COMP3031: Grammars and Expressions

Qiong Luo

HKUST

Language Description

A language description has two parts:

- Syntax: defines the sequence of symbols that make up valid programs.
- Semantics: specifies what the programs mean.

Example: Describe what real numbers are?

Informal rules:

- A real has three parts: the integer part, the dot "." symbol, and the fraction part.
- The integer part is a string, including the empty string, of digitals.
- The fraction part is a non-empty string of digitals.
- A digital is a member of { 0,1,2,3,4,5,6,7,8,9}.

Examples of reals: 0.1, 23.6, .45
Examples of non-reals: 23., 12.a, 2363

BNF for reals

Backus-Naur Form (BNF) for reals:

```
   <real-number> ::= <integer-part>.<fraction>
  <integer-part> ::= <empty> | <digit-sequence>
      <fraction> ::= <digit-sequence>
<digit-sequence> ::= <digit> | <digit><digit-sequence>
         <digit> ::= 0|1|2|3|4|5|6|7|8|9
```

BNF is a notation for writing *context-free grammars*.

Context-Free Grammars

A context-free grammar has four parts:

- A set of tokens or terminals (atomic symbols).
  Example: 0,1,2,3,4,5,6,7,8,9, and . in the reals example.
- A set of nonterminals (variables defined by the grammar, denoting compound expressions). Example: <real-number>, <integer-part>.
- A set of rules called production for defining the nonterminals. A production has the following general form:

  ```
  ‘‘nonterminal’’ ::= ‘‘A string of terminals
                      and nonterminals’’
  ```

  Example: <real-number> ::= <integer-part>.<fraction>
- A nonterminal chosen as the starting nonterminal; it represents the main construct of the language that the grammar is designed for (expressions, programs, and reals).

BNF

Backus-Naur Form (BNF) is a particular way of writing context-free grammars in which:

- Terminals are as they are.
- Nonterminals are always enclosed by $<$ and $>$. Example: <real-number>.
- The special empty string is written as <empty>.
- Sets of productions with common heads:

  `X ::= W1, X ::= W2, ..., X ::= Wn`

  are abbreviated using the special "or" symbol "|":

  X ::= W1 | W2 | $\cdots$ | Wn

Parse Trees

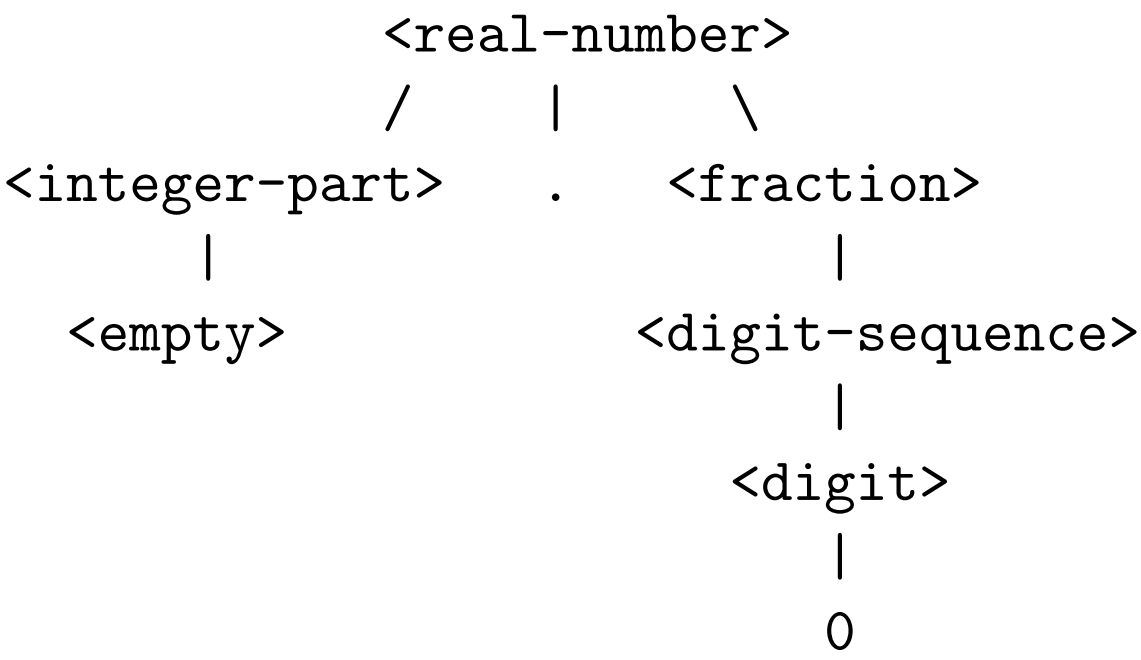
Given a context-free grammar:

Parse trees. A parse tree is a tree satisfying the following properties:

- Each leaf is labeled with a terminal or <empty> (the empty string).
- Each nonleaf node is labeled with a nonterminal.
- The root of the tree is labeled with the starting nonterminal.
- The label of a nonleaf node is the left side of some production and the labels of the children of the node, from left to right, form the right side of that production.

Parse Trees (Example)

A parse tree that generates .0:



Parse trees depict how legal expressions (programs) are generated by the underlying grammar.

Language and Ambiguity

Definition. A parse tree *generates* the string formed by reading the terminals at its leafs from left to right. The language generated by the grammar is the set of all strings that are generated by some parse trees.

Syntactic Ambiguity

A grammar is *(syntactically) ambiguous* if some string in its language is generated by more than one parse tree.

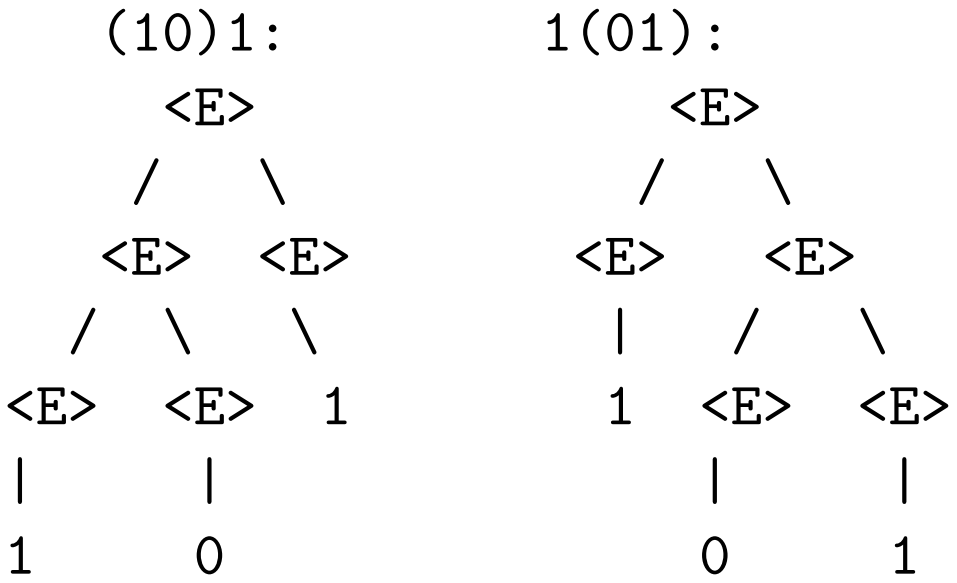
Ambiguity is a central problem in every language design. Ambiguous grammars are undesirable because when a string can be generated by two different parse trees, its meaning will often be ambiguous.

