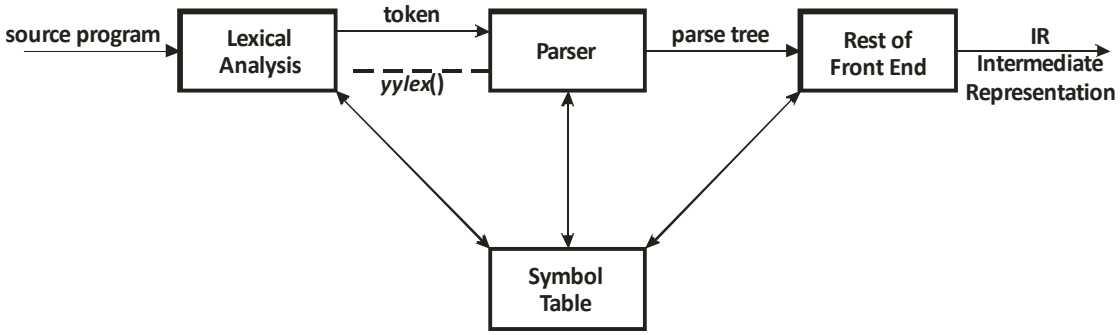

Figure 1. Position of the parser in the compiler model

Context free grammars:
A *context-free grammar* has four components:
1. A set of *tokens*, known as *terminal symbols* or *terminals*.
2. A set of *nonterminal symbols* or *nonterminals*.
3. A set of productions where each production consists of a nonterminal symbol, called the *left side* of the production, an arrow, and a sequence of tokens and/or nonterminal symbols, called the *right side* of the production.
4. A designation of one of the *nonterminal* symbol as the *start* symbol.

Notation
1. Terminal symbols are expressed in **bold** print.
2. Nonterminal symbols are *italicized*.

Example 1. Write a grammar for an arbitrarily long expression consisting of single digits separated by either the plus sign or the minus sign.

	left side		*right side*
1	*list*	→	*list* **+** *digit*
2	*list*	→	*list* **–** *digit*
3	*list*	→	*digit*
4	*digit*	→	**0**
5	*digit*	→	**1**
6	*digit*	→	**2**
7	*digit*	→	**3**
8	*digit*	→	**4**
9	*digit*	→	**5**
10	*digit*	→	**6**
11	*digit*	→	**7**
12	*digit*	→	**8**
13	*digit*	→	**9**

Table 1. Set of productions for the grammar of **Example 1.**

1. The set of terminal symbols (tokens), *T*={**+ - 0 1 2 3 4 5 6 7 8 9**}
2. The set of nonterminal symbols, *N*={*list digit*}

3. The set of productions *P*. Refer to table 1.
4. The starting nonterminal symbol *list*.

Example 2. Write a grammar for the language *micro*.

	left side		*right side*
1	*program*	→	**begin** *statement-list* **end**
2	*statement-list*	→	*statement*
3	*statement-list*	→	*statement-list* **;** *statement*
4	*statement*	→	**id :=** *expression*
5	*statement*	→	**read (** *id-list* **)**
6	*statement*	→	**write (** *expression-list* **)**
7	*id-list*	→	**id**
8	*id-list*	→	*id-list* **, id**
9	*expression-list*	→	*expression*
10	*expression-list*	→	*expression-list* **,** *expression*
11	*expression*	→	*primary*
12	*expression*	→	*expression additive-operator primary*
13	*primary*	→	**(** *expression* **)**
14	*primary*	→	**id**
15	*primary*	→	**intlit**
16	*additive-operator*	→	**+**
17	*additive-operator*	→	**-**

Table 2. Set of productions for the *micro* grammar of **Example 2.**

1. The set of terminal symbols (tokens), *T*={**begin end read write id intlit ; := ( ) + -**}
2. The set of nonterminal symbols,
   N={*program statement-list statement id-list expression-list expression primary additive-operator*}
3. The set of productions *P*. Refer to table 2.
4. The starting nonterminal symbol *program*.

