Programming Languages and Compilers (CS 421)

Sasa Misailovic

4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2024/CS421C

Based on slides by Elsa Gunter, which are based in part on previous slides by Mattox Beckman and updated by Vikram Adve and Gul Agha

Moving On With Richer Expressions

- How do we extend the grammar to support nested additions, e.g., 1 * (0 + 1)

```
<exp>      ::= <factor>
             |  <factor> +  <exp>

<factor>   ::=  <bin>
             |   <bin>  *  <factor>

<bin>      ::=  0  | 1
```

Moving On With Richer Expressions

- How do we extend the grammar to support nested additions, e.g., 1 * (0 + 1)

```
<exp>      ::= <factor>
             |  <factor> +  <exp>

<factor>   ::=  <bin>
             |   <bin>  *  <factor>

<bin>      ::=  0  | 1 | ( <exp> )
```

Moving On With Richer Expressions

- How do we extend the grammar to support other operations, subtraction and division?

```
<exp>      ::= <factor>
             | <factor> + <exp> | <factor> - <exp>

<factor>   ::= <bin>
             | <bin> * <factor> | <bin> / <factor>

<bin>      ::= 0 | 1 | ( <exp> )
```

Disambiguating a Grammar

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***a*** to have <u>higher precedence</u> than ***b***, which in turn has <u>higher precedence</u> than ***m***, and such that ***m*** <u>associates to the left</u>.
- Think of a,b,m as operators
  e.g., a is "++", b is "!" , m is "*"

Disambiguating a Grammar

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***a*** to have <u>higher precedence</u> than ***b***, which in turn has <u>higher precedence</u> than ***m***, and such that ***m*** <u>associates to the left</u>.

- <exp> ::= <exp> m <not_m> | <not_m>
- <not_m> ::= b <not_m> | <not_b_m>
- <not_b_m> ::= <not_b_m>a | 0 | 1

Disambiguating a Grammar – Take 2

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***b*** to have higher precedence than ***m***, which in turn has higher precedence than ***a***, and such that ***m*** associates to the right.

Disambiguating a Grammar – Take 2

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***b*** to have higher precedence than ***m***, which in turn has higher precedence than ***a***, and such that ***m*** associates to the right.
- <exp> ::=
  <no_a_m> | <no_m> m <no_a>| <exp> a
- <no_a> ::= <no_a_m> | <no_a_m> m <no_a>
- <no_m> ::= <no_a_m> | <exp> a
- <no_a_m> ::= b <no_a_m> | 0 | 1

Disambiguating a Grammar – Take 3

- <exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want ***a*** has higher precedence than ***m***, which in turn has higher precedence than ***b***, and such that ***m*** associates to the right.
- For you…

Disambiguating Grammars – Dangling Else

- stmt ::= ...
  | **if** (expr) stmt
  | **if** (expr) stmt **else** stmt

- How can we parse
  if (e1) if (e2) s1 else s2 ?

Disambiguating Grammars – Dangling Else

- Try: let us try to differentiate if we have **if** inside the **then** branch or not….
- stmt = open_stmt | closed_stmt
- open_stmt ::= **if** (expr) stmt
  | **if** (expr) closed_stmt **else** open_stmt
- closed_stmt ::= non_if_statement
  | **if** (expr) closed_stmt **else** closed_stmt

- How can we parse **if** (e1) **if** (e2) s1 **else** s2 now ?