Ambiguity (Example)

The following grammar for binary strings:

```
<E> ::= <E><E> | 0 | 1
```

is ambiguous because 101 has two parse trees:



But the same language can be generated by the unambiguous grammar

```
<E> ::= 0<E> | 1<E> | 0 | 1
```

Ambiguity is often a property of a grammar, not of a language.

Dangling-Else Ambiguity

Consider the following grammar:

```
<S> ::= if <E> then <S>
<S> ::= if <E> then <S> else <S>
```

This grammar is ambiguous because the following construct:

```
if <E> then if <E> then <S> else <S>
```

can be parsed in two ways:

```
if <E> then (if <E> then <S> else <S>)
  or
if <E> then (if <E> then <S>) else <S>
```

The dangling-else ambiguity is typically resolved by matching an "else" with the nearest unmatched "if", i.e. the first case.

Derivations

A derivation is a sequence of strings:

$$S = W_0 \Rightarrow W_1 \Rightarrow \cdots \Rightarrow W_n,$$

- S is the starting nonterminal.
- W_{i+1} is obtained from W_i by replacing a nonterminal in W_i by the right side of one of its productions.
- W_n is a string consisting entirely of terminals.

Such a derivation is said to be for W_n. The following is one for 21.89 in our reals example:

```
<real-number> => <integer-part>.<fraction>
              => <digit-sequence>.<fraction>
              => <digit><digit-sequence>.<fraction>
              => 2<digit-sequence>.<faction>
              => 2<digit-sequence>.<digit-sequence>
              => ... => 21.89
```

Derivation and Parse Trees

Theorem. A string is in the language generated by a context-free grammar iff (if and only if) there is a derivation for it.
Outline of Proof: If a string is generated by a parse tree, then there is a derivation for it. Conversely, if a string is generated by a derivation, then there is a parse tree for it.

Regular Grammars and Expressions

Regular Grammars are a subset of CFGs in which all productions are in one of the following forms:

Right-Linear Grammar

- `<A> ::= x`
- `<A> ::= x<B>`

Left-Linear Grammar

- `<A> ::= x`
- `<A> ::= <B>x`

where A and B are non-terminals and x is a string of terminals.

RE Example1: Right-Linear Grammar

```
<S> ::= a<A>
<S> ::= b<B>
<S> ::= <empty>
<A> ::= a<S>
<B> ::= bb<S>
```

What is the regular language this RG generates?

Regular Expressions

Regular expressions (RE) are succinct representations of RGs using the following notations.

Sub-Expression	Meaning
x	the single char ‘x’
.	any single char except the newline
[abc]	char class consisting of ‘a’,‘b’, or‘c’
[∧abc]	any char except ‘a’,‘b’,‘c’
r*	repeat ”r” zero or more times
r+	repeat ”r” 1 or more times
r?	zero or 1 occurrence of ”r”
rs	concatenation of RE ”r” and RE ”s”
(r)s	”r” is evaluated and concatenated with ”s”
r | s	RE ”r” or RE ”s”
\x	escape sequences for white-spaces and special symbols: \b \n \r \t

Precedence of Regular Expression Operators

The following table gives the order of RE operator precedence from the highest precedence to the lowest precedence.

Function	Operator
parenthesis	( )
counters	* + ? [ ]
concatenation	
disjunction	|

RE Example2: Regular Expression Notations

RE	Meaning
abc	the string "abc"
a+b+	$\{a^m b^n : m, n \geq 1\}$
a*b*c	$\{a^m b^n c : m, n \geq 0\}$
a*b*c?	$\{a^m b^n c \text{ or } a^m b^n : m, n \geq 0\}$
xy(abc)+	$\{\text{xy(abc)}^n : n \geq 1\}$
xy[abc]	{xya, xyb, xyc}
xy(a|b)	{xya, xyb}

Questions: What are the following REs?

- foo|bar*
- foo|(bar)*
- (foo|bar)*

RE Example3: Regular Expressions

REs are commonly used for pattern matching in editors, word processors, commandline interpreters, etc.
The REs used for searching texts in Unix (vi, emacs, perl, grep), Microsoft Word v.6+, and Word Perfect are almost identical.
Examples:

- identifiers in C++:
- real numbers:
- email addresses:
- white spaces:
- all C++ source or include files:

Expressions in PLs

Consider the formula for computing one of the roots of the quadratic equation:

$$root = \frac{-B + \sqrt{B^2 - 4 * A * C}}{2 * A}.$$

- Need a lot of instructions in assembly or machine languages.
- Moreover, the programmer would have to:
  - provide storage for intermediate results;
  - do optimization: Can the two references to the value of B be combined? In what order should the operations be performed to minimize temporary storage and make best use of the hardware?

Expressions (Cont'd)

- In a high level language like FORTRAN, this can be done as a single expression almost directly:

```
ROOT = (-B+SQRT(B**2-4*A*C)) / (2*A)
```

- The notation is compact, natural, and familiar to any one who needs to do such computation.
- The language processor (compiler) rather than the programmer concerns itself with temporary storage and optimization.
- The availability of expressions in higher-level languages is one of their major advantages over machine and assembly languages.

Problems of Expressions in PLs

Expressions are powerful and natural, but they raise new problems. Take the FORTRAN expression for the quadratic formula:

- Is it syntactically correct?
- How do we know, for example in $B**2 - 4*A*C$, the "-" should take place *after* the computation of $4*A*C$?
- Does the procedure that implements SQRT have any side effects?

What Are Expressions

An expression has the following general form:

$$f(e_1, ..., e_n)$$

where f is a function (*operator*) with n arguments, and $e_1, ..., e_n$ are expressions.

