A Simple Imperative Language
Operational Semantics
(= “meaning”)

Homework #1 Out Today
• Due One Week From Now
• Take a look tonight
• My office hours are on Wednesday

Medium-Range Plan
• Study a simple imperative language IMP
  - Abstract syntax (today)
  - Operational semantics (today)
  - Denotational semantics
  - Axiomatic semantics
  - ... and relationships between various semantics (with proofs, peut-être)
  - Today: operational semantics
    • Follow along in Chapter 2 of Winskel

Syntax of IMP
• **Concrete syntax**: The rules by which programs can be expressed as strings of characters
  - Keywords, identifiers, statement separators vs. terminators (Niklaus!!), comments, indentation (Guido!!)
• Concrete syntax is important in practice
  - For readability (Larry!!), familiarity, parsing speed (Bjarne!!), effectiveness of error recovery, clarity of error messages (Robin!!)
• **Well-understood principles**
  - Use finite automata and context-free grammars
  - Automatic lexer/parser generators

(Note On Recent Research)
• If-as-and-when you find yourself making a new language, consider GLR (elkhound) instead of LALR(1) (bison)
• Scott McPeak, George G. Necula: *Elkhound: A Fast, Practical GLR Parser Generator*. CC 2004: pp. 73-88
• As fast as LALR(1), more natural, handles basically all of C++, etc.
Abstract Syntax

• We ignore parsing issues and study programs given as abstract syntax trees
  - I provide the parser in the homework ...
• Abstract syntax tree is (a subset of) the parse tree of the program
  - Ignores issues like comment conventions
  - More convenient for formal and algorithmic manipulation
  - All research papers use ASTs, etc.

IMP Abstract Syntactic Entities

• int integer constants (n ∈ ℤ)
• bool boolean constants (true, false)
• L locations of variables (x, y)
• Aexp arithmetic expressions (e)
• Bexp boolean expressions (b)
• Com commands (c)

- (these also encode the types)

Abstract Syntax (Aexp)

• Arithmetic expressions (Aexp)
  e ::= n for n ∈ ℤ
  | x for x ∈ L
  | e₁ + e₂ for e₁, e₂ ∈ Aexp
  | e₁ - e₂ for e₁, e₂ ∈ Aexp
  | e₁ * e₂ for e₁, e₂ ∈ Aexp

• Notes:
  - Variables are not declared
  - All variables have integer type
  - No side-effects (in expressions)

Abstract Syntax (Bexp)

• Boolean expressions (Bexp)
  b ::= true
  | false
  | e₁ = e₂ for e₁, e₂ ∈ Aexp
  | e₁ ≤ e₂ for e₁, e₂ ∈ Aexp
  | ¬¬ ¬¬ b for b ∈ Bexp
  | b₁ ∧∧ ∧∧ b₂ for b₁, b₂ ∈ Bexp
  | b₁ ∨∨ ∨∨ b₂ for b₁, b₂ ∈ Bexp

“Boolean”

• George Boole
  - 1815-1864
• I’ll assume you know boolean algebra ...

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p ∧ q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Abstract Syntax (Com)

• Commands (Com)
  c ::= skip
  | x := e x ∈ L ∧ e ∈ Aexp
  | c₁ ; c₂ c₁, c₂ ∈ Com
  | if b then c₁ else c₂ c₁, c₂ ∈ Com ∧ b ∈ Bexp
  | while b do c c₁, c₂ ∈ Com ∧ b ∈ Bexp

• Notes:
  - The typing rules are embedded in the syntax definition
  - Other parts are not context-free and need to be checked separately (e.g., all variables are declared)
  - Commands contain all the side-effects in the language
  - Missing: pointers, function calls, what else?
Popular Culture

“Ah. You seek meaning.”
“Yes.”
“Then listen to the music, not the song.”
-- Kosh and Talia, Deathwalker

“Angel... How did you get in here?”
“I was invited. The sign in front of the school... Formatia trans sicere educatorum.”
“Enter all ye who seek knowledge.”
“What can I say? I’m a knowledge seeker.”
-- Jenny Calendar and Angelus, Passion

Why Study Formal Semantics?

• Language design (denotational)
• Proofs of correctness (axiomatic)
• Language implementation (operational)
• Reasoning about programs
• Providing a clear behavioral specification
• “All the cool people are doing it.”
  - You need this to understand PL research
• “First one’s free.”

Consider This Java

```java
x = 0;
try {
x = 1;
break mygoto;
} finally {
x = 2;
raise NullPointerException;
}
x = 3;
mygoto:
x = 4;
```

• What happens when you execute this code?
• Notably, what assignments are executed?

