UNION COLLEGE

# Assignment #2: Building the Symbol Table Stack

Course: *CSC-375 Compiler Design* – Professor: *Aaron Cass*
Due date: *DUE DATE*

### Objectives

- Deepen your understanding of the Rust programming language
- Learn how to construct and manage symbol tables in a compiler
- Thoroughly test the Symbol Table Stack to assure functionality

**An explanation of symbol tables:** In compiler design, a symbol table is a data structure used by a compiler to keep track of scope and binding information about names. Symbol tables are essential for semantic analysis, which checks for correct use of variables, functions, classes, and other entities. They allow the compiler to quickly verify and retrieve information about these entities, facilitating tasks such as type checking, memory allocation, and scope resolution. In this assignment, you will build the structure to manage a stack of symbol tables, representing different levels of scope within a program.

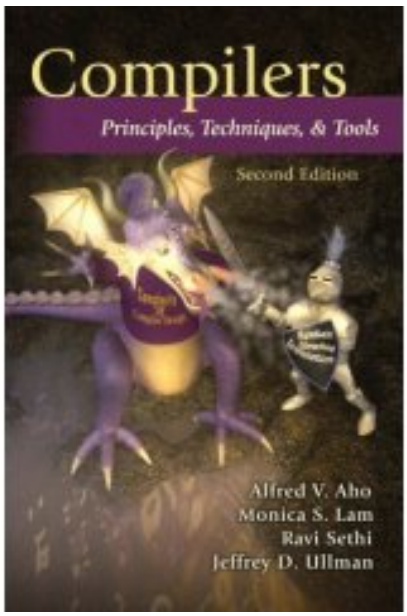


Figure 1: A more comprehensive explanation of the symbol table and the symbol table stack can be found in sections 1.2.7 (page 11) and 2.7 (page 85) of the dragon book.

## Symbol Table Stack Structure

### lib.rs:

This file defines the modules related to the symbol table stack and makes them available for other modules to see. It can be found in the *frontend/* folder. All modules each have documentation in this file to show what the module is used for. The symbol table stack involves several submodules: the symbol table, symbol info, and the stack itself.

You don't have to do anything to this file.

### core.rs:

This file contains the core logic for generating the symbol table stack and routing AST nodes to appropriate handlers. While the structure for **SymbolValue, SymbolInfo, SymbolTable, and SymbolTableStack** are given, additional functions for utilizing these struct's and enum's will need to be implemented. You will also need to implement the following core functions:

- **gen_sym_table_stack**: Drives the symbol table stack generation process and returns the original AST and the generated symbol table stack, or errors if any.
- **sym_table_stack_router**: Routes the proper top-level expression for an ASTNode.

### block.rs:

This file contains the implementation of block statements (e.g., functions, loops) within the symbol table stack. You will need to implement the following functions:

- **sym_table_fn**: Adds a function type to the current scope.
- **sym_table_struct**: Adds a struct type to the current scope.
- **sym_table_enum**: Adds an enum type to the current scope.
- **sym_table_while**: Adds a while loop to the current scope.
- **sym_table_do_while**: Adds a do-while loop to the current scope.
- **sym_table_for**: Adds a for loop to the current scope.
- **sym_table_if_else**: Adds an if-else block to the current scope.
- **sym_table_switch**: Adds a switch statement to the current scope.

**statement.rs:**

This file contains functions that define statements and variables within blocks. The following functions need to be implemented:

- **sym_table_init**: Adds a new variable into the current scope.
- **sym_table_assign**: Assigns a new value to a variable in the current scope.

**base_tests.rs:**

This file contains test definitions for base case tests. Your job is to implement and expand upon these tests to assure proper scope management.

**combination_tests.rs:**

This file should be used as a test suite for combination tests of elements tested in **base_tests.rs**. Make sure to comprehensively test these combinations.

**edge_tests.rs:**

This file should be used as a test suite for boundary cases. This suite should comprehensively test the addition and functionality of SyntaxElements within the Symbol Table Stack.

**error_tests.rs:**

This file should be used as a test suite for error handling. Comprehensively test that every instance that may raise an error is handled properly.

### Follow these guidelines to build the symbol table stack

1. Do not change any of the function signatures, i.e., leave all parameters, parameter types, and return types as you found them.
2. Follow all instructions given in the comments where they exist, and take any potential hints into consideration when making your design.
3. Remember you are managing a stack of symbol tables, each representing a different scope. Ensure that your implementation correctly handles scope transitions.
4. Use the **SymbolInfo**, **SymbolTable**, and **SymbolTableStack** structs effectively to maintain the integrity of the symbol table stack.
5. Maintain clear and consistent error handling. Use **Result** and **ErrorType** to propagate errors appropriately.

### Cheat sheet for manipulating symbol tables

It might not be obvious how to manipulate symbol tables effectively. Here are some tips on how to do this.

1. When adding a new symbol to a table, ensure you use the **add** method of **SymbolTable**. For example, if you want to add a function, create a **SymbolInfo** object and use **SymbolTable::add**.
2. When transitioning scopes, use the **push** and **pop** methods of **SymbolTableStack** to manage the stack of tables. For example, when entering a new scope, push a new table onto the stack.
3. To retrieve symbols from the current scope, use the **get** method of **SymbolTable**. Ensure you handle cases where the symbol might not be found.
4. Maintain clear and consistent error handling. Use **Result** and **ErrorType** to propagate errors appropriately.