Project 2: An LLVM Code Generator for C--

ECE 466/566 Spring 2020

ECE 466 students may work in teams of 2,

ECE 566 students must work individually.

**~~Due: March 20, 2020, 11:59 pm~~**

**Due: March 31, 2020, 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**](#_cpezb019f24z) **2**

[**Description**](#_ky1eg34slfyx) **2**

[**Additional Specifications (466 vs. 566)**](#_f1feeevvwz2h) **2**

[Features for 466](#_ueqkfyn8knie) 2

[Features for 566](#_onjm0gfr92qo) 3

[**Implementation**](#_d1sqxcv8vbfy) **6**

[Update your Git Repository](#_tw4ax8bhwt93) 6

[Code Development and Testing Setup](#_vnir4gqh5nud) 6

[Use the Projects Directory](#_lqnzgdxjyimp) 7

[Use the Simple Directory](#_2zw1uibs874c) 8

[Development, Testing, and Debugging](#_27zc70yttyu4) 9

[**Advanced Bison Features**](#_f61cpb1f3k5a) **9**

[**Working with Variables**](#_agavjaufq63u) **10**

[**Extra Files**](#_kg4n7jahdjtp) **11**

[**Getting Help**](#_9tddpjewk62l) **11**

[**Grading**](#_jr44ooo0c8ws) **11**

[**Appendix: Full Scanner and Grammar Spec for C--**](#_2jma3ceik3fn) **12**

[Scanner](#_4tl9tk9g4t8i) 12

[Keywords](#_9oyepurf79lr) 12

[Other tokens](#_2a8idiq48v9y) 12

[Grammar](#_x3h500wgp0bk) 13

# 

# Objectives

* Implement an LLVM bitcode generator for a subset of the C programming language.
* Interpret a language specification and 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 called C--. The grammar is shown below in the Appendix. Most programs in C-- can be interpreted as having the same meaning they would in C/C++.

However, there are some notable differences between the two languages:

* C-- only allows three declared types: int, int\*, and void.
  + void may only be used with functions, which can have a void return type.
  + There are no explicitly declared arrays or structs. Also, there are no typedefs.
  + However, pointers may be used as the base address of arrays.
  + In terms of LLVM IR, int is represented as an i64 and int\* is an i64\*.
* There are no function prototypes, so functions must always be defined before they are called.
* C-- has very minimal support for type checking and type inference. In other words, it does not perform type promotions and casts automatically. All such casts must be explicitly stated in the program. For example, suppose you want a pointer initialized to 0. 0 is really an int, so we have to convert it:

int \* ptr = inttoptr(0);

inttoptr is a special operator that converts from an int to an int\* type. Likewise, to go back to an int use ptrtoint:

int back\_to\_int;

back\_to\_int = ptrtoint(ptr);

These operators have the same meaning as their LLVM IR counter parts.

* As part of the simplified type logic, some operators produce single bit boolean values (i1 in LLVM IR), referred to as bool from here on for convenience. Specifically, these operators are:

==, !=, <=, >=, >, <

Note, they correspond to the predicates of icmp. If they are subsequently used by branches or bitwise-operators, then they do not need any conversion back to int by the programmer. And, when used by bitwise-operators, both operands must be of type bool. It’s illegal for the two operands to have different types. But, if bools are used by expressions other than bitwise-operators, then the bool must be explicitly converted back to int in the source program using either sext or the zext operators. These have the same meaning as their LLVM IR counterparts. Here are some examples, assuming all variables are declared as int:

|  |  |  |
| --- | --- | --- |
| **Expression** | **Legal/Illegal** | **Explanation** |
| (x > 0) & (x < 100) | Legal | Both (x>0) and (x<100) produce i1, then they are bitwise-anded together. |
| x & (x==1) | Illegal | x is an int but x==1 is a bool. This is a type mismatch and raises a semantics error. The grammar allows it, but it’s not a meaningful program. |
| x + sext(x > 0) | Legal | x > 0 makes a bool, but sext converts it back to int. |
| zext(bool(x) ^ (x>0)) | Legal | the bool() operator converts x from int to bool. Then we can bitwise-xor it with result of (x>0) because they are both booleans. Then, the whole thing is converted back to int. |
| x == y+1 | Legal | Produces a bool result. |

