Limbaje formale si tehnici de compilare

Laborator 4

Eliminarea recursivitatii stangi si factorizarea prefixelor comune

Pentru implementarea unui analizor sintactic descendent recursiv (ASDR) este necesar ca regulile sintactice sa indeplineasca unele conditii:

- nu trebuie sa existe recursivitate stanga (A ::= A α | β), deoarece cand A se va implementa printr-o functie, aceasta functie s-ar apela pe ea insasi intr-o recursivitate infinita. Eliminarea recursivitatii stangi se face aplicand urmatoarea formula:

  $\mathbf{A ::= A\ \alpha_1 \mid \ldots \mid A\ \alpha_m \mid \beta_1 \mid \ldots \mid \beta_n} \longrightarrow \begin{array}{l}\mathbf{A ::= \beta_1\ A' \mid \ldots \mid \beta_n\ A'} \\ \mathbf{A' ::= \alpha_1\ A' \mid \ldots \mid \alpha_m\ A' \mid \varepsilon}\end{array}$

  Exemplu: „exprOr ::= exprOr OR exprAnd | exprAnd" → „exprOr ::= exprAnd exprOr1" si „exprOr1 ::= OR exprAnd exprOr1 | ε"

- nu trebuie sa existe prefixe comune intre mai multe alternative (A ::= α β_1 | α β_2), deoarece daca ar aparea secventa α la intrare nu s-ar sti pe care alternativa sa se mearga. Factorizarea prefixelor comune se face aplicand urmatoarea formula:

  $\mathbf{A ::= \alpha\ \beta_1 \mid \ldots \mid \alpha\ \beta_n} \longrightarrow \mathbf{A ::= \alpha\ (\ \beta_1 \mid \ldots \mid \beta_n\ )}$

Literele grecesti semnifica orice expresie. A' este o regula noua care poate avea orice nume nefolosit inca. ε (epsilon) este alternativa vida si este indeplinita de orice secventa de intrare (inclusiv vida). Prezenta ε ca membru intr-o alternativa face ca acea alternativa sa devina optionala: „α | ε" = „α?".

Analizor sintactic descendent recursiv (ASDR)

ASDR este relativ simplu de implementat manual si ofera o flexibilitate marita. Implementarea lui se bazeaza pe algoritmul descris in continuare.

Consideram atomii lexicali dispusi intr-o lista. Variabila globala „**Token *crtTk;**" pointeaza la atomul curent (initial primul atom din lista). Ultimul atom din lista are codul **END**.

Fiecare regula sintactica (nonterminal) va fi implementata ca o functie separata. Aceasta functie va returna **true** (1) daca incepand cu atomul curent (cel pointat de **crtTk**) se afla o succesiune de atomi corespunzatoare regulii respective, sau **false** (0) in caz contrar. Functiile care returneaza o valoare booleana se mai numesc si predicate. Daca s-a gasit o succesiune corespunzatoare de atomi, functia va consuma (va trece de) toti atomii care formeaza succesiunea respectiva si in final **crtTk** se va afla pe primul atom de dupa acestia. Daca functia nu gaseste nicio succesiune de atomi corespunzatoare regulii pe care o implementeaza, **crtTk** va ramane nemodificat.

Rezulta astfel ca fiecare predicat care implementeaza un nonterminal este responsabil cu consumarea tuturor atomilor care intra in compunerea acelui nonterminal. Daca un nonterminal contine alte nonterminale, el le va apela pe acestea si daca ele sunt indeplinite, fiecare va consuma proprii sai atomi constituenti si va returna **true**. Daca un predicat nu este indeplinit, pozitia curenta in lista de atomi va ramane neschimbata si va returna **false**.

Toata analiza sintactica va fi realizata apeland nonterminalul de start. Acesta la randul lui va apela celelalte nonterminale.

Pentru consumarea terminalelor (atomii lexicali) se foloseste functia „**consume(int cod)**" care daca la pozitia curenta se afla un atom cu codul specificat il consuma si returneaza **true**, altfel pozitia ramane neschimbata si se returneaza **false**. O posibila implementare a functiei **consume** este urmatoarea:

```
Token    *consumedTk;

int      consume(int code)
{
    if(crtTk->code==code){
        consumedTk=crtTk;
        crtTk=crtTk->next;
        return 1;
        }
    return 0;
}
```

consume mentine un pointer la ultimul atom consumat in variabila **consumedTk**. Aceasta va fi de folos cand de exemplu va fi nevoie sa se testeze o constanta iar apoi sa se utilizeze valoarea sa, atomul respectiv fiind deja consumat.

