15-411 Compiler Design: Lab 5
Fall 2008

Instructor: Frank Pfenning
TAs: Rob Arnold and Eugene Marinelli

Test Programs Due: 11:59pm, Thursday, November 6, 2008
Compilers Due: 11:59pm, Thursday, November 13, 2008
Papers Due: 11:59, Friday, November 14, 2008

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L5* and also implement a first set of basic optimizations. This language extends *L4* in only minor ways, but its dynamic semantics is now *safe* in that attempts to access an array out of bounds must result in an exception.

2 Requirements

As for the earlier labs, you are required to hand in test programs as well as a complete working compiler that translates *L5* source programs into correct target programs written in x86-64 assembly language. In addition you have to submit a PDF file via email to the instructor (`fp@cs`), which describes and evaluates the optimizations you implemented.

3 *L5* Syntax

The syntax of *L5* is defined by the context-free grammar in Figure 1. This is an extension of the grammar for *L4* in the sense that a correct *L4* program should still parse correctly, except where the new token `--` is used. 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 *L4*.

The changes to *L4* are as follows:

- There are now hexadecimal constants, starting with `0x`. Hexadecimal constants n must be in the same range, $0 \leq n < 2^{32}$ as decimal constants.
- There are now conditional expressions, e_1 `?` e_2 `:` e_3.
- There are two new forms of simple statements, e`++` and e`--`.

⟨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⟩ `++` | ⟨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⟩ | ⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩
⟨ident⟩ **(** [⟨exp⟩ [**,** ⟨exp⟩]*] **)** | **new (** ⟨type⟩ [[⟨exp⟩]] **)**

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

⟨intconst⟩ ::= [**0-9**][**0-9**]* | **0x**[**0-9a-fA-F**][**0-9a-fA-F**]* (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 *L5*

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	conditional expression
`= += -= *= /= %=` `&= ^= |= <<= >>=`	right	assignment operators

Figure 2: Precedence of operators, from highest to lowest

4 New *L5* Constructs

Hexadecimal constants are explained above.

Conditional Expressions

Both branches of a conditional expression may have the same arbitrary type τ.

$$\frac{e_1 : \textbf{int} \quad e_2 : \tau \quad e_3 : \tau}{e_1 \; ? \; e_2 : e_3 : \tau}$$

The operational meaning is as in C.

We also extend the rule on disallowing dereference of the null pointer: an expression $*e$ is ill-typed if e has type τ^* for all types τ. Previously, this could only happen if $e = \textbf{NULL}$, but now e could also be conditional were both branches or null, or themselves again conditionals with null branches, and so on.

Increment and Decrement Statements

The simple statement e`++` is the same as e `+=` 1 and e`--` is the same as e `-=` 1. In particular, e must be a legal lvalue in both statements. The fact that these statements are simple means they can appear as the initial or step statement of a for loop.

5 *L5* Safety Requirements

The behavior of *L4* was purposely unspecified in certain situations. When the *L5* compiler is called with the `--unsafe` switch this behavior remains undefined. This is also the default, for backward compatibility.

When the *L5* compiler is called with the `--safe` switch, the unspecified behavior is now explicated as outlined below.

Memory-related exceptions must be signaled as the exception 11 (`SIGSEGV`). We refer to this as *illegal memory reference*.

Pointers

As in *L4*, derefencing the null pointer is an *illegal memory reference*. Since the page at address 0 is read/write protected by the operating system, attempting to read from 0 will yield the appropriate exception.

Structs

When the address of a struct is computed as 0, an *illegal memory reference* must be reported. This may happen when computing `*p` for a pointer `p` declared with `var p : s*` for a struct `s`. In this case we must signal an exception instead of basing address calculations for fields on 0, because with a sufficiently large offset we may skip past the read/write protected pages at the beginning of memory.

Arrays

If an array A was allocated with `new(`τ`[`n`])` for a non-negative n, any attempt to access $A[i]$ where $i < 0$ or $i \geq n$ must generate an *illegal memory reference*.

Similarly to structs, when the address of an array is computed as 0, an *illegal memory reference* must be reported. This may happen when computing `*p` for a pointer `p :` τ`[]*`.

