Lezione 1 – Linguaggi per la descrizione di algoritmi

Programmazione

Modulo 2 - Programmazione e progettazione

Unità didattica 1 – Dalla programmazione non strutturata alla strutturata

Marco Anisetti

Università degli Studi di Milano - SSRI - CdL Online

Linguaggi per la descrizione di algoritmi(1)

Linguaggi informali: Linguaggio Naturale

Linguaggi semi-formali: Pseudo-codice, Flow chart (diagrammi di flusso)

- Specifiche iniziali, ancora intelleggibili all'essere umano
- Pseudo-codice: se $A > 0$ allora $A = A + 1$ altrimenti $A = 0$

Formale: Linguaggio di programmazione

Linguaggi semi-formali e formali

Linguaggi grafici: linguaggi pensati per un esecutore umano

- Flow chart
- UML (Unified Modeling Language)

Linguaggi di programmazione: adatti ad un esecutore automatico, specificatamente il calcolatore

- C, Pascal, Java, VisualBasic, ecc.

Linguaggio naturale

Il linguaggio naturale non si usa per la descrizione di programmi perchè è inadatto per una macchina:

- Ambiguo
- Vago
- Complicato

Nessuno ha ancora costruito una macchina che capisce l'italiano (o l'inglese) ma ci stanno provando con ottimi risultati usando ML

- NLP - ChatGPT

Pseudo-codice

Risulta più chiaro percepirlo quando si sa già scrivere del codice piuttosto che al contrario

Si rischia di scrivere in pseudo-naturale al posto che pseudo-codice

Keyword o insieme di keyword che hanno una semantica precisa (es goto)

Bisogna ricordarsi sempre dell’esecutore

Esempio MCD

Sequenza di istruzioni, decisione (if then), controllo di flusso (goto)

Istruzioni per catturare input ed associarlo a variabili (**read**)

Istruzioni per scrivere in output il risultato (**write**)

Esempio MCD

Sequenza di istruzioni, decisione (if then), controllo di flusso (goto)

Istruzioni per catturare input ed associarlo a variabili (**read**)

Istruzioni per scrivere in output il risultato (**write**)

1. **read (x)**
2. **read (y)** *(assumiamo x>y)*
3. **r <- *x* mod *y***
4. **if r = 0, then goto 8**
5. **x <- y**
6. **y <- r**
7. **goto 3**
8. **write (“MCD è”, *y*)**

Esempio

Problema: Dividere un numero naturale *x* per un numero naturale *y*, ricavando il resto *r*

Formalizzazione del problema tramite l'istruzione:

$r \leftarrow x \bmod y$

VINCOLO: mod non è eseguibile va scomposto in sottrazioni ripetute

Esempio

Problema: Dividere un numero naturale *x* per un numero naturale *y*, ricavando il resto *r*

Formalizzazione del problema tramite l'istruzione:

$r \leftarrow x \bmod y$

VINCOLO: mod non è eseguibile va scomposto in sottrazioni ripetute

1. **read(x)**
2. **read(y)**
3. **r ← x;**
4. **if r < y then goto 7**
5. **r ← r – y;**
6. **goto 4**
7. **write ("il resto è", r)**

Grammatica del linguaggio usato

< program >::=< row> | < program > <row>
<row> ::= <linenumber> < istr>
<linenumber> ::= <number> ‘‘.’’
…
< istr>::= <scrittura> | <lettura>| <assegnamento> | <decisione> | <salto>
<scrittura>::= ‘‘write’’ <parametro>
<parametro>::= ‘‘(’’ <stringa> ‘‘,’’ <nomevariabile> ‘‘)’’
….
<decisione>::= ‘‘if’’ <condizione> ‘‘then’’ <salto>
<salto>::= ‘‘goto’’ <number>
…

Linguaggio di programmazione

Ideato per tradurre algoritmi in un linguaggio comprensibile al calcolatore

- Aderenza dettagliata alla macchina (assembler)
- Livello più vicino al programmatore (alto livello)
  - Diversi paradigmi di programmazione, strumenti differenti

Fase di analisi: il linguaggio non va scelto ma conta potrebbe contare il paradigma

Paradigma di programmazione: il modello concettuale con il quale si concepisce un programma

Assembler

Il linguaggio più vicino al linguaggio macchina

Traduzione quasi immediata in linguaggio binario (quello veramente eseguito dalla macchina)

Tutti i linguaggi di alto livello vengono tradotti in una forma di assembler prima di essere tradotti in binari e quindi eseguibili

Il processo relativo si chiama assemblatore

- Per i linguaggi di alto livello si parla di compilatore o interprete

M.C.D in assembler

```
        LOAD R01, 101
        LOAD R02, 102
label1: DIV R01, R02
        MUL R01, R02
        LOAD R02, 101
        SUB R02, R01
        JZERO R02, fine
        LOAD R01, 102
        STORE R01, 101
        STORE R02, 102
        JUMP label1
fine:   LOAD R01, 102
        STORE R01, 103
```

Esempio tratto dal libro: Pighizzini, Ferrari, ”Dai fondamenti agli oggetti”

M.C.D in assembler

```
        LOAD R01, 101
        LOAD R02, 102
label1: DIV R01, R02
        MUL R01, R02
        LOAD R02, 101
        SUB R02, R01
        JZERO R02, fine
        LOAD R01, 102
        STORE R01, 101
        STORE R02, 102
        JUMP label1
fine:   LOAD R01, 102
        STORE R01, 103
```

Esempio tratto dal libro: Pighizzini, Ferrari, ”Dai fondamenti agli oggetti”

M.C.D in assembler

```
        LOAD R01, 101
        LOAD R02, 102
label1: DIV R01, R02
        MUL R01, R02
        LOAD R02, 101
        SUB R02, R01
        JZERO R02, fine
        LOAD R01, 102
        STORE R01, 101
        STORE R02, 102
        JUMP label1
fine:   LOAD R01, 102
        STORE R01, 103
```

