

Context Free Grammars

Formal grammars were first studied by linguists, then they were applied to computer languages. **Noam Chomsky** of MIT first defined the notion of a *context-free grammar* (CFG). BNF is a **meta-syntax** that is usually conforms to the rules for a context-free grammar and is used to specify the **concrete syntax** of a programming language. Think of a BNF grammar as a *finite meta language* for describing the rules for how the syntactic components of a programming language can be used. In other words, a formal language that is used to completely and unambiguously describe a programming language.

A context-free grammar for a language imposes a *hierarchical structure* on the syntax of a program. An entire program can be represented by a **parse tree**, which is a hierarchical representation of the syntactic constructs in a program. Each particular language construct is represented by a unique parse tree, and the entire program is thus a collection of parse trees rooted at the top by a special grammar symbol called the **start symbol**.

The job of the **syntax analyzer** or **parser** is to verify that an input text, having been converted into a token stream by the **lexical analyzer**, is *syntactically correct*—meaning that the parser can construct a valid parse tree for the entire input text representing a legal source program in the target language. A context-free grammar is thus a specification of the legal syntax of a language and BNF grammar rules guide the parsing of a source program.

A context free grammar has four parts:

1. a set of symbols called **terminals**, which are the *tokens* defined by the language.
2. a set of symbols called **non-terminals**, which are *meta variables* representing syntactic constructs (e.g., expressions, statements, type declarations, etc.) in the language
3. a set of **production rules** or **grammar rules** for specifying how legal syntactic constructs are formed.
4. a special non-terminal symbol chosen as the **start symbol** for the set of production/grammar rules.

Lex is a tool that is used to read a lexical specification and generate a lexical analyzer subprogram (in C) that can tokenize an input text based on a definition of the tokens or lexemes for a language. Yacc is a tool that is used to read a BNF grammar specification (called a YACC grammar) and generate a syntax analyzer program (in C and has the 'main' procedure). The Yacc syntax analyzer calls the lexer generated by Lex to fetch one token at a time and tries to build a legal parse tree by matching sequences of tokens to the right hand side of rules in the Yacc grammar. If it succeeds, then the input text is successfully parsed and is declared a syntactically correct program.

Writing a Context-Free Grammar

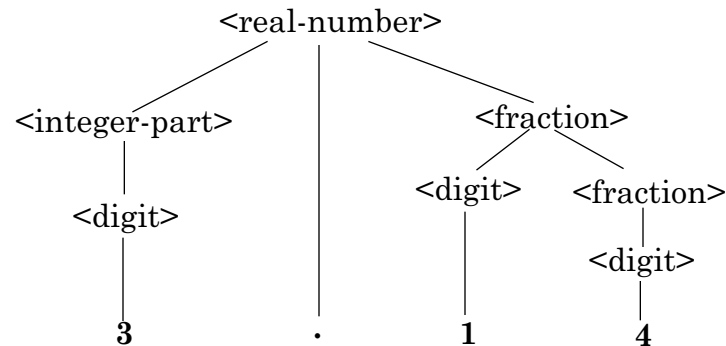
A popular context-free grammar notation is **BNF** (Backus-Naur Form), which is used to write the grammar for a language. Once the BNF grammar is written, it serves as a specification for how to write syntactically correct programs in some programming language. There are many different kinds of BNF notation, all similar but different. For example, the following generic BNF grammar is used for specifying floating point numbers:

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

'::=' indicates a production/grammar rule, and means the left hand side (LHS) "is defined by" the non-terminal and terminal symbols on the right hand side (RHS). In a CFG, there is always only one non-terminal symbol on the LHS and there are one or more non-terminal and terminal symbols on the RHS. Symbols inside of '<>' brackets are non-terminals. The '|' symbol is a logical separator used only on the RHS as a short-hand notation for specifying alternative choices for the RHS of a production rule.

All other symbols in a rule are terminal symbols (i.e., tokens from the language), such as the decimal point '.' and the numerals 0-9.

Observe that the grammar defines a recursive tree structure. In a parse tree, the root and intermediate nodes are always non-terminal symbols, while the leaf nodes are always terminal symbols:



Grammar Production Rules

Productions rules are a compact *recursive* notation for specifying how to derive legal syntactic constructs.

```
<num> ::= <digit> | <digit> <num>
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

This compact rule is equivalent to writing two alternative choices (logical ‘or’) for how to define a number. Note that the second alternative is **right recursive**. Could we write this rule left recursively?

```
<num> ::= <digit>
<num> ::= <digit> <num>
```

The numeric terminal

string 789 is **derived** by the following recursive application of the production rules:

