Formal Semantics

Formal Semantics

- Chapter 2 and 3 formal definitions of syntax with BNF
- And how to make a BNF that generates correct parse trees: “where syntax meets semantics”
- Parse trees can be simplified into *Abstract Syntax Trees* (AST’s)
- Now… the rest of the story: formal definitions of programming language semantics

Outline

- Natural semantics and F# interpreters
  - Language One
  - Language Two: adding variables
  - Language Three: adding functions
- Text implemented the interpreter in Prolog
- Ours implemented in F#

Defining Language One

- A little language of integer expressions:
  - Constants
  - The binary infix operators **+** and *****, with the usual precedence and associativity
  - Parentheses for grouping
- Lexical structure: tokens are **+**, *****, **(**, **)**, and integer constants consisting of one or more decimal digits

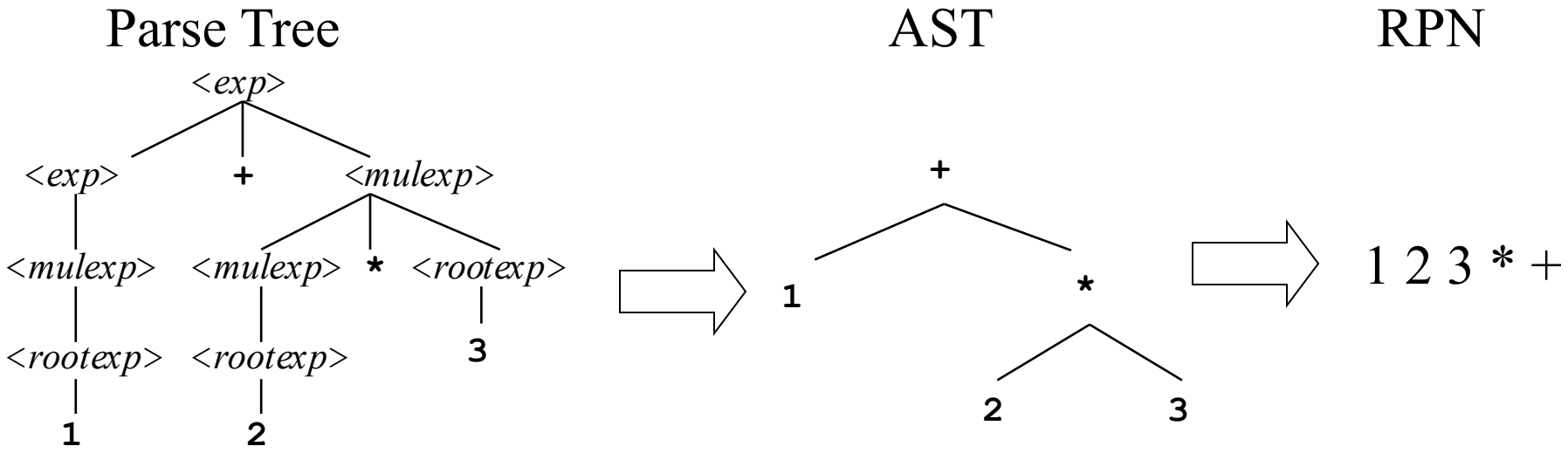
Syntax: Phrase Structure

<exp> **::=** *<exp>* **+** *<mulexp>* | *<mulexp>*
<mulexp> **::=** *<mulexp>* ***** *<rootexp>* | *<rootexp>*
<rootexp> **::=** **(***<exp>***)** | *<constant>*

- (A subset of F# expressions, Java expressions, and Prolog terms)
- This grammar is unambiguous
- Both operators are left associative, and ***** has higher precedence than **+**

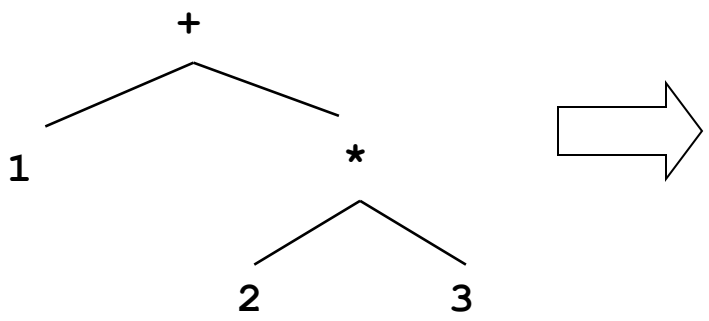
Parse Trees And AST's

- The grammar generates parse trees
- The AST is a simplified form: same order as the parse tree, but no non-terminals
- Can be evaluated in post-order as RPN



Continuing The Definition

- That is as far as we got in Chapters 2 and 3
- One way to define the semantics of the language is to implement an interpreter in F#.
- Start with AST as interpreter's input:




```
plus(
  const(1),times(
                 const(2),const(3)))
```

- Post-order evaluation of AST.

1 2 3 * +

Abstract Syntax

- Note: the set of legal AST's can be defined by a grammar, giving the *abstract syntax* of the language

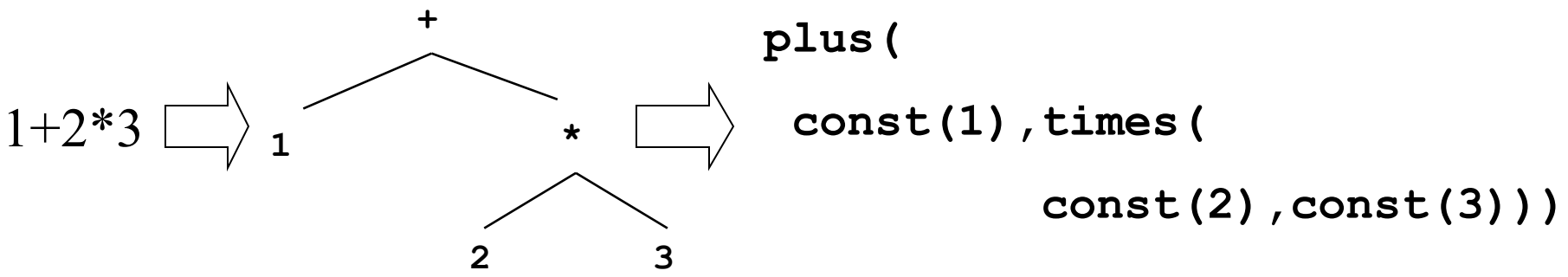
<*exp*> ::= **plus(**<*exp*>**,**<*exp*>**)**
| **times(**<*exp*>**,**<*exp*>**)**
| **const(**<*constant*>**)**

- An abstract syntax can be ambiguous, since the order is already fixed by parsing with the original grammar for *concrete syntax*

Syntax (not semantics)

<*exp*> **::=** <*exp*> **+** <*mulexp*> | <*mulexp*>
<*mulexp*> **::=** <*mulexp*> ***** <*rootexp*> | <*rootexp*>
<*rootexp*> **::=** **(**<*exp*>**)** | <*constant*>

- A subset of F# expressions.
- This grammar is unambiguous.



