15-411 Compiler Design, Lab 4 (Spring 2026)

Seth and co. 

Test Cases Due: 11:59 pm, Saturday, March 21, 2026
Compiler Due: 11:59 pm, Saturday, March 28, 2026

1 Introduction

Lab 4 asks you to implement the language L4. This language extends L3 with pointers, arrays, and structs. With the ability to store global state, you should be able to write a wide variety of interesting programs. As always, correctness is paramount, but you should take care to make sure your compiler runs in reasonable time.

2 L4 Syntax

The lexical specification of L4 is changed by adding '`[`', '`]`', '`.`', and '`->`' as lexical tokens; see Figure 1. Whitespace, token delimiting, and comments are unchanged from L3.

The syntax of L4 is defined by the (no longer 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`. Note that according to this precedence table, `*x++` parses as `*(x++)`, which isn't valid syntax. While we could allow `*x++` because `*x` is unambiguously an lvalue and `(*x)++` is unambiguously a statement, we disallow it. This is both to match the precedence rules and also to avoid confusion with the different semantics that statement has in C.

L4 syntax is not context-free

As noted above, the grammar presented for L4 is no longer context free. Consider, for example, the statement

```
foo * bar;
```

If `foo` is a type name, then this is a declaration of a `foo` pointer named `bar`. If, however, `foo` is *not* a type name, then this is a multiplication expression used as a statement.

For those of you using parser combinator libraries, you will be able to backtrack from a parse decision based on whether an identifier is a type name, so this case should not be a problem. However, those of you using parser generators will have a harder time—the decision whether to shift or reduce might be made well ahead of when an identifier is determined to be a typename or not. Solving this ambiguity is a bit tricky.

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

The most common way to handle this context sensitivity is known as the "lexer hack"[1]. The lexer and parser share global mutable state about what typedefs have been parsed. When the lexer encounters an identifier, it checks whether the identifier has been typedef'd, emitting different tokens based on whether it is a typedef ident or not. Dually, when the parser encounters a typedef at the top level (as a ⟨gdecl⟩), it updates the global state with the new typedef'd identifier. Further, since the lexer emits a special token for typedef'd identifiers, the ambiguity is no longer ambiguous. This was the solution intended by the designers of C, as described in a footnote of the ANSI C book.

The lexer hack can be difficult to implement correctly. For instance, your lexer likely performs a lookahead to find the longest matching rule, and this lookahead can lead to tokens being lexed 'earlier' than you might expect. Consider:

```
typedef int foo;
foo func();
```

In this case, by the time the parser parses `typedef int foo;` the lexer may already have lexed the `foo` at the beginning of the next line, and the global state may not have been updated to recognize `foo` as a typedef ident.

Instead of trying to orchestrate a complicated dance between lexer and parser, you could also consider keeping the 'lexer hack' localized to the lexer. Typedef gdecls are rather easy to pick out in a token stream (in regex: `TYPEDEF (any token except SEMICOLON)+ IDENT SEMICOLON`), so your lexer can simply look for this pattern in the stream of tokens being generated. Every time the lexer sees what *appears* to be a typedef declaration, it adds the identifier to the set of typedef'd identifiers. This solution keeps the mutable state local to the lexer, and ensures each typedef'd identifier is added to the state before the lexer moves on to the next token.

[1] See the Wikipedia page here.

