15-411 Compiler Design, Fall 2014
Lab 6 - Implementing C1

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

Tests due 11:59pm, Tuesday, November 25, 2014
Compilers due 11:59pm, Thursday, December 4, 2014
Papers due 11:59pm, Tuesday, December 9, 2014
Update 1, Mon Nov 17: Your compiler does not need to provide unsafe mode.
Update 2, Mon Nov 17: The library interface and data layout has been specified.

1 Introduction

The main goal of the lab is to explore compiling additional advanced language features of C0 and also in C1. In C0, we have characters and strings; in C1 we have `void*` and function pointers. We code-name the language L5 because it still lacks support for contracts and `#use` compiler directives (the latter being handle by command-line arguments as in L4). With a preprocessor that collects and inline compiler directives, your implementation should be able to compile all code from 15-122 and more, including generic implementations of the data structures we discussed.

2 The L5 Language

We have two new primitive types, `char` and `string`, and two new compound types, `void*` and function types.

2.1 Characters

Characters are a special type to represent components of strings. They are written in the form `'c'`, where c can be any printable ASCII character, as well as the following escape sequences `\t` (tab), `\r` (return), `\f` (formfeed), `\a` (alert), `\b` (backspace), `\n` (newline), `\v` (vertical tab), `\'` (quote), `\"` (doublequote), `\0` (null). The default value for characters is `\0`. Characters can be compared with `==`, `!=`, `<`, `<=`, `>=`, `>` according to their ASCII value, which is always in the range from 0 to 127, inclusively.

For the purpose of correct interaction with libraries, characters should be represented by a 1-byte word. They do not need to be aligned.

2.2 Strings

Strings have the form `"c1...cn"`, where $c_1, \dots, c_n$ are ASCII characters as above, including the legal escape sequences except for `NUL` (`\0`), which may not appear in strings. The double-quote character itself " must be quoted as \" so it is not interpreted as the end of the string. The default value for type `string` is the empty string `""`. Strings can not be compared directly with comparison operators, because in a language such as C the comparison would actually apply to the addresses of the strings in memory, with unpredictable results. Appropriate comparison functions are provided by the string library.

For the purpose of correct interaction with libraries, strings should be represented as `NUL`-terminated character sequences as in C, stored on the heap. Hence, strings are a *small type*, where a value of type string is the address of the string in memory. Strings are always write-only and not synonymous with character arrays as in C. Library functions support explicit conversion between string and character arrays.

2.3 Compiler Directives

Your compiler should ignore, as comments, `#use <lib>` or `#use "filename"` compiler directives. Instead, it should accept a command-line argument `-l <lib>` as in Lab 4.

2.4 Generic Pointers

We have a new form expression, a *cast*, which is used only in a very specific way.

```
<exp> ::= ... | (<tp>) <exp>
```

The form `(void*)e` casts the expression e of type $t\,*$ to be of type `void*`. Operationally, this new pointer references a pair consisting of a runtime representation of the type $t\,*$ (the *tag*) and the pointer value of e.

The second form `(t*)e` where $t \neq$ `void` casts an expression e of type `void*` to have type $t\,*$. If the tag agrees with the type $t\,*$, it strips off the tag and returns the underlying pointer of type $t\,*$. If the tags do not agree, an appropriate runtime exception (`SIGSEGV`) is raised and the program is terminated.

Casting does not affect the null pointer, which remains `NULL` and serves as the default value of type `void*`.

In unsafe mode, the casts only have significance at compile-time and no tagging or untagging will be performed, because it is assumed that the tag would match. This behavior is consistent with C. You do not need to implement unsafe mode since it creates difficulties with respect to the standard library.

Files in the `tests0/` directory provide some examples of correct and incorrect uses of casts.

2.5 Function Pointers

We add a new unary prefix operator `&` pronounced "*address of*", which can only be applied to functions and has the same precedence as other unary prefix operators such as `*`. We can dereference a function pointer and apply it to a sequence of arguments with a new form of function call.

```
<unop> ::= ... | &
<exp> ::= ... | (* <exp>) ( [<exp> (, <exp>)*] )
```

In order to use function pointers we need to be able to assign them types. For this purpose, we allow a particular idiomatic use of `typedef` which is consistent with but much more restrictive than C and declares a *function type name* `<fnid>` which occupies the same name space as (ordinary) type names.

```
<gdefn> ::= ...
          | typedef <tp> <fnid> ( [<tp> <vid> (, <tp> <vid>)*] ) ;
<tp> ::= ... | <fnid>
```

Note that this is exactly the same form as a function declaration (also called a *function prototype*) preceded by the `typedef` keyword.

