Names and Types

standard hue names
Dr. Philip Cannata
PLAI Chapters 4, 5 and 6

Chapter 4, Page 27 – “To add functions to WAE, we must define their concrete and abstract syntax. In particular, we must both describe a function definition, or declaration, and provide a means for its use, also known as an application or invocation.”

Chapter 5, Page 33 – “Well, if the program has size n (measured in abstract syntax tree nodes), then each substitution sweeps the rest of the program once, making the complexity of this interpreter at least $O(n^2)$. That seems rather wasteful; surely we can do better. How do we avoid this computational redundancy? We should create and use a repository of deferred substitutions (in the lisp world this is usually called an Environment). Concretely, here’s the idea. Initially, we have no substitutions to perform, so the repository is empty. Every time we encounter a substitution (in the form of a with or application), we augment the repository with one more entry, recording the identifier’s name and the value (if eager) or expression (if lazy) it should eventually be substituted with. We continue to evaluate without actually performing the substitution.”

Chapter 5, Page 36 – “Definition 10 (Static Scope) In a language with static scope, the scope of an identifier’s binding is a syntactically delimited region. A typical region would be the body of a function or other binding construct. In contrast: Definition 11 (Dynamic Scope) In a language with dynamic scope, the scope of an identifier’s binding is the entire remainder of the execution during which that binding is in effect.”
Chapter 5, Page 37 – “Therefore, at the point of invoking a function, our new interpreter must “forget” about the current substitutions . . . That is, we use the empty repository to initiate evaluation of a function’s body, extending it immediately with the formal parameter but no more.”

Chapter 6, Pages 41 & 42 – “first-order Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in F1WAE are of this nature, which explains the 1 in the name of the language.

higher-order Functions can return other functions as values.

first-class Functions are values with all the rights of other values. In particular, they can be supplied as the value of arguments to functions, returned by functions as answers, and stored in data structures.
PLAI Chapters 4, 5 and 6

Chapter 6, Page 46 –

{with {x 3} {with {f {fun {y} {+ x y}}} {with {x 5} {f 4}}} (let ((x 3)) (let ((f (lambda (y) (+ x y)))) (let ((x 5)) (f 4))))}

We call this constructed value a **closure** because it “closes” the function body over the substitutions that are waiting to occur.”

Chapter 6, Page 47 –

“**Crucially, while we evaluate arg-expr in ds, the repository active at the invocation location, we evaluate the function’s body in its “remembered” repository.**

The C programming language implements a middle-ground by allowing functions (technically, pointers to functions) to serve as values but allowing them to be defined only at the top-level. Because they are defined at the top-level, they have no deferred substitutions at the point of declaration; this means the second field of a closure record is always empty, and can therefore be elided, making the function pointer a complete description of the function. This purchases some small implementation efficiency, at the cost of potentially great expressive power, as the examples in Section 6.5 illustrate.
This Course

Java (Object Oriented)

ACL2 (Propositional Induction)

Algorithmic Information Theory (Information Compression and Randomness) - Kolmogorov Complexity

Orc (Parallel Computing)

GpH (Parallel Computing)

RDF (Horn Clause Deduction, Semantic Web)

High Level Languages

Jython in Java

Relation

A Snapshot of Programming Language History
Names

Concepts

• Variables
• Symbol Table
• Scope
• Resolving References
• Static and Dynamic Scoping
• Visibility
• Overloading
• Lifetime
• Binding
Symbol Table

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.
4. Given a name reference, search the dictionary on top of the stack:
   a) If found, return the binding.
   b) Otherwise, repeat the process on the next dictionary down in the stack.
   c) If the name is not found in any dictionary, report an error.
Scope Concepts

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. }
21. void sort (float a[ ], int size) {
22.     int i, j;
23.     for (i = 0; i < size; i++)      // i, j, a, size are local
24.         for (j = i + 1; j < size; j++) {
25.             if (a[j] < a[i]) {
26.                 float t;           // t is local; i, j, a, size are non-local
27.                 t = a[i];
28.                 a[i] = a[j];
29.                 a[j] = t;
30.             }
31.         }
32.     }
33. }

The lifetime of a variable is the time interval during which the variable has been allocated a block of memory.