Example 3. Write a grammar for expressions.

	left side		*right side*
1	*expression*	→	*expression* **+** *term*
2	*expression*	→	expression **–** term
3	*expression*	→	*term*
4	*term*	→	*term* ***** *factor*
5	*term*	→	*term* **/** *factor*
6	*term*	→	*factor*
7	*factor*	→	**(** *expression* **)**
8	*factor*	→	*id*

Table 3. Set of productions expressions

1. The set of terminal symbols (tokens), *T*={ **id ( ) + - * /**}
2. The set of nonterminal symbols,
   N={*expression*, *term*, *factor* }

3. The set of productions *P*. Refer to table 3.
4. The starting nonterminal symbol *expression*.

Example 3. Write an abbreviated grammar for expressions.

	left side		*right side*
1	*E*	→	*E* **+** *T*
2	*E*	→	*E* **–** *T*
3	*E*	→	*T*
4	*T*	→	*T* ***** *F*
5	*T*	→	*T* **/** *F*
6	*T*	→	*F*
7	*F*	→	**(** *E* **)**
8	*F*	→	**id**

Table 3. Set of productions expressions

5. The set of terminal symbols (tokens), *T*={ **id ( ) + - * /**}
6. The set of nonterminal symbols,
   N={*E*, *T*, *F* }
7. The set of productions *P*. Refer to table 3.
8. The starting nonterminal symbol *E*.

Derivations

Productions are rewriting rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.

Example: Consider the grammar of example 3 and derive **id+id*id**

Rule	*left side*		*Right side*
1	*E*	→	*E* **+** *T*
4		→	*E* **+** *T* ***** *F*
8		→	*E* + *T* ***** **id**
6		→	*E* + *F* ***** **id**
8		→	*E* + **id** ***** **id**
3		→	*T* + **id** ***** **id**
6		→	*F* + **id** ***** **id**
8		→	**id** + **id** ***** **id**

Table 4. Rightmost derivation of **id+id*id** from *E*

Consider $\alpha A\beta$ where α and β are strings of grammar symbols that can include both terminal and nonterminal symbols. *A* is a nonterminal symbol. Suppose $A \to \gamma$ is a production. We write $\alpha A\beta \Rightarrow \alpha\gamma\beta$. The symbol ⇒means "derives in one step." When $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$rewrites $\alpha_1$to $\alpha_n$we say $\alpha_1$derives α_n. The symbol $\overset{*}{\Rightarrow}$means "derives in zero or more steps." Likewise the symbol $\overset{+}{\Rightarrow}$means "derives in one or more steps."

1. $\alpha \overset{*}{\Rightarrow} \alpha$, for any string α.
2. If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \overset{*}{\Rightarrow} \gamma$,then $\alpha \overset{*}{\Rightarrow} \gamma$

Derivation order.

1. $\alpha \underset{lm}{\overset{*}{\Rightarrow}} \beta$ In leftmost derivations, the leftmost nonterminal in each sentential form is always chosen. Parsers that employ leftmost derivations are top-down and often use recursion. Such parsers are called LL meaning **L**eft-to-right scan of the input source and **L**eftmost derivations.
2. $\alpha \underset{rm}{\overset{*}{\Rightarrow}} \beta$ In rightmost derivations, the rightmost nonterminal in each sentential form is always chosen. Parsers that employ rightmost derivations are called bottom-up or LR parsers for **L**eft-to-right scan of the input source **R**ightmost derivation.

Parse Trees and Derivations.

	left side		*right side*
1	*E*	→	*E* **+** *E*
2	*E*	→	*E* ***** *E*
3	*E*	→	**-** *E*
4	*E*	→	**(** *E* **)**
5	*E*	→	**id**

Table 5. Ambiguous grammar for expressions

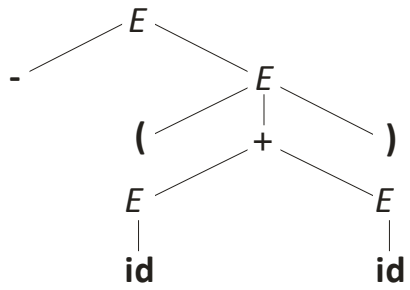

Figure 2. Parse tree for **–(id+id)**

	left side		*right side*
3	*E*	→	**-** *E*
4		→	**-** **(** *E* **)**
1		→	**-** **(***E* **+** *E***)**
5		→	**-** **(***E* **+ id)**
5		→	**- (id + id)**

Table 6. Derivation for figure 2.

