University of Toronto

CSC 488 / CSC2107 Compilers and Interpreters **Winter 2022**

Assignment 4: MiniC Symbol Table and Semantic Checking (12%)

Due: Feb. 28, 2022

In this assignment, you will learn how to build the MiniC symbol table and check semantics in `C++`.

Semantic Checking Workflow in MiniC (cont'd from Assignment 3 Workflow)

After generating the AST, in `main.cpp`, the object `verifier` (of class `VerifyAndBuildSymbols`) visits each AST node starting from `prog`. After visiting each node, check the node's relationship with respect to the language, then build the symbol table and report semantics errors in `VerifyAndBuildSymbols.cpp`. In MiniC, we are specifically interested in the Function symbol table that maps the name to a `FuncSymbolEntry` and the Variable symbol table that maps the name to a `VarSymbolEntry`.

Data Structures for Semantic Checking

The related data structure of A4 is shown in Table 1. Your task is to:

- Define and implement member functions of SymbolTable in `SymbolTable.h`.
- Fill in the Function Symbol Table for the `Program` node and Variable Symbol Tables for `ScopeStatement` nodes and `program` node.
- Fill in error messages in `Errors` vector in class `VerifyAndBuildSymbols`. The error message format should follow `a4_errors.txt`. In `VerifyAndBuildSymbols.cpp`, there are detailed hints that show which errors should be where.

File	**Description**
`SymbolTable.h`	The definitions of Function & Variable Symbol table. (You can ignore `llvm::Value` in this assignment)
`Program.h`	The root of the program, which contains the pointer tp a Function Symbol table
`ASTNode.h`	Contains a member variable: a pointer of Variable Symbol table.
`VerifyAndBuildSymbols.h`	Derived class from `ASTVisitor`. It contains semantics errors and print them in `main.cpp`.

Table 1: File Description

Tips

- You can learn about `std::map` operations from cppreference
- Be sure you know how to visit each node starting from program in class `ASTVisitor`.

- Class `VerifyAndBuildSymbols` should have a similar structure as class `ASTPrinter` visit methods. Generally, you want to visit the node, check for any errors, and then add the appropriate function or variable into your symbol table. You should also know about how virtual and overridden function calls work in the Base class and Derived class from the Tutorial.

- You can change any files. You can add any functions or member functions for each class. The files of interest are shown in Table 1.

- There are many useful functions in `ASTNode.cpp` when you check the node to report errors. e.g. You can fill in the definition for the `locateDeclaringTableForVar()` function if you think it is useful for checking parent scopes for the appropriate variable symbol table.

- You can ignore some member functions if you choose not use them, such as `setRoot()` in `ASTNode.cpp`.

- You should use the error printing format shown in `a4_errors.txt` and the hints (and their order) in `VerifyAndBuildSymbols.cpp` to check for semantic errors in the program.

- You can build the symbol table and check errors at the same time.

- The Minic language specification says ”The operators of level 3 do not associate, so a==b==c is illegal.” You’re free to check for this error, but it will not be tested in following assignments. The sample solution will show the answer in comments.

- You don’t need to care about the line number or column of an error when printing an error message. Just make sure you print it.

- **Do not think about very extreme or strange cases!** The tests will not closely examine edge cases. This allows for some personal interpretation and differences in the compiler design. For example, extreme cases like this will not be tested:

  ```
  int Hello(int a, int b, int c);
  bool Hello(bool a, int b, bool c, int d){ return 1; }
  ```

  Errors will occur in different positions of the AST. Multiple errors should be reported in the order of visiting AST node.

Public Autotester

We have provided a public autotester `a4tester.zip` on Quercus. After compiling your code, change `asst4.py` to your minicc executable file path. Then simply run:

```
    $./asst4.py
OR in verbose mode
    $./asst4.py -v
```

The A4 tester will examine semantics error messages with different MiniC benchmarks. The error messages are listed in `a4_errors.txt`, which is released as a separate file. We may also release additional files as part of the skeleton code.

Deliverables

Compress the whole project folder as a `zip` file. This zip file must match the structure of the skeleton code, and it must compile in our A1 environment, or you may receive a 0 on the automarker. On Markus, please submit:

- The `zip` file. We will directly unzip your submission, build it from source and run it against public and private tests.
- A brief explanation of your implementation and testing plan as a `txt` file.