Limbaje Formale si Translatoare
Laborator 5

Martie 2021

1 Scopul Lucrarii

1. Prezentarea unor sugestii legate de pregatirea specificatiilor Yacc
2. Intelegerea conceptelor de precedenta si asociativitate
3. Intelegerea modalitatilor de tratare a erorilor
4. Implementarea unor exemple pentru aprofundarea notiunilor teoretice

2 Notiuni Teoretice

2.1 Pregatirea Specificatiilor Yacc

Aceasta sectiune contine cateva sugestii pentru pregatirea unor specificatii Yacc clare si usor de modificat.

2.1.1 Stil pentru intrare

(a) Folositi litere mari pentru tokeni si litere mici pentru non-terminale.

(b) Puneti regulile gramaticale si actiunile pe linii separate pentru a permite modificarea lor separata.

(c) Puneti toate regulile cu aceeasi parte stanga impreuna. Puneti partea stanga numai o data si scrieti toate regulile ce urmeaza incepand cu bara verticala.

(d) Puneti punct si virgula ; numai dupa ultima regula cu o parte stanga data si pe o linie separata, permitand astfel adaugarea usoara de noi reguli.

(e) Indentati corpul regulilor prin doua "tab"-uri.

(f) Aliniati doua puncte : cu barile verticale ce descriu corpul regulii pentru a alinia regulile cu aceeasi parte stanga una sub alta

2.1.2 Recursivitatea stanga

Analizorul folosit de YACC incurajeaza utilizarea in regulile gramaticale a recursivitatii stanga. Reguli de forma:

```
list    :   item
        |   list ',' item
        ;
seq     :   item
        |   seq     item
        ;                               ,
```

apar adesea cand se scriu specificatii de liste sau de secvente. In aceste cazuri, prima regula va fi redusa numai pentru primul item, iar a doua regula va fi redusa pentru al doilea si toate cele ce ii urmeaza.

Cu reguli recursive dreapta, cum este de exemplu:

```
seq :       item
    |       item seq
    ;                       ,
```

rezulta un analizor mai mare si articolele sunt vazute si reduse de la dreapta la stanga. Apare insa problema stivei interne a analizorului care e in pericol de a se "umple" cand din intrare e citita o secventa foarte lunga. De aceea, e recomandat ca utilizatorul sa foloseasca recursivitatea stanga daca e posibil.

E bine sa se analizeze daca o secventa cu zero elemente are vreo semnificatie, si in cazul in care are, e indicat sa se includa in specificatia de secventa o regula vida, ca in urmatorul exemplu:

```
seq :       /*empty*/
    |       seq     item
    ;
```

Prima regula va fi redusa intotdeauna o singura data, inainte ca primul item sa fie citit, iar a doua regula va fi redusa pentru fiecare item citit. Permitand secvente vide se creste generalitatea, dar pot sa apara conflicte daca YACC-ul trebuie sa decida ce secventa vida a "vazut", cand inca nu a citit destul ca sa stie acest lucru.

2.1.3 Cuvinte Rezervate

Unele limbaje de programare permit utilizatorului sa foloseasca pentru etichete sau nume de variabile cuvante (ca "if", "do", "else") care in mod normal sunt rezervate, in ideea ca o asemenea utilizare nu intra in conflict cu folosirea legala a acestor nume in limbajul de programare. Acest lucru e greu de realizat in cadrul oferit de YACC. In general, e bine ca cuvantele cheie sa fie rezervate, deci sa nu se permita utilizarea lor ca nume de variabile.

2.2 Precedenta

2.2.1 De ce?

In situatiile in care ne lovim de conflicte, cum ar fi shift/reduce in cazul operatiilor aritmetice, actiunea automata realizata de Yacc si anume shift nu este intotdeauna solutia preferata. Din acest motiv, Yacc permite suprascrierea precedentelor operatorilor pentru a specifica cand sa se faca shift si cand reduce.

Un exemplu intuitiv in care putem vizualiza de ce avem nevoie de conceptul de precedenta este dat de urmatorul fragment al unei gramatici ambigue.

```
expr :  expr '=' expr
     |  expr '+' expr
     |  expr '-' expr
     |  expr '*' expr
     |  expr '/' expr
     |  NAME
     ;
```

Consideram ca gramatica data este ambigua pentru ca de exemplu pentru intrarea "1 - 2 * 3", parsarea se poate face in doua feluri diferite. De exemplu, daca parserul vede tokenii '1', '-' si '2', iar urmatorul token este '*' atunci:

(a) ar putea sa realizeze o actiune reduce folosind operatorul de scadere din expr, dupa care un shift pentru operatorul '*'. Efectul in cazul acesta va fi din punct de vedere algebric "(1-2) * 3". Daca realizam mai intai reduce, stiva va incuraja o asociativitate stanga.

