Syntax

The study of the rules whereby words or other elements of sentence structure are combined to form grammatical sentences.

The American Heritage Dictionary

Syntactic analysis

- The purpose of the syntactic analysis is to determine the structure of the input text;
- The syntactic structure is defined by a grammar.

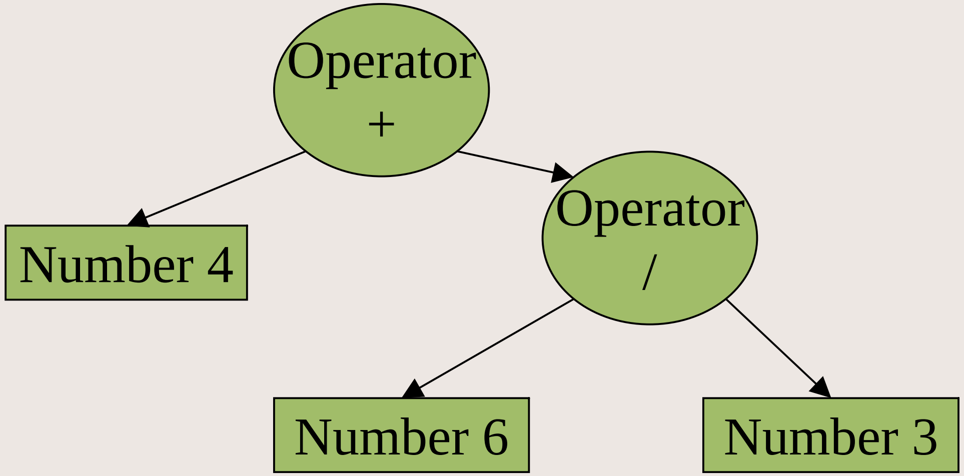
Example of syntactic analysis

`4+12/3`

Lexical Analysis

(Number 4) (Operator +) (Number 12) (Operator /) (Number 3)

Syntactic Analysis



Operator
+

Number 4

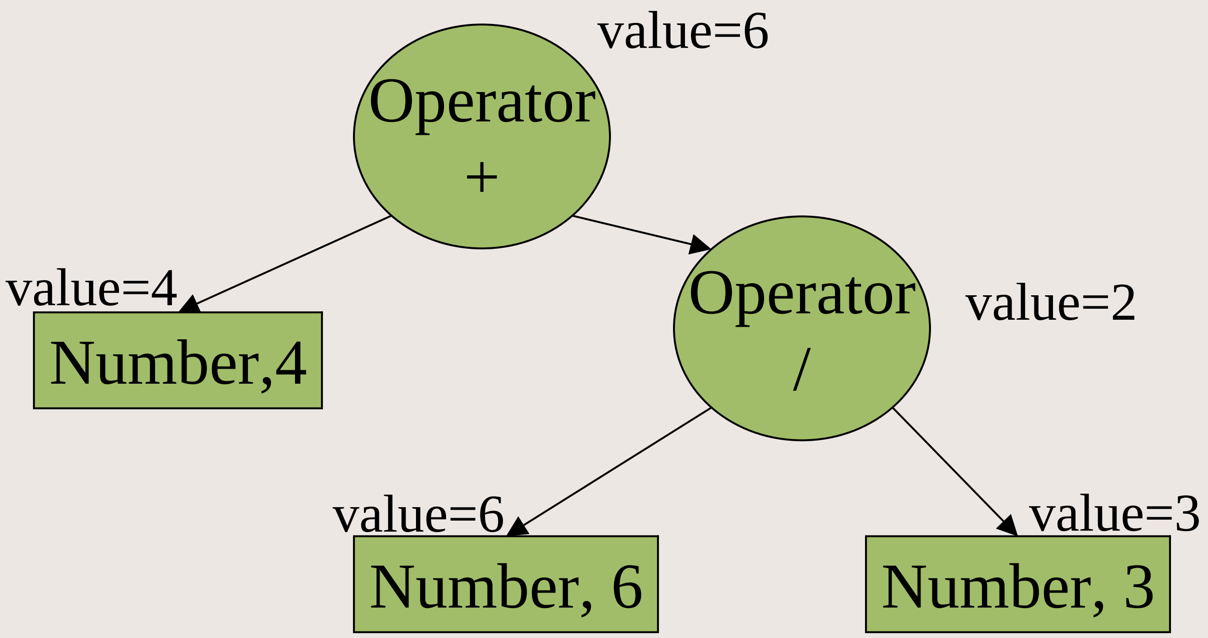
Operator
/

Number 6

Number 3

Semantic analysis

It is the evaluation of the meaning of each (terminal and non-terminal) symbol, achieved by evaluating the semantic attributes either in ascending or descending order.