14.20.2 Execution of try-catch-finally

• A try statement with a finally block is executed by first executing the try block. Then there is a choice:
  - If the finally block completes normally, then the try statement completes normally.
  - If the finally block completes abruptly for reason S, then the try statement completes abruptly for reason S.
• If execution of the try block completes abruptly because of a throw of a value V, then there is a choice:
  - If the run-time type of V is assignable to the parameter of any catch clause of the try statement, then the try statement completes abruptly for reason S.
  - If the throw of value V is caught by a finally block, then the try statement completes abruptly for reason S.

Can’t we just nail this somehow?

• Bonus points: what size shorts is this spectacular specimen sporting?

Ouch! Confusing.

• Wouldn’t it be nice if we had some way of describing what a language (feature or program) means ...
  - More precisely than English
  - More compactly than English
  - So that you might build a compiler
  - So that you might prove things about programs
Analysis of IMP

• Questions to answer:
  - What is the "meaning" of a given IMP expression/command?
  - How would we go about evaluating IMP expressions and commands?
  - How are the evaluator and the meaning related?

Three Canonical Approaches

• Operational
  - How would I execute this?
  - "Symbolic Execution"

• Axiomatic
  - What is true after I execute this?

• Denotational
  - What is this trying to compute?

An Operational Semantics

• Specifies how expressions and commands should be evaluated

• Depending on the form of the expression
  - 0, 1, 2, . . . don’t evaluate any further.
  - They are normal forms or values.
  - e1 + e2 is evaluated by first evaluating e1 to n1 , then evaluating e2 to n2 . (post-order traversal)
  - The result of the evaluation is the literal representing n1 + n2.
  - Similarly for e1 * e2

• Operational semantics abstracts the execution of a concrete interpreter
  - Important keywords are colored & underlined in this class.

Semantics of IMP

• The meanings of IMP expressions depend on the values of variables
  - What does "x*5" mean? It depends on "x"!

• The value of variables at a given moment is abstracted as a function from L to \( \mathbb{Z} \) (a state)
  - If \( x \mapsto 8 \) in our state, we expect "x*5" to mean 13

• The set of all states is \( \Sigma = L \rightarrow \mathbb{Z} \)

• We shall use \( \sigma \) to range over \( \Sigma \)
  - \( \sigma \), a state, maps variables to values

Notation: Judgment

• We write:
  \[ \langle e, \sigma \rangle \Downarrow n \]

• To mean that \( e \) evaluates to \( n \) in state \( \sigma \).

• This is a judgment. It asserts a relation between \( e, \sigma \) and \( n \).

• In this case we can view \( \Downarrow \) as a function with two arguments (\( e \) and \( \sigma \)).

Operational Semantics

• This formulation is called natural operational semantics
  - or big-step operational semantics
  - the \( \Downarrow \) judgment relates the expression and its "meaning"

• How should we define
  \[ \langle e_1 + e_2, \sigma \rangle \Downarrow ... ? \]
### Notation: Rules of Inference

- We express the evaluation rules as **rules of inference** for our judgment
  - called the **derivation rules** for the judgment
  - also called the **evaluation rules** (for operational semantics)
- In general, we have one rule for each language construct:

  \[
  \langle e_1, \sigma \rangle \Downarrow n_1, \langle e_2, \sigma \rangle \Downarrow n_2, \langle e_1 + e_2, \sigma \rangle \Downarrow n_1 + n_2
  \]

  This is the only rule for \( e_1 + e_2 \).

### Rules of Inference

<table>
<thead>
<tr>
<th>Hypothesis (_1), \ldots, Hypothesis (_n)</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash b : \text{bool} )</td>
<td>( \Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \tau )</td>
</tr>
</tbody>
</table>

- For any given proof system, a finite number of rules of inference (or schema) are listed somewhere
- Rule instances should be easily checked
- What is the definition of “NP”?

### Derivation

- Tree-structured (conclusion at bottom)
- May include multiple sorts of rules-of-inference
- Could be constructed, typically are not
- Typically verified in polynomial time

### Evaluation Rules (for Aexp)

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle n, \sigma \rangle \Downarrow n )</td>
<td>( \langle x, \sigma \rangle \Downarrow \sigma(x) )</td>
</tr>
</tbody>
</table>

- This is called **structural operational semantics**
  - rules defined based on the structure of the expression
- These rules do not impose an order of evaluation!

