Flex , **Bison** and the ACSE compiler suite

Marcello M. Bersani

LFC – Politecnico di Milano

Syntactic analysis

- The syntactic analysis recognizes **structures** of a language
  - The **structure** is defined by a grammar G
- Structure is constituted by atomic elements
  - Tokens recognized by the lexical analysis

$$P \longrightarrow (P) \mid (P) P \mid n P \mid n$$

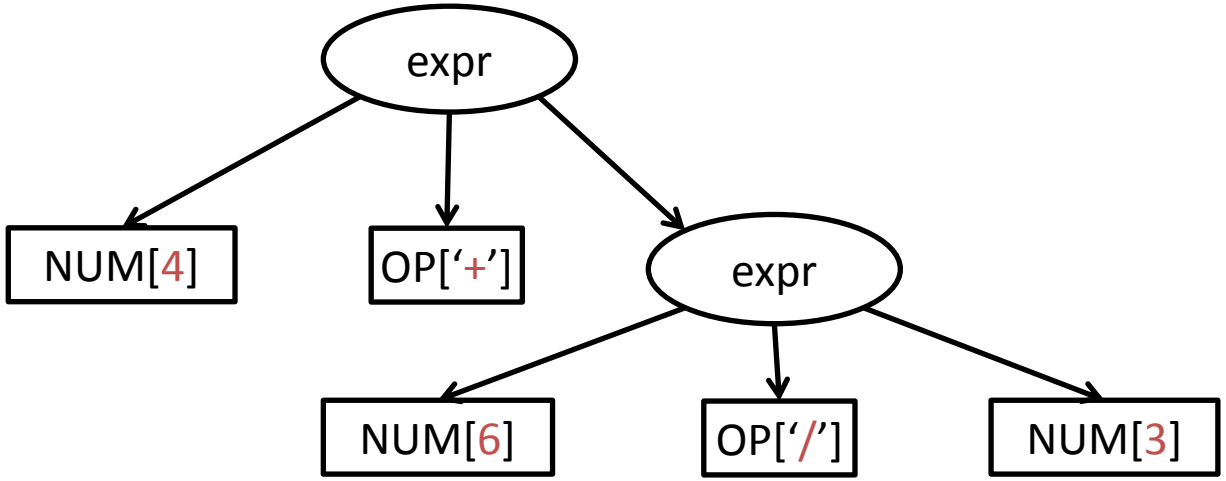
Syntactic analysis

Input 4 + 6 / 3

Sematic value of token between [] is **yylval** of Bison

Lexical NUM[4] - OP[‘+’] - NUM[6] - OP[‘/’] - NUM[3]

Syntactic

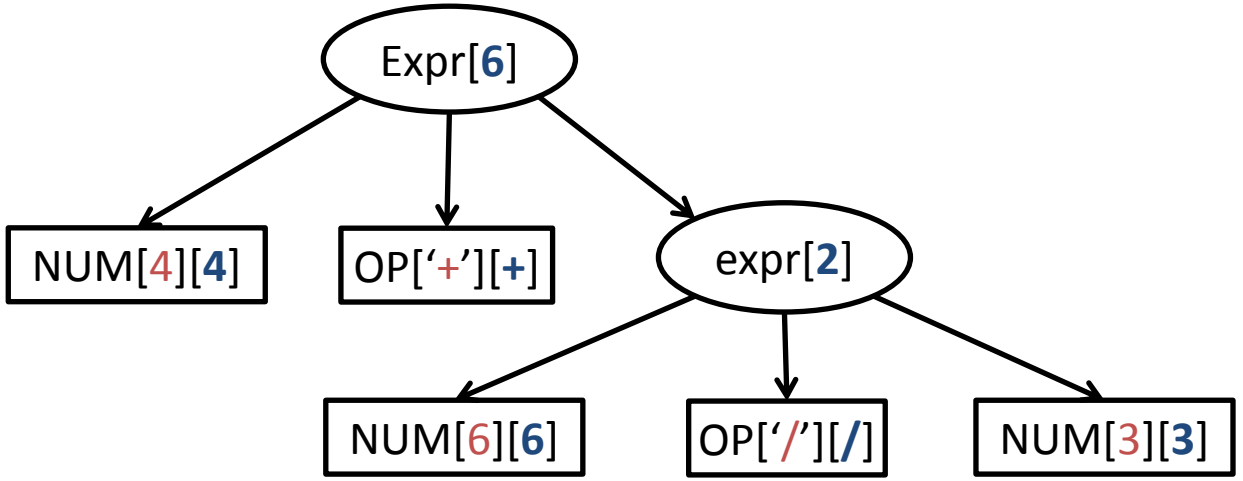


Semantic analysis

Input 4 + 6 / 3

Lexical NUM[4] - OP[‘+’] - NUM[6] - OP[‘/’] - NUM[3]

