15-411 Compiler Design, Fall 2013
Lab 4

Instructor: Frank Pfenning
TAs: Flávio Cruz, Tae Gyun Kim, Rokhini Prabhu, Max Serrano

Test Programs Due: 11:59pm, Tuesday, October 21, 2014
Compilers Due: 11:59pm, Tuesday, October 28, 2014

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`.

Again, note that this grammer is *not* context free. Just trying to parse this language with a parser generator in a naive way will *not* result in correct behavior. Some hints on how to parse this language are provided in the next section.

3 L4 Elaboration

We will not provide explicit elaboration rules for everything in L4. Please note you will need to preserve at least some size information during elaboration to generate correct code.

Detail: 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;
```

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

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, 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 could 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⟩); you may wish to parse one global declaration at a time, with suitable changes to the start-of-parse and end-of-parse symbols). 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();
```

⟨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

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.

4 L4 Static Semantics

Top level declarations

New in the grammar are struct declarations and definitions. See Lecture 15 for some static semantics rules. Not explicitly stated there are the following:

- 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.
- Like type definitions, struct declarations and definitions can appear in external files.

Typechecking

The type checking and static semantics rules for L4 are covered in the notes for Lecture 14 on *Mutable Store* (for pointers and arrays) and Lecture 15 on *Structs*. In addition:

- Postfix operators can be applied to destinations. However, there is an additional restriction that statements of the form $*d$`++` and $*d$`--` are disallowed even though they would fit the grammar. This is to avoid confusion with the different semantics such a statement would have in C.

5 L4 Dynamic Semantics

The dynamic semantics of L4 is covered in the notes for Lecture 14 on *Mutable Store* (for pointers and arrays) and Lecture 15 on *Structs*. Increment and decrement statements `d++;` and `d--;` can be elaborated to `d += 1;` and `d -= 1;`, respectively.

For this lab, you should follow the data representations described in the lecture notes. Specifically, booleans should be represent as 4-byte values that are either 0 or 1. This should guarantee interoperability with the standard 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. At this point we do not know if we will test this provision.

6 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 via Autolab 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 exception` *n*	program must compile but raise runtime exception *n*
`//test exception`	program must compile but raise *some* runtime exception
`//test error`	program must fail to compile due to an L4 source error

followed by the program text. In L4, the exceptions defined are `SIGABRT` (6) (assertion failure), `SIGFPE` (8) (arithmetic exception), `SIGSEGV` (11) (memory-related exception) and `SIGALARM` (14) (timeout).

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.

All test files should be collected into a directory `tests/` (containing no other files) and submitted via the Autolab server.

L4 is a sophisticated and reasonably expressive language. 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 which differ in only small ways in terms of the behavior exercised. 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, and minor variants thereof. 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 l4c
```

should generate the appropriate files so that

```
% bin/l4c <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 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/l4c -l 15411.h0 $test.l4`. Here, `15411.h0` is the header file mentioned in the elaboration and static semantics sections.

The runtime for this lab contains functions to perform output. The output functions write to `stderr`, while the result of the program is printed to `stdout`. `15411.h0` contains a listing and a small amount of documentation.

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 ABI.

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.

Using the Subversion Repository

Handin and handout of material is via the course subversion repository.

The handout files for this course can be checked out from our subversion repository via

```
% svn checkout https://svn.concert.cs.cmu.edu/15411-f14/groups/<team>
```

where `<team>` is the name of your team. You will find materials for this lab in the `lab4` subdirectory. Or, if you have checked out `15411-f14/groups/<team>` directory before, you can issue the command `svn update` in that directory.

After adding and committing your handin directory to the repository with `svn add` and `svn commit` you can hand in your tests or compiler through Autolab.

```
https://autolab.cs.cmu.edu/15411-f14
```

Once logged into Autolab, navigate to the appropriate assignment and select

```
Checkout your work for credit
```

from the menu. It will perform one of

```
% svn checkout https://svn.concert.cs.cmu.edu/15411-f14/groups/<team>/lab4/tests
% svn checkout https://svn.concert.cs.cmu.edu/15411-f14/groups/<team>/lab4/compiler
```

to obtain the files directories to autograde, depending on whether you are handing in your test files or your compiler.

If you are submitting multiple versions, please remember to commit your changes to the repository before asking the Autolab server to grade them! And please do not include any compiled files or binaries in the repository!

What to Turn In

Hand in on the Autolab server:

- 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. The directory `tests/` should only contain your test files and be submitted via subversion as described above. The server will test your test files and notify you if there is a discrepancy between your answer and the outcome of the reference implementation. You may hand in as many times as you like before the deadline without penalty. If you feel the reference implementation is in error, please notify the instructors. The compiled binary for each test case should run in 2 seconds with the reference compiler on the lab machines; we will use a 5 second limit for testing compilers.

  Test cases are due **11:59pm on Tuesday, October 21, 2014**.

- The complete compiler. The directory `compiler/` should contain only the sources for your compiler and be submitted via subversion. The Autolab server will build your compiler, run it on all extant test files, link the resulting assembly files against our runtime system (if compilation is successful), execute the binaries (each with a 5 second time limit), and finally compare the actual with the expected results. You may hand in as many times as you like before the deadline without penalty.

  Compilers are due **11:59pm on Tuesday, October 28, 2014**.

7 Notes and Hints

The rules for struct declarations and definitions 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.

Data now can have different sizes, and you need to track this information throughout the various phases of compilation. We suggest you read Section 4 of the Bryant/O'Hallaron notes on x86-64 Machine-Level Programming available from the course Resources page, especially the paragraph on move instructions and the effects of 32 bit operators in the upper 32 of the 64 bit registers. Some notes can also be found in Section 6 of the notes for Lecture 15.