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 6th.

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 `struct`) along with optional declarators ("function returning *T*," "callback 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. However, a function may be defined only once. The same identifier may be used to declare different objects in different scopes.

Each structure in Simple C has its own scope for its fields. Two different structures may have a field with a common name, but any particular structure has at most one field with a given name. Furthermore, the names of structures (called structure tags) are kept in a separate namespace from that of other identifiers. Thus, it is possible to have a structure with the same name as a variable. A structure name may be used to declare pointers to the structure before the structure itself is completely defined. Such a use of the structure name is called a use of an ***incomplete type***. A structure name can be defined only once.

3 Semantic Rules

3.1 Translation units

translation-unit → ϵ
| *type-definition translation-unit*
| *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 *type-definition*, *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 → **int**
| **char**
| **struct id**

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 [E2], ignoring the parameters as any previous declaration will have an unspecified parameter list. Additionally, if *specifier* is a structure type, then *pointers* must be non-empty [E5].

3.3 Parameters

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

parameter → *specifier pointers* **id**
| *specifier pointers* (∗ **id**) ( )

Each parameter is declared in the current scope, and must not have been previously declared in the current scope [E3]. The type of a scalar parameter is that of *specifier* along with any pointer declarators specified as part of *pointers*. If *specifier* is a structure type, then *pointers* must be non-empty [E5]. The type of a function pointer parameter is "callback returning *T*," where *T* has a specifier of *specifier* along with any pointer declarators. If *T* is a structure type, the *pointers* must be non-empty [E5].

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

declarations → ϵ
| *declaration declarations*

declaration → *specifier declarator-list* ;

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

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

Each variable is declared in the current scope. If the variable is a global variable, then any previous declaration must be identical [E2]. If the variable is a local variable, then the variable must not be previously declared in the current scope [E3]. The type of the variable is that of *specifier* along with any specified pointer and array declarators. If *specifier* is a structure type, then the type must be complete or *pointers* must be non-empty [E6].

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*. Any previous declaration must be identical [E2], ignoring any parameters specified as part of a previous function definition. If *specifier* is a structure type, then *pointers* must be non-empty [E5].

The type of a function pointer variable is "callback returning *T*," where *T* has a specifier of *specifier* along with any pointer declarators specified as part of *pointers*. If *T* is a structure type, then *pointers* must be non-empty [E5]. The restrictions applied to ordinary variables on their redeclaration also apply to these variables.

3.5 Structure definitions

type-definition → **struct id** { *declaration declarations* } ;

The scope of the structure begins immediately before the first *declaration* and persists until immediately after the *declarations*, at which point the type definition is considered complete. The structure must not have been previously defined [E1].

3.6 Statements

statement → { *declarations statements* }

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

3.7 Expressions

primary-expression → **id**

The identifier must be declared in the current scope or in an enclosing scope [E4].

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***. Your compiler should indicate any errors by writing the appropriate error messages to the ***standard error***:

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

E2. `conflicting types for '`*name*`'`

E3. `redeclaration of '`*name*`'`

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

E5. `pointer type required for '`*name*`'`

E6. `'`*name*`' has incomplete type`

Each error message must be prefixed with the line number in the form "`line` *number*`: `". The line number may vary slightly from the examples. Any messages written to the ***standard output*** will be ignored. A function definition always replaces any previous declaration or definition, even if erroneous. However, an erroneous redeclaration of a function or variable is discarded and the original declaration kept. Note that global objects cannot yield error [E3] only [E2], and that local objects cannot yield error [E2] only [E3]. If multiple error messages apply, issue only the first error message listed.

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, the number of levels of indirection due to pointer declarators, and a declarator (scalar, array, or function). 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.