Programming Languages Using PLCC

What a programming course that uses PLCC might look like, and what's possible

Common Approaches to Programming Languages

- Tour of languages:
  - all breadth, no depth
  - Language *du jour* changes (but the fundamentals don't)
- Tour of language constructs (with example languages):
  - A little more depth, more focus on language fundamentals
- Build compiler/interpreter (assembly and/or a simulated machine):
  - Requires more details than is necessary, which means less time for the fundamentals
- Build language implementations using industrial strength tools lex, yacc, javacc, antlr
  - Generated code is complicated, so tools must be used without understanding

PLCC Approach

- Design and implement languages using PLCC
    - Tokens are defined using regular expressions
    - Syntactic rules defined using BNF
    - Generated parser is LL(1)
        - Comprehensible Left-Recursive parser with a 1-token look ahead.
        - Token matching is equally explainable: longest, first match.
    - Java is used for defining semantics
- For each construct
    - Syntax defined in PLCC's grammar language (see above)
    - Semantics is defined in BNF
    - Semantics defined in Java

Students gain a deep understanding of languages without having to dive into assembly or work with tools that hide too much details.

Learning Goals

- Know some of the history of languages (what's been tried).
- Learn and evaluate new language constructs more easily.
- Become familiar with some of the common tools for language design and implementation.
- Become better developers through a deeper understanding of programming languages.

Learning Objectives

- Differentiate between lexical, syntactic, and semantic analysis.
- Use regex to define tokens.
- Use BNF to define small language constructs.
- Use/implement operations that evaluate across a parse tree
- Draw environment diagrams that accurately illustrate the environment that a given expression will evaluate within.
- Accurately evaluate expressions containing nested static scopes, closures, recursion, selection, and polymorphism, all within the context of different call-semantics.
- Identify and explain the risks of side effecting code and how to mitigate them.

Context

- Semester: 15 week course (+1 for finals)
- 2-3 sessions per week, at least 150 minutes per week
- 3 exams, each focused on material since the last exam
- 6-12 homeworks, depending on size
- Upper-level in CS

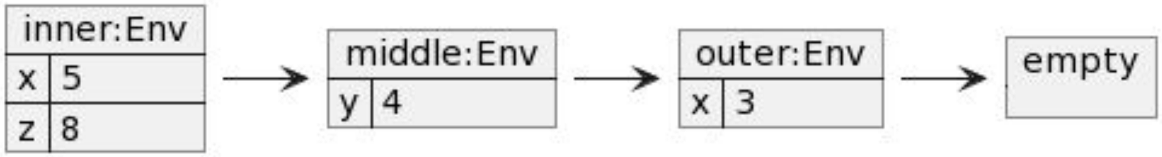
Schedule

- Weeks 1-2: Foundations
  - Lexical Analysis
  - Syntactic Analysis
  - Semantic Analysis
  - Environments
- Weeks 3-6: Build a functional language
  - Literals, symbols, let-expressions
    - Static scoping
  - If expression
  - Functions
    - Higher-ordered
    - Closures
    - Static scoping
    - Letrec
- Weeks 7-10: Call semantics
  - Side-effects
  - Pass by value
  - Pass by reference
  - Pass by name
  - Pass by need
- Week 10-11: Static Type System
- Week 11-13: Object-Oriented Language
- Week 14-15: Topics
  - Properties
  - Concurrency
  - Logic-Based Language
  - etc.

Weeks 1-2: Lexical, Syntactic, and Semantic Analysis

- Lexical analysis
  - Tokens
  - Regex
  - Longest, first match
- Syntactic analysis
  - BNF
  - Repetition Rule
  - LL(1) Parser
- Semantic analysis
  - Parse tree
  - Java snippets to walk tree

Weeks 1-2: Environments

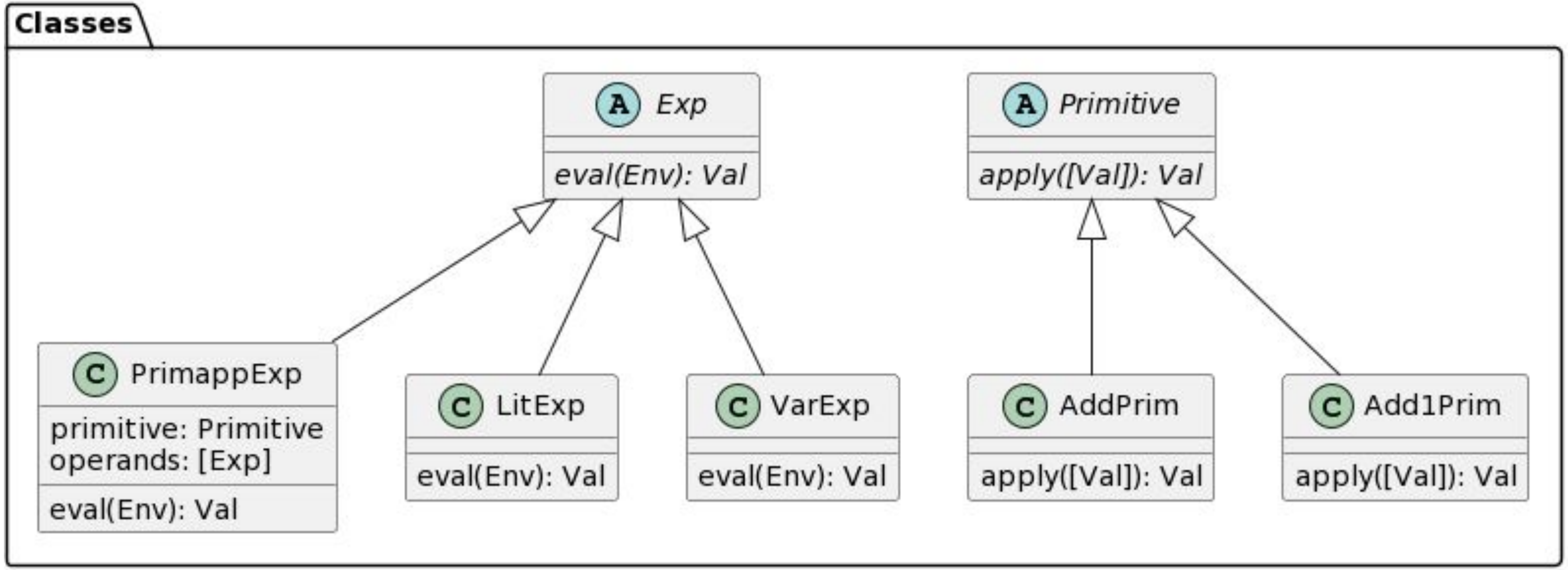


Env.apply() semantics:

