How to Work with Flex & Bison

Linguagens de programação

Ana Madureira amd@isep.ipp.pt António Silva ass@isep.ipp.pt
José Marinhojsm@isep.ipp.pt José Tavares jrt@isep.ipp.pt
Paulo Ferreira pdf@isep.ipp.pt

Instituto Superior de Engenharia do Porto

April 2021

Required Software

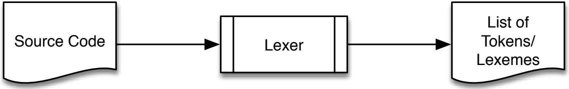
- flex and bison
- Available on OSX or Linux (32 or 64 bit)
- Linux Virtual Machines (Virtual Box or WSL)
- ssh servers at DEI
- Check if they are installed on the machine that you are using:

```
flex -V ; bison -V
```

What is a lexer?

- Grabs a file and spits out tokens and lexemes
- The first phase (or pass) of a compiler
- It may leave you confused

What does a lexer do?



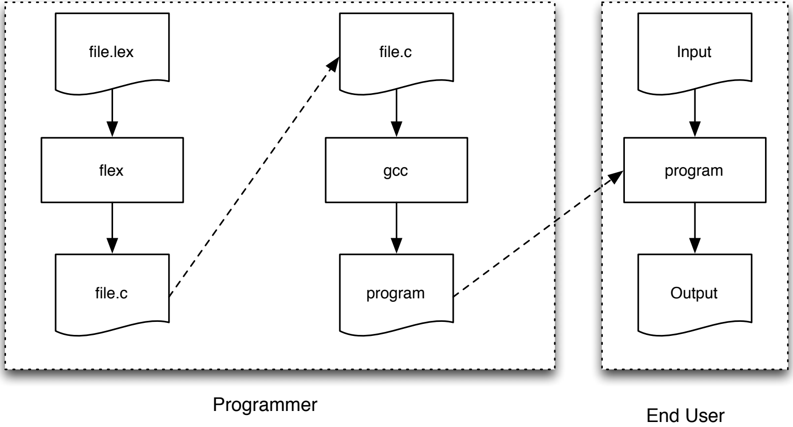
varx=varx+1;
y=0.65;

IDENT, "varx"
EQUAL
IDENT, "varx"
PLUS
INTEGER, 1
SEMICOLON
IDENT, "y"
EQUAL
FLOAT, 0.65
SEMICOLON

Confusing stuff

- Flex is not a lexer, is a tool for writing lexers
- Uses regular expressions
- The produced lexer is rule driven, and has not a sequential execution pattern
- It uses and produces C code
- It mixes C code with a very specific syntax

The flow of execution



Typical command lines

- `flex -t file.lex > file.c`
- `gcc -Wall file.c -lfl -o fich` (Linux)
- `gcc -Wall file.c -ll -o fich` (OSX or BSD)
- `./fich <textfile.txt`

Tip 1: use a makefile

Tip 2: use `.lex` for the file *extension*, not `.flex`

Three sections on a lex file

Separated by %% – Definitions and options – Pairs of patterns/actions – C code

```
%{
 int aCount=0;
%}
%option nounput
%option noinput
%%
a  aCount++;
%%
             int main()
             {
              yylex( ) ;
              printf("Number of 'a's: %d\n" , aCount);
              return 0;
              }
```

Section 1 – Definitions

- Options for the generated lexer
- Macros to be used on the patterns of the following section
- Pieces of required C code (includes and others)

```
%{
 int aCount=0;
%}
%option nounput
%option noinput
%%
```

Section 2 –Pairs of patterns/actions

- Regular expressions are written using the flex rules (don't look on the Internet regexps for other languages/systems)
- The action is written in C
- One C instruction ending with `;` or more inside `{ }`
- Must start on the same line!

```
%%
a   aCount++;
%%
```

Section 3 – C code

- One can place the `main` code function here, if we are not using other modules.
- The `yylex()` function is the generated lexer function

```
%%
             int main()
             {
              yylex( ) ;
              printf("Number of 'a's: %d\n" , aCount);
              return 0;
              }
```

demo01

This input file:

```
abcdaa
aaa
zw
```

Gives this as output:

```
bcd

zw
Number of 'a's: 6
```

Conclusion: Text that is not matched, is copied to the output, text that matches “disappears”

demo02

A small change:

```
%{
 int aCount=0;
%}
%option nounput
%option noinput
%%
a  { aCount++; printf("->%s<-",yytext); }
%%
             int main()
             {
              yylex( ) ;
              printf("Number of ’a’s: %d\n" , aCount);
              return 0;
              }
```

demo02

This input file:

```
abcdaa
aaa
zw
```

Gives this as output:

```
->a<-bcd->a<-->a<-
->a<-->a<-->a<-
zw
Number of 'a's: 6
```

