

# Compiler Lab

Introduction to tools

**Lex** and **Yacc**

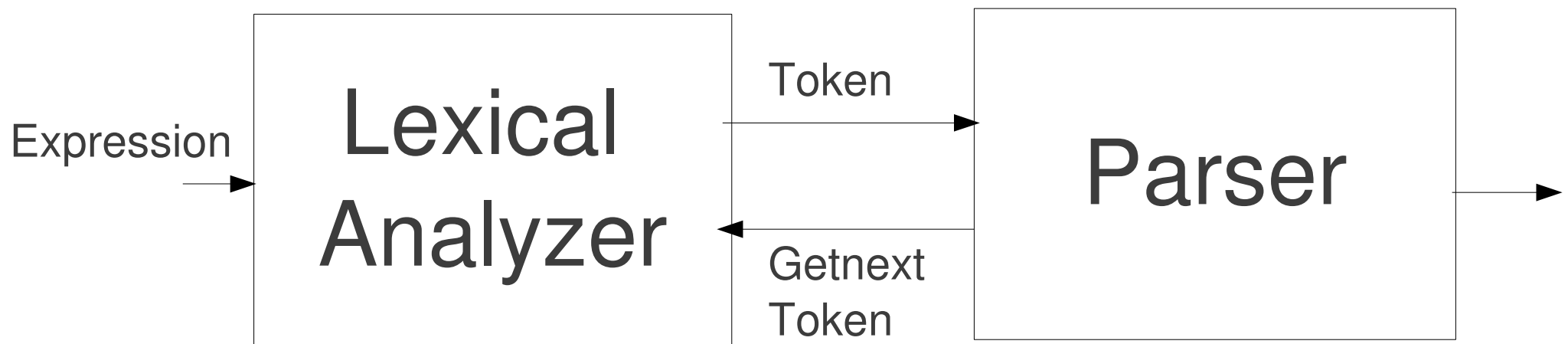
08.01.2015

# Assignment1

- Implement a simple calculator with tokens recognized using **Lex/Flex** and parsing and semantic actions done using **Yacc/Bison**.

# Calculator

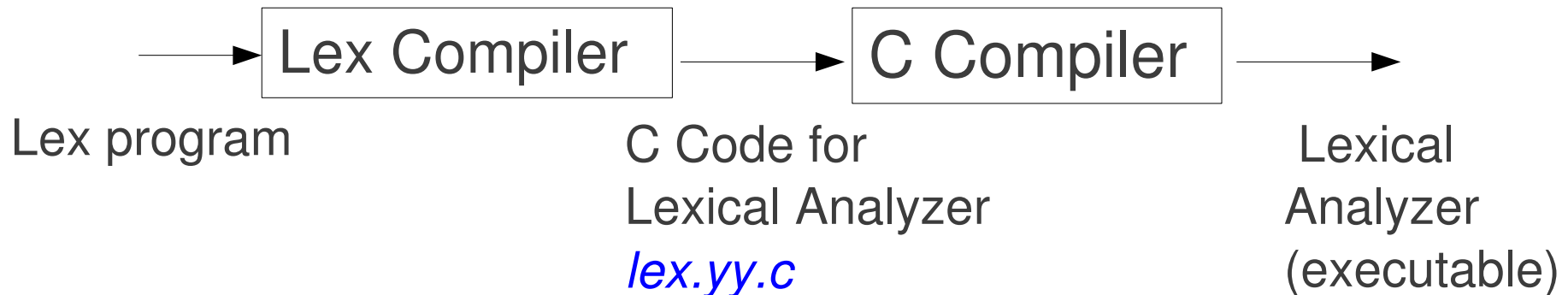
- Input:  $12 + 34 * 6$
- Separate into tokens :  $\langle \text{NUM}, 12 \rangle$   $\langle \text{ARITHOP}, + \rangle$ ,  
 $\langle \text{NUM}, 34 \rangle$   $\langle \text{ARITHOP}, * \rangle$   $\langle \text{NUM}, 6 \rangle$ 
  - Use a Lexical Analyzer
    - Patterns for various tokens
- Check the syntax ( $12+34*$  is not a valid expression)
  - Use a parser
    - Grammar rules
- Evaluate ?



