F) Typed “Pointer”: Pascal is among the pioneer languages to introduce typed pointers (memory locations), for linked structures. C and C++ also have typed pointers. Java has no pointers, but it has references (pointers to structures, class instances, no need to dereferences, and it is nonsense to apply arithmetic ops on them!).

In PASCAL: \[ \text{var } p: ^\text{real}; \quad x: \text{real}; \quad c : \text{char}; \] \[ \begin{align*}
\text{begin } \text{new}(p); & \quad p^ := 3.14159; \quad c := p^; \quad \{ \text{compiler error!} \}
\end{align*} \]

In C&C++:
\[ \text{int } *\text{ptr}; \quad \text{int } \text{count, init}; \quad *** \quad **** \]
\[ \text{ptr }= \&\text{init}; \quad \text{count }= *\text{ptr}; \]

They also have a non-types pointers “\text{void } *\text{ptr}” as generic pointers of type any; in case we need to have a function to deal with memory blocks of byte/words (any type). Just define the formal parameters to be “\text{void } *\text{ptr}”, to deal with any actual parameter pointer type you send to the function!

Questions: Is the use of non-types pointers secure? What type of tradeoff is that of the “\text{void } *\text{ptr}” facility?

We can have pointers to any type, not only basic types (int, real, ..).
Ex: \[ \text{var } p: ^\text{plane}; \quad (* \text{plane is a record type*}) \]
\[ **** \]
\[ p^\text{.parked }:= \ldots. \]

- A pointer with the value “\text{nil}” points to nowhere! (what is the type of a nil pointer?)

( C&C++ use value 0 instead of nil)

- A common mistake by programmers is to equate the definition of a “pointer” with and “address”, why is wrong to do so?

- A “pointer” is a high level abstract concept, as Java views it a “reference” to an object. Languages like Ada-83, Modula-3, and Pascal has only one way to create a new pointer value via a built-in function “new()” that allocate an object (of the pointer type) and returns a pointer to it.
- But, an address is a low-level concept of the actual word location in memory. In C, C++, Ada-95 one can create an address to a nonheap object (simple, non-composite, word) using an “address of” operator.

- After finishing the use of storage locations allocated in the code; C, C++, Pascal, Modula-2 require that the programmer “explicitly” reclaim (free) them. Lisp, ML, Modula-3, Ada, and Java languages have implicit automatic reclamation of unused objects (implicit garbage collection).

- Discuss implicit versus explicit garbage collection (GC) of unused spaces. Implicit GC: how often? Constant overhead. Explicit: Trusting the user to do it! If user prompt in doing it → Efficient, but insecure if the user de-allocates an already used space (creating dangling reference)!

**No dynamic arrays, or blocks, in PASCAL!! (why?)**
No need for dynamic arrays because we have the typed pointers (efficient no range check at run time). No need for blocks because of the associated run time overhead of AR allocation/de-allocation, upon handling blocks’ entry and exist, respectively.

**Passing Functions/Procedures as Parameters, Securely:**

PASACL **is the first imperative language** to “explicitly” pass functions/procedures as parameters, “securely”.

Ex: 

```
function test (function f (x: integer): real; x : real) real;
begin  test :=  f(x * x) – f(-x*x) end;
```

**Name Structure:**

The way to bind names to their meanings in PASCAL is done via the following Six bindings mechanisms:

1- Constant section  
2- Type section  
3- Variable section  
4- Procedure/Function declaration  
5- Implicit Enumeration  
6- Label declaration (yes PASCAL has “goto” a label has to be declared in the scope of its use: **var dest : label;**)

**Control Structure:**

The following are very important contribution of PASCAL:

The definite “for” loop:  
```
for <name> := <expr> (to/downto) <expr> do <statement>
```

The indefinite “while” loop:  
```
while <condition> do <statement>
```
The indefinite “repeat” loop: \texttt{repeat} \texttt{<statement>} \texttt{until} \texttt{<condition>}

The “case” statement: \texttt{case} \texttt{<expr>} \texttt{of}
\texttt{\quad \texttt{<case clause>};}
\texttt{\quad \texttt{<case clause>};}
\texttt{\quad ***}
\texttt{\quad \texttt{<case clause>}
\texttt{end}}

where:
\texttt{<case clause> ::= <constant>, <constant>, \ldots, <constant>: <statement>}

Example: \texttt{case I of}
\begin{align*}
1 & : \texttt{begin **** end}; \\
2, 3 & : \texttt{begin **** end}; \\
4 & : \texttt{begin **** end}
\end{align*}
\texttt{end}

\textbf{Parameters are passed by:}
\begin{itemize}
\item 1) value
\item 2) reference (3- constant was used! Similar to Ada “input”, pp: 202-3)
\end{itemize}

The PASCAL global declaration and by-reference has the “aliasing” that might lead to insecure name access, security loophole!

