15-411 Compiler Design, Fall 2016 Lab 1

Jan and co.

Test Programs Due: 9:00am, September 13
Compilers Due: 9:00am, September 20

1 Introduction

Writing a compiler is a major undertaking. In this course, we will build not just one compiler, but several! Each compiler will build on the previous one, so careful thought and design are very important in the first labs. You *will* be re-using and re-writing code. To get you off to a good start, we provide you with a compiler for a simple language called L1. The compiler we have provided targets a very simple "abstract assembly" language with an infinite number of registers and a very simple instruction set which includes just arithmetic instructions.[1]

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 change that you will have to make is modifying the instruction selector and coming up with some scheme for register allocation. It must be possible to assemble and link the target programs (that is, the x86-64 assembly output from your compiler) with our runtime environment using `gcc`, producing a standard executable. You may also have to make some small changes to the lexer and parser to be in compliance with the current L1 specification.

Projects should be done either individually or in pairs. You are strongly encouraged to work in teams of two. Each team (or individual) is assigned a group name by the instructor. You will be assigned a group name after completing the group committment form at

`https://goo.gl/forms/tL0Mu9MgHnDQO5FQ2` .

You need a Github account to complete the form. You will not be able to download the starter code or hand in labs without accounts for the `git` repository, so you must have a group name.

The first project is neither the most difficult nor the most time consuming assignment in the course. Especially if you implement a more trivial register allocator, the total amount of code you will have to write is relatively small. Nevertheless, as this is your first attempt at working with the compiler code, there is a relatively large amount of material 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.

Keep in mind that all of the following may consume a substantial amount of time:

- Sorting out administrative problems. Making sure that you have git access and getting used to git.

[1] There are some limitations to the starter code we give you. Please consult the last section on Supported Programming Languages for more on these.

- Getting to know your partner. Working with a partner is an important aspect of this class. It is important to schedule time to work, find a preferred working environment, and develop a good group dynamic early in the semester. *We strongly suggest that you schedule time for reading and discussing each other's code at least twice weekly.*
- Reading and possibly porting the entire starter code so that you understand every bit of what your compiler is doing. This is essential, because you will be editing every stage of the compiler in future labs, which will include any starter code that we distribute to you.
- Getting used to the libraries available for your programming language of choice. The libraries that your compiler depends on will likely include a parser combinators, a LL(1) parser generator or a LALR(1) parser generator, and possibly a separate lexer generator. Please make sure that you can find the specification documents for the libraries you use.
- Last, but certainly not the least, generating code requires attention to detail. Please be prepared to read through the Intel Developer Manuals for precise behaviour of the instructions you will emit, and the GNU assembler documentation for the syntax that you should use. As a matter of academic integrity, you shouldn't be reading the code of any publicly existing compilers during this course. However, reading any and all external resources about x86-64 assembly and looking at the assembmly emitted by compilers like `gcc` is both allowed and encouraged.

To emphasize again, *all the projects in this course are cumulative.* Therefore, falling behind in Lab 1 could be disastrous. Please get an early start, and remember that we're here to help.

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 a fragment of the C0 introductory programming language, and is similar to the "straight-line programs" language from Chapter 1 of the textbook.

Lexical Tokens

The concrete syntax of L1 is based on ASCII character encoding.

Whitespace and Token Delimiting

In L1, whitespace is either a space, horizontal tab (`\t`), vertical tab (`\v`), carriage return (`\r`), linefeed (`\n`), or formfeed (`\f`) character in ASCII encoding. Whitespace is ignored, except that it terminates tokens. For example, `+=` is one token, while `+ =` is two tokens.

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). Contracts in C0 start with `//@` or `/*@`; they are simply treated as comments in L1.

⟨program⟩	::=	**int main ()** ⟨block⟩	
⟨block⟩	::=	`{` ⟨stmts⟩ `}`	
⟨stmts⟩	::=		
	|	⟨block⟩	
	|	⟨stmt⟩ ⟨stmts⟩	
⟨stmt⟩	::=	⟨decl⟩ **;**	
	|	⟨simp⟩ **;**	
	|	**return** ⟨exp⟩ **;**	
⟨decl⟩	::=	**int ident**	
	|	**int ident** `=` ⟨exp⟩	
⟨simp⟩	::=	⟨lvalue⟩ ⟨asnop⟩ ⟨exp⟩	
⟨lvalue⟩	::=	**ident**	
	|	**(** ⟨lvalue⟩ **)**	
⟨exp⟩	::=	**(** ⟨exp⟩ **)**	
	|	⟨intconst⟩	
	|	**ident**	
	|	⟨exp⟩ ⟨binop⟩ ⟨exp⟩	
	|	`-` ⟨exp⟩	
⟨intconst⟩	::=	**decnum**	(in the range $0 \leq \text{intconst} \leq 2^{31}$)
	|	**hexnum**	(in the range `0x00000000` to `0xffffffff`)
⟨asnop⟩	::=	`=` | `+=` | `-=` | `*=` | `/=` | `%=`	
⟨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

```
<ident>      ::= [A-Za-z_][A-Za-z0-9_]*

<num>        ::= <decnum> | <hexnum>
<decnum>     ::= 0 | [1-9][0-9]*
<hexnum>     ::= 0[xX][0-9a-fA-F]+

<unop>       ::= -
<binop>      ::= + | - | * | / | %
<asnop>      ::= = | += | -= | *= | /= | %=

<reserved>   ::= --
```

