Compiler Lab

Introduction to tools
Lex and Yacc

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
- Meaningful lexical units

  12   +   34   *   6

- Syntactically valid

  12+34* is not syntactically valid

- Evaluate ?

Calculator

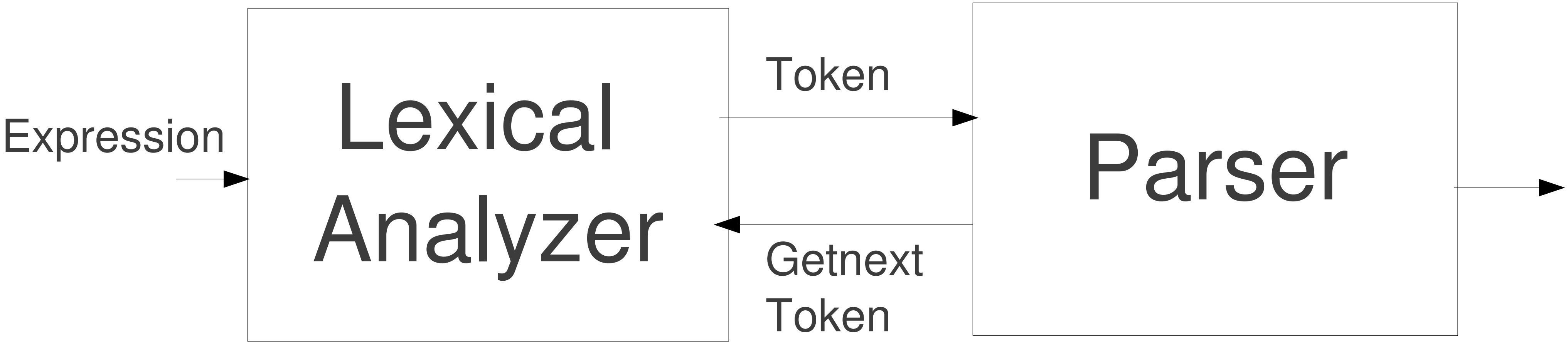
- Input: 12 + 34 * 6
- Lexical units

  12   +   34   *   6

- Separate into tokens :
  - <NUM, 12>   <ARITHOP, +>   <NUM, 34>   <ARITHOP, *>   <NUM, 6>
  - Use a Lexical Analyzer
    - Patterns for various tokens

Calculator

- Input: 12 + 34 * 6
- Lexical Analyzer gives tokens : <NUM, 12> <ARITHOP, +> <NUM, 34> <ARITHOP, *> <NUM, 6>
- Check the syntax (12+34* is not syntactically valid)
  - Use Syntax Analyzer (parser)
    - Grammar rules
- Evaluate ?