Semantic
Syntactic



Parser

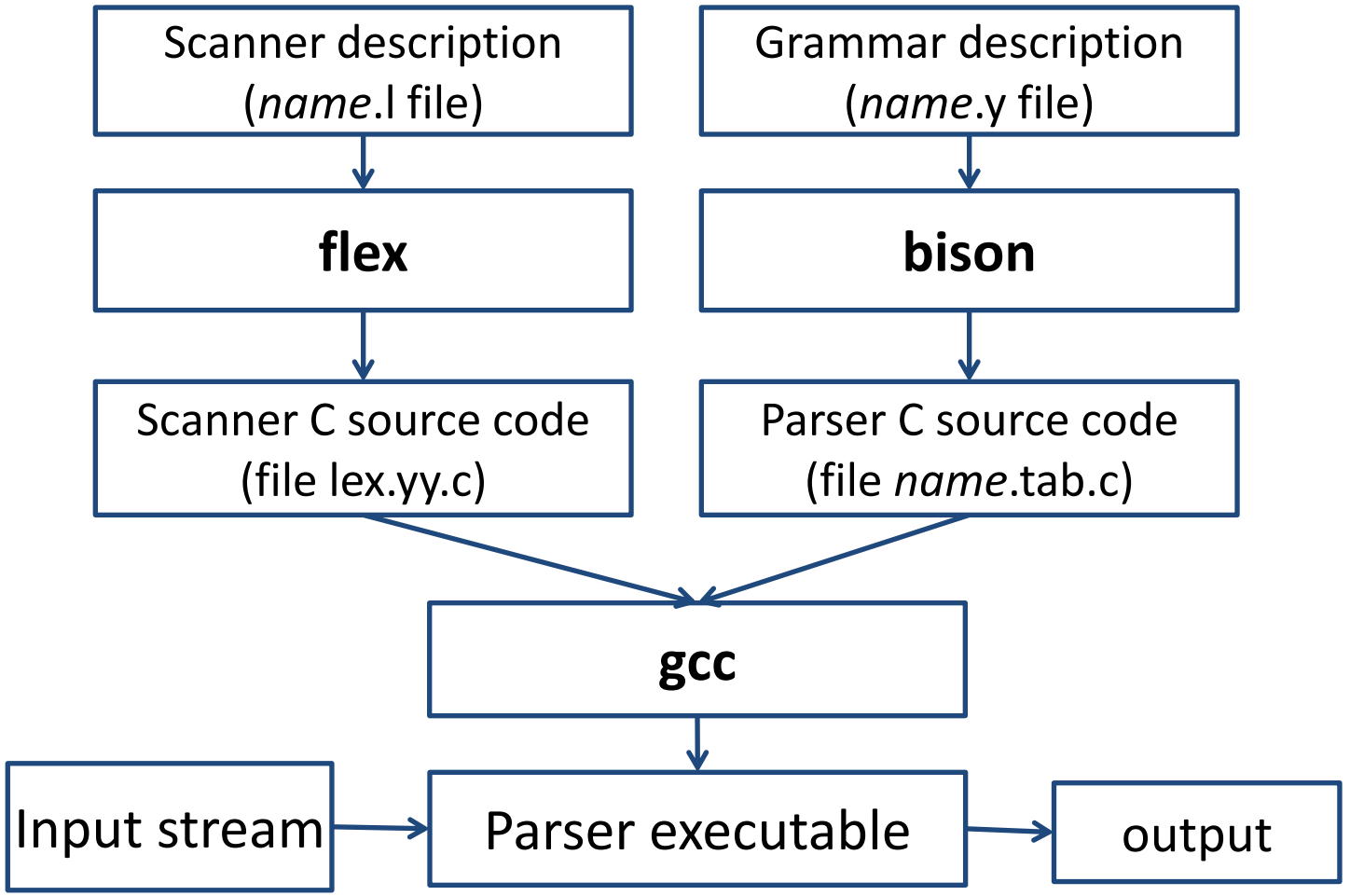
- A program that performs syntactic analysis.
  - LL (left to right-leftmost)
  - LR (left to right-rightmost)
- Parsing is enriched with functionalities to produce output
  - Grammar rules are associated with **actions**

