

# Flex & Bison

Limbaje formale și translaatoare (Compilatoare)

March 18, 2020

Mihai-Lica Pura

# Cuprins

- ▶ flex (lex)
- ▶ bison (yacc)

# lex/flex - Cuprins

- ▶ Structura fisierului de intrare
- ▶ Funcționarea lex/flex
- ▶ lex/flex libl
- ▶ Analiza lexicala contextuala
- ▶ Depanarea

- ▶ este un generator de analizoare lexicale în limbajul C
  - ▶ **flex tokdef.l** → **lex.yy.c**
- ▶ **tokedef.l** - fișier de intrare pentru flex
  - ▶ definește *sintaxa atomilor lexicali* ai limbajului sursă (cu ajutorul *expresiilor regulate*)
  - ▶ definește *acțiunile* semantice asociate fiecărei clase de atomi astfel definită
  - ▶ definiția + acțiunea asociată = regulă
  - ▶ în mod obișnuit o acțiune se încheie cu returnarea tipului de atom lexical identificat, pentru a putea fi folosit în continuare de către analizorul sintactic

- ▶ **lex.yy.c** - fișierul generat de flex pe baza celui de intrare
  - ▶ reprezintă codul sursă al analizorului lexical definit
  - ▶ conține codul funcției `yylex()`

# Fisierul de intrare lex/flex

- Structura fisierului de intrare pentru lex/flex (fisierul .l) este:

```
%{  
define and include directives of cpp  
%}  
definitions
```

```
%%
```

```
rules
```

```
%%
```

```
user defined routines
```

## Ex1 din arhiva

```
%{  
int charcount=0, linecount=0;  
%}  
%%  
. charcount++;  
\n {linecount++; charcount++;}  
%%  
int main()  
{  
yylex(); // End-of-file Ctrl+D  
printf("There were %d characters in %d lines \n", charcount,  
linecount);  
return 0;  
}
```

## Fisierul de intrare lex/flex

- ▶ codul dintre `% {` si `% }` este inclus ca atare la începutul fisierului sursa generat
- ▶ regulile sunt formate din definitiile atomului lexical (expresii regulate) si actiuni corespunzatoare
- ▶ cele doua componente ale unei reguli sunt despartite printr-un spatiu
- ▶ la scrierea regulilor este obligatoriu sa se înceapa de pe prima coloana a fiecarei linii
- ▶ actiunea asociata unei definitii a unui atom lexical se scrie în limbajul de programare C. Daca ea este formata din mai multe instructiuni, acestea trebuie scrise între paranteze accolade.



## Fisierul de intrare lex/flex

- ▶ "user defined routines" poate contine definitii de functii
- ▶ componenta de baza este functia *main()* care contine apelul functiei *yylex()*, functie care reprezinta de fapt analizorul lexical generat
- ▶ daca aceasta parte este goala, lex/flex va adauga în mod automat o functie *main* implicita, de forma:

```
int main(int argc, char *argv[])  
{  
    yylex();  
    return 0;  
}
```

# Utilizarea lex/flex

- ▶ 1. Scrierea fisierului de intrare pentru lex/flex (e.g. tokdef.l)
- ▶ 2. Generarea fisierului sursa a analizorului lexical (lex.yy.c)
  - ▶ **lex tokdef.l SAU**
  - ▶ **flex tokdef.l**
- ▶ 3. Compilarea fisierului sursa a analizorului lexical si generarea executabilului
  - ▶ **cc lex.yy.c -o alex -ll SAU**
  - ▶ **cc lex.yy.c -o alex -lfl**
- ▶ 4. Lansarea in executie
  - ▶ **./alex**

## Ex2 din arhiva

```
%{  
#include<iostream>  
using namespace std;  
%}  
%%  
.  
 { cout<<"CAR";}  
\n {char*s = new char[4]; }  
%%  
int main()  
{  
cout<<"Incepe";  
yylex()  
cout<<"Gata";  
return 0;  
}
```

# Utilizarea lex/flex cu cod C++

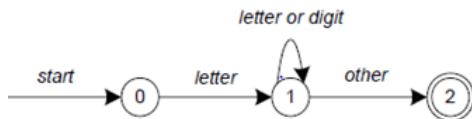
- ▶ 1. Scrierea fisierului de intrare pentru lex/flex (e.g. tokdef.l)
- ▶ 2. Generarea fisierului sursa a analizorului lexical (lex.yy.c)
  - ▶ **lex tokdef.l SAU**
  - ▶ **flex.tokdef.l**
- ▶ 3. Compilarea fisierului sursa a analizorului lexical si generarea executabilului
  - ▶ **g++ lex.yy.c -o alex -ll SAU**
  - ▶ **g++ lex.yy.c -o alex -lfl**
- ▶ 4. Lansarea in executie
  - ▶ **./alex**

# Functionarea lex/flex

- ▶ Expresiile regulate sunt traduse de catre lex/flex într-un program care implementează modul de lucru al unui automat finit determinist.
- ▶ Pe baza starii curente si a valorii urmatorului caracter de la intrare, se determina noua stare prin indexarea într-o tabela a starilor generata de catre program.

## Exemplu de functionare lex/flex

letter(letter|digit)\*



```
start:  goto state0
```

```
state0: read c  
        if c = letter goto statel  
        goto state0
```

```
statel: read c  
        if c = letter goto statel  
        if c = digit goto statel  
        goto state2
```

```
state2: accept string
```

## Arhitectura functiei *yylex()*

```
int yylex()
{...
new__symbol:
switch (c = getchar();){
case 'a': ... case 'Z': return identifierOrKeyword();
case '0': ... case '9': return number();
case '(': ... case ')': return specSymbol();
case '\\': return characterOrString();
case EOF: return YYEOF;
default: error(INVALID_TOKEN); goto new__symbol;
}
```

## Sectiunea definitiilor

- ▶ 1. Cod C - Codul dintre %{ si %} va fi copiat ca atare la începutul fisierului lex.yy.c.
  - ▶ directive catre preprocesor (include, definitii s.a.)
  - ▶ definitii de variabile
  - ▶ prototipurile functiilor care vor fi definite în a treia sectiune
  - ▶ ș.a.
- ▶ 2. Definitii de aliasuri pentru expresiile regulate – Definitiiile elementelor de limbaj care vor fi folosite în regulile din a doua sectiune
  - ▶ letter [a-zA-Z]
  - ▶ digit [0-9]
  - ▶ punct [,;:!?]
  - ▶ nonblank [^ \t]



## Sectiunea definitiilor

- ▶ 3. Definitii de stari - Atunci când o regula depinde de context, se pot defini stari care sa permita identificarea corespunzatoare a atomilor lexicali:
  - ▶ %s NUMESTARE

Exista o stare predefinita si anume starea INITIAL.

## Secțiunea regulilor

- ▶ Contine o serie de perechi definitie-actiune pentru câte o clasa de atomi lexicali
- ▶ Definițiile claselor de atomi lexicali sunt expresii regulate scrise si pe baza aliasurilor definite în prima secțiune
- ▶ Actiunile asociate sunt o instructiune C sau mai multe instructiuni C între paranteze acolade.
- ▶ Între definitie si actiunea asociata trebuie sa fie cel puțin un spatiu (unul sau mai multe spații, unul sau mai multe taburi, dar nu poate fi sfârșitul de linie)
- ▶ Adica actiunea asociata unei definitii este obligatoriu sa înceapa de pe aceeasi linie cu aceasta.

## Sectiunea regulilor

- ▶ Scrierea unei reguli trebuie sa înceapa de pe prima coloana a liniei.
- ▶ Orice text din sectiunea de reguli care nu începe de pe prima coloana a unei linii este copiat ca atare în fisierul sursa generat `lex.yy.c`.
- ▶ Daca mai multe definitii se potrivesc textului de intrare, se va lua acea definitie care potriveste textul cel mai lung.
- ▶ Daca mai multe definitii potrivesc un text de aceeasi lungime, va fi luata prima în ordinea în care au fost scrise în sectiunea de reguli.

## Secțiunea regulilor

- ▶ `.` Match any character except newlines.
- ▶ `\n` A newline character.
- ▶ `\t` A tab character.
- ▶ `^` The beginning of the line.
- ▶ `$` The end of the line.
- ▶ `<expr>*` Zero or more occurrences of the expression.
- ▶ `<expr>+` One or more occurrences of the expression.

## Secțiunea regulilor

- ▶ `<expr>?` Zero or one occurrences of the expression.
- ▶ `(<expr1>|<expr2>)` One expression or another.
- ▶ `[<set>]` A set of characters or ranges, such as `[a-zA-Z]`.
- ▶ `[^ <set>]` The complement of the set, for instance `[^ \t]`.
- ▶ `"a+b"` Liberal `"a+b"`; C escapes still work!
- ▶ `<expr> {n }` `n` occurrences of the expression.
- ▶ `<expr> {n, m }` `n`, `n+1`, `n+2`, ... , `m` occurrences of the expression.

## Secțiunea regulilor

▶ Expression	Matches
▶ abc	abc
▶ abc*	ab abc abcc abccc ...
▶ abc+	abc abcc abccc ...
▶ a(bc)?	one of: a, b, c
▶ [abc]	one of: a, b, c
▶ [a-z]	any letter, a-z

## Secțiunea regulilor

- ▶ `[a\ -z]` one of: a, -, z
- ▶ `[-az]` one of: -, a, z
- ▶ `[A-Za-z0-9]+` one or more alphanumeric characters
- ▶ `[\t \n]+` whitespace
- ▶ `[^ab]` anything except: a,b
- ▶ `[a^b]` one of: a, ^, b
- ▶ `[a|b]` one of: a, |, b
- ▶ `a|b` one of: a,b
- ▶ `(abc){2,4}` abcabc or abcabcabc  
or abcabcabcabc

## Secțiunea regulilor

- ▶ Dacă pentru anumite succesiuni de caractere nu a fost prevăzută nici o regulă, atunci acestora li se aplică **regula implicită lex/flex** și anume:  
**potrivirea și copierea caracter cu caracter la ieșire**
- ▶ Ex:
- ▶ Cel mai scurt fișier de definiții pentru (f)lex este:
  - ▶ %

Acțiunea: Intrarea este copiată în ieșire, caracter cu caracter.



- ▶ **char\*yytext** - succesiunea de caractere din intrare care se potrivește definiției curente
- ▶ **int yyleng** - lungimea succesiunii de caractere din intrare care se potrivește definiției curente (în fapt lungimea sirului de caractere din *yytext*)

## Ex3 din arhiva

```
%{  
int charcount=0, linecount=0, wordcount=0;  
%}  
letter[^ \t \n]  
%%  
{letter}+ {wordcount++; charcount+=yyleng;}  
. charcount++;  
\n {linecount++; charcount++;}  
%%  
int main()  
{  
yylex(); // End-of-File Ctrl+D  
printf("There were %d word with %d characters in %d  
lines \n".wordcount,charcount,linecount);  
return 0;  
}
```

- ▶ **FILE\* yyout**      **outputfile**
- ▶ **FILE\* yyin**        **inputfile**
- ▶ Functia *yylex()* citește textul de analizat prin intermediul variabilei *yyin*, și scrie ieșirea prin variabila *yyout*
- ▶ În mod implicit:
  - ▶ **yyin = stdin**, adică standard input (tastatura) (EOF pentru *stdin* se poate obține prin **CTRL+D**)
  - ▶ **yyout = stdout**, adică standard output (consola)
  - ▶
- ▶ în mod implicit analizorul lexical generat va analiza textul introdus de la tastatura și va scrie ieșirea în consolă

## Ex4 din arhiva

```
%{  
int lineno;  
%}  
%%  
^(.*)\n fprintf(yyout, "%4d \t %s", ++lineno, yytext);  
%%  
  
int main(int argc, char*argv[]){  
    yyin=fopen(argv[1], "r");  
    yyout=fopen(argv[2], "w");  
    yylex();  
    fclose(yyin);  
    fclose(yyout);  
}
```

- ▶ **int yywrap(void)**
- ▶ atunci când *yylex*, citind din *yyin*, ajunge la EOF se apeleaza automat functia *yywrap()*
- ▶ **yywrap()** returneaza
  - ▶ 1 daca nu mai exista intrare de analizat
  - ▶ 0 daca mai exista cel puțin o intrare de analizat
- ▶ functia **yywrap()** împreuna cu **yyin** si **yyout** poate fi folosita pentru a analiza consecutiv mai multe fisiere de intrare si respectiv pentru a scrie consecutiv în mai multe fisiere de iesire

