Compiler Construction
Final Project

Due: Thursday, February 21, by 13:00

Last modified on Sunday, January 5, 2013

Contents

General Instructions

1. Due Dates. The final project is due by 13:00. For every late weekday (including Fridays & Saturdays), you will be deducted a penalty of 5%. This penalty will apply in all but two situations, as specified by the Faculty of Natural Science:

   - You're called for military reserve duty (מילואים). You will be required to provide suitable documentation from the IDF.
   - You're suffering from a medical condition that required your hospitalization for longer than a day. In such case, you must provide suitable documentation by a qualified physician.

   In both cases, you are excused from handing in the assignment. Your grade will be computed as a function of the remaining coursework.

2. Group work. The assignments are designed so that you can complete them alone. You may, if you so choose, work in pairs. You may not work in larger groups.

3. Academic Dishonesty. You are expected to conduct yourself with honesty and sincerity throughout this course, as befitting a university student.

   - You may not discuss your course work with *anyone* other than your partner for this immediate assignment, or the teaching staff for this course.

- You may not use portions of code from previous semesters, or from other students.

4. Programming Style. Your programs will be graded for style as well as for correctness. Your code is expected to be clean, neat, readable, efficient, safe, and extensible. Your ML & Scheme code is expected to be functional, unless the problem states explicitly that you may use side-effect. Your code in C, Java, & assembly are expected to build and execute flawlessly, without generating warnings or errors. You can lose (or gain) points for style.

1 Code Generation Into Pseudo-Assembly Language

1.1 What you need to do

For this problem, you will need to write a procedure `code-gen`, which takes a parsed expression *after* the `annotate-tail-calls` and `pe->lex-pe` stages, and generates CISC-like pseudo-assembly instructions for it, using the architecture *arch* posted to the website.

An easy way to structure the procedure `code-gen` is to to have, at its heart, a huge `cond`-expression, that dispatches on all the types of parsed expressions in your intermediate language. Thus, for a constant, `code-gen` should call the procedure `code-gen-const`, for a free variable, `code-gen` should call the procedure `code-gen-free-var`, etc. Note that `code-gen` is recursive, and for an `if3`-expression, code-gen should call the procedure `code-gen-if3`, which then calls code-gen on the three relevant sub-expressions, combining the three strings to produce a string that will be the return value of `code-gen-if3`, and subsequently of `code-gen`, when called with an `if3`-expression.

To combine large strings out of smaller ones, you can use the Scheme procedure `string-append`, which, like the corresponding procedure for lists, is a variadic lambda. Here's a simple example:

```
(define ^^label
  (lambda (name)
    (let ((n 0))
      (lambda ()
        (set! n (+ n 1))
        (string-append name
          (number->string n))))))
(define ^label-if3else (^^label "Lif3else"))
(define ^label-if3exit (^^label "Lif3exit"))
(define nl (list->string (list #\newline)))
(define code-gen-if3
  (lambda (e)
    (with e
     (lambda (if3 test do-if-true do-if-false)
       (let ((code-test (code-gen test))
             (code-dit (code-gen do-if-true))
             (code-dif (code-gen do-if-false))
             (label-else (^label-if3else))
             (label-exit (^label-if3exit)))
         (string-append
         code-test nl ; when run, the result of the test will be in R0
          "CMP(R0, SOB_BOOLEAN_FALSE);" nl
          "JUMP_EQ(" label-else ");" nl
```

```
code-dit nl
"JUMP(" label-exit ");" nl
label-else ":" nl
code-dif nl
label-exit ":"))))))
```

The macro `SOB_BOOLEAN_FALSE` should be defined appropriately.[1] As discussed in class, the correctness of our compiler rests on the *induction hypothesis* that the result of a call to the `code-gen` on a parsed expression $\mathcal{PE}$ is assembly code, the execution of which will place the *value* of $\mathcal{PE}$ in the result register `R0`. Therefore, if by assuming that our induction hypothesis holds for `test`, `do-if-true`, and `do-if-false`, it follows that the code produced by `code-gen-if3` will also place the result of the given `if3`-expression in the register `R0`.

