NEPAL (**N**aive **E**xperimental **P**rogramming **A**cademic **L**anguage)

- Paradigma imperativo + funzionale
- Programma strutturato in quattro sezioni: *tipi*, *variabili*, *funzioni*, *corpo*
- Tipi di dati atomici: **bool**, **int**, **real**, **string**
- Costruttori di tipo: **record**, **array** (elastico)
- Espressioni condizionali
- Operatori logici valutati in corto circuito
- Corpo della funzione = espressione
- Corpo del programma = sequenza di istruzioni
- Istruzioni: input, output, assegnamento, **if**, **while**, **foreach**
- Commenti: `// Questo è un commento`

Esempio di Programma

```
types
  Numeri = [int];

variables
  numeri_da_ordinare: Numeri;

functions
  inserisci(n: int, numeri: Numeri): Numeri // inserimento di un numero in un arrary ordinato
    if empty(numeri) then
      [n]
    else
      if n <= head(numeri) then
           [n] ++ numeri
      else
        [head(numeri)] ++ inserisci(n, tail(numeri))
      end
    end
  end

  ordina(numeri: Numeri): Numeri // ordinamento di un array per inserzione
    if empty(numeri) then
      []
    else
      inserisci(head(numeri), ordina(tail(numeri)))
    end
  end

run
  write "Inserisci una sequenza di numeri da ordinare:\n";
  read numeri_da_ordinare;
  write "Sequenza di numeri ordinata:\n";
  write ordina(numeri_da_ordinare);
end
```

Tipi Atomici

- `bool`

```
ok: bool;
...
ok = true;
```

- `int`

```
i: int;
...
i = 36;
```

- `real`

```
x: real;
...
x = 12.45;
```

- `string`

```
nome: string;
...
nome = "Luisa";
```

Tipo Record

```
persona: {nome: string, eta: int, studente: bool};
...
persona = {"anna", 25, false};
```

```
types

  Persona = {nome: string, eta: int, studente: bool};

...

variables

  persona: Persona;
```

Tipo Array

```
numeri: [int];
...
numeri = [1,2,3,4,5];
```

```
parole: [string]
...
parole = ["alfa","beta"];
```

```
flags: [bool];
...
flags = [true,false,true];
```

```
persone: [{nome: string, eta: int, studente: bool}];
...
persone = [{"anna",25,false},{"luigi",16,true},{"maria",42,false}];
```

```
types

  Persona = {nome: string, eta: int, studente: bool};
...

variables

  persone: [Persona];
```

Costruzione di Valori Strutturati mediante Espressioni

```
persona, p: {nome: string, eta: int, studente: bool};
...
persona = {p.nome, n+m, is_student(p.nome)};
```

```
fibonacci: [int];
...
fibonacci = [fib(0),fib(1),fib(2),fib(3),fib(4),fib(5),fib(6),fib(7),fib(8),fib(9)];
```

```
numeri: [int];
...
numeri = [n+m-1, 25, fib(i)*fib(j)];
```

```
p1, p2, p3: Persona;
persone: [Persona];
...
persone = [p1, p2, p3, nuova_persona("angela", 38, false)];
```

Funzioni Polimorfe su Array: `empty`, `head`, `tail`

```
a, b: [int];
iniziali: {testa1: int, testa2: int};
...
if empty(a) or empty(b) then
  write "L'array non può essere vuoto";
else
  iniziali = {head(a), head(b)};
end;
```

```
somma(numeri: [int]): int
  if empty(numeri) then
    0
  else
    head(numeri) + somma(tail(numeri))
  end
end
```

Operatori Polimorfi su Array

- | *expr* | (cardinalità)

```
a, b: [int];
tot: int;
...
a = [1,2,3];
b = [7,12,25,0,1];
tot = |a| + |b|;        // tot = 8
```

- ++ (concatenazione)

```
pari, dispari, cifre: [int];
...
pari = [0,2,4,6,8];
dispari = [1,3,5,7,9];
cifre = pari ++ dispari;  // cifre = [0,2,4,6,8,1,3,5,7,9]
```

