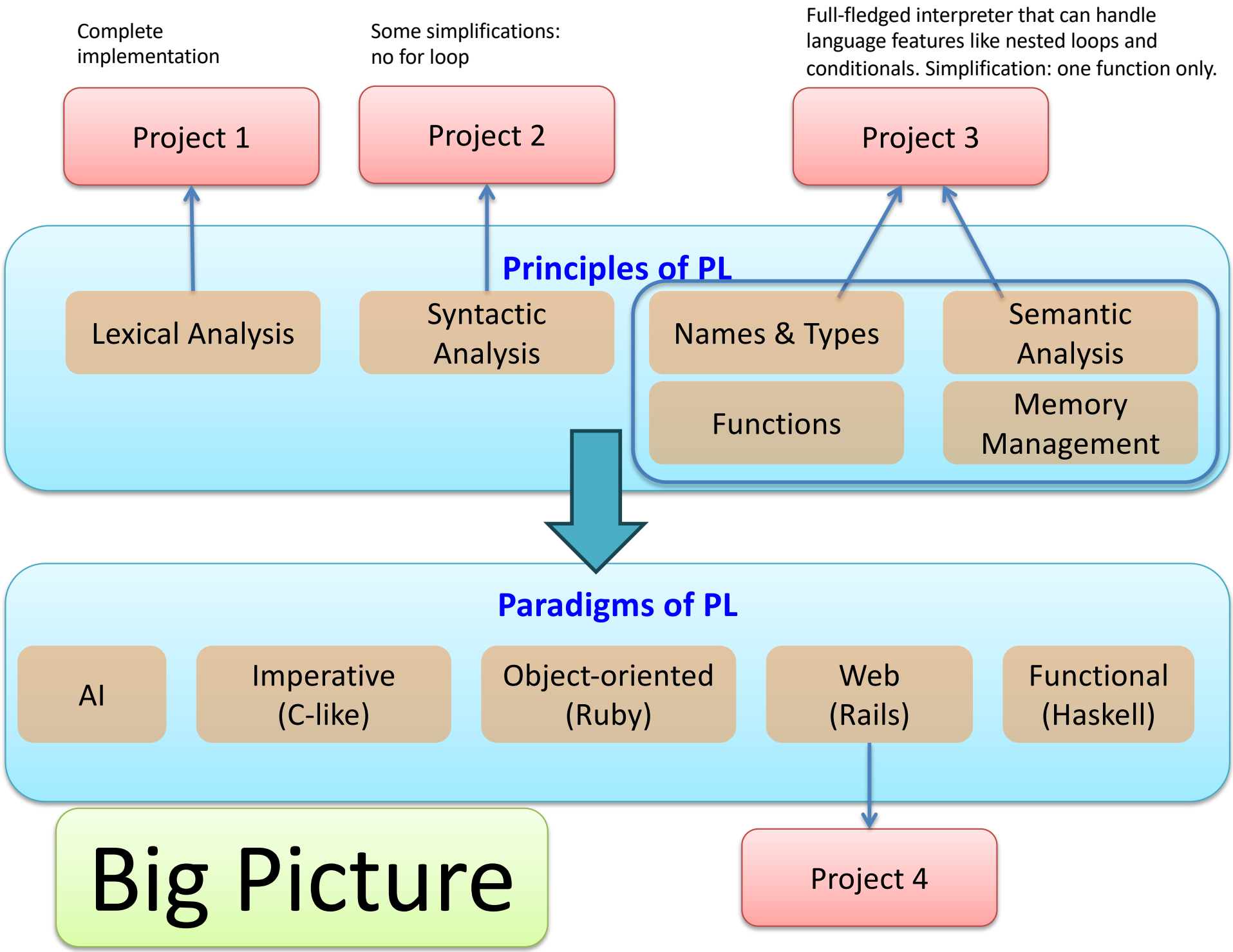
CSCI-2320
Principles of Programming Languages

Names

Ref: Tucker-Noonan [Ch. 4]


Complete implementation
Some simplifications: no for loop
Full-fledged interpreter that can handle language features like nested loops and conditionals. Simplification: one function only.
Project 1
Project 2
Project 3
Principles of PL
Lexical Analysis
Syntactic Analysis
Names & Types
Semantic Analysis
Functions
Memory Management
Paradigms of PL
AI
Imperative (C-like)
Object-oriented (Ruby)
Web (Rails)
Functional (Haskell)
Big Picture
Project 4

Self study from the textbook

- Pointers and L-value and R-value
- Forward reference
- Visibility vs. lifetime

Syntactic Issues

- Lexical rules for names
- Collection of reserved words or keywords
- Case sensitivity
- Early languages: no
- C-like: yes
- PHP: partly yes, partly no

Binding

L-value vs. R-value

L-value: use of a variable name to denote its address.

