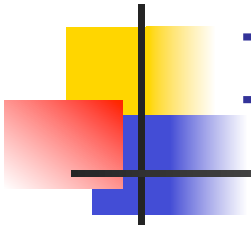
Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

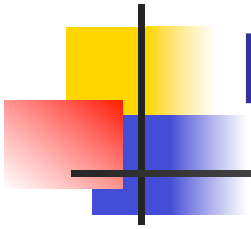
Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next "state"
  - To get final value, put in a loop

Natural Semantics Example

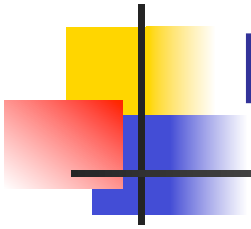


- compute_exp (Var(v), m) = look_up v m
- compute_exp (Int(n), _) = Num (n)
- …
- compute_com(IfExp(b,c1,c2),m) =

  if compute_exp (b,m) = Bool(true)

  then compute_com (c1,m)

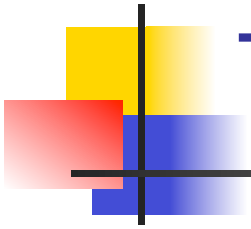
  else compute_com (c2,m)



Natural Semantics Example

- compute_com(While(b,c), m) =
  if compute_exp (b,m) = Bool(false)
  then m
  else compute_com
  (While(b,c), compute_com(c,m))

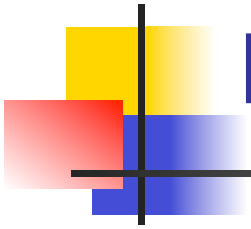
- May fail to terminate - exceed stack limits
- Returns no useful information then



Transition Semantics

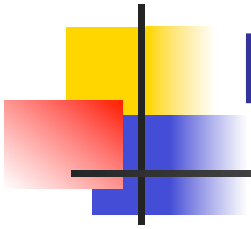
- Form of operational semantics
- Describes how each program construct transforms machine state by *transitions*
- Rules look like

  (C, *m*) --> (C′, *m′*) or (C,*m*) --> *m′*
- *C, C′* is code remaining to be executed
- m, m′ represent the state/store/memory/ environment
  - Partial mapping from identifiers to values
  - Sometimes *m* (or *C*) not needed
- Indicates exactly one step of computation



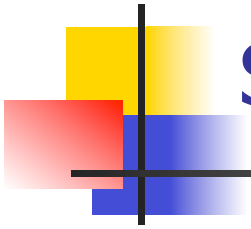
Expressions and Values

- *C, C'* used for commands; *E, E' for* expressions; *U,V* for values
- Special class of expressions designated as *values*
  - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
  - Other possibilities exist



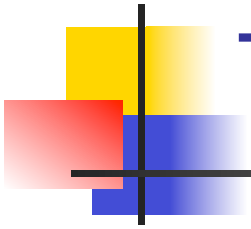
Evaluation Semantics

- Transitions successfully stops when E/*C* is a value/memory
- Evaluation fails if no transition possible, but not at value/memory
- Value/memory is the final *meaning* of original expression/command (in the given state)
- Coarse semantics: final value / memory
- More fine grained: whole transition sequence



Simple Imperative Programming Language

- $I \in Identifiers$
- $N \in Numerals$
- B ::= true | false | B & B | B or B | not B | $E < E$ | $E = E$
- $E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid - E$
- C::= skip | $C;C$ | I ::= E
  | if B then C else C fi | while B do C od



Transitions for Expressions

- Numerals are values
- Boolean values = {true, false}
- Identifiers: (*I*,*m*) --> (*m*(*I*), *m*)

Boolean Operations:

- Operators: (short-circuit)

(false & *B*, *m*) --> (false,*m*)

(true & *B*, *m*) --> (*B*,*m*)

$$\frac{(B, m) \rightarrow (B'', m)}{(B \,\&\, B', m) \rightarrow (B'' \,\&\, B', m)}$$

(true or *B*, *m*) --> (true,*m*)

(false or *B*, *m*) --> (*B*,*m*)

$$\frac{(B, m) \rightarrow (B'', m)}{(B \text{ or } B', m) \rightarrow (B'' \text{ or } B', m)}$$

(not true, m) --> (false,*m*)

(not false, m) --> (true,*m*)

$$\frac{(B, m) \rightarrow (B', m)}{(\text{not } B, m) \rightarrow (\text{not } B', m)}$$

Relations

$$\frac{(E, m) \dashrightarrow (E'', m)}{(E \sim E', m) \dashrightarrow (E'' \sim E', m)}$$

$$\frac{(E, m) \dashrightarrow (E', m)}{(V \sim E, m) \dashrightarrow (V \sim E', m)}$$

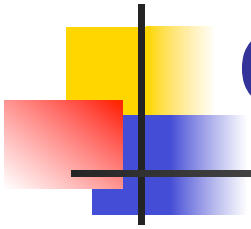
$(U \sim V, m) \dashrightarrow$ (true,m) or (false,m)
depending on whether $U \sim V$ holds or not

