CMSC 430
Introduction to Compilers
Fall 2018

Operational Semantics
Syntax vs. semantics

• Syntax = grammatical structure
• Semantics = underlying meaning

• Sentences in a language can be syntactically well-formed but semantically meaningless
  - if ("foo" > 37) { oogbooga(3); "baz" * "qux"; }

• ocamllex and ocamlyacc enforce syntax
  - (Though could play tricks in actions to check semantics)
Syntax vs. semantics (cont’d)

• General principle: enforce correctness at the earliest stage possible
  ▪ Keywords identified in lexer
  ▪ Balanced ()’s enforced in parser
  ▪ Types enforced afterward

• Why?
  ▪ Earlier in pipeline ⇒ simpler to think about
  ▪ Reporting errors is easier
    - Less transformation from original program
    - Errors may be easier to localize
  ▪ Faster algorithms for detecting violations
    - Higher chance could employ them interactively in IDE
Detour: Natural deduction

- We are going to use *natural deduction* rules to describe semantics
  - So we need to understand how those work first

- Natural deduction rules provide a syntax for writing down proofs
  - Each rule is essentially an axiom
  - Rules are composed together
    - The result is called a *derivation*
  - The things rules prove are called *judgments*
H1 H2 ... Hn  name

C

- H1 ... Hn are hypotheses, C is the conclusion
- “If H1 and H2 and ... and Hn hold, then C holds”
IMP: A language of commands

\[
a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1
\]
\[
b ::= b_0 \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \land b_1 \mid b_0 \lor b_1
\]
\[
c ::= \text{skip} \mid X := a \mid c_0 ; c_1 \mid \text{if } b \text{ then } c_0 \text{ else } c_1 \mid \text{while } b \text{ do } c
\]

- \( n \in \mathbb{N} = \text{integers}, \ X \in \text{Var} = \text{variables}, \ b_0 \in \text{Bool} = \{\text{true, false}\} \)
- This is a typical way of presenting a language
  - Notice grammar is for ASTs
    - Not concerned about issues like ambiguity, associativity, precedence
- Syntax stratified into commands (c) and expressions (a,b)
  - Expressions have no side effects
- No function calls (and no higher order functions)
- So: How do we specify the semantics of IMP?
Program state

• IMP contains imperative updates, so we need to model the program state
  ▪ Here the state is simply the integer value of each variable
  ▪ (Notice can’t assign a boolean to a variable, by syntax!)

• State:
  ▪ $\sigma : \text{Var} \rightarrow \mathbb{N}$
  ▪ A state $\sigma$ is a mapping from variables to their values
Judgments

• Operational semantics has three kinds of judgments
  ▪ \( \langle a, \sigma \rangle \rightarrow n \)
    - In state \( \sigma \), arithmetic expression \( a \) evaluates to \( n \)
  ▪ \( \langle b, \sigma \rangle \rightarrow \text{bv} \)
    - In state \( \sigma \), boolean expression \( b \) evaluates to true or false
  ▪ \( \langle c, \sigma \rangle \rightarrow \sigma' \)
    - Running command \( c \) in state \( \sigma \) produces state \( \sigma' \)

• Can immediately see only commands have side effects
  ▪ Only form whose evaluation produces a new state
  ▪ Commands also do not return values
  ▪ Note this is math, so we express state changes by creating the new state \( \sigma' \). We can’t just “mutate” \( \sigma \).
Arithmetic evaluation

\[
\langle n, \sigma \rangle \rightarrow n
\]
\[
\langle X, \sigma \rangle \rightarrow \sigma(X)
\]

\[
\langle a_0, \sigma \rangle \rightarrow n_0
\]
\[
\langle a_1, \sigma \rangle \rightarrow n_1
\]

\[
\langle a_0 + a_1, \sigma \rangle \rightarrow n_0 + n_1
\]

\[
\langle a_0 - a_1, \sigma \rangle \rightarrow n_0 - n_1
\]

\[
\langle a_0, \sigma \rangle \rightarrow n_0
\]
\[
\langle a_1, \sigma \rangle \rightarrow n_1
\]

\[
\langle a_0 \times a_1, \sigma \rangle \rightarrow n_0 \times n_1
\]
• Notes:
  
  ▪ Rule for variables only defined if \( X \) is in \( \text{dom}(\sigma) \). Otherwise the program goes wrong, i.e., it has no meaning.
  
  ▪ Hypotheses of last three rules stacked to save space.
  
  ▪ Notice difference between syntactic operators, on the left side of arrows, and mathematical operators, on the right side of arrows.
  
  ▪ One rule for each kind of expression
    - These are syntax-directed rules.
  
  ▪ In the rules, we use terminals and non-terminals in the grammar to stand for anything producible from them.
    - E.g., \( n \) stands for any integer; \( \sigma \) for any state; etc.
  
