Project 1: A Simple Language for Bitwise Processing

ECE 466/566 Spring 2024

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

Due: February 16, 2024, 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.

Description

In this project, you will implement an expression language, called P1 for convenience. The P1 language is designed to make bitwise processing easier than it is in C. For example, we can make a variable equal to a bit of another variable just by specifying the bit number at the end of the variable name:

```
in none
x = 5  // x2=1, x1=0, x0=1
y = x2 // set y to the third bit (index of 2) of variable x
final y // y = 1
```

Or, we can re-arrange the bits of a number and insert an extra bit:

```
in none
x = 15              //x3=1,x2=1,x1=1,x0=1
y = x3,x2,0,x1,x0   // now y = 27
final y
```

There are also a number of other features. Let's look at the tokens and grammar in more detail.

Tokens and Grammar

P1 is described by the following tokens and grammar:

Keyword/Regular Expression	Token Name	Description
`in`	`IN`	Keyword used to specify parameters.
`none`	`NONE`	Keyword that indicates no parameters.
`reduce`	`REDUCE`	Keyword for reduce operations.
`expand`	`EXPAND`	Keyword for bitwise-expand operations.
`final`	`final`	Indicates return value.
`[a-zA-Z]+`	`ID`	These hold temporary results.
`[0-9]+`	`NUMBER`	A decimal number.
`=`	`ASSIGN`	Assignment.
`~`	`INV`	Bitwise-invert operation.
`!`	`BINV`	Single-bit invert operation.
`&`	`AND`	Bitwise-and operation.
`|`	`OR`	Bitwise-or operation.
`^`	`XOR`	Bitwise-xor operation.
`-`	`MINUS`	Unary minus operation.
`+`	`PLUS`	Addition operation.
`*`	`MULTIPLY`	Multiply operation.
`/`	`DIVIDE`	Divide operation.
`%`	`MOD`	Modulo operation.
`(`	`LPAREN`	Open parentheses.

)	RPAREN	Close parentheses.
[	LBRACKET	Open bracket.
]	RBRACKET	Close bracket.
,	COMMA	Comma.
:	COLON	Used by ensembles to construct numbers.
\n	ENDLINE	Newline marker.
"//".*\n		Comment in source code.

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

```
program: inputs statements_opt final;

inputs:   IN params_list ENDLINE
| IN NONE ENDLINE;

params_list: ID
| params_list COMMA ID;

final: FINAL ensemble endline_opt;

endline_opt: %empty | ENDLINE;

statements_opt: %empty
            | statements;

statements:   statement
            | statements statement ;

statement: ID ASSIGN ensemble ENDLINE
  /* Next two rules for 566 only */
| ID NUMBER ASSIGN ensemble ENDLINE
| ID LBRACKET ensemble RBRACKET ASSIGN ensemble ENDLINE;

ensemble:  expr
| expr COLON NUMBER // 566 only
| ensemble COMMA expr
| ensemble COMMA expr COLON NUMBER; //566 only
```

```
expr:   ID
| ID NUMBER
| NUMBER
| expr PLUS expr   // addition
| expr MINUS expr  // subtraction
| expr XOR expr    // bitwise-XOR
| expr AND expr    // bitwise-AND
| expr OR expr     // bitwise-OR
| INV expr         // bitwise-invert
| BINV expr        // least-significant bit invert
| expr MUL expr    // multiply
| expr DIV expr    // divide
| expr MOD expr    // modulo
| ID LBRACKET ensemble RBRACKET
| LPAREN ensemble RPAREN
/* 566 only after this */
| LPAREN ensemble RPAREN LBRACKET ensemble RBRACKET
| REDUCE AND LPAREN ensemble RPAREN
| REDUCE OR LPAREN ensemble RPAREN
| REDUCE XOR LPAREN ensemble RPAREN
| REDUCE PLUS LPAREN ensemble RPAREN
| EXPAND  LPAREN ensemble RPAREN;
```

Examples and Meaning

A complete program includes a statement about arguments and the final result, with optional statements in between. The simplest program is one that has no arguments and returns 0:

```
in none  // no arguments, no statements
final 0  // final result is 0
```

This program in P1 is equivalent to the following function in LLVM IR:

```
define i32 @example_1() {
entry:
  ret i32 0
}
```

Here's another example:

```
in none     // no arguments
```

```
a = 1       // three statements
b = 1
c = 1
final a,b,c // put the three bits together into a single number
```

Might produce this:

```
define i32 @example_2() {
entry:
  ret i32 7
}
```

Explanation of Each Rule

Some aspects of this grammar need more explanation and are shown below with an example and description. Other rules are similar to prior tutorials and not described in depth here.

