Programming Language Theory

Operational Semantics for IMP
Looking back, looking forward

- Done: **IMP** syntax, structural induction

- Today: **IMP** operational semantics
  - One of the two or three most important lectures of course

- Tonight: You could (almost?) finish Homework 1
Review: **IMP** abstract syntax

**IMP**’s abstract syntax is defined inductively (using BNF):

\[
\begin{align*}
    s &::= x := e \mid \text{skip} \mid s ; s \mid \text{if } e s s \mid \text{while } e s \\
    e &::= c \mid x \mid e + e \mid e * e \\
    (c &\in \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \\
    (x &\in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \ldots\})
\end{align*}
\]

We haven’t yet said what programs *mean!* (Syntax is boring)

Encode our “social understanding” about variables and control flow.

Emphasis on “social understanding”:
- we define the meaning of a program language
- the meaning of a programming language is not dictated
Outline

- Semantics for expressions
  - Informal idea; the need for *heaps*
  - Definition of heaps
  - The evaluation *judgement* (a relation form)
  - The evaluation *inference rules* (the relation definition)
  - Using inference rules
    - *Derivation trees* as interpreters
    - Or as *proofs* about expressions
  - *Metatheory*: Proofs about the semantics
- Then semantics for statements
  - (rinse and repeat)
Informal idea

Given expression $e$, what integer $c$ does it evaluate to?

$$1 + 2 \quad \text{and} \quad x + 2$$
Informal idea

Given expression $e$, what integer $c$ does it evaluate to?

$$1 + 2 \quad x + 2$$

It depends on the values of variables (of course).

Use a heap $H$ to encode a total function from variables to constants.

- Could use partial functions,
  but then $\exists H$ and $e$ for which there is no $c$

We’ll define a relation over triples of $H$, $e$, and $c$.

- Will turn out to be a (total) function
  if we view $H$ and $e$ as inputs and $c$ as output.

- With our metalanguage, it is easier to define a relation
  and then prove that it is a function (if, in fact, it is).
Heaps

An abstract syntax for heaps:

\[ H ::= \cdot | H, x \mapsto c \]

Describe heaps as a data structure.

A lookup-function for heaps:

\[
H(x) = \begin{cases} 
  c & \text{if } H = H', x \mapsto c \\
  H'(x) & \text{if } H = H', y \mapsto c \text{ and } x \neq y \\
  0 & \text{if } H = \cdot 
\end{cases}
\]

Last case avoids “errors” (makes the function total)

“What heap to use” will arise in the semantics of statements

- For expression evaluation, “we are given an \( H \)"
The judgement

We will write:

\[ H;e \Downarrow c \]

to mean “\( e \) evaluates to \( c \) under heap \( H \)”. 

We just made up metasyntax \( H;e \Downarrow c \) to follow PL convention and to distinguish it from other relations.

It is just a relation on triples of the form \( (H, e, c) \).
The judgement

We will write:

\[ H; e \downarrow c \]

to mean “\( e \) evaluates to \( c \) under heap \( H \)”.

We just made up metasyntax \( H; e \downarrow c \) to follow PL convention and to distinguish it from other relations.

It is just a relation on triples of the form \((H, e, c)\).

We can write \( \cdot, x \mapsto 3; x + y \downarrow 3 \), which will turn out to be \textit{true} (this triple will be in the relation we define).

Or \( \cdot, x \mapsto 3; x + y \downarrow 6 \), which will turn out to be \textit{false} (this triple will not be in the relation we define).
Inference rules

\[
\begin{align*}
\text{CONST} & \quad H; c \Downarrow c \\
\text{ADD} & \quad H; e_1 \Downarrow c_1 \quad H; e_2 \Downarrow c_2 \\
& \quad H; e_1 + e_2 \Downarrow c_1 + c_2 \\
\text{VAR} & \quad H; x \Downarrow H(x) \\
\text{MULT} & \quad H; e_1 \Downarrow c_1 \quad H; e_2 \Downarrow c_2 \\
& \quad H; e_1 \ast e_2 \Downarrow c_1 \ast c_2
\end{align*}
\]

Top: *hypotheses*
Bottom: *conclusion*

By definition, if all hypotheses hold, then the conclusion holds.

