Project 1: A Simple Code Generator Targeting LLVM IR

ECE 466/566 Spring 2020

ECE 466 students may work in teams of 2,

ECE 566 students must work individually.

**Due: February 17, 2019, 11:59 pm**

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

# Objectives

* Implement an LLVM bitcode generator for a simple programming language.
* 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

## Tokens and Grammar

In this project, you will implement a Lisp inspired programming language that we’ll call P1 that’s described by the following tokens and grammar:

|  |  |  |
| --- | --- | --- |
| **Keyword/Regular Expression** | **Token Name** | **Description** |
| [a-zA-Z\_][a-zA-Z\_0-9]\* | ID | These hold temporary results |
| [0-9]+ | NUM | A decimal number |
| - | MINUS | Unary minus operation. |
| + | PLUS | Addition operation. |
| \* | MULTIPLY | Multiply operation. |
| / | DIVIDE | Divide operation. |
| ( | LPAREN | Open parentheses. |
| ) | RPAREN | Close parentheses. |
| setq | SETQ | Keyword used to create a variable. |
| min | MIN | Keyword used to calculate minimum of values. |
| max | MAX | Keyword used to calculate maximum of values. |
| aref | AREF | Keyword used to access an array. |
| setf | SETF | Keyword used to modify an array. |
| make-array | MAKEARRAY | Keyword used to create an array. |

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

program : exprlist ;

exprlist: exprlist expr | expr;

expr: LPAREN MINUS token\_or\_expr\_list RPAREN

|LPAREN PLUS token\_or\_expr\_list RPAREN

|LPAREN MULTIPLY token\_or\_expr\_list RPAREN

|LPAREN DIVIDE token\_or\_expr\_list RPAREN

|LPAREN SETQ ID token\_or\_expr RPAREN

|LPAREN MIN token\_or\_expr\_list RPAREN

|LPAREN MAX token\_or\_expr\_list RPAREN

|LPAREN AREF ID token\_or\_expr RPAREN

// Next two rules for 566 only

|LPAREN SETF token\_or\_expr token\_or\_expr RPAREN

|LPAREN MAKEARRAY ID NUM token\_or\_expr RPAREN

;

token\_or\_expr\_list: token\_or\_expr\_list token\_or\_expr | token\_or\_expr;

token\_or\_expr: token | expr;

token: ID | NUM;

## Examples and Meaning

Programs in the P1 language correspond to a single function in the C language. Here’s a really simple example program in P1:

(+ 1 2 3)

This says to add 1+2+3 and return the result. Note that after the + we can have an arbitrary number of operands, 1 or more) that are getting added together. Or, we can define variables and use them:

(setq a 100)

(setq b 200)

(+ a b)

This program would produce the value 300. Even more concisely, we could write this program as follows:

(+ (setq a 100) (setq b 200))

This shows that many of the expressions can be nested.

Going deeper, P1 assumes that all programs take an integer and integer array as an argument and returns an integer. It would look something like this if written in C code:

int p1\_program(int arg\_size, int \*arg\_array)

{

// do something

return 0;

}

Hence, we can access this array using the array reference keyword, aref:

(aref arg\_array 0)

This would return the value at the first position of the arg\_array. To get the value at the last position of the array, we can do this:

(aref arg\_array (+ arg\_size (- 1)))

arg\_array always refers to the input arg\_array and arg\_size tells us the size of the array.

## Explanation of Each Rule

Some aspects of this grammar need more explanation:

|  |  |  |
| --- | --- | --- |
| **Rule** | **Example** | **Description** |
| LPAREN MINUS token\_or\_expr\_list RPAREN | (- 1) | This expr performs a unary minus. The token\_or\_expr\_list should only be a single item. The expr produces a value equal to the negation of its operand. |
| LPAREN PLUS token\_or\_expr\_list RPAREN | (+ 1 2 3) | This expression adds its operands and produces the sum. MULTIPLY expression works the same way, but it produces the product instead. |
| LPAREN DIVIDE token\_or\_expr\_list RPAREN | (/ 5 2 2) | Like PLUS and MULTIPLY, DIVIDE also applies to all of the operands. But, here the order matters. The first operand is divided by the second, then the result is divided by the third, and so on. In C code, it could be written like this:  5 / 2 / 2  We only perform signed integer division. |
| LPAREN SETQ ID token\_or\_expr RPAREN | (setq x 5)  (+ x 3) | Create a variable called x and initialize it to 5. Future references to x produce the value of the most recent setq. For example, the second expression produces the value 8. |
| LPAREN MIN token\_or\_expr\_list RPAREN  | LPAREN MAX token\_or\_expr\_list RPAREN | (min 2 3 1)  (setq x 5)  (max 2 3 10 (+ x 6)) | min and max expressions return the smallest values in their list of operands. The first expression produces 1, while the third produces 11. |
| LPAREN AREF ID token\_or\_expr RPAREN | (aref arg\_array 1) | aref allows us to access an element of an array. We assume that all programs are given as input an array called arg\_array. This returns the value held at index 1. |
| The following rules are for ECE 566 only. | | |
| LPAREN SETF token\_or\_expr token\_or\_expr RPAREN | (setf (aref arg\_array 1) 10) | setf allows us to set the location within an array. The first operand must be the result of an aref expression. The second operand can be the result of an expression or a token. The expression produces a result equal to the value stored into the array. |
| LPAREN MAKEARRAY ID NUM token\_or\_expr RPAREN | (make-array a 100 0) | This says to make an array of length 100 that’s initialized to 0. The first operand must be an ID. The second operand must be a length, and the third operand must be the initialization value. If nested in another expression, it produces its initialization value as its result. |
|  |  |  |

You will implement the rules for a parser that generates LLVM bitcode for any program written in the P1 language.