# Additional Specifications (466 vs. 566)

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 64-bit integer types are allowed. You may assume no pointers.
2. Each C-- file has global variable and function definitions. There may be one or more global variables or functions.
   1. Functions may have parameters and local variables.
   2. Each variable is declared on a line by itself and may optionally be initialized.
   3. You do not need to support call expressions (calling other functions).
3. No post-fix and pre-fix increment/decrement operators (++/--).
4. No operators supporting pointers, structs, unions, or arrays need be supported.
5. No logical operations (&&, ||). But, there are bitwise operations (&, |, ^).
6. Statements include only **if-then-else**, **while**, **for**, **do**, and **expression** statements. No **break** or **continue.** 
   1. **if** must always have a matching **else**.
   2. If-then-else statements may be nested arbitrarily.
   3. Loops are **not** nested in other loops, but any other statement may appear in the body of a loop.
7. Relational operators (!=, ==, <=, <, >, >=) produce a single bit boolean value. Follow the guidance earlier in the doc for how to handle expressions that include bools.
8. You may implement additional features for bonus points:
   1. Implement some or all 566 features and earn extra points per 566 test case that passes.
   2. To earn bonus credit, your bonus code must be operational at the same time as your normal submission. It cannot be disabled or commented out. If it prevents the 466 test cases from working, you will lose points for those test cases. So, do not attempt the bonus until you have all the 466 cases working.

## Features for 566

1. Types that must be supported: int (i64), int\* (i64\*), and void (for function returns).
2. Each C-- source file may include global variables and multiple function definitions.
   1. Functions may have parameters and local variables of any allowed type. The may return void, int, or int\* types.
   2. All variables are declared on a line by itself and may optionally be initialized.
   3. Functions may call other functions, including themselves; recursion is allowed and must be supported.
3. No post-fix and pre-fix increment/decrement operators (++/--).
4. No operators on structs or unions need be supported.
5. Relational operators (!=, ==, <=, <, >, >=) produce a single bit boolean value. Follow the guidance earlier in the doc for how to handle expressions that include bools.
6. Statements include **if-then-else**, **while**, **do-while**, **for**, **expression** statements, **break** and **continue**.
   1. **if** will always have a matching **else**.
   2. Statements may be **arbitrarily** nested.
7. Arrays are supported only through pointer type variables (dereferencing/array access). There are no declared arrays.
8. Pointer and operations on pointers, like Dereference (\*) and address-of (&) must be supported. Implicit conversion between integer and pointer types is not allowed. The programmer must explicitly perform type conversions in the source code. Arithmetic operations on pointers is forbidden. Relational operations on pointers is allowed provided that both operands are pointer type (e.g. you can compare against null pointer).
9. Bonus points:
   1. You may add support for switch statements for 10 bonus points.
   2. To earn bonus credit, your bonus code must be operational at the same time as your normal submission. It cannot be disabled or commented out. If it prevents the 566 test cases from working, you will lose points for those test cases. So, do not attempt the bonus until you have all the 566 cases working!

Your compiler should take in a C-- 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 one or more of the functions 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. Many other required features are already implemented for you. I encourage you to read through the source code before you begin your implementation so that you better understand the parts you need to implement.

