Stacks cont'd

Ata Kaban
University of Birmingham
Stacks for calculation

- Example:

\[(5 + 2) \times \sqrt{x \times x + y \times y + 8}\]

What order are operations applied in?

\[(5 + 2) \times \sqrt{x \times x + y \times y + 8}\]

1 6 5 2 4 3 7

\[\sqrt{\text{means}}
\]

“square root of” (SQRT)
e.g. SQRT(x\times x + y\times y)
Reverse Polish notation

- Order of operations is as written
- No brackets needed
- Powerful use of stack (= operand stack) to store intermediate results

Number or variable: push on stack
Operation: apply to top elements on stack

E.g.  
\[
\begin{array}{c}
8 \\
3 \\
6 \\
10 \\
\ldots \\
\end{array} \quad \text{apply +} \quad \begin{array}{c}
11 \\
6 \\
10 \\
\ldots \\
\end{array}
\]
\[(5 + 2) \times \sqrt{x \times x + y \times y} + 8\]

Reverse Polish: push operands, then operate. We get:

\[5 \ 2 \ + \ x \ x \ \ast \ y \ y \ \ast \ + \ \sqrt{\ast} \ 8 \ +\]
Suppose that $x$ has value 3, and $y$ has value 4.

Now, evaluate the expression. (Top of stack is on right.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5,2</td>
</tr>
<tr>
<td>+</td>
<td>7</td>
</tr>
<tr>
<td>$x$</td>
<td>7,3</td>
</tr>
<tr>
<td>$x$</td>
<td>7,3,3</td>
</tr>
<tr>
<td>$*$</td>
<td>7,9</td>
</tr>
<tr>
<td>$y$</td>
<td>7,9,4</td>
</tr>
<tr>
<td>$y$</td>
<td>7,9,4,4</td>
</tr>
<tr>
<td>$*$</td>
<td>7,9,16</td>
</tr>
<tr>
<td>+</td>
<td>7,25</td>
</tr>
<tr>
<td>SQRT</td>
<td>7,5</td>
</tr>
<tr>
<td>$*$</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>35,8</td>
</tr>
<tr>
<td>+</td>
<td>43</td>
</tr>
</tbody>
</table>
Notation for operand stack

To show what an operation does to stack:
E.g. subtraction

stack before

..., val1, val2 \rightarrow ... \rightarrow ..., val1-val2

stack after

pop these two, do subtraction, push result

the rest is unchanged
Any expression can be converted to reverse Polish
• Then easy to execute with a stack

• Applications
  1. Humans use reverse Polish directly
     - E.g. some pocket calculator – HP in 1970s, 80s
  • Forth programming language has two stacks:
     - Operand stacks for calculations
     - Return stack for module calls
Applications of Reverse Polish

2. Compile to a reverse Polish form that is then executed.
   • e.g. Postscript format, for printable files
     - executed by printers
   • e.g. Java byte code
     - uses operand stacks for calculations

   • In Java, each method call has its own operand stack.
Stack instead of registers

- Use 2 stacks
  - return stack for subroutine return
  - operand stack for reverse Polish calculations
- Don't need a, b, c registers
- Advantages:
  - More space for calculations
  - Opcodes don't need to specify registers
- Disadvantages:
  - Harder to know where things are on stack
What is an operand?

• Underlying meaning:
  Whatever an operator operates on.

• Two meanings here (don't confuse them):
  1) Extra bytes after the instruction opcode in memory, e.g.
    \texttt{ld a 42}
  2) Entries in the operand stack.

  2) Entries in the operand stack.
Machine instructions as stack operators

[Forget the Toy CPU – remember these mnemonics (JVM)]

• Arithmetic:
  
  e.g. **add** - adds top 2 stack entries
  
  ..., val1, val2  →  ..., val1+val2

  e.g. **sub** - subtracts top 2 stack entries
  
  ..., val1, val2  →  ..., val1-val2

  e.g. **neg** - negates top stack entry
  
  ..., val  →  ..., -val

• Similarity: **mul**, **div**, **rem**
Bitwise boolean operations

- Boolean operations on one bit
  0=false, 1=true

<table>
<thead>
<tr>
<th>OR</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AND</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XOR</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Can do these bit-wise on binary values. Example for XOR:

```
0 0 1 1 1 0 1 1 1 1 ............0 1 1 0 0 1 0 0 0
0 0 0 1 0 0 1 0 0 1 ............1 1 1 1 0 1 1 0 0
-------------------------------------------------------------
0 0 1 0 1 0 0 1 1 0 ............1 0 0 1 0 0 1 0 0
```

XOR: done on top 2 stack entries (similar to: or, and):

..., val1, val2 → ..., val1 XOR val2
Pushing constants on the stack

\[ \text{push } N \rightarrow \ldots, N \]

\[ \text{load } x \rightarrow \ldots, \text{value of } x \]

- Pretend we also can push values of variables [more on that next time]

instruction needs extra byte
Example: \[ 5 \ 2 + x \ x \ * \ y \ y \ * + \sqrt{} * \ 8 + \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>push 5</td>
<td>5</td>
</tr>
<tr>
<td>push 2</td>
<td>5,2</td>
</tr>
<tr>
<td>add</td>
<td>7</td>
</tr>
<tr>
<td>load x</td>
<td>7,3</td>
</tr>
<tr>
<td>load x</td>
<td>7,3,3</td>
</tr>
<tr>
<td>mul</td>
<td>7,9</td>
</tr>
<tr>
<td>load y</td>
<td>7,9,4</td>
</tr>
<tr>
<td>load y</td>
<td>7,9,4,4</td>
</tr>
<tr>
<td>mul</td>
<td>7,9,16</td>
</tr>
<tr>
<td>add</td>
<td>7,25</td>
</tr>
<tr>
<td>call \sqrt{}</td>
<td>7,5</td>
</tr>
<tr>
<td>mul</td>
<td>35</td>
</tr>
<tr>
<td>push 8</td>
<td>35,8</td>
</tr>
<tr>
<td>add</td>
<td>43</td>
</tr>
</tbody>
</table>

From slide 5

more next time
Jumps

• Unconditional jumps
  • Operand stack not used

• Conditional jumps
  
  `ifeq N` // jumps to N if val=0
  
  ..., val → ...

  `if_cmpeq N` // jumps to N if val1=val2

  ..., val1, val2 → ...

`compare`
Conditional jumps with other comparisons

jump if \( val \begin{cases} = 0 \\ < \\ \leq \\ \neq \\ > \\ \geq \end{cases} \) 

6 operators: ifeq, iflt, ifle, etc. 
also: if_cmpeq, if_cmplt, etc.
Summary

You have now seen:

- Registers for calculating
- Operand stacks for calculating
- Return stacks for saving return addresses and registers

Next:

JVM puts them together: Bytecode
On to Exercise 3 and Exercise 4 on the 2-page Notes on Stacks (handed out yesterday).

- You will need to use an algorithm to convert math expressions from the usual “infix” notation to reverse-Polish. This algo is called Dijkstra's Shunting-Yard Algorithm.
  - The attachment explains how it works
  - You can also read the Wikipedia page on this algo https://en.wikipedia.org/wiki/Shunting-yard_algorithm