Controllo di Sequenza: Espressioni e Comandi

Controllo di Sequenza

Le strutture che definiscono il controllo della sequenzializzazione delle operazioni (sia primitive sia definite dall'utente) possono essere suddivise in tre gruppi:

- **espressioni**
- **comandi**
- **sottoprogrammi**

In aggiunta vi sono delle strutture che presiedono il controllo della trasmissione dei dati tra sottoprogrammi (**controllo dati**) che discuteremo più avanti.

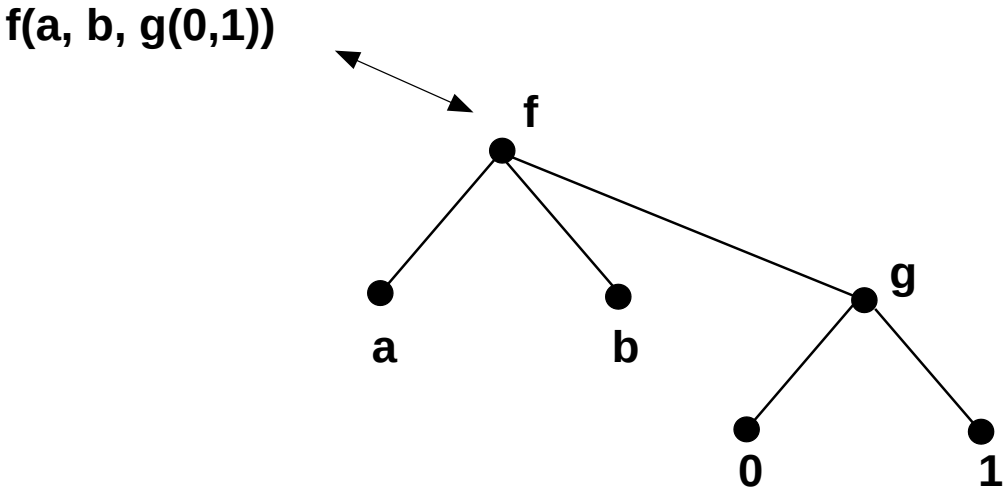
Controllo di Sequenza per le Espressioni

espressione $\Rightarrow$ per valutazione: sintassi + semantica

rappresentazione dell'espressione:

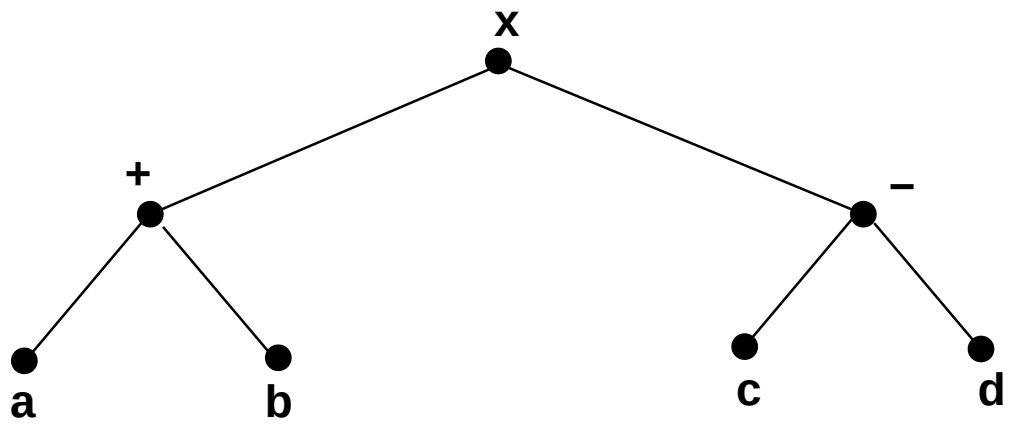
- sintattica astratta
- sintattica concreta
- a run-time
- data dalla regola di valutazione

La rappresentazione ad albero (astratta) chiarisce la struttura di controllo dell'espressione



Gli alberi lasciano comunque parte dell'ordine di valutazione non specificato (nell'esempio qual è l'ordine nella valutazione tra `a`, `b`, `0` e `1`?). Inoltre, attenti agli effetti collaterali.

