

CSC324H Principle of Programming Languages (Week 11)

Yilan Gu

yilan@cdf.toronto.edu

<http://www.cs.toronto.edu/~yilan/324f09>

1

Types, Values and Scopes

3

Administrative Issue

- Assignment 5 due Dec. 3 (6:00pm sharp), and cannot be later than Dec. 4 (6:00pm sharp)
- Reference/Reading for week 11 and week 12 will be distributed in class, as well as out of the instructor's office (SF3209)
- Arrangement of next week:
 - One more tutorial
 - Talk about Final exam
 - Distribute cheating sheet for final exam
 - Something more about PL Design
 - Review

2

Basic concepts

- Names
- Storage
- Types
- Scopes and referencing environments
- Functions as parameters

These concepts are useful in understanding most programming languages (and programs).

Reference: Sebesta, chapter 5

4

Names

- A name identifies a variable (or other thing)
- The identified variable has attributes:
 - name
 - memory address
 - type
 - value
 - scope
 - lifetime

5

Terminology: static vs dynamic

- Static: "during compilation"
 - more generally, before the program begins to run
 - "compile-time"
- Dynamic: "while the program is running"
 - "run-time"
- Some things that can be either static or dynamic:
 - binding
 - errors
 - storage allocation
- The "static" keyword in C, C++, Java is related but still different.
 - "just once"
 - "invisible"
 - "class-related"

7

Memory Addresses

- Usually considered unchangeable – a variable can't be moved to a different address.

```
int i, j, *p;
p = &i;
p = &j;
```

 - The variable p isn't moving; it stays in the same memory location but stores different values (its values are addresses).
- But if you use the same variable name in different functions, the name means different addresses in different places.
- Some languages allow names to be redefined.
 - Scheme: define, set!
 - Python: depends on the way we look at things
 - All variables really store references, and the variables themselves can't be moved.

6

Binding

- **Binding** is an association between an attribute and an entity, or between an operation and a symbol.

```
count = count + 5;           // C assignment
```

 - The type of count is bound at compile time
 - The meaning of + is bound at compile time
 - The value of count is bound at the executing time with this statement.
- A name is bound to a variable statically in most of the languages we're used to.
- In other languages, the name-variable binding is dynamic.
 - The same name can be re-bound to a different variable.
 - e.g. Scheme

8

Binding

- **Binding** is an association between an attribute and an entity, or between an operation and a symbol.
`count = count + 5; // C assignment`
 - The type of count is bound at compile time
 - The meaning of + is bound at compile time
 - The value of count is bound at the executing time with this statement.
- A name is bound to a variable statically in most of the languages we're used to.
- In other languages, the name-variable binding is dynamic.
 - The same name can be re-bound to a different variable.
 - e.g. Scheme

9

Dynamic Name Binding

- Dynamic binding: While a program is running, you can redefine a name to mean a different variable.
- Can't be done in C or Java
`int i = 5;`
`i = 6;`
 - The name denotes the same variable, though the value changes.
- C++'s reference variables provide an alternative name for the same variable.
`int i = 0;`
`int& k = i`
`k = 6 // Now i == 6 too.`
 - But in fact you can't change what k refers to.

10

Dynamic Name Binding

- Here we have dynamic name binding in pseudo-code:
`define i : int`
`i = 5`
`define i : string`
`i = "hi"`
 - The same name comes to denote some different thing (where "thing" really means a memory location).
- Languages like Scheme and ML allow this.

11

Dynamic Binding to Storage

- We said binding of a variable to a memory address is "Usually considered unchangeable".
 - But that doesn't mean that storage binding is always static.
 - Stack-dynamic: e.g. local variables in functions
 - Heap-dynamic: nameless "variables" (we usually think of them as blocks of memory rather than as variables) allocated at run-time using malloc, new, or implicitly allocated at run-time during object creation (e.g. in Python).
 - If variables are appearing and disappearing, then their connection to a memory location is dynamic.
 - But while the variable exists, this connection is unchangeable (in C and similar languages).
- ```
int f() {
 int i; // different calls of f() have different i's
 ...
}
```

12

## Stack-Dynamic Variables

- Suppose factorial is a recursive function. In each call of factorial on the stack, there is a distinct variable n, and each such variable n has a fixed location (while it exists).

|                       |
|-----------------------|
| Factorial(1)<br>n = 1 |
| Factorial(2)<br>n = 2 |
| Factorial(3)<br>n = 3 |

