Limbaje Formale si Translatoare
Laborator 4

Februarie 2021

1 Scopul Lucrarii

1. Studiul generatorului de analizoare sintactice YACC
2. Efectuarea unor exercitii practice de utilizare a acestuia

2 Notiuni Teoretice

2.1 Introducere

Pornind de la o gramatica independenta de context, YACC-ul genereaza un set de tabele pentru un automat de analiza LALR(1)(*LookAhead*, *Left to right*, *Rightmost derivation*. Pentru a putea utiliza mediul YACC, trebuie urmati pasii urmatori.

1. Instalare: Pentru instalare, utilizati in terminal comanda

   ```
   sudo apt-get install bison
   ```

2. Compilare Yacc: YACC-ul se compileaza astfel:

   ```
   yacc [-vlt] [-o nume_fis.c] [-d] fisier_sursa_yacc.y
   ```

   unde:

   - **-v** genereaza fisierul *y.output*, care descrie tabelele de analiza si raporteaza conflictele generate de ambiguitatile din gramatica.
   - **-d** genereaza fisierul y.tab.h cu instructiunile #define ce asociaza codurile token fixate de YACC cu numele de tokeni declarate de utilizator. Aceasta optiune permite unor fisiere sursa, altele decat y.tab.c (implicit) respectiv nume_fis.c (optiunea -o) sa acceseze codurile token.

   Pentru o descriere completa a optiunilor, rulati $yacc\ -h$

3. Compilare lex & yacc: Pentru a putea compila programul generat de analizorul lexical si cel semantic, rulati:

   ```
   gcc [lex.yy.c/fisier_generat_lex.c] y.tab.c
   ```

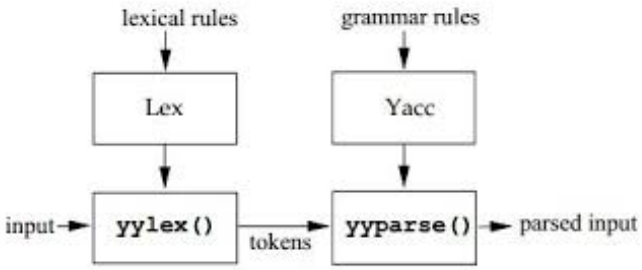

Figura 1: Analiza lexicala si sintactica cu generatoarele *lex* si *yacc*

Puteti folosi *-ly -ll* pentru a folosi functii predefinite in bibliotecile yacc/lex, precum main, yyerror; main din biblioteca yacc apeleaza functia yyparse.

In cazul in care aveti erori la utilizarea -ly, instalati libbison-dev (for Ubuntu:

```
apt install libbison-dev
```

Observatii:

- Cel mult un proces YACC poate fi activ intr-un director la un moment dat deoarece numele de fisiere generate sunt fixe: **y.tab.h** si **y.tab.c**
- Functia generata de YACC (*yyparse*) foloseste o functie de analiza lexicala. Se poate folosi functia **yylex** generata de *lex*, sau se poate implementa o functie (fig. 1)
  1. Functia de analiza lexicala returneaza urmatorul *toke*(numit si *simbol terminal* sau atom) de la intrare
  2. *yyparse* organizeaza *tokenii* pe baza *regulilor gramaticale*. O structura recunoscuta de analizorul sintactic (yyparse) se numeste $simbol\ non-terminal$.
  3. cand o regula gramaticala e potrivita, se invoca actiunea corespunzatoare (actiunea e optionala). Actiunile au posibilitatea de a returna valori si de a utiliza valorile altor actiuni.

Utilizatorul YACC-ului pregateste o specificatie a procesului de intrare, ce include reguli ce pot descrie:

1. structura de intrare (productiile gramaticii)
2. codul de executat la recunoasterea acestor reguli
3. o rutina low-level pentru a efectua procesul de intrare

Clasa specificatiilor acceptate e foarte generala: gramatici LALR(1) cu reguli de dezambiguizare.
In unele cazuri YACC-ul nu reuseste sa produca un analizor sintactic. De exemplu, specificatiile pot fi contradictorii sau ele pot cere un mecanism de recunoastere mai puternic decat cel disponibil in YACC. In prima situatie e vorba de erori, iar cea de-a doua poate fi in general corectata facand analizorul lexical mai puternic sau rescriind o parte din regulile gramaticale.