Ambiguity.

A grammar is ambiguous if there exists more than one parse tree for some sentence in the grammar. A grammar is ambiguous if there is more than one rightmost or leftmost derivation of a sentence in the grammar.

Consider the grammar of table 4 and the sentence **id+id*id**

	left side		*right side*
1	*E*	→	*E* **+** *E*
2	*E*	→	*E* **+** *E* ***** *E*
5	*E*	→	*E* **+** *E* ***** **id**
5	*E*	→	*E* **+ id * id**
5	*E*	→	**id + id * id**

Table 7. Rightmost derivation of **id+id*id** number 1

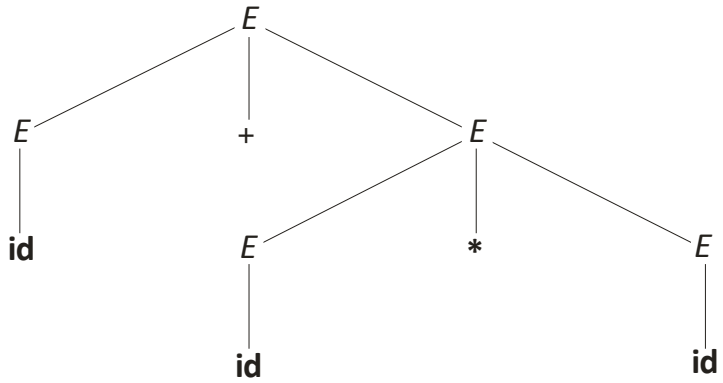

Figure 3. Rightmost derivation of **id+id*id** number 1

	left side		*right side*
2	*E*	→	*E* ***** *E*
5	*E*	→	*E* *** id**
1	*E*	→	*E* **+** *E* *** id**
5	*E*	→	*E* **+ id * id**
5	*E*	→	**id + id * id**

Table 6. Rightmost derivation of **id+id*id** number 2

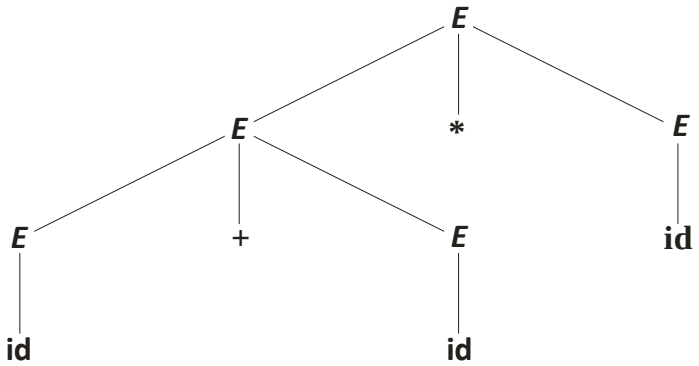

Figure 4. Rightmost derivation of **id+id*id** number 2

Yacc is a tool that will generate a parser given an LR(0) grammar.

Structure of a Yacc Grammar

… definition section …
%%
… rules section …
%%
… user subroutine section …

Symbol Conventions

Typically, non-terminal symbols are given in lowercase and terminal symbols are assigned all capital letters. For example, the rule:

program → program-head declarations program-body **.**

would be expressed for a yacc grammar as

```
program:
    program_head declarations program_body PERIOD
```

Note that hyphens have been changed to underscores to satisfy the C++ rules for identifiers and the period at the extreme right on the right hand side (RHS) of the rule has been changed to a capitalized spelling.