13

## Heap-Dynamic Variables

- A heap storage example (in C):
- Observe that the above code creates two heap-dynamic variables (each having the same size as an integer).

```
int *p;
// p on a heap storage at location 1196
p = malloc(sizeof(int));
// now p == 4104
*p = 17
p = malloc(sizeof(int));
// now p == 12596
*p = -23;
```

|                  |
|------------------|
| at 1196          |
|                  |
| at 4104:<br>17   |
|                  |
| at 12596:<br>-23 |

14

## The Lifetime of a Variable

- Stack-dynamic storage is allocated on function entry, and deallocated on function exit.
  - Storage management is always automatic, and not left to the programmer.
- Heap-dynamic storage is allocated on demand. It is freed either on demand or as needed:
  - Freed on demand
    - If heap freeing is managed by programmers, as in C and C++.
  - Freed when needed
    - If heap freeing is managed by an automatic garbage collector, as in Scheme, Java, Python.

15

## Types

- "X is a strongly typed language."
  - Where X might be Java, Python, ...
- Strong typing may not be clearly defined, but Sebesta's description helps:
  - A language is strongly typed if "type errors are always detected."
  - This implies that "the types of all operands can be determined, either at compile time or at run time."
  - Sebesta, pg. 219.
- Strongly typed languages include:
  - Python
  - Java
  - C#
- What about C/C++?
  - Not strongly typed, since there is no type checking when using union types.

16

## Static vs. Dynamic Typing

- This is separate from the issue of strongly vs not-strongly typed.
- Static typing:
  - Types are declared by the programmer.
  - Or, types are deduced from code by the compiler. For example, in the language ML:

```
fun square (x : int) = x*x;
```

    - Obviously fun returns an int, and you don't have to say so because the ML compiler can figure it out.
- Dynamic typing:
  - Types are determined at run-time.
  - e.g. Scheme, Python

17

## Scope

- **Scope:** the part of a program where a variable can be referred to.
- In block-structured languages, scope generally consists of:
  - The block where the variable is declared, plus
  - blocks contained by that block.

19

## Types in Records

- A record is a C-type struct.

```
struct { int a; char b; };
struct { int x; char y; };
```
- Are these the same type?
  - When using name type compatibility: no
    - Name type compatibility: Types are only the same if they have the same name.
    - Used by C++, Java, Python.
  - When using structure type compatibility: yes
    - Structure type compatibility: Types are the same as long as they have the same structure.
    - Used by C

18

## Blocks in C

```
int f(...) {
 int x;
 ... // can refer to x here
 for (...) {
 int y;
 int x; // another x; this declaration "hides" the old x
 ... // can refer to x (the new x) and y here
 }
 ... // can refer to x (the original x) here
}
```

20

## Blocks in Python

```
def outer():
 x = 3;
 def sub1(): # We can't modify x but we can refer to it.
 print 'sub1: x =', x
 def sub2(): # Or we can define another x which "hides" the old one.
 x = -5
 print 'sub2: x =', x
 sub1()
 sub2()
 print 'outer: x =', x

>>> outer()
sub1: x = 3
sub2: x = -5
outer: x = 3
```

21

## Caller vs Ancestor

```
procedure A:
 var x : integer;
 x := 2;
 procedure sub1:
 x := x + 1;
 end sub1;
 procedure sub2:
 var x : integer;
 x := 1;
 sub1; // Which x is changed by this call?
 end sub2;
end A;
```

- Using static scoping, A's x is changed.
- Using dynamic scoping, sub2's x is changed.

23

## Dynamic Caller, Static Ancestor

- Caller: the procedure that at run time initiated the execution of the current procedure.
  - This is dynamic, not fixed at compile-time.
- Ancestor: the procedure which encloses the code of the current procedure, in the text that the programmer wrote.
  - This is fixed at compile-time.
  - Just read the code to find out.
- In a block-structured language:
  - Static scoping: what names can be used is determined by ancestry.
    - Used in most languages.
  - Dynamic scoping: what names can be used is determined by calling history.
    - Not used very much. One example: The original LISP.

22

## Scope vs Lifetime

- Scope is the part of the program text where you can refer to a variable.
- Lifetime is the time period of the program's execution when the variable exists.
- They may be related but not the same.
- A variable may exist during the execution of code that is not within its scope:

```
void b() {
 ... /* x isn't visible here */
}
void a() {
 int x;
 b(); /* x exists while this call is running */
}
```

24

# Referencing Environment

- **Referencing environment** is the set of names that can be used at a particular point in a program
- Determining the referencing environment is simple in a language with static scope.
  - The set of usable names is the set of names that are in-scope and not hidden.
  - This can be found from the program text.
- Determining the referencing environment is harder in a language with dynamic scope.
  - This can only be found by understanding execution, since it's depends on the calling history.

