Jam2000 Assembly

```
(ldi R0 0)
(ldi R1 1)


(ldi R2 10)
(sub R2 R0 R2)
(bez R2 7)
(add R0 R0 R1)
(jmpi 2)


(halt)
```

Number addresses like 7 and 2 are a pain...

Jam2000 Assembly and Labels

```
(ldi R0 0)                 (ldi R0 0)
(ldi R1 1)                 (ldi R1 1)

                           (label LOOP)
(ldi R2 10)                (ldi R2 10)
(sub R2 R0 R2)      =      (sub R2 R0 R2)
(bez R2 7)                 (bez R2 DONE)
(add R0 R0 R1)             (add R0 R0 R1)
(jmpi 2)                   (jmpi LOOP)

                           (label DONE)
(halt)                     (halt)
```

Jam2000 Assembly and Constants

```
                          (const COUNT 10)
(ldi R0 0)                (ldi R0 0)
(ldi R1 1)                (ldi R1 1)

(label LOOP)              (label LOOP)
(ldi R2 10)               (ldi R2 COUNT)
(sub R2 R0 R2)      =     (sub R2 R0 R2)
(bez R2 DONE)             (bez R2 DONE)
(add R0 R0 R1)            (add R0 R0 R1)
(jmpi LOOP)               (jmpi LOOP)

(label DONE)              (label DONE)
(halt)                    (halt)
```

Jam2000 Assembly and Data

```
(jmpi PROG)

(label DRAW-CHAR)
(ldi R7 1)
....
(label DONE)
(jmpx R2)

(data FONT-TABLE
      0 0 0 ....)

(label PROG)
....
```

but there's no reason anymore to put `PROG` at the end

Running the Assembler

```
% racket as.rkt < loop.jam > loopdisk

% racket jam.rkt loopdisk
```

- `%` means "this is a command line"; your actual prompt may be different
- **`racket`** is the executable
- **`as.rkt`** is the argument to **`racket`**, which is a Racket program to run
- every command-line program has a default input and output stream, and **`as.rkt`** reads a Jam2000 assembly program from its input stream and writes a Jam2000 disk to its output stream
- **`< loop.jam`** redirects the input stream to read from **`loop.jam`**
- **`> loopdisk`** redirects the input stream to read from **`loopdisk`**

Jam2000 Assembly

A Jam2000 instruction in S-expression form is an ***instruction***, possibly using a `name` in place of a number

A Jam2000 assembly program is a sequence of ***declarations***

A ***declaration*** is either

- An ***instruction***
- `(label name)`
- `(const name num)`
- `(data name num ...)` where a `name` can be used in place of a `num`

Jam2000 Assembly

- An **instruction** corresponds to a machine code
- `(label name)` has no machine code, but declares `name` to be replaced with the count of machine codes that precede the `label` declaration
- `(const name num)` has no machine code, but declares `name` to be replaced with `num`
- `(data name num ...)` generates the machine-code sequence `num ...` and declares `name` to be replaced with the number machine codes that precede the `data` declaration

Assembling

```
COUNT = 10
LOOP = 2
DONE = 7
```

⇒

```
(const COUNT 10)
(ldi R0 0)                      9
(ldi R1 1)                    119

(label LOOP)
(ldi R2 COUNT)               1029
(sub R2 R0 R2)              20220
(bez R2 DONE)                 703
(add R0 R0 R1)              10010
(jmpi LOOP)                   201

(label DONE)
(halt)                          0
```

From High Level to Low Level

```
(define (sum n)
  (cond
   [(zero? n) 0]
   [else (+ n (sum (sub1 n)))]))

(sum ...)
```

From High Level to Low Level

```
(define (sum n a)
  (cond
   [(zero? n) a]
   [else (sum (sub1 n) (+ n a))]))

(sum ... 0)
```

From High Level to Low Level

```
(define n 0)
(define a 0)

(define (sum)
  (cond
   [(zero? n) a]
   [else (set! a (+ n a))
         (set! n (sub1 n))
         (sum)]))

(set! n ...)
(sum)
```

From High Level to Low Level

```
(define n 0) ; argument register
(define a 0) ; register

(define (sum) ; label
  (cond
   [(zero? n) a]
   [else (set! a (+ n a))
         (set! n (sub1 n))
         (sum)])); jump

(set! n ...)
(sum) ; jump
```

From High Level to Low Level

```
(define n 0)  ; argument register
(define a 0)  ; register

(define (sum)  ; label
  (if (zero? n)  (done)  ; branch
      (begin
        (set! a (+ n a))
        (set! n (sub1 n))
        (sum))))  ; jump
(define (done) a)

(set! n ...)
(sum)  ; jump
```

From High Level to Low Level

If all values are numbers...

... and if you can convert to tail form

then

- functions become labels
- conditionals become branches
- result at a final label

From High Level to Low Level: Nested Conditionals

```
(if (or (< n 10) (> n 5))
    (something)
    (nothing))
```

⇒

```
(if (< n 10) (something)
    (if (> n 5) (something)
        (nothing)))
```

From High Level to Low Level: Nested Conditionals

```
(if (and (< n 10) (> n 5))
    (something)
    (nothing))
```

⇒

```
(if (>= n 10) (nothing)
    (if (<= n 5) (nothing)
        (something)))
```

```
        (if Z X Y) = (if (not Z) Y X)

(not (and X Y)) = (or (not X) (not Y))

          (< Y X) = (not (>= X Y))
```

Shallow Function Calls

```
(define (sqr n)
  (* n n))

(+ (sqr 3) (sqr 4))
```

Shallow Function Calls

```
(define (sqr n)
  (* n n))

(set! a (sqr 3))
(set! b (sqr 4))
(+ a b)
```

