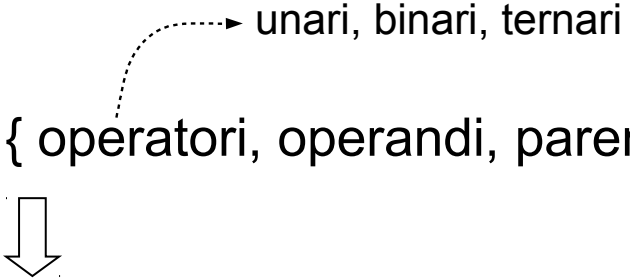


Espressioni

- Espressione = meccanismo fondamentale per esprimere computazioni in un LP
→ importante comprenderne la semantica!
- Valutazione automatica di expr aritmetiche = obiettivo primario dei primi LP (Fortran)
- Expr aritm. in LP composta da { operatori, operandi, parentesi, call di funzioni }

specifica una computazione aritmetica
- Necessarie 2 azioni < recupero del valore degli operandi (**valutazione**)
applicazione dell'operazione su tali valori (**esecuzione**)

Espressioni Aritmetiche: Scelte Progettuali

1. Quali regole di **precedenza** degli operatori?
2. Quali regole di **associatività** degli operatori?
3. Quale **ordine** di valutazione degli operandi?
4. Quali restrizioni sui possibili **effetti collaterali** della valutazione degli operandi?
5. Permessi **overloading** degli operatori?
6. Quale grado di **mixing**? (operandi di diverso tipo)

Ordine di Valutazione degli Operatori: **Precedenza**

$$\begin{array}{l} \text{3} \quad \text{4} \quad \text{5} \\ \boxed{A + B * C} \end{array} \begin{cases} \rightarrow (A + B) * C = \text{35} \\ \leftarrow A + (B * C) = \text{23} \end{cases}$$

↓

Più in generale: operatori classificati in una gerarchia di priorità di valutazione

- **Regole di precedenza:**

FORTRAN	C++	Ada
**	++, --	**, abs
*, /	+, - (unari)	*, /, mod
+, -	*, /, %	+, - (unari)
	+, - (binari)	+, - (binari)

Ordine di Valutazione degli Operatori (ii): **Associatività**

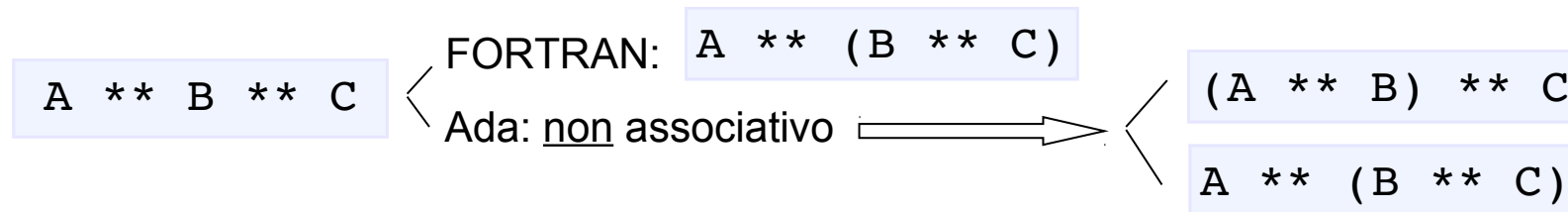
$A - B + C - D \Rightarrow$ se +, - con stessa precedenza: quale ordine di valutazione?



