CMSC 430
Introduction to Compilers
Fall 2015

Operational Semantics

Syntax vs. semantics

- Syntax = grammatical structure
- Semantics = underlying meaning

- Sentences in a language can be syntactically well-formed but semantically meaningless
  - *“Colorless green ideals sleep furiously.”* — Syntactic Structures, Noam Chomsky, 1957.
  - 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*

Structure of a rule

$$\frac{H1\ H2\ \ldots\ Hn}{C}\ \text{name}$$

- H1 ... Hn are *hypotheses*, C is the *conclusion*
- "If H1 and H2 and ... and Hn hold, then C holds"

Example: Logic

$$\frac{A \qquad B}{A \wedge B}\ \wedge\text{-I} \qquad \frac{A \wedge B}{A}\ \wedge\text{-E\_L} \qquad \frac{A \wedge B}{B}\ \wedge\text{-E\_R}$$

$$\frac{A}{A \vee B}\ \vee\text{-I\_L} \qquad \frac{B}{A \vee B}\ \vee\text{-I\_R} \qquad \cfrac{A \vee B \qquad \cfrac{\cfrac{A}{\cdots}}{C} \qquad \cfrac{\cfrac{B}{\cdots}}{C}}{C}\ \vee\text{-E}$$

$$\frac{}{\text{true}}\ \text{true} \qquad \cfrac{\cfrac{\cfrac{A}{\cdots}}{B}}{A \Rightarrow B}\ \Rightarrow\text{-I} \qquad \frac{A \qquad A \Rightarrow B}{B}\ \Rightarrow\text{-E}$$

(modus ponens)

Example: Logic (cont'd)

$$\dfrac{\dfrac{\dfrac{A}{\cdots}}{C} \quad \dfrac{\dfrac{A}{\cdots}}{\neg C}}{\neg A} \ \neg\text{-I}$$

(reductio ad absurdum)

$$\dfrac{A \qquad \neg A}{B} \ \neg\text{-E}$$

(noncontradiction)

- Note these are axioms from classical logic

Example derivations

$$\dfrac{\dfrac{A \wedge (B \vee C)}{A}}{(A \wedge (B \vee C)) \Rightarrow A}$$

$$\dfrac{\dfrac{A \vee (A \wedge B) \qquad A \qquad \dfrac{A \wedge B}{A}}{A}}{A \vee (A \wedge B) \Rightarrow A}$$

IMP: A language of commands

a ::= n | X | a0+a1 | a0-a1 | a0×a1

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

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

- n ∈ N = integers, X ∈ Var = variables, bv ∈ Bool = {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:
  - $\sigma : \mathrm{Var} \rightarrow \mathrm{N}$
  - A state σ is a mapping from variables to their values

Judgments

- Operational semantics has three kinds of judgments
  - $\langle a, \sigma \rangle \rightarrow n$
    - In state σ, arithmetic expression a evaluates to n
  - $\langle b, \sigma \rangle \rightarrow bv$
    - In state σ, boolean expression b evaluates to true or false
  - $\langle c, \sigma \rangle \rightarrow \sigma'$
    - Running command c in state σ produces state σ'
- 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 σ'. We can't just "mutate" σ.

Arithmetic evaluation

$$\frac{}{\langle n, \sigma \rangle \rightarrow n} \qquad \frac{}{\langle X, \sigma \rangle \rightarrow \sigma(X)}$$

$$\frac{\langle a0, \sigma \rangle \rightarrow n0 \quad \langle a1, \sigma \rangle \rightarrow n1}{\langle a0+a1, \sigma \rangle \rightarrow n0+n1} \qquad \frac{\langle a0, \sigma \rangle \rightarrow n0 \quad \langle a1, \sigma \rangle \rightarrow n1}{\langle a0\text{-}a1, \sigma \rangle \rightarrow n0\text{-}n1}$$

$$\frac{\langle a0, \sigma \rangle \rightarrow n0 \quad \langle a1, \sigma \rangle \rightarrow n1}{\langle a0\times a1, \sigma \rangle \rightarrow n0\times n1}$$

Arithmetic evaluation (cont'd)

- Notes:
  - Rule for variables only defined if X is in dom(σ). 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; σ for any state; etc.
  - Order of evaluation irrelevant, because there are no side effects

Sample derivation

- $1+2+3$
- $(2*x)-4$ in $\sigma = [x \mapsto 3]$

Correspondence to OCaml

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

$$\frac{}{\langle \mathrm{true}, \sigma\rangle \to \mathrm{true}} \qquad \frac{}{\langle \mathrm{false}, \sigma\rangle \to \mathrm{false}} \qquad \frac{\langle b, \sigma\rangle \to bv}{\langle \neg b, \sigma\rangle \to \neg bv}$$

$$\frac{\langle a0, \sigma\rangle \to n0 \quad \langle a1, \sigma\rangle \to n1}{\langle a0{=}a1, \sigma\rangle \to n0{=}n1} \qquad \frac{\langle a0, \sigma\rangle \to n0 \quad \langle a1, \sigma\rangle \to n1}{\langle a0{\le}a1, \sigma\rangle \to n0{\le}n1}$$

$$\frac{\langle b0, \sigma\rangle \to bv0 \quad \langle b1, \sigma\rangle \to bv1}{\langle b0{\wedge}b1, \sigma\rangle \to bv0{\wedge}bv1} \qquad \frac{\langle b0, \sigma\rangle \to bv0 \quad \langle b1, \sigma\rangle \to bv1}{\langle b0{\vee}b1, \sigma\rangle \to bv0{\vee}bv1}$$