Combinand aspectele descrise mai sus, rezulta ca:

- **atomii lexicali (terminalele)** vor fi intotdeauna consumati folosind functia **consume**
- **nonterminalele** se consuma apeland predicatele care implementeaza acele nonterminale

Este posibil ca in testarea unei reguli sa se poata avansa in interiorul ei, fara a se ajunge totusi la final. In acest caz trebuie sa existe o modalitate de a se reveni la pozitia initiala in lista de atomi (cea cu care a inceput testarea), pentru a se mentine conditia ca daca un predicat nu este indeplinit, el nu va schimba pozitia atomului curent. Aceasta se poate realiza simplu setand chiar la inceputul predicatului o variabila cu valoarea initiala a **crtTk**: „**Token *startTk=crtTk**;**”**. Daca se doreste revenirea, aceasta se face prin „**crtTk=startTk;**”.

In continuare se vor da unele sugestii de implementare a gramaticilor in format EBNF.

SECVENTA (a b …)

Se implementeaza testand pe rand fiecare componenta a sa si returnand true doar daca toate sunt indeplinite.

ruleWhile ::= WHILE LPAR expr RPAR stm

```
int      ruleWhile()
{
Token    *startTk=crtTk;
if(consume(WHILE)){
    if(consume(LPAR)){
        if(expr()){
            if(consume(RPAR)){
                if(stm()){
                    return 1;
                    }else tkerr(crtTk,"missing while statement");
                }else tkerr(crtTk,"missing )");
            }else tkerr(crtTk,"invalid expression after (");
        }else tkerr(crtTk,"missing ( after while");
    }
crtTk=startTk;
return 0;
}
```

Se poate constata faptul ca pentru a se returna **true** trebuie sa se indeplineasca toate componentele secventei. **WHILE**, **LPAR** si **RPAR** sunt terminale si atunci s-au consumat folosind functia **consume**. **expr** si **stm** sunt noterminale si atunci s-au apelat predicatele lor pentru a fi consumate.

In cazul secventelor se pot genera mesaje de eroare foarte clare, deoarece se stie ce trebuie sa urmeze in secventa. De exemplu dupa „while” trebuie neaparat sa urmeze o paranteza deschisa, altfel este eroare. Singura ambiguitate

ar fi faptul ca daca expresia pe care o testeaza „while” este gresita, de exemplu „while(-/a)”, expr va returna **false**, ceea ce in acest caz nu inseamna totusi ca expresia lipseste, ci ca e gresita. Ar fi fost posibil si ca ea sa lipseasca: „while()”, dar acest caz e mult mai rar. Din acest motiv s-a ales un mesaj de eroare pentru cazul cel mai probabil. Mesajul complet ar fi fost „missing or invalid expression after (”. In cazul implementarilor performante s-ar fi putut elimina chiar si aceasta ambiguitate, prin testarea daca paranteza inchisa este consecutiva celei deschise.

In aceasta implementare particulara, daca se tine cont de faptul ca **tkerr** termina complet programul (folosind functia **exit** sau generand o exceptie), deci nu se mai revine in codul **ruleWhile**, acesta poate fi scris simplificat astfel:

```
int     ruleWhile()
{
    if(!consume(WHILE))return 0;
    if(!consume(LPAR))tkerr(crtTk,"missing ( after while");
    if(!expr())tkerr(crtTk,"invalid expression after (");
    if(!consume(RPAR))tkerr(crtTk,"missing )");
    if(!stm())tkerr(crtTk,"missing while statement");
    return 1;
}
```

Daca nu s-a consumat atomul **WHILE** nu este nevoie sa se revina la **startTk**, deoarece **crtTk** a ramas neschimbat (acest aspect este valabil si la versiunea anterioara, dar s-a folosit **startTk** pentru exemplificare).

Se poate constata ca si in acest caz, la fel ca in versiunea anterioara, nu se genereaza eroare daca lipseste **WHILE** ci doar se returneaza 0. Aceasta este din cauza ca o instructiune poate fi si altceva, nu doar „while” (de exemplu poate fi: if, return, for, …). Daca absenta „while” ar fi generat o eroare, atunci practic nu s-ar mai fi dat voie sa existe si alte instructiuni a caror testare s-ar fi facut dupa „while”. In schimb daca s-a consumat **WHILE**, deci a inceput secventa specifica acestei instructiuni, urmatoarele componente sunt obligatorii asa cum sunt ele descrise in gramatica, deci absenta lor va genera erori.

Din cele prezentate mai sus rezulta ca este important de analizat cand se genereaza erori si cand doar se semnaleaza faptul ca predicatul nu a fost indeplinit. Regula este urmatoarea: **Daca intr-o anumita situatie exista o singura posibilitate, neindeplinirea acesteia va genera o eroare. Daca exista mai multe posibilitati, neindeplinirea uneia dintre ele va semnala doar predicat neindeplinit, dar se va genera eroare daca niciuna dintre aceste posibilitati nu a fost indeplinita.**

ALTERNATIVA (a | b)

Se implementeaza testand pe rand fiecare componenta si returnand **true** la prima adevarata. Daca niciuna nu este adevarata, in final se returneaza **false**.

factor ::= ID | CT_INT | LPAR expr RPAR

```
int     factor()
{
    Token *startTk=crtTk;
    if(consume(ID)){
        return 1;
        }
    if(consume(CT_INT)){
        return 1;
        }
    if(consume(LPAR)){
        if(expr()){
                if(consume(RPAR)){
                    return 1;
                    }else tkerr(crtTk,"missing )");
                }else tkerr(crtTk,"invalid expression after (");
        crtTk=startTk; // restaureaza crtTk la valoarea de intrare
```

```
        }
    return 0;
}
```

Aceasta regula contine o alternativa a carei ultima parte este o secventa. Alternativa a fost implementata folosind o succesiune de „if”-uri, fiecare „if” testand o componenta a alternativei. Daca oricare componenta este satisfacuta, se returneaza succes. Daca niciuna dintre compontente nu este satisfacuta, in final se returneaza 0.

