15-411 Compiler Design: Lab 2
Fall 2007

Instructor: Frank Pfenning
TAs: David McWherter and Noam Zeilberger

Test Programs Due: 11:59pm, Tuesday, September 25, 2007
Compilers Due: 11:59pm, Tuesday, October 2, 2007
Revision 1: Mon Sep 24, 2007

1 Introduction

The goal of the lab is to implement a complete compiler for the language *L2*. This language extends *L1* by conditionals, loops, functions, and some additional operators. This means you will have to change all phases of the compiler from the first 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 1, you are required to hand in test programs as well as a complete working compiler that translates *L2* 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 *L2* Syntax

The syntax of *L2* 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.

$$
\begin{array}{lcl}
\langle\text{program}\rangle & ::= & \langle\text{function}\rangle^{*} \\
\langle\text{function}\rangle & ::= & \langle\text{type}\rangle\ \langle\text{ident}\rangle\ \textbf{(}\ \langle\text{paramlist}\rangle\ \textbf{)}\ \langle\text{body}\rangle \\
\langle\text{paramlist}\rangle & ::= & \varepsilon \mid \langle\text{ident}\rangle\ \textbf{:}\ \langle\text{type}\rangle\ [\ \textbf{,}\ \langle\text{ident}\rangle\ \textbf{:}\ \langle\text{type}\rangle\ ]^{*} \\
\langle\text{body}\rangle & ::= & \textbf{\{}\ \langle\text{decl}\rangle^{*}\ \langle\text{stmt}\rangle^{*}\ \textbf{\}} \\
\langle\text{decl}\rangle & ::= & \textbf{var}\ \langle\text{ident}\rangle\ [\ \textbf{,}\ \langle\text{ident}\rangle\ ]^{*}\ \textbf{:}\ \langle\text{type}\rangle\ \textbf{;} \\
\langle\text{type}\rangle & ::= & \textbf{int} \\
\langle\text{stmt}\rangle & ::= & \langle\text{simp}\rangle\ \textbf{;} \mid \langle\text{control}\rangle \mid \textbf{;} \\
\langle\text{simp}\rangle & ::= & \langle\text{ident}\rangle\ \langle\text{asop}\rangle\ \langle\text{exp}\rangle \\
\langle\text{control}\rangle & ::= & \textbf{if}\ \textbf{(}\ \langle\text{exp}\rangle\ \textbf{)}\ \langle\text{block}\rangle\ [\ \textbf{else}\ \langle\text{block}\rangle\ ] \mid \\
 & & \textbf{while}\ \textbf{(}\ \langle\text{exp}\rangle\ \textbf{)}\ \langle\text{block}\rangle \mid \textbf{for}\ \textbf{(}\ [\ \langle\text{simp}\rangle\ ]\ \textbf{;}\ \langle\text{exp}\rangle\ \textbf{;}\ [\ \langle\text{simp}\rangle\ ]\ \textbf{)}\ \langle\text{block}\rangle \mid \\
 & & \textbf{continue}\ \textbf{;} \mid \textbf{break}\ \textbf{;} \mid \textbf{return}\ \langle\text{exp}\rangle\ \textbf{;} \\
\langle\text{block}\rangle & ::= & \langle\text{stmt}\rangle \mid \textbf{\{}\ \langle\text{stmt}\rangle^{*}\ \textbf{\}} \\
\langle\text{exp}\rangle & ::= & \textbf{(}\ \langle\text{exp}\rangle\ \textbf{)} \mid \langle\text{intconst}\rangle \mid \langle\text{ident}\rangle \mid \langle\text{unop}\rangle\ \langle\text{exp}\rangle \mid \langle\text{exp}\rangle\ \langle\text{binop}\rangle\ \langle\text{exp}\rangle \mid \\
 & & \langle\text{ident}\rangle\ \textbf{(}\ [\ \langle\text{exp}\rangle\ [\ \textbf{,}\ \langle\text{exp}\rangle\ ]^{*}\ ]\ \textbf{)} \\
\langle\text{ident}\rangle & ::= & [\textbf{A-Z\_a-z}][\textbf{0-9A-Z\_a-z}]^{*} \\
\langle\text{intconst}\rangle & ::= & [\textbf{0-9}][\textbf{0-9}]^{*} \qquad \text{(in the range } 0 \leq \text{intconst} < 2^{32}\text{)} \\
\langle\text{asop}\rangle & ::= & \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \mid \texttt{\&=} \mid \texttt{\^{}=} \mid \texttt{|=} \mid \texttt{<<=} \mid \texttt{>>=} \\
\langle\text{binop}\rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \\
 & & \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{\&} \mid \texttt{\^{}} \mid \texttt{|} \mid \texttt{<<} \mid \texttt{>>} \\
\langle\text{unop}\rangle & ::= & \texttt{!} \mid \texttt{\~{}} \mid \texttt{-}
\end{array}
$$

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 *L2*

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 *L2* Static Semantics

The *L2* 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 (`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. The order of the 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.
- Functions must terminate with an explicit `return` statement. This is checked by verifying that each control flow path originating at the top of a function ends in a `return` statement. See a more detailed explanation below.
- There must be a function `_l2_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 one properties.