25

# Referencing environments in C and Java

- C: two sets of names are available:
  - names defined locally within a function
  - names defined globally, externally to functions
    - Some of these invisible, through "static" declarations (in other files).
- Java: many sets of names
  - names defined locally within a method
  - instance variables for the current object
  - class variables for the current object's class
  - public instance variables for other objects
    - These objects must themselves be part of the referencing environment.
  - public class variables for other classes
- Java has no set of globally-defined names like C's.

27

- An example of Ada skeletal program:

```
procedure Example is
 A, B: Integer;
 ...
 procedure Sub1 is
 X, Y: Integer;
 begin -- of sub1
 ... <-----X and Y of Sub1, A and B of Example
 end;
 procedure Sub2 is
 X: integer;
 ...
 procedure Sub3 is
 X: integer;
 begin -- of sub3
 ... <-----X of Sub3, A and B of Example
 end;
 begin -- of sub2
 ... <-----X of Sub2, A and B of Example
 end;
 begin -- of Example
 ... <----- A and B of Example
 end;
```

26

# A Dynamic-Scope Example

- In a C-like language with dynamic scoping:

```
void f1() { int a, b; 1 ... }
void f2() { int b, c; 2 ... f1(); }
void main() { int c, d; 3 ... f2(); }
```
- Which variables are accessible at points 1, 2 and 3?
  - At point 1: f1's a, f1's b, f2's c, main's d.
  - At point 2: f2's b, f2's c, main's d.
  - At point 3: main's c, main's d.

28

## "First-Class" Status for a Type

- A first-class type is a type with values that can be:
  - created at run time
  - assigned to variables
  - returned from functions
  - passed as an argument
  - exist without a name
- You can think of values of a first-class type as "things" in your program.
  - "Things" that you can work with
- Are functions first-class?

29

## Are functions things in C?

- C functions can be:
  - passed as parameters
  - returned as function results
  - assigned to variables
  - but, only in the form of pointers to statically-defined functions
- They cannot be created at run time.
- They cannot exist without a name.
- That is, C doesn't have anything like a lambda expression.

31

## A C Example

```
double integrate(double *f(double), double a, double b)
{ ...
 sum += f(x);
 ...
}
double myFun (double x)
{
 return 3*x*x + 2*x + 1;
}
int main(void)
{ ...
 printf(integrate(myFun, 0.5, 12));
 ...
}
```

30

## “Functions” in Java?

- Functions in Java are methods of objects or classes.
- They cannot be referred to except by calling them.
- They can be "created" at run time by instantiating an anonymous class:  

```
MyFile x = new MyFile() {int getFileType() { return 3;}};
```

  - This defines (and instantiates) a nameless class that extends MyFile.
  - But the code for the method exists at compile time.
- Unlike C functions, Java methods cannot be passed as parameters

32

## So, are functions first-class?

- Not in C or Java.
- Yes, in functional languages: Scheme, Lisp, ML
- Almost yes, in Python:
  - Recall that lambda expressions in Python can only consist of a single expression.
    - This restricts our ability to create functions at run time.
- Prolog (which we'll be looking at next) doesn't exactly have functions, but its terms can behave like functions and can be created at run time.

33

## Example: shallow vs deep

- Looks like C but isn't:

```
int x;
f2() { printf(x); }
f3() { int x = 3;
 f4(f2); }
f4(void f())
 { int x = 4; f(); }
x = 1;
f3();
```

- Shallow: prints \_\_\_\_\_
- Deep: prints \_\_\_\_\_
- See the next slide for answers...

35

## Referencing in Parameter Functions

- If you pass a function as a parameter, what names can it refer to?
- *Shallow binding*: the names available where the function is actually called.
- *Deep binding*: the names available where the function was defined.
  - This is what is done in Python.
- This is not the same as the distinction between dynamic and static binding!
- Reference: Sebesta, Section 9.6.

34

## Answers

- Shallow: 4
- Deep: 1
- And 3? That's ad-hoc binding.
  - Ad-hoc binding: The names that are available are those available in the function that passes (as opposed to receives) the parameter.

