The Final Project

Mayer Goldberg, Avi Hayoun, Shahaf Shperberg

December 21, 2020

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.
- Late submissions result in a reduction of 5 points per day up to 5 days, any submission after more than 5 days will be not accepted. A late submission is, **by definition**, *any time* past the official deadline, according to the departmental *submission system*, i.e. submission of this assignment at 14:01 at the day of the deadline would consider a day late.
- 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 Constructing the full compiler

We provide you with an extensive skeleton that contains many of the implementation details, signatures and auxiliary functions needed for various parts of the code generation and compilation processes. We suggest that you make use of this code, but do not require it; you are allowed to make any changes to the provided code/signatures, 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).

We are providing you with six files:

- `code-gen.ml`, which contains a suggested module structure for the code-generation phase of the compiler pipeline. The module in this file contains three functions to implement:

  - `make_consts_tbl : expr' list -> (constant * (int * string)) list`, which is meant to be used to generate a three-column constants table (const, relative address of the constant, and the assembly representation of the constant), as shown in class (see Chapter 6, slide 36).
  - `make_fvars_tbl : expr' list -> (string * int) list`, which is meant to be used to generate a two-column free variables table (variable name, relative address of the free variable), as shown in class (see Chapter 6, slide 63).
  - `generate : (constant * (int * string)) list -> (string * int) list -> expr' -> string`, which is meant to generate the assembly string representing a single `expr'`, using the constants table and the free variables table.

  The file itself contains detailed explanations of the module signature.

- `compiler.ml`, which combines the four parts of the compilation pipeline, and provides additional auxiliary functions and definitions for reading from files and outputting self-contained assembly programs. This file contains comments explaining the structure of the assembly. It also contains an association list (`primitive_names_to_labels`) that maps `fvar` names of primitive functions to their labels in the assembly generated by the module defined in `prims.ml` (see details for `prims.ml` below). Assuming you use the mechanisms in `compiler.ml`, you should add additional pairs for the low-level procedures you are required to implement (see 4.1).

- `compiler.s`, an assembly file containing various helper `nasm` macros to simplify the code generation process, data types for implementing the Scheme objects, along with the procedure `write_sob_if_not_void` that prints scheme-objects pointed to by `rax` to *stdout*.

  - A note on `write_sob_if_not_void`: When printing symbols, our printing routine will sometimes print the hexadecimal ascii code of the first character instead of the character itself. For example, the symbol `3a` will be printed as `\x33;a`, but the symbol `a3` will be printed as `a3`. This method of printing is largely compatible with Chez, with minimal differences, but does not really make a difference: `(equal? '\x33;a '3a)` yields `#t` in Chez.

- `prims.ml`, which contains code to generate the assembly implementations of many of the low-level standard library procedures. The generation of these low level procedures is explained in detail in comments in the file itself.

- `stdlib.scm`, a Scheme file containing implementations of many of the high-level standard library procedures

- A `Makefile` that adheres to the requirements detailed in Section 6.

3.1 The behavior of `compiler.ml`

When executing `compiler.ml`, the following steps are performed:

- The name of a Scheme source file to compile (e.g., `foo.scm`) is extracted from from the ocaml command line arguments array and stored in the variable `inflie`.

- The contents of `stdlib.scm` and `infile` are read into memory, and catenated into a single string `code`.

- `code` is processed by the *reader* you wrote in *assignment 1*, returning a list of *sexprs*.
- The list of *sexprs* is tagged by the *tag parser* you wrote in *assignment 2*, returning a list of *exprs*.
- Each *expr* in the is annotated by the `run_semantics` procedure you wrote in *assignment 3*, resulting in a list of *expr's*.
- The tables for constants and free-variables are constructed.
- `generate` is applied to each *expr'* , resulting in a list of snippets of x86-64bit assembly instructions.
- A call to a printing routine (`write_sob_if_not_void` implemented in `compiler.s`) is appended to each snippet. This facilitates the desired output format and behavior of the executables your compiler will generate.
- The assembly snippets are all catenated together into a single string and store in the variable `code_fragment`.
- A *prologue* is prepended and an *epilogue* is appended to `code-fragment`, resulting in a self-contained assembly language program.
- The self contained assembly program is printed to `stdout`.