Daca secventa finala nu este indeplinita in totalitate, a fost necesar sa se restaureze pozitia lui **crtTk** de la intrarea in predicat, deoarece este posibil sa se consume doar LPAR sau LPAR si expr, dar nu si RPAR. In aceste cazuri pozitia initiala trebuie refacuta inainte de a se returna 0.

Optionalitatea (a?)

Deoarece in acest caz **a** poate sa existe sau nu, se incearca sa se consume acesta, dar fara sa se tina cont de rezultatul incercarii.

typeName ::= typeBase arrayDecl?

```
int     typeName()
{
    if(!typeBase())return 0;
    arrayDecl();
    return 1;
}
```

Se poate constata ca daca s-a trecut de **typeBase** nu mai conteaza faptul ca **arrayDecl** (care este optional) exista sau nu exista. Daca el exista va fi consumat, dar in oricare situatie rezultatul final nu va fi influentat. Existenta **arrayDecl** se poate testa cu un „if” daca de aceasta depind anumite procesari in fazele ulterioare.

Alternativa vida epsilon (ε) se poate implementa in acelasi mod, tinand cont de faptul ca **„a | ε” = „a?”**.

Repetitia optionala (a*)

Se include testarea lui **a** intr-o bucla care dureaza atata timp cat s-a putut consuma **a**. In cazuri mai simple **a** se poate scrie direct ca si conditie a lui „while” (**while(a){…}**), altfel se poate folosi o bucla infinita din care se iese cand nu se mai poate consuma **a** (**while(1){ if(!a())break; }**).

stmCompound: LACC (declVar | stm)* RACC ;

```
int     stmCompound()
{
    if(!consume(LACC))return 0;
    while(1){
        if(declVar()){
            }
        else if(stm()){
            }
        else break;
        }
    if(!consume(RACC))tkerr(crtTk,"missing } or syntax error");
    return 1;
}
```

In acest caz o intreaga alternativa (**declVar | stm**) a trebuit sa poata fi repetabila optional. Toata alternativa a fost pusa intr-o bucla infinita, din care se iese doar atunci cand nicio componenta a ei nu mai poate fi indeplinita. Se poate constata faptul ca nu conteaza de cate ori (sau niciodata) a fost indeplinita alternativa, deoarece nicaieri nu se contorizeaza aceasta. O varianta pentru repetarea alternativei poate fi „**while(declVar() || stm()){…}**”.

Repetitia cu cel putin o existenta (a+) se poate implementa analogic, tinand cont de faptul ca **„a+” = „a a*”**.

Observatii

Dupa ce algoritmul care implementeaza un ASDR a fost prezentat, se poate intelege de ce nu este permisa recursivitatea stanga. De exemplu daca s-ar fi implementat direct o regula de forma „**exprOr ::= exprOr OR exprAnd | exprAnd**", aceasta ar fi avut un inceput de forma „**int exprOr(){ if(exprOr()){ …**". In aceasta situatie, primul apel din interiorul **exprOr** este tot **exprOr** si aceasta fara ca intre cele doua apeluri sa se consume vreun atom. De aici rezulta o recursivitate infinita, fara nicio modificare a pozitiei curente in atomii lexicali.

Deoarece predicatele se apeleaza unele pe altele, este necesar ca ele sa fie scrie in ordinea corecta limbajului C, unde un simbol mai intai trebuie declarat si apoi folosit. De exemplu predicatul **declVar** care este folosit in **unit** va trebui scris inainte de acesta. Cand doua predicate au referinte incrucisate unul la celalalt (ex: **stmCompound** cu **stm**), atunci antetul celui de-al doilea (**int stm();**) se poate declara inaintea primului, pentru a fi vizibil din acesta.

In aceasta faza rolul analizorului sintactic este doar analizarea programului de intrare pentru a se vedea daca este corect sau nu din punct de vedere sintactic. Daca programul este corect, nu se va genera nicio eroare.

Atentie: aproape toate fazele ulterioare ale compilatorului vor adauga cod in interiorul analizorului sintactic, deci acesta trebuie inteles foarte bine si implementat cat mai „elegant".

Aplicatia 4.1: *Sa se scrie predicatele corespunzatoare regulilor* ***exprAdd*** *si* ***exprPrimary****.*

Tema: sa se scrie analizorul sintactic complet pentru limbajul AtomC.