Conclusion: we can use `yytext` to get the matched text
Warning: `yytext` is always a C *string* – array of chars!

demo03 – Eating other chars

```
%{
 int aCount=0;
%}
%option nounput
%option noinput
%%
a  { aCount++; printf("->%s<-",yytext); }
.  ;     /* any other char will  disappear */
%%
             int main()
             {
              yylex( ) ;
              printf("Number of ’a’s: %d\n" , aCount);
              return 0;
              }
```

demo03 – Eating other chars

This input file:

```
abcdaa
aaa
zw
```

Gives this as output:

```
->a<-->a<-->a<-
->a<-->a<-->a<-

Number of ’a’s: 6
```

demo04 – Eating also newline

```
%{
 int aCount=0;
%}
%option nounput
%option noinput
%%
a  { aCount++; printf("->%s<-",yytext); }
.|\n  ;     /* any other char including newline will  disappear */
%%
             int main()
             {
              yylex( ) ;
              printf("Number of ’a’s: %d\n" , aCount);
              return 0;
              }
```

demo04 – Eating also newline

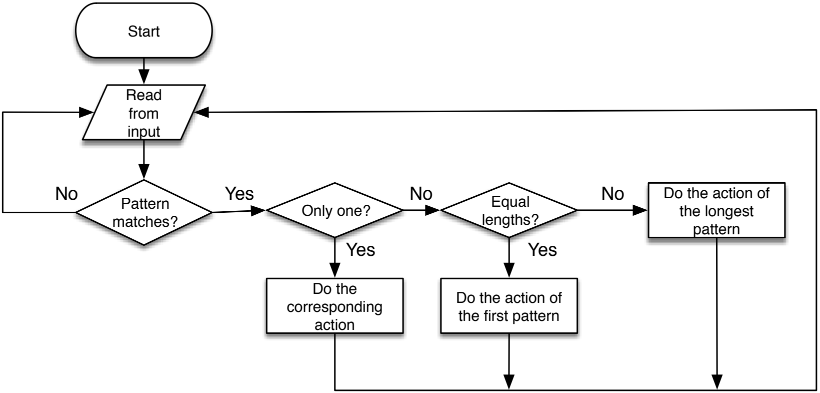
This input file:

```
abcdaa
aaa
zw
```

Gives this as output:

```
->a<-->a<-->a<-->a<-->a<-->a<-Number of 'a's: 6
```

How (the program generated by) Flex behaves


Start
Read from input
Pattern matches?
No
Yes
Only one?
No
Yes
Do the corresponding action
Equal lengths?
No
Yes
Do the action of the first pattern
Do the action of the longest pattern

demo05 – Macros and longest match

```
PAT2 aa
PAT3 aaa
PAT4 aaaa
%option nounput
%option noinput
%%
{PAT2}   printf("Pat2");
{PAT3}   printf("Pat3");
{PAT4}   printf("Pat4");
%%
             int main()
             {
              yylex( ) ;
              return 0;
              }
```

demo05 – Macros and longest match

This input file:

```
aa
aaa
aaaa
aaaaa
aaaaaa
```

Gives this as output:

```
Pat2
Pat3
Pat4
Pat4a
Pat4Pat2
```

demo06 – Macros and first match

```
PAT1 aaa
PAT2 [a-z][a-z][a-z]
%option nounput
%option noinput
%%
{PAT2}   printf("Pat2");
{PAT1}   printf("Pat1");
%%
             int main()
             {
              yylex( ) ;
              return 0;
              }
```

demo06 – Macros and first match

This input file:

```
bbb
aaa
aaa
```

Gives this as output:

```
Pat2
Pat2
Pat2
```

demo07 – Macros and first match

```
PAT1 aaa
PAT2 [a-z][a-z][a-z]
%option nounput
%option noinput
%%
{PAT1}   printf("Pat1");
{PAT2}   printf("Pat2");
%%
             int main()
             {
              yylex( ) ;
              return 0;
              }
```

demo07 – Macros and first match

This input file:

```
bbb
aaa
aaa
```

Gives this as output:

```
Pat2
Pat1
Pat1
```

Things to watch for

- warning: ‘yyunput’ defined but not used — use %option nounput on the first section
- warning: ‘input’ defined but not used — use %option noinput on the first section
- Comments in Flex are C comments /*...*/ not C++ comments (//)!
- Comments in the second section cannot start on the first column, or they will be considered patterns
- Spaces can be written between — ” ” — or between square brackets — [ ]
- a|b|c is ok, a | b | c is not ok!