Expression

Lexical Analyzer

Token

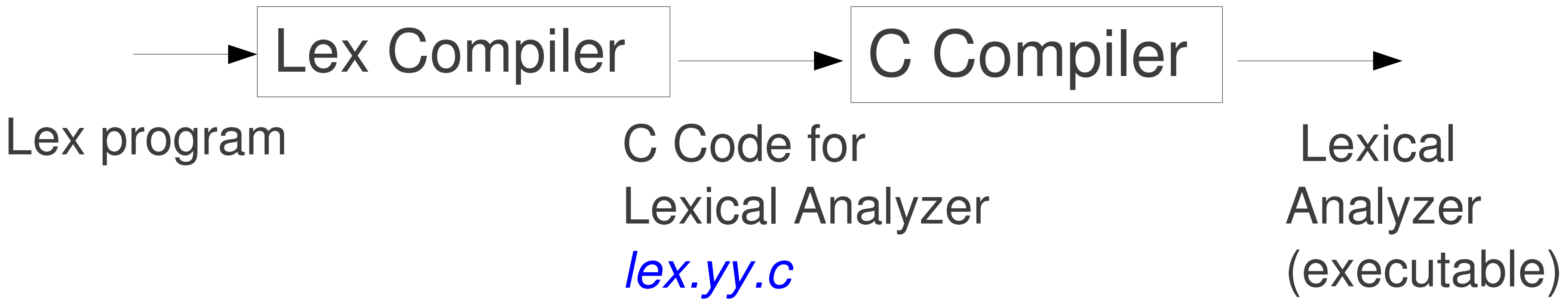
Getnext Token

Parser

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 --> E+E
E --> 0 | 1 | 2

Some strings in the language

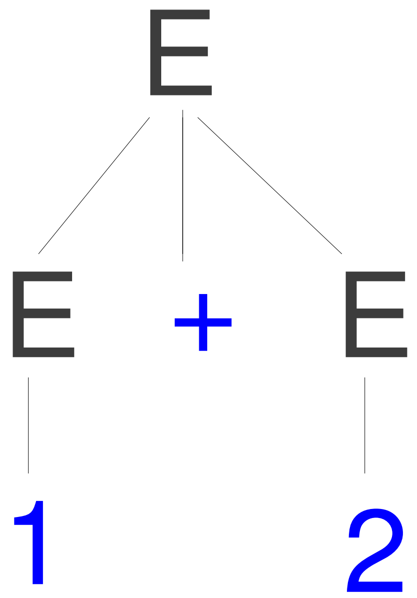
1+2   1+0+2   2+2+1+0

Some strings not in the language

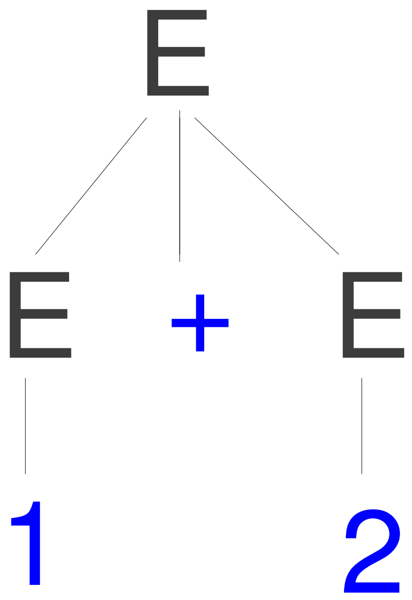
1+   +2+3   1++   12

Parse Tree

Input : 1+2



Parse Tree



Parse Tree for

1+2+1

1+

+2+3

Syntactic validity

E --> E+E

E --> 0 | 1 | 2

Valid strings: 1+2, 1+0+2, 2+2+1+0 ...

Can build a parse tree for any of these strings

Invalid strings:1+, +2+3, 1++, 12

Parse tree construction fails

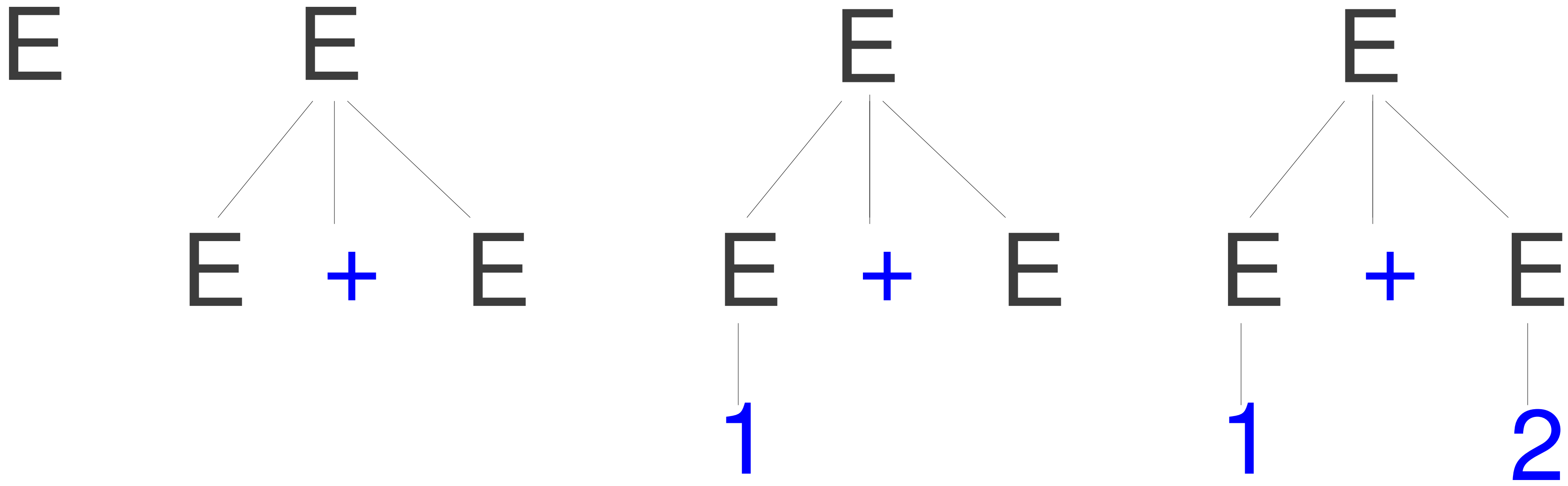
Parsing Algorithm

Input: Grammar, string to be checked
Output: Yes / No (string is syntactically valid or not)

- Parser simulates the steps in building a parse tree for the given string
- Two approaches – Top-down, Bottom-up