(b) ar putea sa realizeze mai intai o actiune shift pentru operatorul '*', apoi pentru '3', urmata de un reduce. Efectul in cazul acesta va fi "1 - (2 * 3)". Daca realizam mai intai shift, stiva va incuraja o asociativitate dreapta.

Astfel, pentru a decide intre cele doua variante, ar trebui ca Yacc sa foloseasca precedenta operatorilor, si astfel ca '*' sa aiba precedenta mai mare.

Situatia de mai sus solutia e clara, dar in situatia in care avem o intrare de tipul "1 - 2 - 5", ne intrebam daca semantic, el trebuie inteles ca "(1 - 2) - 5" sau ca "1 - (2 - 5)". In aceste cazuri se prefera prima varianta, adica asociativitate stanga, iar pentru operatori de tip assignment (=), se prefera asociativitate dreapta.

Vedem asadar ca multe dintre constructiile folosite pentru expresii aritmetice pot fi in mod natural descrise prin notiunea de nivele de precedenta ale operatorilor, impreuna cu informatie legata de asociativitatea stanga sau dreapta. Deci, se pot folosi gramatici ambigue, dar cu reguli de dezambiguizare potrivite, pentru a crea analizoare mai rapide si mai usor de scris decat cele construite din gramatici neambigue.

2.2.2 Ce?

Yacc-ul permite specificarea acestor alegeri legate de precedenta operatorilor prin declaratiile descrise mai jos. Acestea sunt plasate in sectiunea de declaratii si contin o lista de tokeni ce reprezinta operatorii a caror precedenta si asociativitate se declara.

- Declaratia **%left** face ca toti operatorii specificati sa aiba asociativitate de stanga
- Declaratia **%right** face ca toti operatorii specificati sa aiba asociativitate de dreapta
- O alternativa, **%nonassoc** face ca operatorii indicati sa nu fie asociativi, si indica o eroare de sintaxa la run-time la intalnirea aceluiasi operator de doua ori in aceeasi linie de intrare
- Declaratia **%precedence** permite doar definirea precedentei, fara asociativitate. Astfel, orice conflict legat de asociativitate va fi aruncat sub forma unei eroare de compilare

- Cuvantul cheie **%prec** schimba nivelul de precedenta asociat cu o regula gramaticala particulara. El apare imediat dupa corpul regulii, inainte de actiune sau de punct si virgula, si este urmat de un nume de token sau de literal ce va impune nivelul de precedenta. Precedenta regulii devine cea a numelui de token sau a literalului astfel specificat.

Prioritati

- **Toti tokenii de pe aceeasi linie vor fi tratati ca avand acelasi nivel de precedenta si asociativitate**, iar un token declarat cu %left, %right sau %nonassoc nu mai trebuie declarat si cu %token.
- **Precedenta relativa a diferitilor operatori e controlata de ordinea in care sunt declarati** alaturi de una din declaratiile de mai sus. Astfel, prima declaratie de predecenta sau asociativitate din fisier declara operatorii cu cea mai slaba precedenta, iar ultima, pe cei cu cea mai ridicata.

2.2.3 Cand?

Exemplul 1

In situatia data de exemplul cu operatorii aritmetici descris la sectiunea 2.2.1, o solutie pentru rezolvarea conflictelor ar fi declararea tokenilor pentru operatori astfel:

```
%right          '='
%left           '+' '-'
%left           '*' '/'
```

Liniile date descriu faptul ca plus si minus sunt asociative stanga si au precedenta mai mica decat inmultirea si impartirea, iar egal este asociativ dreapta, cu precedenta mai mica decat ceilalti operatori. In felul acesta, declaratiile de mai sus vor structura intrarea:

```
a = b = c * d - e - f * g
```

sub forma

```
a = (b = (((c * d) - e) - (f * g)))
```

Cand se foloseste acest mecanism, in general trebuie acordata o anumita precedenta si operatorilor unari, deoarece deseori un operator unar si unul binar pot avea aceeasi reprezentare simbolica, dar precedente diferite. Un exemplu e '-'-ul unar si binar. Minusului unar i se poate acorda aceeasi precedenta ca si inmultirii, sau chiar mai mare, in timp ce minusul binar are o precedenta mai mica decat inmultirea. Pentru a realiza acest lucru, se poate folosi cuvantul cheie %prec astfel:

```
%left           '+' '-'
%left           '*' '/'

%%

expr        :   expr '+' expr
            |   expr '-' expr
            |   expr '*' expr
            |   expr '/' expr
            |   '-' expr %prec '*'
            |   NAME
            ;
```

Exemplul 2

Exista si situatii cand utilizarea %left, %right sau %nonassoc ar putea ascunde anumite conflicte, caz in care se prefera utilizarea %precedence.

