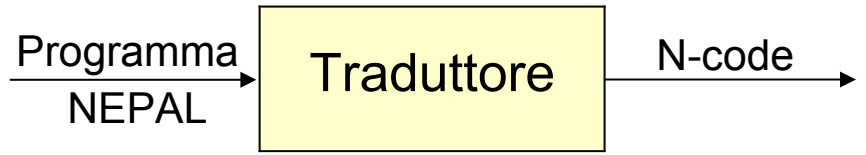
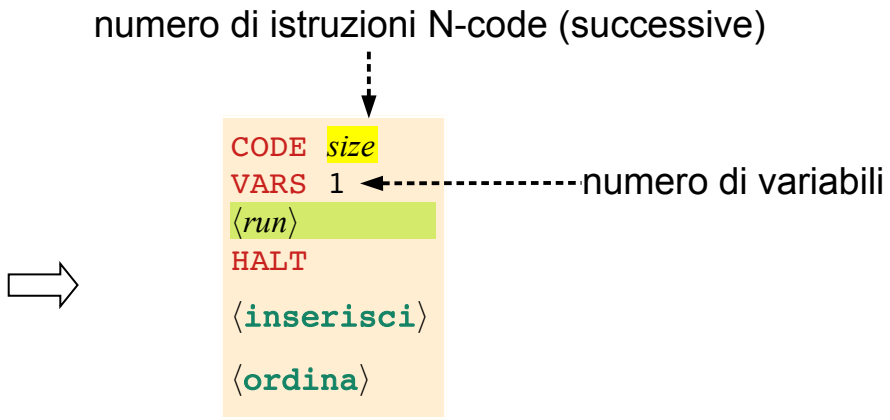
Generazione di Codice Intermedio N-code

```
types
  Numeri = [int];

variables
  numeri_da_ordinare: Numeri;

functions
  inserisci(n: int, numeri: Numeri): Numeri
    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
    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
```





Generazione di Codice Intermedio (ii)

- Codice indirizzabile direttamente (senza uso di label)
- Indirizzo di una istruzione N-code = posizione della istruzione nel codice (valore intero: 0, 1, ...)
- Descrittori degli oggetti (variabili e parametri) allocati nell'ordine in cui vengono dichiarati
- Identificazione degli oggetti (variabili e parametri): mediante ***oid***, l'object identifier dell'oggetto nel suo ambiente (numerazione relativa: 0, 1, 2, ...)

Dichiarazione di Variabili

Dichiarazione di una variabile → `NEWO` *size num*

```
min, max: int;
media: real;
nome: string;
ok: bool;
p: {nome:string, eta:int, studente:bool};
people: [{nome:string, eta:int, studente:bool}];
```

```
NEWO |int| 1
NEWO |int| 1
NEWO |real| 1
NEWO |ptr| 1
NEWO |int| 1
NEWO ⟨|ptr|+|int|+|int|⟩ 1
NEWO ⟨|ptr|+|int|+|int|⟩ 0
```

Note:

- *size* = dimensione della variabile o (nel caso di array) dell'elemento dell'array
- *num* = numero di elementi (nel caso di array, altrimenti `1`)
- Tipo `bool` rappresentato da `int`
- Dimensione `ptr` = dimensione puntatore (a carattere)
- Gestione delle stringhe mediante hash table (per garantire unicità del puntatore)

Referenza a Costante Atomica

- Costante booleana `ok = true;` ⇨ `LOCI 1`
- Costante intera `i = 25;` ⇨ `LOCI 25`
- Costante reale `x = 17.36;` ⇨ `LOCR 17.36`
- Costante stringa `nome = "Anna";` ⇨ `LOCS "Anna"`

Nota: Valori booleani (`true`, `false`) → rappresentati da interi (`1`, `0`)

Referenza a Parametro di Funzione o Variabile

1. Locale (parametro)

```
j + 25
```

⇨

```
LOAD 1 ^j
```

```
alfa + beta
```

⇨

```
LOAD 1 ^beta
```

2. Globale (variabile)

```
i = n - 12;
```

⇨

```
LOAD 0 ^n
```

```
delta = gamma;
```

⇨

```
LOAD 0 ^gamma
```

Note:

- Argomento di `LOAD` = *env* (environment), *oid* (object identifier)
- *env* = 0 → globale (variabile); *env* = 1 → locale (parametro)
- `^nome` indica l'object identifier dell'oggetto di nome `nome` (quindi, un intero: 0, 1, 2, ...)