# Additional Specifications

1. Generate a function in LLVM bitcode that takes an integer (i32) and an integer pointer (i32\*) as argument and returns an i32.
2. The name for the function is taken from the name of the input file. If the file is called tmp.p1, the name of the function must be tmp. *(I’ve provided code for this already, but you may not change it.)*
3. The generated function will always return an integer type. The return value is the value produced by the last expr in the program.
4. You may only generate one basic block. Your generated LLVM IR must not have any additional basic blocks and may not use control flow (ifs or loops or calls).
5. Your generated code may not use a loop or call a function to perform an operation.
6. The generated function should be added to a module and dumped as a legal LLVM bitcode module. This module will then be 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. If you detect illegal code during parsing (e.g. undefined variables), print an error message and abort. For runtime errors, like divide by zero, do not try to generate code to prevent them. Just allow them to raise exceptions when the generated code runs.

# Implementation

## Update your Git Repository

You may need to update your repository to get the latest version of code. Then rebuild everything:

* 1. cd path/to/ncstate\_ece566\_spring2020
  2. git pull
     1. 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...*

* + - 1. git stash

Now, go back and re-execute git pull.

Make sure you start docker if you are relying on it to build and test your code:

docker-compose run projects

## Code Development and Testing Setup

There are two ways to use the course infrastructure to implement your project. You may implement your project in either C or C++ using the p1/C++ or p1/C project directory provided in the **projects/tools** folder. Look inside the directories to find some starter code.

Or, you may use the simple/p1/C or simple/p1/C++ folder. No configuration is needed, you only need to make sure that llvm-config is available from your command line. Use this command to check:

which llvm-config

*Most students find this approach simpler to use and understand*.

The next two subsections explain how to get going depending on the method you choose.

### Use the Simple Directory (preferred)

If you are using the simple directory, then simply verify that llvm-config is your path:

which llvm-config

If the command is not found, make sure you add the path to your PATH variable. For example, on VCL it would look something like this:

export PATH=$PATH:/where/llvm-config/is/found

After llvm-config is confirmed, then follow the remaining steps.

1. Build the code in the simple/p1/C or simple/p1/C++ directory.

cd ncstate\_ece566\_spring2020/simple/p1/C++

make

1. Test the code:

cd tests

make test

1. Debug a test case (note, you may need to use the debug version of the docker image):

cd tests

make DEBUG=1 test\_00.p1.bc

### This will run lldb on your program, the debugger that’s bundled with llvm/clang. [For more info on how to use lldb](https://lldb.llvm.org/lldb-gdb.html).

### 

### Use the Projects Directory

If you are using the projects folder, then make a build folder, where you would build your source files.

1. Make a directory.

mkdir -p projects/build

cd projects/build

1. For C, use the cmake configure command inside the build directory to choose C:

cmake **-DUSE\_C=1** ..

1. For C++, run cmake configure command inside the build directory to choose C++:

cmake -DUSE\_CPP=1 ..

1. Run the cmake command to build the code.

cmake --build .

1. Your project will be tested using the wolfbench repository configured using the tool you implement. Configure your testing directory as follows:
   1. Make directory for testing. I recommend under the main repo directory:
      1. cd path/to/ncstate\_ece566\_spring2020/
      2. mkdir p1-test
      3. cd p1-test
   2. Assuming that you are in the docker container, you can configure the project like this:
      1. For C:
         1. ../wolfbench/configure --enable-p1=/ncstate\_ece566\_spring2020/simple/p1/C/p1
      2. For C++:
         1. ../wolfbench/configure --enable-p1=/ncstate\_ece566\_spring2020/simple/p1/C++/p1
      3. Then, build the test code:
         1. make all test
      4. If you want the output to be less verbose, run it this way:
         1. make -s all test
      5. If you encounter a bug, you can run your tool in a debugger this way:
         1. cd test
         2. make clean
         3. make DEBUG=1

This will launch lldb on your tool with one of the input files. You can set breakpoints directly in the p1.y file within rules.

This will run lldb on your program, the debugger that’s bundled with llvm/clang. [For more info on how to use lldb](https://lldb.llvm.org/lldb-gdb.html).

## Development, Testing, and Debugging

1. At the end of the make test run, you will see a percentage of how many tests passed. At first, the project will not complete the full tests because the scanner is not fully implemented and you’ll encounter syntax errors. After you fix these, the initial version of the code will pass a few percent 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.
2. Please keep the following in mind as you use the infrastructure:
   1. 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!*
   2. 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 the Google Group 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 needs to submit in ECE 466, but both names must appear in the comments at the top of all modified files.

Warning: the TAs may amend these steps. Check back closer to the due date.

**Uploading instructions:**

Upload your source files (p1.y, p1.lex, main.cpp/c) to the 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 additional warnings or errors than the code provided

10 points: Code is well commented and written in a professional coding style

60 points: Fraction of tests that pass (15 test cases)

20 points: Fraction of secret tests that pass (these may overlap with provided tests)

**ECE 466**

10 points: Compiles properly with no additional warnings or errors than the code provided

10 points: Code is well commented and written in a professional coding style

60 points: Fraction of provided tests that pass (10 test cases)

20 points: Fraction of secret tests that pass (these may overlap with provided tests)