## Ex5 din arhiva

```
%{
#include <string.h>
int second_file=0;
char *nume2;
int lineno;

}%
%%
^(.*) \n fprintf(yyout, "%4d \t % s", ++lineno, yytext);
%%
int main(int argc, char *argv[]) {
    nume2=(char*)malloc(sizeof(char)*(strlen(argv[2])+1));
    strcpy(nume2, argv[2]);
    yyin = fopen(argv[1], "r");
    yyout = fopen(argv[3], "w");
    yylex()
    fclose(yyin);
    fclose(yyout);
}
```

## Ex5 din arhiva

```
int yywrap()  
{  
    if(second_file ==0)  
    {  
        fclose(yyin);  
        yyin=fopen(ume2, "r");  
        second_file=1;  
        return 0;  
    }  
    else  
        return 1;  
}
```

- ▶ `int yymore(void)` - append the next token to *yytext*
- ▶ `int yyless(int n)` - truncate the current token to *n* characters
- ▶ `int input(void)` - extract the next symbol from input
- ▶ `int unput(int c)` - return the symbol *c* back to the input
- ▶ `int yylineno` - current line number



# Depanare lex/flex

- ▶ Codul generat de catre lex/flex în lex.yy.c contine si instructiunii de depanare.
- ▶ Acestea pot fi activate prin specificarea parametrului **-d** la etapa de generare a fisierului sursa a analizorului lexical.
- ▶ Adica:
  - ▶ lex -d tokdef.l
  - ▶ flex -d tokdef.l

# Dependenta de context

- ▶ Daca aplicarea unei reguli depinde de context, atunci analizorul lexical generat trebuie sa fie în masura sa faca o analiza în functie de context.
- ▶ Prin context se înțelege locatia în cadrul codului sursa care trebuie analizat a succesiunii de caractere la care se refera regula respectiva.
- ▶ Pentru a trata astfel de operatii, lex/flex permite utilizarea:
  - ▶ **right state**: aplicarea unei reguli depinde de ceea ce urmeaza dupa succesiunea de caractere respective
  - ▶ **left state**: aplicarea unei reguli depinde de ceea ce a fost înaintea succesiunii de caractere respective

# Dependenta de context

## Exemplu

- ▶ Sa se genereze un analizor lexical care sa modifice numele claselor, metodelor si variabilelor conform conventiei Java de denumire: numele claselor încep cu litera mare, iar numele variabilelor si ale metodelor încep cu litera mica.
- ▶ atomii lexicali corespunzatori celor trei tipuri de identificatori au aceeasi definitie, si anume:
  - ▶ `[a-zA-Z][a-zA-Z0-9]*`
- ▶ actiunea asociata definitiilor trebuie însa sa fie diferita:
  - ▶ pentru numele de clase trebuie capitalizata prima litera
  - ▶ pentru numele de variabile si de metode, prima litera trebuie sa fie transformata în litera mica
- ▶ **cum se face diferența?**

## Left state

```
public class MainClass {  
    private int aField  
  
    public void aMethod() {  
  
    }  
}
```

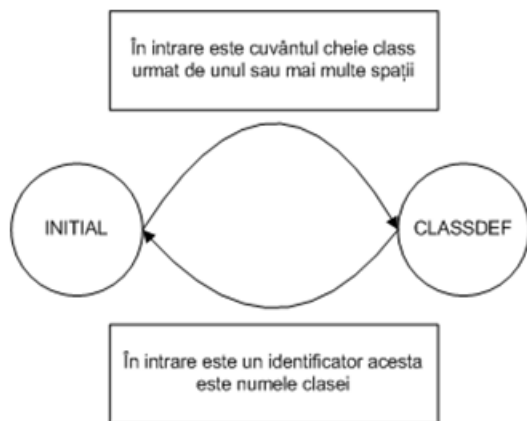
## Left state

- ▶ Daca gândim problema ca fiind una de dependenta de context de tip stânga, atunci:
  - ▶ daca **înainte** de identificator a fost cuvântul cheie **class** urmat de cel puțin un spatiu, atunci identificatorul este un nume de clasa si trebuie capitalizat primul lui caracter
  - ▶ daca înainte de identificator a fost orice altceva, atunci el este un nume de variabila sau de metoda si primul sau caracter trebuie transformat în litera mica

## Implementarea left state

- ▶ regula este prefixata cu numele unei stari, sub forma:  
**<NUME\_STARE> definitie\_atom\_lexical actiune**
- ▶ regula va fi evaluata numai daca analizorul lexical se afla în starea <NUME\_STARE> respectivă
- ▶ schimbarea starii curente a analizorului lexical se poate face în partea de actiuni a unei reguli cu ajutorul macroului BEGIN:  
**NUME\_STARE>definitie\_atom\_lexical {actiuni;  
BEGIN NUME\_ALTA\_STARE;}**
- ▶ toate starile trebuie definite în sectiunea de definitii a primei parti a fisierului de intrare printr-o instructiune de forma:  
**%s NUME\_STARE**
- ▶ Starea initiala a analizorului este **INITIAL**

## Implementare left state



## Ex7 din arhiva

```
%{  
    #include<string.h>  
}%  
%s CLASSDEF  
  
%%  
  
<INITIAL>  
[a-zA-Z][a-zA-Z0-9]*{yytext[0]=(char)tolower(yytext[0]);  
fprintf(yyout, "%s", yytext); }  
<INITIAL>"class"[]+{ ECHO; BEGIN CLASSDEF; }  
<CLASSDEF>  
[a-zA-Z][a-zA-Z0-9]*{yytext[0]=(char)tolower(yytext[0]);  
fprintf(yyout, "%s", yytext); BEGIN INITIAL;  
}  
.ECHO;  
\n ECHO;  
%%
```



## Right state

```
public class MainClass {  
    private int aField  
  
    public void aMethod() {  
  
    }  
}
```

## Right state

- ▶ Daca gândim problema ca fiind una de dependenta de context de tip dreapta, atunci:
  - ▶ daca dupa identificator urmeaza cel puțin un spatiu si **apoi o paranteza acolada deschisa**, atunci identificatorul este un nume de clasa si trebuie capitalizat primul lui caracter
  - ▶ daca dupa identificator urmeaza orice altceva, atunci el este un nume de variabila sau de metoda si primul sau caracter trebuie transformat în litera mica

# Implementarea right state

- ▶ specificarea contextului dreapta se face cu ajutorul caracterului  
/
- ▶ regula respectiva va avea forma:  
**definitie\_atom\_lexical/context actiune**
- ▶ succesiunea de caractere care defineste contextul dreapta ramâne în sirul de intrare si va fi analizata ulterior de catre analizorul lexical

## Ex6 din arhiva

```
%{  
    #include<string.h>  
}%  
%%  
[a-zA-Z] [a-zA-Z0-9]* / [\t \n]+[{]  
{yytext[0]=(char)toupper(yytext[0]); fprintf(yyout, "%s", yytext);}  
[a-zA-Z] [a-zA-Z0-9]*  
yytext[0]=(char)tolower(yytext[0]);fprintf(yyout, "%s", yytext);}  
.ECHO;  
\n ECHO;  
%%
```

# Exemplu

- Suppose we want to clean up sloppy spacing and punctuation in typed text. For example in this text:

*This    text    (all of it    )has occasional lapses , in  
punctuation(sometimes pretty bad) ,( sometimes not so) .*

*(Ha!) is this: fun? Or what!*

# Exemplu

We have

- ▶ Multiple consecutive blank line: those should be compacted.
- ▶ Multiple consecutive spaces, also to be compacted.
- ▶ Space before punctuation and after opening parentheses.
- ▶ Missing spaces before opening and after closing parentheses.
- ▶ The last item is a good illustration of where state comes in: a closing paren folloed by punctuation is allowed, but followed by a letter it is an error to be corrected.

## Ex8 din arhiva

```
punct [.,;!?]
text[a-zA-Z]
%%
)" " "+/ {punct}      {printf(" ");}
)""/{text}      {printf(" ");}
{text}+" " "+/" "      {while(yytext[yylen-1]==")yylen--;
ECHO;}
({punct}|{text}+)/"("      {ECHO; printf(" ");}
"(" " " "+/{text}      {while(yytext[yylen-1]==")yylen--;
ECHO;}
{text}+" " "+/{punct}
{while(yytext[yylen-1]==")yylen--; ECHO;}
^+      ;
" " "+      {printf(" ");}
.      {ECHO;}
\n / \n \n      ;
\n      {ECHO;}
```

## Ex9 din arhiva

punct[,.;!~?]

text [a-zA-Z]

%s OPEN

%s CLOSE

%s TEXT %s PUNCT

%%



## Ex9 din arhiva

```
" "+  
;   
<INITIAL>"(" {ECHO; BEGIN OPEN;}   
<TEXT>"(" {printf(" ") ECHO; BEGIN OPEN;}   
<PUNCT>"(" {printf(" ") ECHO; BEGIN OPEN;}   
)" " {ECHO; BEGIN CLOSE;}   
<INITIAL>{text}+ {ECHO; BEGIN TEXT;}   
<OPEN>{text}+ {ECHO; BEGIN TEXT;}   
<CLOSE>{text}+ {printf(" "); ECHO; BEGIN TEXT;}   
<TEXT>{text}+ {printf(" "); ECHO; BEGIN TEXT;}   
<PUNCT>{text}+ {printf(" "); ECHO; BEGIN TEXT;}   
{punct}+ {ECHO; BEGIN PUNCT;}   
\n {ECHO; BEGIN INITIAL;}   
%%
```

# Cuvinte rezervate

- ▶ Daca limbajul pentru care este generat analizorul lexical are un numar mare de cuvinte cheie, atunci este mai eficient sa folosim lex/flex pentru a potrivi identificatorii si sa alegem noi din cod care dintre acestia sunt cuvinte cheie, comparându-i, de exemplu, cu intrarile dintr-o tabela de cuvinte rezervate.
- ▶ adica in loc de:

```
"if"                return IF;  
"then"              return THEN;  
"else"              return ELSE;  
{letter}({letter}|{digit})* {  
    ▶ yytlval.id=symLookup(yytext);  
    ▶ return IDENTIFIER;  
    ▶ }
```

# Cuvinte rezervate

- ▶ sa se foloseasca:

```
{letter}({letter}|{digit})* {  
    int i;  
    if((i=resWord(yytext))!=0)  
        return(i);  
    yylval.id=symLookup(yytext);  
    return (IDENTIFIER);  
}
```

# Implementarea unui analizor lexical

- ▶ Utilizarea lex/flex pentru a implementa un analizor lexical pentru un limbaj de programare:
  - ▶ - definirea claselor de atomi lexicali
  - ▶ - ş.a
- ▶ Ex:

# Limitările lex/flex

- ▶ Cauze: provin din modul în care este implementat - prin automate finite deterministe
- ▶ Consecinta: lex/flex are numai stari si tranzitii între stari
- ▶ De exemplu, lex/flex nu poate fi utilizat pentru a recunoaste structuri îmbricate, cum ar fi parantezele
- ▶ Tratarea structurilor îmbricate cu ajutorul lui (f)lex se poate face numai prin utilizarea unei stive (implementata si gestionata de catre programator).

# Depanarea codului sursa

- 1 Tinem minte numarul liniei/liniilor din cadrul fisierului .I, de unde dorim sa facem debug (il vom folosi pentru a seta breakpoint-urile)

De exemplu, pentru a face debug in functia main, pentru fisierul **ex.I** cu continutul de pe slide-ul urmator, vom tine minte numarul **8**.

## Depanarea codului sursa

```
%{  
int lineno;  
%}  
%%  
^(.*)\n fprintf(yyout, "%4d \t %s", ++lineno, yytext);  
%%  
int main(int argc, char*argv[]){  
    colorredyyin=fopen(argv[1], "r";  
    yyout=fopen(argv[2], "w");  
    yylex();  
    fclose(yyin);  
    fclose(yyout);  
}
```

## Depanarea codului sursa

- 2 Folosim lex/flex pentru a genera codul sursa al instrumentului de analiza lexicala, pe baza fisierului ex.l.

**flex ex.l**

- 3 Compilam pentru debug codul sursa obtinut, adaugând optiunea -g pentru a genera un executabil

**gcc lex.yy.c -g -o executabil -lfl**



## Depanarea codului sursa

- 4 Lansam debuggerul **gdb** trasmitându-i ca si parametru numele executabilului pe care dorim sa îl depanam  
**gdb executabil**
- 5 Setam breakpoint-urile folosind comanda **-b sau break**, numele fisierului .l si numarul liniei pe care dorim sa setam breakpoint-ul  
**b ex.l:8**
- 6 Repetam pasul al 5-lea pentru fiecare breakpoint pe care dorim sa îl setam

# Depanarea codului sursa

## 7 Lansam procesul de depanare cu comanda **run**

- ▶ Daca executabilul asteapta sa primeasca argumente la lansarea în executie (de exemplu numele fisierelor de intrare si de iesire) acestea vor fi furnizate ca si argumente ale comenzii **run**. De exemplu

**run in.txt out.txt**

- ▶ Daca executabilul lucreaza cu *stdin* si *stdout*, dupa comanda **run** el va astepta sa introducem textul de analizat.

