Appunti di Linguaggi di Programmazione

Secondo Modulo
Corso di Informatica

Alessandro Amella

27 dicembre 2024

Indice

1 Nomi e ambiente

1.1 Nomi

1.1.1 Definizione

Un **nome** è una sequenza di caratteri utilizzata per denotare qualcos'altro.
Esempi:

- `const pi = 3.14;` (**pi** denota la costante 3.14)
- `int x;` (**x** denota una variabile)
- `void f(){...};` (**f** denota la definizione di una funzione)

1.1.2 Identificatori e Oggetti Denotati

- Nei linguaggi, i nomi sono spesso **identificatori** (token alfanumerici), ma possono essere anche altro (es: `+`, `:=`).
- L'uso di un nome serve ad indicare l'**oggetto denotato**.
- I nomi rendono gli oggetti simbolici più facili da ricordare.
- I nomi forniscono **astrazione** per:
  - Dati (variabili, tipi, ecc.)
  - Controllo (sottoprogrammi)

1.1.3 Oggetti Denotabili

Un **oggetto denotabile** è un'entità a cui può essere associato un nome.
Tipi di nomi:

- **Definiti dall'utente**: variabili, parametri formali, procedure, tipi definiti dall'utente, etichette, moduli, costanti definite dall'utente, eccezioni.
- **Definiti dal linguaggio**: tipi primitivi, operazioni primitive, costanti predefinite.

Terminologia:

- **Legame (binding)** o **associazione**: la relazione tra un nome e un oggetto.

1.1.4 Binding Time

Il **binding time** indica quando avviene l'associazione tra un nome e un oggetto.

- **Statico (avviene prima o durante l'esecuzione)**
  - *Progettazione del linguaggio*: tipi primitivi, nomi per operazioni e costanti predefinite.
  - *Scrittura del programma*: definizione di alcuni nomi (variabili, funzioni, ecc.) il cui legame sarà completato più tardi.
  - *Compilazione (+ collegamento e caricamento)*: legame di alcuni nomi (variabili globali).
- **Dinamico (avviene durante l'esecuzione)**
  - *Esecuzione*: legame definitivo di tutti i nomi non ancora legati (es: variabili locali ai blocchi).

1.1.5 Terminologia Italiana

- **binding** = legame = associazione
- **environment** = ambiente
- **scope** = portata, estensione (anche: ambito, campo d'azione)
- **lifetime** = vita, o tempo di vita

1.2 Ambiente

1.2.1 Definizione

L'**ambiente** è l'insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma e in uno specifico momento dell'esecuzione.

1.2.2 Dichiarazione

La **dichiarazione** è un meccanismo (implicito o esplicito) con il quale si crea un'associazione nell'ambiente.

1.2.3 Ambiguità e Aliasing

- Lo stesso nome può denotare oggetti distinti in punti diversi del programma.
- **Aliasing**: nomi diversi denotano lo stesso oggetto.
  - Passaggio per riferimento
  - Puntatori
  - Esempio concettuale:

```
1 X
2 Y
3 10 5
```

    Se X e Y fossero alias, cambiando X cambierebbe anche Y.

1.3 Blocchi

1.3.1 Struttura dell'Ambiente

Nei linguaggi moderni, l'ambiente è strutturato mediante l'uso di **blocchi**.

1.3.2 Definizione

Un **blocco** è una regione testuale del programma, identificata da un segnale di inizio e uno di fine, che può contenere dichiarazioni locali a quella regione.

Esempi di sintassi per blocchi:

- `begin...end` (Algol, Pascal)
- `{...}` (C, Java)
- `(...)` (Lisp)
- `let...in...end` (ML)

Tipi di blocchi:

- **Anonimo (o in-line)**
- **Associato ad una procedura**

1.3.3 Motivazioni per l'uso dei Blocchi

- **Gestione locale dei nomi**:

```
{
  int tmp = x;
  x = y;
  y = tmp;
}
```

  - Chiarezza del codice
  - Libertà di scelta dei nomi

- **Ottimizzazione della memoria**: Con un'opportuna allocazione della memoria.
- **Supporto alla ricorsione**.

1.3.4 Annidamento dei Blocchi

- I blocchi si possono sovrapporre solo se **annidati**.
- **Regola di visibilità (preliminare)**: Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che nasconde, o maschera, la precedente).
- Esempio concettuale di annidamento:

```
/* moltiplicazione tra matrici: A = B*C */

for (i=0; i<n; i++) {        /* ciclo più esterno */
  for (j=0; j<m; j++) {      /* ciclo intermedio */
    a[i][j] = 0.0;
    for (k=0; k<l; k++) {    /* ciclo più interno */
      a[i][j] += b[i][k] * c[k][j];
    }
  }
}
```

1.3.5 Suddivisione dell'Ambiente

L'ambiente in uno specifico blocco può essere suddiviso in:

- **Ambiente locale**: Associazioni create all'ingresso nel blocco.
  - Variabili locali
  - Parametri formali
- **Ambiente non locale**: Associazioni ereditate da altri blocchi.
- **Ambiente globale**: Quella parte di ambiente non locale relativo alle associazioni comuni a tutti i blocchi.
  - Dichiarazioni esplicite di variabili globali
  - Dichiarazioni del blocco più esterno
  - Associazioni esportate da moduli, ecc.

1.3.6 Esempio di Ambienti

```
1  my $x=3;              # dichiarazioni per l'ambiente globale
2  my $y=$x;
3  {                     # inizio sottoblocco
4      my $z="foo";       # dichiarazioni per l'ambiente locale
5      my $x=777;
6      $y="$y$z$x";
7      }
8  print "y vale $y, x vale $x\n";
9  print (defined($z) ? "z vale $z\n" : "z non esiste\n");
10
11 OUTPUT> y vale 3foo777, x vale 3
12 OUTPUT> z non esiste
```

1.4 Regole di Scope

1.4.1 Operazioni sull'Ambiente

- **Creazione associazione nome-oggetto denotato (naming)**: dichiarazione locale in blocco.
- **Riferimento oggetto denotato mediante il suo nome (referencing)**: uso di un nome.
- **Disattivazione associazione nome-oggetto denotato**: una dichiarazione maschera un nome.
- **Riattivazione associazione nome-oggetto denotato**: uscita da blocco con dichiarazione che maschera.
- **Distruzione associazione nome-oggetto denotato (unnaming)**: uscita da blocco con dichiarazione locale.

1.4.2 Operazioni sugli Oggetti Denotabili

- Creazione
- Accesso
- Modifica (se l'oggetto è modificabile)
- Distruzione
- La creazione e la distruzione di un oggetto non coincidono con la creazione e la distruzione dei legami per esso.

1.4.3 Eventi Fondamentali e Lifetime

1. Creazione di un oggetto
2. Creazione di un legame per l'oggetto
3. Riferimento all'oggetto, tramite il legame
4. Disattivazione di un legame
5. Riattivazione di un legame
6. Distruzione di un legame
7. Distruzione di un oggetto

- Il tempo tra 1 e 7 è la **vita** (o il **tempo di vita**: lifetime) dell'oggetto.
- Il tempo tra 2 e 6 è la vita dell'associazione.

1.4.4 Vita dell'Oggetto vs. Vita del Legame

- La vita di un oggetto non coincide con la vita dei legami per quell'oggetto.
- **Vita dell'oggetto più lunga di quella del legame:**
  - Variabile passata ad una procedura per riferimento (Pascal: `var`)

```
procedure P (var X: integer); begin ... end;
...
var A: integer;
...
P(A);
```

Durante l'esecuzione di P esiste un legame tra X e un oggetto che esiste prima e dopo tale esecuzione.

- **Vita dell'oggetto più breve di quella del legame:**
  - Area di memoria dinamica deallocata

```
int *X, *Y;
...
X = (int *) malloc (sizeof (int));
Y = X;
...
free (X);
X = null;
```

Dopo la `free` non esiste più l'oggetto, ma esiste ancora un legame ("pendente") per esso (Y): **dangling reference**.

1.4.5 Interpretazione della Regola di Visibilità

Come deve essere interpretata la regola di visibilità in presenza di procedure (blocchi eseguiti in posizioni diverse dalla loro definizione) e ambiente non locale (e non globale)?

1.4.6 Scope Statico vs. Scope Dinamico

```
{
  int x = 10;
  void foo() {
    x++;
  }
  void fie() {
    int x = 0;
    foo();
  }
  fie();
}
```

Quale `x` incrementa `foo`?

- **Scope statico**: Un riferimento non-locale in un blocco B è risolto nel blocco che include sintatticamente B.
- **Scope dinamico**: Un riferimento non-locale in un blocco B è risolto nel blocco che è eseguito immediatamente prima di B.

1.4.7 Esempio Scope Statico vs. Dinamico

```
int x = 4;
void foo() { print x; }
void bar() {
  int x = 0;
  foo();
}
main() { bar(); }
```

- **Scope statico**: stampa 4
- **Scope dinamico**: stampa 0

1.4.8 Vantaggi dello Scope Statico

- **Indipendenza dalla posizione di chiamata**: il corpo di `foo` è parte dello scope della `x` più esterna.
- **Indipendenza dai nomi locali**: la modifica del locale `x` in `y` dentro `fie` non ha effetto in scope statico.
- **Principio di indipendenza**: Ridenominazioni consistenti dei nomi locali di un programma non devono avere effetto sulla semantica del programma stesso.

```
{
  int x = 10;
```

```
  void foo() { x++; }
  void fie() { int x = 0; foo(); }
  fie();
  foo();
}
```

La chiamata di `foo` interna a `fie` e quella nel `main` accedono alla stessa variabile: la `x` esterna.

1.4.9 Esempio di Scope Dinamico

```
// Scope Dinamico: specializzare una funzione
visualizza è una procedura che rende a colore sul video una certa forma
```

```
/ the familiar languages, we consider the
// following pseudo code as our example. It
// prints 20 in a language that uses dynamic
// scoping.

int x = 10;

// Called by g()
int f()
{
   return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
   int x = 20;
   return f();
}

main()
{
  printf(g());
}
```

1.4.10 Confronto Scope Statico e Dinamico

- **Scope statico (scoping statico, statically scoped, lexical scope)**:
  - Informazione completa dal testo del programma.
  - Le associazioni sono note a tempo di compilazione.
  - Principi di indipendenza.
  - Concettualmente più complesso da implementare ma più efficiente.
  - Linguaggi: Algol, Pascal, C, Java, ...
- **Scope dinamico (scoping dinamico, dynamically scoped)**:
  - Informazione derivata dall'esecuzione.

  - Spesso causa di programmi meno “leggibili”.
  - Concettualmente più semplice da implementare, ma meno efficiente.
  - Linguaggi: Lisp (alcune versioni), Perl.
- Differiscono solo in presenza congiunta di:
  - Ambiente non locale e non globale
  - Procedure

1.4.11 Attenzione al C

- Algol, Pascal, Ada, Java permettono di annidare blocchi di sottoprogrammi.
- Non possibile in C:
  - Funzioni definite solo nel blocco più esterno.
  - Dunque, in una funzione l’ambiente è partizionato in locale e globale.
  - Non si presenta il problema dei non-locali.
- Questo non vuol dire che la regola di scoping (statico o dinamico) sia indifferente in C!
- Significa solo che è semplice determinare dove risolvere un non-locale: ogni non-locale viene risolto nell’ambiente globale (ovvero: ci sono solo locali e globali).
- Esempio in C:

```c
int x = 10;
void foo() { x++; }
void fie() { int x = 0; foo(); }
main() { fie(); foo(); }
```

1.4.12 Scope Dinamico Simulato in C con Macro

```c
int x = 10;
int next_x(int delta) { return x + delta; }
#define NEXT_X(delta) x + delta
main() {
  int x = 5;
  printf("%d, %d\n", next_x(4), NEXT_X(4)); // Stampa 14, 9
}
```

Una situazione di "scope dinamico" si realizza in C con le "macro". La definizione di C dice che `define` corrisponde ad una sostituzione testuale.

1.4.13 Determinare l’Ambiente

L’ambiente è dunque determinato da:

- Regola di scope (statico o dinamico)
- Regole specifiche, p.e.:
  - Quando è visibile una dichiarazione nel blocco in cui compare?
- Regole per il passaggio dei parametri
- Regole di binding (shallow o deep) - intervengono quando una procedura P è passata come parametro ad un’altra procedura mediante il formale X (discusso più avanti).