Top-down Parsing

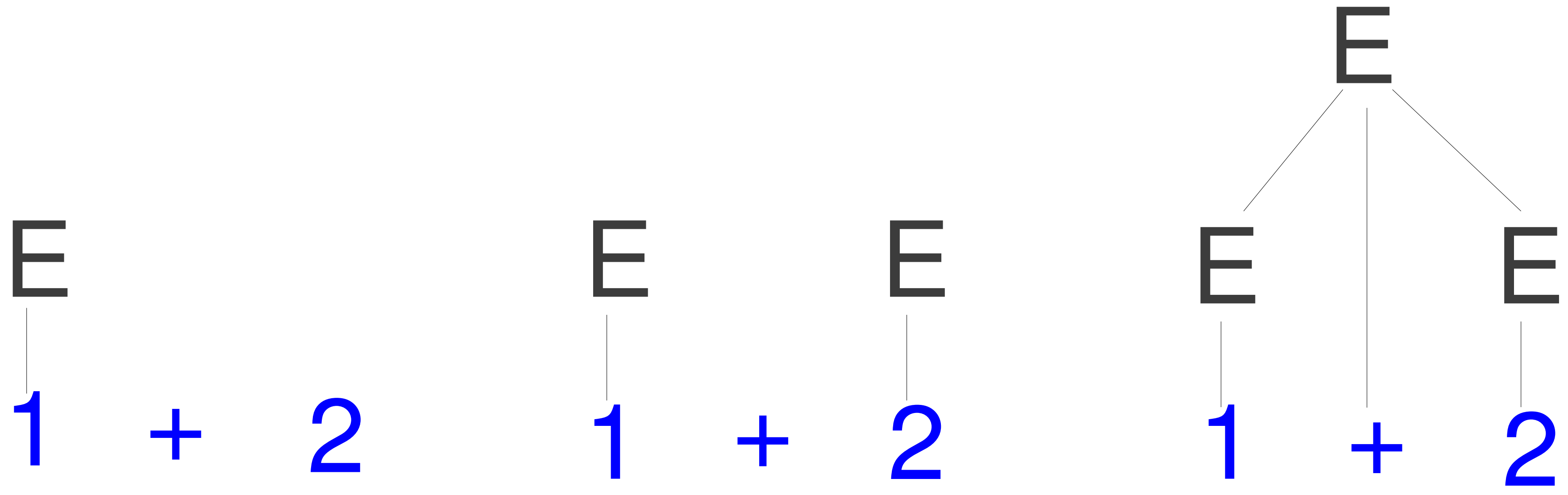
Input : 1+2



Builds a parse tree top down, starting from the root

Bottom-up Parsing

Input : 1+2



Builds a parse tree bottom-up, starting from the leaves
A sequence of reductions

