Principles of Programming Languages
Lecture 19

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig; Concepts of PL by Sebesta; Prog. in Prolog by Clocksin and Mellish; PL Concepts and Constructs by Sethi; ML for the Working Prog. By Paulson; PL Pragmatics by Scott

User-defined Operators

- **Operators such as +,-,*,… are defined for the language types**

- **Some languages enable the programmer to add new semantics for existing operators**

- **Enhances the writeability of the program but makes readability slightly harder**

User-defined Operators

```
class Cube {                          // C++
        public:
                Cube::Cube(float inx, float iny, float inz);
                Cube operator+ (const Cube &rhs);
                float Cube::getX();
                float Cube::getY();
                float Cube::getZ();
        private:
                float x;
                float y;
                float z;
   };
```

```
Cube::Cube(float inx, float iny, float inz) {
   x = inx; y = iny; z = inz;
}
Cube Cube::operator+ (const Cube & rhs) {
   float newx;
   if (x > rhs.x)   newx = x
   else  newx = rhs.x; ……
   return Cube(newx,newy,newz);
}
```

```
int main () {
Cube Compaq      = Cube(33.0,17.0,3.0);
Cube Powerbook = Cube(39.0,16.0,1.8);
Cube Combo     = Compaq + Powerbook;

}
```

User-defined Operators

```
class Car:                # Python
    def __init__(self,Brand,EngineSerial,carclr):
        self.Brand    = Brand
        self.Serial   = EngineSerial
        self.carclr   = carclr

    def __eq__(self,rhs):
        return self.Serial == rhs.Serial


if __name__ == "__main__":
    car1 = Car("Honda",111,"white")
    car2 = Car("Honda",111,"red")
    if car1 == car2:
        print "they are equal"
    else:
        print "they are not equal"
```

Assignment Statement

- **Syntax:**

  X = <expression>

  X := <expression>

  X <- <expression>

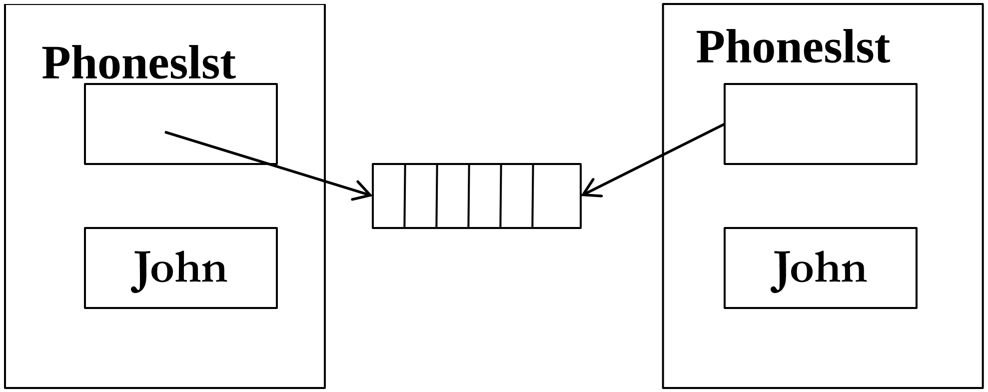
- **Semantics**
  - **Evaluate right hand side first, the result is assigned to left hand side**
  - **Make left hand side and right hand side equal**

- **With operator overloading, assignment gets a little bit more complicated**

Assignment Statement

- Assuming = operator is implemented for class Person

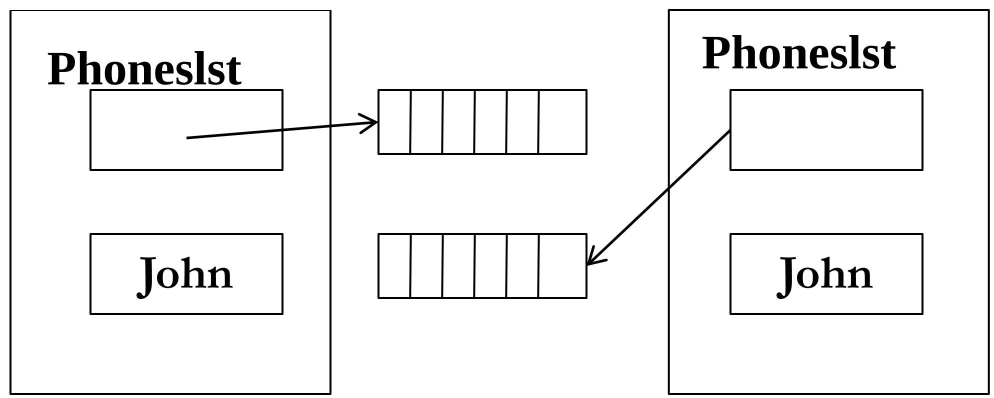
X = new Person(“John”) Y = X



Shallow Copy

Deep Copy

X = new Person(“John”) Y = X



Assignment Statement

- With operator overloading, assignment gets a little bit more complicated

- Deep copy is very powerful but very expensive.
  - E.g. a data structure with 1Mn nodes

