CS378 - A Formal Model of the JVM
Lecture 3

http://www.cs.utexas.edu/users/moore/classes/cs378-jvm/

Semester           Spring, 2012
Unique Id           53105
Instructor          J Strother Moore
Email               moore@cs.utexas.edu
Office              MAI 2014
Office Hours        MW 1:00–2:00
Represent Stacks

Pick a representation for stacks. Define push so that (push i stk) pushes the item i onto the stack represented by stk.

Define top to take a non-empty stack and return the topmost item.

Define pop to take a non-empty stack and return the stack obtained by removing the topmost item.
Programming Note

The symbols `push` and `pop` are already defined in the standard ACL2 symbol package.

To define these functions in an ACL2 session we have to create a new package, which we’ll call the “M1” package.

I’ll show you the incantation for that later.
Accessing List Elements

Define \texttt{nth} so that \((\texttt{nth } i \ x)\) returns the \(i^{th}\) (0-based) element of list \(x\). You may assume \(x\) has at least \(i+1\) elements.
Updating List Elements

Define update-nth so that
(update-nth i v x)
“changes” the list x so that the \(i^{th}\) (0-based) element is v. It leaves the other elements unchanged. Actually, it returns a new list; you can’t modify an object in ACL2. You may assume x has at least \(i+1\) elements.
States

A state contains a program counter, a list of local variable values, a stack, and a program.

Define make-state to take four objects and return a state containing them.

Define pc, locals, stack, and program to take a state and return the corresponding part of it.
Instructions

An instruction contains an “op-code” and 0, 1, or 2 “operands.” Define op-code to return the op-code of an instruction.

Define arg1 to return the first operand of an instruction that has 1 or 2 operands.

Define arg2 to return the second operand of an instruction that has 2 operands.
Next Instruction

A program is a list of instructions and a pc is a natural number.

The “next instruction” of a state is the instruction of the state’s program indicated by the pc.

Define (next-inst s) to return the next instruction of state s. You may assume the concept is well-defined.
# M1 Instruction Set

The M1 instructions are represented by entries of the form shown below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>layout of opcode and args</td>
</tr>
<tr>
<td>Stack</td>
<td>stack before ⇒ stack after</td>
</tr>
<tr>
<td>Description</td>
<td>longer description</td>
</tr>
</tbody>
</table>

Stacks are displayed with the topmost item on the right. Unless otherwise noted, the program counter is always incremented by one.
ILOAD

Operation  push local $n$
Format     $(\text{ILOAD } n)$
Stack      $\ldots \Rightarrow \ldots, v$
Description The value $v$ of local variable $n$ is pushed onto the stack.
<table>
<thead>
<tr>
<th><strong>ICONST</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td>push constant</td>
</tr>
<tr>
<td><strong>Format</strong></td>
<td><code>(ICONST \ c)</code></td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td><code>\ldots \Rightarrow \ldots, c</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The constant ( c ) is pushed onto the stack.</td>
</tr>
<tr>
<td><strong>IADD</strong></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td>add two integers</td>
</tr>
<tr>
<td><strong>Format</strong></td>
<td>(IADD)</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>$\ldots, v_1, v_2 \Rightarrow \ldots, r$</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Both $v_1$ and $v_2$ must be integers. The values are popped from the stack. Their sum, $r$, is pushed onto the stack.</td>
</tr>
</tbody>
</table>
**ISUB**

**Operation** subtract two integers

**Format** (ISUB)

**Stack** \(\ldots, v_1, v_2 \Rightarrow \ldots, r\)

**Description** Both \(v_1\) and \(v_2\) must be integers. The values are popped from the stack. The result, \(r\), is \(v_1 - v_2\) and is pushed onto the stack.
<table>
<thead>
<tr>
<th><strong>IMUL</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td>multiply two integers</td>
</tr>
<tr>
<td><strong>Format</strong></td>
<td>(IMUL)</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>(\ldots, v_1, v_2 \Rightarrow \ldots, r)</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Both (v_1) and (v_2) must be integers. The values are popped from the stack. Their product, (r), is pushed onto the stack.</td>
</tr>
</tbody>
</table>
ISTORE