```
<num>      => <digit> <num>
           => 7 <num>
           => 7 <digit> <num>
           => 7 8 <num>
           => 7 8 <digit>
           => 7 8 9
```

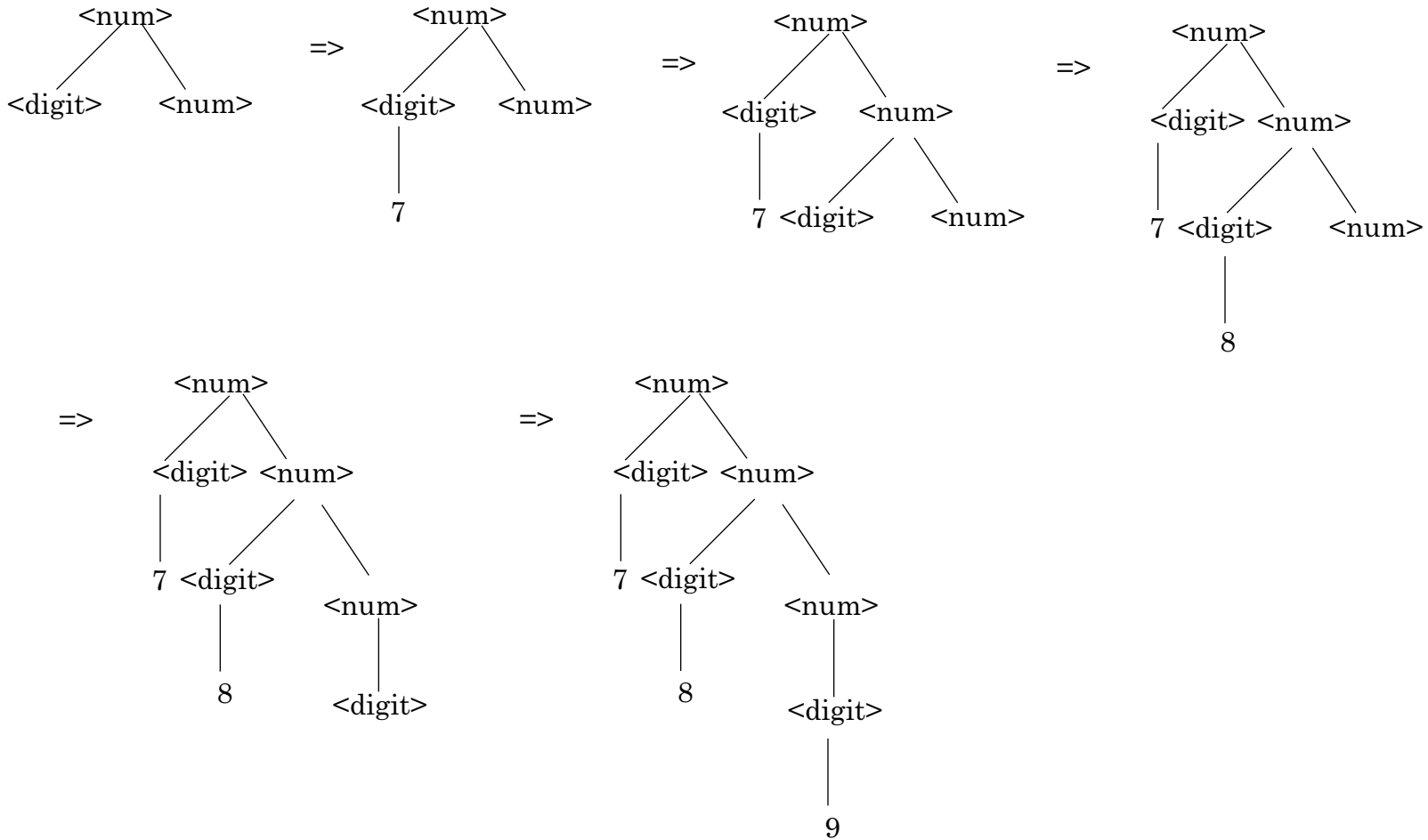
Notice that we always chose to expand the left-most non-terminal symbol at each step. This is called a **top-down left-most derivation**. A left-most derivation effectively constructs a parse tree in a top-down, left-to-right fashion.

A special non-terminal symbol <empty> is sometimes defined on the RHS of a production rule, meaning that an application of the rule can lead to the empty string (i.e., string of length zero). This is how you specify a rule that has an optional component. For example, the following rule generates strings of balanced parentheses:

```
<balanced> ::= ( <balanced> ) | <empty>
<balanced> => ( <balanced> ) => (( <balanced> )) => ((<empty>)) =>(() )
```

Parse Trees

A top-down left-most derivation of a terminal string using a sequence of production rules is equivalent to constructing a parse tree for the string, in a recursive top-down, left-to-right fashion. This technique is called **Recursive Descent Parsing**.



Parse Trees and Parsing

A **parse tree** as defined by a grammar satisfies the following conditions:

1. each leaf is labeled with a terminal symbol (token) or <empty> (the empty terminal symbol is often written use the lower-case greek symbol epsilon ϵ)
2. each non-leaf (interior) node is labeled with a non-terminal symbol
3. each interior node is the LHS of a production rule and the children nodes form the RHS side of the rule when read from left to right
4. the root is labeled with the starting nonterminal symbol for the (sub)set of production rules applied.

Observe that given a string of terminal symbols or tokens, you should be able to “run the productions in reverse” by scanning the terminal string left-to-right and constructing a parse tree **bottom-up** (right-to-left) by finding a suitable sequence of production rules so that the terminal symbol can be “reduced” to a non-terminal symbol on the LHS of some production rule. By repeating the process, you can determine whether or not a given string of terminal symbols is in fact defined by the grammar. This process is called **shift-reduce** parsing.

Shift-reduce parsing is a powerful and common technique for implementing the syntax analyzer for a compiler. The lexical analyzer produces a stream of tokens (terminals) that are fed into the syntax analyzer, which tries to successfully run a sequence of production rules in reverse, to *reduce* the token stream to the start symbol of the grammar.

If each sequence of tokens can be reduced to the LHS of some production rule, all the way to the start symbol for the grammar, then the “parse” is declared to be successful, and the input program is deemed to be syntactically correct. If the parser is unable to reduce some token or sequence of tokens to a valid rule in the grammar, then there is a **syntax error**, and the parse fails.

In practice, parsers are often automatically generated by **parser-generator** tools. YACC stands for Yet Another Compiler-Compiler, meaning it is a compiler for a compiler. There are many others. YACC generates a **bottom-up shift-reduce parser**. There are other parser generator tools that generate **top-down recursive-descent** parsers. Bottom-up parsers are usually “faster” while top-down parsers are “easier” to understand, implement by hand, and debug. You can implement a top-down parser by hand by writing a recursive procedure for each grammar rule.

Lexical Syntax

The syntax of a programming language is specified in terms of “atomic units” called *tokens* or *terminals*.

A lexical analyzer must map sequences of characters (*lexemes*) into keywords, identifiers, operators, numbers, strings, comments, and whitespace. Usually, we can use a compact *lexical syntax* or *lexical notation* to express the mapping of a lexeme in the language to a token that the lexical analyzer can pass on to the syntax analyzer.

In practice, a token is a symbolic name for a unique integer number that uniquely identifies a token string. The syntax analyzer “sees” the integer representing the token string, not the string itself. For example:

```
"+"      { return(PLUS); }      // PLUS = 401
"-"      { return(MINUS); }    // MINUS = 402
"*"      { return(MULT); }     // MULT = 403
"/"      { return(DIV); }     // DIV = 404
```

A token can also be specified in terms of a compact notation for representing various ways to form strings of characters and/or digits. For example, the token for an identifier is commonly required to begin with a lower or upper case alphabetic character or underscore ‘_’, followed by zero or more upper/lower case characters, digits 0-9 or ‘_’.

```
[a-zA-Z_][a-zA-Z0-9_]* { return ID; } // '*' is Kleene star and means zero or more
```

Similarly, numbers might be represented in decimal notation, octal notation, or hexadecimal notation. A hex number begins with ‘0x’ or ‘0X’ followed by one or more digits or upper/

```
[1-9][0-9]*           { return(DECIMALINT); }
0[0-7]*              { return(OCTALINT); }
(0x|0X)[0-9a-fA-F]+  { return(HEXINT); } // '+' means one or more occurrences
```

White space consists of tab ‘\t’, a blank character ‘\ ’, newline ‘\n’, and return ‘\r’. A lexical analyzer usually skips whitespace, and may count the number of lines in the source program by counting end-of-line characters.

```
[\t\ ]+ { skip (); }
[\n\r]  { newline++; skip(); }
```

Lexical Syntax

Most languages have several reserved words; for example, the following are all the **keywords** in ANSI C++ (bold are the new keywords added to C):

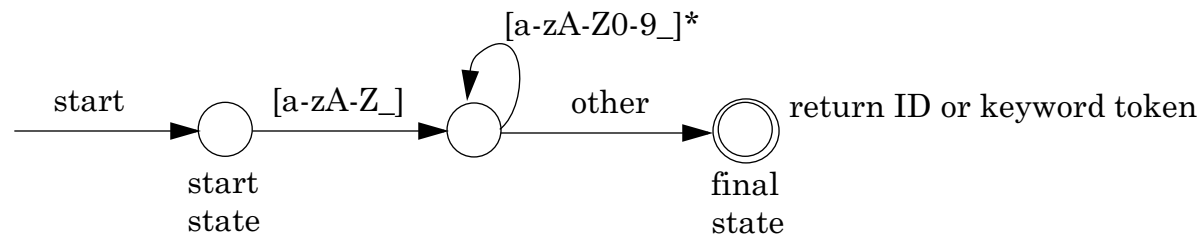
```
"asm", "auto", "bool", "break", "case", "catch", "char", "class", "const",
"const_cast", "continue", "default", "delete", "do", "double", "dynamic_cast",
"else", "enum", "explicit", "extern", "false", "float", "for", "friend", "goto",
"if", "inline", "int", "long", "mutable", "namespace", "new", "operator",
"private", "protected", "public", "register", "reinterpret_cast", "return",
"short", "signed", "sizeof", "static", "static_cast", "struct", "switch",
"template", "this", "throw", "true", "try", "typedef", "typeid", "typename",
"union", "unsigned", "using", "virtual", "void", "volatile", "wchar_t", "while"
```

Each keyword will then have a corresponding token definitions in the lexical analyzer. A token is just an integer value, represented by a symbolic name, that is a 1-1 mapping to the actual string representation of all keywords:

```
"asm"      { return(ASM); } // token 301
"auto"     { return(AUTO); } // token 302
"bool"     { return(BOOL); } // token 303
"break"    { return(BREAK); } // token 304
...

```

In practice, a program called a *scanner generator* (e.g., Lex and Flex) can read such lexical definitions and automatically generate code for an algorithm that implements the lexical analyzer (called the scanner). The scanning of the input by the lexical analyzer is carried out by a *deterministic finite automaton (DFA)*, which is an abstract state machine (algorithm) for recognizing and separating out tokens from an input stream of characters. A DFA for recognizing identifiers can be represented using a transition diagram:



Concrete and Abstract Syntax

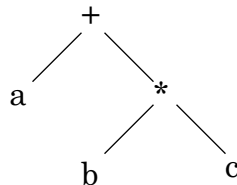
Different languages can use different *concrete syntax* for representing expressions; however, syntactically different expressions with the same meaning must have a common *abstract syntax*. The concrete syntax consists of the *literal symbols* used to syntactically form the expression. The abstract syntax captures the *semantics* of the expression.

For example, arithmetic expressions can be defined using **infix**, **prefix**, or **postfix** notation, i.e., different arrangements of symbols to form different concrete syntaxes.

How do we determine the meaning of the expression $a + b * c$? In other words, how do we evaluate expressions?

infix: $a + b * c$
prefix: $+ a * b c$
postfix: $a b c * +$

Assuming that the language with the concrete infix syntax implements the “standard” rules for operator **precedence** and **associativity**, then all three expressions have the same *meaning* (i.e., the same semantics). The abstract syntax in this case can be represented using an **expression tree** whose structure captures the exact meaning of the expression. NOTE: an expression tree is not the same thing as a parse tree as an expression tree contains only terminal symbols:



Each subtree represents a subexpression and so the meaning of the expression must be unambiguous.

The above expression tree computes the expression $(a + (b * c))$. Is this correct?

Questions: What are the results of inorder, preorder and postorder traversals of the above expression tree?

Expression Notations

With infix notation, we have to establish **operator precedence rules** and **associativity rules**. Higher precedence operators **bind** operands “tighter” than lower precedence operators. By convention, multiplication/division operators have higher precedence than addition/subtraction operators. Parentheses override evaluation order.

What is the meaning of the following simple arithmetic expression?

$$3 + 4 * 5$$

The correct expression tree for $3 + 4 * 5$ is the one that results in the value 23 ($3 + (4 * 5)$) when the expression tree is evaluated by executing an **inorder** tree traversal.



Recall that an inorder traversal is a recursive left walk of the tree, starting at the root. In an inorder traversal, an expression subtree is evaluated as though its operator and operands were explicitly grouped by ().

So, in this case, an inorder evaluation of the tree on the left results in

$$(3 + 4) * 5 = 35$$

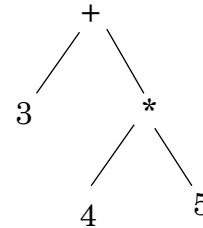
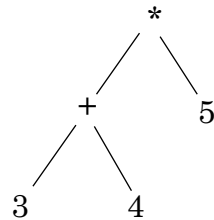
While the tree on the right results in:

$$3 + (4 * 5) = 23$$

When parsing expressions of a programming language, we want to construct our expression trees so that an inorder traversal yields the correct arithmetic result, based on operator precedence and associativity, automatically.

Expression Notations

What form of expressions result from **postorder** and **preorder** traversals of the following expression trees?.



Postfix expressions resulting from postorder traversals:

$$3\ 4\ +\ 5\ * = 7\ 5\ * = 35$$

$$3\ 4\ 5\ * + = 3\ 20 + = 23$$

Prefix expressions resulting from preorder traversals:

$$* + 3\ 4\ 5 = * 7\ 5 = 35$$

$$+ 3 * 4\ 5 = + 3\ 20 = 23$$

A distinct advantage of postfix notation is that it can easily be mechanically evaluated with the help of a **operand stack** data structure. So the following procedure can be used to evaluate an expression tree for a simple calculator:

- scan & parse an input expression and create an expression tree conforming to the proper precedence and associativity rules for arithmetic operators.
- perform a postorder traversal such that when a leaf operand node is visited, the value of the operand at the node is pushed onto the stack
- when a non-leaf binary operator node is visited, pop two values from the stack, apply the operator to the values, and push the result back on the stack. For unary operators, pop only one argument.
- at the end of the traversal, pop the value from the top of the stack and print it as the result.

Mixed Expression Notation

Most modern programming languages use a mixture of infix, prefix, and postfix arithmetic notation. In Java/C/C++, we might write the assignment statement:

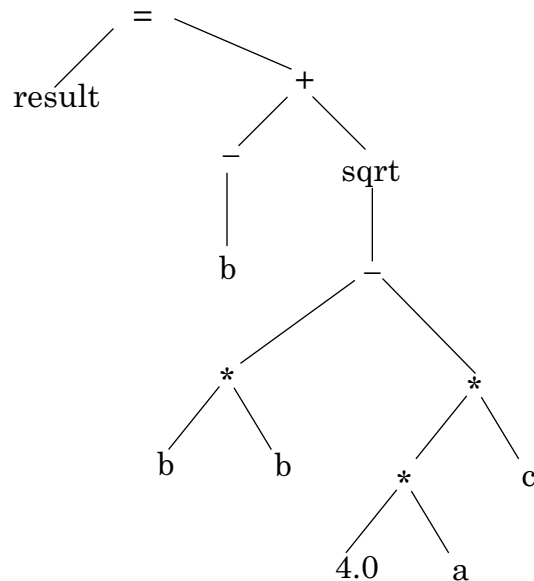
```
result = -b + sqrt(b*b - 4.0 * a * c);
```

Some operators have *arity 2* (binary) and some have arity 1 (unary). Unary operators have the highest precedence. Both the “unary minus” and the square root function are unary prefix operators. Unary operators often present a small difficulty when writing an expression in prefix or postfix notation. In the case of unary minus, we have to indicate the arity of the operator since it uses the same symbol as binary minus.

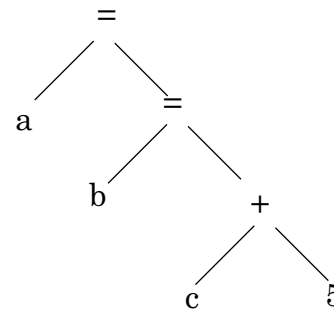
prefix: = result + ₋₁ b sqrt ₋₂ * b b * * 4.0 a c

postfix: result b ₋₁ b b * 4 a * c * ₋₂ sqrt + =

Here is the complete expression tree:



Note that the assignment operator has the lowest precedence and is right associative. Right associative operators cause expression trees to grow down to the right. For example: `a = b = c + 5`



Expression Interpretation

It is fairly easy to implement an interpreter for a simple arithmetic expression language.

You can easily implement a simple read-eval-print-loop (REPL) expression interpreter for an integer calculator:

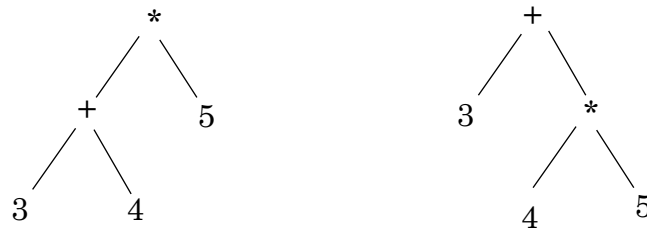
- **read** an arithmetic expression and translate it into an internal *expression tree*
- **evaluate** the expression tree
- **print** the result
- **loop** again

How would you implement an algorithm to evaluate (interpret) the following expression:

3 + 4 * 5

This expression is in infix notation, which means that the operators are placed in-between the operands of the expression. We read the expression left-to-right and construct an expression tree that unambiguously represents the expression semantically. Operands are placed at the leaf nodes of the tree and operators are placed at the non-leaf nodes.

Since there is more than one expression tree that can be formed from the above infix expression, how do we write a program to construct the correct one? We need a lexical analyzer to recognize the operators and operands and a syntax analyzer to construct the expression trees based on operator precedence and associativity of operators!



Special Postfix, Prefix and “Mixfix” Operators in Java/C/C++

Java and C/C++ define special prefix and postfix increment and decrement operators that can be applied to integral values (integers or pointers):

