15-411 Compiler Design, Spring 2023
Lab 3

Jan, Ethan, Isabel, James, and Thea! ♡

Compilers Due: 11:59 pm, Friday, March 3, 2023
Test Feedback Due: 11:59 pm, Monday, March 6, 2023

1 Introduction

The goal of the lab is to implement a complete compiler for the language L3. This language extends L2 with the ability to define functions and call them. This means you will have to change all phases of the compiler from the second lab. One can write some interesting recursive and iterative functions over integers in this language. Correctness is still paramount, but performance starts to become a bit more of an issue as we run larger and more interesting test cases.

L3 Global Declarations

In L2, we only had one top-level declaration (main), and our program consisted of a single function body. In L3, we allow for multiple top-level/global declarations (gdecls). These global declarations should be familiar from C0/C:

- fdecl: declares a function (i.e. a function prototype).

$$\langle \text{fdecl} \rangle ::= \langle \text{type} \rangle\ \textbf{ident}\ \langle \text{param-list} \rangle\ \textbf{;}$$

- fdefn: defines a function.

$$\langle \text{fdefn} \rangle ::= \langle \text{type} \rangle\ \textbf{ident}\ \langle \text{param-list} \rangle\ \langle \text{block} \rangle$$

- typedef: defines a transparent alias for a type, as in C0/C.

$$\langle \text{typedef} \rangle ::= \textbf{typedef}\ \langle \text{type} \rangle\ \textbf{ident}\ \textbf{;}$$

An L3 program thus consists of a list of gdecls.

L3 Header Files

Starting in L3 we would like to be able to link programs against a separate runtime (e.g. C0 standard library functions). But your compiler needs to know what external functions are available, and what their interface is. For this purpose we add the notion of "header files", much like in C0/C.

- L3 programs may be compiled with up to one *header file*
- Header files have the same syntax as program files
- Header files **CANNOT contain** function definitions, only function *declarations* and typedefs.
- Your compiler **cannot** assume that the header file parses and typechecks correctly.
- Your compiler can assume each function declared in the header will be available as an external symbol of the same name, when linking the program together.

The exact constraints on how global declarations interact with those in the L3 source file are complicated. See the Static Semantics section for details. Also, the x86 output needs to treat external functions (those declared in the header file) differently from functions declared/ defined in the L3 source file. See the Runtime section for details.

2 L3 Syntax

The lexical specification of L3 remains mostly unchanged from that of L2, with the addition of a comma (`,`) as a lexical token. The syntax of L3 is a superset of L2, as presented in Figure 1. A quick summary of the changes:

- New syntax rules for global declarations, including function declarations and typedefs.
- Function calls as expressions
- A new statement assert(e) with its meaning from C0 and C, aborting the program if e evaluates to false.
- The void syntactic construct, used in place of a return type for functions that do not return a value. [1]

Ambiguities in the 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`.

[1]Note: Some sources will refer to void as a type; this is rather inaccurate, since one cannot have values of "type" void. You may find it useful to think of functions as having an optional return type; either they return some type τ, or they do not have a return type (which we denote in the syntax as void). Using void anywhere other than in function return types should be considered a parse error.

