15-411 Compiler Design: Lab 2
Fall 2010

Instructor: Andre Platzer
TAs: Anand Subramanian and Nathan Snyder

Test Programs Due: 11:59pm, Tuesday, September 21, 2010
Compilers Due: 11:59pm, Tuesday, September 28, 2010

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 concrete syntax of *L2* is based on ASCII character encoding of source code.

3.1 Lexical Token

L2 source files are tokenized into the tokens listed in figure 1. The *L2*grammar does not necessarily accept all the tokens produced upon tokenizing source code. However, the additional tokens will remain in the lexical specification to maintain forward compatibility with future labs and C0.

ident	::=	`[A-Za-z_][A-Za-z0-9_]*`
num	::=	⟨*decnum*⟩ | ⟨*hexnum*⟩
⟨*decnum*⟩	::=	`0 | [1-9][0-9]*`
⟨*hexnum*⟩	::=	`0[xX][0-9a-fA-F]+`

⟨*special characters*⟩	::=	`! ~ - + * / % << >>`
		`< > == != & ^ | && ||`
		`= += -= *= /= %= <<= >>= &= |= ^=`
		`-> . -- ++ ( | ) [ ] ;`

⟨*reserved keywords*⟩	::=	`struct typedef if else while for continue break`
		`return assert true false NULL alloc alloc_array`
		`int bool void char string`

Terminals referenced in the grammar are in **bold**. Other classifiers not referenced within the grammar are in ⟨*angle brackets and in italics*⟩. **ident**, ⟨*decnum*⟩ and ⟨*hexnum*⟩ are described using regular expressions.

Figure 1: Lexical Tokens

Whitespace and Token Delimiting

In *L1*, whitespace is either a space, tab (`\t`), linefeed (`\n`), carriage return (`\r`) or formfeed (`\f`) character in ASCII encoding. Whitespace is ignored, except that it delimits tokens. Note that white space is not a REQUIREMENT to terminate a token. For instance, `()` should be tokenized into a left parenthesis followed by a right parenthesis according to the given lexical specification. However, white space delimiting can disambiguate two tokens when one of them is present as a prefix in the other. For example, `+=` is one token, while `+ =` is two tokens. The lexer should produce the longest valid token possible. For instance, `--` should be lexed as one token, not two.

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

`main` shall also be treated as a keyword in addition to those listed in figure 1, since the language has no other concept of functions or function names.

3.2 Grammar

The syntax of *L2* is defined by the context-free grammar in Figure 2. Ambiguities in this grammar are resolved according to the operator precedence table in Figure 3 and the rule that an `else` provides the alternative for the most recent eligible `if`.

⟨program⟩ ::= **int main ()** ⟨block⟩

⟨block⟩ ::= `{` ⟨decls⟩ ⟨stmts⟩ `}`

⟨type⟩ ::= **int** | **bool**

⟨decls⟩ ::= ϵ | ⟨decl⟩ ⟨decls⟩

⟨decl⟩ ::= ⟨type⟩ **ident ;** | ⟨type⟩ **ident =** ⟨exp⟩ **;**

⟨stmts⟩ ::= ϵ | ⟨stmt⟩ ⟨stmts⟩

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

⟨simp⟩ ::= **ident** ⟨asop⟩ ⟨exp⟩ | **ident** ⟨postop⟩

⟨simpopt⟩ ::= ϵ | ⟨simp⟩

⟨esleopt⟩ ::= ϵ | **else** ⟨stmt⟩

⟨control⟩ ::= **if (** ⟨exp⟩ **)** ⟨stmt⟩ ⟨elseopt⟩ | **while (** ⟨exp⟩ **)** ⟨stmt⟩

| **for (** ⟨simpopt⟩ **;** ⟨exp⟩ **;** ⟨simpopt⟩ **)** ⟨stmt⟩

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

⟨exp⟩ ::= **(** ⟨exp⟩ **)** | ⟨intconst⟩ | **true** | **false** | **ident**

| ⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ | ⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩

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

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

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

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

⟨unop⟩ ::= `!` | `~` | `-`

⟨postop⟩ ::= `++` | `--`

The precedence of unary and binary operators is given in Figure 3. Non-terminals are in ⟨angle brackets⟩. Terminals are in **bold**. The absence of tokens is denoted by ϵ.