36

## Procedural Programming Design

37

## Subprograms: introduction

- **Subprograms are really a control abstraction**
  - It is one of two fundamental abstraction mechanisms (what's the other one?)
- **Characteristics:**
  - A subprogram has a single entry point
  - Caller is suspended during execution of the called subprogram
  - Control always returns to the caller when the called subprogram's execution terminates
  - Master/slave model
- **A subprogram can access data in two ways:**
  - Direct access to non local variables
  - Parameter passing
- **Reference: Sebesta Chapter 9.1-9.6**

38

## Subprograms: introduction cont'd

- **Advantages:**
  - Allow better reuse:
    - Savings from memory space to coding time
    - The details of the program computation are hidden
  - Increase readability of programs:
    - Exposing their logical structure
    - Hiding the small scale details
- **Each programming paradigm implement subprograms in a different way:**
  - Imperative: block of code that can be called
    - Procedure:
      - Group user-specified statements in a single body
      - Define a new statement in the language
    - Function:
      - Structurally resemble procedures.
      - Semantically built on mathematical functions; no side effects and return a value
      - Much like user-defined operators
  - Functional: lambda expression
  - Logic: horn clause

39

## Components

- **Name**
- **Parameters (optionally with types)**
  - Formal Parameters (parameter)
    - Local variable to the subprogram whose value is received from caller
  - Actual Parameter (argument)
    - Info passed from caller to callee
- ***Subprogram header:*** name + formal parameters
- **Body; a syntactic construct in the language, could be:**
  - Block, i.e. declarations and statements
  - Expression
  - Conjunction of terms
- **Optional result (with/without a type)**
- **E.g.**

```
int foo(int nCount , struct strctData) { // C
for(int nIndex=0; nIndex < nCount; nIndex++)
printf(strctData.name[nIndex]);
```

40

## Syntax Examples

```
// Ada: function nested in a procedure
procedure Display_Even_Numbers is
 <declarations>
 function even (number:integer) return boolean is
 begin
 <statements>
 end even;
begin
 <statements>
end Display_Even_Numbers;
```

```
// Pascal: procedure
procedure count(k: array[1..5] of real);
const
 <constant-declarations>
type
 <type-declarations>
var
 <variable-declarations>
// nested procedures and functions go here
begin
 <statements>
end;
```

```
// Fortran: subroutine
SUBROUTINE SUM(MATRIX,ROWS,COLS)
 INTEGER ROWS,COLS
 REAL MATRIX(ROWS,COLS)
 <statements>
RETURN
END
```

```
// Algol60: procedure
real procedure average(A,n);
real array A; integer n;
begin
 real sum; sum:= 0;
 for i := 1 step 1 until n do
 sum := sum + A[i];
 average:= sum/n;
end;
```

41

## Subprograms Implementation Issues

- **The general notion of a subprogram leaves a number of points unspecified:**
  - How to pass parameters when the subprogram is called?
  - How to maintain local state and control information?
  - How to access non-local names within a subprogram body?

42

## Parameters Passing

- **Positional association:**
  - Arguments associated with formals one-by-one from left to right
  - E.g. C, Pascal, Scheme, Java
 

```
public plot(int x, int y, boolean penup)

plot(some_int, somey, flag);
```
- **Keyword association:**
  - Uses keywords to match arguments to formals. Order is irrelevant.
  - E.g.
 

```
procedure plot (x, y: in real; penup: in boolean)

plot(0.0, 0.0, penup=> true)
plot(penup=> true, x=>0.0, y => other_variable)
```
- **Optional arguments:**
  - Implementation:
    - Extra arguments are packaged into some structure
    - Passed to special parameter
  - E.g.

```
// C language, dots for optional arg
int set(char *item, int num, ...);

main(){
 nReturn = set("pear", 1, "grape");
 nReturn = set("pear", 2, "apple", "g");
}
43
int set(char *item, int num, ...){blah ..}
```

## Parameter Passing

- **Default Parameter Values:**
  - If argument is not initialized, it is assigned default value
  - Some languages use default parameters to simulate variable number of parameters.
  - E.g.

```
// Ada
type field is integer range 0..50;
type number_base is integer range 2..20;
default_width : field := 25;
default_base : number_base := 10;
....
....
procedure put (item : in integer;
 width : in field := default_width;
 base : in number_base := default_base);
begin
end put;
```

```
// Pascal
function foo (item: Integer;
 width: field = default_width;
 base: number_base = default_base;
) : Integer;

begin
 <statements>
end;
....
....
begin
 foo(10); // ok
 foo(10, 5); // ok?
end.
```

