Blocks, Procedures, and Modules

1

Blocks

- A block is a sequence of executable statements that are treated as a unit
  - while *expression* do *statement*
  - while *expression* do
    begin *executable statements* end
- A block is the scope-defining unit
  - *B*: declare
    declarations
    begin
    executable statements
    end *B*;

2

Environments

- An *environment* is a set of identifier bindings
- A *block* introduces an *environment*
- The *local environment* is the collection of bindings declared in current block
- The *nonlocal environment* is the collection of bindings not declared in current block

3

Scope

- variables declared in outer blocks are visible to inner blocks
- variables declared in inner blocks are *not* visible to outer blocks
- If an identifier is declared in both blocks of a nested block, the scope of the outer declaration is *eclipsed* by the scope of the inner declaration

4

Scope Rules in Ada

```
Outer: declare
   A, B: INTEGER;
   Inner: declare
      B: INTEGER;          -- hide Outer.B
      C: INTEGER;
   begin
      B := A;
      C := Outer.B
   end Inner;
begin
   A := B;
   A := Inner.C;           -- illegal
end Outer;
```

5

Static & Dynamic Scoping

- *Static scoping* or *lexical scoping* means that a procedure is called in the *environment of its definition*
- *Dynamic scoping* means that a procedure is called in the *environment of its caller*
- Using dynamic scoping *the choice of the name* for a variable can make or break a program
- Using static scoping a *strongly typed* language is possible

6

An Example

```
Outer: declare
   b: Boolean := true;          -- b in the definition's environment
   procedure P is
   begin
      print (b);                -- variable b not local to P
   end P;
begin
   Inner: declare
      b: Boolean := false;      -- b in the caller's environment
   begin
      P;                        -- invocation of P
   end Inner;
end Outer;
```

7

An Example

```
declare
   function discr(a, b, c: FLOAT) returns FLOAT is
   begin return (b**2 - 4*a *c); end discr;
   procedure roots (a, b, c: FLOAT; r1, r2: out FLOAT) is
   begin
      -- body of procedure roots
   end roots;
begin
   declare
      function discr(x, y, z: FLOAT) returns FLOAT is
      begin return (math.sqrt(x**2 + y**2 + z**2); end discr;
   begin
      -- a call to roots does not work as expected
   end;
end;
```

8

Activation Records

- An *activation record* captures the state of a procedure invocation
- An *instruction pointer* pointing to the current instruction
- *Local variables* in the local environment
- An *environment pointer* pointing to nonlocal environments

9

Activation Records

- It is natural to implement the organization of activation records with a *stack*
- An activation record is *pushed* onto the stack when a procedure is *invocated*
- An activation record is *poped* from the stack when a procedure is *exited*
- The *run-time system* of a language is responsible for managing this stack

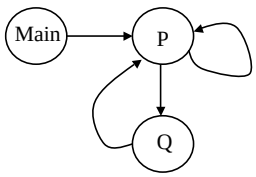
10

Accessing Nonlocal Environment

```
Main: declare
   procedure P is
      procedure Q is
      begin
         -- call P
      end Q;
   begin
      -- call P again recursively sometimes
      -- call Q sometimes
   end P;
begin
   -- call P;
end Main;
```

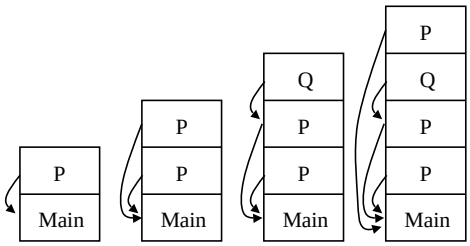
11

Call Graph



12

Run-Time Stack



13

Static Links

- *Static links* link the statically enclosing blocks on the stack according to their nesting in the program
- The *static distance* is the difference in static nesting between a variable use and its declaration
- The *offset* is the displacement of a variable in the activation record

