CSC435/535: Assignment 4
(Due: 11:55pm, Friday 1 August 2014)

Overview

The final stage of the course project is to complete some missing features in the Cb code generation stage.

Preamble

1. Code which implements Assignment 3 is provided. You can choose to use this code instead of your own implementation of Assignment 3, or you can mix and match as you wish.
   Do note however that a few very small additions were made to files previously made available and these are noted below.
2. The code supplied for this assignments implements code generation for most of the Cb language. A fairly exhaustive list of missing features can be created by scanning through the source code of the `LLVMVisitor2.cs` file and looking for one-line comments which say *TODO*.
   The major omissions are arrays, while loops and the Boolean operators && and `||`.
3. The supplied code uses LLVM tools to generate assembly language code for three different platforms: 32-bit Windows with Cygwin, 64-bit Linux, and 64-bit Mac OS X. If you have installed LLVM on a different platform, it should be reasonably easy to support that too.

Compiling and Running Cb Programs

Once you have compiled and built an executable of the supplied cbc program, you can translate and run as follows.

1. First, pick one of the three platforms named above. The default is the 64-bit linux system running on the CSC teaching server, accessible via a secure shell connection at linux.csc.uvic.ca. For the other two platforms, you must specify one of these two options on the command line:
   ```
   -target="i686-pc-mingw32"
   -target="x86_64-apple-macosx10.9.3”
   ```
   If we have a Cb source file named `sample.cs`, then the command
   ```
   cbc sample.cs
   ```
   will create a text file named `sample.ll`. (This file contains code in the LLVM intermediate language.)
2. If necessary, transfer the `sample.ll` file to the machine where it is to be executed.
3. Translate the LLVM file to assembly language by use of the llvm tool ‘`llc`’. For example,
   ```
   llc sample.ll
   ```
   will create an assembler file named `sample.s`.
4. Assemble and link your program with the C library using the Gnu compiler tools. For example,
   ```
   gcc sample.s -o prog.exe
   ```
   will generate an executable file named `prog.exe`.
5. Run the executable file in the usual way.

The Supplied Materials

The new source code files are as follows.

`LLVMVisitor1.cs`	Traverses the AST outputting type definitions needed for class instances and outputting the static class constants.
`LLVMVisitor2.cs`	Traverses the AST outputting code for all the methods. *This code is incomplete* – not all AST nodes are handled.
`LLVM.cs`	The support code for generating LLVM output is organized as a single class named `LLVM` which is split across 7 separate files (using the C# partial class feature). This first file contains the constructor and a few other important methods. It is also defines a small class `LLVMValue` which is ubiquitous throughout the code. That class records what is held by a LLVM temporary and what datatype it holds.
`LLVM-Definitions.cs`	This file contains only boiler-plate code which is copied almost verbatim to the LLVM output. This generated code implements the builtin methods such as `Console.WriteLine` and `Int32.Parse`.
`LLVM-ConstantHandling.cs`	Provides methods for defining and accessing Cb constants in the LLVM code.
`LLVM-CreateClassDefn.cs`	Used by `LLVMVisitor1`; it provides a method for scanning over a class definition and generating the type definitions and the VTable needed when class instances are created by code generated in pass 2.
`LLVM-UtilityMethods.cs`	The methods in this file are mostly concerned with manipulating `LLVMValue` objects.
`LLVM-WriteMethods.cs`	Various methods in this file generate LLVM code for Cb language constructs.
`LLVM-SSAMethods.cs`	These methods support SSA code generation.

In addition to the new materials listed above, the following files required some small additions.

`CbType.cs`	The `CbField` class has a new field named `Index` which records its position in the structure used for instances of the class to which the field belongs. The `CbMethod` class has a new field named `VTableIndex` which specifies the method's position in the class's VTable.

CbSymTab.cs	A new method named Clone creates a duplicate of the symbol table. A new method named GetEnumerator returns an enumerator which can be used to sequence through all the entries in the symbol table. And the SymTabEntry class has a new field named SSAName which records the most recent name generated by the SSA algorithm for that variable in the symbol table.