⟨program⟩	::=	ϵ | ⟨gdecl⟩ ⟨program⟩
⟨gdecl⟩	::=	⟨fdecl⟩ | ⟨fdef⟩ | ⟨typedef⟩ | ⟨sdecl⟩ | ⟨sdef⟩
⟨fdecl⟩	::=	⟨ret-type⟩ **ident** ⟨param-list⟩ **;**
⟨fdef⟩	::=	⟨ret-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** **;**
⟨sdecl⟩	::=	**struct ident** **;**
⟨sdef⟩	::=	**struct ident** `{` ⟨field-list⟩ `}` **;**
⟨field⟩	::=	⟨type⟩ **ident** **;**
⟨field-list⟩	::=	ϵ | ⟨field⟩ ⟨field-list⟩
⟨ret-type⟩	::=	⟨type⟩ | **void**
⟨type⟩	::=	**int** | **bool** | **ident** | ⟨type⟩ ***** | ⟨type⟩[] | **struct ident**
⟨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⟩ **.** **ident** | ⟨lvalue⟩ `->` **ident**
	|	***** ⟨lvalue⟩ | ⟨lvalue⟩ [⟨exp⟩] | **(** ⟨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** | **NULL** | ⟨unop⟩ ⟨exp⟩
	|	⟨exp⟩ ⟨binop⟩ ⟨exp⟩ | ⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩ | **ident** ⟨arg-list⟩
	|	⟨exp⟩ **.** **ident** | ⟨exp⟩ `->` **ident** | **alloc** **(** ⟨type⟩ **)** | ***** ⟨exp⟩
	|	**alloc_array** **(** ⟨type⟩ **,** ⟨exp⟩ **)** | ⟨exp⟩ [⟨exp⟩]
⟨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 L4

Operator	Associates	Meaning
`() [] -> .`	n/a	explicit parentheses, array subscript, field dereference, field select
`! ~ - * ++ --`	right	logical not, bitwise not, unary minus, pointer dereference, 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

3 L4 Semantics

The static and dynamic semantics for Lab 4 are scattered throughout the lecture notes discussing static semantics, dynamic semantics, mutable store, and structs. Here are some specific pointers:

- Elaboration needs to deal with the fact that `A[f(x)] += 3` cannot be elaborated into assign$(A[f(x)], A[f(x)]+3)$, because calling $f(x)$ might have an effect, like printing or writing to a pointer.
- You may need to preserve some information about each variable's width during elaboration to generate correct code. You may need to have both 4- and 8-byte versions of certain operations in your IRs, which can lead to some amount of code duplication in your middle/backend.
- Struct names and field names are in different namespaces, so `struct foo { int foo; };` is legal. Moreover they are in different namespaces with function names and type names.
- Structs in L4 can be both declared and defined, in both header and L4 source files. They can be declared many times, but defined only once (either in the header or in the L4 source). Struct declarations are not semantically meaningful; struct types may be referred to without

having been previously declared (so long as their size information is not needed, see the next bullet point).

- A struct must have already been defined in order to access fields of values of that struct type. This is inherited from C, and ensures that offsets can be calculated without referring to information later in the file. You will probably want to calculate and store the field sizes and offsets for each struct when the struct is defined.

The default library for this lab, `15411-l4.h0`, is a modification of the previous library that uses 8-byte floating point values stored in pointers rather than 4-byte floating point values stored in integers. Many test cases make use of this library.

For this lab, you do *not* need to lay out structs in a way that is compatible with C, but you are encouraged to do so. You should respect the machine's alignment requirements so that integers and booleans are aligned at least 0 modulo 4 and addresses at least 0 modulo 8, but beyond that, we do not require strict adherence. One reason for this flexibility is that we allow you to represent structs and arrays containing boolean values however you want. Here is what we will potentially test:

- Integers must be stored in memory as 4 continuous bytes (little-endian, as usual on x86-64)
- Pointers must be stored in memory as 8 continuous bytes (little-endian, as usual on x86-64)
- Structs and arrays which contain only ints (and other structs which contain only ints, and so on) must store the ints continuously, in order. So if a is the address of the struct or array, then a is the address of the first struct field or array element, $a + 4$ is the address of the second struct field or array element, and so on.
- Ditto for structs and arrays which contain only pointers, except that $a + 8$ is the address of the second struct field or array element, etc.

We expect your generated code to explicitly capture memory errors (null pointer access, array index out of bounds, etc.), rather than counting on the operating system to notice and raise `SIGSEGV` (11). The autograder enforces this by expecting programs to raise `SIGUSR2` (12) when a memory error occurs. You can raise this signal by calling the standard `raise(sig)` function.

4 Runtime Environment

Your compiler is free to assume types and header files match the functions available in the runtime.

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. See Lab 3 for the full conventions you should follow.

