INF108: Compilation

Louis Jachiet

Conventions for compilation

When *compiling* it is important to follow strict conventions:

- bugs are hard to track
- bugs can be hard to trigger
- for compatibility with other tools, it is required

Conventions for P1

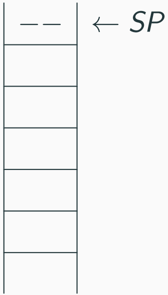
Convention 1

- Each expression is stored on the stack
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x

The effect of an expression e is therefore:

- SP is decreased by 4
- the value of e is stored in $0(SP)$

For instance for (1+2)-(3*4)



Conventions for P1

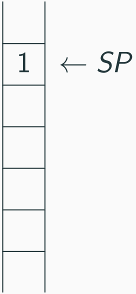
Convention 1

- Each expression is stored on the stack
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x

The effect of an expression e is therefore:

- SP is decreased by 4
- the value of e is stored in $0(SP)$

For instance for (1+2)-(3*4)



Conventions for P1

Convention 1

- Each expression is stored on the stack
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x

The effect of an expression e is therefore:

- SP is decreased by 4
- the value of e is stored in $0(SP)$

For instance for (1+2)-(3*4)

Conventions for P1

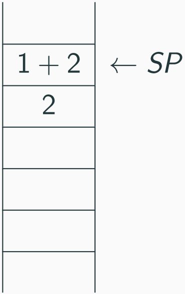
Convention 1

- Each expression is stored on the stack
- Each expression moves *SP* by exactly 4
- Each global variable x is stored in data at label `var_`x

The effect of an expression e is therefore:

- *SP* is decreased by 4
- the value of e is stored in $0(SP)$

For instance for (1+2)-(3*4)



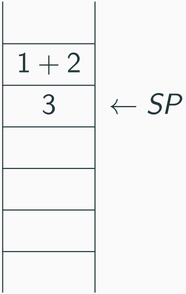
Convention 1

- Each expression is stored on the stack
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x

The effect of an expression e is therefore:

- SP is decreased by 4
- the value of e is stored in $0(SP)$

For instance for (1+2)-(3*4)



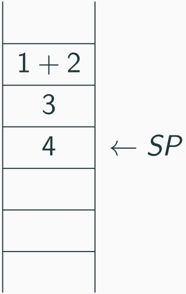
Convention 1

- Each expression is stored on the stack
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x

The effect of an expression e is therefore:

- SP is decreased by 4
- the value of e is stored in $0(SP)$

For instance for (1+2)-(3*4)



Convention 1

- Each expression is stored on the stack
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x

The effect of an expression e is therefore:

- SP is decreased by 4
- the value of e is stored in $0(SP)$

For instance for (1+2)-(3*4)

$1+2$	
$3*4$	$\leftarrow SP$
4	

Conventions for P1

Convention 1

- Each expression is stored on the stack
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x
- Each sub-expression can only modify data below SP

The effect of an expression e is therefore:

- SP is decreased by 4
- the value of e is stored in $0(SP)$

For instance for (1+2)-(3*4)

$(1+2)-(3*4)$	$\leftarrow SP$
$3*4$	
4	

What about let-in?

Semantics for let-in

Whats is the semantics of a let-in construct?

Semantics for let-in

Whats is a semantics?

Semantics for let-in

Let us note $[\![e]\!]_v$ the value given to e when evaluated in the environment v.

Semantics for let-in

$$[\![a \text{ op } b]\!]_v = [\![a]\!]_v \text{ op } [\![b]\!]_v$$

$$[\![\text{let } x = a \text{ in } b]\!]_v = [\![b]\!]_{v[x \to [\![a]\!]_v]}$$

where $v[x \to y]$ denotes the function $l \to \begin{cases} y & \text{when } x = l \\ v(l) & \text{otherwise} \end{cases}$

Conventions for P1

Convention 1

- Each expression is stored on the stack
- SP moves by exactly 4
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x
- Each sub-expression can only modify data below SP

Where to find our local variables?

Conventions for P1

Convention 1

- Each expression is stored on the stack
- SP moves by exactly 4
- Each expression moves SP by exactly 4
- Each global variable x is stored in data at label `var_`x
- Each sub-expression can only modify data below SP

Convention 2

Convention 1 + maintain the offset for each variable when compiling

Conventions for P1

Convention 1

- Each expression is stored on the stack
- *SP* moves by exactly 4
- Each expression moves *SP* by exactly 4
- Each global variable x is stored in data at label `var_`x
- Each sub-expression can only modify data below *SP*

Convention 2

Convention 1 + maintain the offset for each variable when compiling

kind of tedious...

