Operational Semantics 3
Compilation

Jim Royer

CIS 352

March 1, 2016

Interpreters and compilers

interpreter

$$\text{source code} \xrightarrow[\text{and parser}]{\text{via lexer}} \text{abstract syntax} \xrightarrow[\text{interpreter}]{\text{via evaluator/}} \text{value}$$

compiler

$$\text{source code} \xrightarrow[\text{and parser}]{\text{via lexer}} \text{abstract syntax} \xrightarrow{\text{via compiler}} \text{object code}$$
$$\xrightarrow{\text{via linker}} \text{executable} \xrightarrow[\text{interpreter}]{\text{via hardware}} \text{value}$$

There are many variations on the above.

Problem 1: Compile Aexp to a stack-based VM

Aexp

$v \in$ **Num** (Numeric Values) $\quad a \in$ **Aexp** (Arithmetic expressions)

$$a \ ::= \ v \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 * a_2)$$

A big-step semantics for Aexp

$$\textit{Num:}\ \frac{}{v \Downarrow v} \qquad \textit{Eval-}\circledast\textit{:}\ \frac{a_1 \Downarrow v_1 \qquad a_2 \Downarrow v_2}{(a_1 \circledast a_2) \Downarrow v}\ (v = v_1 \circledast v_2)$$

where $\circledast = +, -,$ and $*$.

What is our target VM?

Our target VM, 1

Memory banks

- 256 many 8 bit words
- so 8-bit addresses and 8-bit contents

used to store the stack, object code, and (later) registers.

Registers (internal)

PC = program counter (points to the current instruction)
SP = stack pointer (points to the top of the stack + 1)

Arithmetic

- mod 256 many 8 bit words
- So 255+1 = 0. (IMPORTANT!!!!)

Our target VM, 2

instructions

- Halt
- Push n
- Pop
- Add
- Sub
- Mult

What do the instructions do?

To precisely nail this down, we define a transition system given by a small-step operational semantics:

$$(pc, sp, stk) \Rightarrow (pc', sp', stk')$$

where:

obj	=	the object code	($\approx$ an array)
stk	=	the stack	($\approx$ an array)
pc	=	the program counter	($\approx$ an index into obj)
sp	=	the stack pointer	($\approx$ an index into stk)

Rule format

$$\textit{name:}\quad \frac{\ldots \textit{premises} \ldots}{obj \vdash (pc, sp, stk) \Rightarrow (pc', sp', stk')}\ \text{(side conditions)}$$

Our target VM, 3

$$\textit{Push:}\quad \frac{}{obj \vdash (pc, sp, stk) \Rightarrow (pc+2, sp+1, stk[sp \mapsto n])}\ \begin{pmatrix} obj[pc] = push, \\ obj[pc+1] = n \end{pmatrix}$$

$$\textit{Pop:}\quad \frac{}{obj \vdash (pc, sp, stk) \Rightarrow (pc+1, sp-1, stk)}\ (obj[pc] = pop)$$

$$\textit{Add:}\quad \frac{}{obj \vdash (pc, sp, stk) \Rightarrow (pc+1, sp-1, stk[sp-2 \mapsto n])}\ (*)$$

$$\vdots$$

$(*)$ $obj[pc] = add$ and $n = stk[sp-2] + stk[sp-1]$

N.B. Since pointer arithmetic is mod 256, underflow and overflow are wrap-arounds.

Evaluation/Compilation rules for Aexp

Num:	$\dfrac{}{v \Downarrow v}$	BSS rule
Num_{trans}:	$\dfrac{}{v \downarrow [Push\, v]}$	compilation rule
Plus:	$\dfrac{a_1 \Downarrow v_1 \qquad a_2 \Downarrow v_2}{(a_1 + a_2) \Downarrow v}\ (v = v_1 + v_2)$	BSS rule
$Plus_{trans}$:	$\dfrac{a_1 \downarrow I_1 \qquad a_2 \downarrow I_2}{(a_1 + a_2) \downarrow I_1 ++ I_2 ++ [Add]}$	compilation rule
	$\vdots$	

Haskell implementation in `vm0.hs`.

Questions

- Is the translation well-behaved?
  (In what condition does each expression leave the stack?)
- Is the translation correct?
  (No, we could easily overflow the stack.)
  (Yes, if we stay within size bounds. How to prove this?)