1.4.14 Regole Specifiche di Visibilità

- **Dichiarazioni all'interno del blocco:**
  - *A partire dalla dichiarazione e fino alla fine del blocco:*
    - Java: dichiarazione di una variabile

```
{
  a = 1; // Errore!
  int a;
  ...
}
```

  - *Sempre (dunque anche prima) della dichiarazione:*
    - Java: dichiarazione di un metodo (permette metodi mutuamente ricorsivi)

```
{
  void f() { g(); } // Ok
  void g() { f(); } // Ok
  ...
}
```

1.4.15 Problemi di Scope in Pascal

- Lo scope di una dichiarazione è l'intero blocco dove essa appare - eccetto i buchi.
- Ogni identificatore deve essere dichiarato prima di essere usato.

```
const a = -1;
procedure pippo;
const b = a; // Errore di semantica statica (o b = -1 in alcuni casi)!
...
const a = 0;
```

1.4.16 Altro Esempio di Problemi in Pascal

```
procedure pippo;
...
end (*pippo*);

procedure A;
...
 procedure B
 begin
   ...
   pippo; // Errore di semantica statica
 end (*B*);
...

procedure pippo;
begin ... end;
```

1.4.17 Mutua Ricorsione

Come gestire la mutua ricorsione (funzioni, tipi) in linguaggi dove un nome deve essere dichiarato prima di essere usato?

- Rilasciare tale vincolo per funzioni e/o tipi.
- Esempi:
  - Java per i metodi
  - Pascal per tipi puntatore
  - Definizioni incomplete (Ada, Pascal, C)

Esempi di Mutua Ricorsione: Ada:

```
type elem;
type lista is access elem;
type elem is record
  info: integer;
  next: lista;
end;
```

Pascal:

```
type lista = ^elem;
elem = record
  info : integer;
  next : lista;
end;
```

Java:

```
{
  void f() { g(); }
  void g() { f(); }
}
```

C:

```
struct elem;
struct elem{
  int info;
  elem *next;
};
```

1.5 Esercizi I

1.5.1 Esercizio 1

Con la notazione $C^{L_k}_{L_i \to L_e}$ indichiamo un compilatore dal linguaggio L_i al linguaggio L_e scritto in L_k. Con I^j_L indichiamo un interprete scritto in L per il linguaggio L_j. Se P è un programma in L_i e x un suo dato, $I^i_L(P, x)$ indica l'applicazione dell'interprete a P e x. Si dica se la seguente scrittura

ha senso $I_t(C^i_{k\to e}, H_t)$. Se la risposta è "no", si motivi tale fatto; se è "si" si dica qual è il risultato ottenuto.

Soluzione: La scrittura ha senso se e solo se il compilatore $C^i_{k\to e}$ è un programma scritto nel linguaggio che l'interprete I_t è in grado di interpretare, ovvero L_t. Quindi, affinché l'espressione abbia senso, è necessario che $i = t$.

Se la scrittura ha senso (cioè se $i = t$), il risultato dell'espressione è l'esecuzione del compilatore $C^t_{k\to e}$ (perché $i = t$) sull'input H_t. Assumendo che H_t sia un input valido per il compilatore (anche se un compilatore tipicamente non prende un "dato" nel senso tradizionale, potrebbe rappresentare delle opzioni o un file di input), il risultato sarebbe un programma equivalente a quello scritto in L_k, ma tradotto in L_e. Quindi, il risultato è un programma in L_e.

1.5.2 Esercizio 2

Ricordando che $I^j_{L_g}$ denota un interprete scritto in L_g che interpreta programmi scritti in L_j, e che $C^{L_i}_{L_o\to L_3}$ denota un compilatore scritto in L_i che traduce programmi scritti in L_o in equivalenti programmi scritti in L_3, affinché la seguente espressione

$$I_X(C^{L_Y}_{L_g\to L_3}, C^{L_i}_{L_o\to L_Z})$$

abbia senso, quali linguaggi devono essere assegnati alle variabili X, Y e Z? Nel caso in cui Z assuma valore L_g, il programma calcolato serve a qualcosa?

Soluzione: Affinché l'espressione abbia senso, l'interprete I_X deve essere in grado di interpretare il primo argomento, che è il compilatore $C^{L_Y}_{L_g\to L_3}$. Un compilatore è un programma, e in questo caso è scritto in L_Y. Quindi, deve essere $X = Y$.

Il secondo argomento dell'interprete è $C^{L_i}_{L_o\to L_Z}$, che viene trattato come il "dato" in input al programma interpretato. Il programma interpretato è il compilatore $C^{L_Y}_{L_g\to L_3}$. Un compilatore prende come input un programma scritto nel suo linguaggio sorgente. Quindi, il "dato" $C^{L_i}_{L_o\to L_Z}$ deve essere un programma scritto in L_g. Questo significa che $L_o = g$.

Non ci sono restrizioni dirette su Z affinché l'espressione abbia senso sintatticamente. Quindi:

- X deve essere uguale a Y.
- g deve essere uguale a L_o.
- Z può essere qualsiasi linguaggio.

Nel caso in cui Z assuma valore L_g, il secondo argomento diventa $C^{L_i}_{L_o\to L_g}$. Siccome $L_o = g$, questo è un compilatore da L_g a L_g scritto in L_i. L'interprete I_Y (ricordiamo che $X = Y$) esegue il compilatore $C^{L_Y}_{L_g\to L_3}$ con input il compilatore $C^{L_i}_{L_g\to L_g}$. Il compilatore $C^{L_Y}_{L_g\to L_3}$ prende un programma in L_g e lo traduce in L_3. In questo caso, il programma in input è il compilatore $C^{L_i}_{L_g\to L_g}$. Quindi, il risultato è la traduzione del compilatore da L_g a L_g (scritto in L_i) in un programma equivalente scritto in L_3. Questo programma risultante è un compilatore da L_g a L_g scritto in L_3. Questo processo è legato al concetto di *meta-compilazione* e può essere utile, ad esempio, per portare un compilatore su una nuova piattaforma.

1.5.3 Esercizio 3

Con la notazione $C^{L_h}_{L_i\to L_p}$ indichiamo un compilatore da L_i a L_p scritto in L_h. Con I^j_L indichiamo un interprete scritto in L per il linguaggio L_j; se P è un programma in L_i e x un suo dato, $I^i_L(P, x)$ indica l'applicazione dell'interprete a P e x. Si dica in meno di 10 parole cosa è $I_{L_y}(C^{L_h}_{L_i\to L_y})$.

Soluzione: È un compilatore da L_i a L_y se $h = y$.

1.5.4 Esercizio 4

La jae espressione

$$I_{L_y}(a^g_{L_o}, I^i_i)$$

calcola qualcosa di utile? Se rimpiazziamo, nell'espressione sopra, la seconda occorrenza di I con I^o_o, cosa otteniamo?

Soluzione: L'espressione $I_{L_y}(a^g_{L_o}, I^i_i)$ ha senso se $a^g_{L_o}$ è un programma scritto in L_y. La notazione $a^g_{L_o}$ è un po' ambigua, ma se intendiamo che a è un programma scritto in L_o e stiamo cercando di interpretarlo con un interprete scritto in L_y, allora affinché l'espressione esterna abbia senso, $a^g_{L_o}$ deve essere un programma in L_y. Se così fosse, l'interprete I_{L_y} esegue il programma $a^g_{L_o}$ con input I^i_i. I^i_i è un interprete per L_i scritto in L_i. L'utilità dipende da cosa fa il programma a_{L_y} con l'interprete I^i_i come input.

Se rimpiazziamo la seconda occorrenza di I con I^o_o, otteniamo:

$$I_{L_y}(a^g_{L_o}, I^o_o)$$

Anche in questo caso, affinché l'espressione abbia senso, $a^g_{L_o}$ deve essere un programma in L_y. L'interprete I_{L_y} esegue il programma $a^g_{L_o}$ con input I^o_o. I^o_o è un interprete per L_o scritto in L_o. Se a è effettivamente un programma in L_o, allora il risultato potrebbe essere l'esecuzione di a (interpretato da qualche meccanismo interno ad a) sull'interprete I^o_o come dato. L'utilità dipende dal comportamento specifico di a.

1.6 Esercizi II

1.6.1 Esercizio 1

Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudo-linguaggio che usa scoping statico e passaggio di parametri per valore. La primitiva `write(x)` permette di stampare un valore intero.

```
1  {
2      int x = 2;
3      void pippo(value int y){
4          x = x + y;
5      }
6      {
7          int x = 5;
8          pippo(x++);
9          write(x);
10     }
11     write(x);
12 }
```

(si ricordi che un comando della forma `foo(w++)`; passa a `foo` il valore corrente di `w` e poi incrementa `w` di uno)

Soluzione: Analizziamo il codice passo passo:

1. Viene dichiarato un intero `x` con valore 2 nello scope esterno.
2. Viene dichiarata la funzione `pippo`. Nello scope di `pippo`, `x` si riferisce alla `x` dello scope esterno (scoping statico).
3. Si entra in un nuovo blocco.
4. Viene dichiarato un nuovo intero `x` con valore 5 in questo scope interno. Questa `x` nasconde la `x` dello scope esterno all'interno di questo blocco.

5. Viene chiamata la funzione `pippo` con l'espressione `x++` come argomento. Il valore attuale di `x` (quella interna, che è 5) viene passato alla funzione, e poi `x` viene incrementata a 6.

6. All'interno di `pippo`, `y` riceve il valore 5. La riga `x = x + y;` nello scope di `pippo` modifica la `x` dello scope esterno: $2 + 5 = 7$.

7. Si esegue `write(x)`; in questo scope interno, `x` è quella dichiarata localmente, che ha valore 6. Quindi viene stampato **6**.

8. Si esce dal blocco interno.

9. Si esegue `write(x)`; ora `x` si riferisce alla `x` dello scope esterno, che è stata modificata da `pippo` e ha valore 7. Quindi viene stampato **7**.

Output:

```
6
7
```

1.6.2 Esercizio 2

Si consideri il seguente frammento di programma scritto in uno pseudo-linguaggio che usi scoping dinamico e dove la primitiva `read(Y)` permette di leggere nella variabile `Y` un intero dall'input standard, mentre `write(X)` permette di stampare il valore della variabile `X`. Si dica quali sono (o qual è) i valori stampati.

```
1  int X;
2  K = 14;
3  int Y;
4  void fie {
5      foo;
6      X = 0;
7  }
8  void foo {
9      int X;
10     KX = 5;
11 }
12 read(Y);
13 if Y > 0 then {
14     int X;
15     X = 4;
16     fie;
17 }
18 else {
19     fie;
20 }
21 write(X);
```

Soluzione: Assumiamo che l'input per `read(Y)` sia un valore maggiore di 0.

1. Vengono dichiarate le variabili globali `X` e `Y`.

2. Viene dichiarata la funzione `fie`.

3. Viene dichiarata la funzione `foo`.

4. Viene letto un valore dall'input e assegnato a `Y`.

5. Se `Y` è maggiore di 0 (assumiamo di sì):
   (a) Viene dichiarata una variabile locale `X` e le viene assegnato il valore 4.
   (b) Viene chiamata la funzione `fie`.
   (c) All'interno di `fie`, viene chiamata `foo`.
   (d) All'interno di `foo`, viene dichiarata una variabile locale `X`.
   (e) L'istruzione `KX = 5;` contiene un errore di sintassi (probabilmente intendeva `X = 5;`). Con scoping dinamico, `X` si riferisce alla `X` dichiarata più recentemente nella catena di chiamate, che è quella locale a `foo`. Quindi, la `X` locale a `foo` assume il valore 5.
   (f) Si torna a `fie`. L'istruzione `X = 0;` con scoping dinamico si riferisce alla `X` dichiarata più recentemente nello scope corrente o negli scope chiamanti. In questo caso, la `X` più recente è quella globale, quindi la `X` globale assume il valore 0.
6. Se `Y` non fosse maggiore di 0, il blocco `else` eseguirebbe `fie`, con lo stesso effetto sulla `X` globale.
7. Infine, viene eseguito `write(X)`. Con scoping dinamico, `X` si riferisce alla `X` più recente nello scope corrente, che è la `X` globale.

