Compilers: Backend to Frontend
and Back to Front Again

Abdulaziz Ghuloum

September 17, 2006

Inside Cover

Inside Title Page

Copyright Page

Preface

Compilers are percieved to be magical artifacts, carefully crafted by the wizards, and unfathomable by the mere mortals. This paper attempts to dispel this myth. We build a simple compiler for a simple language in a step-by-step fashion. The input language accepted by the compiler starts minimal, and grows as our knowledge of how to build compilers grows. The final language is almost Scheme.

Although the compiler is written in the Scheme programming language, only minimal knowledge of Scheme is required. Essentially, the reader is assumed to be comfortable reading and writing recursive Scheme functions to the level presented in *The Little Schemer*. Additionally, we recommend the freely available tutorial *Teach Yourself Scheme in Fixnum Days* for people familiar with other programming languages but not Scheme. *The Scheme Programming Language* is an invaluable resource for understanding Scheme's semantics. You will find it most useful when you give up in thinking how `list?` detects circular data structures.

Our compiler targets the Intel-386 architecture, the dominant architecture for personal computing. The output of our compiler is assembly code that can be assembled by `gas`, the GNU assembler, which is freely available for most operating systems. No knowledge of assembly language or the Intel-386 architecture is assumed beyond the basics: binary numbers, memory layout, and basic pointers. If you are familiar with arrays in `C`, and know how the bit-level operations (*and*, *or*, *xor*, and *not*) work, then you're good to go.

Enjoy!

Contents

Chapter 1

Basic Concepts

in Code Generation [FIXME: Explain what this chapter is all about, what the reader will encounter, and what will be accomplished by the end.]

1.1 Integers

We start our journey by writing the simplest compiler ever. Our compiler takes a small integer (fixnum) as input and produces a native-code program that prints that integer. This input language follows the grammar listed in the following "Input Language" box.

Input Language:

$$\langle \text{Expr} \rangle \rightarrow \text{fixnum}$$

A fixnum is an integer than can be represented on the target machine. On a 32-bit architecture such as the Intel-386, the largest number that can be represented immediately cannot use more than 32 bits of data. For signed numbers represented in two's compliment notation, the range of numbers that fit in 32-bits is $-2^{31} \leq n \leq 2^{31} - 1$. Examples of binary representation of numbers are:

$$\begin{array}{rr}
0 = 00\cdots0000_b & -1 = 11\cdots1111_b \\
1 = 00\cdots0001_b & -2 = 11\cdots1110_b \\
2 = 00\cdots0010_b & -3 = 11\cdots1101_b \\
3 = 00\cdots0011_b & -4 = 11\cdots1100_b \\
4 = 00\cdots0100_b & -5 = 11\cdots1011_b \\
\vdots & \vdots \\
2^{31} - 1 = 01\cdots1111_b & -2^{31} = 10\cdots0000_b
\end{array}$$

Now we turn to the implementation. We will write a simple runtime system in `C` to serve as a scaffold for debugging our compiler. The `C` runtime will execute the assembly code and prints a human-readable output. Suppose that the entry point to our assembly-code is called `scheme_entry`. A simple `C` program `runtime.c` (figure 1.4, p. 9) serves our purpose.

We need to test that our runtime actually works before we proceed to writing our compiler. We test it by writing a `scheme_entry` function in `C`. Here are the contents of the file `ctest.c`:

```
int scheme_entry(){
  return 7; /* should print 7 */
}
```

Let's test it from the unix command line using the GNU C compiler, gcc:

```
$ gcc -Wall ctest.c runtime.c -o test
[bunch of warnings]
$ ./test
7
$
```

Now that we know that our runtime is at least capable of printing 7, let's see how gcc did it. We can get gcc to produce not only the final executable, but also the intermediate assembly code produced. Let's try it and see what comes out:

```
$ gcc --omit-frame-pointer -S ctest.c
$ cat -n ctest.s | expand -t 4 | sed 's/ *//'
1       .file   "ctest.c"
2       .text
3   .globl scheme_entry
4       .type   scheme_entry, @function
5   scheme_entry:
6       movl    $7, %eax
7       ret
8       .size   scheme_entry, .-scheme_entry
```

The meaning of each like is as follows: Line 1 lists the source file name. Line 2 starts the text section. We won't get into what this means now except to say that this is a segment in memory where the compiled code will reside when loaded. Line 3 says that the label scheme_entry should be considered a global label, and should be included in the symbol table so that the linker/loader can find it. Line 4 gives a type to the label: @function in this case. Line 5 is the label of the entry point to our function. Line 6 loads the value of the constant 7 into the register %eax The register %eax serves as the *return value register*. Line 7 causes execution to return from the function thus the caller now has the value 7 as the result of calling the function. Line 8 records the size of the function. We note that the lines 1 and 8 are optional.

The only character that was actually interesting in the whole file was 7. The rest can be treated as boilerplate code by our compiler. All we have to do is write a program that when given a fixnum value as input, it will print out this output after filling out the number.

To facilitate automated testing of our first compiler and runtime, we include a `test-driver.scm` file and a test suite of some input programs along with their expected output. Our compiler is a function `emit-program` of one argument: the input program. All it has to do is print the assembly-code similar to the one listed above. In order to direct the output of the compiler to the appropriate file, the function `emit` that is supplied by the driver must be used for printing. Our incomplete attempt is listed in figure 1.3, p. 8. The test suite provided attempts to target common pitfals and corner cases. It is by no means comprehensive. The reader is urged to extend the test-suite regularly by writing tests that specifically target areas of weaknesses in the compiler. Running the tests should produce the following to assure us that all is correct:

```
> (test-all)

Performing immediates tests
~~~~~~~~~~~~~~~~~~~~~~~~~~~
[1/10]: 0 ... ok
...
[10/10]: -536870912 ... ok
Passed all 10 tests
```

Exercises

1. Ensure that the compiler listed in figure 1.1, p. 5 and the `C` runtime work on your computer. Make sure your compiler passes all the tests provided in the file `tests-1.1-req.scm`. It's a good idea to weed out any warnings `gcc` may emit.

```
(load "test-driver.scm")
(load "tests-1.1-req.scm")

(define (emit-program x)
  (unless (integer? x) (error ---))
  (emit "    .text")
  (emit "    .globl scheme_entry")
  (emit "    .type scheme_entry, @function")
  (emit "scheme_entry:")
  (emit "    movl $~s, %eax" x)
  (emit "    ret"))
```

Figure 1.1: Compiler for integers.

```
#include <stdio.h>

int main(int argc, char** argv){
  printf("%d\n", scheme_entry());
  return 0;
}
```

Figure 1.2: Runtime system for the integers compiler.

1.2 Immediate Constants

Our next compiler extends the first compiler by adding a variety of immediate values. A valid input to our compiler is either a small integer (fixnum), a boolean value (`#t` or `#f`), a character (`#\A`, `#\7`, etc.), or the empty list `()` as listed in the following language:

Input Language:

$$\langle \text{Expr} \rangle \rightarrow \text{fixnum} \mid \text{boolean} \mid \text{char} \mid \text{null}$$

Because the output of the compiler must print the final value, it must have a way of distinguishing the fixnums from the other immediates. There are many techniques for representing the type of objects at runtime. One technique that we use is called the "tagged pointer representation." A value in this representation is divided into two parts: tag and content. The tag represents the type of the objects, and the content represents its value. We choose the following convention for our tags:

- Fixnums always have `#b00` in the two least-significant bits. The most-significant 30-bits represent the actual value of the fixnum.
- All other immediates have `#b1111` in the four least-significant bits.
- The other unused tags will be used at later parts of the compiler to represent pairs, vectors, strings, symbols, etc.

Reserving two bits for the fixnum tag limits the range of numbers that can be represented by a fixnum. With 30-bits of data, our range becomes $-2^{29} \leq n \leq 2^{29} - 1$ or $-536870912 \leq n \leq 536870911$. We accept this limitation for our first compiler and fix it at later time, when we get around to implementing arbitrary-precision arithmetic.

The boolean values `#f` and `#t` will be represented as the boolean numbers 00101111_b and 01101111_b. We have chosen two numbers that differ by one bit only to make distinguishing booleans from other objects easy. An object can be masked with 10111111_b to determine if these bits are 00101111_b.

Characters can be represented using immediate values by using two bytes: one byte to represent the tag (00001111_b) of a character and another

byte to represent its value. For example, the character `#\A` has the ASCII value 65. Therefore, it is represented as an immediate by first shifting the number 65 eight bits to the left, then tagging it with the tag value 15 to produce 16655[1].

Scheme's empty-list object (a.k.a. `()`, nil, null) can be represented by the value 00111111_b.

Exercises

1. Complete the `C` runtime and the Scheme compiler. Make sure your compiler passes all the tests provided in the file `tests-1.2-req.scm`. It's a good idea to weed out any warnings `gcc` may emit.

[1] Most implementations of Scheme use the ASCII numerics for their characters. Check your implementation's manual to see if the function `char->integer` provides the actual ASCII value of a character.