1. read (x)
2. read (y) *(assumiamo x>y)*
3. r <- *x* mod *y*
4. if r = 0, then goto 8
5. x <- y
6. y <- r
7. goto 3
8. write (“MCD è”, *y)*

Esempio tratto dal libro: Pighizzini, Ferrari, ”Dai fondamenti agli oggetti”

M.C.D in assembler

```
        LOAD R01, 101
        LOAD R02, 102
label1: DIV R01, R02
        MUL R01, R02
        LOAD R02, 101
        SUB R02, R01
        JZERO R02, fine
        LOAD R01, 102
        STORE R01, 101
        STORE R02, 102
        JUMP label1
fine:   LOAD R01, 102
        STORE R01, 103
```

1. read (x)
2. read (y) *(assumiamo x>y)*
3. r <- *x* mod *y*
4. if r = 0, then goto 8
5. x <- y
6. y <- r
7. goto 3
8. write (“MCD è”, *y)*

Esempio tratto dal libro: Pighizzini, Ferrari, ”Dai fondamenti agli oggetti”

M.C.D in assembler

```
        LOAD R01, 101
        LOAD R02, 102
label1: DIV R01, R02
        MUL R01, R02
        LOAD R02, 101
        SUB R02, R01
        JZERO R02, fine
        LOAD R01, 102
        STORE R01, 101
        STORE R02, 102
        JUMP label1
fine:   LOAD R01, 102
        STORE R01, 103
```

1. read (x)
2. read (y) *(assumiamo x>y)*
3. r <- *x* mod *y*
4. if r = 0, then goto 8
5. x <- y
6. y <- r
7. goto 3
8. write (“MCD è”, *y)*

Esempio tratto dal libro: Pighizzini, Ferrari, ”Dai fondamenti agli oggetti”

Diagrammi di flusso (Flow chart)

I diagrammi di flusso sono un formalismo grafico di descrizione degli algoritmi.

I diversi tipi di istruzioni che caratterizzano questo formalismo sono rappresentati tramite **blocchi di varia forma**, connessi da frecce.

Orientato principalmente ad un esecutore umano

Ha il pregio di mettere ben in evidenza il control flow (la presenza di cicli, di salti, di biforcazioni, ecc..)

Descrizione tramite flow chart

Assegnamenti racchiusi in rettangoli

Decisioni racchiuse in rombi

Flusso definito da frecce

Ripetizione si nota un flusso che se percorso permette di ripetere istruzioni

- Contiene almeno una decisione che determina la fine della ripetizione

N.B. ci sarebbero anche trapezi per indicare istruzioni di input e output

MCD in flow chart

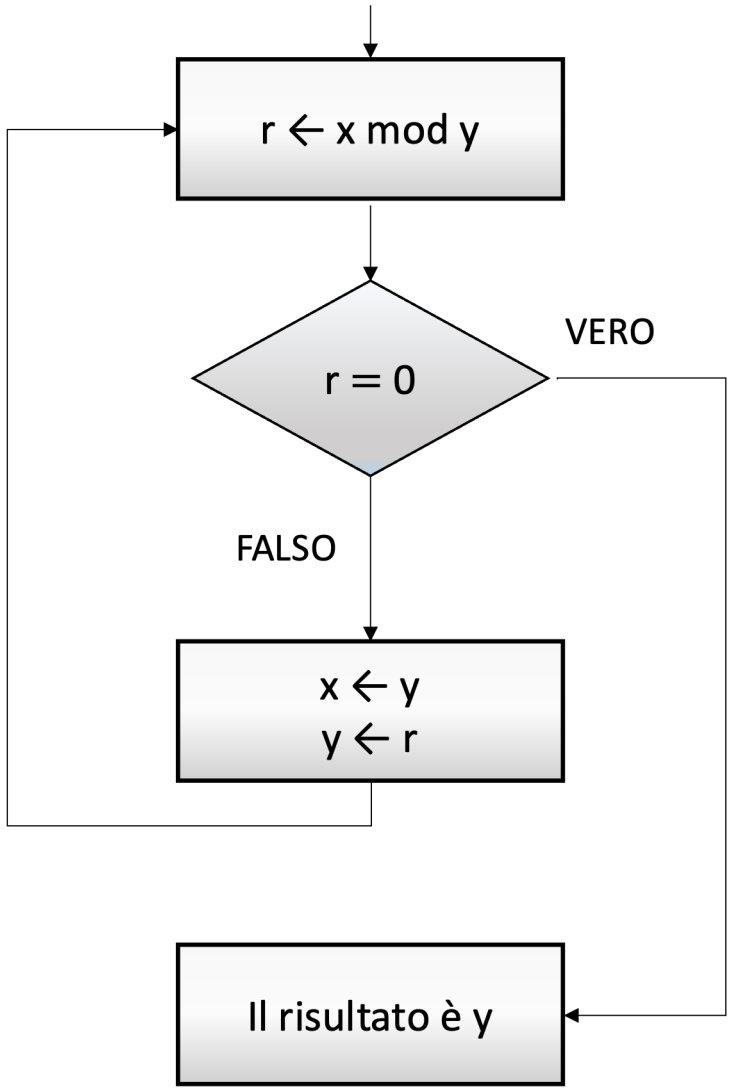
1. read (x)
2. read (y) *(assumiamo x>y)*
3. r <- *x* mod *y*
4. if (r = 0), then goto 8
5. x <- y
6. y <- r
7. goto 3
8. write (“Il risultato è”, *y)*

MCD in flow chart

1. read (x)
2. read (y) *(assumiamo x>y)*
3. **r <- *x* mod *y***
4. **if (r = 0), then goto 8**
5. **x <- y**
6. **y <- r**
7. **goto 3**
8. **write (“Il risultato è”, *y*)**

MCD in flow chart

1. read (x)
2. read (y) *(assumiamo x>y)*
3. **r <- *x* mod *y***
4. **if (r = 0), then goto 8**
5. **x <- y**
6. **y <- r**
7. **goto 3**
8. **write (“Il risultato è”, *y*)**