Conventions for P1

Convention 3, we do not move SP but use an offset

- Each expression is stored on the stack
- SP does not move
- Each expression is given an offset O
- Each global variable x is stored in data at label $\texttt{var\_}x$
- Each local variable is stored in data at some reserved space at an offset $O' > O$
- Each sub-expression can only modify data below SP+offset

We maintain the offset for local variables recursively

What about functions?

Semantics for functions

$$[\![a \text{ op } b]\!]_v = [\![a]\!]_v \text{ op } [\![b]\!]_v$$

$$[\![\text{let } x = a \text{ in } b]\!]_v = [\![b]\!]_{v[x \to [\![a]\!]_v]}$$

$$[\![\text{f}(e)]\!]_v = [\![\text{let x}{=}e \text{ in def(f)}]\!]_v$$

Semantics for functions

$$\llbracket a \text{ op } b \rrbracket_v = \llbracket a \rrbracket_v \text{ op } \llbracket b \rrbracket_v$$

$$\llbracket \text{let } x = a \text{ in } b \rrbracket_v = \llbracket b \rrbracket_{v[x \to \llbracket a \rrbracket_v]}$$

$$\llbracket \text{f}(e) \rrbracket_v = \llbracket \text{let x=}e\text{ in def(f)} \rrbracket_v$$

Really?

Semantics for functions

$$[\![a \text{ op } b]\!]_{v_{loc} \cup v_{glob}} = [\![a]\!]_{v_{loc} \cup v_{glob}} \text{ op } [\![b]\!]_{v_{loc} \cup v_{glob}}$$

$$[\![\text{let } x = a \text{ in } b]\!]_{v_{loc} \cup v_{glob}} = [\![b]\!]_{v_{loc}[x \to [\![a]\!]_{v_{loc} \cup v_{glob}}] \cup v_{glob}}$$

$$[\![\text{f}(e)]\!]_{v_{loc} \cup v_{glob}} = [\![\text{f}]\!]_{[x \to y] \cup v_{glob}} \text{ with } y = [\![\text{e}]\!]_{v_{loc} \cup v_{glob}}$$

Conventions for P1

Convention 3

- Each expression is stored on the stack
- SP does not move
- Each expression is given an offset O
- Each global variable x is stored in data at label `var_`x
- Each local variable is stored in data at some reserved space at an offset $O' > O$
- Each sub-expression / sub-function can only modify data below $SP + O$

Can we make this work with functions?

Conventions for P1

Convention 3

- Each expression is stored on the stack
- SP does not move
- Each expression is given an offset O
- Each global variable x is stored in data at label `var_`x
- Each local variable is stored in data at some reserved space at an offset $O' > O$
- Each sub-expression / sub-function can only modify data below $SP + O$

Can we make this work with functions?

YES, we just need to move SP when calling functions!

Conventions for P1

Convention 4

- Each expression is stored on the stack
- SP **moves only for function calls**
- Each expression is given an offset O
- Each global variable x is stored in data at label `var_`x
- Each local variable is stored in data at some reserved space at an offset $O' > O$
- Each sub-expression / sub-function can only modify data below $SP + O$
- **Each function stores RA at $O - 4(SP)$ and its argument at $O(SP)$**

That is not very optimized...

Yes but:

- It is better to be correct than optimized

That is not very optimized...

Yes but:

- It is better to be correct than optimized
- We can adapt it a little

Conventions for P1

Convention 5

- Each expression result is stored on the $V0$
- SP moves only for function calls
- Each expression is given an offset O
- Each global variable x is stored in data at label `var_`x
- Each local variable is stored in data at some reserved space at an offset $O' > O$
- Each sub-expression can only modify data below $SP + O$
- Each function stores RA at $O - 4(SP)$ and its argument at $O(SP)$

Conventions for P1

Convention 5

- Each expression result is stored on the $V0$
- SP moves only for function calls
- Each expression is given an offset O
- Each global variable x is stored in data at label `var_`x
- Each local variable is stored in data at some reserved space at an offset $O' > O$
- Each sub-expression can only modify data below $SP + O$
- Each function stores RA at $O - 4(SP)$ and its argument at $O(SP)$

Warning When doing binop(e_1, e_2), we **need** to store the result of e_1 on the stack!

How to be sure to be correct?

