Final Project (Due: Friday, 11/01/19, 10:00AM)

Mayer Goldberg

December 30, 2018

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 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 *va'adat mishma'at* 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
- Your code should generate absolutely no warnings or error messages. If it does, you get a grade of zero.
- No late submissions will be accepted, no exceptions. A late submission is, **by definition**, *any time* past the official deadline, according to the departmental *submission system*.
- Please read this document completely, from start to finish, before beginning work on the assignment.

2 Before beginning the final project

This project depends on and uses the code you wrote and submitted in assignments 1-3. If you haven't done so yet, you should go back and make any corrections and/or changes to the assignments to make them work as perfectly as possible.

3 Writing the code generator

We are providing you with much of the implementation details of various parts of the code generation and compilation processes. Due to time constraints we strongly advise that you make use of this code. However, this is not a requirement; you are allowed to make any changes to the provided code, as long as the executable file created using your compiler behaves correctly, and the provided interface to your compiler is not broken (explained further in Section 6).

In this assignment you are asked to implement the procedure:

- in `code-gen.ml`:
  - `make_consts_tbl : expr' list -> (constant * ('a * string)) list`
    This function takes a list of expr's generated by the semantic analyzer and outputs an association list representing the constants table.
    Note that the use type variable ='a= allows you flexibility in the way you define the addresses of constants. Additionally, in case you choose to use labels for constant-addressing, the code we provide assumes that the string representing each constant includes its corresponding label.
  - `make_fvars_tbl : expr' list -> (string * 'a) list`
    This function takes a list of expr's generated by the semantic analyzer and outputs an association list representing the free-variables table.
    Note that the use type variable ='a= allows you flexibility in the way you define the addresses of the free-variables.
  - `generate : (constant * ('a * string)) list -> (string * 'a) list -> expr' -> string`
    This function takes the constants table, the free-variables table and a single expr' and returns a string containing the assembly code representing the evaluation of the expr'.
- in `compiler.ml`:

  - `get_const_address : constant -> string`
    This function takes a single constant and returns a string representing its absolute address.
  - `get_fvar_address : string -> string`
    This function takes a single free-variable name and returns a string representing its absolute address.
  - `primitive_names_to_labels : (string * string) list` is a mapping from free-variable names of primitive library procedures to their corresponding assembly code labels. This list already contains the mappings for the subset of the primitive standard library procedures which we have provided. You are encouraged to add mappings for the primitive standard library procedures you are required to implement yourselves (listed below).
  - `epilogue : string` should be a string containing the implementations of the primitive standard library procedures you are required to implement yourselves (listed in Section 4).

You are given `compiler.ml`, which takes the name of a Scheme source file (e.g., `foo.scm`). It then performs the following:

- Reads the contents of the Scheme source file into a string.
- Prepends the content of `stdlib.scm` to the string.
- Reads the expressions in the string, using the *reader* you wrote in *assignment 1*, returning a list of *sexprs*.
- Tags the list of *sexprs* using the *tag parser* you wrote in *assignment 2*, returning a list of *exprs*.
- Applies to each *expr* in the above list the `run_semantics` procedure you wrote in *assignment 3*, returning a list of *expr's*.
- Constructs the constants table, and the free-variables table.
- Calls `generate` to generate a string of x86-64 assembly instructions for every expr'.
- Adds a *prologue* and *epilogue* to the string, so that it becomes a self-contained assembly language program.
- Outputs the string containing the assembly language program.

You are given a file `compiler.s` that defines the data types for implementing the Scheme objects, and an assembly-language routine `write_sob_if_not_void`, that prints the scheme-object pointed to by the address in `rax` to *stdout*.

In addition, we provide you with a `Makefile` which takes the name of the input file without the extension; Assuming you run the `Makefile` with the argument `foo`, the following steps are performed:

- `compiler.ml` is applied to `foo.scm` which is assumed to exist, and the output is stored in `foo.s`
- `nasm` is applied to `foo.s`, outputting a 64-bit object file named `foo.o`

- `gcc` is used to link `foo.o` with some functions found in the standard C library, producing the executable file `foo`.

The resulting executable should run under Linux, and print to *stdout* the values of each of the expressions in the original Scheme source file.

4 Run-time support

Certain elementary procedures need to be available for the users of your compiler. Some of these standard library ("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), `-` (variadic), `boolean?`, `car`, `cdr`, `char->integer`, `char?`, `cons`, `eq?`, `equal?`, `integer?`, `integer->char`, `length`, `list` (variadic), `list?`, `make-string`, `make-vector`, `map` (variadic), `not`, `null?`, `number?`, `pair?`, `procedure?`, `float?`, `set-car!`, `set-cdr!`, `string-length`, `string-ref`, `string-set!`, `string?`, `symbol?`, `symbol->string`, `vector`, `vector-length`, `vector-ref`, `vector-set!`, `vector?`, `zero?`.

We have provided you with the implementation of many of these procedures, either as primitive assembly implementations (which can be found in the file `prims.s`) or as high-level scheme implementations (which can be found in the file `stdlib.scm`). We strongly encourage you to look through and understand the provided implementations.

Some of the built-in procedures are left for you to implement as primitive assembly procedures: `apply` (variadic), `car`, `cdr`, `cons`, `set-car!`, and `set-cdr!`.

4.1 A note about floating point numbers

Chez Scheme supports multiple numeric representations that are not part of your compiler. As a result, the implementations of the arithmetic functions in your compiler are slightly different than in Chez:

- Adding/subtracting/multiplying/dividing two integers should return an integer.
  - This means that division of integers should return the whole part of the result, and discard the remainder, e.g. `(equal? (/ 7 2) 3)` should return `#t`.
- Adding/subtracting/multiplying/dividing two floats should return a float.
- Adding/subtracting/multiplying/dividing one integer and one float should return a float.