Il valore stampato è il valore della `X` globale, che è stata impostata a 0 all'interno di `fie`.
Output:

```
0
```

1.6.3 Esercizio 3

Si consideri il seguente frammento di programma scritto in uno pseudo-linguaggio che usi scoping dinamico e dove la primitiva `read(Y)` permette di leggere nella variabile `Y` un intero dall'input standard, mentre `write(X)` permette di stampare il valore della variabile `X`. Si dica quali sono i valori stampati.

```
1  int X = 0;
2  int Y;
3  void pippo() {
4      X++;
5  }
6  void pluto() {
7      X++;
8      pippo;
9  }
10 read(Y);
11 if Y > 0 then {
12     int X = 5;
13     pluto();
14 }
15 else {
16     pluto();
17 }
18 write(X);
```

Soluzione: Assumiamo che l'input per `read(Y)` sia un valore maggiore di 0.

1. Viene dichiarata la variabile globale `X` inizializzata a 0.
2. Viene dichiarata la variabile globale `Y`.

3. Viene dichiarata la funzione `pippo`. Con scoping dinamico, `X` in `pippo` si riferirà alla `X` nello scope chiamante.
4. Viene dichiarata la funzione `pluto`.
5. Viene letto un valore per `Y`.
6. Se `Y > 0`:
   (a) Viene dichiarata una variabile locale `X` inizializzata a 5.
   (b) Viene chiamata `pluto`.
   (c) All'interno di `pluto`:
      i. `X++`: Con scoping dinamico, `X` si riferisce alla `X` dichiarata più recentemente, che è quella locale al blocco `if`, quindi la `X` locale diventa 6.
      ii. `pippo;`: Viene chiamata `pippo`.
      iii. All'interno di `pippo`: `X++`: Con scoping dinamico, `X` si riferisce alla `X` nello scope chiamante, che è la `X` locale al blocco `if`. Quindi, la `X` locale diventa 7.
7. Se `Y <= 0`, il blocco `else` esegue `pluto()`. In questo caso, all'interno di `pluto` e `pippo`, `X` si riferirebbe alla `X` globale, incrementandola due volte.
8. Infine, viene eseguito `write(X)`. Con scoping dinamico, `X` si riferisce alla `X` nello scope corrente, che è la `X` globale.

Nel caso in cui `Y > 0`, la `X` globale non viene modificata all'interno del blocco `if`. Le modifiche avvengono sulla `X` locale al blocco `if`.

Output:

```
0
```

1.6.4 Esercizio 4

Si dica cosa stampa il seguente frammento in uno pseudolinguaggio scope statico (si ricordi che l'espressione `X++` restituisce il valore della variabile `x` e successivamente incrementa `x` di uno).

```
1  int x = 3;
2  void foo(int y) {
3      int x = 5;
4      x = x + y;
5      x = x + y;
6      write(x);
7      write(y);
8  }
9  foo(x++);
10 write(x);
```

Soluzione: Analizziamo il codice con scoping statico:

1. Viene dichiarata la variabile globale `x` con valore 3.
2. Viene dichiarata la funzione `foo`. All'interno di `foo`, `x` si riferisce alla `x` locale di `foo`.
3. Viene chiamata `foo(x++)`. Il valore attuale di `x` (globale) è 3, quindi 3 viene passato come argomento a `foo`. Successivamente, `x` globale viene incrementata a 4.
4. All'interno di `foo`:

(a) Viene dichiarata una variabile locale `x` con valore 5.

(b) `x = x + y;`: La `x` locale diventa $5 + 3 = 8$.

(c) `x = x + y;`: La `x` locale diventa $8 + 3 = 11$.

(d) `write(x);`: Stampa il valore della `x` locale, che è **11**.

(e) `write(y);`: Stampa il valore di `y`, che è 3 (passato per valore), quindi stampa **3**.

5. Dopo la chiamata a `foo`, viene eseguito `write(x);`. Questa `x` si riferisce alla `x` globale, che è stata incrementata a 4. Quindi stampa **4**.

Output:

```
11
3
4
```

1.6.5 Esercizio 5

Si dica cosa stampa il seguente frammento in uno pseudolinguaggio con passaggio per riferimento e scope statico.

```
1  int x = 2;
2  void foo(reference int y){
3      x x + 1;
4      y y + 10;
5      x x * y;
6      write(x);
7  }
8  {
9      int x = 10;
10     foo(x);
11     write(x);
12 }
```

Soluzione: Analizziamo il codice con scoping statico e passaggio per riferimento:

1. Viene dichiarata la variabile globale `x` con valore 2.
2. Viene dichiarata la funzione `foo`. All'interno di `foo`, la `x` si riferisce alla `x` globale.
3. Si entra in un nuovo blocco.
4. Viene dichiarata una variabile locale `x` con valore 10.
5. Viene chiamata `foo(x);`. La `x` passata come argomento è quella locale con valore 10. Poiché il passaggio è per riferimento, `y` diventa un alias per la `x` locale.
6. All'interno di `foo`:
   (a) `x x + 1;`: Si riferisce alla `x` globale, quindi la `x` globale diventa $2 + 1 = 3$.
   (b) `y y + 10;`: `y` è un riferimento alla `x` locale, quindi la `x` locale diventa $10 + 10 = 20$.
   (c) `x x * y;`: Si riferisce alla `x` globale moltiplicata per il valore attuale di `y` (che è la `x` locale). La `x` globale diventa $3 \times 20 = 60$.
   (d) `write(x);`: Stampa il valore della `x` globale, che è **60**.

7. Dopo la chiamata a `foo`, viene eseguito `write(x);`. Questa `x` si riferisce alla `x` locale, che è stata modificata all'interno di `foo` tramite il passaggio per riferimento e ha valore 20. Quindi stampa **20**.

Output:

```
60
20
```

1.6.6 Esercizio 6

Si dica cosa stampa il seguente frammento in uno pseudolinguaggio con passaggio per valore e scope statico.

```
int x 3;
int y 4;
void foo(int y, int z) {
    int x = 5;
    y = y + 1;
    if (z==y) write(x);
    else write (y);
}
foo(x,x);
write(x);
write(y);
```

Soluzione: Analizziamo il codice con scoping statico e passaggio per valore:

1. Vengono dichiarate le variabili globali `x` con valore 3 e `y` con valore 4.
2. Viene dichiarata la funzione `foo`. All'interno di `foo`, `x` si riferisce alla `x` locale di `foo`.
3. Viene chiamata `foo(x,x);`. I valori delle `x` globale (3) vengono passati per valore a `y` e `z` di `foo`. Quindi, all'interno di `foo`, `y` vale 3 e `z` vale 3.
4. All'interno di `foo`:
   (a) Viene dichiarata una variabile locale `x` con valore 5.
   (b) `y = y + 1;`: Il parametro `y` (locale a `foo`) diventa $3 + 1 = 4$.
   (c) `if (z==y)`: Si confronta `z` (che è 3) con `y` (che è 4). La condizione è falsa.
   (d) `else write (y);`: Viene stampato il valore di `y` (locale a `foo`), che è **4**.
5. Dopo la chiamata a `foo`, viene eseguito `write(x);`. Questa `x` si riferisce alla `x` globale, che vale 3. Quindi stampa **3**.
6. Viene eseguito `write(y);`. Questa `y` si riferisce alla `y` globale, che vale 4. Quindi stampa **4**.

Output:

```
4
3
4
```

1.6.7 Esercizio 7

Si dica cosa stampa il seguente frammento in uno pseudolinguaggio con scope dinamico

```
int x = 4;
void foo(int y) {
    int w;
    x = x + y;
    w = y;
    write(w);
    write(y);
}
{
    int x = 10;
    foo(x);
    write(x);
}
```

Soluzione: Analizziamo il codice con scoping dinamico:

1. Viene dichiarata la variabile globale `x` con valore 4.
2. Viene dichiarata la funzione `foo`.
3. Si entra in un nuovo blocco.
4. Viene dichiarata una variabile locale `x` con valore 10.
5. Viene chiamata la funzione `foo(x);`. L'argomento passato è il valore della `x` locale, che è 10. Quindi, all'interno di `foo`, `y` assume il valore 10.
6. All'interno di `foo`:
   (a) Viene dichiarata una variabile locale `w`.
   (b) `x = x + y;`: Con scoping dinamico, `x` si riferisce alla `x` nello scope chiamante, che è la `x` locale del blocco corrente (valore 10). Quindi, la `x` locale diventa $10 + 10 = 20$.
   (c) `w = y;`: `w` assume il valore di `y`, che è 10.
   (d) `write(w);`: Stampa il valore di `w`, che è **10**.
   (e) `write(y);`: Stampa il valore di `y`, che è **10**.
7. Dopo la chiamata a `foo`, viene eseguito `write(x);`. Con scoping dinamico, `x` si riferisce alla `x` nello scope corrente, che è la `x` locale del blocco, modificata a 20. Quindi stampa **20**.

Output:

```
10
10
20
```

1.6.8 Esercizio 8

Si dica cosa viene stampato dal seguente frammento di codice scritto in uno pseudo-linguaggio che usa scoping statico e passaggio di parametri per valore e per riferimento.

```
int x = 0;
void pippo(value int y, rif int z){
    z = x + y + z;
}
{
    int x = 1;
    int y = 10;
    int z = 20;
    pippo(x++, x);
    pippo(x++, x);
    write(x);
}
write(x);
```

Soluzione: Analizziamo il codice con scoping statico e passaggio di parametri per valore e riferimento:

1. Viene dichiarata la variabile globale `x` con valore 0.
2. Viene dichiarata la funzione `pippo`. All'interno di `pippo`, `x` si riferisce alla `x` globale.
3. Si entra in un nuovo blocco.
4. Vengono dichiarate le variabili locali `x` con valore 1, `y` con valore 10 e `z` con valore 20.
5. Prima chiamata a `pippo(x++, x);`:
   - `x++`: Il valore attuale di `x` (locale) è 1, che viene passato per valore al parametro `y` di `pippo`. Poi `x` (locale) viene incrementato a 2.
   - `x`: Il valore attuale di `x` (locale) è 2, che viene passato per riferimento al parametro `z` di `pippo`. Quindi `z` in `pippo` diventa un alias per la `x` locale.
   - All'interno di `pippo`: `z = x + y + z;`. La `x` è quella globale (0), `y` è 1, e `z` si riferisce alla `x` locale (attualmente 2). Quindi, la `x` locale diventa $0 + 1 + 2 = 3$.
6. Seconda chiamata a `pippo(x++, x);`:
   - `x++`: Il valore attuale di `x` (locale) è 3, che viene passato per valore al parametro `y` di `pippo`. Poi `x` (locale) viene incrementato a 4.
   - `x`: Il valore attuale di `x` (locale) è 4, che viene passato per riferimento al parametro `z` di `pippo`. Quindi `z` in `pippo` diventa un alias per la `x` locale.
   - All'interno di `pippo`: `z = x + y + z;`. La `x` è quella globale (0), `y` è 3, e `z` si riferisce alla `x` locale (attualmente 4). Quindi, la `x` locale diventa $0 + 3 + 4 = 7$.
7. Viene eseguito `write(x);`. Questa `x` si riferisce alla `x` locale, che ha valore 7. Quindi stampa **7**.
8. Si esce dal blocco.
9. Viene eseguito `write(x);`. Questa `x` si riferisce alla `x` globale, che ha valore 0. Quindi stampa **0**.

Output:

```
7
0
```

2 Memoria

2.1 Tipi di Allocazione della Memoria

La vita di un oggetto in memoria è gestita, generalmente, attraverso tre meccanismi di allocazione:

- **Statica**: La memoria viene allocata a tempo di compilazione. Pensa alle variabili globali, quelle che stanno lì fin dall'inizio.
- **Dinamica**: La memoria viene allocata durante l'esecuzione del programma. Qui abbiamo due sottotipi principali:
  - **Pila (Stack)**: Gli oggetti vengono allocati e deallocati seguendo una politica LIFO (Last-In, First-Out). Immagina una pila di piatti: l'ultimo che metti è il primo che togli.
  - **Heap**: Gli oggetti possono essere allocati e deallocati in qualsiasi momento e ordine. È come uno spazio condiviso dove chiedi un pezzo quando ti serve e lo rilasci quando hai finito. I puntatori giocano un ruolo fondamentale qui!