Language One: F# Interpreter

```
type AST =
        | Const of int
        | Plus of AST * AST
        | Times of AST * AST;;

let rec val1 exp =
  match exp with
     | Const n -> Const n
     | Plus (Const v1, Const v2) -> Const (v1+v2)
     | Plus (E1, E2) -> val1(Plus(val1(E1), val1(E2)))
     | Times (Const v1, Const v2) -> Const (v1*v2)
     | Times (E1, E2) -> val1(Times(val1(E1),
                val1(E2)));;
```

```
val1 (Const(4));;
  val it : AST = Const 4

val1 (Plus(Const(4),Const(5)));;
  val it : AST = Const 9

val1 (Plus(Const(5),Times(Const(3),Const(4))));;
  val it : AST = Const 17
```

Exercise 1

```
1. let rec val1 exp =
2.   match exp with
3.     | Const n -> Const n
4.     | Plus (Const v1, Const v2) -> Const (v1+v2)
5.     | Plus (E1, E2) -> val1(Plus(val1(E1),
          val1(E2)))
6.     | Times (Const v1, Const v2) -> Const (v1*v2)
7.     | Times (E1, E2) -> val1(Times(val1(E1),
                 val1(E2)));;
```

1. Give the Language One representation for: **`2+3*4`**
2. Give the algebraic representation of Language One:
   val1(Times(Const 2,Plus(Const 3,Const 4));;
3. Trace the lines executed for:
   a) **val1(Const 3);;**
   b) **val1(Times(Const 3,Const 4);;**
   c) **val1(Times(Const 2,Plus(Const 3,Const 4));;**

Problems

- What is the value of a constant?
  - Interpreter says **`val1(Const n).`**
  - This means that the value of a constant in Language One is whatever the value of that same constant is *in F#*
  - Unfortunately, different implementations of F# could handle this differently

Value Of A Constant

```
type AST =
      Const of int;
let val1 (Const(n)) = Const(n);
```

```
val1(Const(2147483647));;

 val it : AST = Const 2147483647

val1(Const(2147483648));;

  error FS1147: This number is outside the
allowable range for 32-bit signed integers
```

- Our F# implementation treats constant values greater than 2^{31}-1 as errors
- Did we mean Language One to do this?

Value Of A Sum

```
type AST =
      | Const of int
      | Plus of AST * AST;;

let rec val1 exp =
  match exp with
    | Const n -> Const n
    | Plus (Const v1, Const v2) -> Const (v1+v2);;
```

```
val1(Plus(Const(2147483646),Const(1)));;

  val it : AST = Const 2147483647


val1(Plus(Const(2147483647),Const(1)));;

  val it : AST = Const -2147483648
```

- Our F# implementation wraps on addition of values greater than 2^{31}-1 as errors
- Did we mean Language One to do this?

Defining Semantics By Interpreter Implementation

- Our **`val1`** is not satisfactory as a definition of the semantics of Language One
- "Language One programs behave the way this interpreter says they behave, *running under this implementation of F# on this computer system*"
- We need something more abstract, not defined in terms of a specific implementation

Natural (Operational) Semantics

- A formal notation to capture the same basic rules in **`val1`**
- Define the relation between an AST and the result of evaluating it
- Use the symbol $\rightarrow$ for this relation, writing $E \rightarrow v$ to mean that the AST E evaluates to the value v
- For example, semantics should establish: **`Times(Const 2,Const 3)`** $\rightarrow$ **`Const(6)`**

Logic Inference Rules

- Logic inference rules are written in the form:

$$\frac{\texttt{conditions}}{\texttt{conclusion}}$$

- Example: commutative property of addition

$$\frac{\texttt{a + b = c}}{\texttt{b + a = c}}$$

- Example: transitive property of implication.

$$\frac{\texttt{a} \rightarrow \texttt{b, b} \rightarrow \texttt{c}}{\texttt{a} \rightarrow \texttt{c}}$$

- Axiom: Inference rule with no condition, always holds

$$\frac{}{\texttt{a + 0 = a}}$$

A Rule In Natural Semantics

$$\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{\texttt{times}(E_1, E_2) \rightarrow v_1 \times v_2}$$

- Conditions: **E_1** reduces to **v_1** and **E_2** reduces to **v_2**
- Conclusion: **`times(`E_1`, `E_2`)`** reduces to **v_1 `x` v_2**
- The same idea as our F# **`Times`** reduction rule where:

`Times(Const 4, Const 2)` ⟶ **`Const(4*2)`** ⟶ **`Const(8)`**

```
| (Times(Const v1,Const v2)) -> Const(v1*v2)
| (Times(E1, E2)) -> val1(Times(val1(E1),val1(E2)))
```

Language One, Natural Semantics

$$\mathbf{const}(n) \rightarrow eval(n)$$

```
| (Const(n)) -> Const(n)
```

$$\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{\mathbf{plus}(E_1, E_2) \rightarrow v_1 + v_2}$$

```
| (Plus(Const(v1),Const(v2))) ->
                        Const(v1+v2)

| (Plus(E1, E2)) =
    val1(Plus(val1(E1),val1(E2)))
```

$$\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{\mathbf{times}(E_1, E_2) \rightarrow v_1 \times v_2}$$

```
| (Times(Const(v1),Const(v2))) ->
                        Const(v1*v2)

| (Times(E1, E2)) ->
    val1(Times(val1(E1),val1(E2)))
```

- *eval* is identity since: `val1 (Const(n)) = Const(n)`

Reduction by Natural Semantics

plus(times(const(4),const(2)),const(1))⟶	plus(const(4) x const(2)),const(1))	5.
	⟶ plus(const(8), const(1))	1.
	⟶ const(8) + const(1)	4.
	⟶ const(9)	2.