# Infrastructure

## Save pending work

You will need to update your repository to get the latest version of code. Then rebuild everything. Either commit or stash any pending work:

* + - 1. git commit -a -m”some changes I made blah blah blah”

*Or...*

* + - 1. git stash

### Case 1: Basic cloned repository

Pull the latest copy and then merge conflicts:

git pull

### Case 2: Remote tracking branch (Advanced option from Project 0)

If you have the advanced setup from Project 0 with a remote tracking branch, do this:

git checkout work # replace work with your branch of choice

Replay your commits on top of the class repo:

git pull --rebase ece566 master

### Run Docker

After updating git, 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

I’m providing you starter code and a testing framework. There are two ways to use the course infrastructure to implement your project: using the simple directory or the projects directory. In both cases, C and C++ are supported.

### 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/p2/C or simple/p2/C++ directory.

cd ncstate\_ece566\_spring2020/simple/p2/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 have used the projects directory before, skip to Step 4. Otherwise, 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-p2=/ncstate\_ece566\_spring2020/projects/build/tools/p2/C/p2
      2. For C++:
         1. ../wolfbench/configure --enable-p2=/ncstate\_ece566\_spring2020/projects/build/tools/p2/C++/p2
      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, Interactive Testing, and Debugging

1. At the end of the make -k test run, you will see a percentage of how many tests passed. At first, the project will not complete the full tests because the IR generated will be illegal. After you fix these, the initial version of the code may pass a few tests, but very few.

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.

1. 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.
2. More info about debugging to come soon. In the meantime, printf debugging can be effective. However, printf isn’t the best function to call. If you want to see the contents of a Value object, do this:2

Value \*value = ...; // initialize value

value->print(errs(),true);

1. You may also test your code interactively from the command line. After you build the p2 binary, run it like this:

./p2 -v -use-stdin

Then start typing code and end the program with //END, and you’ll see something like this:

int f() { } //END

; ModuleID = 'p2'

source\_filename = "p2"

define i64 @f() {

entry:

}

# 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 your directory. First, clean out any unnecessary files.

make clean

rm [any other test files you created that aren’t needed - don’t delete your code!]

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

cd path/to/simple

tar czf p2.tgz ./p2/C++

or

cd path/to/simple

zip –r p2.zip ./p2/C++

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

60 points: Fraction of tests that pass

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

+10 bonus points if you fully implement the switch statement correctly

**ECE 466**

10 points: Compiles properly with no warnings or errors

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

60 points: Fraction of provided tests that pass

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

+1 bonus points for additional 566 test cases that you pass, not to exceed 10 points total.

Calculated at discretion of the instructor.

+10 bonus points if you fully implement the switch correctly

# Appendix: Full Scanner and Grammar Spec for C--

## Scanner

### Keywords

|  |
| --- |
| int void for while if else switch case default do inttoptr ptrtoint zext sext bool return break continue |

### Other tokens

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Token** | **Name** | **Token** | **Name** | **Token** | **Name** |
| ; | SEMICOLON | - | MINUS | = | ASSIGN |
| : | COLON | \* | STAR | & | AMPERSAND |
| , | COMMA | / | DIV | | | BITWISE\_OR |
| { | LBRACE | % | MOD | ^ | BITWISE\_XOR |
| } | RBRACE | <= | LTE | << | LSHIFT |
| ( | LPAREN | >= | GTE | >> | RSHIFT |
| ) | RPAREN | < | LT | ~ | BITWISE\_INVERT |
| [ | LBRACKET | > | GT | //EOF | MYEOF |
| ] | RBRACKET | == | EQ |  |  |
| + | PLUS | != | NEQ |  |  |

## Grammar

translation\_unit: external\_declaration

| translation\_unit external\_declaration

| translation\_unit MYEOF

;

external\_declaration: function\_definition

| global\_declaration

;

function\_definition: type\_specifier ID LPAREN param\_list\_opt RPAREN

compound\_stmt

| type\_specifier STAR ID LPAREN param\_list\_opt RPAREN

compound\_stmt

;

global\_declaration: type\_specifier STAR ID opt\_initializer SEMICOLON

| type\_specifier ID opt\_initializer SEMICOLON

;

declaration\_list: declaration

| declaration\_list declaration

;

type\_specifier: INT | VOID

;