Referenza a Costruttore di Record

Generazione delle espressioni dei campi del record + impacchettamento (PACK)

```
...
if persona == {nome, n+10, true} then ...
...
```

```
LOAD 0 ^nome
LOAD 0 ^n
LOCI 10
ADDI
LOCI 1
PACK 3 ⟨|ptr|+|int|+|int|⟩ 1
```

Note:

- Argomenti di PACK = *numero campi (atomici)*, *dimensione record*, 1 (numero elementi)
- In generale: generazione di codice per tutte le espressioni dei valori dei campi

Referenza a Costruttore di Array

Generazione delle espressioni degli elementi dell'array + impacchettamento (PACK)

```
p: Persona;
persone: [Persona];
...
persone = [p, {"anna", 21, true}];
```

```
LOAD 0 ^p
LOCS "anna"
LOCI 21
LOCI 1
PACK 3 ⟨|ptr|+|int|+|int|⟩ 1
PACK 2 ⟨|ptr|+|int|+|int|⟩ 2
```

Note:

- Argomenti di PACK = *numero elementi*, *dimensione elemento*, *numero elementi*
- In generale: generazione di codice per tutte le espressioni degli elementi dell'array

Referenza ad Attributo di Record mediante Fielding

Load address del record (`LODA`) + indirect load (`INDL`)

```
persona : {nome: string, eta: int, studente: bool};
...
write persona.eta;
```

```
LODA 0 ^persona
INDL °eta |int|
```

Note:

- Argomento di `LODA` (*load address*) = *env* (environment), *oid* (object identifier)
- *env* = 0 → globale (variabile); *env* = 1 → locale (parametro)
- Argomenti di `INDL` (*indirect load*) = *offset campo*, *size campo*
- `°eta` indica l'offset del campo `eta` all'interno del record

Referenza di Elemento di Array mediante Indexing

Load dell'indirizzo dell'array +
Computazione del valore dell'indice +
Check del valore dell'indice (`CIDX`) +
Load dell'indirizzo dell'elemento indicizzato (`IXAD`) +
Indirect load dell'elemento indicizzato (`INDL`)

```
numeri: [int];
i: int;
...
write numeri[i+2];
```

```
LODA 0 ^numeri
LOAD 0 ^i
LDCI 2
ADDI
CIDX
IXAD |int|
INDL 0 |int|
```

```
persone : [{nome: string, eta: int, studente: bool}];
i, j: int;
...
write persone[i*j].eta;
```

```
LODA 0 ^persone
LOAD 0 ^i
LOAD 0 ^j
MULI
CIDX
IXAD ⟨|ptr|+|int|+|int|⟩
INDL °eta |int|
```

Note:

- `CIDX`: nella macchina virtuale, se indice fiuori dal range → terminazione esecuzione
- Argomento di `IXAD` (indexed *address*) = fattore di scala
- Argomenti di `INDL` (*indirect load*) = *offset*, *size*

Assegnamento di Identificatori

Load dell'indirizzo della variabile da assegnare (LODA) +
Computazione della espressione di assegnamento +
Store (STOR)

id = *expr*;

```
LODA 0 ^id
⟨expr⟩
STOR
```

```
persona : {nome: string, eta: int, studente: bool};
...
persona = {"Luigi", 42, false};
```

```
LODA 0 ^persona
LOCS "luigi"
LOCI 42
LOCI 0
PACK 3 ⟨|ptr|+|int|+|int|⟩ 1
STOR
```

Nota: STOR: senza argomenti

Assegnamento di Campo di Record

Computazione dell'indirizzo del record +
Caricamento dell'offset del campo da assegnare +
Computazione dell'indirizzo del campo da assegnare (`IXAD`) +
Computazione della espressione di assegnamento +
Store (`STOR`)

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

```
LODA 0 ^persona
LOCI °eta
IXAD 1
LOCI 25
STOR
```

```
persone : [{nome: string, eta: int, studente: bool}];
i, j: int;
...
persone[i+j].eta = 25;
```

```
LODA 0 ^persone
LOAD 0 ^i
LOAD 0 ^j
ADDI
CIDX
IXAD ⟨|ptr|+|int|+|int|⟩
LOCI °eta
IXAD 1
LDCI 25
STOR
```