```
(load "test-driver.scm")
(load "tests-1.2-req.scm")
(load "tests-1.1-req.scm")

(define fxshift        2)
(define fxmask      #x03)
(define bool_f      #x2F)
(define bool_t      #x6F)
(define wordsize       4) ; bytes

(define fixnum-bits (- (* wordsize 8) fxshift)

(define fxlower (- (expt 2 (- fixnum-bits 1))))

(define fxupper (sub1 (expt 2 (- fixnum-bits 1))))

(define (fixnum? x)
  (and (integer? x) (exact? x) (<= fxlower x fxupper)))

(define (immediate? x)
  (or (fixnum? x) (boolean? x) ---))

(define (immediate-rep x)
  (cond
   [(fixnum? x) (ash x fxshift)]
   ---))

(define (emit-program x)
  (unless (immediate? x) (error ---))
  (emit "    .text")
  (emit "    .globl scheme_entry")
  (emit "    .type scheme_entry, @function")
  (emit "scheme_entry:")
  (emit "    movl $~s, %eax" (immediate-rep x))
  (emit "    ret"))
```

Figure 1.3: Compiler for immediate constants.

```
#include <stdio.h>

/* define all scheme constants */
#define bool_f               0x2F
#define bool_t               0x6F
#define fx_mask              0x03
#define fx_tag               0x00
#define fx_shift                2

/* all scheme values are of type ptrs */
typedef unsigned int ptr;

static void print_ptr(ptr x){
  if((x & fx_mask) == fx_tag){
   printf("%d", ((int)x) >> fx_shift);
  } else if(x == bool_f){
   printf("#f");
  }
  ---
  } else {
   printf("#<unknown 0x%08x>", x);
  }
  printf("\n");
}

int main(int argc, char** argv){
  print_ptr(scheme_entry());
  return 0;
}
```

Figure 1.4: A simple `runtime.c` file is used to run the assembly code generated by our compiler and print out the value returned.

1.3 Unary Primitives

The compiler we've written so far accepts a language composed of immediate constants only. Hopefully you have completed the exercises at the end of section 1.2 and your compiler can compile fixnums, booleans, null, and characters to assembly code. Now we turn our attention to extending the language to include some unary primitives. Unary primitives are those that accept one argument. The input language to our compiler is now:

Input Language:

⟨Expr⟩ → ⟨Imm⟩
| (*prim* ⟨Expr⟩)
⟨Imm⟩ → fixnum | boolean | char | null

Some of the unary primitives we need to support are: `fxadd1`, `fxsub1`, `char->fixnum`, `fixnum->char`, `fxzero?`, `null?`, `not`, `fixnum?`, `boolean?`, and `char?`.

Since there are many different primitives that we have to deal with now and many more that we will deal with later on, it makes sense to use some abstractions that simplify our code. First, we recognize that the language we are accepting is growing. So, we need to deal with the types of expressions that we deal with in a modular fashion. Instead of writing one function `emit-program` like we did in section 1.2, we break the function into two parts: one that emits the code for an arbitrary expression, and one that inserts the function stub around it. The following implementation of `emit-program` calls `emit-expr` which produces the code resposible for placing the result in the return-value register `%eax`.

For every case in our input language we place a corresponding case in the `emit-expr` function. Two cases are listed in the code: one for immediates and one for primitive calls. The immediate case is similar to the one from the section 1.2. The primitive call case first checks that the number of arguments passed to the primitive is correct before emitting the code for the primitive call.

The information associated with every primitive must be stored somewhere in the compiler. Given a name, we might want to ask the following questions about it: (1) Is it a primitive? (2) How many arguments does it take? and (3) What procedure handles emitting the code for the primitive?

In the compiler listed in figure 1.5, p. 14, we chose to store this information in the property list of the symbol denoting the primitive. The three properties `*is-prim*`, `*arg-count*`, and `*emitter*` answer the three previous questions. Other ways of storing the information are possible including association lists, hash tables, etc. Using property lists is perhaps the simplest and most efficient representation for our purpose.

Our compiler infrastructure is almost complete. Its code is listed in figure 1.5, p. 14. We are now ready to tackle the problem of implementing the primitives of our language. The first primitive that we tackle is `fxadd1`. The primitive `fxadd1` takes one argument, which must evaluate to a fixnum, and returns that value incremented by 1. The implementation of `fxadd1` should first emit the code for evaluating the argument. Evaluating that code at runtime would place the value of the argument at the return-value register `%eax`. The value placed in `%eax` should therefore be incremented and the new computed value should be placed back in `%eax`. Remember though that all the fixnums in our system are shifted to the left by two. So, a `fxadd1` instruction translates to an instruction that increments `%eax` by 4. To add this primitive to our compiler, we use the `define-primitive` syntactic form as follows:

```
(define-primitive (fxadd1 arg)
  (emit-expr arg)
  (emit "    addl $~s, %eax" (immediate-rep 1)))
```

The machine instruction `addl` in this usage takes a constant value and a register and increments the value of that register by that constant value.

The implementation of the primitive `char->fixnum` should evaluate its argument, which must evaluate to a character, then convert the value to the appropriate fixnum. Since we defined the tag for characters to be 00001111_b and the tag for fixnums to be 00_b, it suffices to shift the character value to the right by six bits to obtain the fixnum value. The primitive `fixnum->char` should shift the fixnum value to the left, then tag the result with the character tag. Tagging a value is performed using the instruction `orl`. The code for implementing the primitive `fixnum->char` follows:

```
(define-primitive (fixnum->char arg)
  (emit-expr arg)
  (emit "    shll $~s, %eax" (- charshift fxshift))
  (emit "    orl $~s, %eax" chartag))
```

All the primitives so far were implemented using simple arithmetic and bit-twiddling operations. Implementing predicates such as `fixnum?` is not as simple. First, after the argument to `fixnum?` is evaluated, the lower two bits of the result must be extracted and compared to the fixnum tag 00_b. If the comparison succeeds, we return the true value, otherwise we return the false value. Extracting the lower bits using the fixnum mask is done using the bitwise-and instructions `and`/`andl`[2]. The result is compared with the fixnum tag using the `cmp`/`cmpl` instruction. The Intel-386 architecture provides many instructions for conditionally setting the lower half of a register by either a 1 or a 0 depedning on the relation of the objects involved in the comparison. One such instruction is `sete` which sets the argument register to 1 if the two compared numbers were equal and to 0 otherwise. A small glitch here is that the `sete` instruction only sets a 16-bit register. To work around this problem, we use the `movzbl` instruction that sign-extends the lower half of the register to the upper half. Since both 0 and 1 have 0 as their sign bit, the result of the extension is that the upper bits will be all zeros. Finally, the result of the comparison is shifted to the left by an appropriate number of bits and or'ed with the false value 00101111_b to obtain either the false value or the true value 01101111_b.