Definition Section

The definition section can contain

- *literal block*
  Declarations necessary for grammar actions and user subroutines are placed in the *literal block*. The *literal block* includes all **.h** files. A *literal block* is enclosed between **%{** and **%}** on separate lines as shown below.
  %{
  … C++ macro preprocessor definitions, declarations, and code …
  %}

- **%union** declarations
  The **%union** declaration associates terminal and non-terminal symbols with C-types. Identifiers defined in **%token** declarations and **%type** declarations are given specific types to be exploited in actions in the rules section. For example

```
%union {
    string* token;
    SList* slist;
}
%%
%token <token> ID
%type <slist> identifier_list
%%
identifier_list:
    ID
     {tfs << endl << "identifier_list -> ID(" << (*$1) << ")";
      $$=new SList;
      $$->Insert(*$1);
     }
identifier_list:
    identifier_list COMMA ID
     {tfs << endl << "identifier_list -> " << (*$1) << " , ID(" << (*$3) << ")";
      $$->Insert(*$3);
     }
```

  Notes:
  1. The symbolic type *token* is assigned the type *string**** in the **%union** declaration.
  2. The symbolic type *slist* is assigned the type *SList**** in the **%union** declaration.
  3. The **%token** declaration defines the terminal symbol *ID* to have the type assigned to the symbol type *token*. The terminal symbol *ID* now has type *string****.
  4. The **%type** declaration defines the non-terminal symbol *identifier_list* to have the type assigned to the symbol type *slist*. The non-terminal symbol now has type *SList****.
  5. The rule, *identifier-list* → **id**, expressed in the yacc grammar as
     ```
     identifier_list:
         ID
     ```
     has symbolic references to actual symbols represented by the terminal and non-terminal symbols in the rule. $1 refers to the first symbol on the RHS of the rule. $2, refers to the second symbol on the RHS of the rule. If there were five symbols on the RHS, $5 would refer to the fifth symbol on the RHS. $$ refers to the non-terminal on the LHS.

     In this case, because of prior **%union** and **%token** declarations, $1 has type *string**** and contains a pointer to the actual string recognized by the scanner and parser.

$$ has type SList* because of prior **%union** and **%type** declarations.

We first created a new *SList*, a string list, and, then we inserted the first identifier, a string, in the *SList*.

- **%token** declarations
  %token declarations are used to define terminal symbols. Terminal symbols defined by **%token** declarations are made available to a scanner implemented using ***lex***. File y.tab.h is created when ***yacc*** is invoked. File **y.tab.h** assigns positive integer values to terminal symbols defined using **%token** declarations. The values assigned to the terminal symbols are their token codes not the actual values represented by the token. A token is an integer code and a spelling. The spelling is the string of characters recognized by the scanner for that token.

  To make the strings recognized by the scanner available to the parser for the example above, you must add the following statement to the scanner.

  *yylval***.***token***=new** *string***(***yytext***);**

  Variable *yytext* has type **char*** and points to the most recent string of characters recognized by the scanner.

- **%type** declarations
  %type declarations perform much the same function as **%token** declarations with the difference that **%type** declarations are designed for non-terminal symbols whereas **%token** declarations are reserved for terminal symbols. **%type** declarations work in concert with **%union** declarations. **%union** declarations associate a C type with a symbolic type name. The symbolic type name is associated with a non-terminal symbol by a **%type** declaration.

Rules Section

The rules section contains

- grammar rules
  A rule of the grammar has a Left Hand Side (LHS) and a Right Hand Side (RHS). For example, consider the following expression grammar below with actions enclosed between **{** and **}**.
- actions containing C++ code

%union {
    double *real***;**
    string* *strlit*
}
%token PLUS MINUS STAR SLASH LPAREN RPAREN
%token <*real***> REALIT**
%token <*strlit***> ID**
%type **<***real***>** *expression term factor*
%%
statement:
    ID ASSIGN expression **{***cout* **<<** *endl* **<< (*$1) " := " << $3;}**
*expression***:**
    term **{$$=$1;}**
*expression***:**
    expression **PLUS** *term* **{$$=$1+$3;}**
expression:
    expression **MINUS** *term* **{$$=$1-$3;}**
*term***:**
    factor **{$$=$1;}**
*term***:**
    term **STAR** *factor* **{$$=$1*$3;}**
*term***:**
    term **SLASH** *factor* **{$$=$1/$3;}**
*factor***:**
    REALIT **{$$=$1;}**
*factor***:**
    LPAREN *expression* **RPAREN** **{$$=$2;}**
*factor***:**
    MINUS *factor* **{$$=-$1;}**