2.2 Allocazione Statica

2.2.1 Caratteristiche

- Un oggetto allocato staticamente ha un indirizzo di memoria fisso, assoluto, che rimane lo stesso per tutta la durata dell'esecuzione del programma. È come avere un posto riservato.
- Tipicamente, vengono allocati staticamente:
  - Variabili globali (quelle dichiarate fuori da qualsiasi funzione o blocco).
  - Variabili locali di sottoprogrammi che non sono ricorsivi (altrimenti si creerebbe casino con più istanze).
  - Costanti il cui valore è noto a tempo di compilazione.
  - Tabelle utilizzate dal supporto a run-time del linguaggio (per il controllo dei tipi, garbage collection, ecc.).
- Spesso, per queste zone di memoria si utilizzano aree protette, per evitare scritture accidentali.

2.2.2 Limitazioni: La Ricorsione

L'allocazione statica non supporta la ricorsione. Immagina una funzione che chiama se stessa:

```
1 SUBROUTINE ERROR (N)
2   IF (N.LE.1) RETURN
3   CALL ERROR(N-1)
4   PRINT N
5 END
```

Se provassimo a eseguirla con allocazione statica, avremmo un'unica area di memoria per la variabile 'N' e per l'indirizzo di ritorno.

La prima chiamata di 'ERROR' memorizzerebbe il valore di 'N' e l'indirizzo di ritorno. La successiva chiamata sovrascriverebbe questi valori, causando la perdita dell'indirizzo di ritorno originale e un comportamento errato del programma.

2.3 Allocazione Dinamica: Pila (Stack)

2.3.1 Necessità della Pila

Quando c'è di mezzo la ricorsione, l'allocazione statica non basta più. Durante l'esecuzione, possono esistere più "istanze" della stessa variabile locale di una procedura.

2.3.2 Record di Attivazione (Activation Record o Frame)

Ogni volta che viene invocato un sottoprogramma (o si entra in un blocco), viene creata una porzione di memoria chiamata **record di attivazione** (RdA) o **frame**. Questo RdA contiene informazioni specifiche per quella particolare invocazione:

- Indirizzo di ritorno (dove tornare una volta finito il sottoprogramma).
- Parametri passati al sottoprogramma.
- Variabili locali del sottoprogramma.
- Risultati intermedi dei calcoli.

Analogamente, anche un blocco di codice (come quelli delimitati da ‘‘) può avere il suo record di attivazione, anche se è una gestione più semplificata.

2.3.3 La Pila: La Struttura LIFO Perfetta

La **pila** (stack) è la struttura dati ideale per gestire i record di attivazione perché le chiamate a procedura (anche ricorsive) e i blocchi sono naturalmente annidati uno dentro l'altro. L'ultima funzione chiamata è la prima a terminare, proprio come una pila di piatti.

Anche in linguaggi senza ricorsione, la pila può essere utile per memorizzare le variabili locali e risparmiare memoria.

2.3.4 Gestione della Pila

La gestione della pila è orchestrata da:

- **Sequenza di chiamata**: Il codice eseguito dal chiamante immediatamente prima di invocare la procedura. Si occupa di preparare i parametri e salvare l'indirizzo di ritorno.
- **Prologo**: Il codice eseguito all'inizio del blocco o della procedura chiamata. Crea il record di attivazione e alloca spazio per le variabili locali.
- **Epilogo**: Il codice eseguito alla fine del blocco o della procedura. Recupera il valore di ritorno (se presente) e dealloca lo spazio del record di attivazione.
- **Sequenza di ritorno**: Il codice eseguito dal chiamante subito dopo che la procedura ha terminato. Recupera il risultato (se presente) e continua l'esecuzione.

2.3.5 Puntatore al Record di Attivazione (SP)

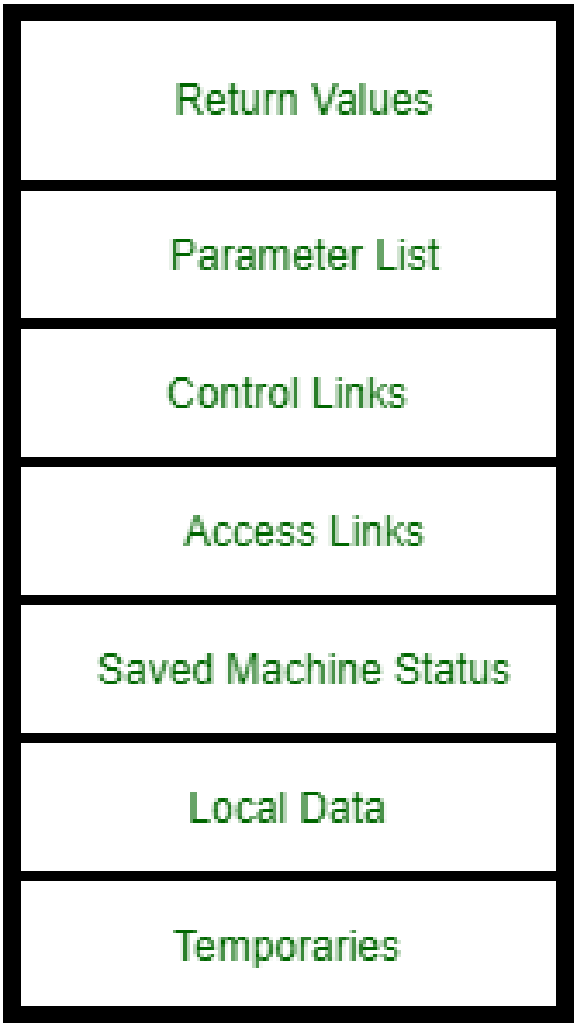
L'indirizzo di un RdA non è noto a tempo di compilazione. Un registro speciale, chiamato **Puntatore al Record di Attivazione** (SP - Stack Pointer), punta al RdA del blocco o della procedura attualmente in esecuzione.

Le informazioni all'interno di un RdA sono accessibili tramite un **offset** rispetto al valore di SP:

$$\text{indirizzo-informazione} = \text{contenuto(SP)} + \text{offset}$$

L'offset è calcolabile staticamente dal compilatore, quindi l'accesso alle variabili locali è efficiente.

2.3.6 Record di Attivazione per Blocchi In-line



Quando si entra in un blocco:

- Si effettua un **Push** sulla pila, creando spazio per il nuovo record di attivazione.
- Il **link dinamico** del nuovo RdA viene impostato al valore corrente di SP (puntando al RdA precedente).
- SP viene aggiornato per puntare al nuovo RdA.

Quando si esce da un blocco:

- Si effettua un **Pop** dalla pila, eliminando il RdA corrente.
- SP viene ripristinato al valore del link dinamico del RdA appena rimosso.

In molti linguaggi, per i blocchi anonimi (in-line) non c'è una vera e propria manipolazione della pila. Il compilatore raccoglie tutte le dichiarazioni dei blocchi annidati e alloca spazio per tutte le variabili locali in un unico record di attivazione della funzione che li contiene. Questo può portare a un potenziale spreco di memoria, ma evita l'overhead della gestione della pila per ogni blocco.

2.3.7 Record di Attivazione per Procedure: Dettagli

Il record di attivazione per una procedura contiene più informazioni:

- **Link Dinamico (Control Link)**: Un puntatore al record di attivazione precedente sulla pila (quello della procedura chiamante). Fondamentale per il ritorno dalla funzione.
- **Link Statico**: (Vedremo dopo, serve per lo scope statico).
- **Indirizzo di Ritorno**: L'indirizzo nel codice del chiamante dove riprendere l'esecuzione.
- **Indirizzo del Risultato**: La locazione dove memorizzare il valore di ritorno della funzione.
- **Parametri**: I valori passati alla funzione.
- **Variabili Locali**: Lo spazio per le variabili dichiarate all'interno della funzione.
- **Risultati Intermedi**: Spazio temporaneo per calcoli.

Il **Puntatore RdA** punta all'inizio del record di attivazione corrente. Il puntatore al top della pila indica la cima dello spazio attualmente allocato per la pila.

Perché Link Dinamico e Puntatore RdA?

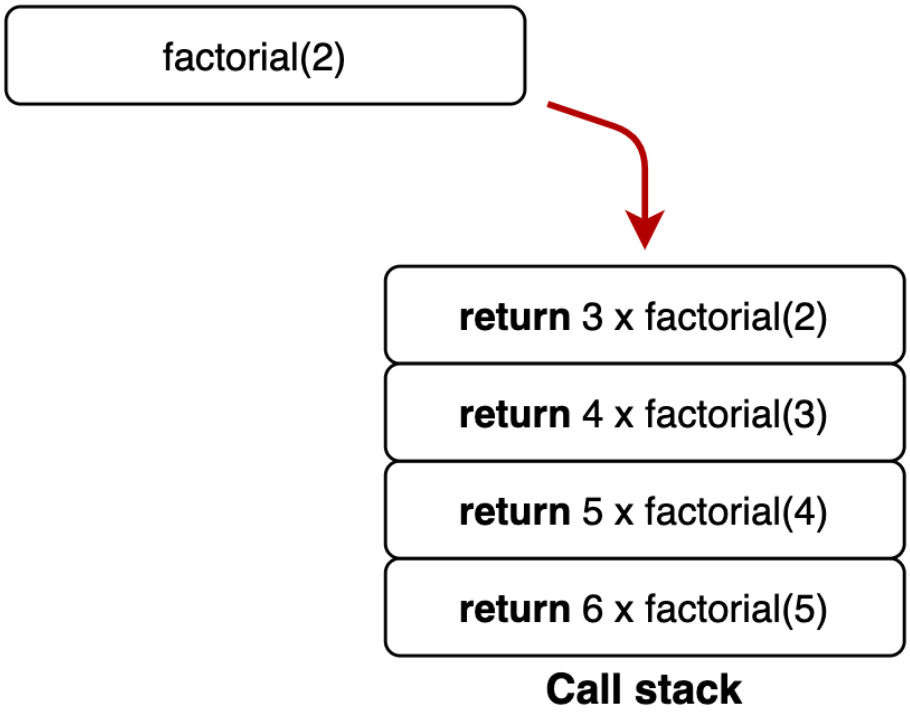
- Gli RdA non hanno tutti la stessa dimensione (a causa delle variabili locali). Il link dinamico permette di sapere dove inizia l'RdA precedente per poter fare "pop".
- Un RdA può contenere dati di dimensione variabile a run-time (es: array). Il puntatore RdA fornisce un punto di riferimento fisso per calcolare gli offset delle variabili locali (tranne quelle di dimensione variabile, gestite in modo speciale).

2.3.8 Esempio di Chiamata di Funzione (fattoriale)

Consideriamo la funzione fattoriale:

```
1 int fact (int n) {
2   if (n <= 1) return 1;
3   else return n * fact(n-1);
4 }
```

Quando chiamiamo 'fact(3)', la pila evolve così:



Ogni chiamata a 'fact' crea un nuovo record di attivazione con il proprio valore di 'n', l'indirizzo di ritorno e spazio per il risultato intermedio.

2.3.9 Gestione della Pila: Ingresso e Uscita

Ingresso in un blocco o procedura:

- Allocazione dello spazio per il record di attivazione sulla pila.
- Inizializzazione dei parametri e delle variabili locali (eventuale).
- Trasferimento del controllo al codice del blocco o della procedura.

Uscita da un blocco o procedura:

- Restituzione del valore di ritorno (se la procedura è una funzione).
- Ripristino dei registri della CPU (incluso il puntatore SP).
- Deallocazione dello spazio sulla pila (il puntatore SP viene riportato al valore precedente).
- Ripristino del valore del contatore di programma (PC) per continuare l'esecuzione dal punto corretto.

2.4 Allocazione Dinamica: Heap

2.4.1 Caratteristiche dell'Heap

L'**heap** è una regione di memoria dove blocchi di dimensione variabile possono essere allocati e deallocati in momenti arbitrari. È come un grande magazzino dove si prendono e si lasciano oggetti quando serve.

L'heap è necessario quando il linguaggio permette:

- Allocazione esplicita di memoria a run-time (tramite puntatori e strutture dati dinamiche come liste e alberi).