Figure 3: Lexical Tokens

Reserved Keywords

The following are reserved keywords or lexical tokens and cannot appear as a valid token in any place not explicitly mentioned as a terminal in the grammar.

```
struct typedef if else while for continue break
return assert true false NULL alloc alloc_array
int bool void char string
```

Many of these keywords are unused in L1. However, the specification treats these as keywords to maintain forward compatibility of valid C0 programs.

Other Tokens

L1 may treat certain strings as tokens even though they never appear as terminals in the grammar. We do this in order to maintain forward compatibility with the remaining labs and C0. In most cases, using a token such as `<<` (eventually representing a left shift) in L1 will simply lead to a lexing or parsing error. The token `--` represents special case. An expression such as `--5` or `x--2` must be rejected in L1 for reasons of forward compatibility (write `-(-5)` or `x- -2` instead). The token is listed as *reserved* in Figure 3.

3 L1 Static Semantics

The L1 language does not have a very interesting type system. Most constraints imposed by the type system are for the time being imposed by the grammar instead.

Declarations

Though declarations are a bit redundant in a language with only one type and no interesting control flow construct, we require every variable in the function to be declared (with the correct type, in this case `int`) before being used, altough statements and declarations can be mixed. We do this to ensure that the valid L1 programs are forward compatible with respect to future labs, and C0. Variables may not be redeclared.

Initialization Checking

C0 requires that along each control flow path that starts from a variable declaration, that variable is initialized before it is used.[2] Using a variable before it is initialized must therefore generate a compile-time error message. An odd case arises when there is no control flow path connecting the use of a variable to its declaration. In L1, this can arise when a `return` statement separates the declaration of a variable from its use. In such a case, the variable need not be initialized. However, each variable must still be declared and the use lie in the scope of the declaration.

Return Checking

C0 requires that every control flow path in the body of a must end with a return statement (unless the return type is `void`). To maintain forward compatibility, we require that L1 programs contain a return statement, but not necessarily only one or as the last statement.

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.

[2] Even if a variable declaration itself is after a `return` statement and therefore can't be executed, we still treat it as the beginning of a valid control flow path for this purpose.

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.

Decimal constants c in a program must be in the range $0 \leq c \leq 2^{31}$, where $2^{31} = -2^{31}$ according to two's complement modular arithmetic. Hexadecimal constants must fit into 32 bits.

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 the relevant instructions.

5 Lab Requirements

For this lab, you are required to hand in a complete working compiler for L1 that produces correct target programs written in Intel x86-64 assembly language.

We also require that you document your code. Documentation includes both inline documentation and a file `compiler/lab1/README` that explains the design decisions underlying the implementation along with the general layout of the sources. If you use publicly available libraries, you are required to indicate their use and source in the `README` file. If you are unsure whether it is appropriate to use external code, please discuss it with course staff.

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 source files and test programs must be formatted and handed in via Github as specified below.

Test Program

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 div-by-zero`	program must compile but raise SIGFPE
`//test error`	program must fail to compile due to an L1 source error

All test files should be submitted in the directory

`tests`

in the root directory of the repository. This directory should contain no other files.

Compiler Source Files

The files comprising the compiler itself should be collected in a subdirectory of the `compiler` directory named `lab1`. The `compiler` directory contains a `Makefile`, which you may edit, but do not need to edit.

Issuing the shell command

```
% make lab1
```

from within the `compiler` directory should generate the appropriate files so that

```
% bin/c0c <args>
```

will run your L1 compiler. 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. The command

```
% make clean
```

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

Important: You should also update the `README` file and insert a description of your code and algorithms used at the beginning of this file.

Your compiler is also expected to recognize a flag `-t` which, when present on the command line, stops the compiler immediately after typechecking and before the rest of the compiler runs. The exit code of your compiler should indicate success (0) if the code is well-formed, and failure (1) otherwise. If your compiler indicates success when run with `-t`, then it should be able to compile the file without further errors. Your compiler should also recognize the flags `-ex86-64` and `-O0`, but these flags can be ignored for now. These flags will be used for later assignments; they are explained in file `compiler/c0c-spec.txt`.

Runtime Environment

Your target code will be linked against a very simple runtime environment. The runtime contains a function `main()` which calls a function `_c0_main` and then prints the returned value. If your compiler is given a well-formed input file `foo.l1` as a command-line argument, it should generate a target file called `foo.l1.s` in the same directory as `foo.l1`. The file `foo.l1.s` will be linked with the runtime into an executable using the command `gcc -m64 foo.l1.s ../runtime/run411.c`. This means that your compiler must generate target code for a function called `_c0_main`, and that the `return` statement at the end of the L1 source program should be compiled into an x86-64 `ret` instruction. According to the calling conventions, the register `%eax` must hold the return value. Your `_c0_main` function must preserve all callee-save registers so that our main function can work correctly.