- inner.apply("x"); // 5
- inner.apply("y"); // 4
- inner.apply("z"); // 8
- middle.apply("x"); // 3
- middle.apply("y"); // 4
- middle.apply("z"); // Exception

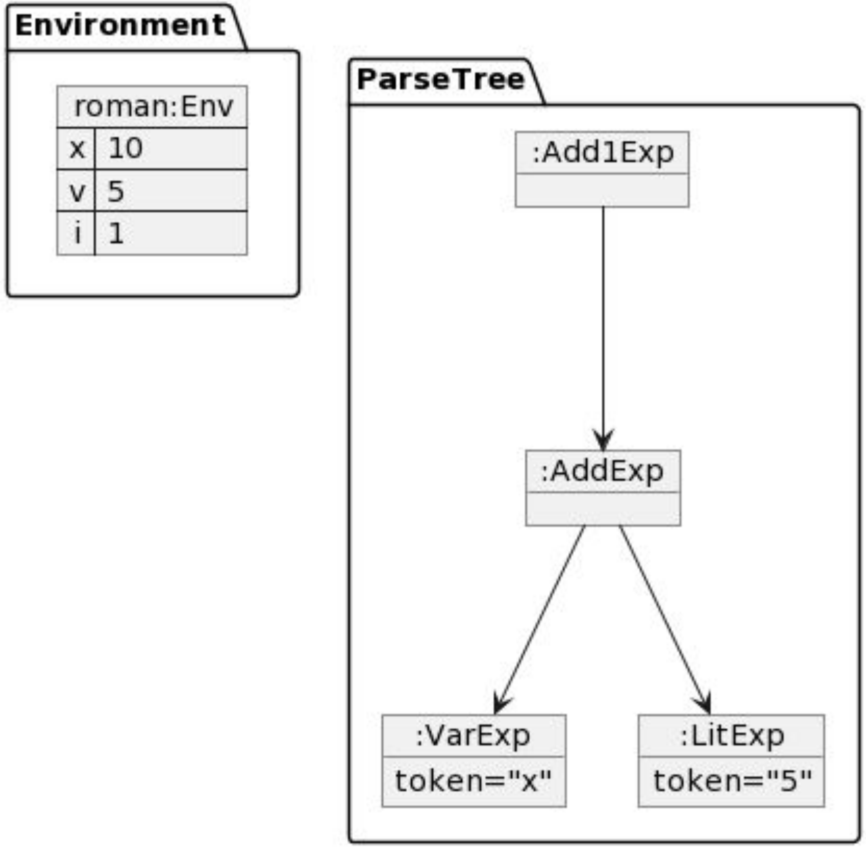
Weeks 3-6: Define functional language: V0-V1

```
<prog> ::= <exp>
<exp>:VarExp ::= <VAR>
<exp>:LitExp ::= <INT>
<exp>:PrimappExp ::= <prim> LPAREN <operands> RPAREN
```



Weeks 3-6: Define functional language: V0-V1

```
add1( +(x,5) )   % => 15
```

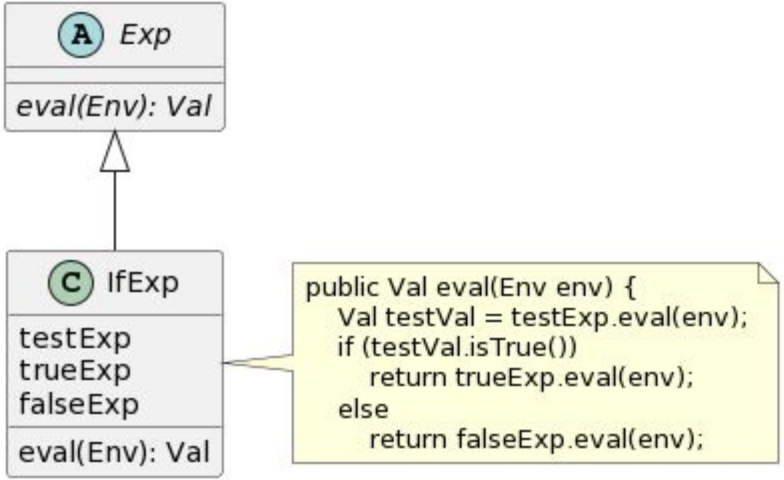


Note that, during semantic analysis, code in the parse tree is executed that searches the environment (not in parse tree) for the value of **x**.

Weeks 3-6: Define functional language: V2

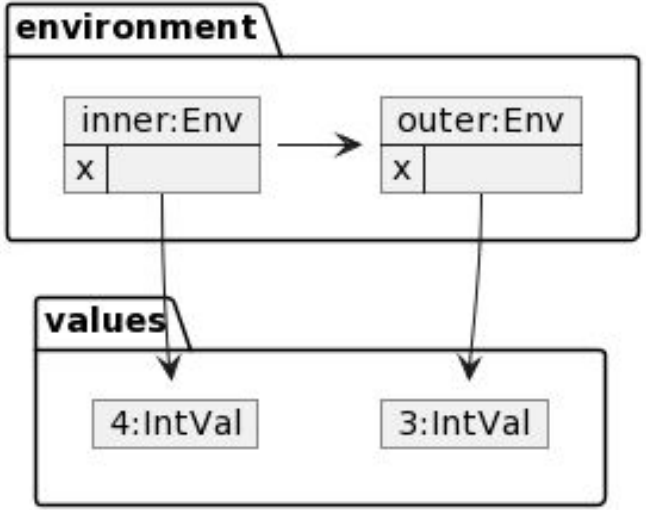
```
<exp>:IfExp ::=
  IF <exp>testExp THEN <exp>trueExp ELSE <exp>falseExp
```

```
if 1 then 3 else 4    % => 3
if 0 then 3 else 4    % => 4
```