Regole di associatività $\left\{ \begin{array}{l} \text{sinistra (tipicamente)} \\ \text{destra (esponenziazione)} \end{array} \right.$

Esempi:

$$A - B + C \equiv (A - B) + C$$



APL: $\left\{ \begin{array}{l} \text{precedenza: uguale per tutti} \\ \text{associatività: destra} \end{array} \right. \Rightarrow A \times B + C \equiv A \times (B + C) = \mathbf{27}$

Ordine di Valutazione degli Operatori (iii): **Associatività**

- \exists operatori aritmetici matematicamente associativi \rightarrow valore indipendente dalle regole di associazione

$A + B + C$



Compilatore: può riordinare le valutazioni per ragioni di ottimizzazione

Però: floating-point \approx approssimazione del concetto di numero reale



Operazioni sui floating-point non necessariamente associative!
(solo quando i risultati intermedi sono rappresentabili correttamente nel tipo)

Anche: Interi $A + B + C + D$ in cui $\begin{cases} A, C \gg 0 \\ B, D \ll 0 \end{cases}$



$(A + B) + (C + D)$ ok

$(A + C) + (B + D)$ overflow

Ordine di Valutazione degli Operatori (iv)

- Parentesi: artificio per alterare le regole di precedenza/associatività

`(A + B) * C`

→ teoricamente: potrebbero sostituire le regole, ma scomodo!

- Espressioni condizionali: operatore ternario **?:** (C, C++, Java)

```
if num = 0 then
    media := 0
else
    media := somma/num;
```

```
media = (num == 0 ? 0 : somma/num);
```

Ordine di Valutazione degli Operandi

- Operando $\left\{ \begin{array}{l} \text{costante} \\ \text{variabile} \end{array} \right\}$ recupero da memoria
- espressione \rightarrow valutazione di tutti gli operandi prima che possa essere usata

- Ordine: rilevante quando \exists effetti collaterali nella valutazione degli operandi

$A + \text{FUN}(A) \Rightarrow \text{FUN} \left\{ \begin{array}{l} \text{senza} \\ \text{con} \end{array} \right\} \text{effetto collaterale} \Rightarrow \text{ordine} \left\{ \begin{array}{l} \text{irrilevante} \\ \text{rilevante} \end{array} \right.$

Hp: $\text{FUN}(A) \left\{ \begin{array}{l} \text{assegna } A \leftarrow 20 \\ \text{restituisce } A/2 \text{ prima di} \end{array} \right.$

$A := 10;$
 $B := A + \text{FUN}(A)$ $\left\{ \begin{array}{l} A, \text{FUN}(A) \rightarrow B=15 \\ \text{FUN}(A), A \rightarrow B=25 \end{array} \right.$

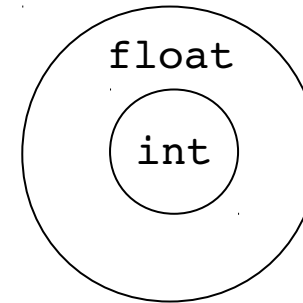
- Scelte dei LP $\left\{ \begin{array}{l} \text{Pascal, Ada: ordine di valutazione lasciato all'implementazione} \\ \text{Java: da sinistra a destra} \end{array} \right.$

Overloading degli Operatori

- Stesso simbolo con diversi significati Java: + $\begin{cases} \text{somma} \\ \text{concatenazione di stringhe} \end{cases}$
 - Accettabile purchè non comprometta $\begin{cases} \text{leggibilità} \\ \text{affidabilità} \end{cases}$
 - C: & $\begin{cases} \text{and bit a bit} \\ \text{indirizzo di} \end{cases}$ `if (x & y)` se omesso, reinterpretazione del predicato di selezione
 - - $\begin{cases} \text{unario} \\ \text{binario} \end{cases}$ `x = y - z` se omesso, reinterpretazione della espressione di assegnamento
 - Simboli distinti → più $\begin{cases} \text{leggibili} \\ \text{convenienti} \end{cases}$
 - C $\begin{cases} \text{reale} \\ \text{media} = \text{somma}/\text{cont}; \end{cases}$
 - Pascal $\begin{cases} \text{intero} \\ \text{media} := \text{somma}/\text{cont}; \end{cases}$ \Rightarrow troncamento $\begin{cases} / \rightarrow \text{reali} \\ \text{div} \rightarrow \text{interi} \end{cases}$
 - C++: overloading definito dall'utente `A * B + C * D` (matrici)
- \Updownarrow
`addm(mulm(A,B), mulm(C,D))`

