Formal Languages and Compilers

Lab11

-Bison-

Bison

- .y extension
- Grammars (set of formation rules) for Bison are described using a variant of Backus Naur Form (BNF) (formal, mathematical way to describe context-free grammars).

Example of BNF:

```
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

“::=” means **“is defined as”**
“|” means **“or”**

BNF vs Extended BNF (EBNF)

- Decimal numbers in BNF

```
<expr> ::= '-' <num> | <num>

<num> ::= <digits>
| <digits> '.' <digits>

<digits> ::= <digit>
| <digit> <digits>

<digit> ::= '0' | '1' | '2' |
'3' |
'4' | '5' | '6' | '7' | '8' |
'9'
```

- Decimal numbers in EBNF

```
<expr> := '-'? <digit>+ ('.'
<digit>+)?

<digit> := '0' | '1' | '2' |
'3' | '4' |
'5' | '6' | '7' | '8' | '9'
```

“?” means **“0 or 1 occurrences”** (“[...]” can also be used)

“+” means **“1 or more occurrences”**

Grammar example

```
E -> E + E
E -> E * E
E -> id
```

Lexeme	Token type
7	Number
*	Product
3	Number
+	Addition
5	Number

- E: expression (nonterminal, appear on right-hand side)
- Id: identifier (terminal-tokens returned by flex, appear on the left-hand side)
- An expression can be the sum of two expressions, the product of two expressions or an identifier
- E.g. 7*3+5
- Start with the left-side of the expression:
  - 7 is an id -> an id is an expression
  - Match one of the rules based on the next token because 7 is an expression
  - The product expression is matched because the * is the next token
  - Move to the next token which is 3. 3 is an id->an id is an expression
  - Repeat

Bison file structure

- %{ C declarations %} or %code
- %% Bison declarations
- %% Grammar rules

$$-holds the result of the rule

$n-holds the value of the n^th^ term from the right side

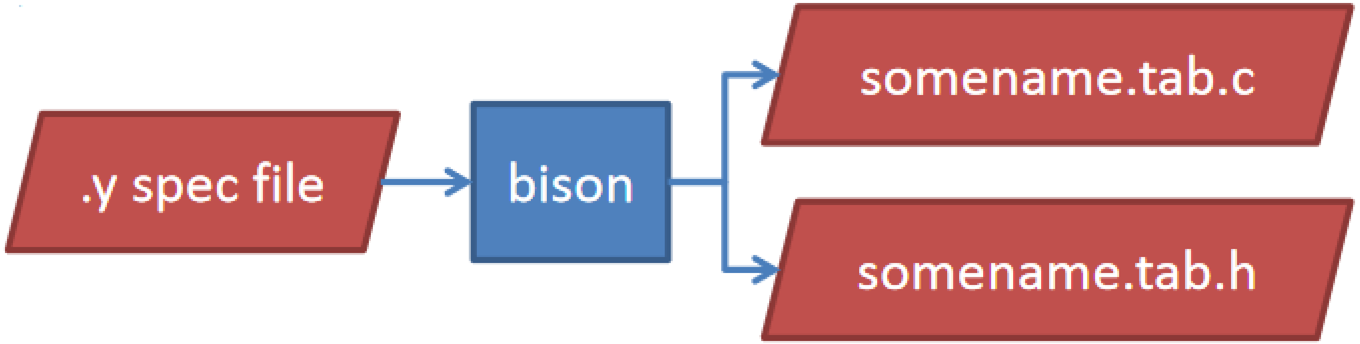
- %% C code

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}
%token INTEGER

%%
program:
        program expr '\n'     { printf("%d\n", $2);}
        |
        ;
expr:
        INTEGER                 { $$ = $1; }
        | expr '+' expr         { $$ = $1 + $3; }
        | expr '-' expr         { $$ = $1 - $3; }
        ;
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

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

```
------------------------------

Part to be embedded into the *.c

------------------------------
%Definition Section
(token declarations)
------------------------------
Production rules section:
define how to "understand" the
input language, and what actions
to take for each "sentence".