What is a parser

- A parser is a program that performs syntactic analysis.
- It can typically be:
  - LL (left to right-leftmost); or
  - LR (left to right-rightmost).
- LL parsers can be constructed by hand or automatically.
- LR parsers are usually too complex to be constructed manually.

`bison`: a parser generator

- **`bison`** is a free implementation of **`yacc`** (originally by AT&T) that comes standard with most Unix distributions; **`yacc`** is the absolute standard compiler compiler;
- Learn more on **`bison`** at the following address:

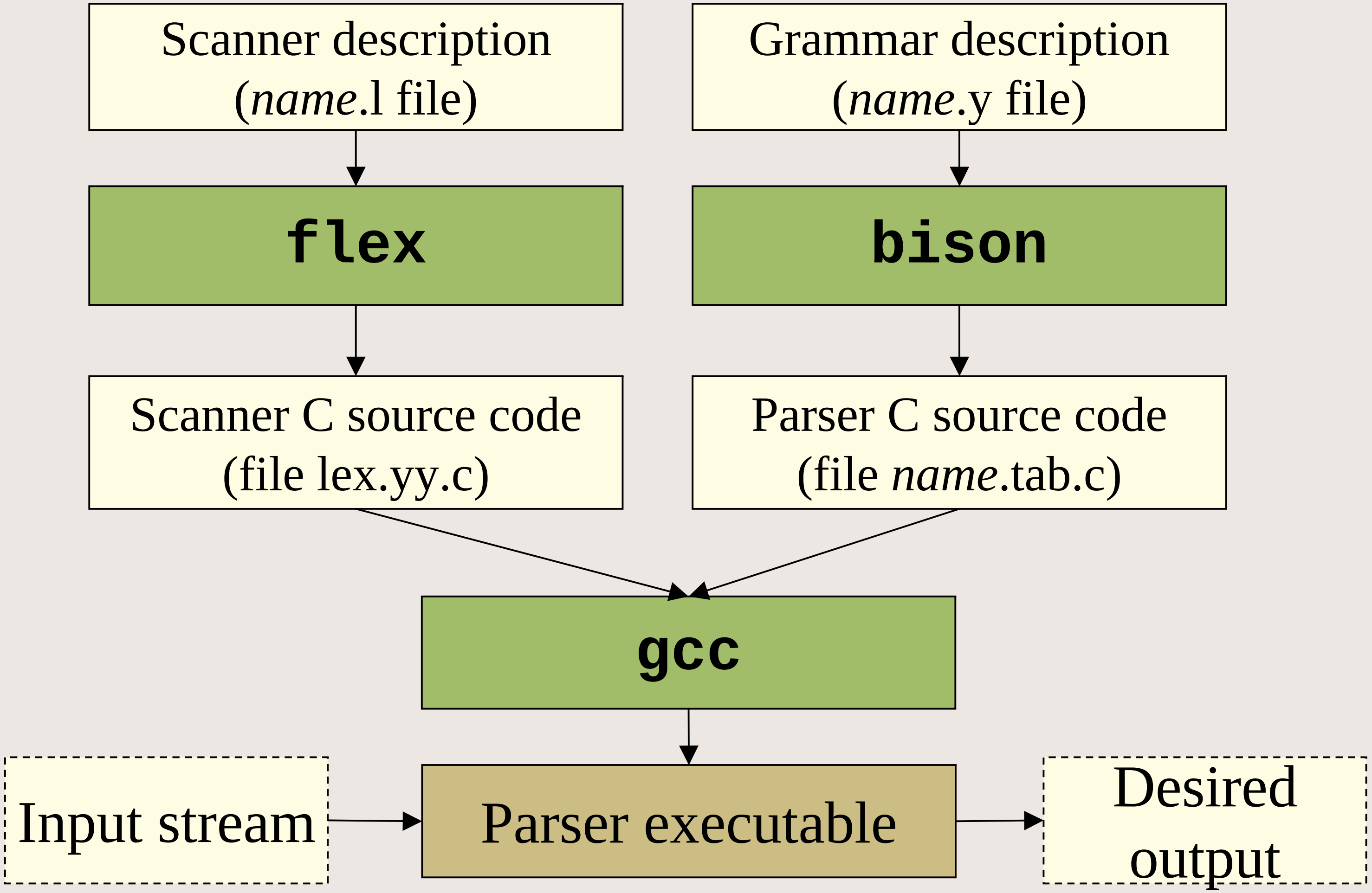
  www.gnu.org/software/flex/flex.html
- **`bison`** is free, and distributed under the terms of GNU General Public License (GPL).
- A useful book to understand **`bison`** is:

  John Levine, Tony Mason & Doug Brown

  lex & yacc, 2nd Edition

  O'Reilly

Designing a parser with **`bison`** and **`flex`**



Scanner description
(*name*.l file)

`flex`

Scanner C source code
(file lex.yy.c)

Grammar description
(*name*.y file)

`bison`

Parser C source code
(file *name*.tab.c)

`gcc`

Input stream

Parser executable

Desired output

5 easy steps to build a parser

- Specify the tokenizer in **`flex`** format.
- Specify the grammar in **`bison`** format.
- Write the desired semantic actions associated to each syntax rule.
- Write the controlling function.
- Write the error-reporting function.

The format of **bison** grammars

```
%{
    C definitions
%}
    bison definitions
%%
    Grammar Rules
%%
    C user code
```

Comments enclosed in /* */ may appear in any of the sections.

A first example

A Reverse Polish Notation calculator.

Grammar Rules:

S → S E | epsilon

E → number

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

An RPN Calculator in **`bison`**

Definitions

```
%{
#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

%%
```

Grammar Rules

```
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    { $$ = $1 / $2;}
       /* Exponentiation */
       | exp exp OP_EXP    { $$ = pow ($1, $2);}
       /* Unary minus    */
       | exp UN_MINUS      { $$ = -$1;    }
       ;
%%
```

Driver and error routines

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

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

Bison definitions and grammar rules

- Hot to define a token (a terminal symbol):

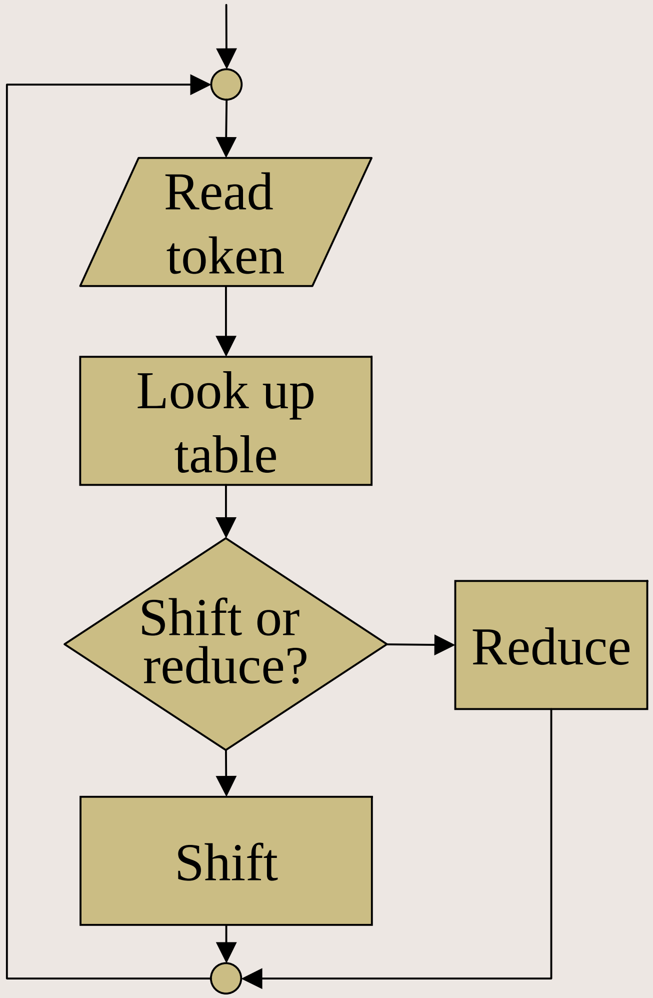
```
%token TOKEN_NAME
```

- How to define a grammar rule:

```
S : A1 … An { semantic action }
  | B1 … Bm { semantic action }
  ;
```