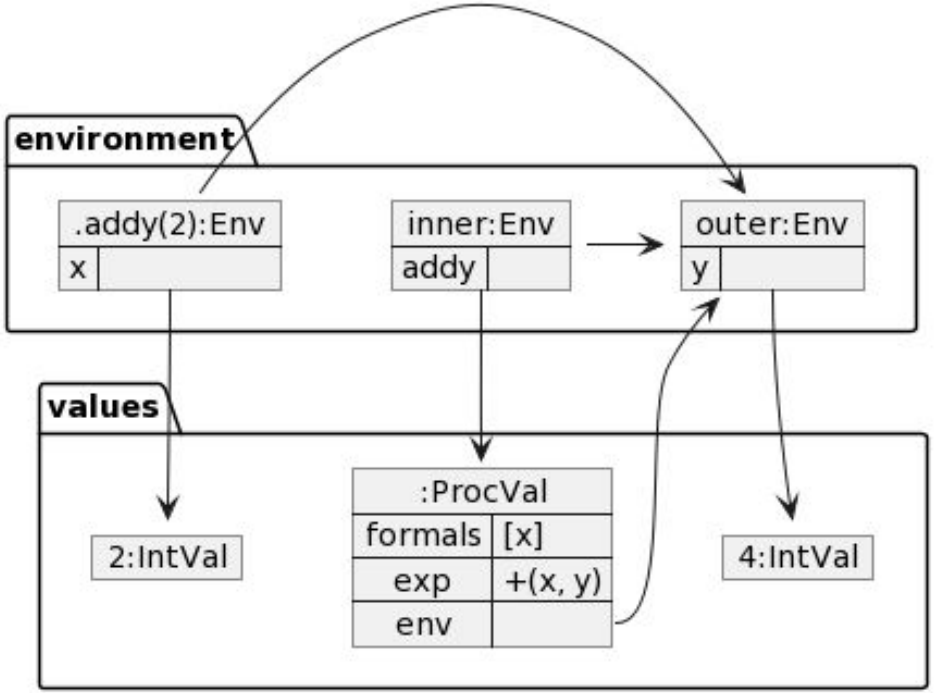
Weeks 3-6: Define functional language: V3

```
let          % outer
    x = 3
in
    let      % inner
        x = add1(x)
    in
        +(x, x) % => ?
```



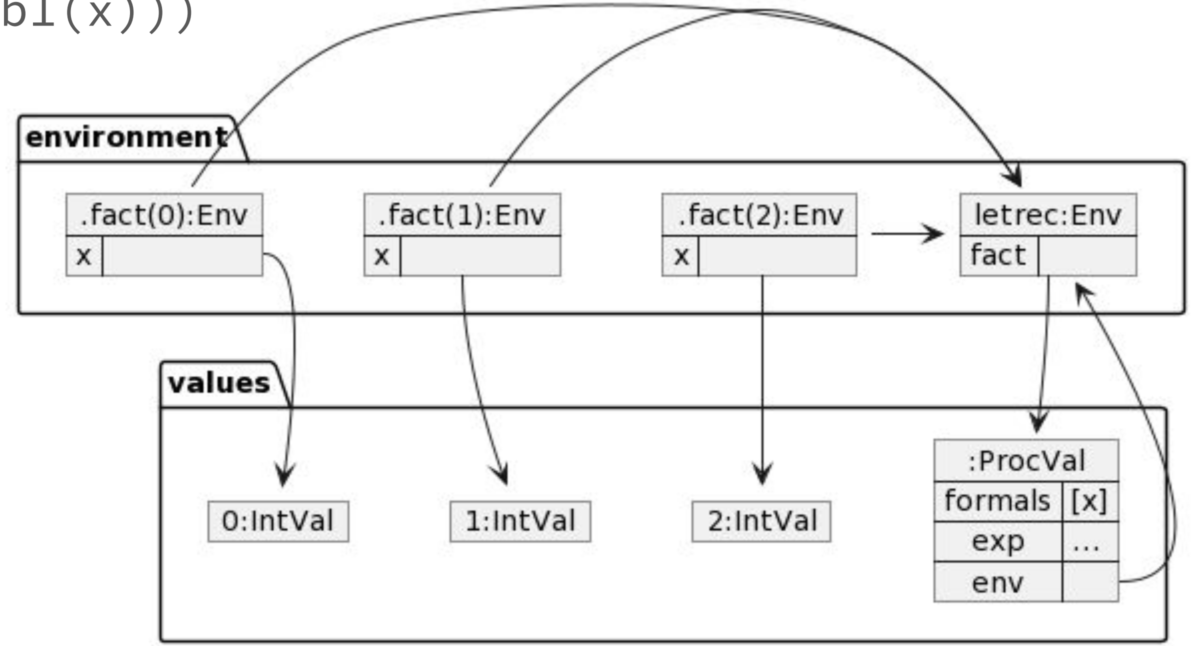
Weeks 3-6: Define functional language: V4

```
let
    y = 4
in
    let
        addy = proc(x) +(x, y)
    in
        . addy(2)     % => 6
```



Weeks 3-6: Define functional language: V5-V6

```
letrec
  fact = proc(x)
    if zero?(x) then 1
    else *(x, .fact(sub1(x)))
in
  .fact(2)
```