Proofs

Semantics for the input

We have seen:

- $[\![a \text{ op } b]\!]_{v_{loc} \cup v_{glob}} = [\![a]\!]_{v_{loc} \cup v_{glob}} \text{ op } [\![b]\!]_{v_{loc} \cup v_{glob}}$
- $[\![\text{let } x = a \text{ in } b]\!]_{v_{loc} \cup v_{glob}} = [\![b]\!]_{v_{loc}[x \to [\![a]\!]_{v_{loc} \cup v_{glob}}] \cup v_{glob}}$
- $[\![\mathrm{f}(e)]\!]_{v_{loc} \cup v_{glob}} = [\![\mathrm{f}]\!]_{[x \to y] \cup v_{glob}}$ with $y = [\![e]\!]_{v_{loc} \cup v_{glob}}$

Semantics for the input

We have seen:

- $[\![a \text{ op } b]\!]_{v_{loc} \cup v_{glob}} = [\![a]\!]_{v_{loc} \cup v_{glob}} \text{ op } [\![b]\!]_{v_{loc} \cup v_{glob}}$
- $[\![\text{let } x = a \text{ in } b]\!]_{v_{loc} \cup v_{glob}} = [\![b]\!]_{v_{loc}[x \to [\![a]\!]_{v_{loc} \cup v_{glob}}] \cup v_{glob}}$
- $[\![\mathrm{f}(e)]\!]_{v_{loc} \cup v_{glob}} = [\![\mathrm{f}]\!]_{[x \to y] \cup v_{glob}}$ with $y = [\![e]\!]_{v_{loc} \cup v_{glob}}$

- How to deal with read?

Proofs

Semantics for the input

We have seen:

- $[\![a \text{ op } b]\!]_{v_{loc} \cup v_{glob}} = [\![a]\!]_{v_{loc} \cup v_{glob}} \text{ op } [\![b]\!]_{v_{loc} \cup v_{glob}}$
- $[\![\text{let } x = a \text{ in } b]\!]_{v_{loc} \cup v_{glob}} = [\![b]\!]_{v_{loc}[x \to [\![a]\!]_{v_{loc} \cup v_{glob}}] \cup v_{glob}}$
- $[\![\mathrm{f}(e)]\!]_{v_{loc} \cup v_{glob}} = [\![\mathrm{f}]\!]_{[x \to y] \cup v_{glob}}$ with $y = [\![e]\!]_{v_{loc} \cup v_{glob}}$

- How to deal with read?
- How to deal with print?

Proofs

We can make $[\![e]\!]_v$ return a value and a state.

- for read: $[\![\text{read } x]\!]_{\text{state}} = (\text{state}[x \to \text{read}()])$
- for print: $[\![\text{print } e]\!]_{\text{state}} = (\text{state} + \text{out}(v_e))$ with $(v_e, \text{state}_e) = [\![e]\!]_{\text{state}}$

Proofs

We can make $[\![e]\!]_v$ return a value and a state.

For expression not much changes:

- $[\![a \text{ op } b]\!]_{\text{state}} = [\![a]\!]_{\text{state}} \circ_{\text{op}} [\![b]\!]_{\text{state}}$
  where $(v_1, \text{state}_1) \circ_{\text{op}} (v_1, \text{state}_2) = (v_1 \text{ op } v_2, \text{state}_1)$

Proofs

We can make $[\![e]\!]_v$ return a value and a state.

For expression not much changes:

- $[\![a \text{ op } b]\!]_{\text{state}} = [\![a]\!]_{\text{state}} \circ_{\text{op}} [\![b]\!]_{\text{state}}$
  where $(v_1, \text{state}_1) \circ_{\text{op}} (v_1, \text{state}_2) = (v_1 \text{ op } v_2, \text{state}_1)$
- $[\![\text{let } x = a \text{ in } b]\!]_{\text{state}} = (v_b, \text{state}_1)$ where
  $(v_b, \text{state}_2) = [\![b]\!]_{\text{state}[x \to v_a]}$ and $(v_a, \text{state}_1) = [\![a]\!]_{\text{state}}$

Proofs

We can make $[\![e]\!]_v$ return a value and a state.

For expression not much changes:

- $[\![a \text{ op } b]\!]_{\text{state}} = [\![a]\!]_{\text{state}} \circ_{\text{op}} [\![b]\!]_{\text{state}}$
  where $(v_1, \text{state}_1) \circ_{\text{op}} (v_1, \text{state}_2) = (v_1 \text{ op } v_2, \text{state}_1)$
- $[\![\text{let } x = a \text{ in } b]\!]_{\text{state}} = (v_b, \text{state}_1)$ where
  $(v_b, \text{state}_2) = [\![b]\!]_{\text{state}[x \to v_a]}$ and $(v_a, \text{state}_1) = [\![a]\!]_{\text{state}}$
- $[\![\text{f}(e)]\!]_{\text{state}} = (v_f, \text{state})$ where $(v_f, \text{state}_f) = [\![\text{f}]\!]_{\text{state}[x \to y]}$ and
  $(v_e, \text{state}_e) = [\![e]\!]_{\text{state}}$

Proofs

We can make $[\![e]\!]_v$ return a value and a state.