Each rule is a *schema* that you “instantiate consistently”:

- Rules “work” “for all” \( H, c, e_1 \), etc.
- But “each” \( e_1 \) has to be the “same” expression.
Instantiating rules

Example instantiation:

\[
\cdot, y \mapsto 4;3 + y \downarrow 7 \quad \cdot, y \mapsto 4;5 \downarrow 5
\]

\[
\cdot, y \mapsto 4;(3 + y) + 5 \downarrow 12
\]

Instantiates:

\[
\text{ADD} \\
H;e_1 \downarrow c_1 \quad H;e_2 \downarrow c_2 \\
H;e_1 + e_2 \downarrow c_1 + c_2
\]

with:

\[
H = \cdot, y \mapsto 4 \\
e_1 = 3 + y \quad c_1 = 7 \\
e_2 = 5 \quad c_2 = 5
\]
Derivations

A \textit{(complete) derivation} is a tree of instantiations with axioms at the leaves.

Example:

\[
\begin{align*}
\cdot, y &\mapsto 4;3 \downarrow 3 & \text{CONST} \\
\cdot, y &\mapsto 4;4 \downarrow 4 & \text{VAR} \\
\cdot, y &\mapsto 4;3 + y \downarrow 7 & \text{ADD} \\
\cdot, y &\mapsto 4;(3 + y) + 5 \downarrow 12 & \text{ADD} \\
\end{align*}
\]

In theorems and proofs, we write “\(H;e \downarrow c\)” to mean “there exists a derivation with \(H;e \downarrow c\) at the root”.

Matthew Fluet
Programming Language Theory
Lecture 03
Relations

What relation do our inference rules define?

- Let $R_0$ be the empty relation (no triples).
- For $i > 0$, let $R_i$ be $R_{i-1}$ union all $H;e \Downarrow c$ such that we can instantiate some inference rule to have conclusion $H;e \Downarrow c$ and all its hypotheses in $R_{i-1}$.
- $R_i$ is all triples at the bottom of height-$j$ complete derivations for $j \leq i$.
- Let $R_\infty = \bigcup_{i \geq 0} R_i$. This is the relation we defined.
- $R_\infty$ is all triples at the bottom of complete derivations.

For the math folks:

$R_\infty$ is the smallest relation closed under the inference rules.
What are these things?

We can view the inference rules as defining an *interpreter*.

- Complete derivation shows (recursive) calls to the “eval exp” function.
  - Recursive calls from conclusion to hypotheses.
  - Syntax-directed means the interpreter need not “search” or “guess”.

- See SML code in next lecture and Homework 2.

Or we can view the inference rules as defining a *proof system*.

- Complete derivation proves facts (from other facts) starting with axioms.
  - Facts established from hypotheses to conclusions.

Note: Our semantics is *syntax-directed*.

- Exactly one inference rule for each variant of syntax.
On to statements

A statement doesn't produce an integer constant.
On to statements

A statement doesn't produce an integer constant.

It produces a new, possibly different, heap.
- If it terminates.

\[ H_1; s \downarrow H_2. \]

Would be a partial function from \( H_1 \) and \( s \) to \( H_2 \).

When would it not be defined?

Works fine; could be a homework problem.

Instead, we will define a "small-step" semantics and then "iterate" to "run the program".
On to statements

A statement doesn't produce an integer constant.

It produces a new, possibly different, heap.
▶ If it terminates.

We could define $H_1; s \downarrow H_2$.
▶ Would be a partial function from $H_1$ and $s$ to $H_2$.
  ▶ When would it not be defined?
▶ Works fine; could be a homework problem.
On to statements

A statement doesn't produce an integer constant. It produces a new, possibly different, heap.

▶ If it terminates.

We could define $H_1; s \downarrow H_2$.

▶ Would be a partial function from $H_1$ and $s$ to $H_2$.
  ▶ When would it not be defined?

▶ Works fine; could be a homework problem.

Instead, we will define a “small-step” semantics and then “iterate” to “run the program”.

$$H_1; s_1 \rightarrow H_2; s_2$$
Statement semantics

\[ H_1; s_1 \rightarrow H_2; s_2 \]

**Assign**

\[
\begin{align*}
H; e & \downarrow c \\
\hline \\
H; x := e & \rightarrow H, x \mapsto c; \text{skip}
\end{align*}
\]

**SeqSkip**

\[
\begin{align*}
H; \text{skip} & \rightarrow H; s
\end{align*}
\]

**SeqStep**

\[
\begin{align*}
H; s_1 & \rightarrow H'; s'_1 \\
H; s_1 ; s_2 & \rightarrow H'; s'_1 ; s_2
\end{align*}
\]

**IfT**

\[
\begin{align*}
H; e & \downarrow c \\
\hline \\
H; \text{if } e \ s_1 & \rightarrow H; s_1
\end{align*}
\]

**Iff**

\[
\begin{align*}
H; e & \downarrow c \\
\hline \\
H; \text{if } e \ s_1 & \rightarrow H; s_2
\end{align*}
\]
Statement semantics (cont’d)

What about while e s?