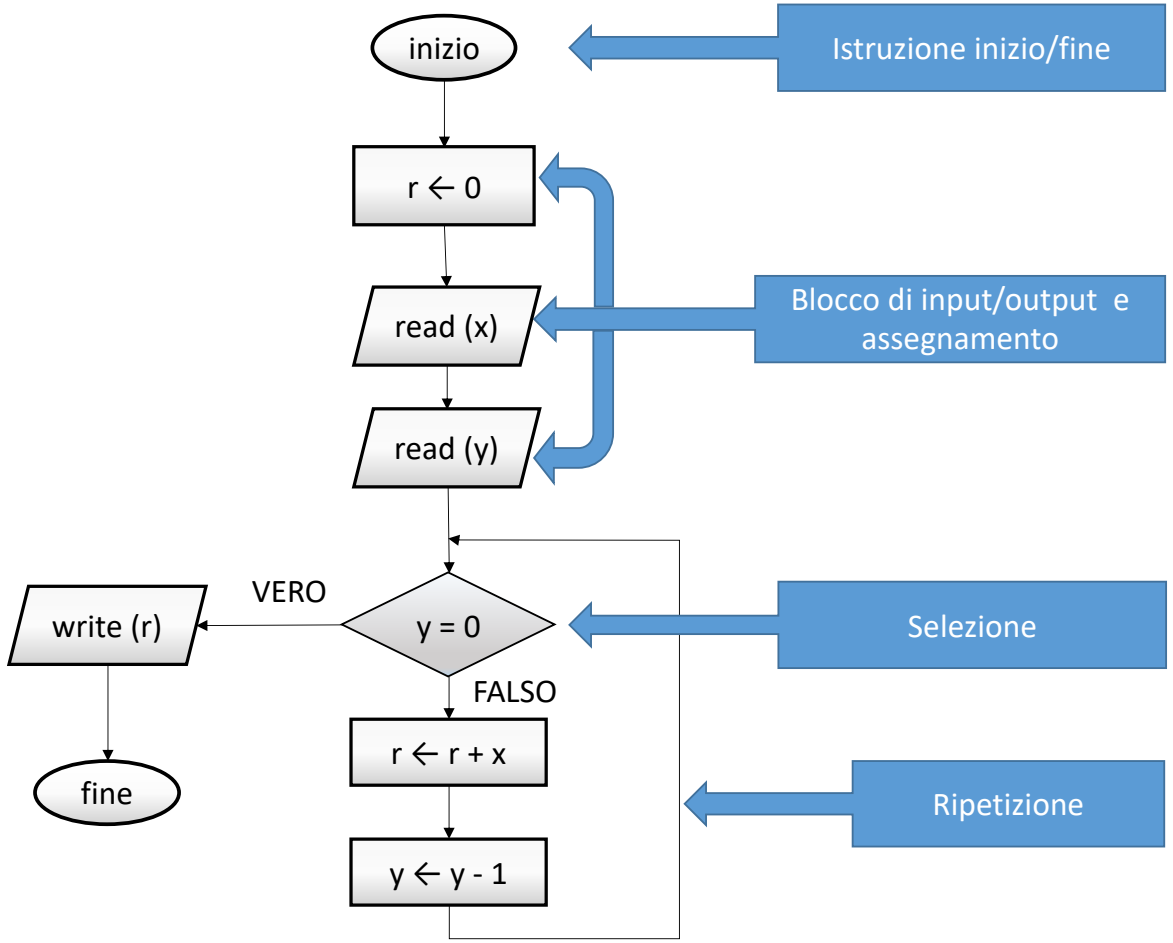
Moltiplicazione per somme

Immaginiamoci di dover scrivere il programma che esegue una moltiplicazione come somme successive

Quale idea risolutiva ci viene in mente?

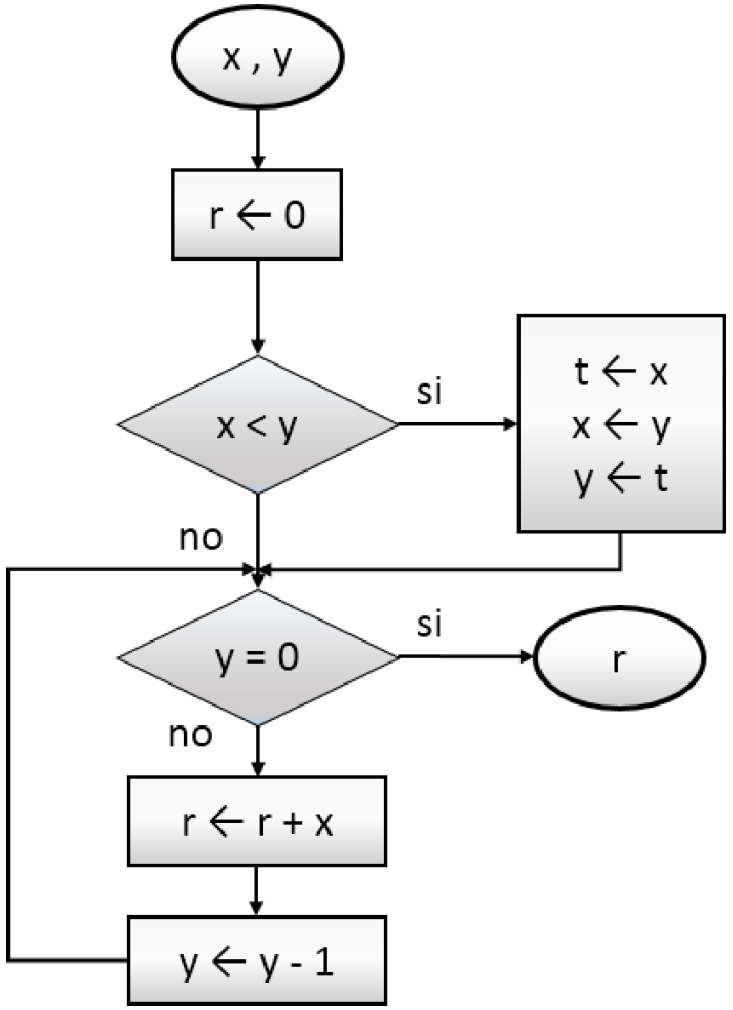
Scriviamolo in forma di diagramma di flusso