Continut fisier .l

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

%%
[a-z]    ;
0        return *yytext;
1        return *yytext;
```

Continut fisier .y

```
%{
#include <stdio.h>

int yylex();
void yyerror(const char *s);

%}
%%
S: '0' S '1'      {printf("Productia S->0S1");}
   |/*epsilon*/   {printf("Productia S->epsilon");}
  ;
```

Figura 2: Exemplu lex/yacc - recunoastere string 0^n1^n

2.2 Specificatii de Baza

Un fisier *.y* cu specificatii consta din 3 sectiuni separate prin %%:

```
declaratii
%%
reguli
%%
programe
```

Identificatorii ce apar in specificatii refera fie tokeni fie simboluri non-terminale. YACC-ul cere ca numele de tokeni sa fie declarate. In plus, adesea e de dorit sa se includa analizorul lexical ca parte a fisierului de specificatii, care poate sa includa si alte programe. Sectiunea de declaratii poate fi vida.

2.2.1 Reguli

Forma regulilor:

```
A: CORP;
```

unde A reprezinta un nume de *nonterminal*, iar CORP reprezinta o secventa de zero sau mai multe *nume* si *literale*. Doua puncte, si punct si virgula, sunt punctuatii YACC.

Elemente ce pot aparea in reguli:

- nume. Numele folosite in corpul unei reguli gramaticale pot reprezenta **tokeni** sau **non-terminale**.
  - Numele pot avea orice lungime,
  - Trebuie sa inceapa cu o litera
  - sunt case-sensitive
  - pot fi alcatuite din:

    ```
    [a-z],[A-Z], ., _, [0-9]
    ```

- literale. Un literal consta dintr-un caracter inclus intre apostroafe. Ca in C, "backslash"-ul, e un caracter "escape". Toate secventele "escape" din C sunt recunoscute:

```
'\n' = "newline"
'\r' = "return"
'\'' = apostrof
'\\' = "backslash"
'\t' = "tab"
'\b' = "backspace"
'\f' = "formfeed"
'\xxx' = xxx in octal
```

"Blanc"-urile, "tab"-urile si "newline"-urile sunt ignorate si ele nu pot sa apara in nume sau simboluri multi-caracter rezervate. Comentariile pot sa apara oriunde e legal un nume si ele trebuie incluse intre /*...*/. Sectiunea de reguli cuprinde una sau mai multe reguli gramaticale, de forma:

Exemplu Un exemplu de fisier cu specificatii yacc se poate observa in figura 2. Este folosita gramatica $(\{0,1\},\{S\},S,P)$, unde:

$$P = \{S \to 0S1|\varepsilon\}.$$

Limbajul acestei gramatici e format din siruri de 0 si 1 cu structura: incep cu 0, continua cu oricat de multi de 0, urmat de acelasi numar de 1. Analiza lexicala recunoaste caracterele '0' si '1', iar analiza sintactica verifica ordinea.

Observatie 1. Daca exista mai multe reguli gramaticale cu aceeasi parte stanga, se poate folosi bara verticala '|' pentru a evita rescrierea. In plus, punct si virgula de la sfarsitul unei reguli pot fi eliminate cand urmeaza o bara verticala.

Astfel regulile:

```
A : B C D;
A : E F;
A : G;
```

pot fi scrise si ca

```
A : B C D
  | E F
  | G
  ;
```

Nu e obligatoriu ca toate regulile gramaticale cu aceeasi parte stanga sa apara impreuna, dar e recomandat.

Observatie 2. Daca un simbol non-terminal recunoaste vid, aceasta se scrie de exemplu:

```
empty : ;
```

Observatie 3. Numele ce reprezinta tokeni trebuie declarate, si aceasta se face cel mai simplu folosind cuvantul cheie %token in sectiunea de declaratii:

```
%token  nume1 nume2 nume3 ...
```

Observatie 4. Orice nume nedefinit in sectiunea de declaratii e presupus a reprezenta un non-terminal. Orice simbol non-terminal trebuie sa apara in partea stanga a cel putin unei reguli.

Observatie 5. Dintre simbolurile non-terminale, simbolul de start are o importanta particulara. **Simbolul de start** reprezinta cea mai larga si mai generala structura descrisa de regulile gramaticale. Simbolul de start este considerat implicit a fi partea stanga a primei reguli gramaticale. E de dorit insa sa se declare simbolul de start in mod explicit, in sectiunea de declaratii, folosind:

```
%start      simbol_de_start
```

Observatie 6. Sfarsitul intrarii e semnalat de un token special, numit marcator de sfarsit ("end-marker"). Daca tokenii cititi pana la "end-marker" formeaza o structura ce se potriveste cu simbolul de start, dupa ce e intilnit "end-marker"-ul analizorul revine la apelantul sau si accepta intrarea.

Daca marcatorul de sfarsit e "vazut" in orice alt context, exista o eroare. Analizorul lexical are sarcina de a returna "end-marker"-ul. In mod normal acesta este o valoare de I/E ca "end-of-file" sau "end-of-record".

2.3 Analiza Lexicala

Utilizatorul trebuie sa furnizeze un analizor lexical care sa citeasca fluxul de intrare si sa comunice tokenii (cu valorile lor, daca se doreste) analizorului sintactic. Analizorul lexical e o functie ce returneaza intregi, numita yylex(). Functia returneaza numarul terminalului ("token-number"), reprezentand tipul de token citit. Daca exista o valoare asociata cu acel token, ea trebuie atribuita variabilei externe **yylval**.

2.3.1 Consistenta Tokenilor

Analizorul sintactic si cel lexical trebuie sa foloseasca aceleasi numere de token, pentru ca ele sa poata comunica corect. Numerele pot fi alese de YACC sau de utilizator. In ambele cazuri, se foloseste mecanismul #define din C pentru a permite analizorului lexical sa returneze simbolic aceste numere.

Sa presupunem ca numele de token DIGIT a fost definit in sectiunea de declaratii a fisierului de specificatii YACC. Portiunea relevanta a analizorului lexical ar putea arata astfel:

```
yylex()
{
    extern int yylval;  int c;
    c=getchar();
    switch(c) {
      case '0':
      case '1':
      .........
      case'9' :
                    yylval=c-'0';
                    return(DIGIT);
    }
}
```

Intentia este de a returna un cod de token DIGIT si valoarea sa numerica. Cand codul pentru analizorul lexical e plasat in sectiunea "programe" a fisierului de specificatii, identificatorul DIGIT va fi definit ca numarul de token asociat cu token-ul DIGIT. Acest mecanism conduce la analizoare lexicale limpezi si usor de modificat.

2.3.2 Tokeni Rezervati

O capcana ce poate aparea este necesitatea de a evita orice nume de token care e rezervat sau semnificativ in C sau in analizor. De exemplu, utilizarea numelor de token if sau while va cauza aproape sigur dificultati mari la compilarea analizorului lexical. Numele de token error e rezervat pentru tratarea erorilor si trebuie utilizat in mod corespunzator.

Dupa cum s-a mentionat, numerele de token pot fi alese de YACC sau de utilizator. Implicit, ele sunt alese de YACC. Numarul de token implicit pentru un literal caracter este valoarea numerica a caracterului in setul de caractere local. Celorlalte nume le sunt asociate numere de token incepand cu 257.

2.3.3 Criterii pentru Denumire

Pentru a asigna un numar unui token, inclusiv literalelor, prima aparitie a numelui de token sau a literalului in sectiunea de declaratii poate fi imediat urmata de un intreg nenegativ. Acest intreg e tratat a fi numarul de token al numelui sau al literalului. Numele si literalele ce nu sunt definite prin acest mecanism isi pastreaza definitia lor implicita.

E important ca toate numerele de token sa fie distincte. End_marker-ul trebuie sa aiba numarul de token 0 sau negativ. Acest numar de token nu poate fi redefinit de utilizator. Analizorul lexical trebuie sa fie pregatit sa returneze zero sau un numar negativ la intalnirea sfarsitului intrarii. Un instrument util pentru constructia de analizoare lexicale este LEX-ul, care genereaza analizoare potrivite pentru a lucra in armonie cu analizoarele sintactice generate de YACC.

2.4 Actiuni Yacc

Utilizatorul poate asocia fiecarei reguli gramaticale actiuni de realizat cand respectiva regula e recunoscuta in fluxul de intrare. Aceste actiuni pot returna valori si pot obtine valorile returnate de actiunile precedente. Mai mult, analizorul lexical poate returna valori pentru tokeni.

O actiune e o secventa de instructiuni C. Mai jos avem exemple de reguli gramaticale cu actiuni:

```
A    :        '(' B ')'          {
                                      hello(1, "abc");
                                 }
XXX :          YYY ZZZ           {
                                      printf ("a message\n");
                                      flag=25;
                                 }
```

Simbolul dolar, '$', e un semnal special pentru YACC. Pentru a returna o valoare, actiunea seteaza pseudo-variabila $$ la acea valoare. De exemplu, o actiune care nu face altceva decat sa returneze valoarea 1, e: $$= 1;

Pentru a obtine valori returnate de actiuni precedente sau de analizorul lexical se pot folosi pseudo-variabilele: $1, $2, ... , care refera valorile returnate de componentele din partea dreapta a regulii curente, citind de la stanga la dreapta. Astfel, pentru "A : B C D;" $2 are valoarea returnata de C, iar $3 valoarea returnata de D. Sa consideram:

```
expr : '(' expr ')' ;
```

Valoarea returnata de aceasta regula e de obicei valoarea lui expr dintre paranteze. Acest lucru poate fi precizat astfel:

```
expr : '(' expr ')' { $$=$2; }
```

Implicit, valoarea unei reguli e valoarea primului sau element, $1. Astfel, regulile gramaticale de forma: A : B; nu au nevoie de actiunea explicita $$=$1;.

In exemplele de mai sus, toate actiunile apar la sfarsitul regulilor. Uneori se doreste obtinerea controlului inainte ca o regula sa fie complet analizata. YACC-ul permite scrierea unei actiuni si in interiorul unei reguli. Aceasta actiune poate returna o valoare, accesibila actiunilor de la dreapta sa, si ea poate la randul ei accesa valorile returnate de simbolurile de la stanga sa. Astfel, in regula:

```
A : B            { $$=1;}
    C            { x=$2; y=$3;}
    ;
```

Efectul consta in setarea lui x la 1 si a lui y la valoarea returnata de C. Actiunile ce nu termina o regula, sunt tratate de YACC prin manipularea unui nou nume de simbol non-terminal si a unei noi reguli ce asociaza acest nume sirului vid. Actiunea interioara este actiunea "comutata" la recunoasterea acestei reguli adaugate. YACC-ul trateaza exemplul de mai sus astfel:

```
$ACT : /* empty */ { $$=1;}
     ;
A    : B $ACT C { x=$2; y=$3;}
     ;
```

In multe aplicatii, iesirea nu e realizata direct de actiuni. Se construieste mai degraba o strucura de date (cum ar fi un arbore de derivare) in memorie, si se aplica transformari respectivei structuri inainte de a se genera iesirea. Arborii de derivare sunt usor de construit, daca sunt date rutinele necesare pentru a construi si mentine structura arborescenta dorita. Sa presupunem ca exista o functie node(), scrisa astfel incit apelul node (L,n1,n2), creeaza un nod cu eticheta L, descendentii n1 si n2, si returneaza indexul nodului nou creat. Atunci arborele de derivare poate fi construit cu actiuni ca:

```
expr : expr '+' expr    { $$ = node('+', $1, $3); }
```

Utilizatorul poate defini alte variabile pentru a fi folosite de actiuni. Declaratiile si definitiile pot sa apara in sectiunea de declaratii, incluse intre marcatorii %{ si %} . Aceste declaratii si definitii au un domeniu global, deci sunt cunoscute atit de instructiunile actiune cit si de analizorului lexical. De exemplu, declaratia de mai jos plasata in sectiunea de declaratii face variabila var1 accesibila tuturor actiunilor.

```
%{ int var1 = 0; %}
```

Analizorul YACC foloseste numai identificatori ce incep cu 'yy', motiv pentru care utilizatorul trebuie sa evite astfel de nume.

3 Desfasurarea lucrarii

3.1 Testare Exemple

Se vor testa si analiza exemplele prezentate.

3.2 Expresii Matematice

Se va analiza urmatorul exemplu, ce prezinta specificatia YACC corespunzatoare gramaticii discutate la lucrarea anterioara (varianta 1):

```
E ->    T | E+T | E-T
        T -> F | T*F | T/F
        F -> n | (E)                .