- Intuitively, do s and loop if e > 0.

\[
\text{WHILE} \\
\text{H;while e s} \Rightarrow \text{H;if e (s ; while e s) skip}
\]

Many other equivalent definitions possible.
Program semantics

Defined $H; s \rightarrow H'; s'$, but what does a whole program “s” mean/do?

Iterate:

$$H_1; s_1 \rightarrow H_2; s_2 \rightarrow H_3; s_3 \rightarrow \cdots$$

with each step justified by a complete derivation using our single-step statement semantics.
Let \( H_1; s_1 \rightarrow^n H_2; s_2 \) mean “\( H_1; s_1 \) becomes \( H_2; s_2 \) after \( n \) steps”.

Let \( H_1; s_1 \rightarrow^* H_2; s_2 \) mean “\( H_1; s_1 \) becomes \( H_2; s_2 \) after 0 or more steps”.

▶ “there exists some \( n \) such that \( H_1; s_1 \rightarrow^n H_2; s_2 \)”

Pick a special “answer” variable \( \text{ans} \).

The program \( s \) produces \( c \) if \( ;; s \rightarrow^* H; \text{skip} \) and \( H(\text{ans}) = c \).

Does every \( s \) produce a \( c \)?
Example program execution

\[ x := 3 ; (y := 1 ; \textbf{while} x (y := y \times x ; x := x + -1)) \]

Let’s write some of the state sequence. You can justify each step with a full derivation. Let \( s = (y := y \times x ; x := x + -1) \).

\[ \cdot ; x := 3 ; (y := 1 ; \textbf{while} x s) \]
Example program execution

\[ x := 3 ; (y := 1 ; \textbf{while} x (y := y \ast x ; x := x + -1)) \]

Let’s write some of the state sequence. You can justify each step with a full derivation.

Let \( s = (y := y \ast x ; x := x + -1) \).

\[
\cdot ; x := 3 ; (y := 1 ; \textbf{while} x s) \\
\frac{\text{assign}}{\text{seqStep}} \rightarrow \cdot, x \leftrightarrow 3 ; \text{skip} ; (y := 1 ; \textbf{while} x s)
\]
Example program execution

\[
x := 3 \; (y := 1 \; \text{while} \; x \; (y := y \times x \; \text{x} := x + -1))
\]

Let’s write some of the state sequence.
You can justify each step with a full derivation.
Let \( s = (y := y \times x \; \text{x} := x + -1) \).

\[
\cdot \; x := 3 \; (y := 1 \; \text{while} \; x \; s)
\]

\[
\begin{align*}
\text{ASSIGN} & \quad \rightarrow \quad \cdot, x \mapsto 3 \; \text{skip} \; (y := 1 \; \text{while} \; x \; s) \\
\text{SEQSTEP} & \quad \rightarrow \quad \cdot, x \mapsto 3 \; y := 1 \; \text{while} \; x \; s
\end{align*}
\]
Example program execution

\[ x := 3 \ ; \ (y := 1 \ ; \ \textbf{while} \ x \ (y := y \times x \ ; \ x := x + -1)) \]

Let’s write some of the state sequence.
You can justify each step with a full derivation.
Let \( s = (y := y \times x \ ; \ x := x + -1) \).

\[ \cdot \ ; \ x := 3 \ ; \ (y := 1 \ ; \ \textbf{while} \ x \ s) \]

\[
\begin{align*}
\text{ASSIGN} & \quad \rightarrow \quad \cdot, \ x \mapsto 3 \ ; \ \text{skip} \ ; \ (y := 1 \ ; \ \textbf{while} \ x \ s) \\
\text{SEQSTEP} & \quad \rightarrow \quad \cdot, \ x \mapsto 3 \ ; \ y := 1 \ ; \ \textbf{while} \ x \ s \\
\text{SEQSKIP} & \quad \rightarrow \quad \cdot, \ x \mapsto 3, \ y \mapsto 1 \ ; \ \text{skip} \ ; \ \textbf{while} \ x \ s
\end{align*}
\]
Example program execution

\[ x := 3 \ ; \ (y := 1 \ ; \textbf{while} \ x \ (y := y \times x \ ; \ x := x + -1)) \]