    expr: expr + expr {do_something();}

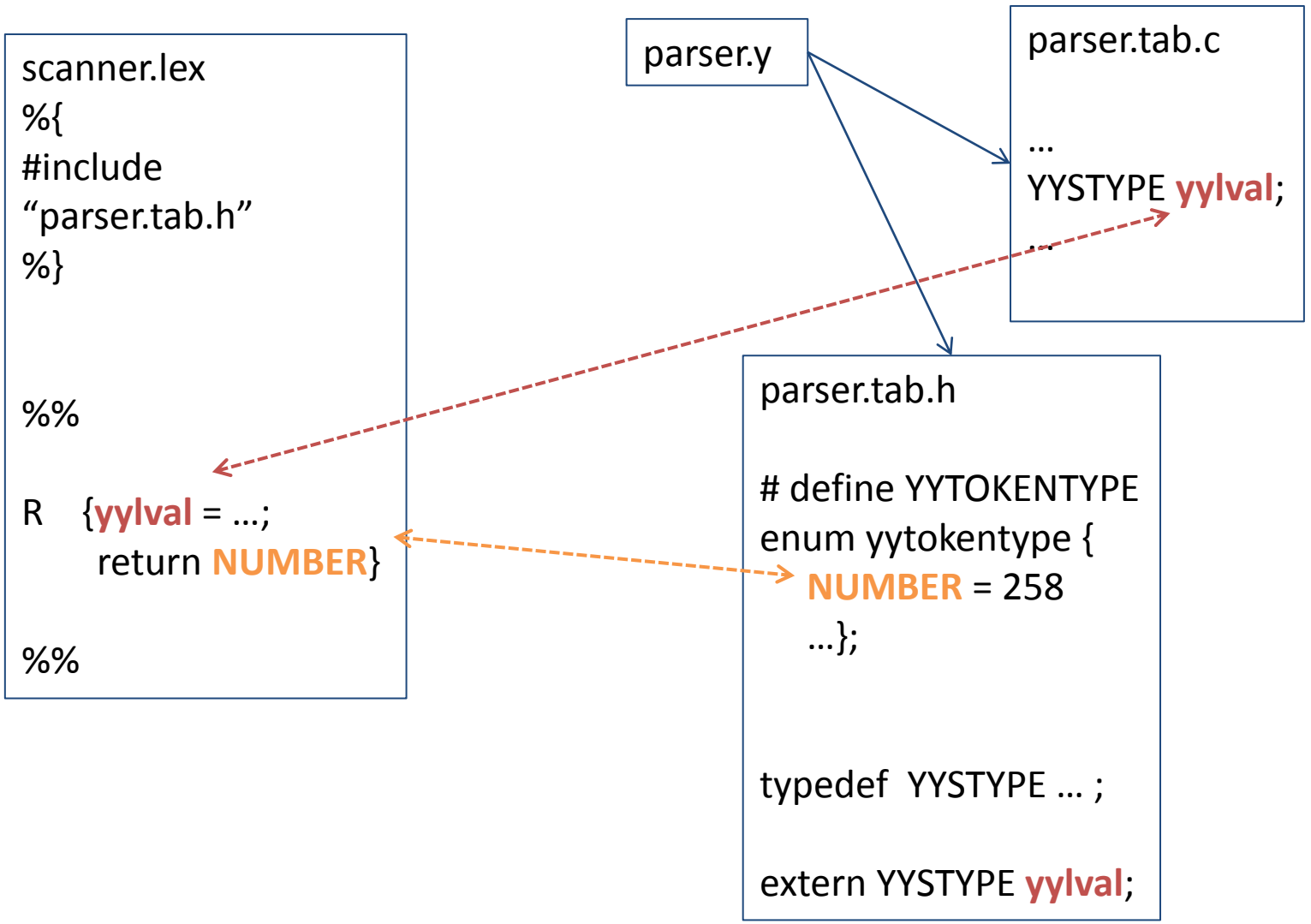
- **Bison** is a free parser generator

www.gnu.org/software/bison/

Interaction Flex/Bison



Interaction Flex/Bison I



Bison input

%{ C definitions }%

Bison definitions

%%
Grammar rules
%%

C user code

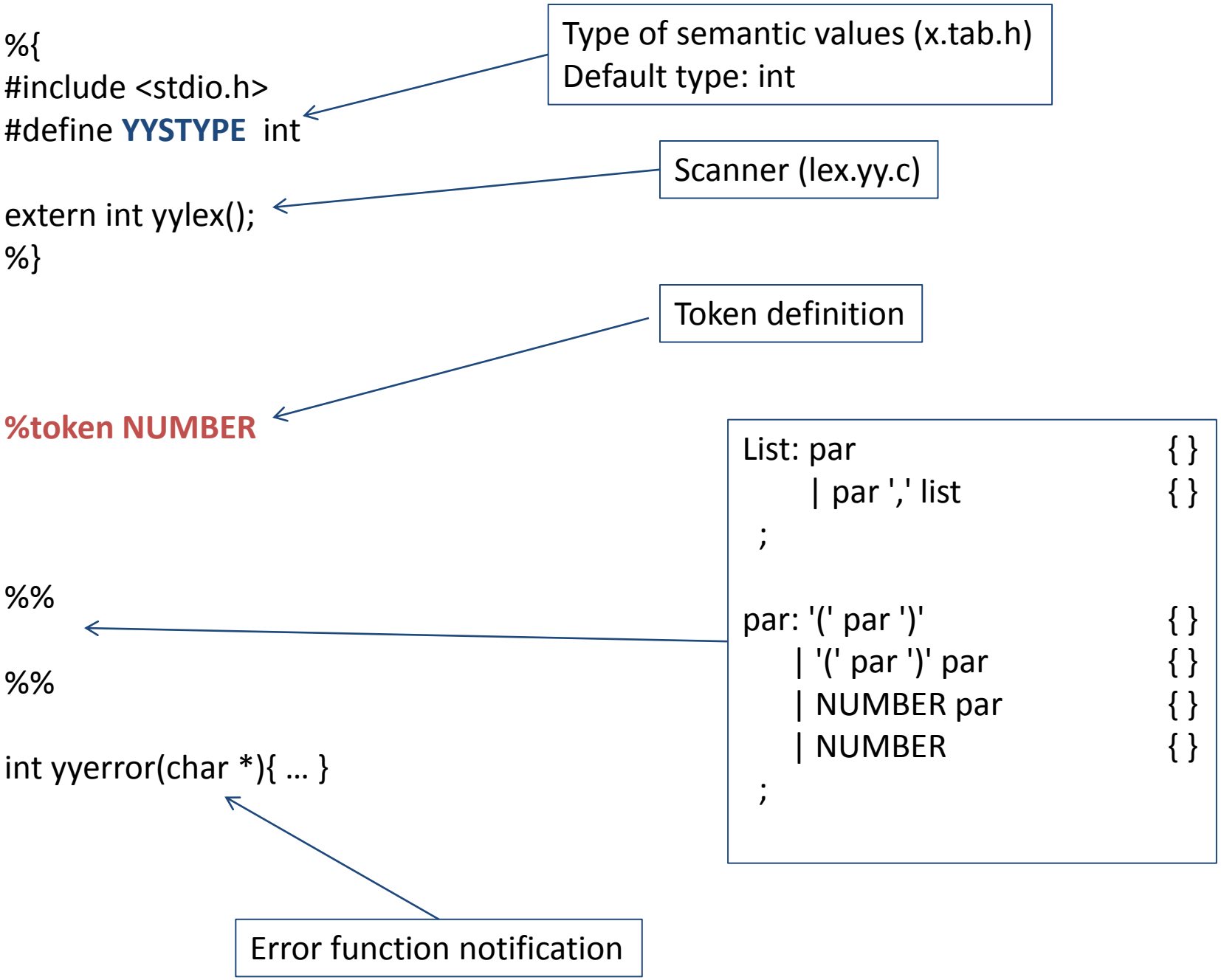
Example 1

- Define a parser for the grammar

$L \rightarrow P \mid P, L$

$P \rightarrow (P) \mid (P)\ P \mid n\ P \mid n$

Remark: L is **right-recursive**


%{
#include <stdio.h>
#define YYSTYPE int
extern int yylex();
%}
%token NUMBER
%%
%%
int yyerror(char *){ … }
Type of semantic values (x.tab.h)
Default type: int
Scanner (lex.yy.c)
Token definition
List: par { }
| par ',' list { }
;
par: '(' par ')' { }
| '(' par ')' par { }
| NUMBER par { }
| NUMBER { }
;
Error function notification

