15-411 Compiler Design, Lab 1 (Spring 2024)

Jan and co.

Test Cases Due: 11:59 PM, Januaray 26, 2024
Compilers Due: 11:59 PM, February 2, 2024

1 Introduction

Writing a compiler is a major undertaking. In this course, you will write not just one compiler, but several! Each compiler will build on the previous one, so careful thought and design are crucial in the first labs. You *will* be reusing and rewriting code. To get you off to a good start, we provide you with a compiler for a small language called L1. The provided compiler targets a simple abstract assembly language with an infinite number of registers and an instruction set consisting solely of arithmetic operations.

For this project, your task is to extend this compiler to translate L1 source programs into x86-64 assembly target programs. To do this, the main change that you will have to make is modifying the instruction selector and dealing with the now-finite number of registers. 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.

The first project is neither the most difficult nor the most time consuming assignment in the course. *Thus, we highly recommend using this time to implement register allocation for the entirety of lab1 and not just for the optional (but strongly encouraged) checkpoint.*

Although the total amount of code you will have to write is relatively small, 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. Make sure you thoroughly understand the concepts of instruction selection and register allocation before attempting to implement anything.

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

- Sorting out administrative problems: making sure that you have GitHub access and getting used to git.
- 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 team 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. For parsing alone, your compiler might depend on a parser combinator library, a LL(1) parser generator, or a LALR(1) parser generator, and possibly even on a separate lexer generator. Please make sure that you can find the specification documents for the libraries you use.
- Lastly, but certainly not least, generating code requires attention to detail. Please be prepared to read through the Intel Developer Manuals for precise behavior 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 assembly emitted by compilers like* `gcc` is both allowed and encouraged. The latter is sometimes helpful if the x86 manual is unclear.

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 ident ()** ⟨block⟩	(the typechecker ensures this is "main")
⟨block⟩	::=	`{` ⟨stmts⟩ `}`	
⟨stmts⟩	::=	ϵ (empty)	
	|	⟨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 $0 \leq$ 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

There is one ambiguity in the C0 specification for comments: it is not specified whether the string

```
5 *//one line comment
3
```

should be parsed as "`5*3`" (parsing and ignoring a line comment) or as a syntax error (since the closing block comment marker, "`*/`" was not matched with a corresponding opening block comment marker). We will not test your compiler with any programs containing the string "`*//`", and you should not submit any test cases containing this string.

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

Future labs will have a larger set of meaningful tokens than the L1 grammar. Most of these tokens don't conflict with the L1 grammar and can be tokenized by the lexer even if they are not used by the parser. For instance, `<<` will eventually represent a left shift, but in L1 is not a valid terminal, so any program containing `<<` should lead to a lexer error, or if your compiler already tokenizes it, a parser error.

A notable exception is `--`. In future labs, `--` will be tokenized as part of the decrement statement, so the string `--5` should lead to a parser error. However, the L1 grammar theoretically allows `--5` to be parsed as `-(-(5))`. To maintain forward compatibility, we require you to lex `--` as a single token to disallow such expressions. For this reason, the `--` token appears in the `<reserved>` syntax category 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.[1] 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 function 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.

[1] 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.

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 to a value, 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 that the expression in the program's `return` statement evaluates to.

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 outside the range of integers represented by 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 the following:

- A set of *10* test cases written in L1. These test cases should follow the specification described in the Test Program section. Keep in mind that the purpose of a test case is to find bugs in your compiler, so you should try to write test cases that cover different cases of the language. Be creative, but also be considerate, as your test cases will be used as grading criteria for your peers (so for example you should not write test cases that run super slow unless sophisticated optimizations are implemented). You can read existing test cases that we provide you to find

some inspirations. Note that the deadline for test cases is *different* from the deadline for the compiler itself.

- A complete working compiler for L1 that produces correct target programs written in Intel x86-64 assembly language.
- Documentation: this 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 source code. 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.

Your test cases and compiler source files must be handed in via GitHub and Gradescope.

Test Program

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

`//test return i`	program must compile, execute without error, and return i
`//test div-by-zero`	program must compile but raise `SIGFPE` in its execution
`//test error`	program fails to comply with the L1 specification and must fail to compile

All test files should be in the directory

`tests`

in the root directory of the repository. This directory should contain no other files. You should follow the naming convention `<team name>-<file name>.l1` where `<team name>` is your team name in all lower case and `<file name>` is a descriptive name for the test case.