Nota: Argomento di `IXAD` = 1

Assegnamento di Elemento di Array

Load dell'indirizzo dell'array (LODA) +
Computazione dell'indice +
Check del valore dell'indice (CIDX) +
Computazione dell'indirizzo dell'elemento da assegnare (IXAD) +
Computazione della espressione di assegnamento +
Store (STOR)

```
i, j: int;
numeri: [int];
...
numeri[i+j] = 36;
```

```
LODA 0 ^numeri
LOAD 0 ^i
LOAD 0 ^j
ADDI
CIDX
IXAD |int|
LOCI 36
STOR
```

```
p: {nome: string, eta: int, studente: bool}
persone : [{nome: string, eta: int, studente: bool}];
i: int;
...
persone[i+3] = p;
```

```
LODA 0 ^persone
LOAD 0 ^i
LOCI 3
ADDI
CIDX
IXAD ⟨|ptr|+|int|+|int|⟩
LOAD 0 ^p
STOR
```

Operazioni Logiche: `and`, `or`

$logic\text{-}expr \rightarrow expr_1 \; expr_2$

and

```
⟨expr1⟩
SKPF offset
⟨expr2⟩
SKIP 2
LOCI 0
```

or

```
⟨expr1⟩
SKPF 3
LOCI 1
SKIP exit
⟨expr2⟩
```

```
a, b, c, g: bool;
...
a = (b and c) or g;
```

```
LODA 0 ^a
LOAD 0 ^b
SKPF 3
LOAD 0 ^c
SKIP 2
LOCI 0
SKPF 3
LOCI 1
SKIP 2
LOAD 0 ^g
STOR
```

Note:

- Valutazione in corto circuito
- `SKIP` = salto incondizionato relativo
- `SKPF` = salto condizionato (a false) relativo
- Argomento di `SKIP`, `SKPF` = dimensione del salto $\begin{cases} exit = |\langle expr_2 \rangle| + 1 \\ offset = |\langle expr_2 \rangle| + 2 \end{cases}$

Operazioni di Confronto: ==, !=

comp-expr → *expr*$_1$ *expr*$_2$

⇨

==	!=
⟨*expr*$_1$⟩	⟨*expr*$_1$⟩
⟨*expr*$_2$⟩	⟨*expr*$_2$⟩
EQUA	NEQU

```
i, j, k: int;
b: bool;
...
b = (i == j or j != k);
```

⇨

```
LODA 0 ^b
LOAD 0 ^i
LOAD 0 ^j
EQUA
SKPF 3
LOCI 1
SKIP 4
LOAD 0 ^j
LOAD 0 ^k
NEQU
STOR
```

Note:

- EQUA, NEQU: polimorfe (applicabili ad ogni tipo di oggetto)
- Costanti stringa non duplicate (confronto fra puntatori) → hash table

Operazioni di Confronto: >, >=, <, <=

$comp\text{-}expr \rightarrow expr_1 \; expr_2$

	>	>=	<	<=
int	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ GTHI	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ GEQI	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ LTHI	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ LEQI
real	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ GTHR	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ GEQR	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ LTHR	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ LEQR
string	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ GTHS	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ GEQS	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ LTHS	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ LEQS

```
i, j: int;
s, t: string
b: bool;
...
b = i > j or s < t;
```

```
LODA 0 ^b
LOAD 0 ^i
LOAD 0 ^j
GTHI
SKPF 3
LOCI 1
SKIP 4
LOAD 0 ^s
LOAD 0 ^t
LTHS
STOR
```

Nota: Operatori applicabili solo a numeri (`int`, `real`) o stringhe

Operazioni di Confronto: `in`

$comp\text{-}expr \rightarrow expr_1 \; expr_2$ ⇨

$\langle expr_1 \rangle$
$\langle expr_2 \rangle$
`MEMB`

```
p: Persona;
persone: [Persona];
...
if p in persone then ...
```

⇨

```
LOAD 0 ^p
LOAD 0 ^persone
MEMB
```

Nota: Tipo di $expr_1$ = tipo dell'elemento dell'array $expr_2$

Operazioni Aritmetiche: +, -, *, /

$math\text{-}expr \rightarrow expr_1 \; expr_2$ ⇨

	+	-	*	/
int	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ ADDI	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ SUBI	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ MULI	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ DIVI
real	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ ADDR	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ SUBR	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ MULR	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ DIVR

```
i, j, k: int;
...
i = (i + 5) * (j – k);
```