Flow chart: moltiplicazione per somme



Trovare delle formulazioni alternative più efficienti per la moltiplicazione per somme

Soluzione



Esercizio

Problema: Dividere un numero naturale *x* per un numero naturale *y*, ricavando il quoziente intero *q* e il resto *r*

Formalizzazione: tramite l'istruzione:

$(q, r) \leftarrow x \; div \; y$

VINCOLO: div non è eseguibile va scomposto in sottrazioni ripetute

Esercizio

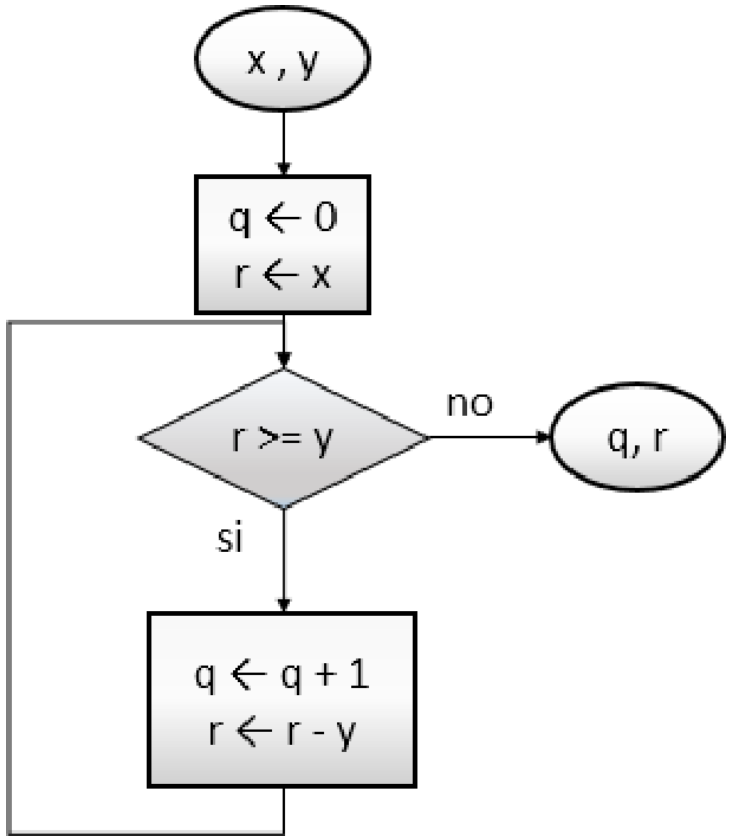
Problema: Dividere un numero naturale *x* per un numero naturale *y*, ricavando il quoziente intero *q* e il resto *r*

Formalizzazione: tramite l'istruzione:

$(q, r) \leftarrow x \; div \; y$

VINCOLO: div non è eseguibile va scomposto in sottrazioni ripetute

Soluzione



Considerazioni

Quando il problema è di natura matematica: formalizzazione matematica

Un programma descrive delle trasformazioni di stato delle proprie variabili

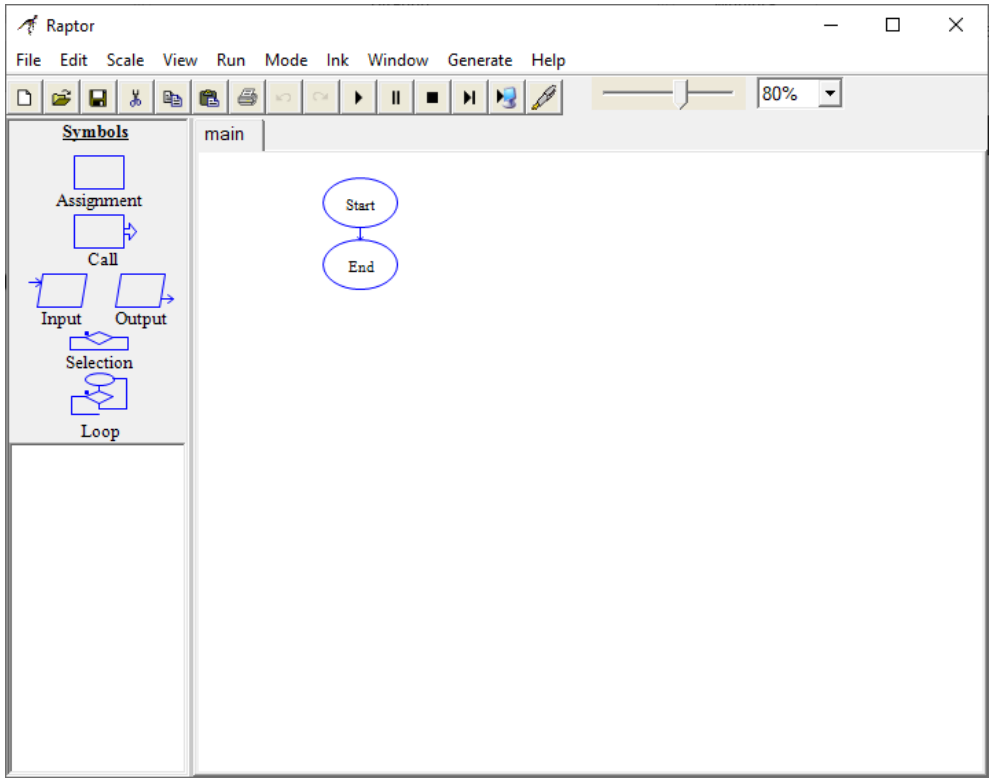
- Processo (programma in esecuzione)

Il formalismo dei flow chart

- Evidenzia graficamente dei costrutti
- Evidenzia il flusso del programma
- Generalmente contiene operazioni definite in un linguaggio di programmazione formalizzato