# Calculator : Lexical Analysis

# Flex / Lex - A Lexical analyzer generator

- Input : Patterns (similar to regular expressions)
- Generates C code for a lexical analyzer(scanner)



# A portion of Lex program

%%

[0-9] { return NUM; }

[A-Z] { return ID; }

%%

Pattern

Action

## //Declaration section

digit [0-9] *// regular definition*

letter [a-zA-Z]

%%

## //Rules section

{digit}+ { return NUM;}

{letter} ({letter} | {digit})\* { return ID;}

%%



*%%*

```
{digit}+ {yylval = atoi (yytext); return NUM;}
```

*%%*

*yytext* : pointer to the string matched (lexeme)

*yylval* : attribute value for the token

# Lexical Analyzer, Parser communication

- LA should pass a **token name** and associated **attribute value** to the parser
  - e. g. **<NUM, 123>**
  - *return NUM*
    - returns a token name to the parser
  - *yylval = atoi (yytext)*
    - attribute value associated with the token

*//lex file, calcu.l*

```
%{  
    #include "y.tab.h"  
    extern int  yylval;  
}%  
digit [0-9]  
%%  
{digit}+  {yylval = atoi (yytext); return NUM;}  
[*+\n]    { return *yytext;}  
.  
{ }  
%%
```

# Calculator: Parsing

# Bison / Yacc- A parser generator

- Input : Context Free Grammar rules for the language
- Generates C code for a parser for your language