  ▪ Order of evaluation irrelevant, because there are no side effects.
Sample derivation

- $1+2+3$
- $(2\times x)-4$ in $\sigma = [x \mapsto 3]$
Correspondence to OCaml

(* a ::= n | X | a0+a1 | a0-a1 | a0×a1 *)

```ocaml
type aexpr =
| AInt of int
| AVar of string
| APlus of aexpr * aexpr
| AMinus of aexpr * aexpr
| ATimes of aexpr * aexpr

let rec aeval sigma = function
| AInt n -> n
| AVar n -> List.assoc n sigma
| APlus (a1, a2) -> (aeval sigma a1) + (aeval sigma a2)
| AMinus (a1, a2) -> (aeval sigma a1) - (aeval sigma a2)
| ATimes (a1, a2) -> (aeval sigma a1) * (aeval sigma a2)
```
## Boolean evaluation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \text{true}, \sigma \rangle )</td>
<td>true</td>
</tr>
<tr>
<td>( \langle \text{false}, \sigma \rangle )</td>
<td>false</td>
</tr>
<tr>
<td>( \langle \neg b, \sigma \rangle )</td>
<td>\neg \text{bv}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle a_0, \sigma \rangle )</td>
<td>(n_0)</td>
</tr>
<tr>
<td>( \langle a_1, \sigma \rangle )</td>
<td>(n_1)</td>
</tr>
<tr>
<td>( \langle a_0 = a_1, \sigma \rangle )</td>
<td>(n_0 = n_1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle b_0, \sigma \rangle )</td>
<td>(\text{bv}_0)</td>
</tr>
<tr>
<td>( \langle b_1, \sigma \rangle )</td>
<td>(\text{bv}_1)</td>
</tr>
<tr>
<td>( \langle b_0 \land b_1, \sigma \rangle )</td>
<td>(\text{bv}_0 \land \text{bv}_1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle b_0 \lor b_1, \sigma \rangle )</td>
<td>(\text{bv}_0 \lor \text{bv}_1)</td>
</tr>
</tbody>
</table>
Sample derivations

• \neg false \land true

• 2 \leq X \lor X \leq 4 \text{ in } \sigma = [X \mapsto 3]
Correspondence to OCaml

(* b ::= bv | a0=a1 | a0≤a1 | ¬b | b0∧b1 | b0∨b1 *)

type bexpr =
| BV of bool
| BEq of aexpr * aexpr
| BLeq of aexpr * aexpr
| BNot of bexpr
| BAnd of bexpr * bexpr
| BOr of bexpr * bexpr

let rec beval sigma = function
| BV b -> b
| BEq (a1, a2) -> (aeval sigma a1) = (aeval sigma a2)
| BLeq (a1, a2) -> (aeval sigma a1) <= (aeval sigma a2)
| BNot b -> not (beval sigma b)
| BAnd (b1, b2) -> (beval sigma b1) && (beval sigma b2)
| BOr (b1, b2) -> (beval sigma b1) || (beval sigma b2)
Command evaluation

\[
\begin{align*}
\langle \text{skip}, \sigma \rangle & \rightarrow \sigma \\
\langle a, \sigma \rangle & \rightarrow \text{a} \\
\langle X:=a, \sigma \rangle & \rightarrow \sigma[X\mapsto\text{a}]
\end{align*}
\]

\[
\begin{align*}
\langle \text{c0}, \sigma \rangle & \rightarrow \sigma_0 \\
\langle \text{c1}, \sigma_0 \rangle & \rightarrow \sigma_1 \\
\langle \text{c0}; \text{c1}, \sigma \rangle & \rightarrow \sigma_1
\end{align*}
\]

• Here \(\sigma[X\mapsto\text{a}]\) is the state that is the same as \(\sigma\), except \(X\) now maps to \(\text{a}\)
  - \((\sigma[X\mapsto\text{a}])(X) = \text{a}\)
  - \((\sigma[X\mapsto\text{a}])(Y) = \sigma(Y)\)    \(X \neq Y\)

• Notice order of evaluation explicit in sequence rule
Command evaluation (cont’d)

• Two rules for conditional
  - Just like in logic we needed two rules for $\land$-E and $\lor$-I
  - Notice we specify only one command is executed
Command evaluation (cont’d)