⟨program⟩	::=	ϵ $\mid$ ⟨gdecl⟩ ⟨program⟩
⟨gdecl⟩	::=	⟨fdecl⟩ $\mid$ ⟨fdefn⟩ $\mid$ ⟨typedef⟩
⟨fdecl⟩	::=	⟨ret-type⟩ **ident** ⟨param-list⟩ **;**
⟨fdefn⟩	::=	⟨ret-type⟩ **ident** ⟨param-list⟩ ⟨block⟩
⟨param⟩	::=	⟨type⟩ **ident**
⟨param-list-follow⟩	::=	ϵ $\mid$ **,** ⟨param⟩ ⟨param-list-follow⟩
⟨param-list⟩	::=	**( )** $\mid$ **(** ⟨param⟩ ⟨param-list-follow⟩ **)**
⟨typedef⟩	::=	**typedef** ⟨type⟩ **ident ;**
⟨type⟩	::=	**int** $\mid$ **bool** $\mid$ **ident**
⟨ret-type⟩	::=	⟨type⟩ $\mid$ **void**
⟨block⟩	::=	**{** ⟨stmts⟩ **}**
⟨decl⟩	::=	⟨type⟩ **ident** $\mid$ ⟨type⟩ **ident =** ⟨exp⟩
⟨stmts⟩	::=	ϵ $\mid$ ⟨stmt⟩ ⟨stmts⟩
⟨stmt⟩	::=	⟨simp⟩ **;** $\mid$ ⟨control⟩ $\mid$ ⟨block⟩
⟨simp⟩	::=	⟨lvalue⟩ ⟨asop⟩ ⟨exp⟩ $\mid$ ⟨lvalue⟩ ⟨postop⟩ $\mid$ ⟨decl⟩ $\mid$ ⟨exp⟩
⟨simpopt⟩	::=	ϵ $\mid$ ⟨simp⟩
⟨lvalue⟩	::=	**ident** $\mid$ **(** ⟨lvalue⟩ **)**
⟨elseopt⟩	::=	ϵ $\mid$ **else** ⟨stmt⟩
⟨control⟩	::=	**if (** ⟨exp⟩ **)** ⟨stmt⟩ ⟨elseopt⟩
	$\mid$	**while (** ⟨exp⟩ **)** ⟨stmt⟩
	$\mid$	**for (** ⟨simpopt⟩ **;** ⟨exp⟩ **;** ⟨simpopt⟩ **)** ⟨stmt⟩
	$\mid$	**return** ⟨exp⟩ **;** $\mid$ **return ;**
	$\mid$	**assert (** ⟨exp⟩ **) ;**
⟨arg-list-follow⟩	::=	ϵ $\mid$ **,** ⟨exp⟩ ⟨arg-list-follow⟩
⟨arg-list⟩	::=	**( )** $\mid$ **(** ⟨exp⟩ ⟨arg-list-follow⟩ **)**
⟨exp⟩	::=	**(** ⟨exp⟩ **)** $\mid$ **num** $\mid$ **true** $\mid$ **false** $\mid$ **ident**
	$\mid$	⟨unop⟩ ⟨exp⟩ $\mid$ ⟨exp⟩ ⟨binop⟩ ⟨exp⟩
	$\mid$	⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩ $\mid$ **ident** ⟨arg-list⟩
⟨asop⟩	::=	`=` $\mid$ `+=` $\mid$ `-=` $\mid$ `*=` $\mid$ `/=` $\mid$ `%=` $\mid$ `&=` $\mid$ `^=` $\mid$ `|=` $\mid$ `<<=` $\mid$ `>>=`
⟨binop⟩	::=	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `<` $\mid$ `<=` $\mid$ `>` $\mid$ `>=` $\mid$ `==` $\mid$ `!=`
	$\mid$	`&&` $\mid$ `||` $\mid$ `&` $\mid$ `^` $\mid$ `|` $\mid$ `<<` $\mid$ `>>`
⟨unop⟩	::=	`!` $\mid$ `~` $\mid$ `-`
⟨postop⟩	::=	`++` $\mid$ `--`

The precedence of unary and binary operators is given in Figure 2. Non-terminals are in ⟨angle brackets⟩. Terminals are in **bold**. The absence of tokens is denoted by ϵ.

Figure 1: Grammar of L3

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 2: Precedence of operators, from highest to lowest

3 L3 Statics

The static semantics of L3 is substantially more complex than that of L2. Much of the non-uniform behavior of C has been replicated in L3, especially with respect to name collisions and shadowing. Design your implementation carefully, to ensure it matches the (quite complex) behavior described in this section.

Function Signatures

A *function signature* summarizes the argument types of a function, as well as the return type, if the function has one.

$$(\tau_1, \tau_2, \ldots, \tau_n) \to [\tau]$$

In the void return case, we write $(\tau_1, \ldots, \tau_n) \to \cdot$ to indicate the lack of a return type.

Declaration Context Δ

The context Δ (*de*lta, for *de*finitions and *de*clarations) will be used to track function declarations and definitions. We have a judgment that f has a given signature in Δ, written:

$$\Delta \vdash f : (\tau_1, \ldots, \tau_n) \to \tau$$

We also have a judgment that f is defined in Δ, written:

$$\Delta \vdash f \ \mathsf{defined}$$

We will write $\Delta, f : (\tau_1, \ldots, \tau_n) \to \tau$ to add a signature to the context, and $\Delta, f\ \mathsf{defined}$ to mark that f is defined. There are some restrictions on this, detailed in the relevant sections below.

Typedef Context Ω

During typechecking we must keep track of which identifiers have been typedef'd, and what their definition is.

