Pascal

- ALGOL’s like language, introduced by Niklaus Wirth (1971), but more **reliable**, **efficient**, and **simple** for **pedagogic** purposes (including for system programming).

- PASCAL introduced a **much richer type system** than ALGOL:
  
  A) “**type**” and “**Const**” (why?) declarations.
  
  B) “**Enumeration Types**”: To handle non-numeric data, eliminating the insecurity of overworking integers as such needed types. (examples in: C, Pascal, and Ada)

  ```
  type
      month = (Jan, Feb, March, Apr, …, Dec);
      Day   = (Sun, Mon, Tue, …, Sat);
  
  In C : enum StudentClass {Fresh, Soph, Junior, Senior}
  enum EmployeeGender {Male, Female}
  In Ada : type Days is ((Sun, Mon, Tue, …, Sat);
  
  They are **true ADTs**, with **user** defined data type, and **system** built-in operators: :=, succ, pred, =, ≠, <, >, ≥, ≤
  
  **Advantages:**
  
  1- High level application oriented, allowing the language to cover wider area of applications.
  2- Efficient use of memory to represent the type values.
  3- **Secure** use of type elements, the compiler protects against any meaningless operations by the users on elements of the defined types. Pred(Jan) and succ (Sat) will produce compile time errors.

  **Question:** Can we still use the standard input/output commands for enumerated types?

  **Problems? Yes:**
  
  1- No Input/output built in operations! (why?)
  2- Overloaded enumerated literal constants when appearing in different definitions at the same environment.

  **Ex:** type favoriteColors = (red, yellow, magenta, brown, aqua, blue, green);
      TrafficLightColors = (red, yellow, green);

      for color in ‘red’ .. ’green’ do
Notice that “color” is implicitly typed with the specified discrete range by the compiler. The discrete range is ambiguous(!) since the compiler will not know to which type it belongs: favoriteColors or TrafficLightColors?

Hence, C and Pascal do not allow the same element name to be used in more than one enumeration type in the same scope.

Solution by Ada-- Ada allows it but to resolve the name overloading:

```plaintext
for color in favoriteColors (‘red’) .. favoriteColors (‘green’) do
```

C) Subrange Types: Pascal (and other languages like Ada) allows the programmer to define subranges of only discrete types (enumerated, int, characters), where it inherits all of its parent defined set of operations.

```plaintext
type uppercase = 'A'..'Z'; index = 1..100; WeekDays = Mon..Fri;
```

Advantages:
1- Enhances readability.
2- Security, errors due the assignment of out of range value will be detected at compile time, in case of literal constant value; or at run time, in case of expression/variable assignment.
3- Efficient memory representation of the type values.

*Notice* that when the “ subrange” inherits its parent’s set of operations, it introduces “security loophole”. For example, “dayOfMonth = 1..31”, it is maybe ok to add/subtract the days of month, but what about dividing/multiplying them??!! Subranges are built in Algol 68, Pascal, and Ada; but not C++/Java(?)

Do you consider the use of meaningless (inherited)parent’s operation on its subrange type a security loophole? Justify!
D) “Set Type”:
Advantages:
1- High level and application oriented
2- Efficient representation.
3- ADT and readability.

```pascal
type favoriteColors = (red, yellow, magenta, brown, aqua, blue, green);
colorset = set of favoriteColors;

var set1, set2 : colorset;
digits : set of char;

begin
  set1 := [red, blue, yellow, aqua];
  set2 := [brown, green];
  T := [1..10]; (* set of integers 1 to 10*)
  S := [1, 2, 5, 7, 12, 13];
  T := T * S; (* T will have 1,2,5,7 *)
  (* the operator * is used instead of the intersection ∩ *)
  if T = [1, 2, 5, 7] then …
  digits := ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'];
  read (ch);
  if ch in digits the …. (* the in operation is used for membership *)

