15-411 Compiler Design: Lab 4
Fall 2010

Instructor: Andre Platzer
TAs: Anand Subramanian and Nathan Snyder

Test Programs Due: 11:59pm, Thursday, October 21, 2010
Compilers Due: 11:59pm, Thursday, October 28, 2010

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 – these features should be familiar from the C programming language. This feature set will require you to change all phases of the compiler from the third lab. You should now be able to write a great variety of interesting programs with the language.

2 Requirements

As for Lab 3, you are required to hand in test programs as well as a complete working compiler that translates *L4* source programs into correct target programs written in x86-64 assembly language. When encountering an error in the input program (which can be a lexical, grammatical, or static semantics error) the compiler should terminate with a non-zero exit code and print a helpful error message. To test the target programs, we will assemble and link them using `gcc` on the lab machines and run them under fixed but generous time limits.

3 *L4* Syntax

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

⟨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** | ⟨type⟩ `*` | ⟨type⟩ `[]` | **struct ident** | **ident**

⟨block⟩ ::= `{` ⟨decls⟩ ⟨stmts⟩ `}`

⟨decls⟩ ::= ϵ | ⟨decl⟩ ⟨decls⟩

⟨decl⟩ ::= ⟨type⟩ **ident** **;** | ⟨type⟩ **ident** **=** ⟨exp⟩ **;**

⟨stmts⟩ ::= ϵ | ⟨stmt⟩ ⟨stmts⟩

⟨stmt⟩ ::= ⟨simp⟩ **;** | ⟨control⟩ | **;** | ⟨block⟩

⟨simp⟩ ::= ⟨exp⟩ ⟨asop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨postop⟩ | ⟨exp⟩

⟨simpopt⟩ ::= ϵ | ⟨simp⟩

⟨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⟩ **)** | ⟨intconst⟩ | **true** | **false** | **ident**

| ⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ | ⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩ |

| **ident** ⟨arg-list⟩ | ⟨exp⟩ **.** **ident** | ⟨exp⟩ `->` **ident**

| ⟨exp⟩ `[` ⟨exp⟩ `]` | `*` ⟨exp⟩ | **alloc** `(` ⟨type⟩ `)`

| **alloc_array** `(` ⟨type⟩ **,** ⟨exp⟩ `)` | **NULL**

⟨intconst⟩ ::= **num** (in the range $0 \leq$ **num** $< 2^{32}$)

⟨asop⟩ ::= `=` | `+=` | `-=` | `*=` | `/=` | `%=` | `&=` | `^=` | `|=` | `<<=` | `>>=`

⟨binop⟩ ::= `+` | `-` | `*` | `/` | `%` | `<` | `<=` | `>` | `>=` | `==` | `!=`

| `&&` | `||` | `&` | `^` | `|` | `<<` | `>>`

⟨unop⟩ ::= `!` | `~` | `-`

⟨postop⟩ ::= `++` | `--`

Figure 1: Grammar of *L4*

Operator	Associates	Meaning
`() [] -> .`	left	parens, array subscript, field dereference, field select
`! ~ - * ++ --`	right	logical not, bitwise not, unary minus, pointer dereference increment, decrement
`* / %`	left	integer times, divide, modulo
`+ -`	left	plus, minus
`<< >>`	left	(arithmetic) shift left, right
`< <= >= >`	left	comparison
`== !=`	left	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: Operator precedence, from highest to lowest

4 *L4* Elaboration

We will not provide elaboration rules for *L4*. However, if you are interested in performing type-based optimizations in the future it is suggested that you elaborate to a form which makes the dynamic semantics obvious while still preserving type information. You may also wish to handle the more lexically oriented aspects of the static semantics in the elaborator, as in *L3*.

5 *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, a struct 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 pointer comparisons also. 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`

lvalues

Finally, we need to introduce a notion of lvalues to determine what it is acceptable to assign to, what should not be assigned to. lvalues classify the forms of expressions that are allowed to appear on the left side of an assignment.

⟨lvalue⟩ ::= **ident** | ⟨lvalue⟩ `.` **ident** | ⟨lvalue⟩ `->` **ident** | ⟨lvalue⟩ `[` ⟨exp⟩ `]` | `*` ⟨lvalue⟩

- Note that definition of lvalues is recursive over lvalues. This is not strictly necessary to give a well-formed semantics. It is a stylistic decision. Therefore, expressions such as `*f()` are disallowed as lvalues.

- Postfix operators can also be applied only to lvalues. There is an additional restriction: statements of the form `*exp++;` and `*exp--;` are disallowed. This is in order to avoid confusion with C. In C the above expression would return the value found at a pointer, and then perform pointer arithmetc. *L4* does not have any concept of pointer arithmetc, and elaboration would rewrite that expression to `*exp = *exp + 1;`

- lvalues are required 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 any other expression.

- There are many valid strategies for implementing lvalue checking – some of them push more of the logic to the parser, and other push more of the logic towards the typechecker and elaborator. These can all produce legitimate results. However, beware if you are using a LALR parser generator – the error messages that your compiler will be very cryptic.

6 *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.

7 *L4* Compilation and 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.h test.l4`. The linking 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.

8 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. 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.

Test Files

Test files should have extension `.l4`. They are to be formatted and handed in as in *L3*.

You can also feel free to call any of the functions in `l4rt.h`. You may find them helpful for diagnostic purposes, and they will also test that you adhere to the calling convention correctly.

The language is now capable of expressing a huge variety of computations, so you should be able to produce some interesting test cases. Any bugs that slip through in *L4* will make safely applying optimizations in the next lab very difficult, so try to thoroughly exercise the new language

constructs. However, you should refrain from submitting large numbers of stress tests or writing tests that abuse the differences between the gcc-based reference compiler and the student compilers. Some specific restrictions on test cases were given in the lab4 prerelease email.

Compiler Files

The compiler sources are to be handed in as in *L3*. The compiler and the make target should be called `l4c`.

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-f10/<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-f10/<team>` directory before, you can issue the command `svn update` in that directory.

After first adding (with `svn add` or `svn copy` from a previous lab) and committing your handin directory (with `svn commit`) to the repository you can hand in your tests or compiler by selecting

```
S3 - Autograde your code in svn repository
```

from the Autolab server menu. It will perform one of

```
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<team>/lab4/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15411-f10/<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 as error and at least two others raise a runtime exception. The directory `tests/` should only contain your test files and be submitted via subversion or as a tar file 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 8 second limit for testing compilers.

  Test cases are due **11:59pm on Thu Oct 21, 2010**.

- The complete compiler. The directory `compiler/` should contain only the sources for your compiler and be submitted via subversion or as a tar file as described above. 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 8 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 Thu Oct 28, 2010**.

9 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 note 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.