15-411 Compiler Design, Fall 2020
Lab 4

Seth and co.

Test Programs Due: 11:59 pm, Tuesday, November 3, 2020
Checkpoint Due: 11:59 pm, Thursday, November 5, 2020
Compilers Due: 11:59 pm, Thursday, November 12, 2020

1 Introduction

The goal of the lab is to implement a complete compiler for 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 languages L1, L2, and 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.

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

However, backtracking raises the specter of a performance bug, so you will need to closely consider performance.

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; below, we describe two approaches.

One way to handle this is to perform an ambiguous parse: use one rule to parse both the declaration form and the expression form. Then undo an incorrect decision during elaboration. This approach will almost certainly involve some other adjustment to various pieces of your grammar and lexer. Looking over grammars and parsers from past years, we cannot honestly recommend this technique—results seem to be largely contorted with low confidence in correctness.

Another option is to prevent incorrect decisions from being made. New type identifiers are introduced at the top level (as a ⟨gdecl⟩), and so the parser can update mutable state to record type identifiers as such. With this approach, the lexer can produce different tokens for type and non-type identifiers. This was the solution intended by the designers of C, if one considers the relevant footnote in their book.

This solves the parsing problem, but raises another. The lexer performs a lookahead in order to find the longest match. This affects the lexing of a token which is used immediately after it is introduced—consider:

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

In this case, if the parser parses `typedef int foo;` the lexer may already have lexed the `foo` at the beginning of the next line, so be careful! Despite this potential issue, we recommend this approach because the grammar will continue to look natural and follow the actual understanding of the language syntax.

⟨program⟩	::=	ϵ | ⟨gdecl⟩ ⟨program⟩
⟨gdecl⟩	::=	⟨fdecl⟩ | ⟨fdef⟩ | ⟨typedef⟩ | ⟨sdecl⟩ | ⟨sdef⟩
⟨fdecl⟩	::=	⟨type⟩ **ident** ⟨param-list⟩ **;**
⟨fdef⟩	::=	⟨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⟩
⟨type⟩	::=	**int** | **bool** | **ident** | **void** | ⟨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 described in the lecture notes discussing static semantics, dynamic semantics, mutable store, and structs. Some specific points:

- Elaboration will need to deal with the fact that `A[f(x)] += 3` cannot be elaborated into $\mathsf{assign}(A[f(x)], A[f(x)]+3)$, because calling $f(x)$ might have an effect, like printing or writing to a pointer.
- You will have to preserve at least some size information during elaboration to generate correct code. We suggest you refer to the current (third) edition of Bryant and O'Hallaron's *Computer Systems: A Programmer's Perspective*, or to the current semester's 15-213 notes. Pay particular attention to the effects of 32 bit operators in the upper 32 bits of the 64 bit registers.
- Struct *definitions* obey scoping rules of the other global declarations, that is, they are available only after their point of definition. However, structs may be *declared* implicitly; see Section 2 of the notes.

- The rules for struct declarations and definitions, mostly inherited from C, are carefully engineered so that it should be possible to compute the size and field offsets of each struct without referring to anything found later in the file. You probably want to store the sizes and field offsets in global tables.
- Like type definitions, struct declarations and definitions can appear in external files.
- We expect your generated code to explicitly capture memory errors, rather than counting on the operating system to notice and raise `SIGSEGV` (11). In order to enforce that, the signal associated with memory errors will be `SIGUSR2` (12). You can raise this signal explicitly with the standard `raise(sig)` function.

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. We encourage you to use this library in some of your test cases!

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 will 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, where a, the address of the struct or value of the array, 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.

4 Project Requirements

For this project, you are required to hand in test cases and a complete working compiler for L4 that produces correct target programs written in Intel 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 lab machines.

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 error, and at least two of which must execute correctly and return a value.

Test Files

Test programs should have 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`
`//test abort`	program must compile and run but raise `SIGABRT`
`//test memerror`	program must compile and run and raise `SIGUSR2`
`//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 L4, the exceptions defined are `SIGABRT` (6), `SIGFPE` (8), and `SIGUSR2` (12).

Since the language now supports function calls, the runtime environment contains external functions providing output capabilities (see the runtime section for details). However, we do not check that the output is correct, merely that correct values are eventually returned from library calls.

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. You are strongly encouraged, but not required, to write tests that take advantage of this library.

If your tests use a header file that you wrote, your test *must* start with the line `//test error` or `//test typecheck`. If you include a note that explains what your header file and your test is doing, we might change it to allow the autograder to try to compile (`//test compile`) or run (anything else). Only tests utilizing header files that you wrote should begin with `//test typecheck`.

L4 is the first Turing-complete language explored in this class. As such, you should be able to write some very interesting tests, perhaps adapted from the programs and libraries you wrote in the 15-122 course on *Principles of Imperative Computation* that uses C0.

Please do *not* submit test cases that are only slightly different from each other in terms of the behavior exercised. In addition, we would like some fraction of your test programs to perform "interesting" computations; 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 variations thereon. Please use your imagination!

Compiler Files

The files comprising the compiler itself should be collected in a directory `compiler/` which should contain a `Makefile`. **Important:** You should also update the `README` file and insert a description of your code and algorithms used at the beginning of this file. Even though your code will not be read for grading purposes, we may still read it to provide you feedback. The `README` will be crucial information for this purpose.

Issuing the shell command

```
% make lab4
```

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.

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-l4.h0 $test.l4`. Here, `15411-l4.h0` is the header file mentioned above. You may not assume that the header file parses and typechecks correctly.

The `15411-l4.h0` header file describes a library for manipulating double-precision floating-point values. The implementation of this library can be found in `lab4/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. As a reminder from lab 3, 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 ABI.
- 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.

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 lab machines; the produced assembly must conform to this environment.

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, November 3, 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. You will submit to the **Test 4** assesment 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, November 5, 11:59 pm A compiler which can typecheck the language of lab 4. You will submit to the **Lab 4 Checkpoint** assessment on Notolab, containing the same files as the full Lab 4 (see below). The autograder will build your compiler, run it on all existing test files using the "`-t`" switch, and log which files successfully compiled and which did not. If a test fails to compile unexpectedly (or the reverse), we will mark the test as failed. The checkpoint will grade **typechecking only**.

Before Thursday, November 12, 11:59 pm The complete compiler. You will submit to the **Lab 4** assessment on Notolab. The directory `compiler/lab4` should contain only the sources for your compiler and be submitted as described above. 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 (each with a 6 second time limit), and finally compare the actual with the expected results.

Please note that similar to previous labs, we will count your highest submission across all submissions. Also, 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.

Checkpoint Scoring

Your compiler will be graded against the test cases, and your score is computed as follows. Note that the checkpoint will only see whether a given test compiles successfully (or not), and will mark tests as passed/failed accordingly.

20 * (% passed of new) +
60 * (% passed of large & only & basic)
= checkpoint_subtotal

penalty = min(40, (1 point per failure) + (0.5 points per timeout))

checkpoint_total = checkpoint_subtotal - penalty

Lab Scoring

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

20 * (% passed of new) +
60 * (% passed of large & only & basic)
= lab_subtotal

penalty = min(40, (1 point per failure) + (0.5 points per timeout))

lab_total = lab_subtotal - penalty

Scoring

Your final score for Lab 4 will be computed as

total = max(0.2 * checkpoint_total + 0.8 * lab_total, lab_total)