PASCAL

The Algol language introduced many new concepts into language design, and as a consequence, spawned a number of new languages (e.g. PL/I) all of which were very complex and unmanageable. Pascal was a teaching language designed to reduce this burgeoning language complexity.

Another idea at the time was to develop *extensible languages*, based on a small kernel of functionality. Extensions added application specific functionality. However, extensible languages turned out to be very inefficient, since variable extensions made parsing and compiling difficult. As well, extensions were reduced to kernel functions, adding another level of language complexity which could not be optimized. Since kernel errors were not expressed in the application specific language, diagnosis was not transparent.

Pascal combines simplicity with generality, a result of learning how the innovations of Algol can be efficiently and elegantly combined.

New concepts introduced in Pascal include:

- *enumeration types*
  using names rather than numbers to represent finite sets
  compiled like an array in contiguous memory, efficient
  e.g.: type DayOfWeek = {Mon, Tues, Wed, Thur, Fri, Sat, Sun}

- *subrange types*
  for contiguous subgroups
  e.g.: type Weekday = {Mon .. Fri} of DayOfWeek

- *set types*
  for arbitrary collections
  efficient, encoded as binary array indicating set membership
  bit level operations for union and intersection
  subsets easy to define, e.g.: S := [1,3,5,6]
  e.g.: type set of 1..9

- *strong typing of arrays*
  static array types since typing is determined at compile-time
  cannot write dynamic array manipulation procedures

- *name structures* include bindings for
  constants, types, variables, functions, labels

- *case statement*

Pascal's control structures embody the principles of structured programming. Control structures have one entry and one exit point. All statements can be compound. Pascal eliminated the idea of block structure, a precursor to structured programming.

C

The C language mixes characteristics of several language generations, it is an amalgam of structured high-level features, low level implementation features, and even machine-level features. It lacks support for nested procedures and modular programming, and is machine architecture specific. It's creator Dennis Ritchie says: "C is quirky, flawed, and enormously successful."

Summary of Block Structuring

Activation Records

An activation record represents the state of a procedure or function call. It holds all the information relevant to one execution unit, or activation. Thus a procedure consists of

1. the program code — fixed, static, not part of the activation record
2. the activation record — dynamic, keeps track of context and computational results

The activation record itself consists of

1. ip: the *instruction pointer* to the next statement to be executed after the procedure call returns. Also called the *resumption address*.

2. ep: the *environment pointer* identifies the bindings and scope of variables

   2a: local context: names declared by the procedure;
   local parameters and variables.
   Also the *static link* to the nonlocal scope

   2b. nonlocal context: names declared by *surrounding* procedures
   Also the *dynamic link* to the activation record of the caller.

A static link is required to locate the environment of the definition. A dynamic link is required to locate the environment of the caller.

Environments

An *environment* is simply a binding list of names and their values. Which names are in an environment is determined by the scoping rules of the language. *Scoping rules* define how to locate the values of names which are not immediately local to the procedure being executed. The *context of a procedure* is the set of names declared by that procedure, together with the names declared in the surrounding procedures, with "surrounding" being defined by scoping rules. Every name and variable is local to some procedure, the default being the top level, or main procedure. The activation record of that procedure contains the name and its binding.

FORTRAN activation records are compiled statically; names are assigned a permanent memory address. Languages which permit recursion require dynamic creation of activation records, and dynamic searching of the context for variable bindings, since more than one copy of a procedure may be active at the same time. Since a primary cost in computation is finding variables and values, it is impractical to dynamically search for the context of every variable. Instead a *two-coordinate method* is used:

1. the *ep* accesses the activation record of the current environment (calling procedure).
2. an *offset* locates the variable within the activation record

Scoping

Static scoping: a procedure executes in the environment of its definition (syntactic structure)

Dynamic scoping: a procedure executes in the environment of its caller (semantic structure)

Procedure Activation

To activate a procedure:

1. Save the state of the caller
   Put the current ip in the caller's activation record.
   The local ep is already in the caller's activation record.
   The nonlocal ep is already in the static link of the caller's activation record.

2. Create an activation record for the called procedure
   Put the actual bindings of parameters in the parameter part.
   These must be evaluated first, returning either
   1) a value (call by value), or
   2) an address (call by reference), or
   3) a thunk (call by name) [thunk=address returning function]
   Add the static link to the environment of definition.
   The ip is not relevant until the called procedure calls a procedure itself.
   The dynamic link points to the activation record of the caller.

3. Enter the called procedure in the context of the new activation record.

To exit a procedure, basically reverse the above process.

1. Delete the called procedure's activation record.
2. Restore the state of the caller.

Closures

In languages which can pass procedures as parameters, the procedure is passed as closure. A ***closure*** is a ip-ep pair:

1. The ip contains the entry address of the actual procedure.
2. The ep contains a pointer to the environment of definition.

In order to *pass functions* as parameters, the entire activation record approach must be changed, leading to a functional programming regime.

Blocks

A *block* is a container for a collection of operations. Technically, blocks are implemented the same as are procedures. Blocks are degenerate procedures; the ep and its dynamic link are not needed in the block activation record.

Displays

An alternative method to searching up a scoping chain for nonlocal variables is to have all accessible contexts stored in an array which is searched directly when a nonlocal is referenced. This produces constant look-up times.

Efficiencies

Both static and dynamic environments can be nested, often many levels deep. To locate a nonlocal variable, the static environment must be searched outwardly. This is very inefficient. Here is a listing of the costs of various static operations. *SD* stands for the static distance between the context using a variable or procedure and the context defining the variable or procedure.

Operation	*Memory references*	*Display references*
local variable	1	2
variable access	SD + 1	2
procedure call	SD + 3	6
procedure return	2	5
pass procedure	SD + 2	
formal procedure call	5	
goto	SD	

We can see than when operations are deeply nested, display is better than searching scoping chains.