Terminal symbols

- Three ways to define a terminal
  - %token
  - Single character ‘…’
  - String “…”
- **yylex()** always returns
  - a positive value when a token is read
  - nonpositive when EOF is read

Remark on lists

Left-recursive

L -> P | L, P

L -> L,P -> L,P,P -> P,P,P

Stack:

P	[reduce]
L	[shift]
L,P	[reduce]
L	[shift]
L,P	[reduce]
L	

Right-recursive

L -> P | P, L

L -> P,L -> P,P,L -> P,P,P

Stack:

P	[shift]
P,P	[shift]
P,P,P	[reduce]
P,P,L	[reduce]
P,L	[reduce]
L	

Reverse Polish Calculator

Grammar Rules:

S → S E | ε

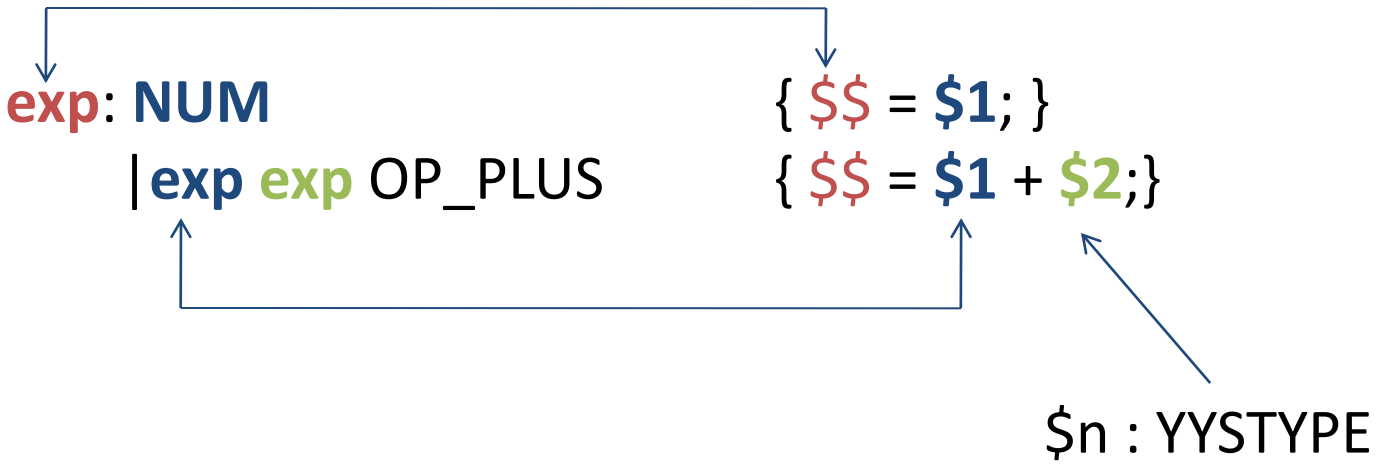
E → n

E → E E + | E E - | E E * | E E / | E E ^ | E -

```
%{
#define YYSTYPE double
#include <math.h>
%}

%token NUM
%token OP_PLUS
%token OP_MINUS
%token OP_MUL
%token OP_DIV
%token OP_EXP
%token UN_MINUS
%token NEWLINE

%%

%%



int yyerror(char * s){
  printf("%s\n",s);
}

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

```
input: /* empty */
         | input line
         ;

line:  NEWLINE
       | exp NEWLINE { printf ("\t%.10g\n", $1); }
       ;

exp:     NUM                  { $$ = $1; }
      | exp exp OP_PLUS       { $$ = $1 + $2;}
      | exp exp OP_MINUS      { $$ = $1 - $2;}
      | exp exp OP_MUL        { $$ = $1 * $2;}
      | exp exp OP_DIV        { $$ = $1 / $2;}
      | exp exp OP_EXP        { $$ = pow ($1, $2);}
      | exp UN_MINUS          { $$ = -$1; }
    ;
```

Rules and semantic values



- $$ is accessible **only** in the **last** semantic action
  - $$ = $1 is the default action

How to compile bison files

`bison -d rpn.y`

- bison outputs file “name.tab.h”
  - YYSTYPE
  - tokens

```
#ifndef YYSTYPE
#define YYSTYPE double
#endif

#define  NUM      257
#define  OP_PLUS          258
#define  OP_MINUS         259
#define  …