```
(define-primitive (fixnum? arg)
  (emit-expr arg)
  (emit "    and $~s, %al" fxmask)
  (emit "    cmp $~s, %al" fxtag)
  (emit "    sete %al")
  (emit "    movzbl %al, %eax")
  (emit "    sal $~s, %al" bool-bit)
  (emit "    or $~s, %al" bool-f))
```

[2]Instructions prefixed with the letter `l` operate on 32-bit values while the instructions lacking the letter `l` operate on 16-bit values.

The remaining predicates can be implemented by following the previous examples.

Exercises

1. Finish the implementation of the compiler code presented in figure 1.5, p. 14.

2. Add all the missing primitives: `fxsub1`, `fixnum->char`, `null?`, `not`, `boolean?`, and `char?`. The primitive `not` takes any kind of value and returns `#t` if the object is `#f`, otherwise it returns `#f`. Make sure your compiler passes all the tests in `tests-1.3-req.scm`.

3. Implement the primitive `fxlognot` which takes a fixnum value and returns a fixnum with all the bits flipped. Pay special attention to preserving the 00_b tag for the resulting fixnum. Add the appropriate test cases.

```
(define-syntax define-primitive
  (syntax-rules ()
   [(_ (prim-name arg* ...) b b* ...)
    (begin
      (putprop 'prim-name '*is-prim* #t)
      (putprop 'prim-name '*arg-count*
        (length '(arg* ...)))
      (putprop 'prim-name '*emitter*
        (lambda (arg* ...) b b* ...)))]))

(define (primitive? x)
  (and (symbol? x) (getprop x '*is-prim*)))

(define (primitive-emitter x)
  (or (getprop x '*emitter*) (error ---)))

(define (primcall? expr)
  (and (pair? expr) (primitive? (car expr))))

(define (emit-primcall expr)
  (let ([prim (car expr)] [args (cdr expr)])
    (check-primcall-args prim args)
    (apply (primitive-emitter prim) args)))

(define (emit-expr expr)
  (cond
   [(immediate? expr) (emit-immediate expr)]
   [(primcall? expr)  (emit-primcall expr)]
   [else (error ---)]))

(define (emit-program expr)
  (emit-function-header "scheme_entry")
  (emit-expr expr)
  (emit "    ret"))
```

Figure 1.5: Compiler framework for modular primitives

1.4 Conditional Expressions

The next extension to the language is to add conditional expressions. A conditional expression is a three-part `if` expression. The first part is the test, and is evaluated first. If the value of the test is not `#f`, then the second part, the *consequent* branch, is evaluated and its value is returned. Otherwise, the third part, the *alternate* branch, is executed. The input language to our next compiler is as follows:

Input Language:

⟨Expr⟩ → ⟨Imm⟩
| `(if` ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩`)`
| `(`*prim* ⟨Expr⟩`)`
⟨Imm⟩ → fixnum | boolean | char | null

In order to perform conditional execution, we need a way of telling the machine what piece of code needs to be executed. Labels are used to mark the targets of jump instructions. We have already seen one label in our previous compiler: the `scheme_entry` label. To create more labels that are all unique, we use the simplest scheme. All local labels would have the form `L_0`, `L_1`, etc.. We define the procedure `unique-label` which returns a new label (a string) everytime it is called.

```
(define unique-label
  (let ([count 0])
    (lambda ()
      (let ([L (format "L_~s" count)])
        (set! count (add1 count))
        L))))
```

To emit the code for the form `(if test conseq altern)`, first we emit the code for evaluating the `test`. Evaluating the `test` would place some value in the `%eax` register. We compare that value to the value of `#f`, and if they're equal, we jump to some label, `alt-label`, where the code for the `altern` branch is placed. If the value of `%eax` is not `#f`, then we let the execution proceed to the code of the `conseq` branch. When the `conseq`

branch finally finishes, the execution must skip over the code of the `altern` branch to the join point (`end-label`).

The first change to our compiler is to make it recognize the `if` as a new form. We modify the function `emit-expr` to allow it to handle `if` expressions as follows:

```
(define (emit-expr expr)
  (cond
   [(immediate? expr) (emit-immediate expr)]
   [(if? expr)        (emit-if expr)]
   [(primcall? expr)  (emit-primcall expr)]
   [else (error ---)]))
```

Next, we implement the procedure that handles the `if` form as follows:

```
(define (emit-if expr)
  (let ([alt-label (unique-label)]
        [end-label (unique-label)])
    (emit-expr (if-test expr))
    (emit "    cmp $~s, %al" bool-f)
    (emit "    je ~a" alt-label)
    (emit-expr (if-conseq expr))
    (emit "    jmp ~a" end-label)
    (emit "~a:" alt-label)
    (emit-expr (if-altern expr))
    (emit "~a:" end-label)))
```

The new machine instructions introduced by this code are (1) `je`, which performs a jump if the two compared values are equal, and (2) `jmp`, which performs an unconditional jump.

Exercises

1. Complete the missing functionality from your compiler and ensure that it passes all the tests in `tests-1.4-req.scm`.

2. In addition to the `if` construct that was just added to our language, many languages provide other forms of conditional expressions. The Scheme form `(and test* ...)` is often implemented as a syntactic transformer as follows:

```
(define-syntax and
  (syntax-rules ()
   [(and) #t]
   [(and test) test]
   [(and test test* ...)
    (if test (and test* ...) #f)]))
```

   Add support to the `and` form to your compiler. Once `and` is working, implement the `(or test* ...)` form.

3. Our basic treatment of conditional expressions is correct but inefficient in many cases. For example, `(if (fxzero? e0) e1 e2)` performs two comparisons: one compares the value of `e0` to the constant `0` and builds the appropriate boolean value. The other compares the resulting boolean value to `#f` to dispatch the appropriate branch. The reason for this inefficiency is that we treated predicates as value primitives even though we were not interested in obtaining the boolean value. How would you modify your compiler to minimize the number of comparisons performed?

1.5 Binary Primitives

So far, all of the constructs of the language we compile were simple enough that we actually only needed one register, `%eax`, to evaluate them. In this section, the input language becomes more interesting by including binary primitives. Binary primitives take not one but two arguments. Here is what our new language looks like:

Input Language:

⟨Expr⟩ → ⟨Imm⟩
| `(if` ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩`)`
| `(`*prim* ⟨Expr⟩`)`
| `(`*prim* ⟨Expr⟩ ⟨Expr⟩`)`
⟨Imm⟩ → fixnum | boolean | char | null

Some of the binary primitives we need to support at this stage are: `fx+`, `fx-`, `fxlogand`, `fxlogor`, `fx=`, `fx<`, `fx<=`, `fx>`, `fx>=`, `char=`, `char<`, `char<=`, `char>`, and `char>=`.

Suppose we would like to compile the following program:

```
(fx+ (fx- (fx- 30 3) 3) (fx- 6 5))
```

In compiling the call to the primitive `fx+`, we can choose to emit the code for the left subexpression, which we hope will compute the value 24 and place it in the `%eax` register. We are stuck now since evaluating the right subexpression will overwrite the value in `%eax` by the value 1. What we need is a way to save the value of the first computation somewhere, evaluate the second computation, then use the saved value to compute the final result.

We use the stack data structure (figure 1.6, p. 19) to hold the intermediate values of computations. The stack is an array of adjacent memory cells. Each cell in the stack holds one value. We dedicate one of the machine's general purpose registers, `%esp`, to hold a pointer to base of the stack. The cell at the base of the stack holds a value, called the *return point*. We won't have to deal with it until section 1.7. We just mention it here so that you don't accidentally overwrite it. So, the first available cell is located one machine word (4 bytes) above the base of the stack. We assume that the

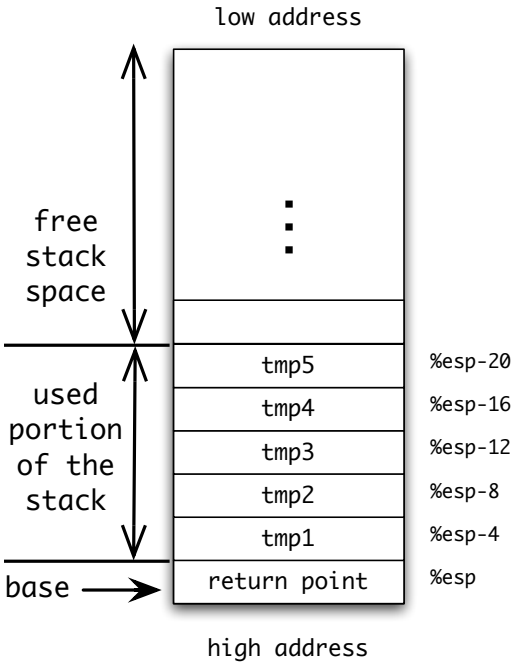


Figure 1.6: Illustration of the stack data structure. The base of the stack is held constant in the `%esp` register. Individual values that are saved on the stack are accessed by their relative offset from the base of the stack.

stack is large enough to hold all intermediate values. The issue of stack overflow is saved for later parts of the tutorial.

To implement binary primitives in our compiler, we need to make a few changes. All of the functions that emit code need to be modified to take an extra argument: the stack index. This index is a non-negative integer that tells us how much of the stack is used and thus cannot be modified. Therefore, everytime we need to save a new value, we would know at which offset the value can be saved without clobbering any other saved values. The index is moved before we emit code for the other subexpression to ensure that its evaluation won't clobber our saved value.

Notice that the stack base pointer, `%esp`, is held at a static position at runtime. The *stack index* is the dynamic value that changes at *compile time* to allocate and deallocate space on the stack. Therefore, stack allocation is achieved by simple bookkeeping at compile-time (figure 1.7, p. 22).

Now let's see how the primitive `fx+` is actually implemented. The primitive is defined to take three arguments: the stack index, and the two operands to `fx+`. It evaluates one of the operands, saves its value, evaluates the second operand (with a decremented index), then adds the saved value to the result of the second operand.

```
(define-primitive (fx+ si arg1 arg2)
  (emit-expr sc arg1)
  (emit "    movl %eax, ~s(%esp)" si)
  (emit-expr (- si wordsize) arg2)
  (emit "    addl ~s(%esp), %eax" si))
```

Handling primitives of higher arities is not different from handling binary primitives. The first argument is evaluated and saved in a stack location; the `si` is moved to preserve the argument; and the process continues for all arguments. Once all arguments are evaluated, we can access their values by the indices at which they were saved.

Creating the stack

The discussion of stacks so far assumed that the base of the stack was in the `%esp` register. Now we show the changes we need to make to our `C` runtime and our compiler to properly initialize the value of the `%esp` register.

From the `C` side, we need to allocate a sufficiently large area of memory to serve as our stack. The code in figure 1.8, p. 23 illustrates the basic idea. The procedure `allocated_protected_space` allocates a block of memory surrounded by two protected pages. These protected pages aid us by crashing the program early should it try to access the memory locations immediately below or above the stack. A pointer to the base of the stack is passed to the `scheme_entry` procedure.

At the `scheme_entry` side, our stack base argument will be located at `%esp`+8 (`C` uses `%esp` as a stack pointer that moves dynamically as you add and remove things to it). So, the first thing we need to do is save the value of the `C` stack pointer into one of the scratch registers: `%ecx`. We then call a label that will perform the actual computation. Once execution returns, we restore the `C` stack pointer back from `%ecx` then return to the `main` procedure. Here is how our assembly code for `scheme_entry` should look like:

```
scheme_entry:
   movl %esp, %ecx
   movl 4(%esp), %esp
   call L_scheme_entry
   movl %ecx, %esp
   ret
```

Exercises

1. Modify your `startup.c` file so that the `main` procedure allocates a stack and passes it to `scheme_entry` as described in this section. Also, modify your compiler so that it outputs the modified `scheme_entry` code. Make sure that your compiler still works on all of the previous tests.

2. Implement all the binary primitives listed at page 18. Make sure your compiler passes all the tests in `tests-1.5-req.scm`.

3. Implement the primitive `fx*`. Remember that all our numbers are scaled by 4 and that the result of the multiplication should also be scaled by 4. Which of the following two equations would you use?

$$4xy = \frac{(4x \cdot 4y)}{4}$$

$$4xy = \left(\frac{4x}{4}\right) \cdot 4y$$

   Make sure you add enough tests to ensure that your implementation of `fx*` is correct. Add a test that makes one of the two equations fail and the other succeeds.

4. In our implementation of `fx+`, we saved the value of the first operand before evaluating the second operand. However, in many cases, one of the operands may be a constant. If one of the arguments is a constant, we can evaluate the other argument first, then add its value to the constant operand. Think about how you would go about implementing such optimization.

```
(define-syntax define-primitive
  (syntax-rules ()
   [(_ (prim-name si arg* ...) b b* ...)
    (begin
      (putprop 'prim-name '*is-prim* #t)
      (putprop 'prim-name '*arg-count*
        (length '(arg* ...)))
      (putprop 'prim-name '*emitter*
        (lambda (si arg* ...) b b* ...)))]))

(define (emit-primcall si expr)
  (let ((prim (car expr)) (args (cdr expr)))
    (check-primcall-args prim args)
    (apply (primitive-emitter prim) si args)))

(define (emit-expr si expr)
  (cond
   [(immediate? expr) (emit-immediate expr)]
   [(if? expr)        (emit-if si expr)]
   [(primcall? expr)  (emit-primcall si expr)]
   [else (error ---)]))

(define (emit-program expr)
  (emit-function-header "L_scheme_entry")
  (emit-expr (- word-size) expr)
  (emit "    ret")
  (emit-function-header "scheme_entry")
  (emit "    movl %esp, %ecx")
  (emit "    movl 4(%esp), %esp")
  (emit "    call L_scheme_entry")
  (emit "    movl %ecx, %esp")
  (emit "    ret"))
```

Figure 1.7: Compiler framework for modular primitives. The compiler keeps tracks of passing the stack index, `si`, to all the primitive emitters.

```
static char* allocate_protected_space(int size){
  int page = getpagesize();
  int status;
  int aligned_size = ((size + page - 1) / page) * page;
  char* p = mmap(0, aligned_size + 2 * page,
                 PROT_READ | PROT_WRITE,
                 MAP_ANONYMOUS | MAP_PRIVATE,
                 0, 0);
  if (p == MAP_FAILED){ --- }
  status = mprotect(p, page, PROT_NONE);
  if(status != 0){ --- }
  status = mprotect(p + page + aligned_size, page, PROT_NONE);
  if(status != 0){ --- }
  return (p + page);
}

static void deallocate_protected_space(char* p, int size){
  int page = getpagesize();
  int status;
  int aligned_size = ((size + page - 1) / page) * page;
  status = munmap(p - page, aligned_size + 2 * page);
  if(status != 0){ --- }
}

