SYSTEMS II — PROJECT 2
Revision 1
Compiler code generation

1 The goal of this project

A compiler is of little use if it does not ultimately generate assembly code that implements the provided source code. In generating this assembly code, the compiler must implement a model for mapping variable names to memory spaces, and for performing procedure calls (including stack management). In this project, you will fill in key components to the code generation portion of your compiler.

2 The code

In Project 1, you developed a compiler that used objects to build the internal representation of the source code that it parsed. For this project, you are going to grab new compiler code that provides some of the capability to *generate* assembly code from the parsed program.

2.1 Getting it

The code for this project is provided in a form that will replace some existing classes in your compiler from Project 1. Therefore, you should follow the steps below carefully to make a copy of your compiler, and then to overwrite portions of that code with the newly provided classes. Notice that the newly provided Project 2 code **does not contain** `Parser.java`, ensuring that your work on that class in Project 1 will be unaffected.[1] To construct your Project 2 code, do the following on the department workstations/servers:

```
$ cd cs26 [or perhaps into your shared directory]
$ tar -xzvpf ~sfkaplan/public/cs26/project-2.tar.gz
$ cd project-2
$ cp ../project-1/Parser.java .
```

2.2 Understanding it

Once you completed the previous project, you had a compiler that would, when a given program was successfully parsed, hold a pointer to a `Program` object. This object, in turn, holds a `List` of `Declaration` objects, each of which is either a (global, `Static`) `Variable` or a `Procedure` (definition, not call). Each of the `Procedure`s in turn point to more objects (e.g., `Dynamic` variables, various types of `Statement`s and `Expression`s) that define that procedure.

[1] Of course, you should not have modified any other class as part of Project 1. If you did, good luck with integrating the changes from the provided Project 2 code with your own.

The compiler already calls on the `bind()` method in each such object, passing it the declarations. Objects that contain pointers to other objects (e.g., a `Procedure` that contains a pointer to a `Statement`) turn around and call the `bind()` method in those objects. In those objects that contain a reference to some *symbol*—the name of another procedure or variable—the `bind()` method looks through the declarations provided and finds the one that matches the reference. That is, if an `Expression` uses a variable named `quux`, then its `bind()` expression finds the object that was created when `quux` was declared, and then keeps a pointer to that object for later.

After the symbols are bound, the compiler is ready to generate code. To do so, it follows the same cascading chain of method calls used to perform the binding. The compiler initiates this cascade by calling the `toAssembly()` method in its `Program` object. That method, in turn, calls on each (global, `Static`) `Variable` and each `Procedure` to generate assembly code for itself. Each `Procedure`, in generating its own assembly, calls on its `Statement` (e.g., *begin-end* statements, *if-then-else* statements, *while* loops) to generate its own code.

The problem with declarations: The picture for code generation is not quite as simple as it seems. Each `Declaration` (that is, each `Procedure` and `Variable`[2]) may be used in multiple ways throughout a program, and each of these uses requires different assembly code to be generated. Specifically, for each declared procedure or variable, the program may need assembly for:

- **Definition:** Generate the instructions or constants that are defined by this declaration. For `Procedures`, the instructions for its definition are those for when the procedure it called. For `Static` variables, labeled constants in the assemblers `.Numeric` mode are generated, thus reserving space for the variable and providing it an initial value. `Dynamic` variables have no definitional assembly—their space is not created statically within the assembly code, but dynamically by executing the procedure call mechanisms at run-time. Each `Declaration` has a `toAssemblyForDefinition()` method to generate this code.

- **Evaluation:** When a `Variable` is evaluated (either `Static` or `Dynamic`) as part of an `Expression`, then we need assembly that extracts the value from the space bound to the variable and pushes it atop the stack. `Procedures`, however, are evaluated when a `ProcedureCall` triggers the assembly generated by the definition. Therefore, a `Procedure` is never evaluated in the direct manner that a `Variable` is, and has no evaluative code generation. Each `Declaration` has a `toAssemblyForEvaluation()` method, but in `Procedures`, that method aborts compilation.

- **Reference:** With the *reference operator* (`&`), a program can obtain the address of any declared symbol. For `Static` varaibles and `Procedures`, referencing it will yield the address at which that item is loaded into main memory from the executable image. (The address of a `Procedure` is the address at which its *entry point*—its first instruction—is loaded.) When referencing a `Dynamic` variable, one obtains the address at which the current instance of that variable exists on the stack. Each `Declaration` has a `toAssemblyForAddress()` method that generates the assembly code that pushes its address atop the stack.

[2]These are *subclasses* of `Declaration`. If you don't know about *inheritence*, ask me. It's not terribly important to understand it deeply here, but a superficial description will help you follow the compiler code.

The simplicity of statements: All `Statements` (e.g., *begin-end*, *if-then* and *if-then-else*, *while* loops) and all `Expressions` (which are subclasses of `Statements` that produce a result that is pushed onto the stack) are simpler than `Declarations` when it comes to code generation. A `Statement` is only used in one manner from this perspective: it is called upon to inject the code that performs the tasks it describes. Therefore, each `Statement` contains a single code generation method named `toAssembly()` that produces a sequence of instructions that implement the statement. For example, a `While` statement generates code to perform a loop.

3 Your assignment

You will find that some of the `toAssembly()` methods (and their ilk) have had their method bodies removed and replaced with a comment that reads, "
texttt// FIX ME". You can find these by using the `grep` command, like so:

```
$ grep "FIX ME" *.java
```

For each class that needs to have its assembly generation method filled in, do so. Then write yourself small, testing programs, compile them, assemble the result, and run the final executable image on the simulator to see if your code works. By the end, you should have a compiler that you can use to write some kernel code.

4 How to submit your work

Use the `cs26-submit` command to turn in your programs. From your `project-2` directory, do this to submit **all** of the classes, including ones written by both you and me:

```
cs26-submit project-2 *.java
```

This assignment is due at **11:59 pm** on **Thursday, March 4.**

A Revision history

- **Revision 0 [2010-Feb-18]:** The initial, complete version.
- **Revision 1 [2010-Feb-26]:** Updated due date.