- How semantic actions are specified, and values treated:
  - The semantic value of the non-terminal in the left-hand side of the production is referred as **`$$`**
  - The semantic values of the symbols in the right-hand side are referred as **`$1……$n`**
  - The default semantic action is **`{ $$ = $1; }`**

How the generated parser works



Read token

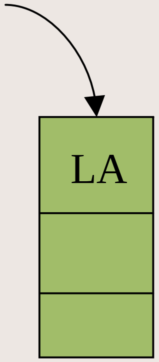
Look up table

Shift or reduce?

Reduce

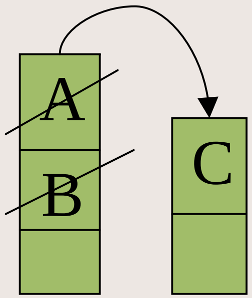
Shift

Shift



The current look-ahead symbol (LA) is pushed on top of the stack.

Reduce



rule:

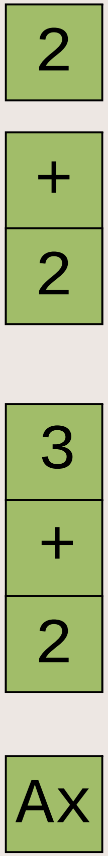
`C : B A ;`

Symbols constituting the right-hand side of a rule (in reverse order) are recognized. They are popped, and the corresponding left-hand side is pushed.

The trace of a parser execution

- Input tokens: **2 + 3**
- LA = **2**
- Shift
- LA = **+**
- Shift
- LA = **3**
- Shift
- LA = *<end of input>*
- Reduce
- Stop

Stack States



Integration with **`flex`**

- Compile the parser source with **`–d`** option.
- **`bison`** outputs a file named ***`name`*****`.tab.h`**, which contains the token definitions and the type definition for return values.
- The above file should be included in the **`flex`** input; **`YYSTYPE`** should be defined (the type of tokens' semantic values).
- The lexical actions must store the semantic value of each token in the global **`yylval`** variable (declared in generated header file).

Example of integration

rpn.tab.h

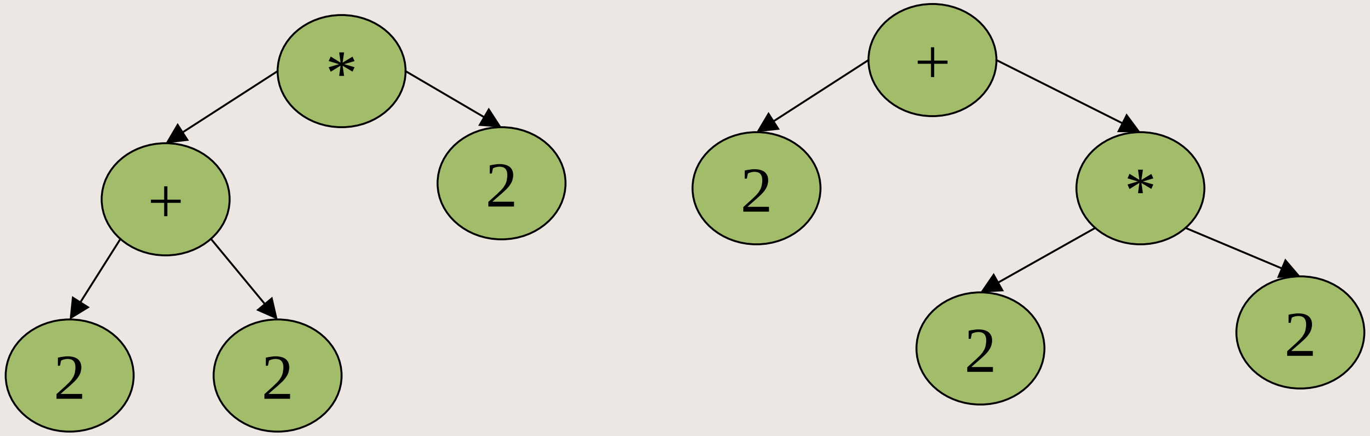
```
#ifndef YYSTYPE
#define YYSTYPE int
#endif

#define  NUM      257
#define  OP_PLUS  258
#define  OP_MINUS 259
#define  OP_MUL   260
#define  OP_DIV   261
#define  OP_EXP   262
#define  UN_MINUS 263
#define  NEWLINE  264

extern YYSTYPE yylval;
```

rpn.lex

```
%{
#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;}
```

An infix notation calculator

- Grammar rules:
  S → (S)S | S+S | S-S | S*S | S/S | S^S | -S
  S → number
- This grammar is ambiguous: there are sentences which can be derived in multiple ways, e.g. 2+2*2.



