Final Project (Due: TBA (12:00pm (צהריים), TBA, 2017)

Mayer Goldberg

January 7, 2017

Contents

1 General

- You may work on this assignment alone, or with a single partner. You may not join a group of two or more students to work on the assignment. If you are unable to find or maintain a partner with whom to work on the assignment, then you will work on it alone.
- You should be very careful to test your work before you submit. Testing your work means that all the files you require are available and usable and are included in your submission.
- Your work should run in Chez Scheme. We will not test your work on other platforms. **Test, test, and test again:** Make sure your work runs under Chez Scheme the same way you had it running under Racket. We will not allow for re-submissions or corrections after the deadline, so please be responsible and test!
- Make sure your code doesn't generate any unnecessary output: Forgotten debug statements, unnecessary print statements, etc., will result in your output being considered incorrect, and you will lose points. You will not get back these points by appealing or writing emails and complaints, so **please** be careful and make sure your code runs properly

2 Changes to this document

-

3 Compiler Optimization: Removal of write-after-write instructions (10%)

For this problem you will implement the procedure `remww`, which takes a list of assembly instructions, in some abstract representation, and returns a list of assembly instructions in the same representation, in which redundant instructions have been removed. For the purpose of this problem, we only consider instructions that effect *registers* rather than *memory*. Instructions are denoted by

- A name for the instruction
- A list of registers from which the instruction *reads*
- A list of registers to which the instruction *writes*

Registers are denoted by their index, so that register `R4` is simply denoted by the number `4`, etc.

For example, the instruction `(inst2 (1 2) (2))` reads from registers 1 & 2 and writes to register 2.

An instruction is redundant if it has no visible effects on the register set, meaning, if the effects were entirely overwritten by later instructions, before any instruction had read from these registers. For example, in the sequence

```
((inst1 () (1))
 (inst2 () (1)))
```

As you can see, both `inst1` and `inst2` write to register 1. It **really** doesn't matter what these instructions actually do. Whatever effect `inst1` has had, it has been completely overwritten by `inst2`. No instruction has had a chance to read from register 1 before `inst2`, so in fact, `inst1` is redundant, and can be removed. The resulting list of instruction shall contain only `((inst2 () (1)))`.

Consider a more involved example:

```
((inst1 (1) (1))
 (isnt2 (1) (1))
 (inst3 () (1)))
```

In this example, `inst3` makes `inst2` redundant, and subsequently, `inst1` becomes redundant too! Such optimizations can have a landslide effect of removing many instructions.

Here is a larger example, generated automatically from my code, and here is how the `remww` procedure is used:

```
> (remww '((g46494 (6 1) (6))
   (g46495 (3 6 0) (0 4))
   (g46496 (1) (3))
   (g46497 (5 0 6) (4 2))
   (g46498 (7 1 3) ())
   (g46499 (7) ())
```

```
   (g46500 (0 1) ())
   (g46501 (4 7) (7))
   (g46502 (5 7 0) ())
   (g46503 (5 2 0) (6 2))
   (g46504 (6 4 5) (0))
   (g46505 (1 2) (1))
   (g46506 (6 2 7 5) (7))
   (g46507 (1) ())
   (g46508 (1) ())
   (g46509 (0 3 2) (5 0))
   (g46510 (3 6) (4))
   ))
((g46494 (6 1) (6)) (g46495 (3 6 0) (0 4)) (g46496 (1) (3))
  (g46497 (5 0 6) (4 2)) (g46501 (4 7) (7))
  (g46503 (5 2 0) (6 2)) (g46504 (6 4 5) (0))
  (g46505 (1 2) (1)) (g46506 (6 2 7 5) (7))
  (g46509 (0 3 2) (5 0)) (g46510 (3 6) (4)))
```

3.1 How we shall grade this part of your code

A solution will be posted, against which you may test your own code. Your code shall be graded against the output of my own solution.

This part of the assignment shall be worth 10% of the grade of the final project. The code generate, run-time, etc., shall be worth the remaining 90%. We suggest you spend most of your time working on the compiler, and tackle this problem when you're reasonably certain you shall be done on time.

3.2 What to submit

Write you code in the file `remww.scm`.