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

- n:=3; f:=1; while n ≥ 1 do f := f * n; n := n - 1
Correspondence to OCaml

(* c ::= skip | X:=a | c0;c1 | if b then c0 else c1 | 
    while b do c *)

type cmd =
| CSkip
| CAssn of string * aexpr
| CSeq of cmd * cmd
| CIf of bexpr * cmd * cmd
| CWhile of bexpr * cmd

let rec ceval sigma = function
| CSkip -> sigma
| CAssn (x, a) -> (x:(aeval sigma a))::sigma
    (* note List.assoc in aeval stops at first match *)
| CSeq (c0, c1) ->
    let sigma0 = ceval sigma c0 in ceval sigma0 c1
    (* or “ceval (ceval sigma c0) c1” *)
| CIf (b, c0, c1) ->
    if (beval sigma b) then (ceval sigma c0)
    else (ceval sigma c1)
| CWhile (b, c) ->
    if (beval sigma b)
    then ceval sigma (CSeq (c, CWhile(b,c)))
    else sigma
Big-step semantics

- Semantics given are “big step” or “natural semantics”
  - E.g., \( \langle c, \sigma \rangle \rightarrow \sigma' \)
  - Commands fully evaluated to produce the final output state, in one, big step
- Limitation: Can’t give semantics to non-terminating programs
  - We would need to work with infinite derivations, which is typically not valid
  - (Note: It is possible, though, using a co-inductive interpretation)
Small-step semantics

• Instead, can expose intermediate steps of computation
  - $a \rightarrow_{\sigma} a'$
    - Evaluating $a$ one step in state $\sigma$ produces $a'$
  - $b \rightarrow_{\sigma} b'$
    - Evaluating $b$ one step in state $\sigma$ produces $b'$
  - $\langle c, \sigma \rangle \rightarrow_{1} \langle c', \sigma' \rangle$
    - Running command $c$ in state $\sigma$ for one step yields a new command $c'$ and new state $\sigma'$

• Note putting $\sigma$ on the arrow is just a convenience
  - Good notation for stringing evaluations together
    - $a_0 \rightarrow_{\sigma} a_1 \rightarrow_{\sigma} a_2 \rightarrow_{\sigma} \ldots$
  - Put 1 on arrow for commands just to let us distinguish different kinds of arrows
Small-step rules for arithmetic

\[ X \rightarrow_{\sigma} \sigma(X) \]

\[
\begin{align*}
a0 & \rightarrow_{\sigma} a0' \\
a0+a1 & \rightarrow_{\sigma} a0'+a1
\end{align*}
\]

\[
\begin{align*}
a1 & \rightarrow_{\sigma} a1' \\
n+a1 & \rightarrow_{\sigma} n+a1'
\end{align*}
\]

\[
\begin{align*}
p=m+n & \rightarrow_{\sigma} p \\
n+m & \rightarrow_{\sigma} p
\end{align*}
\]

- Similarly for \(-\) and \(\times\)
- Notice no rule for evaluating integer \(n\)
  - An integer is in *normal form*, meaning no further evaluation is possible
- We’ve fixed the order of evaluation
  - Could also have made it non-deterministic
Context rules

- We have some rules that do the “real” work
  - The rest are context rules that define order of evaluation

- Cool trick (due to Hieb and Felleisen):
  - Define a context as a term with a “hole” in it
    - \[ C ::= □ \mid C + a \mid n + C \mid C - a \mid n - C \mid C \times a \mid n \times C \]
  - Notice the terms generated by this grammar always have exactly one □, and it always appears at the next position that can be evaluated
  - Define \( C[a] \) to be \( C \) where □ is replaced by \( a \)
    - Ex: \( ((□+3) \times 5)[4] = (4+3) \times 5 \)
  - Now add one, single context rule:

\[
\begin{align*}
  a & \rightarrow_σ a' \\
  C[a] & \rightarrow_σ C[a']
\end{align*}
\]
Small-step rules for booleans

• Very similar to arithmetic expressions
  ▪ Too boring to write them all down...
Small-step rules for commands

• Let’s define contexts, to get that out of the way
  - $C ::= □ | X:=C | C;c1 | \text{if } C \text{ then } c0 \text{ else } c1$

• Now the rules (plus the context rule): 

<table>
<thead>
<tr>
<th>Command Expression</th>
<th>Reduces to</th>
<th>Next State Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle X:=n, \sigma \rangle$</td>
<td>$\rightarrow_1$</td>
<td>$\langle \text{skip}, \sigma[x\mapsto n] \rangle$</td>
</tr>
<tr>
<td>$\langle \text{skip}; c1, \sigma \rangle$</td>
<td>$\rightarrow_1$</td>
<td>$\langle c1, \sigma \rangle$</td>
</tr>
<tr>
<td>$\langle \text{if true then } c0 \text{ else } c1, \sigma \rangle$</td>
<td>$\rightarrow_1$</td>
<td>$\langle c0, \sigma \rangle$</td>
</tr>
<tr>
<td>$\langle \text{if false then } c0 \text{ else } c1, \sigma \rangle$</td>
<td>$\rightarrow_1$</td>
<td>$\langle c1, \sigma \rangle$</td>
</tr>
<tr>
<td>$\langle \text{while } b \text{ do } c, \sigma \rangle$</td>
<td>$\rightarrow_1$</td>
<td>$\langle \text{if } b \text{ then } (c; \text{ while } b \text{ do } c) \text{ else } \text{skip}, \sigma \rangle$</td>
</tr>
</tbody>
</table>
Lambda calculus