Function types, named by a `<fnid>` are large types and, moreover, function values cannot be allocated on the stack or heap. That is, we store and pass only pointers to functions, not functions themselves. Function type names are treated *nominally*, which means that two distinct function type names are considered different, even if their definitions happen to be the same.

Files in the `tests0/` directory provide some examples of correct and incorrect uses of function pointers.

2.6 Library Interface

Several of the standard C0 libraries, namely `conio`, `string`, and `file`, together with the earlier L4 library called 15411 are collected into `15411c1`. So when we call you compiler we will supply `-l 15411c1.h0`.

Because the library now contains characters, strings, and arrays, the layout of certain data structures must be more precisely specified than in Lab 4. We have:

C0	C
`int`	`int` (4 bytes)
`bool`	`bool` (1 byte, from `stdbool.h`)
`char`	`char` (1 byte)
`string`	`char*` (`NUL`-terminated)
`t[]`	(see below)

Arrays are represented as pointers to the beginning of a struct

```
struct c0_array_header {
  c0_int count;
  c0_int elt_size;
};
```

which is immediately followed in memory by an array of `count * elt_size` bytes. Please see the library implementation in `15411c1.c` for more detail.

If you find this representation inconvenient, you may substitute your own definition of `15411c1.c`. However, you cannot substitute `15411c1.h0`, since it is used to type-check L5 sources. For this reason, you must hand in a file `15411c1.c` file in your `compiler/` directory (which could just be a copy of the one we supply in the `runtime/` directory.

3 Requirements

You are required to hand in three separate items:

1. Additional test cases that explore the novel features of L5,
2. the working compiler and runtime system for L5,
3. a term paper describing and critically evaluating your project.

3.1 Tests

The tests should be concerned with verifying that your compiler is correct on characters, strings, generic pointers, and function pointers.

3.2 Compilers

Your compilers should treat the language L5 as described below. You need only implement safe mode. Unsafe mode is optional and may require a separate implementation of the library.

3.3 Term Paper

Your paper should follow this outline.

1. Introduction. This should provide an overview of your implementation and briefly summarize the results you obtained.
2. Compilation. Describe the data structures, code, and information generated by the compiler in order to support the new language features.
3. Analysis. Critically evaluate the language, your compiler and runtime system and sketch future improvements one might make to its design.

The term paper will be graded. There is no hard limit on the number of pages, but we expect that you will have approximately 5-10 pages of reasonably concise and interesting analysis to present.

4 Deadlines and Deliverables

Your test cases and compilers must be committed into `lab6c1` directory in the same way that you submitted your tests and compilers in previous assignments.

4.1 Test Files (due 11:59pm on Tue Nov 25)

You should submit at least 20 test cases in a directory `tests/` named `$name.l5` that explore the new language features. Your compilers will be tested on the tests submitted by other teams implementing this option for Lab 6, and all the collected test cases from Labs 1–4.

Note that the compiler for C0 available on Andrew will not be able to handle `*.l5` files. You should be able to use the compiler at

```
/afs/cs.cmu.edu/academic/class/15411-f14/bin/cc0
```

to validate your test cases.

4.2 Compiler Files (due 11:59pm on Thu Dec 4)

All files should be collected in a directory `compiler/` which should contain a `Makefile` and a library file `15411c1.c`. **Important:** You should also update the `README` file and insert a roadmap to your code. This will be a helpful guide for the grader.

Issuing the shell command

```
% make l5c
```

should generate the appropriate files so that

```
% bin/l5c <args>
```

will run your L5 compiler with switches as for Lab 5 The command

```
% make clean
```

should remove all binaries, heaps, and other generated files.

4.3 Term Paper (due 11:59pm on Tue Dec 9)

Submit your term paper in PDF form via Autolab before the stated deadline. Early submissions are much appreciated since it lessens the grading load of the course staff near the end of the semester. **You may not use any late days on the term paper describing your implementation of Lab 6!**