Example Grammar

E --> E + E

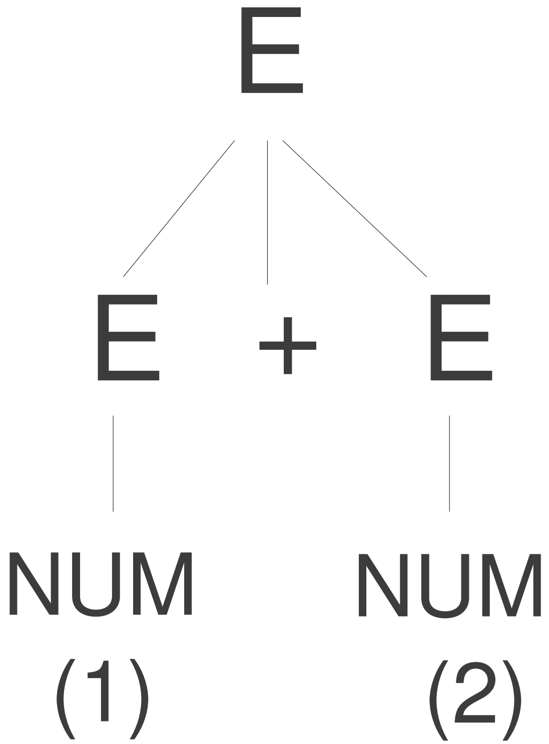
E --> NUM

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

Parse Tree

Input : 1+2

Tokens: NUM, +, NUM



E

E + E

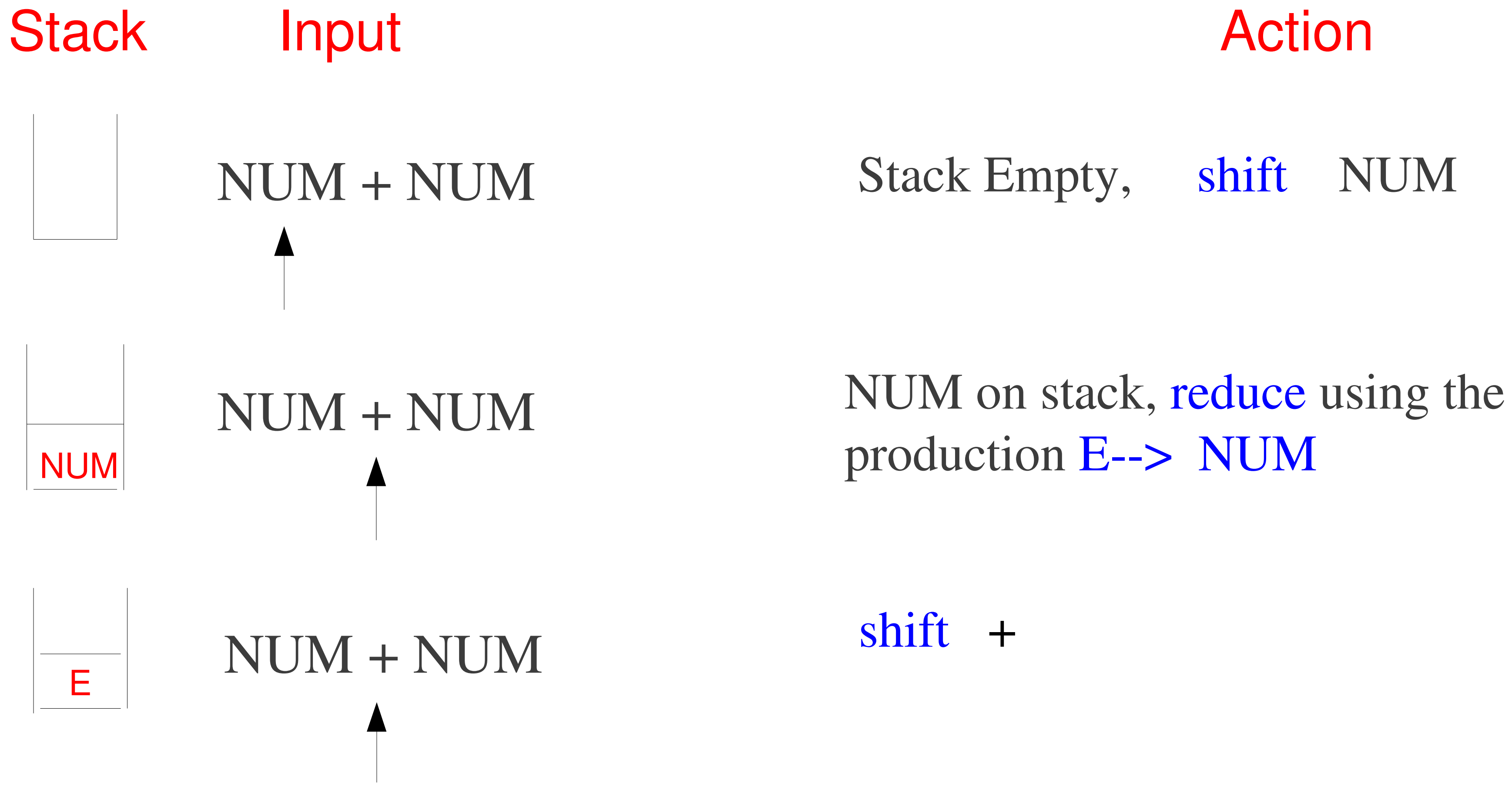
NUM (1)

NUM (2)

