15-411 Compiler Design: Lab 4
Fall 2008

Instructor: Frank Pfenning
TAs: Rob Arnold and Eugene Marinelli

Test Programs Due: 11:59pm, Tuesday, October 21, 2008
Compilers Due: 11:59pm, Tuesday, October 28, 2008
Revision 1: Mon Oct 20, 2008
Revision 2: Sat Oct 25 (see last page)

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L4*. This language extends *L3* by structs (in the sense of C), pointers, and arrays. This means the main change from the third lab is that you will have to deal with *memory references*. Because pointers on the x86-64 architecture are 64 bits wide, this also means you will have to deal with differently sized data. Again, performance of both the compiler and the compiled code are only minor considerations at this stage. However, the threshold for compilation time will be slightly tighter, so you may need to start to pay attention to some performance issues.

2 Requirements

As for Labs 1, 2, and 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 syntax of *L4* is defined by the context-free grammar in Figure 1. This is an extension of the grammar for *L3* in the sense that a correct *L3* program should still parse correctly if it does not use any of the new keywords. 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`. Comments are as in *L3*.

⟨program⟩ ::= ⟨gdecl⟩* ⟨function⟩*

⟨gdecl⟩ ::= **extern** ⟨type⟩ ⟨ident⟩ **(** ⟨paramlist⟩ **) ;** |
struct ⟨ident⟩ **{** [⟨ident⟩ **:** ⟨type⟩ **;**]* **} ;**

⟨function⟩ ::= ⟨type⟩ ⟨ident⟩ **(** ⟨paramlist⟩ **)** ⟨body⟩

⟨paramlist⟩ ::= ε | ⟨ident⟩ **:** ⟨type⟩ [**,** ⟨ident⟩ **:** ⟨type⟩]*

⟨body⟩ ::= **{** ⟨decl⟩* ⟨stmt⟩* **}**

⟨decl⟩ ::= **var** ⟨ident⟩ [**,** ⟨ident⟩]* **:** ⟨type⟩ **;**

⟨type⟩ ::= **int** | ⟨ident⟩ | ⟨type⟩ ***** | ⟨type⟩ **[ ]**

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

⟨simp⟩ ::= ⟨exp⟩ ⟨asop⟩ ⟨exp⟩ | ⟨exp⟩

⟨control⟩ ::= **if (** ⟨exp⟩ **)** ⟨block⟩ [**else** ⟨block⟩] |
while (⟨exp⟩ **)** ⟨block⟩ | **for (** [⟨simp⟩] **;** ⟨exp⟩ **;** [⟨simp⟩] **)** ⟨block⟩ |
continue ; | **break ;** | **return** ⟨exp⟩ **;**

⟨block⟩ ::= ⟨stmt⟩ | **{** ⟨stmt⟩* **}**

⟨exp⟩ ::= **(** ⟨exp⟩ **)** | ⟨intconst⟩ | **NULL** |
⟨ident⟩ | ***** ⟨exp⟩ | ⟨exp⟩ **.** ⟨ident⟩ | ⟨exp⟩ **[** ⟨exp⟩ **]** | ⟨exp⟩ `->` ⟨ident⟩
⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ |
⟨ident⟩ **(** [⟨exp⟩ [**,** ⟨exp⟩]*] **)** |
new (⟨type⟩ [[⟨exp⟩]] **)**

⟨ident⟩ ::= [**A-Z_a-z**][**0-9A-Z_a-z**]*

⟨intconst⟩ ::= [**0-9**][**0-9**]* (in the range $0 \leq \text{intconst} < 2^{32}$)

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

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

⟨unop⟩ ::= `!` | `~` | `-`

The precedence of unary and binary operators is given in Figure 2.
Non-terminals are in ⟨angle brackets⟩, optional constituents in [brackets].
Terminals are in **bold**.

Figure 1: Grammar of *L4*

Operator	Associates	Meaning
`() [] -> .`	left	parens, subscript, field dereference, field select
`! ~ - *`	right	logical not, bitwise not, unary minus, dereference
`* / %`	left	integer times, divide, modulo
`+ -`	left	plus, minus
`<< >>`	left	(arithmetic) shift left, right
`< <= > >=`	left	comparison
`== !=`	left	equality, disequality (integer or pointer)
`&`	left	bitwise and
`^`	left	bitwise exclusive or
`|`	left	bitwise or
`&&`	left	logical and
`||`	left	logical or
`= += -= *= /= %=` `&= ^= |= <<= >>=`	right	assignment operators

Figure 2: Precedence of operators, from highest to lowest

4 *L4* Runtime Objects

The *L4* language has four kinds of identifiers: those standing for functions (g), those standing for variables (x), those standing for structs (s), and those standing for field names (f). These are in separate name spaces, so a function, a variable, a struct, and a field may have the same name without conflict. Reserved words of the grammar (`extern`, `struct`, `var`, `int`, `if`, `else`, `while`, `for`, `continue`, `break`, `return`, `NULL`, `new`) cannot be used as any kind of name.

