LUCRAREA 6

Analiza sintactică folosind CUP

Cuprinsul lucrării:

- Generatorul de parseri sintactici CUP
- Sintaxa unui fişier
- Definirea gramaticii
- Conflicte de tip shift/reduce

Unele exemple şi figuri din acest capitol au fost preluate şi adaptate din [5].

6.1 Generatorul de parseri sintactici CUP

CUP (Constructor of Useful Parsers) este un program care generează parseri sintactici transformând o gramatică independentă de context într-un program Java care parsează secvenţa de simboluri de intrare în funcţie de regulile gramaticii. Gramatica este specificată într-un fişier cu extensia .cup, iar pentru fiecare regulă a gramaticii este menţionată şi o acţiune care se va executa când regula respectivă ajunge să fie aplicată.

CUP crează un parser de tipul LALR (Look-Ahead Left-to-Right), care reprezintă o formă particulară de parser LR(1). Parserul sintactic astfel generat trebuie integrat cu un analizor lexical. În acest sens, în prezenta lucrare se va utiliza un lexer generat folosind JLex, şi se va arăta cum CUP

şi JLex pot fi utilizate împreună pentru a scrie un program complet care implementează analiza lexicală şi sintactică.

Pentru acest laborator se va folosi un mediu de lucru gata configurat care conţine atât codul sursă pentru CUP şi JLex cât şi un exemplu minimal de compilator descris prin cele două fişiere de specificaţii `example.lex` şi `example.cup`[4]. Fişierele pot fi descărcate de la adresa:

`https://github.com/gmh33/Java-CUP-jLex-Example`

6.2 Integrarea JLex şi CUP. Clasa Symbol

Pentru a integra JLex şi CUP este nevoie de a specifica în cadrul fişierului JLex faptul că ieşirea lexer-ului va fi interfaţată cu parserul generat de CUP. Acest lucru înseamnă în primul rând că atomii lexicali generaţi de lexer nu vor mai fi de tipul specificat de utilizator, ci vor fi obligatoriu de tipul `Symbol`, o clasă predefinită din CUP. Pentru aceasta, în prima secţiune a fişierului JLex (cea dedicată codului de utilizator) se adaugă linia: `import java_cup.runtime.Symbol;`

În continuare, în a doua secţiune a fişierului se introduce directiva `%cup` care va specifica JLex faptul că atomii lexicali vor fi trimişi mai departe către un program generat de CUP. Orice altă directivă care specifică tipul de atomi, de exemplu `%integer` trebuie îndepărtată, întrucât tipul de atomi care va fi returnat este `Symbol`.

Un obiect de tip Symbol are mai multe date membre, ce pot fi văzute la sfârşitul fişierului `CUP/java_cup/runtime/Symbol.java`. Dintre acestea, de interes sunt `sym` şi `value`. Data membră `sym` va primi o valoare din clasa `sym`, ce va fi creată de CUP, iar `value` va primi o valoare dată de utilizator în fişierul JLex şi este de tipul generic `Object`, ceea ce înseamnă că poate primi orice tip de dată.

De interes sunt şi constructorii clasei `Symbol`. Aceştia sunt apelaţi în fişierul JLex atunci când se specifică acţiunea ce urmează a fi executată la găsirea unui atom de un anumit tip. Dacă atomul nu are o valoare asociată (spre exemplu, un şir de caractere sau o valoare numerică), se va apela constructorul cu prototipul `public Symbol(int sym_num)`, spre exemplu la întâlnirea unui operator: `return new Symbol(sym.PLUS)`. În cazul în care atomului trebuie să i se asocieze o valoare, se apelează constructorul `public Symbol(int id, Object o)`, ca de exemplu la întâlnirea unui identificator, situaţie în care trebuie returnat pe lângă tipul de atom şi denumirea identificatorului sub forma unui `String`: `return new Symbol(sym.ID, new String(yytext()))`.

6.3 Fişierul CUP

Fişierul de specificaţii CUP are de regulă extensia `.cup` şi conţine descrierea gramaticii independente de context pe baza căreia CUP va crea clasele `parser` şi `sym` (care stau la baza parser-ului propriu-zis). Acest fişier începe de regulă cu liniile de cod care vor fi copiate direct în parser. În exemplul dat, primele linii se referă la includerea unor anumite clase Java:

```
import java_cup.runtime.*;
import java.util.*;
```

În Codul sursă 6.1, se utilizează directiva `parser code` pentru a crea funcţia `main` direct în clasa `parser`.

