15-411 Compiler Design: Lab 1
Fall 2010

Instructor: Andre Platzer
TAs: Anand Subramanian and Nathan Snyder

Test Programs Due: 11:59pm, Tuesday, September 7, 2010
Compilers Due: 11:59pm, Tuesday, September 14, 2010

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. [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 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. [2] Each team (or individual) is assigned a group name by the instructor. If you do not yet have a group name, please contact the instructor as soon as possible. You will not be able to download the starter code or hand in labs without accounts for the `svn` repository and the Autolab grading system, so you must have a group name.

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.

[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.

[2]There are certain circumstances under which working individually may be impractical. Once again, please consult the last section

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, tab (`\t`), 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). 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.

Reserved Keywords

The following are reserved keywords 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. For the purposes of *L1*, `main` shall also be treated as a keyword, since the language has no other concept of functions or function names.

Other Tokens

ł1 also treats 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. The tokens are listed in 3.

Note that `postop` has higher precedence than `unop`. Therefore, we should first attempt to lex consecutive occurrences of `-` as the postfix decrement.

$$
\begin{array}{lll}
\langle\text{program}\rangle & ::= & \textbf{int main () \{ } \langle\text{decls}\rangle\ \langle\text{stmts}\rangle\ \textbf{\}} \\
\langle\text{decls}\rangle & ::= & \\
 & \mid & \langle\text{decl}\rangle\ \langle\text{decls}\rangle \\
\langle\text{decl}\rangle & ::= & \textbf{int ident ;} \\
\langle\text{stmts}\rangle & ::= & \\
 & \mid & \langle\text{stmt}\rangle\ \langle\text{stmts}\rangle \\
\langle\text{stmt}\rangle & ::= & \langle\text{simp}\rangle\ \textbf{;} \\
 & \mid & \textbf{return}\ \langle\text{exp}\rangle\ \textbf{;} \\
\langle\text{simp}\rangle & ::= & \textbf{ident}\ \langle\text{asnop}\rangle\ \langle\text{exp}\rangle \\
\langle\text{exp}\rangle & ::= & \textbf{(}\ \langle\text{exp}\rangle\ \textbf{)} \\
 & \mid & \langle\text{intconst}\rangle \\
 & \mid & \textbf{ident} \\
 & \mid & \langle\text{exp}\rangle\ \langle\text{binop}\rangle\ \langle\text{exp}\rangle \\
 & \mid & \texttt{-}\ \langle\text{exp}\rangle \\
\langle\text{intconst}\rangle & ::= & \textbf{num} \qquad\qquad (\text{in the region } 0 \leq \text{intconst} < 2^{32}) \\
\langle\text{asop}\rangle & ::= & \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \\
\langle\text{binop}\rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}
\end{array}
$$

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>       ::= =  |  +=  |  -=  |  *=  |  /=  |  %=  |  <<=  |  >>=
<postop>      ::= --  |  ++
```

Figure 3: Lexical Tokens

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 used in the function to be declared at the top of the function with the correct type (in this case `int`). We do this to ensure that the valid *L1* programs are forward compatible with respect to future labs, and C0.

Initialization Checking

Programs that attempt to reference a variable before assigning to it should cause the compiler to generate a compile-time error message.

Return Checking

C0 requires that every control flow path in a function should be able to return. To maintain forward compatibility, we require that L1 programs contain a return statement, but not necessarily only once or at 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.

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. In addition to the code generator and the register allocator, you are also expected to modify the provided lexer so that it takes the reserved keywords and other lexical tokens into account.

We also require that you document your code. Documentation includes both inline documentation, and a README document which explains the design decisions underlying the implementation, and the general layout of the sources. If you use publicly available libraries or source, you are required to indicate their use and source in the README file.

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` *n*	program must compile but raise runtime exception *n*
`#test error`	program must fail to compile due to an *L1* source error

followed by the program text. Defined exceptions for this lab is only `SIGFPE` (8), which raised by division by 0 or division overflow. All test files should be collected into a directory `test/` (containing no other files) and submitted via the Autolab server.

The reference compiler may display a warning on `#test error` if your test case accidentally exercises a language feature that might be a part of a future lab or C0. Please do not ignore this warning. Do not hand in tests that cause this warning.

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. Even though your code will not be read for grading purposes, we may still read it to provide you feedback. The `README` file will be crucial information for that purpose.

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.

Using the Subversion Repository

The recommended method for handout and handin is the course subversion repository.

The handout files for this course can be checked out from our subversion repository via

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<team>
```

where `<team>` is the name of your team. You will find materials for this lab in the `lab1` subdirectory. Or, if you have checked out `15411-f10/<team>` directory before, you can issue the command `svn update` in that directory.

After first adding (with `svn add` or `svn copy` from a previous lab) and committing your handin directory (with `svn commit`) to the repository you can hand in your tests or compiler by selecting

```
S3 - Autograde your code in svn repository
```

from the Autolab server menu. It will perform one of

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<team>/lab1/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<team>/lab1/compiler
```

to obtain the files directories to autograde, depending on whether you are handing in your test files or your compiler.

If you are submitting multiple versions, please remember to commit your changes to the repository before asking the Autolab server to grade them! And please do not include any compiled files or binaries in the repository!

Uploading tar Archives

For emergency purposes, we may continue to support a deprecated method for handout and handin: the download and upload of tar archives from the Autolab server. Please refrain from using this option as much as possible.

For the test cases, bundle the directory `tests` as a tar file `tests.tar` with

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

to be submitted via the Autolab server.

For the compiler, bundle the directory `compiler` as a tar file `compiler.tar`. 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. For example:

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

to be submitted via the Autolab server. Please do not include any compiled files or binaries in your hand-in file!

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 Autolab grader agrees with the driver results that you use for development, and also as insurance against a last-minute rush. The last hand-in will count.

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. The directory `tests/` should only contain your test files and be submitted via subversion as described above. 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 7**.

- The complete compiler. The directory `compiler/` should contain only the sources for your compiler and be submitted via subversion as described above. 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 14**.

6 Notes and Hints

Much can be learned from studying the reference implementation. In addition, you should read the 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 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.

Development Guidelines

- Format your code to a line width of no more than 100 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.
- 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!
- 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, which 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. Continuing the tradition of the course, we distribute starter code in Standard ML and Java, both of which form a significant part of the undergraduate curriculum at Carnegie Mellon. We strongly recommend that students who are well versed in Standard ML 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 and SML.

In the Fall 2010 version of this course, we have also added brand new support for two more languages: Haskell and Scala. The starter code for these languages don't use parser generators or lexer generators. Feel free to swap in a parser generated by Happy (for haskell) or Scala-bison (for scala).

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.

If you wish to use a programming language that is significantly more verbose than Standard ML, we STRONGLY recommend that you find a partner in order to avoid being overwhelmed by the sheer volume of code. 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.

SML Style Guide

For Standard ML, which we expect to be the most commonly used language, we recommend the following stylistic guidelines regarding the use of its module system. Improper use may easily lead too an explosion of boilerplate, or make the code difficult to read..

- 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. Control the information that you wish to leak using `where` declarations.
- Use functors only when you instantiate them more than once.
- 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.`'.