Programming Languages & Compilers

Three Main Topics of the Course

I

New Programming Paradigm

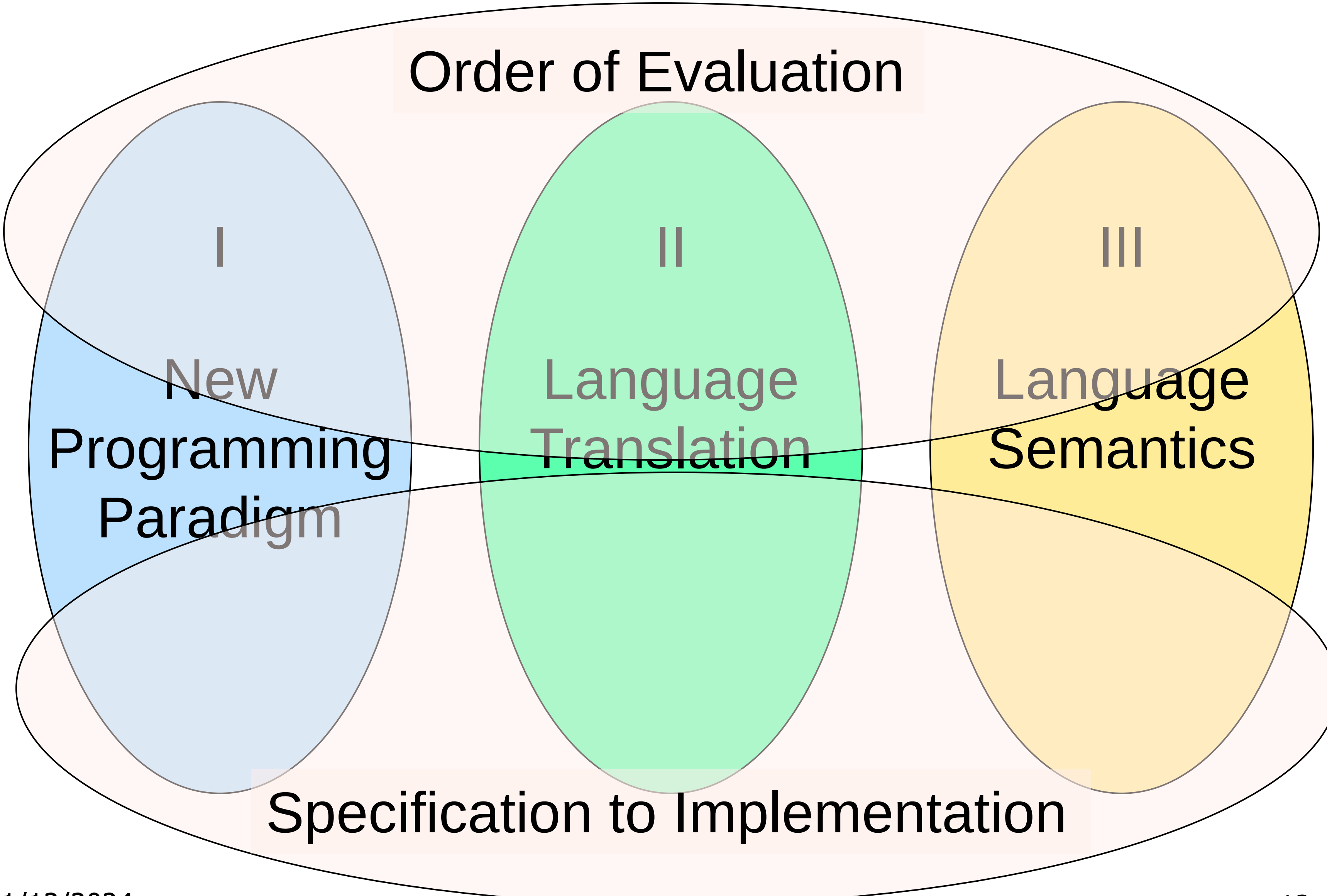
II

Language Translation

