

# Defining Language THREE

- We add the following to Language TWO
  - ML-style function values
  - ML-style function application

# THREE: Syntax

THREE:

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

- A subset of ML expressions
- This grammar is unambiguous
- Function application has highest precedence
- A sample Language Three expression:

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

# THREE: 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 `Formal` parameter and `Body`

- So for the Language Three program

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

we have this AST:

```
apply(fn(x, times(var(x), var(x))),  
      const(3))
```

# Representing Functions

- A representation for functions:
  - `fval (Formal , Body)`
  - `Formal` is the formal parameter variable
  - `Body` is the unevaluated function body
- So the AST node `fn (Formal , Body)` evaluates to `fval (Formal , Body)`
- (Why not just use the AST node itself to represent the function? You'll see...)

# THREE:Prolog Interpreter

```
val3 (plus (X,Y) ,Context,Value) :- ...  
val3 (times (X,Y) ,Context,Value) :- ...    Same as for  
val3 (const (X) ,_,X) .                    Language Two  
val3 (var (X) ,Context,Value) :- ...  
val3 (let (X,Exp1 ,Exp2) ,Context,Value2) :- ...
```

---

```
val3 (fn (Formal,Body) ,_, fval (Formal,Body) ) .  
val3 (apply (Function,Actual) ,Context,Value) :-  
    val3 (Function,Context, fval (Formal,Body) ) ,  
    val3 (Actual,Context, ParamValue) ,  
    val3 (Body, [bind (Formal, ParamValue) |Context] ,Value) .
```

```
?- val3 (apply (fn (x) times (var (x) , var (x)) ,  
|           const (3)) ,  
|           nil , X) .
```

```
X = 9
```

```
Yes
```

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

# Question

- What should the value of this Language Three program be?

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

- Depends on whether scoping is static or dynamic

```
?- val3(let(x, const(1) ,
|         let(f, fn(n, plus(var(n) , var(x))) ,
|           let(x, const(2) ,
|             apply(var(f) , const(0)))) ,
|         nil, X) .
```

X = 2

Yes

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

*Oops—we defined Language Three  
with dynamic scoping!*

# Dynamic Scoping

- We got dynamic scoping
- 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 can we fix this so that Language Three uses static scoping?

# Representing Functions, Again

- Add context to function representation:
  - `fval (Formal, Body, Context)`
  - `Formal` is the formal parameter variable
  - `Body` is the unevaluated function body
  - `Context` is the context to use when calling it
- So the AST node `fn (Formal, Body)` evaluated in `Context`, produces to `fval (Formal, Body, Context)`
- `Context` works as a *nesting link* (Chapter 12)

# Three:Prolog Interpreter (Static Scoping)

```
val3 (fn (Formal , Body) , _ , fval (Formal , Body) ) .
```



```
val3 (fn (Formal , Body) , Context , fval (Formal , Body , Context) ) .
```

```
val3 (apply (Function , Actual) , Context , Value) :-  
  val3 (Function , Context , fval (Formal , Body) ) ,  
  val3 (Actual , Context , ParamValue) ,  
  val3 (Body , bind (Formal , ParamValue , Context) , Value) .
```



```
val3 (apply (Function , Actual) , Context , Value) :-  
  val3 (Function , Context , fval (Formal , Body , Nesting) ) ,  
  val3 (Actual , Context , ParamValue) ,  
  val3 (Body , [bind (Formal , ParamValue) | Nesting] , Value) .
```

```
?- val3(let(x,const(1) ,
|         let(f,fn(n,plus(var(n) ,var(x) ) ) ,
|           let(x,const(2) ,
|             apply(var(f) ,const(0) ) ) ) ) ,
|         nil,X) .
```

X = 1

Yes

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

*That's better: static scoping!*

```
?- val3(let(f,fn(x,let(g,fn(y,plus(var(y),var(x))),
|               var(g))),
|               apply(apply(var(f),const(1)),const(2))),
|               nil,X).
```

X = 3

Yes

```
let
  val f = fn x =>
    let val g = fn y => y+x in
      g
    end
in
  f 1 2
end
```

*Handles ML-style higher  
order functions.*

# Three: Natural Semantics (Dynamic Scoping)

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

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

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

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

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

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

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

# THREE: Natural Semantics (Static Scoping)

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



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

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



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