15-411 Compiler Design: Lab 2
Fall 2009

Instructor: Frank Pfenning
TAs: Ruy Ley-Wild and Miguel Silva

Test Programs Due: 11:59pm, Tuesday, September 22, 2009
Compilers Due: 11:59pm, Tuesday, September 29, 2009

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L2*. This language extends *L1* by conditionals, loops, and some additional operators. This means you will have to change all phases of the compiler from the first lab. One can write some interesting iterative programs over integers in this language. Correctness is still paramount, but performance starts to become a minor issue because the code you generate will be executed on a set of test cases with preset time limits. These limits are set so that a correct and straightforward compiler without optimizations should receive full credit.

2 Requirements

As for Lab 1, you are required to hand in test programs as well as a complete working compiler that translates *L2* source programs into correct target programs written in x86-64 assembly language. When encountering an error in the input program (which can be a lexical, grammatical, or static semantics error) the compiler should terminate with a non-zero exit code and print a helpful error message. To test the target programs, we will assemble and link them using `gcc` on the lab machines and run them under fixed but generous time limits.

3 *L2* Syntax

The syntax of *L2* is defined by the context-free grammar in Figure 1. Ambiguities in this grammar are resolved according to the operator precedence table in Figure 2 and the rule that an `else` provides the alternative for the most recent eligible `if`.

Whitespace

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

⟨program⟩ ::= **{** ⟨stmt⟩* **}**

⟨stmt⟩ ::= ⟨simp⟩ **;** | ⟨control⟩ | **;**

⟨simp⟩ ::= ⟨ident⟩ ⟨asop⟩ ⟨exp⟩

⟨control⟩ ::= **if (** ⟨exp⟩ **)** ⟨block⟩ [**else** ⟨block⟩] |

while (⟨exp⟩ **)** ⟨block⟩ | **for (** [⟨simp⟩] **;** ⟨exp⟩ **;** [⟨simp⟩] **)** ⟨block⟩ |

continue ; | **break ;** | **return** ⟨exp⟩ **;**

⟨block⟩ ::= ⟨stmt⟩ | **{** ⟨stmt⟩* **}**

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

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

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

⟨asop⟩ ::= `=` | `+=` | `-=` | `*=` | `/=` | `%=` | `&=` | `^=` | `|=` | `<<=` | `>>=`

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

| `&&` | `||` | `&` | `^` | `|` | `<<` | `>>`

⟨unop⟩ ::= `!` | `~` | `-`

The precedence of unary and binary operators is given in Figure 2.
Non-terminals are in ⟨angle brackets⟩, optional constituents in [brackets].
Terminals are in **bold**.

Figure 1: Grammar of *L2*

Operator	Associates	Meaning
`()`	n/a	explicit parentheses
`! ~ -`	right	logical not, bitwise not, unary minus
`* / %`	left	integer times, divide, modulo
`+ -`	left	integer plus, minus
`<< >>`	left	(arithmetic) shift left, right
`< <= > >=`	left	integer comparison
`== !=`	left	integer equality, disequality
`&`	left	bitwise and
`^`	left	bitwise exclusive or
`|`	left	bitwise or
`&&`	left	logical and
`||`	left	logical or
`= += -= *= /= %= &= ^= |= <<= >>=`	right	assignment operators

Figure 2: Precedence of operators, from highest to lowest

Comments

L2 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.

4 *L2* Static Semantics

Reserved words of the grammar (`if`, `else`, `while`, `for`, `continue`, `break`, `return`) cannot be used as variable names.

Regarding control flow, several properties must be checked.

- Each (finite) control flow path through the program must terminate with an explicit `return` statement. This ensures that the program does not terminate with an undefined value.
- Each `break` or `continue` statement must occur inside a `while` or `for` loop.

Regarding variables, we need to check one property.

- On each control flow path through the program, each variable must be defined by an assignment before it is used. This ensures that there will be no references to uninitialized variables.

In order to rigorously define the two static checks related to control flow, we postulate abstract syntax trees for statements s according to

$$s ::= \text{assign}(x, e) \mid \text{if}(e, s, s) \mid \text{while}(e, s) \mid \text{continue} \mid \text{break} \mid \text{return}(e) \mid \text{nop} \mid \text{seq(s,s)}$$

where e stands for an expression and x for an identifier. Do not be confused by the fact that this looks like a grammar: the terms on the right hand side describe trees, not strings. Your implementation may of course differ from this—we use this merely as a means of specifying when programs are well-formed. We assume here that for loops have been rewritten to equivalent while loops according to the transformation described in the comments on the dynamic semantics below, and similarly for compound assignments such as `+=`.