- f is called an n-ary operator, and $e_1, ..., e_n$ are called *operands* of the operator. n=0 => constants; n=1 => unary operators; n=2 => binary operators; n=3 => ternary operator.

Infix Notation

If f is a binary operator, then we can write $f(e1, e2)$ as $e1\ f\ e2$ using the so called *infix notation*:

$$a + b,\ \ a * b,\ \ a - b,\ \ a/b,\ \ a \uparrow b,\ \ a == b,\ \ a < b.$$

- Many familiar binary operators are in infix notation.
- A problem is that it needs rules for precedence and associativity or otherwise uses parentheses:
  - $2 * 3 + 4 = ?$
  - $2 + 3 + 4 = ?$

Precedence

Rules of precedence decide which operators get to apply to their operands first:

> *if operator R has precedence than Q, then an expression involving both R and Q will be evaluated by first applying R to its operands.*

Example. Given the following precedence hierarchy:

$\uparrow$ (exponentiation)
$*$ (multiplication), $/$ (division)
$+$, $-$

we have:

$$2 \uparrow 2 * 3 - 6/2 = 4 * 3 - 6/2 = 12 - 3$$

Associativity

Rules of associativity decide how computations are done when there are multiple occurrences of operator at the same level of precedence.

- Left associative means that expressions are grouped from left to right;
- Right associative means that expressions are grouped from right to left.

Example. Normally, $+$ and $-$ are at the same level of precedence and are both left associative:

$$6+2-8+18-2+4 = 8-8+18-2+4 = 0+18-2+4 = 18-2+4 = 16+4$$

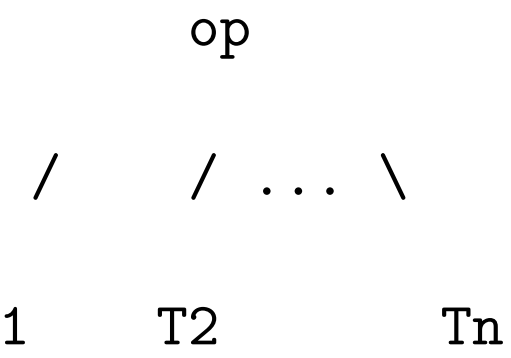
(What if $+$ is left but $-$ is right associative?)

Normally $\uparrow$ is right associative, so

$$120 \uparrow 2 \uparrow 1 \uparrow 3 = 120 \uparrow 2 \uparrow 1 = 120 \uparrow 2$$

Tree Representation of Expressions

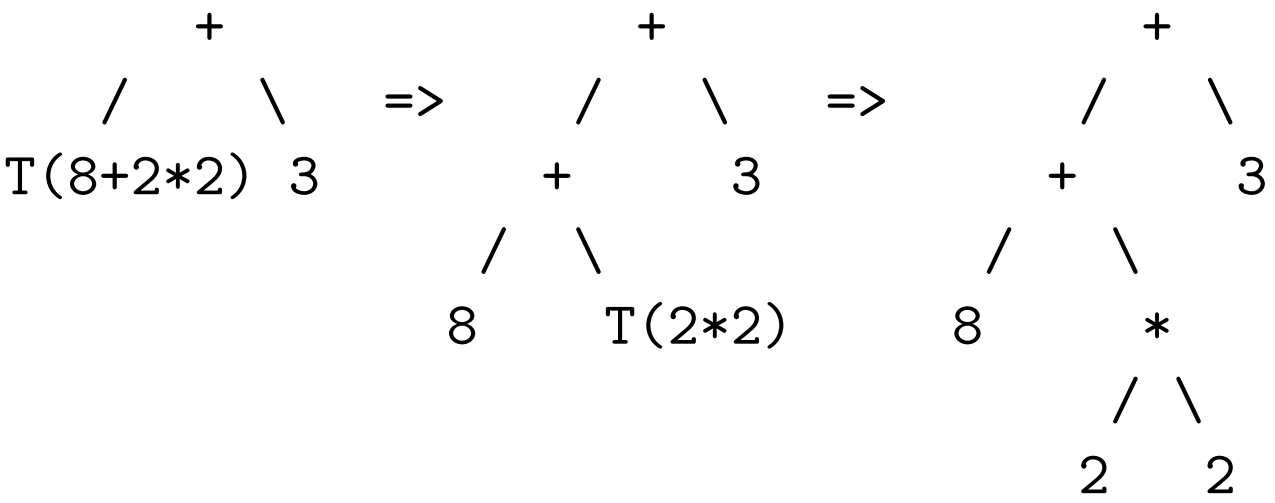
1. The tree representation of a constant (0-ary operator) consists of a single node labeled by the constant
2. The tree representation of the expression $op(E_1, ..., E_n)$ consists of a node labeled by op, and n subtrees for $E_1, ..., E_n$:



where T_i, $1 \leq i \leq n$, is the tree representation of E_i.

Tree Representation (Example)

The tree representation of $8 + 2 * 2 + 3$ is:



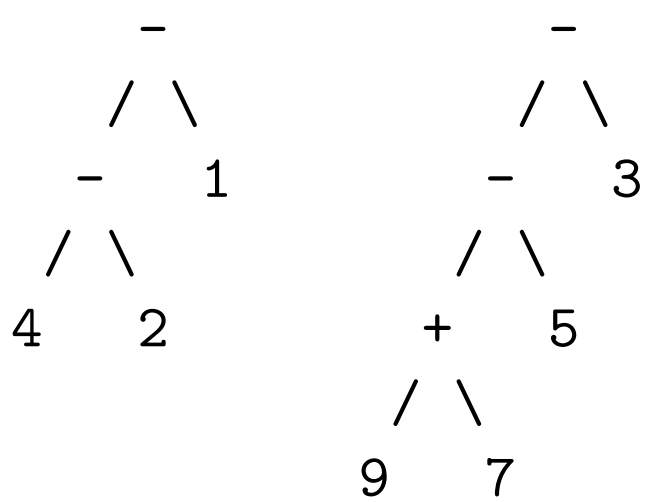
Grammars for Expressions

A desirable grammar for expressions is one such that parse trees according to this grammar are as close as possible to tree representations of the expressions.