What is (not) Flex?

- Not the one from Macromedia/Adobe!
- Not two very old Operating Systems (FLEX – 1976 and FlexOS – 1986)!

The Bison History

- In the beginning there was yacc – Yet Another Compiler Compiler
- Bison is the name of the GNU version of yacc
- The purpose is to help/automatise the writing of compilers/interpreters

Why is it so confusing?

- It is a program that writes other programs. Writing programs is complicated!

Why is it so confusing?

- It is a program that writes other programs. Writing programs is complicated!
- It is used in association with `flex`. They must cooperate in a correct way.

Why is it so confusing?

- It is a program that writes other programs. Writing programs is complicated!
- It is used in association with `flex`. They must cooperate in a correct way.
- It uses grammars. To use it one should be able to write correct grammars.

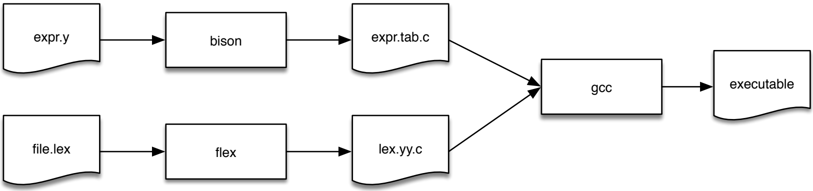
Why is it so confusing?

- It is a program that writes other programs. Writing programs is complicated!
- It is used in association with `flex`. They must cooperate in a correct way.
- It uses grammars. To use it one should be able to write correct grammars.
- Is uses the C language. Please learn C before using Bison.

Why is it so confusing?

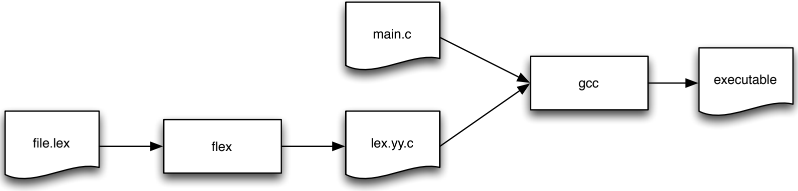
- It is a program that writes other programs. Writing programs is complicated!
- It is used in association with `flex`. They must cooperate in a correct way.
- It uses grammars. To use it one should be able to write correct grammars.
- Is uses the C language. Please learn C before using Bison.
- The alternative is writing yourself all the required code. That means `bison` is very simple!

The flow of execution



- We must use `flex` and `bison` and `gcc`
- Everything must fit together!
- Lets take it slowly

Case 1



- Using flex and gcc
- Shows how to get the data from flex (like bison does)

Communication with `flex`

- The required **Token types** must be defined on the two sides

  `main` should call `yylex()` until the end of file is found

  The return value of `yylex()` is the type of token found

  The value that corresponds to the found token can be read from the `yylval` global variable

Example: demo01

Having this text file (`demo01/sample.txt`), we want to split it in numbers and operators (just plus for now):

```
23+4+55
```

We start by making a definitions file (`demo01/defs.h`), with all the token types:

```
/*  demo01/defs.h  */

typedef enum{ZERO,INTEGER,PLUS} token;

```

The `ZERO` token is defined to reserve the zero value that signalizes the end of file.

Contents of the lex file

```
%{    /* demo01/file.lex  */
     #include "defs.h"
     extern int yylval;
%}

%option noinput
%option nounput
%%

[0-9]+         { yylval = atoi(yytext); return INTEGER;}
\+             return PLUS;
<<EOF>>        return 0;   /* file has ended -- not needed */
.              { /* ignore everything else */ }
%%
  /* no code here */
```

Definitions

- Include of the token definitions
- The variable `yylval` is on `main.c`
- Options to silence some warnings

```
%{    /* demo01/file.lex  */
     #include "defs.h"
     extern int yylval;
%}

%option noinput
%option nounput
%%
```

Pairs of patterns/actions

- Finding a token, `yylex()` should return it's type
- If the value is important, it should be placed on `yylval`
- On the the end of file `yylex()` should return zero
- The code section has nothing

```
%%

[0-9]+         { yylval = atoi(yytext); return INTEGER;}
\+             return PLUS;
<<EOF>>        return 0;   /* file has ended -- not needed */
.              { /* ignore everything else */ }
%%
  /* no code here */
```