The whole program is represented here as a single statement s, because a sequence of statements $\{s_1; s_2; \ldots\}$ is represented as a single statement $\text{seq}(s_1, \text{seq}(s_2, \ldots))$. In an implementation it may be more convenient to use lists explicitly.

Checking Proper Returns

We check that all finite control flow path through a program end with an explicit `return` statement. We say that s *returns* if executing s, if it terminates, will always end with a return statement. Overall, we want to ensure that the whole program, represented as a single statement s, returns according to this definition. If not, the compiler must signal an error.

$\text{assign}(x, e)$	does not return
$\text{if}(e, s_1, s_2)$	returns if both s_1 and s_2 return
$\text{while}(e, s)$	does not return
$\text{return}(e)$	returns
nop	does not return
$\text{seq}(s_1, s_2)$	returns if either s_1 returns (and therefore s_2 is dead code) or s_2 returns

We do not look inside loops (even though the bodies may contain `return` statements) because the body might not be executed at all. Because we do not look inside loops, we do not need rules for `break` or `continue`.

Checking Variable Initialization

We check that along all control flow paths, any variable is defined before use. First, we specify when a statement s *defines* a variable x. We read this as: Whenever s finishes normally, it will have defined x. This excludes cases where s returns, executes a `break` or `continue` statement, or does not terminate.

$\text{assign}(x, e)$	defines only x
$\text{if}(e, s_1, s_2)$	defines x if both s_1 and s_2 define x
$\text{while}(e, s)$	defines no x (because the body may not be executed)
break	defines all x (because it does not finish normally)
continue	defines all x (because it does not finish normally)
$\text{return}(e)$	defines all x (because it does not finish normally)
nop	defines no x
$\text{seq}(s_1, s_2)$	defines x if either s_1 or s_2 does

We also say that an expression e *uses* a variable x if x occurs in e. In our language, e may have logical operators which will not necessarily evaluate all their arguments, but we still say that a variable occurring in such an argument is used.

We now define which variables are *live* in a statement s, that is, their value may be used in the execution of s.

y is live in assign(x, e)	if y is used in e
y is live in if(e, s_1, s_2)	if y is live in s_1 or s_2
y is live in while(e, s)	if y is used in e or live in s
y is live in break	never (the jump target is accounted for elsewhere)
y is live in continue	never (the jump target is accounted for elsewhere)
y is live in return(e)	if y is used in e
y is live in nop	never
y is live in seq(s_1, s_2)	if y is live in s_1
	or y is live in s_2 and *not* defined in s_1

Static analysis should reject a program s if there is any variable y that is live in s.

This definition is based on the static control flow graph. For example, the program

```
{ return 1; x = y + 1; }
```

is valid because the statement `x = y + 1` can not be reached along any control flow path from the beginning of the program. Formally, the statement `return 1` is taken to define all variables, including `y`, so that `y` is not live in the whole program even though it is live in the second statement.

The restriction on `return` statements and initializing variables guarantee that the meaning of every valid program is deterministic: it will either return a unique value, raise a `div` exception, or fail to terminate.

5 *L2* Dynamic Semantics

In most cases, statements have the familiar operational semantics from C. Conditionals, `for`, and `while` loops execute as in C. `continue` skips the rest of the statements in a loop body and `break` jumps to the first statement after a loop. As in C, when encountering a `continue` inside a `for` loop, we jump to the *step* statement. Both `break` and `continue` always apply to the innermost loop they occur in.

We recommend expanding a `for` loop into a corresponding `while` loop. This expansion will streamline your compiler and also make static analysis much easier. Briefly,

$$\texttt{for}\ (s_{init}\,;\ e\,;\ s_{step})\ s_{body}$$

becomes

$$s_{init}\,;\ \texttt{while}\ (e)\ \{s'_{body}\,;\ s_{step}\}$$

where s'_{body} is the result of replacing any occurrence of `continue` in s_{body} referring to this loop by s_{step} ; `continue`.

The meanings of the special assignment operators are as in *L1*, where x *op*`=` e is the same as x `=` x *op* e.

Integer Operations

Since expressions do not have effects (except for a possible `div` exception that might be raised) the order of their evaluation is irrelevant.

The integers of this language are signed integers in two's complement representation with a word size of 32 bits. The semantics of the operations is given by modular arithmetic as in *L1*. Recall that division by zero and division overflow must raise a runtime division exception. This is the only runtime exception that should be possible in the language, except for those defined by the runtime environment such as stack overflow.