Rule	Snippet	Description
`program`	`in none` `in a, b, c`	Declare the input parameters of the program. If none, it's like a void function in LLVM IR.
`params_list`	`a` `a, b`	Single argument in a list. Multiple arguments in a list.
`final`	`final 1` `final 1,1,1,1`	return 1 return 15
`statements_opt`		A program may have zero or more statements.
`statement: ID ASSIGN ensemble ENDLINE`	`x = y+z`	Assign a variable an ensemble. An ensemble is usually just an expression, but ensembles allows us to put expressions together in new ways. See next example.
`ensemble:`	`1,1,0,1`	An ensemble is a way of piecing bitwise values together to make new values. This ensemble evaluates to 13. It can be seen as a binary vector converted to int. The least-significant bit is always on the far right.

		The result of any ensemble is always a 32-bit integer.
	`x,y,z`	Each comma means shift over one bit. So, this expression means `(z<<0) | (y<<1) | (x<<2)`.
	`x:4,y:2,z:1`	We can modify the shifting amount using the colon operator: `(z<<1) | (y<<4) | (x<<9)` Note, the colons plus commas are cumulative.
`expr`	`x = 5`	**Every expression is interpreted as producing a 32 bit integer. But, in some cases, only the least-significant bit matters.**
`expr: ID`	`x = y`	`y` is an `ID` used in an expression. If y has not yet been assigned, it's treated as 0 and a warning is printed to stderr, eg: fprintf(stderr,"%s is used without first being defined. Assuming 0.",%1). The variable name must appear in the warning.
`expr: NUMBER`	`x = 1`	expr can be NUMBER by itself.

`expr: ID NUMBER`	`y = 5`
	`z = y1`
	`w = y2`
		y1 will parse as ID NUMBER. This means get bit at index=1 from y **and move it to be the least significant bit of the expression. So, z will be come 0, but w will become 1.**
		
		Bit indices in integers always start at 0, so in this case y1 is equal to 0.
		
		Because y is a 32-bit integer, the only legal indices are 0 to

		31. But, if an out of range value is given, it's behavior is left unspecified. In such a case, you should choose an appropriate non-failing interpretation.
`expr: INV expr`	`y = 0` `x = ~y`	Invert all of the bits in y. This flips all 32 bits. Here, x becomes an int with all bits 1, in other words equivalent to -1.
`expr: BINV expr`	`y = 0` `x = !y`	Invert only the least-significant bit of y. In this case, x becomes 1.
`expr: ID LBRACKET ensemble RBRACKET`	`y = 5` `x = y[2]`	The bracket notation allows us to access a specific bit using an index. Here y[2] gets the bit with index=2 from y and puts it in x. This means that a single bit from y is put in the LS-bit of x. All other bits of x are 0. Because y is a 32-bit integer, the only legal indices are 0 to 31. But, if an out of range value is given, it's behavior is left unspecified. In this case, you should choose an appropriate non-failing interpretation.
`EXPAND LPAREN ensemble RPAREN`	`x = expand (1)`	The expand operator will fill up all 32-bits of an integer with the LS-bit of its argument. In this case, x will become an integer with all ones.