------------------------------
< C auxiliary subroutines >
Any user code. For example,
a main function to call the
parser function yyparse()



------------------------------
```

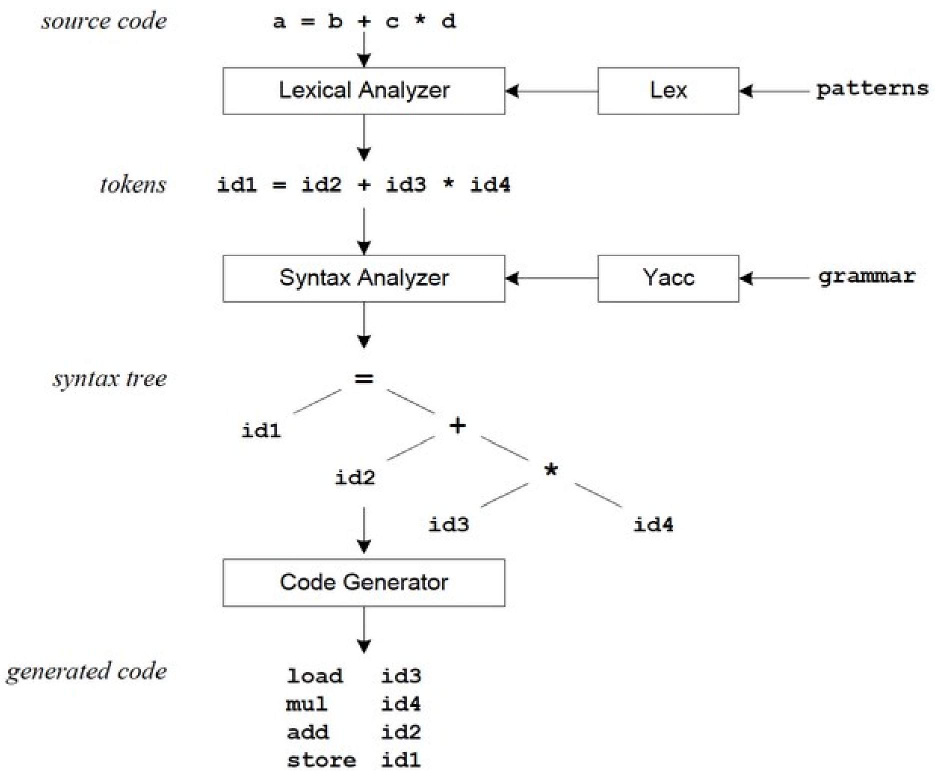
Bison

- The .y file is used to generate a parser in file .c and an include file .h (used with flex)
- The parser analyzes a sequence of tokens and attempts to determine the grammatical structure in relation to a particular grammar.

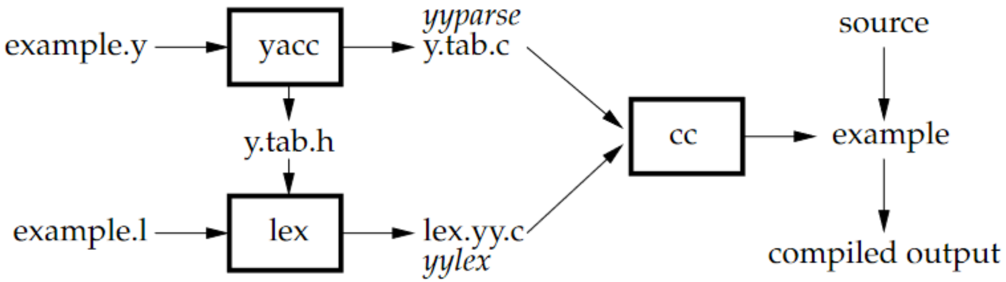


Flex & Bison

- Compilation sequence: ->



- Building a compiler:



Example – simple calculator

- .l file (Flex file)

```
%option noyywrap

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

#define YY_DECL int yylex()

#include "example.tab.h"

%}

%%

[ \t]	;
[0-9]+		{yylval.ival = atoi(yytext); return T_INT;}
\n	{return T_NEWLINE;}
"+"	{return T_PLUS;}
"-"	{return T_MINUS;}

%%
```

Example – simple calculator

- .y file (Bison file)