For expression not much changes:

- $[\![a \text{ op } b]\!]_{\text{state}} = [\![a]\!]_{\text{state}} \circ_{\text{op}} [\![b]\!]_{\text{state}}$
  where $(v_1, \text{state}_1) \circ_{\text{op}} (v_1, \text{state}_2) = (v_1 \text{ op } v_2, \text{state}_1)$
- $[\![\text{let } x = a \text{ in } b]\!]_{\text{state}} = (v_b, \text{state}_1)$ where
  $(v_b, \text{state}_2) = [\![b]\!]_{\text{state}[x \to v_a]}$ and $(v_a, \text{state}_1) = [\![a]\!]_{\text{state}}$
- $[\![\text{f}(e)]\!]_{\text{state}} = (v_f, \text{state})$ where $(v_f, \text{state}_f) = [\![\text{f}]\!]_{\text{state}[x \to y]}$ and
  $(v_e, \text{state}_e) = [\![e]\!]_{\text{state}}$

Except if we want to take exceptions into account...

Proofs

We can make $[\![e]\!]_v$ return a value and a state.

- with $(v_a, \text{state}_a) = [\![a]\!]_{\text{state}}$ and $(v_b, \text{state}_b) = [\![b]\!]_{\text{state}}$ then either $[\![a \text{ op } b]\!]_{\text{state}} = (v_a \circ_{\text{op}} v_b, \text{state})$ or $[\![a \text{ op } b]\!]_{\text{state}} = \perp$ when $(v_a, \text{state}_a) = \perp$ or $(v_b, \text{state}_b) = \perp$

Proofs

We can make $[\![e]\!]_v$ return a value and a state.

- with $(v_a, \text{state}_a) = [\![a]\!]_{\text{state}}$ and $(v_b, \text{state}_b) = [\![b]\!]_{\text{state}}$ then either $[\![a \text{ op } b]\!]_{\text{state}} = (v_a \circ_{\text{op}} v_b, \text{state})$ or $[\![a \text{ op } b]\!]_{\text{state}} = \perp$ when $(v_a, \text{state}_a) = \perp$ or $(v_b, \text{state}_b) = \perp$

We can continue a long time like this...

Back to semantics

Denotational semantics

Denotational semantics is a way of formalizing the semantics of a AST by giving domains representation what programs do and composition rules.

Back to semantics

Denotational semantics

Denotational semantics is a way of formalizing the semantics of a AST by giving domains representation what programs do and composition rules.

The $[\![e]\!]_v$ notation is typically a denotational semantics.

Back to semantics

$$\frac{}{Cst(i), \sigma_l, \sigma_g \rightarrow i, \sigma_g}$$

Back to semantics

$$\frac{}{Cst(i), \sigma_l, \sigma_g \to i, \sigma_g} \qquad \frac{}{Var(x), \sigma_l, \sigma_g \to \sigma(x), \sigma_g}$$

Back to semantics

$$\frac{}{Cst(i), \sigma_l, \sigma_g \rightarrow i, \sigma_g} \qquad \frac{}{Var(x), \sigma_l, \sigma_g \rightarrow \sigma(x), \sigma_g}$$

$$\frac{e_1, \sigma_l, \sigma_g \rightarrow v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \rightarrow v_2, \sigma_g}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \rightarrow v_1 \text{ op}_{\text{int}} \ v_2, \sigma_g}$$

Back to semantics

$$\frac{}{Cst(i), \sigma_l, \sigma_g \to i, \sigma_g} \qquad \frac{}{Var(x), \sigma_l, \sigma_g \to \sigma(x), \sigma_g}$$

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \to v_2, \sigma_g}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \to v_1 \text{ op}_{\text{int}} \ v_2, \sigma_g}$$

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l[x/v_1], \sigma_g \to v_2, \sigma_g}{\text{let } x = e_1 \text{ in } e_2, \sigma_l, \sigma_g \to v_2, \sigma_g}$$

Back to semantics

$$\frac{}{Cst(i), \sigma_l, \sigma_g \to i, \sigma_g} \qquad \frac{}{Var(x), \sigma_l, \sigma_g \to \sigma(x), \sigma_g}$$

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \to v_2, \sigma_g}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \to v_1 \text{ op}_{\text{int}}\ v_2, \sigma_g}$$

