15-411 Compiler Design: Lab 3
Fall 2008

Instructor: Frank Pfenning
TAs: Rob Arnold and Eugene Marinelli

Test Programs Due: 11:59pm, Tuesday, October 7, 2008
Compilers Due: 11:59pm, Tuesday, October 14, 2008

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L3*. This language extends *L2* by functions. This means you will have to change all phases of the compiler from the second lab. One can write some interesting recursive and iterative functions over integers in this language. Correctness is still paramount, but performance starts to become an issue because the code you generate will be executed on a set of test cases with preset time limits. These limits are set so that a correct and straightforward compiler without optimizations should receive full credit.

2 Requirements

As for Lab 2, 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. 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. 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 *L3* Syntax

The syntax of *L3* is defined by the context-free grammar in Figure 1. 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

L2 source programs may contain C-style comments of the form `/* ... */` for multi-line comments and `//` for single-line comments. Multi-line comments may be nested (and of course the delimiters must be balanced). Also, `#` should be considered as starting a single-line comment as such lines will be used as directives for testing and possibly other uses later in the class.

⟨program⟩ ::= ⟨gdecl⟩* ⟨function⟩*

⟨gdecl⟩ ::= **extern** ⟨type⟩ ⟨ident⟩ **(** ⟨paramlist⟩ **) ;**

⟨function⟩ ::= ⟨type⟩ ⟨ident⟩ **(** ⟨paramlist⟩ **)** ⟨body⟩

⟨paramlist⟩ ::= ε | ⟨ident⟩ **:** ⟨type⟩ [**,** ⟨ident⟩ **:** ⟨type⟩]*

⟨body⟩ ::= **{** ⟨decl⟩* ⟨stmt⟩* **}**

⟨decl⟩ ::= **var** ⟨ident⟩ [**,** ⟨ident⟩]* **:** ⟨type⟩ **;**

⟨type⟩ ::= **int**

⟨stmt⟩ ::= ⟨simp⟩ **;** | ⟨control⟩ | **;**

⟨simp⟩ ::= ⟨ident⟩ ⟨asop⟩ ⟨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⟩ | ⟨ident⟩ | ⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ |
⟨ident⟩ **(** [⟨exp⟩ [**,** ⟨exp⟩]*] **)**

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

⟨intconst⟩ ::= [**0-9**][**0-9**]* (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 *L3*

Operator	Associates	Meaning
`()`	n/a	function call, explicit parentheses
`! ~ -`	right	logical not, bitwise not, unary minus
`* / %`	left	integer times, divide, modulo
`+ -`	left	integer plus, minus
`<< >>`	left	(arithmetic) shift left, right
`< <= > >=`	left	integer comparison
`== !=`	left	integer equality, disequality
`&`	left	bitwise and
`^`	left	bitwise exclusive or
`|`	left	bitwise or
`&&`	left	logical and
`||`	left	logical or
`= += -= *= /= %= &= ^= |= <<= >>=`	right	assignment operators

Figure 2: Precedence of operators, from highest to lowest

4 *L3* Static Semantics

The *L3* language has two kinds of identifiers: those standing for functions and those standing for integers. These are in separate name spaces, so a function and a variable may have the same name without conflict. Reserved words of the grammar (`extern`, `var`, `int`, `if`, `else`, `while`, `for`, `continue`, `break`, `return`) cannot be used as function or variable names.

Regarding functions, several properties must be checked.

- Every function that is called must be declared somewhere in the file, either using `extern` or with a definition. The order of the defined functions in the file is irrelevant, but a function may not be declared more than once.
- Functions must be called with the correct number of arguments. Since there is only one basic type in the language, namely `int`, this is the extent of type-checking required.
- Each function must explicit `return` with a value. This is checked by verifying that every finite control flow path originating at the top of a function ends in a `return` statement. A precise specification of this condition is given in Lab 2.
- There must be a function `main()` which returns an integer as the result of the overall computation (and not an exit status).
- Each `break` or `continue` statement must occur inside a `for` or `while` loop.