For this we introduce Ω (*om*ega for *om*... nevermind, this one makes no sense). Ω is a mapping from typedef'd symbols to their definitions. We have a judgment that, in context Ω a type t "resolves" to τ, where $\tau \in \{\mathbf{int}, \mathbf{bool}\}$:

$$\Omega \vdash t \rightsquigarrow \tau$$

If t is **int** or **bool**, then $t = \tau$, while if t is an identifier, we look it up in Ω:

$$\frac{\tau \in \{\text{int, bool}\}}{\Omega \vdash \tau \rightsquigarrow \tau} \qquad\qquad \frac{\Omega(t) = \tau}{\Omega \vdash t \rightsquigarrow \tau}$$

Note that typedefs can typedef a symbol to another typedef'd symbol, but for simplicity we will say that Ω should map symbols to their fully resolved type (i.e. **int** or **bool**). Assuming t is a symbol and τ is a resolved type, we will write $\Omega, t \mapsto \tau$ for the context with updated mapping.

Program Typechecking

An L3 program consists of a list of gdecls from a header file, and a list of gdecls from an L3 source file. Declarations are processed in the file order, from top down, with header declarations processed before program declarations. Note that you cannot assume the header typechecks; its declarations are subject to the same restrictions as program declarations, with the additional restriction that function *definitions* are not allowed in headers.

A global declaration g typechecks in context $\Delta; \Omega$ iff the following judgment is derivable:

$$\Delta;\ \Omega\ \vdash\ g\ \ \rightsquigarrow\ \ \Delta';\ \Omega'$$

The $\Delta'; \Omega'$ represent the updated contexts, capturing that each gdecl adds information to the context.

Notes:

- The `main()` function is considered implicitly declared with its expected function signature BEFORE the declarations in the L3 source, and can therefore be forward-referenced in source programs.
- Header functions are considered defined before the L3 source.

Typedefs

Typedefs are relatively straightforward; if valid, the typedef adds a mapping to Ω.

$$\frac{t' \notin \mathsf{Dom}(\Delta) \qquad t' \notin Dom(\Omega) \qquad \Omega \vdash t \rightsquigarrow \tau}{\Delta;\ \Omega\ \vdash\ \textbf{typedef}\, t\ t'\ \ \rightsquigarrow\ \ \Delta;\ \Omega, t' \mapsto \tau}$$

The premises require that the symbol t' is not already declared as a typedef NOR as a function.

Once an identifier is defined as a typedef, **ALL** later identifiers in the program (function names, parameter/local variable names, other typedefs, . . .) cannot collide with the typedef'd identifier. We will reiterate this in all relevant rules below.

Function Declaration

Function declarations insert a new declaration to Δ. They have somewhat complex typing constraints:

- The name must not be a typedef
- Parameter names must not be typedefs
- Parameter list should have no duplicate parameter names
- If a function with the same name has been declared, the prior signature must match the new declaration (same argument and return types, though not necessarily the same parameter names).

This last rule is quite important – a program can re-declare any function, even those declared in the header, if the declaration signature matches the previous/existing signature for that function. The constraints are summarized by the following two rules:

$$\frac{f \notin \mathsf{Dom}(\Delta) \qquad f \notin \mathsf{Dom}(\Omega) \qquad [\Omega \vdash t \rightsquigarrow \tau] \qquad \Omega \vdash t_i \rightsquigarrow \tau_i}{\Delta;\ \Omega\ \vdash\ [t]\ \ f(t_1\ x_1,\ \ldots,\ t_n\ x_n);\ \ \rightsquigarrow\ \ \Delta, f : (\tau_1, \ldots, \tau_n) \to [\tau];\ \Omega}$$

$$\frac{\Delta \vdash f : (\tau_1, \ldots, \tau_n) \to [\tau] \qquad [\Omega \vdash t \rightsquigarrow \tau] \qquad \Omega \vdash t_i \rightsquigarrow \tau_i}{\Delta;\ \Omega\ \vdash\ [t]\ \ f(t_1\ x_1,\ \ldots,\ t_n\ x_n);\ \ \rightsquigarrow\ \ \Delta;\ \Omega}$$

The square brackets are intended to indicate that these rules apply for functions with no return type; the premises in square brackets simply are ignored in the case of no return type.

Function Definition

Function definitions simultaneously declare and define a function. The declaration derived from a definition should typecheck with the same restrictions as above. In particular, if a function is already declared, the function signature must match the prior declaration.