# An example

$E \rightarrow E + E$

$E \rightarrow 0 \mid 1 \mid 2$

Some strings in the language

$1+2$

$1+0+2$

$2+2+1+0$

Some strings not in the language

$1+$

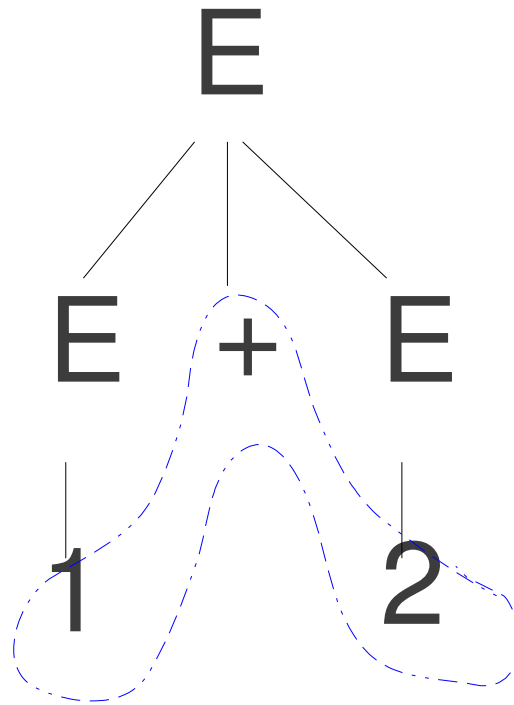
$+2+3$

$1++$

$12$

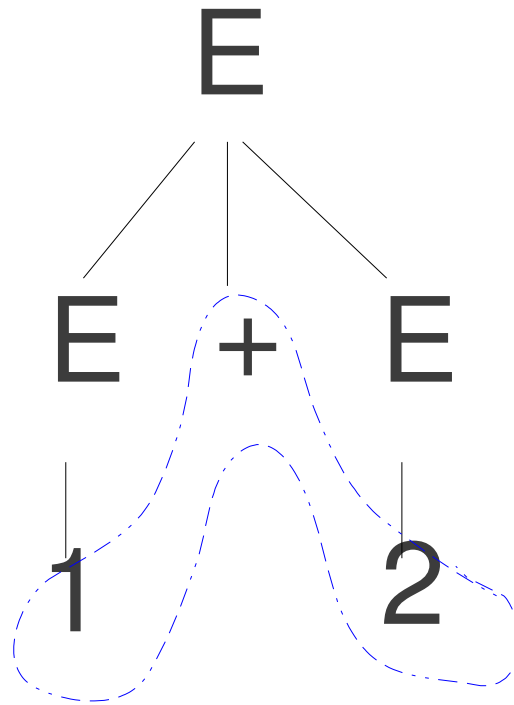
# Top-down Parsing

Input : 1+2



# Bottom-up Parsing

Input : 1 + 2





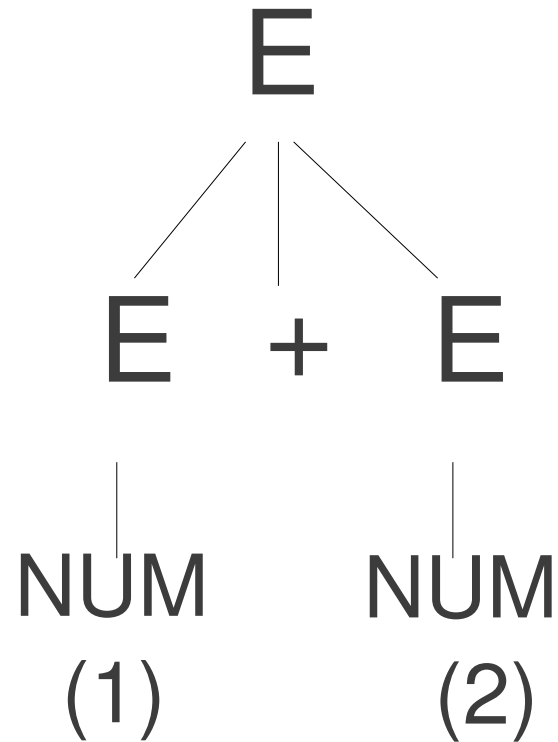
$$E \rightarrow E + E$$
$$E \rightarrow \text{NUM}$$

**NUM** is a token returned by the Lexical Analyzer, for an input matching **[0-9]+**

# Parse Tree

Input : 1+2

Tokens: NUM, +, NUM



# Bottom up parsing – Implementation using a stack

Input: 1 + 2

Tokens: NUM + NUM

Stack

Input

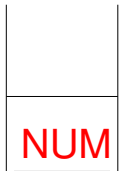
Action



NUM + NUM



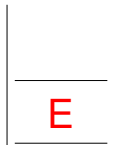
Stack Empty, **shift** NUM



NUM + NUM



NUM on stack, **reduce** using the production  $E \rightarrow \text{NUM}$



NUM + NUM



**shift** +

# Bottom up parsing – Implementation using a stack

Input: 1 + 2

Tokens: NUM + NUM

Stack

Input

Action

+
E

NUM + NUM

shift NUM

NUM
+
E

NUM + NUM

NUM on stack, reduce using the production  $E \rightarrow \text{NUM}$

E
+
E

NUM + NUM

E+E on stack, reduce using the production  $E \rightarrow E + E$

# Bottom up parsing – Implementation using a stack

Input: 1 + 2

Tokens: NUM + NUM

Stack

Input

Action



NUM + NUM



E on stack, no more input tokens,  
**accept** the string

**Action of the parser on input 1+2+ ?**

# Shift / reduce parser

A sequence of **Shift / Reduce** actions and finally **accept/reject**

YACC generates **LALR** parser, a kind of Shift/reduce parser

Given grammar rules:

$S \rightarrow E$

$E \rightarrow E + E$

$E \rightarrow \text{NUM}$

## Yacc Specification

%%

pgm : expr { ... }  
;

expr : expr '+' expr { .....

expr : NUM { .....