V0-V6: Summary

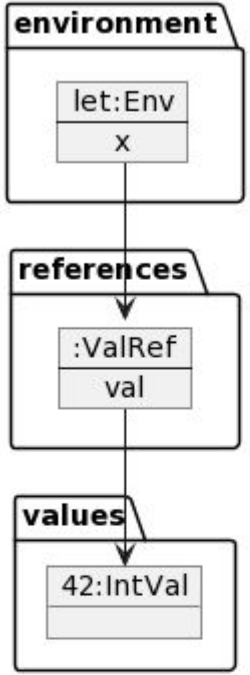
- Functional, expression language
- Closures
- Static-scoping
- Supports recursion
- Selection through if expressions
- Integer primitives
- Boolean values defined as 0 is False, non-0 is True
- Two types: integers and procs

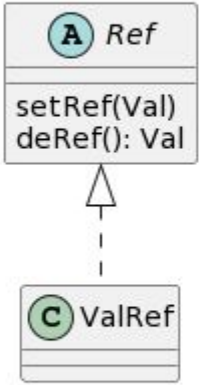
Students practice understanding language concepts in terms of regex, BNF, semantics in Java, parse trees, and environment diagrams.

Weeks 7-10: Call Semantics: SET

```
let
    x = 42
in {
    set x = +(x, 2) ;
    x
}
```

Symbols are now bound to ***mutable*** Refs instead of Vals.
Enables side effects.

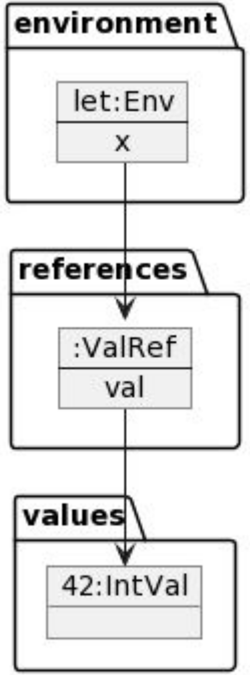


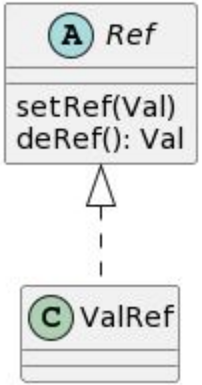


Weeks 7-10: Call Semantics: SET

```
let
    x = 42
in {
    set x = +(x, 2) ;
    x
}
```

xRef.deRef()

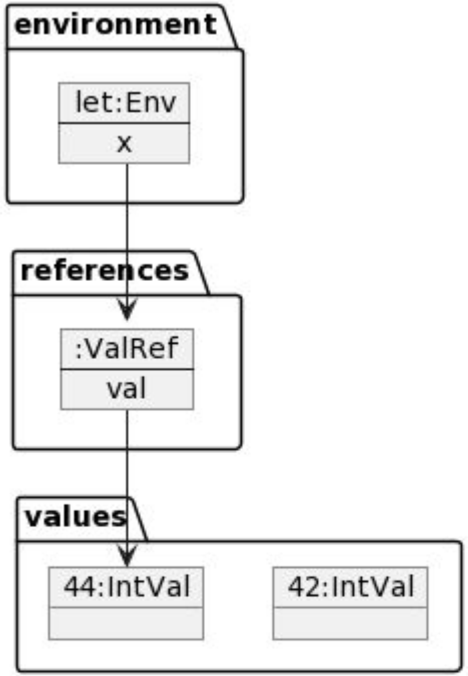


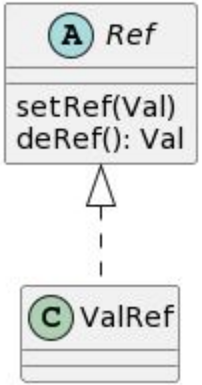


Weeks 7-10: Call Semantics: SET

```
let
    x = 42
in {
    set x = +(x, 2) ;
    x
}
```

xRef.setRef(val);

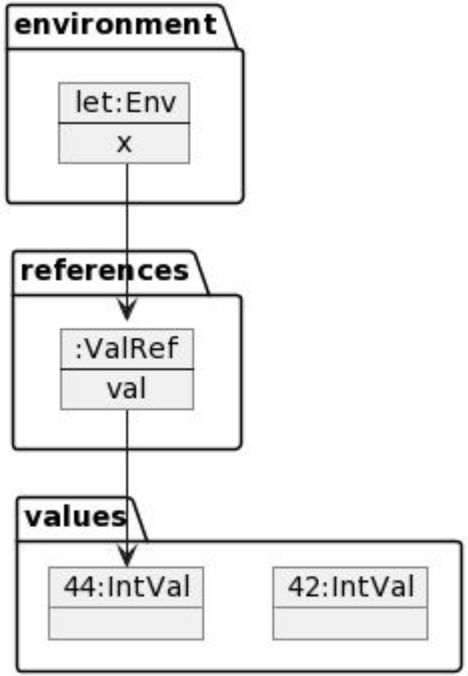


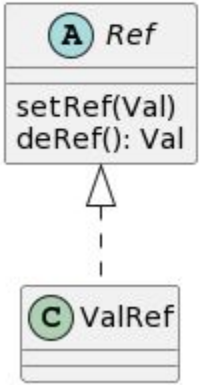


Weeks 7-10: Call Semantics: SET

```
let
    x = 42
in {
    set x = +(x, 2) ;
    x
}
```

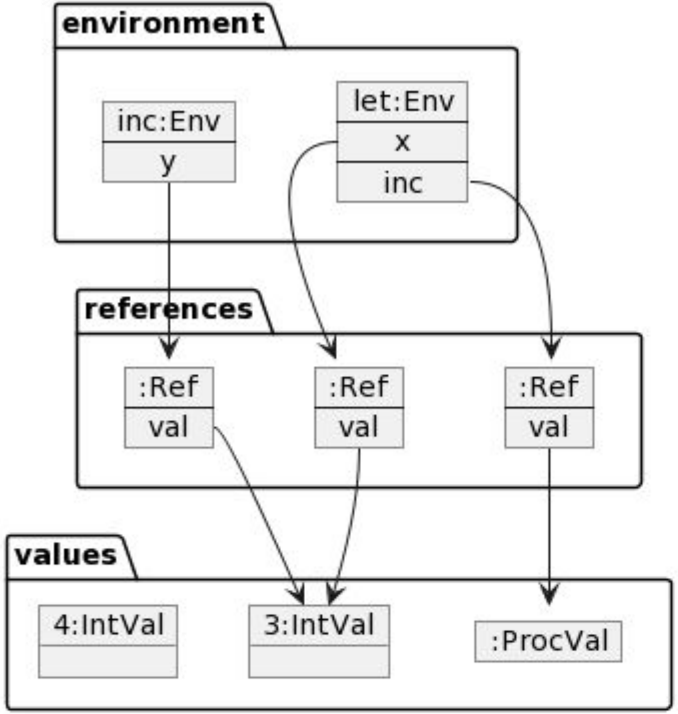
xRef.deRef()