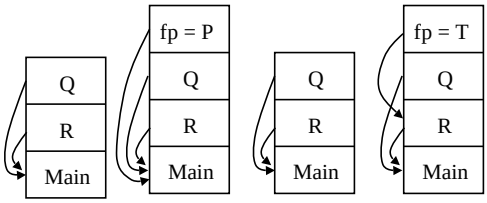
14

Procedural Parameters

```
MODULE Main;
   TYPE F = PROCEDURE (x: INTEGER);
   PROCEDURE P (x: INTEGER) = BEGIN (* … *) END P;
   PROCEDURE Q (fp: F) = BEGIN fp(5) END Q;
   PROCEDURE R ( ) =
      PROCEDURE T (x: INTEGER) = BEGIN (* … *) END T;
   BEGIN
      Q(P); Q(T);
   END R;
BEGIN
   R( );
END Main;
```

15

Procedural Parameters



16

Procedural Parameters

- What gets passed to Q to represent P or T
- What code must be emitted to implement the call fp(5)

17

Calling Sequence

- Transmit the parameters
- Save the return address
- Set the dynamic link
- Set the static link
- Start the execution of the called procedure

18

Closure

- A *closure* is a pair of *pointers*, to the *instructions* of a procedure and to the *nonlocal environment* of a procedure
- A *closure* is passed as the value of a procedural parameter

19

Returned Procedural Values

```
MODULE Main;
  TYPE F = PROCEDURE (x: INTEGER);
  PROCEDURE P (x: INTEGER) = BEGIN (* … *) END P;
  PROCEDURE R ( ): F =
    PROCEDURE T (x: INTEGER) = BEGIN (* … *) END T;
  BEGIN
    IF … THEN RETURN P ELSE RETURN T END;
  END R;
BEGIN
  EVAL R( );
END Main;
```

20

Calling Procedures

- Actual parameters are the arguments passed to a procedure
- Formal parameters are place holders used in the procedure
- Most programming languages use positional correspondence between actuals and formals
- Another correspondence is name association; this requires knowing the names of formal parameters

21

Calling Procedures

```
Procedure ACTIVATE (PROCESS: in PROCESS_NAME;
        AFTER: in PROCESS_NAME := NO_PROCESS;
        WAIT: in DURATION := 0.0;
        PRIOR: in BOOLEAN := FALSE);

ACTIVATE(X);
ACTIVATE(X, AFTER => Y);
ACTIVATE(X, WAIT => 60.0, PRIOR => TRUE);
ACTIVATE(X, Y, 10.0, FALSE);
```

22

Exception Handling

- An exception is a situation that makes it impossible for a program to proceed with normal processing
- Three major ways to raise an exception
  - An error detected by hardware
  - An error detected by system software
  - An error detected by user-program

23

Exception Handling

- Three ways to handle exceptions
  - Ignore them
  - Terminate immediately
  - Correct them and continue with normal execution
- Four steps for processing an exception
  - Detect the exception
  - Pass control to exception handler
  - Execute exception handler
  - Continue with normal execution

24

Exception Handling in Ada

```
A: declare
   I, J, K: INTEGER;
   singular: exception;
begin
   I := J * K;                          -- overflow exception
   if det = 0 then raise singular;      -- user-defined exception
   F := June;                           -- constraint_error exception
exception
   when NUMERIC_ERROR => …
   when singular => …
end A;
```

25

Exception Handling in Ada

```
procedure P is
   ERROR: exception;
   procedure Q is  begin
      R;   -- if ERROR raised here, situation (1)
   exception
      when ERROR => exception handler E1
   end Q;
   procedure R is  begin
      -- if ERROR raised here, situation (2)
   end R;
begin
   -- if ERROR raised here, situation (3)   Q;
exception
   when ERROR => exception handler E2
end P;
```

26

Encapsulation

- The aggregation of program components into larger units is called encapsulation
- Constructs for encapsulation are called modules
- Modules can be used as high-level entities, and used to build even higher-level entities
- Modules can be compiled separately

27

Data Abstraction

- An *abstract type* is a type *representation* bundled together with a fixed number of *procedures*
- The *representation* of the type is hidden from the users of the type
- The provided *procedures* are the only way in which the users can manipulate the representation of the type

28

Data Abstraction

- A type whose structure is hidden is called an *opaque* type
- A type whose structure is not hidden is called a *transparent* type
- A module that provides an abstract type is a *server*
- A program that uses the abstract type is a *client*