1. **const(4)x const(2)⟶const(8)**
2. **const(8)+const(1)⟶const(9)**
3. **const(n)⟶ const(n)**

4. $$\frac{E_1 \to v_1 \quad E_2 \to v_2}{\mathbf{plus}(E_1, E_2) \to v_1 + v_2}$$

5. $$\frac{E_1 \to v_1 \quad E_2 \to v_2}{\mathbf{times}(E_1, E_2) \to v_1 \times v_2}$$

Natural Semantics, Note

- There may be more than one rule for a particular kind of AST node
- Two rules for F#-style if-then-else

1. For the conditions where **`E`**$_1$ reduces to **`true`** and **`E`**$_2$ reduces to **`v`**$_2$, the conclusion is **`if(E`**$_1$**`,E`**$_2$**`,E`**$_3$**`)`** reduces to **`v`**$_2$

$$\frac{E_1 \rightarrow true \quad E_2 \rightarrow v_2}{\texttt{if}(E_1, E_2, E_3) \rightarrow v_2}$$

2. For the conditions where **`E`**$_1$ reduces to **`false`** and **`E`**$_3$ reduces to **`v`**$_3$, the conclusion is **`if(E`**$_1$**`,E`**$_2$**`,E`**$_3$**`)`** reduces to **`v`**$_3$

$$\frac{E_1 \rightarrow false \quad E_3 \rightarrow v_3}{\texttt{if}(E_1, E_2, E_3) \rightarrow v_3}$$

Exercise 1.5 - Natural Semantics

Reduce the following:

1. times(plus(const(7),const(1)),const(2)))
2. times(times(const(4),const(2)),const(2)))

1. **const(8)x const(2)→ const(16)**

2. **const(4)x const(2)→ const(8)**

3. **const(7)+const(1)→ const(8)**

4.
$$\frac{E_1 \to v_1 \quad E_2 \to v_2}{\mathbf{plus}(E_1, E_2) \to v_1 + v_2}$$

5.
$$\frac{E_1 \to v_1 \quad E_2 \to v_2}{\mathbf{times}(E_1, E_2) \to v_1 \times v_2}$$

Outline

- Natural semantics and F# interpreters
  - Language One
  - Language Two: adding variables
  - Language Three: adding functions

Defining Language Two

- Language One serves as a foundation for:
- Language Two, which adds:
  - Variables
  - An F#-style **`let`** expression for defining and binding to a value

Syntax (not semantics)

Language One

```
<exp>  ::= <exp> + <mulexp>  |  <mulexp>
<mulexp>  ::= <mulexp> * <rootexp>  |  <rootexp>
<rootexp>  ::= (<exp>)  |  <constant>
```

Language Two

```
<exp>  ::= <exp> + <mulexp>  |  <mulexp>
<mulexp>  ::= <mulexp> * <rootexp>  |  <rootexp>
<rootexp>  ::= let val <variable> = <exp> in <exp> end
                |  (<exp>)  |  <variable>  |  <constant>
```

- Syntax is a subset of F# expressions
- Grammar is unambiguous
- A sample Language Two expression:

```
let y = 3 in y*y
```

Abstract Syntax

- Two more kinds of AST nodes:
  - `type AST =`
    `| Const of int`
    `| Times of AST * AST`
    `| Var of string`
    `| Let of string * AST * AST;;`
  - **`Var "X"`** for a reference to a variable **`X`**
  - **`Let("X",E1,E2)`** for a **`let`** expression that evaluates **`E2`** in an environment where the variable **`X`** is bound to the value of **`E1`**
- So for the Language Two program

  `let y = 3 in y*y`

  we have this AST:

  `Let("y",Const 3,Times(Var "y",Var "y"))`

  `string * AST  *  AST`

Language Two: Examples

let Z = 6 in Z

```
val2 (Let("Z",Const 6, Var "Z")) [];;
  val it : AST = Const 6
```

let Z = 6 in Z+5

```
val2 (Let("Z",Const 6,Plus(Var "Z",Const 5)) [];;
  val it : AST = Const 11
```

Language Two: Examples

val Y = 8

```
val2 (Var "Y") [("Y",Const 8)];;
  val it : AST = Const 8
```

let Z = Y in Z

```
val2 (Let("Z", Var "Y",Var "Z")) [("Y",Const 8)];;
  val it : AST = Const 8
```

let Z = 6 in Z+Y

```
val context = [("Y",Const 8)];;

val2 (Let("Z",Const 6,Plus(Var "Z",Var "Y"))) context;;

  val it : AST = Const 14
```

Language Two: Examples

```
let y = 3 in
  let x = y*y in
    x+x
```

```
val2 (Let("Y",Const 3,
        Let("X",Times(Var "Y",Var "Y"),
                Plus(Var "X",Var "X")))) [];;

  val it : AST = Const 18
```

Exercise 2

1. Give the F# syntax for:

```
val2 (Let("Z",
          Const 5,
          Times(Var "Z",Const 5)))
     [];;
```

2. Give the Language Two syntax for:

```
let Z = 3 in Z*5
```

3. Give the Language Two syntax for:

```
let Z = 3 in Z*5+Z
```

Language Two: Examples

```
val2 (Let("Y",Const 5,
          Let("Y",Const 6, Var "Y"))) [];;

  val it = Const 6 : AST
```

```
let y = 5
let y = 6
in
    y
```

Note: this is a rebinding error in F#!