Weeks 7-10: SET - Pass-by-Value

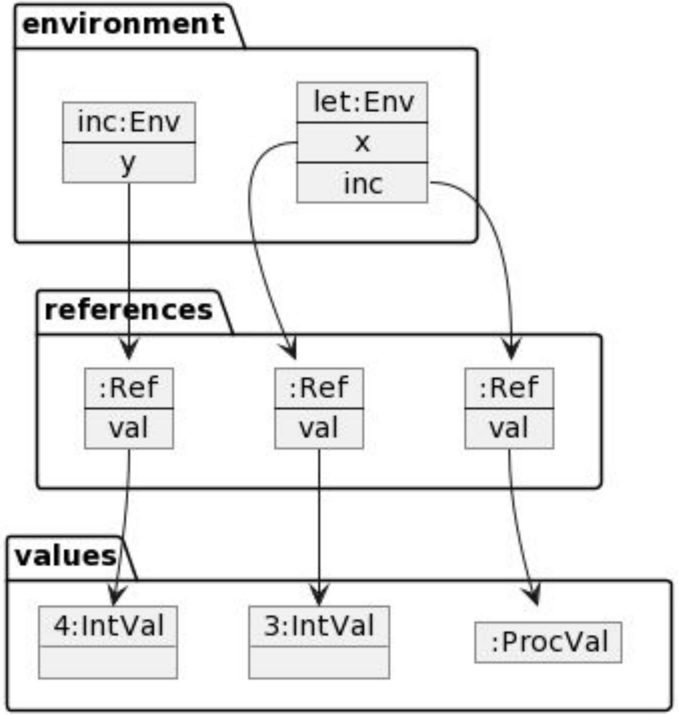
```
let
     x = 3
     inc = proc(y) set y = add1(y)
in {
    . inc(x) ;
    x % => ??
}
```



Just after the call, **before** inc's body evals.

Weeks 7-10: SET - Pass-by-Value

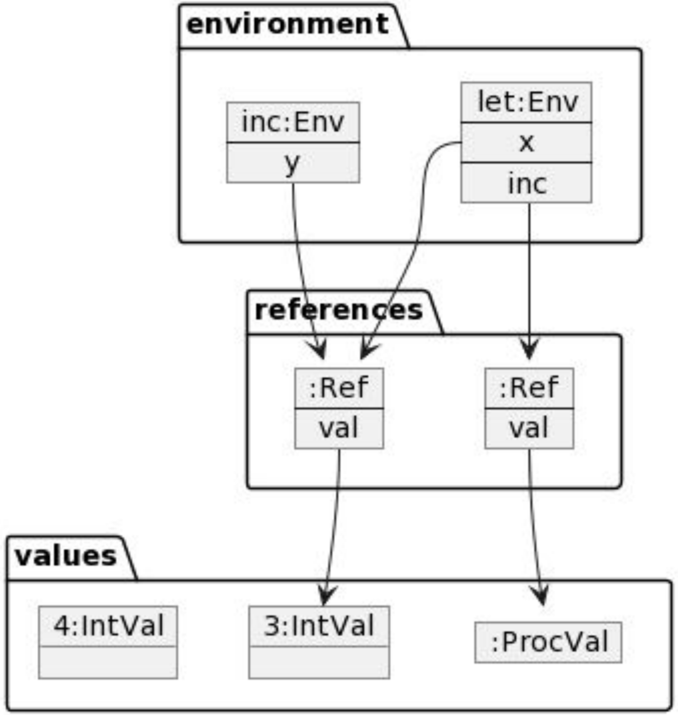
```
let
     x = 3
     inc = proc(y) set y = add1(y)
in {
    . inc(x) ;
    x % => 3
}
```



After .inc(x)

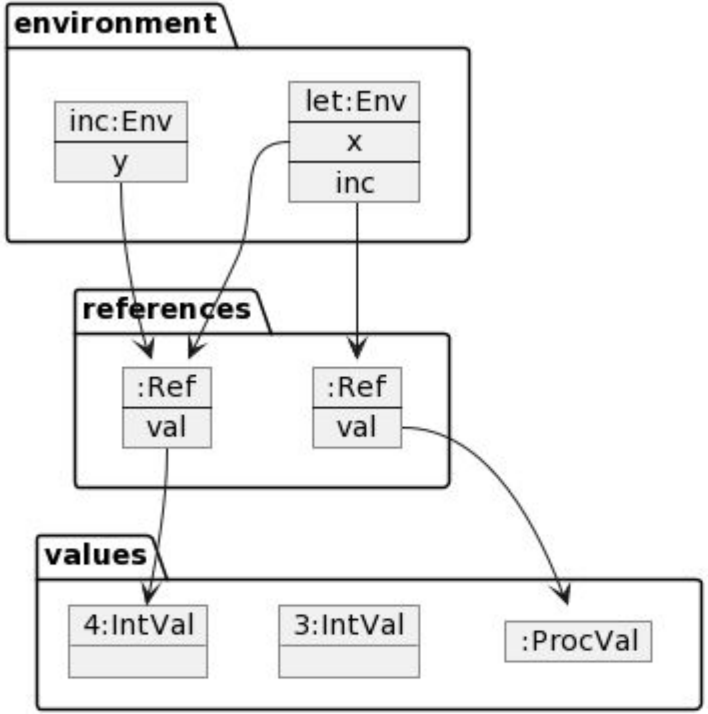
Weeks 7-10: REF - Pass-by-Reference

```
let
      x = 3
      inc = proc(y) set y = add1(y)
in {
     . inc(x) ;
     x % => ??
}
```