extern YYSTYPE yylval;
```

Bison/Flex

- The above file should be included in the **flex** input (**YYSTYPE**)

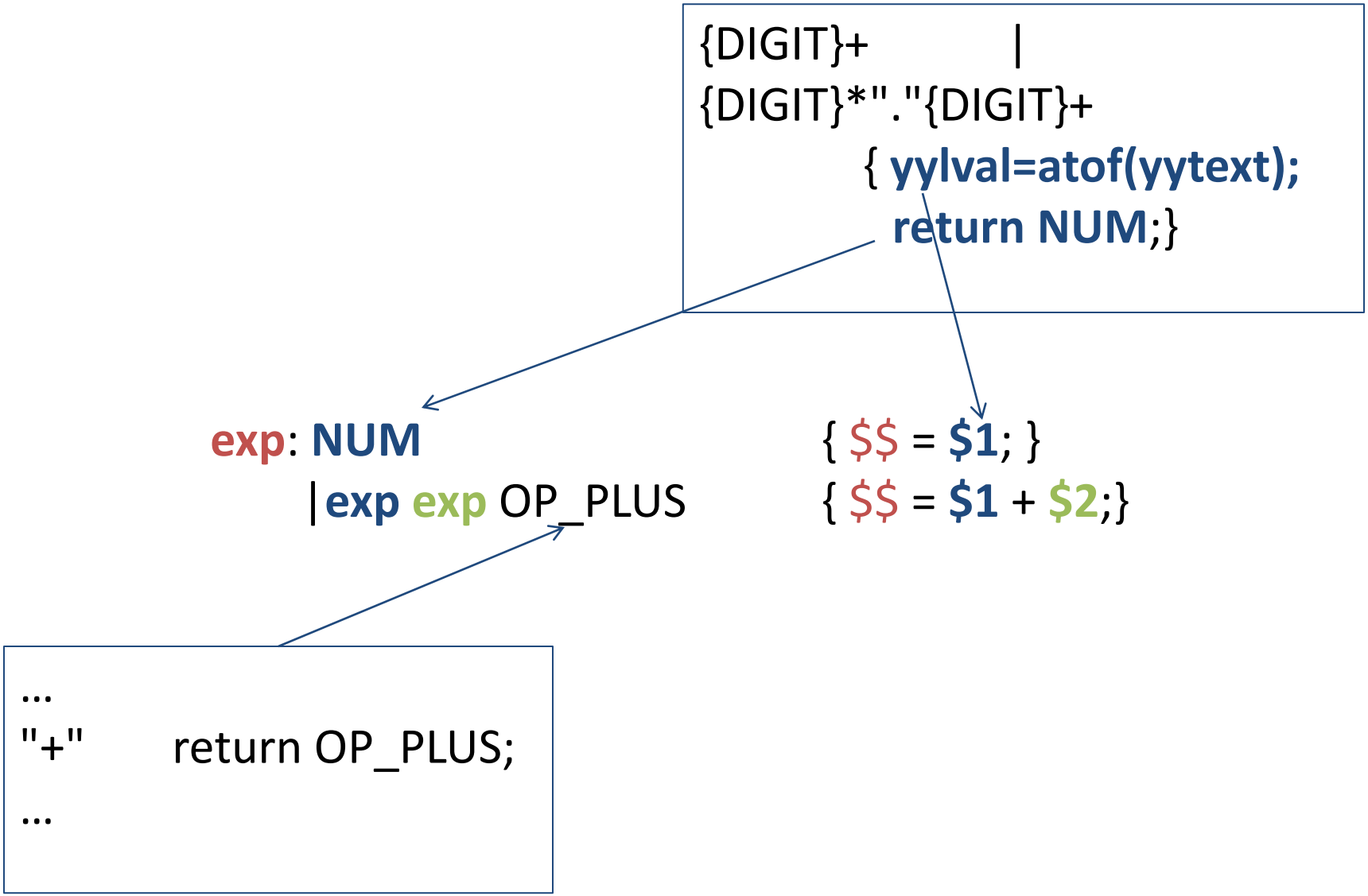
- The lexical actions store the semantic value of each token in **yylval** variable (declared in generated header file)
  - **yylval** is global for non-reentrant parser
  - **yylval** is a YYSTYPE* for reentrant parser

RPN lexer

```
%{
#define YYSTYPE double
#include "rpn.tab.h"
#include <stdlib.h>
%}

%option noyywrap

DIGIT    [0-9]
BLANKS   [ \t]


%%
%%
```

```
{BLANKS}+
"+"         return OP_PLUS;
"-"         return OP_MINUS;
"/"         return OP_DIV;
"*"         return OP_MUL;
"^"         return OP_EXP;
"n"         return UN_MINUS;
"\n"        return NEWLINE;
{DIGIT}+          |
{DIGIT}*"."{DIGIT}+
            { yylval=atof(yytext);
              return NUM;}
```

Interaction



Build the whole

1. **bison** –d rpn.y
2. **flex** rpn.lex
3. **gcc** rpn.tab.c lex.yy.c

Infix calculator

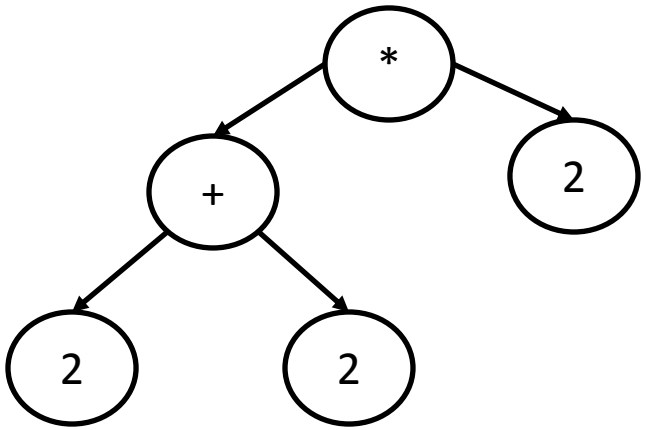
Grammar Rules:

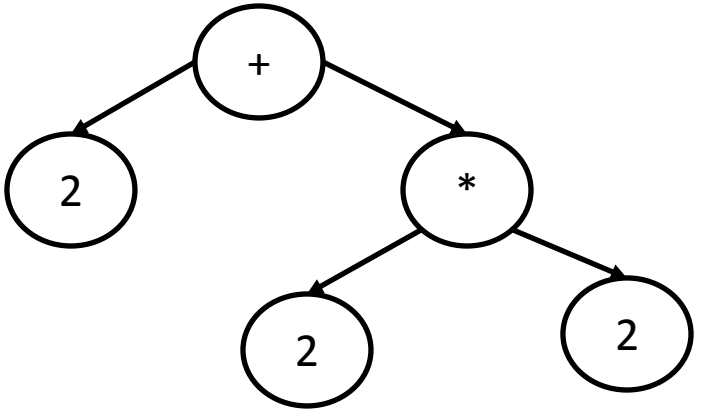
$S \rightarrow E \mid \varepsilon$

$E \rightarrow n \mid E + E \mid E * E \mid (E)$

Ambiguity / shift-reduce conflict

- String 2+2*2





E -> E*E -> E*2 -> E+E*2
-> E+2*2 -> 2+2*2

E -> E+E -> E+E*E -> E+E*2
-> E+2*2 -> 2+2*2

How to solve ambiguity

- Rewrite the grammar into an equivalent form

$S \rightarrow E \mid \varepsilon$

$E \rightarrow E + M \mid M$

$M \rightarrow T * M \mid T$

$T \rightarrow n \mid (E)$

- Exploit Bison convention
  - Promote shift instead of reduce
  - Unless operator precedence/associativity defined

If-then-else