In the above code, we didn't bother to generate C comments. Repeating what was stressed in class, we **urge** you to generate C comments in the code you generate. These comments will help you find bugs in the code you generate, and will save you a lot of time.

1.2 Assumptions about the source language

The source language that your compiler will handle is a subset of the Scheme programming language, with several extensions and variations:

The only numerical type that will be supported by our compiler will be the integer. We will not be supporting the full numerical tower in Scheme. We will not be supporting *bignums*, that is, integers of arbitrary precision. For the purpose of this project, the microarchitectural word size is perfectly fine for the internal representation of integers. Remember, though, that Scheme integers are still tagged data types, so don't forget to set the tag!

1.3 Assumptions about 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 microarchitectural, 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.

[1] And we repeat the recommendation given in class to use C macros wherever possible. This will save you a great deal of debugging, and help make the code clearer and cleaner.

- **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 microarchitecture 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* microarchitecture, 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* microarchitecture, 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.

1.4 The builtin Scheme procedures you need to implement

You will need to implement the following list of primitive procedures: `apply`, `<` (variadic), `=` (variadic), `>` (variadic), `+` (variadic), `/` (variadic), `*` (variadic), `-` (variadic), `boolean?`, `car`, `cdr`, `char->integer`, `char?`, `cons`, `eq?`, `integer?`, `integer->char`, `make-string`, `make-vector`, `null?`, `number?`, `pair?`, `procedure?`, `remainder`, `set-car!`, `set-cdr!`, `string-length`, `string-ref`, `string-set!`, `string->symbol`, `string?`, `symbol?`, `symbol->string`, `vector-length`, `vector-ref`, `vector-set!`, `vector?`, `zero?`. These procedures should be coded either in the *cisc* microarchitecture, or implemented in Scheme and pre-loaded before any user code gets executed.

We will provide you with the Scheme file "`support-code.scm`" that contains the definitions of various procedures that come with every Scheme system. These procedures are defined in terms of the above hand-coded procedures, and should be *auto-loaded* into the Scheme system (i.e., should be available to every user of your compiler). You are free to use my code, or you may write your own, or you may decide to support some of these procedures directly in pseudo-assembly. This is up to you.

The final thing your code should do, before it exits, is to print the value in `R0`. You can do this by calling the subroutine `WRITE_SOB` in the file `cisc/lib/scheme/write_sob.asm`. Of course, you will need to modify this subroutine to support additional types.

1.5 How to build your executable

You will need to provide the procedure `compile-scheme-file`, which shall take two strings as input: The name of the Scheme source file (for example, "*in.scm*"), and

the name of the assembly/C target file (for example, "*out.c*"). You will provide a *makefile* to compile and link the C file into an exacutable (for example, "*out*").

You may get compile-time warnings and errors from *gcc*, which would mean that you are generating code that is incorrect or unsafe. In such case, open the file (for example, "*out.c*") in a text editor, study the location of the warning or error, and try to identify which part in the code generator is responsible for generating this code. You can then change the code generator, re-load into Scheme, re-run on some input, and try to compile again.

You may get run-time errors, your code could go into an infinite loop, or cause some other problem. In this case, you could compile your code with the `-g` flag (for generating debug information) and try to debug it using the *gdb* — GNU debugger, however this route may prove very tedious. You might be better off trying to modify the file "`user-scheme-code.c`" by including print statements that will help you converge on the bug(s).

1.6 How to write to files in Scheme

This section is a quick-and-dirty introduction to writing to files in Scheme. You will need to know how to create and write to files, because your code generator is supposed to create a text file and write to it.

Reading and writing to/from Scheme files is done via *file ports*. To create a file output port, use the command `(open-output-file` *`filename`*`)`, where *`filename`* is a string containing the name of the file. To create a file input port, use the command `(open-input-file` *`filename`*`)`, where *`filename`* is a string containing the name of the file. When you are done working with the file, remember to close the file port by either `(close-output-port` *`port`*`)` or `(close-input-port` *`port`*`)`.