- Oggetti di dimensioni variabili (stringhe, insiemi, ecc.).
- Oggetti la cui vita non segue una logica LIFO (cioè, la cui durata non è legata all'ordine di chiamata delle funzioni).

2.4.2 Sfide della Gestione dell'Heap

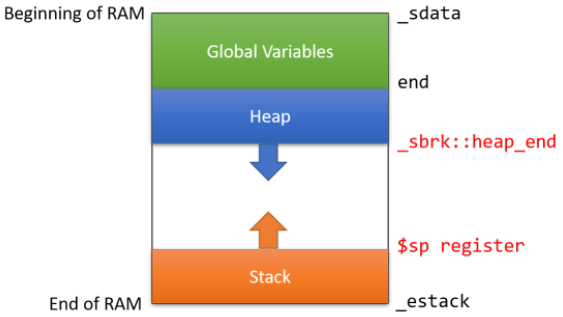
La gestione dell'heap non è banale e presenta diverse sfide:

- **Gestione efficiente dello spazio**: Evitare la frammentazione della memoria.
- **Velocità di accesso**: L'accesso alla memoria nell'heap può essere meno diretto rispetto alla pila.

2.4.3 Organizzazione dell'Heap

Heap con blocchi di dimensione fissa:

- L'heap è diviso in blocchi di dimensione predefinita (tipicamente piccola, qualche parola).
- Inizialmente, tutti i blocchi liberi sono collegati in una **lista libera**.
- L'allocazione richiede di trovare uno o più blocchi contigui liberi.
- La deallocazione restituisce i blocchi alla lista libera.



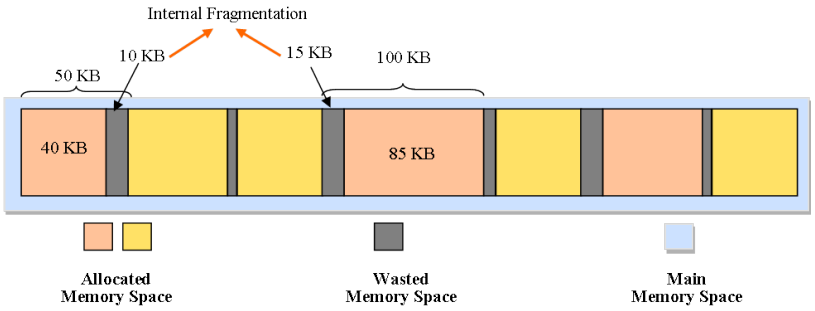
Heap con blocchi di dimensione variabile:

- Inizialmente, l'heap è un unico grande blocco.
- L'allocazione richiede di trovare un blocco libero di dimensione sufficiente.
- La deallocazione restituisce il blocco alla lista libera.

2.4.4 Frammentazione

La gestione dell'heap è complicata dalla **frammentazione**:

- **Frammentazione Interna**: Quando viene allocato un blocco più grande del necessario, lo spazio in eccesso all'interno del blocco viene sprecato.
- **Frammentazione Esterna**: Ci sono blocchi liberi sufficienti per soddisfare una richiesta di allocazione, ma non sono contigui, quindi non possono essere utilizzati.



2.4.5 Gestione della Lista Libera

La **lista libera** (LL) tiene traccia dei blocchi di memoria non allocati nell'heap. Le strategie per gestirla includono:

Con un'unica lista libera:

- Inizialmente, la lista contiene un solo blocco grande quanto l'heap.
- Ad ogni richiesta di allocazione, si cerca un blocco di dimensione adeguata:
  - **First Fit**: Si usa il primo blocco abbastanza grande trovato.
  - **Best Fit**: Si usa il blocco più piccolo che sia comunque abbastanza grande.
- Se il blocco scelto è molto più grande del necessario, può essere diviso, e la parte inutilizzata viene reinserita nella lista libera.
- Quando un blocco viene deallocato, viene restituito alla lista libera. Se ci sono blocchi liberi adiacenti, possono essere "fusi" per ridurre la frammentazione.

Con liste libere multiple:

- Si mantengono più liste libere, ognuna per blocchi di una certa dimensione.
- La ripartizione dei blocchi tra le liste può essere statica o dinamica.
- **Buddy System**: Un esempio di gestione dinamica. Ci sono k liste, dove la lista i contiene blocchi di dimensione 2^i. Se si richiede un blocco di dimensione 2^j ma non è disponibile, si divide un blocco di 2^{j+1}. Quando un blocco di 2^m viene deallocato, viene riunito al suo "buddy" (l'altro blocco di 2^m adiacente) se anche quello è libero.
- **Fibonacci System**: Simile al Buddy System, ma usa i numeri di Fibonacci invece delle potenze di 2 per determinare le dimensioni dei blocchi.

2.5 Implementazione delle Regole di Scope

2.5.1 Scope Statico: Catena Statica

Per implementare lo scope statico, ogni record di attivazione contiene un **link statico**. Questo link punta al record di attivazione dell'ambiente in cui la procedura è stata **definita** (non chiamata).

Quando una funzione 'foo' accede a una variabile non locale 'x', il sistema risale la catena statica per trovare il record di attivazione corretto dove 'x' è definita.

2.5.2 Determinare il Link Statico

Il chiamante è responsabile di determinare il link statico del chiamato. Il chiamante "conosce" l'annidamento statico dei blocchi. Quando la procedura 'Ch' chiama la procedura 'P', ci sono alcuni casi:

- Se la definizione di 'P' è immediatamente inclusa in 'Ch' (annidamento 0), il link statico di 'P' sarà lo stesso puntatore RdA di 'Ch'.
- Se la definizione di 'P' è in un blocco k passi "fuori" da 'Ch', allora il chiamante 'Ch' risale la propria catena statica di k passi e passa quel puntatore come link statico a 'P'.

2.5.3 Suddivisione dei Compiti (Compilatore e Run-time)

Compilatore:

- Associa un "livello di annidamento" k ad ogni chiamata di funzione.
- Associa ad ogni nome un indice h:
  - $h = 0$: nome locale.
  - $h \neq 0$: nome non locale definito h blocchi "sopra".

Sequenza di chiamata/prologo (Run-time):

- Risale la catena statica di k passi.
- Inizializza il puntatore di catena statica del chiamato.

Costo:

- Per ogni chiamata, k passi nella catena statica.
- Ad ogni accesso a una variabile non locale, h passi nella catena statica.

2.5.4 Ottimizzazione: Il Display

Per ridurre il costo di accesso alle variabili non locali, si può usare la tecnica del **display**. Il display è un array dove il i-esimo elemento contiene un puntatore all'RdA del sottoprogramma attivo al livello di annidamento i.

Se il sottoprogramma corrente è annidato a livello j, un oggetto in uno scope esterno di h livelli si trova accedendo al display all'indice $j - h$.

2.5.5 Gestione del Display

È il chiamato a gestire il display. Quando una procedura 'Ch' chiama 'P' (a livello di annidamento j), 'P' salva il valore corrente di 'Display[j]' nel proprio RdA e poi inserisce nel 'Display[j]' un puntatore al proprio RdA.

2.5.6 Catena Statica vs. Display

Mentre il display permette accessi più veloci alle variabili non locali, è più costoso da mantenere durante le chiamate di funzione. Nella pratica, l'annidamento profondo è raro, quindi la catena statica con ottimizzazioni (come tenere i puntatori più usati nei registri) è spesso sufficiente. Il display non è comunemente usato nelle implementazioni moderne.

2.5.7 Scope Dinamico: A-List e CRT

Nello scope dinamico, la visibilità di un nome dipende dalla sequenza di chiamate a run-time.

A-List (Association List): Le associazioni tra nomi e oggetti sono memorizzate negli RdA. Quando si cerca un nome, si risale la pila (la catena dinamica) fino a trovare la prima associazione per quel nome.

Tabella Centrale dei Riferimenti (CRT): Per evitare la scansione lineare dell'A-List, si può usare una tabella hash che contiene tutti i nomi distinti del programma. Ad ogni nome è associata una lista delle sue associazioni (la più recente è la prima). L'accesso è (quasi) costante.

2.5.8 Costi di A-List e CRT

- **A-List:** Tempo di accesso lineare alla profondità della lista.
- **CRT:** Accesso costante, ma gestione più complessa durante l'ingresso/uscita dai blocchi.

3 Controllo

3.1 Controllo del Flusso

Il controllo del flusso in un programma determina l'ordine in cui le istruzioni vengono eseguite. Le strutture fondamentali sono:

- **Sequenziale**: Le istruzioni vengono eseguite una dopo l'altra, nell'ordine in cui appaiono nel codice.
- **Condizionale**: Permette di eseguire blocchi di codice diversi in base al valore di un'espressione booleana (es: `if-then-else`).
- **Iterativo**: Permette di ripetere l'esecuzione di un blocco di codice più volte (es: `while`, `for`).
- **Ricorsione**: Una tecnica in cui una funzione chiama se stessa per risolvere un problema.

3.2 Espressioni

Un'espressione è una combinazione di operandi e operatori che produce un valore. Le notazioni principali sono:

- **Infissa**: L'operatore si trova tra gli operandi (es: `a + b`). È la notazione più comune nell'algebra tradizionale.
- **Prefissa (Polacca)**: L'operatore precede gli operandi (es: `+ a b`). Non richiede regole di precedenza o parentesi.
- **Postfissa (Polacca Inversa)**: L'operatore segue gli operandi (es: `a b +`). Facile da valutare con una pila.

3.2.1 Semantica delle Espressioni: Notazione Infissa

La semantica delle espressioni infisse è determinata da:

- **Precedenza degli operatori**: Definisce l'ordine in cui vengono eseguite le operazioni (es: la moltiplicazione ha precedenza sull'addizione).
  - Esempio di precedenza: `a + b * c` è interpretato come `a + (b * c)`.
  - Esempio di problemi in Pascal: `if A < B and C < D then ...` (errore se A, B, C, D non sono booleani).

- **Associatività degli operatori**: Definisce l'ordine di esecuzione per operatori con la stessa precedenza (es: l'associatività a sinistra di solito per somma e sottrazione).
  - Esempio di associatività: `a - b - c` è interpretato come `(a - b) - c`.
- **Uso delle parentesi**: Le parentesi possono essere usate per forzare un ordine di valutazione specifico (es: `(15 - 4) * 3`).

3.2.1.1 Precedenza degli Operatori in Diversi Linguaggi

La precedenza degli operatori varia tra i linguaggi di programmazione. Ecco alcuni esempi:

- **Fortran**: Operatori logici come `.NOT.`, `.AND.`, `.OR.`, aritmetici come `**`, `*` e `/`, `+` e `-`, e di confronto.
- **Pascal**: Operatori `@`, `not`, `^`, `*, /, div, mod, and`, `+, -, or`, e di confronto.
- **C**: Include operatori di post-incremento/decremento, pre-incremento/decremento, logici bit-a-bit, aritmetici, di confronto, logici, condizionali e di assegnamento.
- **Ada**: Simile a Pascal ma con operatori come `abs`, `not`, `**`, `*, /, mod, rem`, `+, -` (unari e binari), `&` (concatenazione), operatori di relazione, e logici.

3.2.2 Semantica delle Espressioni: Notazione Postfissa

La valutazione di un'espressione in notazione postfissa è semplice e utilizza una pila:

1. Leggi il prossimo simbolo dell'espressione.
2. Se il simbolo è un operando, mettilo sulla pila.
3. Se il simbolo è un operatore:
   (a) Applica l'operatore agli operandi immediatamente precedenti sulla pila.
   (b) Memorizza il risultato.
   (c) Elimina l'operatore e gli operandi dalla pila.
   (d) Metti il risultato sulla pila.
4. Ripeti finché l'espressione non è vuota.

È fondamentale conoscere l'arità di ogni operatore (il numero di operandi che richiede).

3.2.3 Semantica delle Espressioni: Notazione Prefissa

La notazione prefissa è più semplice della infissa:

- Non richiede regole di precedenza.
- Non richiede regole di associatività.
- Non richiede l'uso di parentesi.

La valutazione avviene usando una pila, ma è leggermente più complessa rispetto alla notazione postfissa perché è necessario tenere traccia del numero di operandi attesi per ciascun operatore.

