# Immediate Values

| Type | Mask | Tag | Size |
|---|---|---|---|
| Integer | 0x03 | 0x00 | 30 bits |
| Character | 0xFF | 0x0F | 24 bits |
| Boolean | 0x7F | 0x1F | 1 bit |
| Empty List | 0xFF | 0x2F | 0 bits |

Table 1: Masks, tags and sizes of immediate values.

In this example, the immediate value 42 is compiled.

```
>42

 define i64 @scheme_entry()   /* Header
 {
   entry:
   %tmp = alloca i64                ..  */

   store i64 168, i64* %tmp   /* Body */

   %ret = load i64, i64* %tmp /* Footer
   ret i64 %ret
 }                                  ..  */
```

# Local Variables

```
(let ((var1 exp1)       /* Bindings
      (var2 exp2) ..)      ..        */
        body ..)        /* Body      */
```

```
>(let ((x 1) (y 2)) (add x y))

define i64 @scheme_entry()       /* Header
{
entry:
%tmp = alloca i64                               .. */

// Bindings

store i64 4, i64* %tmp           /* Emit 1 into
%var10 = load i64, i64* %tmp     symbol x (var10) .. */

store i64 8, i64* %tmp           /* Emit 2 into
%var11 = load i64, i64* %tmp     symbol y (var11) .. */

// Body

store i64 %var11, i64* %tmp      /* Load x
%var12 = load i64, i64* %tmp                    .. */

store i64 %var10, i64* %tmp      /* Load y
%var13 = load i64, i64* %tmp                    .. */

%var14 = add i64 %var13, %var12  /* Addition
store i64 %var14, i64* %tmp                     .. */

%ret = load i64, i64* %tmp       /* Footer
ret i64 %ret                               .. */
}
```

# Conditionals

```
(if condition consequent alternative)
```

```
// A program to return the lower of x and y.
(let ([x 3]
      [y 5])
     (if (< x y) x y))
define i64 @scheme_entry()
{
 entry:
 %tmp = alloca i64
 store i64 12, i64* %tmp
 %var10 = load i64, i64* %tmp              // Let bindings
 store i64 20, i64* %tmp
 %var11 = load i64, i64* %tmp
 store i64 %var11, i64* %tmp
 %var17 = load i64, i64* %tmp
 store i64 %var10, i64* %tmp
 %var18 = load i64, i64* %tmp
 %var19 = icmp slt i64 %var18, %var17
 %var20 = zext i1 %var19 to i64
 %var21 = shl i64 %var20, 7
 %var22 = or i64 %var21, 31
 store i64 %var22, i64* %tmp
 %var15 = load i64, i64* %tmp
 %var16 = icmp eq i64 %var15, 159          // Compare Conditional
 br i1 %var16, label %lab12, label %lab13  // Branch
 lab12:                                    // Conseq
 store i64 %var10, i64* %tmp
 br label %lab14
 lab13:                                    // Altern
 store i64 %var11, i64* %tmp
 br label %lab14
 lab14:
 %ret = load i64, i64* %tmp
 ret i64 %ret
 }
```

# A Scheme to LLVM Compiler

## An Incremental Approach to Compiler Construction

Approaching compiler construction can be a daunting task. Ghuloum argues that too much emphasis is placed on ``individual passes of the compiler'', often losing focus on the bigger picture. A humorous example is supplied by Ghuloum in which Appel's book on compiler construction presents 11 chapters on the different stages, but only 6 pages on ``Putting it all together''.

However, an issue with this paper is that is dated from 2006 and many students have attempted this already. The paper targets x86 assembly. My goal was to adopt a more modern approach towards compiler construction by targeting LLVM instead while also documenting the process to highlight the differences between using x86 and LLVM. Furthermore, my goal was to gain a deeper insight into compiler construction, investigate advanced memory management and optimisation techniques. Finally, my aim was to learn Scheme, a programming language I was unfamiliar with at the start.

I implemented a subset of Scheme's core forms, extended forms and a large collection of primitives in addition to the variety of elements required to construct them. In addition, I developed a sophisticated C runtime and a testing framework to compliment them. Furthermore, I extended the project by applying advanced memory management techniques including garbage collection and various optimisation techniques.

# Heap Objects

| Type | Mask | Tag |
|---|---|---|
| Pairs | 0x07 | 0x01 |
| Closures | 0x07 | 0x02 |
| Vectors | 0x07 | 0x05 |
| Strings | 0x07 | 0x06 |

Table 4: Masks and Tags of data structures.

| Primcall | Description |
|---|---|
| (pair? p) | Returns true if p is a Pair. |
| (cons a b) | Constructs (a . b) |
| (car '(a . b)) | Returns a |
| (cdr '(a . b)) | Returns b |

Table 5: Primitives related to Pairs.

| Primcall | Description |
|---|---|
| (vector? vec) | Returns true if vec is a Vector.. |
| (make-vector len val) | Constructs #(val, val, ..) of size len. |
| (vector-ref vec ind) | Accesses vec[ind] |
| (vector-set! vec ind val) | Sets vec[ind] = val. |

Table 6: Primitives related to Pairs.

# Garbage Collection

```
(begin
 (let ([a (make-vector 10 (cons 1 2))]) a)
 (gc)
)
```

## Mark and Sweep Algorithm

```
Marked Vect 0
Marked Clsr 0
Marked Pair 0
Collected 13 memory locations
```

# Lambdas

| Primcall | Description |
|---|---|
| (lambda? (formal args) body) | Constructs a lambda |

Table 2: Primitives related to lambdas.

## Closure Conversion

```
    (let ((x 5))
      (lambda (y) (lambda () (add x y))))

(let ((x 5))
  (lambda (y) (x) (lambda () (x y) (add x y))))

    (labels
       ([f0 (code () (x y) (add x y)])
        [f1 (code (y) (x) (closure f0 x y))])
    (let ((x 5)) (closure f1 x)))
```

# Variable Assignment

| Primcall | Description |
|---|---|
| (set! var val) | Assigns var val. |

Table 7: Primitives related to Assignment.

```
(let ((f (lambda (c) ()
      (cons (lambda (v) (c) (set! c v))
            (lambda () (c) c)))))
(let ((p (funcall f 0)))  // Call f (0) to retrieve Pair and set C to 0
    (funcall (car p) 12) // Call ((Car p) 12) to set C to 12
    (funcall (cdr p))))  // Call ((Cdr p)) to retrieve C


 (let ((f (lambda (c) ()
      (let ((c (make-vector 1 c)))
         (cons (lambda (v) (c) (vector-set! c 0 v))
      (lambda () (c) (vector-ref c 0)))))))
 (let ((p (f 0)))
      ((car p) 12)
      ((cdr p))))
```

# Optimiser

## Constant Propogation and Interpretive Evaluation

```
(let ([x (add 1 2)] [y 3]) (add x y))

(let ([x 3] [y 3]) (add x y))

(let () 6)
```