How to resolve ambiguity

- Either rewrite the grammar in a non-ambiguous form:

```
S → S + E | S – E | E
E → E / M | E * M | M
M → T ^ M | - T | T
T → number | (S)
```

- or use operator precedence declarations provided by **bison**

Infix notation calculator in **bison**

Definitions

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

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

Grammar Rules

```
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         { $$ = $1 / $3;    }
       /* Unary minus    */
       | OP_MINUS exp %prec NEG { $$ = -$2;        }
       /* Exponentiation */
       | exp OP_EXP exp         { $$ = pow($1,$3); }

       | LP exp RP              { $$ = $2         }
       ;
%%
```

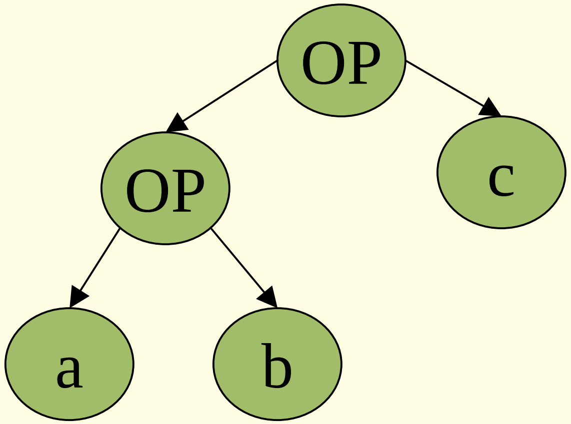
Driver and Error routines

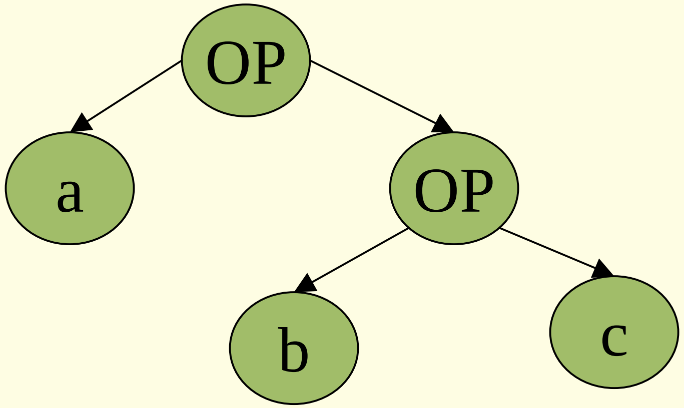
```
int yyerror(char * s){
  printf("%s\n",s);
}
```

