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 20% of your project grade. Your program is due at 11:59 pm, Sunday, January 23rd.

2 Syntactic Structure

The following rules constitute the syntax rules for Simple C:

translation-unit	→	ϵ
	|	*type-definition translation-unit*
	|	*global-declaration translation-unit*
	|	*function-definition translation-unit*
type-definition	→	**struct id** { *declaration declarations* } ;
global-declaration	→	*specifier global-declarator-list* ;
global-declarator-list	→	*global-declarator*
	|	*global-declarator* , *global-declarator-list*
global-declarator	→	*pointers* **id**
	|	*pointers* **id** ( )
	|	*pointers* **id** [**num**]
	|	*pointers* (∗ **id**) ( )
pointers	→	ϵ
	|	∗ *pointers*
specifier	→	**int**
	|	**char**
	|	**struct id**
function-definition	→	*specifier pointers* **id** (*parameters*) { *declarations statements* }
parameters	→	**void**
	|	*parameter-list*
parameter-list	→	*parameter*
	|	*parameter* , *parameter-list*
parameter	→	*specifier pointers* **id**
	|	*specifier pointers* (∗ **id**) ( )
declarations	→	ϵ
	|	*declaration declarations*
declaration	→	*specifier declarator-list* ;

declarator-list → *declarator*
| *declarator* , *declarator-list*

declarator → *pointers* **id**
| *pointers* **id** [**num**]
| *pointers* (∗ **id**) ( )

statements → ϵ
| *statement statements*

statement → { *declarations statements* }
| **return** *expression* ;
| **while** (*expression*) *statement*
| **for** (*assignment* ; *expression* ; *assignment*) *statement*
| **if** (*expression*) *statement*
| **if** (*expression*) *statement* **else** *statement*
| *assignment* ;

assignment → *expression* = *expression*
| *expression*

expression → *expression* || *expression*
| *expression* && *expression*
| *expression* == *expression*
| *expression* != *expression*
| *expression* <= *expression*
| *expression* >= *expression*
| *expression* < *expression*
| *expression* > *expression*
| *expression* + *expression*
| *expression* - *expression*
| *expression* ∗ *expression*
| *expression* / *expression*
| *expression* % *expression*
| (*specifier pointers*) *expression*
| ! *expression*
| - *expression*
| & *expression*
| ∗ *expression*
| **sizeof** *expression*
| **sizeof** (*specifier pointers*)
| *expression* [*expression*]
| *expression* (*expression-list*)
| *expression* ( )
| *expression* . **id**
| *expression* -> **id**
| **id**
| **num**
| **string**
| **character**
| (*expression*)

expression-list → *expression*
| *expression* , *expression-list*

Operators	Associativity	Arity	Output
`[] () . ->`	left	binary	`index call dot arrow`
`(` *specifier pointers* `) & * ! - 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.

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.

Your program will only be given ***syntactically correct*** programs as input. However, it is strongly advised that you should 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 an `enum` for multi-character tokens such as identifiers, numbers, keywords, and operators (e.g., `ID`, `NUM`, `RETURN`, `AND`).

To implement a recursive-descent parser, you will need to eliminate left recursion and left-factor the given grammar. 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.

The only tricky step occurs when encountering a left parenthesis, which could begin either a type cast or a parenthesized expression. However, in former case, but not the latter, the next token must be a type specifier. Therefore, you will need to add an extra token of lookahead to your parser.

Left-factoring needs to be performed at several obvious places (e.g., *declarator*). Also, at the global level, we cannot immediately tell if we have a type definition, function definition, or a global declaration. This problem can be solved by left-factoring the grammar, combining the rules for *type-definition*, *function-definition*, *global-declaration*, and *global-declarator-list*.