

Administrative Issue

CSC324H Principle of Programming Languages (Week 12)

Yilan Gu

yilan@cdf.toronto.edu

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

1

- **Assignment 5 cannot be later than Dec. 4 (6:00pm sharp)**
 - **A4 solution is posted, A5 solution will be posted on Friday night**
 - **Final exam**
 - Time: Dec, 9, 7:00pm - 10:00pm,
 - Location: WI1016 :
- WI - - Wilson Hall, New College, 40 Willcocks Street (east of Spadina Avenue)
- One **green** cheat sheet, 20% policy
 - Other paper is not allowed
 - Contents will be reviewed in today's lecture
- **Office hour for final exam:**
 - **Monday Dec. 7, 3:20-4:50pm SF3207**
 - **Wednesday Dec. 9, 10:30am-12:00pm, 1:30pm-4:50pm drop by SF3209.**

2

Programming Design

- Type, value and scope
- Subprogram
- Parameter passing
 - Pass by value
 - Pass by result
 - Pass by value-result
 - **Pass by reference**
 - **Pass by name**
- **More on subprogram, blocks and scopes**
- **Review**
- **Course evaluation**

3

Procedural Programming Design

4

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: Sebasta Chapter 9.1-9.6**

5

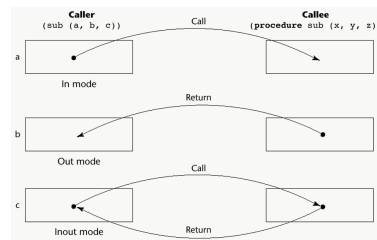
Subprograms Implementation Issues

- **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?

6

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)



7

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
}
    
```

8

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

9

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”?

11

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

10

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

12

Subprograms: parameter passing example

```

int p = 0;

foo(int x, int y) {
    x = y + p;
    printf("%d %d %d ", x, y, p);
}

main() {
    int a[3];

    a[0] = 3;
    a[1] = 2;
    a[2] = 1;

    foo(p, a[p]);
    printf("%d\n", p);
}

```

Pass-by-	Output
Value	3 3 0 0
Value-result	3 3 0 3
Reference	3 3 3 3

Subprograms: activation

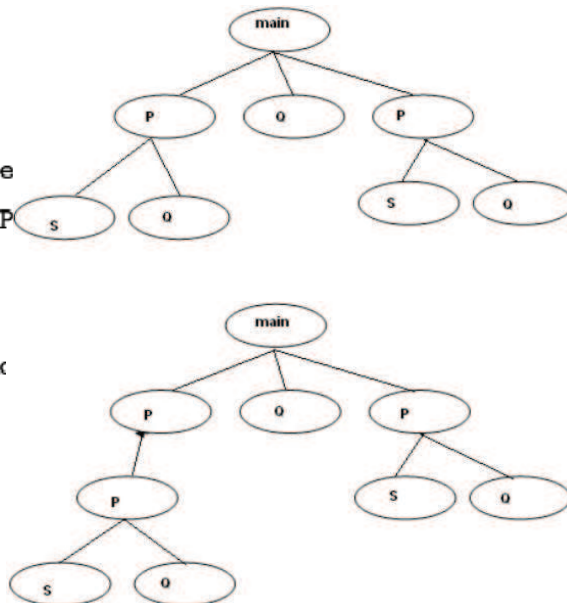
- Each execution of a subprogram is called an **activation**.
- **Life-time of a subprogram:**
 - Begins when control enters activation (call)
 - Ends when control returns from activation
- **Activation tree:**
 - Shows flow of control from one activation to the other
 - **Root:** main program.
 - **Edges (control links):** call from one procedure to another (left to right) control
 - **Leaves:** procedures that call no other procedures

Activation Tree Examples