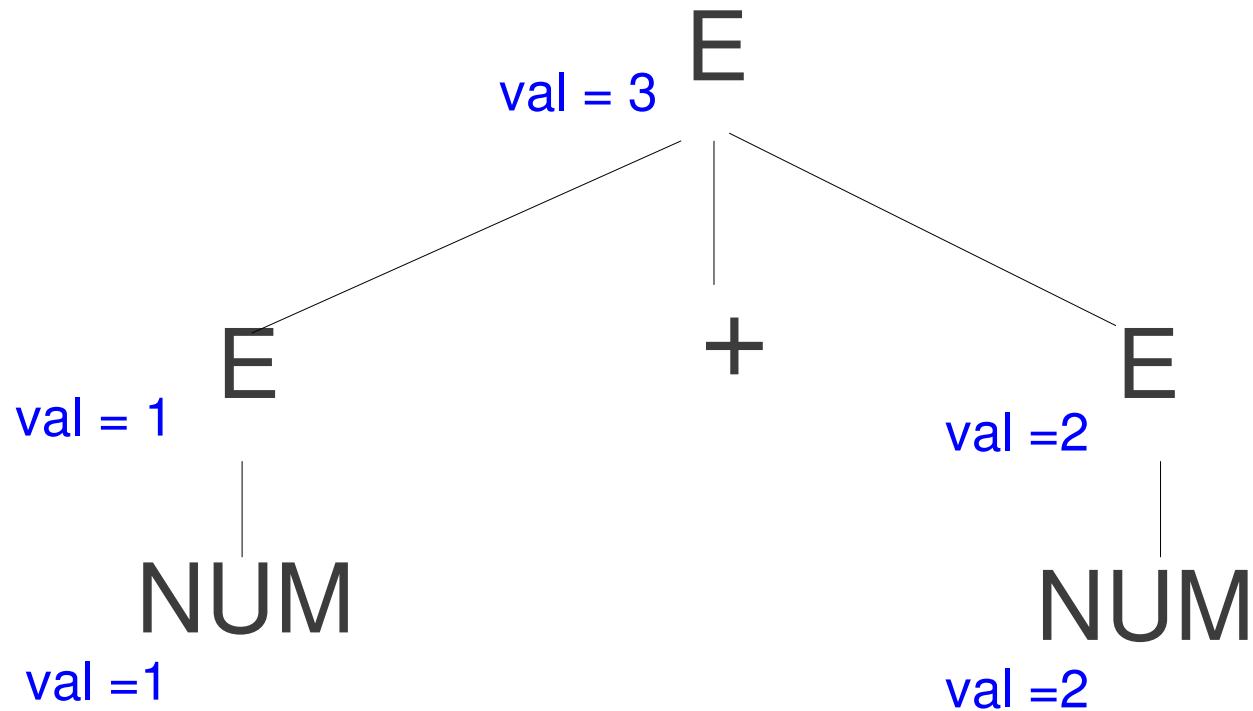
%%

# Calculator: Evaluation



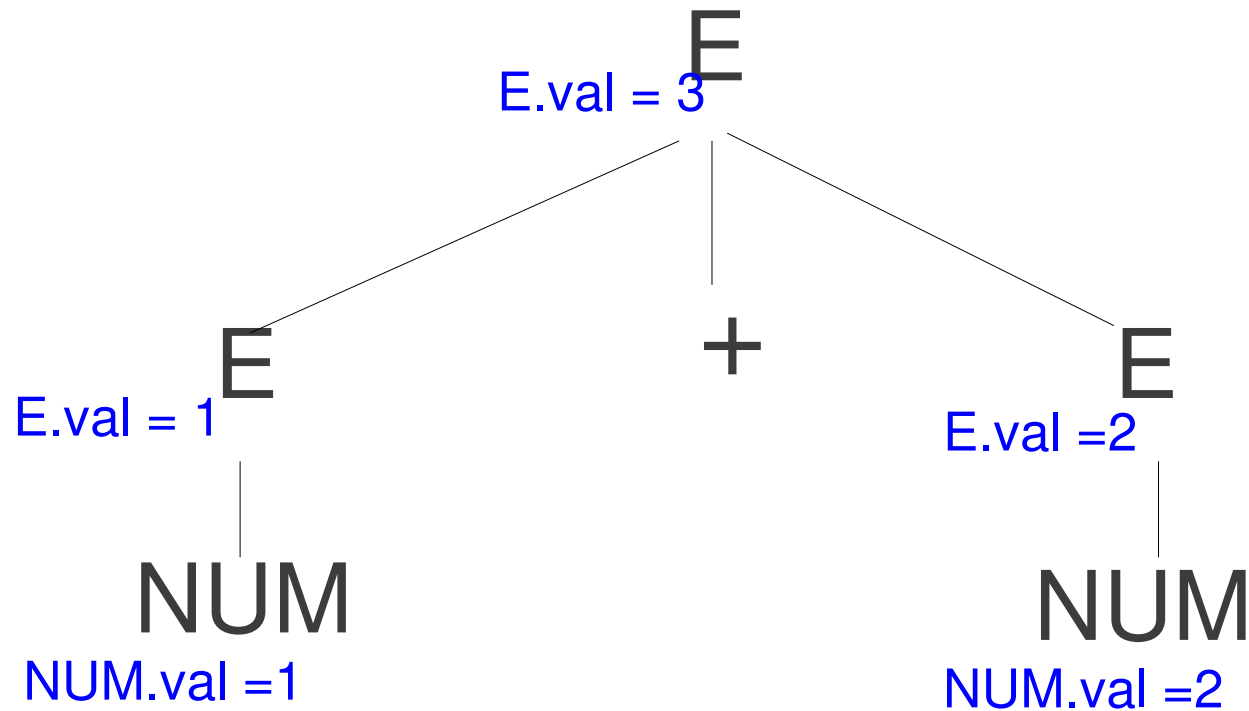
# Evaluation during parsing

Input : 1 + 2



# Evaluation during parsing

Input : 1 + 2



# Syntax Directed Definition

- Associate an *attribute* with each grammar symbol
  - **NUM** - associated numeric value, *lexval*, denoted *NUM.lexval*