Proposition

Suppose

- a is an *Aexp* expression
- I_a is the sequence of instructions the compiler generates for a
- I_a is loaded into the code bank from address ℓ_0 to address ℓ_1.

Then $(\ell_0, sp, stk) \Rightarrow^* (\ell_1 + 1, sp + 1, stk[sp \mapsto v])$, where $a \Downarrow v$, *provided* there is no stack overflow or underflow.

Proof: By an easy structural induction on a.

Problem 2: Compile LC to a stack-based VM

LC Syntax and Base Types

(The Δ's mark changes.)

Phases	P	$::=$	$C \mid E \mid B$
Commands	C	$::=$	**skip** $\mid \ell := E \mid C; C$
		$\mid$	**if** B **then** C **else** $C \mid$ **while** B **do** C
Integer Expressons	E	$::=$	$n \mid !\ell \mid E \circledast E \quad (\circledast \in \{+, -, \times, \dots\})$
Boolean Expressons	B	$::=$	$b \mid E \circledast E \quad (\circledast \in \{=, <, \geq, \dots\})$
8-Bit Integers	$n \in \mathbb{Z}_{256}$	$=$	$\{0, 1, \dots, 255\}$ (Δ)
Booleans	$b \in \mathbb{B}$	$=$	$\{$ true, false $\}$
Locations	$\ell \in \mathbb{L}$	$=$	$\{\ell_0, \ell_1, \dots, \ell_{255}\}$ (Δ)

$!\ell \equiv$ the integer currently stored in ℓ

What is our target VM?

Our target VM, 1

memory banks

- 256 many 8 bit words
- so 8-bit addresses and 8-bit contents

used to store the stack, object code, and user registers.

Registers (internal)

pc = program counter (points to the current instruction)
sp = stack pointer (points to the top of the stack + 1)

Registers (user)

- 256-many 8-bit registers
- Named ℓ_0 through ℓ_{255} (or alternatively, 0 through 255)

Our target VM, 2

instructions

- Halt
- Push n
- Pop
- Fetch n
- Store n
- Iadd
- Isub
- Imult
- Ilt
- Jmp
- Jz
- Jnz

What do the instructions do?

Transition system on VM configs: $(pc, sp, stk, regs)$.

obj	=	the object code
stk	=	the stack
regs	=	the user registers
pc	=	the program counter
sp	=	the stack pointer

Rule format

$$\textit{name:}\ \frac{\dots \textit{premises} \dots}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp', stk', regs')}\ (*)$$

$(*)$ = side-conditions

Our target VM, 3

$$\textit{Fetch:}\ \frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc+2, sp+1, stk[sp \mapsto v], regs)}\ (*)$$

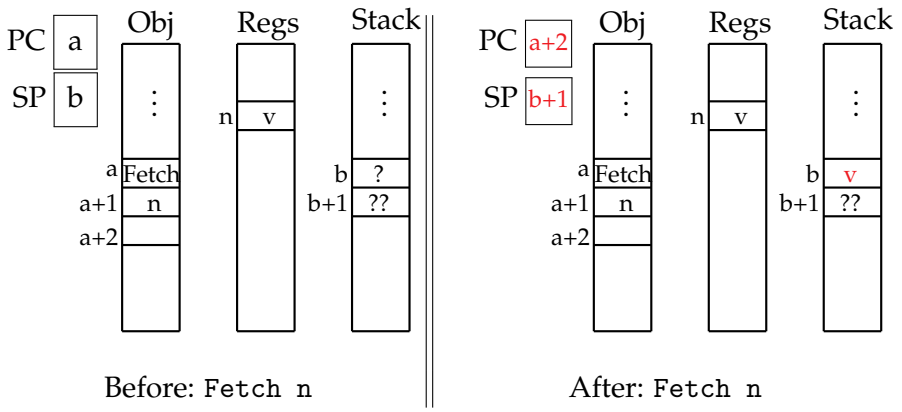
$$\textit{Store:}\ \frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc+2, sp, stk, regs[n \mapsto v])}\ (**)$$

$(*)$ $obj[pc] = fetch, \quad obj[pc+1] = n, \quad regs[n] = v$

$(**)$ $obj[pc] = store, \quad obj[pc+1] = n, \quad stk[sp-1] = v$

N.B. *Store* does **not** pop the stack!!!

