Section 5: Pascal

Evolution of Software Languages

Theo D'Hondt

Bachelor of Computer Science
Faculty of Sciences and Bio-Engineering Sciences
Vrije Universiteit Brussel
Academic Year 2015-2016
One direction was: ‘universal languages’:

- ALGOL 68 was proposed by the academic community as ‘the’ universal programming language
- IBM had proposed PL/I as ‘FORTRAN VI’
- COBOL had the quasi-monopoly in the ‘commercial’ world

These languages are ‘big’ and unmanageable; PL/I is considered as an example of the ‘Swiss Army knife‘ approach
Extensible languages

• An other approach are ‘extensible languages’: a kernel and constructors

• There was a need for a simple kernel (e.g. ALGOL 60-subset)

• There were several mechanisms for language extension constructors
Definition of additional operators via function definitions:

```pascal
operator "#"(x, y);
value x,y;
integer x,y;
begin
    return abs(x-y)
end
```
Extensible languages

Using ‘syntax macro's'

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

Total := sum from k=1 to 10 of x[k]
```
Extensible languages

- Drawback:
  - ‘extensible languages’ are inefficient: a variable syntax makes code-optimisation impossible
  - it is nearly impossible to link syntax errors in an ‘extensible languages’ program to the original program text
  - are used with ‘extensible datatypes’ (and possibly macros)
Pascal was designed as:

- successor to ALGOL 60
- alternative to ALGOL 68
- extremely simple language
- a viable alternative to FORTRAN:
  - powerful and still efficient
  - general purpose
- with symbolic data structures
STRUCTURED PROGRAMMING

O.-J. DAHL
Universitet i Oslo,
Matematisk Institut,
 Blindern, Oslo, Norway

E. W. Dijkstra
Department of Mathematics,
Technological University,
Eindhoven, The Netherlands

C. A. R. Hoare
Department of Computer Science,
The Queen’s University of Belfast,
Belfast, Northern Ireland

1972
ACADEMIC PRESS
LONDON AND NEW YORK

Lecture Notes in Computer Science
Edited by G. Goos and J. Hartmanis

18

Kathleen Jensen
Niklaus Wirth

PASCAL
User Manual and Report

http://dl.acm.org/citation.cfm?id=1243380&CFID=588262404&CFTOKEN=69334518

Pascal was both a language and a system:

• short report (as in ALGOL 60)

• capable of supporting itself

• the first Pascal-system ===> compiler if ±6000 lines Pascal (on a CDC), public domain

• later the p-system, also public domain, was the basis for most µcomputer versions
Innovations in data types

Enumeration:

- CHAR became a ‘first-class type’
- Enumerated types implement the security principle
- the ordering relation is generalised to symbolic data:
  
  \[
  = \leftrightarrow \leq \geq \lessdot \overset{\text{PRED}}{\text{SUCCE}} \text{ORD}
  \]

- Enumerated types introduce the first real abstract data types
- Enumerated types are efficient with respect to both time and space:
  
  \[
  \text{colour} = (\text{red, orange, green, blue, indigo, violet})
  \]

only 3 bits/value!
Innovations in data types

Subranges:

• subrange types are a further implementation of the security principle

• subrange types ensure space optimisation

\[\text{small} = 0..15\]

only four bits/value

• subranges also implement the abstraction principle
Innovations in data types

Sets:

- almost perfect example of a data structure
- high level
- very compact: 1 bit per element
- very efficient

```
VAR s1, s2, s3: SET OF 'A'..'Z';
...
s1 := s2 + s3
...
...c IN s1...
```

SA1  s2
SA2  s3
BX6  X1+X2
SA6  s4

SA1  c
SA2  s1
SB7  X1-'A'
LX2  X2,B7
Innovations in data types

Arrays:

• more consistent than ALGOL 60

\[ a = \text{array} \left[ \text{index} \right] \text{of component} \]

\[ a: \{\text{indexes}\} \rightarrow \{\text{components}\}: i \mapsto a[i] \]

• enormously safer than in FORTRAN
Innovations in data types

Design criteria for arrays:

- All types are fully determined at compile-time.
- Both index and component are types, hence arrays are static.
- An array is a type.
- Parameter binding is strong, hence formal array types do not allow parametrised index types.

The original Pascal is therefore in practice difficult to use; reason: conflict between design criteria that seem very reasonable taken separately.
Records:

• inspired by COBOL

• arrays are dynamically indexed and have uniformly typed components; records are statically indexed but can have variously typed components

• record-qualification can be analysed by the compiler, hence efficient

• the with-command ‘opens up’ an environment and is the precursor of e.g. the modula2 import statement
Innovations in data types

Records with variants are an important abstraction of the COBOL ‘redefines’ but are not 100% safe, even with an explicit tag field.

definitions:

type
    Complex = record
        case kind: (cartesian, polar);
        cartesian:(R, I: real);
        polar:(A, R: real)
    end

var
    horrible = record
        case boolean of
            true:(p: ^char);
            false:(seg,off:integer)
        end

Innovations in data types

Pointers and files:

• Inspired by pointers in PL/I, C, ...

• Combined with strong typing => typed pointers

• File-buffers are inspired by file description and data records in COBOL, but adapted to the notion of (explicit) strong typing
Name binding

• labels, constants (explicitly and enumerated), types, variables, functions en procedures

• constants are a (poor) compile-time substitute for dynamic arrays in ALGOL 60

• the record allows the binding of a name to a composition of data; the procedure allows the binding of a name to a composition of commands

• procedures can be nested and lexical scope applies
Name binding

- records and procedures introduce visibility rules
- declarations apply downstream within surrounding block
- formal parameters are visible in own body
- homonyms are allowed between separate levels
- Pascal was designed for a ‘single-pass’ compiler, hence only downstream visibility with procedures and forward pointer declarations
Innovations in control

- disappearance of the ALGOL block
- no solution for the ‘missing end’
- simplification of the for:
  - allowing an abstract for-index
  - eliminating side effects
  - generating efficient code
- explicit while and repeat
- structured switch becomes case
  - much more consistent
  - no loss of efficiency
  - contribution to abstraction
Innovations in procedures

- no call-by-name but call-by-reference
- call-by-name via functional parameters
- call-by-value as in ALGOL 60
- call-by-constant would have been more efficient, but ...