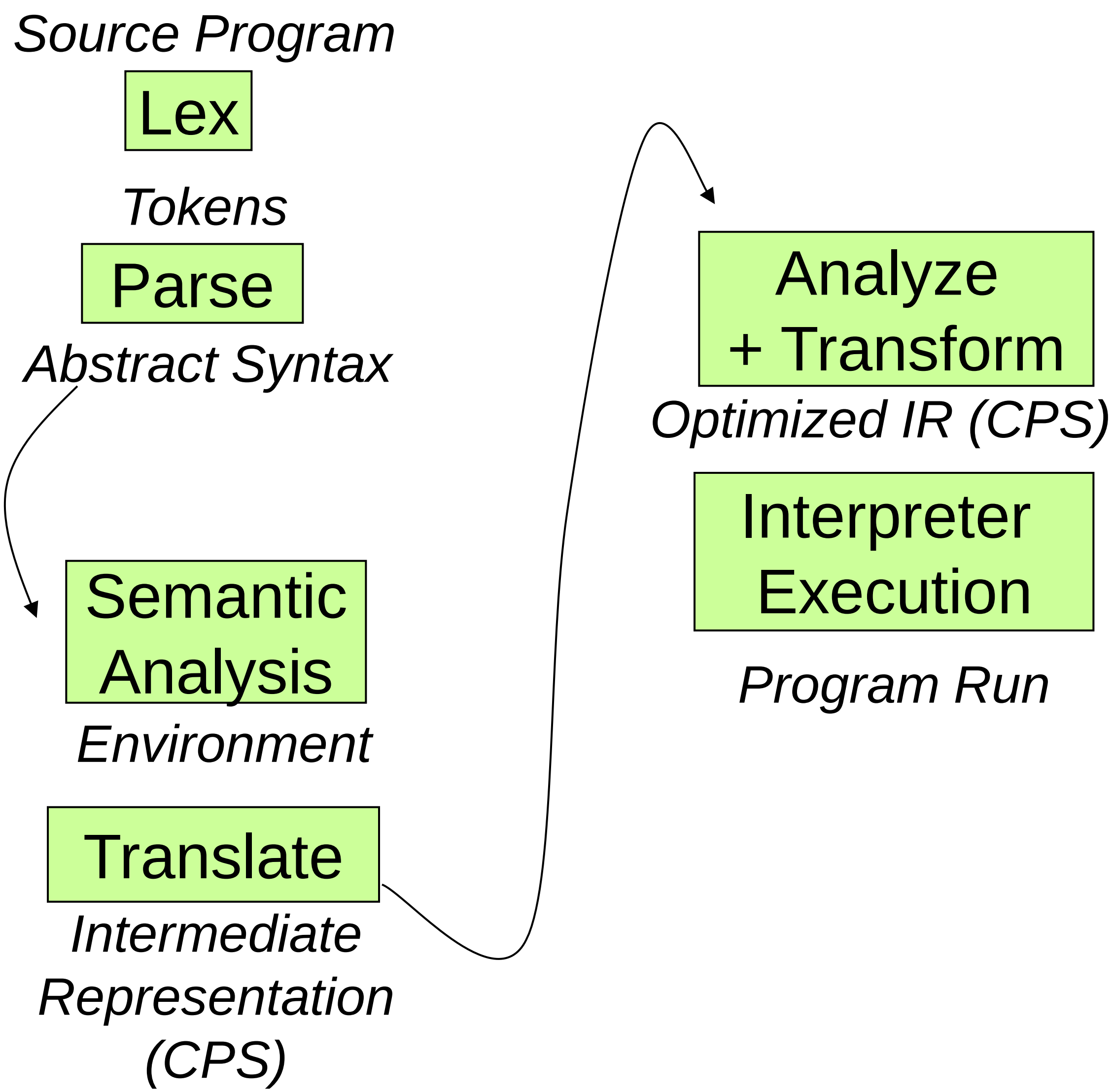
III

Language Semantics

Programming Languages & Compilers



Major Phases of a PicoML Interpreter



Where do we stand?

