CSCE 4650/5650

Assignment 4

Due: 11:59 PM on Friday, April 18, 2025

Requirements:

1. Design and implement symbol table management routines to store all the identifiers used in a program. Specifically, you should have operations to:
    a. Add an identifier to the symbol table,
        i. There are three categories of identifier: variable, function, class.
    b. Look up an identifier in the symbol table to see if it is there or not, returning the symbol table entry information if it is, and
    c. Print the symbol table.

    The symbol table will have 2 levels, one for the top (global) level consisting of all functions, and one for variables defined within functions.
    a. Function identifiers – need to hold following attributes:
        i. The function return type,
        ii. The function's local symbol table containing all the identifiers within the function's scope (which may include the arguments), as described in the information needed for variable identifiers below, and
        iii. The intermediate code for the function represented as string (empty string for this assignment).
    b. Variable identifiers – need to hold the following attribute:
        i. The type of variable.
    c. Class identifiers – need to hold the following attributes:
        i. The class's local symbol table containing all the member identifiers and member functions within the class's scope.

    Note that types will be integer, object (class name) or array.

2. Integrate semantic actions into your parser to build the symbol table.

3. Output the symbol table, either in its entirety or the local symbol table for each class as well as that of the functions within the class. If you choose to output the symbol table incrementally; you may output this after parsing each function.

Suggestions:

A good starting point will be to test and understand the demo of parser generator for PL/0 language using CUP tool and printing symbol table entries. For more, check out the Parser-SDT-SymbolTable under "**Parser-SDT-SymbolTable (Java)**" section on the Canvas for necessary files and necessary steps to compile/run.

Design:

Following is a basic design of the classes necessary for implementing the symbol table management. But you can design your own classes or modify the classes as needed.

SymbolTableEntry	Class for symbol table item.
Category Type String SymbolTable	Category of the symbol: FUNCTION, VARIABLE, CLASS Type of the variable id, function return type, class name, or array Function code. Empty for this assignment. Symbol table to the local environment.
SymbolTableEntry(Category, Type); …	Constructor taking category and type of symbol table entry. (add more constructors as necessary)
SymbolTable	Class for symbol table.
TreeMap<String, SymbolTableEntry> SymbolTable	Hash table for holding entries of the symbol table. Symbol table reference of its parent.

SymbolTable(SymbolTable)
void enterVar(String, Type);
void enterFunc(String, Type, SymbolTable);
void enterClass(String, SymbolTable);
 	
SymTableEntry lookup(String);
 	
void print();
SymbolTable parent();
…
	Constructor taking parent's symbol table reference.
	Adds an identifier to the symbol table.
	Adds enters a function id, its return type and its local symbol table into the symbol table.
	Add class enters a class id and its local symbol table into the symbol table.
	
	Look up an identifier in the symbol table to see if it is there or not, returning the symbol table entry information if it is.
	
	Print the symbol table.
	Returns symbol table reference of current environment's parent.

Type	Base class for representing different *derived* types: integer, array, class name.
Declarator	Class for holding identifier's id and type.
Id Type	Identifier string value. Function name, variable name, or class name. Type of the identifier.

Declarator(Id, Type);
Id id();
Type type();
…
	Constructor taking id and type.
	Returns id.
	Returns type of the id.

Semantic Actions in Parser:

The following is an example for semantic actions needed in CUP file for entering int variable type into symbol table to get you started. You need to add semantic actions in the productions in appropriate places.

```
…
action code {: SymbolTable currentEnv, env; :};

terminal String ID;
terminal String INTEGER;
…
non terminal SymbolTable program;
non terminal String simpleTypeName;
non terminal Declarator declarator;
…
program ::= INCLUDE IOSTREAM USING NAMESPACE STD SEMICOLON
            {: env = new SymbolTable (null); /* start new scope */ :}
            externalDefinitionList {: RESULT = env; :} ;

declaration ::= classSpecifier SEMICOLON
                  | declarator:decl SEMICOLON {: env.enterVar(decl.id(), decl.type()); :} ;

simpleTypeName ::= classId:id  | INT {: RESULT = "int" :} ;

declarator ::= simpleTypeName:type objectId:id
                  {: RESULT = new Declarator (id, type); :}
                   | declarator:decl LBRACK INTEGER:dimension RBRACK ;

objectId ::= ID:id {: RESULT = id; :} ;
```

Sample Output:

See Sample Output on the canvas for more.

```
$ cup -parser TinyCPPParserST -symbols Symbol -expect 1 TinyCPPST.cup

$ jflex TinyCPP.jflex

$ javac -cp .:java-cup-11b-runtime.jar TinyCPPParsST.java

$ java -cp .:java-cup-11b-runtime.jar TinyCPPParsST < tests/test1.cc
```

```
Source Program
--------------

// test1.cc

#include <iostream>

using namespace std;

int main () {
  int my_list [100];
  int my_list_tl [100];
  int r; int h; int i;
  {
    r = 2;
    while (r < 5) {
      my_list [r - 2] = r;
      r = r + 1;
    }
    h = my_list [0];
    i = r;
    while (i > 0) {
      my_list_tl [i - 1] = my_list [i];
      i = i - 1;
    }
  }
  cout << h;
  cout << my_list_tl [0];
  return 0;
}

Identifier Table for Source Program
-----------------------------------

Id         Category Type
--         -------- ----
main       FUNCTION int

Identifier Table for main
-------------------------

Id         Category Type
--         -------- ----
h          VARIABLE int
i          VARIABLE int
my_list    VARIABLE array (100, int)
my_list_tl VARIABLE array (100, int)
r          VARIABLE int
```

Submission:

- You will electronically submit to the **Assignment 4** dropbox in Canvas by the due date.
- Submit all the necessary files that are needed to successfully parse the test files.
    - Cup file (TinyCPPST.cup)
    - Jflex file (TinyCPP.jflex)
    - Main parser program/driver (TinyCPPParsST.java)
    - Symbol table (SymbolTable.java)
    - Other utility files (ErrorMessage.java)
- Program submissions will be checked using a code plagiarism tool against other solutions, including those found on the Internet, so please ensure that all work submitted is your own. Any student determined to have cheated may receive an **'F'** in the course and will be reported for an academic integrity violation.
- Until you are comfortable working on our Linux CSE machines, as a safety precaution, do not edit your program (using vim or nano) after you have submitted your program where you might accidentally re-save the program, causing the timestamp on your file to be later than the due date. If you want to look (or work on it) after submitting it, make a copy of your submission and work off that copy. Should there be any issues with your submission, this timestamp on your code on the CSE machines will be used to validate when the program is completed.