`expr: REDUCE AND LPAREN ensemble RPAREN`	`z = expand(1)`
	`y = reduce& z`
	`w = reduce| z`
	`s = reduce+ z`
		The reduce& operation loops over all of the values in z and &'s them together into a single bit. Then, it sets the LS-bit of the resulting expression. In this case, y gets the result, and &-ing a bunch of 1s produces 1, so y = 1. We also have reduce| and reduce^ and reduce+ variants. For reduce|, w = 1 as

		well. For reduce+, s would equal 32.
`statement: ID NUMBER ASSIGN ensemble ENDLINE`	`p = 0` `p1 = 1`	This statement rule allows us to set a single bit of a variable. p1 on the left means to set only the bit at index=1 using the LS-bit of the right-hand size expression. After these two statements, p = 2. **Note, it's illegal to set a bit of p before defining p.**

`statement: ID LBRACKET ensemble RBRACKET ASSIGN ensemble ENDLINE`	`p = 0`
	`x = 1`
	`p[x] = 1`
		This statement rule is similar to the previous one except that it allows an arbitrary index specified in brackets rather than a constant index.

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

Additional Specifications

Already Provided in the Starter Code

1. Generate a function in LLVM bitcode that takes as many arguments as specified by the program. The names of the parameters in the LLVM IR do not need to match the names in the P1 program, but this is a nice feature if you can do it.
2. The name for the function generated in LLVM must be taken from the name of the input file. If the file is called `tmp.p1`, the name of the function must be `tmp`.
3. 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.

Rules Your Implementation Must Follow

4. The generated function will always return an integer type. The return value is the ensemble value produced by the final rule.
5. **You may only generate one basic block**. Your generated code (output LLVM IR) must not have any additional basic blocks and may not use control flow (ifs or loops or calls).
6. **Your generated code may not use a loop or call a function to perform an operation.** However, the compiler code you write may have loops and function calls. Please make sure you understand the distinction between generated code and compiler

code.

7. If you detect illegal code during parsing (may or may not be possible), you may use the YYABORT macro to ensure that the parser exits in error. For runtime errors, like divide by zero, do not try to detect them at compile time or generate code to prevent them. Just allow them to raise exceptions in the execution of the generated code.

Implementation

Update your Git Repository

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

a. `cd path/to/ncstate_ece566_spring2024`
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`.

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

```
docker-compose up -d
```

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.

CLion Setup

Import the project into CLion and configure it to use the LLVM toolchain you setup in Project 0. Repeat the steps in that document if you don't remember all of the steps. Then, build and run your code as usual for CLion.

Note, for testing, CMake will detect the available tests. You can choose each test one by one from the drop down configuration menu. Or, you can connect to the container and run them manually as shown below.

Manual Setup

Make a build directory in the source folder, configure it using cmake, and then build and test

your code:

C	`cd ncstate_ece566_spring2024/projects/p1/C`
	`mkdir build`
	`cd build`
	`cmake ..`
	`make`
	If all goes well:
	`make test`
	If tests fail, you can see more information by running like this:
	`ctest -V`

C++	`cd ncstate_ece566_spring2024/projects/p1/C++`
	`mkdir build`
	`cd build`
	`cmake ..`
	`make`
	If all goes well:
	`make test`
	If tests fail, you can see more information by running like this:
	`ctest -V`

Development, Testing, and Debugging

1. At the end of the `make test` run, you will see how many tests passed or failed. At first, the project will not complete the tests because the scanner and parser are 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:
    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 someone else's solution!*
    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 all tests that are 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 some hints in comments within the code.

LLVM API	**Purpose**
Builder.getInt32(int)	Build a 32 bit integer.
Builder.CreateXor(Value*,Value*)	Create Xor instruction.
Builder.CreateShl(Value*,Value*)	Create shift left instruction.
Builder.CreateAnd(Value*,Value*)	Create bitwise-and instruction.
Builder.CreateOr(Value*,Value*)	Create bitwise-or instruction.
Builder.CreateAdd(Value*,Value*)	Create addition instruction.
Builder.CreateSub(Value*,Value*)	Create subtraction instruction.
Builder.CreateNot(Value*)	Create bitwise invert.
Builder.CreateMul(Value*,Value*)	Create Mulitply instruction.
Builder.CreateUDiv(Value*,Value*)	Create unsigned integer division.
Builder.CreateSRem(Value*,Value*)	Create modulo instruction
Builder.CreateLShr(Value*,Value*)	Create logical shift right (insert 0 bit into MS-bit).
Builder.CreateAShr(Value*,Value*)	Create arithmetic shift right (insert sign-bit into MS-bit).
Builder.CreateRet(Value*)	Create return instruction.