This behavior is already implemented in the built-in arithmetic procedures in the code your were given.

5 How we shall test your compiler

In this assignment, we will treat your compiler as a black box, using the provided `Makefile` as the interface. This means that we will input a text file and expect an executable, which we will run. We will not look at the results or behavior of any of the internal components of your compiler.

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

1. Run `foo.scm` through Chez Scheme, and collect the output in a list.
2. Run your compiler on `foo.scm`, obtaining an executable `foo`. We shall then execute `foo` and collect its output in a list.
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 `nasm` failed to assemble the assembly file, if `gcc` failed to link your file, if the resulting executable caused a segmentation fault, if your code generated unnecessary output, or if the `equal?` predicate in Chez Scheme returned anything other than `#t` when comparing the two lists.

Assuming your compiler is in `~/compiler` and you have a Scheme file to compile `~/foo.scm`, you can perform similar tests using the following shell command:

```
make -f ./compiler/Makefile foo;\
set o1=`scheme -q < foo.scm`; set o2=`./foo`;\
echo "(equal? '($o1) '($o2))" > test.scm;\
scheme -q < test.scm
```

The expected result is `#t`.

6 Submission Guidelines

In this course, we use the `git` DVCS for assignment publishing and submission. You can find more information on `git` at `https://git-scm.com/`.

To begin your work, pull the updated assignment template from the course website by navigating to your local repository folder from the latest assignment you submitted and executing the command: `git pull`

If you haven't work on any before starting this work, you can simply clone a fresh copy of the repository by executing the command:

`git clone https://www.cs.bgu.ac.il/~comp191/compiler`

This will create a copy of the assignment template folder, named `compiler`, in your local directory. The template contains four (11) files:

- `reader.ml`
- `tag-parser.ml`
- `semantic-analyser.ml`
- `pc.ml`
- `readme.txt`
- `prims.s`
- `compiler.s`
- `compiler.ml`
- `code-gen.ml`

- `stdlib.scm`
- `Makefile`

The `Makefile` is the interface file for your compiler. You are free to make changes to the `Makefile`, as long as it maintains the correct behavior — location-independence:

Recall the snippet from section 5:

```
make -f ./compiler/Makefile foo;\
set o1=`scheme -q < foo.scm`; set o2=`./foo`;\
echo "(equal? '($o1) '($o2))" > test.scm;\
scheme -q < test.scm
```

Our tests will be run in a similar manner: The makefile will be called from *outside* the `compiler` folder, and the executable file `foo` should be created (or moved) to the location from which the `Makefile` was called.

This behavior is non-trivial, since the compiler process defined in `compiler.ml` and `Makefile` load various files (such as `stdlib.scm`, `compiler.s`, `prims.s`) which are expected to be available.

If you break the location-independence of the `Makefile`, we will be unable to test and grade your assignment. Do not do this.

Other than breaking the interface, you are allowed to add any code and/or files you like.

Among the files you are required to edit is the file `readme.txt`.

The file `readme.txt` should contain

1. 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.
2. 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 *va'adat mishma'at*, in pursuit of disciplinary action.

Submissions are only allowed through the submission system.

You are required to submit a **patch file** of the changes you made to the assignment template. See instructions on how to create a patch file below.

You are provided with a structure test in order to help you ensure that our tests are able to run on your code properly. Make sure to run this test on your final submission.

6.1 Creating a patch file

Before creating the patch, review the change list and make sure it contains all the changes that you applied and noting more. Modified files are automatically detected by git but new files must be added explicitly with the 'git add' command:

```
git add -Av .; git commit -m "write a commit message"
```

At this point you may review all the changes you made (the patch):

```
git diff origin
```

Once you are ready to create a patch, simply make sure the output is redirected to the patch file:

```
git diff origin > compiler.patch
```

After submission (but before the deadline), it is strongly recommended that you download, apply and test your submitted patch file. Assuming you download `compiler.patch` to your home directory, this can be done in the following manner:

```
cd ~
git clone https://www.cs.bgu.ac.il/~comp191/compiler fresh_compiler
cd fresh_compiler
git apply ~/compiler.patch
```

Then test the result in the directory `fresh_compiler`.

Finally, remember that your work will be tested on lab computers only! We advise you to test your code on lab computers prior to submission!

7 Tests, testing, correctness, completeness

Please start working on the assignment ASAP! Please keep in mind that:

- We will not (and cannot) provide you with anything close to full coverage. It's your responsibility to test and debug and fix your own code! We're nice, we're helpful, and we'll do our best to provide you with many useful tests, but this is not and cannot be all that you rely on!
- We encourage you to contribute and share tests! Please do not share OCaml code.
- This assignment can be tested using OCaml and Chez Scheme. You don't really need more, beyond a large batch of tests...
- We encourage you to compile your code and test frequently.

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 files your work depends on 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 have points deducted. The graders are instructed not to accept any late corrections or re-submissions under any circumstances.

8.1 A final checklists

1. You completed the module skeleton provided in the `compiler.ml` and `code-gen.ml` files
2. Your compiler matches the behavior presented in class
3. The `Makefile` is location-independent
4. Your compiler runs correctly using OCaml, `nasm` and `gcc` on the departmental Linux image
5. You completed the readme.txt file that contains the following information: (a) Your name and ID (b) The name and ID of your partner for this assignment, assuming you worked with a partner. (c) A statement asserting that the code you are submitting is your own work, that you did not use code found on the internet or given to you by someone other than the teaching staff or your partner for the assignment.
6. You submitted you work in its entirety