Where we are

- Done: OCaml tutorial, “IMP” syntax, structural induction
- Now: Operational semantics for our little “IMP” language
  - Most of what you need for Homework 1
  - (But Problem 4 requires proofs over semantics)
IMP’s abstract syntax is defined inductively:

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

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

Encode our “social understanding” about variables and control flow
Semantics for expressions

1. Informal idea; the need for heaps
2. Definition of heaps
3. The evaluation judgment (a relation form)
4. The evaluation inference rules (the relation definition)
5. Using inference rules
   - Derivation trees as interpreters
   - Or as proofs about expressions
6. Metatheory: Proofs about the semantics

Then semantics for statements
   - ...
Informal idea

Given $e$, what $c$ does $e$ evaluate to?

$$1 + 2$$

$$x + 2$$
Informal idea

Given $e$, what $c$ does $e$ evaluate to?

\[ 1 + 2 \quad x + 2 \]

It depends on the values of variables (of course)

Use a heap $H$ for 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 function if we view $H$ and $e$ as inputs and $c$ as output
- With our metalanguage, easier to define a relation and then prove it is a function (if, in fact, it is)
**Heaps**

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

A lookup-function for heaps:

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

- Last case avoids “errors” (makes function *total*).
- “What heap to use” will arise in the semantics of statements.
- For expression evaluation, “we are given an H’”
The judgment

We will write: \( H ; e \downarrow c \)

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

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

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

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

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

**CONST**

\[
H ; c \downarrow c
\]

**VAR**

\[
H ; x \downarrow H(x)
\]

**ADD**

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

**MULT**

\[
\frac{H ; e_1 \downarrow c_1 \quad H ; e_2 \downarrow c_2}{H ; e_1 * e_2 \downarrow c_1 * c_2}
\]

Top: hypotheses
Bottom: conclusion (read first)

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

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

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

Example instantiation:

\[
\begin{align*}
\cdot, y \rightarrow 4 & \triangleleft 3 + y \triangleright 7 \\
\cdot, y \rightarrow 4 & \triangleleft 5 \triangleright 5 \\
\cdot, y \rightarrow 4 & \triangleleft (3 + y) + 5 \triangleright 12
\end{align*}
\]

Instantiates:

\[
\begin{align*}
\text{ADD} \\
H \triangleright e_1 \triangleleft c_1 \quad H \triangleright e_2 \triangleleft c_2
\end{align*}
\]

\[
H \triangleright e_1 + e_2 \triangleleft c_1 + c_2
\]

with

\[
H = \cdot, y \rightarrow 4
\]

\[
e_1 = (3 + y)
\]

\[
c_1 = 7
\]

\[
e_2 = 5
\]

\[
c_2 = 5
\]
Derivations

A *(complete)* derivation is a tree of instantiations with *axioms* at the leaves

Example:

\[
\begin{align*}
\cdot, y &\rightarrow 4 ; 3 \downarrow 3 \\
\cdot, y &\rightarrow 4 ; y \downarrow 4 \\
\cdot, y &\rightarrow 4 ; 3 + y \downarrow 7 \\
\cdot, y &\rightarrow 4 ; (3 + y) + 5 \downarrow 12 \\
\end{align*}
\]

By definition, \( H ; e \downarrow c \) if there exists a derivation with \( H ; e \downarrow c \) at the root
Back to relations

So what relation do our inference rules define?

- Start with empty relation (no triples) $R_0$

- 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 hypotheses in $R_{i-1}$
  - So $R_i$ is all triples at the bottom of height-$j$ complete derivations for $j \leq i$

- $R_\infty$ is the relation we defined
  - 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 “evaluate expression” function
  - Recursive calls from conclusion to hypotheses
  - Syntax-directed means the interpreter need not “search”

- See OCaml code in Homework 1

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
Some theorems

- Progress: For all $H$ and $e$, there exists a $c$ such that $H ; e \Downarrow c$

- Determinacy: For all $H$ and $e$, there is at most one $c$ such that $H ; e \Downarrow c$

We rigged it that way...
what would division, undefined-variables, or gettime() do?

Proofs are by induction on the the structure (i.e., height) of the expression $e$
On to statements

A statement does not produce a constant
On to statements

A statement does not produce a constant

It produces a new, possibly-different heap.

▶ If it terminates
On to statements

A statement does not produce a 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 \( H_2 \)
  ▶ Works fine; could be a homework problem
On to statements

A statement does not produce a 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 \( H_2 \)

▶ Works fine; could be a homework problem

Instead we’ll 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**

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

**SEQ1**

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

**SEQ2**

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

**IF1**

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

**IF2**

\[
\frac{H ; e \Downarrow c \quad c \leq 0}{H ; \text{if } e \; s_1 \; s_2 \rightarrow H \; s_2}
\]
Statement semantics cont’d

What about \textbf{while }e\textbf{ s} (do s and loop if e > 0)?
What about `while e s` (do `s` and loop if `e > 0`)?

