CSE 302: Compiler Design — Lab 1

2019-09-12

1 INTRODUCTION

This lab is intended to get you set up with your chosen programming language, development libraries, and associated tools. We will provide skeleton implementations in two languages: Java and OCaml. *You are free to use any language you wish*, but you will have to start from scratch if you don't choose one of these languages.

The goal of this lab will be to write a simple code generator for straight line code, i.e., code that contains no control structures. To keep things very simple, you will target a very limited subset of the C programming language; you will then use a standard C compiler to produce the final binary. You don't need to know any assembly language for this lab. The source and target languages are explained in the next two sections.

This lab will be assessed. It is worth 5% (i.e., 1 point) of your final grade. To get full credit, you must submit the final version of your compiler in 1 week, i.e., strictly before `2019-09-19 00:00:00 CEST`. Every additional hour past this deadline incurs a $(20/24)\% = 0.83\%$ penalty.

Before you begin

1. Set up your version control system. We recommend using `git` and private repositories on Github. E-mail a link to your repository to the course instructor before the due date so we can check out/-clone your code.
2. Quickly scan section 4 to find out the requirements for building and running your compiler. We will be grumpy if you don't follow these requirements carefully, since we expect to do lots of automated testing as part of the grading process.

2 SOURCE LANGUAGE: BX0

The source language, `BX0`, is a purely calculational language. There are no control structures such as loops, conditionals, or functions, nor is there any way to produce any output except by means of the `print` statement. This language works with data of a single type, signed 64 bit integers in 2's complement representation. This can represent all integers in the range $[-2^{63}, 2^{63})$.

The grammar of the language is shown in figure 1. A ⟨program⟩ consists of a sequence of ⟨statement⟩s, each terminated with a semi-colon, although the final semi-colon is optional. There are two kinds of statements: computation statements (called ⟨move⟩s) and ⟨print⟩ statements. A ⟨move⟩ computes the value of an ⟨expression⟩ and stores the result in a ⟨variable⟩. A ⟨print⟩ statement can be used to print out the value of a single ⟨variable⟩ on a line by itself on the standard output.

An ⟨expression⟩ is made up of ⟨number⟩s and ⟨variable⟩s combined with the usual arithmetic operators (`+`, `-`, `*`, `/`, `%`) and bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`). The bitwise operators work on the underlying bit pattern of the word, so two's complement identities such as `x + ~x == -1` will be true. Note that the right shift operator, `>>`, stands for *arithmetic* right shift in 2's complement arithmetic, meaning that the new bits to the left are copies of the sign bit, not `0`s.

```
⟨program⟩ ::= ⟨statement⟩ (";" ⟨statement⟩)* ";"?

⟨statement⟩ ::= ⟨move⟩ | ⟨print⟩
⟨move⟩ ::= ⟨variable⟩ "=" ⟨expression⟩
⟨print⟩ ::= "print" ⟨expression⟩

⟨expression⟩ ::= ⟨variable⟩ | ⟨number⟩
               | ⟨expression⟩ ⟨binop⟩ ⟨expression⟩ | ⟨unop⟩ ⟨expression⟩
               | "(" ⟨expression⟩ ")"

⟨variable⟩ ::= [A-Za-z_][A-Za-z0-9_]*                    (except keywords)
⟨number⟩ ::= [0-9]+                          (must be in range [−2^63, 2^63))
⟨binop⟩ ::= "+" | "-" | "*" | "/" | "%" | "<<" | ">>" | "&" | "|" | "^"
⟨unop⟩ ::= "-" | "~"
```

Figure 1: The BX0 language

BX0 (currently) does not have any Booleans, and therefore does not support (in)equations or Boolean operators. These will be added in the future when the language gets support for conditionals. The table below lists the operators, their arity, and their precedence. Note that higher precedence binds tighter, so `1 + 2 * 3` would implicitly stand for `1 + (2 * 3)` and not `(1 + 2) * 3`.

operator	description	arity	precedence
`|`	bitwise or	binary	10
`^`	bitwise xor	binary	20
`&`	bitwise and	binary	30
`<<`, `>>`	bitwise shifts	binary	40
`+`, `-`	addition, subtraction	binary	50
`*`, `/`, `%`	multiplication, division, modulus	binary	60
`-`	integer negation	unary	70
`~`	bitwise complement	unary	80

Here are two example BX0 programs.

```
x = 10;
print x;
y = 2 * x;
print y;
z = y / 2;
print z;
w = z - x - y;
print w;
u = w + w;
print u;
v = -u;
print v;
m = x + y * 2;
print m;
n = (x + y) * 2;
print n;
```

Simple tests

```
x = 0;
y = 0;
print x;
z = x;
x = y;
y = y + z;
print x;
z = x;
x = y;
y = y + z;
print x;
z = x;
x = y;
y = y + z;
print x;
```

Fibonacci

These two programs do not cover every kind of construct in BX0. Write your own tests as well.

