Project 2: An LLVM Code Generator for MiniC

ECE 466/566 Fall 2015

ECE 466 students may work in teams of 2,
ECE 566 students must work individually.

~~**Due: October 1, 11:59pm**~~
Due: October 3, 11:59pm

You are encouraged to comment directly on this document rather than posting questions on Piazza about the specs, as comments make it easier to understand context.

Objectives

- Implement an LLVM bitcode generator for a subset of the C programming language.
- Interpret a language specification and a grammar.
- Explore the semantics of the C language.
- Apply theoretical concepts of the control-flow graph to code generation.
- Build ***control flow*** with the LLVM API.

Description

This project is based on a subset of the C programming language. For convenience, we'll call it MiniC. Many features of C have been eliminated to make this project easier. Also, the supported subset differs between 466 and 566 so be sure to understand the differences well.

Features for 466:

1. Only 32-bit Integer type.
2. Global and local variables, including parameters, are allowed.
3. No post-fix and pre-fix increment/decrement operators (++/--).
4. No operators supporting pointers, structs, unions, or arrays need be supported.
5. No short-circuited **&&** and **||** operations. You may assume both operands evaluate at the same time.
6. Statements include only **if-then-else**, **while**, and **expression** statements. No **break** or **continue.** Also, **if** must always have a matching **else**.
7. Loops are **not** nested, but **if-then-else** may appear inside of a **while**, and if-then-else statements may be nested.
8. There will be only one function per source file, and it need not be named main.

Features for 566:

1. 32-bit Integer type and pointer to 32-bit integers.
2. Global and local variables, including parameters, are allowed.
3. No post-fix and pre-fix increment/decrement operators (++/--).
4. No operators on structs, unions, or arrays need be supported.

5. Dereference (*) and address-of (&) must be supported.
6. No short-circuited **&&** and **||** operations. You may assume both operands evaluate at the same time.
7. Statements include **if-then-else**, **while**, **for**, **expression** statements, **break** and **continue**. Also, **if** must always have a matching **else**.
8. Statements may be **arbitrarily** nested.
9. There will be only one function per source file, and it need not be named main.

You will be provided a lexer and parser that already implement the necessary features of the MiniC language. You will need to implement the actions during parsing to build the C program in LLVM IR. Some actions have already been implemented for you. You may re-implement those actions if you choose, but you are encouraged to understand them and leverage them in your implementation to cut down on your workload. Also, I encourage you to re-use code from P1 if it applies.

Your compiler should take in a MiniC program and create a legal LLVM bitcode file. The generated function should be added to a module and dumped as a legal LLVM bitcode module. This module will be linked with a C program that calls the function and tests that it is logically correct. Support for building and dumping Modules is already integrated in the base P2 code, but I encourage you to read it and understand it.

Project Infrastructure and Support

1. You should implement your project by extending the code in either the p2 or p2cpp tool directory:
   a. `ECE566Projects/projects/tools/p2`
   b. `ECE566Projects/projects/tools/p2cpp`

2. You may need to update your repository to get the latest version of code. Then rebuild everything:
   a. `cd path/to/ECE566Projects`
   b. `git pull`
      i. If this command fails, it's because you have modified files. You can either commit them or stash them (but not both). A commit will keep your changes in the local directory, but *stash will remove your changes and save them* elsewhere. Pick the best one for your case:
         1. `git commit -a -m"some changes I made blah blah blah"`
            Or...
         2. `git stash`

      Now, go back and re-execute `git pull`.
   c. `cd path/to/ECE566Projects/projects/build`

   d. Note: If your build directory is already configured for the language you want, then

you can skip this step and go straight to building (step e). Otherwise, follow the instructions for the language you prefer. For C, remove the build directory and re-run the cmake configure command inside the build directory to choose C (and I think the default language is C++ for p2):

i. `export LLVM_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake`

ii. `cmake **-DUSE_C=1** -DLLVM_CMAKE_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake ..`