More operations on sets: <, > strict subset (S1<S2) (S2>S1) : Boolean
<> not equal sets (S1<>S2) : Boolean
≤, ≥ subset or equal (S1≤ S2 S2 ≥ S1) : Boolean
− difference between two sets (S1− S2) : set
+ Union of two sets (S1 + S2): set
= equal sets (S1 =S2): Boolean
```

E) Record Type: One of the most important contributions by Pascal as a “heterogeneous” data structure that aggregates a group of related, but different data types fields that pertain to an object (e.g., employee, student, etc). It is called “structure” in C.

Variant Records: (pages 187-189): Similar to the “union” in the C language, variant records aim at sharing memory again, yet aliasing different fields’ names on the same memory location which will be a potential insecurity problem, especially with no general mechanism to enforce initialization of the aliased memory before switching fields’ names.
Typed “Pointer”: Pascal is among the pioneer languages to introduce typed pointers to 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, as in C!).

In Pascal:
```
var p: ^ real; x: real; c: char;
begin new (p); p^ := 3.14159;
c := p^;   {compiler error, storing real into char!}
```

In C&C++:
```
int *ptr; int count, init; *** ****

ptr = &init; /* ptr points to init*/
count = *ptr; /* store init in count */
```

They also have a non-types pointers “void *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 “void *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 “void *ptr” facility in C?

We can have pointers to other non-basic type (e.g., records).
```
Ex: var p: ^ plane; (* plane is a record type*)
***
p^.parked := ....
```
- A pointer with the value “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 it 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, Java, 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 non-heap 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 should it be done? (Overhead) **Explicit:** Trusting that the user will do it right! If users are prompt in doing it → Efficient, but **insecure** if the user de-allocates an already used space (creating **dangling** reference that points to an unintended/invalid object!)

**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:**

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

**Example:**

```pascal
function test (function f (x: real): 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** binding’s mechanisms:

1- **Constant** section  2- **Type** section  3- **Variable** section
4- Procedure/Function declaration  5- Implicit Enumeration (mid p174)
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: \[ \text{for } \text{name} := \text{expr} \text{ (to/downto) expr } \text{ do } \text{statement} \]

The indefinite “while” loop: \[ \text{while } \text{condition} \text{ do } \text{statement} \]

The indefinite “repeat” loop: \[ \text{repeat } \text{statement} \text{ until } \text{condition} \]

The “case” statement: \[ \text{case } \text{expr} \text{ of } \]
\[ \text{case clause}; \]
\[ \text{case clause}; \]
\[ *** \]
\[ \text{case clause} \]
\[ \text{end} \]

where:
\[ \text{case clause} := \text{constant}, \text{constant}, \ldots, \text{constant}: \text{statement}; \]

Example: \[ \text{case I of } \]
\[ 1 : \text{begin **** end; } \]
\[ 2, 3 : \text{begin **** end; } \]
\[ 4 : \text{begin **** end } \]
\[ \text{end} \]

Parameters are passed by:

1) value  2) reference (3- constant was used! Similar to Ada “input”, pp: 202-3)

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

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.

• Algol, Pascal, C, and C++ have “go-to”, but not Modula and Java.
**BNF Grammar (Backus-Naur Form)**

*Meta*language that describes another language’s *syntax*.

It is equivalent to *context free grammar* (CFG): $(\{T\}, \{N\}, \{P\}, S)$ where

- $\{T\}$: set of all terminal symbols: $0, 1, 2, \ldots$, $a, b, c, \ldots$, $+, -, \ldots$.
  - Symbols that can not be reduced further more.
- $\{N\}$: set of all nonterminal symbols: *statement-sequence*, *if-statement*, *expression*, ....
  - Symbols that need to be further reduced (replaced with other expansion symbols) (expanded).
- $\{P\}$: set of production rules,
  - the left hand side symbol must be a “nonterminal”

**Ex:** $\langle\text{unsigned integer}\rangle ::= \langle\text{digit}\rangle | \langle\text{unsigned integer}\rangle \langle\text{digit}\rangle$

$S$: starting symbol ($S$ belongs to $\{N\}$)

**Ex:** $\langle\text{program}\rangle ::= \text{program} \langle\text{header}\rangle ; \langle\text{declaration-section}\rangle ; \langle\text{program-body}\rangle \text{ end}$

$\langle\text{program}\rangle$ is the starting symbol in this case.

A *regular grammar* is either a *left* or *right* grammar.

A *right regular grammar* is same as CFG, but all production rules $P$ are one of the following rules:

1- $A \rightarrow a$ - $A$ is a non-terminal in $N$ and $a$ is terminal in $T$
2- $A \rightarrow a B$ - $A$ and $B$ are non-terminal in $N$ and $a$ is terminal in $T$
3- $A \rightarrow \varepsilon$ (empty string) - $A$ is a non-terminal in $N$

A *left regular grammar* is same as above except for rule 2,

where “$A \rightarrow Ba$” replaces of $A \rightarrow aB$

A regular grammar can be both, right and left grammar, otherwise it would be CFG.

