Other Features of Procedures

- **overloading**
  - multiple procedures with same name
  - must differ in signature
    - result type part of signature?

- **overriding**
  - same signature
  - may still be able to access original method

- **default parameters**
  - can omit actual parameters
  - must be last parameters

```pl
procedure inc (item: in out integer; by: in integer := 1;
               mul: in integer :=1) is
begin
  item := item*mul+by;
end inc;
```

- **named parameters**
  - parameter association by name rather than position

```pl
inc(by=>2,mul=>3,item=>i);
inc(i,mul=>3);
```
Functions

• procedures which return a value
  - differentiation, e.g. none in C, ALGOL 60
• specifying result
  - assignment to function name vs return statement
• result can be considered extra result parameter
• return type
  - simple type – ALGOL 60, Java, Pascal
    • but also references
  - structured type – Ada, C, C++, FORTRAN 90, Algol 68
• order of evaluation
  - partial ordering
    • efficient code generation
  - side effects
• modifying parameters?!? For shame!
  - side effects
  - Ada, C++ const
Examples

- **Fortran 90**
  
  ```fortran
  FUNCTION TWICE(A)
    INTEGER TWICE
    INTEGER A
    TWICE = A + A
  END FUNCTION TWICE
  ```

- **Algol 60**
  
  ```algol
  integer procedure twice(a);
  value a; integer a;
  twice := a + a
  ```

- **C++**
  
  ```cpp
  int twice(int a) const {
    return a + a;
  } // twice
  ```

- **Java**
  
  ```java
  int twice(int a) {
    return a + a;
  } // twice
  ```

- **Pascal**
  
  ```pascal
  function twice(a : Integer):Integer;
  begin
    twice := a + a
  end; {twice}
  ```

- **Ada**
  
  ```ada
  function twice(a : in Integer)
  return Integer is
  begin
    return a + a;
  end twice;
  ```
Operators

- operators use infix, functions use prefix
- functions define new operations
- declaring new operators
  - really functions with different syntax

function “+”(today : in Day; n : in Integer) return Day is
begin
  return Day’val((Day’pos(today) + n) rem 7);
end;

- choice of operator
  - existing operator symbols
  - other symbols
- precedence rules
Storage Management

• storage for local variables
• static
  - pre-allocation - load-time (declaration-reference) binding
    • efficient access
  - size of each item known at compile time
  - no data item has multiple simultaneous occurrences
    ⇒ no recursion
  - FORTRAN
  - interesting notes:
    • since memory is pre-allocated, it's always being reserved, even when not in use
    • if a function is revisited, the old values may still be there
• stack-based
  - block-structured languages
  - binding at block entry/de-allocation at exit
    • stack of activation records (AR); run-time stack
  - storage only for locals of active methods
  - **addresses as offsets**
    • hardware support
  - *Ask me about return values*
program Main(output);
  var b, d : Integer;

  procedure z;
  var e, f : Integer;
  begin
    ...
  end {z};

  procedure y;
  var g, h : Integer;
  begin
    ...
z; ...
  end {y};

  begin
    ...
y;
    z; ...
  end.

Start program

Enter y

Enter z from y

Exit z

Exit y

Enter z from Main

Exit z
Activation Records

- local variables & **parameters**
- system information
  - return address
  - restore stack?
  - dynamic chain
- access
  - non-locals (inherited) not at fixed place
    - different calls give different stack structure
  - static chain
    - access is distance up chain plus offset
program Main(output);
var b, d : Integer;

procedure z;
  var e, f : Integer;
begin
  ...
end {z};

procedure y;
  var g, h : Integer;
  procedure x;
  var j, k : Integer;
begin
  ...
end {x};
begin
  ...
end {y};

begin
  ...
end;

end.
Run-Time stack

when \( z \) has been called from \( y \)  
when \( x \) has been called from \( y \)
Activation Records