Uno strumento per programmare con i flow chart

https://raptor.martincarlisle.com/



NOTA: abbiamo dei blocchi da usare non abbiamo la massima libertà di disegnare ogni flusso possibile come sulla carta

Lezione 2 – Programmazione non strutturata

Programmazione

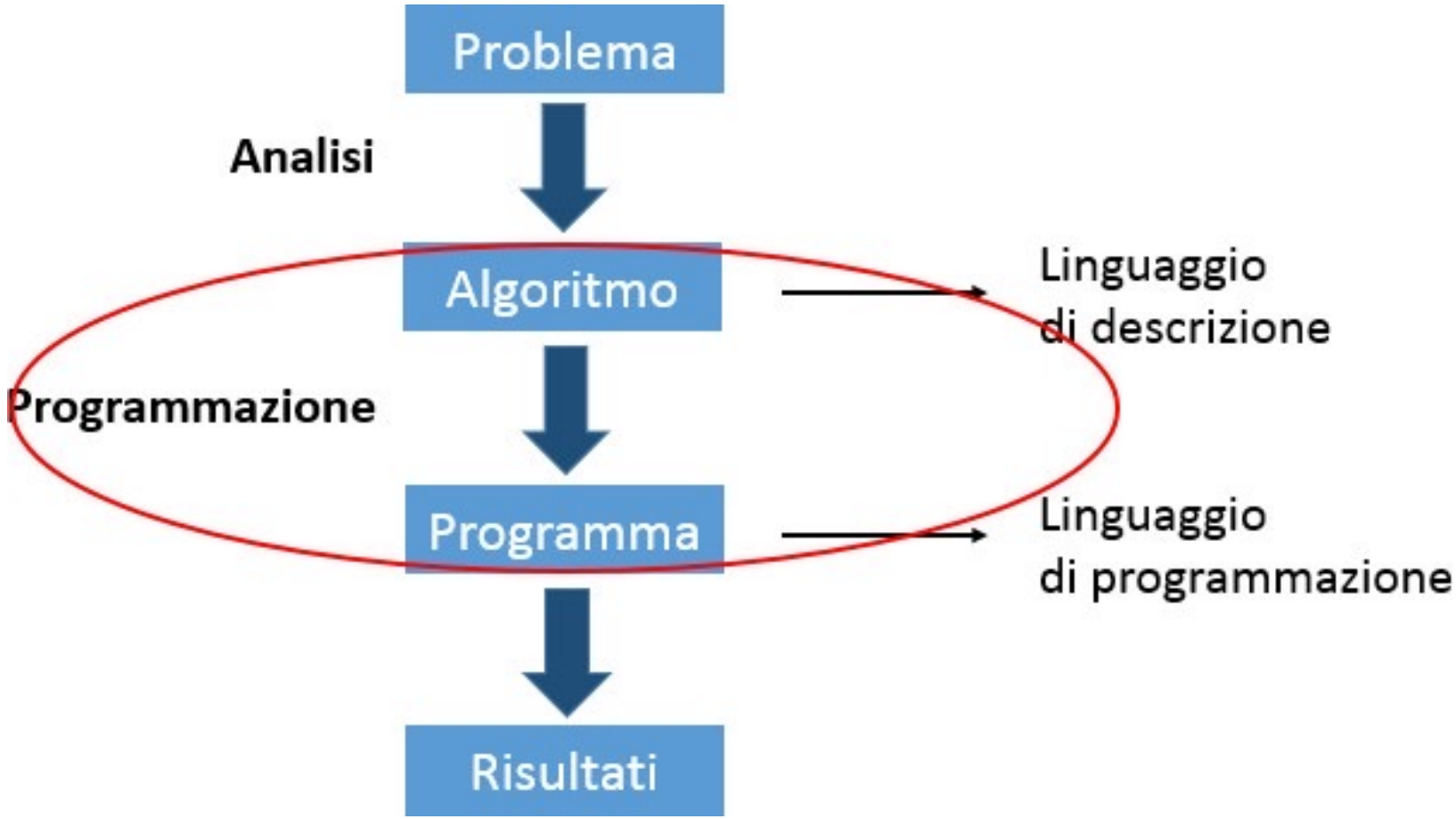
Modulo 2 - Programmazione e progettazione

Unità didattica 1 – Dalla programmazione non strutturata alla strutturata

Marco Anisetti

Università degli Studi di Milano - SSRI - CdL Online

Problema - algoritmo – programma



Modello di Macchina ad Accesso Casuale - RAM

Strumento per procedure sequenziali

- Di solito usato per valutare il costo computazionale

Ne vediamo una versione semplificata per semplicità

Programma: sequenza finita di istruzioni

Istruzioni semplice: read, write, assegnamento (:=), selezione (if), salto (goto)

Accesso alla memoria (Mem) simile ad accesso a celle dei nastri indicando il posto tra []

Codice assembler macchina astratta (RAM)

```
1:  Mem[0]:=0
2: read(Mem[1])
3: if Mem[1]≥ 0 then goto 5
4: goto  7
5:  Mem[3]:=Mem[0]-Mem[1]
6: if Mem[3]≥ 0 then goto  16
7:  write(Mem[1])
8:  read(Mem[2])
9: Mem[3]:=  Mem[2]-Mem[1]
10: if Mem[3]≥ 0 then goto  12
11: goto  14
12:  Mem[3]:=Mem[1]-Mem[2]
13: if Mem[3]≥ 0 then goto  8
14: Mem[1]:=Mem[2]  +Mem[0]
15: goto  3
16:  halt
```

Si riesce a capire cosa fa?