Handling Associativity. Consider infix expressions formed by applying $+$ and $-$ to numbers:

$$4-2-1 \qquad 9+7-5-3$$

Their tree representations:

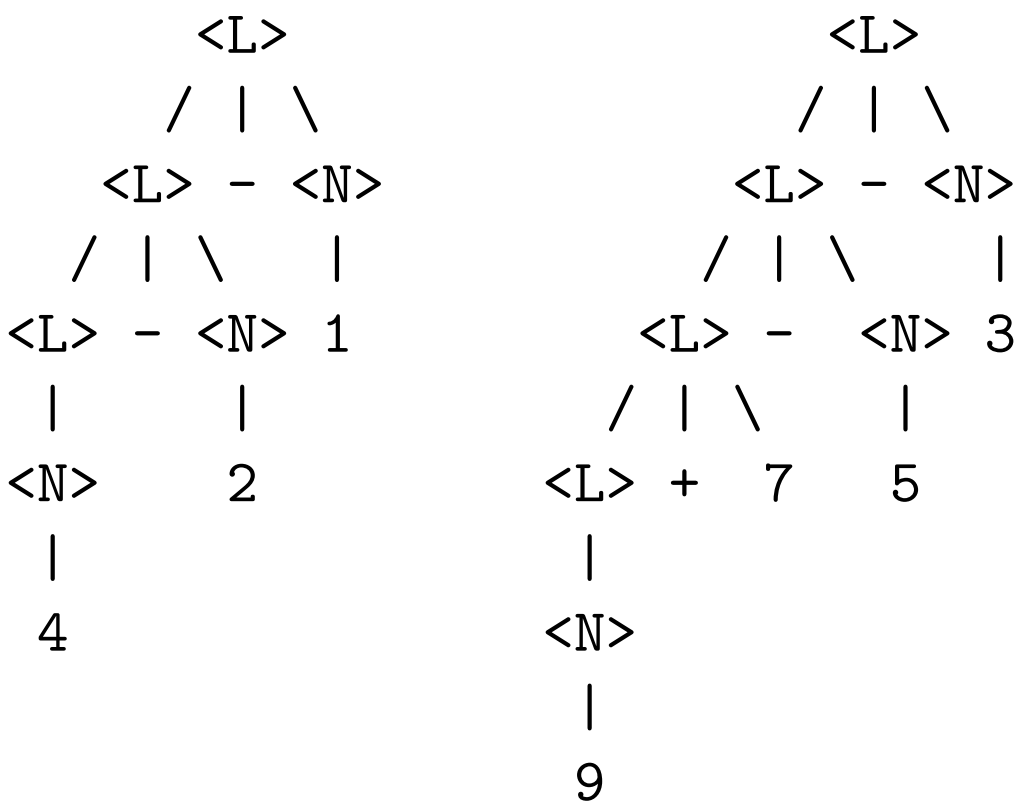


Grammars for Expressions (Cont'd)

A desirable grammar for such expressions is:

```
<L> ::= <L> + <N> | <L> - <N> | <N>
<N> ::= 0 | 1 | 2 | ...
```

Two corresponding parse trees:



Grammars for Expressions (Cont'd)

A less desirable grammar is:

```
<R> ::= <N> + <R> | <N> - <R> | <N>
<N> ::= 0 | 1 | ...
```

In general:

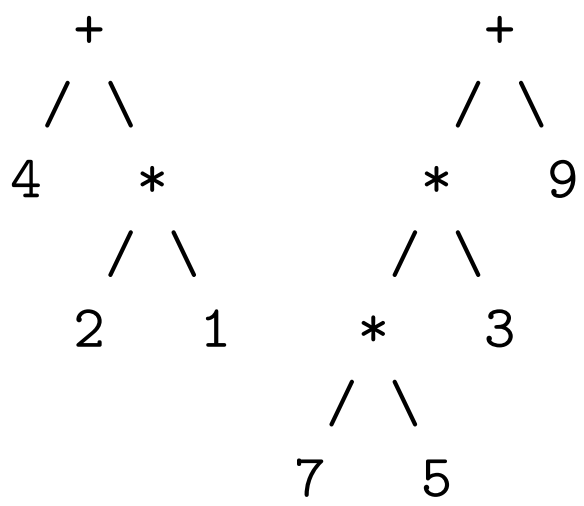
- For right associativity, the nonterminal should appear on the right.
- Vise versa for left associativity.

Grammars for Expressions (Cont'd)

Handling Associativity and Precedence. Consider infix expressions formed by applying $*$ and $+$ to numbers:

$$4 + 2 * 1 \qquad 7 * 5 * 3 + 9$$

Their tree representations:

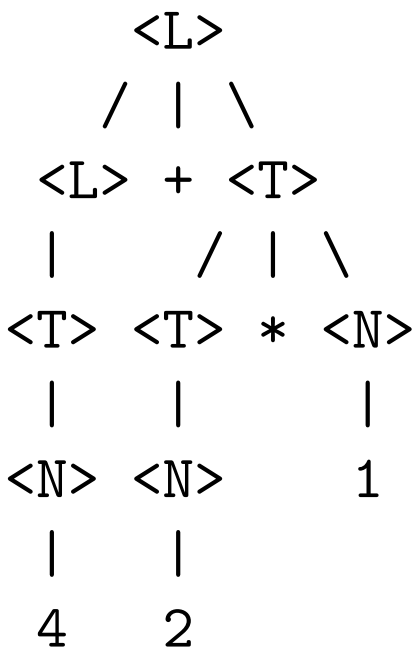


Grammars for Expressions (Cont'd)

A desirable grammar for such expressions is:

```
<L> ::= <L> + <T> | <T>
<T> ::= <T> * <N> |  <N>
<N> ::= 0 | 1 | 2 | ...
```

A corresponding parse tree:



Prefix Notation

In prefix notation, the expression $op(e_1, ..., e_n)$ is written as $op\ e_1\ \cdots\ e_n$:

```
+ 20 30 = 50      * + 20 30 2 = * 50 2 = 100
* 20 + 30 10 = * 20 40 = 800
```

Prefix notation is easy to decode during a left-to-right scan of an expression:

> *if it begins with $+$, then the next expression after $+$ must be the first operand of $+$, and the expression after that must be the second operand of $+$.*