Codul sursă 6.1: Crearea funcţiei `main` direct în clasa `parser`.

```
parser code {:
        /* this part is included in the parser class */
        public static void main (String args[]) throws
           Exception{
                new parser(new Yylex(System.in)).parse();
        }
:}
```

Constructorul apelat primeşte ca parametru un nou obiect de tip `Yylex`, prin aceasta specificându-se faptul că parserul integrează un lexer definit prin clasa `Yylex` ce trebuie în prealabil generată folosind JLex.

Urmează enumerarea simbolurilor gramaticii, mai întâi cele terminale (vezi Codul sursă 6.2). În general, acestea sunt aceleaşi cu atomii lexicali returnaţi de lexer, iar CUP copiază denumirile specificate în clasa `sym` şi atribuie fiecăreia o valoare numerică. Orice simbol terminal menţionat aici dar care nu e prezent în nicio regulă a gramaticii va determina afişarea unui mesaj de avertizare în etapa generării parserului.

Codul sursă 6.2: Enumerarea simbolurilor terminale.

```
terminal VAR,BEGIN,END,IF,THEN,WHILE,DO,ODD;
terminal EQ,NEQ,LT,LTE,GT,GTE,LPAREN,RPAREN,PERIOD;
terminal COMMA,SEMI,ASSIGN,PLUS,MINUS,MULT,DIV,PRINT;
terminal Integer INT;
terminal String ID;
```

Specificaţiile CUP continuă cu enumerarea simbolurilor neterminale (vezi Codul sursă 6.3), fiecare dintre ele trebuind să se regăsească ulterior în membrul stâng al unei producţii. La fel, orice simbol neterminal menţionat aici dar nefolosit sau inaccesibil va genera un mesaj de avertizare la momentul generării parserului. Un lucru important de reţinut este că simbolurile

terminale sunt enumerate în ordinea crescătoare a precedenţei (priorităţii) acestora.

Codul sursă 6.3: Enumerarea simbolurilor neterminale.

```
non terminal Program program;
non terminal Block block;
non terminal VarLst var_lst;
non terminal StmtLst stmt_lst;
non terminal Statement statement;
non terminal Condition condition;
non terminal Expr expr, term, factor;
```

După enumerarea simbolurilor gramaticii urmează explicitarea producţiilor acesteia (vezi Codul sursă 6.4). Fiecare simbol neterminal se regăseşte în membrul stâng al cel puţin unei producţii, iar fiecare producţie este însoţită şi de o acţiune semantică. Aceste acţiuni sunt folosite pentru a produce anumite efecte la aplicarea unei anumite reguli, mai exact pentru a genera codul de asamblare corespunzător. Regulile gramaticii sunt aplicate după potrivirea membrului drept, datorită faptului că parserul este unul de tip LR.

Codul sursă 6.4: Explicitarea producţiilor gramaticii.

```
expr ::=
                  term:t
                        {: RESULT = t; :}
                | PLUS term:t
                        {: RESULT = t; :}
                | MINUS term:t
                        {: RESULT = new InvExpr(t); :}
                | expr:e PLUS term:t
                        {: RESULT = new PlusExpr(e,t); :}
                | expr:e MINUS term:t
                        {: RESULT = new MinusExpr(e,t); :}
                ;
```

În cadrul fiecărei producţii a gramaticii, fiecare terminal are asociată o variabilă care preia valoarea terminalului respectiv. Această variabilă este definită prin simbolul : urmat de numele variabilei. Spre exemplu, `expr ::= term:t`, unde `t` este variabila care area valoarea simbolului terminal `term`. Folosind aceste variabile se pot specifica anumite acţiuni care implică simbolurile terminale. Pentru acelaşi exemplu, acţiunea este `{: RESULT = t; :}`, ceea ce înseamă că valoarea atribuită lui `RESULT` este transmisă automat către neterminalul din partea stângă a producţiei, în cazul de faţă `expr`.

În cazul unei alte producţii a simbolului neterminal `expr`, şi anume `expr :== MINUS term:t`, se observă faptul că acţiunea semantică utilizează variabila `t` dar în acelaşi timp crează un nou obiect care este trimis prin `RESULT` către `expr`, şi anume: `{: RESULT = new InvExpr(t); :}`. Crearea de astfel de obiecte semantice, sau în alte situaţii funcţii de acţiune semantică se impune în situaţiile în care acţiunile care trebuie întreprinse pentru o anumită producţie au un nivel de complexitate ridicat. În acest fel, fişierul de specificaţii CUP îşi păstrează o structură lizibilă.