Figure 2: Grammar of *L2*

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

Figure 3: Precedence of operators, from highest to lowest

4 *L2* Elaboration

As the name is intended to suggest, Elaboration is the process of elaborating the grammar that we have to one that is simpler and more well behaved – the abstract syntax. We describe elaboration separately because it is logically intended as a separate pass of compilation that happens immediate after parsing. Unfortunately, justifying the way we choose to elaborate a language may depend on an intuitive understanding of the operational behavior of the concrete language. Therefore, we recommend that you check what you see in this section against the description of the static and dynamic semantics of *L2* that appear in the next two sections.

Your implementation may of course employ a different elaboration strategy —but we will rely on the following elaboration strategy extensively in the description of the static semantics, and your implementation must behave in an equivalent manner. As always, document any design decisions you make.

We propose the following tree structure as the abstract syntax for statements s:

$$\begin{aligned} s ::= & \text{ assign}(x, e) \mid \text{if}(e, s, s) \mid \text{while}(e, s) \mid \text{for}(s, e, s, s) \\ \mid & \text{ continue} \mid \text{break} \mid \text{return}(e) \\ \mid & \text{ nop} \mid \text{seq}(s, s) \mid \text{declare}(x, t, s) \end{aligned}$$

where e stands for an expression, x for an identifier, and t for a type. Do not be confused by the fact that this looks like a grammar: the terms on the right hand side describe trees, not strings. 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.

Here are the suggested inference rules to elaborate blocks to trees:

$$\frac{}{\{\} \rightsquigarrow \text{nop}} \qquad \frac{\text{statement} \rightsquigarrow s \quad \{\text{remaining body}\} \rightsquigarrow s'}{\{\text{statement}; \text{remaining body}\} \rightsquigarrow \text{seq}(s, s')}$$

$$\frac{\{\text{remaining body}\} \rightsquigarrow s'}{\{\tau x;\ \text{remaining body}\} \rightsquigarrow \text{declare}(x, \tau, s')}$$

$$\frac{e \rightsquigarrow e' \quad \{\text{remaining body}\} \rightsquigarrow s'}{\{\tau x = e;\ \text{remaining body}\} \rightsquigarrow \text{declare}(x, \tau, \text{seq}(\text{assign}(x, e'), s'))}$$

Please check these elaboration rules against the scoping rules given in the static semantics.

As can be seen from the syntax, if statements always have both a "then" branch and an "else" branch represented by the two statements in that order. For loops have the "initializer" statement, the conditional expression, the "step" statment, and the loop body in that order. Elaborating if, else, while, for, break, continue and return should be fairly straightforward, and we do not give any rules for them.

As in *L1* assignment statements of the form `a op= b` are elaborated to be equivalent to `a = a op b`. In addition, `a++` is equivalent to `a = a + 1` and `a--` is equivalent to `a = a - 1`.

Unlike statements, expressions are already in a compact and well behaved representation. We only elaborate the logical operators. `a && b` elaborates to `a ? b : false`, and `a || b` elaborates to `a ? true : b`.

5 *L2* Static Semantics

5.1 Type Checking

Since our grammar and our type system have become a little more interesting, we now give some rules for type-checking. These rules are fairly informal. You may be interested in the material covered in 15-312 and 15-317 if you wish to understand the techniques used to formally treat type systems.

Type-checking Statements v. Type-checking Expressions

Our grammar allows expressions to appear with statements, but there is no way to embed statements within expressions. As a consequence it is meaningful to have a judgment of the form "$e : \tau$" to convey that an expression e has the type τ, but the same is meaningless when discussing statements. Therefore, for statements, we have the judgment "$s\,valid$".

