15-411 Compiler Design, Fall 2021
Lab 3

Seth and co.

Test Programs Due: Tuesday, October 12, 2021 at 11:59 PM
Compilers Due: Thursday, October 21, 2021 at 11:59 PM

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.

2 L3 Syntax

The lexical specification of L3 remains unchanged from that of L2, except that comma (`,`) is now a lexical token. The syntax of L3 is a superset of L2, as presented in Figure 1. Ambiguities in this grammar are resolved according to the same rules of precedence as in L2. The main extension is to allow definitions of functions and transparent definitions of types. We also have function calls as expressions, as well as a new statement assert(e) with its meaning from C0 and C, aborting the program if e evaluates to false. Finally, we have the new type void, used only as the return type for functions not intended to return a value.

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`.

⟨program⟩	::=	ϵ | ⟨gdecl⟩ ⟨program⟩
⟨gdecl⟩	::=	⟨fdecl⟩ | ⟨fdefn⟩ | ⟨typedef⟩
⟨fdecl⟩	::=	⟨type⟩ **ident** ⟨param-list⟩ **;**
⟨fdefn⟩	::=	⟨type⟩ **ident** ⟨param-list⟩ ⟨block⟩
⟨param⟩	::=	⟨type⟩ **ident**
⟨param-list-follow⟩	::=	ϵ | **,** ⟨param⟩ ⟨param-list-follow⟩
⟨param-list⟩	::=	**(** **)** | **(** ⟨param⟩ ⟨param-list-follow⟩ **)**
⟨typedef⟩	::=	**typedef** ⟨type⟩ **ident** **;**
⟨type⟩	::=	**int** | **bool** | **ident** | **void**
⟨block⟩	::=	`{` ⟨stmts⟩ `}`
⟨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⟩ **;** | **return** **;**
	|	**assert** **(** ⟨exp⟩ **)** **;**
⟨arg-list-follow⟩	::=	ϵ | **,** ⟨exp⟩ ⟨arg-list-follow⟩
⟨arg-list⟩	::=	**(** **)** | **(** ⟨exp⟩ ⟨arg-list-follow⟩ **)**
⟨exp⟩	::=	**(** ⟨exp⟩ **)** | **num** | **true** | **false** | **ident**
	|	⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩
	|	⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩ | **ident** ⟨arg-list⟩
⟨asop⟩	::=	`=` | `+=` | `-=` | `*=` | `/=` | `%=` | `&=` | `^=` | `|=` | `<<=` | `>>=`
⟨binop⟩	::=	`+` | `-` | `*` | `/` | `%` | `<` | `<=` | `>` | `>=` | `==` | `!=`
	|	`&&` | `||` | `&` | `^` | `|` | `<<` | `>>`
⟨unop⟩	::=	`!` | `~` | `-`
⟨postop⟩	::=	`++` | `--`

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 Elaboration

Each kind of global declaration (gdecl) is based on a closely corresponding construct in C.

- fdecl: declares a function; like a function prototype in C.
- fdefn: defines a function; like a function definition in C.
- typedef: defines an alias for a type that is transparent to the type system, just as in C.

The semantics are complicated because the language also intrinsically supports a foreign function interface through a mechanism of headers. Consult the section on the compile-time environment for details on how headers are supplied. For now, it is sufficient to know that a header is a list of gdecls that satisfy the following conditions:

- They are treated as if they are available to the L3 program *before* the gdecls in the L3 sources proper.
- They can contain type definitions and function declarations, but *not* function definitions. These are instead supplied by the runtime environment in the linking phase.
- The fact that a function declaration was supplied in a header is an essential distinguishing factor that is preserved for the purpose of the static semantics.

To expeditiously capture the similarity and differences between global declarations in headers and in L3 sources, we assume that the header is parsed with the same rules as for sources, and the resulting declarations are tagged as external.

A program is now simply is list of global declarations and definitions. It must elaborate and then store the elaborated (internal) form of function declarations, definitions, and type definitions.