Arithmetic Expressions

$$\frac{(E, m) \text{ --> } (E'', m)}{(E \text{ op } E', m) \text{ --> } (E'' \text{ op } E', m)}$$

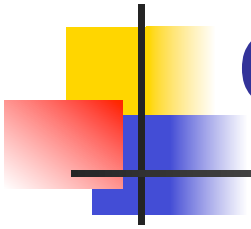
$$\frac{(E, m) \text{ --> } (E', m)}{(V \text{ op } E, m) \text{ --> } (V \text{ op } E', m)}$$

$(U \text{ op } V, m) \text{ --> } (N, m)$ where N is the specified value for $U \text{ op } V$



Commands - in English

- skip means done evaluating
- When evaluating an assignment, evaluate the expression first
- If the expression being assigned is already a value, update the memory with the new value for the identifier
- When evaluating a sequence, work on the first command in the sequence first
- If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory



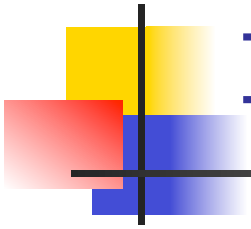
Commands

$$(\text{skip}, m) \dashrightarrow m$$

$$\frac{(E,m) \dashrightarrow (E',m)}{(I::=E,m) \dashrightarrow (I::=E',m)}$$

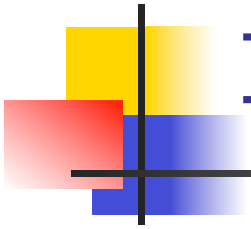
$$(I::=V,m) \dashrightarrow m[I \leftarrow\!\!\text{-- } V]$$

$$\frac{(C,m) \dashrightarrow (C'',m')}{(C;C', m) \dashrightarrow (C'';C',m')} \qquad \frac{(C,m) \dashrightarrow m'}{(C;C', m) \dashrightarrow (C',m')}$$



If Then Else Command - in English

- If the boolean guard in an if_then_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.

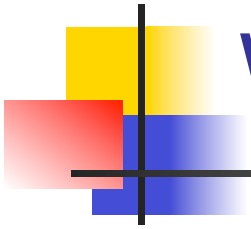


If Then Else Command

(if true then C else C' fi, m) --> (C, m)

(if false then C else C' fi, m) --> (C', m)

$$\frac{(B,m) \text{ --> } (B',m)}{\begin{array}{c}(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \\ \text{--> } (\text{if } B' \text{ then } C \text{ else } C' \text{ fi}, m)\end{array}}$$

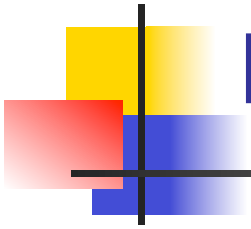


While Command

(while *B* do *C* od, *m*)

--> (if *B* then *C*; while *B* do *C* od else skip fi, m)

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.



Example Evaluation

- First step:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,

{x -> 7})

--> ?

Example Evaluation

- First step:

$$\frac{(x > 5, \{x \rightarrow 7\}) \rightarrow ?}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \rightarrow 7\}) \rightarrow ?}$$

Example Evaluation

- First step:

$$\frac{\dfrac{(x,\{x \to 7\}) \to (7, \{x \to 7\})}{(x > 5, \{x \to 7\}) \to ?}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \to 7\}) \to ?}$$

Example Evaluation

- First step:

$$\frac{\dfrac{(x,\{x \to 7\}) \to (7, \{x \to 7\})}{(x > 5, \{x \to 7\}) \to (7 > 5, \{x \to 7\})}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \to 7\}) \to ?}$$

Example Evaluation

- First step:

$$\frac{\dfrac{(x,\{x \to 7\}) \to (7, \{x \to 7\})}{(x > 5, \{x \to 7\}) \to (7 > 5, \{x \to 7\})}}{(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \to 7\})}$$

$$\to (\text{if } 7 > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \to 7\})$$

Example Evaluation

- Second Step:

$$\frac{(7 > 5, \{x \to 7\}) \to (\text{true}, \{x \to 7\})}{(\text{if } 7 > 5 \text{ then } y:=2 + 3 \text{ else } y:=3 + 4 \text{ fi}, \{x \to 7\}) \to (\text{if true then } y:=2 + 3 \text{ else } y:=3 + 4 \text{ fi}, \{x \to 7\})}$$

- Third Step:

$$(\text{if true then } y:=2 + 3 \text{ else } y:=3 + 4 \text{ fi}, \{x \to 7\}) \to (y:=2+3, \{x \to 7\})$$

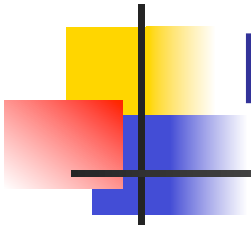
Example Evaluation

- Fourth Step:

$$\frac{(2+3, \{x\text{-}> 7\}) \text{ --> } (5, \{x \text{ -> } 7\})}{(y:=2+3, \{x\text{->}7\}) \text{ --> } (y:=5, \{x\text{->}7\})}$$