The *L4* language has some complexity in its new constructs, dealing with memory (structs, pointers, and arrays). In order to describe these concisely we use an abstract syntax notation. Eliding some expressions which are as in *L3*:

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \textbf{int} \mid s \mid \tau* \mid \tau[\,] \\ \text{Expressions} & e & ::= & c \mid \textbf{null} \mid x \mid *e \mid e.f \mid e[e] \mid e{\rightarrow}e \mid \textbf{new}\,\tau \mid \textbf{new}\,\tau[e] \mid \cdots \end{array}$$

We also have to consider the runtime objects created during the execution of a program. We divide these into two classes: small values and large values. Small values can be held in registers, large values must be in memory. We use a for an address of an object in memory and n for an integer.

$$\begin{array}{lrcl} \text{Small Values} & w & ::= & a \mid n \\ \text{Large Values} & W & ::= & \{f_1{=}V_1; \ldots f_n{=}V_n;\} \mid [V_0, \ldots, V_{n-1}] \\ \text{Values} & V & ::= & w \mid W \end{array}$$

The first kind of large value is a struct with fields $f_1, \ldots, f_n$, the second an array with n elements. On occasion, we need to know the exact size of objects of a given type, $|\tau|$. We compute this as follows from the type:

$$\begin{array}{lcl} |\mathbf{int}| & = & 4 \\ |\tau*| & = & 8 \\ |\tau[\,]| & = & 8 \\ |s| & = & \|f_1{:}\tau_1; \ldots f_n{:}\tau_n;\| \end{array}$$

The size of a struct, written here as $\|f_1{:}\tau_1; \ldots f_n{:}\tau_n;\|$ is computed by traversing the fields from left to right, adding padding as necessary so that alignment restrictions on the fields are satisfied. **int**'s are 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. Padding may need to be added at the end of a struct so that its total size is a multiple of its most strictly aligned field.

Arrays $[V_0, \ldots, V_{n-1}]$ cannot be directly embedded in a struct or other array, since their size is unknown at compile time. This is the reason that a value of type $\tau[\,]$ is *small*: it is merely the starting address of the array in memory. Arrays must be aligned according to the requirement of their elements. The start of an array is returned by a call to the `malloc` or `calloc` library function which guarantees an address 0 mod 8, which is the strictest alignment required in our language.

Addresses are represented as 8 byte (unsigned) integers in the machine. They are obtained from the runtime system which allocates new memory, or by addition (address arithmetic). There is a special address 0 which is never returned by the runtime system and will be used to denote the null pointer. Memory access is represented as $M[a]$, which reads from memory when used as a value and writes to memory when used on the left-hand side of an assignment $M[a] \leftarrow w$. The number of bytes read or written depends on the size of the small value w.

Along similar lines, we write $V[x]$ for the value of a variable x. On the left hand side $V[x] \leftarrow w$ assigns to x; on the right hand side $V[x]$ denotes the current value of x.

The most important judgments we make are typing, written $e : \tau$, and evaluation, written $e \Rightarrow w$. The first is part of the *static semantics*, which must satisfy some additional conditions that are stated informally. The latter constitutes the *dynamic semantics*, but is also only partially formal since evaluating expressions has effects that we only describe informally. As a general convention, expressions must be evaluated from left to right so that any side effects happen in a deterministic order.

We now present both the static and dynamic semantics for each new construct (structs, pointers, arrays) in turn.

Structs

Statically, several properties must be checked for struct declarations.

- Every type identifier s used in a struct declaration or function must be declared with an explicit `struct` s `{...};` somewhere in the file. The order of `struct` declarations is irrelevant, but the syntax restricts the file so that they can be mixed with `extern` declarations but must precede all function definitions.
- All struct declarations in a file must have distinct names s.

- Field names in a struct declaration must all be distinct. However, different struct declarations may reuse field names.
- Fields of structs can refer to other structs which can in turn refer to other structs and so on. However, any circular reference from a struct to itself following such a chain must go through a pointer or array type.

