Programming Language Concepts

4003-450-01

Spring Quarter 20113

Project 1: Symbol Tables & Blocks

Due

- As shown on myCourses Project 1 submission dropbox. Team project (after teams are formed).

Objectives

- Become familiar with the functional programming paradigm using a "functional" subset of Scheme.

Overview

Compilers and interpreters maintain and use symbol tables to keep track of identifiers (symbols) and their attributes (such as type information). Symbols are added to this table when an identifier declaration is parsed and later used to check the types of subsequent expressions and to generate efficient code. Additionally, many procedural languages support the block (or compound statement) concept in which new identifiers can be declared. When an identifier name is reused within a block, the declaration within the block typically hides the outer one. Identifiers cannot be reused with the same block to indicate different type information. So the symbol table must keep track of versions of identifiers in each block in which they are declared. The compiler must somehow track block entry and exit so that the symbol table allows identifiers to be reused within a new block and disallows this when the block is left.

In this assignment, you will develop a structure and functions to implement a simplified symbol table for a highly simplified version of a block structured language such as Java or C/C++. The symbol table keeps track of variables and their types.

```
1     class c {               ;; Outer blocks begins with identifier c declared here
2        int a; float b;      ;; Outer identifiers a and b declared here
3           ...               ;; Outer identifiers a, b and c in scope here
4           {                 ;; Inner block begins
5              char a; int d; ;; Inner identifier a and local d declared here
6              ...            ;; Inner identifiers a, d; outer b, c in scope here
7           }                 ;; Inner block ends
8        ...                  ;; Outer identifiers a, b and c in scope here
9     }                       ;; Outer blocks ends
```

Because we haven't dealt with parsing or scoping (yet!), your Scheme program need not parse the above program using the "right" kinds of techniques! For now, simply assume that a fictitious future parser will use your program to set up the symbol table making appropriate calls to the functions described below.

The data representation for the symbol table may be selected in any way you deem appropriate. One way of representing a symbol table is the following at each of the lines indicated on the above program. For example, the symbol table may look like this at line 0:

```
()
```

And like this at line 2:

```
((c class 0))
```

And like this at line 3:

```
(((c class 0) (a int 0) (b float 0)))
```

And like this at line 6:

```
(((a char 1) (d int 1)) ((c class 0) (a int 0) (b float 0)))
```

And like this at line 8:

```
 (((c class 0) (a int 0) (b float 0)))
```

In this representation, a new list is created for each new block entered with outermost block having level 0. Identifiers are record with their types and block nesting level. For example, the identifier `c` is typed as a `class` defined at level `0`, and the identifier `d` typed as an `int` defined at level `1`.

Other representations are also possible; don't feel compelled to use this representation because you might find an easier-to-use representation, i.e., easier in terms of implementing the functions mentioned below.

It is advisable to do this assignment in two phases that are described below. Also, just as you use several methods in designing a Java class, it makes sense to define and use extra helper functions in Scheme (or any functional programming language). Following these two strategies (phased development and helper functions) will simplify your life considerably and also improve the odds of completing this assignment on time!

Phase 1: Without Blocks (80% of grade)

Define Scheme functions to manipulate a simple symbol table. In this phase, you need to omit the support for blocks to simplify getting the basic functionality up and running.

Function	Arguments	Effect
`table-create`	none	Returns an empty symbol table
`set-identifier`	`table, identifier, type`	Checks to see if the identifier is already in the table. If so, an error message is printed out and the table is not modified; otherwise, the identifier and its type and level are added to the table. The function returns the resulting symbol table.
`get-identifier`	`table, identifier`	Returns the type and level of that identifier in the given symbol table. If the identifier is not in the table, the function returns the empty list.
`table-print`	`table`	Formats and prints out the symbol table entries, their types and levels. The table should be "pretty printed," i.e., the output must illustrate the block levels of the different identifier entries.

Phase 2: With Blocks (20% of grade)

Now add functions to support a block-structured symbol table, and modify other functions (as needed).

Function	Arguments	Effect
`enter-block`	`table`	Returns a symbol table that allows subsequent sets and gets will take place in the context of the new block.
`exit-block`	`table`	Returns the symbol table representing the previous outer block.

For example, `set-identifier` and `get-identifier` must now be capable of dealing with multiple block levels from most recently entered to least recently entered. Also, `table-print` must be changed to print nested blocks identifiers and their types appropriately. Each level of nesting requires the addition of a new inner symbol table, and your program should support as many levels as nesting as may be needed.

[Aside: in the coming weeks of the quarter, we will be studying the use of formal grammars to describe programming languages, and examine better techniques for generating and managing symbol tables. For now, just treat this as a simple Scheme assignment, i.e., writing possibly recursive functions on lists.]

Scheme Usage

Recall that the reason for discussing Scheme in this class is to focus on the functional paradigm. While Scheme supports other functionality, you must focus on the purely functional subset of Scheme in this project.

Use only the following primitives in your assignment. If other primitives are needed, check with the instructor first.

- ' (quote)
- List constructors: cons, append, list
- Math functions: +, -, *, /
- List selectors: car, cdr, and related sugared shortcuts such as cadr, cddr, caddr, etc.
- Conditionals and relational operators: if, cond, and, or, not
- Comparisons and predicates: list?, null?, <, >, <=, >= , = , equal?, eq?, member
- Other functions discussed in class: map, reverse, etc.
- define and let

Note: Using iterative constructs such as loops, or assignment statements will result in an F grade!

Style. Use the same structure (as was done in Lab 1) to ensure all team member names and email ids are included. Each function must have a header that describes the function. Use comments to explain your code. Use descriptive variables names, proper indentation, use spaces (not tabs) to indent, and keep each line shorter than 60 characters long.

Submission

Submit your modified Scheme file to the myCourses dropbox as follows.

```
project1.scm
```

Notes

- To give you a rough outline, I am making available a simpler version of the driver file that will be used to test your code. You may download this file, `project1_simpleDriver.scm`, from the course webpage, and extend it as needed to exercise your code. The driver file first loads your `project1.scm` file and then exercises the required functions. Of course, the final driver file that will be used to exercise your code will need to be more exhaustive in its coverage of the project's functionality!
- As you can observe, the instructor's test script requires you to name your functions and file as stated; if you don't, the script will simply generate a failing grade! So take care to avoid severe penalties.

Please submit only the named file before the deadline.
Note: emailed or hardcopy submissions will not be accepted.