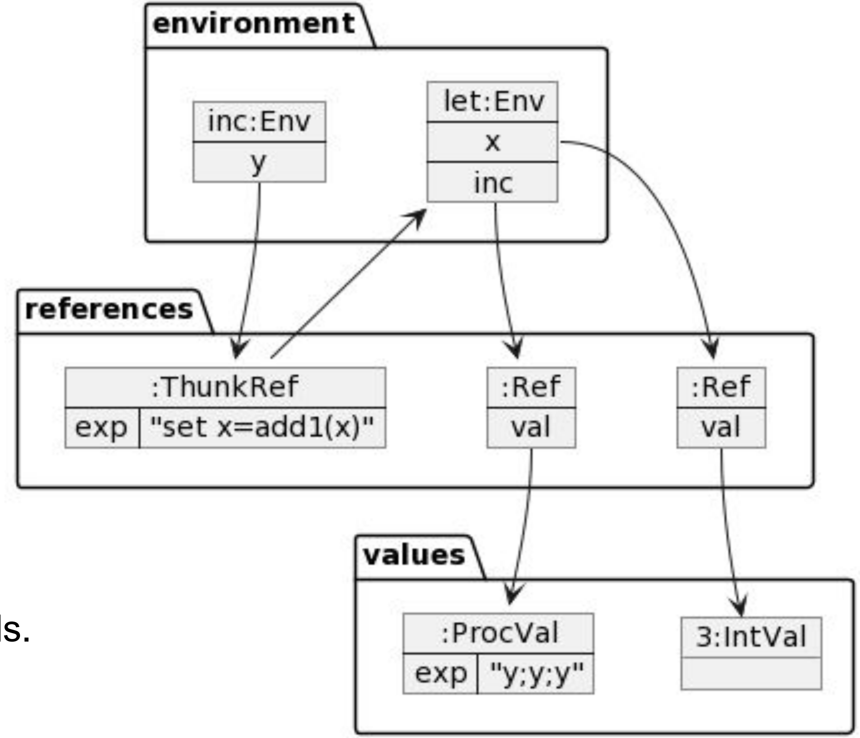
Weeks 7-10: REF - Pass-by-Reference

```
let
     x = 3
     inc = proc(y) set y = add1(y)
in {
    . inc(x) ;
    x % => 4
}
```



Weeks 7-10: NAME - Pass-by-Name (thunk)

```
let
    x = 3
    inc = proc(y) { y; y; y }
in {
   . inc(set x = add1(x)) ;
   x % => ??
}
```

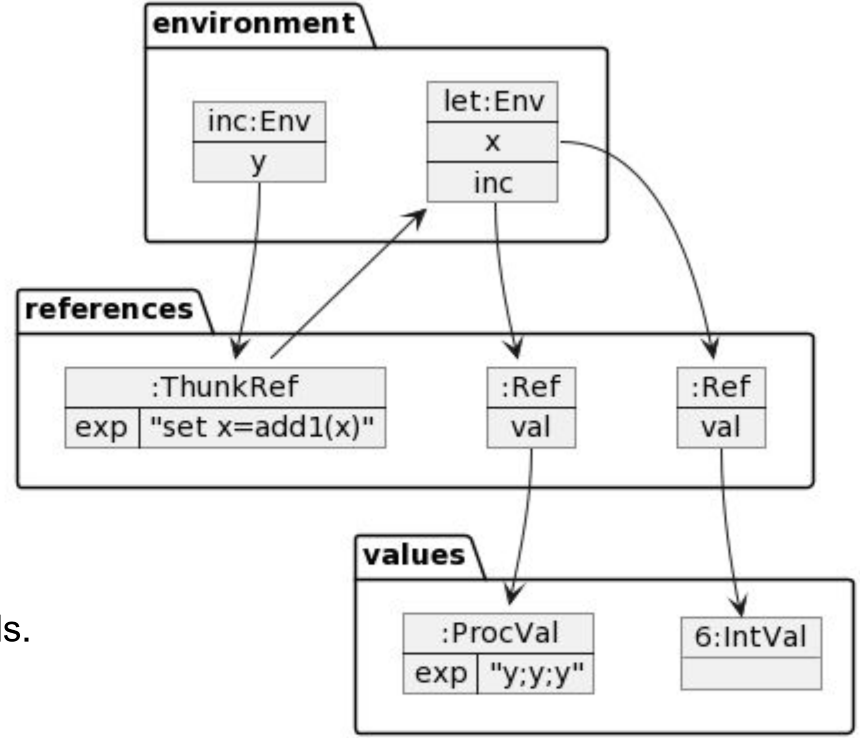


A thunk is similar to a proc without formals.
But it's a Ref.
thunk.deRef() evals its exp in calling env.

Weeks 7-10: NAME - Pass-by-Name (thunk)

