Final Project (Due: Wednesday at 12:00pm (צהריים), January 12, 2023)

Mayer Goldberg

December 21, 2022

Contents

1 General **1**

2 Introduction **2**

3 The files for the project **2**

4 The constants table **3**

5 The free-variables table **3**

6 The `CODE_GENERATION` signature **4**

7 Code generation **4**

8 Setting up the time-time environment **5**

9 How to compile and run code in Scheme **6**

10 Dynamically-allocated memory **9**

11 Final Words **11**

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 Introduction

This is the final project in the *Compiler-Construction Course*. At the end of this project, you should have a working compiler that can compile a large subset of Scheme into Intel x86, 64-bit assembly language, and then, using the Newtide Assembler (nasm), assemble it into a standalone executable.

This was a rather large project for me, as I had to write it all from scratch. You, however, are receiving the vast majority of the code, with a small number of places left for you to implement. What is left for you to do is not long, nor difficult, but you shall need to be able to write (one procedure) in assembly language, and write code to generate assembly language in other places.

You should start working on the project as soon as possible. I have already *closed the cycle* for you, meaning, that once you complete the constants and free-vars tables, you should be able to start compiling simple expressions. This was done intentionally, so as to encourage you to run the compiler as early as possible, and to test often. If you do not test often, it shall be very difficult for you to detect and fix bugs. Ideally, you should test after each change to the code. If you do this, and your code is faulty, you shall have a pretty good idea where the problem lies. If, however, you wait until the end of the project to start compiling and debugging, you most likely shall fail to meet the deadlines. When you debug "at the end", it is harder to find the errors, because there are more untested areas in your program, where the bugs can hide.

This project **must** be done under linux. Make sure you have linux installed and working, including the full, working toolset of *gcc*, *make*, *nasm*, *ocaml*, *Chez Scheme*.

3 The files for the project

You are give the following files *to be added* to your code:

- `code-gen.ml`: This is the additional code in ocaml you are getting. This code implements the constants table, the free-variables table, the interface for the `Code_Generation` module, the `Code_Generation` module itself, and many support functions. This is where most of your work shall take place. You may combine it with your *reader*, *tag-parser*, and *semantic analyser* code, into a single file called `compiler.ml`, or you may prefer to split your code among several files, but the graders shall expect to find a working `compiler.ml` that can be loaded into ocaml for the purpose of testing your compilers, so make sure it is available.
- `prologue-1.asm`: This is a template header for the assembly language generated by your compilers. It includes the definitions of the run-time type-information (RTTI), and various macros that make writing the compiler much easier. You should famililarize yourself with the code.

- `prologue-2.asm`: This is some more boilerplate code that is sandwiched in side the start of your assembly-language output. Nothing very interesting going on in this file.
- `epilogue.asm`: This is the file that ends the `main` function, where compiled user-code is located, followed by over 1600 lines of assembly-language routines. The **only** thing you shall need to add to this file is the implementation for the procedure `apply`, located at the label `L_code_ptr_bin_apply`, with a comment `;;; PLEASE IMPLEMENT THIS PROCEDURE` just above it. This is the **only** assembly-language routine you shall need to write for this project. Of course, if you have the time and the inclination, you are welcome to write more...
- `makefile`: This is the *makefile* that I use to assemble the code generated by the compiler. Please use it often (hint!), but otherwise leave it unchanged.
- `init.scm`: This a Scheme source file that is loaded by your compiler, ahead of any user-code, and provides some additional built-in procedures. The lower-level procedures are implemented in assmebly-language and are located in the `epilogue.asm` file. But anything that can be implemented practically in Scheme, really should be, and this file contains over 90 procedures that were implemented in Scheme.

In total, the language you are implementing supports over 130 different procedures, most of which are imeplemented in Scheme.

4 The constants table

Constructing the *constants table* requires coördination among several procedures:

- `remove_duplicates`: A general utility procedure that is used both for the *constants table* as well as the *free-variable table* (see next section). The purpose is to take a list of objects and remove duplicate items, result in what amounts to a *finite set*. Recall that while the same constants may appear several times in the Scheme source, each constant appears only once in the *constants table*. This implies *maximal sharing* of constant data.
- `collect_constants`: This routine runs recursively over an `expr'`, collecting all constants.
- `add_sub_constants`: This routine takes a list of `sexpr`, where each constant appears only once, and returns a larger lists, where each sub-constant appears before the constant itself. This list may [again] contain duplicates, and is *topologically sorted*. A second application of `remove_duplicates` should *still* result in a *topologically sorted* list.
- `search_constant_address`: This procedure searches the address of a constant in the constant table. It's used both by the procedure that constructs the constant table as well as by the code generator.
- `make_constants_table`: This is the actual procedure that constructs the *constants table* and that coördinates the above procedures.