Postfix Notation

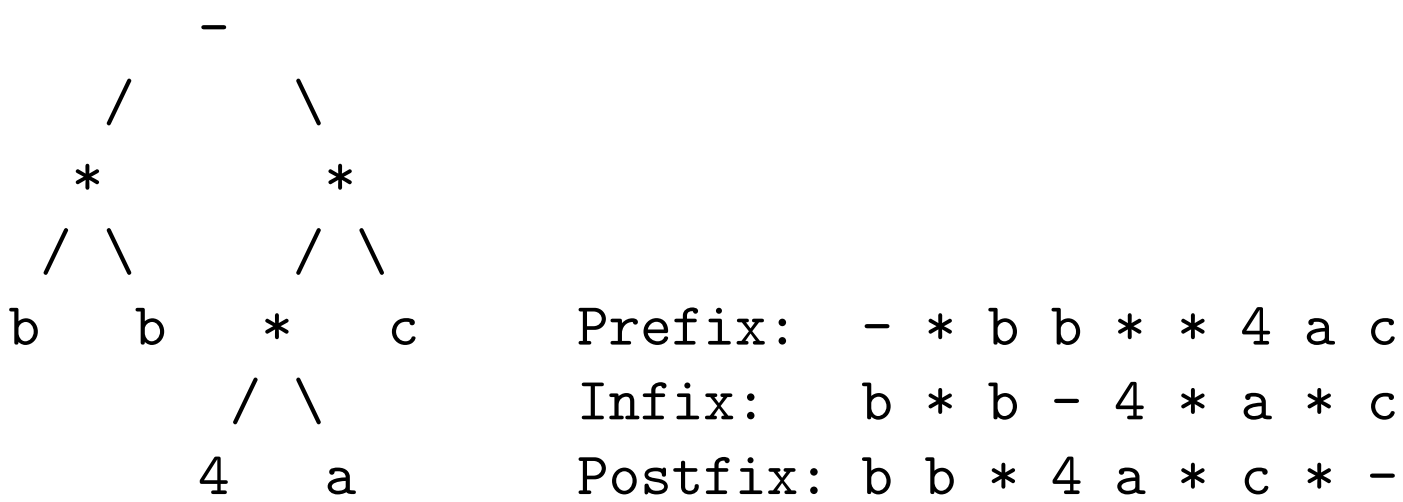
In postfix notation, the expression $op(e_1, ..., e_n)$ is written as $e_1 \cdots e_n\ op$:

```
20 30 + = 50    20 30 + 2 * = 50 2 * = 100
20 30 10 + * = 20 40 * = 800
```

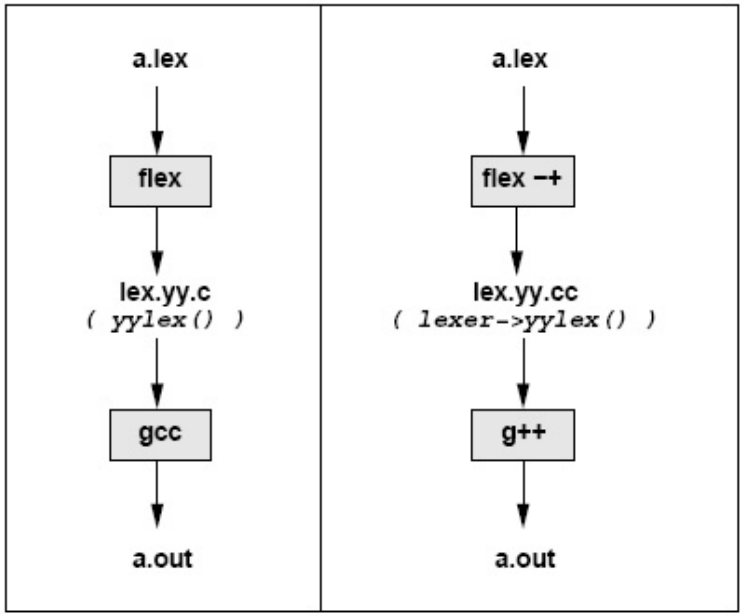
Expressions in postfix notation can be easily evaluated using a stack data structure.

Comparisons

- Prefix or postfix notations do not need parentheses nor do they need rules of precedence and associativity.
- Their disadvantages are that they are hard for *humans* to comprehend.
- Tree representation is independent of particular notations used:



flex : Fast Lexical Analyzer



- flex is a program that generates scanners or lexical analyzers.
- flex reads a description of a scanner written in a lex file and outputs a C or C++ program containing a routine called "yylex()" in C or "(FlexLexer*)lexer→yylex()" in C++.
- Compile the `lex.yy.cc` to `a.out` which will be the lexical analyzer.

flex Example 1: count.lex

```
%option noyywrap

%{
int numlines=0, numchars=0;
%}
%%
\n   ++numlines; ++numchars;
.    ++numchars;
%%
int main(int argc, char **argv)
{    yylex();
     printf("# of lines = %d, # of chars = %d\n",
            numlines, numchars);
     return 0;
}
```

Run "flex count.lex", and then run a C compiler on the resulting source code lex.yy.c.

Example 1 in C++, Version 1, count-cc1.lex

```
%option noyywrap
%{
#include <iostream>
  using namespace std;
int numlines=0, numchars=0;
%}

%%
\n      ++numlines; ++numchars;
.       ++numchars;
%%
int main(int argc, char **argv)
{  yylex();
   cout << "# of lines = " << numlines << endl
        << "# of chars = " << numchars << endl;
   return 0; }
```

Run "flex count-cc1.lex", and then run a C++ compiler on the resulting source code lex.yy.c.

Example 1 in C++, Version 2: count-cc2.lex

```
%option noyywrap
%{
#include <iostream>
  using namespace std;
int numlines=0, numchars=0;
%}
%%
\n      ++numlines; ++numchars;
.       ++numchars;
%%
int main(int argc, char **argv)
{  FlexLexer* lexer = new yyFlexLexer;
   lexer->yylex();
   cout << "# of lines = " << numlines << endl
        << "# of chars = " << numchars << endl;
   return 0; }
```

Run "flex -+ count-cc2.lex" and then run a C++ compiler on the resulting C++ source code lex.yy.cc

flex Input File Format

```
%{
    text to be copied exactly to the output
%}
flex Definitions

%%
    Rules = Patterns in RE + Actions in C or C++
%%

User Codes (in C or C++)
```

- Patterns, written in REs, must start on the first column, and action must start on the same line as its pattern.
- In the Definitions or Rules sections, any indented text or text enclosed in "%{" and "%}" is copied verbatim to the output.

How the Input is Matched?

- The generated lexical analyzer should have a loop calling the function `yylex()` for each input file.
- Each call to `yylex()` will scan the input from left to right looking for strings that match any of the RE patterns.
- If it finds more than 1 match, it takes the longest match.
- If it finds 2 matches of the same length, it takes the first rule.
- When there is a match,

```
extern char* yytext = /* content of matched string */
extern int yyleng = /*length of the matched string */
```

- If no rule is given, the default rule is to echo the input to the output.

flex Example 2: Default Rule

```
%option noyywrap

%%
%%

int main(int argc, char **argv)
{
    yylex();
    return 0;
}
```

How the Input is Matched? ..