Bottom up parsing – Implementation using a stack

Input: 1 + 2

Tokens: NUM + NUM



Stack Input Action

NUM + NUM — Stack Empty, shift NUM

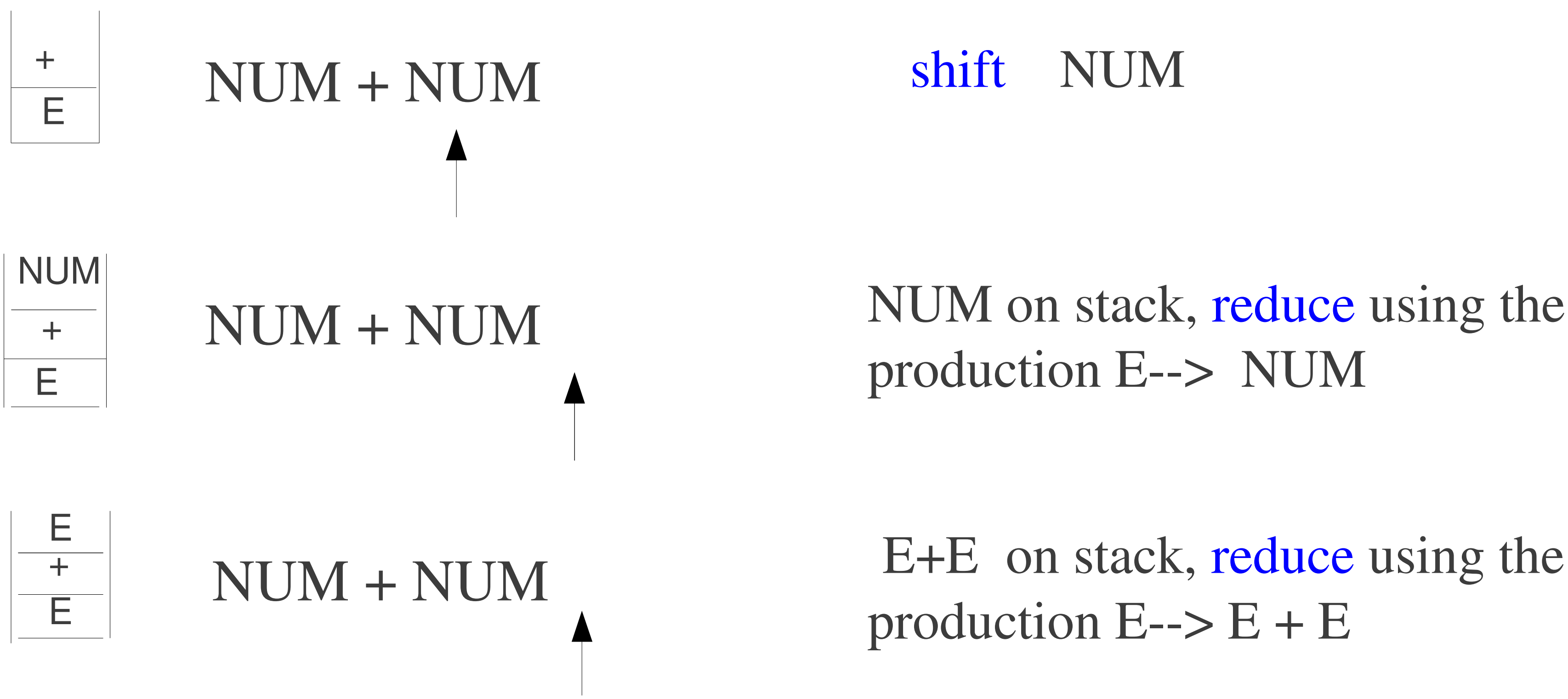
NUM | NUM + NUM — NUM on stack, reduce using the production E--> NUM

E | NUM + NUM — shift +

Bottom up parsing – Implementation using a stack

Input: 1 + 2

Tokens: NUM + NUM



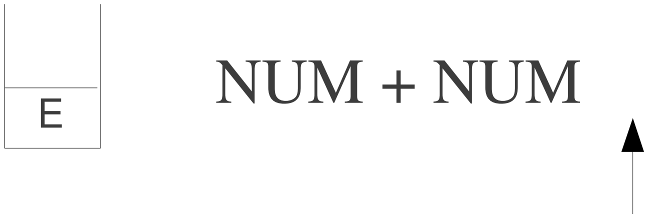
Stack	Input	Action
+ E	NUM + NUM	shift NUM
NUM + E	NUM + NUM	NUM on stack, reduce using the production E--> NUM
E + E	NUM + NUM	E+E on stack, reduce using the production E--> E + E

Bottom up parsing – Implementation using a stack

Input: 1 + 2

Tokens: NUM + NUM

Stack Input Action



E

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 -> E
E -> E + E
E -> NUM
```

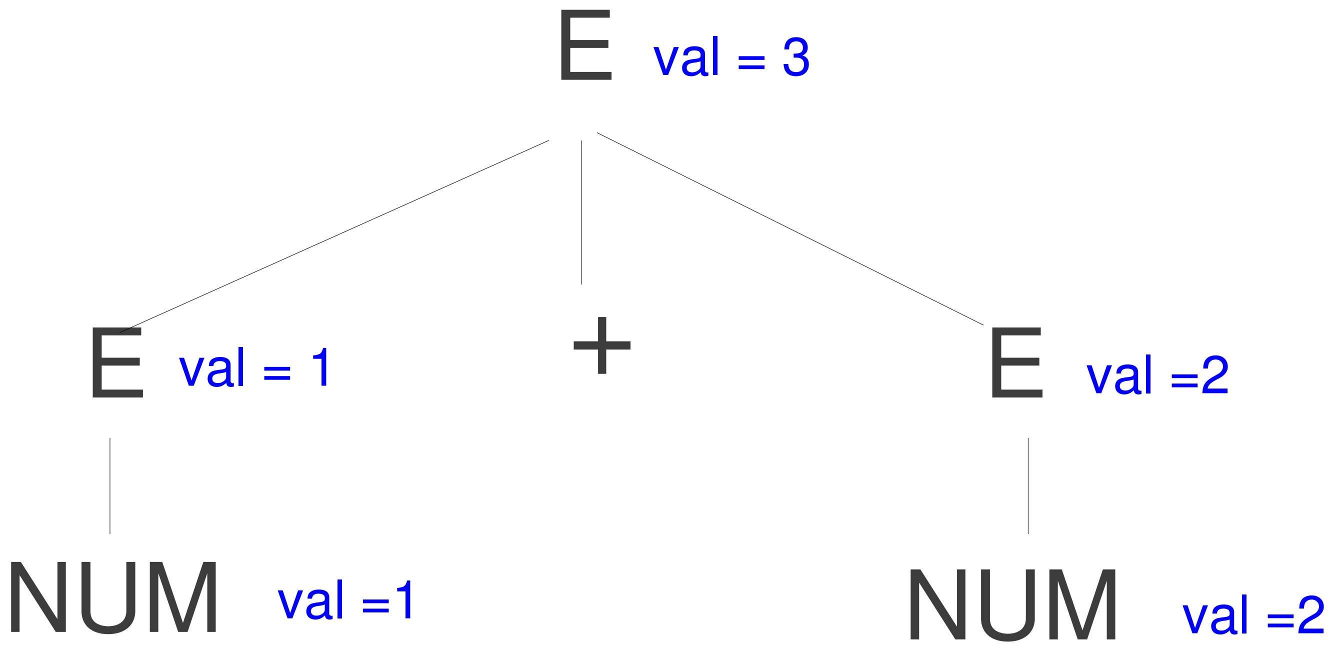
Yacc Specification