```

main
  procedure P
  begin
    procedure S begin ... e
    if random(1) < 1 then P
    else { S(); Q() }
    end P;
  procedure Q begin ... enc
  P;
  Q;
  P;
end

```



Activation Records

- **Run-time stack contains an activation record for each active procedure.**
- **Each activation record includes:**
 - **Return** address (within caller)
 - **Static link:** a pointer to the activation record of the static parent, i.e. the activation record of the procedure that contains the definition of the owner of this record.
 - **Dynamic link:** a pointer to the activation record of caller
 - Storage for **parameters**
 - Storage for **local variables**

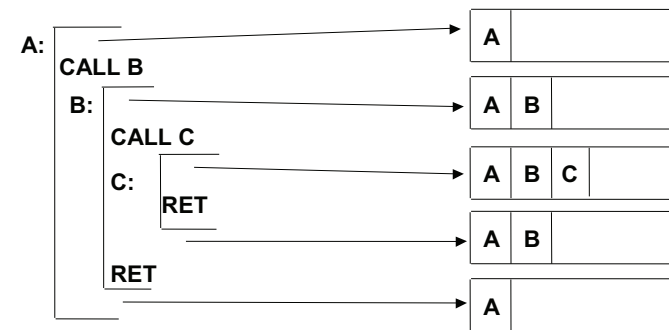
Local
Local
Local
Local
Local
Local
Parameter
Parameter
Dynamic link
Static link
Return address

How would you access the non-local variables?

Subprograms: stack frames

- **Running a program corresponds to a traversal of one of its activation trees.**
- **We can represent the traversal of the tree using a stack**
 - Each item in the stack is called a frame
 - The stack of frames not only maintains the call sequence info, but also keeps track of the local and non-local environment for each procedure.

Subprograms: stack frames cont'd



A → B → C

- Some machines provide a memory stack as part of the architecture (e.g. VAX)
- Sometimes stacks are implemented via software convention (e.g. MIPS)

Activation & Run-time Stack

- **On a call:**
 - Setup stack frame on top of run-time stack (current context)
 - Do the real work of the procedure body
- **On a return:**
 - Release stack frame and restore caller's context (as new top of stack)

Scope (revisit with subprogram)

- **The textual region of the program in which a specific set of variable bindings are active is called the *scope*.**
 - *Visible*
 - *Local*
 - *on-local*
- **The same identifier may refer to different things in different parts of the program**
- **Elaboration:**
 - Opening a new scope and creating appropriate bindings.
 - Upon entering a block

Scope & Blocks

- A block is a section of code in which local variables are allocated and de-allocated at the start/end of the block.

- **Examples:**

```
fun cube(x = x*x*x);
```

```
fun f(a::b::) = a+b
```

```
| f [] = 0;
```

```
while (i < 0) {
  int c = i*i*i;
  p += c;
  q += c;
  i -= step;
}
```

declare

```
LCL : FLOAT;
begin
...
end
```

```
(let ((a 1)
      (b foo)
      (c))
      (setq a (* a a))
      (bar a b c))
```

Scope, Blocks & Nesting

- What happens if a block contains another block, and both have definitions of the same name?

- **Example:**

```
let
  val n = 1
in
  let
    val n = 2
  in
    n
  end
end
```

Static Scope

- **Defines scope in terms of the lexical structure of the program**
 - A name begins life where it is declared and ends at the end of its block.
 - A scope of a variable is known before execution
 - Can be fully determined and bindings made at compile time
 - When writing a program one typically chooses the most recent, active binding made at compile time
 - E.g. C, C++, Pascal, Java, Fortran, Basic...

- **E.g.**

```
int main() { // basic scope
  int k;
  .....
}
```

```
void testfunc() { // nested scope
  int a; // a enters scope;
  for ( int b=1; b<10; b++) { // b in scope
    int c; // c enters scope
    ...
  } // b,c leave scope
} // a leaves scope
```

Static Scope

A name begins life where it is declared and ends at the end of its block.

- **E.g.**

