# Final Project (Due: TBA (12:00pm (צהריים), TBA, 2018)

Mayer Goldberg

January 6, 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 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

- Please read this document completely, from start to finish, before beggining work on the assignment.

## 2 Changes to this document

-

## 3 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.

## 4 Writing the code generator

For this problem you are asked to implement the two 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 x86 architecture, such that when the execution of these instructions is complete, the value of `e` should be in the *result register* `RAX`.

The procedure `compile-scheme-file` takes the name of a Scheme source file (e.g., `foo.scm`), and the name of a x86 assembly target file (e.g., `foo.s`). 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, using the *reader* you wrote in *asssignment 1*, returning a list of *sexprs*.
- Applies to each *sexpr* in the above list the following, in order:
  - `parse`
  - `remove-applic-lambda-nil`
  - `box-set`
  - `pe->lex-pe`
  - `annotate-tc`

  This process is demonstrated in the `pipeline` function below. You may use it as is, or implement your own pipeline.
- Constructs the constants table, the symbol table (linked-list), and the global variable table.
- Calls `code-gen` to generate a string of x86 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 `RAX`) if it is not the *void object*.

  - You shall be given a file that defines the data types for implementing the Scheme objects (other than *symbols*, which you shall have to implement on your own), and an assembly-language routine `write_sob`, that takes a single argument off of the stack, and displays its printed representation on *stdout*. This file is written in the 64-bit layer of the Intel x86 assembly language, which we encourage you to use in your final project. You may use this file, but are not obligated to do so. If you choose to use the 32-bit layer of the Intel x86 assembly language, you will need to re-design the layout for the various data types that constitute Scheme objects, as well as use `EAX` as the result register (instead of `RAX`).

- 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 assemble using `nasm` (the *Newtide Assembler* you used in the course *architecture and systems programming lab*, which is a pre-requisite to this course).

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

You are free to generate combination of x86 instructions, but we advise you to follow the outline of the code generator presented in class.

```
(define pipeline
  (lambda (s)
    ((star <sexpr>) s
     (lambda (m r)
       (map (lambda (e)
              (annotate-tc
               (pe->lex-pe
                (box-set
                 (remove-applic-lambda-nil
                  (parse e))))))
         m))
     (lambda (f) 'fail))))

(define file->list
  (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)))))))
        (run)))))
```

### 4.1 The target language

The target language, i.e., the language that your code generator will output, is the Intel x86 assembly language. We encourage you to use the 64-bit layer, and to this end, we are providing you with definitions for the relevant Scheme objects (with the exception of *symbols*, which shall have to implement on your own), and the `write_sob` procedure written in the 64-bit layer, to give you a head start on testing your compiler.

You are not obligated, however, to use our code, or our data structures, or even to use the 64-bit layer: You may modify the code and the definitions, write your own, and use the 32-bit layer if you so please.

Do keep in mind that the calling conventions for system calls and C functions for the 64-bit layer are different from the corresponding conventions for the 32-bit layer.

## 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` (*not* variadic), `<` (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` (variadic), `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

In this assignment, we will treat your compiler as a black box. 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 intermediate steps.

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

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

The expected result is `#t`.

# 7 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, clone the assignment template from the course website:

```
git clone https://www.cs.bgu.ac.il/~comp181/assignments/project
```

This will create a copy of the assignment template folder, named `project`, in your local directory. The template contains two (2) files:

- Makefile (the assignment interface)
- readme.txt

The file `Makefile` is the interface file for your assignment. The definitions in this file will be used to test your code. If you do not implement the required functionality described below, we will be unable to test and grade your assignment.

You are allowed to add any code and/or files you like to your final submission.

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 ,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.

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 any instructions precisely. Specifically, before you submit your final version, please take the time to make sure your code loads and runs properly in a fresh Scheme session.

### 7.1 The Makefile - your project's interface

You are required to submit a **Makefile** as the interface of your compiler. A blank one is provided in the initial git repository. You must fill in the details.

The Makefile you submit should fully encapsulate your compiler; the way we will test your project is by running your Makefile, passing it a Scheme file name as a parameter. For example, if we want to compile a file called `bob.scm`, we will run the command `make bob.scm` on the command line. The expected output is an executable named `bob`.

In order to support this functionality, your Makefile must perform multiple steps for a given filename:

- Apply your compiler to the input file, producing an assembly file
- Apply `nasm` to the assembly file, producing an object file
- Apply `gcc` to the object file, producing an executable file

In order to support arbitrary filenames in your Makefile, we recommend looking into using the `%` symbol (explained in GNU Make pattern rules) and the GNU Make automatic variables, specifically you might find `$@` helpful.

### 7.2 Creating a patch file

Before creating the patch review the change set 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 for submission, simply make sure the output is redirected to the patch file:

```
git diff origin > project.patch
```

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

```
cd ~
git clone https://www.cs.bgu.ac.il/~comp181/assignments/project fresh_project
cd fresh_project
git apply ~/project.patch
```

Then test the result in the directory `fresh_project`.

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!

## 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 a `patch` file, if files are missing, if functions don't work as they are supposed to, if `readme.txt` or the statement asserting authenticity of your work are 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.