15-411 Compiler Design, Lab 2 (Spring 2026)

Seth and co. 🪶

Test Cases Due: 11:59 PM, February 4, 2026
Compiler Due: 11:59 PM, February 14, 2026

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 L2 Syntax

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

2.1 Lexical Tokens

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.

Whitespace and Token Delimiting

In L2, whitespace is either a space, horizontal tab (`\t`), vertical tab (`\v`), linefeed (`\n`), carriage return (`\r`) or formfeed (`\f`) character in ASCII encoding. Whitespace is ignored, except that it delimits tokens. Note that whitespace 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.

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`

Tokens that are referenced as terminals in the grammar in Figure 2 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

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).

Reserved Keywords

The reserved keywords in Figure 1 cannot appear as a valid token in any place not explicitly mentioned in the grammar. In particular, they cannot be used as identifiers. Some of these keywords are unused in L2. However, they are treated as keywords to maintain forward compatibility with valid C0 programs.

2.2 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`.

If you don't explicitly address the `else` rule, you may end up with ambiguities in your grammar. See https://en.wikipedia.org/wiki/Dangling_else for an idea on how to address the `else` rule.

Hint (unrelated to the above): Consider how you are parsing the following: `(x);`

3 L2 Elaboration

As the name is intended to suggest, *elaboration* is the process of transforming the literal parse tree into one that is simpler and more well behaved – the abstract syntax tree. Much of this may be accomplished directly in the semantic actions that accompany the grammar rules, but sometimes

⟨program⟩ ::= **int main ()** ⟨block⟩

⟨block⟩ ::= `{` ⟨stmts⟩ `}`

⟨type⟩ ::= **int** | **bool**

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

⟨stmts⟩ ::= ϵ | ⟨stmt⟩ ⟨stmts⟩

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

⟨simp⟩ ::= ⟨lvalue⟩ ⟨asop⟩ ⟨exp⟩ | ⟨lvalue⟩ ⟨postop⟩ | ⟨decl⟩ | ⟨exp⟩

⟨simpopt⟩ ::= ϵ | ⟨simp⟩

⟨lvalue⟩ ::= **ident** | **(** ⟨lvalue⟩ **)**

⟨elseopt⟩ ::= ϵ | **else** ⟨stmt⟩

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

| **while (** ⟨exp⟩ **)** ⟨stmt⟩

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

| **return** ⟨exp⟩ **;**

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

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

⟨intconst⟩ ::= **num**

⟨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

a separate pass or multiple passes are advisable. We describe elaboration separately because it is logically intended as a separate pass of compilation that happens immediately 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.

Note that, with L2, some of these semantics may be implemented without a separate elaboration pass by parsing directly to the elaborated code. However, we recommend performing a separate elaboration pass, which can help with resolving ambiguity in future labs, which, like C, have languages that cannot be parsed entirely in a context-free way.

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:

$$\begin{aligned} s ::= &\ \mathsf{assign}(x, e) \mid \mathsf{if}(e, s, s) \mid \mathsf{while}(e, s) \mid \mathsf{return}(e) \\ \mid &\ \mathsf{nop} \mid \mathsf{seq}(s, s) \mid \mathsf{declare}(x, \tau, s) \end{aligned}$$

where e stands for an expression, x for an identifier, and τ 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 $\mathsf{seq}(s_1, \mathsf{seq}(s_2, \ldots,))$ (assuming s_1 and s_2 are not declarations). You may find it more convenient to use lists in your implementation.

Here are some suggested inference rules that describe how to elaborate sequences of statements ($\langle$stmts$\rangle$ in the grammar) into abstract syntax trees.

$$\frac{}{\epsilon \rightsquigarrow \mathsf{nop}} \qquad \frac{\langle\text{stmt}\rangle \rightsquigarrow s \quad \langle\text{stmts}\rangle \rightsquigarrow s'}{\langle\text{stmt}\rangle\ \langle\text{stmts}\rangle \rightsquigarrow \mathsf{seq}(s, s')}$$

$$\frac{\langle\text{type}\rangle \rightsquigarrow \tau \quad \textbf{ident} \rightsquigarrow x \quad \langle\text{stmts}\rangle \rightsquigarrow s'}{\langle\text{type}\rangle\ \textbf{ident}; \langle\text{stmts}\rangle \rightsquigarrow \mathsf{declare}(x, \tau, s')}$$

Here, the third rule should take precedence over the second so that scopes are resolved properly. Note that if variables are initialized when they are declared then one may be tempted to write the following elaboration rule:

$$\frac{\langle\text{type}\rangle \rightsquigarrow \tau \quad \textbf{ident} \rightsquigarrow x \quad \langle\text{exp}\rangle \rightsquigarrow e \quad \langle\text{stmts}\rangle \rightsquigarrow s}{\langle\text{type}\rangle\ \textbf{ident} = \langle\text{exp}\rangle; \langle\text{stmts}\rangle \rightsquigarrow \mathsf{declare}(x, \tau, \mathsf{seq}(\mathsf{assign}(x, e), s))}$$

But this form of elaboration is suspect since it appears to assert that variable x should have type τ in the expression e, while in L2 (and C0), a "recursive" declaration such as `int x = x+1` is not allowed. It is permissible for elaboration to perform simple checks (after parsing) and reject programs that should fail to compile.