  - **E** - an attribute, *val*, denoted *E.val*

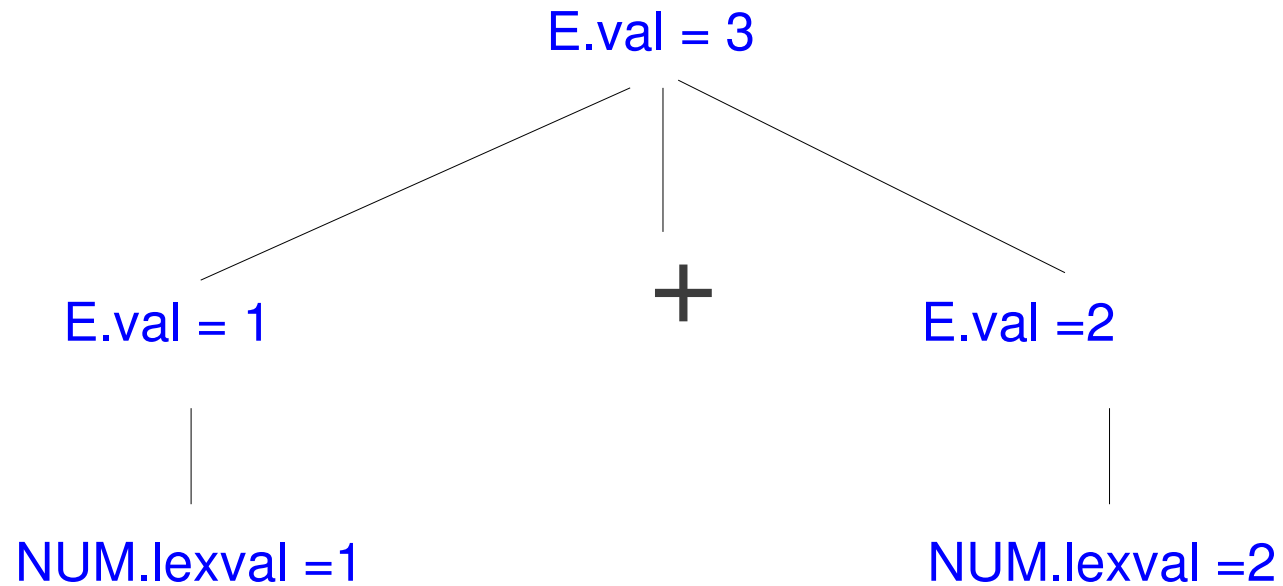
$E \rightarrow E1 + E2 \quad \{ E.val = E1.val + E2.val \}$   
 $E \rightarrow NUM \quad \{ E.val = NUM.lexval \}$

## Semantic actions with each production

- Computes the attribute value
  - Action taken when the parser reduces using the corresponding production