C code on main.c

```
#include <stdio.h>
#include "defs.h"    /* demo01/main.c  */
int yylval;  int yylex();
int main()
  {
    token tok;
    do {
             tok=yylex();  /*  call flex  */
             switch(tok)
             {
                case INTEGER:
                  printf("received the int: %d\n",yylval); break;
                case PLUS:
                 printf("received the plus token\n");
                default:  break;
             }
        }
        while (tok!=0);
        return 0;
  }
```

C code on `main.c`

Explaining:

- The token definitions are included
- The `yylval` variable is defined here (an integer for now).
- The `yylex()` function is defined on the flex generated code
- The main function just calls `yylex()` and checks what type of token it has received

demo01

This input file:

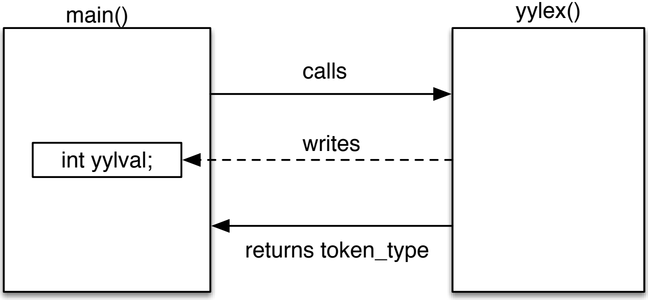
```
23+4+55
```

Gives this as output:

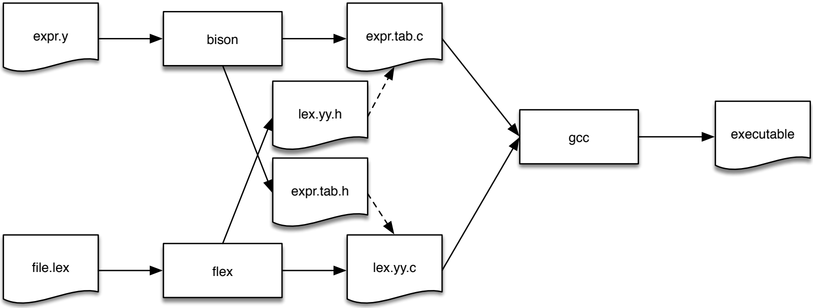
```
received the int: 23
received the plus token
received the int: 4
received the plus token
received the int: 55
```

What can be said?

- There are declarations that need to be cross referenced
- Besides the return of the `yylex()` function, the `yylval` variable is important.



With Bison



Each of the programs generates an header file for the C code generated by the other

Now with Bison – demo02

A small change:

```
%{    /* demo02/file.lex  */
   #include "expr.tab.h"
%}

%option noinput
%option nounput
%option header-file="lex.yy.h"
%%

[0-9]+         { yylval = atoi(yytext); return INTEGER;}
\+             return PLUS;
.              { /* ignore everything else */ }
%%
  /* no code here */
```

Definitions

- Include of the bison generated header file
- Options to silence some C compiler warnings (function declared but not used)
- Option to generate an header file

```
%{    /* demo02/file.lex  */
   #include "expr.tab.h"
%}

%option noinput
%option nounput
%option header-file="lex.yy.h"
%%
```

Pairs of patterns/actions

- Finding a token, `yylex()` should return it's type
- If the value is important, it should be placed on `yylval`
- On the the end of file `yylex()` should return zero
- The code section has nothing

```
%%

[0-9]+         { yylval = atoi(yytext); return INTEGER;}
\+             return PLUS;
.              { /* ignore everything else */ }
%%
  /* no code here */
```

Now the Bison file – demo02

```
%{  /* demo02/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%token INTEGER PLUS
%%
   S   : S EXPR
         |
         ;
   EXPR :  INTEGER  { printf("Found an int: %d\n", $1) ; }
           |  PLUS  { printf("Found a plus\n");}
        ;
%%
int main() {
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

Now the Bison file – Part 1: Definitions

- Include of the `flex` generated header file
- Declaration of the `yyerror()` function, that will be called in case of error
- Definition of the tokens (it is automatically placed in the `expr.tab.h` file)

```
%{  /* demo02/expr.y   */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%token INTEGER PLUS
%%
```

Now the Bison file – Part 2: Grammar

- Grammar definition
- This a very silly grammar in order to be simple!
- The first rule is (by default) the starting rule
- We can have zero or more “expressions”
- Each expression is an integer or a plus
- $1 is the first item of the triggered rule

```
%%
   S    : S EXPR
         |
         ;
   EXPR :  INTEGER  { printf("Found an int: %d\n", $1) ; }
           |  PLUS  { printf("Found a plus\n");}
        ;
%%
```

Now the Bison file – Part 3: C code

- `main()` and `yyerror()` functions
- Simple and easy to understand
- `yyparse()` is the function that Bison generates

```
%%
int main() {
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

demo02

This input file:

```
23+4+55
```

Gives this as output:

```
Found an int: 23
Found a plus
Found an int: 4
Found a plus
Found an int: 55
```

Type of yyval ?