Regarding variables, we need to check two properties.

- Every variable must be declared, either as a function parameter or an explicit local variable with a `var` declaration. Function parameters and local variables are local to a function and have nothing to do with parameters or local variables in other functions. Variables may not be redeclared, that is, names of parameters to a given function and its local variables must all be pairwise distinct. There are no global variables.
- Each local variable must be defined by an assignment before it is used. This is checked by verifying that along all control flow paths originating at the beginning of the function, each local variable is assigned to before it is used. This ensures that there will be no references to uninitialized variables. A precise specification of this condition is given in Lab 2.

The restriction on `return` statements and initializing variables guarantee that the meaning of every valid program is deterministic: it will either return a unique value, raise a `div` exception, or fail to terminate.

5 *L3* Dynamic Semantics

In most cases, statements have the familiar operational semantics from C. Conditionals, `for`, and `while` loops execute as in C. `continue` skips the rest of the statements in a loop body and `break` jumps to the first statement after a loop. As in C, when encountering a `continue` inside a `for` loop, we jump to the *step* statement.

The meanings of the special assignment operators are as in *L2*, where x *op*`=` e is the same as x `=` x *op* e.

Integer Operators

Since expressions contain division and modulus operations and function calls, the order of evaluation is important and specified to be from left to right. We recommend to make evaluation order explicit for potentially effectful expressions in the translation to IR trees to give optimizations and instruction selection more leeway on pure expressions.

The integers of this language are signed integers in two's complement representation with a word size of 32 bits. The semantics of the operations is given by modular arithmetic as in *L2*. Recall that division by zero and division overflow must raise a runtime division exception. The shift operations `<<` and `>>` are logical shifts, where the shift quantity is masked to the lowest 5 bits. The comparison operators `<`, `<=`, `>`, `>=`, `==`, and `!=` have their standard meaning on signed integers as in the definition of C and always return either `0` or `1`.

Logical Operators

The operators `&&`, `||` and `!` are the so-called logical operators. The first two evaluate from left to right and short-circit evaluation when appropriate. They interpret an argument of `0` as false and non-zero as true, and always produce either `0` (for false) or `1` (for true).

Function Calls

Function calls $f(e_1, \ldots, e_n)$ must evaluate their arguments from left to right before passing the resulting values to f.

6 Project Requirements

For this project, you are required to hand in test cases and a complete working compiler for *L2* 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` n	program must compile but raise the runtime exception n
`#test error`	program must fail to compile due to a *L3* source error

followed by the program text. The default exception is 8 (`SIGFPE`) which is raised for division by zero. All test files should be collected into a directory `test/` (containing no other files) and submitted via the Autolab server.

We would like some fraction of your test programs to compute "interesting" functions on specific values; please briefly describe such examples in a comment in the file. Disallowed are programs computing Fibonacci numbers, factorials, greatest common divisors, and minor variants thereof. Please use your imagination!