5 The free-variables table

You are given most of the code for constructing the *free-variables table*, however, you shall need to complete one procedure: `collect_free_vars`. As its name suggests, the procedure runs recursively over the structure of `expr'`, collecting all free variable names.

6 The `CODE_GENERATION` signature

The code$_{\text{generation}}$ module contains many procedures, but as the final state in the compiler, it has nothing to export other than the interface to the completed compiler. I provided you with two functions that represent two ways to run the compiler:

```
module type CODE_GENERATION =
  sig
    val compile_scheme_string : string -> string -> unit
    val compile_scheme_file : string -> string -> unit
  end;;
```

- `compile_scheme_file`: This function takes two strings, `file_in` (the Scheme source file) and `file_out`, the name of the assembly-language file that contains the output of the compiler. It reads the definitions in `file_in`, appends them to the contents of the `init.scm` file, reads, tag-parses, analyses, and generates code for these expressions, and places the assembly instructions in the file `file_in`.
- `compile_scheme_string`: This function takes two strings, `file_out`, the name of the assembly-language file that contains the output of the compiler, and `user`, which contains Scheme expressions to be compiled on-the-fly. This function is intended for testing and debugging whilst you are working on the project. You would not ordinarily compile and run Scheme code in this way, but it is very convenient during the development cycle.

7 Code generation

The heart of this project is the `code_gen : expr' list -> string`. This procedure takes a list of `expr'` and returns a string of assembly code that evaluates and prints these expressions. `code_gen` contains two mutually-recursive procedures, `run : expr' -> string` and `runs : expr' list -> string`. The procedure `run` recurses over the structure of `expr'`, generating snippets of code for sub-expressions, and combining them to get the code that corresponds to the original expression. The `runs` procedure drives `run` and creates code for lists of expressions.

In class, when going over the code-generation for applications, we described two ways of cleaning up the stack upon return from a non-tail-call:

1. The caller is responsible (C calling-conventions)
2. The callee is responsible (Pascal calling-conventions)

The slides describe the cleanup as the responsibility of the caller, following the conventions of C. In class, however, I mentioned that the Intel x86 architecture supports both conventions, and that I thought that it would be simpler to support procedures with variable arity as well as the tail-call optimization if we adhered to the Pascal calling-conventions.

When writing the code-generator, I adhered to the Pascal conventions, and indeed, just as I had expected, the experience was nearly bug-free and absolutely painless! So the code for this project does not correspond to the slides when it comes to cleaning the stack, and adheres instead to the Pascal conventions discussed during the lectures. Please check the recordings if you need to refresh your memory.

Essentially, x86 supports two forms of the `ret` (*return from subroutine*) instruction:

1. `ret`, which pops off the return address and jumps to it. This makes it harder for the *callee* to clean up the stack, leaving the job to the *caller* (C convention)
2. `ret` *`bytes`*, where the `ret` instruction takes an immediate (literal) operand that denotes how many bytes to pop off the stack *after* performing the return. This makes it easier for the *callee* to cleanup the stack (Pascal convention)

The macro

```
%define AND_KILL_FRAME(n) (8 * (2 + n))
```

makes it especially easy for the *callee* to clean up the stack, by specifying the number of parameters the function uses (excluding the *argument count* and the *lexical environment*!).

For example, the procedure `cons` takes two arguments. So the definition of `cons` ends with `ret AND_KILL_FRAME(2)`, which should be read as *return and kill the frame, which consists of 2 parameters*. The *envrironment* and the *arguments count* are included auto-magically:

```
L_code_ptr_cons:
        ENTER
        cmp COUNT, 2
        jne L_error_arg_count_2
        mov rdi, (1 + 8 + 8)
        call malloc
        mov byte [rax], T_pair
        mov rbx, PARAM(0)
        mov SOB_PAIR_CAR(rax), rbx
        mov rbx, PARAM(1)
        mov SOB_PAIR_CDR(rax), rbx
        LEAVE
        ret AND_KILL_FRAME(2)
```

8 Setting up the time-time environment

The variable `global_bindings_table` contains the bindings between the names of primitive Scheme procedures that are hand-written in assembly language, and the actual assembly-language labels where these procedures are defined in the file `epilogue.asm`. For example, in the `global_bindings_table` you shall see the line `("cons", "L_code_ptr_cons");`. What this line means is that the Scheme procedure `cons` is hand-written in assembly language after the label `L_code_ptr_cons` (in the `epilogue.scm` file). This binding is necessary for creating a *closure-object* for `cons` and placing it in the corresponding free variable that shall go in the `.bss` section:

```
free_var_13: ; location of cons
        resq 1
```