begin read(*x*); *x*:=*x*+2; write(*x*) end

```
Token:Code=267 Name=     BEGIN line=  1 col=  1 Spelling="begin"
Token:Code=269 Name=      READ line=  1 col=  7 Spelling="read"
Token:Code=263 Name=    LPAREN line=  1 col= 11 Spelling="("
Token:Code=273 Name=IDENTIFIER line=  1 col= 12 Spelling="x"
#007 IDENTIFIER_list->IDENTIFIER
Token:Code=264 Name=    RPAREN line=  1 col= 13 Spelling=")"
#005 READ ( IDENTIFIER_list )
#002 statement_list->statement
Token:Code=262 Name= SEMICOLON line=  1 col= 14 Spelling=";"
Token:Code=273 Name=IDENTIFIER line=  1 col= 16 Spelling="x"
Token:Code=265 Name=    ASSIGN line=  1 col= 17 Spelling=":="
Token:Code=273 Name=IDENTIFIER line=  1 col= 19 Spelling="x"
#014 primary->IDENTIFIER
Token:Code=259 Name=      PLUS line=  1 col= 20 Spelling="+"
#016 addop-> +
Token:Code=272 Name=    INTLIT line=  1 col= 21 Spelling="2"
#015 primary->INTLIT
#012 expression->primary addop primary
#004 IDENTIFIER := expression
#003 statement_list->statement_list ; statement
Token:Code=262 Name= SEMICOLON line=  1 col= 22 Spelling=";"
Token:Code=270 Name=     WRITE line=  1 col= 24 Spelling="write"
Token:Code=263 Name=    LPAREN line=  1 col= 29 Spelling="("
Token:Code=273 Name=IDENTIFIER line=  1 col= 30 Spelling="x"
#014 primary->IDENTIFIER
Token:Code=264 Name=    RPAREN line=  1 col= 31 Spelling=")"
#011 expression->primary
#009 expression_list->expression
#006 WRITE ( expression_list )
#003 statement_list->statement_list ; statement
Token:Code=268 Name=       END line=  1 col= 33 Spelling="end"
#001 program->BEGIN statement_list END
```

```
rm mcrpar.cpp
rm mcrlex.cpp
rm *.o
rm mcr
make -f makemicro
```

```
#-----------------------------------------------------------------------
# File makemcr creates a micro language compiler
#-----------------------------------------------------------------------
# Author: Thomas R. Turner
# E-Mail: trturner@uco.edu
# Date:   January, 2012
#-----------------------------------------------------------------------
# Copyright January, 2012 by Thomas R. Turner.
# Do not reproduce without permission from Thomas R. Turner.
#-----------------------------------------------------------------------
#-----------------------------------------------------------------------
# Object files
#-----------------------------------------------------------------------
obj  =          mcrpar.o \
                mcrlex.o \
                mcr.o
#-----------------------------------------------------------------------
# Bind the subset Pascal Scanneer
#-----------------------------------------------------------------------
mcr:            ${obj}
                g++ -o mcr ${obj} -lm -ll
#-----------------------------------------------------------------------
# File mcr.cpp processes command line arguments
#-----------------------------------------------------------------------
mcr.o:          mcr.cpp mcrlex.h
                g++ -c -g mcr.cpp
#-----------------------------------------------------------------------
# File mcrlex.cpp is the lex-generated scanner
#-----------------------------------------------------------------------
mcrlex.cpp:   mcrlex.l mcrlex.h
                lex mcrlex.l
                mv lex.yy.c mcrlex.cpp
#-----------------------------------------------------------------------
#-----------------------------------------------------------------------
mcrlex.o:       mcrlex.cpp mcrlex.h
                g++ -c -g mcrlex.cpp
#-----------------------------------------------------------------------
# Create files mcrpar.cpp and mcrtkn.h from file mcrpar.y
#-----------------------------------------------------------------------
mcrtkn.h      \
mcrpar.cpp:  mcrpar.y
                    yacc -d -v mcrpar.y
                    mv y.tab.c mcrpar.cpp
                    mv y.tab.h mcrtkn.h
#-----------------------------------------------------------------------
# Compile the parser mcrpar.y
#-----------------------------------------------------------------------
mcrpar.o:       mcrpar.cpp mcrpar.h
                g++ -c -g mcrpar.cpp
#-----------------------------------------------------------------------
```