%term NUMAR
%left '+' '-'
%left '*' '/'
%%
lista       :       /* vida */
            |       lista '\n'
            |       lista expr '\n' { printf("\t%d\n",$2); }
            ;
expr        :       NUMAR                   { $$=$1; }
            |       expr '+' expr   { $$=$1+$3; }
            |       expr '-' expr   { $$=$1-$3; }
            |       expr '*' expr   { $$=$1*$3; }
            |       expr '/' expr   { $$=$1/$3; }
            |       '(' expr ')'    { $$=$2; }
            ;
%%

#include <stdio.h>
#include <ctype.h>
char *cmd;

main(int argc, char* argv)
{   cmd=argv[0];
    yyparse();
}

yylex()
{   int c;
    while (((c=getchar())==' ') || (c=='\t'));
    if(c==EOF) return 0;
    if( isdigit(c) )
    {
        ungetc(c,stdin);
        scanf("%d",&yylval);
        return NUMAR;
    }
    return c;
}
```

```
yyerror(char* s)
{   fprintf(stderr,"%s : %s",cmd,s);
    return;
}
```

3.3 Calculator de Birou

Se va analiza urmatorul exemplu ce prezinta specificatia YACC pentru un mic calculator de birou, cu 26 de registri etichetati prin literele mici 'a' ...'z', si care accepta expresii aritmetice alcatuite cu operatorii +, -, *, /, % (modulo), & (SI pe bit), | (SAU pe bit) si atribuirea. Daca o expresie are atribuire pe nivelul superior, valoarea nu este tiparita. In caz contrar, valoarea obtinuta e tiparita. Ca in C, daca un intreg incepe cu 0 e presupus a fi octal, altfel e presupus a fi zecimal. Exemplul arata cum sunt folosite precedentele si asociativitatile si prezinta o acoperire simpla a erorilor. Simplificarea majora e in faza de analiza lexicala, care e mult mai simpla decat in cazurile uzuale, iar iesirea e produsa imediat, linie cu linie. De notat modul in care intregii zecimali si octali sunt cititi de catre regulile gramaticale, operatie care, probabil, ar putea fi mai bine facuta de catre analizorul lexical.

```
%{
#include <stdio.h>
#include <ctype.h>
int regs[26];
int baza;
%}
%start      lista
%token      CIFRA LITERA
%left       '|'
%left       '&'
%left       '+' '-'
%left       '*' '/' '%'
%left       MINUSUNAR
%%
lista       :
            |       lista   instr   '\n'
            |       lista   error   '\n'  { yyerrok; }
            ;
instr       :       expr                        { printf("%d\n",$1); }
            |       LITERA '=' expr             { regs[$1]=$3; }
            ;
expr        :       '(' expr ')'                { $$=$2; }
            |       expr '+' expr               { $$=$1+$3; }
            |       expr '-' expr               { $$=$1-$3; }
            |       expr '*' expr               { $$=$1*$3; }
            |       expr '/' expr               { $$=$1/$3; }
            |       expr '%' expr               { $$=$1%$3; }
            |       expr '&' expr               { $$=$1&$3; }
            |       expr '|' expr               { $$=$1|$3; }
            |       '-' expr %prec MINUSUNAR    { $$=-$2; }
            |       LITERA                      { $$=regs[$1]; }
            |       numar
```

```
            ;
numar       :       CIFRA                       {
                                                    $$=$1;
                                                    baza=($1==0)?8:10;
                                                }
            |       numar CIFRA                 { $$=baza*$1+$2; }
            ;

%%

yylex()
{
    int     c;
    while( (c=getchar()) == ' ');
    if(islower(c))
    {
        yylval=c-'a';
        return(LITERA);
    }
    if(isdigit(c))
    {
        yylval=c-'0';
        return(CIFRA);
    }
    return(c);
}
```

4 Intrebari si Dezvoltari

1. Folosind figura 2, creati fisiere cu specificatii lex si yacc pentru a recunoaste stringuri 0^n1^n.
2. Se vor studia exemplele din laborator pentru fisierele dc1, dc2.
3. Considerati ca avem un limbaj in care pe fiecare linie putem scrie cate un numar. Cu lex recunoasteti numerele si newline, orice altceva e considerat eroare. Cu yacc verificati daca aveti linii ce contin un singur numar, respectiv daca aveti mai multe astfel de linii si cresteti fiecare numar cu 10.
4. Se modifica problema anterioara, pe fiecare linie avem

   ```
   n: numar
   ```

   caracterul n, urmat de colon sau spatii, urmat de un numar. La fel ca la problema anterioara, cresteti numarul cu 10 daca fisierul e corect.
5. Folositi Yacc impreuna cu lex pentru exemplul db_config.
   (a) Realizati un scanner care doar verifica daca tokenii apar in ordinea corecta
   (b) Realizati un parser care si afiseaza valoarea pentru fiecare element configurabil (ex. db_type).
6. Sa se scrie specificatia YACC pentru o gramatica pentru expresii logice booleene.