CS 4100
Pascal Highlights

From *Principles of Programming Languages: Design, Evaluation, and Implementation (Third Edition),* by Bruce J. MacLennan, Chapter 5, and based on slides by Istvan Jonyer

1

Chapter 5: Return to Simplicity: Pascal

- 1964 IBM: PL/I (Programming Language one) evolves to be a huge language
  - Union of Fortran, Algol and COBOL (rather than their intersection)
  - Swiss Army Knife Approach
  - Language is hard to use
    - Proponents say, enough to learn subset of PL/I
    - In reality, due to feature interaction, this is not possible
- Hard (or even futile) to design to design a language that is everything to all programmers

2

Extensible Languages

- Another approach is to design a small 'kernel' language and make it extensible
  - Kernel provides basic functionality
  - Extensibility should please everyone

3

Extensions: Operators

- Operator extension (vs overload)
  - Ability to create new operators
  - Example: symmetric difference of real numbers

```
operator 2 x # y;
  value x, y; real x, y;
  begin
    return abs(x – y)
  end
```

  - Allows:

```
    if l # r > 0 then …
```

- C++ has operator overload, variation of this

4

Extensions: Syntax

- *Syntax macros* allowed general syntax extension

```
real syntax sum from i = lb to ub of elem;
  value lb, ub;
  integer lb, ub, i; real elem;
  begin real s; s := 0;
   for i := lb step 1 until ub do
        s := s + elem;
   return s;
  end;
```

  - Allows:

```
   total := sum from k = 1 to N of Wages[k];
```

5

Issues with Extensibility

- Inefficiency
  - New syntax is translated to kernel constructs
  - Inefficiencies are magnified
- Poor diagnostics
  - Compiler errors are issued at kernel-level, which may be confusing to programmer
  - Language is hard to read, since people make up their own syntax
- Upside
  - Research on minimal requirement for PL's

6

Move Toward Simplicity

- Niklaus Wirth suggests changes to Algol-60
  - Non-numeric data types
  - Removing baroque features
  - Maintain efficiency (compile and run-time)
  - Can be taught systematically
- Implements Algol-W (after changes are rejected by Algol committee)
  - Evolves into Pascal, competed in 1970

7

Pascal - 3rd Generation

- Developed 1968-1970
  - 29 page report
- Revised 1972
- International Standard 1982
- Popular teaching language

8

Pascal's Syntax

- Pascal's syntax is like Algol's (p. 171)
- Major changes
  - program … end
  - procedure <declarations> begin <statements> end;
  - var, const, type
  - *for*-loop: simplified
  - case-statement

9

var, const, type

- const
  - Constant parameter declaration

```
const Max = 900;
```

- type
  - Type declarations introduced by "type"

```
type index = 1 .. Max;
```

- var
  - Variables declared after "var"

```
var
  i: index;
  sum, ave, val: real;
```

10

Data Structures

- Primitives are like Algol's
  - real, integer, Boolean, **char**
  - Char holds one character
    - Strings are arrays of chars

11

Enumeration Types: Issues

- Problem:
  - How to manipulate non-numeric data?
  - Mon, Tue, Wed,… Male/Female,
- Using number is very confusing (error prone)
  - today := 1; // Monday
  - tomorrow := today + 1; // next day
  - Issues: Sunday: 0 or 1? Start week with Monday?
- Assign numbers to meaningful variables
  - Mon = 1, Tue = 2, … male = 0, female = 1, …
- Security Issue: compiler allows meaningless operations
  - Year : = (month + male)/DayOfWeek

12

Enumeration Types

- Pascal introduces enumeration types

```
type
  month = (Jan, Feb, Mar, Apr, May, …);
  sex = (male, female);
var
  thisMonth : month;
  gender : sex;
begin
  thisMonth := Apr;
  gender := female;
```

- Supported operations for all enumerated types
  :=, succ, pred, =, <>, <, =, >, <=, >=