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

Operator associativity

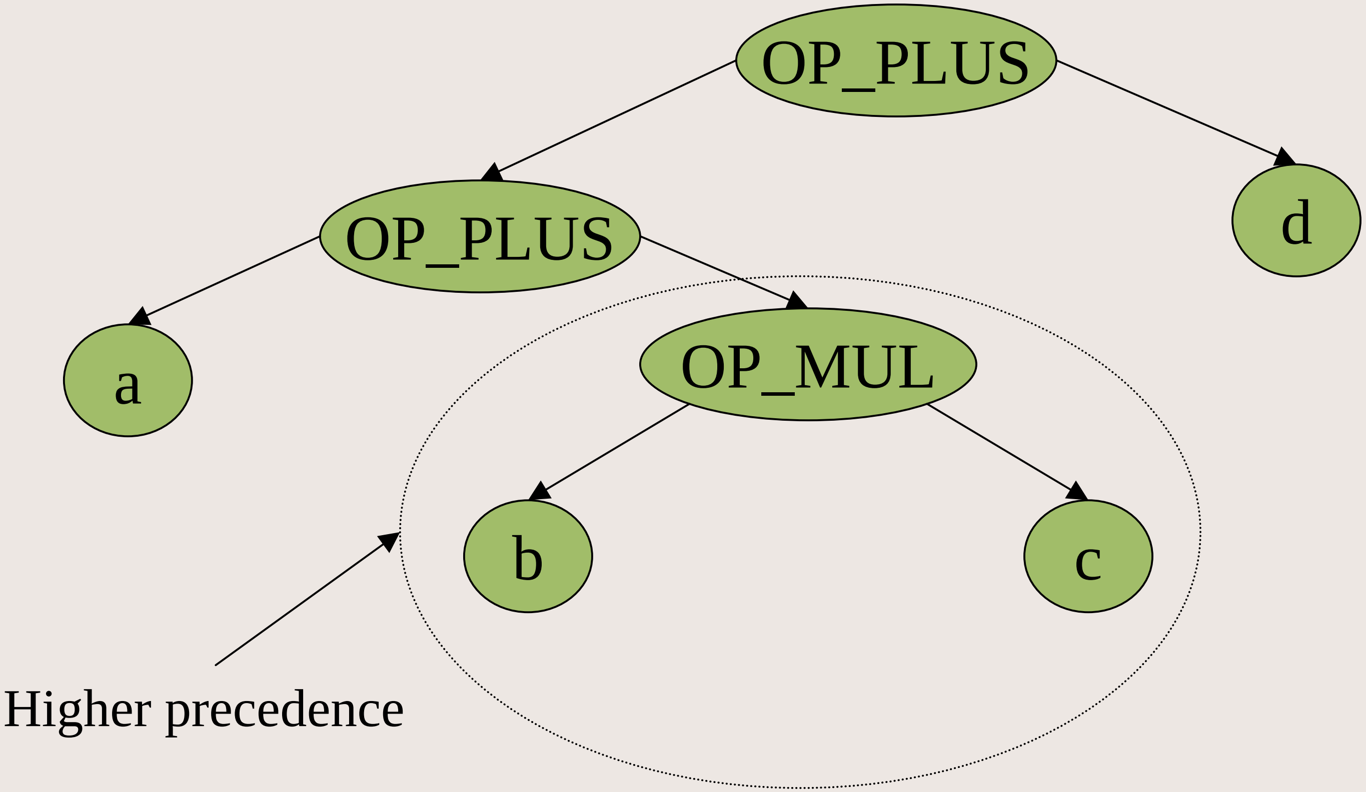
- Consider the following sentence: “a *op* b *op* c”, (where *op* is an operator);
- Should the above expression be interpreted as “(a *op* b) *op* c” or as “a *op* (b *op* c)” ?
- This depends on the operator associativity:


Parsetree if *op* is left-associative


Parsetree if *op* is right-associative

Operator precedence

`a OP_PLUS b OP_MUL c OP_PLUS d`



Operator precedence declarations

Available declaration forms:

- **`%right`** ***`op`***
  specifies right-associativity of operator *op*;
- **`%left`** ***`op`***
  specifies left-associativity of operator *op*;
- **`%nonassoc`** ***`op`***
  specifies no associativity: “a *op* b *op* c” must be considered a syntax error.

Operator precedence

- All the operators declared in the same precedence declaration have equal precedence, and nest together according to their associativity:

  e.g.: `%left OP_PLUS OP_MINUS`

- Operators declared later have the higher precedence and are grouped first:

  e.g.:

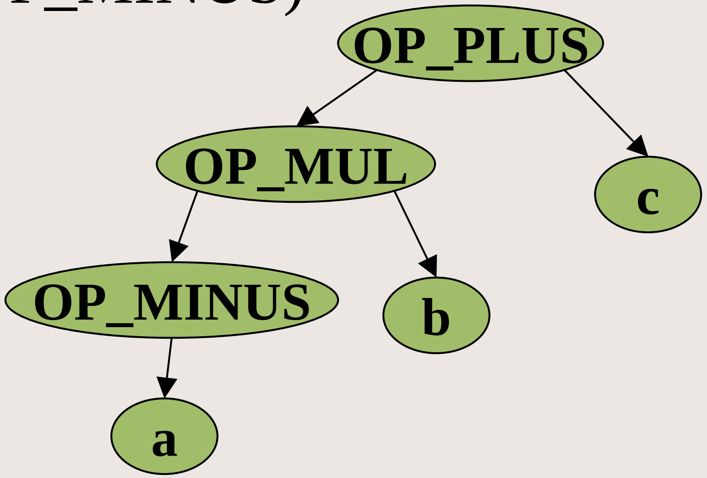
```
%left OP_PLUS OP_MINUS
%left OP_MUL  OP_DIV
```