• recursion
  • stack-based supports recursion
  • each invocation has its own storage
  • multiple ARs for different invocations of same procedure

```c
int factorial(int n) {
    if (n > 1)
        return n * factorial(n - 1);
    else
        return 1;
}// factorial
```
Activation Records

factorial(3)
→ 3 * factorial(2)
→ 3 * 2 * factorial(1)
→ 3 * 2 * 1

data area for the call
  factorial(1)

data area for the call
  factorial(2)

data area for the call
  factorial(3)

data area for the main program
Forward References

• Pascal
  - one-pass compiler
  - defined before use
    • scope sometimes extends from declaration to the end of the block; not starting at beginning of block
  - mutually-recursive procedures?
  - forward reference
    • signature without body
procedure operand(var ca : Character); forward;

procedure exp(var cb : Character);
begin
    operand(cb);
    while cb = '+' do
        operand(cb)
end {exp};

procedure operand(var ca : Character);
begin
    read(ca);
    if ca = '(' then
    begin
        exp(ca);
        if ca <> ')' then writeln(“missing brackets”)
    end else
    if ca <> 'a' then writeln(“missing operand”);
    read(ca)
end {operand};
Forward References

• function prototypes in C & C++

    void operand(char *);

• function specification vs function body in Ada

    function a(b : Integer) return Real;

    function a(b : Integer) return Real is
    begin ... end a;

• ALGOL 60, Algol 68 & Java don’t require defined before use
  – compiler must do multiple passes
Subprogram as Parameters

• functions and procedures passed as parameters
  - basic principle in functional languages
  - less widely used in imperative languages
• specification of function parameters
  - types of their parameters?

```plaintext
function slope(f(y : Real) : Real; x1, x2 : Real) : Real;
begin
  if x1 = x2
  then slope := 0
  else slope := (f(x2) - f(x1)) / (x2 - x1)
end {slope};
```
function straight(x : Real) : Real;
    begin
    straight := 2 * x + 1
    end {straight};

function tan(x : Real) : Real;
    begin
    tan := sin(x) / cos(x)
    end {tan};

slope(straight, 3.8, 3.83);
slope(tan, 3.8, 3.85);
Subprogram as Parameters

• procedure types and procedure variables
  – e.g. Modula-2

```modula2
TYPE realfunc = PROCEDURE(REAL) : REAL;

var fx : realfunc;
fx := straight;

PROCEDURE slope(f : realfunc; x1, x2 : REAL) : REAL;
BEGIN
  IF x1 = x2 THEN
    RETURN 0
  ELSE
    RETURN (f(x2) - f(x1)) / (x2 - x1)
  END
END slope;
```
• generics in Ada
  - compile-time parameters
  - generic instantiation
    • creates distinct functions

generic
  with function f(y: Real) return Real;
function slope(x1, x2 : Real) return Real;

function slope(x1, x2 : Real) return Real is
begin
  if x1 = x2 then
    return 0;
  else
    return (f(x2) – f(x1)) / (x2 – x1);
  end slope;

function straight_slope is new slope(f => straight);
function tan_slope is new slope(f => tan);
Structured Data

• data structures
  - components
• arrays
  - features
• records (structures)
  - features
• dynamic data structures
  - linked structures
  - pointers and dynamic storage allocation
Arrays

• attributes
  - element type
  - dimensionality
    • bounds
    • subscript type
  - Pascal
    type Matrix = array [1 .. 10, 0 .. 15] of Real;
    var a : Matrix;
• computable index
  - bounds error
    • Pascal → run-time error
    • Ada & Java → exception thrown
    • C & C++ → access to some location outside the array
  - notation () vs []
• multi-dimensional array vs array of arrays

type Row = array [0 .. 15] of Real;
type Matrix = array [1 .. 10] of Row;
var b : Matrix;

\[ \text{a}[2,3] \quad \text{vs} \quad \text{b}[2][3] \]