Let’s write some of the state sequence.
You can justify each step with a full derivation.
Let \( s = (y := y \times x \ ; \ x := x + -1) \).

\[
\begin{align*}
\text{\textbf{seqStep}} & \quad \rightarrow \quad \cdot \ ; \ x := 3 \ ; \ (y := 1 \ ; \textbf{while} \ x \ s) \\
\text{\textbf{assign}} & \quad \rightarrow \quad \cdot, x \mapsto 3 \ ; \textbf{skip} \ ; \ (y := 1 \ ; \textbf{while} \ x \ s) \\
\text{\textbf{seqSkip}} & \quad \rightarrow \quad \cdot, x \mapsto 3 \ ; \ y := 1 \ ; \textbf{while} \ x \ s \\
\text{\textbf{assign}} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1 \ ; \textbf{skip} \ ; \textbf{while} \ x \ s \\
\text{\textbf{seqSkip}} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1 \ ; \textbf{while} \ x \ s
\end{align*}
\]
Example program execution

\[
x := 3 ; (y := 1 ; \textbf{while} x (y := y * x ; x := x + -1))
\]

Let’s write some of the state sequence.
You can justify each step with a full derivation.
Let \( s = (y := y * x ; x := x + -1) \).

\[
\begin{align*}
\cdot & ; x := 3 ; (y := 1 ; \textbf{while} x s) \\
\text{\\textsc{assign}} \quad & \quad \rightarrow \cdot, x \mapsto 3 ; \textbf{skip} ; (y := 1 ; \textbf{while} x s) \\
\text{\\textsc{seqstep}} \quad & \quad \rightarrow \cdot, x \mapsto 3 ; y := 1 ; \textbf{while} x s \\
\text{\\textsc{seqskip}} \quad & \quad \rightarrow \cdot, x \mapsto 3, y \mapsto 1 ; \textbf{skip} ; \textbf{while} x s \\
\text{\\textsc{assign}} \quad & \quad \rightarrow \cdot, x \mapsto 3, y \mapsto 1 ; \textbf{while} x s \\
\text{\\textsc{seqskip}} \quad & \quad \rightarrow \cdot, x \mapsto 3, y \mapsto 1 ; \textbf{if} x (s ; \textbf{while} x s) \textbf{skip}
\end{align*}
\]
Example program execution (cont’d)

\[
\text{WHILE } \rightarrow \cdot, x \mapsto 3, y \mapsto 1; \text{ if } x (s; \text{while } x \ s) \text{ skip}
\]
Example program execution (cont’d)

\[
\text{WHILE} \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1; \text{if } x (s; \text{while } x \ s) \text{ skip}
\]

\[
\text{IFT} \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1; (y := y \times x; x := x + -1); \text{ while } x \ s
\]
Example program execution (cont’d)

\[
\text{WHILE } → ·, x ↦ 3, y ↦ 1 ; \text{if } x (s ; \text{while } x s) \text{ skip}
\]

\[
\text{IFT } → ·, x ↦ 3, y ↦ 1 ; (y := y \times x ; x := x + -1) ; \text{while } x s
\]

\[
\text{ASSIGN } \xrightarrow{\text{SEQSTEP}} → ·, x ↦ 3, y ↦ 1, y ↦ 3 ; (\text{skip} ; x := x + -1) ; \text{while } x s
\]
Example program execution (cont’d)

\[ \text{WHILE} \rightarrow \cdot, x \mapsto 3, y \mapsto 1; \text{if } x (s; \text{while } x s) \text{ skip} \]

\[ \text{IFT} \rightarrow \cdot, x \mapsto 3, y \mapsto 1; (y := y \times x; x := x + -1); \text{while } x s \]

\[ \frac{\text{ASSIGN}}{\text{SEQSTEP}} \rightarrow \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; (\text{skip}; x := x + -1); \text{while } x s \]

\[ \frac{\text{SEQSKIP}}{\text{SEQSTEP}} \rightarrow \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; x := x + -1; \text{while } x s \]
Example program execution (cont’d)