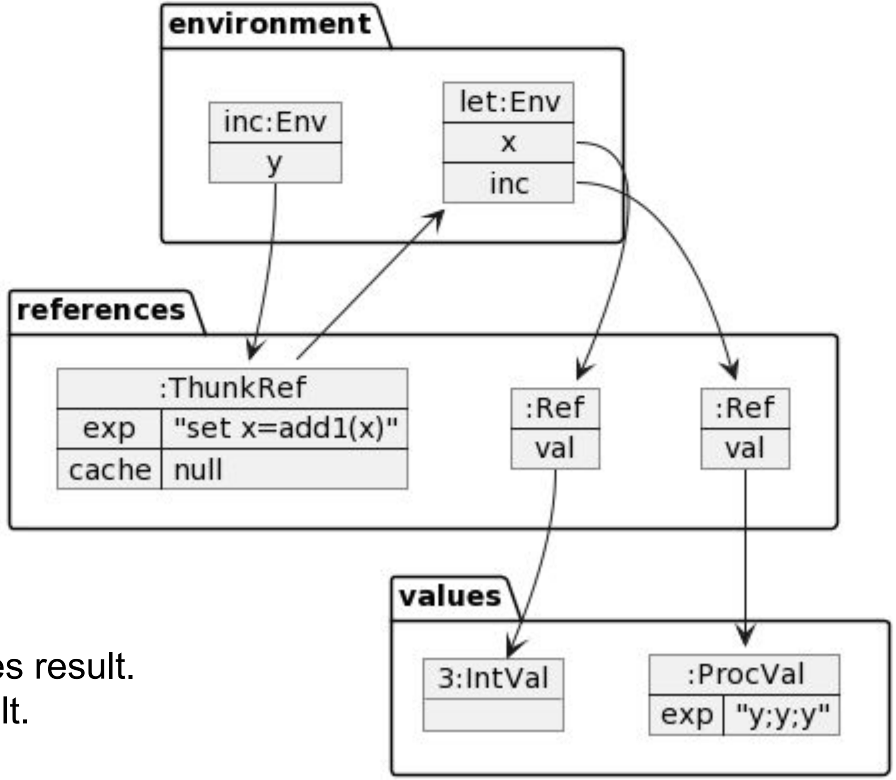
```
let
      x = 3
      inc = proc(y) { y; y; y }
in {
   . inc(set x = add1(x)) ;
   x % => 6
}
```

A thunk is similar to a proc without formals.
But it's a Ref.
thunk.deRef() evals its exp in calling env.



Weeks 7-10: NEED - Pass-by-Need

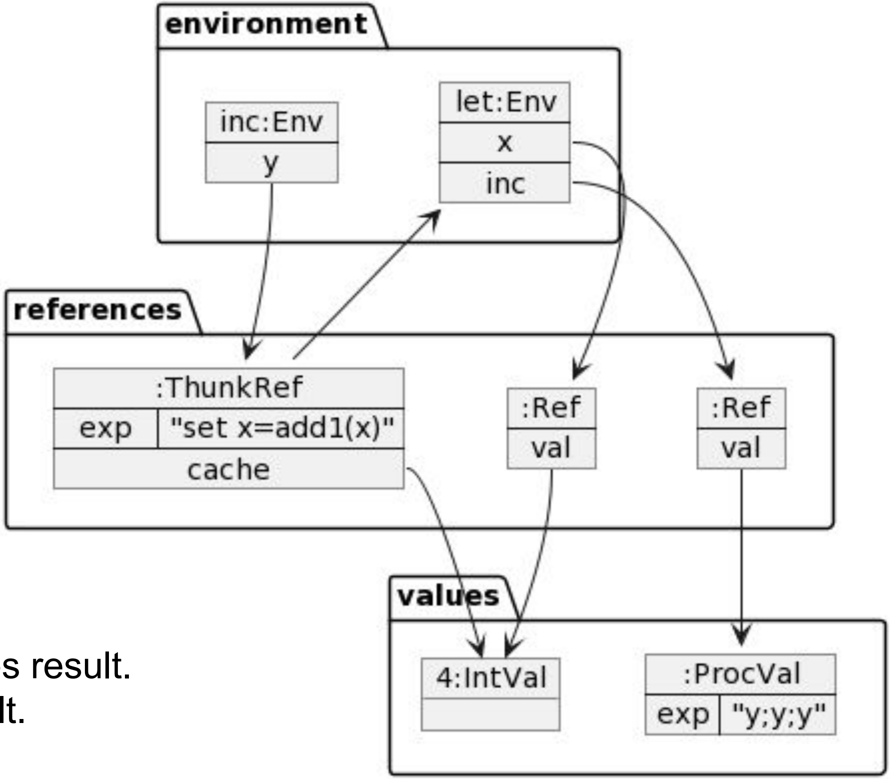
```
let
     x = 3
     inc = proc(y) { y; y; y }
in {
    . inc(set x = add1(x)) ;
    x % => ??
}
```



On first deRef(), thunk evaluates exp and caches result.
Subsequent deRef(), thunk returns cached result.
Lazy evaluation.

Weeks 7-10: NEED - Pass-by-Need

```
let
     x = 3
     inc = proc(y) { y; y; y }
in {
    . inc(set x = add1(x)) ;
    x % => 4
}
```