- yylval is usually an integer
- We can redefine the type of yylval using:

  `#define YYSTYPE ...`

  See demo03
- Definitions that should be seen also from flex should be placed inside a `%code requires { ... }` block

demo03 – Other types

```
%{  /* demo03/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%code requires { #define YYSTYPE float
}
%token FLOAT PLUS
%%
   S    : S EXPR
         |
         ;
   EXPR :  FLOAT  { printf("Found a float: %f\n", $1) ; }
           |  PLUS  { printf("Found a plus\n");}
        ;
%%
int main() {
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

demo03 – Other types

```
%{    /* demo03/file.lex  */
   #include "expr.tab.h"
%}

%option noinput
%option nounput
%option header-file="lex.yy.h"
%%

[0-9]+\.[0-9]+      { yylval = atof(yytext); return FLOAT;}
\+             return PLUS;
.              { /* ignore everything else */ }
%%
  /* no code here */
```

demo03

This input file:

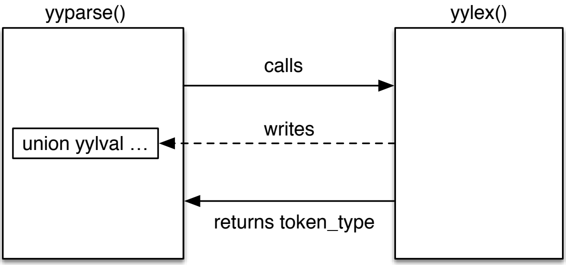
```
23.0+4.1+55.9
```

Gives this as output:

```
Found a float: 23.000000
Found a plus
Found a float: 4.100000
Found a plus
Found a float: 55.900002
```

Unions

- It is useful to send different data types from `flex` to `bison` using `yylval`
- We can do it declaring `yylval` as an union
- This is so common that `bison` already supports it
- Besides declaring the union, we must also declare the type of each token so bison can automatically read the correct value from the `yylval` union
- The type of each token is declared indirectly using the name of the field in the union



demo04 – Unions

```
%{  /* demo04/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%union {
         int number;
         char op;
       }
%token <number> INTEGER
%token <op> OPERATOR
%%
   S    : S EXPR
         |
         ;
   EXPR :  INTEGER  { printf("Found an int: %d\n", $1) ; }
           |  OPERATOR   { printf("Found an op: %c\n",$1);}
        ;
%%
int main() {
      return yyparse();
}
```

demo04 – Unions

```
%{    /* demo04/file.lex  */
   #include "expr.tab.h"
%}
%option noinput
%option nounput
%option header-file="lex.yy.h"
%%
[0-9]+         { yylval.number=atoi(yytext); return INTEGER;}
[-+*/]         { yylval.op=yytext[0];  return OPERATOR; }
.              { /* ignore everything else */ }
%%
```

- On the lex side we just use the `yylval` union

demo04

This input file:

```
23+4*55-2/4
```

Gives this as output:

```
Found an int: 23
Found an op: +
Found an int: 4
Found an op: *
Found an int: 55
Found an op: -
Found an int: 2
Found an op: /
Found an int: 4
```

A technique used many times

- The token types defined by `bison` correspond to numbers over 256
- If we are interested in single ASCII chars we can use the “token type” to pass those chars straight to `bison`
- This can be used with or without using unions
- As usual, an example is required…

demo05 – Passing chars

```
%{  /* demo05/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%token INTEGER
%%
   S    : S EXPR
          | ;
   EXPR :  INTEGER  { printf("Found an int: %d\n", $1) ; }
           | ’+’   { printf("Found a plus\n");}
           | ’-’   { printf("Found a minus\n");}
           | ’*’   { printf("Found a multiply\n");}
           | ’/’   { printf("Found a divide\n");}
        ;
%%
int main() {
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

demo05 – Passing chars

```
%{    /* demo05/file.lex  */
   #include "expr.tab.h"
%}

%option noinput
%option nounput
%option header-file="lex.yy.h"
%%

[0-9]+         { yylval = atoi(yytext); return INTEGER;}
[-+*/]         return yytext[0] ; /* the first/only char of yytext */
.              { /* ignore everything else */ }
%%
```

demo05

This input file:

```
23+4*55-2/4
```

Gives this as output:

```
./demo05   <sample.txt
Found an int: 23
Found a plus
Found an int: 4
Found a multiply
Found an int: 55
Found a minus
Found an int: 2
Found a divide
Found an int: 4
```

How to use the $...