\[
\begin{align*}
\text{WHILE} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1; \text{if } x (s; \text{while } x \ s) \text{ skip} \\
\text{IFT} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1; (y := y \times x; x := x + -1); \text{while } x \ s \\
\text{ASSIGN} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; (\text{skip}; x := x + -1); \text{while } x \ s \\
\text{SEQStep} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; x := x + -1; \text{while } x \ s \\
\text{ASSIGN} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2; \text{skip}; \text{while } x \ s
\end{align*}
\]
Example program execution (cont’d)

\[
\text{WHILE} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; \text{if } x \ (s; \text{while } x \ s) \ \text{skip} \\
\text{IFT} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; (y := y \times x; x := x + -1) ; \text{while } x \ s \\
\text{ASSIGN} \ \text{SEQSTEP} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; (\text{skip}; x := x + -1) ; \text{while } x \ s \\
\text{SEQSTEP} \ \text{SEQSTEP} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; x := x + -1; \text{while } x \ s \\
\text{SEQSkip} \ \text{SEQSTEP} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2; \text{skip}; \text{while } x \ s \\
\text{SEQSTEP} \ \text{SEQSTEP} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2; \text{while } x \ s
\]
Example program execution (cont’d)

\[
\begin{align*}
\text{WHILE} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; \text{if } x (s; \text{while } x s) \text{ skip} \\
\text{IFT} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; (y := y \times x; x := x + -1) ; \text{while } x s \\
\text{ASSIGN} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; (y := y \times x; x := x + -1) \text{ while } x s \\
\text{SEQSTEP} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; y \mapsto 3; (\text{skip } x := x - 1) \text{ while } x s \\
\text{SEQSKIP} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; y \mapsto 3; x \mapsto x - 1 \text{ while } x s \\
\text{ASSIGN} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; y \mapsto 3; x \mapsto 2 \text{ skip } \text{ while } x s \\
\text{SEQSTEP} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; y \mapsto 3; x \mapsto 2 \text{ while } x s \\
\text{SEQSKIP} & \rightarrow \cdot, x \mapsto 3, y \mapsto 1; y \mapsto 3; x \mapsto 2 \text{ while } x s \\
\text{WHILE} & \rightarrow \ldots, y \mapsto 3, x \mapsto 2; \text{if } x (s; \text{while } x s) \text{ skip} \\
\ldots
\end{align*}
\]
Example program execution (cont’d)

\[
\begin{align*}
\text{WHILE} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1 \ ; \ \text{if } x \ (s \ ; \ \text{while } x \ s) \ \text{skip} \\
\text{IFT} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1 \ ; \ (y := y \ast x \ ; \ x := x + -1) \ ; \ \text{while } x \ s \\
\text{ASSIGN} \quad \text{SEQSTEP} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3 \ ; \ (\text{skip} \ ; \ x := x + -1) \ ; \ \text{while } x \ s \\
\text{SEQSTEP} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3 \ ; \ x := x + -1 \ ; \ \text{while } x \ s \\
\text{ASSIGN} \quad \text{SEQSTEP} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2 \ ; \ \text{skip} \ ; \ \text{while } x \ s \\
\text{SEQSTEP} & \quad \rightarrow \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2 \ ; \ \text{while } x \ s \\
\text{WHILE} & \quad \rightarrow \quad \ldots, y \mapsto 3, x \mapsto 2 \ ; \ \text{if } x \ (s \ ; \ \text{while } x \ s) \ \text{skip} \\
\ldots & \quad \rightarrow \quad \ldots, y \mapsto 6, x \mapsto 0 \ ; \ \text{skip}
\end{align*}
\]
Where we are

We have defined $H;e \downarrow c$ and $H;s \rightarrow H';s'$
and extended the latter to give a whole program $s$ a meaning.

We have used “operational semantics”:

- Definition by interpretation:
  - program means what an interpreter \textit{(written in a metalanguage)} says it means
  - interpreter for an \textit{abstract machine} \textit{(sometimes, very abstract)}
- The way we did expressions is “large-step” \textit{(or, “natural”)}
- The way we did statements is “small-step” \textit{(or, “structured”)}
Where we are

We have defined \( H;e \downarrow c \) and \( H;s \rightarrow H';s' \) and extended the latter to give a whole program \( s \) a meaning.

We have used “operational semantics”:

- Definition by interpretation:
  - program means what an interpreter (written in a metalanguage) says it means
  - interpreter for an abstract machine (sometimes, very abstract)
- The way we did expressions is “large-step” (or, “natural”)
- The way we did statements is “small-step” (or, “structured”)

Large-step does not distinguish errors and divergence.