The BNF is powerful enough to describe the following *syntactic* issues in a programming language definition:

1- Lists of similar constructs: *statement-sequence*, *declaration-sequence*, ...
2- The order in which different constructs must appear: a label must start with a letter not a digit.
3- Nested structures to any depth: nested statements
4- Matching parentheses: $(((((( A+B)))))))$
5- Operator precedence: the "/" has higher precedence over the "+"
6- Operator Associativity:

BNF for expressions:
<expr> ::= <expr> <add-op> <term>                (* left associative BNF*)
  | <term>
<term>  ::= <term> <mult-op> <factor>      (*precedence: mult-op > add-op*)
  | <factor>
<factor> ::= <id>  |  <number>  | - <factor> | (<expr>)

<add-op> ::= − | +
<mult-op> ::= / | *

In general, there are two way to evaluate this assignment: X := A – B + C

i) If the BNF expresses **left associative** then the above is execute as
X := (A – B) + C

Left associative: <expr> ::= <expr> <add-op> <term>

ii) But in case of **right associative** then the above is execute as
X := A – (B + C);

Right associative: <expr> ::= < term> <add-op> <expr>

**Yet, BNF can never describe any language semantics issues** (context sensitive issues!) (e.g., variable not declared, over/underflow, label length is not acceptable, . . .).

**Extended BNF:**
The same BNF power but more descriptive and it increases the readability of the BNF!

Extended BNF uses the following extensions to the regular BNF:

1- “optional” parts in the rhs of a rule:
Ex: <if-stmt> ::= if (<logic-expr>) then <stmt> [else <stmt>];
<integer> ::= [ + | − ] <unsigned-integer>

2- zero or more repetitions of some constructs using braces.
Ex: <stmt-list> ::= <stmt> {; <stmt>}

3- At least one occurrence: \((kleene +)\)
Ex: <unsigned integer> ::= <digit>+

4- multiple-choice options:
Ex: <for-stmt> ::= for <id> := <expr> (to|downto) <expr> do <stmt>
EBNF of the a different expression implementation:

\[
\begin{align*}
<\text{expr}> &::= <\text{term}> \{ ( +|-) <\text{term}> \} \\
<\text{term}> &::= <\text{factor}> \{(\ast | /) <\text{factor}> \} \\
<\text{factor}> &::= <\text{expr}> \{**<\text{expr}>\} \quad \text{-- The "**" symbol indicates the exponent operator} \\
<\text{expr}> &::= (<\text{expr}>) | \text{id}
\end{align*}
\]
Attribute Grammar to Describe Context Sensitive (static semantics) Languages Aspects:

It associates a set Attr(F) of attributes to each non-terminal symbol in the grammar, Attr(F) has two disjoint sets of attributes: S(F) -- synthesized & I(F) -- inherited attributes; where S(F) is synthesized bottom-up and then I(F) is to be inherited top-down over the program parse tree.

<unsigned-int> ::= 
    <digit> 
    {Value (<unsigned-int>) ← Value (<digit>)}  
    $$\cdot value ← $$1.value, synthesizing the inherited value of $$1(<digit>) into $$ (<unsigned-int>).
    | <unsigned-int> <digit> 
    {Value (<unsigned-int>) ← 10 * Value (<unsigned-int>)  
    + Value (<digit>)}  
    $$\cdot value ← 10*$$1.value + $$2.value, synthesizing the inherited values of $$2(<digit>) added to the multiplication of $$1 (<unsigned-int>) by 10).

(*the accumulated/obtained values of <unsigned-int> and <digit> are examples of inherited attribute(top-down in the parse tree); yet <unsigned-int> is synthesized attribute (bottom-up in the parse tree) *)

To check the static semantics issue of overflow:
If (Value (<unsigned-int>) ≥ n) Then [output error overflow”]

<digit> ::= 0 
{Value (<digit>) ← 0} -- synthesizing the value 0 into $1  
$$\cdot value ← $$1.value
| 1 
{Value (<digit>) ← 1} -- synthesizing the value 1 into $1  
$$\cdot value ← $$1.value
| 2 
{Value (<digit>) ← 2} -- synthesizing the value 2 into $1
Questions: Give more examples of synthesized and inherited attributes in any Algol program parse tree.

[Hint: think of the type declaration section and its utilization in the program body for type checking]