```
%%
pgm  : expr        { ... }
     ;
expr :   expr '+' expr      {.....}
     ;
expr :   NUM          {.....}
     ;
%%
```

Calculator: Evaluation

Evaluation during parsing

Input : 1 + 2



E val = 3

E val = 1

+

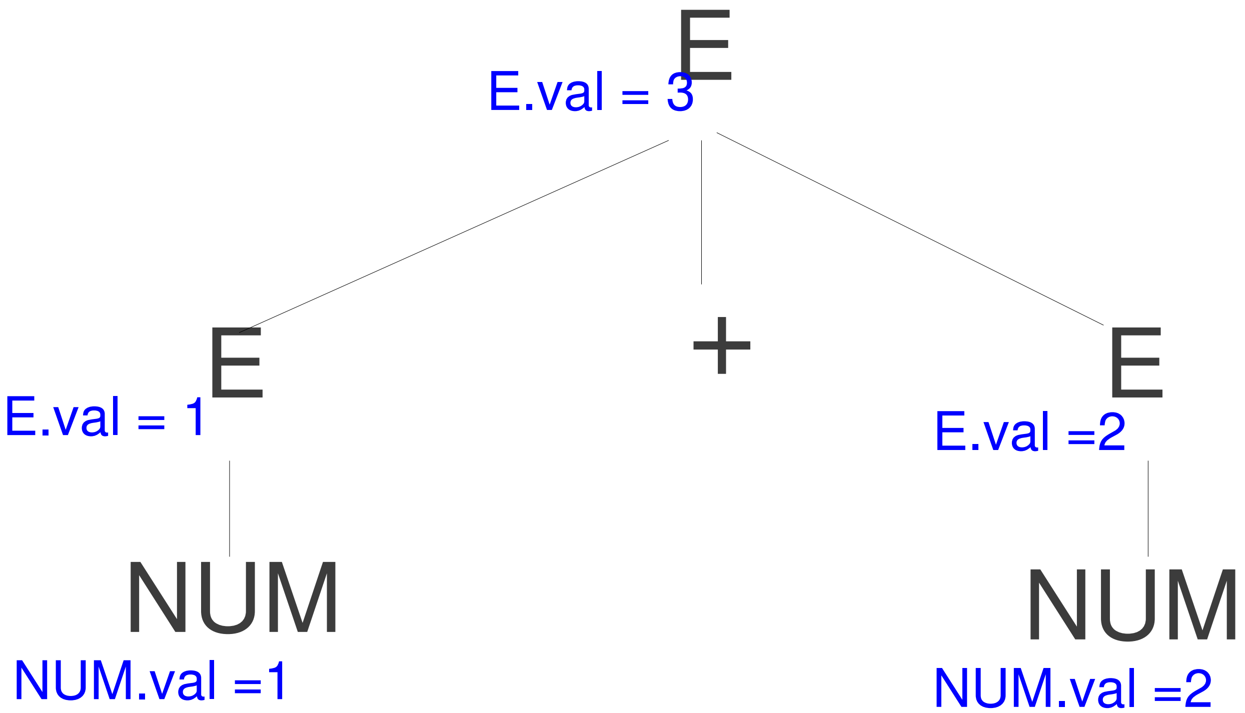
E val =2

NUM val =1

NUM val =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 --> E1 + E2 { E.val = E1.val + E2.val }

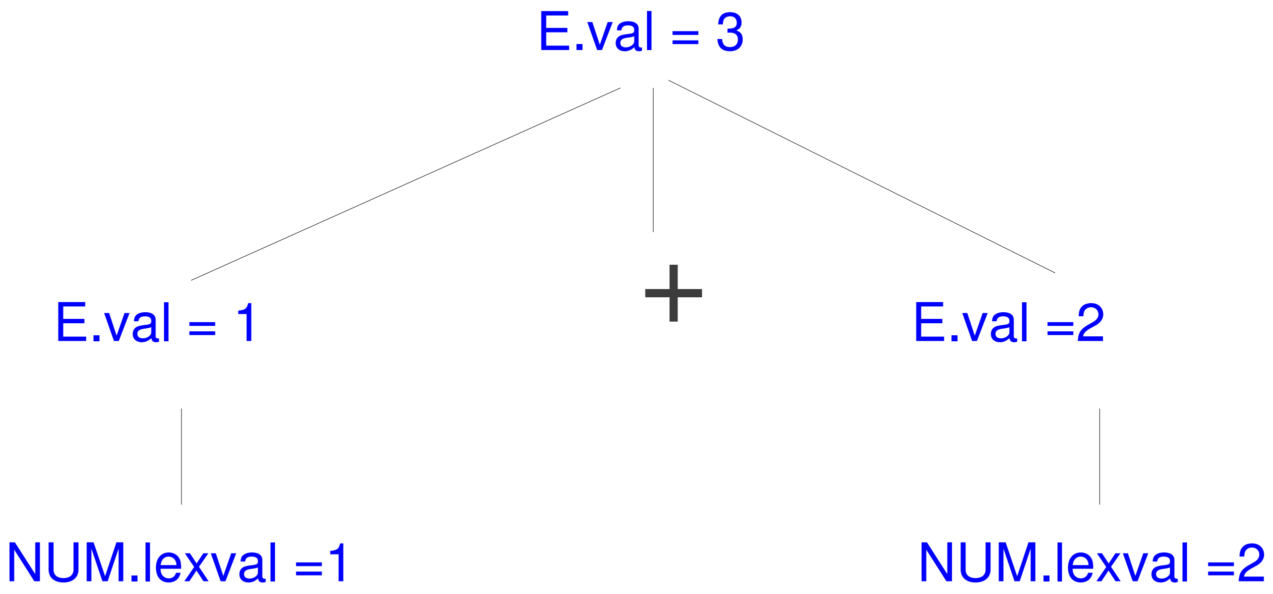
E --> NUM { E.val = NUM . lexval }

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



E.val = 3

E.val = 1

+

E.val =2

NUM.lexval =1

NUM.lexval =2

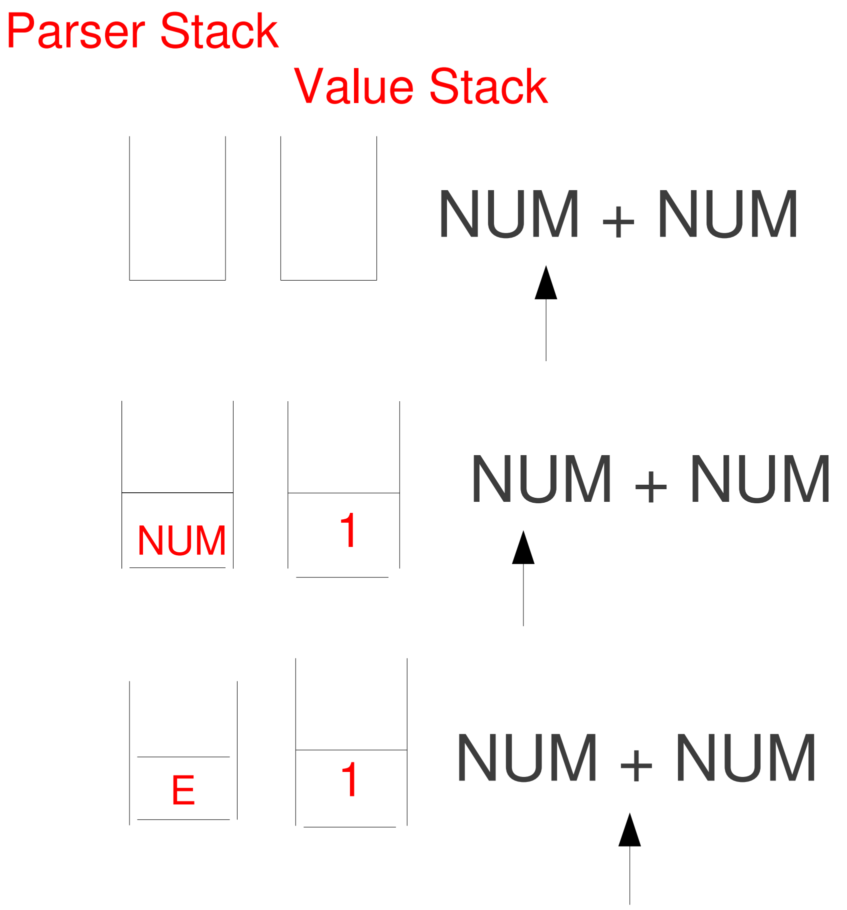
Evaluation during parsing: Implementation

- 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

NUM + NUM