- But we defined IMP to have no errors
- And expressions never diverge

Large-step simpler than small-step when appropriate.
Judgements and Inference Rules

There is a lot of convention built into judgements and inference rules:

▶ conclusion must be a judgement
▶ hypotheses must be judgements or logical formulae
▶ judgements in conclusion or hypotheses may be constrained to elements of a specific syntactic form
▶ metavariables that appear in conclusion are $\forall$ quantified
▶ metavariables that do not appear in conclusion are $\exists$ quantified
▶ repeated metavariables are the “same”
  ▶ contrast with BNF, where repeated metavariables are “different”
Judgements and Inference Rules

This,

\[
\begin{align*}
H; e_1 \downarrow c_1 & \quad H; e_2 \downarrow c_2 \\
\hline \\
H; e_1 + e_2 \downarrow c_1 + c_2
\end{align*}
\]

is equivalent to

\[
\begin{align*}
e = e_1 + e_2 & \\
\hline \\
H; e_1 \downarrow c_1 & \quad H; e_2 \downarrow c_2 & \quad c = c_1 + c_2 \\
\hline \\
H; e \downarrow c
\end{align*}
\]

which is read:

For all heaps \( H \), expressions \( e \), and constants \( c \),
if there exist expressions \( e_1 \) and \( e_2 \) and constants \( c_1 \) and \( c_2 \)
and \( e = e_1 + e_2 \), \( H; e_1 \downarrow c_1 \), \( H; e_2 \downarrow c_2 \), and \( c = c_1 + c_2 \) are all true (provable),
then \( H; e \downarrow c \) is true (provable).
Judgements and Inference Rules

There is a lot of convention built into judgements and inference rules:

- conclusion must be a judgement
- hypotheses must be judgements or logical formulae
- judgements in conclusion or hypotheses may be constrained to elements of a specific syntactic form
- metavariables that appear in conclusion are $\forall$ quantified
- metavariables that do not appear in conclusion are $\exists$ quantified
- repeated metavariables are the “same”
  - contrast with BNF, where repeated metavariables are “different”
Judgements and Inference Rules

Judgements and inference rules are versatile PL notation.

Could have defined syntax using a judgement and inference rules:

\[
\begin{align*}
& e \quad \text{exp} \\
\quad & c \quad \text{exp} \\
\quad & e_1 \quad \text{exp} \quad e_2 \quad \text{exp} \\
& e_1 + e_2 \quad \text{exp} \\
\quad & x \quad \text{exp} \\
\quad & e_1 \quad \text{exp} \quad e_2 \quad \text{exp} \\
& e_1 * e_2 \quad \text{exp}
\end{align*}
\]
Judgements and Inference Rules

Judgements and inference rules are versatile PL notation.

Could have defined heap lookup using a judgement and inference rules:

\[
H \circ x \leadsto c
\]

**EMPTY**

\[
\cdot \circ x \leadsto 0
\]

**HIT**

\[
H', x \mapsto c \circ x \leadsto c
\]

**MISS**

\[
x \neq y' \quad H' \circ x \leadsto c
\]

\[
H', y' \mapsto c' \circ x \leadsto c
\]
Judgements and Inference Rules

Judgements and inference rules are versatile PL notation.

Could have defined multi-step statement semantics using judgements and inference rules:

\[ H_1; s_1 \rightarrow^n H_2; s_2 \]

\[ \frac{H_1; s \rightarrow^0 H; s}{H; s \rightarrow^n H; s} \]

\[ \frac{H_1; s_1 \rightarrow^n H_2; s_2}{H_1; s_1 \rightarrow^{n+1} H_3; s_3} \]

\[ H_1; s_1 \rightarrow^* H_2; s_2 \]

\[ \frac{H_1; s_1 \rightarrow^n H_2; s_2}{H_1; s_1 \rightarrow^* H_2; s_2} \]

Note: With these inference rules, complete derivations of \( H_1; s_1 \rightarrow^* H_2; s_2 \) are unique.
Judgements and Inference Rules

Judgements and inference rules are versatile PL notation.

