CS 4100
Pascal Highlights

March 30, 2011

Based on slides by Istvan Jonyer
Book by MacLennan

Set Types

- Pascal provides facilities for sets

```
set of <ordinal type>
```

  - Ordinal type: enumeration, char, Boolean, subrange
  - Not integer or real

```
var S, T: set of 1..10;
```

  - S, T can hold a set of numbers between 1 and 10
    - vs a single number between 1 and 10:

```
var S, T: 1..10;
```


Efficiency of Sets

- Set types are restricted to be ordinal to be efficient

```
var S, T: set of 1..10;
```

  - S, T take only 10 bits to represent: 1 bit for each number
    - Bit = 0 means number is not is set
    - Bit = 1 means number is in set
  - `S := [1,2,3,5,7];`

	1	2	3	4	5	6	7	8	9	10
S =	1	1	1	0	1	0	1	0	0	0

Set Operations

- Initialization/Assignment
  []

```
T := [1..6];
```

- Membership
  in

```
if 4 in T then …
```

- Union, intersection, difference
  +, *, -

```
S * T, S + T, …
```

- Comparisons
  - Subset, equality, non-equality
  - <=, >=, =, <>
  - Proper subset (<) is not provided

Efficiency of Sets

- Sets are implemented using bit masks
  - Therefore, operations on sets can be implemented using logical operations
  - Intersection: logical *and*
  - Union: logical *or*
  - Difference: logical *exclusive or*
- Logical operations are the fastest a computer can do
- Memory efficiency: 1 bit per element

5

Sets

- Considered an example of elegance
  - High-level
  - Readable
  - Efficient
  - Secure

6

Elegance Principle

- Confine your attention to things that *look* good because they *are* good

7

Array Types

- Arrays are more general than Algol's
  - Base type of arrays can be non-primitives
  - Index types are introduced
  - Subscripts can be other than integers
    - Char, subrange, enumerated types

```
var A: array [1..100] of real;
var Occur: array [char] of integer;
var HoursWorked: array [Mon..Fri] of 0..24;

for day := Mon to Fri do
  TotalHours := TotalHours + HoursWorked[day];
```

8

Dimensions

- Only single-dimension arrays are allowed!!!
- However:
  - Base type of array can be another array!!!

```
var M: array [1..20] of array [1..100] of real;
```

  - Dereferencing: `M[3][5]`
- *Syntactic sugar*:

```
var M: array [1..20, 1..100] of real;
M[3, 5]
```

(Doesn't affect functionality, sweeter for human use.)

9

Static Arrays Only

- Algol's dynamic arrays are not supported
  - Type checking is done at compile time
  - Array bounds are part of array type
  - Hence, only static arrays are supported

10

Record Types

- Pascal provides the ability to group heterogeneous data
  - Versus homogeneous, using arrays
  - Can contain any other type, even other records

```
type person =
    record
          name: string;
          age: 16..100;
          salary: 10000..100000;
          sex: (male, female);
          hireDate: date;
    end;
    string = array [1..30] of char;
```

11

Dereferencing Records

- Dereferencing is done using the '.'

```
var today: date;
newhire.age := 25;
newhire.hireDate := today;
newhire.hireDate.month := Mar;
if newhire.name[1] = 'A' then …
employee[en].hireDate.year := 2004;
```

- Opening one record for multiple access

```
with newhire do
  begin
   age := 25;
   hireDate := today;
   hireDate.month := Mar;
  end;
```

12

Variant Records

- Pascal supports saving storage using variant records; allows alternative structures
  - Not all components of a record may be used at the same time
    - E.g.: Plane altitude and location on ground
  - C: union
    - Union is unsafe as it allows access to any member
  - Pascal attempts to solve this security problem
    - Access only members allowed by *tag field*
    - Initialization not required after tag value change, so type system can be circumvented after all…

13

Variant Record Example

```
type plane = record
  flight: 0..999;
  equipment: (B727, A343, B747);
  case status: (inAir, taxi, atTerminal) of
  inAir: (
      altitude:   0..999999;
      heading:          0..359);
  taxi: (
      location:   airport;
      runway:     runwayNumber);
  atTerminal: (
      parked:     airport;
      gate:       1..100);
end;
```

14

Pointers

- Pascal provides *typed pointers*, which are more secure than untyped ones

```
var p: ↑real;
   x: real;
   c: char;
begin
   new(p);
   p↑ := 3.14159;
   c := p↑;          {Illegal!}
end;
```

  - If P was untyped (`p: ↑pointer`), assignment to c would be allowed (and meaningless)

15

Type Equivalence

- Type checking requires that only variables with identical types can be compared/assigned to each other
- What does 'identical' mean?
  - Structural equivalence
    - Types having the *same structure* are identical

```
var x: record id: integer; w: real end;
var y: record id: integer; w: real end;
```

  - Name equivalence
    - Types having the *same name* are identical

16

Structural equivalence

```
type person = record id:integer; weight real; end
type car = record id:integer; weight real; end
var x: person;
var y: car;
x:= y;
```

- Legal by structural equivalence
- Probably don't want
- Name equivalence fixes this - person and car are different names

17

Name Structures

- Name binding mechanisms in Pascal
  - Constant bindings
  - Type bindings
  - Variable bindings
  - Procedure and function bindings
  - Implicit enumeration bindings
  - Label bindings

18

Constants

- Pascal introduces constant declarations

  const <name>=<constant>;

  const MaxArray = 100;
  - Allows the naming of constants in program
  - Numbers should not be used in programs
- Application of Abstraction Principle

19

Constants - Limitations

- Constant cannot be described by an expression
  - Illegal:

  const MaxArray = MaxData - 1;
- Expressions are not allowed in variable and type declarations
  - Illegal:

  var A: **array** [0.. MaxData – 1] **of real;**

20

Procedure Constructor

- Procedure declaration in Pascal has a strict structure

```
procedure <name>(<formals>)
   <label declarations>
   <const declarations>
   <type declarations>
   <var declarations>
   <procedure and function declarations>
begin
   <statements>
end
```

- Similar to Algol's
  - Scope essentially the same
    - Declarations: entire block including declarations and statements
    - Formals: local declarations and statements
- Names bound before they are used to support one-pass compilation

21