Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/sp2023

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

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*)

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

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

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

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

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

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

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

Relations

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

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

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

Arithmetic Expressions

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

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

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

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

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

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

$$(I::=V,m) \dashrightarrow m[I \leftarrow V]$$

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

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)

$$\frac{(B,m) \dashrightarrow (B',m)}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \;\text{-->}\; (\text{if } B' \text{ then } C \text{ else } C' \text{ fi}, m)}$$

What should while transition to?

(while B do C od, m) → ?

Wrong! BAD

$$\frac{(B, m) \rightarrow (B', m)}{(\text{while } B \text{ do } C \text{ od}, m) \longrightarrow (\text{while } B' \text{ do } C \text{ od}, m)}$$

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:

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

Example Evaluation

- First step:

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

Example Evaluation

- First step:

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

Example Evaluation

- First step:

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

$$\begin{array}{c}\text{--> } (\text{if } 7 > 5 \text{ then } y{:=}2 + 3 \text{ else } y{:=}3 + 4 \text{ fi},\\ \{x \text{ -> } 7\})\end{array}$$

Example Evaluation

- Second Step:

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

- Third Step:

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

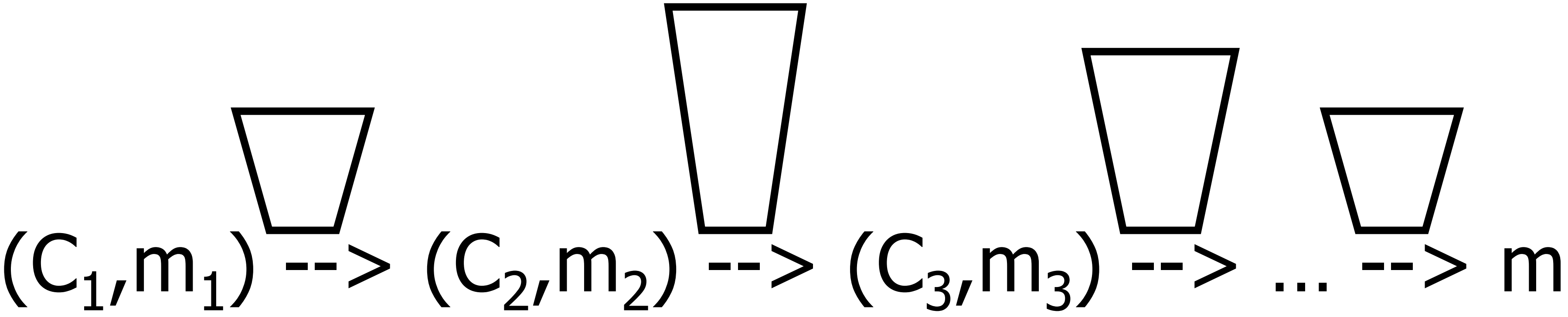
-->(if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

-->(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



(C_1,m_1) --> (C_2,m_2) --> (C_3,m_3) --> … --> m

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

Programming Languages & Compilers

III : Language Semantics

Operational Semantics

Lambda Calculus

Axiomatic Semantics

Lambda Calculus - Motivation

- Aim is to capture the essence of functions, function applications, and evaluation
- λ–calculus is a theory of computation
- “The Lambda Calculus: Its Syntax and Semantics”. H. P. Barendregt. North Holland, 1984

Lambda Calculus - Motivation

- All *sequential programs* may be viewed as functions from input (initial state and input values) to output (resulting state and output values).
- λ-calculus is a mathematical formalism of functions and functional computations
- Two flavors: typed and untyped

Untyped λ-Calculus

- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: λ x. e

    (Function creation, think fun x -> e)
  - Application: $e_1\ e_2$
  - Parenthesized expression: (e)

Untyped λ-Calculus Grammar

- Formal BNF Grammar:
  - <expression> ::= <variable>
    | <abstraction>
    | <application>
    | (<expression>)
  - <abstraction>
    ::= λ<variable>.<expression>
  - <application>
    ::= <expression> <expression>