Could have defined multi-step statement semantics using judgements and inference rules:

\[ H_1; s_1 \rightarrow^* H_2; s_2 \]

\[
\frac{H; s \rightarrow^* H; s}{H; s \rightarrow^* H; s}
\]

\[
\frac{H_1; s_1 \rightarrow^* H_2; s_2 \quad H_2; s_2 \rightarrow^* H_3; s_3}{H_1; s_1 \rightarrow^* H_3; s_3}
\]

\[
\frac{H_1; s_1 \rightarrow H_2; s_2}{H_1; s_1 \rightarrow^* H_2; s_2}
\]

Note: With these inference rules, complete derivations of \( H_1; s_1 \rightarrow^* H_2; s_2 \) are not unique.
Judgements and Inference Rules

Judgements and inference rules are versatile PL notation.

Could have defined program semantics using a judgement and inference rule:

\[ s \rightarrow^* c \]

\[ \cdot; s \rightarrow^* H; \text{skip} \quad H @ \text{ans} \rightsquigarrow c \]

\[ s \rightarrow^* c \]
Review: **IMP** abstract syntax (programs and heaps)

\[
\begin{align*}
s &::= \ x := e \mid \text{skip} \mid s ; s \mid \text{if } e \ s \ s \mid \text{while } e \ s \\
e &::= \ c \mid x \mid e + e \mid e * e \\
& \quad (c \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \\
& \quad (x \in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \ldots\})
\end{align*}
\]

\[
\begin{align*}
H &::= \cdot \mid H, x \mapsto c
\end{align*}
\]
Review: IMP judgement for heap lookup

\[ H @ x \leadsto c \]

**EMPTY**

\[
\cdot @ x \leadsto 0
\]

**HIT**

\[
H', x \mapsto c @ x \leadsto c
\]

**MISS**

\[
x \neq y' \\
H' @ x \leadsto c \\
H', y' \mapsto c' @ x \leadsto c'
\]
**Review: IMP operational semantics for expressions** (big-step)

\[ H; e \downarrow c \]

**CONST**

\[
\begin{array}{c}
H; c \downarrow c \\
\end{array}
\]

**VAR**

\[
\begin{array}{c}
H @ x \rightsquigarrow c \\
H; x \downarrow c \\
\end{array}
\]

**ADD**

\[
\begin{array}{c}
H; e_1 \downarrow c_1 \quad H; e_2 \downarrow c_2 \\
H; e_1 + e_2 \downarrow c_1 + c_2 \\
\end{array}
\]

**MULT**

\[
\begin{array}{c}
H; e_1 \downarrow c_1 \quad H; e_2 \downarrow c_2 \\
H; e_1 * e_2 \downarrow c_1 * c_2 \\
\end{array}
\]
Review: **IMP** operational semantics for statements (small-step)

\[ H_1; s_1 \rightarrow H_2; s_2 \]

**ASSIGN**

\[
\frac{H; e \downarrow c}{H; x := e \rightarrow H, x \mapsto c; \text{skip}}
\]

**WHILE**

\[
\frac{H; \text{while } e \ s \rightarrow}{H; \text{if } e (s; \text{while } e \ s) \ \text{skip}}
\]

**SEQ_SKIP**

\[
\frac{H; \text{skip } ; s \rightarrow H; s}{H; \text{skip } ; s} \]

**SEQ_STEP**

\[
\frac{H; s_1 \rightarrow H'; s'_1}{H; s_1 ; s_2 \rightarrow H'; s'_1 ; s_2}
\]

**IFT**

\[
\frac{H; e \downarrow c \qquad c > 0}{H; \text{if } e \ s_1 \ s_2 \rightarrow H; s_1}
\]

**IFF**

\[
\frac{H; e \downarrow c \qquad c \leq 0}{H; \text{if } e \ s_1 \ s_2 \rightarrow H; s_2}
\]
Preview: Establishing properties about all expressions

We can prove properties about all expressions (i.e., about \textbf{IMP}):

- **Progress:**
  
  For all $H$ and $e$, there exists $c$ such that $H;e \downarrow c$.

- **Determinacy:**
  
  For all $H$, $e$, $c_1$, and $c_2$,
  
  if $H;e \downarrow c_1$ and $H;e \downarrow c_2$, then $c_1 = c_2$.

We rigged it that way... 

- **What would division, undefined variables, or \texttt{rand()} do?**

Proofs are by induction on the structure of the expression $e$.

- **Proofs require lemmas for “progress” and “determinacy” of heaps.**

- **Details in a few lectures and Homework 3.**
Preview: Establishing properties about a program

We can prove a property about a terminating program by “running” it.

Example: Our last program terminates with $x$ holding 0.
Establishing properties about a program

We can prove a property about a terminating program by “running” it.