Codice assembler macchina astratta (RAM)

```
1: Mem[0]:=0
2: read(Mem[1])
3: if Mem[1]≥ 0 then goto 5
4: goto 7
5: Mem[3]:=Mem[0]-Mem[1]
6: if Mem[3]≥ 0 then goto 16
7: write(Mem[1])
8: read(Mem[2])
9: Mem[3]:= Mem[2]-Mem[1]
10: if Mem[3]≥ 0 then goto 12
11: goto 14
12: Mem[3]:=Mem[1]-Mem[2]
13: if Mem[3]≥ 0 then goto 8
14: Mem[1]:=Mem[2] +Mem[0]
15: goto 3
16: halt
```

Non è semplice capire cosa il codice fa guardandolo

Vediamo un costrutto condizionale if-then e degli assegnamenti

Il problema principale è seguire i **goto** incondizionati

Codice assembler macchina astratta (RAM)

```
1: Mem[0]:=0
2: read(Mem[1])
3: if Mem[1]≥ 0 then goto 5
4: goto 7
5: Mem[3]:=Mem[0]-Mem[1]
6: if Mem[3]≥ 0 then goto 16
7: write(Mem[1])
8: read(Mem[2])
9: Mem[3]:= Mem[2]-Mem[1]
10: if Mem[3]≥ 0 then goto 12
11: goto 14
12: Mem[3]:=Mem[1]-Mem[2]
13: if Mem[3]≥ 0 then goto 8
14: Mem[1]:=Mem[2] +Mem[0]
15: goto 3
16: halt
```

Non è semplice capire cosa il codice fa guardandolo

Vediamo un costrutto condizionale if-then e degli assegnamenti

Il problema principale è seguire i **goto** incondizionati

Il programma prende in ingresso una sequenza di numeri tipo 112223144 e ritorna la sequenza senza ripetizioni consecutive 12314

Programmazione non strutturata (1)

Non sempre posso avere una documentazione addizionale

Il programma deve avere una sua leggibilità indipendente da documentazione aggiuntiva

Al crescere delle dimensioni di un programma, diventa sempre più difficile.

Il problema prevalente è sui salti incondizionati

Un salto condizionato è più facile da comprendere per via della condizione che lo genera

Programmazione non strutturata: una programmazione che lascia la massima libertà sul flusso quindi usa i salti incondizionati

Programmazione non strutturata (2)

E’ stata per molti anni l’unica possibile

I salti non condizionati sono stati l’unico modo per avere un flusso differente dal flusso sequenziale

Alcuni salti incondizionati mappano strutture iterative

- esempi nei flowchart

Altri però non hanno nessuna corrispondenza diretta con tali strutture

Questo genere di salti non condizionati (non in relazione con condizioni di ciclo) sono quelli più critici da valutare

Flow chart e programmazione non strutturata

I flow chart possono essere usati anche a solo scopo di documentazione di un codice

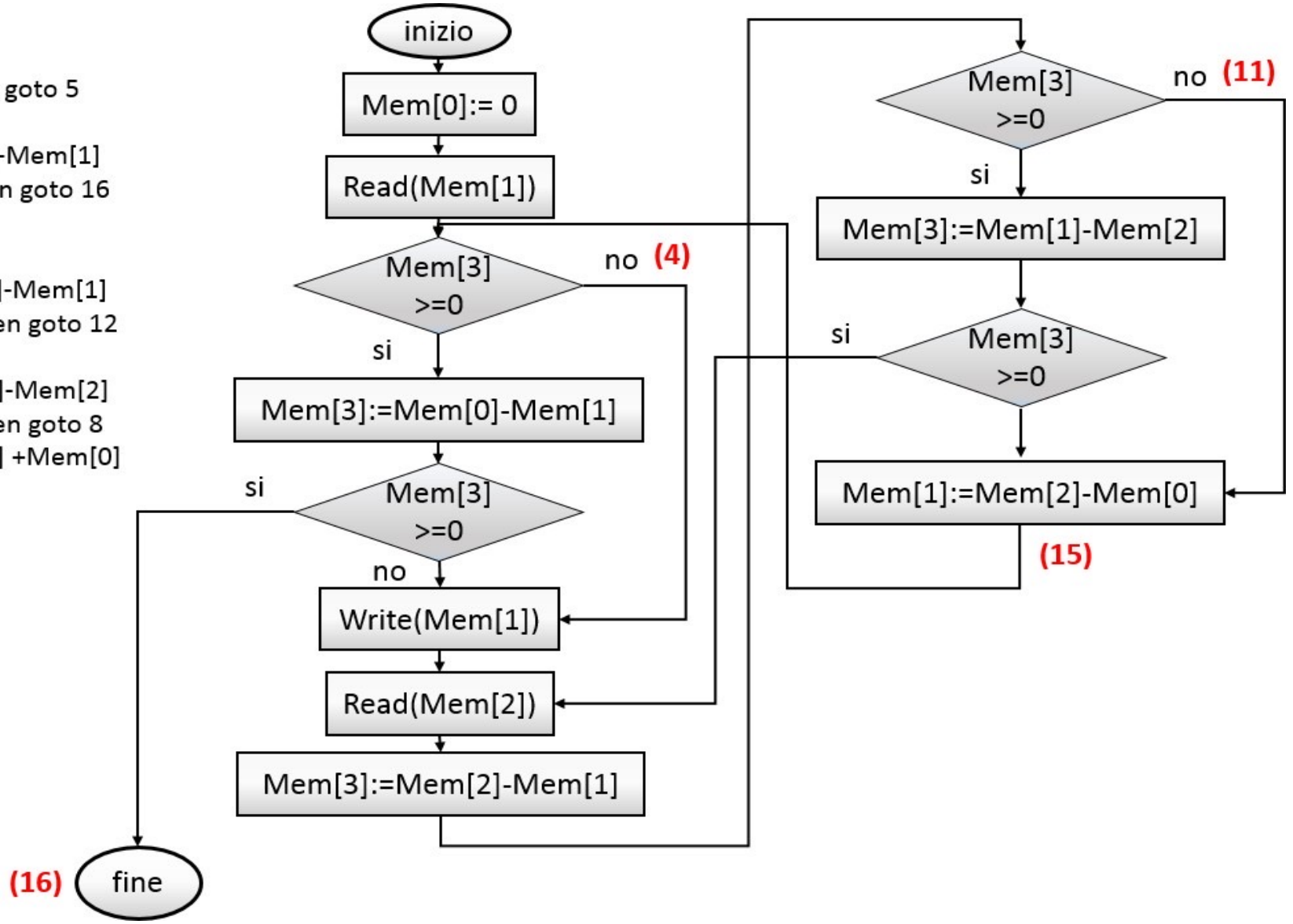
Come esercizio si provi a scrivere il flow chart del programma per macchina RAM visto in precedenza