Getting Help

History shows that my specs are sometimes incomplete, incorrect, or confusing. 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. But, **DO NOT POST YOUR SOLUTION OR CODE. Discussion of your code must be saved for 1:1 discussions with the instructor or TAs.**

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 all of your source files (e.g. p1.y, p1.lex, p1.cpp/p1.c) to the Project 1 assignment page. We will test your code using the test cases provided along with some some secret cases we did not provide. If you have additional source files, you should also include your modified CMakeLists.txt.

You are not allowed to alter how the CMake files 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 566 tests that pass (~45 test cases)
20 points: Fraction of secret tests that pass (these may overlap with provided tests)
+10 points: Bonus - make the top 5 leaderboard for optimal code generated, measured in number of LLVM IR instructions. In the case of a tie of more than 5 students, points will be distributed at the instructor's discretion. To be eligible, all tests must pass. Gradescope generates a top 5 leaderboard with secret cases included.

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 466 provided tests that pass (~20 test cases)
20 points: Fraction of secret tests that pass (these may overlap with provided tests)
+10 points: Bonus - implement some of the 566 rules and pass their test cases. Rubric: 1 point per additional test case with a 10 point cutoff.

Appendix 1: Leaderboard - post your instruction count here!

To determine your score, follow these steps:

```
cd p1/C++/build/tests
./llvm-inst-count *.bc
```

This will report a total number of instructions for all test cases. If you make your own test cases, exclude those in the tally. Report the total in the middle column below along with your name and date. Please post your score in sorted order from lowest to highest. Lower counts at the top! Note, these do not guarantee bonus points. I will validate the rankings based on final code submissions.

Name or Alias	Instruction Count	Date

Appendix 2: Making Your Own Test Cases

The test cases are managed by CMake but it is easy to extend them to add your own. In the p1/C++/tests directory, you need to create two files, one for the p1 program and one that is a C program that tests the p1 program. Here's an example, let's make a program and tester called mytest, shown in the table below.

The important thing is to make the C code evaluate that mytest.p1 produced the correct code. So, we write a C function that is equivalent and compare their outputs.

mytest.p1	mytest.c

in a
final a
	#include <stdio.h>
	
	int mytest(int a); // comes from p1 file
	
	// equivalent code in C
	int mytest_tester(int a)
	{
	  return a;
	}
	
	int main()
	{
	  for(int i=-100; i<100; i++)
	  {
	  int ret = mytest(i);
	  int test = mytest_tester(i);
	  if (test != ret) {
	  printf("mytest(%d) should be %d, but got %d.\n",i,test,ret);
	  return 1; //TEST FAILED
	  }
	
	  }
	  // TEST SUCCEEDED
	  return 0;
	}

Once you're convinced your code works, open tests/CMakeLists.txt and add a line at the end like this:

```
p1_simple_test(mytest  unofficial)
```

It's critical that the both files are named mytest.p1 and mytest.c. This matching prefix must be the first string in the p1_simple_test command. The second string "unofficial" can be anything. But, I recommend making it your name or "unofficial" so you remember that it's your own test case and not mine.

Now, to see it run, just do this:

```
cd p1/C++/build
make
make test
```

When CMake runs your test, it only looks to see what main returns, either success (0) or failure (1). So, it's fine to print out a bunch of stuff to help debug your code. You can run the binary directly by doing this:

```
cd build/tests
./mytest
```

That will show you all of the output from your test case.

Appendix 3: Debugging Tips

Use lldb. Or, add some prints that tell you the file and line number, you can do it like this:

```
printf("file = %s line = %d some message", __FILE__, __LINE__);
```

This is a classic trick for making it easier to figure out where the message is coming from. __FILE__ is a macro that returns the file name, and __LINE__ is a macro that returns the line number. They would also work with std::cout.