• $e ::= x \mid \lambda x.e \mid e\ e$

• Recall
  - Scope of $\lambda$ extends as far to the right as possible
    - $\lambda x.\lambda y.x\ y$ is $\lambda x.(\lambda y.(x\ y))$
  - Function application is left-associative
    - $x\ y\ z$ is $(x\ y)\ z$
  - Beta-reduction takes a single step of evaluation
    - $(\lambda x.e1)\ e2 \rightarrow e1[e2/x]$
A nonderministic semantics

- Why are these semantics non-deterministic?
...with context rules

- \( C ::= □ | \lambda x.C | C e | e C \)

\[
\frac{e \rightarrow e'}{C[e] \rightarrow C[e']} \\
(\lambda x.e1) e2 \rightarrow e1[e2/x]
\]
The Church-Rosser Theorem

• If $a \rightarrow^* b$ and $a \rightarrow^* c$, there there exists $d$ such that $b \rightarrow^* d$ and $c \rightarrow^* d$

• Church-Rosser is also called *confluence*
Normal Form

• A term is in *normal form* if it cannot be reduced
  - Examples: $\lambda x.x$, $\lambda x.\lambda y.z$

• By Church-Rosser Theorem, every term reduces to at most one normal form
  - Warning: All of this applies only to the pure lambda calculus with non-deterministic evaluation

• Notice that for our application rule, the argument need not be in normal form
Not Every Term Has a Normal Form

• Consider
  - \( \Delta = \lambda x. x \ x \)
  - Then \( \Delta \ \Delta \ \rightarrow \ \Delta \ \Delta \ \rightarrow \ldots \)

• In general, self application leads to loops
  - ...which is where the \( Y \) combinator comes from (see 330)
Lazy vs. Eager Evaluation

• Our non-deterministic reduction rule is fine in theory, but awkward to implement

• Two deterministic strategies:
  - **Lazy**: Given $(\lambda x. e_1) e_2$, do not evaluate $e_2$ if $e_1$ does not “need” $x$
    - Also called left-most, call-by-name (c.b.n.), call-by-need, applicative, normal-order (with slightly different meanings)
  - **Eager**: Given $(\lambda x. e_1) e_2$, always evaluate $e_2$ fully before applying the function
    - Also called call-by-value (c.b.v.)
C.b.n. small-step semantics

- $e ::= x \mid \lambda x.e \mid e \ e$

\[
\begin{align*}
(\lambda x.e_1) \ e_2 & \rightarrow e_1[e_2/x] \\
\end{align*}
\]

- $e_1 \rightarrow e_1'$
- $e_1 \ e_2 \rightarrow e_1' \ e_2$

- Must evaluate function position until we get to a lambda
- Apply as soon as we know what fn we’re applying
- Do not evaluate “under” and lambda
- Do not evaluate the argument

- In context form:
  - $C ::= □ \mid C \ e$
**C.b.v. small-step semantics**

- \( e ::= x \mid \nu \mid e \ e \)
- \( \nu ::= \lambda x.e \)

\[
(\lambda x.e) \ \nu \rightarrow e[\nu\backslash x]
\]

- Must evaluate function position until we get to a lambda
- Evaluate function posn *before* argument posn
  - Not important here, but matters if we add side effects
- Do not evaluate “under” and lambda
- Argument must be fully evaluated before the call

- In context form:
  - \( C ::= \Box \mid C \ e \mid \nu \ C \)
C.b.n. versus c.b.v. in theory

• Call-by-name is *normalizing*
  - If $a$ is closed and there is a normal form $b$ such that $a \rightarrow^* b$ under the non-deterministic semantics, then $a \rightarrow^* d$ for some $d$ under c.b.n. semantics

• Call-by-value is not!
  - There are some programs that terminate under call-by-name but not under call-by-value
    - E.g., $(\lambda x.(\lambda y)y)(\Delta \Delta)$
      - Where $\Delta = \lambda x.x\ x$
      - The non-terminating argument $(\Delta \Delta)$ is discarded under c.b.n., but c.b.v. attempts to evaluate it
C.b.n. vs. c.b.v. in practice

- Lazy evaluation (call by name, call by need)
  - Has some nice theoretical properties
  - Terminates more often
  - Lets you play some tricks with “infinite” objects
  - Main example: Haskell

- Eager evaluation (call by value)
  - Is generally easier to implement efficiently
  - Blends more easily with side effects
  - Main examples: Most languages (C, Java, ML, etc.)