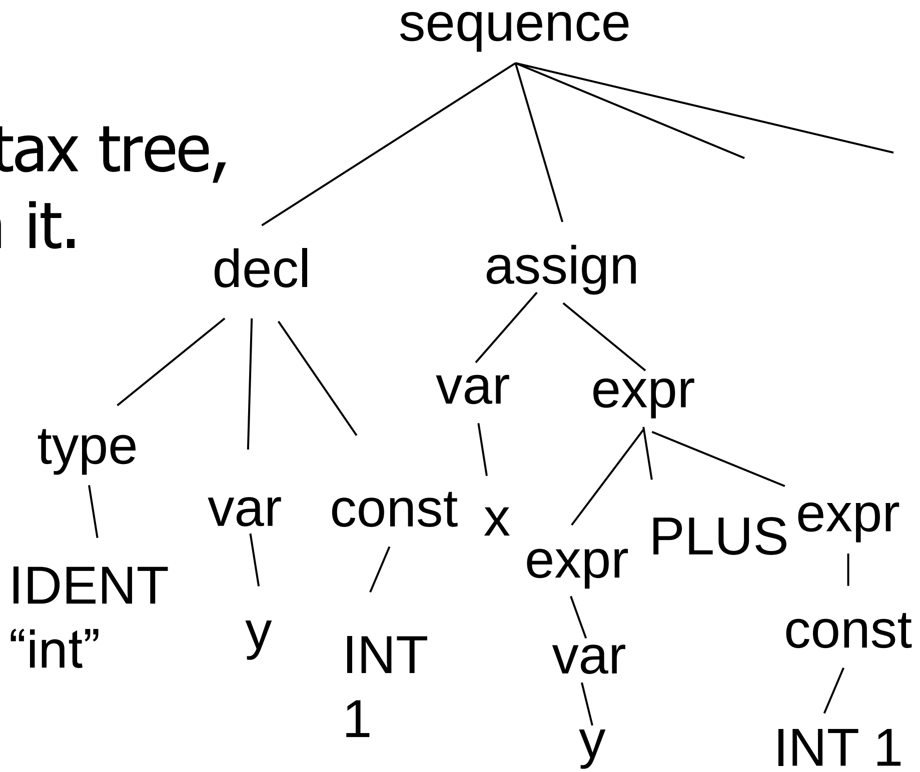
- Conceptually:

Input: “int y = 1; x = y + 1”

Lexer: [IDENT “int”, IDENT “main”, LPAREN, RPAREN, LCURLY, RCURLY]

Parser: turns this list into a syntax tree, so we can more easily work with it.

Typechecker: makes sure all variables are properly typed by traversing the syntax tree