• Java
  - 0 lower bound
  - arrays are “objects”
  - multi-dimensional uses array-of-array model
Access to Elements

• efficient implementation
• mapping function
  - contiguous allocation
  - row-major vs column major
  - contiguous allocation: $a[i,j]$
    \[
    \text{base\_address} + \text{component\_size} \times \\
    (\text{row\_length} \times (i - \text{first\_lower\_bound}) \\
    + j - \text{second\_lower\_bound})
    \]
• access via pointers (C, C++)
  - array name as pointer constant
  - pointer increment
  - dangling pointer
  - parameter passing
    • array name is reference to array
double arow[16];

double total = 0.0;
for (int i = 0; i < 16; i++)
    total += arow[i];

double total = 0.0;
double *b;
b = arow;
for (int i = 0; i < 16; i++)
    total += *b++;
Name-Size Binding

- **static arrays**
  - bounds constant at compile time
  - efficient implementation
  - arrays of max size – waste space
- **semi-dynamic arrays**
  - bounds evaluated at block entry
  - implement as pointer to storage allocated on top of stack
  - slightly less efficient (extra dereference)

```haskell
import Data.Matrix

type Matrix = Array Integer (Integer, Integer) Real

a, b :: Matrix (1 .. m, 1 .. n);
```
• **dynamic arrays**
  - bounds evaluated at statement execution
  - variable may reference arrays of different size (different arrays)

```java
double[] arow = new double[16];

arow = new double[20];
```

• **extensible arrays**
  - bounds of same array may change at execution (extension)
  - expensive implementation
Arrays as Parameters

- bounds as part of type specification
  - Pascal
  - different bounds?

```pascal
type List = array[1 .. 20] of Real;

function sum(a : List) : Real;
var i : Integer;
  total : Real;
begin
  total := 0.0;
  for i := 1 to 20 do
    total := total + a[i];
  sum := total;
end {sum};
```
• conformant array parameters
  - bounds as implicit parameters
  - var parameters
  - value parameters – space allocation

```pascal
function sum(a : array[low .. high : Integer] of Real) : Real;
var i : Integer;
  total : Real;
begin
  total := 0.0;
  for i := low to high do
    total := total + a[i];
  sum := total;
end {sum};
```
• unconstrained arrays
  - Ada
  - array attributes

```pascal

type List is array(Integer range <>) of Real;

function sum(a : List) return Real is
  total : Real := 0.0;
begin
  for i in a'range loop
    total := total + a(i);
  end loop;
return total;
end sum;
```

• arrays as objects
  - Java
  - length attribute
• array parameters as pointers (C, C++)
  – pass attributes as extra parameters
    double sum ( const double a[], int length );
  or
    double sum ( const double* a, int length );
• parameter conformance
  – structural equivalence vs name equivalence
    type First = array[1..10] of Integer;
      Second = array[1..10] of Integer;
    var a : First;
    b : Second;
    c : array [1..10] of Integer;
    d, e : array[1..10] of Integer;
    f : First;
  – Pascal & Ada (name) vs C++ & Java (structural)
Arrays – Misc.

- aggregates
  - array literals or constructors
  type Vector is array (1 .. 6) of Integer;
  a : Vector;
  a := (7, 0, 7, 0, 0, 0);  Ada

  int a[] = {7, 0, 7, 0, 0, 0};  C++
  int a[6] = {7, 0, 7};

  int[] a = {7, 0, 7, 0, 0, 0};  Java

- slices
  - subsections of an array
  - array-of-array – row slices
  - general slices
  a(1 .. 3) := b(4 .. 6);