```
{
  int x = 0;
  foo(x);
  {
    int x = 1;
    bar(x);
  }
  bar(x);
}
```

Classic Block Scope Rule: When using static scope, the scope of a definition is the block containing that definition, from the point of definition to the end of the block, minus the scopes of any redefinitions of the same name in interior blocks

Hence, innermost scope overrides declarations from outer scopes.

Static Scope (recall)

- A name begins life where it is declared and ends at the end of its block.

- **E.g.**

```

program test;
var a : integer;

procedure proc1;
  var b : integer;
  begin
  end;
  ← in scope: b (from proc1), a (from test)

procedure proc2;
var a, c : integer;
begin
  proc1;
  ← in scope: a, c (from proc2)
end;
begin
  proc2;
  ← in scope: a (from test)
end.

```

Subprograms: dynamic scope

- A dynamically-scoped identifier refers to the closest enclosing definition in that specific activation
 - Define scope based on the current state of program execution
 - Scope cannot always be determined by examining the program
 - E.g. some dialects of Lisp/APL and SNOBOL

- **E.g.**

```

int main(){ // basic scope
  int k;
  .....
}

```

Static Scope Implementation

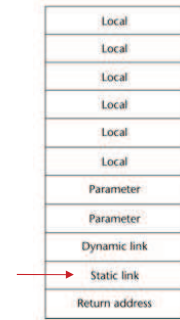
- The compiler/interpreter follow the static link in the activation record

- **E.g.**

```

void testfunc(){ // nested scope
  int a; // a enters scope;
  for ( int b=1; b<10; b++) { // b in scope
    int c; // c enters scope
    if( c < 10 ){ // d enters scope
      int d = a + c;
    } // d leaves scope
    ...
  } // b,c leave scope
} // a leaves scope

```



Dynamic Scope (recall)

- A dynamically-scoped identifier refers to the closest enclosing definition in that specific activation

- **E.g.**

```

program test;
var a : integer;

procedure proc1;
var b : integer;
begin
end;
  ← in scope: b (from proc1) a, c (from proc2)

procedure proc2;
var a, c : integer;
begin
  proc1;
  ← in scope: a, c (from proc2)
end;
begin
  proc2;
  ← in scope: a (from test)
end.

```

Dynamic Scope

- A dynamically-scoped identifier refers to the closest enclosing definition in that specific activation

- E.g.

```

fun g x =
let
  val inc = 1;
  fun f y = y + (inc);
  fun h z =
    let
      val inc = 2;
    in
      f z
    end;
in
  h x
end;

```

What is the value of g 5 using dynamic scope? g 5 = 7

Static vs. Dynamic scope

- **Dynamic scope makes it easier to access variables with lifetime, but it is difficult to understand the semantics of code outside the context of execution.**
 - Implicit parameter passing
- **Static scope is more restrictive – therefore easier to read – but may force the use of more subprogram parameters or global identifiers to enable visibility when required.**

Dynamic Scope Implementation

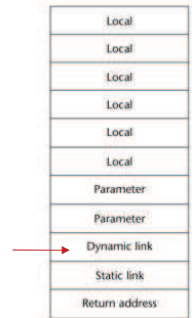
- The compiler/interpreter follow the dynamic link in the activation record

- E.g.