Operation  store into local \( n \)
Format  \((\text{ISTORE } n)\)
Stack  \(\ldots, v \Rightarrow \ldots\)
Description  The value, \( v \), on top of the stack is removed and stored into local \( n \).
GOTO

Operation  jump by $n$
Format      (GOTO $n$)
Stack       $\ldots \Rightarrow \ldots$
Description Execution proceeds at offset $n$ from this instruction, where $n$ may be positive or negative. The target address must be in the current program.
IFEQ

Operation conditional jump by $n$
Format (IFEQ $n$)
Stack $\ldots, v \Rightarrow \ldots$
Description Execution proceeds at offset $n$ from this instruction if $v$ is 0 and at the next instruction otherwise. Pop the stack.
The Single Step Function

Define \texttt{step} so that \((\texttt{step } s)\) takes an M1 state and executes the next instruction. If the next instruction is not one of those given above, halt the machine.
M1 Run

An M1 “schedule” is a list. (Eventually, schedules will specify which thread is to step next, but for now, only the length of the schedule matters.)

Define run so that (run sched s) takes a schedule (of length n) and a state s and steps s n times.
Challenges

1. Build an M1 model yourself. The relevant package declaration is shown below.

2. Write an M1 program to compute factorial.

3. Write an ACL2 expression that uses M1 to compute $n!$ for any natural number $n$. 
(defpkg "M1"
  '(T NIL QUOTE IF EQUAL AND OR
    NOT IMPLIES IFF CONS CAR CDR CONSP ENDP
    LIST LIST* ATOM SYMBOLP + - * / EXPT
    FLOOR MOD NATP INTEGERP NFIX ZP < <=
    > >= LET LET* COND CASE OTHERWISE DEFUN
    DEFTHM THM DEFCONST DEFMACRO PROGN &REST
    MUTUAL-RECURSION IN-PACKAGE DECLARE
    IGNORE XARGS IN-THEORY ENABLE DISABLE
    E/D INCLUDE-BOOK LD I-AM-HERE PBT PCB
    PCB! PE PE! PF PL PR PR! PUFF U UBT UBT!
    O-P O< ACL2-COUNT INTERN-IN-PACKAGE-OF-SYMBOL
    COERCED SYMBOL-NAME STRING CONCATENATE
    STRIP-CARS ASSOC PAIRLIS$ PAIRLIS-X2
    SYNTAXP QUOTE))

(in-package "M1")
(defpkg "M1"
  '(T NIL QUOTE IF EQUAL AND OR
     NOT IMPLIES IFF CONS CAR CDR CONSP ENDP
     LIST LIST* ATOM SYMBOLP + - * / EXPT
     FLOOR MOD NATP INTEGERP NFIX ZP < <=
     > >= LET LET* COND CASE OTHERWISE DEFUN
     DEFTHM THM DEFCONST DEFMACRO PROGN &REST
     MUTUAL-RECURSION IN-PACKAGE DECLARE
     IGNORE XARGS IN-THEORY ENABLE DISABLE
     E/D INCLUDE-BOOK LD I-AM-HERE PBT PCB
     PCB! PE PE! PF PL PR PR! PUFF U UBT UBT!
     0-P 0< ACL2-COUNT INTERN-IN-PACKAGE-OF-SYMBOL
     COERCE SYMBOL-NAME STRING CONCATENATE
     STRIP-CARS ASSOC PAIRLIS$ PAIRLIS-X2
     SYNTAXP QUOTEP))
)

(in-package "M1")

Note that PUSH (i.e., M1::PUSH) is undefined in this package!
The only ACL2 functions you can use are those listed above!