Context-dependent precedence

- Often, the precedence of an operator depends on the context, e.g. unary minus:

  `OP_MINUS a OP_MUL b OP_MINUS c`

  (the first OP_MINUS has higher precedence than OP_MUL which, in turn, has higher precedence than the second OP_MINUS)



Context-dependent precedence

- Declare a precedence for a fictitious terminal symbol as follows:

```
%left '+' '-'
%left '*' '/'
%left UMINUS
```

- Now the precedence of **`UMINUS`** can be used in specific rules, as follows:

```
expr : ...
     | expr '+' expr
     | ...
     | '-' expr %prec UMINUS
     ;
```

Operator precedence resolution

- First, a precedence is assigned to each declared operator, then each rule containing those operators is assigned the same precedence as the last declared symbol in rule;
- Conflicts are resolved by comparing the precedences of the look-ahead symbol and of the rule.

Operator precedence resolution

- If the look-ahead has the higher precedence, **`bison`** chooses to shift, otherwise to reduce.
- If rule and look ahead have the same level of precedence, **`bison`** makes a choice based on associativity:
  - left means reduce
  - right means shift

Infix Notation Calculator with variable storage

- Grammar Rules

```
S → (S)S | S+S | S-S | S*S
        | S/S | S^S | -S
S → number | variable
S → variable = S
```

Infix Notation Calculator with variable storage in **`bison`**

Definitions

```
%{
#include <math.h>
#include "calc.h"
%}

%union {
double    val;
symrec  * tptr;
}

%token NEWLINE
%token LP
%token RP
%token <val>  NUM
%token <tptr> VAR
%type  <val>  exp

%right EQ
%left OP_MINUS OP_PLUS
%left OP_MUL OP_DIV
%left NEG
%right OP_EXP
```

Grammar Rules

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

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

exp: NUM                     { $$ = $1;              }
   | VAR                     { $$ = $1->var;         }
   | VAR EQ exp              { $$ = $3; $1->var = $3;}
   | exp OP_PLUS exp         { $$ = $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          { $$ = pow ($1, $3);    }
   | LP exp RP               { $$ = $2;              }
   ;
```

Semantic Values

- Sometimes, more than one semantic value type is needed;
- In bison this is achieved by the directive

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

Semantic Values (2)

- All the terminal and non-terminal symbols can have only one of the possible type for its own semantic value.
- Non terminals:

`%type <field_x> <token>`

- Terminals:

`%token <field_x> <token>`

- All:

`$<field_x>$`

`$<field_x>1`

...

Error recovery

- When a syntactically incorrect input is encountered, two different behaviors are possible:

1. Stop the parsing immediately, notify a syntax error and call **`yyparse()`** again.
2. Try to recover the error and continue the parsing.

Error recovery (2)

- The first solution is more convenient in an interactive parser.
- The second solution is more convenient in a parser which takes a source file as an input.
- Error recovery in **`bison`** is achieved by adding a rule recognizing the special token ‘**`error`**’ and calling function **`yyerrok()`** in the semantic action.

Shift-reduce conflicts

- Suppose our grammar contains the following productions:

```
if-st: IF expr THEN stmt                      (1)
     | IF expr THEN stmt ELSE stmt            (2)
     ;
```

- When LA=**`ELSE`** the parser could:
  - reduce the four symbols (**`IF, expr, THEN, stmt`**) on top of the stack, according to the first rule; or
  - shift the **`ELSE`** symbol on top of the stack;
- This is the classic “dangling else” conflict.

Shift-reduce conflicts (2)

- The ‘reduce’ behavior associated the **`ELSE`** symbol with the outermost **`IF`**.
- The ‘shift’ behavior associates the **`ELSE`** symbol with the innermost **`IF`**
  (this is the default behavior).
- The first case of ‘dangling else’ appears in the specifications of the Algol 60 programming language.

Useful options

- **`YYACCEPT`**

  it pretends that a valid language sentence has been read; it causes **`yyparse()`** to immediately return 0 (success), ignoring the rest of the input;
- **`YYABORT`**

  it causes **`yyparse()`** to immediately return 1 (failure), ignoring the rest of the input;