- $1 is the first *field*, $2 is the second *field*,...

demo06 – Using *fields*

```
%{  /* demo06/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%token INTEGER SUM MUL
%%
   S    : S LINE
         | ;
   LINE :  SUM INTEGER INTEGER ’\n’ { printf("Sum: %d\n", $2+$3)  ; }
         | MUL INTEGER INTEGER ’\n’ { printf("Mul: %d\n", $2*$3)  ; }
        ;
%%
int main() {
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

demo06 – Using *fields*

```
%{    /* demo06/file.lex  */
   #include "expr.tab.h"
%}
%option noinput
%option nounput
%option header-file="lex.yy.h"
%%
[0-9]+         { yylval=atoi(yytext); return INTEGER;}
mul            return MUL;
sum            return SUM;
\n             return yytext[0];
.              { /* ignore everything else */ }
%%
```

demo06

This input file:

```
mul 3 3
sum 3 3
```

Gives this as output:

```
./demo06   <sample.txt
Mul: 9
Sum: 6
```

More about $. . .

- $$ is the value of the left hand symbol (the whole expression)
- By default $$=$1 if we don’t use it
- We can use it to pass values and make calculations
- Example: we want a total when the file ends on the previous example

demo07 – Using *fields*

```
%{    /* demo07/file.lex  */
   #include "expr.tab.h"
%}
%option noinput
%option nounput
%option header-file="lex.yy.h"
%%
[0-9]+         { yylval=atoi(yytext); return INTEGER;}
mul            return MUL;
sum            return SUM;
\n             return yytext[0];
.              { /* ignore everything else */ }
%%
```

- No changes on the `flex` file

demo07 – Using *fields*

```
%{  /* demo07/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%token INTEGER SUM MUL
%%
S : TOTAL  { printf("Total: %d\n",$1); }
      ;
TOTAL : TOTAL LINE { $$=$1+$2;}
          |    {$$=0;} ;
LINE : SUM INTEGER INTEGER ’\n’ { printf("Sum: %d\n", $2+$3);
                                                        $$=$2+$3;}
    | MUL INTEGER INTEGER ’\n’ { printf("Mul: %d\n", $2*$3);
                                                        $$=$2*$3;}
        ;
%%
int main() {
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

demo07

This input file:

```
mul 3 3
sum 3 3
mul 2 4
```

Gives this as output:

```
Mul: 9
Sum: 6
Mul: 8
Total: 23
```

The Bison file slowly – Part 1: Definitions

- By default all tokens place an integer on `yylval`
- By default all grammar non-terminals also have a type of integer

```
%{  /* demo07/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%token INTEGER SUM MUL
%%
```

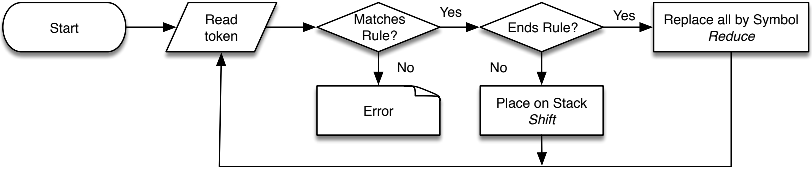
The Bison file slowly – Part 2: Grammar

- We have created a new rule `TOTAL` to provide the sum of the lines (line 9)
- When we find a line, the sum is the previous one plus the value of the line (line 9)
- The value of an empty expression should be zero (line 10)
- When we find a sum or a multiplication, we must calculate the value of the line (lines 11 to 14)

```
 6  %%
 7  S : TOTAL  { printf("Total: %d\n",$1); }
 8        ;
 9  TOTAL : TOTAL LINE { $$=$1+$2;}
10           |    {$$=0;} ;
11  LINE : SUM INTEGER INTEGER ’\n’ { printf("Sum: %d\n", $2+$3);
12                                                          $$=$2+$3;}
13      | MUL INTEGER INTEGER ’\n’ { printf("Mul: %d\n", $2*$3);
14                                                          $$=$2*$3;}
15          ;
16  %%
```

How does Bison works?

- Bison is a bottom-up parser, it starts from the bottom
- It starts reading tokens, and placing those tokens on a stack
- When all the elements of a grammar rule are found, bison empties the stack and places on the stack the corresponding non-terminal
- Shift – means in Bison-talk "insert on the stack"
- Reduce – means in Bison-talk "replace the tokens by the rule"



How to really see what Bison does?