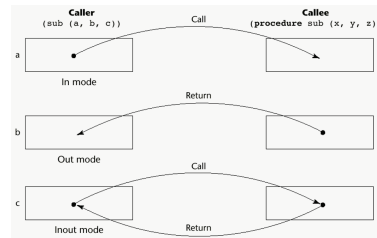
44

## Parameter Passing

- **How to treat arguments? how to implement argument passing?**

- **Semantic models:**

- In mode
- Out mode
- In/out mode



- **Conceptual models of transfer:**

- Physically move a value
- Move an access path (pointer)

- **Implementation models:**

- Pass by value (Java, C, C++, Pascal, Ada, Scheme, Algol68)
- Pass by result (Ada)
- Pass by value-result (some Fortrans, Ada)
- Pass by reference  
(Java objects, C++ with &, some Fortrans, Pascal with var, COBOL)
- Pass by name (Algol 60)

45

## Parameter Passing

- **Pass by Value**

- Initial value of parameters are copied from current values of arguments
- Final values of parameters are “lost” at return time (like local variables)
- E.g.

**Call 1 :**

**k =**

**j =**

**Print 1:**

- Advantage:
  - Arguments protected from change in callee
- Disadvantage:
  - Copying of values takes execution time and space, especially for aggregate values
- PLs: C, Pascal, Ada, Scheme, Algol68

47

## Example for Passing Mode

```
{ c : array[1..10] of integer;
m,n integer;
procedure r (k , j : integer) begin
k := k + 1;
j := j + 2
end r;
...
m := 5;
n := 3;
r(m,n); // call 1
write m, n ; // print 1

m := 2;
c[1] := 1;
c[2] := 4;
c[3] := 8;
r(m,c[m]); // call 2
write m,c[1],c[2],c[3]; // print 2
}
```

46

## Parameter Passing

- **Pass by Result**

- No initial value of parameters
- Final values of parameters are copied back to arguments
- E.g.

Call 1 in the example:

**k =**

**j =**

**Print 1 ?**

- Disadvantage:
  - Requires copying of values, costs time and space, especially for large aggregates.
  - What if the argument is not a variable (e..g  $r(1,2)$  )?
  - What if the variable is used twice in the argument list? (e.g.  $r(m,m)$  )?
- PLs: Ada

48

## Parameter Passing

- **Pass by Value-Result**
  - Initial values of parameters copied from current values of arguments
  - Final values of parameters copied back to arguments
  - Combines functionality of pass by value and pass by result for same parameter
  - E.g.
 

```
Call 1:
 k= j=
Print 1:
Call 2:
 k= j=
Print 2:
```
  - Has all the advantages and disadvantages of value and result together
  - PLs: Fortran, sometimes Ada

49

## Parameter Passing: Aliasing

```
{ y : integer ;
 procedure p (x : integer) begin
 x := x + 1;
 x := x + y
 end p;
 ...
 y := 2;
 p(y);
 write y
}
```

- Pass by Reference:
  - The identifiers x and y refer to the same location in call of p.
  - Result of “write y”?
- Pass by Value-Result:
  - The identifiers x and y refer to different locations in call of p.
  - Result of “write y”?

51

## Parameter Passing

- **Pass by Reference**
  - Formal parameters are pointers to the actual parameters (arguments)
  - Address computations are performed at procedure call
  - Changes to the formal parameters are thus changes to the actual parameters
  - E.g.
 

```
Call 1:
 k= j=
Print 1:
Call 2:
 k= j=
Print 2:
```
  - Advantage:
    - More efficient than copying
  - Disadvantages:
    - Can redefine constants and expressions (e.g. Fortran66: `foo(0,x)` )
    - Aliasing: when there are two or more different names for same storage location.
    - If an error occurs, harder to trace values since some side effected values are in environment of the caller.
  - PLs: Fortran, Pascal var params, Cobol

50

## Parameter Passing

- **Pass by Name**
  - A “name” for the argument is passed in to procedure
  - Like textual substitution of argument in procedure
  - Thus address computations are done whenever parameter is used
  - Like pass-by-reference for scalar parameters
  - E.g.
 

```
Call 1:
 m= n=
Call 2:
 m= c[m]=
```
  - Advantage: same as pass by reference
  - Disadvantage: Inefficient, requires a thunk:
    - essentially a little program is passed that represents the argument
    - evaluates argument in caller’s environment

52