4 Writing the code generator

For this problem you are asked to write in your `compiler.scm` file the following procedures: `code-gen`, and `compile-scheme-file`.

The procedure `code-gen` takes an `pe` and returns for it a string that contains lines of assembly instructions in the CISC architecture, such that when the execution of these instructions is complete, the value of `e` should be in the *result register* `R0`.

The procedure `compile-scheme-file` takes the name of a Scheme source file (e.g., `foo.scm`), and the name of a CISC assembly target file (e.g., `foo.c`). It then performs the following:

- Reads the contents of the Scheme source file, using the procedure `file->string` (provided below).
- Reads the expressions in the string, returning a list of *sexprs*.
- Applies to each *sexpr* in the above list the following, in order:
  - `parse`

  - `eliminate-nested-defines`
  - `remove-applic-lambda-nil`
  - `box-set`
  - `pe->lex-pe`
  - `annotate-tc`

- Constructs the constants table, and the global variable table.
- Calls `code-gen` to generate a string of CISC assembly instructions for all the expressions. After the code for evaluating each expression, there should appear a call to an assembly language routine for printing to the screen the value of the expression (the contents of `R0`) if it is not the *void object*.
- Add a *prologue* and *epilogue* to the string, so that it becomes a self-contained assembly language program.
- Write the string containing the assembly language program to the target file.

The target assembly language file should be compilable via `gcc`. If the name of the file is `foo.c`, the graders should be able to perform at the Linux shell prompt the following:

```
% gcc -o foo foo.c
```

The file `foo` should be a valid executable, and running it should print to *stdout* the values of the expressions in the Scheme source file.

You are free to generate any valid *arch* code, but you are advised to follow the outline of the code generator presented in class.

```
(define file->string
  (lambda (in-file)
    (let ((in-port (open-input-file in-file)))
      (letrec ((run
(lambda ()
  (let ((ch (read-char in-port)))
    (if (eof-object? ch)
(begin
  (close-input-port in-port)
  '())
(cons ch (run)))))))
(list->string
 (run))))))
```

4.1 The target language

The target language, i.e., the language that your code generator will output, is supposed to model a generic CISC-like assembly language for a general register architecture. What this means is that you have a relatively large number of general purpose registers that can all be used for arithmetic, logical and memory operations. The instruction set provides you with operations for computing sums, differences, products, quotients, remainders, bitwise Boolean operations, simple conditionals, branches and subroutines:

- **Registers.** The `cisc.h` file, which defines our micro-architectural, specifies 16 general-purpose registers. Sixteen registers is a reasonable number; We have certainly used less than a half of these, and you may add as many more as you like. , We also have the stack pointer `SP`, the frame pointer `FP`. You may define the stack size to be whatever you like, but `Mega(64)` seems to be a reasonable starting point.

- **Arithmetic Operations.**

You may carry out arithmetic operations between registers and registers or between registers and constants. **No nesting of operations is permitted**, so, for example, you can have `MOV(R0, R3)` or `ADD(R4, R3)`, but you may not have something like `MOV(R0, ADD(R3, R5))`, etc.

- **Branches, Subroutines, Labels.**

You can define any number of labels, and branch to any label. A label in C is defined as a name followed by a colon (e.g., `L34:`). As a rule of thumb, any name that would qualify as a variable or procedure name could also be used as a label. The commands `JUMP`, `JUMP_condition`, `CALL`, `CALLA`, `RETURN`, as well as other commands documented in the manual are available for jumping to labels, calling & returning from subroutines. Consult the documentation & code examples for more details.

As mentioned in class, unlike the situation in assembly language, labels in C are *symbolic* entities and have no addresses. The `gcc` compiler introduces a non-standard *extension*, in which the address of a label can be found via `&&LabelName`, and is a datum of type `void *`. Correspondingly, jumping to an *address*, as opposed to a label, is accomplished by means of another non-standard extension: `goto *addr`.

Our micro-architecture makes extensive use of these extensions to the C standard. This means that this part of the project cannot be done in a compiler other than `gcc`. This compiler is available on the departmental Unix and Linux machines, comes as the default C compiler on Linux & OSX, **and can be downloaded and used under Windows.** For information on how to get `gcc` for Windows, please consult the course syllabus.