# Annotated parse tree

Input : 1 + 2



# Evaluation during parsing: Implementation

- Parser maintains a value stack
  - Lexical Analyzer sets *yylval*
  - the value put on top of the value stack
- Type of value stack
  - default type integer

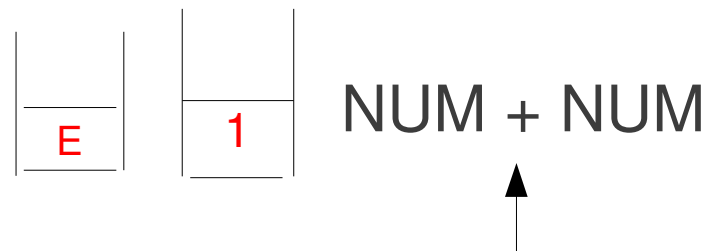
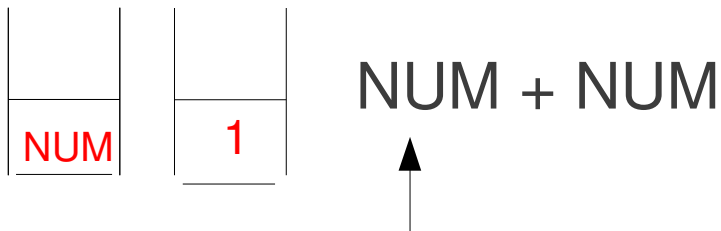
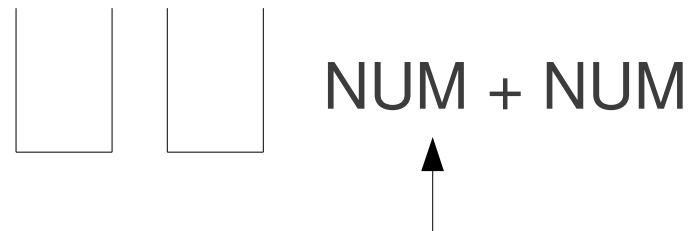
# Bottom up parsing – Attribute evaluation

Input: 1 + 2

Tokens: NUM + NUM

Parser Stack

Value Stack



Action

Stack Empty, **shift** NUM

NUM on stack, 1 on value stack  
**reduce** using the production **E-->**  
**NUM**

**shift** +

# Bottom up parsing – Attribute evaluation

Input: 1 + 2

Tokens: NUM + NUM

Stack

Input

Action

+
E

1

NUM + NUM

shift NUM



NUM
+
E

2
1

NUM + NUM

NUM on stack, 2 on value stack  
reduce (production  $E \rightarrow \text{NUM}$ )





# Bottom up parsing – Attribute evaluation

Input: 1 + 2

Tokens: NUM + NUM

Stack

Input

Action

E
+
E

2
1

NUM + NUM



E+E on stack,  
**reduce** (production  $E \rightarrow E + E$ )  
compute the attribute value

E

3

NUM + NUM



E on stack, value 3 on value stack

$E \rightarrow E1 + E2 \quad \{ E.val = E1.val + E2.val \}$   
 $E \rightarrow NUM \quad \{ E.val = NUM.lexval \}$

In YACC

$E \rightarrow E1 + E2 \quad \{ \$\$ = \$1 + \$3 \}$   
 $E \rightarrow NUM \quad \{ \$\$ = \$1 \}$

$\$i$  : attribute value of grammar symbol  $i$  on the right hand side.

$\$$  : attribute value of grammar symbol on the left hand side.

*By default, each attribute is of type integer*

```
//Yacc file, calcul.y
```

```
%{
```

```
#include "lex.yy.c"
```

```
%}
```

```
%token  NUM
```

```
%%
```

```
pgm    : expr '\n'    {printf("%d\n", $1);  exit(0);};
```

```
expr   :   expr '+' expr    {$$= $1+$3;};
```

```
expr   :   NUM    {$$=$1;};
```

```
%%
```

```
int main(void){
```

```
    return yyparse();}
```

# Building the calculator

```
lex calcu.l
```

```
yacc -d calcu.y
```

```
cc y.tab.c -ll
```

```
./a.out
```