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:

\[
\text{(make-posn 7 99)} \Rightarrow 10
\]

\[
\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 7 & 99 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Compound Data

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

\[(\text{cons } 8 \text{ empty}) \Rightarrow 12\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 7 & 99 & 8 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Compound Data

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

\[
\begin{array}{c}
(\text{cons} \\
9 \\
(\text{cons} \ 8 \ \text{empty}))
\end{array} \quad \Rightarrow \quad 14
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 7 & 99 & 8 & 0 & 9 & 12 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

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 in R0
; accum in R1
; return address in R2
(lez 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 R1
; return in R2
(ldi R0 CONS1)
(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

(ldi R0 0)
(jsr R2 LSUM)
(print R1)
(newline)
(halt)
List Sum

(label LSUM)
; list in R0
; accum in R1
; return address in R2
(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 in R0
; accum in R1
; return address in R2
(ldi R0 CONS1)
(bez R0 LSUM-DONE)
(ld R9 R0)
(ldi R7 1)
(sr R2 LSUM)
(ldx R0 R0 R7)
(add R1 R1 R9)
(print R1)
(jmpi LSUM)
(label LSUM-DONE)
(halt)
(jmpx R2)
List Sum

\( (\text{label LSUM}) \)
; list in R0
; accum in R1
; return address in R2

\( (\text{label MAIN-LSUM}) \)
\( (\text{ldi R0 CONS1}) \)
\( (\text{ldi R1 0}) \)

\( (\text{bez R0 LSUM-DONE}) \)
\( (\text{ld R9 R0}) \)

\( (\text{ldi R7 1}) \)
\( (\text{jsr R2 LSUM}) \)

\( (\text{ldx R0 R0 R7}) \)
\( (\text{add R1 R1 R9}) \)

\( (\text{jmpi LSUM}) \)
\( (\text{print R1}) \)

\( (\text{label LSUM-DONE}) \)
\( (\text{newline}) \)

\( (\text{label LSUM}) \)
\( (\text{halt}) \)

recur on rest
List Sum

(label LSUM)
; list in R0
; accum in R1
; return address in R2
(label MAIN-LSUM)
(ldi R0 CONS1)
(ldi R1 0)
(jsr R2 LSUM)
(print R1)
(newline)
(halt)

(1d R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi LSUM)
(label LSUM-DONE)
(jmpx R2)
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

\[(\text{const } \text{ALLOC-PTR} \ 7)\]

<table>
<thead>
<tr>
<th>1</th>
<th>33</th>
<th>9</th>
<th>80</th>
<th>6</th>
<th>77</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Allocation

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>33</th>
<th>9</th>
<th>80</th>
<th>6</th>
<th>77</th>
<th>2</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>91</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Allocation

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>33</th>
<th>9</th>
<th>80</th>
<th>6</th>
<th>77</th>
<th>2</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Allocation

see REVERSE in list.jam
Non-Loop Recursion

What about *feed-fish*?

```lisp
(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 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