- Fifth Step:

$$(y:=5, \{x\text{->}7\}) \text{ --> } \{y \text{ -> } 5, x \text{ -> } 7\}$$



Example Evaluation

- Bottom Line:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})

--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

-->(if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

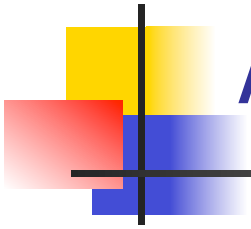
-->(y:=2+3, {x->7})

--> (y:=5, {x->7}) --> {y -> 5, x -> 7}

Transition Semantics Evaluation

- A sequence of steps with trees of justification for each step

- Let -->* be the transitive closure of -->
- Ie, the smallest transitive relation containing -->



Adding Local Declarations

- Add to expressions:
- *E* ::= … | let *I* = *E* in *E′* | fun *I* -> *E* | *E E′*
- fun *I* -> *E* is a value
- Could handle local binding using state, but have assumption that evaluating expressions doesn't alter the environment
- We will use substitution here instead
- **Notation**: *E* [*E′* / *I*] means replace all free occurrence of *I* by *E′* in *E*

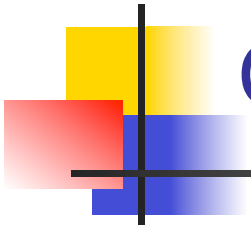
Call-by-value (Eager Evaluation)

$$(\text{let } I = V \text{ in } E, m) \dashrightarrow (E[V/I], m)$$

$$\frac{(E, m) \dashrightarrow (E'', m)}{(\text{let } I = E \text{ in } E', m) \dashrightarrow (\text{let } I = E'' \text{ in } E')}$$

$$((\text{fun } I \rightarrow E)\ V, m) \dashrightarrow (E[V/I], m)$$

$$\frac{(E', m) \dashrightarrow (E'', m)}{((\text{fun } I \rightarrow E)\ E', m) \dashrightarrow ((\text{fun } I \rightarrow E)\ E'', m)}$$

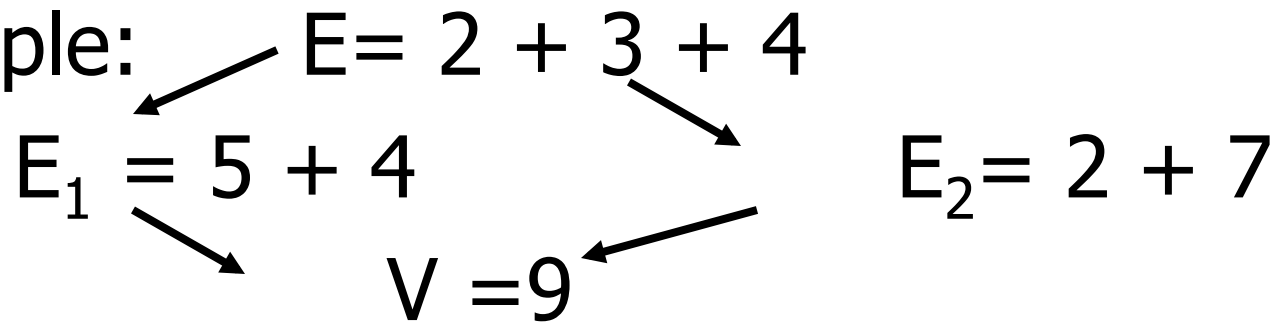


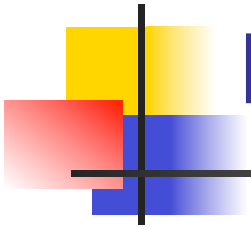
Call-by-name (Lazy Evaluation)

- (let $I = E$ in E', m) --> (E'[E / I],m)
- (((fun I -> E') E, m) --> (E'[E / I],m)
- Question: Does it make a difference?
- It can depending on the language

Church-Rosser Property

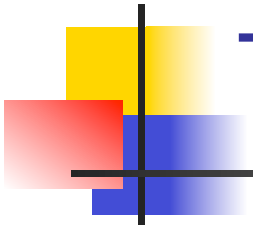
- Church-Rosser Property: If E-->* E_1 and E-->* E_2, if there exists a value V such that E_1 -->* V, then E_2 -->* V
- Also called **confluence** or **diamond property**
- Example:





Does It always Hold?

- No. Languages with side-effects tend not be Church-Rosser with the combination of call-by-name and call-by-value
- Alonzo Church and Barkley Rosser proved in 1936 the λ-calculus does have it
- Benefit of Church-Rosser: can check equality of terms by evaluating them (Given evaluation strategy might not terminate, though)



Transition Semantics for λ-Calculus

- Application (version 1)

$$(\lambda\, x\, .\, E)\; E' \text{ --> } E[E'/x]$$

- Application (version 2)

$$(\lambda\, x\, .\, E)\; V \text{ --> } E[V/x]$$

$$\frac{E' \text{ --> } E''}{(\lambda\, x\, .\, E)\; E' \text{ --> } (\lambda\, x\, .\, E)\; E''}$$