Espressioni Artitmetiche

- `+`, `-`, `*`, `/` : operatori applicabili ai tipi `int` e `real`

```
a, b, c, d: int;
...
a = ((b + c) - (d * 25)) / (a + c);
```

```
x, y, z: real;
...
z = (x + 24.15) * (y – 0.48);
```

- Coercizione di tipo (quando operatore o assegnamento applicato a `int` e `real`):

```
n, tot: int;
x, media: real;
...
media = (n + x) / 2;     // valori di n e 2 trasformati in real
tot = media + x;         // valore di (media + x) trasformato in int
```

- Cast di tipo:

```
n, m, somma: int;
x: real;
...
somma = n + int(x); // valore di x trasformato in int (cast)
n = m + real(n);    // valore di n trasformato in real (cast)
                    // valore di m trasformato in real (coercizione)
                    // valore di (m + real(n)) trasformato in int (coercizione)
```

Operatori di Confronto

- ==, != (applicabili a tutti i tipi)

```
i, j: int;
nome, cognome: string;
ok, flag, strano: bool;
numeri, voti: [int];
...
... if i == j then ...
... if ok != flag then ...
... if numeri == voti then ...
...
strano = nome == cognome;
```

- >, >=, <, <= (applicabili ai tipi `int`, `real` e `string`)

```
i, j: int;
x, y: real;
nome, cognome: string;
...
... if i >= j then ...
... if nome < cognome then ...
```

- `in` (appartenenza di un valore ad un array)

```
n, m: int; numeri: [int];
...
... if n+m in numeri then ...
```

Espressioni Logiche

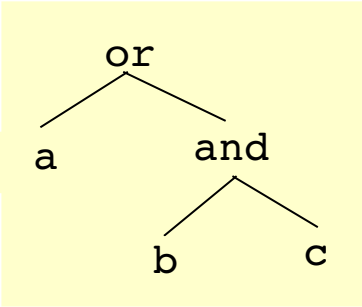
- `and`, `or`, `not` (applicabili al tipo `bool`)

- Valutazione in corto circuito

```
a, b, c, ok: bool;
...
... if ok then
      a or (b and c)
    else ...
```

⟺

```
if a then
  true
elsif b then
  c
else
  false
endif
```



- Integrazione con operazioni di confronto

```
i, j: int;
numeri: [int];
a, b: bool;
...
b = (i == j+2 or a) and numeri == [1,2,3,4,5];
```

Precedenza, Associatività, Ordine di Valutazione

Operatore	*Tipo*	*Associatività*
`and`, `or`	binario	sinistra
`==`, `!=`, `>`, `>=`, `<`, `<=`, `in`	binario	nonassoc
`+`, `-`, `++`	binario	sinistra
`*`, `/`	binario	sinistra
`-`, `not`	unario	destra

precedenza crescente

- Ordine di valutazione degli operandi: da sinistra a destra

Espressione Condizionale

```
a, b, c: int;
...
a = if b > c then b + c else a + 1 end;
```

Istruzioni `read` e `write`

```
n: int;
...
read n;
write "Il fattoriale di ";
write n;
write " è ";
write fattoriale(n);
```

```
a, b: int;
...
read a;
read b;
write if a >= b then a else b end;
```

```
numeri: [int];
...
read numeri
write ordina(numeri);
```

Istruzione `if`

```
n, m: int;
numeri: [int];
...
if n > m then          // if ad una via
  write n + m;
end;

...

if n == m then         // if a due vie
  m = m +1;
else
  m = m - 1;
  write m * n;
end;
```

Istruzioni `while` e `foreach`

```
n: int;
...
read n;
while n >= 0 do
  write fib(n);
  n = n - 1;
end;
```

```
i: int; numeri: [int];
...
i = 0;
while i < |numeri| do
  numeri[i] = numeri[i] + 1;
  i = i + 1;
end;
```

```
prodotti: [{nome: string,
            prezzo: int}];
...
i = 0;
while i < |prodotti| do
  write prodotti[i].nome;
  i = i + 1;
end;
```

```
types
  Persona = {nome: string, eta: int, studente: bool);
...
p: Persona;
persone: [Persona];
nomi: [string];
...
nomi = [];
read persone;
foreach p in persone do
  nomi = nomi ++ [p.nome];    // accumulo dei nomi delle persone
end;
write nomi;
```