Disjoint scopes nested in Outer scope

Symbol Table

Lifetime

1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14> <sort, 21> 1 - 33
2. Function B nested scope: <w, 2> <j, 3> <k, 3> 2 - 7
3. Function A nested scope: <x, 8> <y, 8> <i, 9> <j, 9> 8 - 13
4. Function main nested scope: <a, 15> <b, 15> 15 - 20
5. Function sort nested scope: <a, 21> <size, 21> <i, 22> <j, 22> 21 - 33

Nested scope

Dictionaries

Variables are “local” if they are in the “active” dictionary otherwise they’re non-local.
Scope Concepts

Disjoint scopes nested in Outer scope

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. }
21. void sort (float a[ ], int size) {
22.   int i, j;
23.   for (i = 0; i < size; i++) // i, j, a, size are local
24.     for (j = i + 1; j < size; j++) {
25.       if (a[j] < a[i]) {
26.         float t; // t is local; i, j, a, size are non-local
27.           t = a[i];
28.           a[i] = a[j];
29.           a[j] = t;
30.       }
31.     }
32. }
33. }

The lifetime of a variable is the time interval during which the variable has been allocated a block of memory.

Symbol Table

1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14> <sort, 21> 1 - 33
2. Function B nested scope: <w, 2> <j, 3> <k, 3> 2 - 7
3. Function A nested scope: <x, 8> <y, 8> <i, 9> <j, 9> 8 - 13
4. Function main nested scope: <a, 15> <b, 15> 15 - 20
5. Function sort nested scope: <a, 21> <size, 21> <i, 22> <j, 22> 21 - 33

 Nested scope

Dictionaries

For static scoping, the referencing environment (the set of statements which can validly reference a name) for a name is its defining scope and all nested subscopes.

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

In dynamic scoping, a name is bound to its most recent declaration based on the program’s call history.

Since main (17) → A (10) → B
Reference to i (4) resolves to <i, 9> in A.
Disjoint scopes

A name is visible if its referencing environment (the set of statements which can validly reference a name) includes the reference and the name is not declared in an inner scope.

A name redeclared in an inner scope effectively hides the outer declaration.

Some languages provide a mechanism for referencing a hidden name; e.g.: this.x in C++/Java.

```
1 public class Student {
2    private String name;
3    public Student (String name, ...) {
4       this.name = name;
```

Scope Concepts

```
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. }
21. void sort (float a[], int size) {
22.    int i, j;
23.    for (i = 0; i < size; i++) // i, j, a, size are local
24.        for (j = i + 1; j < size; j++) {
25.            if (a[j] < a[i]) {
26.                float t; // t is local; i, j, a, size are non-local
27.                    t = a[i];
28.                    a[i] = a[j];
29.                    a[j] = t;
30.            }
31.        }
32.    }
33. }
```
Disjoint scopes

Dictionaries

Scope Concepts

Overloading uses the number or type of parameters to distinguish among identical function names or operators.

public class PrintStream extends FilterOutputStream {
    ...
    public void print(boolean b);
    public void print(char c);
    public void print(int i);
    public void print(long l);
    public void print(float f);
    ...

Scope Concepts

1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14> <sort, 21> 1 - 33
2. Function B nested scope: <w, 2> <j, 3> <k, 3> 2 - 7
3. Function A nested scope: <x, 8> <y, 8> <i, 9> <j, 9> 8 - 13
4. Function main nested scope: <a, 15> <b, 15> 15 - 20
5. Function sort nested scope: <a, 21> <size, 21> <i, 22> <j, 22> <t, 26> 21 - 33

Nested scope

Dictionaries

1. int h, i;
2. void B(int w) {
   3.     int j, k;
   4.     h = 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. }
21. void sort (float a[], int size) {
  22.     int i, j;
  23.     for (i = 0; i < size; i++) // i, j, a, size are local
  24.         for (j = i + 1; j < size; j++) {
  25.             if (a[j] < a[i]) {
  26.                 float t; // t is local; i, j, a, size are non-local
  27.                 t = a[i];
  28.                 a[i] = a[j];
  29.                 a[j] = t;
  30.             }
  31.         }
  32.     }
  33. }