⇨

```
LODA 0 ^i
LOAD 0 ^i
LOCI 5
ADDI
LOAD 0 ^j
LOAD 0 ^k
SUBI
MULI
STOR
```

Operazioni di Negazione: -, not

neg-expr → *expr*

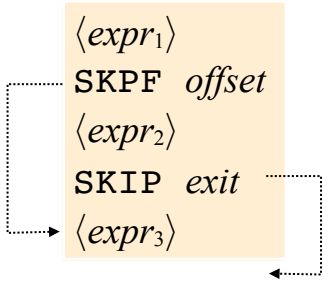
- (int)	- (real)	not
⟨*expr*⟩	⟨*expr*⟩	⟨*expr*⟩
NEGI	NEGR	NEGB

```
i, j, k: int;
a, b: bool;
...
b = i > j * k and not (a or j == -k);
```

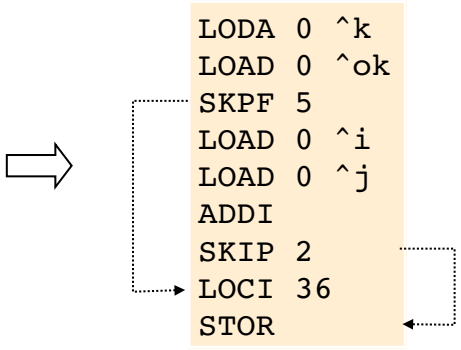
```
LODA 0 ^b
LOAD 0 ^i
LOAD 0 ^j
LOAD ^k
MULI
GRTR
SKPF 11
LOAD 0 ^a
SKPF 3
LOCI 1
SKIP 5
LOAD 0 ^j
LOAD 0 ^k
NEGI
EQUA
NEGB
SKIP 2
LOCI 0
STOR
```

Espressione Condizionale

$if\text{-}expr \rightarrow expr_1 \; expr_2 \; expr_3$




```
i, j, k: int;
ok: bool;
...
k = if ok then i + j else 36 end;
```



Note:

- $offset = |\langle expr_2 \rangle| + 2$
- $exit = |\langle expr_3 \rangle| + 1$

Operatori Built-in

built-in-expr → *expr*

`real(`*expr*`)`	`int(`*expr*`)`	`empty(`*expr*`)`	`head(`*expr*`)`	`tail(`*expr*`)`	`|`*expr*`|`
⟨*expr*⟩ `TORE`	⟨*expr*⟩ `TOIN`	⟨*expr*⟩ `EMPT`	⟨*expr*⟩ `HEAD`	⟨*expr*⟩ `TAIL`	⟨*expr*⟩ `CARD`

```
a, b int;
media: real;
...
media = (real(a) + real(b)) / 2;
```

⇨

```
LODA 0 ^media
LOAD 0 ^a
TORE
LOAD 0 ^b
TORE
ADDR
LOCR 2
DIVR
STOR
```

coercizione diretta di costante intera

```
n int;
x, y: real;
...
y = x + n;
```

⇨

```
LODA 0 ^y
LOAD 0 ^x
LOAD 0 ^n
TORE
ADDR
STOR
```

coercizione dell'intero `n` in reale

```
numeri: [int];
n: int;
flag: bool;
...
n = head(numeri) + |numeri|;
numeri = tail(numeri);
flag =  empty(numeri);
```

⇨

```
LODA 0 ^n
LOAD 0 ^numeri
HEAD
LOAD 0 ^numeri
CARD
ADDI
STOR
LODA 0 ^numeri
LOAD 0 ^numeri
TAIL
STOR
LODA 0 ^flag
LOAD 0 ^numeri
EMPT
STOR
```

Istruzione `read`

read-stat → **id** ⇨ `READ ^id` *format*

```
n: int;
x: real;
nome: string;
flag: bool:
p: {nome: string, eta: int, studente: bool};
persone: [{nome: string, eta: int, studente: bool}];
...
read n;
read x;
read nome;
read flag;
read p;
read persone;
```

⇨

```
READ ^n "i"
READ ^x "r"
READ ^nome "s"
READ ^flag "b"
READ ^p "{nome:s,eta:i,studente:b}"
READ ^persone "[{nome:s,eta:i,studente:b}]"
```

Nota: *format* = stringa che specifica il formato del valore letto

Istruzione `write`

write-stat → *expr*

⇨

⟨*expr*⟩
WRIT *format*

```
n: int;
x: real;
flag: bool;
p: {nome: string, eta: int, studente: bool};
persone: [{nome: string, eta: int, studente: bool}];
...
write n+2;
write x;
write p.nome;
write flag;
write p;
write tail(persone);
```