A definition also (as the name suggests) defines the function. The function must not have been previously defined, and the function body is a block which must typecheck as a statement returning the expected return type:

$$\frac{\begin{array}{c}\Delta;\ \Omega\ \vdash\ [t]\ \ f(t_1\ x_1,\ \ldots,\ t_n\ x_n);\ \ \rightsquigarrow\ \ \Delta';\ \Omega \qquad \Delta \vdash \neg(f\ \text{defined}) \\ [\Omega \vdash t \rightsquigarrow \tau] \qquad \Omega \vdash t_i \rightsquigarrow \tau_i \qquad \Delta';\ \Omega;\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \vdash\ s : [\tau]\end{array}}{\Delta;\ \Omega\ \vdash\ [t]\ \ f(t_1\ x_1,\ \ldots,\ t_n\ x_n)\ \ s\ \ \rightsquigarrow\ \ \Delta', f\ \text{defined};\ \Omega}$$

Note the subtleties in the last premise: f is considered declared in the body s, thus the updated context Δ' is considered when typechecking s. Also, the initial typing context Γ consists of the parameters of f. As with the declaration rules, this rule still applies when f has no return type, with one fewer premise to consider.

In L3, a function is visible after its first declaration, and within its definition. This means a single recursive function does not require a declaration, but two mutually recursive functions require at least one forward declaration.

Statements & Expressions

The typechecking for statements and expressions is largely the same as in L2. As we will see, the statement judgment needs to be augmented with the Δ, Ω contexts from the global declarations, and the expression judgment needs to be augmented with the Δ context.

First, the rule for declarations is now different, because variables may not conflict with typedef'd symbols:

$$\frac{x \notin \mathrm{Dom}(\Omega) \qquad x \notin \mathrm{Dom}(\Gamma) \qquad \Delta;\ \Omega;\ \Gamma, x : \tau\ \vdash\ s : [\rho]}{\Delta;\ \Omega;\ \Gamma\ \vdash\ \mathsf{decl}(\tau, x, s) : [\rho]}$$

There is no restriction on local variables shadowing function names. Note that, in particular, since function parameters are in scope for the entire function, they cannot be shadowed by declarations.

We also have two (three?) new statements, namely non-**int** returns and asserts. Their rules:

$$\frac{}{\Delta;\ \Omega;\ \Gamma\ \vdash\ \mathsf{return;} : \cdot} \qquad \frac{\Delta;\ \Gamma\ \vdash\ e : \rho}{\Delta;\ \Omega;\ \Gamma\ \vdash\ \mathsf{return}(e) : \rho} \qquad \frac{\Delta;\ \Gamma\ \vdash\ e : \mathbf{bool}}{\Delta;\ \Omega;\ \Gamma\ \vdash\ \mathsf{assert}(e) : [\rho]}$$

Next, function calls are now a valid expression, so we need a typechecking rule for them. There is a somewhat odd constraint that the function symbol being called must not have been introduced as a variable in scope:

$$\frac{f \notin \mathrm{Dom}(\Gamma) \qquad \Delta \vdash f : (\tau_1, \ldots, \tau_n) \to \tau \qquad \Delta;\ \Gamma\ \vdash\ e_i : \tau_i}{\Delta;\ \Gamma\ \vdash\ f(e_1, \ldots, e_n) : \tau}$$

Note this rule is only applicable to functions with a return type. Void-returning functions cannot appear in arbitrary expressions. Instead, they can only appear as top-level statements. So, we will add a separate statement rule to allow this:

$$\frac{f \notin \mathrm{Dom}(\Gamma) \qquad \Delta \vdash f : (\tau_1, \ldots, \tau_n) \to \cdot \qquad \Delta;\ \Gamma\ \vdash\ e_i : \tau_i}{\Delta;\ \Omega;\ \Gamma\ \vdash\ f(e_1, \ldots, e_n) : [\rho]}$$

Defined & Used Functions

An important check for the safety of your program is ensuring that all functions *used* in the program are *defined* in the program:

- External functions (= at least one declaration in a header file) are considered defined before the start of the L3 source.
- A function is considered used if it is called ANYWHERE in the program (including unreachable/dead code).
- A function need not be defined at the point in code where it is used (see Statements & Expressions subheader), merely by the end of the program.
- `main()` is implicitly considered used in every program, and thus must always be defined (unlike L1/L2, the presence of `main()` is no longer a syntactic restriction but rather a typechecking restriction!)
- A function that is declared, but not used anywhere in the program, DOES NOT need to be defined. [2]

[2] This is largely due to the behavior of C, where the design of the language is supposed to allow separate compilation of individual sections of the program.