```
If_stmt:
    IF exp THEN stmt
    | IF exp THEN stmt ELSE stmt
;
```

if x then if y then a() else b();

Bison promotes SHIFT

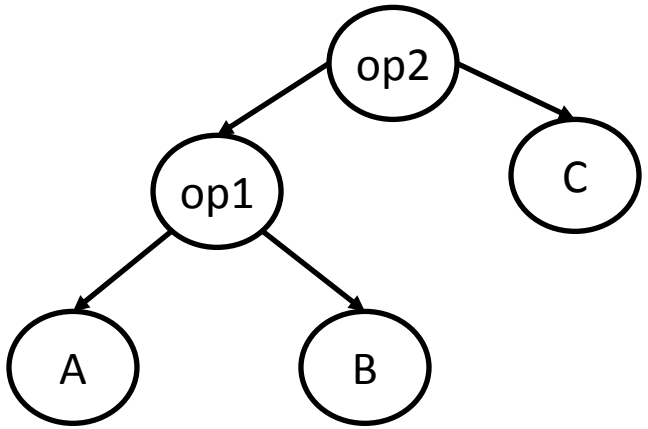
- However shift is not always the solution

Operator precedence

A op1 B op2 C

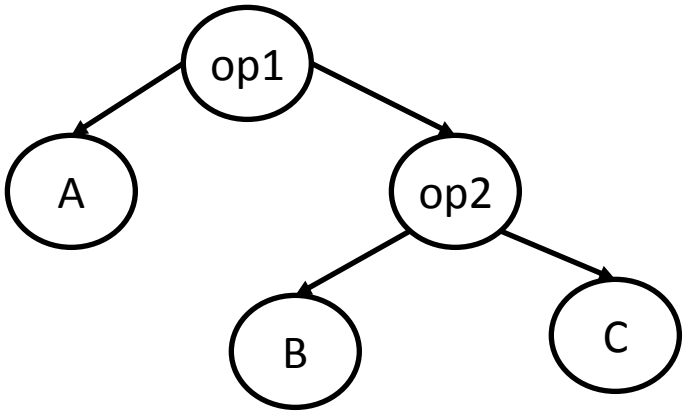
***op1* > op2**

(A op1 B) op2 C



***op1* < op2**

A op1 (B op2 C)

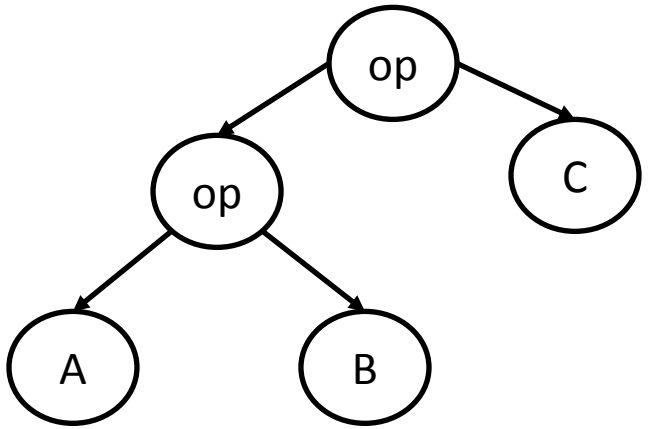


Operator associativity

A op B op C

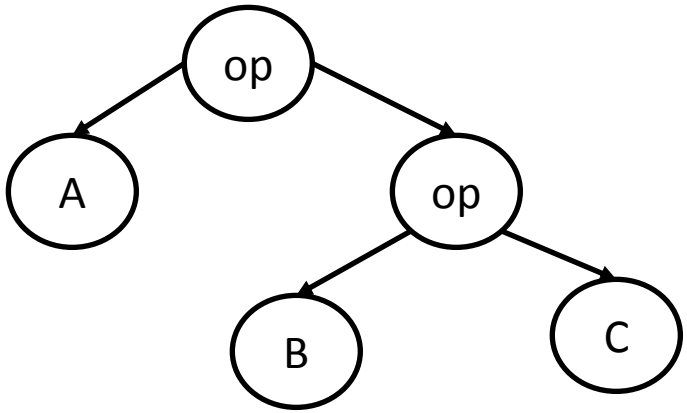
op is **left-associative**

(A op B) op C



op is **right-associative**

A op(B op C)



Definition of precedence/associativity

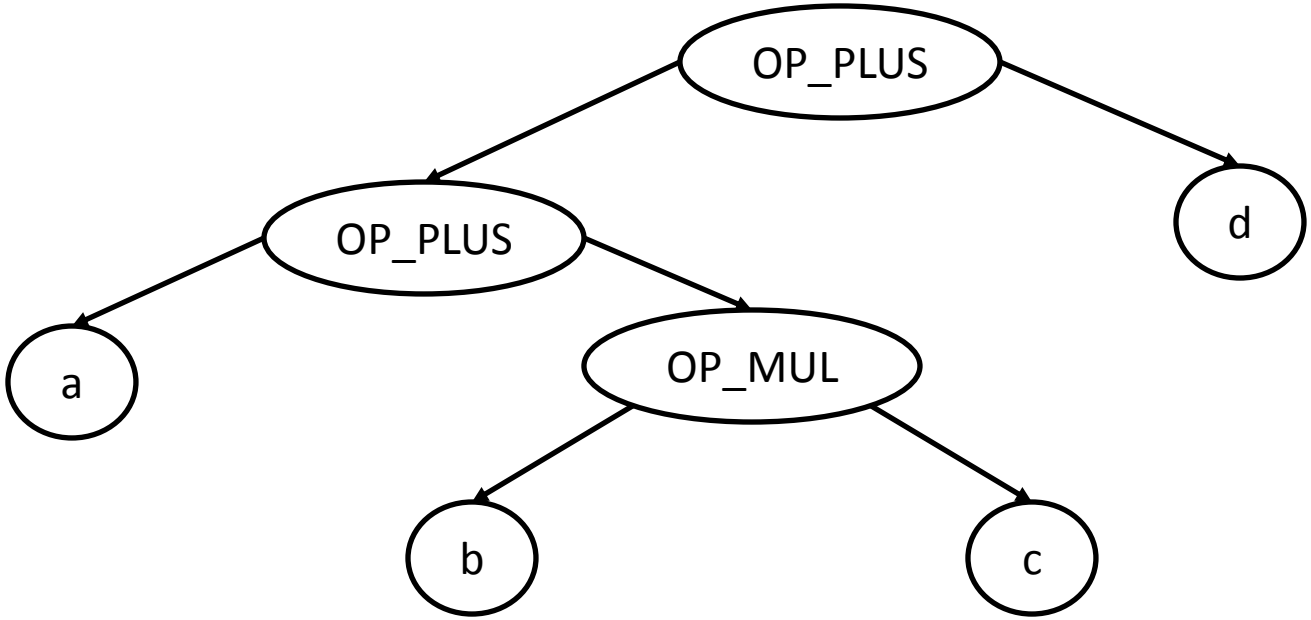
- **%left**, **%right**, **%noassoc** define **associativity**
  - **Only for terminals**
  - Section Bison definitions

%left A1,A2
%right B1,B2 %left D1
%left C1

- Operators in the same declaration have equal precedence
  - A1≡A2, B1≡B2≡D1
- Declaration **order** defines **precedence**
  - Ai < Bi < C1

Precedence in action

%left OP_PLUS,OP_MINUS
%left OP_MUL,OP_DIV



a OP_PLUS **b** OP_MUL **c** OP_PLUS **d**

How Bison solves precedence

- Precedence is assigned to each declared operator
- Each rule R containing those operators is assigned the same precedence as the last declared symbol in rule