13

Enumeration Types

- Advantages
  - High level
    - Lets programmers write what they mean
  - Secure
    - Type checking is performed
    - No meaningless operations
  - Efficient
    - Allows optimization of storage
    - E.g.: Days of week can be stored in 3 bits

14

Subrange Types

- Improve security by allowing variable to take on values meaningful for their use only

```
var DayOfMonth: 1 .. 31;
type Weekday = Mon .. Fri;
```

  - Checking of valid values as part of type checking
  - Many programming errors come down to subrange violations (array out of bounds)
  - Efficient: Allows compact storage of variable
  - Subranges of discrete types are allowed
    - integer, enumerated, char

15

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;
```

16

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 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

17

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

18

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

19

Sets

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

20

Elegance Principle

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

21

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];
```

22

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.)

23

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

24

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;
```

25

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;
```

26

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…

27

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;
```

28

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)

29

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

30

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

31

Name Structures

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

32

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

33

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;
```

34

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

35

Mutual Recursion

```
procedure P(...);
begin
   .
   Q(...);
   .
end;
procedure Q(...);
begin
   .
   P(...);
   .
end;
```

36

Procedure Constructor

- Opposite of top-down
  - Uppermost procedures first, then lower ones they call
- Mutual recursion
  - Cannot define both procedures before one is called
- Pascal's solution
  - "`forward`" declaration of procedures allows recursion, and observation of structure principle

```
procedure Q(...); forward;
```

37

No Blocks

- Pascal eliminates Algol's blocks
  - Compound statements but no blocks
  - Variable declarations are only allowed before *begin* in procedures and functions
  - Simplifies name structures
  - Complicates efficient use of memory
    - Storage shared only between disjoint procedures

38

Control Structures

- Pascal includes more control structures than Algol-60, but they are simpler
  - Provides simple I/O
  - Introduces more structured control structures (structure principle)
    - 1-entry point 1-exit point controls
  - Includes goto (rarely needed)
  - Includes recursive procedures

39

`for`-Loop is Austere

- Pascal removes the baroque for loop, in favor of one simpler than Algol's

```
for <name> := <exp> {to|downto} <exp> do
    <statement>
```

  - Only step size of 1 is allowed (+1 & -1)
    - May be too restrictive
  - Bounds are computed once, on entry
    - Called *definite iterator*
      - Always executes a definite number of times unless goto

40

Leading & Trailing Decision Loops

- Indefinite iterators:
  - Loop is controlled by condition, not counter
  - Condition is tested each time
    - Versus pre-computed in *for*-loop
- Leading Decision loop

```
while <condition> do <statement>
```

- Trailing Decision loop

```
repeat <statement>+ until <condition>
```

- Mid-Decision loop
  - Can be implemented using "**`while true do`**" and **`goto`**

41

Pascal's `case`-Statement

- Pascal introduces the labeled, structured case-statement

```
case <expression> of
  1:    begin <statements> end;
  2, 3: begin <statements> end;
  4:    begin <statements> end;
  ...
end case;
```

  - This case-statement is *self-documenting*

42

Labels in `case`-Statement

- Case labels can be labels from enumeration types

```
case nextFlight.status of
  inAir:      begin <statements> end;
  onGround:   begin <statements> end;
  atTerminal:begin <statements> end;
end case;
```

43

Parameter Passing

- Pass by value
  - Exactly like before, in Algol-60
- Pass by reference
  - Allows output parameters
  - Replaces pass by name
  - Only allows meaningful variables to be written into (unlike Fortran)

44

Pass as Constant

- Pass as constant was originally specified instead of pass by value
  - Like pass by value, but parameter could not be modified in callee
    - Safe
  - Implemented as pass by reference
    - Efficient
  - Replace by pass by value, since pass as constant can be circumvented using scoping (p 202)
    - C++ provides this functionality by explicit pass by reference and *const* definitions (`f(const int &a)`)

45

Two Orthogonal Issues

- Input vs output parameters
- Copy value vs pass address
- Decisions should be separated

46

Goals

- Main goal: good teaching language
  - Reliability
  - Simplicity
  - Efficiency
- Successful!
- Third Generation

47