CSC435/535: Assignment 4
(Due: 11:55pm, Monday 4 April 2016)

Introduction

This assignment asks you to add some missing features to a Goo compiler. In effect, this means adding code to the visitor which performs code generation for a Goo program.

Supplied System

Important: The supplied code does not implement the requirements of Assignment 3. This means that very little type checking is performed on a Goo program. Invalid Goo code will produce unpredictable behavioir.

The supplied code includes a visitor for the parse tree which outputs LLVM's assembly language. The visitor implementation supports the following language features:

- Declarations of variables and constants with simple types (numbers and strings).
- Declarations of arrays.
- Functions and function calls.
- The `fmt.Print`, `fmt.Println` and `fmt.Printf` library functions (though not all formats for `Printf` will work – the format string is passed unmodified to the `printf` function in the C library).
- Assignment statements (including `+= -= *= /= %=`)
- If statements.
- Return statements.
- Expressions using arithmetic operators.
- Expressions using comparison operators.
- Array indexing.
- Type coercion.

Assignment Description

You should add support for the following features to your Goo compiler. They are listed in approximate order of increasing difficulty (except when one feature depends on a previously listed function being available).

1. *While* loops [10 points]
2. The '`+`' and '`-`' unary operators [10 points]
3. The conditional *and* and *or* operators ('`&&`' and '`||`') [10 points]
4. The '`!`' unary operator. [10 points]
5. The *for* statement. [15 points]
6. The address-of ('`&`') and dereferencing ('`*`') operators. [10 points]
7. Variables with struct types and access to the fields inside structs. [15 points]
8. Passing struct values as parameters to functions. [5 points]

9. The Go predefined function *new* to allocate storage for an array or struct (this function must be translated into a call to the C library function `calloc`). [15 points]

Evaluation Criteria

Test Goo programs will be used to evaluate whether each language feature has been implemented correctly. The Goo programs will all be correct code according to the Go compiler. (I.e., your type checking for assignment 3 is not being tested, though correct compilation of some language features being implemented in Assignment 4 do depend on knowing the correct datatypes of constructs in the Goo programs).

If all 9 tasks listed above are implemented and work correctly on the test programs, you can potentially score 100%. However there will be penalties for not meeting the submission requirements (or for submitting the work late).

Help and Other Comments

- If your Assignment 3 program was implemented reasonably successfully, **do not replace** all your Java files with the files provide with this assignment. Instead, you should add the new files (see the table of materials listed below) and you should transfer bug fixes and other small new code additions from the supplied files to your files (again, see the table of materials).
- You can easily discover the correct LLVM code for a Goo language feature by
  - writing a small C program which contains the C equivalent of that feature,
  - using the clang C compiler to compile the C code into LLVM code, and
  - inspecting the LLVM file.

  The clang command to compile `testprog.c` is
  ```
  clang -S -emit-llvm testprog.c
  ```
- You can test whether your LLVM code works by using the `lli` command (`lli` comes with the clang distribution on MacOS and Linux; a Windows version is available on conneX in the Resources/Materials folder).
  If the LLVM file is named `testprog.ll` then the command to use is:
  ```
  lli -force-interpreter testprog.ll
  ```
  (Without the '`-force-interpreter`' flag, `lli` crashes on Windows.)
- You are advised to rebuild the Goo compiler and rerun it on a test file after each coding change, even for relatively small changes. (But you never need to re-run Antlr unless you change the grammar files.)

Submission Requirements

1. You must combine all the source code files into a single compressed archive file and upload that to conneX. Do not include any class files generated by the Java compiler. (There will be more than 100 of them!). Any Java files generated by Antlr can optionally be included in the archive file, but they will be ignored. Please use a filename that begins '`ass4`' and has the appropriate suffix for the file format. The choices are: `ass4.zip` (Zip file), `ass4.tgz` (gzipped tar archive) or `ass4.rar`.
2. The comment at the start of the `GooMain.java` file must be edited so that it lists the names and student numbers of the team members who collaborated on the assignment.