```pascal
program Sick;
   var z:integer;
procedure ZeroEqualsOne(x: integer);
   begin
      writeln(x);
      z := 0;
      writeln(x)
   end;
begin
   x := 1;
   ZeroEqualsOne(x)
end.
```
Innovations in procedures

• procedural parameters are ‘second class objects’
• return-value in a function: same problems as in ALGOL 60

```pascal
function Absurd: integer;
var x , Absurd: real;
begin
  x := Absurd;  {recursion doesn’t work}
  Absurd := x   {return doesn’t work either}
end;
```
Conclusion

• Pascal is also a milestone

• Pascal satisfies the original design objectives (simple, consistent, efficient didactical language)

• Pascal is the basis for many subsequent languages (e.g. ADA)

• Pascal was extended with essentially production-oriented elements (e.g. UCSD pascal)

• Almost all criticism of Pascal is caused by an improper use of Pascal; it is remarkable that Pascal has been successfully used for a number of tasks that it was never designed for
Assignment

1. Investigate functional/procedural parameters in Pascal

2. Evaluate the Pascal program on the next pages
program DSW;

const
  LowAddress = -32767;
  HighAddress = -32737;
  Null = 0;

type
  Address = LowAddress..HighAddress;
  ConsCell = record
      Left: integer;
      Right: integer
  end;

var
  M: array[Address] of ConsCell;
  Root: integer;
  freeList: integer;

procedure collectGarbage;
var
  flag: array[Address] of (tryLeft, tryRight, backLeft, backRight);

procedure markAllFree;
var
  adr: Address;
begin
  for adr := LowAddress to HighAddress do
    flag[adr] := tryLeft
  {markAllFree}

procedure sweepFromRoot (adr: integer);
var
  this, previous: integer;
  continue: boolean;
Example

procedure rotate (var A, B, C: integer);
var
  X: integer;
begin
  X := A;
  A := B;
  B := C;
  C := X
end;{rotate}

begin
  if adr < 0 then
  begin
    this := adr;
    previous := Null;
    continue := true;
    while continue do
      case flag[this] of
        tryLeft:
          if M[this].Left < 0 then
            begin
              flag[this] := backLeft;
              rotate(previous, this, M[this].Left)
            end
        else
          flag[this] := tryRight;
        tryRight:
          begin
            flag[this] := backRight;
            if M[this].Right < 0 then
              rotate(previous, this, M[this].Right)
          end;
      end
    end
  end
end
Example 3

```pascal
backLeft, backRight:
begin
  continue := (previous = Null);
  if continue then
    case flag[previous] of
      backLeft:
        begin
          rotate(M[previous].Left, this, previous);
          flag[this] := tryRight
        end;
      backRight:
        rotate(M[previous].Right, this, previous)
        end
    end
  end
end; {sweepFromRoot}

procedure releaseRemainingFree;
var
  adr: Address;
begin
  for adr := LowAddress to HighAddress do
    if flag[adr] = tryLeft then
      begin
        M[adr].Left := freeList;
        freeList := adr
      end
end; {releaseRemainingFree}
```
Example 4

begin
  markAllFree;
  sweepFromRoot(Root);
  releaseRemainingFree;
end; {collectGarbageNR}

function NewCell: integer;
  var
    oldFreeList: Address;
  begin
    if freeList = Null then
      collectGarbage;
    if freeList = Null then
      Writeln('Memory overflow');
    NewCell := freeList;
    oldFreeList := freeList;
    freeList := M[oldFreeList].Left;
    M[oldFreeList].Left := Null;
    M[oldFreeList].Right := Null
  end; {NewCell}

procedure Initialize;
  var
    adr: Address;
  begin
    for adr := LowAddress to pred(HighAddress) do
      M[adr].Left := succ(adr);
    M[HighAddress].Left := Null;
    freeList := LowAddress;
    Root := Null
  end; {Initialize}

begin
  {test code here}
end.