Symbol Table

1. Lifetime
   1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14> <sort, 21> 1 - 33
   2. Function B nested scope: <w, 2> <j, 3> <k, 3> 2 - 7
   3. Function A nested scope: <x, 8> <y, 8> <i, 9> <j, 9> 8 - 13
   4. Function main nested scope: <a, 15> <b, 15> 15 - 20
   5. Function sort nested scope: <a, 21> <size, 21> <i, 22> <j, 22> <t, 26> 21 - 33

Overloading uses the number or type of parameters to distinguish among identical function names or operators.
Binding

• A *binding* is an association between an entity (such as a variable) and a property (such as its value).

• A binding is *static* if the association occurs before run-time.

• A binding is *dynamic* if the association occurs at run-time.

• Name bindings play a fundamental role.

• The lifetime of a variable name refers to the time interval during which memory is allocated.
Variables

Basic bindings

- Name
- Address
- Type
- Value
- Lifetime
Variables

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

Ex: \( x = \ldots \)

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

Ex: \( \ldots = \ldots x \ldots \)

Some languages support/require explicit dereferencing.

Ex: \( x := !y + 1 \)
int main () {
    int x, y;
    int *p;
    x = *p;
    *p = 123;
    printf("x: %d, y: %d, p: %d", x, y, p);
}

C Pointer Example
Types

- Type Errors
- Static and Dynamic Typing
- Basic Types
- NonBasic Types
  - Recursive Data Types
  - Functions as Types
- Type Equivalence
- Subtypes
- Polymorphism and Generics
- Programmer-Defined Types
Types

A *type* is a collection of values and operations on those values.

Example: Integer type has values ..., -2, -1, 0, 1, 2, ... and operations +, -, *, /, <, ...

Boolean type has values true and false and operations \( \land, \lor, \neg \).

A *type system* is a precise definition of the bindings between the type of a variable, its values and the possible operations on those values. It provides a basis for detecting type errors.

A *type error* is any error that arises because an operation is attempted on a data type for which it is undefined.

Computer types have a finite number of values due to fixed size allocation; problematic for numeric types.

Exception - Haskell type Integer represents unbounded integers.
Types

A language is *statically typed* if the types of all variables are fixed when they are declared at compile time.

A language is *dynamically typed* if the type of a variable can vary at run time depending on the value assigned.

Some languages perform type checking at compile time (e.g., C). Other languages (e.g., Perl) perform type checking at run time. Still others do both (e.g., Java - downcasting - `stringRef = (String) objRef;`).

A language is *strongly typed* if its type system allows all type errors in a program to be detected either at compile time or at run time.

A strongly typed language can be either statically or dynamically typed.
$ cat Test.java
class Animal {
    static int value = 100;
}
class Cat extends Animal { int value = Animal.value + 1; }
class Dog extends Animal { int value = Animal.value + 2; }

public class Test {
    public static void main(String[] args) {
        Cat c = new Cat();
        System.out.println(c.value);
        Dog d = new Dog();
        System.out.println(d.value);
        Animal a = c;
        System.out.println(a.value);
        Object o = c;
        // System.out.println(o.value);
        a = d;
        type(c);
        type(d);
        type(a);
        // type(a.value);
        Dog d2 = (Dog) a;
        a = c;
        Dog d3 = (Dog) a;
    }
    public static void type(Animal a) {
        System.out.println("I am a " + a);
    }
}

Uncomment // System.out.println(o.health);

Uncomment // System.out.println(a.health);
Types

A *type conversion* is a narrowing conversion if the resulting type permits fewer bits, thus potentially losing information.

Otherwise it is termed a *widening* conversion.