3. As in Assignment 3: to make the marker's task easier, we want ONE team member to upload the submitted materials as explained in requirement #1. In addition, we want ALL OTHER team members to upload a tiny text file with the name `team.txt` that lists the names of all the team members, one per line (in any order).
4. The submitted code must not produce any error messages or warning messages when processed by Antlr4 or when the Java files are compiled. Errors that prevent the main program from invoking the code generation visitor will lead to an automatic zero.
5. If your compiler is invoked with just the Goo program filename appearing afterwards on the command-line, your program must generate no output in the console window (assuming the program is correct Go code). The only result of running the compiler should be the creation of the LLVM file (whose name is the same as the input file but with the suffix changed to '`.ll`').
6. The project should be completed in teams of either 2 or, at most, 3 persons. Single member teams are permitted but not encouraged. Your team does not have to contain the same members as for Assignment 2.
7. Late assignments are accepted with a penalty: 10% (of maximum score) for up to 24 hours late, 25% for up to 48 hours late, 50% for up to 72 hours late. You can resubmit as often as you want. The time of the latest submission determines which late penalty, if any, is to be applied.

The Provided Materials

These are the same as for Assignment 3 but with some additional materials and some files which contain small changes. The additional materials and changed materials are listed in Table 1. Any supplied file not listed below should be the same as provided for Assignment 3.

Note: the `lli` command crashes on Windows after executing the LLVM code created from `test3.go`. It also crashes in exactly the same way after executing the LLVM code generated by clang for the C version of the same program. (The reason is a mystery!)

Further Work: Slices (no credit!)

After completing this assignment, many language features will remain absent or incomplete. One of the biggest omissions is the *slice* type. If you were to add this feature, you would probably want to implement a slice value in the LLVM code as a quadruple: <array pointer, end of array pointer, start of slice pointer, end of slice pointer>.

For example, if the Goo program takes a slice `Arr[5,10]` from the array named `Arr` which comprises 20 `int` elements, then the run-time representation of the slice would be the same as this C struct value:

```
struct slice {
    int *arrayBase;  int *arrayEnd;
    int *sliceStart; int *sliceEnd;
} = { Arr, &Arr[20], &Arr[5], &Arr[10] };
```

There should be nothing intrinsically difficult in making this all work, just a lot of tedious code to write!

Table 1: Additional/Changed Materials Provided

File Name	Description
`SymTabVisitor2.java`	The same as in Assignment 3 but with some additions or corrections (marked with the comment // CHANGED IN ASS4)
`Type.java`	Almost the same as in Assignment 3: the `isComplete()` method has been recoded in some of the subclasses. (Look for the // CHANGED IN ASS4 comment)
`FunctionSymbol.java`	Almost the same as in Assignment 3: a `getParameters()` method has been added.
`GooMain.java`	Significantly changed from Assignment 3: it has been extended to invoke the new code generator visitor and supply it with the name of the LLVM output file.
`CGenVisitor.java`	The new visitor which creates LLVM code.
`CGenFunctionCall.java`	A helper class for `CGenVisitor` which creates LLVM code for calls to Goo functions and library functions.
`LLVM.java`	The main class for managing and creating LLVM code.
`LLVMExtras.java`	Contains additional (static) methods for managing and creating LLVM code.
`LLVMValue.java`	An instance of the `LLVMValue` class records the name and type of a temporary variable in the LLVM code.
`LLVMPredefined.java`	Contains boilerplate LLVM code which needs to be output at the beginning and end of a LLVM file.
`test1.go`	Tests assignment, if statement and `fmt.Println`
`test2.go`	Implements the factorial function using recursion
`test3.go`	Initializes and prints an array, using recursion instead of a loop to iterate through the elements.
`test3inC.c`	A C translation of `test3.go`