The left `<<` and right `>>` shift operations are arithmetic shifts. Since our numbers are signed, this means the right shift will copy the sign bit in the highest bit rather than filling with zero. Left shifts always fill the lowest bit with zero. Also, the shift quantity k will be masked to 5 bits and is interpreted positively so that a shift is always between 0 and 31 bits, inclusively. This is also the hardware behavior of the appropriate arithmetic shift instructions on the x86-64 architecture and is consistent with C where the behavior is underspecified.

The comparison operators `<`, `<=`, `>`, `>=`, `==`, and `!=` have their standard meaning on signed integers as in the definition of C.

Logical Operators

The operators `&&`, `||` and `!` are the so-called *logical operators*. They interpret an argument of 0 as false and non-zero as true, and always produce either 0 (for false) or 1 (for true). A tricky aspect of logical and (`&&`) and logical or (`||`) is they evaluate their arguments from left to right and short-circuit evaluation if the left-hand operand yields 0 (for logical and) and 1 (for logical or). In those cases, they do not evaluate their second operand and return 0 (for logical and) and non-zero (for logical or). Your implementation must model this semantics faithfully.

Runtime Environment

As in the first lab, your target code will be linked against a very simple runtime environment. It contains a function `main()` which calls a function `_l2_main()` that your assembly code should provide and export. This function should return a value in `%eax` or raise an exception, according to the language specification. It must also preserve all callee-save registers so that our main function can work correctly. Your compiler will be tested in the standard Linux environment on the lab machines; the produced assembly must conform to this environment.

6 Project Requirements

For this project, you are required to hand in test cases and a complete working compiler for *L2* 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.

Test Files

Test files should have extension `.l2` 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 a *L2* source error

followed by the program text. If the exception number n is omitted, any exception is accepted. All test files should be collected into a directory `test/` (containing no other files) and submitted via the Autolab server.

We would like some fraction of your test programs to compute "interesting" functions on specific values; please briefly describe such examples in a comment in the file. Disallowed are programs computing Fibonacci numbers, factorials, greatest common divisors, and minor variants thereof. Please use your imagination!

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` will be crucial information for this purpose.

Issuing the shell command

```
% make l2c
```

should generate the appropriate files so that

```
% bin/l2c <args>
```

will run your *L2* compiler. The command

```
% make clean
```

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

Using the Subversion Repository

Handin and handout of material is via 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-f09/<team>
```

where `<team>` is the name of your team. You will find materials for this lab in the `lab2` subdirectory. Or, if you have checked out `15411-f09/<team>` directory before, you can issue the command `svn update` in that directory.

After adding and committing your handin directory to the repository with `svn add` and `svn commit` 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-f09/<team>/lab2/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f09/<team>/lab2/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 an compiled files or binaries in the repository!

What to Turn In

Hand in on the Autolab server:

- At least 20 test cases, at least two of which generate as error and at least two others raise a runtime exception. 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. You may hand in as many times as you like before the deadline without penalty. If you feel the reference implementation is in error, please notify the instructors. The compiled binary for each test case should run in 2 seconds with the reference compiler on the lab machines; we will use a 5 second limit for testing compilers.

  Test cases are due **11:59pm on Tue Sep 22, 2009**.

- The complete compiler. The directory `compiler/` should contain only the sources for your compiler and be submitted via subversion. 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 (each with a 5 second time limit), and finally compare the actual with the expected results. You may hand in as many times as you like before the deadline without penalty.

  Compilers are due **11:59pm on Tue Sep 29, 2009**.

7 Notes and Hints

Static Checking

The specification of static checking should be implemented on abstract syntax trees, translating the rules into code. You should take some care to produce useful error messages.

It may be tempting to wait until liveness analysis on abstract assembly to see if any variables are live at the beginning of the program and signal an error then, rather than checking this directly on the abstract syntax tree. There are two reasons to avoid this: (1) it may be difficult or impossible to generate decent error messages, and (2) the intermediate representation might undergo some transformations (for example, optimizations, or transforming logical operators into conditionals) which make it difficult to be sure that the check strictly conforms to the given specification.

Calling Conventions

Your code must strictly adhere to the x86-64 calling conventions. For this lab, this just means that your `_l2_main` function must make sure to save and restore any callee-save registers it uses, and that the result must be returned in `%eax`.

Shift Operators

There are some tricky details on the machine instructions implementing the shift operators. The instructions `sall` k, D (shift arithmetic left long) and `sarl` k, D (shift arithmetic right long) take a shift value k and a destination operand D. The shift either has to be the `%cl` register, which consists of the lowest 8 bits of `%rcx`, or can be given as an immediate of at most 8 bits. In either case, only the low 5 bits affect the shift of a 32 bit value; the other bits are masked out. The assembler will fail if an immediate of more than 8 bits is provided as an argument.

Run-time Environment

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 -m64 -O2 -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 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. 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.