Using the GitHub Repository

To obtain the starter code for this lab executed the following shell commands where `<teamrepo>` is the name of your team in lowercase.

```
  % git clone https://github.com/CMU-15-411/dist.git
  % cd dist
i  % git clone https://github.com/CMU-15-411/<teamrepo>.git compiler
```

The first repository (*dist.git*) is used to distribute starter code, tools, and tests. It will be further populated during the course. The second repository (*teamrepo.git*) is your team's repository. We set up an working environment in which your team's repository is in the subdirectory `compiler`. The implementation of this lab is assumed to be in `compiler/lab1`. More details can be found in the `README.md` file in *dist.git*.

To register a submission with the autograder, you will use git branches in your team's remote repository. Pushing to the branch `test1` will trigger the autograder for the test files of this lab. Pushing to the branch `lab1` will trigger the autograder for compiler of this this lab. There is no penalty for multiple submissions; you are highly encouraged to push to the `lab1` branch early and often.

Do not include any compiled files or binaries in the repository!

What to Turn In

You may turn in code and have it autograded as many times as you like, without penalty. In fact, we encourage you to hand in to verify that the autograder agrees with the driver results that you use for development, and also as insurance against a last-minute rush. The hand-in with the highest grade will count.

You will submit

Before Tuesday, September 13, 9am At least 10 test cases, at least two of which successfully compute a result, at least two of which raise a runtime exception, and at least two of which generate an error. You submit by pushing to the branch `test1` of your team's remote repository on GitHub. The directory `tests` should only contain your test files. The autograder 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.

Before Tuesday, September 20, 9am The complete compiler. You submit by pushing into the branch `lab1` in your team's remote repository on GitHub. The directory `compiler/lab1` should contain only the sources for your compiler and be submitted as described above. The autograder will build your compiler, run it on all existing 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.

6 Notes and Hints

We recommend reading lecture material on instruction selection and register allocation, and the optional textbook if you require further information. The written homework may also provide some insight into and practice with the algorithms and data structures needed for the assignment.

Register Allocation

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.

Code Generation

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 (second or third revision) or the published slides from 15-213. 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 data on the details of the instructions. Note that we use the GNU Assembler, which uses a different syntax than that given in the Intel manuals.

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.

Development Guidelines

- Format your code to a line width of no more than 80 characters.
- Tabs, if used at all, should format well with a width of 8. Some languages like ML do not indent very well with tabs, so we recommend against tabs altogether.
- Use variable names consistently.
- Use comments, but do not clutter the code too much where the meaning is clear from context.
- 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.
- Use git to your advantage – make sure you're using source control to keep track of history on your codebase, and if you're familiar with it, use pull requests to internally review code!
- Do not prematurely optimize. Write clear, simple code first and optimize only as necessary, when bottlenecks have been identified. Timeouts for compilation times are designed to be lenient.
- Be clear about the data structures and algorithms you want to implement before starting to write code.

- You may encounter performance problems in the course of your development. A profiler is definitely a useful tool in identifying the bottlenecks in your compiler. However, you must be careful in interpreting the information provided by the compiler. A particular pass in your compiler might be taking too long either because you inefficiently implemented it, or because it is inherently a hard problem, and a previous pass generated an unusually large input for subsequent passes.

- 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. We recommend that you take this chance to gain experience in incremental software development, a useful skill that is quite orthogonal to modular software development.

7 Supported Programming Languages

This course does not require students to use any specific programming language to implement their compilers. However, we cannot support every programming language in existence. We have distributed starter code for Standard ML, OCaml, Haskell, and Java. We have some slightly out-of-date starter code for other languages, including, F# and Scala, which can be made available as-is upon request. We strongly recommend that students who are well versed in a functional language take that option, because a language that supports algebraic datatypes and pattern matching allows for significantly more compact and readable code for programs that do a lot of symbol processing. If you would like to verify this claim, feel free to compare the volume of starter code in Java to that of the SML or Haskell code.

If you plan to use specific starter code, say for OCaml, then after you download the public class code and your group's repository, you should run

```
% cp -R starter/ocaml compiler/lab1
% cd compiler
% git add lab1
```

before starting your work in the `compiler/lab1` directory.

Students wishing to use other programming languages are encouraged to consult the starter code for other languages ahead of time and adapt the necessary starter code and build infrastructure. Please contact the course staff for any technical support.

We **strongly** recommend that you find a partner in order to avoid being overwhelmed by the sheer volume of code. This is doubly true if you wish to use a programming language that is significantly more verbose than Standard ML. Sources of verbosity may include the lack of algebraic datatypes, lack of a module system, explicit memory management, poor support for parser generators, etc.

Please remember that your code will be the basis for future labs, and that you are working with a partner. This means your code must be readable. This also means that the code should be broken up along natural module boundaries. Finally, be careful in choosing your programming language, because you effectively commit to using it for the rest of the semester.