opt\_initializer: ASSIGN constant\_expression

| // nothing

;

param\_list\_opt:

| param\_list

;

param\_list: param\_list COMMA type\_specifier ID

| param\_list COMMA type\_specifier STAR ID

| type\_specifier ID

| type\_specifier STAR ID

;

statement: expr\_stmt

| compound\_stmt

| selection\_stmt

| iteration\_stmt

| return\_stmt

| break\_stmt

| continue\_stmt

| case\_stmt

;

expr\_stmt: SEMICOLON

| assign\_expression SEMICOLON

;

compound\_stmt: LBRACE declaration\_list\_opt statement\_list\_opt RBRACE

;

declaration\_list\_opt:

| declaration\_list

;

statement\_list\_opt:

| statement\_list

;

statement\_list: statement

| statement\_list statement

;

break\_stmt: BREAK SEMICOLON

;

case\_stmt: CASE constant\_expression COLON

;

continue\_stmt: CONTINUE SEMICOLON

;

selection\_stmt:

IF LPAREN expression RPAREN statement ELSE statement

| SWITCH LPAREN expression RPAREN statement

;

iteration\_stmt:

WHILE LPAREN bool\_expression RPAREN statement

| FOR LPAREN expr\_opt SEMICOLON bool\_expression SEMICOLON expr\_opt RPAREN statement

| DO statement WHILE LPAREN bool\_expression RPAREN SEMICOLON

;

expr\_opt:

| assign\_expression

;

return\_stmt: RETURN SEMICOLON

| RETURN expression SEMICOLON

;

bool\_expression: expression ;

assign\_expression: lvalue\_location ASSIGN expression

| expression ;

expression: unary\_expression

| expression BITWISE\_OR expression

| expression BITWISE\_XOR expression

| expression AMPERSAND expression

| expression EQ expression

| expression NEQ expression

| expression LT expression

| expression GT expression

| expression LTE expression

| expression GTE expression

| expression LSHIFT expression

| expression RSHIFT expression

| expression PLUS expression

| expression MINUS expression

| expression STAR expression

| expression DIV expression

| expression MOD expression

| BOOL LPAREN expression RPAREN

| F2I LPAREN expression RPAREN

| I2F LPAREN expression RPAREN

| I2P LPAREN expression RPAREN

| P2I LPAREN expression RPAREN

| ZEXT LPAREN expression RPAREN

| SEXT LPAREN expression RPAREN

| ID LPAREN argument\_list\_opt RPAREN

| LPAREN expression RPAREN

;

argument\_list\_opt: | argument\_list

;

argument\_list: expression

| argument\_list COMMA expression

;

unary\_expression: primary\_expression

| AMPERSAND primary\_expression

| STAR primary\_expression

| MINUS unary\_expression

| PLUS unary\_expression

| BITWISE\_INVERT unary\_expression

;

primary\_expression: lvalue\_location

| CONSTANT\_INTEGER

;

lvalue\_location: ID

| lvalue\_location LBRACKET expression RBRACKET

| STAR LPAREN expression RPAREN

constant\_expression: unary\_constant\_expression

| constant\_expression BITWISE\_OR constant\_expression

| constant\_expression BITWISE\_XOR constant\_expression

| constant\_expression AMPERSAND constant\_expression

| constant\_expression LSHIFT constant\_expression

| constant\_expression RSHIFT constant\_expression

| constant\_expression PLUS constant\_expression

| constant\_expression MINUS constant\_expression

| constant\_expression STAR constant\_expression

| constant\_expression DIV constant\_expression

| constant\_expression MOD constant\_expression

| LPAREN constant\_expression RPAREN

| I2P LPAREN constant\_expression RPAREN

| P2I LPAREN constant\_expression RPAREN

;

unary\_constant\_expression: constant

| MINUS unary\_constant\_expression

| PLUS unary\_constant\_expression

| BITWISE\_INVERT unary\_constant\_expression

constant: CONSTANT\_INTEGER

;