Conversioni di Tipo

- Conversione
 - allargante `int → float`
 - restringente `float → int`



- Conversione
 - implicita (**coercizione**)
 - esplicita (**cast**)

↓
Stile

- Ada: funzionale
- C: operatore prefisso

```
media := FLOAT(somma)/FLOAT(cont)
```

```
(long int)distanza
```

Conversione di Tipo Implicita

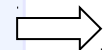
- Coercizione: necessaria quando l'operatore ammette operandi di diverso tipo



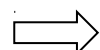
espressione **mista**

- Se $\text{binding}(var, tipo)$ statico \rightarrow compilatore genera codice per la coercizione
 - Vantaggio: flessibilità
 - Svantaggio: indebolimento del compilatore

```
void main()  
{  
  int a,b,c;  
  float d;  
  ...  
  a = b * d;  
  ...  
}
```



errore non individuato



coercizione

$\left\{ \begin{array}{l} b \rightarrow \text{float} \\ \text{expr} \rightarrow \text{int} \end{array} \right.$

- Esempio di coercizione spinta: PL/I

intero
↑ ↑
i + j * s → stringa

⇒ scanning $\left\{ \begin{array}{l} \text{intero} \\ \text{'.' reale} \rightarrow \text{coercizione di } i, j \text{ in reale} \end{array} \right.$

Espressioni Booleane

- **Espressioni relazionali:** $op_1 \text{ relop } op_2 \Rightarrow \text{risultato: booleano (C: intero)}$

→ priorità più bassa degli aritmetici: $a + 1 > b + 2$

- **Espressioni booleane:** coinvolgono $\left\{ \begin{array}{l} \text{expr relazionali} \\ \text{variabili booleane} \\ \text{costanti booleane} \\ \text{operatori booleani} \end{array} \right.$ $\left\{ \begin{array}{l} \text{and} \\ \text{or} \\ \text{not} \end{array} \right.$ espressioni logiche

- Sono definite regole di $\left\{ \begin{array}{l} \text{precedenza} \\ \text{associatività} \end{array} \right.$

- C: \nexists boolean $\underbrace{a > b > c}_{0 \text{ oppure } 1} \Rightarrow b \text{ non confrontato con } c$

Espressioni Booleane (ii)

- **Valutazione in corto circuito**: risultato determinato mediante valutazione parziale della expr

`(12 * x) * (y/12 - 1)` $x = 0 \rightarrow \text{risultato} = 0$

`(x > 0) or (y < 0)` $x > 0 \rightarrow \text{risultato} = \text{true}$

$a \text{ or } b \equiv (a ? \text{true} : b)$

`(x > 0) and (y < 0)` $x \leq 0 \rightarrow \text{risultato} = \text{false}$

$a \text{ and } b \equiv (a ? b : \text{false})$

Esempio: Ricerca di un elemento in una tabella sulla base di una chiave (Pascal)

	id	val
1		
2		
...		

```
i := 1;  
while (i <= TOT) and (tab[i].id <> chiave) do  
    i := i + 1;
```



valutazione completa \rightarrow se \nexists chiave $\rightarrow \text{tab}[\text{TOT}+1] \rightarrow \text{out of range!}$

Espressioni Booleane (iii)

- Pb del corto circuito quando \exists effetti collaterali nella valutazione della expr

`(a > b) || (b++ / 3)` \Rightarrow b incrementato solo quando $a \leq b$

- Ada: corto circuito non implicito $\left\{ \begin{array}{l} \text{AND} \rightarrow \text{and then} \\ \text{OR} \rightarrow \text{or else} \end{array} \right.$

```
i := 1;
while (i <= TOT) and then (tab(i).id /= chiave)
  loop
    i := i + 1;
  end loop;
```

- C, C++, Java $\left\{ \begin{array}{l} \&\&, || \rightarrow \text{corto circuito} \\ \&, | \rightarrow \text{no!} \end{array} \right.$

- Conclusione: meglio avere entrambi gli operatori (come Ada)