15-411 Compiler Design, Fall 2016
Lab 6 - C0 and Beyond

Jan and co.

Tests due 09:00am, Thursday, December 1, 2016
Due 09:00am, Thursday, December 8, 2016
Papers due 09:00am, Tuesday, December 13, 2016

1 Introduction

The main goal of the lab is to explore advanced aspects of compilation. This writeup describes the option of extending your L4/L5 compiler with new language features; other writeups detail other options. The basic languages you deal with in this lab are called L5 and L6 instead of C0 or C1 because they may still lack support for some contracts and for `#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 almost all code from 15-122 and more.

2 The L5 Language

The point of this section is to establish a common baseline for all language extension projects by implementing two new primitive types, `char` and `string`. Both should be treated as *small 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.

Characters should be represented in memory as a 1-byte word. They do not need to be aligned.

2.2 Strings

Strings have the form `"c1...cn"`, where $c_1, \ldots, c_n$ 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. Unlike C strings, L5 strings cannot be modified.

You should put some thought into how strings are represented at runtime. The C library implementation you are given (see below) represents string values as pointers to `NUL`-terminated character sequences, as in C. You can change the implementation of strings if you change the corresponding runtime. You should also be thoughtful about how you deal with constant strings. Ideally these should be allocated in the text part of your compiled binary; check out the way how `gcc` handles strings. The choices you make should be documented in your project report.

2.3 Library Interface

Several of the standard C0 libraries, namely `conio`, `string`, and `file`, together with the earlier library, are collected into the runtime; when we call you compiler we will supply the command-line arguments `-l ../runtime/15411-l5.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*` (8 byte pointer to `NUL`-terminated string)
`t*`	(8 bytes)
`t[]`	(8 bytes, see below)

Arrays are represented as described in lecture: as a pointer a to the first data element of the array, where the number of elements in the array are stored in the four bytes beginning at the address $a - 8$. Please see the library implementation in `run411.c` for more detail. Note that, because the library includes array-bounds checks, it can only be used with safe mode. As a consequence, you are not required to implement the `--unsafe` flag for this assignment. If you do implement the `--unsafe` flag, it will need to be compiled against a different runtime.

You may include a different runtime and have your compiler link against that runtime instead. However, you must support `15411-c1.h0`, since it is used to type-check L5 sources.

3 The L6 Language

Simply implementing the L5 language is not sufficient for passing the final project. The point of this section is to describe how to build on the L5 language to make a compelling final project.

You are expected to follow the outline described in **one** of the sections below. Each approach contains a basic extension that, if done well, will give you a passing grade on the final project, as well as some suggestions on how to approach doing something extra. You are expected do something extra to get full credit on the final project.

Any one of these projects requires you to make decisions at every level of the language: the static and dynamic semantics, the compilation strategies, and potentially the runtime. Your designs and design decisions will need to be carefully documented and critically examined in your term paper.

3.1 Contracts

Contracts are an important part of C0 that does not exist in any of the languages discussed in this class. Contracts exist within either multi-line declarations `/*@ ... @*/` or single line declarations that begin with `//@` and end with a newline character. Lexing is modified within contracts: the `@` character is treated as whitespace. Aside from that, contracts have the following forms:

```
<anno>  ::= //@ <spec>* \n
            /*@ <spec>* @*/
<spec>  ::= requires <exp> ;
          | ensures <exp> ;
          | loop_invariant <exp> ;
          | assert <exp> ;
<stmt>  ::= ... | <anno>+ <stmt>
<exp>   ::= ... | \result | \length ( <exp> )
<gdecl> ::= ...
          | <type> ident <param-list> <anno>+ <block>
```

This grammar introduces an ambiguity, because `/*@ s1 @*/ /*@ s2 @*/ s3` could either be:

- a twice-annotated statement, same as `/*@ s1 s2 @*/ s3`
- an annotated statement with an annotation, same as `/*@ s1 @*/ { /*@ s2 @*/ s3 }`

This is always resolved in favor of having a twice-annotated statement. Furthermore, the following restrictions apply:

- `loop_invariant` contracts are the only contracts that can annotate a loop body.
- `assert` contracts are the only contracts that can annotate other statements.
- `requires` and `ensures` contracts can only appear immediately before the initial "`{`" token in a function definition.
- The expression `\result` can only appear within a `ensures` contract; it represents the value of the function upon exit.
- The expression `\length(e)` can only appear in contracts. It takes an array as its argument, and allows contract code to access the length of an array.

Compared to C0, you probably do not want to support annotations on function declarations. Files in the `testsC/` directory describe provide some examples of correct and incorrect uses of casts.

A runtime flag, `-d`, means that all contracts are turned into assertions in the compiled code. When this flag is omitted, all contracts are still typechecked, but they are removed from the code after typechecking.

In addition to supporting contracts, you are expected to do something extra. Here's a couple of examples of what an project based on this extension might look like:

- Extend the language with new syntax designed to be used in contracts (possibly only in contracts) and give examples to show that it is useful. One example: it is not possible to write the loop invariant "all $A[i]$ for $0 \leq i < n$ are less than x" without writing a helper

function. Array comprehension syntax might help with this. Another example: it is not possible to write a contract in C0 that checks that an in-place sorting function returns a permutation of the original array. Can you design a language extension that addresses this?