Example: Our last program terminates with $x$ holding $0$.

We can prove that a program $s$ diverges:

- for all $H$ and $n$, $s \rightarrow^n H;\text{skip}$ cannot be derived.

Example: $\text{while 1 skip}$
We can prove a property about a terminating program by “running” it.

Example: Our last program terminates with \( x \) holding 0.

We can prove that a program \( s \) diverges:

\[
\text{for all } H \text{ and } n, \; \cdot; s \rightarrow^n H; \text{skip} \text{ cannot be derived.}
\]

Example: \texttt{while 1 skip}

Proof is by induction on \( n \) with a stronger induction hypothesis:
If we can derive \( \cdot; \text{while 1 skip} \rightarrow^n \cdot; s' \)
then \( s' \) is \texttt{while 1 skip}
or \( s' \) is \texttt{if 1 (skip ; while 1 skip) skip}
or \( s' \) is \texttt{skip ; while 1 skip}.

Details in a few lectures.
Establishing properties about all programs

We can prove properties about all programs (i.e., about \textbf{IMP}):

- **Progress:**
  
  For all \( H \) and \( s \), there exists \( H' \) and \( s' \) such that \( H; s \rightarrow H'; s' \).

- **Determinacy:**
  
  For all \( H, s, H_1', s_1', H_2', \) and \( s_2' \), if \( H; s \rightarrow H_1'; s_1' \) and \( H; s \rightarrow H_2'; s_2' \), then \( H_1' = H_2' \) and \( s_1' = s_2' \).
Establishing properties about all programs

We can prove properties about all programs (i.e., about IMP):

- **Progress:**
  
  For all $H$ and $s$, there exists $H'$ and $s'$ such that $H;s \rightarrow H';s'$.

- **Determinacy:**
  
  For all $H$, $s$, $H'_1$, $s'_1$, $H'_2$, and $s'_2$,
  
  if $H;s \rightarrow H'_1;s'_1$ and $H;s \rightarrow H'_2;s'_2$, then $H'_1 = H'_2$ and $s'_1 = s'_2$.

One of these properties is not true...
Establishing properties about all programs

We can prove properties about all programs (i.e., about \texttt{IMP}):

- **Progress:**
  
  For all \(H\) and \(s\), there exists \(H'\) and \(s'\) such that \(H; s \rightarrow H'; s'\).

- **Determinacy:**
  
  For all \(H, s, H'_1, s'_1, H'_2, \) and \(s'_2\), if \(H; s \rightarrow H'_1; s'_1\) and \(H; s \rightarrow H'_2; s'_2\), then \(H'_1 = H'_2\) and \(s'_1 = s'_2\).

One of these properties is not true...
Establishing properties about all programs

We can prove properties about all programs (satisfying a property).
Establishing properties about all programs

We can prove properties about all programs (satisfying a property).

Example: If $H$ and $s$ have no negative constants and $H; s \rightarrow H'; s'$, then $H'$ and $s'$ have no negative constants.
Establishing properties about all programs

We can prove properties about all programs (satisfying a property).

Example: If $H$ and $s$ have no negative constants and $H; s \rightarrow H'; s'$, then $H'$ and $s'$ have no negative constants.

Proof is by structural induction on the derivation $H; s \rightarrow H'; s'$. Details next time.
Establishing properties about all programs

We can prove properties about all programs (satisfying a property).

Example: If $H$ and $s$ have no negative constants and $H; s \rightarrow H'; s'$, then $H'$ and $s'$ have no negative constants.

Proof is by structural induction on the derivation $H; s \rightarrow H'; s'$. Details next time.

Example: If for all $H$, we know $s_1$ and $s_2$ terminate, then for all $H$, we know $s_1 ; s_2$ terminates.
Establishing properties about all programs

We can prove properties about all programs (satisfying a property).

Example: If $H$ and $s$ have no negative constants and $H;s \rightarrow H';s'$, then $H'$ and $s'$ have no negative constants.

Proof is by structural induction on the derivation $H;s \rightarrow H';s'$. Details next time.

Example: If for all $H$, we know $s_1$ and $s_2$ terminate, then for all $H$, we know $s_1 ; s_2$ terminates.

Proof is almost direct (but needs a lemma).

$$H;s_1 ; s_2 \rightarrow^* H';\text{skip} \quad ; s_2 \rightarrow H';s_2 \rightarrow^* H'';\text{skip}$$