Per rappresentare le espressioni in un LP occorre usare una sintassi concreta, ovvero "linearizzare" gli alberi:



- notazione prefissa: × + `a b` − `c d`
- notazione infissa: `(a`+`b)` × `(c`−`d)`
- not. postfissa (polacca inversa): `a b` + `c d` −×

Tutte queste diverse notazioni per le espressioni hanno i loro pregi e possono essere scelte in fase di progetto del linguaggio di programmazione (tipicamente prefissa + infissa).

Rappresentazione delle Espressioni a Run-Time

Per decodificare le espressioni nella loro usuale notazione infissa usata nel codice sorgente, esse vengono tradotte in una forma "eseguibile":

- sequenza in linguaggio macchina
- rappresentazione ad albero + interprete SW
- notazione prefissa o postfissa + interprete SW

Ma il vero problema rimane quale regola/strategia di valutazione, ovvero

quale **semantica** per le espressioni?

Regola di Valutazione Uniforme

Nell'albero si valutano prima i nodi figli (operandi) e poi si applica l'operatore del nodo. È detta anche regola di valutazione **interna**.

Problema 1: funzioni non strette, ovvero che possono essere definite anche se qualcuno degli argomenti non è definito

`if-then-else` è una tipica funzione non stretta. Siccome normalmente è l'unico costrutto non stretto, ad esso si applica una regola di valutazione uniforme particolare:

`if B then` E_1 `else` E_2:

`se B VERO e` E_1 valuta $V_1 \Rightarrow$
(`if B then` E_1 `else` E_2) valuta V_1

`se B FALSO e` E_2 `valuta` $V_2 \Rightarrow$
(`if B then` E_1 `else` E_2) valuta V_2

Regola di Valutazione Esterna

Un operando viene valutato solo quando serve: per calcolare `f(E)` sostituire per prima cosa `E` nel corpo di `f` e poi procedere alla valutazione.

vantaggio: è completa, ovvero se `E` ha un valore allora la regola lo computa

svantaggio: un argomento può essere valutato troppe volte

```
fun f x = x + x;
fun e y = y * y;
```

reg. interna:
f(e 3) → f(3*3) → f(9) → 9+9 → 18

reg. esterna:
f(e 3) → e(3)+e(3) → (3*3)+(3*3) →9+9→18

Le due regole di valutazione uniforme viste (interna ed esterna) corrispondono a due comuni tecniche di passaggio di parametri ai sottoprogrammi, rispettivamente la **by value** e la **by name**, quindi sono note anche con questi nomi.

Esempio ML: ML adotta la call by value:

`problema (a):`

```
fun cond(b, x, y) = if b then x else y;
```

`cond` si comporta come if-then-else?

`problema (b):`

E_1 `orelse` E_2 abbrevia `if` E_1 `then TRUE else` E_2`;`
Quindi vale

```
fun orelse(a, b) = if a then TRUE else b;
```

?

Problema 2: effetti collaterali (assenti nei linguaggi funzionali puri)

Tale problema è indipendente dalla strategia di valutazione usata

```
A + f(B)
```

$\rightarrow$ `f(B)` potrebbe modificare `A` per side-effect

$\Rightarrow$ si perde la commutatività una volta fissato l'ordinamento nella valutazione degli operandi

Esempio:

```
int f(int *x)
    { *x = *x - 1;
      return 1;}
a = 1;
a = a + f(&a);
```

qual è il valore finale di `a`?

Controllo di Sequenza tra Comandi

Notazione:

$$< C, \sigma > \rightarrow \sigma_1$$

comando C eseguito nello stato σ produce stato σ_1

Assumiamo per semplicità che la valutazione di espressioni non provochi effetti collaterali:

$$< E, \sigma > \rightarrow v$$

la valutazione dell'espressione E nello stato σ produce il valore v

Chiamiamo `valutazioni` le 2 espressioni di cui sopra.