3.2.4 Valutazione delle Espressioni e Alberi Sintattici

Internamente, le espressioni sono spesso rappresentate da alberi sintattici. Le diverse notazioni lineari (infissa, prefissa, postfissa) corrispondono a diverse modalità di visita dell'albero:

- Visita simmetrica → notazione infissa.
- Visita anticipata → notazione prefissa.
- Visita differita → notazione postfissa.

Il compilatore o l'interprete utilizzano l'albero per generare codice oggetto o valutare l'espressione. L'ordine di valutazione delle sottoespressioni è importante per:

- **Effetti collaterali**: Se una sottoespressione modifica lo stato del programma (es: chiamate a funzioni che modificano variabili globali), l'ordine di valutazione influisce sul risultato finale.
- **Aritmetica finita**: L'ordine delle operazioni può influenzare la precisione dei calcoli con numeri in virgola mobile.
- **Operandi non definiti**: La valutazione di un'espressione potrebbe portare a errori se un operando non è definito o ha un valore invalido.
- **Ottimizzazione**: Il compilatore può riordinare le operazioni per migliorare l'efficienza, a patto di preservare la semantica.

3.2.4.1 Effetti Collaterali

Considera l'espressione `f(b) * g(b)`. Se `f` modifica `b`, il risultato dipende dall'ordine di valutazione (da sinistra a destra o da destra a sinistra). Alcuni linguaggi vietano funzioni con effetti collaterali nelle espressioni. Java specifica un ordine di valutazione da sinistra a destra.

3.2.4.2 Valutazione Lazy vs. Eager

- **Valutazione Eager**: Tutti gli operandi di un'espressione vengono valutati prima di applicare l'operatore.
- **Valutazione Lazy (Corto-circuito)**: Gli operandi vengono valutati solo se necessario per determinare il risultato dell'espressione.

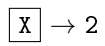
In C, l'espressione `0 ? b : b/0` usa la valutazione lazy: `b/0` non viene valutato perché la condizione è falsa. La valutazione eager in questo caso porterebbe a un errore di divisione per zero.

3.3 Comandi

Un comando è un'entità sintattica la cui valutazione può modificare lo stato del programma (avere un effetto collaterale) senza necessariamente restituire un valore. I comandi sono tipici del paradigma imperativo e non sono presenti nei paradigmi funzionale e logico. In alcuni linguaggi (es: C), l'assegnamento è un comando che restituisce un valore.

3.3.1 Variabili nei Linguaggi Imperativi

Nei linguaggi imperativi (Pascal, C, Ada, ...), una variabile è un contenitore di valori modificabile. Ha un nome e un valore che può essere cambiato tramite il comando di assegnamento.



3.3.2 Assegnamento

L'assegnamento è un'operazione fondamentale nei linguaggi imperativi.

- **l-value**: Un'espressione che denota una locazione di memoria (può comparire a sinistra dell'assegnamento).
- **r-value**: Un'espressione che denota un valore (può comparire a destra dell'assegnamento).

In generale: `l-value := r-value` (in Pascal) o `l-value = r-value` (in C).
La computazione nei linguaggi imperativi avviene principalmente attraverso gli effetti collaterali dell'assegnamento.

3.3.2.1 Modelli di Variabile

- **Linguaggi Funzionali (Lisp, ML, Haskell)**: Le variabili sono simili alle incognite matematiche, denotano un valore e non sono modificabili.
- **Linguaggi Logici**: Simile ai funzionali, ma con la possibilità di modificare il valore associato entro certi limiti.
- **Clu**: Modello a oggetti, anche detto modello a riferimento.
- **Java**: Variabile modificabile per i tipi primitivi, modello a riferimento per i tipi classe.

3.3.2.2 Modello a Riferimento

Simile ai puntatori, ma senza la manipolazione diretta delle locazioni di memoria. Se due variabili `X` e `Y` riferiscono allo stesso oggetto modificabile (es: un oggetto Java), le modifiche fatte tramite `X` si riflettono sull'oggetto riferito da `Y`.

3.3.2.3 Operatori di Assegnamento

- `X := X + 1` (Pascal) vs `X += 1` (C). L'operatore compatto evita il doppio accesso alla locazione di `X` e rende il codice più chiaro.
- Attenzione agli effetti collaterali negli indici degli array: `A[index(i)] := A[index(i)] + 1` potrebbe dare risultati inattesi se `index(i)` ha effetti collaterali.
- C offre diversi operatori di assegnamento, incremento/decremento prefissi e postfissi (`++i`, `i++`, `-i`, `i-`).
- L'incremento di un puntatore in C tiene conto della dimensione del tipo puntato (es: `p += 3` incrementa `p` di `3 * sizeof(*p)` byte).

3.3.2.4 Assegnamento Multiplo e Concatenato

- Assegnamento multiplo: `a, b, c := pippo(d, e, f)` (assegna i risultati di una funzione a più variabili).
- Assegnamento concatenato: `a = b = c = 0` (assegna lo stesso valore a più variabili).

3.3.3 Espressioni e Comandi nei Linguaggi Imperativi

- **Algol 68 (expression-oriented)**: Non c'è una distinzione netta tra espressioni e comandi. Ogni procedura restituisce un valore.
- **Pascal**: Distingue tra espressioni e comandi. Un comando non può comparire dove è richiesta un'espressione e viceversa.
- **C**: Le espressioni possono comparire dove ci si aspetta un comando. L'assegnamento (`=`) è permesso nelle espressioni, il che può portare a costrutti come `if (a = b) { ... }`, dove l'assegnamento viene eseguito e il valore assegnato viene valutato come condizione.

3.3.4 Ambiente e Memoria nei Linguaggi Imperativi

Nei linguaggi imperativi, lo stato del programma è determinato dall'**ambiente** e dalla **memoria**:

- **Ambiente**: Associa i nomi alle locazioni di memoria (Valori Denotabili).
- **Memoria**: Associa le locazioni di memoria ai valori memorizzati (Valori Esprimibili).

La semplice funzione `Stato: Nomi` $\rightarrow$ `Valori` non è sufficiente a causa dell'aliasing (due variabili che denotano la stessa locazione di memoria). I linguaggi funzionali usano solo l'ambiente.

3.4 Comandi per il Controllo della Sequenza

3.4.1 Comando Sequenziale e Blocchi

Il comando sequenziale (';') è il costrutto di base dei linguaggi imperativi. Ha senso solo in presenza di effetti collaterali. In alcuni linguaggi, il ';' è un terminatore di comando piuttosto che un separatore.

Un comando composto (blocco) permette di raggruppare più comandi in una singola unità sintattica. In Algol 68 e C, il valore di un comando composto è il valore dell'ultimo comando eseguito.

3.4.2 GOTO

L'uso del `goto` è stato ampiamente dibattuto negli anni '60 e '70. Considerato utile per:

- Uscire da un ciclo annidato.
- Ritornare da un sottoprogramma.
- Gestire eccezioni.

Tuttavia, è stato in gran parte considerato dannoso perché rende il codice difficile da leggere e da manutenere. I linguaggi moderni forniscono alternative strutturate per il controllo del flusso (cicli `while` e `for`, `if-then-else`, gestione delle eccezioni). Il `goto` non è presente in Java.

3.4.3 Programmazione Strutturata

La programmazione strutturata (anni '70) promuove un approccio di sviluppo del software che enfatizza:

- Design top-down o bottom-up.
- Codice modulare.
- Nomi di identificatori significativi.
- Uso esteso di commenti.

- Tipi di dati strutturati (array, record, ecc.).
- Comandi di controllo strutturati.

I comandi di controllo strutturati (come `for`, `if`, `while`, `case`) hanno un singolo punto di ingresso e un singolo punto di uscita, facilitando la comprensione del flusso di esecuzione.

3.4.4 Comando Condizionale (IF-THEN-ELSE)

Introdotto in Algol 60, il comando condizionale permette l'esecuzione selettiva di codice. Vari linguaggi hanno regole per risolvere l'ambiguità degli `if` annidati:

- **Pascal, Java**: L'`else` si associa al `then` non chiuso più vicino.
- **Algol 68, Fortran 77**: Usano una parola chiave di terminazione (`endif`).
- **Rami multipli espliciti**: `if-elseif-else` (presente in molti linguaggi).

La valutazione dell'espressione booleana di controllo può essere ottimizzata con la valutazione a corto-circuito.

3.4.5 Comando CASE (SWITCH)

Il comando `case` (o `switch`) è un discendente del `goto` calcolato e permette di selezionare un blocco di codice da eseguire in base al valore di un'espressione discreta.

- Diverse versioni nei vari linguaggi (Modula, Pascal, C, Ada, Fortran).
- Ottimizzazione nella valutazione: calcolo dell'indirizzo e salto diretto al ramo corrispondente (tramite tabelle di salto).
- Sintassi tipica di C, C++ e Java: `switch (espressione) { case valore1: ... break; default: ... }`.

3.5 Iterazione

L'iterazione e la ricorsione sono i due meccanismi fondamentali per ottenere la completezza di Turing.

- **Iterazione Indeterminata**: Il numero di ripetizioni non è noto a priori (cicli controllati logicamente: `while`, `repeat`).
- **Iterazione Determinata**: Il numero di ripetizioni è noto all'inizio del ciclo (cicli controllati numericamente: `for`).

3.5.1 Iterazione Indeterminata (WHILE, REPEAT)

Il ciclo `while` ripete un blocco di codice finché una condizione è vera. Il ciclo `repeat-until` (in Pascal) esegue il blocco di codice almeno una volta e continua finché una condizione diventa vera.

3.5.2 Iterazione Determinata (FOR)

Il ciclo `for` (con un indice, un valore iniziale, un valore finale e un passo) esegue un blocco di codice un numero specifico di volte. In un'iterazione determinata, non si possono modificare l'indice, l'inizio, la fine e il passo all'interno del ciclo.

La semantica del `for` dipende dal segno del passo:

- Passo positivo: Il ciclo continua finché l'indice è minore o uguale al valore finale.
- Passo negativo: Il ciclo continua finché l'indice è maggiore o uguale al valore finale.

I linguaggi differiscono su:

- Possibilità di modificare l'indice, l'inizio, la fine e il passo all'interno del ciclo.
- Dove avviene il controllo della condizione di terminazione.
- Quando viene incrementato l'indice.
- Possibilità di saltare dall'esterno all'interno del ciclo.

Nella maggior parte dei linguaggi moderni, non sono permessi cambiamenti all'interno del ciclo e i valori di inizio, fine e passo sono valutati una sola volta all'inizio.

3.5.3 Valore dell'Indice alla Fine del Ciclo

Il valore dell'indice al termine del ciclo varia a seconda del linguaggio:

- L'ultimo valore assegnato (il primo valore che eccede il limite).
- L'ultimo valore valido (richiede un test aggiuntivo).
- Indefinito (Pascal, Fortran IV).

In alcuni linguaggi (Algol W, Algol 68, Ada, Modula-3, C++), l'indice è una variabile locale del ciclo. In Algol 60 e Fortran 77, non si può saltare all'interno di un ciclo con il `goto`.

3.5.4 Loop in C

Il `for` in C è più flessibile e non è strettamente un'iterazione determinata: `for (inizializzazione; condizione; incremento) comando;`. Gli indici possono essere modificati nel corpo del ciclo e il programmatore deve gestire possibili overflow.

3.6 Ricorsione

La ricorsione è un modo alternativo all'iterazione per ottenere la completezza di Turing. Una funzione è ricorsiva se è definita in termini di se stessa.

3.6.1 Definizioni Induttive e Ricorsione

La ricorsione è strettamente legata alle definizioni induttive (es: la definizione dei numeri naturali di Peano). Una funzione ricorsiva definisce il valore per un caso base e poi definisce il valore per i casi successivi in termini di valori precedenti della stessa funzione.

È importante notare che non tutte le definizioni ricorsive sono "corrette" e possono portare a comportamenti inattesi o a non terminazione.

3.6.2 Ricorsione vs. Iterazione

Ogni programma ricorsivo può essere trasformato in uno iterativo equivalente e viceversa. La ricorsione è più naturale nei linguaggi funzionali e logici, mentre l'iterazione è più comune nei linguaggi imperativi. Implementazioni naive della ricorsione possono essere meno efficienti dell'iterazione, ma i compilatori moderni possono ottimizzare il codice ricorsivo (soprattutto la tail recursion).

