15-411 Compiler Design: Lab 4
Fall 2012

Instructor: Andre Platzer
TAs: Ian Gillis and Alex Crichton

Test Programs Due: 11:59pm, Tuesday, October 23, 2012
Compilers Due: 11:59pm, Tuesday, October 30, 2012

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L4*. This language extends *L3* with structs, arrays, and pointers. 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.

A special "dangerous bend" symbol[1] marks some particularly important warnings. These warnings are about "epic fail" issues, not "interesting suggestions," so you *must not* ignore them. Other issues are important too, so please pay attention to them as well.

Black bars on the right side of the text indicate significant changes from the previous lab.

Spec changes during the lab will be hilighted with red bars.

2 *L4* Syntax

The lexical specification of *L4* remains unchanged from that of *L3*. The syntax of *L4* is the superset of *L3* as presented in Figure 2.1. Ambiguities in this grammar are resolved according to the same rules of precedence as in *L4*.

Whitespace and Token Delimiting

In *L4*, whitespace is either a space, horizontal tab (`\t`), vertical tab (`\v`), linefeed (`\n`), carriage return (`\r`) or formfeed (`\f`) character in ASCII encoding. Whitespace is ignored, except that it delimits tokens. Note that white space is not a *requirement* to terminate a token. For instance, `()` should be tokenized into a left parenthesis followed by a right parenthesis according to the given lexical specification. However, white space delimiting can disambiguate two tokens when one of them is present as a prefix in the other. For example, `+=` is one token, while `+ =` is two tokens. The lexer should produce the longest valid token possible. For instance, `++` should be lexed as one token, not two.

Comments

L4 source programs may contain C-style comments of the form `/* ... */` for multi-line comments and `//` for single-line comments. Multi-line comments may be nested (and of course the delimiters must be

[1] See Knuth, The TEXbook, 9th printing, 1986

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

balanced).

2.1 Grammar

The syntax of *L4* is defined by the (no longer context-free!) grammar in Figure 2.1. Ambiguities in this 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`. Again, note that this grammer is *not* context free. Parsing typedefs with a parser generator in the naive way will *not* result in correct behavior.

```
⟨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 | ⟨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⟩ )
⟨esleopt⟩            ::=  ε | else ⟨stmt⟩
⟨control⟩            ::=  if ( ⟨exp⟩ ) ⟨stmt⟩ ⟨elseopt⟩ | while ( ⟨exp⟩ ) ⟨stmt⟩
                      |   for ( ⟨simpopt⟩ ; ⟨exp⟩ ; ⟨simpopt⟩ ) ⟨stmt⟩
                      |   continue; | break; | return ⟨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⟩ | NULL
                      |   alloc_array ( ⟨type⟩ , ⟨exp⟩ ) | ⟨exp⟩ [ ⟨exp⟩ ]
⟨asop⟩               ::=  = | += | -= | *= | /= | %= | &= | ^= | |= | <<= | >>=
⟨binop⟩              ::=  + | - | * | / | % | < | <= | > | >= | == | !=
                      |   && | || | & | ^ | | | << | >>
⟨unop⟩               ::=  ! | ~ | -
⟨postop⟩             ::=  ++ | --
```

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 2: Precedence of operators, from highest to lowest

3 *L4* Elaboration

We will not provide explicit elaboration rules for everything in *L4*. Please note that if you are interested in performing type-directed optimizations in future labs, it is recommended that you perform elaboration in a way which preserves type information.

3.1 Elaboration hints

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

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 will not be a problem.

However, those of you using parser generators will have a harder time—the decision whether to shift or reduce could have been made well ahead of when an identifier is determined whether 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 decl form and the exp form (hint: an existing rule may work!). Then, resolve the incorrect decision during elaboration. This approach will almost certainly involve some other adjustment to various pieces of your grammar and lexer. However, if you are already performing elaboration in a seperate phase, we believe this will be quite manageable.

Another option is to prevent incorrect decisions from being made. New type identifiers are introduced at the gdecl level; you may wish to parse one gdecl at a time (with suitable changes to the start-of-parse and end-of-parse symbols). With this approach, the lexer can produce a different tokens distinguishing type and non-type identifiers.

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 will already have lexed the `foo` at the beginning of the next line. In order to ensure that the lexer has access to context information which includes in the typedef, you can stop parsing a typedef *before* the `;`. If you take this approach, you will need to determine whether the typedef is well formed syntactically in some other way. Be careful here! This approach may involve becoming more familiar with the internals of the parser than you desire.

4 *L4* Static Semantics

gdecls

The following are the static semantics of the newly available gdecls.

- The new gdecls in the grammar all obey the lexical scoping rules of the other gdecls (i.e. they are available only after their point of declaration).
- Like typedefs, structs declarations and definitions can appear in external files.
- Like functions, structs can be declared multiple times but defined at most once. Not all declared structs are required to be defined. Unlike functions, struct definitions are allowed to appear in external files.
- Note that unlike C, structs cannot be used anonymously as types. They must declared with a name before thay can appear in the source of a typedef or within the types of variables and functions.