For each function declaration

$$\tau\ f(\tau_1\, x_1, \ldots, \tau_n\, x_n);$$

or function definition

$$\tau\ f(\tau_1\, x_1, \ldots, \tau_n\, x_n)\{s\}$$

we obtain a declaration $f : (\tau_1, \ldots, \tau_n) \to \tau$ and, in a addition, a corresponding definition of f in the second case.

For each type definition

$$\mathsf{typedef}\ \tau\ \alpha$$

which defines the type name α as a transparent synonym for τ, we obtain a definition $\alpha = \tau$. τ must be a valid type itself, which right now just means that it cannot be **void**.

A context Γ that we can use for type checking, therefore now contains:

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, f{:}(\tau_1, \ldots, \tau_n) \to \tau, \mid \Gamma, \alpha = \tau \mid \Gamma, x{:}\tau$$

While elaborating a program, we must obey the following scoping rules:

- A function may be declared several times, in which case the declarations must be compatible (same argument and return types, though not necessarily the same names for the arguments). External functions (functions with at least one declaration in a header file) must not ever be defined. Other functions that are referenced in a call—even in unreachable code—must be defined exactly once and be compatible with their declaration (if there is one). Functions that are never referenced in a call do not need to be defined, even if they have been declared previously, but they may not be defined more than once.
- The name of a function is visible after its first declaration, and within and after its definition. This means a single recursive function does not require a declaration, but two mutually recursive functions require at least one forward declaration. Note that the `main()` function is considered implicitly declared and referenced before any of the code that is explicit in the `.l3` file.
- The name of a defined type is visible after its definition. Type names may be defined only once.
- Names of functions and variables may not collide with defined type names.

4 L3 Static Semantics

Due to the presence of functions and defined type names, type checking is more complex than in L2. We refer to function parameters or variables declared in the body of functions as *local variables*. As noted in the previous section, the typing context Γ contains declarations for functions $f : (\tau_1, \ldots, \tau_n) \to \tau$, type names $\alpha = \tau$, and local variables $x : \tau$. We shall refer to the last occurrence of f in the function declaration list of Γ with the notation $\Gamma(f)$.

- Function parameters and locally declared variables with overlapping scopes may not have the same name. Among other things, this means that for a given function, all function parameters must have distinct names. However, local variables are allowed to shadow function names. This is similar to the behavior of C. You can emulate this specification, for example, by extending the rules for declarations from L2 as follows:

$$\frac{x \notin \mathrm{Dom}(\Gamma) \quad \Gamma, x : \tau \vdash s \ \mathit{valid}}{\Gamma \vdash \mathsf{declare}(x, \tau, s) \ \mathit{valid}}$$

$$\frac{\Gamma(x) = (\tau_1, \ldots, \tau_n) \to \tau' \quad \Gamma, x : \tau \vdash s \ \mathit{valid}}{\Gamma \vdash \mathsf{declare}(x, \tau, s) \ \mathit{valid}}$$

- A function must be called with the correct number of arguments, and with compatible types. The whole expression has the corresponding return type.

$$\frac{\Gamma(f) = (\tau_1, \ldots, \tau_n) \to \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

- The new type void is allowed only as the return type of a function. It expresses that the function returns no value. Such a function can not be called *inside* an expression, but only directly as a statement.

- A function returning void does not require an explicit return statement at the end of each control flow path starting from the beginning of the function. Any explicit return that happens to be present must have the form "`return;`" with no argument.

- The new statement $\mathsf{assert}(e)$ is typed by

$$\frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \mathsf{assert}(e) \ \mathit{valid}}$$

As you can probably observe, we have imported a lot of the non-uniform behavior of C to L3, especially with respect to name collisions and shadowing. It is plausible to handle these inconsistencies to varying extents in the elaborator and the type checker. You are free to make design decisions that suit your compiler. However, wherever you draw your module boundaries (if at all), think carefully about why your implementation is equivalent to this specification.

The static check that there is a return statement along every control flow path from the beginning of a function is similar to L2, except that we omit this check for functions returning void. The check that all local variables are defined before they are used also proceeds as in L2, where function parameters are considered *defined* at the beginning of the function body. This is because they will have values when the execution of the body of a function commences.