To write to an output port, use the `display` or `write` procedures where with an output-file-port as a second argument: `(display string` *`output-port`*`)` or `(write string` *`output-port`*`)`. There is a difference between the two procedures; Experiment and decide which one does what you need.

1.7 How we will test your compiler

Your compiler will be tested on increasingly complex Scheme code, starting with simple expressions to be evaluated (such as `#f`, `(+ 3 4)`, etc, all the way up to mutually recursive functions defined using `define` to define global variables or using `letrec`-expressions to define local variables). The point of these tests is to make sure that your compiler handles the basic types, supports the primitive operations on them, and can generate working code for all the Scheme expressions we support.

We advise you to test your compiler *incrementally*, as you add features and functionality. Create a large group of tests for which you know the outcome and use a makefile to compile and execute each and every one of these tests. For each sequence of tests, if your makefile redirects the output into some file, you can use the Unix *diff* command to compare this file to a file with the correct answers. Testing and comparing the output can thus be automatized, and you will only be notified when the output doesn't match what you expect. It's worth repeating, so we'll mention it again: **Develop your programs incrementally! Debug and test after each change!**

2 Final integration

You will need to provide the procedure `compile-scheme-file`, which will open the *source file*, convert the text in the file into *tokens*, using your scanner, read the

sexprs, parse them, perform on them semantic analysis and tagging of tail calls, call the code generator on them, and emit the assembly/C instructions to the *target file*. The executable will be built at the *shell*, using a *makefile*.

2.1 What to hand in

- You should hand in a Scheme file, called "`compiler.scm`", that will include all the code you need to compile Scheme files into assembly, and create the output file. The file `compiler.scm` should include your scanner, your reader, your macro-expander, your tag-parser, `pe->lex-pe`, and the code for annotating tail calls.[2]
- You should also hand in a file called "`makefile`", and any additional C source and header files (`*.c` and `*.h`) that are needed by your system.

3 What to submit

You should submit a single archive `final-project.zip` that uncompresses into the directory `final-project`, in which the following files appear:

- `readme.txt` — A text file that begins with the IDs of all the group participants, separated by a comma. For example: 123456789,987654321. Below this line, the file should contains the names of all the partners in the project, their email address, and the following statement at the end of the file:

  > Being cognizant that academic dishonesty is an offense against the regulations of the university, the faculty of natural sciences, the department of computer science, and the compiler construction course, we hereby assert that the work we submit for this assignment is entirely our own. We have made no use of solutions and/or of code from other students in the class, from students who took the class in previous years, or from any other source outside the class (e.g., a friend, the internet, etc). We realize that if we are suspected of academic dishonesty, we shall be called before the disciplinary committee (ועדת משמעת), and that the **minimal penalty** for cheating on an assignment is a failing grade in the course.
  >
  > We proudly and steadfastly maintain that this work is entirely and only our own.

  Failure to include this file, or the above statement, will result in the assignment considered to have been submitted late, and you will need to re-submit it at the usual penalty for a late submission.

- `compiler.scm`
- `makefile`
- The `arch` subdirectory and all its files

[2] If your scanner and reader do not work correctly, please include the code nevertheless, as it will be needed for the בדיקה פרונטלית of your homework. However, you can bypass the scanner and reader by calling the builtin procedure `read`, which when applied to an *input port* returns the next sexpr. Please note that the reader you wrote is supposed to return the list of *all* sexprs in the file, whereas the `read` procedure will only return the next. With very little effort, you can write a simple procedure to `read` one sexpr at a time, and return a list of all the sexprs in the file. This will bypass any defective code you may have written earlier.

- Any additional files that are necessary to compile Scheme source and build the executable.

A final word of caution

As mentioned in class, we strongly urge you, before you submit your homework, to go over the procedure names, the file names, the API to your procedures, *et cetera*, and make sure that your code complies with the requirements stated in the problem. Once submitted, your homework will be tested and graded **as is**. The graders will not make any changes to your code, nor will they let *you* make these changes after the deadline. Please heed this warning, and take a few minutes, before you submit, to re-check that your code conforms to the requirements stated in the problem.