- Struct names have their own namespace.
- Struct definitions place all constituent fields in a separate namespace. This means that all field names in a struct must be distinct, though different structs may share field names.
- Struct definitions makes an equivalent structure **declaration** available within the body of the struct.

Typechecking

- The type checking rules for the new language constructs are covered in the lecture notes on semantic analysis and specifications. The type checker must be upgraded to enforce the distinction between small and large types.
- All local variables, function parameters, and return types must be of small types.
- Equality and disequality are now overloaded for any small type–this includes pointers and arrays as well. Comparisons between pointers are only valid if they are between two pointers of compatible types.
- `NULL` is a valid value of any pointer type. To simplify type checking expressions that explicitly dereference `NULL` (such as `*NULL` or `NULL->a`) are disallowed. However, expressions such as `NULL == NULL` are allowed.
- Structs can be declared in other structs as fields, and arrays of structs can be created. However, for allocations and field placements to succeed, the size of the struct would need to be known in advance. The size of a struct is always known if a well-formed definition is within scope. Therefore, lexical scoping saves you some work here.
- Where a struct declaration is available (but not a definition), the structure can appear as part of a pointer or array type.
- `e->f` can be treated as syntactic sugar for `(*e).f`
- Postfix operators can be applied to lvalues. However, there is an additional restriction that statements of the form `*exp++;` are disallowed. This is to maintain the difference from C semantics, where the above expression would return the value found at a pointer, and then perform pointer arithmetic. *L4* does not have any pointer arithmetic.
- lvalues are equired to have small types, and so are their corresponding right sides. Apart from this requirement, lvalues are subject to the same type correctness requirements as other expressions.

5 *L4* Dynamic Semantics

The dynamic semantics of *L4* extend those of *L3* with rules for the new language constructs. These rules are covered in the lecture notes on semantic analysis and specifications.

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 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 ten 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 *L3* source error

followed by the program text. In *L4*, the exceptions defined are `SIGFPE` (8), `SIGSEGV` (11), `SIGALARM` (14). Note that infinitely recursing programs might either raise 11 or 14, depending on whether they first run out of memory or time. Test programs which exercise this behavior should therefore only verify that *some* exception is raised.

The runtime environment contains external functions providing I/O capabilities (see the runtime section for details). Beginning with *L3*, and now in *L4*, the testing framework takes advantage of this. For a test program `$test.l4`:

- If a file `$test.l4.in` exists, its contents will be available to the program as input during testing.
- If a file `$test.l4.out` exists *and the program returns*, the output of the program must be identical to the contents of the file for the test to pass.

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

The reference compiler may display a warning on `//test error` if your test case accidentally exercises a language feature that might be a part of a future lab or C0. Please do not ignore this warning. Do not hand in tests that cause this warning.

L4 is a sophisticated and reasonably expressive language. You should be able to write some very interesting tests.

Please do *not* submit test cases which differ in only small ways (in terms of 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. We will read your tests!

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 l4rt.h0 $test.l4`. Here, `l4rt.h0` is the ubiquitous header mentioned in the elaboration and static semantics sections.

The runtime for this lab contains functions to perform input and output. The input functions read from `stdin`, and the output functions write to `stderr`. (The result of the program is printed to `stdout`.) If an input function tries to read off the end of a file, the effect will be the same as a stack overflow–that is, `SIGSEGV` (11) will be raised. The runtime provides some other functions, as well; `l4rt.h0` contains a listing and a small amount of documentation.

The runtime environment will contain the function `void *calloc(size_t nobj, size_t nbytes)` which allocates an array of nobj objects of size nbytes all initialized to 0. You should use this to allocate heap memory as necessary, assuming that size_t is a 4 byte unsigned int. *L4* is garbage collected, so there is no explicit freeing of memory.

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. Also recall the provision that all declared functions need not be defined. As in *L3*, all functions must be mangled by adding a `_c0_` prefix. As long as you mangle all your functions, the linker should prevent the complete compilation of any test that does not define all used symbols.

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.

To maintain interoperability with the ABI for C on linux, bools should be given an underlying 32 bit integer representation with false mapping to 0 and true mapping to 1. This runtime detail should in no way be visible in *L4* itself.

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://cvs.concert.cs.cmu.edu/15411-f12/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-f12/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-f12
```

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://cvs.concert.cs.cmu.edu/15411-f12/groups/<team>/lab4/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f12/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 10 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 23, 2012**.

- 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 30, 2012**.

7 Notes and Hints

Structs cannot be used until they are defined (though pointers to them can be) and structs definitions have lexical scope. This makes it 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 Resources page, especially the paragraph on move instructions and the effects of 32 bit operators in the upper 32 of the 64 bit registers.

Your code must strictly adhere to struct alignment requirements, meaning that within a struct each field must be properly aligned, possibly requiring that padding be added within the struct. Ints and bools must be aligned at 0 mod 4, small values of type $\tau*$ and $\tau[\,]$ are aligned at 0 mod 8, and structs are aligned according to their most strictly aligned field. You may also read the Section 3.1.2 in the *Application Binary Interface* description available from the Resources page.