⇨

```
LOAD 0 ^n
LOCI 2
ADDI
WRIT "i"
LOAD 0 ^x
WRIT "r"
LODA 0 ^p
INDL °nome |ptr|
WRIT "s"
LOAD 0 ^flag
WRIT "b"
LOAD 0 ^p
WRIT "{nome:s,eta:i,studente:b}"
LOAD 0 ^persone
TAIL
WRIT "[{nome:s,eta:i,studente:b}]"
```

Nota: *format* = stringa che specifica il formato del valore stampato

Istruzione Condizionale

if-stat → *expr stat-list*

⟨*expr*⟩
SKPF *exit*
⟨*stat-list*⟩

if-stat → *expr* $stat\text{-}list_1$ $stat\text{-}list_2$

⟨*expr*⟩
SKPF *offset*
$\langle stat\text{-}list_1 \rangle$
SKIP *exit*
$\langle stat\text{-}list_2 \rangle$

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

```
LOAD 0 ^a
LOAD 0 ^b
EQUA
SKPF 7
LODA 0 ^c
LOAD 0 ^a
LOCI 1
ADDI
STOR
SKIP 10
LOAD 0 ^d
LOAD 0 ^a
GTHI
SKPF 6
LODA 0 ^b
LOAD 0 ^b
LOAD 0 ^a
SUBI
STOR
```

Ciclo While

while-stat → *expr stat-list*

```
⟨expr⟩
SKPF exit
⟨stat-list⟩
SKIP up
```

```
a, b, c: int;
c = 0;
while a >= b do
  c = c + 1;
  a = a – b;
end;
```

```
LODA 0 ^c
LOCI 0
STOR
LOAD 0 ^a
LOAD 0 ^b
GEQI
SKPF 12
LODA 0 ^c
LOAD 0 ^c
LOCI 1
ADDI
STOR
LODA 0 ^a
LOAD 0 ^a
LOAD 0 ^b
SUBI
STOR
SKIP -14
```

Ciclo Foreach

foreach-stat → **id**$_1$ **id**$_2$ *stat-list*

```
LODA 0 ^index
LOCI 0
STOR
LOAD 0 ^index
LOAD 0 ^id2
CARD
LTHI
SKPF exit
LODA 0 ^id1
LODA 0 ^id2
LOAD 0 ^index
IXAD |elem of ^id2|
INDL 0 |elem of ^id2|
STOR
⟨stat-list⟩
LODA 0 ^index
LOAD 0 ^index
LOCI 1
ADDI
STOR
SKIP up
```

```
numeri: [real]; x, y: real;
...
foreach x in numeri do
  write x+y;
end;
```

```
LODA 0 ^index
LOCI 0
STOR
LOAD 0 ^index
LOAD 0 ^numeri
CARD
LTHI
SKPF 17
LODA 0 ^x
LODA 0 ^numeri
LOAD 0 ^index
IXAD |real|
INDL 0 |real|
STOR
LOAD 0 ^x
LOAD 0 ^y
ADDR
WRIT "r"
LODA 0 ^index
LOAD 0 ^index
LOCI 1
ADDI
STOR
SKIP -20
```

Nota: `index` = variabile ausiliaria (`int`) per indicizzare l'array (numero di indici ausiliari necessari = massimo livello di innestamento dei cicli `foreach` nella sezione `run` del programma)

Chiamata di Funzione

$func\text{-}call \rightarrow expr_1 \ldots expr_n$

$\langle expr_1 \rangle$
$\langle expr_2 \rangle$
...
$\langle expr_n \rangle$
`PUSH` *n*
`JUMP` *entry*
`APOP`

```
j, k: int;
flag, b: bool;
...
flag = b or alfa(j+k, true);
```

```
LODA 0 ^flag
LOAD 0 ^b
SKPF 3
LOCI 1
SKIP 8
LOAD 0 ^j
LOAD 0 ^k
ADDI
LOCI 1
PUSH 2
JUMP ^alfa
APOP
STOR
```

Note:

- `JUMP`: salto assoluto (invece che relativo come in `SKIP` e `SKPF`)
- $expr_1, expr_2, \ldots, expr_n$ = lista dei parametri attuali
- Argomento di `PUSH` (n) = numero di parametri della funzione
- *entry* = indirizzo della prima istruzione del corpo (espressione) della funzione