Representing Environment Contexts

- A representation for environment contexts using lists:
  - **`(Variable,Value)`** = the binding from **`Variable`** to **`Value`**
  - A **context** is a list of zero or more binding terms
- Examples:
  - The context in which **`y`** is bound to Const 3 would be
    `[("y",Const 3)]`
  - The context in which **`x`** is bound to Const 3 and **`y`** to Const 4 would be:
    `[("x",Const 3),("y",Const 4)]`
- Note that the first binding is used when two binding occur:
  - The context in which **`x`** is bound to Const 3, **`y`** to Const 4 and **`x`** to Const 5 would be:
    `[("x",Const 3),("y",Const 4),("x",Const 5)]`

Looking Up A Binding

```
let rec lookup V L =
  let (var,value)::T = L in
  if V=var then value else lookup V T;;
```

- Looks up a binding for **V** in an environment context
- Finds the most recent binding for a given variable **V** when more than one binding
- Following returns Const 3 for first "**x**" found.

```
lookup "x" [("x",Const 3);("x",Const 4)];;
  val it : AST = Const 3
```

- What is returned for "y"?

```
lookup "y" [("x",Const 3);("y",Const 4)];;
```

Language Two: Var Binding Trace

```
1. let rec lookup V L =
2.   let (var,value)::T = L in
3.   if V=var then value else lookup V T;;

4. let rec val2 exp context = match exp with
5.     | Const x -> Const x
6.     | Var V -> lookup V context;;
```

```
val Z = 6;
val X = 3;
Z;
```

```
val2 (Var "Z")
 [("X",Const(3));
  ("Z",Const(6))];;

val it = Const 6 : AST
```

```
6. val2 Var entered
    V = "Z"
    Context=[("X", Const 3),
             ("Z", Const 6)]
1.  lookup entered
      V="Z" T=[("Z",Const 6)]
      value=Const 3 var="X"
1.    lookup entered
        V="Z" T=[]
        value=Const 6 var="Z"
      lookup returned Const 6
    lookup returned Const 6
  val2 returned Const 6
```

Exercise 2.1: Var Bindings

```
1. let rec lookup V L =
2.    let (var,value)::T = L in
3.    if V=var then value else lookup V T;;

4. let rec val2 exp context = match exp with
5.       | Const x -> Const x
6.       | Var V -> lookup V context;;
```

```
val2(
 Var("Z"),[("X",Const(3)); ("Z",Const(6))]);;
```

1. What is the value of "Z"?
2. Trace the above.

```
1. let rec lookup V L =
2.  let (var,value)::T = L in
3.   if V=var then value else lookup V T;;
4. let rec val2 exp context = match exp with
5.   | Const x -> Const x
6.   | Var V -> lookup V context
7.   | Let (V, exp1, exp2) -> val2 exp2 ((V, val2 exp1
     context)::context);;
```

let Z = 6 in Z

```
val2
(Let ("Z",
      Const 6,
       Var "Z")) [];;

val it = Const 6 : AST
```

Binding "Z" in Let line 6

```
(V,val2(Exp1,Context))::Context);
```

binds Z to Const 6 by:

```
("Z", val2(Const 6, []))::[]);
("Z", Const 6)::[])
[("Z", Const 6)]
```

7. val2 Let entered
   V = "Z"
   Exp1 = Const 6
   Exp2 = Var "Z"
   Context = []
5. val2 Const entered X = 6
   val2 returned Const 6
6. val2 Var entered
   V = "Z" Context = [("Z", Const 6)]
1. lookup entered
   V = "Z" T = []
   value = Const 6 var = "Z"
   lookup returned Const 6
   val2 returned Const 6
   val2 returned Const 6

Exercise 2.2: Let Trace

```
1. let rec lookup V L =
2.  let (var,value)::T = L in
3.   if V=var then value else lookup V T;;
4. let rec val2 exp context = match exp with
5.   | Const x -> Const x
6.   | Var V -> lookup V context
7.   | Let (V, exp1, exp2) -> val2 exp2 ((V, val2 exp1
       context)::context);;
```

- Give the corresponding F#.
- Trace the following `val2` executions:

```
1. val2 (Let("Z",Const 6,Var "Z")) [];;

2. val2 (Let("Z",Const 6,Var "Y")) [("Y", Const 8)];;
```

```
1. let rec lookup V L = let (var,value)::T = L in
2.   if V=var then value else lookup V T;;
3. let rec val2 exp context = match exp with
4.    | Const x -> Const x
5.    | Var V -> lookup V context
6.    | Plus (Const x, Const y) -> Const (x+y)
7.    | Plus (x, y) -> val2 (Plus(val2 x context, val2 y context))
        context
8.    | Let (V, exp1, exp2) -> val2 exp2 ((V, val2 exp1
        context)::context);;
```

```
let Z = 6
    in Z+5
```

```
val2
 (Let("Z", Const 6,
   Plus(Var "Z",
    Const 5)))
     [("Z",Const 4)];;

val it AST = Const 11
```

8. val2 Let entered
   V = "Z" Exp1 = Const 6
   Exp2 = Plus (Var "Z", Const 5)
   Context = [("Y", Const 3), ("Z", Const 4)]
4. val2 Const entered X = 6 returned Const 6
7. val2 Plus entered
   X = Var "Z" Y = Const 5
   Context = [("Z", Const 6), ("Z", Const 4)]
5. val2 Var entered
   V="Z" Context =[("Z",Const 6),("Z",Const 4)]
1. lookup entered
   V = "Z" T = [("Z", Const 6),("Z", Const 4)]
   value = Const 6 var = "Z"
   lookup returned Const 6
   val2 returned Const 6
4. val2 Const entered X = 5 returned Const 5
6. val2 Plus entered X=6 Y = 5 returned Const 11
   val2 returned Const 11
val2 returned Const 11

Language Two: Full Interpreter

```
type AST =
  | Const of int
  | Var of string
  | Plus of AST * AST
  | Times of AST * AST
  | Let of string * AST * AST;;

let rec lookup V L = let (var,value)::T = L in
  if V=var then value else lookup V T;;

let rec val2 exp context = match exp with
  | Const x -> Const x
  | Var V -> lookup V context
  | Plus (Const x, Const y) -> Const (x+y)
  | Plus (x, y) -> val2 (Plus(val2 x context, val2 y context))
      context
  | Times (Const x, Const y) -> Const (x*y)
  | Times (x, y) -> val2 (Times(val2 x context, val2 y
      context)) context
  | Let (V, exp1, exp2) -> val2 exp2 ((V, val2 exp1
      context)::context);;
```

Exercise 2.3

```
1. let rec lookup V L = let (var,value)::T = L in
2.   if V=var then value else lookup V T;;

3. let rec val2 exp context = match exp with
4.   | Const x -> Const x
5.   | Var V -> lookup V context
6.   | Plus (Const x, Const y) -> Const (x+y)
7.   | Plus (x, y) -> val2 (Plus(val2 x context, val2 y context))
       context
8.   | Times (Const x, Const y) -> Const (x*y)
9.   | Times (x, y) -> val2 (Times(val2 x context, val2 y
       context)) context
10.  | Let (V, exp1, exp2) -> val2 exp2 ((V, val2 exp
       context)::context);;
```

