

# Computer Engineering 175

## Phase II: Syntax Analysis

“Grammar, which knows how to control even kings.”  
Molière, *Les Femmes Savantes*

### 1 Overview

In this assignment, you will write a recursive-descent parser for the Simple C language. This assignment is worth 15% of your project grade. Your program is due at 11:59 pm, Sunday, January 27th.

### 2 Syntactic Structure

The following rules constitute the syntax rules for Simple C:

<i>translation-unit</i>	→	$\epsilon$   <i>global-declaration translation-unit</i>   <i>function-definition translation-unit</i>
<i>global-declaration</i>	→	<i>specifier global-declarator-list ;</i>
<i>global-declarator-list</i>	→	<i>global-declarator</i>   <i>global-declarator , global-declarator-list</i>
<i>global-declarator</i>	→	<i>pointers id</i>   <i>pointers id ( parameters )</i>   <i>pointers id [ integer ]</i>
<i>pointers</i>	→	$\epsilon$   <i>* pointers</i>
<i>specifier</i>	→	<b>char</b>   <b>int</b>   <b>double</b>
<i>function-definition</i>	→	<i>specifier pointers id ( parameters ) { declarations statements }</i>
<i>parameters</i>	→	<b>void</b>   <i>parameter-list</i>
<i>parameter-list</i>	→	<i>parameter</i>   <i>parameter , parameter-list</i>
<i>parameter</i>	→	<i>specifier pointers id</i>
<i>declarations</i>	→	$\epsilon$   <i>declaration declarations</i>
<i>declaration</i>	→	<i>specifier declarator-list ;</i>
<i>declarator-list</i>	→	<i>declarator</i>   <i>declarator , declarator-list</i>

<i>declarator</i>	→	<i>pointers</i> <b>id</b>
		<i>pointers</i> <b>id</b> [ <b>integer</b> ]
<i>statements</i>	→	$\epsilon$
		<i>statement</i> <i>statements</i>
<i>statement</i>	→	{ <i>declarations</i> <i>statements</i> }
		<b>return</b> <i>expression</i> ;
		<b>while</b> ( <i>expression</i> ) <i>statement</i>
		<b>if</b> ( <i>expression</i> ) <i>statement</i>
		<b>if</b> ( <i>expression</i> ) <i>statement</i> <b>else</b> <i>statement</i>
		<i>expression</i> = <i>expression</i> ;
		<i>expression</i> ;
<i>expression</i>	→	<i>expression</i>    <i>expression</i>
		<i>expression</i> && <i>expression</i>
		<i>expression</i> == <i>expression</i>
		<i>expression</i> != <i>expression</i>
		<i>expression</i> <= <i>expression</i>
		<i>expression</i> >= <i>expression</i>
		<i>expression</i> < <i>expression</i>
		<i>expression</i> > <i>expression</i>
		<i>expression</i> + <i>expression</i>
		<i>expression</i> - <i>expression</i>
		<i>expression</i> * <i>expression</i>
		<i>expression</i> / <i>expression</i>
		<i>expression</i> % <i>expression</i>
		! <i>expression</i>
		- <i>expression</i>
		& <i>expression</i>
		* <i>expression</i>
		<b>sizeof</b> ( <i>specifier pointers</i> )
		( <i>specifier pointers</i> ) <i>expression</i>
		<i>expression</i> [ <i>expression</i> ]
		<b>id</b> ( <i>expression-list</i> )
		<b>id</b> ( )
		<b>id</b>
		<b>real</b>
		<b>integer</b>
		<b>string</b>
		( <i>expression</i> )
<i>expression-list</i>	→	<i>expression</i>
		<i>expression</i> , <i>expression-list</i>

### 3 Assignment

You will write a parser for Simple C, using the given grammar as a starting point. Unfortunately, the given expression grammar is ambiguous. Therefore, you must first disambiguate the grammar without changing the language accepted. To help you in your task, Table 1 shows the precedence and associativity of operators in Simple C.

To illustrate that your parser is working correctly, you will write the operator, as shown in Table 1, used in each expression to the **standard output** after you have matched that expression. For example,  $a + b * c$  would generate mul and then add because the multiplication is done before the addition. In contrast,  $a + b - c$  would generate add and then sub since addition and subtraction have the same precedence but are left associative.

Operators	Associativity	Arity	Output
[ ]	left	binary	index
( <i>specifier pointers</i> ) & * ! - sizeof	right	unary	cast addr deref not neg sizeof
* / %	left	binary	mul div rem
+ -	left	binary	add sub
< > <= >=	left	binary	ltn gtn leq geq
== !=	left	binary	eql neq
&&	left	binary	and
	left	binary	or

Table 1: Operator associativity and precedence listed from highest to lowest.

Your program will only be given **syntactically correct** programs as input. However, it is strongly advised that you test your program against syntactically incorrect programs as a way of finding errors in your implementation.

## 4 Hints

First, you will need to modify your lexical analyzer to return separate tokens for each keyword and operator. The parser will call the lexer when it needs a token. For simplicity, use the ASCII character value of a single-character token (e.g., '+', '-', '\*'), and create a `#define` or `enum` for multi-character tokens such as identifiers, integers, reals, strings, keywords, and operators (e.g., `ID`, `INTEGER`, `REAL`, `RETURN`, `AND`).

After disambiguating your grammar, you will need to eliminate left recursion and left-factor the grammar in order to implement a recursive-descent parser. The first step involves the rules for expressions. You can simply extend the example given in the textbook, writing one function for each level of precedence. Start by writing a parser just for expressions (i.e., the start symbol would be *expression*) and test it on expressions.

Left-factoring needs to be performed at several obvious places (e.g., *declarator*) and two non-obvious places. At the global level, we cannot immediately tell if we have a function definition as opposed to a global declaration. This problem can be solved by left-factoring the grammar, combining the rules for *function-definition*, *global-declaration*, and *global-declarator-list*. In an expression, a left parenthesis could begin either a cast expression or a parenthesized expression. This problem can be solved by carefully combining the rules for prefix and array operators to expose the left factoring. Essentially, if the next token after the left parenthesis is not a type specifier, the expression can be treated as a parenthesized expression followed by optional array operators.