Allocation

When allocation of a pointer or an array fails, an *illegal memory reference* must be reported. Allocation with `new(`τ`[`n`])` for $n < 0$ must fail; other allocations may or may not fail depending on current resource constraints. In any case, `new` is not allowed to return the null pointer.

6 Array Layout

In order to be able to call C libraries, your code must strictly adhere to the convention specified in the Application Binary Interface (ABI) for the x86-64 and C. This is exactly as in *L4* (data alignment, struct layout, calling conventions, stack pointer alignment), except that special considerations are necessary when compiling to safe code.

In order for your code to be able to check whether array access is in bounds, it must store with each array the number of elements of the array. This should be done as follows: if the data elements in an array start at address a, then the (non-negative) integer n recording the number of elements in the array is stored at $a-8, a-7, a-6, a-5$. The bytes at $a-4, \ldots a-1$ are padding, to simplify alignment (the strictest alignment requirement in our language for data is 0 mod 8, and `calloc` returns memory aligned in this manner).

This layout will allow *L5* to correctly call C functions with array addresses in arguments, whether we are using safe or unsafe mode. Arrays generated by C cannot be returned to *L5* in safe mode because their size is in general unknown. Special wrapper code is needed in this case, which may be different for different libraries.

7 Optimizations

In addition to the minor language extension and safe compilation option, you are also required to implement and describe some basic optimizations. Choose 3 from the following menu.

- **Instruction selection.** You may optimize instruction selection to generate more compact or faster code. This includes generating good code for conditions (e.g., avoiding `set` instructions) or loops (e.g., enabling good branch prediction or aligning jump targets), and other improvements on your code.
- **Constant propagation and folding.** Implement constant propagation together with constant folding and eliminating constant conditional branches.
- **Dead code elimination.** Implement dead code elimination using the analysis described in Lecture 5.

- **Eliminating register moves.** Explore techniques for eliminating register moves such as improved instruction selection, copy propagation, register coalescing, and peephole optimization. Register coalescing should be run as a single pass after register allocation as suggested by Pereira and Palsberg and the notes to Lecture 3.
- **Common subexpression elimination.** Implement common subexpression elimination, with or without type-based alias analysis to avoid redundant loads from memory.
- **Other optimizations.** If there is a particular optimization you would like to implement that is not in the above list you may wait until the next lab or contact the instructor with a proposal.

If you have already implemented some of these optimizations, you may revisit and describe them, perhaps improve them further.

You are required to write a short paper of 3–5 pages, to be submitted via email to the instructor as a PDF file. This paper should describe each optimization, how you implemented it, any heuristics you developed for its application, and whether you found it to be effective both on contrived examples and the provided benchmark suite.

8 Regression Testing

As you are implementing optimizations, it is extremely important to carry out regression testing to make sure your compiler remains correct. We are providing a directories `regression/tests*` with the handout, assembled from tests for labs 2, 3, and 4. This may or may not be checked by Autolab upon hand-in, so you should make sure to apply this frequently as you proceed with optimizations. If regression testing is not performed automatically, we will apply it by hand during instructor evaluation.

9 Project Requirements

For this project, you are required to hand in test cases, a complete working compiler for *L5* that produces correct target programs written in Intel x86-64 assembly language, and a description and assessment of your optimizations. 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 `.l5` 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), illegal memory references (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 should be in two categories: (1) those that test the new features of *L5*, namely hexadecimal constants, conditional expression, increment and decrement statements, and safe compilation, and (2) those that test your optimizations.

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 l5c
```

should generate the appropriate files so that

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

will run your *L5* 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 `lab5` 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>/lab5/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15-411/<team>/lab5/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. 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 within 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 Thu Nov 6, 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 Thu Nov 13, 2008**.

- The description of the optimizations. The PDF file of 3–5 pages should describe your optimizations and assess how well they worked in improving the code, over individual tests and the benchmark suite. It should be emailed to the instructor, `fp@cs`.

  Papers are due **11:59 on Fri Nov 14, 2008**.