3 TARGET LANGUAGE: C (A TINY FRAGMENT)

In this first lab you will compile BX0 to a very restricted fragment of C, and then use the C compiler gcc to produce the final binary. The fragment of C you will target will always look like this:

```c
#include <stdio.h>
#include <stdint.h>
static const char* const __print_fmt = "%ld\n";
int main() {
  // variable declarations such as
  int64_t x0;
  int64_t x1;
  // etc.
  // ----------------------------------------
  // your compiled code here, such as
  x0 = 20;
  x1 = 22;
  x0 = x0 + x1;
  printf(__print_fmt, x0);
  // etc.
  // ----------------------------------------
  return 0; // do not change
  // successful executions should have exit code 0
  // unsuccessful executions can have any non-zero exit code or fault
}
```

From the BX0 program you will design the corresponding variable declarations and compiled statements to populate the main() function.

Variable declarations The variable declarations must be listed at the top of the main() function and each variable must have C type int64_t. *No variable declarations are allowed anywhere else*; in particular, they must not occur in the middle of compiled code.

Compiled code The expressions and statements of the BX0 program will be compiled into a list of C instructions. The following is the complete list of allowed C instructions. *You are not allowed to use any other kind of* C *syntax*. In all of the following, x, y, and z refer to (not necessarily pairwise distinct) explicitly declared variables at the top of main(), and the literal number 42 stands for all literal numbers.

Instruction	Description
`x = 42;`	Set a variable to a literal number.
`x = y;`	Copy the value of variable y to variable x.
`x = y • z;`	Compute the value of a binary operator application, where • can be any binary operator (+, -, *, /, %, &, |, ^, <<, >>), and store the result in the variable x. Neither of the arguments y or z can be a literal number.
`x = † y;`	Compute the value of a unary operator application, where † can be any unary operator (-, ~), and store the result in the variable x. The argument y cannot be a literal number.
`printf(__print_fmt, x);`	Print the value of the variable x on a line by itself in the standard output.

4 DELIVERABLES

To get full credit for this lab, you will have to submit the complete source to your compiler, which should be buildable and runnable with Linux. Include the following two files at the top level.

- A `README` (or `README.md`) file listing all the tools required to build the compiler. These tools must be available to install on a standard recent Debian-derived Linux installation (which includes Ubuntu), i.e., for any tool `t`, the following command should work: `apt-get install t`.
- A `Makefile` that can be used to build the compiler.

Building the compiler To build the compiler, it should suffice to execute the following command from the top level directory:

```
$ make
```

This should produce an executable file (or link) called `bx0.exe`. The `.exe` extension is required on all platforms, even MacOS and Linux.

Running the compiler To use your compiler, it should suffice to run the following command on the example input file `example.bx`.

```
$ ./bx0.exe example.bx
```

This should produce two files for the given input file `example.bx`:

- The compiled `C` source file, `example.c`.
- The binary compiled with `gcc`, named `example.exe`.

This is what we expect to see with the `file` command:

```
$ file example.c
example.c: C source, ASCII text
$ file example.exe
example.exe: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
    ..., not stripped
```

Testbench You are encouraged to write your own test `BX0` programs and then routinely test your compiler against your programs. Place every test program in the directory `test/` at the top level of your code. Each test program, say `test.bx`, should be accompanied with the expected output, `test.expected`. When the program is compiled, its output must exactly match the expected output.

```
$ test/test.exe > test/test.actual
$ diff test/test.actual test/test.expected || echo "Outputs do not match"
```

To run all the tests at once, it should be sufficient to run the following from the toplevel directory.

```
$ make testbench
```

5 DESIGNING THE COMPILER

Here are a few hints about writing the compiler from scratch.

Lexing and parsing We strongly recommend using a lexer and parser generator instead of rolling your own. Your language probably has something like the `flex`/`bison` tools that are used in the example skeleton of the compiler in C++. Try to adapt it to your chosen programming language. We will cover lexing and parsing in more detail in upcoming labs.

Maximal munch The suggested code-generation algorithm is *maximal munch* (as seen in lecture). Remember that maximal munch requires a syntax tree, which your parser should produce. Feel free to implement either the top-down or the bottom-up maximal munch algorithm that you saw in the lecture and in the handout.

Improvements While it is not required, you can attempt to improve the code generated by maximal munch using copy-propagation, where instructions of the form `x = y` are removed and uses of `x` replaced with those of `y` (as long as `y` does not itself change). You can simplify the condition by implementing *static single assignment* (SSA) for straight line code as you saw in the lecture.