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 do not need to edit (but which you may 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 the input is not a valid L1 source program, your compiler should exit with a non-zero return code. If the input is valid, the compiler should then exit with a return code of 0 and output the generated code. The file `c0c-spec.txt` gives the full expected behavior of this binary.

The command

```
% make clean
```

should remove all binaries, heap images, 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-saved registers so that our main function can work correctly.

Note for Mac users: When compiling on Mac, the main function must be called `__c0_main` (with 2 underscores). The grading environment exports an environment variable `UNAME` that is set to the result of running `uname` on the grading machine; on the autograding machines, its value is `Linux`. You will likely make use of this platform-detection mechanism if you wish to develop on a Mac.

Using the GitHub Repository

Once you have completed the team registration form, you should receive access to the starter code and your team's repository (with your team name) on GitHub.

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

```
% git clone https://github.com/15-411-s24/dist.git
% cd dist
% git clone https://github.com/15-411-s24/<teamrepo>.git compiler
```

The first repository (*dist.git*) is used to distribute starter code, tools, and tests. Throughout the course, we will push more test cases to this repo.

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*.

Please avoid committing compiled binaries to your team repository. We provide a basic `.gitignore` to this effect, but you may have to ignore more files.

The starter code for OCaml, Rust, and SML lives under the `starter` directory of *dist.git*. For example, if you wanted to copy the OCaml starter code to your team's repository, you would run:

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

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

6 Submission Instructions

Using Gradescope

This semester, Gradescope will serve as the central hub for you to submit your labs and view the autograder results.

Use the GitHub integration to submit labs on Gradescope. This allows you to choose the specific branch of your repository that you want to submit, so you don't necessarily have to have to your final submission in the main branch.

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.

You will submit:

Before Friday, January 26, 11:59 PM Ten (10) test cases, at least 2 of which successfully compute a result, at least 2 of which raise a runtime exception, and at least 2 of which cause a compile-time error. You will submit to the **Test 1** assessment on Gradescope. The directory `tests` should only contain your test files.

Before Friday, February 2, 11:59 PM The complete compiler. You will submit to the **Lab 1** assessment on Gradescope. The directory `compiler/lab1` should contain only the sources for your compiler. 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.

No deadline The *optional, but highly recommended* compiler checkpoint for implementing register allocation. See checkpoint handout for details.

The results of autograding can be viewed on Gradescope.

Autograded Scoring

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 submission with the highest grade will count.

Your score for each submission is computed as follows:

$$\text{subtotal} = 25 * \frac{\text{passed basic tests}}{\text{total basic tests}} + 65 * \frac{\text{passed large \& only tests}}{\text{total large \& only tests}}$$

$$\text{compiler} = \text{subtotal} - (1 \text{ point per compiler failure}) - (0.1 \text{ points per executable timeout})$$

Your total score for the lab is computed as follows:

$$\text{test cases} = 10 * \frac{\min(\text{valid test cases}, 10)}{10}$$

$$\text{total} = \text{test cases} + \text{compiler}$$

7 Notes and Hints

We recommend reading lecture material on instruction selection and register allocation, referring to 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 to avoid inefficient target programs. The recommended algorithm is based on 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 essential that your target code strictly adhere to the calling conventions of the x86-64 architecture. Failing to do so could result in weird, possibly nondeterministic errors—and, more troublingly, these errors may only manifest in later labs. (Function calls, introduced in L3, bring many bugs out of the woodwork.)

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. We will post more detailed information about the grading infrastructure on Piazza, including how you can replicate it yourself.

Development Guidelines

- Format your code to a reasonable maximum line width.
- 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 and GitHub to your advantage—source control facilitates keeping track of the history of your codebase, and if you're familiar with them, GitHub pull requests provide a standard way for performing code reviews.
- 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.
- Think carefully 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 make future development easier because of the inherent difficultly in anticipating what might be needed. Instead, they may complicate or obfuscate the present code. 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.

8 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 OCaml, Rust, and SML.

Students are free to use other programming languages, but they must contact the course staff so that we can ensure the grading infrastructure is ready. For many languages, we can provide incomplete starter code for reference.

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 or less expressive than OCaml. Sources of problems may include 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.