LLVM

The website http://llvm.org/ provides downloads of the LLVM tools (some in compiled form, and all in source code form) plus lots of documentation and tutorials. The most important document is the *LLVM Language Reference Manual*.

If you wish to understand LLVM code, an approach easier than reading the manual is to download clang (if it is available in pre-built form for your platform). This is a C compiler which uses the LLVM tools to generate object code. You can run small experiments to see how each C language feature is translated into LLVM code. The command

```
clang -S -emit-llvm testfile.c
```

generates a file containing LLVM code. That file is named testfile.ll on Linux/Windows but testfile.s on Mac OS X (which is confusing because '.s' is meant to denote assembler code).

If you cannot find a prebuilt clang for your platform, building it from source code is possible but the source code distribution is gigantic, and building both it and the LLVM tools requires several gigabytes of space on your hard drive (as well as keeping your CPU busy for hours while many C++ compilations are being performed). You should also be familiar with makefiles as well as the usual Gnu approach to handling source code distributions.

The only LLVM tool you need for this course project is llc.

The Assignment Requirements

The difficulty level of this assignment is high. Therefore you are being asked to add only these four features to the incomplete implementation provided to you. The first two should be easy and will help you gain some familiarity with the code before proceeding to the remaining two.

Supporting the Unary Minus Operator [20% weight]

The unary minus is left unimplemented. After you've completed this assignment, the code

```
int k;  k = 97;  k = -k; System.Console.Write(k);
```

should produce the obvious result. Treating the -k expression as being the same as 0-k would be the obvious way to tackle this problem.

Supporting the Increment and Decrement Operators [25% weight]

The ++ and -- operators are left unimplemented. Note that treating the statement

```
x++;
```

as though it were the statement x = x + 1; is an obvious way to implement the operation.

Supporting Conditional And, and Or [45% weight]

The two operators && and || are not yet converted into LLVM code. You should combine the operands of these two operators using control flow as outlined on the course slides.

Converting the operands to 0 or 1 boolean values and using the bitwise *and* and *or* operators to combine them is disallowed as a solution. (It has the wrong semantics.)

Even though there is control flow involved in the LLVM code, and this is control flow which causes new basic blocks to be constructed, there are no implications for SSA code generation. Just test values and generate conditional branch instructions. There are no phi functions needed in the code (because comparison expressions in Cb can have no effect on local variables).

Implementating While Loops [10% weight]

This is the hard problem because SSA code is being generated. You should first study how code is generated for an *if* statement, how the symbol table (which now tracks SSA names) is replicated for the *then* and *else clauses*, and how the symbol tables are re-combined and phi functions are generated at the control flow merge point.

The Moessenboeck and Brandis approach should be implementable with some extra features which have been added for that purpose to the provided classes. The M & B approach requires capturing the generated SSA code and renumbering some SSA variables before outputting the code again. The `DivertOutput`, `UndivertOutput`, and `InsertCode` methods in the LLVM class should provide that capability. (Renumbering is textual replacement.)

The `GeneratedNames` property, which is assigned a value by the `Join` method of the LLVM class, can be used to tell you which needs name to be modified.

The rest is up to you to figure out!

Submission Requirements – All As Before

1. You must provide all source files needed to build your program. Do *not* submit any files generated by `gplex` or `gppg`.
2. We will run `gplex` on your `.l/.lex` file, `gppg` on your `.y` file, and then compile all the C# files with the command `csc *.cs`. If these steps do not yield an executable program, you lose points.
3. You must combine all your files including a README file into a single compressed archive file. The only accepted formats for the archive file are as a zipfile (and the filename must have a "`.zip`" suffix) or as a gzipped Unix tar file (and the filename must have a "`.tgz`" suffix).
4. Upload your archive file via conneX.
5. The project is to be completed in teams of either 2 or 3 persons. The ideal size is 2 people. All team members *must* participate. Be sure to identify who the team members are in a separate file named `README.txt`. (If there are any special features of your code to point out, or things you didn't get working right, you can mention them here.)