Before any of either the user-code or the code in `init.scm` gets evaluated, `cons` shall need to be available for use. During the initialization stage, a closure shall be created, the address of which shall be placed in `free_var_13`:

```
; building closure for cons
mov rdi, free_var_13
mov rsi, L_code_ptr_cons
call bind_primitive
```

The procedure that does all the heavy lifting here in `bind_primitive`, which is defined in the `epilogue.asm` file:

```
;;; rdi: address of free variable
;;; rsi: address of code-pointer
bind_primitive:
        ENTER
        push rdi
        mov rdi, (1 + 8 + 8)
        call malloc
        pop rdi
        mov byte [rax], T_closure
        mov SOB_CLOSURE_ENV(rax), 0 ; dummy, lexical environment
        mov SOB_CLOSURE_CODE(rax), rsi ; code pointer
        mov qword [rdi], rax
        LEAVE
        ret
```

9 How to compile and run code in Scheme

I set up a directory called `testing`, where all the chaos of temporary files can be contained without messing up the source files for the compiler. Suppose I want to evaluate the Scheme code: `(equal? '(a (b . c) d) '(a (b . c) d))`. I don't need to place these definitions into a Scheme source-file. Rather, I can compile from a string that contains the Scheme code:

```
utop[4]> Code_Generation.compile_scheme_string
  "testing/goo.asm"
  "(equal? '(a (b . c) d) '(a (b . c) d))";;
!!! Compilation finished. Time to assemble!
- : unit = ()
```

At this point, I have a file `testing/goo.asm` that I need to assemble. Moving to the *bash* shell, I type `make goo`, and the *makefile* does the rest:

```
gmayer@lambda> make goo
nasm -g -f elf64 -l goo.lst goo.asm
gcc -g -m64 -no-pie -o goo goo.o
rm goo.o
```

I now have an executable `goo` that I can run at the shell:

```
gmayer@lambda> ./goo
#t

!!! Used 14178 bytes of dynamically-allocated memory
```

So our code returned `#t`.

The system reports that 14,178 bytes of dynamically-allocated memory were allocated by this program. What was all this memory used for?

The Scheme system provides over 130 procedures that form the run-time environment. Those are the builtin procedures. Creating closures for all of these takes up some memory, especially since most of these procedures are written in Scheme, and use several levels of nested lambda-expressions.

All of this memory represents a constant overhead. We can measure and prove this claim by evaluating an expression that clearly requires no additional resources:

```
utop[6]> Code_Generation.compile_scheme_string "testing/goo.asm" "#t";;
!!! Compilation finished. Time to assemble!
- : unit = ()
```

Followed by:

```
gmayer@lambda> make goo
nasm -g -f elf64 -l goo.lst goo.asm
gcc -g -m64 -no-pie -o goo goo.o
rm goo.o
gmayer@lambda> ./goo
#t

!!! Used 14178 bytes of dynamically-allocated memory
```

Sometimes, however, evaluating expressions shall consume additional resources. Consider 20! (the *factorial* of 20):

```
utop[7]> Code_Generation.compile_scheme_string "testing/goo.asm" "(fact 20)";;
!!! Compilation finished. Time to assemble!
- : unit = ()
```

Followed by

```
gmayer@lambda> make goo
nasm -g -f elf64 -l goo.lst goo.asm
gcc -g -m64 -no-pie -o goo goo.o
rm goo.o
gmayer@lambda> ./goo
2432902008176640000

!!! Used 86498 bytes of dynamically-allocated memory
```

Why such a great increase in memory? The answer is simple: We created plenty of intermediate values, every one of which is a Scheme-object, that takes up memory in the heap. We have no garbage-collector [yet], so this 86K represents one, giant memory-leak! Luckily, the memory was recovered when the process was terminated.

What about the more usual route of compile from a *source file*? Remember the procedure `describe`? It appears in the slides discussing the structure of sexprs. The procedure takes an sexpr and tells us how to generate this sexpr using elementary functions for creating pairs, lists, and vectors.

I placed the definition of `describe` in the source file `testing/goo.scm`, and used it to describe its own source code! This is the contents of the file `testing/goo.scm`:

```
(define describe
  (lambda (e)
    (cond ((list? e)
           `(list ,@(map describe e)))
          ((pair? e)
           `(cons ,(describe (car e))
                  ,(describe (cdr e))))
          ((vector? e)
           `(vector ,@(map describe s)))
          ((or (null? e) (symbol? e)) `',e)
          (else e))))

(describe
  '(define describe
     (lambda (e)
       (cond ((list? e)
              `(list ,@(map describe e)))
             ((pair? e)
              `(cons ,(describe (car e))
                 ,(describe (cdr e))))
             ((vector? e)
              `(vector ,@(map describe s)))
             ((or (null? e) (symbol? e)) `',e)
             (else e)))))
```