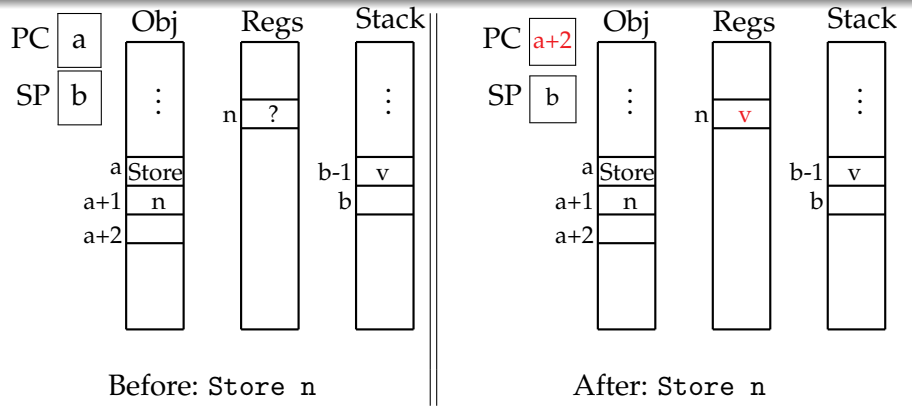
Fetch: Before and after



Fetch: $$\frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc+2, sp+1, stk[sp \mapsto v], regs)} \quad (*)$$

$(*)$ $obj[pc] = fetch, \quad obj[pc+1] = n, \quad regs[n] = v$

Store: Before and after



Store: $$\frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc+2, sp, stk, regs[n \mapsto v])} \quad (**)$$

$(**)$ $obj[pc] = store, \quad obj[pc+1] = n, \quad stk[sp-1] = v$

N.B. *Store* does **not** pop the stack!!!

Our target VM, 4

Ilt: $$\frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc+1, sp-1, stk[(sp-2) \mapsto v], regs)} \quad (*)$$

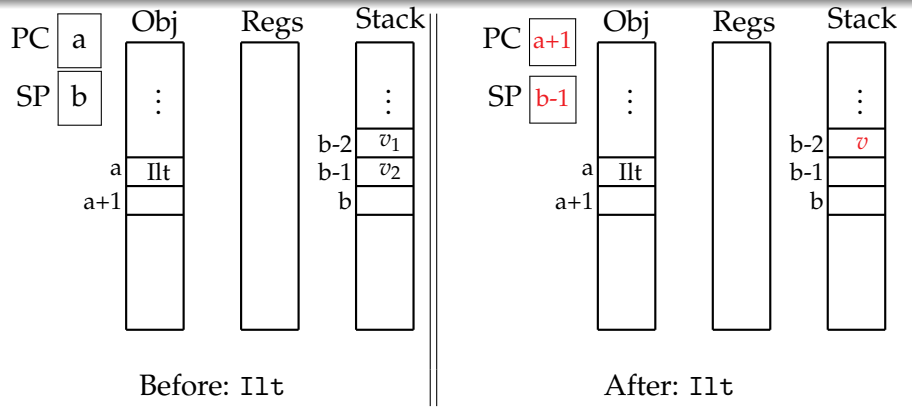
Jmp: $$\frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp, stk, regs)} \quad (**)$$

$(*)$ $obj[pc] = ilt, \quad stk[sp-2] = v_1, \quad stk[sp-1] = v_2,$ and
if $v_1 < v_2$ then $v = 1$ else $v = 0$

$(**)$ $obj[pc] = jmp, \quad obj[pc+1] = n, \quad pc' = pc + n + 1$

N.B. *Jmp* is a relative jump.