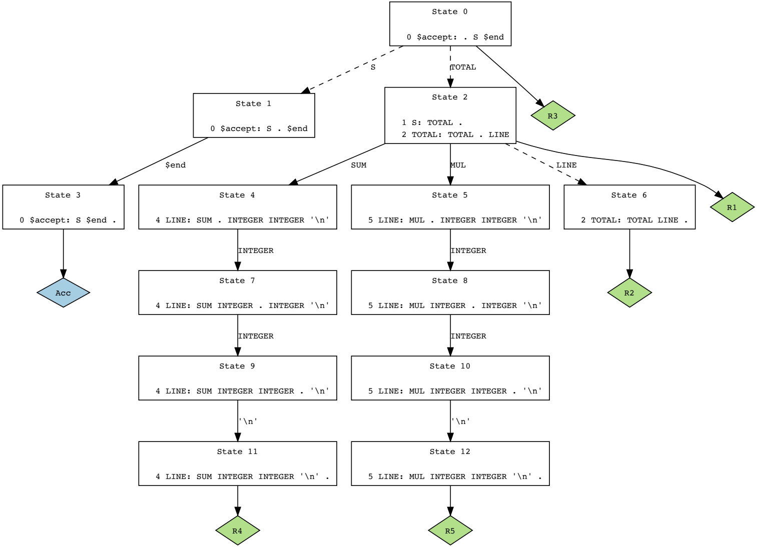
- Listing of the states

  `make report` — creates a `.lst` text file with the Bison states
- Graph with the states

  `make graph` — creates a `.gv` Graphviz file with the Bison states
- Debugging/trace of the grammar

  `%define parse.trace` — on the Bison definitions and
  `yydebug=1;` — on the `main` C function
  One must do both in order to activate traces on Bison!

The graph of demo07



demo08 – Activating *tracing* (excerpt)

```
S : TOTAL  { printf("Total: %d\n",$1);   } ;
TOTAL : TOTAL LINE { $$=$1+$2;}
         |   {$$=0;} ;
LINE : SUM INTEGER INTEGER ’\n’ { printf("Sum: %d\n", $2+$3);
                                                        $$=$2+$3;}
    | MUL INTEGER INTEGER ’\n’ { printf("Mul: %d\n", $2*$3);
                                                        $$=$2*$3;}
        ;
%%
int main() {
      yydebug=1;     /* activate trace */
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

demo09 – A simple grammar

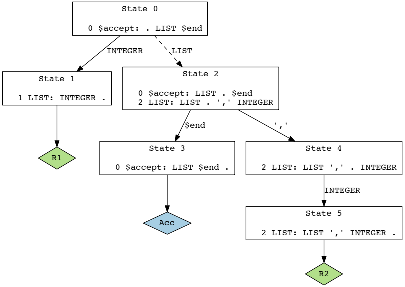
```
%{  /* demo09/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%token INTEGER
%define  parse.trace
%%
   LIST    : INTEGER
         | LIST ’,’ INTEGER
         ;
%%
int main() {
      yydebug=1;
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

demo09 – A simple grammar

- We can trace demo09 with a simple file:

```
10,20,30
```

The graph of demo09

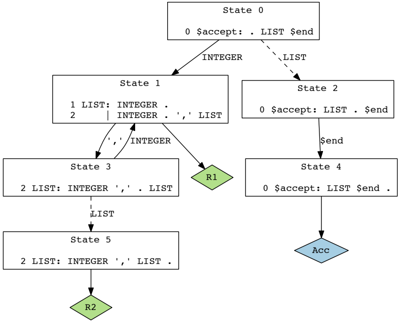


```
Starting parse
Entering state 0
Reading a token: Next token is token INTEGER ()
Shifting token INTEGER ()
Entering state 1
Reducing stack by rule 1 (line 8):
   $1 = token INTEGER ()
-> $$ = nterm LIST ()
Stack now 0
Entering state 2
Reading a token: Next token is token ’,’ ()
Shifting token ’,’ ()
Entering state 4
Reading a token: Next token is token INTEGER ()
Shifting token INTEGER ()
Entering state 5
Reducing stack by rule 2 (line 9):
   $1 = nterm LIST ()
   $2 = token ’,’ ()
   $3 = token INTEGER ()
-> $$ = nterm LIST ()
Stack now 0
Entering state 2
Reading a token: Next token is token ’,’ ()
Shifting token ’,’ ()
Entering state 4
Reading a token: Next token is token INTEGER ()
Shifting token INTEGER ()
Entering state 5
Reducing stack by rule 2 (line 9):
   $1 = nterm LIST ()
   $2 = token ’,’ ()
```

demo10 – A simple grammar – right recursive

```
%{  /* demo10/expr.y  */
     #include "lex.yy.h"
     void yyerror( char * s );
%}
%token INTEGER
%define  parse.trace
%%
   LIST    : INTEGER
         | INTEGER ’,’ LIST
         ;
%%
int main() {
      yydebug=1;
      return yyparse();
}
void  yyerror(char *s){
        printf("Syntactic/semantic error: %s\n",s);
}
```

