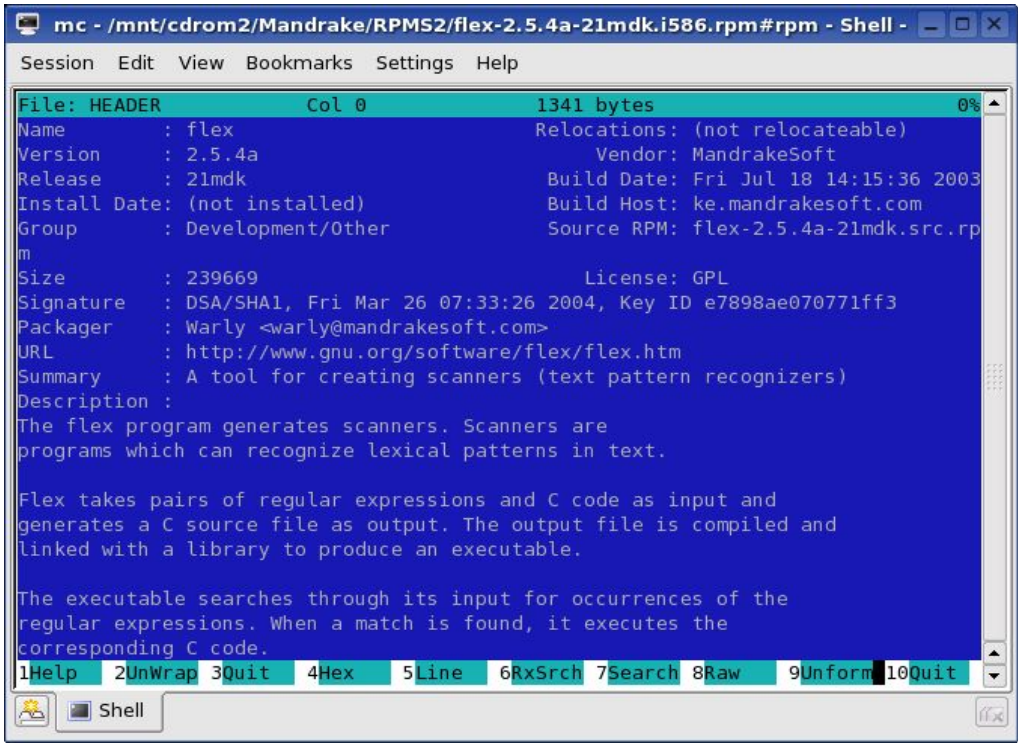
Alegeți scriptul INSTALL și tastați Enter pentru a începe instalarea.



