15-411 Compiler Design: Lab 1
Fall 2008

Instructor: Frank Pfenning
TAs: Rob Arnold and Eugene Marinelli

Test Programs Due: 11:59pm, Tuesday, September 9, 2008
Compilers Due: 11:59pm, Tuesday, September 16, 2008

1 Introduction

Writing a compiler is a major undertaking. In this course, we will build not just one compiler, but several! (Actually, each compiler will build on the previous one. But even so, each project will require a serious amount of work.) To get you off to a good start, we provide you with a compiler for a simple language called *L1*. While it is complete, it does not generate real assembly code, only instructions in a pseudo assembly language with simple instructions and the assumption that it has arbitrarily many registers.

For this project, your task is to extend this compiler to translate *L1* source programs into target programs written in actual x86-64 assembly language. To do this, the main changes that you will have to make are to the instruction selector and the addition of a register allocator. It must be possible to assemble and link the target programs with our runtime environment using `gcc`, producing a standard executable.

Projects should be done either individually or in pairs. You are strongly encouraged to work in teams of two. You may use the discussion board `academic.cs.15-411` to find a partner or email the instructor soon as possible, as the test cases will be due in a week. Please inform the instructor as soon as you have teamed up with someone so we can set up appropriate Autolab accounts for each team.

The first project is not designed to be very time consuming or difficult. In particular, the total amount of code you will have to write is not tremendously large. Nevertheless, as this is your first attempt at working with the compiler code, there is a relatively large amount of code to understand before you can get started, and you will also have to understand thoroughly the concepts of instruction selection and register allocation before attempting to implement anything. We therefore recommend that you get an early start.

2 *L1* Syntax

The compilers we provide to you translate source programs written in *L1*. The syntax of *L1* is defined by the context-free grammar shown in Figure 1. The language is similar to the "straight-line programs" language from Chapter 1 of the textbook.

⟨program⟩ ::= **{** ⟨stmt⟩* **return** ⟨exp⟩ **;** **}**

⟨stmt⟩ ::= ⟨ident⟩ ⟨asop⟩ ⟨exp⟩ **;**

⟨exp⟩ ::= **(** ⟨exp⟩ **)** | ⟨intconst⟩ | ⟨ident⟩ | **-** ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩

⟨ident⟩ ::= [**A-Z_a-z**][**0-9A-Z_a-z**]*

⟨intconst⟩ ::= [**0-9**][**0-9**]* (in the range $0 \leq$intconst$< 2^{32}$)

⟨asop⟩ ::= `=` | `+=` | `-=` | `*=` | `/=` | `%=`

⟨binop⟩ ::= `+` | `-` | `*` | `/` | `%`

The precedence of unary and binary operators is given in Figure 2.
Non-terminals are in ⟨brackets⟩.
Terminals are in **bold**.

Figure 1: Grammar of *L1*

Operator	Associates	Class	Meaning
`-`	right	unary	unary negation
`* / %`	left	binary	integer multiplication, division, modulo
`+ -`	left	binary	integer addition, subtraction
`= += -= *= /= %=`	right	binary	assignment

Figure 2: Precedence of operators, from highest to lowest

Comments

L1 source programs may contain C-style comments of the form `/* ... */` for multi-line comments and `//` for single-line comments. Multi-line comments may be nested (and of course the delimiters must be balanced). Also, `#` should be considered as starting a single-line comment as such lines will be used as directives for testing and possibly other uses later in the class.

3 *L1* Static Semantics

The *L1* language does not have a type system as such. All variables are bound to integers and are implicitly declared when first assigned a value. Programs that attempt to read a variable before assigning to it should cause the compiler to generate a compile-time error message.

