Jam2000 Assembly

(ldi R0 0)
(ldi R1 1)

(ldi R2 10)
(sub R2 R0 R2)
(bez R0 6)
(add R0 R0 R1)
(jmpi 2)

(halt)
Jam2000 Assembly

(ldi R0 0)
(ldi R1 1)

(ldi R2 10)
(sub R2 R0 R2)
(bez R0 6)
(add R0 R0 R1)
(jmpi 2)

(halt)

Number addresses like 6 and 2 are a pain...
Jam2000 Assembly and Labels

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

(ldi R2 10)              (label LOOP)
(sub R2 R0 R2)           (sub R2 R0 R2)
(bez R0 6)               (bez R0 DONE)
(add R0 R0 R1)           (add R0 R0 R1)
(jmpi 2)                 (jmpi LOOP)

(halt)                   (label DONE)
(halt)                   (halt)
```
Jam2000 Assembly and Constants

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

(label LOOP) (label LOOP)
(ldi R2 10) (ldi R2 COUNT)
(sub R2 R0 R2) (sub R2 R0 R2)
(bez R0 DONE) (bez R0 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)
.....
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
Extended Jam2000 Assembly Example: Mandelbrot
Extended Jam2000 Assembly Example: Mandelbrot
Linkers and Loaders

Pieces of a program can be individually assembled, but with some addresses not yet picked

A **linker** puts pieces together, possibly picking some addresses

A **loader** picks remaining addresses at the last minute so that a program can run

(we won’t bother with them for Jam2000)
More Data

• Small numbers: obvious
• Booleans: 1 and 0
• Colors: 0 = black, 9999999 = white, etc.
• Characters: 17 = A, etc.
More Data

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

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

```
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 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:

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

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

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

\[
\begin{array}{c}
(\text{cons} \\
9 \\
(\text{cons} \ 8 \ \text{empty})) \\
\end{array} \Rightarrow \begin{array}{c}
14 \\
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 7 & 99 & 8 & 0 & 9 & 12 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\end{array}
\]
Compound Data

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

\[
\begin{align*}
(\text{cons} & \ 9) \\
(\text{cons} & \ 8 \ \text{empty})) & \Rightarrow 14 \\
\end{align*}
\]

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)
Sum

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

(label MAIN-SUM)

(ldi R0 CONS1)
(ldi R1 0)
(ldi R2 MAIN-SUM-RETURN)

(jmpi SUM)

(label MAIN-SUM-RETURN)

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

(label MAIN-SUM)
(ldi R0 CONS1)
(ldi R1 0)
(ldi R2 MAIN-SUM-RETURN)
(jmpi SUM)
(label MAIN-SUM-RETURN)
(print R1)
(newline)
(halt)
(label SUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 0)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi SUM)
(label SUM-DONE)
(jmpx R2)

(label MAIN-SUM)
(ldi R0 CONS1)
(adi R1 0)
(bez R0 SUM-DONE)
(ldi R2 MAIN-SUM-RETURN)
(jmpi SUM)
(label MAIN-SUM-RETURN)
(print R1)
(newline)
(halt)
(label SUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 SUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi SUM)
(label SUM-DONE)
(jmpx R2)

(label MAIN-SUM)
(1di R0 CONS)
(1di R1 0)
(1di R2 MAIN-SUM-RETURN)
(jmpi SUM)

rest of argument

(jmpi MAIN-SUM-RETURN)
(print R1)
(newline)
(halt)
(label SUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 SUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi SUM)
(label SUM-DONE)
(jmpx R2)

(label MAIN-SUM)

(1di R0 CONS1)
(1di R1 0)
(1di R2 MAIN-SUM-RETURN)

accumulate result

(label MAIN-SUM-RETURN)
(print R1)
(newline)
(halt)
(label SUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 SUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi SUM)
(label SUM-)
(jmpx R2)

(label MAIN-SUM)
(ldi R0 CONS1)
(ldi R1 0)
(ldi R2 MAIN-SUM-RETURN)

(jmpi SUM)

(label MAIN-SUM-RETURN)
(print R1)
(newline)
(halt)
(label SUM)
; list in R0
; accum in R1
; return address in R2
(bez R0 SUM-DONE)
(ld R9 R0)
(ldi R7 1)
(ldx R0 R0 R7)
(add R1 R1 R9)
(jmpi SUM)
(label SUM-DONE)
(jmpx R2)

(label MAIN-SUM)

(ldi R0 CONS1)
(ldi R1 0)
(ldi R2 MAIN-SUM-RETURN)
(jmpi SUM)

(label MAIN-SUM-RETURN)
(print R1)
(newline)
(halt)
Allocation

How about reverse?
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)

\[
\begin{array}{cccccccc}
1 & 33 & 9 & 80 & 6 & 77 & 2 & 8 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Allocation

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

\[
\begin{array}{ccccccccc}
1 & 33 & 9 & 80 & 6 & 77 & 2 & 10 \\
91 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Allocation

$$(\text{const ALLOC-PTR 7})$$

$$(\text{cons 5 (cons 91 empty)}) \; ; = 10$$

<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 \texttt{REVERSE} in \texttt{list.jam}
Non-Loop Recursion

What about \textit{feed-fish}?
Non-Loop Recursion

What about *feed-fish*?

```
(label MAIN-FEED)
....
(ldi R2 MAIN-FEED-RETURN1)
(jmpi FEED) ....
(label MAIN-FEED-RETURN1) ....
```

```
(label FEED)
....
(ldi R2 FEED-BACK)
(jmpi FEED) ....
(label FEED-BACK) ....
```
Non-Loop Recursion

What about \texttt{feed-fish}?

\begin{verbatim}
(label MAIN-FEED)
....
(ldi R2 MAIN-FEED-RETURN1)
(jmpi FEED) ....
(label MAIN-FEED-RETURN1) ....

(label FEED)
....
(ldi R2 FEED-BACK)
(jmpi FEED) ....
(label FEED-BACK) ....
\end{verbatim}

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
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

$$\begin{array}{l}
\textbf{data} & \textbf{STACK} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}$$
Stack

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