Definizione di Funzione

func-decl → *formal-decl-list type expr*

```
FUNC fid
⟨expr⟩
RETN
```

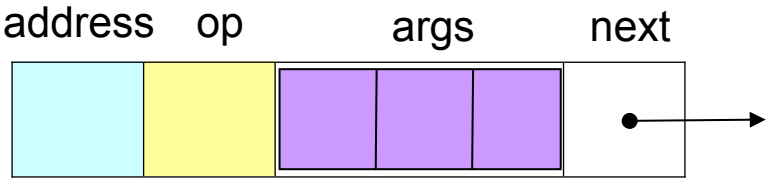
```
alfa(i: int, ok: bool): bool
  ok and i <= 10;
```

```
FUNC ^alfa
LOAD 1 ^ok
SKPF 5
LOAD 1 ^i
LOCI 10
LEQI
SKIP 2
LOCI 0
RETN
```

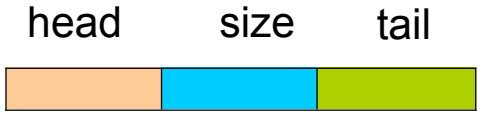
Nota: *fid* = function identifier (1, 2, ...)

Strutture Dati per la Generazione di Codice

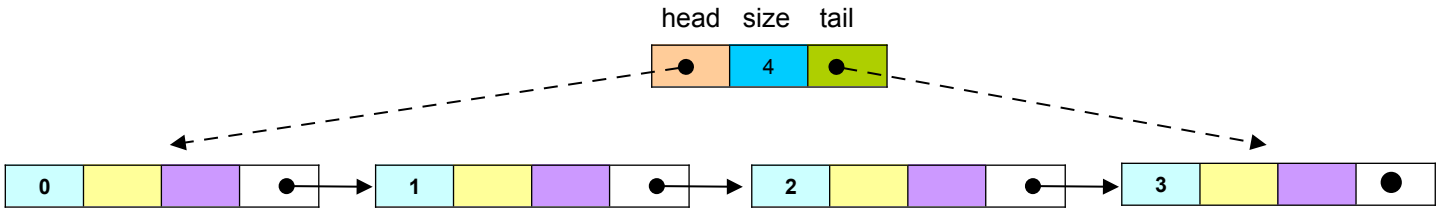
`Stat:`



`Code:`



Rappresentazione di un segmento di codice (lista di istruzioni N-code):



gen.c

```
void relocate(Code code, int offset)
Code appcode(Code code1, Code code2)
Code endcode()
Code concode(Code code1, Code code2, ...)
Stat *newstat(Operator op)
Code makecode(Operator op)
Code makecode1(Operator op, int arg)
Code makecode2(Operator op, int arg1, int arg2)
Code make_func_call(int numparams, int entry)
Code make_locs(char *s)
```

gen.c: relocate()

```
void relocate(Code code, int offset)
{
  Stat *pt = code.head;
  int i;

  for(i = 1; i <= code.size; i++)
  {
    pt->address += offset;
    pt = pt->next;
  }
}
```

gen.c: appcode( )

```
Code appcode(Code code1, Code code2)
{
  Code rescode;

   relocate(code2, code1.size);
   rescode.head = code1.head;
   rescode.tail = code2.tail;
   code1.tail->next = code2.head;
   rescode.size = code1.size + code2.size;
   return rescode;
}
```

gen.c: endcode( ), concode( )

```
Code endcode()
{
  static Code code = {NULL, 0, NULL};

  return code;
}

Code concode(Code code1, Code code2, ...)
{
  Code rescode = code1, *pcode = &code2;

  while(pcode->head != NULL)
  {
    rescode = appcode(rescode, *pcode);
    pcode++;
  }
  return rescode;
}
```

gen.c: newstat(), makecode(), makecode1()

```
Stat *newstat(Operator op)
{
  Stat *pstat;

  pstat = (Stat*)malloc(sizeof(Stat));
  pstat->address = 0;
  pstat->op = op;
  pstat->next = NULL;
  return pstat;
}
```

```
Code makecode(Operator op)
{
  Code code;

  code.head = code.tail = newstat(op);
  code.size = 1;
  return code;
}
```

```
Code makecode1(Operator op, int arg)
{
  Code code;

  code = makecode(op);
  code.head->args[0].ival = arg;
  return code;
}
```

gen.c: make_func_call()

```
Code make_func_call(int numparams, int entry)
{
    return concode(makecode1(PUSH, numparams),
                   makecode1(JUMP, entry),
                   makecode(APOP),
                   endcode());
}
```

gen.c: make_locs()

```
Code make_locs(char *s)
{
    Code code;

    code = makecode(LOCS);
    code.head->args[0].sval = s;
    return code;
}
```