# Depanarea codului sursa

- 8 Programul va rula pâna la primul breakpoint întâlnit în executie si apoi se va opri afisând instructiunea de la acel breakpoint
  - ▶ În acel moment se va putea vizualiza valoarea unei variabile (echivalentul unui watch) cu comanda **p sau print** urmata de numele variabilei a carei valoarea dorim sa o vizualizam  
**p yytext**
- 9 Executia instructiunii curente si oprirea la instructiunea urmatoare se face cu comanda **n sau next**  
**n**

## Depanarea codului sursa

- 10 Executia programului pâna la urmatorul breakpoint setat se face cu comanda **c** sau **continue**

**c**

- 11 Iesirea din debugger se face cu comanda **q** sau **quit**

# Depanarea codului sursa

## Tutoriale gdb

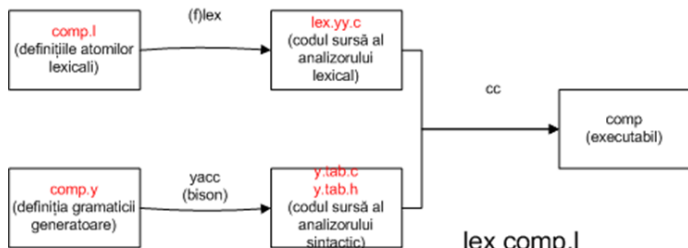
- ▶ <http://www.unknownroad.com/com/rtfm/gdbtut/>
- ▶ <http://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>
- ▶ <http://www.cs.cmu.edu/~gilpin/tutorial/>

## yacc/bison - Cuprins

- ▶ Structura fisierului de intrare
- ▶ Analiza sintactica
- ▶ Funcționarea yacc (bison)
- ▶ Precedența și asociativitatea operatorilor
- ▶ Conflicte shift/reduce
- ▶ Conflicte reduce/reduce
- ▶ Tratarea erorilor
- ▶ Depanarea
- ▶ Transmiterea informațiilor privind locația atomilor lexicali
- ▶ Transmiterea valorii semantice a atomilor lexicali
- ▶ Analiza semantica
- ▶ Generarea codului pentru limbajul expresiilor aritmetice

- ▶ “Yet another compiler’s compiler” este un generator de analizoare sintactice, capabile să parseze un șir de atomi lexicali, care, de obicei, este generat cu ajutorul instrumentului lex/flex.
- ▶ Dacă lex/flex are nevoie de definițiile atomilor lexicali pentru a fi în măsură să îi descopere într-un șir de intrare, yacc/bison are nevoie de definiția limbajului de analizat, dată printr-o gramatică generatoare.

- Generarea unui compilator cu ajutorul instrumentelor lex/flex și yacc/bison:



```
lex comp.l
yacc -d comp.y
cc lex.yy.c y.tab.c -o comp
./comp
```



# Fișierul yacc/bison

- ▶ Fișierul yacc/bison are aceeași structură ca și fișierul lex/flex:

- ▶ ... Definiții

%%

- ▶ ... Reguli

%%

- ▶ ... Funcții

# Secțiunea definiții

- ▶ Poate cuprinde:
- ▶ 1. Cod C/C++
  - Orice cod C/C++ scris între `%{` și `%}` este copiat ca atare în fișierul yacc/bison de ieșire.
  - În această subsecțiune se definesc, de obicei, variabilele și funcțiile care vor fi utilizate ulterior.
- ▶ 2. Definițiile numelor atomilor lexicali
  - În această subsecțiune se definesc numele atomilor lexicali pe care instrumentul de analiză sintactică care va fi generat îi recunoaște.  
e.g. `%token NUME_TOKEN`
  - La rularea yacc/bison, acesta generează un fișier header numit `y.tab.h` în care vor fi definite numele token-ilor folosind directiva `define`.  
e.g. `#define NUME_TOKEN 258`

## Secțiunea definiții

- În subsecțiunea de cod C/C++ a fișierului `lex/flex` corespunzător se va face `include` pentru fișierul `y.tab.h`, astfel încât analizorul lexical să poată utiliza denumirea atomilor lexicali așteptată de către `yacc/bison`.
- Instrumentul de analiză sintactică generat apelează automat funcția `yylex()`.
- La fiecare apel, aceasta trebuie să returneze numele unui atom lexical, așa cum a fost acesta definit în fișierul `yacc/bison`.

### ► 3. Reguli de asociativitate

- Se definesc reguli privind asociativitatea și prioritatea operatorilor.

## Secțiunea reguli

- ▶ Cuprinde regulile de producție ale gramaticii care definește limbajul țintă.
- ▶ Fiecărei părți drepte ale unei reguli de producție poate să îi fie asociată o acțiune.
- ▶ Acțiunile sunt scrise în cod C/C++, și apar între parantezele acolade.
- ▶ yacc/bison presupune că simbolul de start al gramaticii este primul simbol din secțiunea de reguli.
- ▶ Simbolul de start poate fi specificat și explicit prin declarația **%start NETERMINAL**, în prima secțiune.

## Secțiunea funcții

- ▶ Cuprinde definițiile funcțiilor care vor fi utilizate.
- ▶ Funcția main implicită pentru un compilator generat cu yacc/bison și lex/flex este:

```
int main()  
{  
    yyparse();  
    return 0;  
}
```

## Secțiunea funcții

- ▶ Funcția `yyparse()` reprezintă analizorul sintactic generat. Ea apelează automat funcția `yylex()` ori de câte ori are nevoie de următorul atom lexical.
- ▶ Fișierul `lex/flex` corespunzător nu va mai conține funcția `main`, deoarece cele două fișiere sursă generate pe rând de către `yacc/bison` și respectiv de către `lex/flex` vor fi compilate împreună, și deci este nevoie de o singură funcție `main`.
- ▶ Variabilele și funcțiile predefinite pentru lucrul cu fișiere (`yyin`, `yyout`, `yywrap()`) sunt valabile și pentru `yacc/bison`, putând fi utilizate exact în același mod ca și pentru `lex/flex`.

## Ex1 din arhiva - Analiza sintactica

- ▶ Generarea unui analizor sintactic pentru limbajul expresiilor aritmetice.

- ▶ 1. Definirea gramaticii pentru limbajul în cauză.

Ex:  $S \rightarrow E$  ;

$$E \rightarrow E+E \mid E-E \mid E * E \mid E / E \mid (E) \mid NR$$

- ▶ 2. Scrierea fișierului pentru yacc/bison:
  - Definirea numelor atomilor lexicali (terminalelor din gramatică)
  - Completarea regulilor de producție

## Ex1 din arhiva - Analiza sintactica

```
%{  
    #include <stdio.h>  
    int EsteCorecta = 0;  
%}
```

```
%token TOK_ PLUS TOK_ MINUS TOK_ MULTIPLY  
TOK_ DIVIDE TOK_ LEFT TOK_ RIGHT TOK_ ERROR  
%token TOK_ NUMBER
```

```
%start S
```

```
%left TOK_ PLUS TOK_ MINUS  
%left TOK_ MULTIPLY TOK_ DIVIDE
```



## Ex1 din arhiva

%%

S : E ';' { EsteCorecta = 1; }

;

E : E TOK\_PLUS E

|

E TOK\_MINUS E

|

E TOK\_MULTIPLY E

|

E TOK\_DIVIDE E

|

TOK\_LEFT E TOK\_RIGHT

|

TOK\_NUMBER

;

%%

## Ex1 din arhiva

```
int main()
{
    yyparse();
    if(EsteCorecta == 1)
    {
        printf("CORECTA");
    }
    else
    {
        printf("INCORECTA");
    }
    return 0;
}

int yyerror(const char *msg)
{
    printf("Error: %s", msg);
    return 1;
}
```

## Ex1 din arhiva - Analiza sintactica

- ▶ 3. Scrierea fișierului pentru lex/flex, având în vedere atomii lexicali (terminalele gramaticii) definiți în fișierul yacc/bison

## Ex1 din arhiva

```
%{  
    #include "y.tab.h"  
}%  
%%  
"+"      { return TOK_PLUS; }  
"_"      { return TOK_MINUS; }  
"*"      { return TOK_MULTIPLY; }  
"/"      { return TOK_DIVIDE; }  
"("      { return TOK_LEFT; }  
")"      { return TOK_RIGHT; }  
";"      { return ';;'; }  
[1-9][0-9]*|0 { return TOK_NUMBER; }  
.        { return TOK_ERROR; }  
%%
```

## Ex1 din arhiva - Analiza sintactica

### ► 4. Generarea executabilului.

```
yacc -d ex.y
```

```
lex ex.l
```

```
cc lex.yy.c y.tab.c -o ex -lf
```

### ► 5. Execuția

```
./ex <in.txt >out.txt
```

# Funcționarea yacc/bison

- ▶ Punctul de intrare în analiză pentru analizoarele sintactice generate cu ajutorul lui yacc este funcția `yyparse()`.
- ▶ Ori de câte ori analizorul sintactic are nevoie de următorul atom lexical din intrare, el apelează funcția `yylex()`. Aceasta îi întoarce identificatorul următorului atom lexical sau o valoare care arată că nu mai sunt atomi lexicali în intrare.
- ▶ Identificatorii atomilor lexicali sunt cei definiți prin `%token`.
- ▶ Analizoarele sintactice generate de către yacc implementează o analiză sintactică ascendentă: prin operații shift-reduce încearcă să reducă succesiunea de atomi lexicali la simbolul de start al gramaticii.

# Funcționarea yacc/bison

- ▶ Pe măsură ce yacc citește atomi lexicali, îi salvează într-o stivă (shift).
- ▶ În momentul în care ultimii  $n$  atomi lexicali de pe vârful stivei se potrivesc cu partea dreaptă a unei reguli de producție din secțiunea de reguli a fișierului \*.y, aceștia pot fi grupați împreună și înlocuiți cu partea stângă a regulii de producție respective (reduce).
- ▶ Yacc nu face însă o reducere imediat ce ultimii  $n$  atomi lexicali se potrivesc părții drepte ale unei reguli de producție.

## Funcționarea yacc (bison)

- ▶ Atunci când un atom lexical este întors de către `yylex()`, el nu este imediat shiftat pe stivă, ci devine mai întâi **lookahead token** (atomul lexical următor, așa cum este el folosit și în analiza LL(1)).
- ▶ Atunci când este posibilă o reducere, yacc se uită la valoarea lui lookahead token (care nu se află în stivă, ci reprezintă următorul atom lexical care va fi shiftat).
- ▶ În funcție de valoarea sa, yacc face una sau mai multe reduceri.
- ▶ Dacă nu mai este posibilă nicio reducere, lookahead token este shiftat pe stivă, se cere lui `yylex()` următorul atom lexical, iar atomul lexical întors de `yylex()` devine lookahead token.
- ▶ Lookahead token este stocat în variabila **yylchar** (variabilă predefinită). Când această variabilă nu conține un atom lexical, ea are valoarea **YYEMPTY**.



# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

<b>Intrare</b>	1+2!
<b>Stiva</b>	$\epsilon$

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR} !$
- ▶  $E \rightarrow \text{NR}$

Intrare	+2!
Stiva	NR

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

Intrare	+2!
Stiva	E

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

<b>Intrare</b>	+2!
<b>Stiva</b>	
+	
E	

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR} !$
- ▶  $E \rightarrow \text{NR}$

Intrare	
Stiva	!
NR	
+	
E	

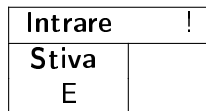
# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR} !$
- ▶  $E \rightarrow \text{NR}$

Intrare	
Stiva	!
E	
+	
E	

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR} !$
- ▶  $E \rightarrow \text{NR}$



# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

Intrare	$\epsilon$
Stiva	
!	
E	

Propoziția este  
incorectă?



# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

<b>Intrare</b>	1+2!
<b>Lookahead</b>	$\epsilon$

<b>Stiva</b>
$\epsilon$

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR} !$
- ▶  $E \rightarrow \text{NR}$

<b>Intrare</b>	+2!
<b>Lookahead</b>	NR

<b>Stiva</b>
$\varepsilon$

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR} !$
- ▶  $E \rightarrow \text{NR}$

<b>Intrare</b>	2!
<b>Lookahead</b>	+

<b>Stiva</b>
NR

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

<b>Intrare</b>	2!
<b>Lookahead</b>	+

<b>Stiva</b>
E

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR} !$
- ▶  $E \rightarrow \text{NR}$

<b>Intrare</b>	<b>!</b>
<b>Lookahead</b>	<b>NR</b>

<b>Stiva</b>
<b>+</b>
<b>E</b>

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

<b>Intrare</b>	$\epsilon$
<b>Lookahead</b>	!

<b>Stiva</b>
NR
+
E

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR} !$
- ▶  $E \rightarrow \text{NR}$

<b>Intrare</b>	$\epsilon$
<b>Lookahead</b>	$\epsilon$

<b>Stiva</b>
!
NR
+
E

# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

<b>Intrare</b>	$\epsilon$
<b>Lookahead</b>	$\epsilon$

<b>Stiva</b>
E
+
E



# Funcționarea yacc/bison

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow E ^ E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow NR !$
- ▶  $E \rightarrow NR$

<b>Intrare</b>	$\epsilon$
<b>Lookahead</b>	$\epsilon$

<b>Stiva</b>
E

Propoziția este  
corectă.

# Precedența și asociativitatea operatorilor

- ▶ Asociativitatea determină modul în care se grupează utilizările repetate ale aceluiași operator.
- ▶ Precedența arată modul în care un operator se grupează cu alți operatori.
- ▶ Asociativitatea atomilor lexicali ai gramaticii se definesc cu ajutorul lui:
  - **%left NUME\_TOKEN** – asociativitate stânga (pt  $x \text{ op } y \text{ op } z$  se grupează mai întâi  $x$  cu  $y$ ).
  - **%right NUME\_TOKEN** – asociativitate dreapta (pt  $x \text{ op } y \text{ op } z$  se grupează mai întâi  $y$  cu  $z$ ).
  - **%nonassoc NUME\_TOKEN** – nicio asociativitate: operatorul nu se poate afla de mai multe ori în același rând ( $x \text{ op } y \text{ op } z$  va fi considerată incorectă sintactic).

# Precedența și asociativitatea operatorilor

- ▶ Precedența atomilor lexicali ai gramaticii rezultă din modul în care a fost definită asociativitatea lor:
  - Atomii lexicali a căror asociativitate a fost definită în aceeași declarație `%left`, `%right` sau `%nonassoc` au aceeași precedență și se grupează potrivit asociativității lor.
  - Atomii lexicali a căror asociativitate a fost definită în declarații diferite au precedențe diferite, după cum urmează: cel pentru care a fost definită ultima dată asociativitatea are precedența cea mai mare, iar cel pentru care asociativitatea a fost definită prima dată are precedența cea mai mică.
- ▶ Ultimul pentru care a fost declarată asociativitatea va fi grupat primul, iar primul pentru care a fost declarată asociativitatea va fi grupat ultimul.

# Precedența și asociativitatea operatorilor

► Ex:

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

► Rezultă:

1. Toate terminalele au asociativitate stânga.
2. ! Are precedența mai mare decât \* și /, iar \* și / au precedența mai mare decât + și -.

## Precedența contextuală

- ▶ Precedența unui operator, care depinde de context, se specifică cu ajutorul declarației **%prec**.
- ▶ Ex: Operatorul minus (-) poate fi operator binar (reprezintă scăderea) sau operator unar (reprezintă semnul unui număr). Atunci când este semn, minus are precedența mai mare decât atunci când reprezintă scăderea.

```
%left '-'  
%left UMINUS  
E → E '-' E  
    | '-' E %prec UMINUS  
    | NR
```

- ▶ Regula '-' E are aceeași precedență ca și UMINUS, deci mai mare decât '-', conform declarațiilor de asociativitate.

# Conflicte shift/reduce

► Ex:

DECL\_IF  $\rightarrow$  if EXP then DECL  
                  |  
                  if EXP then DECL else DECL

- Pe parcursul analizei sintactice, după ce atomii lexicali “if EXP then DECL” sunt shiftați pe stivă, “else” devine lookahead token.
- În momentul acela există două posibilități de continuare:
1. Să reducă “if EXP then DECL” la DECL\_IF.
  2. Să îl shifteze pe “else” pe stivă, pentru ca ulterior să reducă “if EXP then DECL else DECL” la DECL\_IF.

## Conflicte shift/reduce

- ▶ Conform regulilor de producție, ambele operații posibile (atât operația shift, cât și operația reduce) sunt corecte. Rezultă **un conflict shift/reduce**.
- ▶ Modul implicit în care yacc rezolvă aceste conflicte este executând operația shift.
- ▶ Totuși, utilizatorul va fi informat asupra conflictului printr-un mesaj de atenționare.
- ▶ Pentru a evita aceste mesaje de atenționare relativ la conflictele shift/reduce, se poate utiliza declarația **%expect n**: yacc nu va genera mesaje de atenționare atâta timp cât numărul de conflicte shift/reduce este exact n.

# Conflicte shift/reduce

- ▶ Un alt mod de rezolvare a acestor conflicte este utilizarea precedenței contextuale (precedența unui operator depinde de contextul în care acesta se află).
- ▶ Ex:
  - 1. Definim un atom lexical `ifx`, care să reprezinte instrucțiunea `if` fără `else`.
  - 2. Declarăm `ifx` și `else` ca fiind neasociative, și astfel încât `else` să aibă precedența mai mare decât `ifx`.  
`%nonassoc ifx`  
`%nonassoc else`  
(Deoarece asociativitatea lui `else` este definită după cea a lui `ifx`, atunci `else` are precedență mai mare decât `ifx`.)



## Conflicte shift/reduce

- 3. Utilizăm declarația de precedență contextuală **%prec** pentru a specifica că `if` fără `else` are aceeași precedență ca și `ifx`:

```
DECL_IF → if EXP then DECL %prec ifx  
         |  
         if EXP then DECL else DECL
```

În consecință, “`if EXP then DECL`” are aceeași precedență ca și `ifx`, adică mai mică decât a lui `else`. Prin urmare, yacc va face shift, deplasând pe `else` pe stivă.

# Conflicte reduce/reduce

- Un conflict reducere/reducere apare atunci când pentru reducerea unei secvențe de atomi lexicali pot fi aplicate mai multe reguli de producție.

- Ex:

propoziție  $\rightarrow \varepsilon$   
                   $\rightarrow$  cuvântul\_poate  
                   $\rightarrow$  propoziție cuvânt  
cuvântul\_poate  $\rightarrow \varepsilon$   
                   $\rightarrow$  cuvânt

- Un cuvânt poate fi redus la o propoziție prin mai multe căi:
  1. Un cuvânt este redus la cuvântul\_poate, iar acesta este redus la o propoziție.
  2.  $\varepsilon$ , este redus la propoziție, iar propoziție împreună cu cuvânt este redus la propoziție

## Conflicte reduce/reduce

- ▶ Acest tip de conflicte nu afectează validitatea analizei, indiferent de reducerile utilizate: o propoziție validă va fi considerată validă, iar una incorectă va fi găsită ca fiind incorectă.
- ▶ Ceea ce diferă este însă ieșirea programului: aceasta depinde de acțiunea asociată fiecărei reguli de producție. Deci, în funcție de regula utilizată la reducere se va executa o altă acțiune.
- ▶ Yacc rezolvă implicit aceste conflicte aplicând cu prioritate regula care apare prima în secțiunea de reguli a fișierului \*.y.
- ▶ Modul implicit de rezolvare a acestor conflicte este, de obicei, riscant.
- ▶ Cel mai corect este studierea regulilor de producție ale gramaticii și eliminarea manuală a conflictelor.

# Tratarea erorilor

- ▶ Comportamentul implicit al yacc în momentul în care întâlnește o eroare sintactică este să oprească analiza, afișând un mesaj corespunzător.
- ▶ În majoritatea cazurilor ar fi mult mai folositor ca yacc să continue analiza sintactică pentru a descoperi toate erorile din intrare.
- ▶ Pentru a permite utilizatorului să controleze procesul de tratare a erorilor, yacc pune la dispoziție un atom lexical special cu numele **error**.
- ▶ Acesta poate fi utilizat în secțiunea de reguli a fișierului \*.y, la scrierea regulilor de producție ale gramaticii care definește limbajul țintă, pentru a arăta locurile în care se așteaptă să apară erori.

# Tratarea erorilor

- ▶ La întâlnirea unei erori sintactice, yacc scoate de pe stivă câte un element, până în momentul în care atomul lexical **error** este legal (apare într-un context care corespunde regulilor de producție ale gramaticii).
- ▶ Apoi, el se comportă ca și cum error ar fi lookahead token-ul curent, și execută acțiunea corespunzătoare.
- ▶ Lookahead token este apoi resetat la atomul lexical care a generat eroarea.
- ▶ După detectarea unei erori, analizorul sintactic rămâne în starea error până când trei atomi lexicali sunt citați și deplasați cu succes pe stivă.
- ▶ Dacă pe timpul în care analizorul se află în starea error este detectată o nouă eroare, atunci nu se mai afișază un mesaj de eroare, iar atomul lexical de intrare este șters în mod silențios.

# Tratarea erorilor

- ▶ O regulă de forma:

Instrucțiune : error ';'

- ▶ Înseamnă că la apariția unei erori analizorul sintactic va încerca să treacă peste instrucțiunea care a cauzat eroarea, și va face acest lucru sărind până la următorul ';'.
- ▶ Toți atomii lexicali de după error, și înainte de ';' vor fi eliminați, deoarece nu pot fi deplasați pe stivă.
- ▶ La întâlnirea lui ';' error ';' vor fi reduse la Instrucțiune și se va executa acțiunea de "curățare" asociată.

# Tratarea erorilor

- ▶ Se pot utiliza și reguli generale, de forma:

**Instrucțiune : error**

- ▶ Dar în acest caz analizorul sintactic va încerca să treacă peste instrucțiunea care a cauzat eroarea într-un mod la fel de general: folosind analizorul lexical, el va căuta următorul grup de trei atomi lexicali care ar putea urma legal unei instrucțiuni, și va relua analiza începând cu primul dintre aceștia.
- ▶ Dacă începutul unei instrucțiuni nu este suficient de diferit față de ceea ce ar putea apărea în interiorul acesteia, yacc ar putea să găsească un start fals în interiorul instrucțiunii respective, și să genereze o a doua eroare, deși este vorba de aceeași.

# Tratarea erorilor

- ▶ `yerror`;
  - Macro care permite forțarea analizorului sintactic să creadă că s-a revenit complet dintr-o eroare. Pentru erorile care apar ulterior acestui apel se vor genera mesaje corespunzătoare.
- ▶ `yyclearin`;
  - Macro care permite ștergerea lookahead tokenului curent.
- ▶ `YYRECOVERING`;
  - Acest macro reprezintă o expresie care are valoarea 1 dacă analizorul sintactic este în procesul de revenire dintr-o eroare, și 0 în rest.
- ▶ `YYERROR` ;
  - Cauzează imediat o eroare sintactică. Analizorul va începe procesul de revenire din eroare, ca și cum ar fi găsit-o el însuși, dar nu va afișa niciun mesaj. Pentru a afișa un mesaj corespunzător, trebuie apelată manual funcția `yerror`, înainte de apelul macroului `YYERROR`.
- ▶ `YYABORT` ;
  - Revenire imediată din `yyparse()`, cu indicarea unei erori.
- ▶ `YYACCEPT` ;
  - Revenire imediată din `yyparse()`, cu indicarea succesului.