- Actually the variable `yytext` can be specified as a pointer or an array in the flex-definition section.

```
%pointer     /* extern char* yytext */
%array       /* extern char yytext[YYLMAX] */
```

- Using pointer for `yytext` renders faster operation and avoids buffer overflow for large tokens. While it may be modified but you should NOT lengthen it or modify beyond its length (as given by yyleng). Using array for `yytext` allows you to modify the matched string freely.
- You can use "%array" with C programs, but not C++ programs.

flex Example 3: Use of yytext

```
%option noyywrap

%{
#include <stdio.h>
%}

%%
[a-yA-Y]     printf("%c", *yytext+1);
[zZ]         printf("%c", *yytext-25);
.            printf("%c", *yytext);
%%

int main(int argc, char **argv)
{  yylex();
   return 0;
}
```

2 flex Directives: ECHO, REJECT

- ECHO: copy `yytext` to the output
- REJECT: ignore the current match and proceed to the next match.
  - if there are 2 rules that match the same length of input, it may be used to select the 2nd rule.
  - may be used to select the rule that matches less text.

flex Example 4: REJECT

```
%option noyywrap

%{
#include <stdio.h>
%}

%%
a      |
ab     |
abc    |
abcd   ECHO; REJECT;
.|\n   printf("xx%c", *yytext);
%%

int main(int argc, char **argv)
{    yylex(); return 0; }
```

Global Variables/Classes

C Implementation	C++ Implementation
FILE* yyin	abstract base class: FlexLexer
FILE* yyout	derived class: yyFlexLexer
char* yytext	member function: const char* YYText()
int yyleng	member function: int YYLeng()

Miscellaneous

Exceptions about character class RE:

- For character class: special symbols like *, + lose their special meanings and you don't have to escape them. However, you still have to escape the following symbols: \, -,], ∧, etc.
- There are some pre-defined special character class expression enclosed inside "[:" and ":]". e.g.

```
[:alnum:]  [:alpha:]  [:digit:]
[:lower:]  [:upper:]
```

Some important command-line options:

Option	Meaning
-d	debug mode
-p	performance report
-s	suppress default rule; can find holes in rules
-+	generate C++ scanners

flex Example 5: Generating C++ Scanners

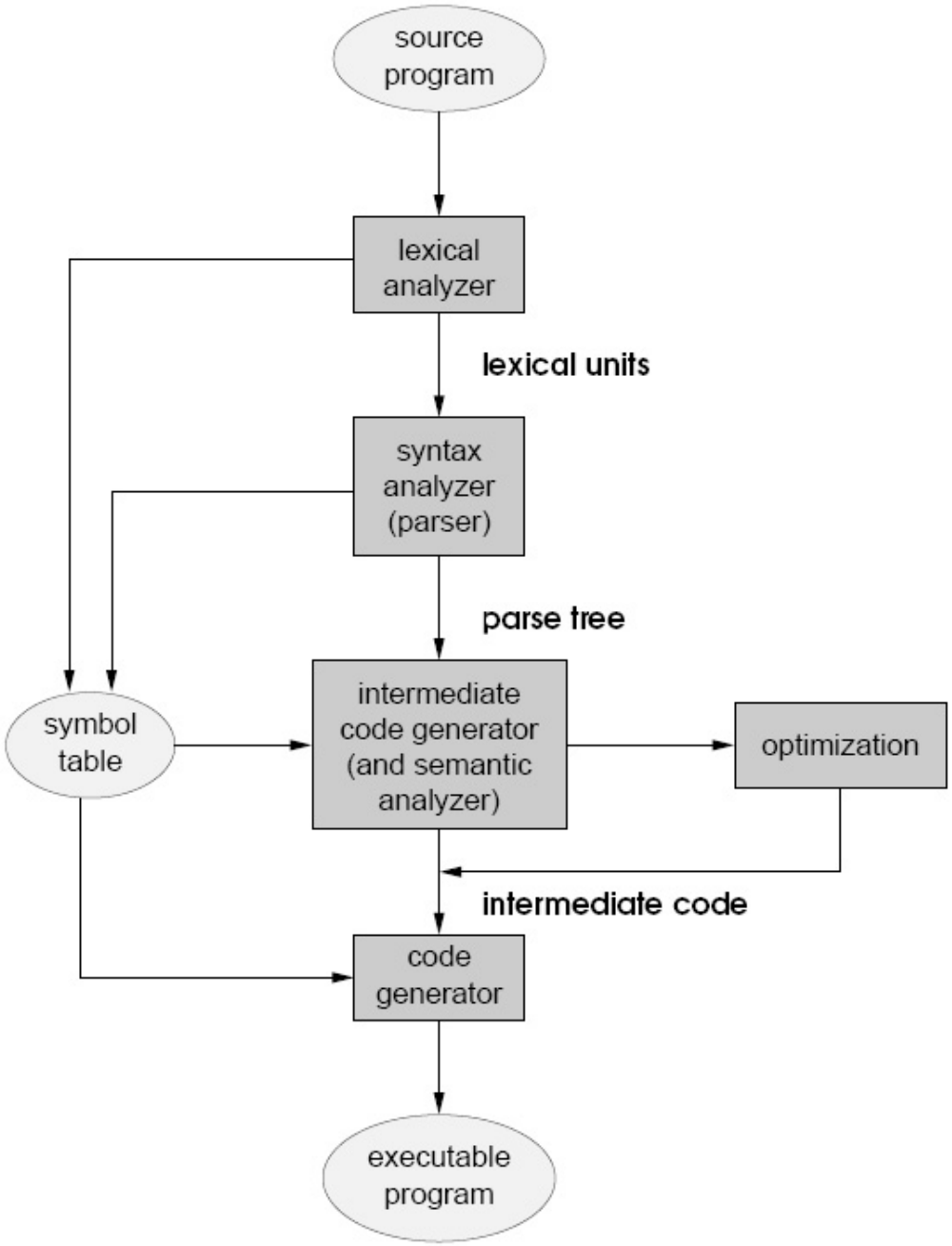
```
%option noyywrap
%{
int mylineno=0;
%}
string   \"[^\n"]+\"
ws       [ \t]+
alpha    [A-Za-z]
dig      [0-9]
name     ({alpha}|{dig}|\$)({alpha}|{dig}|[_.\-/$])*
num1     [-+]?{dig}+\.?([eE][-+]?{dig}+)?
num2     [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
number   {num1}|{num2}
%%
{ws}     /* skip blanks and tabs */
{number} cout << "number " << YYText() << '\n';
{name}   cout << "name " << YYText() << '\n';
{string} cout << "string " << YYText() << '\n';
\n       mylineno++;
```

flex Example 5: Generating C++ Scanners ..