Return Checking

As in L2, the typechecker must ensure that functions with return types must return a value on every control flow path. However, functions with no return type (i.e. void functions) **do not** need to return on every path. They are assumed to implicitly return at the end of the function body.

Variable Initialization

As in L2, the typechecker must ensure that all local variables are defined before they are used. Function parameters are considered *defined* at the beginning of the function body (they will have values when the execution of the body of a function commences).

4 L3 Dynamic Semantics

The dynamic semantics of L3 directly extends the dynamic semantics from L2.

Function calls $f(e_1, \ldots, e_n)$ are very similar to their counterparts in C with the following significant difference: they must evaluate their arguments from left to right before passing the resulting values to f. The same is true for other operators that evaluate their arguments. For example, in $e_1 + e_2$, we must evaluate e_1 before e_2. Since L3 has several kinds of effects (arithmetic exception, nontermination, abort, and even output), this specification now becomes significant (when it was not observable in L2).

The new statement `assert`(e) first evaluates e. If the result is `true`, the assertion succeeds and we just continue with the next statement. If e is false, it raises the `SIGABRT` exception (6). In your code you can achieve this by calling the function `abort()` which takes no arguments.

Expressions may appear as statements, because we now have the concept of expressions with side-effects. These side-effects are quite simple in L3: arithmetic exceptions, program abort, nontermination, and output done by library functions. An error can also arise when a program runs out of stack space. Different systems handle this differently. The autograder runs on an Ubuntu system that seems to consistently raise `SIGBUS` (7) when it runs out of stack space, but other systems may raise `SIGSEGV` (11). You do not need to handle stack overflows specially.

5 Runtime Environment

Programs compiled with a header file are allowed to assume the declared symbols will be present in the runtime, and that the types declared in the header are accurate to the runtime implementation.

The GNU compiler and linker will be used to link your assembly to the implementations of the external functions. You should ensure that the code you generate adheres to the C ABI for Linux on x86-64. In order for the linking to work, you must adhere to the following conventions:

- External functions must be called as named in the header file.
- Non-external functions with name *name* must be called `_c0_`*name*. This ensures that non-external function names do not accidentally conflict with names from standard library which

could cause assembly or linking to fail.

NOTE: If a header file declares a function with name `_c0_...`, your compiler is allowed to crash after typechecking, since the program may not be compilable in a way that adheres to these conventions.

- Non-external functions must be exported from (declared to be *global* in) the assembly file you generate, so that our test harness can call them and verify your adherence to the calling conventions. Use the `.globl <label>` directive.

The runtime environment defines a function `main()` which calls a function `_c0_main()` your assembly code should provide and export. Your compiler will be tested in the standard Linux environment on the docker containers; the produced assembly must conform to this environment.

In order to satisfy the ABI of libraries that take or return the type `bool`, you must implement this type such that `false` maps to `0` and `true` maps to `1` when calling external functions, though you are free to implement your own representations internally if you wish. For the sake of uniformity, it will still suffice to treat boolean values as 32-bit integers; there may be occasion to reconsider this in Lab 4 or 5.

6 Command-Line Interface

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. You are encouraged to implement `-O0` and `-O1` flags, representing different tradeoffs between compilation speed and runtime performance of your compiled program. The autograder will invoke your compiler with whichever flag you have set as the default. You are additionally encouraged to perform analysis on the file you are compiling to determine whether your compiler can feasibly perform more expensive optimizations (such as register allocation) within the specified limit. We recommend falling back to faster, if less optimal, optimization strategies when necessary.

Your compiler should accept an optional command line argument `-l` which must be given the name of a file as an argument. For instance, we will be calling your compiler using the following command: `bin/c0c -l ../runtime/15411-l3.h0 $test.l3`. Here, `15411-l3.h0` is a header file.

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.

7 Testing

Test programs have extension `.l3` 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` (8)
`//test abort`	program must compile and run but raise `SIGABRT`(6)
`//test error`	program must fail to compile due to an L3 source error
`//test typecheck`	program must typecheck correctly
`//test compile`	program must typecheck, compile, and link

If the test program `$test.l3` is accompanied by a file `$test.h0` (same base name, but `h0` extension), then we compile the test treating `$test.h0` as the header file. Otherwise, we treat `../runtime/15411-l3.h0` as the header file for all `l3` tests, and we will pass that header file to your compiler with the `-l` argument. The `15411-l3.h0` header file describes a library for floating point arithmetic and printing operations. The implementation of this library can be found in `lab3/runtime/run411.c`. Our testing framework will ignore any output performed from the printing operations.