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l[x/v_1], \sigma_g \to v_2, \sigma_g}{\text{let } x = e_1 \text{ in } e_2, \sigma_l, \sigma_g \to v_2, \sigma_g}$$

$$\frac{e, \sigma_l, \sigma_g \to v, \sigma_g \qquad \text{body}(f), \{x \to v\}, \sigma_g \to v', \sigma_g}{f(e), \sigma_l, \sigma_g \to v', \sigma_g}$$

Back to semantics

$$\frac{}{\text{read } x, \sigma_g, a :: t, Out \rightarrow \sigma_g[x/a], t, Out}$$

Back to semantics

$$\frac{}{\text{read } x, \sigma_g, a :: t, Out \to \sigma_g[x/a], t, Out}$$

$$\frac{e, \emptyset, \sigma_g \to v, \sigma_g}{\text{print } e, \sigma_g, In, Out \to \sigma_g, In, v :: Out}$$

Back to semantics

$$\frac{}{\text{read } x, \sigma_g, a :: t, Out \rightarrow \sigma_g[x/a], t, Out}$$

$$\frac{e, \emptyset, \sigma_g \rightarrow v, \sigma_g}{\text{print } e, \sigma_g, In, Out \rightarrow \sigma_g, In, v :: Out}$$

$$\frac{stmt, \sigma_g, In, Out \rightarrow \sigma_g^1, In_1, Out_1 \qquad prog, \sigma_g^1, In_1, Out_1 \rightarrow \sigma_g^2, In_2, Out_2}{stmt :: prog, \sigma, In, Out \rightarrow \sigma_2, In_2, Out_2}$$

Back to semantics: adding exceptions

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \to 0, \sigma_g}{e_1 / e_2, \sigma \to E(\mathsf{DivByZero})}$$

Back to semantics: adding exceptions

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \to 0, \sigma_g}{e_1 / e_2, \sigma \to E(\mathrm{DivByZero})}$$

$$\frac{e_1, \sigma_l, \sigma_g \to E(v) \qquad e_2, \sigma_l, \sigma_g \to v_2, \sigma_g}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \to E(v)}$$

Back to semantics: adding exceptions

$$\frac{e_1, \sigma_l, \sigma_g \rightarrow v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \rightarrow 0, \sigma_g}{e_1 / e_2, \sigma \rightarrow E(\text{DivByZero})}$$

$$\frac{e_1, \sigma_l, \sigma_g \rightarrow E(v) \qquad e_2, \sigma_l, \sigma_g \rightarrow v_2, \sigma_g}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \rightarrow E(v)}$$

$$\frac{e_1, \sigma_l, \sigma_g \rightarrow v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \rightarrow E(v)}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \rightarrow E(v)}$$

Back to semantics: adding exceptions

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \to 0, \sigma_g}{e_1/e_2, \sigma \to E(\text{DivByZero})}$$

$$\frac{e_1, \sigma_l, \sigma_g \to E(v) \qquad e_2, \sigma_l, \sigma_g \to v_2, \sigma_g}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \to E(v)}$$

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \to E(v)}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \to E(v)}$$

$$\frac{e_1, \sigma_l, \sigma_g \to (v_1, \sigma) \qquad e_2, \sigma_l, \sigma_g \to E(v)}{\text{let } x = e_1 \text{ in } e_2, \sigma_l, \sigma_g \to E(v)}$$

Back to semantics: adding exceptions

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \to 0, \sigma_g}{e_1 / e_2, \sigma \to E(\mathsf{DivByZero})}$$

$$\frac{e_1, \sigma_l, \sigma_g \to E(v) \qquad e_2, \sigma_l, \sigma_g \to v_2, \sigma_g}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \to E(v)}$$

$$\frac{e_1, \sigma_l, \sigma_g \to v_1, \sigma_g \qquad e_2, \sigma_l, \sigma_g \to E(v)}{e_1 \text{ op } e_2, \sigma_l, \sigma_g \to E(v)}$$

$$\frac{e_1, \sigma_l, \sigma_g \to (v_1, \sigma) \qquad e_2, \sigma_l, \sigma_g \to E(v)}{\text{let } x = e_1 \text{ in } e_2, \sigma_l, \sigma_g \to E(v)}$$

We can continue a long time like this...

Back to semantics

This kind of semantics is called natural or big-step semantics.

Back to semantics

This kind of semantics is called natural or big-step semantics.

Another kind of semantics is the small-step semantics.

