CSCI-2320 Principles of Programming Languages

Names

Ref: Tucker-Noonan [Ch. 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: $\dots = \dots \times \dots$

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)

#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);

return 0;

Scope

Scope

The <u>scope</u> 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 <u>before</u> its declaration

Static and dynamic scoping

- <u>Static scoping:</u> a name is bound to a collection of statements according to its position in the source program.
- <u>Dynamic scoping</u>: 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: <u>Stack</u> 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;			14 void main() {		
2	void B(ir	nt w) {	15 int a, b;		
3	int j, l	k;	16 h = 5; a = 3; b = 2;		
4	i = 2*	Ŵ;	17 A(a, b);		
5	W = W	v+1;	18 B(h);		
6			19		
7	}		$20\}$		
8 void A (int x, int y) {					
9	float	i, j; 1.	Outer scope: <h, 1=""> <i, 1=""> <b, 2=""> <a, 8=""> <main, 14=""></main,></a,></b,></i,></h,>		
10) B(h);	2.	Function B: $ $ $ $		
11	i = 3;	3.	Function A: $, 8> , 8> , 9> , 9>$		
12	<u> </u>	1	<h, 1=""> <i, 1=""> <b, 2=""> <a, 8=""> <main, 14=""></main,></a,></b,></i,></h,>		
13	8 }	4.	Function main: <a, 15=""> <b, 15=""> <h, 1=""> <i, 1=""> <b, 2=""> <a, 8=""> <main, 14=""></main,></a,></b,></i,></h,></b,></a,>		

Dynamic Scoping:

The whole program maintains a single symbol table

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 }

y) {

Call history:	$main (17) \rightarrow A (10) \rightarrow B$
B: A:	<w, 2=""> <j, 3=""> <k, 3=""> <x, 8=""> <y, 8=""> <i, 9=""> <j, 9=""></j,></i,></y,></x,></k,></j,></w,>
main :	<a, 15=""> <b, 15=""> <h, 1=""> <i, 1=""> <b, 2=""> <a, 8=""> <main, 14=""></main,></a,></b,></i,></h,></b,></a,>

Reference to i (4) resolves to $\langle i, 9 \rangle$ 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) {
  float i, j;
9
10 B(h);
11 i = 3;
12 ...
13 }
```

```
14 void main() {
```

```
15 int a, b;
```

```
18 B(h);
```

Call history: main (18) \rightarrow B

B: main: <w, 2> <j, 3> <k, 3> <a, 15> <b, 15> <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Reference to i (4) resolves to $\langle i, 1 \rangle$ in global scope.

Another example: swapping using C (static scoping)

```
#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);
}
                               After swap: x = 100, y = 200
void swap()
{
  int t = x;
  x = y;
  y = t;
}
int main()
{
  myFunction();
  return 0;
}
```

Swapping Perl program (dynamic scoping)

```
x = 5;
y = 10;
myFunction(); # call subroutine - no argument
sub myFunction
ł
    local($x, $y);
    x = 100, y = 200;
    swap(); # no argument
    print "After swap: x = $x, y = $y\n";
}
                            After swap: x = 200, y = 100
sub swap
{
    $t = $x; #Is this the x in myFunciton or the global x?
    x = y;
    y = t;
}
```

Michael Scott (Programming Languages)

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