X = Y // means creating another 1Mn nodes

Assignment Statement Variations

- **Syntax:**

  X,Y = Y,X

- **Semantics**
  - **Swap X and Y**
  - **Equivalent to**

    temp = X
    X = Y
    Y = temp

- **E.g.**
  - **Python**

Assignment Statement Variations

- Syntax:

  X,Y,Z = 10,20,30

- Semantics
  - Multiple assignment in one statement
  - Left most term in right side is evaluated first

- E.g.

  X,Y,Z = 10,X+2,Y+3

  // after evaluation X = 10, Y = 12, Z = 15

Variables, Operators & Expressions Questions

- What rules exist for naming variables?
- Which binding type the language supports?
- Does the language support short circuit evaluation?
- Does the language support bit-wise operators?
- Does the language support user-defined operators?
- If Assignment is overloaded for complex data structured of the language, is it shallow or deep copying?

Components of an Imperative Language

- **Data types**
- **Variables, operators & Expressions**
- **Iteration construct**
- **Branching construct**

→ Subprogram construct

- **Container construct**

Subprograms: introduction

- **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

- **Why is it a good idea?**

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

Subprograms: introduction cont'd

- **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

Subprograms: 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)**

Subprograms: 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;
```

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?

Subprograms: activation

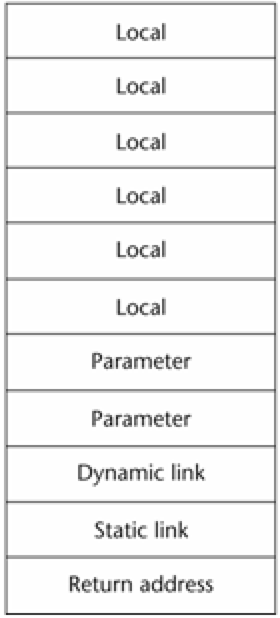
- **Each execution of a subprogram is called an activation.**

- **Life-time of a subprogram:**
  - Begins when control enters activation (call)
  - Ends when control returns from activation

Subprograms: 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



How would you access the non-local variables?

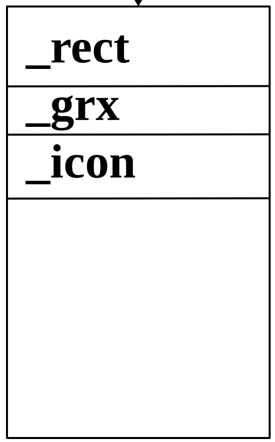
Subprograms: activation records

```
class Window{

  private Rectangle _rect;
  private Graphics  _grx;

  public void draw( ){
    int nX = ......;
    int nY = ......;

    _rect.draw(nX,nY,_grx);
    _icon.draw(img, _grx);
    ....
 }

}
```

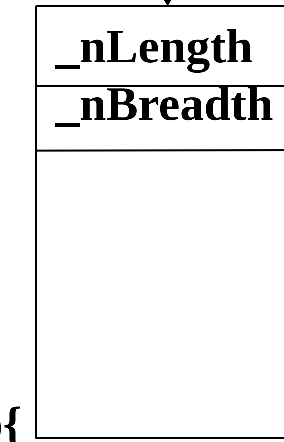
```
class Rectangle {
   private int _nLength,
                     _nBreadth;

   public void draw(int nX,
                              int nY,
                            Graphics grx){
      Color  clr = new Color("Black")
      grx.drawLine( nX, nY, nX+_nLength, nY);
   }
}
```

Subprograms: activation records

```
class Window{

  private Rectangle _rect;
  private Graphics  _grx;

  public void draw( ){
    int nX = ……;
    int nY = ……;

    _rect.draw(nX,nY,_grx);
    _icon.draw(img, _grx);
    ….
  }

}
```

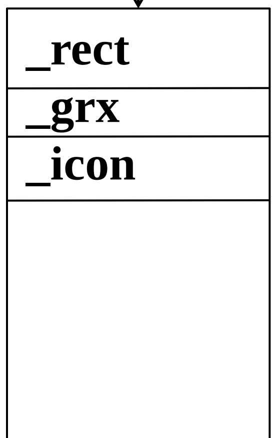


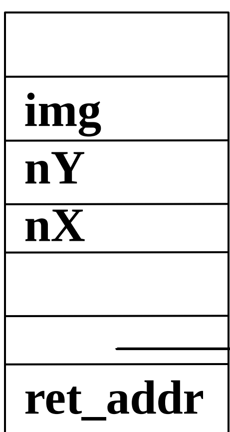