Example: x = …

R-value: use of a variable name to denote its value.

Example: … = … x …

What's the first PL to clearly distinguish L- and R-value?

Self study: Example C code to understand pointers and L-value and R-value (posted on Canvas)

```c
#include <stdio.h>
int main()
{
    int x = 50, y = 10; //50 is stored in l-value of x. So, r-value of x = 50.
    int *p = &y; //r-value of p is l-value of y (i.e., y's address)
    *p = x; //r-value of r-value of p is r-value of x
    x = *p + 100; //r-value of r-value of p + 100 is assigned to l-value of x
    printf("p = %p, *p = %d\nx = %d, y = %d\n", p, *p, x, y);

    p = &x; //r-value of p is changed to the l-value of x (i.e., x's address)
    printf("After changing pointer:\np = %p,
            *p = %d\nx = %d, y = %d\n", p, *p, x, y);

    return 0;
}
```

Scope

Scope

The scope of a name is the collection of statements which can access the name binding.

What are the scopes of i, j, and t?

```
1  void sort (float a[ ], int size) {
2    int i, j;
3    for (i = 0; i < size; i++)
4      for (j = i + 1; j < size; j++)
5        if (a[j] < a[i]) {
6          float t;
7          t = a[i];
8          a[i] = a[j];
9          a[j] = t;
10       }
11 }
```

Scoping error

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
i = 0; // invalid reference to i
```

Design Issues

- Dynamic or static?
- Allow forward reference? (Self study)
  - Using an identifier before its declaration

Static and dynamic scoping

Static scoping: a name is bound to a collection of statements according to its position in the source program.

Dynamic scoping: binding depends on the control flow of the program.

Most modern languages use static (or *lexical*) scoping.

How to implement scoping and name resolution?

Symbol Table (Compiler's Database)

Data structure for scoping and name resolution

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15    int a, b;
16    h = 5; a = 3; b = 2;
17    A(a, b);
18    B(h);
19    ...
20 }
```

i (line 4) vs.
i (line 11)

Symbol table data structure:
Stack of dictionaries

Each dictionary:

{<name, binding>, <name, binding>, …}

Symbol table data structure: Stack of dictionaries

Algorithm:

- { : push a new dictionary
- } : pop the top dictionary
- Var declaration: insert its binding into the top dictionary
- Name reference (not var dec): search stack top to bottom

Static Scoping:

Each function has its own symbol table

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15    int a, b;
16    h = 5; a = 3; b = 2;
17    A(a, b);
18    B(h);
19    ...
20 }
```

i (line 4) vs. i (line 11)

1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
2. Function B: <w, 2> <j, 3> <k, 4>
   <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
3. Function A: <x, 8> <y, 8> <i, 9> <j, 9>
   <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
4. Function main: <a, 15> <b, 15>
   <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Dynamic Scoping:

The whole program maintains a single symbol table

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }
```

Call history: main (17) → A (10) → B

B: <w, 2> <j, 3> <k, 3>

A: <x, 8> <y, 8> <i, 9> <j, 9>

main : <a, 15> <b, 15>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Reference to i (4) resolves to <i, 9> in A.

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }
```

Call history: main (18) → B

B: <w, 2> <j, 3> <k, 3>

main: <a, 15> <b, 15>
<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Reference to i (4) resolves to <i, 1> in global scope.

Another example: swapping using C (static scoping)

```c
#include <stdio.h>
void myFunction();
void swap();

int x = 5, y = 10;

void myFunction()
{
  int x = 100, y = 200;
  swap(); //local var x & y will not be modified
  printf("After swap: x = %d, y = %d\n", x, y);
}

void swap()
{
  int t = x;
  x = y;
  y = t;
}

int main()
{
  myFunction();
  return 0;
}
```

After swap: x = 100, y = 200

Swapping
Perl program (dynamic scoping)

```
$x = 5;
$y = 10;
myFunction(); # call subroutine - no argument

sub myFunction
{
    local($x, $y);
    $x = 100, $y = 200;
    swap(); # no argument
    print "After swap: x = $x, y = $y\n";
}

sub swap
{
    $t = $x; #Is this the x in myFunciton or the global x?
    $x = $y;
    $y = $t;
}
```

After swap: x = 200, y = 100

It is not entirely clear whether the use of dynamic scoping in Lisp and other early interpreted languages was deliberate or accidental. One reason to think that it may have been deliberate is that it makes it very easy for an interpreter to look up the meaning of a name: all that is required is a stack of. Unfortunately, this simple implementation has a very high run-time cost, and experience indicates that dynamic scoping makes programs harder to understand. The modern consensus seems to be that dynamic scoping is usually a bad idea.

Other issues (self study)

- Visibility (re-declaration)
- Overloading
- Lifetime (vs. scope)
  - *Variable can be out-of-scope temporarily but can still be living*