Evidenziare i goto con delle apposite frecce

Individuate eventuali strutture iterative

Soluzione

```
1:  Mem[0]:=0
2:  read(Mem[1])
3:  if Mem[1>=0 then goto 5
4:  goto 7
5:  Mem[3]:=Mem[0]-Mem[1]
6:  if Mem[3]>= 0 then goto 16
7:  write(Mem[1])
8:  read(Mem[2])
9:  Mem[3]:= Mem[2]-Mem[1]
10: if Mem[3]>= 0 then goto 12
11: goto 14
12: Mem[3]:=Mem[1]-Mem[2]
13: if Mem[3]>= 0 then goto 8
14: Mem[1]:=Mem[2] +Mem[0]
15: goto 3
16: halt
```



Esempio: rimozione dei doppi

Possiamo scrivere lo stesso programma dell’esempio usando il formalismo usato nell’esempio dell’ MCD

Esempio: rimozione dei doppi

Possiamo scrivere lo stesso programma dell'esempio usando il formalismo usato nell'esempio dell' MCD

```
1: read(x);
2: if x=0 then goto 8;
3: write(x);
4: read(next);
5: if next=x then goto 4;
6: x <- next;
7: goto 2;
8: ;   (vuol dire che ha finito)
```

Esempio: rimozione dei doppi

Possiamo scrivere lo stesso programma dell'esempio usando il formalismo usato nell'esempio dell' MCD

```
1: read(x);
2: if x=0 then goto 8;
3: write(x);
4: read(next);
5: if next=x then goto 4;
6: x <- next;
7: goto 2;
8: ;   (vuol dire che ha finito)
```

E' più leggibile (si parla di codificata ad un livello più alto)

- a linea 7: abbiamo un goto non condizionato

Esercizio(1) programmare con goto

Dire cosa fa il seguente programma

```
1: read(a);
2: read(b);
3: if (a<b) then goto 7
4: c <- a;
5: a <- b;
6: b <- c;
7: ;
```

Esercizio(1) programmare con goto

Dire cosa fa il seguente programma

```
1: read(a);
2: read(b);
3: if (a<b) then goto 7
4: c <- a;
5: a <- b;
6: b <- c;
7: ;
```

Scambia i valori in a e b

Esercizio(2) programmare con goto

Il goto è ancora presente come costrutto di alcuni linguaggi di programmazione ad alto livello

Proviamo a scrivere il programma che calcola la divisione tra due numeri controllando prima che il divisore sia diverso da zero

Esercizio(2): Soluzione

Una soluzione possibile

1. read(dividendo)
2. read(divisore)
3. if divisore ≠ 0 then goto 6
4. write(“errore: divisione per zero”)
5. goto 8
6. res <- dividendo/divisore
7. write(ris)
8. ;

Programmazione ad alto livello

Fino ad ora abbiamo sempre implicitamente usato un modello di programmazione (paradigma) chiamato **imperativo**

- Parte della macro famiglia del **paradigma procedurale**
  - Descrivere come si risolve un problema

Imperativo: Programmazione intuitiva in cui si ordina all'esecutore di compiere delle azioni (richieste esplicite)

Gli elementi caratterizzanti sono:

- Il valore delle variabili cambia durante l'esecuzione (dinamicità)
- Il programma non è la stessa cosa della computazione (staticità del programma)

Lezione 3 – Variabili ed assegnamenti

Programmazione

Modulo 2 - Programmazione e progettazione

Unità didattica 1 – Dalla programmazione non strutturata alla strutturata

Marco Anisetti

Università degli Studi di Milano - SSRI - CdL Online

Variabili

Paragonabili alle incognite matematiche

Algoritmo: descrive una soluzione per una **classe di problemi**

- abbiamo bisogno di variabili (contenitori di valori) per esprimere le varie istanze di problemi

Si definiscono in termini di:

- **Nome:** identificatore (a cui fare riferimento)
- **Tipo:** insieme dei possibili valori che possono essere assunti (numero, carattere ecc.) ed operazioni che possono essere fatte
- **Valore:** valore attualmente assunto dalla variabile

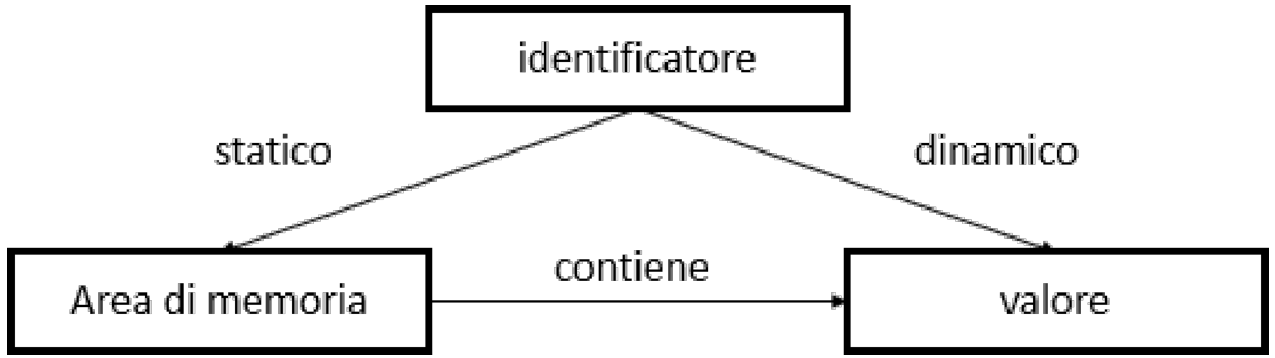
Identificatore - valore

Un identificatore è il nome di un oggetto

L'identificatore corrisponde ad una area di memoria che contiene il valore dell'oggetto