For C++, remove the previous build directory and re-run the cmake configure command inside the build directory to choose C++ (I think for C++ you can skip this part):

iii. `export LLVM_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake`

iv. `cmake **-DUSE_CPP=1** -DLLVM_CMAKE_DIR=/usr/local/llvm/3.6.2/install-debug/share/llvm/cmake ..`

e. `cmake --build .`

3. Your project will be tested using the wolfbench repository configured using the tool you implement. Configure your testing directory as follows:
   a. Before following the next steps, make sure that the llvm-config and clang commands are found in your path:
      i. `which llvm-config`
      ii. `which clang`

      If not, add the directory where clang is installed into your path variable. On VCL, that command will look like this:

      `export PATH=$PATH:/usr/local/llvm/3.6.2/install/bin`

   b. First, enter your wolfbench directory and do a git update to get the files for this project:
      i. `cd path/to/ECE566Projects/`
      ii. `mkdir p2-test;`
      iii. `cd p2-test`

   c. Assuming that wolfbench, p2-test, and projects are all in the same parent directory, you can configure the project as shown below. For ece466 test cases only, add `--enable-ece466` to the configure command:
      i. For C (select the 566 or 466 option):
         1. `../wolfbench/configure --enable-p2=`cd ../projects/build/tools/p2/; pwd`/p2`
         2. `../wolfbench/configure --enable-p2=`cd

```
../projects/build/tools/p2/; pwd`/p2
--enable-ece466
```

ii. For C++ (select the 566 or 466 option):

1. `../wolfbench/configure --enable-p2=`cd ../projects/build/tools/p2cpp/; pwd`/p2`
2. `../wolfbench/configure --enable-p2=`cd ../projects/build/tools/p2cpp/; pwd`/p2 --enable-ece466`

iii. Then, build the test code:

1. `make all test`

iv. If you want the output to be less verbose, run it this way:

1. `make -s all test`

v. If you encounter a bug, you can run your tool in a debugger this way:

1. `make clean`
2. `make DEBUG=1`

This will launch gdb on your tool with one of the input files. You can set breakpoints directly in the parser.y file within rules.

4. At the end of the `make test` run, you will see a percentage of how many tests passed. The initial version of the code will only pass about 5% of tests. Once you pass all the tests, you're pretty much done. At that point, you only have to worry about the secret cases! But, you may need to clean up your code and document it some more before your final submission. Also, note that the provided test cases may not cover all aspects of the spec. It is up to you to perform that testing.

5. You can find the code you need to get started inside projects/tools in the p2 or p2cpp directory. Please keep the following in mind as you use the infrastructure:
   a. You are allowed to modify any of the source files provided, and you are allowed to add your own C files or header files to the project. You may also import open source data structures to help you. *However, you may not import data structures that constitute a full solution, in other words no cheating!*
   b. You should not **substantially** alter how the testing infrastructure works in order to make your code work, as we will use a copy of wolfbench that's unmodified.

LLVM API

The required LLVM API is very similar to that of P1. You will need to make use of the following:

LLVM-C/Core/Types/IntegerTypes
LLVM-C/Core/Types/SequentialTypes (for pointers)
LLVM-C/Core/Values/BasicBlock
LLVM-C/Core/Values/General APIs
LLVM-C/Core/Values/User Value

LLVM-C/Core/Values/Constants/Scalar constants
LLVM-C/Core/Values/Constants/Global Values/Global Variables
LLVM-C/Core/Values/Constants/Global Values/Function Values
LLVM-C/Core/Values/Constants/Global Values/Function Parameters
LLVM-C/Core/Instruction Builder

Advanced Bison Features

So far, we have done just fine with simple Bison features. However, for P2, you will need to use an advanced capability known as a **mid-rule action**. Rather than performing an action only after an entire rule matches, it's often beneficial to perform an action after a partial match. For example, consider this rule:

```
statement:   while LPAREN expression RPAREN statement { /* action */ };
```