## Ex2 din arhiva - Tratarea erorilor

```
%{  
    #include "y.tab.h"  
}%  
  
%%  
"+"      { return TOK_PLUS; }  
"_"      { return TOK_MINUS; }  
"*"      { return TOK_MULTIPLY; }  
"/"      { return TOK_DIVIDE; }  
"("      { return TOK_LEFT; }  
")"      { return TOK_RIGHT; }  
";"      { return ';;' ; }  
[1-9][0-9]* { return TOK_NUMBER; }  
\n        { ; }  
.  
%%
```

## Ex2 din arhiva

```
%{  
    #include <stdio.h>  
    int EsteCorecta = 1;  
  
    int yydebug = 1;  
}%
```

```
%token TOK_PLUS TOK_MINUS TOK_MULTIPLY  
TOK_DIVIDE TOK_LEFT TOK_RIGHT TOK_ERROR  
%token TOK_NUMBER
```

```
%start S
```

```
%left TOK_PLUS TOK_MINUS  
%left TOK_MULTIPLY TOK_DIVIDE
```

## Ex2 din arhiva

%%

S :

|  
E ';' S

|

error ';' S

{ EsteCorecta = 0;}

;

E : E TOK\_PLUS E

|

E TOK\_MINUS E

|

E TOK\_MULTIPLY E

|

E TOK\_DIVIDE E

|

TOK\_LEFT E TOK\_RIGHT

|

TOK\_NUMBER

;

## Ex2 din arhiva

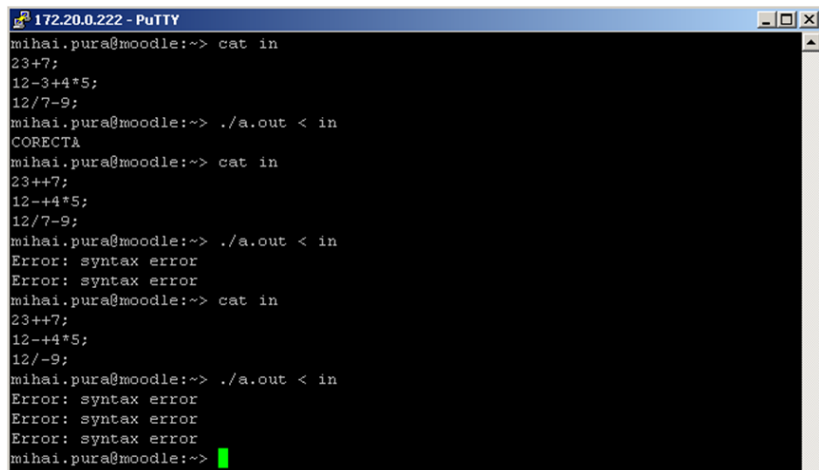
```
int main()
{
    yyparse();

    if(EsteCorecta == 1)
    {
        printf("CORECTA \n");
    }

    return 0;
}

int yyerror(const char *msg)
{
    printf("Error: %s \n", msg);
    return 1;
}
```

## Ex2 din arhiva



```
172.20.0.222 - PuTTY
mihai.pura@moodle:~> cat in
23+7;
12-3+4*5;
12/7-9;
mihai.pura@moodle:~> ./a.out < in
CORECTA
mihai.pura@moodle:~> cat in
23++7;
12-+4*5;
12/7-9;
mihai.pura@moodle:~> ./a.out < in
Error: syntax error
Error: syntax error
mihai.pura@moodle:~> cat in
23++7;
12-+4*5;
12/-9;
mihai.pura@moodle:~> ./a.out < in
Error: syntax error
Error: syntax error
Error: syntax error
mihai.pura@moodle:~> █
```

# Depanarea

- ▶ Activarea pentru analizorul sintactic generat a facilităților de depanare oferite de de către yacc se face astfel:

- 1. La crearea fișierului \*.y, trebuie adăugată declarația:

`int yydebug = 1;`

în secțiunea de cod C a primei secțiuni a fișierului (adică între `%{` și `%}`).

SAU

Se va folosi directiva `%debug`, în secțiunea de definiții a primei secțiuni a fișierului

- 2. La generarea fișierului sursă al analizorului sintactic, trebuie adăugate opțiunile `-debug` și `-verbose`:

`yacc -d *.y -debug -verbose`

# Depanarea

- ▶ Aceste modificări vor duce la:
  - 1. Generarea fișierului **y.output**, al cărui conținut explică automatul finit implementat pentru analiza sintactică a gramaticii date.
  - 2. La rularea analizorului sintactic, se vor afișa informații cu privire la ceea ce se întâmplă în cadrul analizei.
- ▶ La adresa:

## Debugging Lex, Yacc, etc.

<http://www.cs.man.ac.uk/~pjj/cs2121/debug.html>

se găsesc unele dintre cele mai obișnuite erori, precum și modalitățile în care pot fi ele corectate.

## Transmiterea informatiilor privind locatia atomilor lexicali

- ▶ Analizorul lexical (`yylex()`) poate sa transmita analizorului sintactic informatii privind locatia atomilor lexicali în fisierul de intrare prin utilizarea variabilei `yylloc`.
- ▶ Aceasta este o variabila predefinita:
- ▶ `extern YYSTYPE yyloc;`
- ▶ `yylloc` poate fi accesata în codul asociat definitiei unui atom lexical si reprezinta informatiile de locatie pentru atomul lexical respectiv.
- ▶ `YYSTYPE` reprezinta o structure de date cu patru membri:
- ▶ `struct {`
- ▶ `int first_line, last_line;`
- ▶ `int first_column, last_column;`
- ▶ `};`
- ▶ `yylex()` trebuie deci sa completeze valoarea unora sau a tuturor membrilor variabilei `yylloc` cu valorile corespunzatoare calculate pe baza fisierului de intrare.



# Transmiterea informatiilor privind locatia atomilor lexicali

- ▶ Accesarea informatiilor privind locatia atomilor lexicali, se face în fisierul \*.y în codul asociat partii drepte a unei reguli de productie prin utilizarea variabilelor @i, cu i de la 1 la numarul de elemente ale partii drepte a regulii de productie respective.
- ▶ Variabilele @i sunt de tipul YYLTYPE, la fel ca si variabila yylloc. În consecinta, ele au cei patru membrii ai structurii respective.
- ▶ Bineînțeles, se pot obtine valori numai pentru membrii structurii care au fost completati de catre yylex().
- ▶ În prima sectiune a fisierului \*.y se va folosi directiva %locations.

# Transmiterea valorii semantice a atomilor lexicali

- ▶ La fiecare apel al funcției `yylex()`, analizorul lexical returnează tipul următorului atom lexical, așa cum au fost definiți aceștia în fișierul `*.y`, prin **`%token`**.
- ▶ Acest mecanism este suficient pentru analiza sintactică.
- ▶ În cazul în care se dorește scrierea unui interpretor sau generarea de cod, pe lângă tipul atomilor lexicali este nevoie și de valoarea acestora.
- ▶ În cele două exemple de până acum am scris un analizor sintactic pentru limbajul instrucțiunilor aritmetice. În cazul scrierii unui interpretor sau a generării de cod, pentru atomul lexical `NUMAR` este nevoie și de valoarea acestuia, adică de valoarea numerică a succesiunii de caractere care s-a potrivit definiției acestuia.

# Transmiterea valorii semantice a atomilor lexicali

- ▶ În acest scop se poate utiliza variabila predefinită `yylval`.
- ▶ Aceasta este declarată (în fișierul `y.tab.c`), în mod implicit, sub forma:

```
YYSTYPE yylval;
```

- ▶ În mod implicit, tipul de date `YYSTYPE` este definit în fișierul `y.tab.c`, ca fiind:

```
typedef int YYSTYPE;
```

- ▶ `yylval` reprezintă valoarea semantică a lookahead token-ului.
- ▶ În fișierul `*.l`, în codul asociat definiției unui atom lexical, `yylval` poate fi utilizată pentru a transmite analizorului sintactic valoarea semantică a atomului lexical respectiv.

# Transmiterea valorii semantice a atomilor lexicali

- ▶ Pentru a utiliza variabile de tip **struct** sau **class** in interiorul lui **%union**, putem utiliza una dintre cele două variante de mai jos:

- 1. se declară structura sau clasa chiar in **interiorul lui %union**:

```
%union int a; struct Exemplu int a;char b; Ex;
```

- 2. folosim **%code requires** inainte de **%union**:

```
%code requires  
{  
    typedef struct exemplu int a; char b;          EXEMPLU;  
}  
%union int a; EXEMPLU x;
```

- 3. includem definiția intr-un fișier header pe care îl includem in .l înainte de orice și cumva la fel si in .y.

# Transmiterea valorii semantice a atomilor lexicali

- ▶ În fișierul \*.y, în codul asociat părții drepte a unei reguli de producție, se poate accesa valoarea semantică a atomilor lexicali sau a neterminalelor prin utilizarea variabilelor **\$i**, cu i de la 1 la numărul de elemente ale părții drepte a regulii de producție respective.
- ▶ Analog, **\$\$** reprezintă valoarea semantică care va asocia noului neterminal din vârful stivei, după ce are loc reducerea acestuia conform regulii de producție respective.
- ▶ În mod implicit, yacc/bison consideră neterminalele și atomii lexicali ca având o valoarea semantică de tip **int**.

## Ex3 din arhiva - Transmiterea valorii semantice a atomilor lexicali

```
%{  
    #include "y.tab.h"  
    extern int yylval;  
    int lineNo = 1;  
    int colNo = 1;  
%}  
%%  
"+"      { colNo++; return TOK_PLUS; }  
"_"      { colNo++; return TOK_MINUS; }  
"*"      { colNo++; return TOK_MULTIPLY; }  
"/"      { colNo++; return TOK_DIVIDE; }  
"("      { colNo++; return TOK_LEFT; }  
")"      { colNo++; return TOK_RIGHT; }  
";"      { colNo++; return ';' ; }  
0|[1-9][0-9]* { colNo+=strlen(yytext); yylloc.first_line = lineNo;  
yylloc.first_column = colNo; yylval = atoi(yytext); return  
TOK_NUMBER; }  
.  
\n      { lineNo++;colNo=1; }  
%%
```

## Ex3 din arhiva

```
%{
```

```
    #include <stdio.h>
    int EsteCorecta = 1;
    char msg[50];
```

```
%}
```

```
%token TOK_ PLUS TOK_ MINUS TOK_ MULTIPLY
TOK_ DIVIDE TOK_ LEFT TOK_ RIGHT TOK_ ERROR
%token TOK_ NUMBER
```

```
%start S
```

```
%left TOK_ PLUS TOK_ MINUS
```

```
%left TOK_ MULTIPLY TOK_ DIVIDE
```

## Ex3 din arhiva

%%

S :

|  
E ';' S { printf("= %d \n", \$1); }

|  
error ';' S  
    { EsteCorecta = 0; }  
;



## Ex3 din arhiva

```
E : E TOK_PLUS E { $$ = $1 + $3; }
    |
    | E TOK_MINUS E { $$ = $1 - $3; }
    |
    | E TOK_MULTIPLY E { $$ = $1 * $3; }
    |
    | E TOK_DIVIDE E
      {
        if($3 == 0)
        {
          sprintf(msg,"%d:%d Eroare semantica: Impartire la zero!",
@1.first_line, @1.first_column);
          yyerror(msg);
          YYERROR;
        }
        else { $$ = $1 / $3; }
      }
    |
    | TOK_LEFT E TOK_RIGHT { $$ = $2; }
    |
    | TOK_NUMBER { $$ = $1; }
;

%%
```

## Ex3 din arhiva

```
int main()
{
    yyparse();

    if(EsteCorecta == 1)
    {
        printf("CORECTA \n");
    }

    return 0;
}

int yyerror(const char *msg)
{
    printf("Error: %s \n", msg);
    return 1;
}
```

# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

Intrare	1+2	yyval
Lookahead	$\epsilon$	$\epsilon$

Stiva
$\epsilon$

# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

<b>Intrare</b>	1+2	<b>yyval</b>
<b>Lookahead</b>	NR	1

<b>Stiva</b>
$\epsilon$

# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

Intrare	1+2	yyval
Lookahead	+	$\epsilon$

Stiva	
NR	(\$1 = 1)

# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

Intrare	1+2	yyval
Lookahead	+	$\epsilon$

Stiva	
E	$$$ = \$1$

# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

Intrare	1+2	yyval
Lookahead	NR	2

<b>Stiva</b>	
+	
E	(\$2 = 1)

# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

<b>Intrare</b>	1+2	yyval
<b>Lookahead</b>	$\epsilon$	$\epsilon$

<b>Stiva</b>	
NR	(\$1 = 2)
+	
E	(\$3 = 1)



# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

Intrare	1+2	yyval
Lookahead	$\epsilon$	$\epsilon$

<b>Stiva</b>	
E	$$$ = \$1$
+	
E	$(\$3 = 1)$

# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

<b>Intrare</b>	1+2	yyval
<b>Lookahead</b>	$\epsilon$	$\epsilon$

<b>Stiva</b>	
E	(\$1 = 2)
+	
E	(\$3 = 1)

# Transmiterea valorii semantice a atomilor lexicali

- ▶  $E \rightarrow E + E$
- ▶  $E \rightarrow E - E$
- ▶  $E \rightarrow E * E$
- ▶  $E \rightarrow E / E$
- ▶  $E \rightarrow (E)$
- ▶  $E \rightarrow \text{NR}$