Il rapporto tra identificatore e valore è dinamico mentre quello tra identificatore e l'area di memoria è statico

L'identificatore di un variabile dovrebbe essere auto-documentante



Variabili e assegnamenti(1)

Un contenitore di valori che possono variare durante il tempo di esecuzione

Una locazione di memoria che ha una dimensione associata

Riferibile attraverso l'identificatore

Si può variarne il contenuto attraverso un assegnamento

[Istruzione di assegnamento]

variabile ← *espressione*

Esempio [BNF]

<assegnamento>::= <l-value> "<-" <r-value>

Variabili e assegnamenti(2)

[Semantica assegnamento]

(1) Viene calcolato il valore dell'espressione scritta a destra del simbolo ←

(1) Il risultato ottenuto è assegnato alla variabile a sinistra del simbolo ← (nel caso sovrascrivendo l'eventuale valore precedentemente contenuto)

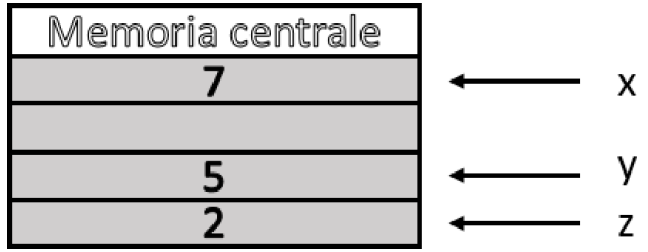
Molti linguaggi, utilizzano per l'assegnamento il simbolo =, usato comunemente per indicare l'uguaglianza

Esempio

x ← y + z

Valuta l'espressione *y* + *z*, recuperando i valori presenti nella celle di memoria relative a *y* e *z*

Assegna alla variabile *x* il risultato di tale somma (sovrascrivendo il valore)



Tipo

Il tipo di una variabile specifica:

- La classe di valori che questa può assumere
- L'insieme delle operazioni che possono essere effettuate su di essa.
- Quanto spazio occupa in memoria

Ad esempio una variabile *x* di tipo intero

- Può assumere come valori solo numeri interi
- Su di essa possono essere effettuate soltanto le operazioni consentite per i numeri interi
- La sua occupazione di memoria è solitamente uguale alla dimensione dei registri dell'elaboratore (es. 32, 64 bit)

Astrazioni: variabili

Il concetto di variabile è un' astrazione del concetto di **locazione di memoria**

La variabile è il **riferimento mnemonico** all'indirizzo della locazione di memoria

- Es. locazione 101 dell'esempio MCD in assembler

L'assegnamento di un valore a una variabile è un'astrazione dell'operazione **STORE** di un ipotetico linguaggio macchina

- Es. STORE R01, 101 dell'esempio MCD in assembler

Astrazioni: tipi

Tutte le variabili sono rappresentate nella memoria come sequenze di bit, tali sequenze possono essere interpretate diversamente in base ai tipi

La tipizzazione delle variabili permette di mantenere il controllo sulla correttezza nello sviluppo del programma

La nozione di tipo fornisce un' astrazione rispetto alla **rappresentazione effettiva dei dati**

Il programmatore può utilizzare variabili di tipi differenti, senza necessità di conoscerne l'effettiva rappresentazione

Dichiarazione delle variabili

Dichiarazione: Un enunciato di linguaggio di programmazione che definisce un **identificatore** e l'informazione a questo correlata

- Dichiarazione esplicita
- Dichiarazione implicita (al primo uso)

Di norma dichiaro quello che il programmatore ha la libertà di definire (es. variabili, funzioni ecc.)

- Molti linguaggi richiedono di dichiarare le variabili utilizzate nel programma in modo esplicito

Esempio [BNF]

<dichiarazione_variabile>::= <tipo> " " <identificatore>

Dichiarazione delle variabili

Alcuni linguaggi (ad esempio Pascal) richiedono che le variabili siano dichiarate tutte all'inizio del programma

Alcuni linguaggi richiedono che siano dichiarate prima del loro utilizzo (ad esempio Java)

Vantaggi nella dichiarazione:

- Accresce la leggibilità dei programmi
- Diminuisce la possibilità di errori
- Facilita la traduzione efficiente in linguaggio macchina (compilatori efficienti)

Modificatori di tipo Costante

Costante è intuitivamente un valore che non cambia per il lasso di tempo in cui viene usato

Costante è differente da letterale

- Un letterale è un carattere che viene inserito in una espressione
  - Esempio x + 5 x variabile 5 letterale

Una costante in programmazione è un **modificatore di tipo** che indica che una variabile non potrà mai cambiare valore

Condizioni - tipi - assegnamenti

Esiste genericamente un tipo associato ad ogni espressione

- dipende dai tipi delle variabili o letterali (ovvero numeri cifre ecc.) utilizzati nell'espressione

Quello che viene valutato in una decisione è una **condizione** ovvero una espressione

- Il tipo di queste espressioni è di solito un predicato che assume un valore di verità
  - Es. uguaglianza, disuguaglianza o una proposizione logica

Esempio BNF

Esempio [BNF]

<decisione>::= “if” <condizione> “then” …

< condizione> ::= <and-expr> | < condizione > “or” <and-expr>

<and-expr> ::= <eq-expr> | <and-expr> “and” <eq-expr>

<eq-expr> ::= <rel-expr> | <eq-expr> “ = ” <rel-expr>

| <eq-expr> “ ≠ ” <rel-expr>

<rel-expr> ::= <expr > |<rel-expr> “<” <expr> | …

Condizioni

Espressioni che assumono un valore di verità

- Il valore di verità è associato ad un tipo particolare detto booleano
- Tale tipo può prendere valore nell'insieme *{vero, falso}*

Quando l'espressione generica è usata in una condizione essa viene valutata ed il ritorno viene interpretato dall'istruzione condizionale