```
int x = 0;
printf("x is %d\n", x++); // post-increment results in 'x is 0'
printf("x is %d\n", ++x); // pre-increment results in 'x is 2'
```

Similarly,

```
int x = 0;
printf("x is %d\n", x--); // post-decrement results in 'x is 0'
printf("x is %d\n", --x); // pre-decrement results in 'x is -2'
```

Some expressions are syntactically legal, but semantically ambiguous and therefore undefined.

```
int x = 0;
x = ++x + x++; // legal syntax, but undefined semantics!!
```

A *mixfix* operator does not fit neatly into the prefix, infix, postfix classification, since it commonly has an *arity* greater than 2. For example, in functional languages, the **if-then-else** conditional is a **tertiary operator** with three operands rather than as a control-flow statement, as in an imperative language like Java/C/C++:

```
if (expr1) then expr2 else expr3 /* a conditional expression, not a statement */
```

If the boolean *expr1* evaluates to true, evaluate *expr2* as the result of the if-then-else expression, otherwise evaluate *expr3* and returns its value as the result. **Question: What is the type of a conditional expression?**

Java/C/C++ define a special *tertiary* conditional operator **?:** (called an inline conditional), that is used precisely this way.

```
(conditional-expr) ? then-expr-part : else-expr-part;
```

For example, the integer min function is usually written in C/C++ as:

```
int min(int a, int b) { int result = (a < b) ? a : b; return result; }
```

Expression Compilation Example

Suppose we have the following C expression, where position, initial and rate are all of type **float**.

```
float position, initial, rate;
position = initial + rate * 60;
```

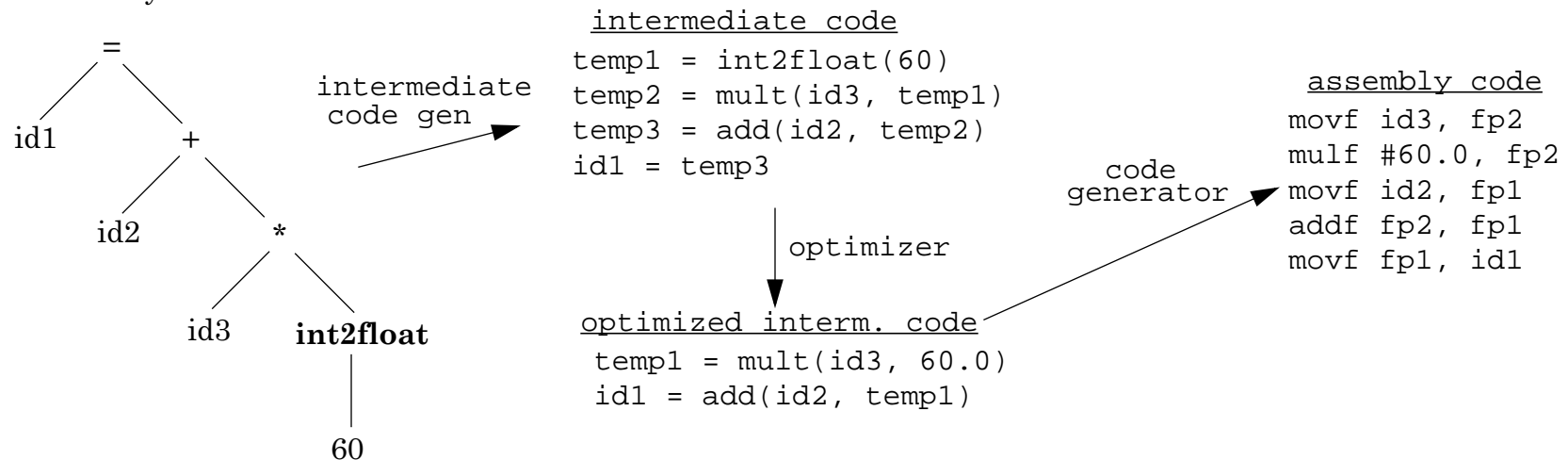
During lexical analysis, this sequence of characters (whitespace included) is analyzed, lexemes are matched against the lexical syntax definitions, and the following token-lexeme pairs are output from the lexical analyzer :