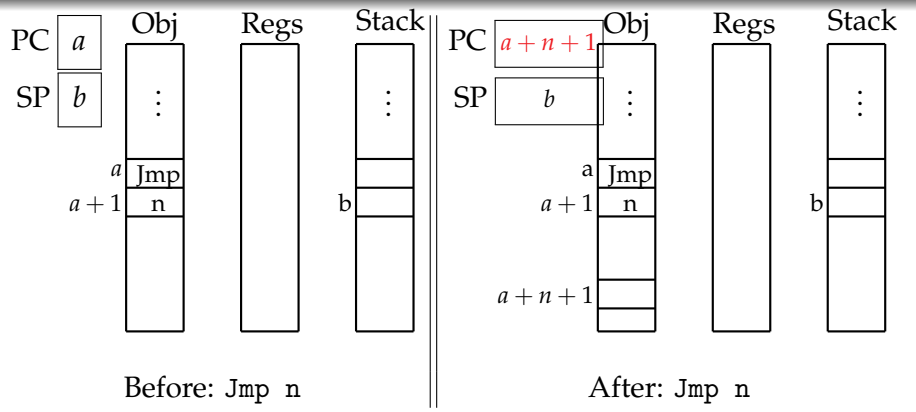
Ilt: Before and After



Ilt: $$\frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc+1, sp-1, stk[(sp-2) \mapsto v], regs)} \quad (*)$$

$(*)$ $obj[pc] = ilt, \quad stk[sp-2] = v_1, \quad stk[sp-1] = v_2,$ and
if $v_1 < v_2$ then $v = 1$ else $v = 0$

Jmp: Before and after



$$\textit{Jmp:}\quad \frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp, stk, regs)}\ (**)$$

$(**)$ $obj[pc] = jmp, \quad obj[pc+1] = n, \quad pc' = pc + n + 1$

N.B. *Jmp* is a relative jump.

Our target VM, 5

$$\textit{Jz:}\quad \frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp-1, stk, regs)}\ (*)$$

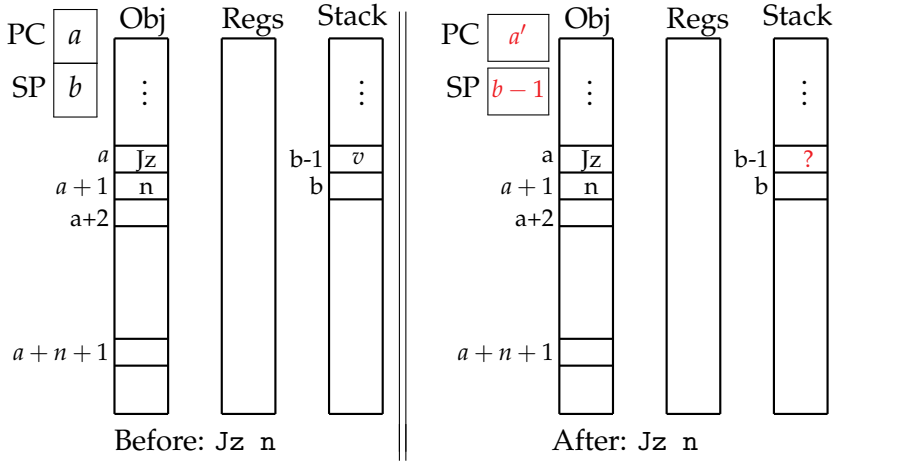
$$\textit{Jnz:}\quad \frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp-1, stk, regs)}\ (**)$$

$(*)$ $obj[pc] = jz,$
$obj[pc+1] = n,$
$stk[sp-1] = v,$ and
if $v = 0$ then $pc' = pc + n + 1$
else $pc' = pc + 2.$

$(**)$ $obj[pc] = jnz,$
$obj[pc+1] = n,$
$stk[sp-1] = v,$ and
if $v \neq 0$ then $pc' = pc + n + 1$
else $pc' = pc + 2.$

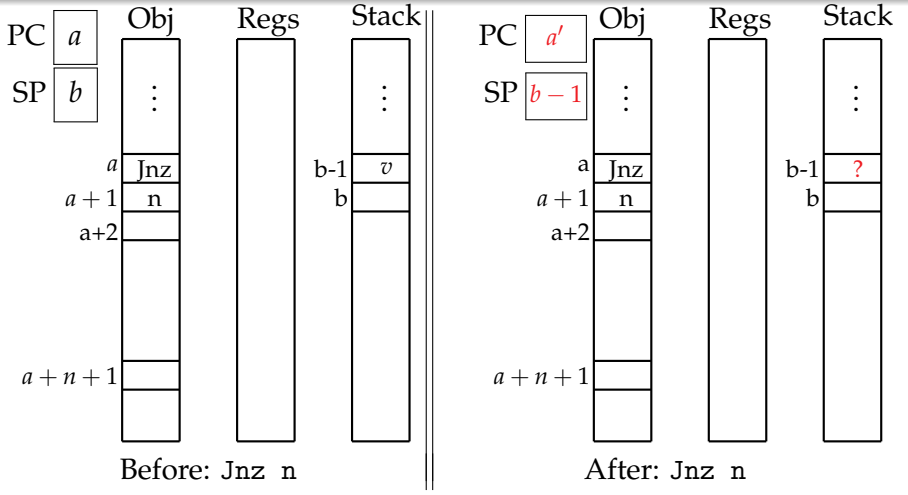
N.B. Both *Jz* and *Jnz* pop the stack!!!
N.B. Both *Jz* and *Jnz* are a relative jumps.