- array operators
  - PL/I and APL
• arrays as function results
  - also operator overloading

function add(left, right : Vector) return Vector is
  total : Vector;
begin
  for i in left’range loop
    total[i] := left[i] + right[i];
  end loop;
  return total;
end add;

c := add(a, b);       c := a + b;
• associative arrays
  - any value as index
  - key – data mapping (Map in Java)
  - implementation as hashtables

```java
Hashtable<String, PeopleInfo> persons = new Hashtable<String, PeopleInfo>();

PeopleInfo p1, p2, p3;
...code to give a value to p1, p2 and p3...
persons.put("J Smith", p1);
persons.put("F Bloggs", p2);
persons.put("A Brown", p3);

PeopleInfo inf = persons.get("F Bloggs");
```
Records & Classes

- record
  - non-homogeneous collection
  - components (fields) accessed by name
  - originally for file processing (COBOL)
  - later for data organization
    - Pascal, C, C++
  - C++
    - both data and function members
    - associate operations with data

- classes
  - derived from records
  - C++ class is same as struct
  - classes without methods - records
  - classes without instance variables - method libraries
Stack Example (Pascal)

- record type for CharStack
- methods taking CharStack parameter
- dot notation for field access
- no grouping or encapsulation

```pascal
type CharStack =
  record
    val : array [1 .. 20] of Char;
    head : 0 .. 20;
  end;

var a, b : CharStack;
  ch : Char;
```
procedure initialise(var stack : CharStack);
begin
  stack.head := 0;
end; {initialise}

function isEmpty(stack : CharStack) : Boolean;
begin
  isEmpty := stack.head = 0;
end; {isEmpty}

procedure push(var stack : CharStack; x : Char);
begin
  if stack.head = 20 then
    writeln('Error: stack full');
  else
    begin
      stack.head := stack.head + 1;
      stack.val[stack.head] := x;
    end;
end; {push}
procedure pop(var stack : CharStack; var x : Char);
begin
    if isEmpty(stack) then
        writeln('Error: stack empty');
    else
        begin
            x := stack.val[stack.head];
            stack.head := stack.head - 1;
        end;
end; {pop}

initialise(a); initialise(b);
push(a, 'f'); push(b, 'g');
pop(a, ch);
Stack Example (Ada)

- group record type and methods in a package
- `pop` cannot be function (only in parameters)
- control visibility
  - private types
    - := and =
  - limited private types
    - no operations
- can change representation
package Stacks is
    type CharStack is limited private;
    procedure push(st : in out CharStack; x : in Character);
    procedure pop(st : in out CharStack; x : out Character);
    function isEmpty(st : CharStack) return Boolean;
private
    type Values is array(1 .. 20) of Character;
    type CharStack is
        record
            val : Values;
            head : Integer range 0 .. 20 := 0;
        end record;
end Stacks;
Stack Example (C++)

• data members for representation
• member functions for operations
  – method bodies separate
• arbitrarily sized stack
  – constructor
  – destructor
class CharStack {
    char *val;
    int head;
    int maxSize;
public:
    CharStack(int size);
    ~CharStack() {delete val;}
    void push(char x);
    char pop();
    int isEmpty() const;
}; // constructor

CharStack :: CharStack(int size) {
    val = new char[size];
    head = -1;
    maxSize = size;
} // constructor

CharStack a(50), b(100);
Stack Example (Java)

- similar to C++
- garbage collection - no need for destructor

```java
public class CharStack {
    private char[] val;
    private int head;
    public CharStack ( int size ) {
        elts = new char[size];
        head = -1;
    } // constructor
    public void push ( char x ) { ... }; // push
    public char pop ( ) { ... }; // pop
    public boolean isEmpty ( ) { ... }; // empty
} // CharStack
```
Variant Records

- static typing is inflexible
  - e.g. class list with undergrad and grad students
  - cannot use array since must be homogeneous
- Pascal & Ada - variant records
  - fixed part
  - variant part
  - discriminant (tag field)
  - single type
  - access to variants
  - modification of discriminant
- object-oriented languages
  - replaced by inheritance and polymorphism
type Status = (undergraduate, postgraduate);
Student =
  record {fixed part}
    name : String;
  case kind : Status of {variant part}
    undergraduate : (advisor : String);
    postgraduate  : (course : String;
      supervisor : String)
  end;

var studentlist : array[1..3000] of Student;

write(studentlist[i].name);
if studentlist[i].kind = undergraduate then
  writeln(studentlist[i].advisor);
else
  writeln(studentlist[i].course,
    studentlist[i].supervisor);
Dynamic Data Structures