\textbf{Input/Output:} Built-in facilities, in Pascal, making use of special syntax to obtain “subroutines” that take a variable list of parameters, some of which are optional, for much more powerful formatting of the input and output values.

- \texttt{Algol}, \texttt{Pascal}, \texttt{C}, and \texttt{C++} have “go-to”, but not \texttt{Modula} and \texttt{Java} (yet, Java it reserves the token as a keyword for its compiler error messages when a C++ programmer mistakenly uses it)!

\textbf{Block Structured Languages}

\textbf{There are two major structures that are maintained for any computation at run time:}

\textbf{A) Soft structure: activation state with the following two parts:}
\begin{itemize}
\item a) \texttt{Fixed} Part: the machine code of the program, and
b) **Variable Part**: The activation record (AR) that holds the state of the “activation” (procedure/function in execution).
   i) **environment part (ep)**: defines the context of an activation; it consists of:
      a) locals and formal parameters.
      b) **Static Link (SL)**: pointer to all visible non-local accessible scopes (starting at the AR of the definer of the activation) of the current activation.
   ii) **Instruction Pointer (IP)**: pointer into the current activation code.
   iii) **Dynamic Link (DL)**: a pointer to the caller’s AR, where the return address is stored.

B) **Hardware structure (virtual processor \( \pi \))**: consists of the following:

   i) **Environment Pointer (EP)**: a pointer to the current activation environment (its AR).
   ii) **Instruction Pointer (IP)**: a pointer to the next instruction to be executed.
   iii) **Stack Pointer (SP)**: a pointer to the next AR record to be placed at the top of the stack, in case of the current activation calls a procedure or function.

**Name Access (local/non-local):**

**Vocabularies:**

i) **Static Chain**: a chain starting from the current AR’s SL until the main program AR, following the SL of all intermediate ARs.
ii) **Dynamic Chain**: a chain starting from the current AR’s DL until the main program AR, following the DL of all intermediate ARs.
ii) **Static Nesting Level (snl)**: the snl of a name use or declaration is the number of surrounding contour diagram boxes around its use or declaration, respectively.

The Static distance (sd) of a name is:

\[
    sd = snl_{use} - snl_{def}
\]

- The compiler will maintain a symbol table that keeps a record of properties (e.g., type, snl_{use}, snl_{def}, sd, and “offset” within the local declarations of its definer, etc) for each name; this will help in compile time type checking and run-time name access (look page 216).

**Question**: Knowing that for any used name in the program, the snl_{use} must be greater than or equal snl_{def} (i.e., sd of the name is positive), is it a sufficient condition to have positive sd for a name to be used? (No, the name has to also be visible)
Accessing Non-Local Names: (static scoping)

In addition to the available offset in the symbol table, the compiler will compute the $sd$ of every non-local name, then it generates machine code to carry out the following at run time:

a) To locate the AR of the environment of definition of the non-local, traverse the static chain $sd$ times in order to get to the defining module’s AR.
b) Add the name’s offset (from the symbol table) to the obtained address of the defining AR, above (in a).

$$AP := M[EP].SL;$$ traverse first link of the static chain to the AR of the definer of the current activation.

$$(sd - 1) \{ AP := M[AP].SL \};$$ traverse remaining links of the static chain, up to the AR of the definer of the non-local.

$${\textbf{fetch}} M[AP + \text{offset}] ;$$ offset with the AR of the definer to get to the non-local name.

The total number of memory referencing is “$sd+1$”, expensive execution time in case of deeply nested modules.

**Question:** Discuss the above concern as a language designer, and as a programmer. The language designer will investigate the language environment, if users tend to have deeply nested modules, and then think of other mechanism not the “static chain”. If the programmers are using a language with “static chain”, then they should not deeply nest proc/functions, for their program to execute faster!

**Question:** What is the scenario that makes the static and dynamic chains the same? When the callers and the definers are the same.

**Question:** Show a scenario where the static and dynamic chains are not the same.
Procedure “Call” Sequence with Static Chain:

\[
\begin{align*}
M[\pi_{SP}].PAR[1] & := \text{evaluate (actual-parameter[1])}; \\
M[\pi_{SP}].PAR[2] & := \text{evaluate (actual-parameter[2])}; \\
\text{***} & \\
M[\pi_{SP}].PAR[n] & := \text{evaluate (actual-parameter[n])}; \\
M[\pi_{EP}].IP & := \pi_{IP}; \text{ save the resume address into the caller’s AR} \\
M[\pi_{SP}].DL & := \pi_{EP}; \text{ set the dynamic link in the callee’s AR} \\
sd * (\pi_{EP} & := M[\pi_{EP}].SL); \text{ get to the environment of the callee’s definer} \\
M[\pi_{SP}].SL & := \pi_{EP}; \text{ set the static link of the callee} \\
\pi_{EP} & := \pi_{SP}; \text{ set the virtual processor EP to the callee AR} \\
\pi_{SP} & := \pi_{SP} + \text{size (callee’s AR)}; \text{ set the hard virtual processor stack pointer ready for the next AR allocation for any future proc/function call!} \\
\pi_{IP} & := \text{entry address of the callee’s code} \\
& \quad \text{; transfer the control to execute at the callee’s code by the proper setting of the virtual processor’s IP.} \\
\text{resume:} & \quad (\text{house keeping code}) \quad \text{; return address}
\end{align*}
\]