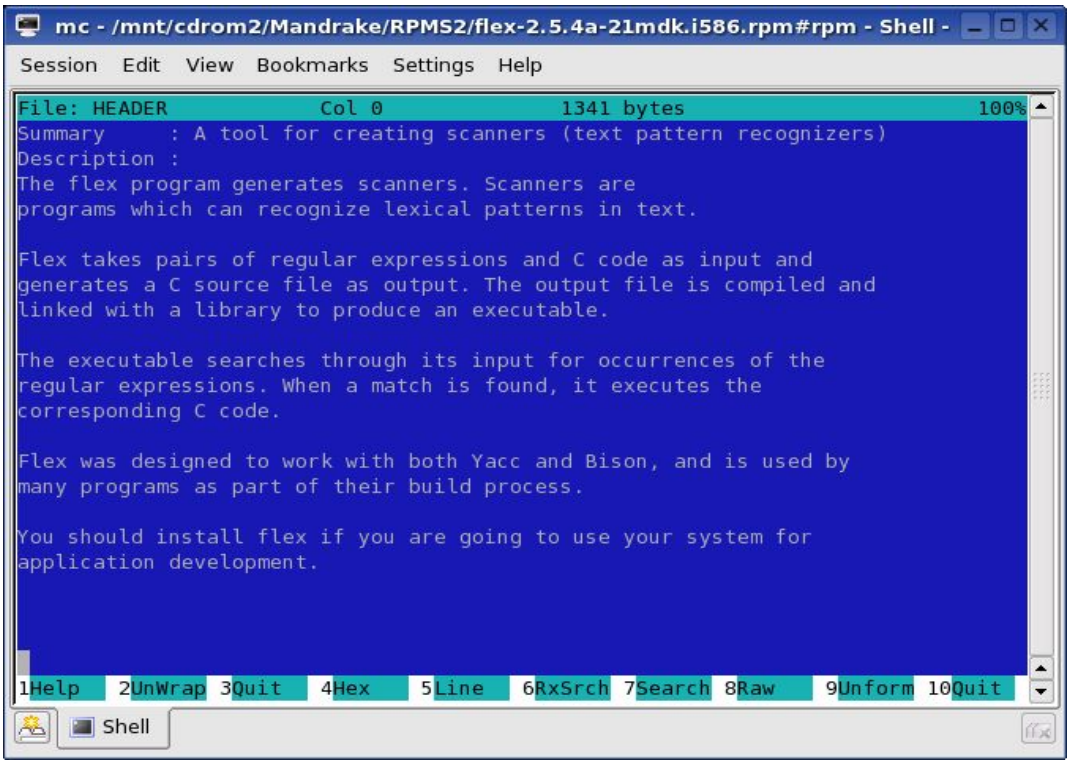
Pentru a moderniza un pachet la o nouă versiune s-ar fi ales scriptul UPGRADE.



HEADER conține informații importante despre pachet.



Folosiți tasta F3 pentru a le consulta înainte de a instala pachetul. Acolo pot fi mai multe ecrane cu text, le puteți face să defileze folosind clasicele taste cu săgeți.



În timpul instalării vom vedea un mesaj anunțând faza “Prepairing” apoi două indicatoare de progres în formă de bare orizontale și în finalul un procent 100% care indică terminarea fiecărei etape. Vor arăta așa:

```
Prepairing:     ########################################## [100%]
    1:flex      ########################################## [100%]
```

Notă: Alte distribuții folosesc alte comenzi pentru instalarea pachetelor (de exemplu apt-get, emerge șamd). Consultați documentația distribuției de Linux pe care o folosiți.

Succes.

Construcția compilatoarelor folosind Flex și Bison

Volumul se adresează atât programatorilor pe platforma UNIX / LINUX care doresc să se instruiască în construcția limbajelor de programare cât și liceenilor și studenților care vor să cunoască modul cum se scrie un compilator folosind limbajul C și generatoarele de programe Yacc și Lex sau clonele lor Flex și Bison disponibile pe sistemele de operare din familia Linux. Prezentarea pas cu pas a componentelor compilatorului unui limbaj asemănător cu C, numit Simple, împreună cu explicarea clară a conceptelor folosite de autori la construcția limbajelor de programare și subsistemelor unui compilator fac din această carte deopotrivă un curs condensat de compilatoare cât și un îndrumar practic pentru realizarea unui compilator cu aceste instrumente celebre: Flex și Bison.

Volumul poate fi folosit ca material de studiu auxiliar ori ca îndrumar de laborator în cadrul cursului de translatoare sau compilatoare atât în mediul politehnic cât și în facultățile de informatică.

Nu pot să vă descriu emoția scrierii primului program care va rula pe interpretorul dumneavoastră sau va fi compilat cu compilatorul dumneavoastră. Și mai ales satisfacția care o veți simți văzând că rulează sau se compilează și se execută fără erori. V-o spune un fost absolvent al primei serii de la o facultate de Informatică serie născută prin transformarea uneia din seriile de la o secție a Facultații de Matematică.

Dan Popa

Categoria: Construcția compilatoarelor

ISBN 973-0-04013-3