Programming Languages & Compilers

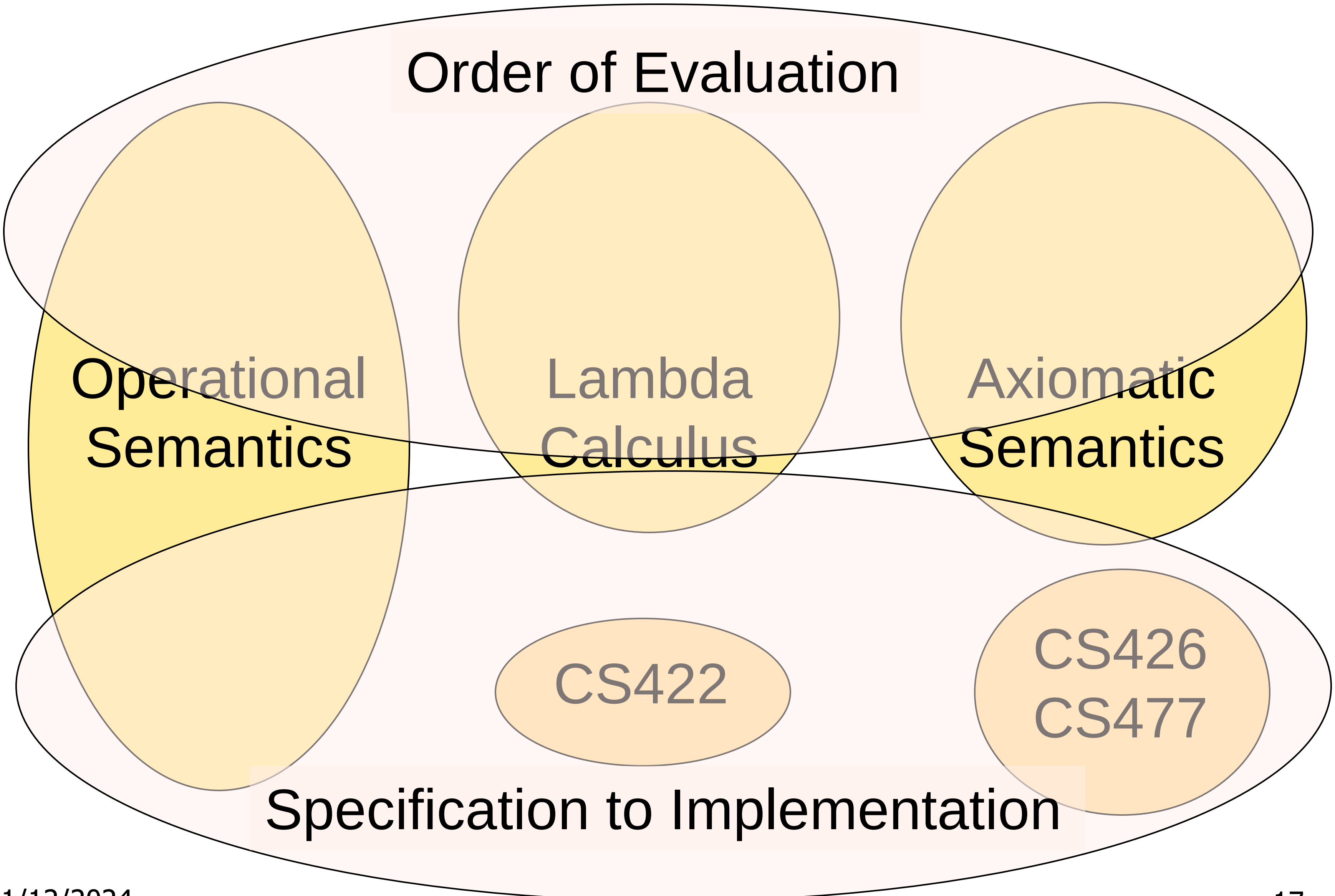
III : Language Semantics

Operational Semantics

Lambda Calculus

Axiomatic Semantics

Programming Languages & Compilers



Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference

Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics

Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution

- Written :

  {Precondition} Program {Postcondition}

- Source of idea of *loop invariant*

Denotational Semantics

- Construct a function $\mathcal{M}$ assigning a mathematical meaning to each program construct
- Lambda calculus (or its variants) often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

Natural Semantics

- Aka Structural Operational Semantics, aka "Big Step Semantics"
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

$$(C, m) \Downarrow m'$$

or

$$(E, m) \Downarrow v$$

Simple Imperative Programming Language

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

Natural Semantics of Atomic Expressions

- Identifiers: $(I,m) \Downarrow m(I)$
- Numerals are values: $(N,m) \Downarrow N$

- Booleans: $(\mathrm{true},m) \Downarrow \mathrm{true}$

  $(\mathrm{false}\ ,m) \Downarrow \mathrm{false}$

Booleans Warmup: Negation

$$\frac{(B,\ m) \Downarrow \text{true}}{(\text{not } B,\ m) \Downarrow \text{false}} \qquad \frac{(B,\ m) \Downarrow \text{false}}{(\text{not } B,\ m) \Downarrow \text{true}}$$

Booleans: Conjunction(&) and Disjunction(or)

$$\frac{(B, m) \Downarrow \text{false}}{(B \,\&\, B', m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{true} \quad (B', m) \Downarrow b}{(B \,\&\, B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(B \text{ or } B', m) \Downarrow \text{true}} \qquad \frac{(B, m) \Downarrow \text{false} \quad (B', m) \Downarrow b}{(B \text{ or } B', m) \Downarrow b}$$

Relations Warmup

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad b = (U = V)}{(E = E', m) \Downarrow b}$$

- By $U \sim V = b$, we mean does (the meaning of) the relation $\sim$ hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V

Relations: General Case (~ is any relational operator)

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad (U \sim V) = b}{(E \sim E', m) \Downarrow b}$$

- By $U \sim V = b$, we mean does (the meaning of) the relation ~ hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V

Arithmetic Expressions

$$\frac{(E, m) \Downarrow U \qquad (E', m) \Downarrow V \quad U \; op \; V = N}{(E \; op \; E', m) \Downarrow N}$$

where N is the specified value for $U \; op \; V$

Commands

Skip: $(\text{skip}, m) \Downarrow m$

Assignment:
$$\frac{(E,m) \Downarrow V}{(I{:=}E,m) \Downarrow m\,[I \text{ <-- } V\,]} \quad (=\{\mathrm{I} \text{ -> } \mathrm{V}\}+\mathrm{m})$$