Untyped λ-Calculus Terminology

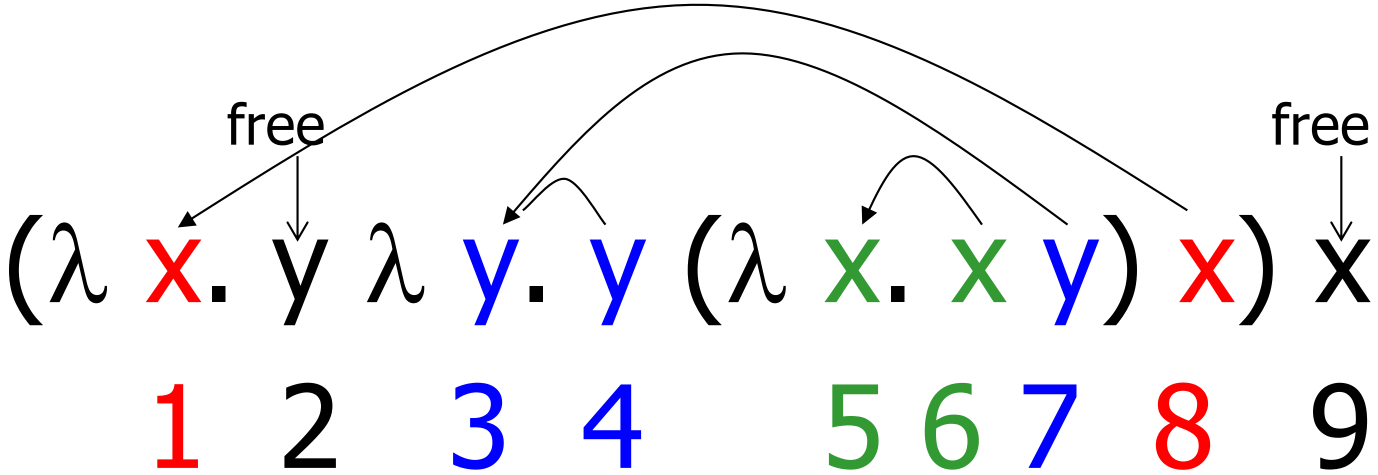
- Occurrence: a location of a subterm in a term
- Variable binding: λ x. e is a binding of x in e
- Bound occurrence: all occurrences of x in λ x. e
- Free occurrence: one that is not bound
- Scope of binding: in λ x. e, all occurrences in e not in a subterm of the form λ x. e′ (same x)
- Free variables: all variables having free occurrences in a term

Example

- Label occurrences and scope:

```
(λ x. y λ y. y (λ x. x y) x) x
   1  2   3  4    5  6 7  8  9
```

Example

- Label occurrences and scope:
- 

(λ x. y λ y. y (λ x. x y) x) x

1 2 3 4 5 6 7 8 9

Untyped λ-Calculus

- How do you compute with the λ-calculus?
- Roughly speaking, by substitution:
- $(\lambda\ x.\ e_1)\ e_2 \Rightarrow^* e_1\ [e_2\ /\ x]$
- * Modulo all kinds of subtleties to avoid free variable capture

Transition Semantics for λ-Calculus

$$\frac{E \text{ --> } E''}{E\ E' \text{ --> } E''\ E'}$$

- Application (version 1 - Lazy Evaluation)

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

- Application (version 2 - Eager Evaluation)

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

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

V - variable or abstraction (value)

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is Turing Complete
  - Can express any sequential computation
- Problems:
  - How to express basic data: booleans, integers, etc?
  - How to express recursion?
  - Constants, if_then_else, etc, are conveniences; can be added as syntactic sugar

Typed vs Untyped λ-Calculus

- The *pure* λ-calculus has no notion of type: (f f) is a legal expression
- Types restrict which applications are valid
- Types are not syntactic sugar! They disallow some terms
- Simply typed λ-calculus is less powerful than the untyped λ-Calculus: NOT Turing Complete (no recursion)

α Conversion

1. α-conversion:
   2. λ x. exp --α--> λ y. (exp [y/x])
3. Provided that
   1. y is not free in exp
   2. No free occurrence of x in exp becomes bound in exp when replaced by y

λ x. x (λ y. x y) -×-> λ y. y(λ y.y y)

α Conversion Non-Examples

1. Error: y is not free in term second

$$\lambda\ x.\ x\ y\ \xcancel{\text{--}\alpha\text{-->}}\ \lambda\ y.\ y\ y$$

2. Error: free occurrence of x becomes bound in wrong way when replaced by y

$$\lambda\ x.\ \underbrace{\lambda\ y.\ x\ y}_{\text{exp}}\ \xcancel{\text{--}\alpha\text{-->}}\ \lambda\ y.\ \underbrace{\lambda\ y.\ y\ y}_{\text{exp}[y/x]}$$

But $\lambda\ x.\ (\lambda\ y.\ y)\ x\ \text{--}\alpha\text{-->}\ \lambda\ y.\ (\lambda\ y.\ y)\ y$

And $\lambda\ y.\ (\lambda\ y.\ y)\ y\ \text{--}\alpha\text{-->}\ \lambda\ x.\ (\lambda\ y.\ y)\ x$