1. Which line(s) reduces variables?
2. Which line(s) reduces Plus?
3. Which line(s) reduces Let?
4. Where exactly is a variable name bound to a value?
5. Trace: **`val2(Var "Y",[("Y",Const 6)]);`**

Exercise 2.3

```
1. let rec lookup V L = let (var,value)::T = L in
2.   if V=var then value else lookup V T;;

3. let rec val2 exp context = match exp with
4.   | Const x -> Const x
5.   | Var V -> lookup V context
6.   | Plus (Const x, Const y) -> Const (x+y)
7.   | Plus (x, y) -> val2 (Plus(val2 x context, val2 y context))
       context
8.   | Times (Const x, Const y) -> Const (x*y)
9.   | Times (x, y) -> val2 (Times(val2 x context, val2 y
       context)) context
10.  | Let (V, exp1, exp2) -> val2 exp2 ((V, val2 exp
       context)::context);;
```

6. Trace:
   val2(Let("Y",Const 3,Var "Y")) [("Y",Const 6)]

7. What is the environment during execution of line 12 for Question 6?

8. Trace:
   val2 (Var "Y") [("Y",Const 6)]

9. Trace:
   val2(Let("Y",Const 3,Var "Y")) [("Y",Const 6)]

Natural Semantics

- As before, we will write a natural semantics to capture the same rules
- Use the symbol $\rightarrow$ for this relation, though it is a different relation
- Write $<E,C> \rightarrow v$ to mean that the value of the AST E in context C is v

Language Two, Natural Semantics

$$\frac{\langle E_1, C\rangle \to v_1 \quad \langle E_2, C\rangle \to v_2}{\langle \mathbf{plus}(E_1, E_2), C\rangle \to v_1 + v_2}$$

$$\langle \mathbf{var}(v), C\rangle \to lookup(C, v)$$

$$\frac{\langle E_1, C\rangle \to v_1 \quad \langle E_2, C\rangle \to v_2}{\langle \mathbf{times}(E_1, E_2), C\rangle \to v_1 \times v_2}$$

$$\langle \mathbf{const}(n), C\rangle \to eval(n)$$

$$\frac{\langle E_1, C\rangle \to v_1 \quad \langle E_2, bind(x, v_1) :: C\rangle \to v_2}{\langle \mathbf{let}(x, E_1, E_2), C\rangle \to v_2}$$

- Still needs the definitions for +, × and *eval*, as well as *bind, lookup,* ::, and the *nil* environment

Language Two, Natural Semantics

$$\frac{\langle E_1, C\rangle \to v_1 \quad \langle E_2, bind(x, v_1) :: C\rangle \to v_2}{\langle \texttt{let}(x, E_1, E_2), C\rangle \to v_2}$$

Rule for let

- Conditions
  - Expression **E_1** in context **C** reduces to value **v_1**
  - Expression **E_2** in context where variable **x** bound to value **v_1** appended to context **C** reduces to value **v_2**
  - Conclusion
    - **let(x,E_1,E_2)** in context **C** reduces to **v_2**

About Errors

- In Language One, all syntactically correct programs run without error
- Not true in Language Two when variable not in environment context:

```
let a = 1 in b
```

- What does the semantics say about this?

Undefined Variable Error

```
val2 (Let("a",Const 1,Var "b")) []
   Exception: The match cases were incomplete
```

- Implementation fails when **`Var("b")`** not in context **`[("a", Const(1))]`**
- Natural semantics requires condition that E_2 (b) reduces to v_2 in context where x is bound to v_1

$$\frac{\langle E_1, C\rangle \to v_1 \quad \langle E_2, bind(x, v_1) :: C\rangle \to v_2}{\langle \mathbf{let}(x, E_1, E_2), C\rangle \to v_2}$$

- There is no *v* for which

  <**`let(a,const 1,var b)`**, *nil*> $\to v$

Static Semantics

- Ordinarily, language systems perform error checks after parsing but before running
  - For static scoping: references must be in the scope of some definition of the variable
  - For static typing: a consistent way to assign a type to every part of the program
- This part of a language definition, neither syntax nor runtime behavior, is called *static semantics*

Static and Dynamic Semantics

- Language Two semantics could be 2 parts:
  - Static semantics rules out runtime errors
  - Dynamic semantics can ignore the issue
- Static semantics can be complicated too:
  - F#'s type inference
  - Java's "definite assignment"
- In this chapter, dynamic semantics only

Note: Dynamic Error Semantics

- In full-size languages, there are still things that can go wrong at runtime
- One approach is to define error outcomes in the natural semantics:

$$\langle \texttt{divide}(\texttt{const}(\texttt{6}),\texttt{const}(\texttt{3})), C\rangle \rightarrow \langle normal, 2\rangle$$

$$\langle \texttt{divide}(\texttt{const}(\texttt{6}),\texttt{const}(\texttt{0})), C\rangle \rightarrow \langle abrupt, zerodivide\rangle$$

- Our languages: semantics for error-free case only

Outline

- Natural semantics and F# interpreters
  - Language One
  - Language Two: adding variables
  - Language Three: adding functions

Defining Language Three

- Build upon Language Two adding:
  - F#-style function values
  - F#-style function application

Syntax (not semantics)

Language One

```
<exp> ::= <exp> + <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= (<exp>) | <constant>
```

Language Two

```
<exp> ::= <exp> + <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= let <variable> = <exp> in <exp>
            | (<exp>) | <variable> | <constant>
```

Language Three

```
<exp> ::= fn <variable> => <exp> | <addexp>
<addexp> ::= <addexp> + <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <funexp> | <funexp>
<funexp> ::= <funexp> <rootexp> | <rootexp>
<rootexp> ::= let <variable> = <exp> in <exp>
            | (<exp>) | <variable> | <constant>
```

Syntax

Language Three

```
<exp>  ::= fn <variable> => <exp>  |  <addexp>
<addexp>  ::= <addexp> + <mulexp>  |  <mulexp>
<mulexp>  ::= <mulexp> * <funexp>  |  <funexp>
<funexp>  ::= <funexp> <rootexp>  |  <rootexp>
<rootexp>  ::= let <variable> = <exp> in <exp>
             |  (<exp>)  |  <variable>  |  <constant>
```