Stack Empty, shift NUM

NUM 1 NUM + NUM

NUM on stack, 1 on value stack

reduce using the production E--> NUM

E 1 NUM + NUM

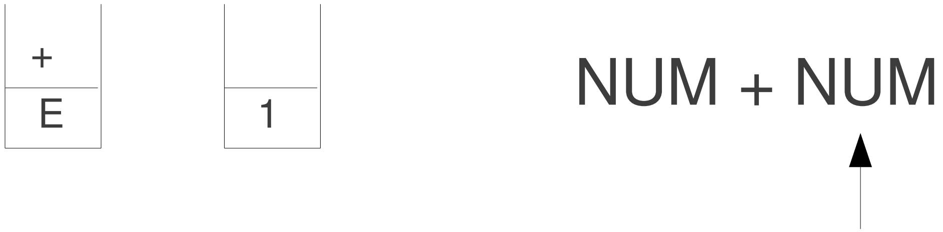
shift +

Bottom up parsing – Attribute evaluation

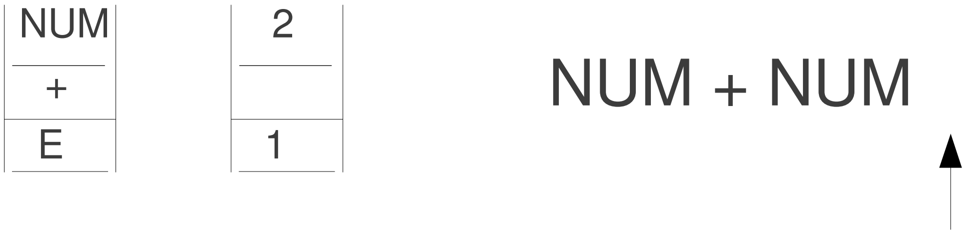
Input: 1 + 2

Tokens: NUM + NUM

Stack Input Action



shift NUM



NUM on stack, 2 on value stack

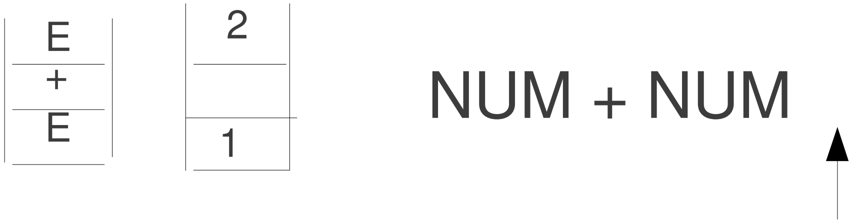
reduce (production E-->NUM)

Bottom up parsing – Attribute evaluation

Input: 1 + 2

Tokens: NUM + NUM

Stack Input Action



E

+

E

2

1

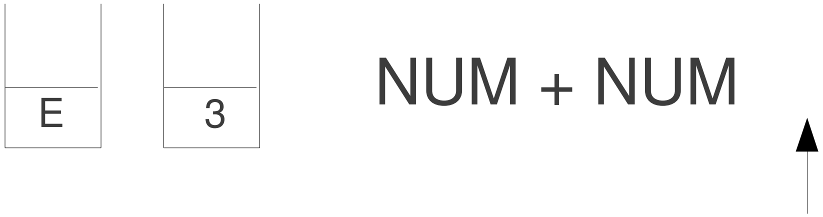
NUM + NUM

E+E on stack,

reduce (production E--> E + E)

compute the attribute value

E.val = E1.val + E2.val



E

3

NUM + NUM

E on stack, value 3 on value stack