- **Conditionals.**

Consult the documentation of the CISC micro-architecture, as well as the sample library, for information on all the conditions that can be tested, and how to use them.

- **Stack Operations.**

Consult the documentation of the CISC micro-architecture, as well as the sample library, for information on the structure of the stack, the frame, the `FPARG` and `STARG` macros, and how to use them.

5 Run-time support

Certain elementary procedures need to be available for the users of your compiler. Some of these "built-in" procedures need to be hand-written, in assembly language. Others can be written in Scheme, and compiled using your compiler. The procedures you need to support include:

`append` (variadic), `apply`, `<` (variadic), = (variadic), `>` (variadic), `+` (variadic), `/` (variadic), `*` (variadic), `-` (variadic), `boolean?`, `car`, `cdr`, `char->integer`, `char?`, `cons`, `denominator`,

`eq?`, `integer?`, `integer->char`, `list` (variadic), `make-string`, `make-vector`, `map`, `not`, `null?`, `number?`, `numerator`, `pair?`, `procedure?`, `rational?`, `remainder`, `set-car!`, `set-cdr!`, `string-length`, `string-ref`, `string-set!`, `string->symbol`, `string?`, `symbol?`, `symbol->string`, `vector`, `vector-length`, `vector-ref`, `vector-set!`, `vector?`, `zero?`.

The arithmetic procedures should work with both integers and rational numbers wherever this makes sense mathematically.

6 How we shall test your compiler

We will run your compiler on various *test files*. For each test file `foo.scm`, we will do three things:

1. Run `foo.scm` through Chez Scheme, and collect the output in a list. If the output consists of several sexprs, we'll just wrap parenthesis around them so we have one huge, happy, valid list.

2. Run your compiler on `foo.scm`, obtaining `foo.c`. We shall then compile `foo.c` using `gcc` to obtain an executable `foo`. We shall then run `foo` and collect its output in a list, just as in the previous item.

3. The list generated in item (1) and the list generated in item (2) shall be compared using the `equal?` predicate that is built-in in Chez Scheme.

If the `equal?` predicate returns `#t`, you get a point. Otherwise, you don't. You could lose a point if the Scheme code broke, if `gcc` failed to compile the CISC assembly language file, if your file generated a segmentation fault, or if the `equal?` predicate in Chez Scheme returned anything other than `#t` when comparing the two lists.

7 What to Submit

Submit the file `final-project.zip`. Under extraction, it should create the directory `final-project` and in it the following files:

1. The file `remww.scm` containing your solution to the *write-after-write* optimization. This file and the code therein are not a part of the compiler pipeline, and shall be graded separately.

2. The file `compiler.scm` containing all the code for this assignment.

3. The file `pc.scm` containing the parsing combinator package you used in your compiler.

4. The file `cisc.h`, and the sub-directory `./lib` containing the libraries of CISC assembly sub-routines

5. Any additional files needed to compile your executable. You may use any number of files you need, but please make sure that the resulting assembly file compile and run as we specified.

6. The file `readme.txt` containing the following information:

   (a) The names and IDs of all the people who worked on this assignment. There should be either your own name, or your name and that of your partner. You may only have one partner for this assignment.

   (b) The following statement:

> I (We) assert that the work we submitted is 100% our own. We have not received any part from any other student in the class, nor have we give parts of it for use to others. Nor have we used code from other sources: Courses taught previously at this university, courses taught at other universities, various bits of code found on the internet, etc.
>
> We realize that should our code be found to contain code from other sources, that a formal case shall be opened against us with , in pursuit of disciplinary action.

Please be careful to check your work multiple times. Because of the size of the class, we cannot handle appeals to recheck your work in case you forget or fail to follow these instructions precisely.

8 A word of advice

The class is very large. We do not have the human resources to handle late submissions or late corrections from people who do not follow instructions. By contrast, it should take you very little effort to make sure your submission conforms to what we ask. If you fail to follow the instructions to the letter, you will not have another chance to submit the assignment:

If you fail to submit `zip` file, if files are missing, if functions don't work as they are supposed to, if the statement asserting authenticity of your work is missing, if your work generates output that is not called for (e.g., because of leftover debug statements), etc., then you're going to get a grade of zero. The graders are instructed not to accept any late corrections or re-submissions under any circumstances.