The main expression construct for structs is $e.f$. It is easily type-checked.

$$\frac{e : s \quad \textbf{struct}\ s\ \{\ldots f{:}\tau; \ldots\}}{e.f : \tau}$$

We write offset(s, f) for the offset of field f is structure s, in bytes. We suggest to compute this information early and store it in a table so that the compiler can access it easily.

The evaluation rule is slightly tricky, and carries a recurring theme. In several circumstances evaluating an expression yields an address of some object in memory. When this object is small, we can return it directly as a small value for further computation. When the object is large, we cannot directly operate on the object, but must return its address instead.

$$\begin{array}{rcll} e.f & \Rightarrow & M[a+k] & \text{if } e \Rightarrow a \text{ and offset}(s,f) = k \\ & & & \text{where } e : s \text{ and } \textbf{struct}\ s\ \{\ldots f{:}\tau; \ldots\} \text{ and } \tau \text{ small} \\ & \Rightarrow & a + k & \text{as above, except that } \tau \text{ large} \end{array}$$

We also have the derived form $e{\rightarrow}f$, which can be desugared into $(*e).f$.

Pointers

The typing for pointer operations are straightforward.

$$\frac{e : \tau*}{*e : \tau} \qquad \frac{}{\textbf{new}\ \tau : \tau*} \qquad \frac{}{\textbf{null} : \tau*}$$

Unfortunately, the typing of the **null** pointer present a challenge, since it potentially has infinitely many types. This is necessary so that expressions such as `p == NULL` or statements such as `p->f = NULL` where the left-hand side has some type $\tau*$ are all well-typed.

We suggest to create a new type **any** used only internally. In key places during type-checking where some types are required to be equal, we can allow **any** on either side or both sides. In programming language terms we say that **null** is *polymorphic*; fortunately it is the only polymorphic construct in the language so you can still implement a relatively simple type checker. The main places where type comparisons take place is in compiling equality (`==`), disequality (`!=`), assignments, and function calls. To avoid pathologies in type-checking, we disallow $*$**null** because it type is ambiguous and may not be resolved by context. You can still reliably dereference the null pointer, should you be so inclined, for example with `var null : int*; null = NULL; *null;`

Assume a function g has prototype

$$\tau\ g(x_1{:}\tau_1, \ldots, x_n{:}\tau_n);$$

Then we have

$$\frac{e_i : \tau_i \quad (1 \le i \le n)}{g(e_1, \ldots, e_n) : \tau}$$

$$\frac{e : \tau \quad (\textbf{return} \text{ in body of } g)}{\textbf{return}\ e\ \textit{valid}}$$

where s *valid* indicates that the statement s is valid.

The operational semantics for $*e$ evaluates e to an address and then returns the memory contents at that address. Dereferencing the null pointer must raise the `SIGSEGV` exception. In an implementation this can be accomplished without any checks, because the operating system will prevent read access to address 0 and raise the appropriate exception.

null	$\Rightarrow$	0	
$*e$	$\Rightarrow$	$M[a]$	if $e \Rightarrow a$ with $a \neq 0$, $e : \tau*$ and τ small
	$\Rightarrow$	a	as above, except that τ large
	raises `SIGSEGV`		if $a = 0$
new τ	$\Rightarrow$	a	where $M[a], \ldots, M[a + \lvert\tau\rvert - 1]$ are freshly allocated locations

The values stored in freshly allocated location must be all 0. This can be achieved with `calloc()` and means that values of type **int** are simply 0, values of type $\tau'*$ are null pointers, all fields of structs are recursively set to 0, and values of array type have address 0 which is akin to a null array reference.

Arrays

Arrays are similar to pointers.

$$\frac{e_1 : \tau[\,] \quad e_2 : \textbf{int}}{e_1[e_2] : \tau} \qquad \frac{e : \textbf{int}}{\textbf{new}\ \tau[e] : \tau[\,]}$$

$e_1[e_2]$	$\Rightarrow$	$M[a + n\lvert\tau\rvert]$	if $e_1 \Rightarrow a$ and $e_2 \Rightarrow n$ with $e_1 : \tau[\,]$, τ small
			$M[a + n\lvert\tau\rvert]$ allocated
	$\Rightarrow$	$a + n\lvert\tau\rvert$	as above, except that τ large
	undefined		otherwise
new $\tau[e]$	$\Rightarrow$	a	if $e \Rightarrow n$ and $M[a], \ldots, M[a + (n-1)\lvert\tau\rvert]$ are freshly allocated

Values in a freshly allocated array are all initialized to 0, as for pointers. The value of an out-of-bounds array access is *undefined* in this version of the language.