Small-step semantics

$$\frac{}{Cst(i), \sigma \rightarrow (i, \sigma)}$$

Small-step semantics

$$\frac{}{Cst(i), \sigma \to (i, \sigma)} \qquad \frac{}{Var(x), \sigma \to (\sigma(x), \sigma)}$$

Small-step semantics

$$\frac{}{Cst(i), \sigma \rightarrow (i, \sigma)}$$

$$\frac{}{Var(x), \sigma \rightarrow (\sigma(x), \sigma)}$$

$$\frac{e_1, \sigma \rightarrow e_1', \sigma}{e_1 \text{ op } e_2, \sigma \rightarrow e_1' \text{ op } e_2, \sigma}$$

Small-step semantics

$$\frac{}{Cst(i), \sigma \rightarrow (i, \sigma)} \qquad \frac{}{Var(x), \sigma \rightarrow (\sigma(x), \sigma)}$$

$$\frac{e_1, \sigma \rightarrow e_1', \sigma}{e_1 \text{ op } e_2, \sigma \rightarrow e_1' \text{ op } e_2, \sigma} \qquad \frac{e_2, \sigma \rightarrow e_2', \sigma}{n \text{ op } e_2, \sigma \rightarrow n \text{ op } e_2', \sigma}$$

Small-step semantics

$$\frac{}{Cst(i), \sigma \to (i, \sigma)}$$

$$\frac{}{Var(x), \sigma \to (\sigma(x), \sigma)}$$

$$\frac{e_1, \sigma \to e_1', \sigma}{e_1 \text{ op } e_2, \sigma \to e_1' \text{ op } e_2, \sigma}$$

$$\frac{e_2, \sigma \to e_2', \sigma}{n \text{ op } e_2, \sigma \to n \text{ op } e_2', \sigma}$$

with $n' = n_1 \text{ op}_{int} n_2$

$$\frac{}{n_1 \text{ op } n_2, \sigma \to n', \sigma}$$

Small-step semantics

$$\frac{}{Cst(i), \sigma \to (i, \sigma)} \qquad \frac{}{Var(x), \sigma \to (\sigma(x), \sigma)}$$

$$\frac{e_1, \sigma \to e_1', \sigma}{e_1 \text{ op } e_2, \sigma \to e_1' \text{ op } e_2, \sigma} \qquad \frac{e_2, \sigma \to e_2', \sigma}{n \text{ op } e_2, \sigma \to n \text{ op } e_2', \sigma}$$

$$\frac{\text{with } n' = n_1 \text{ op}_{int} \ n_2}{n_1 \text{ op } n_2, \sigma \to n', \sigma} \qquad \frac{e, \sigma \to e', \sigma}{\text{let } x = e \text{ in } e_2, \sigma \to \text{let } x = e' \text{ in } e_2, \sigma}$$

Small-step semantics

$$\frac{}{Cst(i), \sigma \to (i, \sigma)} \qquad \frac{}{Var(x), \sigma \to (\sigma(x), \sigma)}$$

$$\frac{e_1, \sigma \to e_1', \sigma}{e_1 \text{ op } e_2, \sigma \to e_1' \text{ op } e_2, \sigma} \qquad \frac{e_2, \sigma \to e_2', \sigma}{n \text{ op } e_2, \sigma \to n \text{ op } e_2', \sigma}$$

$$\frac{\text{with } n' = n_1 \text{ op}_{int} \; n_2}{n_1 \text{ op } n_2, \sigma \to n', \sigma} \qquad \frac{e, \sigma \to e', \sigma}{\text{let } x = e \text{ in } e_2, \sigma \to \text{let } x = e' \text{ in } e_2, \sigma}$$

$$\frac{}{\text{let } x = n \text{ in } e_2, \sigma \to e_2[x/n], \sigma}$$

Big-step semantics for arithmetics

$$\frac{}{Cst(i) \to (i)}$$

$$\frac{e_1 \to v_1 \qquad e_2 \to v_2}{e_1 \text{ op } e_2 \to (v_1 \text{ op}_{\text{int}} \ v_2)}$$

Can we prove that our compilation is correct?

