Assignments in the Compiler Construction Course

Mayer Goldberg

January 20, 2024

Contents

1 General

- You may work on this assignment alone, or with a **single** partner. 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 are not to copy/reuse code from other students, students of previous years, or code you found on the internet. If you do, you are guilty of *academic dishonesty*. All discovered cases of *academic dishonesty* will be forwarded to the disciplinary committee (ועדת משמעת) for disciplinary action. Please save yourself the trouble, and save us the effort and the inconvenience of dealing with you on this level. This is not how we prefer to relate to our students, but *academic dishonesty* is a real problem, and if necessary, we will pursue all disciplinary venues available to us.
- 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.
- **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.
- Please read this document completely, from start to finish, before beginning work on the assignment.

2 How your assignments are organized

Because we are in a time of national crisis and the semester has been truncated to 11 weeks, we are going to work differently than in normal school-years:

- You are now given all programming tasks for this course: Two assignments and one final project.
- You are given a file `compiler.ml` that contains **the vast majority of the code for the compiler you are supposed to write**. This code loads perfectly, so you can begin writing and testing immediately.
- The file `compiler.ml` marks very clearly what you need to do and where you need to do it: In the file you shall find the declaration `exception X_not_yet_implemented of string;;`. This declaration is used to note places in the code where there is some work for you to do. Because it is an *exception*, it does not prevent the code from being loaded and compiled. If you attempt to evaluate an `raise`-expression that raises this exception, and run-time exception shall take place, and the execution of your program shall stop forthwith.
- Notice that the exception `X_not_yet_implemented` takes a string as an argument. This string is used as "documentation", so as to clarify what pieces of code need to be completed for assignment 1, assignment 2, and the final project. Of course, you may work on the later code sooner, and in any case, the purpose is just to help you identify what to work on next.
- You are not to change the signatures of any of the modules in the code.

You shall be given instructions on how to submit your final project towards the end of the semester.

3 Assignment 1

Please locate all expressions of the form `raise (X_not_yet_implemented "hw 1")` that appear in the file `compiler.ml`, and replace them with the appropriate code.

This assignment should be completed entirely in ocaml. This can be done on any computer, and does not require linux.

3.1 Hints

- The interface to the reader is the function `read`, which takes a string and returns an S-expression. You can debug your reader by running this function. You can use the custom printers for S-expressions (`print_sexpr`, `sprint_sexpr`) to print S-expressions, as per the following examples:

```
utop[6]> read;;
- : string -> sexpr = <fun>
utop[7]> read "(a b c)";;
- : sexpr =
ScmPair
 (ScmSymbol "a", ScmPair (ScmSymbol "b", ScmPair (ScmSymbol "c", ScmNil)))
utop[8]> Printf.printf "Here is the S-expression:\n\n%a\n\n"
```

```
print_sexpr (read "(a b c)");;
Here is the S-expression:

(a b c)

- : unit = ()
utop[9]> Printf.printf "Here is the S-expression:\n\n%a\n\n"
print_sexpr (read "((a . b) c)");;
Here is the S-expression:

((a . b) c)

- : unit = ()
utop[10]> Printf.printf "Here is the S-expression:\n\n%a\n\n"
print_sexpr (read "(a . (b . (c . (d . ()))))");;
Here is the S-expression:

(a b c d)

- : unit = ()
```

Notice that S-expressions are printed in their canonical representation (by applying the two *dot-rules*).

- The interface to the tag-parser is the function `parse`, which takes a string and returns an expression. You can debug your tag-parser and macro-expander by running this function. You can use the custom printers for expressions (`print_expr`, `sprint_expr`) to print parsed expressions, as per the following examples:

```
utop[1]> parse;;
- : string -> expr = <fun>
utop[2]> parse "a";;
- : expr = ScmVarGet (Var "a")
utop[3]> parse "(a b c)";;
- : expr =
ScmApplic (ScmVarGet (Var "a"), [ScmVarGet (Var "b"); ScmVarGet (Var "c")])
utop[4]> Printf.printf "And here is the parsed expression:\n\n%a\n\n"
print_expr (parse "(define fact (lambda (n) (if (zero? n) 1
(* n (fact (- n 1))))))");;
And here is the parsed expression:

(define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))

- : unit = ()
```

Notice that tag-parsing has been **undone**, by the custom printer, so this is useful to inspect the code **as understood by the tag-parser**.

4 Assignment 2

Please locate all expressions of the form `raise (X_not_yet_implemented "hw 2")` that appear in the file `compiler.ml`, and replace them with the appropriate code.

This assignment should be completed entirely in ocaml. This can be done on any computer, and does not require linux.

5 Final Project

Please locate all expressions of the form `raise (X_not_yet_implemented "final project")` that appear in the file `compiler.ml`, and replace them with the appropriate code.

The final project requires you write assembly-code in the Intel x86 instruction set, and using the Linux Application Binary Interface (ABI). You can only test your code on an Intel x86 machine running the Linux operation system.

You can debug your code generator using the following interface:

```
val compile_scheme_string : string -> string -> unit
val compile_scheme_file : string -> string -> unit
val compile_and_run_scheme_string : string -> string -> unit
```

- `compile_scheme_string`: This function takes an output file (e.g., `foo.asm`) and a string of Scheme source, and generates the file `foo.asm` by compiling the string into x86 assembly language. You can then compile the assembly file using the Newtide Assembler (*nasm*) and the `makefile` included.
- `compile_scheme_file`: This function takes two filenames: The name of an input file (a Scheme source-file, e.g., `foo.scm`) and the name of an output file (an assembly-language file, e.g., `foo.asm`), and compiles the Scheme source file into assembly language. You can then compile the assembly file using the Newtide Assembler (*nasm*) and the `makefile` included.
- `compile_and_run_scheme_file`: This function is a bit of a stretch, as it uses my own directory structure, and you may need to adapt it for your own environment, but it **really** shortens the debugging cycle, so I recommend you use it. This function takes **the base** of a filename, e.g., `foo`, and a string of Scheme source code, writes the Scheme source to the file `foo.scm`, compiles it into `foo.asm`, and then runs the assembler over it, generating an executable, which it then runs immediately, giving the user instant feedback.

Please consider the following examples of my own compiler, that is, the compiler you have, but with the parts missing in your skeleton file:

```
utop[13]> Code_Generation.compile_and_run_scheme_string "testing/goo"
"(+ 2 3)";;
nasm -g -f elf64 -l testing/goo.lst testing/goo.asm
gcc -g -m64 -no-pie -o testing/goo testing/goo.o
rm testing/goo.o
5

!!! Used 16524 bytes of dynamically-allocated memory
```

```
- : unit = ()
utop[14]> Code_Generation.compile_and_run_scheme_string "testing/goo"
"(letrec ((fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(map fact '(0 1 2 3 4 5 6 7)))";;
nasm -g -f elf64 -l testing/goo.lst testing/goo.asm
gcc -g -m64 -no-pie -o testing/goo testing/goo.o
rm testing/goo.o
(1 1 2 6 24 120 720 5040)

!!! Used 115931 bytes of dynamically-allocated memory

- : unit = ()
```

This kind of immediate feedback is precisely what you need in order to test and debug your compilers effectively. **Please get into the habit of testing frequently!**