And here is how I compile the file:

```
utop[4]> Code_Generation.compile_scheme_file "testing/goo.scm" "testing/goo.asm";;
!!! Compilation finished. Time to assemble!
- : unit = ()
```

And here is how I assemble and run the executable:

```
gmayer@lambda> make goo
nasm -g -f elf64 -l goo.lst goo.asm
gcc -g -m64 -no-pie -o goo goo.o
rm goo.o
gmayer@lambda> ./goo
(list (quote define) (quote describe) (list (quote lambda) (list (quote e))
(list (quote cond) (list (list (quote list?) (quote e)) (list (quote
quasiquote) (list (quote list) (list (quote unquote-splicing) (list (quote
map) (quote describe) (quote e)))))) (list (list (quote pair?) (quote e))
(list (quote quasiquote) (list (quote cons) (list (quote unquote)
(list (quote describe) (list (quote car) (quote e)))) (list (quote unquote)
(list (quote describe) (list (quote cdr) (quote e))))))) (list (list (quote
vector?) (quote e)) (list (quote quasiquote) (list (quote vector) (list
(quote unquote-splicing) (list (quote map) (quote describe) (quote s))))))
(list (list (quote or) (list (quote null?) (quote e)) (list (quote symbol?)
(quote e))) (list (quote quasiquote) (list (quote quote) (list (quote
unquote) (quote e))))) (list (quote else) (quote e)))))

!!! Used 22091 bytes of dynamically-allocated memory
```

10 Dynamically-allocated memory

If you examine the snippets of code in the code-generator, or if you read over the assembly code in `epilogue.asm`, you will notice that there are plenty of calls to `malloc`. But if you examine carefully the `extern` functions, namely the functions with which the code links, you shall not find `malloc`. This definition appears in `prologue-2.asm`, as well as in any executable generated by our compiler:

```
extern printf, fprintf, stdout, stderr, fwrite, exit, putchar
global main
section .text
```

So it should be clear to you that we are *not* linking with the C-library function `malloc`, and we must have our own!

If you examine the `epilogue.asm` file, you shall see the following code:

```
section .bss
memory:
        resb gbytes(1)

section .data
top_of_memory:
        dq memory

section .text
malloc:
        mov rax, qword [top_of_memory]
        add qword [top_of_memory], rdi
        ret
```

This raises several questions:

- Why did I write my own `malloc` of sorts?
- Can `malloc` really be *that simple*??
- How do I ever `free` memory?

The projects in previous years did use the C-library `malloc` function. Students reported that they ran into severe memory shortages rather quickly. At first, we suspected this was due to the lack of a *garbage-collector* in our project: If we are not reclaiming unused memory, surely we shall run out of it at some point... This is obviously true, but the memory shortages students reported were unreasonably severe even for a system that had no *garbage-collector*.

The problem lies with `malloc` itself. When you ask `malloc` to allocate memory, it generally allocates [a lot] more memory than you might imagine. If you ask for n bytes, `malloc` shall, in fact, allocate $2^k \cdot \mathit{BlockSize}$ bytes of memory, for the smallest k that satisfies your request.

So what is the value of *BlockSize*? This is easy to find:

```
section .data
fmt:
        db `%ld\n\0`

extern printf, malloc
global main
section .text
main:
        enter 0, 0

        mov rdi, 1
        call malloc
        push rax
        mov rdi, 1
        call malloc
        pop rsi
        neg rsi
        add rsi, rax
        mov rdi, fmt
        mov rax, 0
        call printf

        leave
        ret
```

And now we can assemble and run our code:

```
gmayer@lambda> make foo
nasm -g -f elf64 -l foo.lst foo.asm
gcc -g -m64 -no-pie -o foo foo.o
rm foo.o
gmayer@lambda> ./foo
32
```

So *BlockSize* equals 32 bytes.

What this teaches us is that `malloc` is wasteful for small requests. If we wish to conserve our memory, we need to allocate a lot of memory at once and parcel it out on our own. This is precisely what I had done in my code.

The `prologue-1.asm` file contains definitions for memory sizes:

```
%define bytes(n) (n)
%define kbytes(n)  (bytes(n) << 10)
%define mbytes(n)  (kbytes(n) << 10)
%define gbytes(n)  (mbytes(n) << 10)
```

I allocate 1 gigabyte of memory in the `.bss` section, and keep a pointer to the top of it in the `.data` section. My implementation of `malloc` is trivially simple because it only parcels out memory and never attempts to `free` it again. Had I needed to `free` memory, I would have had to implement some bookkeeping algorithm, such as the one invented by Knuth and used in the C-library implementation of `malloc`. But since we are not going to `free` any memory, we might as well not have to pay the bookkeeping-fees in the difference between what we request and $2^k \cdot \mathit{BlockSize}$...

11 Final Words

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.