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 <args>
```

will run your *L3* 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 `lab3` 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>/lab3/tests
% svn checkout https://cvs.concert.cs.cmu.edu/15-411/<team>/lab3/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, at least two of which generate as error and at least two others raise a runtime exception. 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 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 Oct 7, 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 Tue Oct 14, 2008**.

7 Notes and Hints

Static Checking

Checking that functions are declared and have the right number of arguments is relatively straightforward, but will require two passes: one to collect all functions and their number of arguments, and one to check that function calls are correct.

Checking that function bodies explicitly return should follow the rules given in the handout for Lab 2, which should be easy to implement on abstract syntax trees. Similarly, checking that variables are initialized before they are used should follow the rules given in Lab 2. Some care should be given to produce a useful error message if these checks are not satisfied.

Compiling Functions

It is not a strict requirement, but we recommend compiling functions completely independently from each other, taking care to respect the calling conventions but making no other assumptions. Interprocedural program analysis and optimization is difficult and, if you do it at all, is better left to a later lab.

Calling Conventions

Your code must strictly adhere to the x86-64 calling conventions. This will make sure that it links correctly with our runtime system. We recommend that you study the corresponding section in the Application Binary Interface (ABI) available under Resources on the course home page. Fortunately, all arguments and return values are of type `int` which greatly simplifies the most general situation, as does the fact *L3* does not have functions with variable numbers of arguments. The first six arguments are passed in fixed registers, any further arguments are passed on the stack. Note that each argument on the stack takes up 8 bytes, even if only a 32-bit integer is passed. Another subtle point is the stack alignment requirement: before calling a function `%rsp` must be a multiple of 16.

We will test your compiler with some programs which call C and assembly functions declared as external in order to verify that your generated code obeys the calling conventions.

The use of the frame pointer `%rbp` is optional, as is the use of the red zone.

There are few functions with more than 6 arguments, so it may be a good strategy to implement parameter passing in registers first (which would lose only a few points) and then add stack-based passing of additional arguments later.

Naming of Functions in Generated Code

Functions declared as external (and therefore not defined in the file) should be referred to by their given name.

Functions defined in the file must be systematically renamed from the source g to the assembly label `_l3_`g. This prevents any conflict between a C standard library function and the *L3* name of a function. This also means that the function `main` has to be named `_l3_main` in the assmbly language file.

The names of all functions g defined in a file should be exported as symbols `_l3_`g. This will allow our test code to call internal functions and verify adherence to calling conventions.

Run-time Environment

The tests will be run in the standard Linux environment on the lab machines; the produced assembly code must conform to those standards. We recommend the use of `gcc -S` to produce assembly files from C sources which can provide template code and assembly language examples.

We will link your code against some library functions you can find in the files `runtime/l3rt.h` and `runtime/l3rt.c`. This will allow us to test your adherence to the C calling conventions on the x86-64 architecture, and it will allow you to call some standard library functions, say, to print messages, painful though it may be in a language without strings. The functions we explicitly define can be declared in *L3* as

```
extern int printchar(c:int);  /* print c as ASCII character */
extern int printint(n:int);   /* print n in decimal */
extern int printhex(n:int);   /* print n in hexadecimal */
```

If your compiler detects any (compile-time) errors in the source program, it should exit with a non-zero return code. If compilation succeeds and target code is generated, the compiler should then exit with a return code of 0.

Good Coding Practices

Please remember that **your code will be read and graded** by the instructor or a teaching assistant. This means your code must be readable. This also means that the code should be broken up along natural module boundaries. Some specific pieces of advice are listed below. As usual, such generalizations have to be taken with a grain of salt, but we hope they may still be useful.

- Every `structure` should have a `signature` which determines its interface.
- Seal structures by ascribing signatures with '`:>`' and not '`:`'. The latter might accidentally leak private information about the structure.

- Use functors only when you instantiate them more than once.
- Format your code to a line width of no more than 100 characters.
- Tabs, if used at all, should have a standard width of 8.
- Do not use `open LongStructureName` because the reader will then be unable to tell where identifiers originate. It can also lead to unfortunate shadowing of names. Instead, use, for example, `structure L = LongStructureName` inside a structure body and qualify identifiers with '`L.`'.
- Use variable names consistently.
- Use comments, but do not clutter the code too much where the meaning is clear from context.
- Avoid excessive uses of combinators such as `foldl` and `foldr`. They tend to be easier to write than to read and understand.
- Develop techniques for unit testing, that is, testing modules individually. This helps limit the problem of nasty end-to-end bugs that are very difficult to track.
- Do not prematurely optimize. Write clear, simple code first and optimize only as necessary, when bottlenecks have been identified. Compiler speed is not a grading criterion!
- Do not prematurely generalize. Solve the problem at hand without looking ahead too much at future labs. Such generalizations are unlikely to simplify later coding because it is generally very difficult to anticipate what might be needed. Instead, they may make the present code harder to follow because of unmotivated pieces.
- Be clear about the data structures and algorithms you want to implement before starting to write code.