Project 1: A Simple Code Generator Targeting LLVM IR

ECE 466/566 Fall 2015

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

Due: September 14, 11:59 pm

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 simple programming language on your own.
- Gain experience reading and interpreting a language specification.
- Gain experience implementing a simple programming language using parser generators: Flex & Bison.
- Practice code generation with LLVM and gain experience using the LLVM software infrastructure.
- Learn to use the course's project infrastructure.

Description

This project is based on the programming language, P1, described by the following tokens and grammar:

Keyword/Regular Expression	Token Name
`vars`	VARS
`$[a-z0-9][a-z0-9]?`	TMP
`[a-zA-Z_]+`	IDENT
`[0-9]+`	NUM
`=`	ASSIGN
`;`	SEMI

-	MINUS
+	PLUS
*	MULTIPLY
/	DIVIDE
,	COMMA
^^	RAISE
<	LESSTHAN

^^ means exponentiation. `3^^2 = 3*3. 4^^5 = 4*4*4*4*4`

The grammar for the P1 language is given by the following rules:

```
program:  decl stmtlist;

decl:      VARS varlist SEMI;

varlist:  varlist COMMA IDENT
| IDENT;

stmtlist: stmtlist stmt
| stmt;

stmt: TMP ASSIGN expr SEMI;

expr:        expr MINUS expr
           | expr PLUS expr
           | MINUS expr
           | expr MULTIPLY expr
           | expr DIVIDE expr
           | expr RAISE expr  /* raise $1 to power of $3, $3 must be
an expression that folds to a constant, and generates error message
if $3 is negative */
           | expr LESSTHAN expr /* produces 0 if false, 1 if true */
           | NUM
           | IDENT
           | TMP ;
```

The arithmetic operators have their usual meaning. Here is an example program for this

language stored in file myfunc.p1:

```
vars a,b,c;

$0 = a*b;
$1 = $0+c;
$3 = $1;
```

The variables specified by the vars keyword represent integer inputs to the list of statements. These are the same as the input arguments of the C-like function you should generate.

All of the TMP tokens represent temporary values computed by an expression. A temporary must be defined before it is used (but they need not be declared in advance in the vars list). The final result of the program is the value assigned to the last temporary. For the program above, the final “return value” is the value assigned to $3.

We can think of this particular program as producing code equivalent to a single C function (with some simplification):

```
int64_t myfunc(int64_t a, int64_t b, int64_t c)
{
      return a*b + c;
}
```

However, we want to generate LLVM bitcode, so it really looks more like this:

```
define i64 @myfunc(i64 %a, i64 %b, i64 %c) {
entry:
  %0 = mul i64 %a, %b
  %1 = add i64 %0, %c
  ret i64 %1
}
```

Note, the C API does not allow you to assign names to parameters. So, you do not need to generate IR that looks exactly like the IR above.

In summary, you will implement the rules for a parser that generates LLVM bitcode for any program written in the P1 language.

Detailed Specification

1. Generate a function in LLVM bitcode that takes as many arguments as there are

variables in the varlist. The order matters. The order in which variables are declared is the same order in which they are passed to the function from left to right.

2. The MULTIPLY/DIVIDE/RAISE operators have highest precedence with left associativity. PLUS/MINUS are next with left associativity. And, LESSTHAN has the lowest precedence with left associativity.
3. The name for the function is taken from the name of the (output) file. So, myfunc.p1 will produce a function named myfunc.
4. The generated function will always return an integer type. Also, the return value must match the value assigned to the last temporary.
5. You may only generate one basic block. You may not use a loop or call a function within the generated code for the exponentiation operator.
6. The generated function should be added to a module and dumped as a legal LLVM bitcode module. This module will be then linked with a C program that calls the function and tests that it is logically correct. However, don't fret about these details because most of the code for dumping LLVM modules will be provided for you.
7. You may implement your project in either C or C++ using the p1 or p1cpp project directory provided in the **projects/tools** folder. Look inside the p1/p1cpp folder and you will find some starter code. But, *it is only a starting point!* You may need to change the scanner and parser.
8. If there is any compile error (e.g. undefined variables), generate the according error message and abort. Don't try to generate error message for runtime errors. Just let core dumped.
9. 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. For C, re-run the cmake configure command inside the build directory to choose C (and I think the default language is C++ for p1):
      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++, 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 .`

10. 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 p1-test;`
       iii. `cd p1-test`

    c. Assuming that wolfbench, p1-test, and projects are all in the same parent directory, you can configure the project like this:
       i. For C:
          1. `../wolfbench/configure --enable-p1=`cd ../projects/build/tools/p1/; pwd`/p1`
       ii. For C++:
          1. `../wolfbench/configure --enable-p1=`cd ../projects/build/tools/p1cpp/; pwd`/p1`
       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.

11. 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 pass 1% or less. 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.

12. You can find the code you need to get started inside projects/tools in the p1 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 Classes and Modules

For this project, you will need to leverage several APIs/Classes within the LLVM compiler. Which ones you use depends on the language you choose to work with. If you are working in C, you should investigate the C API for LLVM located here: http://llvm.org/doxygen/modules.html. You will find a discussion of the Instruction Builder API, which is the one you will use the most. Also, read the comments in the provided file to identify other functions of interest.

For C++, the LLVM Class hierarchy is a good place to start (http://llvm.org/doxygen/annotated.html) , though it is quite large and daunting. You really only need to focus on a few key classes: llvm::IRBuilder, llvm::Module, llvm::Function, llvm::Type, and llvm::StringMap.

However, for both of the languages, I recommend you read the full code I gave you first so that you understand what has already been implemented and what you need to implement. I've already provided helpful hints in comments within the code.

Getting Help

History shows that my specs are sometimes incomplete or incorrect. Please start early so that you can get the help that you need. When you run into problems, please post a question to Piazza as this makes it easier for other students to find help.

Grading

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

Uploading instructions: First, clean your projects folder. Make sure your name (and your teammate's name) is in comments at the top of all your source files. Then, zip up the p1 or p1cpp folder using either zip or tar+gzip to create a single compressed archive.

```
cd path/to/ECE566Projects/projects/tools/
tar czf p1.tgz ./p1
```

or

```
zip –r p1  ./p1
```

Upload the archive in Moodle on Project 1 assignment page. We will test your code using the test cases provided in wolfbench and with some secret cases we did not provide.

You are not allowed to alter how the configure script or Makefiles work as that will break our testing infrastructure. If we cannot test your code by following the same steps as shown above, you will get a **0.**

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)