### Evaluation Rules (for Bexp)

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \text{true}, \sigma \rangle \Downarrow \text{true} )</td>
<td>( \langle e_1, \sigma \rangle \Downarrow n_1, \langle e_2, \sigma \rangle \Downarrow n_2 )</td>
</tr>
<tr>
<td>( \langle \text{false}, \sigma \rangle \Downarrow \text{false} )</td>
<td>( \langle e_1 \leq e_2, \sigma \rangle \Downarrow n_1 \leq n_2 )</td>
</tr>
</tbody>
</table>

### How to Read the Rules?

- **Forward (top-down)** = inference rules
  - if we know that the hypothesis judgments hold then we can infer that the conclusion judgment also holds

- If we know that \( \langle e_1, \sigma \rangle \Downarrow 5 \) and \( \langle e_2, \sigma \rangle \Downarrow 7 \), then we can infer that \( \langle e_1 + e_2, \sigma \rangle \Downarrow 12 \)
How to Read the Rules?

- **Backward (bottom-up)** = evaluation rules
  - Suppose we want to evaluate \( e_1 + e_2 \), i.e., find \( n \) s.t. \( e_1 + e_2 \uparrow n \) is derivable using the previous rules
  - By inspection of the rules we notice that the last step in the derivation of \( e_1 + e_2 \uparrow n \) must be the addition rule
    - the other rules have conclusions that would not match \( e_1 + e_2 \uparrow n \)
    - this is called reasoning by **inversion** on the derivation rules

Evaluation By Inversion

- Thus we must find \( n_1 \) and \( n_2 \) such that \( e_1 \uparrow n_1 \) and \( e_2 \uparrow n_2 \) are derivable
  - This is done recursively
- If there is exactly one rule for each kind of expression we say that the rules are **syntax-directed**
  - At each step at most one rule applies
  - This allows a simple evaluation procedure as above (recursive tree-walk)
    - True for our Aexp but not Bexp. *Why?*

Evaluation of Commands

- The evaluation of a **Com** may have side effects but has **no direct result**
  - What is the result of evaluating a command?
- The "result" of a **Com** is a **new state**:

  \[
  \langle c, \sigma \rangle \downarrow \sigma'
  \]

  - But the evaluation of **Com** might not terminate! **Danger Will Robinson! (huh?)**

Com Evaluation Rules

**Com Evaluation Rules 1**

\[
\begin{align*}
\langle \text{skip}, \sigma \rangle & \downarrow \sigma \\
\langle e_1, \sigma \rangle & \downarrow \sigma' \\
\langle e_2, \sigma \rangle & \downarrow \sigma'' \\
\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle & \downarrow \sigma' \\
\langle b, \sigma \rangle & \downarrow \text{true} \quad \langle c_1, \sigma \rangle \downarrow \sigma' \\
\langle b, \sigma \rangle & \downarrow \text{false} \quad \langle c_2, \sigma \rangle \downarrow \sigma'
\end{align*}
\]

**Com Evaluation Rules 2**

\[
\begin{align*}
\langle x := e, \sigma \rangle & \downarrow \sigma[x := n] \\
\langle x := e, \sigma \rangle & \downarrow \sigma[x := n] \quad \text{Def: } \sigma[x := n](x) = n \\
\sigma[x := n](y) & = \sigma(y)
\end{align*}
\]

- Let’s do **while** together

**Com Evaluation Rules 3**

\[
\begin{align*}
\langle e, \sigma \rangle & \downarrow n \\
\langle x := e, \sigma \rangle & \downarrow \sigma[x := n] \\
\text{Def: } & \sigma[x := n](x) = n \\
\sigma[x := n](y) & = \sigma(y)
\end{align*}
\]

\[
\begin{align*}
\langle b, \sigma \rangle & \downarrow \text{false} \\
\langle b, \sigma \rangle & \downarrow \text{true} \quad \langle c; \text{while } b \text{ do } c, \sigma \rangle \downarrow \sigma' \\
\langle b, \sigma \rangle & \downarrow \text{false} \quad \langle \text{while } b \text{ do } c, \sigma \rangle \downarrow \sigma'
\end{align*}
\]
 Homework

- Homework 1 Out Today
  - Actually out last week ...
  - Due in One Week
- Read at least 1 of these 3 Articles
  - 1. Wegner’s Programming Languages - The First 25 years
  - 2. Wirth’s On the Design of Programming Languages
  - 3. Nauer’s Report on the algorithmic language ALGOL 60
- Skim the optional reading - we’ll discuss opsem “in the wild” next time