Jz: Before and After



$$\textit{Jz:}\quad \frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp-1, stk, regs)}\ \begin{pmatrix}\ldots \text{ \& if } v = 0\\ \text{then } pc' = pc + n + 1\\ \text{else } pc' = pc + 2\end{pmatrix}$$

Jnz: Before and After



$$\textit{Jnz:}\quad \frac{}{obj \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp-1, stk, regs)}\ \begin{pmatrix}\ldots \text{ \& if } v \neq 0\\ \text{then } pc' = pc + n + 1\\ \text{else } pc' = pc + 2\end{pmatrix}$$

Compiling integer and boolean expressions

- The compiler for integer expressions is a repeat of what we had before.
- For boolean expressions we maintain the convention that a boolean value is represented by either 0 (for False) or 1 (for True).

$$Bval_{trans}: \quad \frac{}{b \Rightarrow [Push\ v]} \quad \begin{pmatrix} v = 1 \text{ when } b = True, \\ v = 0 \text{ when } b = False \end{pmatrix}$$

$$Lt_{trans}: \quad \frac{ae_1 \Rightarrow I_1 \qquad ae_2 \Rightarrow I_2}{(ae_1 < ae_2) \Rightarrow I_1 ++ I_2 ++ [Ilt]}$$

$$Eq_{trans}: \quad \frac{a_1 \Rightarrow I_1 \qquad a_2 \Rightarrow I_2}{(ae_1 == ae_2) \Rightarrow I_1 ++ I_2 ++ [Isub, Push\ 1, Ilt]}$$

$$\vdots$$

Compiling statements, 1

$$Skip_{trans}: \quad \frac{}{Skip \Rightarrow []}$$

$$Assign_{trans}: \quad \frac{ae \Rightarrow I_0}{\ell_i := ae \Rightarrow I_0 ++ [Store\ i, Pop]}$$

$$Seq_{trans}: \quad \frac{S_1 \Rightarrow I_1 \qquad S_2 \Rightarrow I_2}{S_1; S_2 \Rightarrow I_1 ++ I_2}$$

Compiling statements, 2

$$If_{trans}: \quad \frac{be \Rightarrow I_0 \quad S_1 \Rightarrow I_1 \quad S_2 \Rightarrow I_2}{\textbf{if } be \textbf{ then } S_1 \textbf{ else } S_2 \Rightarrow I_0 ++ [Jz\ n_0] ++ I_1 ++ [Jmp\ n_1] ++ I_2} \quad (*)$$

$(*)$ $n_0 = 3 + codeLen(I_1)$ and
$n_1 = 1 + codeLen(I_2)$

$$While_{trans}: \quad \frac{be \Rightarrow I_0 \quad S \Rightarrow I_1}{\textbf{while } be \textbf{ do } S \Rightarrow I_0 ++ [Jz\ n_0] ++ I_1 ++ [Jmp\ n_1]} \quad (*)$$

$(*)$ $n_0 = 3 + codeLen(I_1)$ and
$n_1 = -(3 + codeLen(I_0) + codeLen(I_1))$

Implementation in `LCvm.hs` and `LCCompiler.hs`.

Questions

1. What does it mean for the compiler to be correct?
   Any run of a compiled program ends up with the state (register contents) dictated by the operational semantics of LC.
2. How does one prove that?
   Another structural induction on LC code.
3. Does compiled code behave well (e.g., always leaves the stack in some sensible condition)?
4. What about variables, blocks, procedures, etc.?