```
E -->  E1  +  E2     {  E.val = E1.val + E2.val  }
E -->  NUM            {  E.val = NUM . lexval  }
```

In YACC

```
E -->  E1  +  E2     {  $$ = $1 + $3  }
E -->  NUM            {  $$ = $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,  calcu.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
```

Building an expression tree

Semantic actions to build an expression tree

E --> E1 + E2 { E.ptr = *mkNode* (+, E1.ptr, E2.ptr) }

E --> NUM { E.ptr = *mkLeafNode* (NUM.lexval) }

YACC code

```
expr  :   expr '+' expr {$$=mkNode('+', $1, $3);}
    ;
expr  :   NUM {$$=mkLeafNode($1);}
```

Expression Tree

```
%union{      //defines YYSTYPE
    int   ival;
    struct tree_node *nptr;
};

%token    <ival> NUM
%type     <nptr> expr
```

```
%{
   ……
#include "exprtree.h"
   ……
%}

%union{
        int ival;
        struct tree_node *nptr;
};

%token  <ival> NUM
%type   <nptr> expr

%%
pgm:      expr '\n' {printf("%d\n", evaluate($1));}
        ;
expr:     expr '+' expr {$$=mkOperatorNode('+', $1, $3);}
        ;
expr:     NUM {$$=mkLeafNode($1);} ;
%%
```