- A subset of F# expressions
- Grammar is unambiguous
- A sample Language Three expression:

```
(fn x => x * x) 3
```

- Function application <*funexp*> <*rootexp*> has highest precedence

```
(fn x => x * x) 3 + 5 is 14 not 64
```

Parse

1. *<exp>* ::= **fn** *<variable>* **=>** *<exp>* | *<addexp>*
2. *<addexp>* ::= *<addexp>* **+** *<mulexp>* | *<mulexp>*
3. *<mulexp>* ::= *<mulexp>* ***** *<funexp>* | *<funexp>*
4. *<funexp>* ::= *<funexp>* *<rootexp>* | *<rootexp>*
5. *<rootexp>* ::= **let** *<variable>* **=** *<exp>* **in** *<exp>*
   | **(***<exp>***)** | *<variable>* | *<constant>*

(fn x => x * x) 3 + 5

```
1 <exp>= <addexp>
2      = <addexp> + <mulexp>
2      = <mulexp> + <mulexp>
3      = <funexp> + <mulexp>
4      = <funexp> <rootexp> + <mulexp>
4      = <rootexp> <rootexp> + <mulexp>
5      = ( <exp> ) <rootexp> + <mulexp>
1      = (fn <variable> => <exp>) <rootexp> + <mulexp>
1-5    = (fn x => <exp>) <rootexp> + <mulexp>
1      = (fn x => <addexp>) <rootexp> + <mulexp>
2      = (fn x => <mulexp>) <rootexp> + <mulexp>
3      = (fn x => <mulexp> * <funexp>) <rootexp> + <mulexp>
4      = (fn x => <funexp> * <funexp>) <rootexp> + <funexp>
5      = (fn x => <rootexp> * <rootexp>) <rootexp> + <rootexp>
5      = (fn x => <variable> * <variable>) <constant> + <constant>
       = (fn x => x * x) 3 + 5
```

Exercise 3 - Give the parse tree and AST

(fn x => x * x) 3 + 5

```
<exp> = <addexp>
      = <addexp> + <mulexp>
      = <mulexp> + <mulexp>
      = <funexp> + <mulexp>
      = <funexp> <rootexp> + <mulexp>
      = <rootexp> <rootexp> + <mulexp>
      = ( <exp> ) <rootexp> + <mulexp>
      = (fn <variable> => <exp>) <rootexp> + <mulexp>
      = (fn x => <exp>) <rootexp> + <mulexp>
      = (fn x => <addexp>) <rootexp> + <mulexp>
      = (fn x => <mulexp>) <rootexp> + <mulexp>
      = (fn x => <mulexp> * <funexp>) <rootexp> + <mulexp>
      = (fn x => <funexp> * <funexp>) <rootexp> + <funexp>
      = (fn x => <rootexp> * <rootexp>) <rootexp> + <rootexp>
      = (fn x => <variable> * <variable>) <constant> + <constant>
      = (fn x => x * x) 3 + 5
```

Abstract Syntax

- Two more kinds of AST nodes:
  - **`apply(Function, Actual)`** applies the **`Function`** to the **`Actual`** parameter
  - **`fn(Formal, Body)`** for an **`fn`** expression with the given function **`Formal`** parameter and **`Body`**

```
type AST =
         | Fn of String * AST
         | Apply of AST * AST;;
```

- AST of Language Three: **`(fn x => x * x)`**

`Fn( "x", Times(Var "x",Var "x") )`

Formal Body

- AST of Language Three: **`(fn x => x * x) 3`**

`Apply(Fn("x",Times(Var "x",Var "x")), Const 3)`

Function Actual

Representing Functions

- A representation for functions:
  - **`fn(Formal,Body)`**
  - **`Formal`** is the formal parameter variable
  - **`Body`** is the unevaluated function body
- Application of functions:
  - **`apply( fn(Formal, Body), Actual), Context)`**
  - **`Formal`** is the formal parameter variable
  - **`Body`** is the unevaluated function body
  - **`Actual`** is the actual parameter variable
  - **`Context`** is the environment of execution using dynamic scoping

Language Three: F# Interpreter Datatype definitions

```
type AST =
   | Const of int
   | Plus of AST * AST
   | Times of AST * AST
   | Var of string
   | Let of string * AST * AST
   | Fn of String * AST
   | Apply of AST * AST;;
```

Fn Examples

```
(fn x => x * x) 3;
```

```
val3
  (Apply(Fn("x", Times(Var "x",Var "x")),Const 3))
  [];;
val it : AST = Const 9
```

```
val X = 2;
val Y = 3;
(fn A => A * X) Y;
```

```
val3
  (Apply( Fn("A", Times(Var "A",Var "X")), Var "Y"))
  [("X",Const 2);("Y",Const 3)];;
val it : AST = Const 6
```

Fn Examples

Bind a function to a name, invoke function through name.

```
let f = (fun n -> n + n) in
        f 4;;
```

```
val3 (Let("f",
          Fn("n",
             Plus(Var "n",
                  Var "n")),
          Apply(Var "f",
                Const 4)))

     [];;
val it : AST = Const 8
```

Fn Trace

```
1. let val3 exp ctx =  match exp with
2.      | Const x -> Const x
3.      | Fn (formal, body) -> Fn(formal, body);;
```

```
val3 (Fn("z", Const 4))
       [];;


val it : AST = Fn("z", Const 4)
```

```
2. val3 Fn entered
     Formal = "z"
     Body = Const 4
   val3 returned
     Fn ("z", Const 4)
```

Exercise 3.1 - Fn Trace

```
1. let val3 exp ctx =  match exp with
2.      | Const x -> Const x
3.      | Fn (formal, body) -> Fn(formal, body);;
```

```
val3 (Fn("q", Const 7)) [];;
```

1. What is the corresponding F# for the above?
2. What is `Formal` bound to?
3. What is `Body` bound to?
4. Trace the execution.
5. What is the result?