The graph of demo10



```
Starting parse
Entering state 0
Reading a token: Next token is token INTEGER ()
Shifting token INTEGER ()
Entering state 1
Reading a token: Next token is token ’,’ ()
Shifting token ’,’ ()
Entering state 3
Reading a token: Next token is token INTEGER ()
Shifting token INTEGER ()
Entering state 1
Reading a token: Next token is token ’,’ ()
Shifting token ’,’ ()
Entering state 3
Reading a token: Next token is token INTEGER ()
Shifting token INTEGER ()
Entering state 1
Reading a token:
Now at end of input.
Reducing stack by rule 1 (line 8):
   $1 = token INTEGER ()
-> $$ = nterm LIST ()
Stack now 0 1 3 1 3
Entering state 5
Reducing stack by rule 2 (line 9):
   $1 = token INTEGER ()
   $2 = token ’,’ ()
   $3 = nterm LIST ()
-> $$ = nterm LIST ()
Stack now 0 1 3
Entering state 5
Reducing stack by rule 2 (line 9):
   $1 = token INTEGER ()
   $2 = token ’,’ ()
   $3 = nterm LIST ()
-> $$ = nterm LIST ()
Stack now 0
```

Conflicts – when Bison says things are wrong

Reduce-Reduce – Bison has two rules that “match”

The grammar should be corrected to be unambiguous!

Shift-Reduce – Bison has the choice between shifting a token or reducing a rule

Bison always chooses to shift a token!
You have been warned, so you can’t complain!
Simple mnemonic: when choosing between two rules, the longest one wins

demo11 – A simple calculator

- A calculator with 100 memory positions: #0 to #99

This input file:

```
#10=2;
print #10;
#2=#10+20;
print #2;
```

Gives this as output:

```
Printing 2
Printing 22
```

demo11 – The Flex file

```
%{
#include "calc.tab.h"
%}
%option noinput
%option nounput
%option header-file="lex.yy.h"
%%
"print"            {return PRINT;}
"exit"             {return EXIT_COMMAND;}
#[0-9][0-9]?       {yylval.num = atoi(yytext+1); return IDENTIFIER;}
[0-9]+             {yylval.num = atoi(yytext); return NUMBER;}
[ \t\n]            ;
[-+=;]             {return yytext[0];}
.                  {printf("unexpected character");}

%%

```

demo11 – The Bison file part 1

```
%{
#include <stdio.h>     /* C declarations used in actions */
#include <stdlib.h>
// #include <ctype.h>
#include "lex.yy.h"
int memory[100];
void yyerror (char *s);
%}
%union {int num; char id;}
%start LINE
%token PRINT
%token EXIT_COMMAND
%token <num> NUMBER
%token <num> IDENTIFIER
%type <num> LINE EXPR TERM  COMMAND
%type <id> ASSIGNMENT

```

demo11 – The Bison file part 2

```
%%
LINE    : COMMAND ’;’
        | LINE COMMAND ’;’
        ;

COMMAND : ASSIGNMENT         {;}
        | EXIT_COMMAND       {exit(EXIT_SUCCESS);}
        | PRINT EXPR         {printf("Printing %d\n", $2);}
        ;
ASSIGNMENT : IDENTIFIER ’=’ EXPR  { memory[$1]=$3; }
                        ;
EXPR     : TERM                      {$$ = $1;}
        | EXPR ’+’ TERM              {$$ = $1 + $3;}
        | EXPR ’-’ TERM              {$$ = $1 - $3;}
        ;
TERM    : NUMBER                    {$$ = $1;}
        | IDENTIFIER                {$$ = memory[$1];}
        ;

```

demo11 – The Bison file part 3

```
%%
int main (void) {
        return yyparse ( );
}
void yyerror (char *s) {fprintf (stderr, "%s\n", s);}
```

Things to do:

- Add multiplication and division (check if they are ok).
- Add exponentiation (check if it works ok).