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

Many other equivalent definitions possible
Program semantics

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

Our machine iterates: $H_1; s_1 \rightarrow H_2; s_2 \rightarrow H_3; s_3 \ldots$, 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 “becomes after $n$ steps”

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

Pick a special “answer” variable $\texttt{ans}$

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

Does every $s$ produce a $c$?
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 \( s = (y := y \ast x; x := x-1) \).
Example program execution

\begin{verbatim}
x := 3; (y := 1; while x (y := y * x; x := x−1))
\end{verbatim}

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

\begin{verbatim}
.; x := 3; y := 1; while x s
\end{verbatim}
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 \( s = (y := y \times x; x := x - 1) \).

\[
\bullet; x := 3; y := 1; \textbf{while} \ x \ s \\
\rightarrow \quad \cdot, x \mapsto 3; \textbf{skip}; y := 1; \textbf{while} \ x \ s
\]
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 \( s = (y := y \times x; x := x - 1) \).

\[
\begin{array}{c}
\cdot; x := 3; y := 1; \textbf{while} \ x \ s \\
\rightarrow \cdot, x \mapsto 3; \textbf{skip}; y := 1; \textbf{while} \ x \ s \\
\rightarrow \cdot, x \mapsto 3; y := 1; \textbf{while} \ x \ s
\end{array}
\]
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 \( s = (y := y \times x; x := x - 1) \).

\[ ·; x := 3; y := 1; \textbf{while} x \ s \]

\[ \rightarrow \cdot, x \mapsto 3; \textbf{skip}; y := 1; \textbf{while} x \ s \]

\[ \rightarrow \cdot, x \mapsto 3; y := 1; \textbf{while} x \ s \]

\[ \rightarrow^2 \cdot, x \mapsto 3, y \mapsto 1; \textbf{while} x \ s \]
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 \( s = (y := y * x; x := x - 1) \).

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

\[ \rightarrow \ \cdot, x \mapsto 3; \textbf{skip}; y := 1; \textbf{while} x \ s \]

\[ \rightarrow \ \cdot, x \mapsto 3; y := 1; \textbf{while} x \ s \]

\[ \rightarrow^2 \ \cdot, x \mapsto 3, y \mapsto 1; \textbf{while} x \ s \]

\[ \rightarrow \ \cdot, x \mapsto 3, y \mapsto 1; \textbf{if} x (s; \textbf{while} x \ s) \textbf{skip} \]
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 \( s = (y := y \ast x; x := x - 1) \).

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

\[ \rightarrow \cdot, x \mapsto 3; \textbf{skip}; y := 1; \textbf{while} \ x \ s \]

\[ \rightarrow \cdot, x \mapsto 3; y := 1; \textbf{while} \ x \ s \]

\[ \rightarrow^2 \cdot, x \mapsto 3, y \mapsto 1; \textbf{while} \ x \ s \]

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

\[ \rightarrow \cdot, x \mapsto 3, y \mapsto 1; y := y \ast x; x := x - 1; \textbf{while} \ x \ s \]
Continued...

\[ \rightarrow^2 \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; x := x-1; \textbf{while } x \ s \]
Continued...

\[\rightarrow^2 \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; x := x - 1; \textbf{while } x \ s\]

\[\rightarrow^2 \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2; \textbf{while } x \ s\]
Continued...

\[\vdash^2 \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; \ x := x-1; \ \textbf{while} \ x \ s \]

\[\vdash^2 \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2; \ \textbf{while} \ x \ s \]

\[\vdash \quad \ldots, y \mapsto 3, x \mapsto 2; \ \textbf{if} \ x \ (s; \ \textbf{while} \ x \ s) \ \textbf{skip}\]
Continued...

\[\rightarrow^2 \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; \ x := x - 1; \ \textbf{while} \ x \ s\]

\[\rightarrow^2 \quad \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2; \ \textbf{while} \ x \ s\]

\[\rightarrow \quad \ldots, y \mapsto 3, x \mapsto 2; \ \textbf{if} \ x \ (s; \ \textbf{while} \ x \ s) \ \textbf{skip}\]

\[\ldots\]
Continued...

\[\rightarrow^2 \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3; x := x-1; \textbf{while} x \ s\]

\[\rightarrow^2 \cdot, x \mapsto 3, y \mapsto 1, y \mapsto 3, x \mapsto 2; \textbf{while} x \ s\]

\[\rightarrow \ldots, y \mapsto 3, x \mapsto 2; \textbf{if} \ x (s; \textbf{while} x \ s) \textbf{skip}\]

\[\ldots\]

\[\rightarrow \ldots, y \mapsto 6, x \mapsto 0; \textbf{skip}\]
Where we are

Defined $H ; e \downarrow c$ and $H ; s \rightarrow H' ; s'$ and extended the latter to give $s$ a meaning

- The way we did expressions is “large-step operational semantics”
- The way we did statements is “small-step operational semantics”
- So now you have seen both

Definition by interpretation: program means what an interpreter (written in a metalanguage) says it means

- Interpreter represents a (very) abstract machine that runs code

Large-step does not distinguish errors and divergence

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

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

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

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

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

We can prove a program diverges, i.e., for all \( H \) and \( n \),
\[
\bullet; s \rightarrow^n H; \text{skip} \text{ cannot be derived}
\]

Example: \texttt{while 1 skip}
Establishing Properties

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

Example: Our last program terminates with \texttt{x} holding 0.

We can prove a program diverges, i.e., for all \( H \) and \( n \),
\[ \cdot ; s \rightarrow^n H ; \text{skip} \] cannot be derived.

Example: \texttt{while 1 skip}

By induction on \( n \), but requires a stronger induction hypothesis.
More General Proofs

We can prove properties of executing all programs (satisfying another property)

Example: If \( H \) and \( s \) have no negative constants and \( H ; s \rightarrow^* H' ; s' \), then \( H' \) and \( s' \) have no negative constants.

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