As can be seen from the abstract syntax, conditionals always have both a "then" branch and an "else" branch represented by the two statements in that order. Elaborating **if**, **else**, **while**, and **return** should be fairly straightforward, and we do not give any rules for them.

It is probably sensible to elaborate **for** loops into **while** loops, just to simplify the transformations, code generation, etc. in the compiler. For example, we might elaborate with the following rule:

$$\frac{\langle \text{stmt}_1 \rangle \rightsquigarrow \mathit{init} \quad \langle \text{exp} \rangle \rightsquigarrow e \quad \langle \text{stmt}_2 \rangle \rightsquigarrow \mathit{step} \quad \langle \text{stmt}_3 \rangle \rightsquigarrow \mathit{body}}{\textbf{for}\ (\langle \text{stmt}_1 \rangle;\ \langle \text{exp} \rangle;\ \langle \text{stmt}_2 \rangle)\ \langle \text{stmt}_3 \rangle \rightsquigarrow \mathsf{seq}(\mathit{init}, \mathsf{while}(e, \mathsf{seq}(\mathit{body}, \mathit{step})))}$$

assuming that $\langle \text{stmt}_1 \rangle$ and $\langle \text{stmt}_2 \rangle$ are not declarations. If $\langle \text{stmt}_1 \rangle$ is a declaration, its elaboration should be similar to the one above but take scoping into account so that the scope of declaration *init* includes e, *body*, and *step* but nothing else. $\langle \text{stmt}_2 \rangle$ is not allowed to be a declaration and the compiler should issue an error message in that case.

As in L1, assignment statements of the form `a <binop>= b` are elaborated to be equivalent to `a = a <binop> 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 suggest elaborating the logical operators `a && b` to `a ? b : false`, and `a || b` to `a ? true : b`.

4 L2 Static Semantics

4.1 Type Checking

Since our grammar and our type system have become a bit 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. The declaration of a variable in a block is visible only in the subsequent statements in the same block. Variables declared in inner blocks are not allowed to shadow the same variable declared in an enclosing scope. 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, $\mathsf{declare}(x, \tau, s)$, is very convenient for capturing all these phenomena and specifying the rules of type-checking. Here, the declaration of x of type τ 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.

Some special rules apply for scoping of declarations in **for** statements as explain in Section 3.

We write $\Gamma \vdash e : \tau$ 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.

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

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 (like C0) does 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.

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

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

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.

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

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

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

Note that in a type theoretic sense, equality and disequality are *overloaded operators*, which is an example of so-called *ad hoc polymorphism* (in contrast with *parametric polymorphism*). See Section 5 to see how the implementation is affected.

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

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

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

4.2 Control Flow

Regarding control flow, several properties must be checked.

- Each (finite) control flow path through the program starting at the beginning of each function must terminate with an explicit `return` statement. This ensures that the program does not terminate with an undefined value.

Regarding variables, we need to the following.

- On each control flow path through the program connecting the use of a variable to its declaration, it must be defined by an assignment (or initialized) before it is used. This ensures that there will be no references to uninitialized variables.
- The step statement in a for loop may not be a declaration, as already discussed before.

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

Checking Proper Returns

We check that all finite control flow paths through the program starting at the beginning of each function, 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.

$\mathsf{declare}(x, \tau, s)$	returns if s returns
$\mathsf{assign}(x, e)$	does not return
$\mathsf{if}(e, s_1, s_2)$	returns if both s_1 and s_2 return
$\mathsf{while}(e, s)$	does not return
$\mathsf{return}(e)$	returns
nop	does not return
$\mathsf{seq}(s_1, s_2)$	returns if either s_1 returns (and therefore s_2 is unreachable 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.

Checking Variable Initialization

We wish to give a well-formed deterministic dynamic semantics to L2. A program should return a value, raise an arithmetic 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 (initialized) 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 calls **return** or does not terminate.

$\mathsf{declare}(x, \tau, s)$	defines y if s defines y and $y \neq x$
$\mathsf{assign}(x, e)$	defines only x
$\mathsf{if}(e, s_1, s_2)$	defines x if both s_1 and s_2 define x
$\mathsf{while}(e, s)$	defines no x (because the body may not be executed)
$\mathsf{return}(e)$	defines all x within scope (because it transfers control out of the scope)
nop	defines no x
$\mathsf{seq}(s_1, s_2)$	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 needed in the execution of s.

y is live in $\mathsf{declare}(x, \tau, s)$	if y is live in s and y is not the same as x
y is live in $\mathsf{assign}(x, e)$	if y is used in e
y is live in $\mathsf{if}(e, s_1, s_2)$	if y is used in e or live in s_1 or s_2
y is live in $\mathsf{while}(e, s)$	if y is used in e or live in s
y is live in $\mathsf{return}(e)$	if y is used in e
y is live in nop	never
y is live in $\mathsf{seq}(s_1, s_2)$	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, these rules also tell us which variables are live at the beginning of their scope, that is, not initialized before their first use. Static analysis should reject a program if for any $\mathsf{declare}(x, \tau, s)$, 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 `x = y + 1`.