int main(int argc, char** argv){
  int stack_size = (16 * 4096);  /* holds 16K cells */
  char* stack_top = allocate_protected_space(stack_size);
  char* stack_base = stack_top + stack_size;
  print_ptr(scheme_entry(stack_base));
  deallocate_protected_space(stack_top, stack_size);
  return 0;
}
```

Figure 1.8: The new runtime system creates a stack surrounded by two protected pages to guard against stack overflow and underflow.

1.6 Local Variables

Now that we know how to use the stack to save intermediate values of a computation, we can generalize the concept by giving names to these intermediate values. We extend the language of our compiler once more by adding two new syntactic forms: the binding construct `let` and variable references. The ellipsis '`...`' in the grammar below denote repetition. Whenever the form '*x* `...`' appears, it means that zero or more of *x* are accepted. It is equivalent to the Kleene-star construct of regular languages.

Input Language:

⟨Expr⟩ → ⟨Imm⟩
| *var*
| `(if` ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩`)`
| `(let ([`*var* ⟨Expr⟩`] ...)` ⟨Expr⟩`)`
| `(`*prim* ⟨Expr⟩ `...)`
⟨Imm⟩ → fixnum | boolean | char | null

The construct `let` consists of a list of bindings `([`*var* ⟨Expr⟩`] ...)` and a body expression. Evaluating a `let` expression starts with evaluating every expression appearing on the right-hand-side of the bindings list. Once these expressions are evaluated, the variables on the left-hand-side are bound to their corresponding values. The variables bound by a `let` become visible in the body of the `let` only. They are not visible in the right-hand-side expressions since the binding of the variables happens only *after* all the right-hand-side expressions are evaluated.

In order to implement `let` in our compiler, we need to maintain an environment that maps local variables to the location in which their values are saved. Because all our variables are saved on the stack, our environment maps every variable to its stack index. Among possible representations for the environment, we will utilize association lists (alists for short) because of their simplicity.

Again, we need to make a few changed to our compiler. Just like we added the stack index as an extra argument to all of our procedures, we now need to add an environment. We also need to add the extra clauses for `let` and *var* to our `emit-expr` procedure.

```
(define (emit-expr si env expr)
  (cond
   [(immediate? expr) (emit-immediate expr)]
   [(variable? expr)  (emit-variable-ref env expr)]
   [(if? expr)        (emit-if si env expr)]
   [(let? expr)       (emit-let si env expr)]
   [(primcall? expr)  (emit-primcall si env expr)]
   [else (error ---)]))
```

The procedure `emit-let` takes a `let` expression and emits the code for evaluating all the right-hand-sides of the bindings, saving the values, extending the environment, and processing the body in the new environment. The value of the `let` expression is the value returned by the body of the `let`. Below is the code for `emit-let`. Notice that as we progress in writing the compiler, we use more abstractions to make the code more readable and less dependent on the representation.

```
(define (emit-let si env expr)
  (define (process-let bindings si new-env)
    (cond
     [(empty? bindings)
      (emit-expr si new-env (let-body expr))]
     [else
      (let ([b (first bindings)])
        (emit-expr si env (rhs b))
        (emit-stack-save si)
        (process-let (rest bindings)
           (next-stack-index si)
           (extend-env (lhs b) si new-env)))]))
  (process-let (let-bindings expr) si env))
```

Variables are easy to handle now. When we need to emit the code for a variable reference, all we have to do is lookup that variable's stack index in the environment and emit a load from that location.

```
(define (emit-variable-ref env var)
  (cond
   [(lookup var env) => emit-stack-load]
   [else (error ---)]))
```

Exercises

1. Finish the implementation of `let`. Make sure your compiler passes the tests in `tests-1.6-req.scm`.

2. The Scheme standard does not specify the order of evaluation for the right-hand-side expressions of a `let`. The standard provides another binding form, `let*`, that evaluates all of its bindings sequentially and makes the bound variables available in all subsequent bindings in addition to the body of the `let*`. For example, the on the left below evaluates to 2 since the `x` in the inner `let` refers to the `x` bound by the outer `let`. The example on the right evaluates to 3 since the `x` in the right-hand-side of `y` is the `x` bound by the inner `let*`.

```
(let ([x 1])
  (let ([x (fx+ x 1)]
        [y (fx+ x 1)])
    y))
```

```
(let* ([x 1])
  (let* ([x (fx+ x 1)]
         [y (fx+ x 1)])
    y))
```

   Add support for `let*` to your compiler and test it using the file `tests-1.6-opt.scm`.

1.7 Procedures

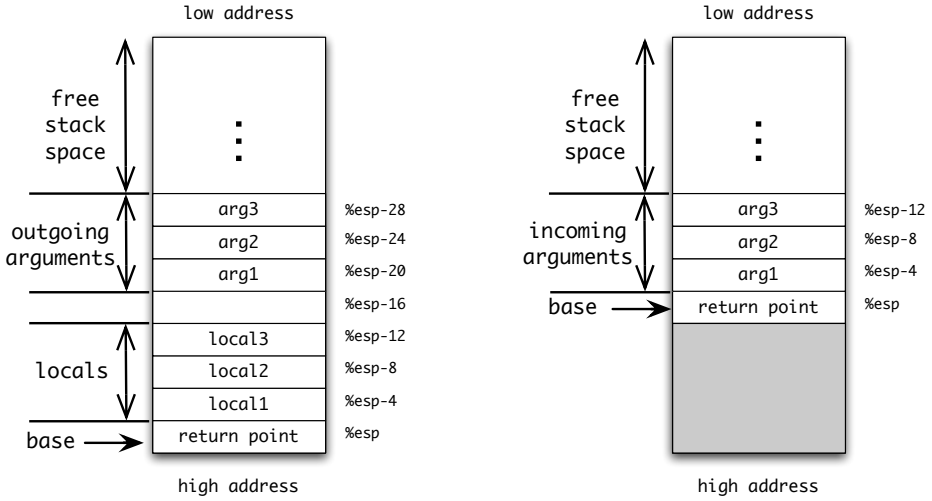
The compilers we have written up to the last section are a little more than a calculator. Every program they compiled was simple enough that we could have computed the result at compile time and replaced the program by its answer. Real programming languages have procedures. Procedures allow the programmer to abstract common computations into reusable components that are applied many times with different arguments. In this section, we extend our language by adding procedure declarations and procedure calls as shown in the grammar below.

Input Language:

⟨Program⟩ → ⟨Expr⟩
| `(letrec ([`*lvar* ⟨Lambda⟩`] ...)` ⟨Expr⟩`)`
⟨Lambda⟩ → `(lambda (`*var* `...)` ⟨Expr⟩`)`
⟨Expr⟩ → ⟨Imm⟩
| *var*
| `(if` ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩`)`
| `(let ([`*var* ⟨Expr⟩`] ...)` ⟨Expr⟩`)`
| `(app` *lvar* ⟨Expr⟩ `...)`
| `(`*prim* ⟨Expr⟩ `...)`
⟨Imm⟩ → fixnum | boolean | char | null

The language now contains an optional top-level `letrec` form which serves as a declaration for procedure names. The `letrec` form is similar in structure to the `let` form with the restriction that all the right-hand-sides of the bindings must be procedures. These procedures are created with the `lambda` form that includes the *formal arguments* and the body of the procedure.

The language also contain a `app` form. The first operand to a `app` is a variable which must be a name of a procedure bound by the top-level `letrec`. The other arguments are ordinary expressions, which must be evaluated first to obtain their values. We call these values the *actual* arguments. The procedure call binds the procedure's *formal* arguments to the the *actual* arguments with which the procedure was called. Once the execution of the body of the procedure is done, execution returns to the `app` that invoked the procedure, and the value of the `app` becomes the value of the body of the procedure.



(A) Caller's View (B) Callee's View

Figure 1.9: The view of the stack from (A) the Caller's side right before making the call, and (B) the Callee's side on entry to the procedure.

The caller and the callee must agree on where the parameters will be placed. To simplify matters, we choose to pass all arguments on the stack. The caller evaluates all of its arguments and saves them sequentially on the stack, leaving one cell empty for the return point (figure 1.9 above). The caller then adjusts the base pointer to make sure that the callee receives the base pointer adjusted to its base. Once the procedure returns, the caller moves the base pointer back to where it was before making the call. Two machine instructions are used for making procedure calls: `call` and `ret`:

- The `call` instruction performs the following: (1) computes the return point (i.e. the address of the instruction following the `call` instruction), (2) decrements the value of `%esp` by 4, (3) saves the return point at `0(%esp)`, then (4) directs the execution to the target of the call.

- The `ret` instruction performs the following: (1) it loads the return point address from `0(%esp)`, (2) increments the value of `%esp` by 4, then (3) directs the execution to the return point.

Now we show how to emit code for `letrec`, `lambda`, and `app` forms.

First, processing the `(letrec ([`*lvar* ⟨Lambda⟩`] ...)` ⟨Expr⟩`)` form must first create a set of labels, one for each of the *lvar*s. The procedure `unique-label` can be used to create these labels. Once we have the labels, we create the initial environment which maps the *lvar*s to their corresponding labels. The code for each `lambda` is emitted next followed by the code for the body of the `letrec`.