Assignment

Assignment is now more general, because the left-hand side need not be a variable, but could be a more complicated expression so we can write, for example, `*p = *q` to copy the contents of q to the location denoted by p. The left-hand side of an assignment must be a so-called *lvalue* ("left

value"), which is a certain kind of expression. We define lvalues, assuming that the expression is already known to be well-typed.

$$\text{Lvalues} \quad v \quad ::= \quad x \mid *e \mid e.f \mid e_1[e_2]$$

For an assignment to be a valid statment, the lvalue on the left-hand side must have a small type.

$$\frac{v : \tau \quad e : \tau \quad \tau \; \mathit{small}}{v = e \; \mathit{valid}}$$

We also have a new statement which is just an expression e which is evaluated for possible effects. e is also required to have small type and its computed value is discarded so that it can be seen as shorthand for $x = e$ where x is a fresh variable not used anywhere else.

To show how assignment evaluates we use some notational slight of hand.

$$\begin{array}{rl} v = e & \text{has the effect of } V[x] \leftarrow w \\ & \text{if } v \Rightarrow V[x] \text{ and } e \Rightarrow w \\ & \\ & \text{has the effect of } M[a] \leftarrow w \\ & \text{if } v \Rightarrow M[a] \text{ and } e \Rightarrow w \end{array}$$

Because we check that v is an lvalue and that the type of e is small, this should cover all possible cases. Note that v must be evaluated before e.

The meaning of complex assignment operators now also changes and is no longer a syntactic expansion because expressions can have side effects. An assignment

$$v \; \mathit{op}\texttt{=} \; e$$

should execute as

$$\begin{array}{ll} V[x] \leftarrow V[x] \; \mathit{op} \; w & \text{if } v \Rightarrow V[x] \text{ and } e \Rightarrow w \\ M[a] \leftarrow M[a] \; \mathit{op} \; w & \text{if } v \Rightarrow M[a] \text{ and } e \Rightarrow w \end{array}$$

Functions

Regarding functions, several properties must be checked.

- Every function that is called must be declared somewhere in the file, either using `extern` or with a definition. The order of the functions in the file is irrelevant, but a function may not be declared more than once.
- Functions must be called with the correct number of arguments of the correct type.
- All function arguments and function results must be of small type.
- Functions must terminate with an explicit `return` statement. This is checked by verifying that each finite control flow path originating at the top of a function ends in a `return` statement. See a more detailed explanation in the description of *L2*.
- There must be a function `main()` which returns an integer as the result of the overall computation (and not an exit status).
- Each `break` or `continue` statement must occur inside a `for` or `while` loop.

Variables

Regarding variables, we need to check three properties, the last of which is new for this language.

- Every variable must be declared, either as a function parameter or an explicit local variable with a `var` declaration. Function parameters and local variables are local to a function and have nothing to do with parameters or local variables in other functions. Variables may not be redeclared, that is, names of parameters to a given function and its local variables must all be pairwise distinct. There are no global variables.
- Each local variable must be defined by an assignment before it is used. This is checked by verifying that along all control flow paths originating at the beginning of the function, each local variable is assigned to before it is used. This ensures that there will be no references to uninitialized variables. A precise specification of this condition is given in Lab 2.
- Any variable, either a function parameter or locally declared variable, must have a small type. Variables of large type are not permitted. This simplifies parameter passing considerably.

The lack of bounds check and this language means that, unlike *L3*, there are programs in *L4* whose result is not defined. Such programs are considered to be "erroneous", although the compiler must permit them. For your test cases, you may not submit programs whose behavior is undefined.

Overloaded Operators

Equality (`==`) and disequality (`!=`) are overloaded, applying to both integers and pointers. See the discussion above on pointers. Operationally, the compiler will have to choose a 4 byte or 8 byte comparison for the two versions, which suggests after type-checking and during translation to intermediate code there should be two explicitly different operators.

5 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` 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 error`	program must fail to compile due to an *L4* source error

followed by the program text. If an exception number is missing, any exception is accepted. Defined exceptions are at least `SIGFPE` (8), `SIGSEGV` (11), and `SIGALRM` (14). These are raised by division by 0 or division overflow (8), dereference of 0, stack overflow, or out-of-memory situations (11), or by a time-out (14). All test files should be collected into a directory `test/` (containing no other files) and submitted via the Autolab server.