Variable Declarations and Contexts

As in *L1* variables need to be declared with their types before they can be used. Unlike in *L1*, declaration can appear at the beginning of any block. The declaration of a variable is not visible outside the block of its declaration. Note that variables declared in inner blocks do *not* shadow variables declared in enclosing scopes. Therefore, multiple declarations of the same identifier may be present in the body of `main` if and only if no two of them are visible within the same block.

The abstract syntax for declarations, i.e. $\text{declare}(x, t, s)$ is very convenient for capturing all these phenomena and specifying the rules of type-checking. Here, the declaration of x of type t is visible only to statement s, and your elaborator should take care to elaborate statements while correctly preserving the lexical scope of declarations. Making the scope explicit in the abstract syntax makes it easy for you to build up a context of variables declarations available while type-checking any particular statement or expression.

We say "$\Gamma \vdash e : \tau$" to express that an expression e is type-checked under the context Γ which keeps track of all declarations of variables and their types. The following inference rules demonstrate the function of the context.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{x : \tau' \notin \Gamma \text{for any} \tau' \quad \Gamma, x : \tau \vdash s\ \mathit{valid}}{\Gamma \vdash \text{declare}(x, \tau, s)\ \mathit{valid}}$$

Here, x stands for any identifier.

The types

`int` is no longer the only type. We have *bool* which is inhabited by `true` and `false`. *L2* (and C0) do not allow implicit or explicit coercion between integral and boolean values. This is a major point of departure from C and other (in)famous languages.

Statements

We have already explained how declarations work. Here are the remaining significant rules.

$$\frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash s_1 \ \mathit{valid} \quad \Gamma \vdash s_2 \ \mathit{valid}}{\Gamma \vdash \textbf{if}(e, s_1, s_2) \ \mathit{valid}} \qquad \frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash s \ \mathit{valid}}{\Gamma \vdash \textbf{while}(e, s) \ \mathit{valid}}$$

$$\frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash s_1 \ \mathit{valid} \quad \Gamma \vdash s_2 \ \mathit{valid} \quad \Gamma \vdash s_3 \ \mathit{valid}}{\Gamma \vdash \textbf{for}(s_1, e, s_2, s_3) \ \mathit{valid}}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \textbf{assign}(x, e) \ \mathit{valid}} \qquad \frac{\Gamma \vdash s_1 \ \mathit{valid} \quad \Gamma \vdash s_2 \ \mathit{valid}}{\Gamma \vdash \textbf{seq}(s_1, s_2) \ \mathit{valid}} \qquad \frac{\Gamma \vdash e : \mathit{int}}{\Gamma \vdash \textbf{return}(e) \ \mathit{valid}}$$

The rule for return statements is still very rudimentary because we have only one function in the program, and it is required to return an int. The remaining statements always have a valid type structure.

Expressions

The following are the rules to check expressions for type correctness.

$$\frac{}{\Gamma \vdash \textbf{true} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \textbf{false} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \textbf{intconst} : \textbf{int}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int} \quad \text{relop} \in \{<, \leq, >, \geq\}}{\Gamma \vdash e_1 \text{ relop } e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \text{polyeq} \in \{==, !=\}}{\Gamma \vdash e_1 \text{ polyeq } e_2 : \textbf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \quad \Gamma \vdash e_2 : \textbf{bool} \quad \text{logop} \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1 \text{ logop } e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \textbf{!}\ e : \textbf{bool}}$$

Note that in a type theoretic sense, equality and disequality are *Overloaded Operators*, not polymorphic operators. See the dynamic semantics to see how the implementation is affected.

$$\frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 \text{ binop } e_2 : \textbf{int}} \qquad \frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash \text{unop } e : \textbf{int}}$$

Here, binop and unop are all the remaining binary and unary operators in the grammar not covered by the rules for booleans.

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (e_1 \ \textbf{?}\ e_2 \ \textbf{:}\ e_3) : \tau}$$

5.2 Control Flow

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.

We define these checks more rigorously on the abstract syntax as follows.