```
//Definitions Section
%{
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*Lex  includes  this  file  and  utilizes  the  definitions  for  token  values.
To  obtain  tokens  yacc  calls  yylex.  Function  yylex  has  a  return  type
of  int  that  returns  a  token */

extern int yylex();
extern int yyparse();
extern FILE* yyin;

/*yyerror is used to generate an error in case something is wrong.
It's also called in main function */
void yyerror(const char* s);
%}

//Rules Section
%union {
        int ival;
        float fval;
}
//This is where YACC generates a parser in file y.tab.c and an include file y.tab.h
%token<ival> T_INT
%token T_PLUS T_MINUS
%token T_NEWLINE

%type<ival> expression


%start calculation
```

```
%%

calculation:
      | calculation line
;

line: T_NEWLINE
    | expression T_NEWLINE { printf("\tResult: %i\n", $1); }
;

/*The  rules  section  resembles  the  BNF  grammar  discussed  earlier.
The  left-hand  side  of  a  production,  or  nonterminal,
is entered left-justified and followed by a colon.
This is followed by the right-hand side of the production.
Actions associated with a rule are entered in braces.*/

;

expression: T_INT                 { $$ = $1; }
      | expression T_PLUS expression     { $$ = $1 + $3; }
      | expression T_MINUS expression    { $$ = $1 - $3; }

;

%%

int main() {
    yyin = stdin;

    do {
        yyparse();
    } while(!feof(yyin));

    return 0;
}

void yyerror(const char* s) {
    fprintf(stderr, "Parse error: %s\n", s);
    exit(1);
}
```

Running FLEX & Bison

Running Bison on Linux

- In order to run Bison on Linux you need to use Terminal.
- Step 1: In Terminal, you need to change the directory where the code is, using the command ‘cd’ (e.g. cd /Desktop/Ex/ExB)
- Step 2: In order to run Bison, FLEX must be run first. Write flex <filename> (e.g. flex example.l)
- Step 3: To run Bison, enter the command bison –d <filename>(e.g. bison –d example.y)
- Step 4: The above commands will create a file name lex.yy.c and name.tab.c. These files will be compiled by using the commands cc y.tab.c lex.yy.c -lfl.
- Step 5: To run the program, the command ./a.out must be introduced.

```
alle@alle-VirtualBox:~/Desktop/Ex/ExB$ flex example.l
alle@alle-VirtualBox:~/Desktop/Ex/ExB$ bison -d example.y
alle@alle-VirtualBox:~/Desktop/Ex/ExB$ cc example.tab.c lex.yy.c -lfl
alle@alle-VirtualBox:~/Desktop/Ex/ExB$ ./a.out
10-7+5
        Result: 8
```

Running FLEX & Bison

Running Bison on Windows

- In order to run Bison on Windows you need to use Command Prompt.
- Step 1: In CMD, you need to change the directory where the code is, using the command ‘cd’ (e.g. cd D:\Facultate\Predat\FormalLanguagesAndCompilers\Lab\Ex\Ex4L3)
- Step 2: In order to run Bison, FLEX must be run first. Write flex <filename> (e.g. flex example.l)
- Step 3: To run Bison, enter the command bison –d <filename>(e.g. bison –d example.y)
- Step 4: The above commands will create a file name lex.yy.c and name.tab.c. These files will be compiled by using the commands gcc lex.yy.c name.tab.c.
- Step 5: To run the program, the command a.exe must be introduced.

```
D:\Facultate\Predat\FormalLanguagesAndCompilers\Lab\Ex\Ex4L3>flex example.l

D:\Facultate\Predat\FormalLanguagesAndCompilers\Lab\Ex\Ex4L3>bison -d example.y

D:\Facultate\Predat\FormalLanguagesAndCompilers\Lab\Ex\Ex4L3>gcc lex.yy.c example.tab.c

D:\Facultate\Predat\FormalLanguagesAndCompilers\Lab\Ex\Ex4L3>a.exe
7+10-6
        Result: 11
```

Exercise

- Add multiplication and division to the exercise done in lab.

Homework

- Add m^n to the exercise done in lab.
- Add mathematical parenthesis “(“, “)” to the exercise done in lab.