Shallow Function Calls

```
(define (sqr)
  (set! r (* n n)))

(set! n 3)
(sqr)
(set! a r)
(set! n 4)
(sqr)
(set! b r)
(+ a b)
```

Shallow Function Calls

```
(define (sqr)
  (set! r (* n n)))

(define (go)
  (set! n 3)
  (sqr)
  (got-a))
(define (got-a)
  (set! a r)
  (set! n 4)
  (sqr)
  (got-b))
(define (got-b)
  (set! b r)
  (+ a b))
(go)
```

Shallow Function Calls

```
(define (sqr)
  (set! r (* n n))
  (next))

(define (go)
  (set! n 3)
  (set! next got-a)
  (sqr))
(define (got-a)
  (set! a r)
  (set! n 4)
  (set! next got-b)
  (sqr))
(define (got-b)
  (set! b r)
  (+ a b))
(go)
```

Shallow Function Calls

```
(define (sqr)
  (set! r (* n n))
  (next))

(define (go)
  (set! n 3)
  (jsr! next sqr got-a))
(define (got-a)
  (set! a r)
  (set! n 4)
  (jsr! next sqr got-b))
(define (got-b)
  (set! b r)
  (+ a b))
(go)
```

Shallow Function Calls

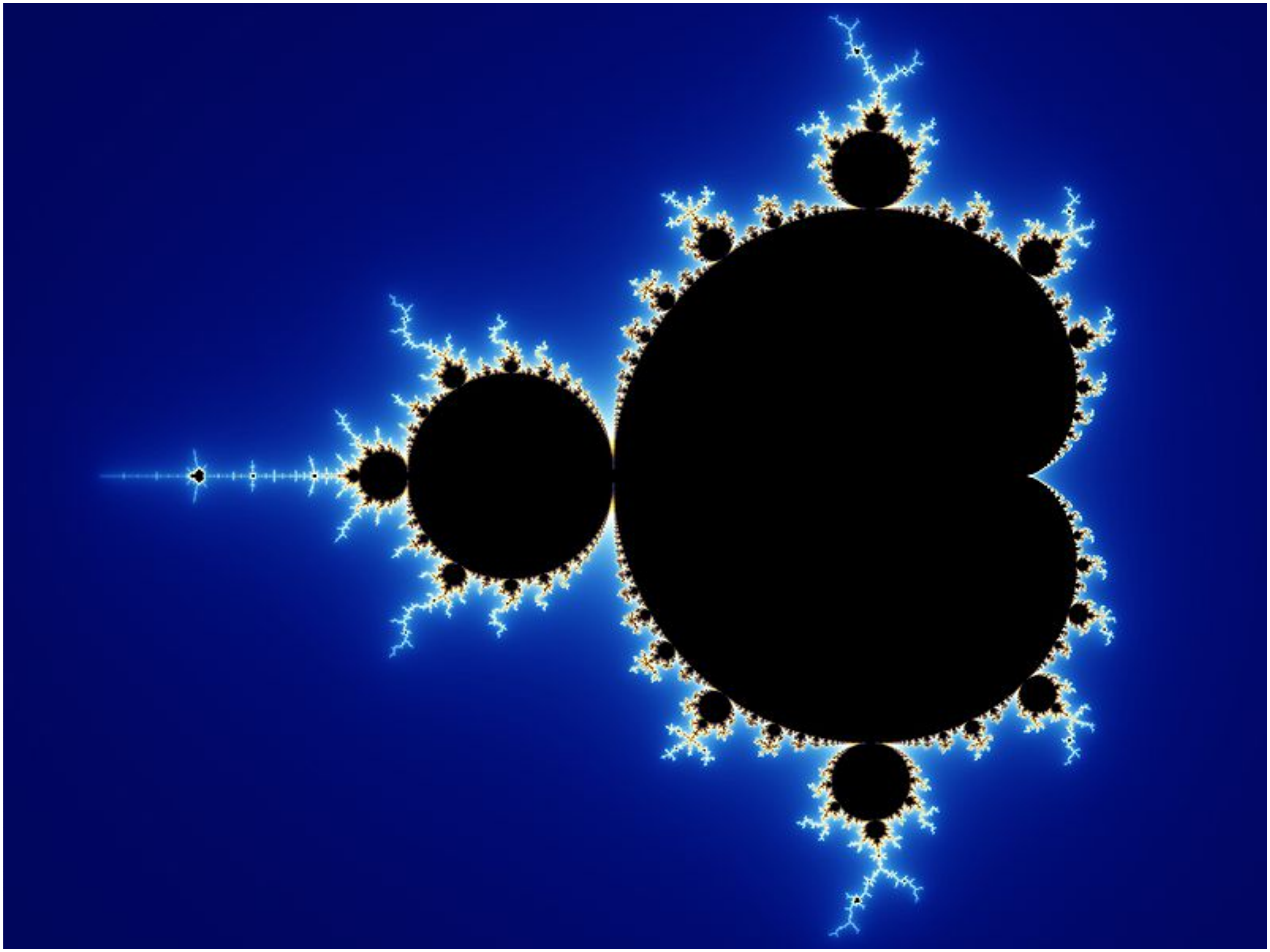
```
(label SQR)
(mul R0 R1 R1)
(jmpx R2)

(label GO)
(ldi R1 3)
(jsr R2 SQR)

(mov R3 R0)
(ldi R1 4)
(jsr R2 SQR)

(add R0 R0 R3)
```

Extended Jam2000 Assembly Example: Mandelbrot



Extended Jam2000 Assembly Example: Mandelbrot