Tests that use a `.h0` header file might typecheck but fail to link because they refer to functions that aren't provided by the system: the header file can even describe a library that can't possibly be implemented (see `busy.l3` and `busy.h0` for an example).

Tests which use a `.h0` header file might also take advantage of functions provided in `libc` or `libgcc` (see `rand.l3` and `rand.h0` for an example). But many of these functions can cause the test cases to behave badly. Therefore, tests that have a custom header file *must* start with the line `//test error` or `//test typecheck`, and will not be executed by the autograding harness.

8 Submission

For this project, you are expected to hand in a complete working compiler for L3 that produces correct target programs written in Intel x86-64 assembly language.

Please note that any submission past **11:59 pm** on the due dates for either the tests or the compiler will result in the usage of late days. This policy will apply even if the late submission(s) has a lower grade than any previous submission before the deadline. However, we will still count your highest score across all submissions.

Compiler

You should submit the source code of your compiler to the **Lab 3** assessment on Gradescope. The directory `compiler/lab3` 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 files comprising the compiler itself should be collected in a directory `compiler/` which should contain a `Makefile`.

Issuing the shell command

```
% make lab3
```

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

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

will run your L3 compiler. The command

```
% make clean
```

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

Important: You should also update the `README` file to include

1. a description of your compiler's structure.
2. major design decisions or specific algorithms utilized.
3. any publicly available external libraries you used.

Even though we will not be grading you on your code quality, we will be reading your code for the code review (see below), and to provide you feedback. The `README` will be crucial for us to understand what is going on in your compiler.

You are permitted to make use of external, publicly available libraries in your compiler, where appropriate. If you are unsure whether it is appropriate to use an external library, please discuss it with course staff.

Autograded Scoring

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

Your score for each submission is computed as follows:

$$\text{subtotal} = 25 * \frac{\text{passed basic tests}}{\text{total basic tests}} + 45 * \frac{\text{passed large \& only tests}}{\text{total large \& only tests}}$$
$$\text{total} = \text{subtotal} - (\text{1 point per compiler failure}) - (\text{0.1 points per executable timeout})$$

Test Case Feedback

For this lab, you will be required to submit a file called `test-feedback.txt` to Gradescope with test case feedback. We will be using this feedback to trim down our test case suite and identify which tests are the most useful for debugging your compilers. You must choose **at least 20** distinct test cases from the L3 test suite and provide short feedback on each of them.

This feedback can be either "positive", meaning you would recommend the test be kept, or "negative", meaning you would recommend the test be removed. Since we are aiming to trim down the test suite, you should give negative feedback for **at least 10** of your chosen tests.

Each entry in the text file should be formatted as follows:

```
* <test-file-name> <positive/negative> <feedback>
```

For example:

```
* mytest.l3 positive caught a bug in our register allocator
* yourtest.l3 negative this test is exactly the
  same as thattest.l3
```

Note that you may include multiple lines in your feedback, as long as each entry starts at the beginning of a line with a `*`.

Code Review

We will be holding code review sessions between March 14-16. The purpose of the code review is to make sure that both partners understand the whole compiler, from the lexing and parsing all the way to x86-64 generation. We will ask questions pertaining to any part of the compiler, and both partners are expected to be able to answer them. The instructors will be looking through your code and READMEs before meeting with you, so make sure that the READMEs are updated to reflect the organization of your code.

We will post sign ups for slots on Piazza as soon as we finalize the schedules. Be sure to check Piazza frequently for updates.

9 Notes and Hints

Elaboration

We *highly* recommend using an explicit elaboration pass, or even multiple elaboration passes, rather than transforming source code on the fly during parsing. A well-designed elaboration pass can significantly streamline your compiler and reduce the amount of work it takes to implement later phases of your compiler.

Static Checking

Follow the rules for static analysis closely. You should take some care to produce useful error messages when typechecking; this phase is notoriously prone to small or subtle bugs in L3, and good error messages can be a life saver when debugging your typechecker.

Compiling Functions

It is not a strict requirement, but we recommend compiling functions completely independently from each other, taking care to respect the calling conventions and making no other assumptions. Interprocedural program analysis and optimization is difficult and, if you do it at all, is better left to a later lab.

Calling Conventions

Your code must strictly adhere to the x86-64 calling conventions. Please refer to the course webpage for resources on the ABI and calling conventions. A particular point of note: `%rsp` must be 16-byte aligned. GCC often ignores this rule when it isn't actively using floating point numbers, because the requirement is only consequential when the floating point stack is in use.