Procedure “Return” Sequence with Static Chain:

1) \(\pi_{SP} := \pi_{SP} - \text{size (callee’s AR)}\); pop the callee’s AR from the stack

2) \(\pi_{EP} := M[\pi_{EP}].DL\); reactivate the virtual processor back into the caller
3) $\pi_{IP} := M[\pi_{EP}].IP$ ; goto the resume address and start executing at the caller’s code

Procedural Parameters Are Represented by “Closures”:

- We know the values of an integer or real type, but for the type “procedure/function” the question is: what is the state (value) of such type, and how to represent it?
- The answer is the “closure”: A “closure” is a pair <$ip, ep$>

Where: 
- $ip$ is the code instruction pointer
- $ep$ is a pointer to the environment of the definition of the procedure/function that is represented with such closure.
- The compiler will evaluate a closure for every actual procedure parameter, (or locally declared name); and passes it to its corresponding formal, in the caller activation record (or keep it in the activation record of the host of declared proc/func). Then, inside the callee, if such formal parameter procedure (function) is called, the compiler will generate the following machine code calling sequence.

**Calling Formal Parameter Procedure (fp)**

$M[\pi_{SP}].PAR[1] :=$ evaluate (actual-parameter[1]);

***

$M[\pi_{SP}].PAR[n] :=$ evaluate (actual-parameter[n]);
$M[\pi_{EP}].IP := \pi_{IP}$ ; save resume address into the caller’s AR
$M[\pi_{SP}].DL := \pi_{EP}$ ; set the dynamic link in the callee’s AR

$M[\pi_{SP}].SL := fp.ep$ ; fill in the SL of fp’s AR, with the environment pointer that is extracted form the fp’s closure
$\pi_{EP} := \pi_{SP}$ ; set the virtual processor EP to the callee AR
$\pi_{SP} := \pi_{SP} +$ size (callee’s AR) ; set the hard virtual processor stack pointer ready for the next AR
allocation for any future proc/function call, while in fp!

\[ \pi_{IP} := fp.ip \] ; go to the code of fp

* Notice that we assumed that the fp’s closure exists and we used it, but at some point the compiler has to build it as follows:

**How to Build the Closure (for procedure p)?**

M[\pi_{SP}].PAR[i].ip := entry (p); build the ip part of the closure of procedure p
\[ \pi_{AP} := M[\pi_{EP}].SL \] ; traverse the first static link
(sd-1) * (\pi_{AP} := M[\pi_{AP}].SL) ; get to the environment of the p’s definition
M[\pi_{SP}].PAR[i].ep := \pi_{AP} ; build the ep part of the closure of procedure p

Look page 226, figure 6.5 for examples of formal procedure parameters.

**Functions/procedures As First Class Citizens (FCC)**

- Until now we passed functions/procedures as parameters; what about returning them as values from other functions (i.e., treating them as FCC)?
- Function Composition:

  1. Program Test;
  2. type
  3. fun : function (integer): integer;
  4. var
  5. m: integer;
  6. h, incr, sqr : fun;

  7. function Compose (f, g : fun): fun;

  8. function apply_copm (x : integer): int;
  9. begin
  10. return (f (g (x)) );
11.   end;
12.   begin (* Compose *)
13.       return (apply_comp);
14.   end (* Compose *);
15.   begin (* Test*)
16.       h := Compose (incr, sqr);
17.       m := h (3)                           (* m = 10 *)
18.   end. (* Test*)

If we follow the run time stack dynamic formation, we will find out that
upon exiting from Compose line 16, there will be a closure <ip,ep>apply_comp
With its ep pointing to its definer “Compose”, but we already exited from
Compose! Hence, there is a problem of an orphan “ep”!!!!!!!!!!!!!!!!!!!!!!

To solve the above problem, we must give up the “stack” as a model of
computation, and use a “heap”, where we can keep an activation record even
after we exit its module!

Thus, for languages that return functions/procedures from other functions,
we can not use the “stack” as a model of computation, at run time; instead
we use a “heap”.

**Retention:**
It is the ability to retain the AR of a procedure/function, even after exiting its
code. Such retention can not be achieved in languages that utilize the “stack”
as their model of computation. It is used in languages that uses “heap” as
their model of computation.

Question: Would the feature of “retention” violate the static scoping rules
manifested by the contour diagram semantics????
Answer: Yes! Adeeply nested function can be returned to a higher snl,
exposing it to be called by a lesser snl procedure!