Sequencing:
$$\frac{(C,m) \Downarrow m' \qquad (C',m') \Downarrow m''}{(C;C', m) \Downarrow m''}$$

If Then Else Command

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

$$\frac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

While Command

$$\frac{(B,m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B,m)\Downarrow\text{true} \quad (C,m)\Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m')\Downarrow m''}{(\text{while } B \text{ do C od}, m) \Downarrow m''}$$

Example: If Then Else Rule

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

Example: If Then Else Rule

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

Example: Arith Relation

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

Example: Identifier(s)

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

Example: Arith Relation

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

Example: If Then Else Rule

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

Example: Assignment

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

Example: Arith Op

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

Example: Numerals

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

Example: Arith Op

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

Example: Assignment

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

Example: If Then Else Rule

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

Comment

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explictly*
- Clash of constructs apparent in awkward semantics

Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed

Interpreter

- An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations

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 Interpreter Implementation

- Identifiers: $(k,m) \Downarrow m(k)$
- Numerals are values: $(N,m) \Downarrow N$
- Conditionals: $\dfrac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$ $\dfrac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$

```
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 Interpreter Implementation

- Loop: $\dfrac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$ $\dfrac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$

```
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*)

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

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

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

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

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

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

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

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

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

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

$(U \; op \; V, m) \dashrightarrow (N, m)$ 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)

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

What should while transition to?

(while B do C od, m) → ?

Wrong! BAD

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

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:

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

--> ?

Example Evaluation

- First step:

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

Example Evaluation

- First step:

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

Example Evaluation

- First step:

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

Example Evaluation

- Second Step:

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

- Third Step:

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

$$\longrightarrow (y{:=}2{+}3, \{x \to 7\})$$

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->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

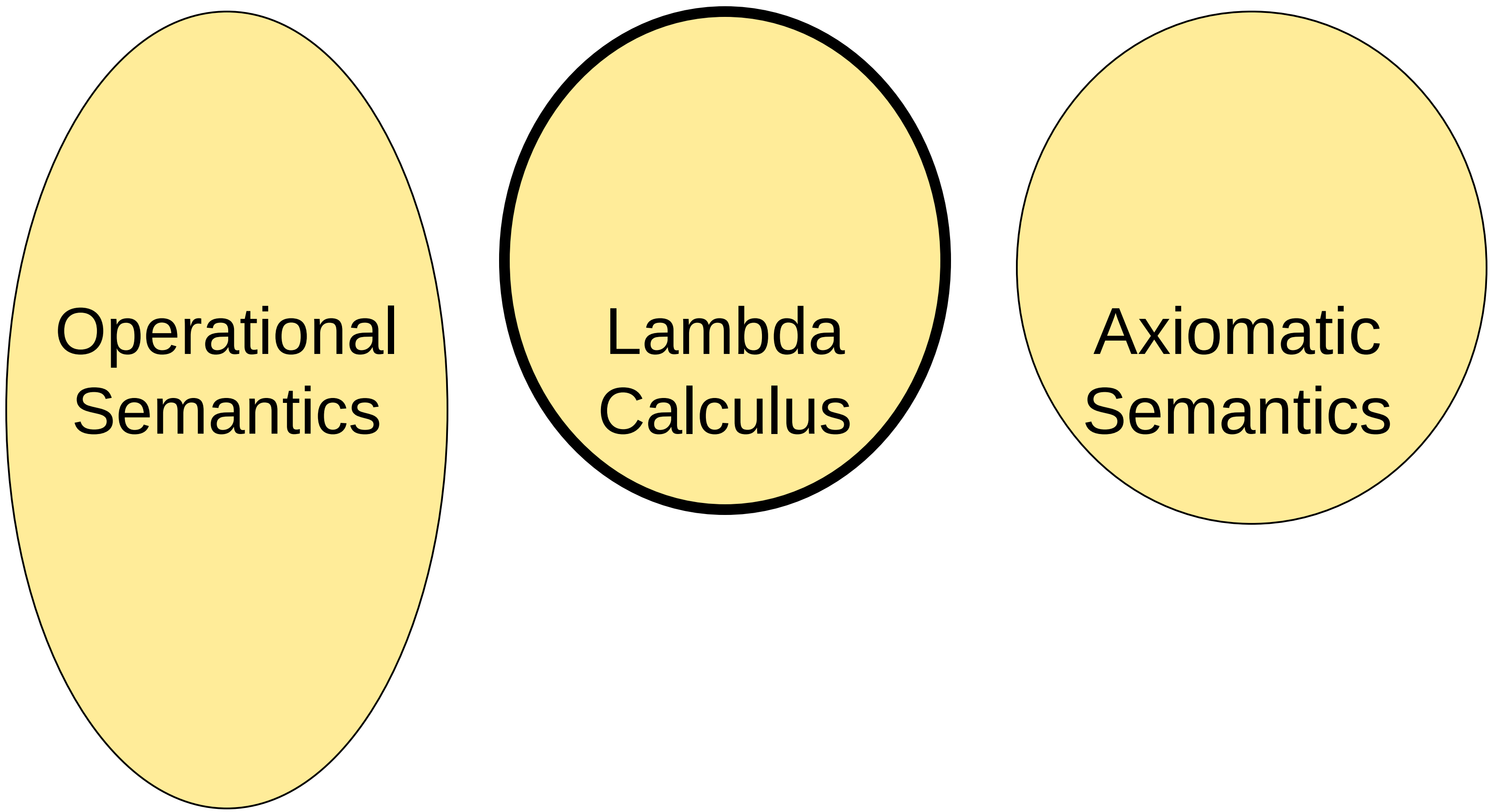
- A sequence of steps with trees of justification for each step