3.6.3 Ricorsione in Coda (Tail Recursion)

Una chiamata a funzione `g` all'interno di `f` è una "chiamata in coda" se `f` restituisce direttamente il valore restituito da `g` senza ulteriori calcoli. Una funzione è tail recursive se tutte le sue chiamate ricorsive sono chiamate in coda. La tail recursion può essere ottimizzata dal compilatore per evitare l'allocazione di nuovi record di attivazione sulla pila, rendendola efficiente come un ciclo.

3.6.4 Esempio: Fattoriale Ricorsivo vs. Tail-Recursive

La versione standard del fattoriale non è tail-recursive, mentre una versione tail-recursive utilizza un parametro aggiuntivo per accumulare il risultato intermedio.

3.6.5 Esempio: Numeri di Fibonacci

La versione ricorsiva standard dei numeri di Fibonacci ha una complessità esponenziale, mentre una versione tail-recursive ha una complessità lineare e un uso di memoria costante.

4 Astrazione sul Controllo

4.1 Introduzione all'Astrazione

4.1.1 Astrazione nei Linguaggi di Programmazione

I linguaggi di programmazione (LP) forniscono al progettista strumenti per implementare modelli astratti. Gli LP stessi sono astrazioni del calcolatore sottostante.

Esempi di astrazione:

- **Alto livello di astrazione:**
  - `while` *vs* `goto`
  - Classi e metodi *vs* procedure
  - Tipi di dato (astratti) *vs* tipi non strutturati (`int`)
- **Astrazione sul controllo e sui dati.**

4.1.2 Astrazione sul Controllo

Permette di scrivere un pezzo di codice (P) senza conoscere il contesto in cui verrà usato (S).

Benefici:

- Fornisce astrazione funzionale al progetto.
- Ogni componente offre servizi al suo ambiente.
- L'astrazione descrive il comportamento esterno e nasconde i dettagli interni.
- Interazione limitata al comportamento esterno.
- Comunicazione attraverso:
  - Parametri
  - Ambiente globale (da evitare perché distrugge l'astrazione)

4.1.3 Astrazione sui Dati

Definizione: Un tipo di dato è definito dai suoi valori e dalle operazioni consentite su quei valori.

- Esempio: `integer` = `[-maxint..maxint]` e operazioni {`+,-,*,div,mod`}.
- Le operazioni sono l'unico modo per manipolare un dato di quel tipo.
- Esempio: non sono possibili operazioni di shift su valori `integer` a meno che non siano esplicitamente definite.

Astrazione sui dati:

- La rappresentazione (implementazione) dei dati.
- L'implementazione delle operazioni.
- Queste sono inaccessibili all'utente, protette da una capsula che le isola.

Linguaggi senza astrazione sui dati (nativa):

- FORTRAN, Pascal, C

4.2 Parametri

4.2.1 Terminologia

- **Dichiarazione/Definizione:**

```
int f (int n) { return n+1; }
```

- **Uso/Chiamata:**

```
x = f(y+3);
```

  - `n`: Parametro formale
  - `y+3`: Parametro attuale

4.2.2 Pragmatica

Flusso di informazioni tra chiamante e chiamato.

- `main` → `proc`:

```
x = f(y+3);
```

- `main` ← `proc`:

```
procedure Uno (var y: integer) ; begin y:=1 end;
```

- `main` ↔ `proc`:

```
procedure Succ (var y: integer) ; begin y:=y+1 end;
```

4.2.3 Ambiente non locale

Considerare l'accesso a variabili non locali all'interno di una funzione.

4.2.4 Passaggio dei Parametri

- Quale flusso informativo è realizzato?
- Sono ammesse espressioni come attuali?

```
int f (int x) { return x+1; }
int g (int &x) { return x+1; } // C++ reference
n = f(n+1);
n = g(n);
```

- Modifiche al formale si ripercuotono sull'attuale?

```
void foo (int x) { x = x+1; }
void fie (int &x) { x = x+1; } // C++ reference
int y;
y= 1;
foo (y); // Qui y vale 1
fie (y); // Qui y vale 2
```

4.3 Modalità di Passaggio dei Parametri

4.3.1 Passaggio per Valore

- Il valore dell'attuale è assegnato al formale, che si comporta come una variabile locale.
- Pragmatica: `main` → `proc`.
- L'attuale può essere qualsiasi espressione.
- Modifiche al formale non si riflettono sull'attuale.

```
void foo (int x) { x = x+1; }
int y = 1;
foo(y+1); // Qui y vale 1
```

- Il formale `x` è una variabile locale (sulla pila).
- Alla chiamata, l'attuale `y+1` è valutato e il valore è assegnato al formale `x`.
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante.
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila).
- Non è possibile trasmettere informazioni da `foo` al chiamante mediante il parametro.
- Può essere costoso per dati grandi: copia.
- Linguaggi che lo usano (di default o unicamente): Java, Scheme, Pascal (default), C.

4.3.2 Passaggio per Riferimento (o per Variabile)

- È passato un riferimento (indirizzo) all'attuale. I riferimenti al formale sono riferimenti all'attuale (aliasing).
- Pragmatica: `main` ← `proc`.
- L'attuale deve essere un L-valore ("una variabile").
- Modifiche al formale si riflettono sull'attuale.

```
1 void foo (int &x){ x = x+1;}
2 int y=1;
3 foo(y); // Qui y vale 2
```

- Viene passato un riferimento (indirizzo; puntatore).
- Il formale `x` è un alias di `y`.
- Al ritorno da `foo`, viene distrutto il (solo) legame tra `x` e l'indirizzo di `y`.
- Trasmissione bidirezionale tra chiamante e chiamato.
- Efficiente nel passaggio, ma indirezione nel corpo.
- Pascal (`var`); in C simulato passando un puntatore.

4.3.3 Passaggio per "Riferimento" in C

- Deve essere garantita una pragmatica `main` ↔ `proc`! Altrimenti non si potrebbe scrivere una `swap(x,y)`!!
- Ricorda: array e puntatori sono interoperabili. Passare un array è passare un riferimento.
- Esempio della funzione `swap`:

```
1 void swap (int *a, int *b) {
2     int t = *a; *a=*b; *b=t;}
3 int v1, v2;
4 swap(&v1, &v2) ;
```

- Viene passato (per valore!) un riferimento; non viene modificato il parametro formale, ma il dato a cui il parametro formale si riferisce.
- Stessa situazione in Java per i tipi "classe".

4.3.4 Passaggio per Costante (o Read-Only)

- Il passaggio per valore garantisce la pragmatica `main` → `proc` a spese dell'efficienza.
- Dati grandi sono copiati anche quando non sono modificati.
- Passaggio read-only (Modula-3; ANSI C: `const`).
- Nella procedura non è permessa la modifica del formale (controllo statico del compilatore: no alla sintassi di assegnamento; non passato per riferimento ad altre procedure).
- Implementazione a discrezione del compilatore:

- Parametri "piccoli" passati per valore.
- Parametri "grandi" passati per riferimento.

- In Java: `final`.

```
void foo (final int x){ //qui x non può essere modificato
}
```

4.3.5 Passaggio per Risultato

- Duale del passaggio per valore. Pragmatica: `main` ← `proc`.
- Esisteva in Algol-W.

```
void foo (int &x) {x = 8;} // Pseudo-codice
int y=1; // qui y vale 8
foo (y) ;
```

- Il formale `x` è una variabile locale (sulla pila).
- Al ritorno da `foo`, il valore di `x` è assegnato all'attuale `y`.
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante.
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila).
- Non è possibile trasmettere informazioni dal chiamante a `foo` mediante il parametro.
- Costoso per dati grandi: copia.
- Ada: `out`.

4.3.6 Passaggio per Valore/Risultato

- Insieme di valore + risultato. Pragmatica: `main` ↔ `proc`.
- Esisteva in Algol-W.

```
void foo (int x) // Pseudo-codice
{ x = x+1; }
int y= 8; // qui y vale 9
foo (y) ;
```

- Il formale `x` è a tutti gli effetti una variabile locale (sulla pila).
- Alla chiamata, il valore dell'attuale è assegnato al formale.
- Al ritorno, il valore del formale è assegnato all'attuale.
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante.
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila).
- Costoso per dati grandi: copia.
- Ada: `in out` (ma solo per dati piccoli; per dati grandi passa riferimento).

4.3.7 Value-result vs. Riferimento

```
void foo (int x, int y, int z) { // Sintassi di Java; ma questi modi non
↪  disponibili in Java!
    x = 1;
    y = 2;
    z = x;
}
int a = 3;
int b = 0;
foo(a, a, b);
```

	Value-result				Riferimento			
	a	y	x	b	a	y	x	b
Iniziale	3	3	3	0	3	3	3	0
Dopo x=1	3	3	1	0	1	1	1	0
Dopo y=2	3	2	1	0	2	2	1	0
Dopo z=x	3	2	1	1	1	1	1	1

4.3.8 Valore e Riferimento: Morale

Passaggio per valore:

- Semantica semplice: il corpo non ha necessità di conoscere come la procedura verrà chiamata (trasparenza referenziale).
- Implementazione abbastanza semplice.
- Potenzialmente costoso il passaggio; efficiente il riferimento al parametro formale.
- Necessita di altri meccanismi per comunicare `main` ↔ `proc`.

Passaggio per riferimento:

- Semantica complessa; aliasing.
- Implementazione semplice.
- Efficiente il passaggio; un po' più costoso il riferimento al parametro formale (un indiretto).

4.3.9 Valore, e non Riferimento

- I vantaggi del passaggio per valore, in particolare la trasparenza referenziale, suggeriscono linguaggi con passaggio solo per valore.
- Più meccanismi separati per ottenere il passaggio per riferimento:
  - Puntatori in C.
  - Variabili in modello a riferimento (tipi classe) in Java.

4.3.10 Passaggio per Nome

- **Regola di copia:** Una chiamata alla procedura P è la stessa cosa che eseguire il corpo di P dopo aver sostituito i parametri attuali al posto dei parametri formali.
- "Macro espansione", realizzata in modo semanticamente corretto.
- In Algol-W era il default.

```
int y;
void foo (/* name */ int x) {x= x + 1; } // Pseudo-codice
y=1;
foo(y) ; // y = y+1;
```

- In questo caso l'effetto è quello del passaggio per riferimento (che non esiste in Algol W).
- Caso più delicato:

```
int y;
void fie (/* name */ int x) { // Pseudo-codice
    int y;
    x=x+1; y= 0;
}
int y = 1;
fie(y) ;
```

- Conflitto (e "cattura") di variabili!
- Domanda: in quale ambiente avviene la valutazione dell'attuale dopo la sostituzione?
- Scoping statico: in quello del chiamante! Le due variabili `y` sono diverse! `fie` incrementa l'attuale (`y`) attraverso il formale `x` e crea e distrugge una nuova `y`.
- Viene passata una coppia: `<exp, amb>`.
  - `exp` è il parametro attuale (testo, non valutato).
  - `amb` è l'ambiente di valutazione (in scoping statico).
- Ogni volta che il formale è usato, `exp` è valutata in `amb`.

```
int y = 1;
void fie (int x )
{
    int y;
    x=x+1; y=0;
}
fie(y);
```

- Aliasing possibile tra formale e attuale; l'attuale deve valutare ad un L-val se il formale compare a sinistra di assegnamento.
- Pragmatica: `main` $\leftrightarrow$ `proc`.
- Costoso: passaggio di ambiente + valutazione ogni volta.
- Solo Algol 60 e W. Ma implementazione fondamentale...

4.3.11 Value-result vs. Nome

```
void fie (int x, int y) { // Sintassi di Java; ma questi modi non disponibili in
↪  Java!
    x = x+1;
    y = 1;
```

```
}
int i=1;
int[] A = new int[5];
A[1]=4;
fie (i, A[i]); // Esercizio: Riferimento e value-result hanno in questo caso lo
↪  stesso comportamento.
```

	Value-result		Nome	
	y	x	y	x
Iniziale	A[1] = 4	1	A[1] = 4	1
Dopo x=x+1	A[1] = 4	2	A[1] = 4	2
Dopo y=1	1	2	A[1] = 1	2

4.3.12 Passaggio per Nome: Implementazione

- Come passare la coppia `<exp,env>`?
  - Un puntatore al testo di `exp`.
  - Un puntatore di catena statica (sullo stack) al record di attivazione del blocco di chiamata.
- Perché chiude un'espressione, cioè una chiusura. Elimina le sue variabili libere legandole nell'ambiente del chiamante.
- Le chiusure servono a passare funzioni come argomenti ad altre procedure.
- Parametro per nome = funzione nascosta senza argomenti che valuta il parametro nell'ambiente del chiamante (un *thunk*, nel gergo Algol).