You may notice that the functions `c0_alloc` and `c0_alloc_array` are implemented in `run411.c`. This is because `run411.c` is a modified version of the c0 reference runtime, which needs these functions since the reference compiler targets C, rather than assembly. Please **do not** use these functions. You must implement `alloc` and `alloc_array` yourself. In practice, this means you should be calling `calloc` in your generated assembly.

5 Testing

Test programs have the extension `.l4` 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 memerror`	program must compile and run and raise `SIGUSR2` (12)
`//test error`	program must fail to compile due to an L4 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.

If the test program `$test.l4` is accompanied by a file `$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-l4.h0` as the header file for all `l4` tests, and we will pass that header file to your compiler with the `-l` argument. The `15411-l4.h0` header file describes a library for double-precision floating point arithmetic and printing operations; our testing framework will ignore any output performed from the printing operations.

As in L3, tests that use a custom header file *must* start with the line `//test error` or `//test typecheck`, to avoid dangerous or badly-behaved external calls.

L4 is the first properly Turing-complete language we will compile in this class. As such, there are some very interesting tests in the testing suite. Previous students of this course[2] have left many easter eggs to be discovered. You, too, can and should be creative and leave fun test cases for your peers and future students to find.

Starting from L4 onward, you **may not use AI/LLMs to generate test cases**. Please put some care into the test cases you submit – try to target interesting parts of both the L4 syntax and semantics. Either handcraft your test cases, or write scripts to generate more complex test cases.

[2] It's me! I'm previous students!! -James

6 Extending your LLVM Backend (Optional)

In Lab LLVM, you added a new backend to your compiler targetting the LLVM IR instead of x86-64 assembly. We hope the lab helped you solidify your SSA implementation and develop a modular compiler capable of supporting multiple compilation pipelines. If you have time and wish to add L4 support to your LLVM backend, you are welcome to do so.

The two new LLVM IR instructions you need are `store <type> <val>, ptr <pointer>`, which writes the value `val` of type `type` to the address in `pointer`, and `load <type>, ptr <pointer>`, which returns the value of type `type` stored at `pointer`. LLVM does not have heap allocation built-in, so those should translate to the same function calls you will make in your x86-64 backend. Note that `alloca` allocates a pointer (or struct/array) on the *stack*, not the heap; since L4 and C0 in general does not support structs and arrays in the stack, nor taking the address of variables (that can be stored in stack space), `alloca` will *not* be useful for implementing the L4 features.

7 Submission

For this project, you are expected to hand in a complete working compiler for L4, as well as test cases as usual. The due dates are:

A late day is computed based on the active submission on Gradescope. So if your active submission is past **11:59 pm** on the due dates then it will use a late day.

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 Saturday, March 21, 11:59 pm Ten (10) test cases, at least 1 for each of the first 6 categories mentioned in section 5. All of your test cases should utilize new language features introduced in L4. You should submit to **Test 4** assessment on Gradescope. The directory tests should only contain your test files. Please name your test files with the extension `.l4` prefixed with your team name in lower case.

Before Saturday, March 28, 11:59 pm The complete compiler. You will submit to the **Lab 4** assessment on Gradescope. The directory `compiler/lab4` 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.

Compiler

The files comprising the compiler itself should be collected in a directory `compiler/` which should contain a `Makefile`. Issuing the shell command

```
% make lab4
```

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

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

will run your L4 compiler. The command

```
% make clean
```

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

Important: As with L3, you should update your `README` file to include

- a description of your compiler's high level structure.
- major design decisions or specific algorithms utilized.
- any publicly available external libraries you used.

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 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, in accordance with the course policy. 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:

$$\text{subtotal} = 25 * \frac{\text{passed basic tests}}{\text{total basic tests}} \; + \; 65 * \frac{\text{passed large \& only tests}}{\text{total large \& only tests}}$$

$$\text{compiler} = \text{subtotal} - (1 \text{ point per compiler failure}) - (0.1 \text{ points per executable timeout})$$

Your total score for the lab is computed as follows:

$$\text{test cases} = 10 * \frac{\min(\text{valid test cases}, 10)}{10}$$

$$\text{total} = \text{test cases} + \text{compiler}$$