```
(define (emit-letrec expr)
  (let* ([bindings (letrec-bindings expr)]
         [lvars (map lhs bindings)]
         [lambdas (map rhs bindings)]
         [labels (unique-labels lvars)]
         [env (make-initial-env lvars labels)])
    (for-each (emit-lambda env) lambdas labels)
    (emit-scheme-entry (letrec-body expr) env)))
```

Handling a `(lambda (`*var* `...)` ⟨Expr⟩`)` form is tackled next. The `lambda` form must bind its formal arguments to the appropriate locations in which these formals arrive. We know that the return point will be located at `%esp`, so the formals will be located at indices $-4, -8, -12, \ldots$. Notice that our environment now has two different kinds of bindings: integers denoting stack locations and strings denoting `letrec`-bound labels.

```
(define (emit-lambda env)
  (lambda (expr label)
    (emit-function-header label)
    (let ([fmls (lambda-formals expr)]
          [body (lambda-body expr)])
      (let f ([fmls fmls] [si (- wordsize)] [env env])
        (cond
         [(empty? fmls) (emit-expr si env body)]
         [else
          (f (rest fmls)
             (- si wordsize)
             (extend-env (first fmls) si env))])))))
```

Finally, we have to emit code for the `(app` *`lvar`* ⟨Expr⟩ `...)` form. As stated earlier, a `app` must first evaluate all of its arguments saving them at the appropriate stack locations, then it must adjust the base pointer (`%esp`) before issuing the `call` instruction, and finally, it should readjust the base pointer to where it was before the `call` was issued.

```
(define (emit-app si env expr)
  (define (emit-arguments si args)
    (unless (empty? args)
      (emit-expr si env (first args))
      (emit-arguments (- si wordsize) (rest args))))
  (emit-arguments (- si wordsize) (call-args expr))
  (emit-adjust-base (+ si wordsize))
  (emit-call si (lookup (call-target expr) env))
  (emit-adjust-base (- (+ si wordsize))))
```

Exercises

1. Implement procedures in your compiler. Make sure you pass all the tests in `tests-1.7-req.scm` as well as all previous tests.

2. The Intel-386 architecture is pretty esoteric in what it does in the `call` and `ret` instructions. RISC architectures in particular do not provide such instructions; instead, they provide a `jmp`-like instruction only. How would you implement procedure calls and return using the `jmp` instruction only? Implement your idea and test it.

1.8 Iteration via Proper Tail Calls

The evaluation of procedures' bodies so far followed a simple code generation strategy: code for the body expression was emitted first (to place the result of the expression in the register `%eax`), followed by a call to the `ret` instruction. Let's inspect the the following program that sums all numbers from 0 to 10:

```
(letrec ([sum (lambda (n ac)
                (if (fxzero? n)
                    ac
                    (app sum (fxsub1 n) (fx+ n ac))))])
  (sum 10 0))
```

The code generated for the procedure `sum` looks similar to the following commented assembly code:

```
L_1:
    movl -4(%esp), %eax    /* load n                          */
    cmpl $0, %eax          /* compare it to 0                 */
    sete %al               /* if zero, set %al to 1, else 0   */
    movzbl %al, %eax       /* %eax is either 1 or 0           */
    sa $6, %al             /* %eax is now #x4F or 0           */
    or $0x2F, %al          /* %eax is #x2F (#f) or #x6F (#t)  */
    cmp $0x2F, %al         /* compare %eax to false           */
    je L_2                 /* goto altern branch              */
    movl -8(%esp), %eax    /* load ac                         */
    jmp L_3                /* goto end                        */
L_2:
    movl -4(%esp), %eax    /* load n                          */
    addl $-4, %eax,        /* subtract 1                      */
    movl %eax, -16(%esp)   /* save at arg1                    */
    movl -4(%esp), %eax    /* load n                          */
    addl -8(%esp), %eax    /* add to ac                       */
    movl %eax, -20(%esp)   /* save at arg2                    */
    addl $-8, %esp         /* adjust base pointer             */
    call L_1               /* call sum again                  */
    addl $8, %esp          /* readjust base pointer           */
L_3:
    ret                    /* return at end of procedure      */
```

The code generated consumes three cells of stack space at every recursive call to the function `sum`. What's more important is that after returning from the call to `sum`, there is nothing more to be done other than returning to the caller. Recognizing procedure calls whose value will only be returned and treating them in a way that does not leak any space is called *Proper Tail Calls*.

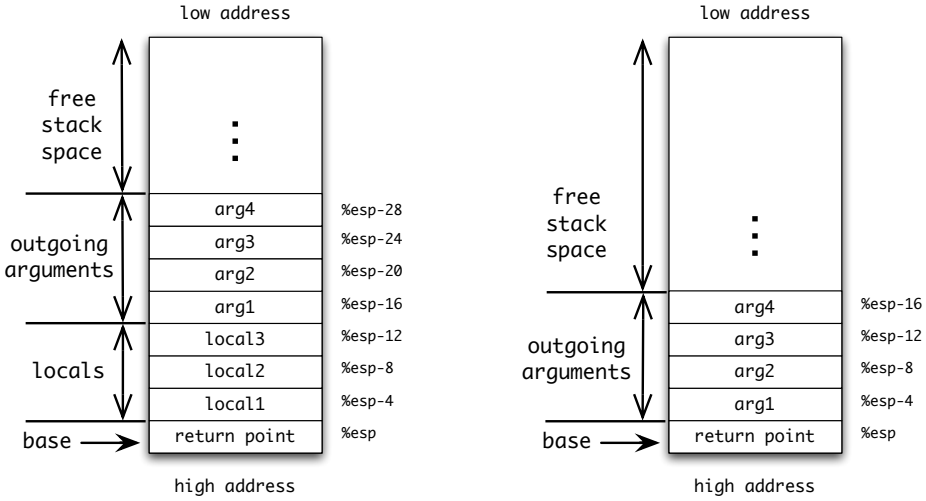
A procedure call is called a tail call if it occurs in a tail position. A tail position in our language is defined recursively as follows:

- The body of a procedure is in tail position.
- If a `let` expression is in tail position, then the body of the `let` is in tail position.
- If the conditional expression `(if test conseq altern)` is in tail position, then the `conseq` and `altern` branches are also in tail position.
- All other expressions are not in tail position.

We modify our compiler to recognize tail positions first. First, we add a new procedure, `emit-tail-expr`, which is called at the initial tail position (i.e. the body of a `lambda` form). The procedure `emit-tail-expr` is similar to `emit-expr`, except that it calls the appropriate tail procedures:

```
(define (emit-tail-expr si env expr)
  (cond
   [(immediate? expr) (emit-tail-immediate expr)]
   [(variable? expr)  (emit-tail-variable-ref env expr)]
   [(if? expr)        (emit-tail-if si env expr)]
   [(let? expr)       (emit-tail-let si env expr)]
   [(primcall? expr)  (emit-tail-primcall si env expr)]
   [(app? expr)       (emit-tail-app si env expr)]
   [else (error ---)]))
```

Immediates, variable references, and primcalls in tail position are treated similarly to their non-tail counterpart. The only difference is that a `ret` instruction must be emitted after emitting the code that evaluates the simple expression.



(A) Before Tail Call (B) At Tail Call

Figure 1.10: One way of implementing proper tail calls is by collapsing the tail frame. The figures show the placement of the arguments on the stack above the local variables (A), then moving the arguments down to overwrite the current frame immediately before making the tail jump (B).

The procedures `emit-tail-if` and `emit-tail-let` are also similar `emit-if` and `emit-let` except that they call `emit-tail-expr` instead of `emit-expr` for the expression subparts that are in tail position (i.e. the body of a `let` and the two branches of an `if`).

For procedure calls in tail position, something special must be done. Our simple strategy (illustrated in figure 1.10 above) works as follows:

- All arguments are evaluated in some order. The values of these arguments are saved at some stack locations. This is similar to how non-tail calls evaluated their arguments.
- Since the local variables are not needed after evaluating the arguments, all the arguments are moved down to stack locations adjacent to where the return-point is.
- Instead of emitting a `call` instruction, we emit a `jmp` instruction to the target of the call. A `jmp` instruction does not modify the base pointer, nor does it modify the return-point.

Exercises

1. The tests in `tests-1.8-req.scm` are tail-call intensive. Verify that running any of the tests on your previous compiler indeed exhausts the stack space and consequently crashes the program. Implement the basic algorithm for proper handling of tail calls. Test your compiler on the same test file. Note that some tests may take long time to terminate.

2. The basic algorithm, although correct, is naive and terribly inefficient. Suggest simple ways in which the basic algorithm may be improved. Look at the assembly output of some programs to get a feel for some of the useless work that the basic algorithm performs.

1.9 Heap Allocation

Our simple compiler so far supported only the simplest data forms: the immediates. We are now ready to add some data structures (pairs, vectors, and strings) along with the primitives needed to construct these data structures and access their data.

The first and simplest data structure to support is the pair. A pair is a data structure that holds two values (the `car` and `cdr` fields[3]). The primitive `cons` takes two arguments and constructs a new pair composed of these two values. The primitives `car`/`cdr` take a pair and return the value stored in the `car`/`cdr` field. The `pair?` predicate tests if its argument is a pair; and like all predicates, it returns `#t` or `#f` depending on whether the argument was a pair or not. Naturally, the pair data structures must hold its two values along with a type tag that distinguishes it from the immediates and the other data structures.

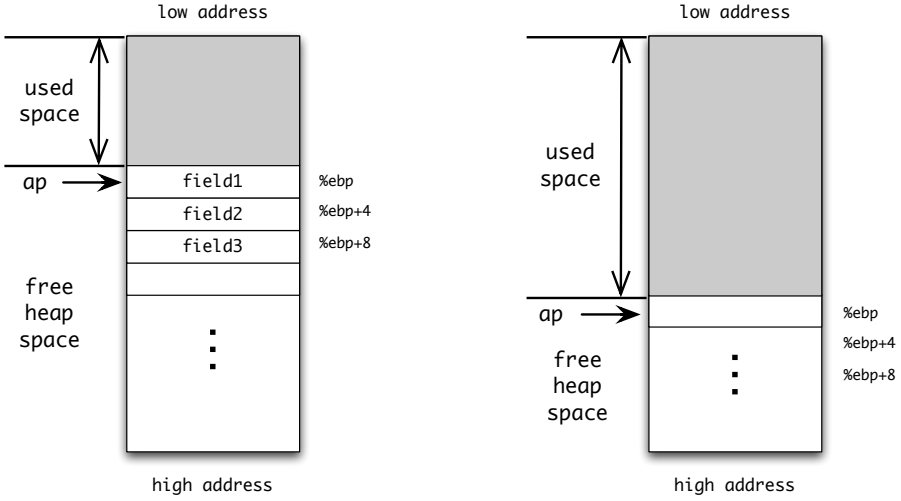
Several representations for pairs are possible. We choose a simple yet efficient representation. Two words of memory are allocated for the pair to hold its `car` and `cdr` fields. Remember that our fixnum and boolean values were represented by integers whose lower bits served as a tag and higher bits served as a content. We use a similar representation for pair pointers by tagging the address of the allocated memory with a *pair-tag*.

For tagged-pointers to work on pairs, we need a reliable way of dividing the 32-bit pointer into disjoint tag and address parts. This is possible if we enforce some constraints on what constitutes a valid address. If we only allow allocation on 8-byte boundaries, then the lower 3-bits of all objects' addresses will be 000_b and the higher 29-bits discriminate the objects from one another. By *or*ing the address of the object with a 3-bit tag that distinguishes the type of the object, we obtain a tagged pointer containing both the object's type and its address in memory.

Eight unique tag combinations are possible with 3-bits of data. Three of the eight combinations were already used for tagging fixnums (000_b and 100_b), and the other immediates (111_b). We are left with five tags for our nonimmediate objects.

We assign the tags as follows: 001_b for pairs, 010_b for closures (discussed later in section **??**), 011_b for symbols (discussed in section **??**), 101_b for vectors, and 110_b for strings.

[3]The names are stuck for historical reasons



(A) Before allocation (B) After allocation

Figure 1.11: Illustration of the heap. The allocation pointer is is held in the `%ebp` register and its value is always aligned on 8-byte boundaries. Individual objects are allocated at address `%ebp` and the allocation pointer is bumped to the first boundary after the object.

A pointer to heap-space from which we allocate our objects is held in a dedicated register, `%ebp`. Allocation is performed by bumping the value of `%ebp` by the size of the object. If the size of the object is not a multiple of 8, then we bump `%ebp` to the next address that is a multiple of 8.

Pairs require 8-bytes of memory to hold its `car` and `cdr` fields. Therefore, code generated for `cons` should, after evaluating its arguments, copy the value of the first argument to *car-offset*`(%ebp)`, copy the value of the second argument to *cdr-offset*`(%ebp)`, compute the pointer to the pair (by *or*ing the *pair-tag* with the value of `%ebp`), then bump the value of `%ebp` by *size-pair*. Notice that the value of `%ebp` remains a multiple of 8 after allocating a pair since the size of the pair is 8.

The predicate `pair?` is implemented the same way as the predicates `fixnum?` and `boolean?`. The argument is masked with 111_b and the result is compared to the *pair-tag*.

The primitives `car` and `cdr` each takes a pair as an argument, so, the value of the argument would already be tagged, or displaced by the amount

of *pair-tag*. The fields themselves are located at *car-offset* and *cdr-offset* from the *address* of the pair. Subtracting the tag from the offset yields the address of the field from the value of the pointer. For example, if the `car`/`cdr` fields are located at offsets 0 and 4, and the tag of the pair is 1, then the code for extracting the `car` field is:

```
movl -1(%eax), %eax
```

and the code for extracting the `cdr` field is:

```
movl 3(%eax), %eax
```

assuming the argument is received in `%eax` and the value of the primitive call is placed in `%eax`.

Creating the heap and saving the C registers

Creating a heap space is no different from creating a stack. The procedure `allocate_protected_space` shown in figure 1.8, p. 23 can be used to allocate a large block of memory to serve as our heap.

We slightly modify our strategy for saving the contents of the `C` registers. Previously, we saved the value of the `%esp` register in the scratch register `%ecx`. As we use more registers in our compiler (like `%ebp`), we need to free more of the `C` register. At this point, it's better to implement a full context-switching mechanism to save and restore all of the `C` register.

The `main` procedure (shown in figure 1.12, p. 39) creates a `context` struct and passes a pointer to it to the `scheme_entry` along with the stack/heap pointers. The `context` struct has eight fields to hold the values of the eight machine registers.

Each register name is commented with a scratch/preserve marker to as defined by the *Application Binary Interface* which is adopted by the UNIX-based operating systems on the Intel-386 architecture. Scratch register are those registers that the callee (`scheme_entry`) can freely modify while preserve registers are those whose value must be preserved.

On entry to `schem_entry`, the context is loaded in the `%ecx` register and all the values that must be preserved are stored. Next, the allocation pointer is loaded into `%ebp` and the stack pointer is loaded into `%esp` before control is transferred to `L_scheme_entry`. The contents of the registers are restored before returning to `main`. See figure 1.13, p. 39 for a detailed assembly sequence.

Exercises

1. Complete the support for pairs in your compiler and the `C` runtime system. Test your implementation using `tests-1.9-cons.scm`.

2. In addition to pairs, many languages provide means of allocating blocks of memory (often called vectors, arrays, etc.). Scheme's vectors can be created by calling `make-vector`. The primitive `make-vector` takes two arguments: a number indicating the size of the vector, and a value used to fill the elements of the vector. For example, calling `(make-vector 4 #t)` returns `#(#t #t #t #t)`. A vector of length N is represented by a block of memory of size $(4N + 4)$ bytes to store the vector length in addition to the vector elements. In addition to the primitive `make-vector`, the primitives `vector?`, `vector-length`, `vector-ref`, and `vector-set!` are needed to inspect and manipulate vectors. Add support for the vector primitives to your compiler and runtime and test it using `tests-1.9-vectors.scm`. (Hint: Pay proper attention to keeping the allocation-pointer aligned at at 8-byte boundaries.)

3. Strings are in many ways similar to vectors. The only difference is is that strings contain characters only as content. This difference allows us to represent strings more compactly in memory: a string of size N requires $(N + 4)$ bytes of memory to store its length and content. Implement the primitives `make-string`, `string?`, `string-length`, `string-ref`, and `string-set!`. Test your implementation using `tests-1.9-strings.scm`. (Hint: Use `gcc` to figure out the assembly instructions for storing and loading a single byte from a string.)

