Assignment 4

Submission Date: 24th March 2015

Here is some background repeated from the previous stage:

1. As you process the program, the information contained in the declarations is stored in the *symbol table*. We have seen that this information includes the the types of variables and functions and the sizes and offsets of variables.
2. When you process the non-declarative part, then two things happen:
   (a) The information in the symbol table is used to ensure that the variables are used within the scope of their declarations and are used in a type-correct manner. If they are not, then the program is rejected.
   (b) If the program is syntactically and semantically (type and scope) correct, a tree structure called an AST is generated. This structure along with the information in the symbol table is used to generate code.

In this (fourth) assignment, we shall construct the symbol table, do the semantic checks and then generate ASTs using the AST construction scheme that was developed in the third assignment.

I suggest that you create the following two level symbol table structure.

1. A global symbol table (gst) that maps function names to their local symbol tables.
2. A local symbol table for every function that contains the *relevant* information for the parameters and the local variables of the function. You have to decide and defend which information is relevant. Here is something that you want to keep in mind: The AST and the information in the symbol table should be enough to generate code in the fifth stage.

Having constructed the symbol table, you will now process the non-declarative part of the program once again to create an AST, re-using the code that you had developed during the third assignment. However, you will now perform semantic checks to ensure that semantically incorrect programs are rejected. While doing this, you try to mimic as closely as possible the behaviour of gcc—any program that GCC rejects, you shall also reject.

Your output for good programs will consist of:

1. A dump of the symbol table of each function in a format of your choice, and
2. A dump of the AST of a set of functions that the user should be able to select.

3. For bad programs, it should mention the error that caused the program to be rejected. The requirement is a message like "Type mismatch in line 34", or "Undeclared variable in line 78".

Here are the decisions that we have taken so far:

1. For $x + y$, do the following:
   (a) If x and y are both ints, resolve the + to $+_{int}$. The AST (when printed) will be `(PLUS_INT x y)`.
   (b) If x and y are both $float$s, resolve the + to $+_{float}$. The AST (when printed) will be `(PLUS_FLOAT x y)`.
   (c) If x is int and y is $float$, cast x to a float and resolve the + to $+_{float}$. The AST (when printed) will be `(PLUS_FLOAT (TO_FLOAT x) y)`.
2. For $x\&\&y$, No casting is required. The result is 1 if both operands are non-zero, and 0 otherwise. The type of the expression is integer.
3. For $x < y$, if x is int and y is $float$, cast x to a float and resolve the $<$ to $<_{float}$. The AST (when printed) will be `(LT_FLOAT (TO_FLOAT x) y)`. The result will be integer.
4. The implementation does not require you to pass arrays as parameters.
5. The structure of the activation record is:

```
space for return value       |  stack grows
space for parameters         |  towards lower
space for old ebp            |  addresses
space for locals             v
space for saved registers
```

   Don't bother about saved registers right now.
6. How to layout local variables and parameters is your choice. You need not do it in the same way as GCC.
7. At the first error, print a message reporting the cause of the error and location clearly and exit. Here is an (incomplete) list of errors that you have to look out for:
   (a) All form of type errors. Examples: Array index not being integer. Variables being declared as being the `void` type.
   (b) All form of scoping errors.
   (c) Non-context-free restrictions on the language. For example, array indexed with less indices than its dimension and functions being passed with lesser than required number of parameters.