Sample derivations

- $\neg$false $\wedge$ true
- $2 \leq X \vee X \leq 4$ 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

$$\frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \sigma}$$

$$\frac{\langle c0, \sigma \rangle \rightarrow \sigma 0 \quad \langle c1, \sigma 0 \rangle \rightarrow \sigma 1}{\langle c0; c1, \sigma \rangle \rightarrow \sigma 1}$$

$$\frac{\langle a, \sigma \rangle \rightarrow n}{\langle X{:=}a, \sigma \rangle \rightarrow \sigma[X \mapsto n]}$$

- Here $\sigma[X \mapsto a]$ is the state that is the same as σ, except X now maps to a
  - $(\sigma[X \mapsto a])(X) = a$
  - $(\sigma[X \mapsto a])(Y) = \sigma(Y) \quad X \neq Y$
- Notice order of evaluation explicit in sequence rule

Command evaluation (cont'd)

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true} \qquad \langle c0, \sigma \rangle \rightarrow \sigma 0}{\langle \text{if } b \text{ then } c0 \text{ else } c1, \sigma \rangle \rightarrow \sigma 0}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false} \qquad \langle c1, \sigma \rangle \rightarrow \sigma 1}{\langle \text{if } b \text{ then } c0 \text{ else } c1, \sigma \rangle \rightarrow \sigma 1}$$

- Two rules for conditional
  - Just like in logic we needed two rules for $\wedge$-E and $\vee$-I
  - Notice we specify only one command is executed

Command evaluation (cont'd)

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true} \qquad \langle c; \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma'}$$

Sample derivations

- n:=3; f:=1; while n $\geq$ 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 σ produces a'
  - $b \rightarrow_{\sigma} b'$
    - Evaluating b one step in state σ produces b'
  - $\langle c, \sigma \rangle \rightarrow_{1} \langle c', \sigma' \rangle$
    - Running command c in state σ for one step yields a new command c' and new state σ'
- Note putting σ on the arrow is just a convenience
  - Good notation for stringing evaluations together
    - $a0 \rightarrow_{\sigma} a1 \rightarrow_{\sigma} a2 \rightarrow_{\sigma} \ldots$
  - Put 1 on arrow for commands just to let us distinguish different kinds of arrows

Small-step rules for arithmetic

$$\frac{}{X \to_\sigma \sigma(X)}$$

$$\frac{a0 \to_\sigma a0'}{a0+a1 \to_\sigma a0'+a1} \qquad \frac{a1 \to_\sigma a1'}{n+a1 \to_\sigma n+a1'} \qquad \frac{p=m+n}{n+m \to_\sigma p}$$

- Similarly for - and ×
- 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 ::= □ | C+a | n+C | C-a | n-C | C×a | n×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) × 5)[4] = (4+3) × 5
  - Now add one, single context rule:

$$\frac{a \rightarrow_\sigma a'}{C[a] \rightarrow_\sigma C[a']}$$

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 | if C then c0 else c1 | while C do c
- Now the rules (plus the context rule):

$\langle$X:=n, σ$\rangle$	$\rightarrow_1$	$\langle$skip, σ[x↦n]$\rangle$
$\langle$skip; c1, σ$\rangle$	$\rightarrow_1$	$\langle$c1, σ$\rangle$
$\langle$if true then c0 else c1, σ$\rangle$	$\rightarrow_1$	$\langle$c0, σ$\rangle$
$\langle$if false then c0 else c1, σ$\rangle$	$\rightarrow_1$	$\langle$c1, σ$\rangle$
$\langle$while b do c, σ$\rangle$	$\rightarrow_1$	$\langle$if b then (c; while b do c) else skip, σ$\rangle$

Lambda calculus

- e ::= x | λx.e | e e
- Recall
  - Scope of λ extends as far to the right as possible
    - λx.λy.x y is λx.(λy.(x y))
  - Function application is left-associative
    - x y z is (x y) z
  - Beta-reduction takes a single step of evaluation
    - (λx.e1) e2 → e1[e2\x]

A nonderministic semantics

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1[e2\backslash x]}$$

$$\frac{e \rightarrow e'}{(\lambda x.e) \rightarrow (\lambda x.e')}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

$$\frac{e2 \rightarrow e2'}{e1\ e2 \rightarrow e1\ e2'}$$

- Why are these semantics non-deterministic?

...with context rules

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

$$\frac{e \rightarrow e'}{C[e] \rightarrow C[e']}$$

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1[e2\backslash x]}$$

The Church-Rosser Theorem

- If $a \to^* b$ and $a \to^* c$, there there exists d such that $b \to^* d$ and $c \to^* d$
  - Proof: http://www.mscs.dal.ca/~selinger/papers/lambdanotes.pdf

- Church-Rosser is also called confluence

Normal Form

- A term is in *normal form* if it cannot be reduced
  - Examples: λx.x, λx.λy.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 $\Delta\ \Delta \to \Delta\ \Delta \to \cdots$

- 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 (λx.e1) e2, do not evaluate e2 if e1 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 (λx.e1) e2, always evaluate e2 fully before applying the function
    - Also called **call-by-value (c.b.v.)**

C.b.n. small-step semantics

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

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1[e2\backslash x]} \qquad \frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

- 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:
  - $C ::= \square \mid C\ e$

C.b.v. small-step semantics

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

$$\frac{}{(\lambda x.e)\ v \to e[v \backslash x]}$$

$$\frac{e1 \to e1'}{e1\ e2 \to e1'\ e2}$$

$$\frac{e2 \to e2'}{v\ e2 \to v\ e2'}$$

- 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:
  - $C ::= \square \mid C\ e \mid v\ C$

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.)