Homework #7

Question 1 (10 pt.)

Consider the following grammar for function declarations:

$$
\begin{aligned}
F &\rightarrow T \ \text{id} \ (\ A \ ) \ ; \\
A &\rightarrow \epsilon \\
&\ \ | \ B \ T \ \text{id} \\
B &\rightarrow \epsilon \\
&\ \ | \ B \ T \ \text{id} \ , \\
T &\rightarrow \text{int} \\
&\ \ | \ \text{float} \\
&\ \ | \ \text{void}
\end{aligned}
$$

Write a parser based on Flex and Bison that produces LLVM code for function declarations. Remember that a function declaration is a line of code specifying the prototype of a function, including the function name, return type, and argument list. A function declaration does not include a function body.

Function declarations in LLVM do not include argument names, which means that you can ignore the string attributes associated with the argument identifiers. However, the name of the LLVM function should be the same as the C function name. This is an example of the execution of the parser:

Input:

```
float f(int x, float y, int z);
```

Output:

```
; ModuleID = 'TestModule'
declare float @f(i32, float, i32)
```

Upload the scanner in a file named `scanner.l`, and the parser in a file named `parser.y`. Your program should take a file name as a command-line argument, and should compile correctly on the COE achines by running the following sequence of commands:

```
$ bison -oparser.c parser.y -d -v
$ flex -oscanner.c scanner.l
$ g++ parser.c \
        scanner.c \
        -o parser \
        `llvm-config --cppflags` \
        `llvm-config --ldflags` \
        -lLLVM-3.4 \
        -std=c++11
```

Here are some hints that will guide you through the implementation:

- You can use the following Bison union to track the semantic values for the grammar symbols:

```
%union
{
   // For 'A' and 'B'
   std::vector<llvm::Type *> *types;

   // For 'Type'
   llvm::Type *type;

   // For 'id'
   char *name;
}
```

  Nonterminals *A* and *B* are associated with a vector of types, which gathers the argument types as production rules are reduced. This vector is included in the union as a pointer to a vector, as opposed to a vector itself. The reason is that a union does not allow for fields with an implicit constructor invocation, since the compiler has no way to know which constructor of which field to invoke when the union is instantiated. Thus, you must make sure to first instantiate a new vector dynamically using the C++ `new` operator, before you start inserting elements into it.

- The following LLVM functions will be helpful. You can find additional information for them, and other related functions, in the LLVM Doxygen online documentation.

```
static FunctionType *FunctionType::get(
        Type *Result,
        ArrayRef<Type *> Params,
        bool isVarArg);

Constant *Module::getOrInsertFunction(
        StringRef Name,
        FunctionType *Ty);
```

  Notice that LLVM type `ArrayRef` is compatible with `std::vector`, and thus you can directly pass the collected vector of types into the second argument of function `FunctionType::get()`.