29

Packages in Ada

```
package RationalNumbers is
   type Rational is
      record
         Numerator: INTEGER;
         Denominator: INTEGER range 1 .. INTEGER'LAST;
      end record;
   function Equal (x, y: Rational) return BOOLEAN;
   function "+" (x, y: Rational) return Rational;
   function "-" (x, y: Rational) return Rational;
end RationalNumbers;
```

30

Packages in Ada

```
package body RationalNumbers is
  procedure SameDenominator (x, y: in out Rational) is
    NewDenominator: INTEGER :=
                  x.Denominator * y.Denominator;
  begin
    x.Numerator := x.Numerator * y.Denominator;
    y.Numerator := y.Numerator * x.Denominator;
    x.Denominator := NewDenominator;
    y.Denominator := NewDenominator;
  end SameDenominator;
  …
end RationalNumbers;
```

31

Packages in Ada

```
  function Equal (x, y: Rational) return BOOLEAN is
    u: Rational := x;
    v: Rational := y;
  begin
    SameDenominator(u, v);
    return (u.Numerator = v.Numerator);
  end Equal;
  function "*" (x, y: Rational) return Rational is
  begin
    return ( (
      Numerator => x.Numerator * y.Numerator,
      Denominator => x.Denominator * y.Denominator) );
  end "*";
```

32

Packages in Ada

```
  function "+" (x, y: Rational) return Rational is
    u: Rational := x;
    v: Rational := y;
  begin
    SameDenominator(u, v);
    return ( (
      Numerator => u.Numerator + v.Numerator,
      Denominator => u.Denominator) );
  end "+";
```

33

Packages in Ada

```
package RationalNumbers is
  type Rational is private;
  function Equal (x, y: Rational) return BOOLEAN;
  function "+" (x, y: Rational) return Rational;
  function "-" (x, y: Rational) return Rational;
private
  type Rational is
    record
      Numerator: INTEGER;
      Denominator: INTEGER range 1 .. INTEGER'LAST;
    end record;
end RationalNumbers;
```

34

Modules in Module-3

```
INTERFACE Rational;
  TYPE T = RECORD
      Numerator := 0;
      Denominator : [1 .. LAST(INTEGER)] := 1;
  END;
  PROCEDURE
    Equal (READONLY x, y: T): BOOLEAN;
  PROCEDURE Plus (READONLY x, y: T): T;
  PROCEDURE Times (READONLY x, y: T): T;
END Rational.
```

35

Modules in Module-3

```
MODULE Rational;
  PROCEDURE SameDenominator (VAR x, y: T) =
  VAR NewDenominator :=
                x.Denominator * y.Denominator;
  BEGIN
    x.Numerator := x.Numerator * y.Denominator;
    y.Numerator := y.Numerator * x.Denominator;
    x.Denominator := NewDenominator;
    y.Denominator := NewDenominator;
  END SameDenominator;
  …
BEGIN END Rational;
```

36

Modules in Module-3

```
PROCEDURE Equal (READONLY x, y: T): BOOLEAN =
VAR   u := x;  v := y;
BEGIN
  SameDenominator(u, v);
  RETURN (u.Numerator = v.Numerator);
END Equal;
PROCEDURE Times (READONLY x, y: T): T =
BEGIN
  RETURN ( T {
    Numerator := x.Numerator * y.Numerator,
    Denominator := x.Denominator * y.Denominator} );
END Times;
```

37

Modules in Module-3

```
PROCEDURE Plus (READONLY x, y: T): T =
VAR
  u := x;
  v := y;
BEGIN
  SameDenominator(u, v);
  RETURN ( T {
    Numerator := u.Numerator + v.Numerator,
    Denominator := u.Denominator} );
END Plus;
```

38

Modules in Module-3

```
INTERFACE Rational;
  TYPE T <: REFANY;
END Rational.

MODULE Rational;
REVEAL
   T = BRANDED REF RECORD
      Numerator: INTEGER := 0;
      Denominator: [1 .. LAST(INTEGER)] := 1;
  END;
BEGIN END Rational.
```