```
class Rectangle {
   private int _nLength,
               _nBreadth;

   public void draw(int nX,
                    int nY,
                    Graphics grx){
     Color  clr = new Color("Black")
     grx.drawLine( nX, nY, nX+_nLength, nY);
   }
}
```



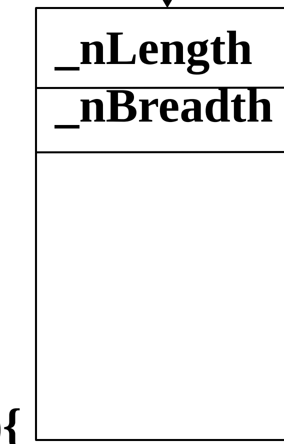
Subprograms: activation records

```
class Window{

  private Rectangle _rect;
  private Graphics  _grx;

  public void draw( ){
     int nX = ......;
     int nY = ......;

     _rect.draw(nX,nY,_grx);
     _icon.draw(img, _grx);
     ....
  }

}
```






```
class Rectangle {
   private int _nLength,
               _nBreadth;

   public void draw(int nX,
                    int nY,
                    Graphics grx){
     Color  clr = new Color("Black")
     grx.drawLine( nX, nY, nX+_nLength, nY);
   }
}
```



Subprograms: activation records

```
class Window{

  private Rectangle _rect;
  private Graphics  _grx;

  public void draw( ){
    int nX = ......;
    int nY = ......;

    _rect.draw(nX,nY,_grx);
    _icon.draw(img, _grx);
    ....
  }

}
```

```
class Rectangle {
  private int _nLength,
              _nBreadth;