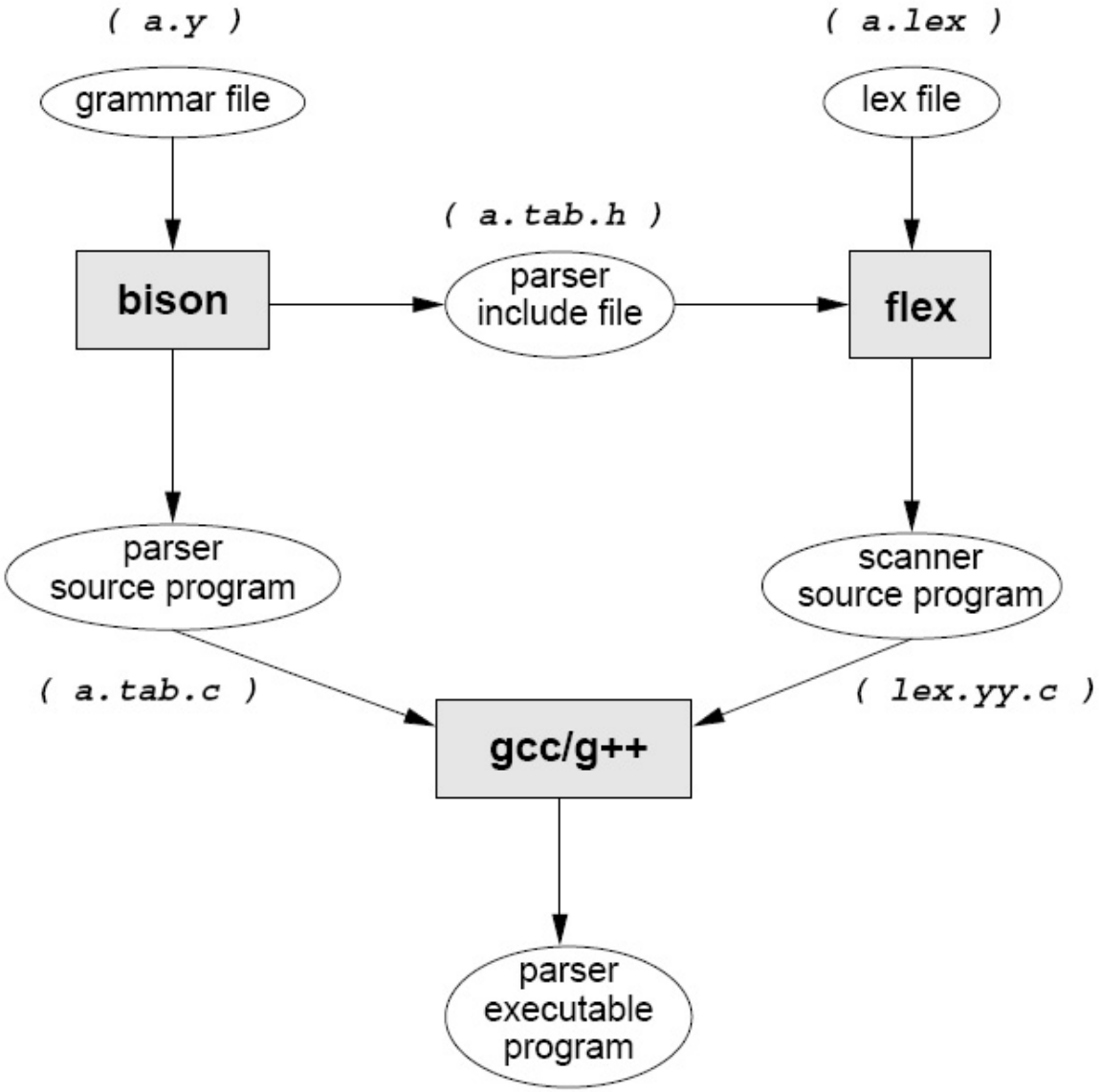
```
"/*"      {   int c;
              while((c = yyinput()) != 0)
              {       if (c == '\n')
                        ++mylineno;
                      else if (c == '*')
                      {   if ((c = yyinput()) == '/') break;
                          else   unput(c); }
               }
            }
.         /* cout << "unrecognized " << YYText() << endl; */
%%

int main(int argc, char** argv)
{  FlexLexer* lexer = new yyFlexLexer;
   while (lexer->yylex() != 0)
         ;
   return 0;
}
```

Compilation



Syntax analysis Using bison and flex



bison: Introduction

- bison is a program that generates a parser for a given CFG. It is compatible with perhaps the first popular parser generator called "yacc" (Yet Another Compiler's Compiler).
- bison reads a description of a CFG written in a bison Grammar File, and output a C program containing a routine called `yyparse()`.
- The default name of the output C program is "`*.tab.c`". Compile `*.tab.c` to `a.out` which will be the parser.
- bison can only parse a subset of CFGs called LALR(1) grammars, using a bottom-up parsing algorithm with one look-ahead token.
- bison only generates a parser and does NOT provide a scanner automatically. To get both a parser and a scanner:
  - run both bison and flex
  - put the lexical analysis codes in the section "Additional C Codes".

bison Grammar File Format

```
%{
C Declarations
%}

bison Declarations

%%
Grammar Rules + Actions
%%

Additional C code
```

- Similar to flex, any statements between the "%{" and "%}", as well as the additional C codes are copied verbatim to the output.

Example 1: Reverse Polish Notation Calc rpn-all-calc.y

```
%{
#define YYSTYPE double
#include <math.h>
%}
%token NUM
%% /* grammer rules and actions follow */
input: /* empty */
     | input line
     ;
line: '\n'
    | exp '\n' { printf("\t%.10g\n", $1); }
    ;
exp: NUM { $$ = $1; }
   | exp exp '+' { $$ = $1 + $2; }
   | exp exp '-' { $$ = $1 - $2; }
   | exp exp '*' { $$ = $1 * $2; }
   | exp exp '/' { $$ = $1 / $2; }
   | exp exp '^' { $$ = pow($1, $2); }
   | exp 'n' { $$ = -$1; }
   ;
%%
```

bison Example 1 ..

```
/* additional C code */
#include <ctype.h>
#include <stdio.h>
int yylex(void)
{   int c;
    while ((c = getchar()) == ' '||c == '\t') ; /* skip white spaces
    if (c == '.' || isdigit(c)) /* process numbers */
    {   ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUM;    }
    if (c == EOF) return 0;
    return c; }
int main() { return yyparse(); }
int yyerror(const char* s) { printf("%s\n", s); return 0; }
```

Run "bison rpn-all-calc.y", and then use a C compiler to compile the resulting source code rpn-all-calc.tab.c.