Our target language

With two variables and a stack:

- $push(i)$ for $i \in \mathbb{N}$
- push(a op $_{int} b$)
- a=pop()
- b=pop()
- stmt_1 ; stmt_2

Semantics of our language

$$\frac{}{\text{push}(i), s, v_a, v_b \rightarrow i :: s, v_a, v_b}$$

$$\frac{}{\text{push}(a \text{ op}_{int}\ b), s, v_a, v_b \rightarrow (v_a \text{ op}_{int}\ v_b) :: s, v_a, v_b}$$

$$\frac{}{a = \text{pop}(), i :: s, v_a, v_b \rightarrow s, i, v_b}$$

$$\frac{}{b = \text{pop}(), i :: s, v_a, v_b \rightarrow s, v_a, i}$$

$$\frac{\text{stmt}_1, s, v_a, v_b \rightarrow s', v_a', v_b' \qquad \text{stmt}_2, s', v_a', v_b' \rightarrow s'', v_a'', v_b''}{\text{stmt}_1; \text{stmt}_2, s, v_a, v_b \rightarrow s'', v_a'', v_b''}$$

Our compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  - $Compil(e_1)$;
  - $Compil(e_2)$;
  - $b = \text{pop}()$;
  - $a = \text{pop}()$;
  - $\text{push}(a \text{ op}_{int} \ b)$

Our compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  - $Compil(e_1)$;
  - $Compil(e_2)$;
  - $b = \text{pop}()$;
  - $a = \text{pop}()$;
  - $\text{push}(a \text{ op}_{int}\ b)$

We want to prove:

$$\frac{}{e \to i} \quad \Rightarrow \quad \frac{}{Compil(e), [], 0, 0 \to [i], v_a, v_b}$$

Our compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  - $Compil(e_1)$;
  - $Compil(e_2)$;
  - $b = \text{pop}()$;
  - $a = \text{pop}()$;
  - $\text{push}(a \text{ op}_{int} \ b)$

We want to prove:

$$\frac{}{e \to i} \quad \Rightarrow \quad \frac{}{Compil(e), s, v_a, v_b \to i :: s, v'_a, v'_b}$$

Proving the correctness of our compiler

The compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  $Compil(e_1); Compil(e_2); b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int}\ b)$

We will proceed by induction on the expressions.

Proving the correctness of our compiler

The compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  $Compil(e_1); Compil(e_2); b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int} \ b)$

We will proceed by induction on the expressions. For constants:

$$\frac{}{Cst(i) \to i} \quad \Rightarrow \quad \frac{}{\text{push}(i), s, v_a, v_b \to i :: s, v'_a, v'_b}$$

The compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  $Compil(e_1); Compil(e_2); b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int}\ b)$

We will proceed by induction on the expressions. For constants:

$$\frac{}{Cst(i) \to i} \quad \Rightarrow \quad \frac{}{Compil(Cst(i)), s, v_a, v_b \to i :: s, v'_a, v'_b}$$

Proving the correctness of our compiler (operation part)

The compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  $Compil(e_1); Compil(e_2); b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int} \ b)$

$$\frac{}{e_1 \text{ op } e_2 \to i} \quad \Rightarrow \quad \frac{e_1 \to v_1 \qquad e_2 \to v_2}{e_1 \text{ op } e_2 \to (v_1 \text{ op}_{\text{int}} \ v_2)}$$

Proving the correctness of our compiler (operation part)

The compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  $Compil(e_1); Compil(e_2); b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int}\ b)$

$$\frac{e_1 \to v_1 \qquad e_2 \to v_2}{e_1 \text{ op } e_2 \to (v_1 \text{ op}_{\text{int}}\ v_2)} \quad \Rightarrow \quad \begin{array}{c} \dfrac{}{Compil(e_1), s, v_a, v_b \to v_1 :: s, v'_a, v'_b} \\[2em] \dfrac{}{Compil(e_2), v_1 :: s, v'_a, v'_b \to v_2 :: v_1 :: s, v''_a, v''_b} \end{array}$$

BUT $$\frac{}{b = \text{pop}(), v_2 :: v_1 :: s, v''_a, v''_b \to v_1 :: s, v''_a, v_2}$$

Proving the correctness of our compiler (operation part)

The compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  $Compil(e_1); Compil(e_2); b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int}\ b)$

$$\frac{e_1 \to v_1 \qquad e_2 \to v_2}{e_1 \text{ op } e_2 \to (v_1 \text{ op}_{\text{int}}\ v_2)} \quad \Rightarrow$$

$$\frac{}{Compil(e_1); Compil(e_2), s, v_a, v_b \to v_2 :: v_1 :: s, v_a'', v_b''}$$

BUT
$$\frac{}{b = \text{pop}(), v_2 :: v_1 :: s, v_a'', v_b'' \to v_1 :: s, v_a'', v_2}$$

AND
$$\frac{}{a = \text{pop}(), v_1 :: s, v_a'', v_2 \to s, v_1, v_2}$$

Proving the correctness of our compiler (operation part)

The compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  $Compil(e_1); Compil(e_2); b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int}\ b)$

$$\frac{e_1 \to v_1 \qquad e_2 \to v_2}{e_1 \text{ op } e_2 \to (v_1 \text{ op}_{\text{int}}\ v_2)}$$