Intrare	1+2	yyval
Lookahead	$\epsilon$	$\epsilon$

Stiva	
E	$$$ = \$1 + \$3$

# Transmiterea valorii semantice a atomilor lexicali

- ▶ La scrierea unui interpretor sau a unui compilator pentru un limbaj complex, este insuficient mecanismul de transmitere a valorilor atomilor lexicali prin variabile `yylval` de tip `YYSTYPE` de tip întreg.
- ▶ Atunci când atomii lexicali ai limbajului au tipuri de valori semantice diferite, `YYSTYPE` poate fi definit de tipul **union**, astfel încât variabila `yylval` să aibă un membru special, corespunzător tipului fiecăruia dintre atomii lexicali.
- ▶ Declarația acestuia se face în fișierul `*.y`, prin:
- ▶ **`%union { membrii structurii de tip union ... }`**

# Transmiterea valorii semantice a atomilor lexicali

- ▶ Ca urmare a acestei declaratii, header-ul y.tab.h va contine definitia corespunzatoare a tipului de date YYSTYPE:
- ▶ `typedef union {`
- ▶  `membrii structurii de tip union ...`
- ▶ `} YYSTYPE;`
- ▶ Dupa declararea uniunii, trebuie asociat fiecarui atom lexical cu valoarea semantica, câte un membru al acesteia.
- ▶ Referirea unui membru al uniunii în fisierul \*.y se face prin constructia:
- ▶ `<num_membru_uniune>`

# Transmiterea valorii semantice a atomilor lexicali

- ▶ Asocierea dintre un membru al uniunii si un atom lexical se poate face prin una din declaratiile `%token`, `%left`, `%right` si `%nonassoc`.
- ▶ `%token <nume_membru>` lista nume atomi lexicali
- ▶ `%left <nume_membru>` lista nume atomi lexicali
- ▶ `%right <nume_membru>` lista nume atomi lexicali
- ▶ `%nonassoc <nume_membru>` lista nume atomi lexicali

# Transmiterea valorii semantice a atomilor lexicali

- ▶ Analog, se va utiliza declaratia **%type** pentru a asocia un membru al uniunii, unui neterminal.
- ▶ `%type <nume_membru> lista_neterminale`
- ▶ Acest lucru este necesar pentru codul asociat partii drepte a unei reguli de productie, în care se da o valoare semantica noului vârf al stivei, cel rezultat în urma reducerii conform regulii de productie respective.

## Transmiterea valorii semantice a atomilor lexicali

- ▶ În cazurile în care mecanismul descris este insuficient, se poate preciza “pe loc” tipul valorii semantice a unui atom lexical sau a unui neterminal.
- ▶ Acest lucru se face inserând după primul \$ numele membrului uniunii (între <>), care va fi asociat elementului respectiv (atom lexical sau neterminal).
- ▶  $\$<\text{nume\_membru}>\$$
- ▶  $\$<\text{nume\_membru}>|$
- ▶ Ex:
- ▶ rule : aaa {  $\$<\text{intval}>\$ = 3; \}$
- ▶ |
- ▶ bbb { fun(  $\$<\text{intval}>2, \$<\text{other}>0 \); \}$
- ▶ ;
- ▶ Obs: \$0 reprezinta valorile contextului stânga.



# Transmiterea valorii semantice a atomilor lexicali

- ▶ Membrii structurii de tip union declarate pot fi si variabile de tip **struct/class**, daca acest lucru este necesar.
- ▶ Ex:
- ▶ `%{`
- ▶ `// ...`
- ▶ `typedef struct interval { double lo, hi; } INTERVAL;`
- ▶ `%}`
- ▶ `...`
- ▶ `%union { int ival; double dval; INTERVAL vval; }`

# Analiza semantica

- ▶ Analiza semantica se face în paralel cu analiza sintactica.
- ▶ Ea depinde de limbajul tinta al interpretorului/compilerului.
- ▶ Daca la analiza semantica se constata erori, interpretarea, sau respectiv generarea codului trebuie întrerupta.
- ▶ Analiza sintactica si cea semantica trebuie însa se continue, pentru a gasi si afisa toate eventualele erori.

## Ex4 din arhiva - Analiza semantica

- ▶ %{
- ▶     #include "y.tab.h"
- ▶     int lineNo = 1;
- ▶     int colNo = 1;
- ▶ %}
  
- ▶ %%
- ▶ "+"             { colNo++; return TOK\_PLUS; }
- ▶ "-"             { colNo++; return TOK\_MINUS; }
- ▶ "\*"             { colNo++; return TOK\_MULTIPLY; }
- ▶ "/"             { colNo++; return TOK\_DIVIDE; }
- ▶ "("             { colNo++; return TOK\_LEFT; }
- ▶ ")"             { colNo++; return TOK\_RIGHT; }
- ▶ ";"             { colNo++; return ' '; }
- ▶ "="             { colNo++; return '='; }

## Ex4 din arhiva

- ▶ `0|[1-9][0-9]*` {
  - ▶ `yylloc.first_line = lineNo;`
  - ▶ `yylloc.first_column = colNo;`
  - ▶ `colNo+=strlen(yytext);`
  - ▶ `yylval.val = atoi(yytext);`
  - ▶ `return TOK_NUMBER;` }
- ▶ `"var"` { `colNo+=3; return TOK_DECLARE;` }
- ▶ `"print"` { `colNo+=5; return TOK_PRINT;` }
- ▶ `[a-zA-Z][a-zA-Z0-9]*` {
  - ▶ `yylloc.first_line = lineNo;`
  - ▶ `yylloc.first_column = colNo;`
  - ▶ `colNo+=strlen(yytext);`
  - ▶ `yylval.sir = new char[strlen(yytext)+1];`
  - ▶ `strcpy(yylval.sir,yytext);`
  - ▶ `return TOK_VARIABLE;` }
- ▶ `[ ]` { `colNo++;` }
- ▶ `.` { `colNo++; return TOK_ERROR;` }
- ▶ `\n` { `lineNo++;colNo=1;` }
- ▶ `%%`

## Ex4 din arhiva

- ▶ `%{`
- ▶ `#include <stdio.h>`
- ▶ `#include <string.h>`
- ▶ `int yylex();`
  
- ▶ `int yyerror(const char *msg);`
- ▶ `int EsteCorecta = 1;`
  
- ▶ `char msg[50];`

## Ex4 din arhiva

```
▶ class TVAR
▶ {
▶     char* nume;
▶     int valoare;
▶     TVAR* next;
▶     public:

▶     static TVAR* head;
▶     static TVAR* tail;

▶     TVAR(char* n, int v = -1);
▶     TVAR();
▶     int exists(char* n);
▶     void add(char* n, int v = -1);
▶     int getValue(char* n);
▶     void setValue(char* n, int v);
▶     };
```

## Ex4 din arhiva

- ▶ TVAR\* TVAR::head;
- ▶ TVAR\* TVAR::tail;
  
- ▶ TVAR::TVAR(char\* n, int v)
- ▶ {
- ▶ this->nume = new char[strlen(n)+1];
- ▶ strcpy(this->nume,n);
- ▶ this->valoare = v;
- ▶ this->next = NULL;
- ▶ }
  
- ▶ TVAR::TVAR()
- ▶ {
- ▶ TVAR::head = NULL;
- ▶ TVAR::tail = NULL;
- ▶ }

## Ex4 din arhiva

```
▶ int TVAR::exists(char* n)
▶ {
▶     TVAR* tmp = TVAR::head;
▶     while(tmp != NULL)
▶     {
▶         if(strcmp(tmp->nume,n) == 0)
▶             return 1;
▶         tmp = tmp->next;
▶     }
▶     return 0;
▶ }
```



## Ex4 din arhiva

```
▶ void TVAR::add(char* n, int v)
▶ {
▶     TVAR* elem = new TVAR(n, v);
▶     if(head == NULL)
▶     {
▶         TVAR::head = TVAR::tail = elem;
▶     }
▶     else
▶     {
▶         TVAR::tail->next = elem;
▶         TVAR::tail = elem;
▶     }
▶ }
```

## Ex4 din arhiva

```
▶ int TVAR::getValue(char* n)
▶ {
▶     TVAR* tmp = TVAR::head;
▶     while(tmp != NULL)
▶     {
▶         if(strcmp(tmp->nume,n) == 0)
▶         {
▶             return tmp->valoare;
▶             tmp = tmp->next;
▶         }
▶     }
▶     return -1;
▶ }
```

## Ex4 din arhiva

```
▶ void TVAR::setValue(char* n, int v)
▶ {
▶     TVAR* tmp = TVAR::head;
▶     while(tmp != NULL)
▶     {
▶         if(strcmp(tmp->nume,n) == 0)
▶         {
▶             tmp->valoare = v;
▶         }
▶         tmp = tmp->next;
▶     }
▶ }
▶ TVAR* ts = NULL; %}
```

## Ex4 din arhiva

- ▶ `%union { char* sir; int val; }`
- ▶ `%token TOK_PLUS TOK_MINUS TOK_MULTIPLY  
TOK_DIVIDE TOK_LEFT TOK_RIGHT TOK_DECLARE  
TOK_PRINT`
- ▶ `%token <val> TOK_NUMBER`
- ▶ `%token <sir> TOK_VARIABLE`
- ▶ `%type <val> E`
- ▶ `%start S`
- ▶ `%left TOK_PLUS TOK_MINUS`
- ▶ `%left TOK_MULTIPLY TOK_DIVIDE`
- ▶ `%locations`
- ▶ `%%`

## Ex4 din arhiva

```
▶ S :  
▶   |  
▶   | ';' S  
▶   |  
▶   error ';' S  
▶   { EsteCorecta = 0; }  
▶   ;
```

## Ex4 din arhiva

```
▶ I : TOK_VARIABLE '=' E
▶ {
▶   if(ts != NULL)
▶   {
▶     if(ts->exists($1) == 1)
▶     {
▶       ts->setValue($1, $3);
▶     }
▶     else
▶     {
▶       sprintf(msg,"%d:%d Eroare semantica: Variabila %s
este utilizata fara sa fi fost declarata!", @1.first_line,
@1.first_column, $1);
▶       yyerror(msg);
▶       YYERROR;
▶     }
▶   }
▶ }
```

## Ex4 din arhiva

```
▶ else
▶ {
▶     sprintf(msg,"%d:%d Eroare semantica: Variabila %s este
        utilizata fara sa fi fost declarata!", @1.first_line,
        @1.first_column, $1);
▶     yyerror(msg);
▶     YYERROR;
▶ }
▶ }
▶ |
```

## Ex4 din arhiva

```
▶ TOK_DECLARE TOK_VARIABLE
▶ {
▶   if(ts != NULL)
▶   {
▶     if(ts->exists($2) == 0)
▶     {
▶       ts->add($2);
▶     }
▶     else
▶     {
▶       sprintf(msg,"%d:%d Eroare semantica: Declaratii multiple pentru
variabila %s!", @1.first_line, @1.first_column, $2);
▶       yyerror(msg);
▶       YYERROR;
▶     }
▶   }
▶   else
▶   {
▶     ts = new TVAR();
▶     ts->add($2);
▶   }
▶ }
▶ |
```



## Ex4 din arhiva

```
▶ TOK_PRINT TOK_VARIABLE {
▶   if(ts != NULL)
▶   {
▶     if(ts->exists($2) == 1)
▶     {
▶       if(ts->getValue($2) == -1)
▶       {
▶         sprintf(msg,"%d:%d Eroare semantica: Variabila %s
este utilizata fara sa fi fost initializata!", @1.first_line,
@1.first_column, $2);
▶         yyerror(msg);
▶         YYERROR;
▶       }
▶     else
▶     {
▶       printf("%d",ts->getValue($2));
▶     }
▶   }
▶ }
```

## Ex4 din arhiva

```
▶ else
▶ {
▶     sprintf(msg,"%d:%d Eroare semantica: Variabila %s este utilizata
fara sa fi fost declarata!", @1.first_line, @1.first_column, $2);
▶     yyerror(msg);
▶     YYERROR;
▶ }
▶ }
▶ else
▶ {
▶     sprintf(msg,"%d:%d Eroare semantica: Variabila %s este utilizata
fara sa fi fost declarata!", @1.first_line, @1.first_column, $2);
▶     yyerror(msg);
▶     YYERROR;
▶ }
▶ }
▶ ;
```