An operator or function is *overloaded* when its meaning varies depending on the types of its operands or arguments or result.
$ cat prog.c

#include <stdio.h>
#include <stdlib.h>

int main ()
{
    int x = 1, y = 2;
    float a = 1.1, b = 2.2;

    printf("\n%d, %d, %d, %d\n", x, x < y, x > y, y);
    printf("%f, %f, %f, %f\n", x, x < y, x > y, y);
    printf("%f, %d, %d, %f\n", a, a < b, a > b, b);
    printf("%d, %d, %d, %d\n", a, a < b, a > b, b);
}
$ cat test.java

class test
{
  static int x = 1, y = 2;

  public static void main(String args[])
  {
    System.out.println("x + " + " + x < y + " + " + x > y + " + y");
  }
}

// Doesn’t compile
Type Errors

```java
$ cat test.java

class test{
    static int x = 1, y = 2;
    static float a = 1.1f, b = 2.2f;

    public static void main(String args[])
    {
        System.out.println( x + "", " + (x < y) + ", " + (x > y) + ", " + y );
        System.out.println( a + "", " + (a < b) + ", " + (a > b) + ", " + b );
    }
}
```
Type Errors

$ cat test.java

class test
{
    static int x = 1, y = 2;
    static float a = 1.1f, b = 2.2f;

    public static void main(String args[])
    {
        System.out.println( x + "", " + (x < y) + ", " + (x > y) + ", " + y );
        System.out.println( a + "", " + (a < b) + ", " + (a > b) + ", " + b );
        System.out.println( (x / y) + ", " + (a / b) + ", " + (x / b) + ", " + (x < b) );
    }
}
Basic Types

Terminology in use with current 32-bit computers:

- Nibble: 4 bits
- Byte: 8 bits
- Half-word: 16 bits
- Word: 32 bits
- Double word: 64 bits
- Quad word: 128 bits
Nonbasic Types

- Pointers
- Enumerations
- Arrays
- Strings
- Structures
- Unions
Pointers

- C, C++, Ada, Pascal
- Java???
- Value is a memory address
- Indirect referencing
- Operator in C: *
struct Node {
    int key;
    struct Node* next;
};
struct Node* head;

float sum(float a[], int n) {
    int i;
    float s = 0.0;
    for (i = 0; i<n; i++)
        s += a[i];
    return s;
}

float sum(float *a, int n) {
    int i;
    float s = 0.0;
    for (i = 0; i<n; i++)
        s += *a++;
    return s;
}

void strcpy(char *p, char *q) {
    while (*p++ = *q++);
}

a[i] = *(a + i)
$ cat test.
float x[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};

main() {
    printf("In main, a[1]: %f\n", x[1]);
    printf("x: %d\n", x);
    sum(x, 10);
}

sum( float a[], int n)
{
    printf("a: %d\n", a);
    int i;
    float s = 0.0;
    for (i = 0; i<n; i++)
    {
        s += a[i];
        printf("In loop s, a[i]: %f\n", s, a[i]);
    }
    printf("s: %f\n", s);
}

$ cat test.c
main()
{
    char s1[] = "string1"
    char s2[] = "string2"
    printf("s1, s2: %s, %s\n", s1, s2);
    mystrcpy(s1, s2);
}

mystrcpy(char *p, char *q)
{
    char *sp = p;
    char *sq = q;
    printf("p, q: %s, %s\n", sp, sq);
    while (*p++ = *q++)
    {
        printf("p, q: %s, %s\n", sp, sq);
    }
}
$ cat test.c
int main ( ) {
    int x = 101, y = 202;
    int *p;
    printf("x: %d, y: %d, p: %d\n", x, y, p);
    p = &y;
    printf("x: %d, y: %d, p: %d\n", x, y, p);
    x = *p;
    *p = y;
    printf("x: %d, y: %d, p: %d\n", x, y, p);
    printf("x: %d, &x: %d, *(&x): %d\n", x, &x, *(&x) );
}
Pointers

- Bane of reliable software development
- Error-prone
- Buffer overflow, memory leaks
- Particularly troublesome in C
Structures

- Analogous to a tuple in mathematics
- Collection of elements of different types
- Used first in Cobol, PL/I
- Absent from Fortran, Algol 60
- Common to Pascal-like, C-like languages
- *Omitted from Java as redundant*
Structures

```c
struct employeeType {
    int id;
    char name[25];
    int age;
    float salary;
    char dept;
};
struct employeeType employee;
...
employee.age = 45;
```
Unions

Logically: multiple views of same storage

```c
struct state_table {
    char name[100];
    char type[100];
    int initialized;
    union {
        double d;
        long l;
        char *s;
    } value;
} state_table[MAX];
```