4 *L1* Dynamic Semantics

Statements have the obvious operational semantics, although there are subtleties regarding the evaluation of expressions. Each statement is executed in turn. To execute a statement, the expression on the right-hand side of the assignment operator is evaluated, and then the result is assigned to the variable on the left-hand side, according to the type of assignment operator. The meanings of the special assignment operators are given by the following table, where x stands for any identifier and e for any expression.

x `+=` e	$\equiv$	x `=` x `+` e
x `-=` e	$\equiv$	x `=` x `-` e
x `*=` e	$\equiv$	x `=` x `*` e
x `/=` e	$\equiv$	x `=` x `/` e
x `%=` e	$\equiv$	x `=` x `%` e

The result of executing an *L1* program is the value of the expression in the program's `return` statement.

Integer Operations

The integers of this language are in two's complement representation with a word size of 32 bits. Addition, subtraction, multiplication, and negation have their meaning as defined in arithmetic modulo 2^{32}. In particular, they can never raise an overflow exception.

In order to be able to parse the smallest integer, -2^{31}, we allow integer constants c in the source in the range $0 \leq c < 2^{32}$; constants larger than $2^{31} - 1$ are interpreted modulo 2^{32} as usual in two's complement representation.

The division i/k returns the truncated quotient of the division of i by k, dropping any fractional part. This means it always rounds towards zero.

The modulus $i \,\%\, k$ returns the remainder of the division of i by k. The modulus has the same sign as i, and therefore

$$(i/k) * k + (i \,\%\, k) = i$$

Division i/k and modulus $i \,\%\, k$ are required to raise a `divide` exception if either $k = 0$ or the result is too large or too small to fit into a 32 bit word in two's complement representation.

Fortunately, this prescribed behavior of integer operations coincides with the hardware behavior of appropriate instructions.

5 Project Requirements

For this project, you are required to hand in a complete working compiler for *L1* that produces correct target programs written in Intel x86-64 assembly language. When we grade your work, we will use the `gcc` compiler to assemble and link the code you generate into executables using the provided runtime environment on the lab machines.

Your compiler and test programs must be formatted and handed in via Autolab as specified below. For this project, you must also write and hand in at least six test programs, two of which must fail to compile, two of which must generate a runtime error, and two of which must execute correctly and return a value.

Test Files

Test files should have extension `.l1` and start with one of the following lines

`#test return` i	program must execute correctly and return i
`#test exception`	program must compile but raise a runtime exception
`#test error`	program must fail to compile due to an *L1* source error

followed by the program text. All test files should be collected into a directory `test/` (containing no other files) and bundled as a tar file `tests.tar` with

```
% tar -cvf tests.tar tests/
```

to be submitted via the Autolab server.

Compiler Files

The files comprising the compiler itself should be collected in a directory `compiler/` which should contain a `Makefile`. **Important:** You should also update the `README` file and insert a description of your code and algorithms used at the beginning of this file. This will be a crucial guide for the grader.

Issuing the shell command

```
% make l1c
```

should generate the appropriate files so that

```
% bin/l1c <args>
```

will run your *L1* compiler. The command

```
% make clean
```

should remove all binaries, heaps, and other generated files.

In order to keep the files you hand in to a reasonable size, please clean up the directory and then bundle it as a tar file `compiler.tar`, for example with

```
% cd compiler
% make clean
% cd ..
% tar -cvf compiler.tar compiler/
```

to be submitted via the Autolab server. If you are using `svn` or `cvs`, you may want to add the switch `--exclude .svn` or `--exclude CVS` to the `tar` command for a cleaner hand-in.

What to Turn In

Hand-in on the Autolab server:

- At least 6 test cases, two of which successfully compute a result, two of which raise a runtime exception, and two of which generate an error. Submit `tests.tar` with the directory `tests/` with only the test files via the Autolab server. The server will test your test files and notify you if there is a discrepancy between your answer and the outcome of the reference implementation. If you feel the reference implementation is in error, please notify the instructors.

  Test cases are due **11:59pm on Tue Sep 9**.

- The complete compiler. Submit `compiler.tar` with the directory `compiler/` after applying a `make clean`. The Autolab server will build your compiler, run it on all extant test files, link the resulting assembly files against our runtime system (if compilation is successful), execute the binaries, and finally compare the actual with the expected results.