## Ex4 din arhiva

```
▶ E : E TOK_PLUS E { $$ = $1 + $3; }
▶ |
▶ E TOK_MINUS E { $$ = $1 - $3; }
▶ |
▶ E TOK_MULTIPLY E { $$ = $1 * $3; }
▶ |
▶ E TOK_DIVIDE E
▶ {
▶   if($3 == 0)
▶   {
▶     sprintf(msg,"%d:%d Eroare semantica: Impartire la
zero!", @1.first_line, @1.first_column);
▶     yyerror(msg);
▶     YYERROR;
▶   }
▶   else { $$ = $1 / $3; }
▶ }
```

## Ex4 din arhiva

- ▶ TOK\_LEFT E TOK\_RIGHT
- ▶ {
- ▶ \$\$ = \$2;
- ▶ }
- ▶ |
- ▶ TOK\_NUMBER { \$\$ = \$1; }
- ▶ ;
- ▶ %%

## Ex4 din arhiva

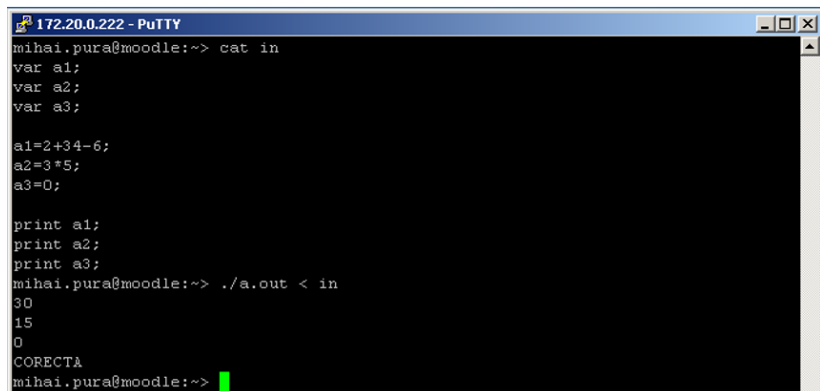
```
▶ int main()
▶ {
▶     yyparse();
▶     if(EsteCorecta == 1)

▶     {
▶         printf("CORECTA\n");
▶     }

▶     return 0;
▶ }

▶ int yyerror(const char *msg)
▶ {
▶     printf("Error: %s\n", msg);
▶     return 1;
▶ }
```

## Ex4 din arhiva



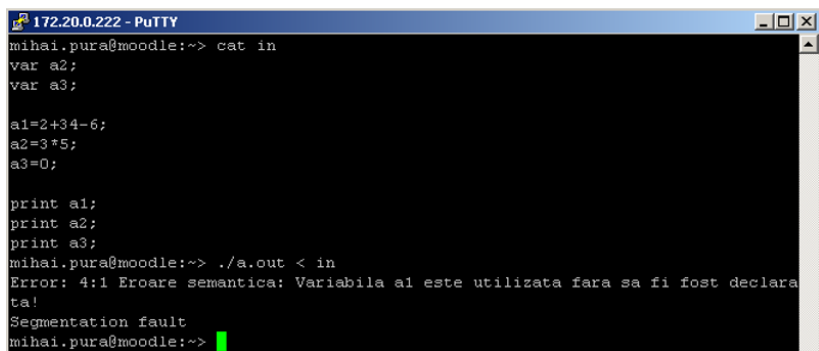
A screenshot of a PuTTY terminal window titled "172.20.0.222 - PuTTY". The terminal shows a user named "mihai.pura@moodle" at a prompt. The user enters "cat in", which displays the contents of a file named "in". The file contains a C program with three variables: "a1", "a2", and "a3". "a1" is assigned the value 2+34-6, "a2" is assigned 3\*5, and "a3" is assigned 0. The program then prints the values of "a1", "a2", and "a3" on separate lines. The user then enters "./a.out < in", which runs the compiled program, producing the same three output lines. The user then types "CORECTA", and the prompt returns. A green cursor is visible at the end of the last line.

```
mihai.pura@moodle:~> cat in
var a1;
var a2;
var a3;

a1=2+34-6;
a2=3*5;
a3=0;

print a1;
print a2;
print a3;
mihai.pura@moodle:~> ./a.out < in
30
15
0
CORECTA
mihai.pura@moodle:~> 
```

## Ex4 din arhiva



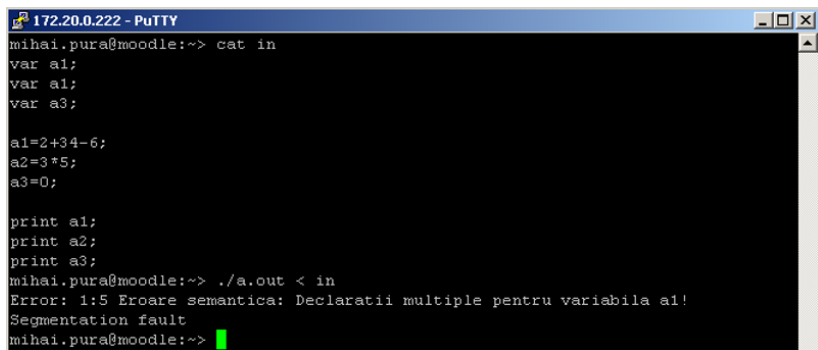
A screenshot of a PuTTY terminal window titled "172.20.0.222 - PuTTY". The terminal shows a user named "mihai.pura@moodle" in the home directory. They first run "cat in" to display the contents of a file named "in". The file contains the following code:

```
var a2;  
var a3;  
  
a1=2+34-6;  
a2=3*5;  
a3=0;  
  
print a1;  
print a2;  
print a3;
```

Next, the user runs "./a.out < in". This results in a compilation error message in Romanian: "Error: 4:1 Eroare semantica: Variabila a1 este utilizata fara sa fi fost declarata!". Below the error message, the text "Segmentation fault" is displayed. The terminal prompt returns to "mihai.pura@moodle:~>".

```
172.20.0.222 - PuTTY  
mihai.pura@moodle:~> cat in  
var a2;  
var a3;  
  
a1=2+34-6;  
a2=3*5;  
a3=0;  
  
print a1;  
print a2;  
print a3;  
mihai.pura@moodle:~> ./a.out < in  
Error: 4:1 Eroare semantica: Variabila a1 este utilizata fara sa fi fost declarata!  
Segmentation fault  
mihai.pura@moodle:~>
```

## Ex4 din arhiva



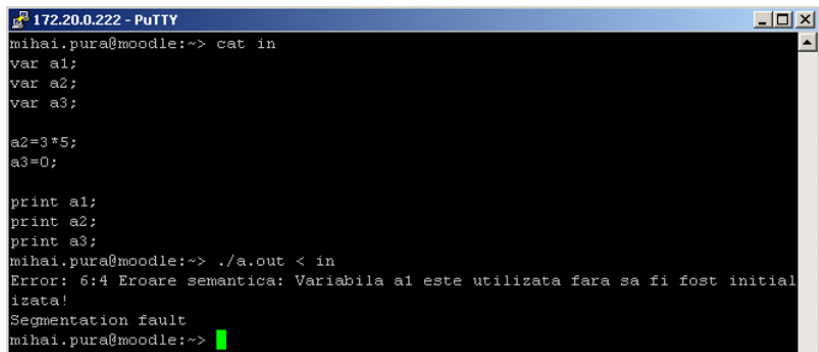
```
172.20.0.222 - PuTTY
mihai.pura@moodle:~> cat in
var a1;
var a1;
var a3;

a1=2+34-6;
a2=3*5;
a3=0;

print a1;
print a2;
print a3;
mihai.pura@moodle:~> ./a.out < in
Error: 1:5 Eroare semantica: Declaratii multiple pentru variabila a1!
Segmentation fault
mihai.pura@moodle:~> 
```



## Ex4 din arhiva



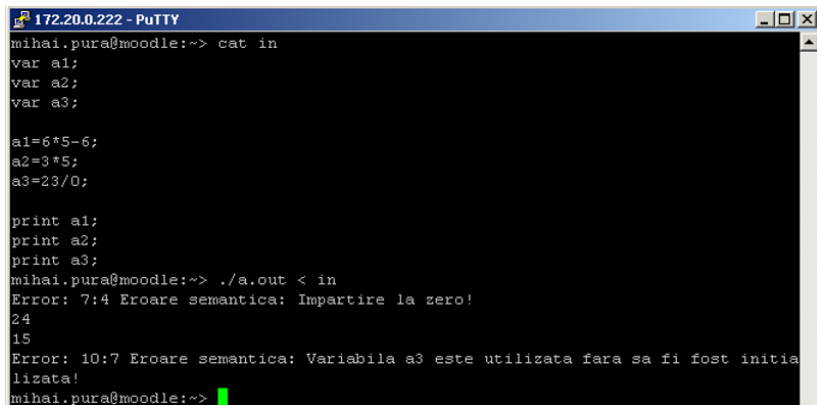
A screenshot of a PuTTY terminal window titled "172.20.0.222 - PuTTY". The terminal shows a user named "mihai.pura@moodle" at a prompt. They first run the command "cat in" to display the contents of a file named "in". The file contains a C program with three variables: "a1", "a2", and "a3". "a1" is declared but not assigned a value. "a2" is assigned the value 15 (3\*5), and "a3" is assigned the value 0. The program then prints the values of "a1", "a2", and "a3". The user then runs the command "./a.out < in" to execute the program. The program outputs "Error: 6:4 Eroare semantica: Variabila a1 este utilizata fara sa fi fost initializata!" followed by "Segmentation fault". The terminal ends with the user's prompt and a green cursor.

```
172.20.0.222 - PuTTY
mihai.pura@moodle:~> cat in
var a1;
var a2;
var a3;

a2=3*5;
a3=0;

print a1;
print a2;
print a3;
mihai.pura@moodle:~> ./a.out < in
Error: 6:4 Eroare semantica: Variabila a1 este utilizata fara sa fi fost initial
izata!
Segmentation fault
mihai.pura@moodle:~>
```

## Ex4 din arhiva



```
172.20.0.222 - PuTTY
mihai.pura@moodle:~> cat in
var a1;
var a2;
var a3;

a1=6*5-6;
a2=3*5;
a3=23/0;

print a1;
print a2;
print a3;
mihai.pura@moodle:~> ./a.out < in
Error: 7:4 Eroare semantica: Impartire la zero!
24
15
Error: 10:7 Eroare semantica: Variabila a3 este utilizata fara sa fi fost initia
lizata!
mihai.pura@moodle:~> █
```

# Arhitectura MIPS

- ▶ Procesorul este format din:
  - ▶ O unitate centrala de procesare (CPU) principala;
  - ▶ Doua co-procesoare:
    - ▶ Unul pentru operatiile cu numere reale;
    - ▶ Unul pentru gestiunea memoriei.
- ▶ Dimensiunea unui cuvânt este de 32 de biti (numarul de biti pe care un CPU îl poate procesa la un moment dat).
- ▶ Procesorul are un numar de 32 de registri, fiecare de dimensiunea unui cuvânt, unii cu destinatii speciale, altii pentru utilizarea generala.
- ▶ Co-procesorul dedicat operatiilor cu numere reale are si el 32 de registrii.
- ▶ Toate instructiunile sunt codate printr-un singur format, de lungimea unui cuvânt.
- ▶ Toate operatiile cu date sunt de tip registru la registru.
- ▶ Referintele la memorie sunt numai de tip load/store.

## Cei 32 de registrii MIPS

zero	holds constant 0
at	reserved for assembler
v0-v1	used to return results of functions
a0-a3	usually used to pass first 4 args to function call
t0-t7	general purpose (caller-saved)
s0-s7	general purpose (callee-saved)
t8-t9	general purpose (caller-saved)
k0-k1	reserved for OS
gp	global pointer to static data segment
sp	stack pointer
fp	frame pointer
ra	return address

# Modul de adresare

- ▶ Masina ofera un singur mod de adresare:
- ▶  $c(rx)$
- ▶ Accesarea locatiei de la offsetul  $c$  (pozitiv sau negativ), relativ la adresa de memorie stocata în registrul  $rx$ .
- ▶ Instructiunile load si store opereaza numai asupra datelor aliniate: o cantitate este aliniata numai daca adresa sa este un multiplu al dimensiunii sale în octeti.

# Formatul spatiului de adrese

- ▶ Spatiul de adrese al unui program pe o masina MIPS este format din mai multe segmente:
- ▶ În partea de jos este **segmentul de text** (contine instructiunile programului) – dimensiune fixa, setata la compilare, care nu se poate modifica.
- ▶ Urmeaza apoi **segmentul de date statice** (contine obiectele a caror adresa si dimensiune sunt cunoscute compilatorului si link-editorului în momentul compilarii) – dimensiune fixa, setata la compilare, care nu se poate modifica.
- ▶ În partea urmatoare se afla **segmentul de date dinamice** numit si **heap** (dimensiunea sa creste în functie de apelurile de alocare de memorie).
- ▶ În partea de sus a spatiului de adrese este **stiva programului**. Aceasta creste în jos, catre **heap**.