```
typedef struct {
  void* eax;    /* 0    scratch  */
  void* ebx;    /* 4    preserve */
  void* ecx;    /* 8    scratch  */
  void* edx;    /* 12   scratch  */
  void* esi;    /* 16   preserve */
  void* edi;    /* 20   preserve */
  void* ebp;    /* 24   preserve */
  void* esp;    /* 28   preserve */
} context;

int main(int argc, char** argv){
  --- allocate heap/stack
  context ctxt;
  print_ptr(scheme_entry(&ctxt, stack_base, heap));
  --- deallocate heap/stack
  return 0;
}
```

Figure 1.12: A `runtime.c` creates a `context` struct to be used by the `scheme_entry` procedure to save the callee-save registers.

```
0. scheme_entry:                    8.   movl 8(%esp), %esp
1.    movl 4(%esp), %ecx            9.   call L_scheme_entry
2.    movl %ebx, 4(%ecx)            10.  movl 4(%ecx), %ebx
3.    movl %esi, 16(%ecx)           11.  movl 16(%ecx), %esi
4.    movl %edi, 20(%ecx)           12.  movl 20(%ecx), %edi
5.    movl %ebp, 24(%ecx)           13.  movl 24(%ecx), %ebp
6.    movl %esp, 28(%ecx)           14.  movl 28(%ecx), %esp
7.    movl 12(%esp), %ebp           15.  ret
```

Figure 1.13: Saving and restoring the context around `L_scheme_entry`.

Appendix A

Tables

A.1 ASCII Table

0	#x0	#\nul	32	#x20	#\sp	64	#x40	#\@	96	#x60	#\‘
1	#x1	#\soh	33	#x21	#\!	65	#x41	#\A	97	#x61	#\a
2	#x2	#\stx	34	#x22	#\"	66	#x42	#\B	98	#x62	#\b
3	#x3	#\etx	35	#x23	#\#	67	#x43	#\C	99	#x63	#\c
4	#x4	#\eot	36	#x24	#\$	68	#x44	#\D	100	#x64	#\d
5	#x5	#\enq	37	#x25	#\%	69	#x45	#\E	101	#x65	#\e
6	#x6	#\ack	38	#x26	#\&	70	#x46	#\F	102	#x66	#\f
7	#x7	#\bel	39	#x27	#\’	71	#x47	#\G	103	#x67	#\g
8	#x8	#\bs	40	#x28	#\(	72	#x48	#\H	104	#x68	#\h
9	#x9	#\tab	41	#x29	#\)	73	#x49	#\I	105	#x69	#\i
10	#xA	#\lf	42	#x2A	#*	74	#x4A	#\J	106	#x6A	#\j
11	#xB	#\vt	43	#x2B	#\+	75	#x4B	#\K	107	#x6B	#\k
12	#xC	#\ff	44	#x2C	#\,	76	#x4C	#\L	108	#x6C	#\l
13	#xD	#\cr	45	#x2D	#\-	77	#x4D	#\M	109	#x6D	#\m
14	#xE	#\so	46	#x2E	#\.	78	#x4E	#\N	110	#x6E	#\n
15	#xF	#\si	47	#x2F	#\/	79	#x4F	#\O	111	#x6F	#\o
16	#x10	#\dle	48	#x30	#\0	80	#x50	#\P	112	#x70	#\p
17	#x11	#\dc1	49	#x31	#\1	81	#x51	#\Q	113	#x71	#\q
18	#x12	#\dc2	50	#x32	#\2	82	#x52	#\R	114	#x72	#\r
19	#x13	#\dc3	51	#x33	#\3	83	#x53	#\S	115	#x73	#\s
20	#x14	#\dc4	52	#x34	#\4	84	#x54	#\T	116	#x74	#\t
21	#x15	#\nak	53	#x35	#\5	85	#x55	#\U	117	#x75	#\u
22	#x16	#\syn	54	#x36	#\6	86	#x56	#\V	118	#x76	#\v
23	#x17	#\etb	55	#x37	#\7	87	#x57	#\W	119	#x77	#\w
24	#x18	#\can	56	#x38	#\8	88	#x58	#\X	120	#x78	#\x
25	#x19	#\em	57	#x39	#\9	89	#x59	#\Y	121	#x79	#\y
26	#x1A	#\sub	58	#x3A	#\:	90	#x5A	#\Z	122	#x7A	#\z
27	#x1B	#\esc	59	#x3B	#\;	91	#x5B	#\[	123	#x7B	#\{
28	#x1C	#\fs	60	#x3C	#\<	92	#x5C	#\\	124	#x7C	#|
29	#x1D	#\gs	61	#x3D	#\=	93	#x5D	#\]	125	#x7D	#\}
30	#x1E	#\rs	62	#x3E	#\>	94	#x5E	#\^	126	#x7E	#\~
31	#x1F	#\us	63	#x3F	#\?	95	#x5F	#_	127	#x7F	#\del

Figure A.1: ASCII Table. Due to typographical problems, some characters may not be very legible. Such characters include 39 (single-quote or apostrophe), 44 (comma), 46 (period), and 96 (back-quote).

A.2 Object Tags

Fixnum:	30-bit fixnum	`00`
Pair:	29-bit address	`001`
Closure:	29-bit address	`010`
Symbol:	29-bit address	`011`
Vector/Record:	29-bit address	`101`
String/Tagged-Object:	29-bit address	`110`
Other Immediate:	29-bit content	`111`

Figure A.2: Primary Pointer Tags.

ASCII chars:	$16 \times$ `0` | 8-bit content	`0000`	`1111`
`#f`:	$24 \times$ `0`	`0010`	`1111`
`#t`:	$24 \times$ `0`	`0011`	`1111`
`()`:	$24 \times$ `0`	`0100`	`1111`
`#!eof`:	$24 \times$ `0`	`0101`	`1111`
`#!void`:	$24 \times$ `0`	`0110`	`1111`
`#!unbound`:	$24 \times$ `0`	`0111`	`1111`

Figure A.3: Immediates Objects

`#!code`:	$24 \times$ `0`	`0000`	`0111`
`#!stack`:	$24 \times$ `0`	`0001`	`0111`
`#!symbol-table`:	$24 \times$ `0`	`0010`	`0111`
`#!hash-table`:	$24 \times$ `0`	`0011`	`0111`
`#!exact-bignum`:	$24 \times$ `0`	`1000`	`0111`
`#!exact-ratnum`:	$24 \times$ `0`	`1001`	`0111`
`#!exact-rectnum`:	$24 \times$ `0`	`1010`	`0111`
`#!inexact`:	$24 \times$ `0`	`1100`	`0111`

Figure A.4: Extended Object Tags