$$(C_1,m_1) \dashrightarrow (C_2,m_2) \dashrightarrow (C_3,m_3) \dashrightarrow \ldots \dashrightarrow m$$

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

Programming Languages & Compilers

III : Language Semantics



Lambda Calculus - Motivation

- Aim is to capture the essence of functions, function applications, and evaluation
- $\lambda-$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:

$$(\lambda\ x.\ y\ \lambda\ y.\ y\ (\lambda\ x.\ x\ y)\ x)\ x$$

1 2 3 4 5 6 7 8 9

Example

- Label occurrences and scope:
- $$(\lambda\ \underset{1}{x}.\ \overset{\text{free}}{\underset{2}{y}}\ \lambda\ \underset{3}{y}.\ \underset{4}{y}\ (\lambda\ \underset{5}{x}.\ \underset{6}{x}\ \underset{7}{y})\ \underset{8}{x})\ \overset{\text{free}}{\underset{9}{x}}$$

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

$$\frac{E \dashrightarrow E''}{E\ E' \dashrightarrow E''\ E'}$$

- Application (version 1 - Lazy Evaluation)

$$(\lambda\ x\ .\ E)\ E' \dashrightarrow E[E'/x]$$

- Application (version 2 - Eager Evaluation)

$$\frac{E' \dashrightarrow E''}{(\lambda\ x\ .\ E)\ E' \dashrightarrow (\lambda\ x\ .\ E)\ E''}$$

$$\frac{}{(\lambda\ x\ .\ E)\ V \dashrightarrow E[V/x]}$$

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

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

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

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

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

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

Congruence

- Let ~ be a relation on lambda terms. ~ is a congruence if
- it is an equivalence relation
- If $e_1 \sim e_2$ then
  - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
  - $\lambda\, x.\ e_1 \sim \lambda\, x.\ e_2$

α Equivalence

- α equivalence is the smallest congruence containing α conversion

- One usually treats α-equivalent terms as equal - i.e. use α equivalence classes of terms

Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

- λ x. (λ y. y x) x --α--> λ z. (λ y. y z) z so
  λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z
- (λ y. y z) --α--> (λ x. x z) so

  (λ y. y z) ~α~ (λ x. x z) so

  (λ y. y z) z ~α~ (λ x. x z) z so

  λ z. (λ y. y z) z ~α~ λ z. (λ x. x z) z
- λ z. (λ x. x z) z --α--> λ y. (λ x. x y) y so

  λ z. (λ x. x z) z ~α~ λ y. (λ x. x y) y
- λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y