4.3.13 Modalità di Passaggio: Riepilogo

Nome	Attuale	Implementazione	Modifica?	Alias?
Var, Riferimento	Riferimento	Riferimento	Sì	Sì
Valore	Valore	Valore	No	No
Valore (ro)	Costante (l-o)	Valore o Riferimento	No	Possibile riferimento
Valore/Risultato	Valore	Valore	Sì	No
Nome	Chiusura	Chiusura	Sì	Sì

4.4 Funzioni di Ordine Superiore

4.4.1 Funzioni come Parametro di Procedure

- Il passaggio per nome è un caso particolare: si passa una funzione senza argomenti; corpo=exp.
- Esempio (ispirato a Pascal):

```
int x=4; int z=0;

int f (int y){ // dichiarazioni di x
    return x*y;
}

void g (int h(int n)){ // quando f sarà chiamata (tramite h) quale x (non
↪  locale) sarà usata?
    int x = 7;
    z = h(3) + x; // in scope statico, certo la x esterna
```

```
}                   // in scope dinamico, ha senso sia la x del blocco di
↪  chiamata che la x interna

int main() {
    int x = 5;
    g(f);
    return 0;
}
```

- **"funarg problems": downward funarg problem.**

4.4.2 Downward Funarg Problem

- Quando una procedura viene passata come parametro, si crea un riferimento tra un nome (parametro formale: `h`) e una procedura (parametro attuale: `f`).
- Problema: quale ambiente non locale si applica al momento dell'esecuzione di `f` (in quanto chiamata via `h`)?
  - Ambiente al momento della creazione del legame `h->f`?
    - **Deep binding** (scope statico).
  - Ambiente al momento della chiamata di `f` via `h`? Può aver senso con scope dinamico.
    - **Shallow binding** (scope dinamico).

4.4.3 Deep e Shallow Binding: Esempio

```
int x=4; int z=0;
int f (int y) { // quale x?
    return x*y;
}
void g ( int h(int n) ) {
    int x = 7;
    z = h(3) + x;
}

int main() {
    int x = 5;
    g(f);
    return 0;
}
```

Scope statico:

- Deep: `x` rossa (esterna).
- Shallow: `x` rossa (esterna).
- La regola di scope è sufficiente!

Scope dinamico:

- Deep: `x` azzurra (blocco di chiamata).
- Shallow: `x` nera (blocco interno di `g`).

4.4.4 Scoping Statico con Funzioni come Parametro (Deep Binding)

```
int x=4; int z =0;
int f (int y){
    return x*y;
}
void g (int h(int n)){
    int x = 7;
    z = h(3) + x;
}

int main() {
    int x = 5;
    g(f) ;
    return 0;
}
```

Come determinare il puntatore di CS alla chiamata di `h` (cioè `f`)? Alla chiamata `g(f)` è associato staticamente:

- un valore `k=0` di annidamento per `g`.
- un valore `k=0` di annidamento per `f`.

4.4.5 Chiusure

- Passare dinamicamente sia il legame col codice della funzione, che il suo ambiente non locale, cioè una chiusura `<code, env>`.
- Alla chiamata di una procedura passata per parametro:
  - Alloca (come sempre) il record di attivazione.
  - Prendi il puntatore di catena statica dalla chiusura.

4.4.6 Riassumendo: Parametri Funzioni (e per nome)

- Chiusure per mantenere puntatore all'ambiente statico del corpo di una funzione.
- Alla chiamata, il puntatore di catena statica determinato mediante la chiusura.
- Tutti i puntatori di catena statica puntano sempre indietro nella pila.
- I record di attivazione possono essere "saltati" per accedere a variabili non locali.
- De-allocazione dei record di attivazione secondo stretta politica a pila (LIFO: last-in-first-out).

4.4.7 Scope Dinamico: Implementazione

- **Shallow binding:** Non necessita di alcuna attenzione. Per accedere a `x`, risali la pila. Uso delle strutture dati solite (A-list, CRT).
- **Deep binding:** Usa necessariamente qualche forma di chiusura per "congelare" uno scope da riattivare più tardi.

4.4.8 Deep vs. Shallow Binding con Scope Statico

- **Scope dinamico:** È possibile sia deep binding (implementato con chiusure) che shallow binding (non necessita di implementazione ulteriore).
- **Scope statico:** Sempre usato deep binding (implementato con chiusure). A prima vista deep o shallow non fa differenza - è la regola di scope statico a stabilire quale non locale usare.
- Non è così: vi possono essere dinamicamente più istanze del blocco che dichiara il nome non-locale (accade in presenza di ricorsione).

4.4.9 Deep e Shallow Binding con Scope Statico: Esempio

```
int x = 1;

void foo(int f(), int y) {
    int z = f() + y; // deep binding?
                     // shallow binding? (non usato!)
}

int fie() {
    return x;
}

int g() {
    return 1;
}

int main() {
    foo(fie, 1);
    foo(g, 1);
    return 0;
}
```

4.4.10 Upward Funarg Problem

- Alcuni linguaggi (e.g., funzionali) permettono di restituire una funzione.
- Se la funzione ha variabili locali, queste devono sopravvivere indipendentemente dalla struttura a pila: hanno vita indefinitamente lunga.
- Esempio:

```
int main() {
    int (*F())() { // Funzione che restituisce un puntatore a funzione che
    ↪  non prende argomenti e restituisce int
        int x = 1;
        int g() { // Sintassi C/Java ma esempio impossibile in questi
        ↪  linguaggi!
            return x + 1;
        }
        return g;
    }

    int (*gg)() = F();
```

```
11      int z = gg();
12      return 0;
13  }
```

- La procedura `F` ritorna una chiusura. Modifica la macchina astratta per gestire una "chiamata a chiusura" come in `gg()`.

4.4.11 Morale: Funzioni come Risultato

- Uso delle chiusure, ma...
- I record di attivazione persistono indefinitamente (perdita proprietà della pila - LIFO).
- Come implementare la "pila" in questo caso:
  - Non deallocare esplicitamente.
  - Record di attivazione sullo heap.
  - Le catene statica e dinamica collegano i record.
  - Invoca il garbage collector quando necessario.

4.5 Eccezioni: "Uscita Strutturata"

- Spesso usate per:
  - Terminare parte di una computazione in condizioni inusuali.
  - Saltar fuori di un costrutto.
  - Passando dati attraverso il salto.
  - Ritornando il controllo al più recente punto di gestione.
  - Record di attivazione non più necessari sono deallocati.
  - Altre risorse, incluso spazio sullo heap, possono essere liberate.
- Due costrutti principali:
  - Dichiarazione del gestore dell'eccezione: posizione e codice di trattamento.
  - Comando/espressione per sollevare eccezione.

4.5.1 Esempio

- La funzione `average` calcola la media di un vettore; se il vettore è vuoto, solleva eccezione.

```
1  class EmptyExcp extends Throwable {int x=0;};
2
3  int average (int [] V) throws EmptyExcp {
4      if (V.length == 0) {
5          throw new EmptyExcp(); // solleva l'eccezione
6      } else {
7          int s=0;
8          for (int i=0; i<V.length; i++) {
9              s=s+V[i];
10         }
11         return s/V.length;
12     }
13 }
```

```
14
15  void foo() {
16      int[] w = {};
17      try { // intrappola e gestisce l'eccezione
18          average(w);
19      } catch (EmptyExcp e) { // gestore (handler) dell'eccezione
20          System.out.println("Array vuoto");
21      }
22  }
```

4.5.2 Sollevare un'Eccezione

- Il gestore è legato in modo statico al blocco di codice protetto.
- L'esecuzione del gestore rimpiazza la parte di blocco che doveva essere ancora eseguita.

```
1   class EmptyExcp extends Throwable {int x=0;};
2
3   int average(int[] V) throws EmptyExcp {
4       if (V.length==0) throw new EmptyExcp();
5       else {
6           int s=0;
7           for (int i=0; i<V.length; i++) s=s+V[i];
8           return s/V.length;
9       }
10  }
11
12  void bar() {
13      int[] W = {};
14      try{
15          average(W);
16      }
17      catch (EmptyExcp e) {
18          System.out.println("Array vuoto");
19      }
20  }
```

4.5.3 Propagare un'Eccezione

- Se l'eccezione non è gestita nella routine corrente:
  - La routine termina, l'eccezione è ri-sollevata al punto di chiamata.
  - Se l'eccezione non è gestita dal chiamante, l'eccezione è propagata lungo la catena.
  - Fino a quando si incontra un gestore o si raggiunge il toplevel, che fornisce un gestore di default.
- Vengono tolti i rispettivi frame dallo stack:
  - Per ogni frame che viene tolto, ripristina lo stato dei registri.
- Ogni routine ha un gestore "nascosto" che ripristina lo stato e propaga le eccezioni lungo la pila.

```java
class X extends Throwable {}

void f() throws X {
    throw new X();
}

void g (int sw) throws X {
    if (sw == 0) {
        f();
    } // try {f();} catch (X e) {System.out.println("in_g");}
}

public static void main(String[] args) {
    try {
        new Main().g(1);
    }
    catch (X e) {
        System.out.println("in main") ;
    }
}
```

Domanda: Quale dei due gestori viene eseguito? Osserva: quando l'argomento di g (qui g(1)) è frutto dell'esecuzione, non si conosce staticamente il gestore...

```java
class X extends Throwable {}

void f() throws X {
    throw new X();
}

void g (int sw) throws X {
    if (sw == 0) {
        f();
    }
    try {
        f();
    } catch (X e) {
        System.out.println("in_g");
    }
}

public static void main(String[] args) {
    try {
        new Main().g(1);
    }
    catch (X e) {
        System.out.println("in main");
    }
}
```

Esempio con Albero Binario:

```
class Zero extends Throwable {}

class Nodo {
    int chiave;
    Nodo FS;
    Nodo FD;
    public Nodo(int chiave, Nodo FS, Nodo FD) {
        this.chiave = chiave;
        this.FS = FS;
        this.FD = FD;
    }
}

int mul (Nodo alb) {
    if (alb == null) {
        return 1;
    }
    return alb.chiave * mul(alb.FS) * mul(alb.FD);
}

int mulAus (Nodo alb) throws Zero{
    if (alb == null) {
        return 1;
    }
    if (alb.chiave == 0) {
        throw new Zero();
    }
    return alb.chiave * mulAus(alb.FS) * mulAus(alb.FD);
}

int mulEff (Nodo alb) {
    try {
        return mulAus (alb);
    }
    catch (Zero e) {
        return 0;
    };
}
```

4.5.4 Implementare le Eccezioni

- **Semplice:**
  - All'inizio di un blocco protetto (`try`):
    - Metti su una pila (può essere quella di sistema) il gestore.
  - Quando un'eccezione è sollevata:
    - Togli il primo gestore sulla pila e guarda se è quello giusto.
    - In caso contrario, solleva di nuovo l'eccezione e ripeti.
  - Inefficiente nel caso – il più frequente – che non si verifichi eccezione:
    - Ogni `try` e routine devono mettere e togliere roba dalla pila.
- **Una soluzione migliore...**

4.5.5 Implementare le Eccezioni, II

- **Migliore:**
- Il compilatore genera una tabella dove, per ogni blocco protetto e per ogni routine, c'è una coppia: `<block_addr, handler_addr>`.
- La tabella è ordinata sul primo elemento (staticamente).
- Al sollevamento di un'eccezione:
  - Ricerca binaria nella tabella del Program Counter sul primo elemento.
  - Trasferimento del controllo all'indirizzo corrispondente del gestore.
- Se il gestore risolleva l'eccezione, ripeti.
- Attenzione: se il gestore è quello nascosto di una routine, la prossima ricerca nella tabella deve avvenire non col PC, ma con l'indirizzo di ritorno della routine.