Computer Engineering 175
Phase III: Symbol Table Construction

"Be angry when you will, it shall have scope."
Shakespeare, *Julius Caesar*, Act IV

1 Overview

In this assignment, you will augment your parser to construct a symbol table for the Simple C language. This assignment is worth 20% of your project grade. Your program is due at 11:59 pm, Sunday, February 10th.

2 Semantic Checking

To perform most semantic checks, your compiler must record information about an identifier (such as its type) at the time of its declaration and then lookup that information when the identifier is used. The symbol table is the central repository for all such information.

In Simple C, a type consists of a type specifier (`char`, `int`, or `double`) along with optional declarators ("function returning *T*," "array of *T*," and "pointer to *T*"). To manage identifiers, Simple C uses static nested scoping: scopes can be nested hierarchically and when an identifier is used after being declared, the closest or innermost declaration is used.

A local variable or parameter may be declared at most once in a scope. However, a global variable or function may be declared multiple times in the global scope so long as all of the declarations are identical (including any parameters, if specified). However, a function may be defined only once. Calling a function without a previous declaration results in the function being ***implicitly declared*** in the global scope as "function returning `int`" with an unspecified parameter list. The same identifier may be used to declare different objects in different scopes.

3 Semantic Rules

3.1 Translation units

translation-unit → ϵ
| *global-declaration translation-unit*
| *function-definition translation-unit*

The scope of the translation unit (i.e., file) begins at the top of the file before any *global-declaration* or *function-definition* and persists until the end of the file.

3.2 Function definitions

function-definition → *specifier pointers* **id** (*parameters*) { *declarations statements* }

specifier → **char**
| **int**
| **double**

pointers → ϵ
| $*$ *pointers*

The function is both ***declared*** and ***defined*** in the current translation unit. The scope of the function begins immediately after the identifier and persists until the end of *statements*. The type of the function is "function returning *T*," where *T* has a specifier of *specifier* along with any pointer declarators specified as part of *pointers*.

The function must not have been previous defined [E1] and any previous declaration must be identical [E3]. (If a previous declaration was implicit, then the parameters are not checked.)

3.3 Parameters

parameters → **void**
| *parameter-list*

parameter-list → *parameter*
| *parameter* , *parameter-list*

parameter → *specifier pointers* **id**

Each parameter is declared in the current scope, and must not have been previously declared in the current scope [E2]. The type of the parameter is that of *specifier* along with any pointer declarators specified as part of *pointers*.

3.4 Declarations

global-declaration → *specifier global-declarator-list* ;

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

global-declarator → *pointers* **id**
| *pointers* **id** [**integer**]
| *pointers* **id** (*parameters*)

declarations → ϵ
| *declaration declarations*

declaration → *specifier declarator-list* ;

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

declarator → *pointers* **id**
| *pointers* **id** [**integer**]

Each variable is declared in the current scope. If the variable is a global variable, then any previous declaration must be identical [E3]. If it is a local variable, then the variable must not be previously declared in the current scope [E2]. The type of the variable is that of *specifier* along with any specified pointer and array declarators.

Each function is declared in the global scope. The type of the function is "function returning *T*," where *T* has a specifier of *specifier* along with any pointer declarators specified as part of *pointers*. A new scope begins immediately after the identifier and persists until the end of *parameters*.

3.5 Statements

statement → { *declarations statements* }

The scope of the block begins before the *declarations* and persists until the end of the *statements*.

3.6 Expressions

primary-expression → **id** (*expression-list*)
| **id** ( )
| **id**

When used as a variable, the identifier must be declared in the current scope or in an enclosing scope [E4]. If the identifier in a function call expression is undeclared, then it is implicitly declared in the global scope to have type "function returning `int`" with an unspecified list of parameters. (For this assignment, you do not need to check if the identifier is a function or a variable.)

4 Assignment

You will design and implement a symbol table for Simple C by augmenting your parser, using the given rules as a guide. You will only be given ***syntactically legal programs as input***. You compiler should indicate any errors by writing the appropriate error messages to the ***standard error***:

E1. `redefinition of '`*name*`'`

E2. `redeclaration of '`*name*`'`

E3. `conflicting types for '`*name*`'`

E4. `'`*name*`' undeclared`

Each error message must be prefixed with the line number in the form "`line` *number*`: `". The line number may vary slightly from the examples. Each error may cause subsequent, cascading errors. Any messages written to the ***standard output*** will be ignored. Any subsequent declaration or definition always replaces any previous declaration or definition, even it erroneous. Note that replacing an implicit function declaration with an explicit one is not erroneous so long as the return type of the explicit declaration is `int` since the parameters are not checked. Also, global objects cannot yield error [E2] only [E3], and that local objects cannot yield error [E3] only [E2]. If multiple errors apply to the same declaration, issue only the first error message listed above.

5 Hints

The Standard Template Library provides several useful data structures. You will probably find it easiest to model the nesting of scopes using a stack. (Either an explicit stack can be used, or each scope can maintain a pointer to its enclosing scope, thus forming a linked-list of scopes.) Each scope itself can be implemented using a map that associates the name of an identifier to information that includes its type. Such an object is called a symbol, which usually has a name, type, and any other necessary information. A type can be modeled as a specifier and the number of levels of indirection due to pointer declarators. Develop classes for types (`Type.cpp`), symbols (`Symbol.cpp`), and scopes (`Scope.cpp`). Finally, develop a separate module (`checker.cpp`) for performing the semantic checks.