```
[ID, "position"] [ASSIGN, '=' ] [ID, "initial"] [PLUS, '+' ] [ID, "rate"] [MULT, '*' ] [NUM, 60] [SEMICOLON, ';' ]
```

For convenience, we can write the tokenized expression as follows (the ';' token is just a syntactic statement separator, which does not need to appear in the expression):

```
id1 = id2 + id3 * 60
```

Note that we need to account for the fact that an *implicit type conversion* is required to convert the integer value 60 to a floating point value, since the type of the expression result is float. Eventually, the expression is transformed to intermediate code, optimized code, and finally the assembly code required to compute the expression, after being assembled by an assembler into machine code.



Syntactic Ambiguity

Given a sequence of terminal symbols that we are trying to reduce to the start symbol of a grammar, what happens if you reach a point where there is more than one production rule that can be applied? Which one do you pick?

```
<expr> ::= <expr> + <expr> | <expr> * <expr> | a | b | c
```

A valid terminal string in the language recognized by this grammar is: $a + b * c$

Let us attempt to derive this string from the start symbol of the grammar using a top-down recursive descent approach. Note that we encounter an **ambiguity** at the first derivation—we have two alternative rules that we can apply. Which one do we pick?

```
<expr> => <expr> + <expr>  
      or  
      => <expr> * <expr>
```

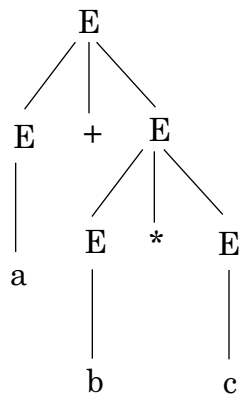
It turns out that it is possible to derive two different parse trees for the terminal string $a + b * c$ using this grammar. The first choice leads to a rightmost derivation and the second choice leads to a leftmost derivation. For a grammar to be **unambiguous**, both a leftmost and a rightmost derivation should lead to the same parse tree. If it is possible to derive a string with two or more different parse trees, then the grammar is said to be **ambiguous**.

We know that the correct choice depends on the operator precedence and associativity rules defined by the language in question. **But, operator precedence and associativity are *semantic rules of the language, not syntactic rules!*** A context-free grammar defines syntactic rules, not semantic rules. It would be very helpful if we could write an expression grammar in such a way that it would only let us derive parse trees that were not only syntactically correct, but semantically correct as well. Note however that it is not always possible to do this.

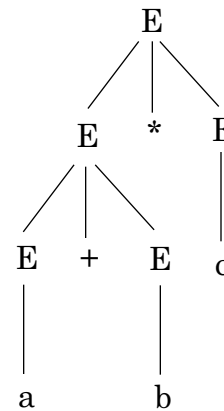
Since the goal of the syntax analyzer is to verify that the input is a syntactically correct program, and to generate a parse tree that can then be used by the intermediate code generator to eventually produce semantically correct machine code, we cannot have the parser generating *syntactically correct*, but semantically incorrect, parse trees for our program. That would be an unmitigated disaster!

Ambiguous Parse Trees

Parse Tree from a rightmost derivation starting from $\langle \text{expr} \rangle + \langle \text{expr} \rangle$



Parse Tree from a leftmost derivation starting with $\langle \text{expr} \rangle * \langle \text{expr} \rangle$



We would not like it if $a + b * c$ evaluated to $(a + b) * c$ in one part of our program and $a + (b * c)$ in another part, depending on which choice the compiler felt like making at compile-time.

Again, if a grammar allows two different parse trees to be generated for the same terminal string in the language, then the grammar is said to be ambiguous. It could be that the language itself is ambiguous, which forces the grammar to be ambiguous. In that case, the language syntax needs to be modified to either eliminate the ambiguity or otherwise ensure that the syntax rules of the language avoid the use of that syntax in a situation in which the ambiguity can lead to a semantically incorrect translation of the source program to machine code.

In rare cases, syntactic ambiguity can be tolerated as long as there is no semantic ambiguity.

Grammar Rules for Expressions

It is sometimes the case that ambiguity can be dealt with by reorganizing a grammar to eliminate there being a choice of rules to apply when reducing a terminal string in a bottom up parse. The trick is to rewrite the ambiguous grammar rules and add some more non-terminal symbols that introduce additional structure which eliminates the ambiguity. Effectively, we are defining syntax rules in such a way that they ensure that the correct semantic rules are applied. Note however that it is NOT always possible to do this, and the syntax analyzer has to be augmented with semantic actions.

In the case of infix integer arithmetic expressions, we can write a grammar to cope with precedence and associativity constraints.

- define a distinct non-terminal symbol for each operator precedence level
- define the RHS of a production rule to “enforce” the proper associativity rule
- define an additional non-terminal symbol for the smallest subexpressions

For example, a grammar for infix arithmetic expressions with two levels of precedence, and factors that consist of either parenthesized expressions or terminal symbols (i.e., identifier and number tokens). The choice of the words ‘term’ and ‘factor’ come from the mathematical use of these words, as in a “sum of terms” and a “multiplicative sequence of factors” e.g., sum $1+2+3\dots+n$ *terms* and $1*2*3*\dots*n$ of n *factors* (i.e., n factorial).