Fn Trace

```
1. let rec val3 exp ctx = match exp with
2.     | Const x -> Const x
3.     | Apply (Fn(formal, body), actual) -> val3 body ((formal, val3
                actual ctx)::ctx)
4.     | Fn (formal, body) -> Fn(formal, body);;
```

```
val3 (Const 4) (("z", val3 (Const 3) [])::[]);;
val3 (Const 4) (("z", Const 3)::[]);;
val3 (Const 4) [("z", Const 3];;
```

Binding of:
z = Const 3

```
(fn z => 4) 3;
```

```
val3
 (Apply(
         Fn("z", Const 4),
         Const 3
        ))
  [];;
val it = Const 4 : AST
```

```
3. val3 Apply entered
        Formal = "z"
        Body = Const 4
        Actual = Const 3
        Context = []

2.    val3 Const entered X=3
      val3 returned Const 3
2.    val3 Const entered X=4
      val3 returned Const 4
   val3 returned Const 4
```

Exercise 3.2 - Fn Trace

```
1. let rec val3 exp ctx = match exp with
2.     | Const x -> Const x
3.     | Apply (Fn(formal, body), actual) -> val3 body ((formal,
           val3 actual ctx)::ctx)
4.     | Fn (formal, body) -> Fn(formal, body);;
```

```
val3( Apply( Fn("q", Const 5), Const 2 )) [] ;;
```

1. What is the corresponding F# for the above?
2. What is `Formal` bound to?
3. What is `Body` bound to?
4. What is `Actual` bound to?
5. What is `Context` bound to?
6. Trace the execution.
7. What is the result?

```
1. let rec lookup V L =
2.  let (var,value)::T = L in
3.   if V=var then value else lookup V T;;

4. let rec val3 exp ctx = match exp with
5.   | Const x -> Const x
6.   | Var V -> lookup V ctx
7.   | Apply (Fn(formal, body), actual) -> val3 body ((formal, val3
       actual ctx)::ctx)
9.   | Fn (formal, body) -> Fn(formal, body);;
```

(fn z => z) 3;

```
val3
 (Apply(
  Fn("z",
     Var "z"
    ),
  Const 3))
 []
;;

val it = Const 3 : AST
```

```
7. val3 Apply entered
        Formal = "z"
        Body = Var "z"
        Actual = Const 3
        Context = []

5.    val3 Const entered  X = 3
      val3 returned Const 3
6.    val3 Var entered  V = "z"   Context = [("z", Const 3)]
1.     lookup entered  V = "z"  T = []  value = Const 3  var = "z"
       lookup returned Const 3
      val3 returned Const 3
  val3 returned Const 3
```

Exercise 3.3 - Fn Trace

```
1. let rec lookup V L =
2.  let (var,value)::T = L in
3.   if V=var then value else lookup V T;;

4. let rec val3 exp ctx = match exp with
5.   | Const x -> Const x
6.   | Var V -> lookup V ctx
7.   | Apply (Fn(formal, body), actual) -> val3 body ((formal, val3
       actual ctx)::ctx)
9.   | Fn (formal, body) -> Fn(formal, body);;
```

```
val3 (Apply( Fn("z", Var "z") Const 2)) [];;
```

1. What is the corresponding F# for the above?
2. What is `Formal` bound to?
3. What is `Body` bound to?
4. What is `Actual` bound to?
5. What is `Context` bound to?
6. Trace the execution.
7. What is the result?

```
1. let rec lookup V L =
2.  let (var,value)::T = L in
3.   if V=var then value else lookup V T;;
4. let rec val3 exp ctx = match exp with
5.   | Const x -> Const x
6.   | Var V -> lookup V ctx
7.   | Plus (Const x, Const y) -> Const (x+y)
8.   | Plus (x, y) -> val2 (Plus(val2 x ctx, val2 y ctx)) ctx
9.   | Apply (Fn(formal, body), actual) -> val3 body ((formal,
        val3 actual ctx)::ctx)
10.  | Fn (formal, body) -> Fn(formal, body);;
```

```
9. val3 Apply entered
     Formal = "z"
     Body = Plus (Var "z", Const 2)
     Actual = Const 3
     Context = []
5.   val3 Const entered X = 3
     val3 returned Const 3
8.   val3 Plus entered
       X = Var "z"
       Y = Const 2
       Context = [("z", Const 3)]
6.     val3 Var entered  V = "z"  Context = [("z", Const 3)]
        lookup entered  V = "z" T = []  value = Const 3 var = "z"
        lookup returned Const 3
       val3 returned Const 3
5.     val3 Const entered  X = 2
       val3 returned Const 2
7.     val3 Plus entered  X = 3  Y = 2
       val3 returned Const 5
     val3 returned Const 5
   val3 returned Const 5
```

```
val3
 (Apply(
  Fn("z",
     Plus(Var "z", Const 2)
    ),
  Const 3))
 []
;;
val it = Const 5 : AST
```

(fn z => z + 2) 3;

Exercise 3.4 - Fn Trace

```
1. let rec lookup V L =
2.  let (var,value)::T = L in
3.   if V=var then value else lookup V T;;
4. let rec val3 exp ctx = match exp with
5.   | Const x -> Const x
6.   | Var V -> lookup V ctx
7.   | Plus (Const x, Const y) -> Const (x+y)
8.   | Plus (x, y) -> val2 (Plus(val2 x ctx, val2 y ctx)) ctx
9.   | Apply (Fn(formal, body), actual) -> val3 body ((formal,
       val3 actual ctx)::ctx)
10.  | Fn (formal, body) -> Fn(formal, body);;
```

```
val3 (Apply( Fn("z", Plus(Var "z",Const 2)),Const 3)) [];;
```

1. What is the corresponding F# for the above?
2. What is `Formal` bound to?
3. What is `Body` bound to?
4. What is `Actual` bound to?
5. What is `Context` bound to?
6. Trace the execution.
7. What is the result?

