More Data

- Small numbers: obvious
- Booleans: `1` and `0`
- Colors: `0` = black, `9999999` = white, etc.
- Characters: `17` = A, etc.
- Empty: `0`
- Structure or cons: ???

Compound Data

To represent compound data, use a number that is an address, and store pieces starting at the address:

```
(make-posn 7 99)  ⇒  10
```

0	0	0	0	0	0	0	0
0	0	7	99	0	0	0	0
0	0	0	0	0	0	0	0

Compound Data

To represent compound data, use a number that is an address, and store pieces starting at the address:

```
(cons 8 empty)
```

⇒ 12

0	0	0	0	0	0	0	0
0	0	7	99	8	0	0	0
0	0	0	0	0	0	0	0

Compound Data

To represent compound data, use a number that is an address, and store pieces starting at the address:

```
(cons
 9
 (cons 8 empty))
```

⇒ 14

0	0	0	0	0	0	0	0
0	0	7	99	8	0	9	12
0	0	0	0	0	0	0	0

This works if we never store a cons at address 0

Lists

As data in Jam2000 assembly:

```
(const EMPTY 0)
(data CONS2 3 EMPTY)
(data CONS1 2 CONS2)
```

List Sum

```
(label LSUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 LSUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi LSUM)
(label LSUM-DONE)
(jmpx R2)
```

```
(label MAIN-LSUM)

(ldi R0 CONS1)
(ldi R1 0)

(jsr R2 LSUM)

(print R1)
(newline)
(halt)
```

List Sum

```
(label LSUM)
; list i
; accum i
; return address in R2
(bez R0 LSUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi LSUM)
(label LSUM-DONE)
(jmpx R2)
```

empty? test on argument

```
(label MAIN-LSUM)

(ldi R0 CONS1)
(ldi R1 0)

(jsr R2 LSUM)

(print R1)
(newline)
(halt)
```

List Sum

```
(label LSUM)
; list in R0
; accum in
; return a
(bez R0 LSUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi LSUM)
(label LSUM-DONE)
(jmpx R2)
```

`first` of argument

```
(label MAIN-LSUM)

(ldi R0 CONS1)
(ldi R1 0)

(jsr R2 LSUM)

(print R1)
(newline)
(halt)
```

List Sum

```
(label LSUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 LSUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi LSUM)
(label LSUM-DONE)
(jmpx R2)
```

```
(label MAIN-LSUM)

(ldi R0 CONS1)
(ldi R1 0)

(jsr R2 LSUM)

(print R1)
(newline)
(halt)
```

rest of argument

List Sum

```
(label LSUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 LSUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 [illegible])
(add R1 R1 R9)
(jmpi LSUM)
(label LSUM-DONE)
(jmpx R2)
```

accumulate result

```
(label MAIN-LSUM)

(ldi R0 CONS1)
(ldi R1 0)

[illegible]sr R2 LSUM)

(print R1)
(newline)
(halt)
```

List Sum

```
(label LSUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 LSUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi LSUM)
(label LSU
(jmpx R2)
```

recur on `rest`

```
(label MAIN-LSUM)

(ldi R0 CONS1)
(ldi R1 0)

(jsr R2 LSUM)

(print R1)
(newline)
(halt)
```

List Sum

```
(label LSUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 LSUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi LSUM)
(label LSUM-DONE)
(jmpx R2)
```

```
(label MAIN-LSUM)

(ldi R0 CONS1)
(ldi R1 0)

(jsr R2 LSUM)

(print R1)
(newline)
(halt)
```

Allocation

How about `reverse`?

A `cons` needs to ***allocate***:

- Designate some address `ALLOC-PTR` to point to free space
- Initialize `ALLOC-PTR` to the area past all the code
- Increment `ALLOC-PTR` for each `cons`

Allocation

```
(const ALLOC-PTR 7)
```

1	33	9	80	6	77	2	8
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Allocation

```
(const ALLOC-PTR 7)
(cons 91 empty) ; = 8
```

1	33	9	80	6	77	2	10
91	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Allocation

```
(const ALLOC-PTR 7)
(cons 5 (cons 91 empty)) ; = 10
```

1	33	9	80	6	77	2	12
91	0	5	8	0	0	0	0
0	0	0	0	0	0	0	0

Allocation

see **`REVERSE`** in **`list.jam`**

Non-Loop Recursion

What about **`feed-fish`**?

```
(label MAIN-FEED)
....
(jsr R2 FEED)
....


(label FEED)
....
(jsr R2 FEED)
....
```

A single return register isn't enough

Continuation

Instead of a single return address, keep a list of return addresses

For **`FEED`**, this list also needs to remember the number to add after returning

Danger: if we don't get rid of the continuation conses, then we might run out of memory

- Discard each cons just before returning

Stack

A ***stack*** is an alternative to a list, especially for continuations

Typically, a register like **`R6`** holds the stack pointer instead of a memory address like **`ALLOC-PTR`**

+ Simpler allocation

+ Simpler discard

− Splits memory between stack and allocation

```
(data STACK
      0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0)
```

Stack

see **`FEED`** in **`list.jam`**