Un exemplu comun ar fi cel legat de ambiguitatea if-else, ce poate fi rezolvat explicit. Conflictul de shift/reduce in acest caz apare in urmatorul stadiu de parsare:

```
if e1 then if e2 then s1 . else s2
```

Conflictul e dat de reducerea regulii 'IF expr THEN stmt', a carei precedenta implicita e cea a ultimului token, adica 'THEN', si de actiunea shift pe tokenul 'ELSE'. In aceasta situatie se prefera sa se ataseze else-ul la cel mai apropiat if. Astfel, pe exemplul dat, ar trebui sa se realizeze o actiune shift asa incat precedenta lui 'ELSE' sa fie mai mare decat precedenta lui 'THEN'. Fiindca nici unul din cei doi tokeni nu ar trebui sa fie implicat intr-un conflict legat de asociativitate, se poate specifica:

```
%precedence THEN
%precedence ELSE
```

In acest fel, indicam doar ca operatorul 'ELSE' are precedenta mai mare, fara a ne lega si de asociativitate. Exista totusi si varianta de a oferi ambilor tokeni precedenta egala, si a folosi asociativitatea pentru a rezolva conflictul:

```
%right "then" "else"
```

2.2.4 Cum?

Precedentele si asociativitatile sunt utilizate de YACC pentru a rezolva conflictele de analiza, si ele produc reguli de dezambiguizare. Formal, regulile lucreaza astfel:

1. In momentul introducerii in fisier a declaratiilor de precedenta, se asigneaza nivele de precedenta simbolurilor terminale/tokenilor declarati.

2. Pe baza aceasta, regulile ce contin acesti tokeni primesc precedenta si asociativitatea ultimului terminal din corpul lor. Daca este folosita constructia %prec, ea se suprapune peste aceasta situatie implicita.

   Nu e necesar ca toate regulile si toti tokenii sa aiba precedenta sau asociativitate.

3. Rezolvarea conflictelor prin introducerea precedentei functioneaza prin compararea precedentei regulii curente din parsare cu precedenta tokenului "lookahead". Daca precedenta tokenului e mai mare, alegerea este sa se faca shift. Altfel, se alege reduce. Daca precedentele sunt egale, alegerea se face in functie de asociativitatea nivelului de precedenta. Asociativitatea stanga implica reducere, asociativitatea dreapta implica deplasare, iar non-asociativitatea implica eroare.

   Conflictele rezolvate prin precedenta nu sunt raportate. Aceasta inseamna ca erorile in specificarea precedentelor trebuie evitate.

Pentru a vizualiza cum s-a rezolvat un conflict, se poate rula Yacc-ul cu -v, in fisierul y.output fiind listate si solutiile pentru conflicte.

3 Desfasurarea Laboratorului

3.1 Precedenta

Transformati o expresie aritmetica din postfix in infix, fara executie. Presupunerea este ca avem doar o singura expresie pe linie.

```
2 3 + ar trebui sa returneze 2+3 sau (2+3)
4 2 3 *+ ar trebui sa returneze 4+2*3
22 3 + 5 6 + * ar trebui sa returneze (22+3) * (5+6)
```

Hints:

(a) folositi %union{char * s; } pentru a transmite din lex un string pentru numere

(b) folositi strdup, strcat, strcpy din string.h pentru a crea un string nou, si concatenati cele doua stringuri necesare

3.2 If-then-else Scanner

Creati un parser pentru o afirmatie de tip If-then-else. Programul ar trebui sa recunoasca daca o propozitie este corecta sau nu (fara executie si momentan fara returnarea arborelui de parsare).

Structuri permise:

```
if (a>2) then a=2
if (3>a) then if (b>=2) then b=2
if (3>a) then if (b>=2) then b=2 else c=10
2
c=10
```

Structuri interzise:

```
if (a=3) then a=2;
if (a>3) then a<2;
```

- In conditia unui if ar trebui sa aveti un operator relational sau de egalitate (<, >, <=, >=, ==, !=).
- Ca si corp pentru un if puteti avea doar asertiuni sau alte afirmatii if.
- Orice afirmatie if se incheie cu ';'.

Hints: posibile reguli pentru Yacc

```
if_stmt  : IF '(' cond ')' THEN stmt ';'
         ;
```

%PREC schimba nivelul de precedenta asociat cu o regula gramaticala specifica
%NONASSOC permite definirea precedentei pentru operatori care nu sunt asociativi

3.3 Arbore de Parsare

Creati un parser pentru problema anterioara care creaza si afiseaza arborele de parsare. Adaugati functii din functions.c drept actiuni in gramatica, dupa modelul descris la problema anterioara.

Mai intai, compilati si analizati functiile definite in functions.c.

```
gcc if_functions.c
```