# SPIM

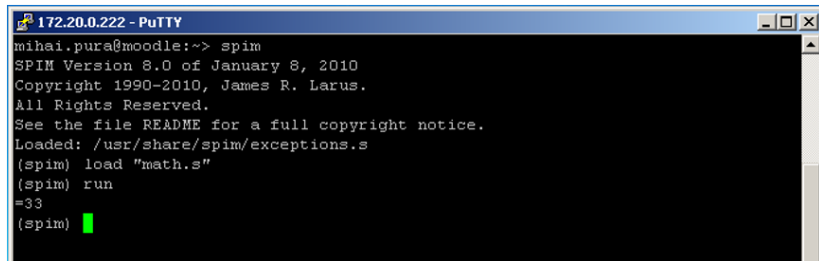
- ▶ SPIM este un simulator pentru MIPS32.
- ▶ Permite citirea si executia de programe scrise în limbaj de asamblare.
- ▶ Nu ruleaza executabile (programe binare compilate).

# Un exemplu de program MIPS

```
▶ .text
▶ .globl main
▶ main:
▶     li      $t1, 1
▶     li      $t2, 32
▶     addu    $t3,$t1,$t2
▶     la      $a0, egal
▶     li      $v0, 4
▶     syscall
▶     move    $a0, $t3
▶     li      $v0, 1
▶     syscall
▶     la      $a0, linie_noua
▶     li      $v0, 4
▶     syscall
▶     li      $v0, 10
▶     syscall
▶     .data
▶ egal:      .asciiz    "="
▶ linie_noua: .asciiz    "\n"
```



# Un exemplu de program MIPS



```
172.20.0.222 - PuTTY
mihai.pura@moodle:~> spim
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/share/spim/exceptions.s
(spim) load "math.s"
(spim) run
=33
(spim)
```

## Ex5 din arhiva - Generarea codului pentru limbajul expresiilor aritmetice

```
▶ %{
▶   #include "y.tab.h"
▶   extern int yylval;
▶   int lineNo = 1;
▶   int colNo = 1;
▶ }%
▶ %%
▶ "+"      { colNo++; return TOK_PLUS; }
▶ "-"      { colNo++; return TOK_MINUS; }
▶ "*"      { colNo++; return TOK_MULTIPLY; }
▶ "/"      { colNo++; return TOK_DIVIDE; }
▶ "("      { colNo++; return TOK_LEFT; }
▶ ")"      { colNo++; return TOK_RIGHT; }
▶ ";"      { colNo++; return ' '; }
▶ 0|[1-9][0-9]* { colNo+=strlen(yytext); yylloc.first_line = lineNo;
yylloc.first_column = colNo; yylval = atoi(yytext); return
TOK_NUMBER; } . { colNo++; return TOK_ERROR; }
▶ \n      { lineNo++;colNo=1; }
▶ %%
```

## Ex5 din arhiva

- ▶ `%{`
- ▶ `#include <stdio.h>`
- ▶ `FILE * yyies = NULL;`
- ▶ `int EsteCorecta = 1;`
- ▶ `char msg[50];`
- ▶ `int Prima = 0;`
- ▶ `%}`
- ▶ `%token TOK_PLUS TOK_MINUS TOK_MULTIPLY TOK_DIVIDE`  
`TOK_LEFT TOK_RIGHT`
- ▶ `%token TOK_NUMBER TOK_ERROR`
- ▶ `%start S`
- ▶ `%left TOK_PLUS TOK_MINUS`
- ▶ `%left TOK_MULTIPLY TOK_DIVIDE`

## Ex5 din arhiva

```
▶ %%  
▶ S :  
▶ |  
▶ E ';' S  
▶ {  
▶     fprintf(yyies, "\tla\t$a0, egal\n\tli\t$v0,  
4\n\tsyscall\n\tmove\t$a0, $t1\n\tli\t$v0, 1\n\tsyscall\n");  
▶     fprintf(yyies, "\tla\t$a0, linie_noua\n\tli\t$v0,  
4\n\tsyscall\n");  
▶     Prima = 0;  
▶ }  
▶ |  
▶ error ';' S  
▶ { EsteCorecta = 0; }  
▶ ;
```

## Ex5 din arhiva

```
▶ E : E TOK_PLUS E
▶     {
▶         fprintf(yyies, "\\tadd\\t$t1,$t1,$t2\\n");
▶         Prima = 1;
▶     }
▶ |
▶ E TOK_MINUS E
▶     {
▶         fprintf(yyies, "\\tsub\\t$t1,$t1,$t2\\n");
▶         Prima = 1;
▶     }
▶ |
```

## Ex5 din arhiva

```
▶ E TOK_MULTIPLY E
▶ {
▶     if(Prima == 2)
▶     {
▶         fprintf(yyies, "\\tmult\\t$t1,$t2\\n");
▶         fprintf(yyies, "\\tmflo\\t$t1\\n");
▶     }
▶     else
▶     {
▶         fprintf(yyies, "\\tmult\\t$t2,$t3\\n");
▶         fprintf(yyies, "\\tmflo\\t$t2\\n");
▶     }
▶     Prima = 1;
▶ }
```

```
▶ |
```

## Ex5 din arhiva

```
▶ E TOK_DIVIDE E
▶ {
▶     if($3 == 0)
▶     {
▶         sprintf(msg,"%d:%d Eroare semantica: Impartire la zero!", @1.first_line,
@1.first_column);
▶         yyerror(msg);
▶         YYERROR;
▶     }
▶     else
▶     {
▶         if(Prima == 2)
▶         {
▶             fprintf(yyies, "\\tdiv\\t$t1,$t2\\n");
▶             fprintf(yyies, "\\tmflo\\t$t1\\n");
▶         }
▶         else
▶         {
▶             fprintf(yyies, "\\tdiv\\t$t2,$t3\\n");
▶             fprintf(yyies, "\\tmflo\\t$t2\\n");
▶         }
▶         Prima = 1;
▶     }
▶ }
▶ |
```

## Ex5 din arhiva

```
▶ TOK_NUMBER
▶ {
▶     if(Prima == 0)
▶     {
▶         fprintf(yyies, "\\tli\\t$t1, %d\\n", $1);
▶         Prima = 1;
▶     }
▶     else if(Prima == 1)
▶     {
▶         fprintf(yyies, "\\tli\\t$t2, %d\\n", $1);
▶         Prima = 2;
▶     }
▶     else
▶     {
▶         fprintf(yyies, "\\tli\\t$t3, %d\\n", $1);
▶         Prima = 3;
▶     }
▶ }
▶ ;
▶ %%%
```



## Ex5 din arhiva

```
▶ int main()
▶ {
▶     yyies = fopen("math.s", "w");
▶     fprintf(yyies, "\t.t.text\n\t.t.globl main\nmain:\n");

▶     yyparse();

▶     fprintf(yyies, "\tli\t$v0, 10\n\tsyscall\n\t.data\nnegal:\t\t.ascii z\n\n\tline_noua:\t\t.ascii z\t\t\t\n\n");
▶     fclose(yyies);

▶     if(EsteCorecta == 1)
▶     {
▶         printf("CORECTA\n");
▶     }

▶     return 0;
▶ }
```

## Ex5 din arhiva

```
▶ int yyerror(const char *msg)
▶ {
▶     printf("Error: %s\n", msg);
▶     return 1;
▶ }
```

## Ex5 din arhiva

► Ex:  $2*3+4+2*2-1-5+2+3*2-4/2+3$ ;

► .text

► .globl main

► main:

► li \$t1, 2

► li \$t2, 3

► mult \$t1,\$t2

► mflo \$t1

► li \$t2, 4

► add \$t1,\$t1,\$t2

► li \$t2, 2

► li \$t3, 2

► mult \$t2,\$t3

► mflo \$t2

## Ex5 din arhiva

► Ex:  $2*3+4+2*2-1-5+2+3*2-4/2+3$ ;

► add \$t1,\$t1,\$t2

► li \$t2, 1

► sub \$t1,\$t1,\$t2

► li \$t2, 5

► sub \$t1,\$t1,\$t2

► li \$t2, 2

► add \$t1,\$t1,\$t2

► li \$t2, 3

► li \$t3, 2

► mult \$t2,\$t3

► mflo \$t2

► add \$t1,\$t1,\$t2

## Ex5 din arhiva

▶ Ex:  $2*3+4+2*2-1-5+2+3*2-4/2+3;$

▶ li            \$t2, 4

▶ li            \$t3, 2

▶ div          \$t2,\$t3

▶ mflo        \$t2

▶ sub          \$t1,\$t1,\$t2

▶ li            \$t2, 3

▶ add          \$t1,\$t1,\$t2

## Ex5 din arhiva

- ▶ `la`            `$a0, egal`
- ▶ `li`            `$v0, 4`
- ▶ `syscall`
  
- ▶ `move`        `$a0, $t1`
- ▶ `li`            `$v0, 1`
- ▶ `syscall`
  
- ▶ `la`            `$a0, linie_noua`
- ▶ `li`            `$v0, 4`
- ▶ `syscall`
  
- ▶ `li`            `$v0, 10`
- ▶ `syscall`
  
- ▶ `.data`
- ▶ `egal:`        `.asciiz "="`
- ▶ `linie_noua:` `.asciiz "\n"`

## Ex5 din arhiva

 172.20.0.222 - PuTTY

```
mihai.pura@moodle:~> cat in
2*3+4+2*2-1-5+2+3*2-4/2+3;
mihai.pura@moodle:~> spim
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/share/spim/exceptions.s
(spim) load "math.s"
(spim) run
=17
(spim) █
```

## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	$\epsilon$
Stiva	
$\epsilon$	





## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	NR
Stiva	
$\epsilon$	



# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	*
Stiva	
NR	



## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	*
Stiva	
E	

li	\$t1, 2
----	---------

# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	2*3+4+2*2-1-5+2+3*2-4/2+3
Look ahead	NR
Stiva	
*	
E	

li    \$t1, 2

# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
NR	
*	
E	

li    \$t1, 2

## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
E	
*	
E	

li	\$t2, 3
li	\$t1, 2

## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
E	

```
mult  $t1,$t2  
mflo  $t1
```

# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	NR
Stiva	
+	
E	

mult \$t1,\$t2
mflo \$t1



## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
NR	
+	
E	

mult \$t1,\$t2
mflo \$t1

## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
E	
+	
E	

li	\$t2, 4
mult	\$t1,\$t2
mflo	\$t1

## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
E	

add    \$t1,\$t1,\$t2
-----------------------

# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	2*3+4+2*2-1-5+2+3*2-4/2+3
Look ahead	+
Stiva	
+	
E	

add \$t1,\$t1,\$t2

# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	2*3+4+2*2-1-5+2+3*2-4/2+3
Look ahead	+
Stiva	
E	
+	
E	

li	\$t2, 2
add	\$t1,\$t1,\$t2

# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
*	
E	
+	
E	

li \$t2, 2
add \$t1,\$t1,\$t2

# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
NR	
*	
E	
+	
E	

li    \$t2, 2
add    \$t1,\$t1,\$t2

# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
E	
*	
E	
+	
E	

li	\$t3, 2
li	\$t2, 2
add	\$t1,\$t1,\$t2



# Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
E	
+	
E	

mult	\$t2,\$t3
mflo	\$t2
add	\$t1,\$t1,\$t2

## Generarea codului pentru limbajul expresiilor aritmetice

Intrare	$2*3+4+2*2-1-5+2+3*2-4/2+3$
Look ahead	+
Stiva	
E	

add    \$t1,\$t1,\$t2
-----------------------

s.a.m.d.

# Bibliografie

- ▶ Bert Hubert, **Lex and YACC primer/HOWTO**  
<http://tldp.org/HOWTO/Lex-YACC-HOWTO.html>
- ▶ Tom Niemann, **Lex & Yacc**  
<http://epaperpress.com/lexandyacc/>
- ▶ Anthony A. Aaby, **Compiler Construction using Flex and Bison**  
<http://foja.dcs.fmph.uniba.sk/kompilatory/docs/compiler.pdf>

# Bibliografie

- ▶ **Programmed Introduction to MIPS Assembly Language**

[http://programmedlessons.org/  
AssemblyTutorial/index.html](http://programmedlessons.org/AssemblyTutorial/index.html)

- ▶ **MIPS Instruction Reference**

[http://www.mrc.uidaho.edu/mrc/people/jff/  
digital/MIPSir.html](http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html)