Language Three: F# Interpreter

To line, same as for Language Two

```
let rec val3 exp context = match exp with
  | Const x -> Const x
  | Var V -> lookup V context
  | Plus (Const x, Const y) -> Const (x+y)
  | Plus (x, y) -> val3 (Plus(val3 x context, val3 y context))
      context
  | Times (Const x, Const y) -> Const (x*y)
  | Times (x, y) -> val3 (Times(val3 x context, val3 y context))
      context
  | Let (V, exp1, exp2) -> val3 exp2 ((V, val3 exp1
      context)::context)
```

```
  | Fn (formal, body) -> Fn(formal, body)
  | Apply (Fn(formal, body), actual) -> val3 body ((formal,
      val3 actual context)::context)
  | Apply (Var v, actual) -> val3 (Apply (val3 (Var v) context,
      actual)) context;;
```

Exercise 4

1. Give the Language 3 syntax for:

```
(fn x => x) 3;
```

2. Give the F# for:

```
val3 (Let("f",
          Fn("n",
             Times(Var "n",
                   Var "n")),
          Apply(Var "f",
                Const 3)))
     [];;
```

Exercise 4

```
1. let rec val3 exp context = match exp with
2.    | Const x -> Const x
3.    | Var V -> lookup V context
4.    | Plus (Const x, Const y) -> Const (x+y)
5.    | Plus (x, y) -> val3 (Plus(val3 x context, val3 y context))
6.      context
7.    | Times (Const x, Const y) -> Const (x*y)
8.    | Times (x, y) -> val3 (Times(val3 x context, val3 y context))
        context
9.    | Let (V, exp1, exp2) -> val3 exp2 ((V, val3 exp1
        context)::context)
12.   | Fn (formal, body) -> Fn(formal, body)
10.   | Apply (Fn(formal, body), actual) -> val3 body ((formal,
        val3 actual context)::context)
11.   | Apply (Var v, actual) -> val3 (Apply (val3 (Var v) context,
         actual)) context;;
```

3. List the first 3 line(s) in the reduction of:

```
val3 (Apply( Fn("z",
                  Plus(Var "z", Const 2)),
             Const 3))
       [];;
```

4. What are the values of **Formal, Body, Actual, Context?**

Language Three Natural Semantics

$$\langle \mathbf{fn}(x, E), C \rangle \rightarrow (x, E)$$

States that **fn(x,E)** in context C reduces to **(x,E)**

```
val3 (Fn(Formal, Body), _) = Fn(Formal, Body)
```

Language Three Natural Semantics

$$\frac{\langle E_1, C\rangle \to (x, E_3) \quad \langle E_2, C\rangle \to v_1 \quad \langle E_3, bind(x, v_1) :: C\rangle \to v_2}{\langle \texttt{apply}(E_1, E_2), C\rangle \to v_2}$$

Rule for **apply**

Conditions

Expression **E1** in context **C** reduces to **(x,E3)**

Expression **E2** in context **C** reduces to value **v1**

Expression **E3** in context where variable **x** bound to value **v1** appended to context **C** reduces to value **v2**

Conclusion

apply(E1,E2) in context **C** reduces to **v2**

```
| Apply( Fn(Formal, Body), Actual) ->
    val3( Body,(Formal,val3(Actual, Context))::Context)
| Apply( Var(V), Actual) ->
    val3( Apply( val3(Var(V), Context), Actual), Context);
```

Question

- Value of this Language Three program depends on whether scoping is static or dynamic.
- What is the value using dynamic and static scoping?

```
let x = 1 in
  let f = fn n => n + x in
    let x = 2 in
      f 0
```

Answer

- Dynamic: 2
- Static: 1

Language Three

```
val3 (Let("x", Const 1,
        Let("f", Fn("n", Plus(Var "n", Var "x")),
          Let("x", Const(2),
            Apply(Var "f", Const 0)))));;

val it : AST = Const 2
```

F#

```
let x = 1 in
  let f = fun n -> n + x in
    let x = 2 in
      f 0;;

val it : int = 1
```

Language Three has dynamic scoping.

Dynamic Scoping

- We defined dynamic scoping by adding each binding to the front of the execution context
- Probably not a good idea:
  - We have seen its drawbacks: difficult to implement efficiently, makes large complex scopes
  - Most modern languages use static scoping
- How to fix this so that Language Three uses static scoping?

Language Three Natural Semantics, Dynamic Scoping

$$\frac{\langle E_1, C\rangle \to v_1 \quad \langle E_2, C\rangle \to v_2}{\langle \mathtt{plus}(E_1, E_2), C\rangle \to v_1 + v_2}$$

$$\frac{\langle E_1, C\rangle \to v_1 \quad \langle E_2, C\rangle \to v_2}{\langle \mathtt{times}(E_1, E_2), C\rangle \to v_1 \times v_2}$$

$$\langle \mathtt{const}(n), C\rangle \to eval(n)$$

$$\langle \mathtt{var}(v), C\rangle \to lookup(C, v)$$

$$\langle \mathtt{fn}(x, E), C\rangle \to (x, E)$$

$$\frac{\langle E_1, C\rangle \to v_1 \quad \langle E_2, bind(x, v_1) :: C\rangle \to v_2}{\langle \mathtt{let}(x, E_1, E_2), C\rangle \to v_2}$$

$$\frac{\langle E_1, C\rangle \to (x, E_3) \quad \langle E_2, C\rangle \to v_1 \quad \langle E_3, bind(x, v_1) :: C\rangle \to v_2}{\langle \mathtt{apply}(E_1, E_2), C\rangle \to v_2}$$

Language Three Natural Semantics, Static Scoping

Dynamic $\langle \mathbf{fn}(x, E), C \rangle \rightarrow (x, E)$

Static $\langle \mathbf{fn}(x, E), C \rangle \rightarrow (x, E, C)$

Dynamic $$\frac{\langle E_1, C \rangle \rightarrow (x, E_3) \quad \langle E_2, C \rangle \rightarrow v_1 \quad \langle E_3, bind(x, v_1) :: C \rangle \rightarrow v_2}{\langle \mathbf{apply}(E_1, E_2), C \rangle \rightarrow v_2}$$

Static $$\frac{\langle E_1, C \rangle \rightarrow (x, E_3, C') \quad \langle E_2, C \rangle \rightarrow v_1 \quad \langle E_3, bind(x, v_1) :: C' \rangle \rightarrow v_2}{\langle \mathbf{apply}(E_1, E_2), C \rangle \rightarrow v_2}$$

About Errors

- Language Three now has more than one type, so we can have type errors: **`1 1`**

```
val3 (Apply(Const 1, Const 1)) [];;
Error - Match failure
```

- Similarly, the natural semantics gives no *v* for which

  <**`apply(const(1),const(1))`**, *nil*>→ *v*

More Errors

- In the dynamic-scoping version, we can also have programs that run forever:

```
let f = fun x -> f x in f 1
```

- Interpreter runs forever on this
- Natural semantics does not run forever—does not *run* at all—it just defines no result for the program