```

1:  a : integer    -- global declaration
2:  procedure first
3:    a := 1
4:  procedure second
5:    a : integer  -- local declaration
6:    first ()
7:  a := 2
8:  if read_integer () > 0
9:    second ()
10: else
11:   first ()
12: write_integer (a)

```

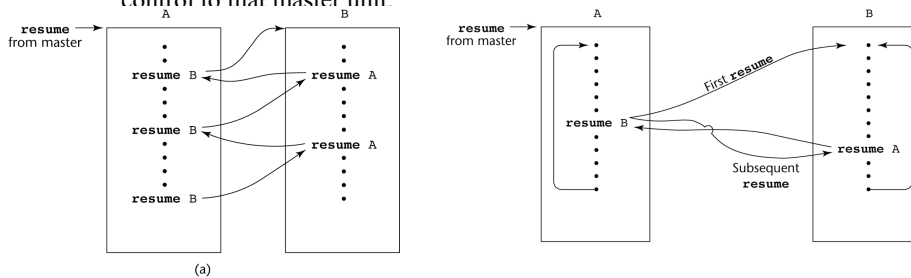


Optimizing Variable Access

- Problem: Accessing non-local names requires following links up the access link chain
- Solution for static scoping only:
 - Maintain a vector of currently-active static-chain frames.
 - Called the display
 - Pioneered in Algol60
 - Makes addresses directly accessible
 - Using a display:
 - If a procedure is at nesting depth n, it may have to follow n-1 static links to find variable addresses
 - Display is an array of pointers to stack frames
 - It must be maintained along with run-time stack

Subprograms: coroutines – an alternative

- **A special kind of subprogram:**
 - Caller and callee are on an equal basis
 - Multiple entry points
 - They can maintain their status between activation. Because of this, the invocation of a coroutine is called a *resume* rather than a call.
- **Coroutines are created in an application by a program unit called the master unit.**
 - When created, coroutines execute their initialization code and then return control to that master unit.



Review

Introduction

- What's a programming language?
- Syntax and semantics of PLs
- Levels: machine, assembly, high-level
- Translation: compilation, interpretation, hybrid
- Paradigms: procedural, functional, logic
- What's a "good" PL?

Formal Language Specification

- Motivation for specifying syntax formally
- Chomsky hierarchy
- Regular languages
 - regular expressions
 - regular grammars
- Context-free languages, context-free grammars
- Ambiguity
 - sources and solutions
 - associativity and precedence of operators
 - inherent ambiguity

Functional Programming/Scheme

- Scheme
- Functional programming
 - Spirit
 - Properties:
 - functions and evaluation, referential transparency, manifest interface principle, no assignment statement, no side effects, functions as first-class values, recursion vs iteration, higher-level language,
- Types of recursion
- let, let*, letrec
- Efficiency: tail recursion, accumulators
- Higher-order procedures: map, fold, reduce, apply, eval
- macro

Logical Programming/Prolog

- Spirit and motivation of logic programming
- Functions vs Relations
- First-Order Logic and Horn Clause
- Prolog clauses: facts, rules and queries
- Prolog vs Logical programming
- Execution of Prolog programs, Prolog search trees
- Unification
- Backward Chaining/Top-down reasoning/goal-directed reasoning
 - Backtracking
- Top-down vs bottom-up reasoning
- Negation as failure, closed-world assumption, (un)safe use of negation, guards
- Cut

Formal Proofs and Lambda Calculus

- Formal Proof based on Scheme programs
- Lambda calculus
 - Terms
 - Equivalence
 - Alpha-equivalence
 - Beta-equivalence and reduction
 - Currying
 - let statement
 - normal form, confluence
 - Fixed-point operator

Procedural Programming Design

- Name, variable, value
- Scope and life-time of a variable
- Subprogram
 - Parameter passing
 - By value
 - By result
 - By value-result
 - By reference
 - By name
 - Activation
 - More about scope with subprograms

What will/will be not covered in Final?

- Include:
 - General introduction about PLs
 - Formal language specification
 - Functional programming/Scheme
 - Formal proof
 - **General knowledge about lambda calculus only**
 - Logical programming/Prolog
- Will **not** include:
 - Procedural programming design
 - Detailed calculations of lambda calculus

Format of Final Exam

- Totally 8 questions
 - 6% of short answer questions, which covers all general knowledges mentioned in previous slide
 - 16% of formal language specification
 - About 28% of functional programming, Scheme and formal proofs
 - About 50% of logic programming, reasoning and Prolog
- Material:
 - Course lecture notes
 - Tutorials
 - Assignments, term test 1 and test 2 solutions
 - Old exam onlines
 - Text book: Mitchell 1,2,3,4,5,15
 - Online resources for Scheme and Prolog

Good luck with the final exams!