3.2 The behavior of `Makefile`

The `Makefile` takes the name of an 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 (specifically, the VM image published on the course website), and print the values of each of the expressions in the original Scheme source file to *stdout*. Take care to make sure that your compiler does prints only the necessary values to `stdout`, and nothing to `stderr`.

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 implemented in assembly language. Others can be implemented in Scheme, and compiled using your compiler. The procedures your compiler shall support include:

`*` (variadic), `+` (variadic), `-` (variadic), `/` (variadic), `<` (variadic), = (variadic), `>` (variadic), `append` (variadic), `apply` (variadic), `boolean?`, `car`, `cdr`, `char->integer`, `char?`, `cons`, `cons*` (variadic), `denominator`, `eq?`, `equal?`, `exact->inexact`, `flonum?`, `fold-left`, `fold-right`, `gcd`

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

We have provided you with implementations for most of these procedures, either as primitive assembly implementations (which are generated by the code defined in `prims.ml`) 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.

4.1 Library functions you must implement

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!`.
  - We recommend that you add your implementations for these procedures to prims.ml. However, you may incorporate your implementations into the compiler however you like (stand-alone assembly file, OCaml string in a variable, etc.).
- As high-level Scheme procedures: `fold-left` (non-variadic!), `fold-right` (non-variadic!), and `cons*` (variadic).
  - We encourage you to use a folding mechanism to implement `cons*`.
  - `stdlib.scm` contains placeholders for these functions, which you must complete.

Note that many of the high-level library functions depend on `apply`, `car`, `cdr`, `cons`, `fold-left` and `fold-right`, and require them to be implemented in order to run (see section 4.2.5).

4.2 Library function we implemented for you

The implementations we provide can be ordered in a hierarchy based on the compiler features on which they depend:

4.2.1 Only depend on free variables and application

The low-level procedures, implemented in `prims.ml`, only depend on your compiler correctly handling free variables (in order to access the closures for these procedures) and regular application. These include:

`*` (binary version), `+` (binary version), `/` (binary version), `<` (binary version), = (binary version), `boolean?`, `char->integer`, `char?`, `denominator`, `eq?`, `exact->inexact`, `flonum?`, `gcd` (binary version), `integer->char`, `make-string` (binary version), `null?`, `pair?`, `procedure?`, `rational?`, `string-length`, `string-ref`, `string-set!`, `string?`, `symbol?`, `symbol->string`.

1. The low-level numeric procedures are implemented similarly to the OCaml arithmetic operators: there are two implementations for each operator, one that operates on pairs of rational numbers, and one that operates on pairs of floating-point numbers. To achieve the behavior of the scheme numeric operators (variadic and not type-restricted), we implement wrapper functions in the high-level library for each numeric operator. These wrappers check the types of the arguments, perform type conversions as needed, and dispatch the arguments to the correct low-level implementation.

Note that subtraction (`-`) has no low-level implementation, and is fully implemented in the high-level library, making use of the addition and multiplication operators (which do have low-level implementations). This is probably not the best decision in terms of performance, and would not be implemented this way in an industrial-grade compiler. However, it is a good example of making use of abstraction, which is a core concept in this course, so we chose to take the opportunity to demonstrate it here as well.

4.2.2 Additionally depend on `LambdaSimple`

The high-level library function `not` additionally depends on you correctly handling `LambdaSimple'` definitions.

4.2.3 Additionally depend on `LambdaOpt`

The high-level library function `list` additionally depends on you correctly handling `LambdaOpt'` definitions.

4.2.4 Additionally depend on correct environment expansion

The implementation of many of the high-level library functions depend on other library functions (both high- and low-level). A naïve implementation of such functions would leave the library implementation vulnerable to re-definitions of free variables. For example:

```
(define foo +)
(define x (lambda (x y) (foo x y)))
(x 2 5) ; => 7
(define foo *)
(x 2 5) ;=> 10
```

The closure `x` contains a direct reference to the free variable `foo`, and so once we redefine the value of `foo`, the behavior of `x` changes.

To protect our library functions from this problem, many high-level library functions store local copies of closures on which they depend, instead of looking up free variables upon executions. These functions include `integer?`, `number?`, and `zero?`.

4.2.5 Additionally depend on procedures you must implement

The implementations of the following high-level library functions additionally depend on you correctly implementing the procedures detailed in section 4.1: `*` (variadic), `+` (variadic), `-` (variadic), `/` (variadic), `<` (variadic), = (variadic), `>` (variadic), `append` (variadic), `equal?`, `gcd` (variadic), `length`, `make-string`, `map` (variadic), `string->list`.