- Implement a variant *purity checking*, which ensures that contracts will not modify memory that can be read outside of the contract. You will probably need your purity checker to reject many programs that the reference compiler accepts; your report should discuss and analyze any differences you find.

- Develop a "semi-safe" mode that assumes that all contracts succeed and uses that assumption to perform optimizations (like removing array bounds checks) based on the presence of contracts, but then does not check those contracts at runtime. Demonstrate performance improvements over safe mode for examples where you can prove (on paper) that all array bounds checks succeed.

3.2 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, a memory error 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*`. Files in the `testsG/` directory provide some examples of correct and incorrect uses of casts.

In unsafe mode (if you support it), 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. In order to support unsafe mode, it should be impossible for the programmer to learn what tag something has, just like it is not possible to check the length of an array.

In addition to supporting generic pointers, you are expected to do something extra. Here's one example of what an project based on this extension might look like:

- The 15-122 take on object oriented programming requires both generic pointers and function pointers, so that structs can contain both data and instructions for how to interpret that data.

  Another approach to object-oriented programming is to allow functions to be associated with particular pointer or struct types. Then, if we call the `hash` function on a generic pointer that was originally an `struct rope*`, we can look up, at runtime, to see how ropes are supposed to be hashed. If there's no `hash` function associated with `struct rope` pointers, this should cause a memory error.

  The really interesting feature of this approach is that it's possible for a hash table implementation to hash generic pointers that it contains without ever knowing what tag those generic pointers have.

Design a reasonable syntax, semantics, and implementation of generic data structures that uses these ideas. Implement a generic hash table.

3.3 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 in C1 are treated *nominally*, which means that two distinct function type names are considered different, even if their definitions happen to be the same. You can support this or not, but you should explain your decision in your project report.

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

In addition to supporting function pointers, you are expected to do something extra. Here's one example of what an project based on this extension might look like:

- In addition to the `&` syntax for creating function pointers, allow anonymous functions to be created by defining a new form of expression `fn ( [<tp> <vid> (, <tp> <vid>)*] ) <block>` that evaluates to a function pointer. These anonymous functions should be allowed to refer to temps in the function(s) that they are declared within, which means that the runtime value of a function will need to be a *closure* containing a function pointer, instead of just a function pointer.

4 Requirements

You are required to hand in three separate items:

1. Additional test cases that explore the novel features of L5/L6,
2. the working compiler and runtime system for L5/L6,
3. a term paper describing and critically evaluating your project.

4.1 Tests

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

4.2 Compilers

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

4.3 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. Extensions. Carefully describe the syntax and semantics of the extensions to your language.
3. Compilation. Describe the data structures, code, and information generated by the compiler in order to support the new language features.
4. Examples. Walk through several code examples that demonstrate the power of your language extensions.
5. 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.

5 Deliverables and Deadlines

All your code should be placed in subdirectories of the `lab6` directory as before. Be sure to push to a branch named `lab6c1`. The autograder will run regression tests against your own tests, the same tests it used in Lab 5, and L5 tests submitted by other groups doing the same project. These tests will *not* directly contribute to your grade. We will grade you based on the code and `README` file(s) you have checked in at the deadline.

Test Files (due 09:00am on Thu Dec 1)

In a directory `lab6/tests/`, you should turn in at least 10 tests named `$name.l5` that deal with characters, strings, and the new libraries, at least 10 tests named `$name.l6` that deal with the basic L6 extensions to your language, and at least 10 tests named `$name.l6` that deal with the L6 extensions that you designed.

It is okay if you have to modify your L6 tests as you work on your compiler. If you need to change anything besides a typo, make sure to mention it in your project report.

Compiler Files (due 09:00am on Thu Dec 8)

As for all labs, the files comprising the compiler itself should be collected in the `lab6/` directory which should contain a `Makefile`. **Important:** You should also update the `README` file and insert a roadmap to your code. This will be a helpful guide for the grader. In particular, since there are likely to be many different projects undertaken, do introduce the project at the very top of the README.

Issuing the shell command

```
% make
```

should generate the appropriate files so that

```
% bin/c0c --exe <args>
```

will run your compiler and produce an executable. It is not necessary to continue supporting any compiler flags besides `-t`.

After running `make`, issuing the shell command

```
% make test
```

should run your own tests and print out informative output. The command

```
% make clean
```

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

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.

Runtime environment (due 09:00am on Thu Dec 8)

Because you may have implemented a new runtime, your compiler should have an additional flag `--exe`. If your compiler is given a well-formed input file `foo.l1` or `foo.l2` as a command-line argument and is also given the `--exe` argument, it should generate a target file called `foo.l1.s` or `foo.l2.s` (respectively) in the same directory as the source file, and should *also* compile the runtime and link it with your generated assembly to create an executable `foo.l1.exe` or `foo.l2.exe` (respectively). We will test your compiler against the regression suite, but the grade reported by the autograder won't directly contribute to your grade.

Term Paper (due 09:00am on Tue Dec 13)

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!**