  Compilers are due **11:59pm on Tue Sep 16**.

6 Notes and Hints

Much can be learned from studying the reference implementation. In addition, you should read the textbook and lecture material on instruction selection and register allocation. The written homework may also provide some insight into and practice with the algorithms and data structures needed for the assignment.

Register Use

We recommend implementing a global register allocator based on graph coloring. While this may be not be strictly necessary for such a simple source language, doing so now will save work in later projects where high-quality register allocation will be important. The recommended algorithm is based on chordal graph coloring as presented in lecture and detailed in the lecture notes. We recommend that you first implement register allocation without spilling, which would get almost full credit since few programs will need more than the registers available on the x86-64 processor.

We do not recommend that you implement register coalescing for this lab, unless you already have a complete, working, beautifully written compiler and some free time on your hands.

Runtime Environment

Your target code will be linked against a very simple runtime environment. The runtime contains a function `main()` which calls a function `_l1_main` and then prints the returned value. If your compiler generates a target file called `foo.s`, it will be linked with the runtime into an executable using the command, `gcc foo.s l1rt.c`. This means that your compiler should generate target code for a function called `_l1_main`, and that the `return` statement at the end of the *L1* source program should be compiled into an x86 `ret` instruction. According to the calling conventions, the register `%eax` will have to hold the return value.

It is extremely important that register usage and calling conventions of the x86-64 architecture are strictly adhered to by your target code. Failure to do so will likely result in weird, possibly nondeterministic errors.

You can refresh your memory about x86-64 assembly and register convention using Randal Bryant and David O'Hallaron's textbook supplement on x86-64 Machine-Level Programming available from the resources page on the course website. The Application Binary Interface (ABI) specification linked from the web page will also be important, if not now, then later in the course. Finally, the processor manual contains useful on the details of the instructions, although we use the GNU Assembler conventions rather than Intel notation.

The tests will be run in the standard Linux environment on the lab machines; the produced assembly code must conform to those standards. We recommend the use of `gcc -S` to produce assembly files from C sources which can provide template code and assembly language examples.

If your compiler detects any (compile-time) errors in the source program, it should exit with a non-zero return code. If compilation succeeds and target code is generated, the compiler should then exit with a return code of 0.

Good Coding Practices

Please remember that **your code will be read and graded** by the instructor or a teaching assistant. This means your code must be readable. This also means that the code should be broken up along natural module boundaries. Some specific pieces of advice are listed below. As usual, such generalizations have to be taken with a grain of salt, but we hope they may still be useful.

- Every `structure` should have a `signature` which determines its interface.
- Seal structures by ascribing signatures with '`:>`' and not '`:`'. The latter might accidentally leak private information about the structure.
- Use functors only when you instantiate them more than once.
- Format your code to a line width of no more than 100 characters.
- Tabs, if used at all, should have a standard width of 8.
- Do not use `open LongStructureName` because the reader will then be unable to tell where identifiers originate. It can also lead to unfortunate shadowing of names. Instead, use, for example, `structure L = LongStructureName` inside a structure body and qualify identifiers with '`L.`'.
- Use variable names consistently.
- Use comments, but do not clutter the code too much where the meaning is clear from context.
- Avoid excessive uses of combinators such as `foldl` and `foldr`. They tend to be easier to write than to read and understand.
- Develop techniques for unit testing, that is, testing modules individually. This helps limit the problem of nasty end-to-end bugs that are very difficult to track.
- Do not prematurely optimize. Write clear, simple code first and optimize only as necessary, when bottlenecks have been identified. Compiler speed is not a grading criterion!

- Do not prematurely generalize. Solve the problem at hand without looking ahead too much at future labs. Such generalizations are unlikely to simplify later coding because it is generally very difficult to anticipate what might be needed. Instead, they may make the present code harder to follow because of unmotivated pieces.
- Be clear about the data structures and algorithms you want to implement before starting to write code.