In Our Last Exciting Episode

I can’t wait to see what’s up with this new technology announcement.

Will we see new scenarios? New devices? Ground-breaking user interfaces?

Welcome.

I’d like to begin by showing this block diagram of our proposed architectural framework.

He’s already lost me. Seven seconds. That’s a new record.

So who’s up for ice cream?

Usability time. So to use the feature, where would you click?

Mmmm... no... colder... colder... warmer...

Warmer... warmer... hot! Hot! Smokin’ hot! There it is!

I think we’re leading the witness a bit. Would you say this is the best feature ever?
Lessons From Model Checking

- To find **bugs**, we need **specifications**
  - What are some good specifications?
- To **convert** a program into a **model**, we need **predicates**/invariants and a **theorem prover**.
  - Which are the important predicates? Invariants?
  - What should we track when reasoning about a program and what should we abstract?
  - How does a theorem prover work?
- **Simple** algorithms (e.g., depth first search, pushing facts along a CFG) can work well
  - … under what circumstances?
The Big Lesson

• To reason about a program (= “is it doing the right thing? the wrong thing?”) we must understand what the program means!
A Simple Imperative Language Operational Semantics (= “meaning”)

I'm hungry. Can I have a snack?
Sure. Help yourself.
You can have an apple or an orange from the fridge.
Even though we're both talking English, we're not speaking the same language.
Homework #0 Due Today

- Can't get BLAST to work?
  - ssh to power1.cs.virginia.edu
  - Plus the BLAST linux binaries
  - cp all of them (e.g., csi*, pblast*, ...) to ~/bin
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 Post-LALR Advances)

• 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 ...

• An 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 ($n \in \mathbb{Z}$)
- **bool** bool 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)

\[ e ::= n \quad \text{for } n \in \mathbb{Z} \]
\[ | x \quad \text{for } x \in L \]
\[ | e_1 + e_2 \quad \text{for } e_1, e_2 \in Aexp \]
\[ | e_1 - e_2 \quad \text{for } e_1, e_2 \in Aexp \]
\[ | e_1 \times e_2 \quad \text{for } e_1, e_2 \in Aexp \]

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

- Boolean expressions (Bexp)

\[ b ::= \text{true} \]

\[ | \ \text{false} \]

\[ | e_1 = e_2 \quad \text{for } e_1, e_2 \in \text{Aexp} \]

\[ | e_1 \leq e_2 \quad \text{for } e_1, e_2 \in \text{Aexp} \]

\[ | \neg b \quad \text{for } b \in \text{Bexp} \]

\[ | b_1 \land b_2 \quad \text{for } b_1, b_2 \in \text{Bexp} \]

\[ | b_1 \lor b_2 \quad \text{for } b_1, b_2 \in \text{Bexp} \]
“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)**
  
  \[
  c ::= \text{skip} \\
  \mid x := e \quad x \in L \land e \in Aexp \\
  \mid c_1 ; c_2 \quad c_1, c_2 \in \text{Com} \\
  \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \quad c_1, c_2 \in \text{Com} \land b \in Bexp \\
  \mid \text{while } b \text{ do } c \quad c \in \text{Com} \land b \in Bexp
  \]

- **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?
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 Legal Java

```java
x = 0;
try {
    x = 1;
    break mygoto;
} finally {
    x = 2;
    raise NullPointerException;
}
mygoto:
```
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 execution of the try block completes normally, then the finally block is executed, and 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 first (leftmost) such catch clause is selected. The value $V$ is assigned to the parameter of the selected catch clause, and the Block of that catch clause is executed. Then there is a choice:
    - If the catch block completes normally, then the finally block is executed. Then there is a choice:
      - If the finally block completes normally, then the try statement completes normally.
      - If the finally block completes abruptly for any reason, then the try statement completes abruptly for the same reason.
    - If the catch block completes abruptly for reason $R$, then the finally block is executed. Then there is a choice:
      - If the finally block completes normally, then the try statement completes abruptly for reason $R$.
      - If the finally block completes abruptly for reason $S$, then the try statement completes abruptly for reason $S$ (and reason $R$ is discarded).
  - If the run-time type of $V$ is not assignable to the parameter of any catch clause of the try statement, then the finally block is executed. Then there is a choice:
    - If the finally block completes normally, then the try statement completes abruptly because of a throw of the value $V$.
    - If the finally block completes abruptly for reason $S$, then the try statement completes abruptly for reason $S$ (and the throw of value $V$ is discarded and forgotten).
- If execution of the try block completes abruptly for any other reason $R$, then the finally block is executed. Then there is a choice:
  - If the finally block completes normally, then the try statement completes abruptly for reason $R$.
  - If the finally block completes abruptly for reason $S$, then the try statement completes abruptly for reason $S$ (and reason $R$ is discarded).
Can’t we just nail this somehow?
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?