- Conflicts are resolved by comparing the precedences of the LA and of the rule R.

How Bison solves precedence

- If LA > R
  - then SHIFT
  - else REDUCE.
- If LA = R (check associativity)
  - R is left → REDUCE R
  - R is right → SHIFT
- If either rule R or LA has no precedence the default is SHIFT

Example of parsing prec

- a +
- a + b
- **a + b** *
- a + b * c
- a + **b * c** +
- a + **E** +
- **E + E** +
- **E** + d 〉
- E 〉

- LA > **R** [shift]
- LA < **R** [reduce]
- LA = **R** [%left->reduce]

a OP_PLUS **b** OP_MUL **c** OP_PLUS **d**

Two different behaviours

Operators +, * are defined both with the same precedence

+ left, * right

1*2+3+4*5=

2 + 3

5 + 4 = 9

9 * 5 = 45

1 * 45

+ right, * left

1*2+3+4*5=

1 * 2

4 * 5

3 + 20 = 23

2 + 23

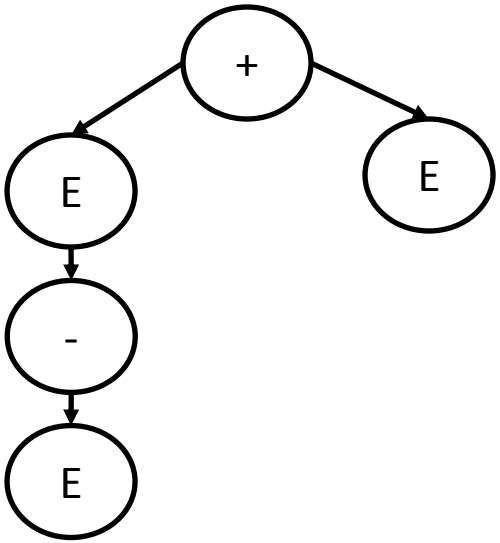
Extension Infix calculator Context precedence rules

Grammar Rules:

S → E | e

E → n | **-E** | E + E | E * E | E - E

- -E + E?
  - -(E+E)
  - <u>(-E)+E</u>



Context precedence of rules

- The precedence of an operator depends on the context
- **%prec** declares the precedence of a rule R by specifiying a terminal symbol whose precedence is used for rule R
  - %left,%right,%nonassoc are only for terminal