```
<expr> ::= <expr> + <term> | <expr> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= ( <expr> ) | id | num
```

Equivalently and for simplicity of notation, we use a modified BNF syntax that uses capital letters for non-terminals and omit the (tedious) use of < > in specifying non-terminals:

```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= ( E ) | id | num
```

How does this rewritten grammar eliminate the ambiguity that previously existed in the earlier expression grammar we discussed?

Unambiguous Expression Grammar

First, let's verify that we can only derive the expression $a + b * c$, in its tokenized form $id + id * id$, from the start symbol E , such that there is only one possible parse tree. Here are the left and right derivations. Draw the parse trees for both and convince yourself that there is only one possible parse tree for any arithmetic expression using the modified expression grammar.

Leftmost:

```
E => E + T
    => T + T
    => F + T
    => id + T
    => id + T * F
    => id + F * F
    => id + id * F
    => id + id * id
```

Rightmost:

```
E => E + T
    => E + T * F
    => E + T * id
    => E + F * id
    => E + id * id
    => T + id * id
    => F + id * id
    => id + id * id
```

Left and Right Recursive Grammar Rules

As we've seen, the following grammar for simple expressions is unambiguous. It is also **left recursive**, because in at least one rule, the leftmost symbol on the RHS of the production is the same as the non-terminal symbol on the LHS:

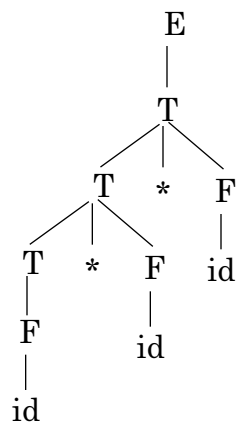
$$\begin{aligned} E &::= E + T \mid E - T \mid T \\ T &::= T * F \mid T / F \mid F \\ F &::= (E) \mid \mathbf{id} \mid \mathbf{num} \end{aligned}$$

A derivation of a terminal string using this grammar results in a parse tree whose corresponding expression tree is left associative.

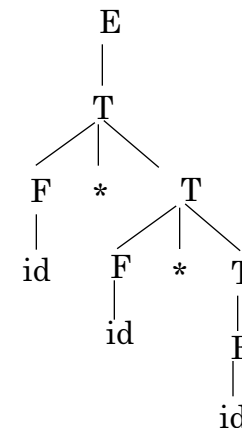
We can write a similar grammar that is **right recursive**. The resulting grammar is unambiguous, but results in a right recursive parse tree whose corresponding expression tree is right associative. **Can you think of any operators that are right associative?**

$$\begin{aligned} E &::= T + E \mid T - E \mid T \\ T &::= F * T \mid F / T \mid F \\ F &::= (E) \mid \mathbf{id} \mid \mathbf{num} \end{aligned}$$

Left recursive parse tree for $\text{id} * \text{id} * \text{id}$
with left-associativity



Right recursive parse tree for $\text{id} * \text{id} * \text{id}$
with right associativity



Yacc Expression Grammar

As you have seen in the basic integer calculator example, a Yacc grammar for expressions does not need to be structured in the previous manner because Yacc can be explicitly told what the precedence and associativity rules are, so that it enforces them automatically.

```
%left PLUS MINUS          /* lowest precedence*/
%left MULT DIV
%nonassoc UNARY           /* highest precedence */
...
%%
...
expr:    LPAREN expr RPAREN      { $$ = $2; }
        | expr MULT expr        { $$ = $1 * $3; }
        | expr DIV expr         { $$ = $1 / $3; }
        | expr PLUS expr        { $$ = $1 + $3; }
        | expr MINUS expr       { $$ = $1 - $3; }
        | MINUS expr %prec UNARY { $$ = -$2; }
        | num
```

If Yacc did not have this capability, then we **WOULD** have to write the rules explicitly using the additional rules for the non-terminal symbols `<term>` and `<factor>` as shown previously. For example, if you were to remove the `%left` directives in the above grammar, and recompile the `icalc.y` file, you would get:

```
% yacc -dv icalc.y
conflicts: 16 shift/reduce
```

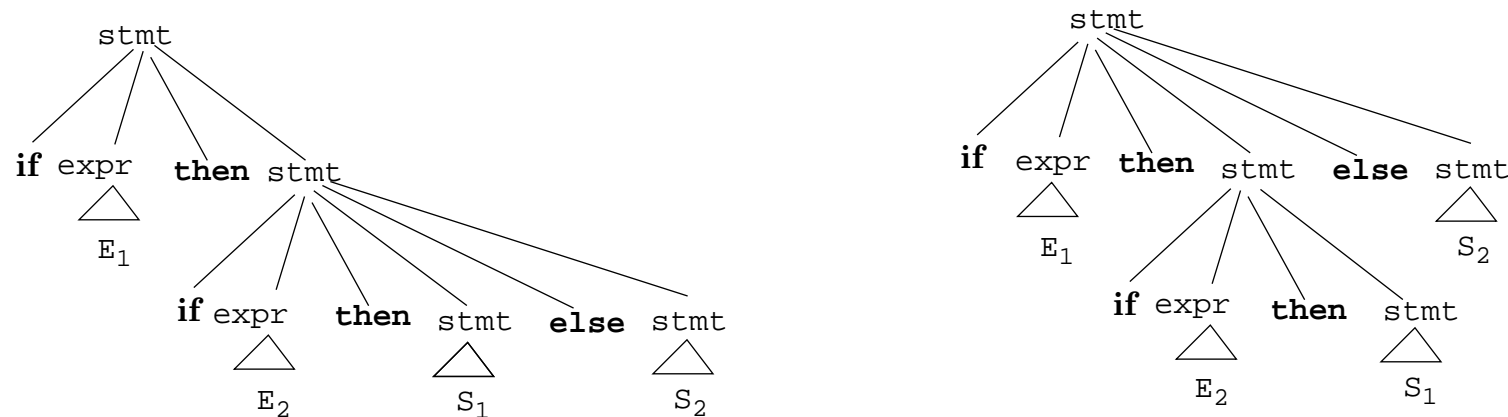
These shift-reduce conflicts are generated because of the ambiguity introduced by the fact that all of these operator tokens now have the same precedence & associativity. So either we use the explicit precedence and associativity directives provided by Yacc, or we restructure the grammar rules as shown previously so that the rules themselves enforce the precedence and associativity.

“Dangling Else” Ambiguity

Practically all imperative programming languages define an if-then-else conditional statement. The obvious grammar for specifying the if-then-else syntactic construct is:

```
stmt ::= if expr then stmt | if expr then stmt else stmt
```

This grammar is ambiguous and leads to two parse trees for the terminal string: if E1 then if E2 then S1 else S2



The first parse tree is the preferred one, since it groups the ‘else’ with the most recent ‘if’. Once again, this is a semantic rule, which can be syntactically handled by incorporating it into the grammar rules. Observe that we really have two different kinds of conditional statement: ‘if-then-else’ and ‘if-then’. The first one is an ‘if’ *matched* with an ‘else’ and the second one is an ‘if-then’ *unmatched* with an ‘else’. To avoid the dangling else problem, we need to enforce the structural condition in the grammar that only a matched statement can follow a ‘then’ terminal in the grammar rules. This forces an ‘else’ to always be matched with the closest ‘if’.

```
<stmt> ::= <matched> | <unmatched>
<matched> ::= if <expr> then <matched> else <matched>
<unmatched> ::= if <expr> then <stmt> | if <expr> then <matched> else <unmatched>
```

This new grammar only allows the parse tree that associates each ‘else’ with the closest previous ‘then’.

Shift-Reduce Conflicts

The previous approach to resolving this ambiguity using grammar rules is often inconvenient, and even problematic in practice. Practical tools like Yacc are able to cope with this type of ambiguity, which is an example of a **shift-reduce** conflict. Here is the Yacc grammar for the C if-statement:

```
%token IF ELSE
...
if_statement: IF '(' expr ')' statement
             | IF '(' expr ')' statement ELSE statement
```

This grammar rule suffers from the dangling else ambiguity; **HOWEVER**, Yacc is able to resolve it by always shifting the **ELSE** token onto the parser's stack, which turns out to be the right thing to do semantically because the **ELSE** token pairs with the closest **IF** token.

The default behavior of Yacc is always to shift when it encounters a shift-reduce conflict, but a warning is generated. For example, the following

```
% yacc -dv c-gram.y
conflicts: 1 shift/reduce
```

If we examine, the `y.output` error file generated by Yacc, we find the following shift-reduce conflict reported:

```
329: shift/reduce conflict (shift 344, red'n 187) on ELSE
state 329
      selection_statement : IF ( expr ) statement_      (187)
      selection_statement : IF ( expr ) statement_ELSE statement
```

An obvious way to resolve this problem is to introduce a new token into the language, **endif**, which would ensure that there is no conflict, but that would mean changing the definition of the C language, and it is too late for that!

```
%token IF ELSE ENDIF
...
if_statement: IF '(' expr ')' statement ENDIF
             | IF '(' expr ')' statement ELSE statement ENDIF
```

Using Precedence Rules to Resolve Shift-Reduce Conflicts

The shift-reduce conflict can also be eliminated by using a bit of cleverness with Yacc's precedence directive `%prec` and introducing a dummy token, `LOWER_THAN_ELSE`, that has lower precedence than `ELSE`.

```
%token IF ELSE
...
%nonassoc LOWER_THAN_ELSE /* dummy token */
%nonassoc ELSE
...
%%
...
if_statement: IF '(' expr ')' statement %prec LOWER_THAN_ELSE
             | IF '(' expr ')' statement ELSE statement
```

This technique explicitly forces the parser to shift an `ELSE` token onto the parser stack since it has higher precedence than the `LOWER_THAN_ELSE` dummy token, thereby paring an `ELSE` token with the most recent `IF` token.

This bit of cleverness is not really required since Yacc will shift by default when it encounters a shift-reduce conflict. But this does eliminate the warning about the shift-reduce conflict since the choice to shift is that of the language designer who wrote the grammar for the language.

Homework Assignment

To be completed by Wed, 7 Feb.

1. Rewrite the following expressions in **prefix** and **postfix** notation. The `sqrt` function is treated as a prefix operator of one argument.

```
x = a * -b + c
```

```
x = a * (b + c)
```

```
x = a * b + c * d
```

```
x = a * (b + c) * -d
```

```
x = (b/2 + sqrt((b / 2) * (b / 2) - a * c)) / a
```

2. Draw **expression tree** for each of the expressions in exercise 2.
3. Study the C language lexical (`scan.l`) and syntactic (`gram.y`) specifications handed out in class. Identify all the **terminal** and **non-terminal symbols**. What is the **start symbol** of the grammar?
4. By hand, using a **top-down recursive descent** approach, derive the complete parse tree for the simple C program below showing each non-terminal interior node and each terminal leaf node:

```
int main() { int x; if (x < 0) return 1; else return 0; }
```