Your test programs must test the new features of *L4*: structs, pointers, and arrays. Since the language is backward compatible except for conflicts with reserved words, we may also use the *L3* test cases for regression testing. We cannot test programs whose results are undefined. It is therefore critical that your test programs do not perform any operations whose meaning is undefined. This includes accessing an array out of bounds or accessing a null array.

We would like some fraction of your test programs to compute "interesting" functions; please briefly describe such examples in a comment in the file. Disallowed are sorting programs.

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. This will be a crucial guide for the grader.

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.

Using the Subversion Repository

The recommended method for handout and handin is 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/15-411/<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 `15-411/<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

```
S5b - Autograde your code in svn repository
```

from the Autolab server menu. It will perform one of

```
% svn checkout https://cvs.concert.cs.cmu.edu/15-411/<team>/lab4/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15-411/<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 an compiled files or binaries in the repository!

Uploading tar Archives

A deprecated method for handout and handin is the download and upload of tar archives from the Autolab server.

For the test cases, bundle the directory `tests` as a tar file `tests.tar` with

```
% tar -cvf tests.tar tests/
```

to be submitted via the Autolab server.

For the compiler, bundle the directory `compiler` as a tar file `compiler.tar`. In order to keep the files you hand in to a reasonable size, please clean up the directory and then bundle it as a tar file. For example:

```
% cd compiler
% make clean
% cd ..
% tar -cvf compiler.tar --exclude CVS compiler/
```

to be submitted via the Autolab server. Please do not include any compiled files or binaries in your hand-in file!

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 5 second limit for testing compilers.

  Test cases are due **11:59pm on Tue Oct 21, 2008**.

- 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 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 Tue Oct 28, 2008**.

6 Notes and Hints

There is no address-of operator (`&`), and variables (be it local variables or procedure parameters) cannot hold large values. The suggested implementation therefore is to allocate structs and arrays

on the heap. Our runtime system provides the function `void* calloc(size_t nobj, size_t nbytes);` for this purpose, which allocates an array of `nobj` objects of size `nbytes` all initialized to 0. Your assembly code should call this function as necessary, assuming `size_t` is `unsigned int` (4 bytes).

There is no corresponding `free`, since we use a conservative garbage collector to reclaim storage.

To check that struct declarations are well-founded, we suggest a simple cycle-checking depth-first search which is in any case necessary to compute field offsets. Global tables to keeps struct sizes, field offsets, and the types of functions seem like an appropriate strategy.

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 the struct alignment requirements explained above. You may also read the Section 3.1.2 in the *Application Binary Interface* description available from the Resources page.

Also, the C calling conventions detailed in Lab 3 must still be obeyed. Since no large values can be passed, this remains relatively straightforward. Also as in Lab 3, functions declared with `extern` must retain their name, while all defined functions g must be exported as symbols `_l4_`g. This prevents any conflicts between a C standard library function and the *L4* name of a function. When a function `_l4_`g is declared with `extern`, there may not be a local function g to avoid ambiguity.

Run-time Environment

The tests will be run in the standard Linux environment on the lab machines; the produced assembly code must conform to those standards. We recommend the use of `gcc -S` to produce assembly files from C sources which can provide template code and assembly language examples.

We will link your code against some library functions you can find in the files `runtime/l4rt.h` and `runtime/l4rt.c`. This will allow us to test your adherence to the C calling conventions on the x86-64 architecture, and it will allow you to call some standard library functions, say, to print messages, painful though it may be in a language without strings. The functions we explicitly define can be declared in *L4* as

```
extern int printchar(c:int);  /* print c as ASCII character */
extern int printint(n:int);   /* print n in decimal */
extern int printhex(n:int);   /* print n in hexadecimal */
```

More may be available as the code is released and updated.

If your compiler detects any (compile-time) errors in the source program, it should exit with a non-zero return code. If compilation succeeds and target code is generated, the compiler should then exit with a return code of 0.

7 Changes

Revision 1. The compiler must check that any external symbol `_l4_`g must not conflict with an internal function g. This prevents confusion between the two symbols while permitting multiple *L4* files to be linked.

Revision 2. The type checker must reject `*NULL` because it can have any type, which leads to problems in type checking. So, for example, `(*NULL).f` might have been legal in the first spec, but type checking is unreasonable since multiple structs may have an `f` field. The new restriction can be checked, for example, by verifying that in $*e$, the expression e does not have type **any**$*$.