Now that we have function calls, we define `main` as an identifier with the special requirement that it can only be defined as a function with no parameters and the return type of `int`. Execution of an L3 program begins in the `main` function. Please refer to the section describing the runtime.

5 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).

6 Project Requirements

For this project, you are required to hand in test cases and a complete working compiler for L3 that produces correct target programs written in AT&T x86-64 assembly language.

We also require that you document your code. Documentation includes both inline documentation and a README document which explains the design decisions underlying the implementation 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 docker containers.

Your compiler and test programs must be formatted and handed in as specified below. For this project, you must also write and hand in at least 20 test programs, at least two of which must fail to compile, at least two of which must generate a runtime exception, at least two of which must execute correctly and return a value, and at least two of which must include header files that you wrote (see below).

Test Files

Test programs should 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`
`//test abort`	program must compile and run but raise `SIGABRT`
`//test error`	program must fail to compile due to an L3 source error
`//test typecheck`	program must typecheck correctly (see below)
`//test compile`	program must typecheck, compile, and link (see below)

followed by the program text. In L3, the exceptions defined are `SIGABRT` (6) and `SIGFPE` (8).

If the test program `$test.l3` is accompanied by a file `$test.h0` (same base name, but `h0` extension), then we will compile the test treating `$test.h0` as the header file. Otherwise, we will 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; our testing framework will ignore any output performed from the printing operations. You are encoraged but not required to write tests that take advantage of this library.

Tests that use a `.h0` header file that you wrote might typecheck but fail to link because they refer to functions that aren't provided by the system: your header file can 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 that you wrote 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, if your tests use a header file that you wrote, your test *must* start with the line `//test error` or `//test typecheck`. Only tests utilizing header files that you wrote should begin with `//test typecheck`.

Now that the language we are compiling supports function calls, 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 which compute Fibonacci numbers, factorials, greatest common divisors, the Ackermann function, and minor variants thereof. Please use your imagination. We will read your tests!

Compiler Source 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 compiler's structure, and any design decisions or specific algorithms utilized at the beginning of this file. Even though your code will not be read for grading the code quality, we may still read it for for the code reviews and to provide you feedback. The `README` will be crucial information for this purpose.

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. 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 compiler structure and any specific algorithms you implemented 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. 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.

Runtime Environment

Your compiler should accept a single, 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 the header file mentioned in the elaboration and static semantics sections. You may not assume that the header file parses and typechecks correctly.

The `15411-l3.h0` header file describes a library for manipuating floating-point values. The implementation of this library can be found in `lab3/runtime/run411.c`. You should assume the types of the implementation match the types in the header file.

The GNU compiler and linker will be used to link your assembly to the implementations of the external functions, so you need not worry much about the details of calling to 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.
- 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.
- 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.

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.

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 Tuesday, October 12 at 11:59 PM At least 20 test cases, at least two of which generate an error, at least two of which raise a runtime exception, and at least two of which return a value, and at which two of which use a header file that you wrote.

You will submit by to the **Test 3** assessment on Notolab. The directory `tests` should only contain your test files. The autograder will test your test files and notify you if there is a discrepancy between your answer and the outcome of the reference implementation. If you feel the reference implementation is in error, please notify the instructors.

Before Thursday, October 21 at 11:59 PM The complete compiler. You will submit to the **Lab 3** assessment on Notolab. 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 results of the autograding are available at Notolab at https://notolab.cs.cmu.edu/.

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.

Code Review

We will be holding code reviews on the week of October 21. We will post sign ups for slots on Piazza as soon as we finalize the schedules. 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 codegen. 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.

Scoring

Your compiler will be graded against the test cases, and your score is computed as follows.

20 * (% passed of new) +
40 * (% passed of large & only & basic) +
20 * (% non-fail of quarantine) +
= subtotal

total = subtotal - (1 point per failure) - (0.5 points per timeout [excluding quarantined])

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 portable.

We again *highly* recommend using an explicit elaboration pass, rather than transforming source code on the fly during parsing. This will make the next lab a significantly smoother experience.

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.

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 but 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.