Notazione per regola di valutazione:

$$\frac{valutaz_1 \cdots valutaz_K}{valutaz_N}$$

se ho le valutazioni $valutaz_1 \cdots valutaz_K$ allora ho la valutazione $valutaz_N$

Esempio

$$\frac{< E, \sigma > \rightarrow v}{< x := E, \sigma > \rightarrow \sigma'}$$

dove σ' è il nuovo stato della macchina per il quale `r-value(x) = v`

Composizione Sequenziale

C_1, C_2 comandi $\Rightarrow (C_1; C_2)$

$$\frac{< C_1, \sigma > \rightarrow \sigma'' \quad < C_2, \sigma'' > \rightarrow \sigma'}{< C_1; C_2, \sigma > \rightarrow \sigma'}$$

...generalizzare al caso n-ario $(C_1; C_2; \cdots C_n)$

Composizione Condizionale

$$\frac{< B, \sigma > \rightarrow vero \quad < C_1, \sigma > \rightarrow \sigma'}{< \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma > \rightarrow \sigma'}$$

$$\frac{< B, \sigma > \rightarrow falso \quad < C_2, \sigma > \rightarrow \sigma'}{< \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma > \rightarrow \sigma'}$$

Costrutti Iterativi

$$\frac{< B, \sigma > \rightarrow F}{< \texttt{while } B \texttt{ do } C, \sigma > \rightarrow \sigma}$$

$$\frac{< B, \sigma > \rightarrow T \quad < C, \sigma > \rightarrow \sigma'' \quad < \texttt{while } B \texttt{ do } C, \sigma'' > \rightarrow \sigma'}{< \texttt{while } B \texttt{ do } C, \sigma > \rightarrow \sigma'}$$

for $\Rightarrow$ numero fissato di iterazioni

idea: `for i` $\in$ `S do C`
dove S è un insieme ordinato $[v_0 \cdots v_k]$

```
i := v0;
C;
i := v1;
C;
...
i := vk;
C;
```

Esempio Pascal:

```
for I:=E1 to E2 do C
```

Vincoli del "buon for":

- `S` non varia durante l'esecuzione
- variabile `i` non modificabile esplicitamente in `C`

Attenzione: il `for` del C non è un vero for:

```
for (E1; E2; E3) C;    ⇒    E1;
                            while (E2) {
                               C;
                               E3;
                            }
```

Costrutti "Cattivi"

1. **goto + etichette**:

```
...
goto 100
...
100:  C
```

2. **salti di tipo escape**. Ad esempio in C:

   break: uscita da switch o da un ciclo
   continue: viene forzata l'iterazione successiva
   return
   exit

Come Istruire l'Interprete

$[C]_\sigma$ = risultato della valutazione di C nello stato σ ovvero

$$\sigma' = [C]_\sigma \quad \Leftrightarrow \quad < C, \sigma > \rightarrow \sigma'$$

$[E]_\sigma$ = risultato della valutazione di E nello stato σ ovvero

$$v = [E]_\sigma \quad \Leftrightarrow \quad < E, \sigma > \rightarrow v$$

Ad esempio:

$$[x := E]_\sigma = \sigma'$$

t.c σ' è ottenuto da σ ponendo

$$\texttt{r-value(x)} = [E]_\sigma$$

Possiamo dare le regole dell'interprete nel modo seguente:

$$[(C_1; C_2)]_\sigma = [C_2]_{[C_1]_\sigma}$$

$$[\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2]_\sigma =$$

$$[C_1]_\sigma \quad se \;\; [B]_\sigma = Vero$$

$$[C_2]_\sigma \quad se \;\; [B]_\sigma = Falso$$

$$[\texttt{while } B \texttt{ do } C]_\sigma =$$

σ se $[B]_\sigma$ = Falso

$[\texttt{while } B \texttt{ do } C]_{[C]_\sigma}$ se $[B]_\sigma$ = Vero

Implementazione

1. le regole date precedentemente definiscono direttamente un interprete

2. se l'implementazione è compilativa oppure se l'interprete è molto vicino a quello di una macchina convenzionale HW, l'implementazione di `if` e `while` è basata su istruzioni di `confronto + salto`