The following example demonstrates how our static analysis is slightly more restricted than necessary on some programs.

```
{ 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. And, indeed, y is live at its declaration.

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

5 L2 Dynamic Semantics

In most cases, statements have the familiar operational semantics from C. Conditionals, **for**, and **while** loops execute as in C.

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

Integer Operations

Since expressions do not have effects (except for a possible arithmetic exception that might be raised) the order of their evaluation is irrelevant. However, in later labs, the evaluation of expressions can have distinguishable effects (such as non-termination), so it makes sense to stick to the order of evaluation specified by C0, which is left-to-right. For example, in $e_1 \otimes e_2$, where $\otimes$ is a binary operator, e_1 should be evaluated first, then e_2, then the operation.

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 arithmetic exception.

As in L1, decimal constants c in a program must be in the range $0 \leq c \leq 2^{31}$, where $2^{31} = -2^{31}$ according to arithmetic modulo 2^{32}. Hexadecimal constants must fit into 32 bits.

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 must be in the range $0 \leq k < 32$. The appropriate arithmetic shift instructions on the x86-64 architecture will instead mask the value to 5 bits to reduce it to this range, so **your compiler must arrange for an arithmetic exception to the raised at run time if the shift quantity is not in range**. In C, the behavior for shifts out of this range is *undefined* and therefore consistent with both L2 and machine code.

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 can have the same underlying representation, these operators only need to be implemented once each.

6 Project Requirements

For this project, you are required to hand in a complete working compiler for L2 that produces correct target programs written in Intel x86-64 assembly language (AT&T syntax).

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

When we grade your work, we will use the `gcc` compiler to assemble and link the code you generate into executables using the provided runtime environment on the lab machines.

Your compiler must be handed in as specified below.

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 div-by-zero`	program must compile but raise `SIGFPE`
`//test error`	program must fail to compile due to an L2 source error

followed by the program text. The only explicit exception defined in L2 is `SIGFPE` (8), which is raised upon division by zero or division overflow. Your compiled code will also need to raise this for a shift quantity that is out of range.

Compiler Source Files

The files comprising the compiler itself should be collected in a subdirectory of the `compiler` directory named `lab2`. The `compiler` directory contains a `Makefile`, which you may edit, but do not need to edit.

Issuing the shell command

```
% make lab2
```

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

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

will run your L2 compiler. If your compiler detects any (compile-time) errors in the source program, it should exit with a non-zero return code. If compilation succeeds and target code is generated, the compiler should then exit with a return code of 0. The command

```
% make clean
```

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

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

Your compiler is also expected to recognize a flag `-t` which, when present on the command line, stops the compiler immediately after typechecking and before the rest of the compiler runs. The exit code of your compiler should indicate success (0) if the code is well-formed, and failure (1) otherwise. If your compiler indicates success when run with `-t`, then it should be able to compile the file without further errors. Your compiler should also recognize the flags `-ex86-64` and `-O0`, but these flags can be ignored for now. These flags will be used for later assignments; they are explained in file `compiler/c0c-spec.txt`. However, for those who are interested, your compiler could be modified to recognize the `-O0` and `-O1` flags. These flags indicate the absence and presence of running the register allocator respectively. Choosing to recognize these flags represents the trade-off between compile time and run time performance of the input code.

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 `_c0_main()` that your assembly code should provide and export. If your compiler is given a well-formed input file `foo.l1` or `foo.l2` as a

command-line argument, it should generate a target file called `foo.l1.s` or `foo.l2.s` (respectively) in the same directory as the source file. The file `foo.l2.s` will be linked with the runtime into an executable using the command `gcc -m64 foo.l2.s ../runtime/run411.c`. According to the calling conventions, the register `%eax` must hold the return value, and your `_c0_main` function must preserve all callee-save registers so that our main function can work correctly.

What to Turn In

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

You will submit:

Before Wednesday, February 4th, 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, at least 2 of which cause a compile-time error. See the corresponding section above for how to format test case files when submitting to the **Test 2** assessment on Gradescope. The directory tests should only contain your test files. **Please name your test files with the extension `.l2`.**

Before Saturday, February 14th, 11:59 pm The complete compiler. You will submit to the **Lab 2** assessment on Gradescope. The directory `compiler/lab2` 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.

The results of the autograding can be viewed on Gradescope.

Autograded Scoring

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

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 adaptable.

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.

Representing boolean values

We suggest that booleans be given an underlying 32 bit integer representation with `false` mapping to `0` and `true` mapping to `1`. There might be occasion to reconsider the sizes of representations in future assignments, but it will not affect this lab, and giving all types a uniform four-byte representation simplifies your task for now.

Calling Conventions

Your code must strictly adhere to the x86-64 calling conventions. For this lab, this just means that your `_c0_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. Since the L2 semantics requires an exception to be raised if $k < 0$ or $k > 31$, you will need to insert a check before the shift at least in some cases.