15-411 Compiler Design: Lab 4
Fall 2007

Instructor: Frank Pfenning
TAs: David McWherter and Noam Zeilberger

Test Programs Due: 11:59pm, Tuesday, November 6, 2007
Compilers Due: 11:59pm, Tuesday, November 13, 2007
Late handins until 11:59pm, Thursday, November 15, 2007 without penalty
Revision 1: Fri Nov 9, 2007

1 Introduction

The main goal of the lab is to implement another compiler for the language *L3* which generates *safe* code. This means that in situations where the behavior of an *L3* program (according to the specification from Lab 3) would be undefined, the new compiler must generate a runtime exception. We refer to this language as *safe L3*. In addition, your implementation must be able to reliably interact with C, which means it must be possible for *L3* code to call C libraries and C code to call *L3* libraries. This should work for code compiled both in safe and unsafe modes.

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 *L3* source programs into correct target programs written in x86-64 assembly language. Your compiler will take a switch as an argument to compile in safe or unsafe mode. 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. When encountering a runtime error, the program should terminate with an exception; no error message needs to be printed. 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 Extended *L3* Syntax

The syntax of *L3* is extended only minimally from Lab 3 in that it allows the declaration of function prototypes at the beginning of a file. We only show the part of the language that has changed in Figure 1. This means there is one new reserved keyword, `extern`, which cannot be used as an identifier.

4 *L3* Safe Static Semantics

The static semantics remains unchanged from unsafe *L3* with the exception of the introduction the new reserved keyword `extern`. In particular, your compiler must accept even patently unsafe

$$
\begin{array}{lcl}
\langle\text{program}\rangle & ::= & \langle\text{gdecl}\rangle^* \ \langle\text{function}\rangle^* \\
\langle\text{gdecl}\rangle & ::= & \textbf{struct}\ \langle\text{ident}\rangle\ \textbf{\{}\ [\ \langle\text{ident}\rangle\ \textbf{:}\ \langle\text{type}\rangle\ \textbf{;}]^*\ \textbf{\}}\ \textbf{;}\ \ | \\
 & & \textbf{extern}\ \langle\text{type}\rangle\ \langle\text{ident}\rangle\ \textbf{(}\ \langle\text{paramlist}\rangle\ \textbf{)}\ \textbf{;}
\end{array}
$$

Figure 1: *L3* Grammar Extension

programs, or those that are guaranteed to raise a runtime exception, but it may produce warnings if you feel inclined to do so.

Functions declared with

`extern` τ g `(`x_1`:`τ_1`,...,`x_n`:`τ_n`);`

can be used inside the file at the given types, but there may not be any further declarations or definitions of these functions in the file. It will be the responsibility of the linking process to provide definitions for these functions at the specified types. Since the linker does not actually check the types, and the language we link again may be inherently unsafe, the safety of *L3* should be understood conditionally: if the externally supplied implementations of the external functions are safe and obey the declared types, then the overall program will be safe.

To facilitate simpler interoperability with C, any function g not declared with `extern` will be renamed to `_l3_`g during the compilation process before linking. This means the *main* function in the *L3* source now needs to be called `main` instead of `_l3_main`, in contrast with Lab 3.

5 *L3* Safety Requirements

The behavior of unsafe *L3* is purposely undefined in a number of situations. In safe *L3* we specify the behavior for these situation.

Initialization

Recall that integer variables are already required to be initialized to 0. Initial values for variables of other types, and data structures explicitly allocated with `new`, are now required to be set as described below.

- Uninitialized variables of type τ`*`. These must now be initialized to `NULL`. This means that any attempt to dereference an uninitialized pointer will generate an *invalid memory reference* exception (`SIGSEGV` signal).
- Uninitialized variables of type τ`[]`. These must be initialized to some value so that any attempt to access an array element will generate an *invalid memory reference* exception.
- Allocated, but uninitialized pointers. These must be initialized to `NULL`.
- Allocated, but uninitialized arrays. Each element of an array must be initialized according to its type.
- Allocated, but uninitialized structs. Each component of a struct must be initialized according to its type, including any embedded structs.

Pointers

Any attempt to dereference the `NULL` pointer must generate an *invalid memory reference* exception. This will be translated into a `SIGSEGV` signal, often printed as "segmentation violation". Your program should exit with the corresponding code of `11` for the lower `7` bits of the exit code.

The operating system guarantees that attempting to access memory location `0` will generate an exception of the kind required.

This specification means that it is no longer legal to optimize, for example, `*p == *p` to `1` unless your compiler can prove that `p` cannot be `NULL` at this program point.

Structs

Structs cannot be passed or assigned to variables, but addresses of structs can be computed. When the computed address a of a struct is `NULL` an *invalid memory reference* exception should be raised, even if memory may not necessarily be accessed at that time. This is to robustly fail for offset calculations based on a, which could lead to addresses whose access may not be guaranteed to raise an exception when the offset is large enough. This also means that an expression such as `*p`, when used as a statement for side effects, should yield an error when `p` is `NULL` even when the type of `p` is large and therefore may not entail a memory access according the operational semantics of *L3*.

Arrays

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

If an attempt is made to allocate an array with `new(`τ`[`n`])` where $n < 0$, a runtime exception must be raised. For simplicity, we use the same *invalid memory reference* exception as for other memory-related errors.

Resource Bounds

Your program might be forced to terminate for a number of reasons, such as exceeding the time or stack limits, or receiving an explicit signal. You do not need to check for or handle such unexpected termination conditions.

The only exception is allocation: if your call to the runtime system for allocating memory fails by returning an address of 0, your program should always raise an *invalid memory reference* exception.

