# 15-411 Compiler Design: Lab 5
# Fall 2010

Instructor: Andre Platzer
TAs: Anand Subramanian, Nathan Snyder

Test Programs Due: 11:59pm, Thursday, November 4, 2010
Checkpoint 1 Due: 11:59pm, Tuesday, November 9, 2010
Final Compiler and Paper Due: 11:59pm, Tuesday, November 16, 2010

## 1 Introduction

The goal of the lab is to implement a complete compiler for the language *L4* and also implement a first set of basic optimizations. The language itself is unchanged from Lab 4, but its dynamic semantics is now *safe* in that certain operations such as attempting to access an array out of bounds must result in an exception.

## 2 Preview of Deliverables

As for the earlier labs, 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. In addition you have to submit a PDF file via email to the course staff at (`15411@symbolaris`), which describes and evaluates the optimizations you implemented.

## 3 Safety Requirements

The behavior of *L4* was purposely unspecified in certain situations. When the Lab 5 compiler is called with the `--unsafe` switch this behavior remains undefined. This is also the default, for backward compatibility.

When the Lab 5 compiler is called with the `--safe` switch, the unspecified behavior is now explicated as outlined below.

There are two memory-related errors that should be reported to the user:

1. *illegal memory references.* For the purpose of automated testing, the program must raise exception 11 (`SIGSEGV`)
2. *array bounds error.* For the purpose of automated testing, the program must raise exception 6 (`SIGABRT`)

### Pointers

As in Lab 4, 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 `alloc_array(`$\tau$`, `$n$`)` for a non-negative $n$, any attempt to access $A[i]$ where $i < 0$ or $i \geq n$ must generate an *array bounds error*.

Similarly to structs, when the address of an array is computed as 0, an *illegal memory reference* must be reported.

### Allocation

When allocation fails, an *illegal memory reference* must be reported. Allocation with `alloc_array(`$\tau$`, `$n$`)` for $n < 0$ must fail with an *array bounds error*; other allocations may or may not fail depending on current resource constraints. Your test cases should use a moderate amount of memory so that one would not expect allocation to fail due to resource constraints. In any case, `alloc` is not allowed to return the null pointer.

## 4 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 Lab 4 (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 the implementation 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 *L4* 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.

## 5 Optimizations

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

1 **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.

2 **Constant propagation and folding.** Implement constant propagation together with constant folding and eliminating constant conditional branches.

3 **Dead code elimination.** Implement dead code elimination using the analysis described in the lecture notes.

4 **Eliminating register moves.** Explore techniques for eliminating register moves such as improved instruction selection, copy propagation, register coalescing, and peephole optimization. We suggest coalescing registers in a single pass after register allocation as suggested by Pereira and Palsberg and the notes to Lecture 3.

5 **Common subexpression elimination.** Implement common subexpression elimination, with or without type-based alias analysis to avoid redundant loads from memory.

6 **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.

You may choose one of the two following categories to determine which optimizations and how many you wish to implement.

A Implement any three optimizations from the menu as described in lecture. You may prefer this option if you wish to implement a greater number of optimizations as simple extensions to your compiler, and observe the effect that these transformations have on the quality of compiled code.

B Use an Intermediate Representation based on Static Single Assignment form.

- You may transform your code to SSA form using either the more common dominance frontier algorithm, or the value numbering algorithm presented in lecture.
- After reaching SSA form, apply at least one optimization from items 2, 3 or 5.
- Finally, eliminate $\phi$ nobes at or prior to register allocation. Remember that the chordal graph allocator provably finds an optimal coloring for programs in SSA form. Reducing the number of moves generated during unSSA is good, but correctness comes first!

You may prefer this option if you are looking for an extra challenge, and if you are planning ahead to add more optimizations in lab 6.

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

Your compiler must take a new option, `-O`*n* (dash capital-O), where `-O0` means no optimizations, `-O1` performs some optimizations, and `-O2` performs the most aggressive optimizations. Part of our evaluation may be based on the performance improvements you achieve.

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.

## 6 Regression Testing

As you are implementing optimizations, it is extremely important to carry out regression testing to make sure your compiler remains correct. The optimizations must be valid for the safe memory semantics. All tests from previous labs are fair game. If regression testing is not performed automatically, we will apply it by hand during instructor evaluation.

## 7 Deliverables and Deadlines

For this project, you are required to hand in test cases, a complete working compiler for *L4* 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 Cases

We require both of the following:

- At least 8 test cases that test the behaviour of safe memory. At least two of these must be expection tests. Since the static semantics has not changed from lab 4, you are not required to hand in any error test cases. However, if you notice something that the previous test suites failed to test for, feel free to add error test cases.

- At least 6 test cases that are interesting from the point of view of optimizations. Do not design these as stress tests which test the runtime of the compiler. Instead, make them interesting with respect to the quality of code produced. These tests may be assembled into a suite of benchmark tests.

  Keep in mind that optimizations are generally designed to benefit the common case of programs, not pathological cases or uncommon programming patterns. For instance, having a program full of dead code is not particularly interesting, though it might help you sanity test dead-code-elimination.

At the checkpoint, your compiler must be able to compile and run these within the time limit with `-O0`. The compiled binary for each test case should run within 1 second with the reference compiler on the lab machines; we will use a 5 second limit for testing student compilers.

At the final handin, your compiler must be able to compile and run the tests within the time limit with `-O2`. Please plan for the fact that compilers are not necessarily required to implement SSA, and even if they do, not necessarily using a fast algorithm. Therefore, as interesting as your test cases may get, they should allow compilers to run optimizations that run in quadratic time with respect to the number of branches in the program, or quadratic with respect to the number of temporaries.

Test cases are due **11:59pm on Thu Nov 04, 2010**.

## Checkpoint 1: Compiler with Safe Memory Semantics

Compilers that successfully implement safe memory semantics are due **11:59pm on Tue Nov 09, 2010**. We don't expect that safe memory semantics to take a very long time to implement. However, there ought to be enough time before the checkpoint to restructure the compiler if required, and enough time after it to implement a few optimizations.

This is a GRADED CHECKPOINT.

## Final Compiler and Project Report

Compilers that implement optimizations are due **11:59 on Tue Nov 16, 2010**. The sources should be handed in through SVN as usual, and must contain documentation that is up to date. This submission will also be automatically tested as usual for correctness. Remember that optimizations are required to preserve memory safety.

The project report should be a PDF file of approximately 2–5 pages, and should be emailed to the course staff at `15411@symbolaris` The report should describe your optimizations and assess how well they worked in improving the code, over individual tests and the benchmark suite.

If you use algorithms that have not been covered in class, cite any relevant sources, and briefly describe how they work. If the algorithms have been covered in class, cite the appropriate lecture notes, and do not waste any space on describing the optimizations. The interactions of the optimizations with each other and the effect of optimizations on the produced code should be given adequate treatment.