- nodes linked together by pointers
- nodes - records/classes
- dynamic allocation
- Pascal
  - record structure
  - pointer type
  - procedures
type Ptr = ^Node;

Node =
  record
    data : Char;
    next : Ptr;
  end;

procedure add(c : Char; var head : Ptr);
var p : Ptr;
begin
  new(p); p^.data := c;
  p^.next := head;
  head := p;
end; {add}
• Ada
  - package for association and visibility
  - initialization at creation
• C
  - self-referential type
    ```
    struct Node {
      char data;
      struct Node *next;
    }
    ```
  - similar to Pascal
• Java
  - no pointers but object variables are references
  - package visibility (class not public) makes Node visible to List but not outside package
class Node {
    char data;
    Node next;
    public Node(char c, Node p) {
        data = c; next = p;
    } // constructor
} // Node

class List {
    private Node head;
    public List() {
        head = null;
    } // constructor
    public void add(char c) {
        head = new Node(c, head);
    } // add
    ...
} // List
• C++
  - similar to Java
  - requires pointers
  - friend class
    ```cpp
    class Node {
        ...
        friend class List;
        ...
    }
    ```
  • List can access private members
Parametric Types

- e.g. stack - code same regardless of component type
- type as a parameter to another type
  - e.g. array of int, stack of char
- Ada
  - generic package
    - type (and other things) as parameter to package
    - generic instantiation
      - compile-time
      - defines a new package
- C++
  - class templates similar to Ada generics
  - automatic generic instantiation
generic
    type Item is private;
    size : Integer;
package Stacks is
    type Stacks is limited private;
    procedure push(st : in out Stack; x : in Item);
    procedure pop(st : in out Stack; x : out Item);
    function isEmpty(st : Stack) return Boolean;
private
    type Values is array(1 .. size) of Item;
    ...
end Stacks;

package StkChar is new Stack(Item => Character, size => 20);
package StkInt is new Stack(Item => Integer, size => 20);
• Java
  - generics added in 1.5
  - doesn’t do generic instantiation, all variants are the same class

```java
public class Stack <E> {
    private E[] val;
    private int head;
    public Stack ( int size ) {
        val = (E[])new Object[size];
        head = -1;
    } // constructor
    public void push ( E item ) { ... } // push
    public E pop ( ) { ... }; // pop
    public boolean isEmpty ( ) { ... }; // empty
} // Stack

Stack<Character> s;
Character c;
s = new Stack<Character>(100);
s.push(c);
```
Strings

• Pascal, Ada
  - as arrays of char
  - fixed length representation
  - only operations that don’t change length
• C, C++
  - arrays of character
  - fixed length representation
  - may vary up to defined length
  - string terminator character
• Java
  - String as library type
    • immutable
  - StringBuffer
    • more efficient processing if length to change
    • essentially extensible varying length string
• pattern matching
  - SNOBOL
  - Perl - regular expressions
• implementation
  - contiguous sequence of char→array
  - varying length?
    • move when must extend
    • use linked structure
    • combination
Sets

• Pascal
  - set of discrete type
  - set operations
    • inclusion (presence), add, remove
  - representation is bitset
• bit strings
  - bit-level operations
  - C, C++, Java - int values
  - Ada - Boolean arrays
Files

- files on secondary storage
  - external to program
- Pascal
  - file type
    - sequence of values of some type
    - sequential access
    - EOF
    - can have file of record type
    - local vs external files
- usually files are considered external and supported by I/O facility