The same parser in C++: rpn-calc-all-cc.y

```
/* reverse polish notation calculator */
%{
#define YYSTYPE double
#include <math.h>
  using namespace std;
#include <iostream>
#include <stdio.h>
int yyerror(const char *s);
int yylex(void);
%}
%token NUM
%% /* grammer rules and actions follow */
input: /* empty */
     | input line ;
line: '\n'
    | exp '\n' { cout << $1 << endl; } ;
exp: NUM { $$ = $1; }
   | exp exp '+' { $$ = $1 + $2; }
   | exp exp '-' { $$ = $1 - $2; }
   | exp exp '*' { $$ = $1 * $2; }
   | exp exp '/' { $$ = $1 / $2; }
   | exp exp '^' { $$ = pow($1, $2); }
   | exp 'n' { $$ = -$1; } ;
```

The same parser in C++

```
%% /* additional C++ code */
#include <ctype.h>
int yylex(void) {
int c;
/* skip white spaces */
while ((c = getchar()) == ' ' || c == '\t');
/* process numbers */
if (c == '.' || isdigit(c)) {
ungetc(c, stdin);
scanf("%lf", &yylval);
return NUM; }
if (c == EOF) return 0;
return c; }
int main() { return yyparse(); }

int yyerror(const char* s) {
cout << s << endl;
return 0; }
```

Run "bison rpn-calc-all-cc.y" and run a C++ compiler on the resulting source code rpn-calc-all-cc.tab.c.

bison Grammar File

C Declarations

- define types and variables
- `#define`'s and `#include`'s

bison Declarations

- declares names of the terminals/non-terminals symbols
- describe operator precedence and associativity
- data types of semantic values of variables

Grammar Rules

- production rules of CFGs

Additional C Codes

- definition of `yylex()`
- definition of `yyerror()` and other supporting routines

Description of bison Grammar Rules

- Three ways to represent terminals:
  (1) character literals. e.g. '+' for the + operator.
  (2) C string constants. e.g. "else" for the keyword **else**.
  (3) C-like identifiers. e.g. NUM (for numbers). The convention is to write it in upper case.
- Non-terminals are represented as C-like identifiers. The convention is to write them in lower case. e.g. exp for `<Expression>`.
- Use : to represent ::=.
- A rule ends with a ';'.
- Example of a (production) rule in bison:

```
if-stmt : ``if'' bool-expr ``then'' stmt ';'
        | ``if'' bool-expr ``then'' stmt ``else'' stmt ';'
        ;
```

Type and Semantic Value

- Most terminals or tokens have
  (1) a type
  (2) a semantic value

  e.g. the integer 123 has:

  type : INTEGER
  semantic value : one hundred twenty-three
- But *some* terminals do NOT. e.g. operator '+'.
- Non-terminals also have semantic values. e.g.
  - the semantic value of a math expression (e.g. E = a + b) is a real number — result computed from its constituents.
  - the semantic value of a compiled expression is a parse tree.

Semantic Actions

- Define the semantics of a program!
- Compute the semantic value of the non-terminal on the LHS of a grammar production rule based on the semantic values of the terminals and non-terminals on the RHS of the rule.
  For example,

```
    expr : expr '+' term   { $$ = $1 + $3 }
where
    $$ = semantic value of ''expr'' on the LHS.
    $1 = semantic value of the 1st token on the
         RHS, which is the non-terminal ''expr''.
    $3 = semantic value of the 3rd token on the
         RHS, which is the non-terminal ''term''.
```

bison Types, Variables, Functions

Entity	Meaning
YYSTYPE	macro for the token type (default: int)
extern YYSTYPE yylval	value of an input token
extern int yyparse(void)	parser function
extern int yyerror(const char*)	error reporting function

- When no action is specified, the default action is: `$$ = $1`.
- Token type code of EOF = any non-positive value (including 0).

bison Example 1 again: rpn-calc.y

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

%token NUM

%% /* Grammer rules and actions follow */
input: /* empty */
     | input line
     ;
line: '\n'
    | exp '\n' { printf("\t%.10g\n", $1); }
    ;
exp: NUM { $$ = $1; }
   | exp exp '+' { $$ = $1 + $2; }
   | exp exp '-' { $$ = $1 - $2; }
   | exp exp '*' { $$ = $1 * $2; }
   | exp exp '/' { $$ = $1 / $2; }
   | exp exp '^' { $$ = pow($1, $2); }
   | exp 'n' { $$ = -$1; }
   ;
%%

int main() { return yyparse(); }
int yyerror(const char* s) { printf("%s\n", s); return 0; }
```

bison Example 1 again: rpn-calc.tab.h

```
bison -d rpn-calc.y
```

produces 2 files:

- rpn-calc.tab.h: Some C declarations needed by the lex file
- rpn-calc.tab.c: Source program of the parser

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define NUM     257

extern YYSTYPE yylval;
```

bison Example 1 again: rpn-calc.lex

```
%option noyywrap

%{
#define YYSTYPE double     /* type for bison's var: yylval */
#include <stdlib.h>        /* for atof(const char*) */
#include "rpn-calc.tab.h"
%}

digits [0-9]
rn     (0|[1-9]+{digits}*)\.?{digits}*
op     [+*^n/\-]
ws     [ \t]+

%%
{rn}   yylval = atof(yytext); return NUM;
{op}   |
\n     return *yytext;
{ws}   /* eats up white spaces */
%%

/* There is NO main function! */
```

bison Example 2: Infix-Notation Calculator

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

%token NUM
%left '-' '+'
%left '*' '/'
%left NEG
%right '^'

%% /* Grammer rules and actions follow */
input:     /* empty */ | input line ;
line:      '\n' | exp '\n' { printf("\t%.10g\n", $1); } ;
exp: NUM { $$ = $1; }
   | exp '+' exp { $$ = $1 + $3; }
   | exp '-' exp { $$ = $1 - $3; }
   | exp '*' exp { $$ = $1 * $3; }
   | exp '/' exp { $$ = $1 / $3; }
   | '-' exp %prec NEG { $$ = -$2; }
   | exp '^' exp { $$ = pow($1, $3); }
   | '(' exp ')' { $$ = $2; } ;
%%

int main() { return yyparse(); }
int yyerror(const char* s) { printf("%s\n", s); return 0; }
```