Checking Proper Returns

We check that all finite control flow paths through a program end with an explicit `return` statement. We say that s *returns* if, every execution of s that 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.

declare(x, t, s)	returns if s returns
assign(x, e)	does not return
if(e, s_1, s_2)	returns if both s_1 and s_2 return
while(e, s)	does not return
for(s_1, e, s_2, s_3)	does not return
return(e)	returns
nop	does not return
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, or might terminate by a break, not a return. Because we do not look inside loops, we do not need rules for `break` or `continue`.

Checking Variable Initialization

We wish to give a well formed deterministic dynamic semantics to *L1*. A program should either return a value, raise a divide-by-zero exception or fail to terminate. In order to do this, our static semantics must enforce the following necessary condition: we need to check that along all control flow paths, any variable is defined before use. Since the language allows the nesting of scopes of declaration, we define the variable initialization property as a local property of a scope which is actually stronger than the guarantee we need to make in order to give a well-formed dynamic semantics to the entire program.

To further streamline the specification, we transform For loops as follows:

$$\text{for}(\text{initializer}, e, \text{step}, \text{body}) \rightarrow \text{seq}(\text{initializer}, \text{while}(e, \text{body}_{\text{step}})$$

Here, $\text{body}_{\text{step}}$ is body with step inserted before every occurrence of continue that doesn't fall within the scope of an inner loop, and at the very end of the loop body. Check the dynamic semantics and make sure you understand why this transformation preserves those semantics. You may also find it an interesting exercise to consider what would go wrong if you were to include this transformation in the elaborator.

Your implementation may be more efficient if you convert while loops to for loops instead.

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.

declare(x, t, s)	defines no variable
assign(x, e)	defines only x
if(e, s_1, s_2)	defines x if both s_1 and s_2 define x
while(e, s)	defines no x (because the body may not be executed)
break	defines all x within scope (because it transfers control out of the scope)
continue	defines all x within scope (because it transfers control out of the scope)
return(e)	defines all x within scope (because it transfers control out of the scope)
nop	defines no x
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, because it might be.

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 declare(x, t, s)	if y is live in s and y is not the same as x
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 used in e or 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

Since scopes are encoded as declare statements, the given strategy has also told us what variables are live at the beginning of a scope, i.e. not initialized before its first use. Static analysis should reject a program if for any declare(x, t, s), the variable x is live in s.

The following example demonstrates how our static analysis is sufficient to guarantee deterministic evaluation:

```
{ int x; int y; 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 following example demonstrates how our static analysis is stronger than a sufficient condition to guarantee deterministic evaluation:

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

We can still give a well formed dynamic semantics for the program. However, static analysis will raise an error because the following *block* is not well formed:

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

Make sure that you have checked that all control flow paths return and all occurrences of `break` and `continue` are inside loops *before* checking for variable initialization. Otherwise, the effect that these statements have on liveness may produce very unhelpful and cryptic error messages for ill-formed programs.

6 *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. The suggested method of elaboration reflects this behavior. Both `break` and `continue` always apply to the innermost loop they occur in.

The ternary operator (`?:`), as in C, must only evaluate the branch that is actually taken. The suggested elaboration of the logical operators to the ternary operator also reflect their C-like short-circuit evaluation.

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.

To maintain interoperability with the ABI for C on linux, bools should be given an underlying 32 bit integer representation with false mapping to 0 and true mapping to 1. This runtime detail should in no way be visible in *L2* itself.

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. Operators `==` and `!=` are overloaded operators that test for the equality of either a pair of ints or a pair of bools. Fortunately, since the two types have the same underlying representation in our runtime, these operators only need to be implemented once each.

7 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-f10/<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-f10/<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-f10/<team>/lab2/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<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 an 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 21, 2010**.

- 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 28, 2010**.

8 Notes and Hints

Elaboration

Please take the recommended elaboration strategy seriously. It significantly streamlines your compiler and reducing the amount of work you do in each remaining pass of a multi-pass compiler. Isolating elaboration also makes your source code more portable.

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

Refer to *L1*.