• Axiomatic
  - What is true after I execute this?
  - Symbolic Execution

• 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.
  - $e_1 + e_2$ is evaluated by first evaluating $e_1$ to $n_1$, then evaluating $e_2$ to $n_2$. (post-order traversal)
    • The result of the evaluation is the literal representing $n_1 + n_2$.
  - Similarly for $e_1 * e_2$

• 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 \( x = 8 \) 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
Program State

- The **state** $\sigma$ is somewhat like “**memory**”
  - It holds the current values of all variables
  - Formally, $\sigma : L \rightarrow \mathbb{Z}$
Q: Cartoons (682 / 842)

• Why is Gargamel trying to capture the Smurfs?
Q: Computer Science

- This American Turing Award winner is notable for his work in the theory of algorithms, a max-flow solver, a bipartite graph matcher, a string search algorithm, and “Reducibility Among Combinatorial Problems” in which he proved 21 problems to be NP-complete. He introduced the standard methodology for proving problems to be NP-complete.
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

\[
\langle e_1 + e_2, \sigma \rangle \downarrow \ldots ?
\]
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**:

\[
\begin{align*}
\langle e_1, \sigma \rangle &\downarrow n_1 \\
\langle e_2, \sigma \rangle &\downarrow n_2 \\
\langle e_1 + e_2, \sigma \rangle &\downarrow n_1 + n_2
\end{align*}
\]

This is the only rule for \( e_1 + e_2 \)
Rules of Inference

\[ \text{Hypothesis}_1 \ldots \text{Hypothesis}_N \]

Conclusion

\[
\Gamma \vdash b : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \tau
\]

- 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>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;n, \sigma&gt; \Downarrow n$</td>
<td>Evaluation of a number.</td>
</tr>
<tr>
<td>$&lt;e_1, \sigma&gt; \Downarrow n_1$</td>
<td>Evaluation of an expression $e_1$.</td>
</tr>
<tr>
<td>$&lt;e_2, \sigma&gt; \Downarrow n_2$</td>
<td>Evaluation of an expression $e_2$.</td>
</tr>
<tr>
<td>$&lt;e_1 + e_2, \sigma&gt; \Downarrow n_1 + n_2$</td>
<td>Evaluation of the sum of two expressions.</td>
</tr>
<tr>
<td>$&lt;e_1 - e_2, \sigma&gt; \Downarrow n_1 - n_2$</td>
<td>Evaluation of the difference of two expressions.</td>
</tr>
<tr>
<td>$&lt;e_1 * e_2, \sigma&gt; \Downarrow n_1 * n_2$</td>
<td>Evaluation of the product of two expressions.</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)

\[<true, \sigma> \Downarrow true\]
\[<false, \sigma> \Downarrow false\]

\[<e_1, \sigma> \Downarrow n_1 \quad <e_2, \sigma> \Downarrow n_2\]
\[<e_1 \leq e_2, \sigma> \Downarrow n_1 \leq n_2\]

\[<e_1 = e_2, \sigma> \Downarrow n_1 = n_2\]

\[<b_1, \sigma> \Downarrow false\]
\[<b_1 \land b_2, \sigma> \Downarrow false\]

\[<b_2, \sigma> \Downarrow false\]
\[<b_1 \land b_2, \sigma> \Downarrow false\]

\[<b_1, \sigma> \Downarrow true\]
\[<b_2, \sigma> \Downarrow true\]

(show: candidate \lor rule) \[<b_1 \land b_2, \sigma> \Downarrow true\]
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
    
    \(<e_1, \sigma> \downarrow 5 \text{ and } <e_2, \sigma> \downarrow 7\), then we can infer that
    
    \(<e_1 + e_2, \sigma> \downarrow 12\)
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. $e_1 + e_2 \Downarrow n$ is derivable using the previous rules
  - By inspection of the rules we notice that the last step in the derivation of $e_1 + e_2 \Downarrow n$ must be the addition rule
    • the other rules have conclusions that would not match $e_1 + e_2 \Downarrow n$
    • this is called reasoning by inversion on the derivation rules
Evaluation By Inversion

- Thus we must find $n_1$ and $n_2$ such that $e_1 \downarrow n_1$ and $e_2 \downarrow n_2$ 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 1

\[ \begin{array}{l}
\text{<skip, } \sigma \text{> } \Downarrow \sigma \\
\text{<c}_1\text{, } \sigma \text{> } \Downarrow \sigma' \quad \text{<c}_2\text{, } \sigma' \Downarrow \sigma'' \\
\text{<c}_1 ; \text{c}_2\text{, } \sigma \Downarrow \sigma'' \\
\text{<b, } \sigma \Downarrow \text{true} \quad \text{<c}_1\text{, } \sigma \Downarrow \sigma' \\
\text{<if b then c}_1\text{ else c}_2\text{, } \sigma \Downarrow \sigma' \\
\text{<b, } \sigma \Downarrow \text{false} \quad \text{<c}_2\text{, } \sigma \Downarrow \sigma' \\
\text{<if b then c}_1\text{ else c}_2\text{, } \sigma \Downarrow \sigma' 
\end{array} \]
Com Evaluation Rules 2

\[ \langle e, \sigma \rangle \downarrow n \]
\[ \langle x := e, \sigma \rangle \downarrow \sigma \left[ x := n \right] \]

Def: \( \sigma \left[ x := n \right](x) = n \)
\( \sigma \left[ x := n \right](y) = \sigma(y) \)

• Let’s do **while** together
Com Evaluation Rules 3

\[
\begin{align*}
\langle e, \sigma \rangle & \Downarrow n \\
\langle x := e, \sigma \rangle & \Downarrow \sigma[x := n] \\
\langle b, \sigma \rangle & \Downarrow \text{false} \\
\langle \text{while } b \text{ do } c, \sigma \rangle & \Downarrow \sigma \\
\langle b, \sigma \rangle & \Downarrow \text{true} \\
\langle c; \text{ while } b \text{ do } c, \sigma \rangle & \Downarrow \sigma' \\
\langle \text{while } b \text{ do } c, \sigma \rangle & \Downarrow \sigma'
\end{align*}
\]

Def: \[\sigma[x := n](x) = n, \quad \sigma[x := n](y) = \sigma(y)\]
Homework

- Homework 0 Due Today
- Homework 1 Due In One Week
- Reading!
  - If this wasn't intuitive, try some of the optional readings for more context.