6 Interoperability with C

In order to be able to call C libraries and vice versa, your code must adhere strictly to the a number of conventions specified in the *Application Binary Interface* (ABI). Specifically:

- Data alignment and struct layout constraints must be strictly observed, including those on stack pointer alignment.
- Calling conventions, including caller- and callee-save registers, argument passing, and return registers must be observed. Note that much of the specification does not apply since complex data (such as structs) cannot be passed to or from functions directly.

One question is how data types are translated between the two languages. The mapping is straightforward except for arrays. First, when *L3* is compiled in unsafe mode the following table relates *L3* types τ to their C counterparts $\hat{\tau}$.

L3	C
`int`	`int`
τ`*`	$\hat{\tau}$`*`
τ`[]`	$\hat{\tau}$`*`
s	`struct` s

In the case of a struct, the fields types are translated correspondingly.

In safe mode the matter is slightly more complex, because with an array we must keep some runtime information on its size. For every *L3* array τ`[]` which has been allocated with n elements, there is a corresponding struct type in C which captures the required layout.

`struct { int num; int padding;` $\hat{\tau}$ `data[`n`]; }*`

Here, the `num` field must be n and the `data` field contains the actual array data. The `padding` is there to make sure that the data are properly aligned (no matter what type they are), assuming the struct itself is aligned at 0 modulo 8.

Note the trailing star which indicates that what is actually passed between C and safe *L3* is the address a of data layed out in this manner, from which the number of array elements can be recovered at offset 0 (address a) and the array data at offset 8 (address $a + 8$).

All functions g that are not declared as `extern` should be mapped to assembler symbols `_l3_`g. This avoids accidental conflicts between *L3* functions and functions or other symbols in the standard C library or other libraries your code is linked against.

You can also define C functions in a separate file and link your *L3* compiled code against them as a library. This way you can do input or output in order to test your *L3* code. The simplest way to achieve this with the provided driver is to include a reference to your C sources of the library functions in `l3rt.h` and `l3rt.c`. However, please keep in mind that the Autolab grader will not have your C sources, so you should not hand in test code with reference to external files except for those provided in the standard runtime files.

You can see from the provided files `l3rt.h` and `l3rt.c` which files we will link against the assembly language you produce. You may explicitly call any functions there as long as the types are both available and compatible with corresponding *L3* types. These may be different in safe and unsafe modes. And don't forget that any external function you call in your *L3* source must be declared with an appropriate prototype.

7 *L3* Runtime Objects

The runtime representation of safe *L3* objects should be the same as for unsafe *L3* except for arrays.

For interoperability, 0 should represent the `NULL` pointer and also the contents of an array variable which has not been allocated. This means the initialization of all variables should be the value 0, although at different sizes.

Consequently, memory allocated by a call to `new` should be initialized with all zeroes, which can be achieved easily by calling `calloc` instead of `malloc`.

Arrays should be laid out as indicated in the previous section via a corresponding C struct declaration. An alternative method whereby the *L3* address of an array is the start of the actual data and the size is to its left may sometimes be more convenient, but makes calling to and from C in the safe version harder.

8 Project Requirements

For this project, you are required to hand in test cases and a complete working compiler for *L3* 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 `.l3` and start with one of the following lines

`#test return` *i*	program must execute correctly and return *i*
`#test exception`	program must compile but raise a runtime exception
`#test error`	program must fail to compile due to a *L3* source error

followed by the program text. All test files should be collected into a directory `test/` (containing no other files) and bundled as a tar file `tests.tar` with

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

to be submitted via the Autolab server.

For this lab, all your new test programs must raise memory exceptions. In addition, your compiler should be able to execute all 314 test programs from Lab 3, plus all 320 test programs from Lab 2. These are provided in the Autolab handout for this lab.

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 l3c
```

should generate the appropriate files so that

```
% bin/l3c --safe <args>
% bin/l3c --unsafe <args>
```

will run your *L3* compiler in safe and unsafe modes, respectively. For backwards compatibility, the default is `--unsafe`. The command

```
% make clean
```

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

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 `compiler.tar`, for example with

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

to be submitted via the Autolab server. If you are using `cvs` instead of subversion, you may want to use the switch `--exclude CVS` instead of `--exclude .svn` for a cleaner hand-in.

What to Turn In

Hand-in on the Autolab server:

- At least 14 test cases, all of which generate a runtime exception related to memory access. You may add additional test programs of all kinds at your discretion. Submit `tests.tar` with the directory `tests/` with only the test files via the Autolab server. 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. 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 Nov 6**.

- The complete compiler. Submit `compiler.tar` with the directory `compiler/` after applying a `make clean`. 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.

  Compilers are due **11:59pm on Tue Oct 13**.

9 Changes

Revision 1. There were two changes: one regarding the handling of names to facilitate reliable linking, and the other regarding struct access.

- Non-external function names must be systematically renamed from the source *g* to the assembly label `_l3_`*g*. This prevents any unintended conflict between a C standard library function and the *L3* name of a function. This also means that the `_l3_main` function is now called `main` in the *L3* source because the uniform renaming creates the right internal symbol.

- When computing `*p` for a pointer `p` declared with `var p : s*`, the operational semantics prescribes that the outcome is the address *a* of a struct, rather than the struct itself (which is a large value and should not be loaded). However, if `p` is `NULL`, we nevertheless need to create an error instead of basing address calculations on `0`. While in most cases it would work to wait for *a* (or $a + \mathit{offset}$ for some field offset) to be accessed, *offset* could be large enough to take us past the read/write protected page of memory at address 0, making the language unsafe. Also, an expression such as `*p` could be used as a statement and should yield an error when `p` is `NULL` regardless of the type of the target.