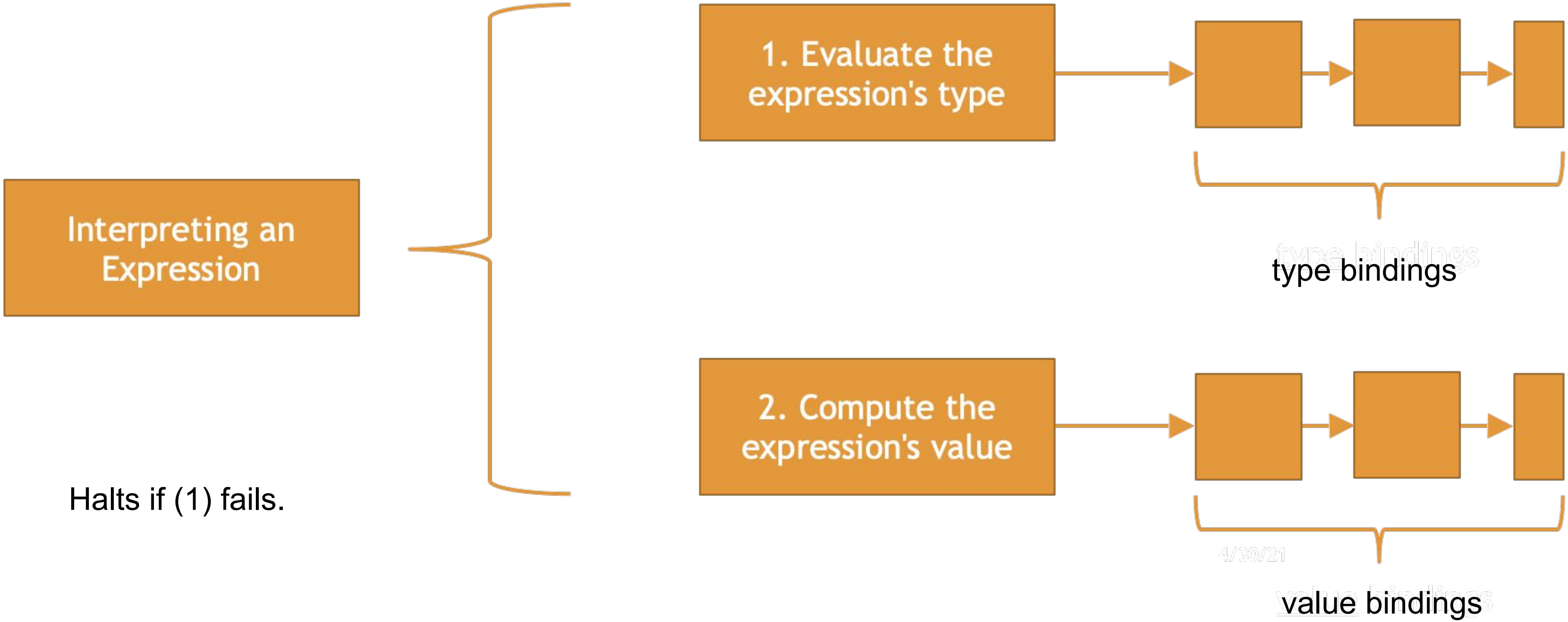
On first deRef(), thunk evaluates exp and caches result.
Subsequent deRef(), thunk returns cached result.
Lazy evaluation.

Weeks 10-11: TYPE0 and TYPE1: Static Type Checking

```
letrec
    fact = proc(x: int): int
        if zero?(x) then 1
        else *(x, .fact(sub1(x)))
in
    .fact(5)
```

```
let
    twice = proc(f: [int => int], x: int): int {
        f(f(x))
    }
    add5 = proc(x: int): int  +(5, x)
in
    twice(add5, 10)   % => 20
```

Type Checking Design



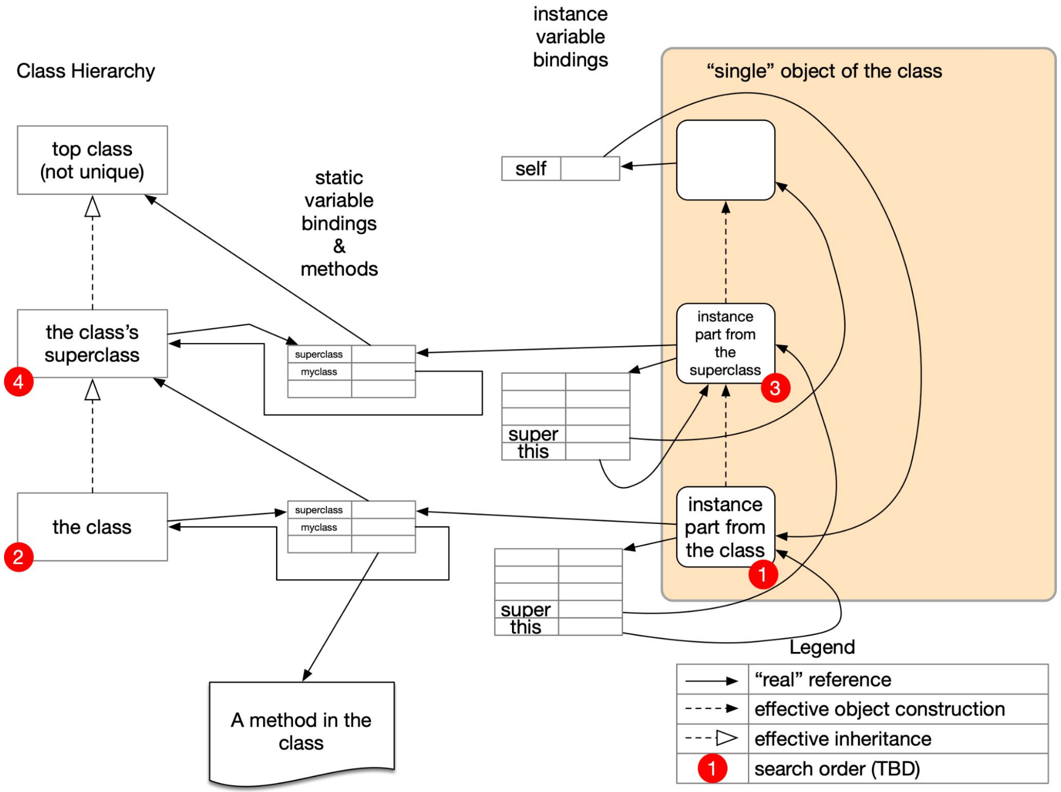
Halts if (1) fails.

Weeks 11-14: OBJ (shown) and PROP (not shown)

```
define shape = class
   method area = proc() -1
end

define rectangle = class extends shape
   field lenn % length
   field widd % width

   method init = proc(lenn,widd) {set <self>lenn=lenn ; set <self>widd=widd ; self}
   method area = proc() *(lenn,widd)
end

define s = new shape
define r = .<new rectangle>init(4,5)
.<r>area() % => 20
.<s>area() % => -1
```

What is an Object?
An Environment!!!!

Weeks 11-14: OBJ: Making objects
"Environments, Environments, Environments"



ABC: Logic-Based-Language

PLCC implementation of ABCDatalog <http://abcdatalog.seas.harvard.edu/>
a subset of Prolog

```
bear(fuzzy).
bear(wuzzy).
bear(X)?    % yields "bear(fuzzy)" and "bear(wuzzy)"
```

Course Materials

https://github.com/ourPLCC/course/
https://plcc.pithon.net/

Let's look at what's there...