The `expression` needs to be placed in a different basic block than `statement`. Recall from P1 that the Builder remembers where to insert code, and if you don't tell it otherwise, it will keep putting new code into the same basic block. So if we wait to perform the action once after matching the whole rule, then expression and statement would be lumped inside one basic block. To have fine grain control, we should use a mid-rule action. We can do this instead:

```
statement:   while LPAREN
{ /* mid rule: create a new basic block for the loop header */}
             expression
{ /* mid rule: insert branch and create new basic block for body */}
             RPAREN  statement
{ /* insert final branch back to header */ };
```

Mid-rule actions alter the numbering of terminals and non-terminals in the rule. **while** is still $1 and **LPAREN** is $2. But, **expression** is now $4, and **statement** is $7. Each mid-rule action is given it's own number, which means $3 refers to the result of the first mid-rule action itself. Of course, you need to assign it a result using $<field name in union>$. You must specify the field name between the dollar signs because there is no other way to specify which field you want to use for the mid-rule action in advance.

Mid-rule actions with examples are covered in detail in Tutorial 3.

Working with Variables

LLVM requires that all registers obey SSA form. The key rule behind SSA form is that all registers are defined only once. Said another way, you can never write twice to the same register.

Fortunately, you can read and write to memory an unlimited number of times. Hence, the simplest way to obey SSA form is by allocating all variables to memory. At each use of the variable, load the variable into a register. At each update of the variable, store the new value back to its memory location. This will guarantee all aspects of SSA form are obeyed.

Here are the implications:

1. At a variable read (whether it be a local, parameter, or a global), load the value from memory into a register. Use a load instruction in LLVM IR.
2. At a variable write in an assignment statement, store the new value into the memory using a store instruction in LLVM IR.
3. For all variables, you need to track their address. The symbol.h header in the C/C++ projects provides an interface for doing just that.

In fact, I've already implemented this code for you, but you should read through the code I provided to make sure you generally understand what it's doing.

Extra Files

To assist with some of the project's requirements, I've added a few supporting files: list.c, list.h, symbol.c and symbol.h. These files provide you with an interface for tracking declarations, their scopes, and even nested loop information. There are already some examples of their use in the provided files. Feel free to modify, extend, or discard these files as you see fit.

Getting Help

History shows that my specs are sometimes incomplete or incorrect. Therefore, please start early. If you run into problems, please add comments/questions to this document or post questions to Piazza.

WARNING: if you wait until the day before it's due to start this assignment, it is unlikely you will succeed. It does require a lot of planning and familiarization with the LLVM API. These are not tasks you can easily rush. Also, the instructor may not respond to last minute cries for help.

Grading

ECE 566 students must work individually, but ECE 466 students are allowed and *encouraged* to form groups of two. Only one student needs to submit in ECE 466, but both names must appear in the submission document and in the comments of all your code.

Uploading instructions: Create an archive of the p2 or p2cpp directory.

cp –R ./projects/tools/p2[cpp] p2[cpp]

Then do this:

cd p2[cpp]
rm –Rf `find . –name Debug+Asserts`
rm [any other test files you created]

Then, zip up the folder using either zip or tar+gzip to create a single compressed archive.

tar czf p2.tgz ./p2[cpp]

or

zip –r p2[cpp] ./p2[cpp]

Upload the archive in Moodle on Project 2 assignment page. Also, please add a note to your submission in the Notes field indicating which language you used. We will test your code using the test cases provided in wolfbench and with some secret cases we did not provide.

The assignment is out of 100 points total. If you make no attempt and submit the provided code without meaningful changes (i.e. white space and comments do not count), you earn 10 points. Otherwise, assigned as follows:

ECE 566
10 points: Compiles properly with no warnings or errors
10 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications
30 points: Fraction of tests that pass
20 points: Fraction of secret tests that pass (these may overlap with provided tests)

ECE 466
10 points: Compiles properly with no warnings or errors
10 points: Code is well commented and written in a professional coding style
30 points: Meets or exceeds all specifications
40 points: Fraction of provided tests that pass
10 points: Fraction of secret tests that pass (these may overlap with provided tests)