39

Modules in Module-3

```
INTERFACE P;
  PublicProtocol = OBJECT
    METHODS translate (h, v: INTEGER)
  END;
  Graphic < PublicProtocol;
END P.
MODULE P;
REVEAL
  Graphic = PublicProtocol BRANDED OBJECT
      x, y: Point.T;
  END;
BEGIN END P.
```

40

Modules in ML

```
signature RationalNumberSig =
sig
  type Rational
  val Eq: Rational * Rational -> bool
  val Add: Rational * Rational -> Rational
  val Mult: Rational * Rational -> Rational
end
```

41

Modules in Module-3

```
structure RationalNumbers: RationalNumberSig =
struct
  type Rational = {Numerator: int, Denominator: int}
  fun Num ({Numerator=x, Denominator=_}:Rational) = x
  fun Den ({Numerator=_, Denominator=x}:Rational) = x
  fun Eq (x, y) = (Num(x)*Den(y) = Num(y)*Den(x))
  fun Add (x, y): Rational =
    {Numerator = Num(x)*Den(y) + Num(y)*Den(x),
     Denominator = Den(x)*Den(y)}
  fun Mult (x, y): Rational =
    {Numerator = Num(x)*Num(y),
     Denominator = Den(x)*Den(y)}
end
```

42

Modules in Module-3

```
abstraction RationalNumbers: RationalNumberSig =
struct
  type Rational = {Numerator: int, Denominator: int}
  fun Num ({Numerator=x, Denominator=_}:Rational) = x
  fun Den ({Numerator=_, Denominator=x}:Rational) = x
  fun Eq (x, y) = (Num(x)*Den(y) = Num(y)*Den(x))
  fun Add (x, y): Rational = { … }
  fun Mult (x, y): Rational = { … }
  fun Rational (n, d) = {Numerator=n, Denominator=d}
end
```

43

Classes in C++

```
/* file rn.h */
class Rational {
private:
  int Numerator, Denominator;
  friend void SameDenominator(Rational *x, Rational *y);
public:
  Rational(int n, int d = 1)
    {Numerator = n; Denominator = d;}
  friend int operator ==(Rational x, Rational y);
  friend Rational operator +(Rational x, Rational y);
  friend Rational operator *(Rational x, Rational y);
};
```

44

Classes in C++

```
#include "rn.h"
void SameDenominator(Rational *x, Rational *y) {
  int NewDenominator =
    x->Denominator * y->Denominator;
  x->Numerator *= y->Denominator;
  y->Numerator *= x->Denominator;
  x->Denominator = y->Denominator = NewDenominator;
}

int operator ==(Rational x, Rational y) {
  SameDenominator (&x, &y);
  return (x.Numerator == y.Numerator);
}
```

45

Classes in C++

```
Rational operator +(Rational x, Rational y) {
  Rational Sum(0);
  SameDenominator (&x, &y);
  Sum.Numerator = x.Numerator + y.Numerator;
  Sum.Denominator = x.Denominator;
  return (Sum);
}
Rational operator *(Rational x, Rational y) {
  Rational Product(0);
  Product.Numerator = x.Numerator * y.Numerator;
  Product.Denominator = x.Denominator * y.Denominator;
  return (Product);
}
```

46

Classes in C++

```
class base {
public:
        void m1( ) { cout << "base::m1\n"; }
        void m2( ) { cout << "base::m2\n"; }
  virtual void m3( ) { cout << "base::m3\n"; }
};
class derived: public base {
public:
  void m2( ) { cout << "derived::m2\n"; }
  void m3( ) { cout << "derived::m3\n"; }
  void m4( ) { cout << "derived::m4\n"; }
};
```

47

Classes in C++

```
int main( ) {
  base O;
  derived P;
  base *Q = new derived;
  derived *R = new base;

  O.m1( ); O.m2( ); O.m3( ); O.m4( );  /* error */
  P.m1( ); P.m2( ); P.m3( ); P.m4( );
  P.base::m3( );
  Q->m1( ); Q->m2( ); Q->m3( ); Q->m4( ); /* error */
  Q->base::m3( );
  return 0;
}
```

48