4.3 A couple more words on compatibility with Chez

Our standard library implementation differs from Chez's in multiple ways, detailed later in this section.

Before diving into the specifics, we would like to stress that these incompatibilities are deliberate, and are present mainly to simplify the implementation of the compiler and/or of the standard library. We will work around such incompatibilities when we test your work.

4.3.1 Argument handling

The most obvious difference between our implementations and Chez's is that, unlike Chez, we do not handle invalid input values or argument counts gracefully: while Chez performs runtime type-checking and argument count validation, printing an informative error when needed, our implementations completely ignore such problems.

The effect of this implementation decision is that calling our library functions with an incorrect number of arguments or arguments of the wrong types may lead to unexpected results and/or segmentation faults.

4.3.2 The `rational?` type predicate

Applying `rational?` to a floating-point value almost always returns `#t` in Chez, while our implementation returns `#f` for all floating point numbers.

4.3.3 Floating-point accuracy

Chez's floating-point value implementation supports significantly more accuracy than our own implementation, due to (among other things) the fact that they support 128-bit representation of floats, while only have support for 64-bit representation. As a result, the rounding errors using our floating-point representation are much more severe than in Chez.

4.3.4 Infinity, NaN, and Division by zero

While Chez correctly handles NaN, positive and negative infinities in its floating-point representation, and division by zero in general, our implementation ignores these edge cases, and does not handle them correctly (or at all).

4.3.5 Support for the parallel-set form (`pset!`)

As you may recall from assignment 3, Chez does not support the `pset!` form, while your compiler is required to implement support for it. Note that there is not need to handle `pset!` specially, if you correctly implemented support for it as a macro in assignment 2.

4.3.6 Addresses of constant values

Recall that Chez does not construct a constants table in in the compiler, and does not maintain a memory segment for constant values like in your compiler. As a result, comparing identical constant values by address (using `eq?`) will often return `#f`, where your compiler should return `#t`. For example, the expression `(eq? '(1 . 2) '(1 . 2))` will evaluate to `#f` (since two separate constant pairs are created in memory), but should evaluate to `#t` when running the executable file generated by your compiler (since there should be only one `'(1 . 2)` in memory).

All of the incompatibilities detailed above are deliberate, and are present mainly to simplify the implementation of the compiler and/or of the standard library. We will work around such incompatibilities when we test your work.

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 provide a text file containing valid code, and expect as output

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 `test0.scm`, we will do three things:

1. Run `test0.scm` through Chez Scheme, and collect the output in a list.
2. Run your compiler on `test0.scm`, obtaining an executable `test0`. We shall then execute `test0` 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?` in Chez Scheme (Chez's implementation of `equal?`, not that in your `stdlib.scm`).

If the `equal?` predicate returns `#t`, you get a point. Otherwise, you don't. You could lose a point if your code OCaml (*reader*, *tag-parser*, *semantic-analyser*, *code-generator*) processed the input incorrectly, if `nasm` failed to assemble the assembly file, if `gcc` failed to link your file, if the resulting executable caused a segmentation fault, if the resulting executable 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 worked on any assignments 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/~comp211/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.ml`
- `compiler.s`
- `compiler.ml`
- `code-gen.ml`
- `stdlib.scm`
- `Makefile`

The `Makefile` is the interface file for your compiler. It is implemented to be location-independent; 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 in (or moved to) the location from which the `Makefile` was called. This behavior is non-trivial, since the compilation process defined in `compiler.ml` and `Makefile` load various files (such as `stdlib.scm`, `compiler.s`, etc.) which are expected to be available.

Your compiler will be tested using a **fresh** copy of `Makefile`. **Any changes you make to `Makefile` will be discarded!**

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/~comp211/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 using the VM image published on the course website! We advise you to test your code on this image prior to submission!

7 Tests, testing, correctness, completeness

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

- We encourage you to contribute and share tests! Please do not share OCaml code.
- This assignment can be tested using OCaml, Chez Scheme and GNU Make. You don't really need more, beyond a large batch of tests…
- We encourage you to compile and test your work 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 (or an equivalent alternative)
2. Your compiler matches the behavior presented in class
3. The `Makefile` is location-independent
4. Your compiler runs correctly using OCaml, `nasm` , `gcc` and Make 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