  public void draw(int nX,
                   int nY,
                   Graphics grx){
    Color  clr = new Color("Black")
    grx.drawLine( nX, nY, nX+_nLength, nY);
  }
}
```

1067

_rect
_grx
_icon

img
nY
nX
ret_addr

_nLength
_nBreadth

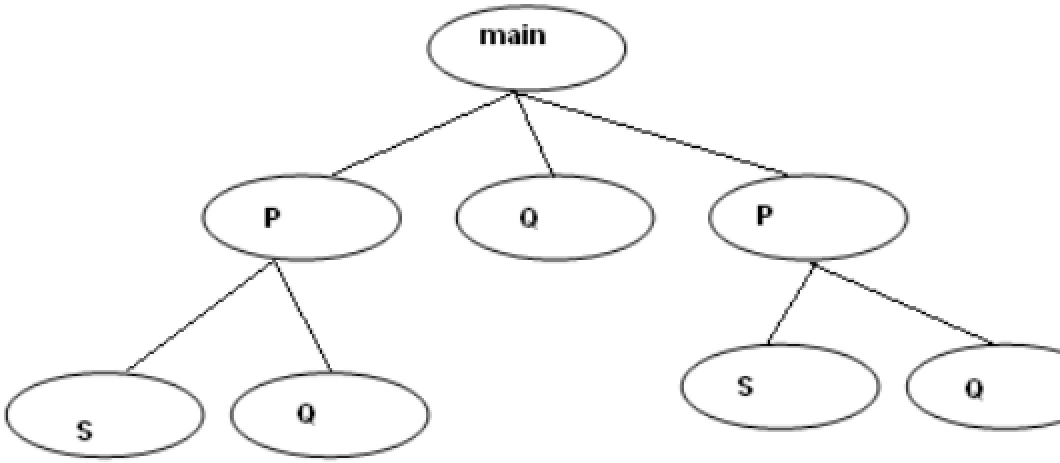
clr
grx
nY
nX
1067

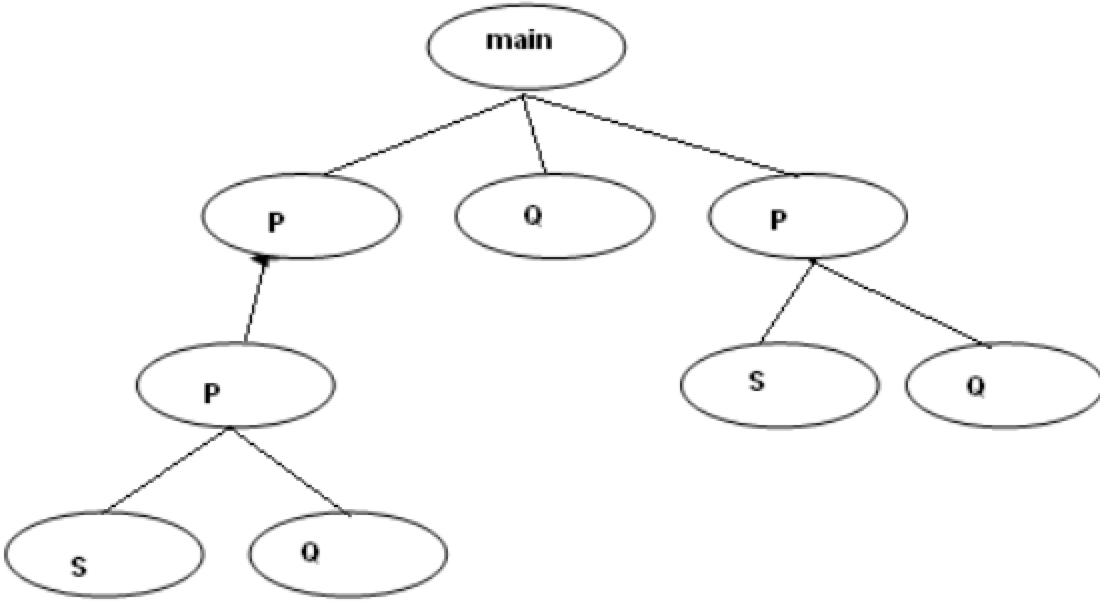
Subprograms: activation tree

- **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

Subprograms: activation tree example

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



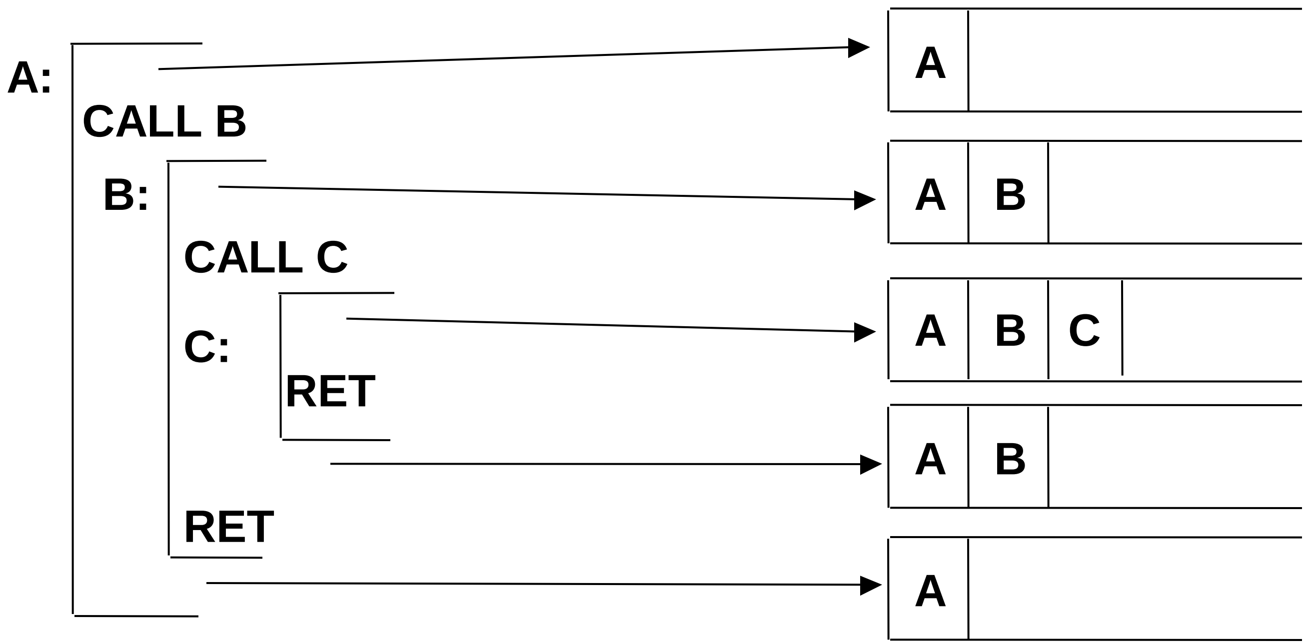


Subprograms: stack frames

- **Running a program corresponds to a traversal of one of its activation trees.**

- **We represent the traversal of the tree using a stack**
  - Each item in the stack is called a frame

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)

Subprograms : 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)

Subprograms: big picture

- **Sample memory layout**
  - A program with 4 sub-programs: A, B, C and D