BUT

$$\frac{}{b = \text{pop}(), v_2 :: v_1 :: s, v_a'', v_b'' \to v_1 :: s, v_a'', v_2}$$

AND

$$\frac{}{a = \text{pop}(), v_1 :: s, v_a'', v_2 \to s, v_1, v_2}$$

AND

$$\frac{}{\text{push}(a \text{ op}_{int}\ b), s, v_1, v_2 \to (v_1 \text{ op}_{int}\ v_2) :: s, v_1, v_2}$$

Proving the correctness of our compiler (operation part)

$$\frac{e_1 \to v_1 \qquad e_2 \to v_2}{e_1 \text{ op } e_2 \to (v_1 \text{ op}_{\text{int}} \ v_2)}$$

BUT

$$\frac{}{b = \text{pop}(), v_2 :: v_1 :: s, v_a'', v_b'' \to v_1 :: s, v_a'', v_2}$$

AND

$$\frac{}{a = \text{pop}(), v_1 :: s, v_a'', v_2 \to s, v_1, v_2}$$

AND

$$\frac{}{\text{push}(a \text{ op}_{int} \ b), s, v_1, v_2 \to (v_1 \text{ op}_{int} \ v_2) :: s, v_1, v_2}$$

THUS

$$\frac{}{b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int} \ b), v_2 :: v_1 :: s, v_1, v_2 \to (v_1 \text{ op}_{int} \ v_2) :: s, v_1, v_2}$$

Proving the correctness of our compiler

The compiler

- $Compil(Cst(i)) = \text{push}(i)$
- $Compil(e_1 \text{ op } e_2) =$
  $Compil(e_1); Compil(e_2); b = \text{pop}(); a = \text{pop}(); \text{push}(a \text{ op}_{int}\ b)$

For operations:

$$\frac{}{e_1 \text{ op } e_2 \to i} \qquad \Rightarrow$$

$$\frac{}{Compil(a \text{ op }_{int} b), s, v_a, v_b \to (v_1 \text{ op}_{int}\ v_2) :: s, v_1, v_2}$$

$\square$

Things are always more complicated...

- We need to deal with exceptions

Things are always more complicated...

- We need to deal with exceptions
- We need to handle (global and local) variables

Things are always more complicated...

- We need to deal with exceptions
- We need to handle (global and local) variables
- We need to compile our intermediate language to assembly

Things are always more complicated...

- We need to deal with exceptions
- We need to handle (global and local) variables
- We need to compile our intermediate language to assembly
- We need to prove that there is a unique value that can be obtained

Things are always more complicated...

- We need to deal with exceptions
- We need to handle (global and local) variables
- We need to compile our intermediate language to assembly
- We need to prove that there is a unique value that can be obtained

  ...

Ok, proofs are too complicated what we do?

Testing!

Testing is **not**:

- a single expression
- a very limited number of expressions
- a multiplication of "1+x", "y*x", "1+3" because that does not really test all cases

Testing!

Testing is **not**:

- a single expression
- a very limited number of expressions
- a multiplication of “1+x”, “y*x”, “1+3” because that does not really test all cases

Tests **should**:

- test all features: all operations, let-in, functions, etc.

Testing!

Testing is **not**:

- a single expression
- a very limited number of expressions
- a multiplication of "1+x", "y*x", "1+3" because that does not really test all cases

Tests **should**:

- test all features: all operations, let-in, functions, etc.
- test all combinations: what happens when - is not on top, what happens when the same variables is bound twice, etc.

Testing!

Testing is **not**:

- a single expression
- a very limited number of expressions
- a multiplication of “1+x”, “y*x”, “1+3” because that does not really test all cases

Tests **should**:

- test all features: all operations, let-in, functions, etc.
- test all combinations: what happens when - is not on top, what happens when the same variables is bound twice, etc.
- test that everything goes well for complex cases

Testing!

Testing is **not**:

- a single expression
- a very limited number of expressions
- a multiplication of "1+x", "y*x", "1+3" because that does not really test all cases

Tests **should**:

- test all features: all operations, let-in, functions, etc.
- test all combinations: what happens when - is not on top, what happens when the same variables is bound twice, etc.
- test that everything goes well for complex cases
- use a sound baseline

Your tests vary between **mediocre**, **really bad** and **absent**...

My tests

On one of my simplest tests, among the **37** submitted projects, only $\sim$ **20** agree on this test (25 after simple fixes):

```
print ((((0+1)/(0+1))*2)-((0+1)*1))
```