Definirea şi explicitarea acestor obiecte şi funcţii de acţiune semantică este făcută într-un fişier java separat, în cazul exemplului din acest laborator, este vorba de fişierul `Program.java`.

6.4 Generarea parserului

Mediul de lucru pentru prezentul laborator are automatizat procesul de compilare şi generare a programelor aferente folosind un fişier de tip makefile (denumit `makefile`). Astfel, în linia de comandă se introduc următoarele comenzi:

```
make init
make all
```

care vor compila sursele JLex şi Cup (`init`) precum şi sursele compilatorului (`all`).

Fără utilizarea unui fişier de tip makefile, paşii necesari sunt descrişi mai jos (ei se regăsesc în makefile, lucru verificabil prin deschiderea acestui fişier cu un editor de text):

```
javac JLex/*.java
javac java_cup/runtime/*.java
javac java_cup/*.java
javac -classpath .  java_cup.Main example.cup
javac JLex.Main example.lex
mv example.lex.java Yylex.java
javac Program.java
javac sym.java parser.java Yylex.java
```

După compilarea surselor, sunt generate două clase Java care descriu implementarea parserului: `sym` şi `parser`. Clasa `sym`, descrisă în fişierul `sym.java` cuprinde definirea de valori numerice întregi pentru fiecare din simbolurile terminale eumerate în fişierul de specificaţii CUP. Clasa `parser`, cuprinsă în fişierul `parser.java`, cuprinde descrierea tabelară a producţiilor, acţiunilor şi regulilor de reducere corespunzătoare.

De asemenea, fişierul include şi codul scris de utilizator la începutul fişierului de specificaţii CUP, în cazul nostru funcţia `main`, care crează un obiect de tipul clasei (`parser`). Tot în fişierul `parser.java` se regăseşte un constructor al clasei care primeşte ca parametru un obiect de tip `Scanner`. Acest Scanner e de fapt lexer-ul generat de JLex, fapt observabil în definiţia clasei `Yylex` din fişierul `Yylex.java`. Prin aceste specificaţii, parserul ştie că va folosi clasa `Yylex` ca lexer, cu alte cuvinte ca sursă de atomi lexicali.

După generare, parserul poate fi lansat în execuţie furnizându-i-se un cod de intrare. În exemplul ilustrat de Codul sursă 6.5, fişierul `test.pl` conţine un program exemplificativ pentru limbajul PL/0, care implică declararea de variabile, operaţii matematice cu acestea, afişări şi o structură repetitivă (buclă de tip `while`).

Limbajul PL/0 este un limbaj minimalist (având o sintaxă mult simplificată[1]), cu aplicaţii în domeniul educaţional, mai exact în studiul şi proiectarea compilatoarelor. A fost dezvoltat de informaticianul elvenţian Niklaus Wirth şi prezentat în cartea acestuia, *Algorithms + Data Structures = Programs* [14].

Codul sursă 6.5: Cod sursă PL/0.

```
var a,b,c;
begin
        a := 1;
        b := 2;
        print c;
        c := a + b;
        print c;
        if odd a then
                a := 0 - a;
        print a;
        b := 10;
        while b # 0 do
        begin
                print b;
                b := b - 1
        end
end.
```

Compilarea şi lansarea în execuţie a acestui program se realizează prin comanda:

```
make test
```

sau se poate explicita numele fişierului prin comanda:

```
make run < test.pl
```

[1] `https://en.wikipedia.org/wiki/PL/0`

Din nou, este folosit fişierul makefile de automatizare a întregului flux. Comenzile de mai sus în fapt substituie comanda:

```
java classpath .  parser < test.pl
```

După executarea parserului pe codul sursă din `test.pl`, pe ecran ar trebui să fie afişate valorile numerice din program în urma execuţiei comenzilor `print`. Urmăriţi codul sursă şi valorile afişate.

6.5 Temă de laborator

La specificaţiile compilatorului de PL/0 prezentate în prezentul laborator, adăugaţi codul necesar, atât în fişierul JLex cât şi în specificaţiile CUP pentru a implementa instrucţiunea repetitivă `for` şi lucrul cu metode (subprograme).