```
%left OP_PLUS, P_MINUS
%right OP_PER, OP_DIV
%left UMINUS
```

Declare a fictitious symbol

```
expr : ...
        | expr OP_PLUS expr
        | OP_MINUS expr %prec UMINUS
        ;
```

When OP_MINUS is followed by expr then the rule has the precedence of UMINUS

Bison grammar

```
%{
#define YYSTYPE double
#include <math.h>
%}
%token NUM
%token NEWLINE

/* operator precedence */
%left OP_PLUS OP_MINUS
%left OP_MUL OP_DIV
%left NEG
%right OP_EXP


%%

%%

int main(){
  yyparse();
}

int yyerror(char * s){
  printf("%s\n",s);
}
```

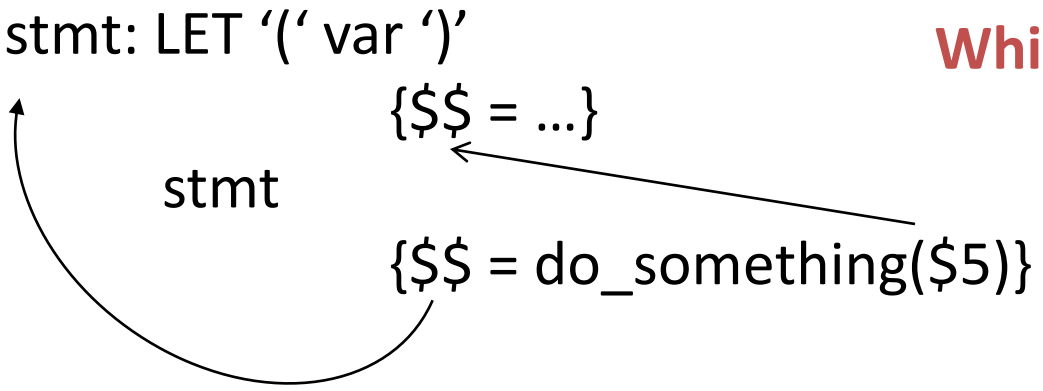
```
input: /* empty */
     | input line
     ;
line:  NEWLINE
     | exp NEWLINE  { printf ("\t%.10g\n", $1); }
     ;
exp:    NUM                 { $$ = $1;        }
     | exp OP_PLUS exp      { $$ = $1 + $3;   }
     | exp OP_MINUS exp     { $$ = $1 - $3;   }
     | exp OP_MUL exp       { $$ = $1 * $3;   }
     | exp OP_DIV exp       { $$ = $1 / $3;   }
     | OP_MINUS exp %prec NEG { $$ = -$2;      }
     | exp OP_EXP exp       { $$ = pow($1,$3); }

     | '(' exp ')'          { $$ = $2         }
     ;
```

Reduce-reduce conflicts

- Two or more rules apply for a reduction
  - Bison chose the first rule appearing in the grammar

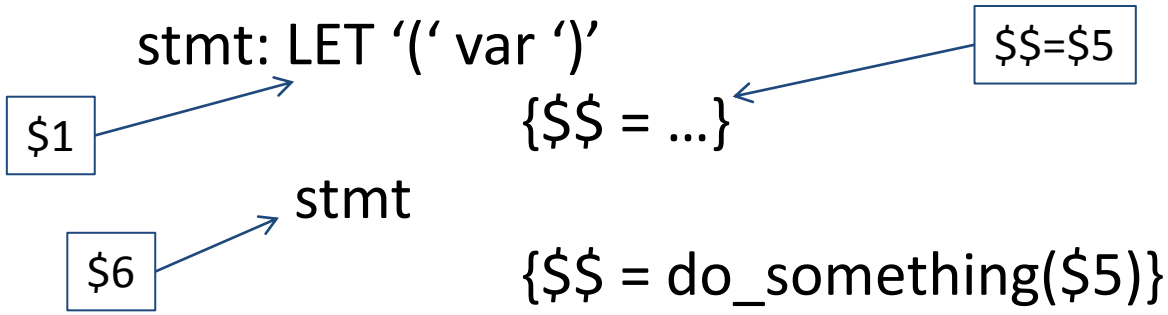
Mid-action rules

- Action within the rule
  - It can refer to semantic value of **previous** component of the rule by $n
  - It may have a semantic value
    - Set by (local) $$
    - Referred by $n in **later** actions



Which type??

Mid-action rules



- Each element of the rule rules has a semantic value $i ($i \in [1,n]$)
  - Terminals, non-terminals
  - Mid-action

More on semantic values

- **Terminals**, **non-terminals** and **rules** may have different semantic value type
  - Directive **%union** in the Bison declaration section

```
%union {
    type1 field1;
    …
    typeN fieldN;
}
```

More on semantic values

```
%union {
        int             i_value;
        char            *str;
        sym_str         *sptr;
}
```

- Names **i_value**, **str**, **sptr** are used
  - in **definitions** of terminals or non-terminals
  - or when **accessing** to semantic values of rules

More on semantic values

%token <i_value> NUMBER
%token <str> STRING

- Token **NUMBER** is an i_value [int]
- Token **STRING** is a str [char*]

%type <sptr> stmt

- Non-terminal **stmt** is a sptr [sym_str*]

Mid-action rules and semantic val.

- Actions within rules are not associated with a symbol name
  - %type can not be used
- The type of semantic value is specified by **$<type>n** construct

```
stmt: LET '(' var ')'
                    {$<context>$ = ...}
      stmt
                    {$$ = do_something($<context>5)}
```

context is a name defined in %union

Interaction Flex/Bison II

parser.y

```
%union {
  int  i_value;
  char *str;
  sym_str  *sptr;
}

%%
…
%%
```

parser.tab.c

```
…
YYSTYPE yylval;
…
```

scanner.lex

```
%{
#include “parser.tab.h”
%}

%%

R    {yylval.i_value = …;
       return NUMBER}

%%
```

parser.tab.h

```
# define YYTOKENTYPE
enum yytokentype {
     NUMBER = 258,
     …};


typedef union YYSTYPE{
  int          i_value;
  char*        str;
   sym_str *sptr;
} YYSTYPE;
extern YYSTYPE yylval;
```

Bison parsing function

- Function implementing the parser is **yyparse()**
  - Returns 0 when parsing is succesful
  - Returns 1 when the input is invalid
    - Syntax error
    - YYABORT is called by an action

- YYACCEPT
  - When invoked within an action returns 0
- YYABORT
  - When invoked within an action returns 1
- YYERROR
  - Rise a syntax error. Produces **error** token (see next)

Some things to be known

- Bison parser are non-reentrant by default
  - **yylval** is static global variable in *parser*.tab.c
  - **yylex()** accesses directly to it
- Reentrant parser can be defined **%pure-parser**
  - **yylex()** is

    int yylex(YYSTYPE * …, …)
  - **yylval** is local in **yyparse()** and evaluates to a pointer in **yylex()**

    yylval->num = …

Some things to be known

- Error handling; when an error occur
  - Discard the whole input
  - Recover the parsing
    - The parser produces the **error** token

```
stmts:
        | stmts stmt '\n'
        | stmts  error '\n'
```

- The parser
  - discards objects from the stack until it reaches a "safe" state in order to shift the **error** token
    - Discarded symbols must be managed (memory leaks)
  - If the LA is not acceptable, shifts until a good one