- Every variable must be declared, either as a function parameter or an explicit local variable with `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.

Local variables declared with `var` will be silently initialized to 0 by the compiler, avoiding any issues with uninitialized variables.

In order to describe how to check for `return` statements, we postulate abstract syntax trees for statements s according to the definition

$$s ::= \mathrm{assign}(x, e) \mid \mathrm{if}(e, s, s) \mid \mathrm{while}(e, s) \mid \mathrm{for}(s, e, s, s) \mid \mathrm{continue} \mid \mathrm{break} \mid \mathrm{return}(e) \mid \mathrm{nop} \mid \mathrm{seq(s,s)}$$

where e stands for an expression and x for an identifier. Do not be confused by the fact that this looks like a grammar: the terms on the right hand side describe trees, not strings. Your implementation may of course differ from this—we use this merely as a means of specifying when function bodies are well-formed.

We check that executing a function always ends in an explicit `return` statement with the following rules. We say that *s returns* if executing s will always ends with a return statement. Overall, we want to ensure that the body of every function returns, according to this definition.

$\mathrm{assign}(x, e)$	does not return
$\mathrm{if}(e, s_1, s_2)$	returns if both s_1 and s_2 return
$\mathrm{while}(e, s)$	does not return
$\mathrm{for}(s_1, e, s_2, s_3)$	does not return
$\mathrm{return}(e)$	returns
nop	does not return
$\mathrm{seq}(s_1, s_2)$	returns if either s_1 returns (in which case s_2 must be dead code) or s_2 returns

We do not look inside loops (even though the bodies may contain `return` statements) because the body might be executed 0 times. Because we do not look inside loops, we do not need rules for `break` or `continue`.

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 *L2* 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. This means that a `for` loop cannot be easily expanded into a `while` loop, unless there is no `continue` in the loop body. Both always apply to the innermost loop that they occur in. Local variables declared with `var` x : `int` must be initialized to `0`.

The meanings of the special assignment operators are as in *L1*, where x *op*`=` e is the same as x `=` x *op* e.

Integer Operations

Since expressions do not have effects (except for a possible `div` exception that might be raised) the order of their evaluation is irrelevant.

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 *L1*. Recall that division by zero and division overflow must raise a runtime division exception. This is the only runtime exception that should be possible in the language, except for those defined by the runtime environment such as stack overflow.

The left `<<` and right `>>` shift operations are arithmetic shifts. Since our numbers are signed, this means the right shift will copy the sign bit in the highest bit rather than filling with zero. Left shifts always fill the lowest bit with zero. Also, the shift quantity k will be masked to 5 bits and is interpreted positively so that a shift is always between 0 and 31 bits, inclusively. This is also the hardware behavior of the appropriate arithmetic shift instructions on the x86-64 architecture and is consistent with C where the behavior is underspecified.

The comparison operators `<`, `<=`, `>`, `>=`, `==`, and `!=` have their standard meaning on signed integers as in the definition of C.

Logical Operators

The operators `&&`, `||` and `!` are the so-called *logical operators*. They interpret an argument of 0 as false and non-zero as true, and always produce either 0 (for false) or 1 (for true). A tricky aspect of logical and (`&&`) and logical or (`||`) is they evaluate their arguments from left to right and short-circuit evaluation if the left-hand operand yields 0 (for logical and) and 1 (for logical or). In those cases, they do not evaluate their second operand and return 0 (for logical and) and non-zero (for logical or). Your implementation must model this semantics faithfully.

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 `.l2` 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 *L2* 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.

We would like some fraction of your test programs to compute "interesting" functions; please briefly describe such examples in a comment in the file.

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 l2c
```

should generate the appropriate files so that

```
% bin/l2c <args>
```

will run your *L2* compiler. 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 20 test cases, at least two of which generate as error and at least two others raise a runtime exception. 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 Sep 25**.

- 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 2**.

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 earlier in the handout, which should be easy to implement on abstract syntax trees. Some care should be given to produce a useful error message.

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 the *L2* 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.

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.

Shift Operators

There are some tricky details on the machine instructions implementing the shift operators. The instructions `sall` k, D (shift arithmetic left long) and `sarl` k, D (shift arithmetic right long) take a shift value k and a destination operand D. The shift either has to be the `%cl` register, which consists of the lowest 8 bits of `%rcx`, or can be given as an immediate of at most 8 bits. In either case, only the low 5 bits affect the shift of a 32 bit value; the other bits are masked out. The assembler will fail if an immediate of more than 8 bits is provided as an argument.

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.

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.

8 Changes

The changes from revision 0 are the following:

- In the grammar, a ⟨simp⟩ statement must be an assignment and can no longer be a **return**, which is now part of the ⟨control⟩ non-terminal.
- Both **continue** and **break** terminals must now be followed by a semicolon '**;**'.
- The meaning of a **continue** inside a **for** loop is now as in C and executes the *step* statement. Therefore a **for** loop can no longer be easily expanded into a **while** loop.