Indice

Capitolo 1

Introduzione agli elementi di un linguaggio di programmazione

I motivi della creazione ed utilizzo di un linguaggio di programmazione di alto livello sono: di fornire una descrizione precisa, ovvero una specifica formale; offrire un interpretazione tramite interprete da compilare.

Le caratteristiche principali che categorizzano i linguaggi di programmazione sono la sintassi e la semantica, la quale può essere statica o dinamica.

1.1 Linguaggi staticamente tipati

Nei linguaggi tipizzati staticamente il *tipo* di variabile viene stabilito nel codice sorgente, per cui si rende necessario che:

- gli operatori e le assegnazioni devono essere usati coerentemente con il *tipo* dichiarato;
- le variabili siano usate consistentemente rispetto la loro dichiarazione.

I vantaggi della staticità risiedono nella preventiva rilevazione degli errori e nell'efficienza di calcolo risultante dalla coerenza tra codice e compilatore.

1.2 Linguaggi dinamicamente tipati

Nei linguaggi di programmazione dinamicamente tipati le variabili sono assegnate ai *tipi* durente l'esecuzione del programma, ne consegue che:

- la semantica statica non è definita;
- un utilizzo incosistenze di variabili, operazioni o assegnazioni generano errori dinamici basati sui tipi.

I linguaggi dinamici per questi motivi risultano essere più semplici ed espressivi.

1.2.1 Esempi di errori

Listing 1.1: Errore di sintassi

```
x = ;
```

Un errore sintattico generico a molti linguaggi è un espressione formattata erroneamente.

Listing 1.2: Errore statico

```
int x=0;
if(y<0) x=3; else x="three"
```

In un linguaggio statico come *Java* l'esempio precedente darebbe un errore in quanto una stringa non può essere convertita in un tipo intero.

Listing 1.3: Errore Dinamico

```
x = null;
if(y<0) y=1; else y=x.value;
```

L'esempio, per $y > 0$, darebbe un errore dinamico sia in *Java*, che in un linguaggio dinamico come *Javascript*.

Capitolo 2

Stringhe

Teorema 1 *Un alfabeto A è un insieme finito non vuoto di simboli.*

Teorema 2 *Sia una stringa in un alfabeto A la successione di simboli in u:*

$$u : [1 \dots n] \to A$$

Sia:

- $[1 \dots n] = m$*, l'intervallo dei numeri naturali tale che:*

$$1 \leq m \geq n;$$

- u *sia una funzione totale;*
- n *sia la lunghezza di* $u : length(u) = n$*.*

Teorema 3 *Un programma è una stringa in un alfabeto A.*

2.1 Esempio di stringhe

2.1.1 Stringa vuota

$$u : [1 \dots 0] \to A$$

Esiste un unica funzione $u : 0 \to A$

Le notazioni standard di una stringa vuoto sono: ε, λ

2.1.2 Stringa non vuota

Si consideri $A = \{'a', \ldots, 'z'\} \cup \{'A', \ldots, 'Z'\}$, l'alfabeto inglese di lettere minuscole e maiuscole. La funzione $u : [1 \ldots 4] \to A$ rappresenta la stringa "$Word$" con:

- $u(1)$ = 'W'
- $u(1)$ = 'o'
- $u(1)$ = 'r'
- $u(1)$ = 'd'

2.1.3 Concatenazione di stringhe

Teorema 4

$$length(u \cdot v) = length(u) + length(v)$$

Per ogni $i \in [1 \ldots length(u) + length(v)]$

$$(u \cdot v)(i) = \textbf{\textit{if}}\ i \leq< length(u) \textbf{\textit{then}}\ u(i) \textbf{\textit{else}}\ v(i - length(u))$$

Monoide

La concatenazione è associativa, ma non commutativa.

La stringa vuota è l'identità dell'elemento.

Induzione

La definizione di u^n per induzione su $n \in \mathbb{N}$:

Base: $u^0 = \lambda$

Passo induttivo: $u^{n+1} = u \cdot u^n$ Per cui u^n si concatena con se stesso n volte.

2.1.4 Insiemi di stringhe

Teorema 5 *Sia A un alfabeto:*

- A^n = *l'insieme di tutte le stringhe in A con lunghezza n;*
- A^+ = *l'insieme di tutte le stringhe in A con lunghezza maggiore di 0;*
- $A^\star$ = *l'insieme di tutte le stringhe in A;*
- $A^+ = \bigcup_{n>0} A^n$;
- $A^\star = \bigcup_{n\geq 0} A^n = A^0 \cup A^+$

2.2 Linguaggio formale

Teorema 6 (Nozione sintattica di linguaggio) *Un linguaggio L in un alfabeto A è un sottoinsieme di $A^\star$*

Esempio: L'insieme L_{id} di tutti gli identificatori di variabile:

$$A = \{'a', \ldots, 'z'\} \cup \{'A', \ldots, 'Z'\} \cup \{'0', \ldots, '9'\}$$

$$L_{\text{id}} = \{'a', 'b', \ldots, 'a0', 'a1', \ldots\}$$

2.2.1 Composizione di operatori tra linguaggi

Le operazioni possono essere di concatenazione o di unione:

- **Concatenazione:** $L_1 \cdot L_2 = \{u \cdot w | u \in L_1, w \in L_2\}$;
- **Unione:** $L_1 \cup L_2$.

2.2.2 Intuizione

Unione

$L = L_1 \cup L_2$: qualsiasi stringa L è una stringa di L_1 o di L_2.

Esempio:

$$L' = \{'a', \ldots, 'z'\} \cup \{'A', \ldots, 'Z'\}$$

Concatenazione

$L = L_1 \cdot L_2$: qualsiasi stringa L è una stringa di L_1, seguita da una stringa di L_2.

Esempio:

$$\{'a', 'ab'\} \cdot \{\lambda, '1'\} = \{'a', 'ab', 'a1', 'ab1'\}$$

$$L_{\text{id}} = L' \cdot A^\star \text{ con } A = \{'a', \ldots, 'z'\} \cup \{'A', \ldots, 'Z'\} \cup \{'0', \ldots, '9'\}$$

2.2.3 Monoide

La concatenazione è associativa, ma non commutativa.

$A^0 (= \{\lambda\})$ è l'identità dell'elemento; quindi A^0 *non è l'elemento neutro*, l'elemento neuro è $0 = \{\}$.

2.2.4 Passo induttivo

L^n è definito per induzione su $n \in \mathbb{N}$: Base: $L^0 = A^0 (= \{\lambda\}$,

Passo induttivo: $L^{n+1} = L \cdot L^n$.

2.2.5 Operatori + e *

- **Addizione:** $L^+ = \bigcup_{n>0} L^n$;
- **Moltiplicazione:** $\star$ viene chiamata *Kleen star, stella di Kleen.*

$$L^\star = \bigcup_{n \geq 0} L^n$$

Sono equivalenti $L^\star = L^0 \cup L^+$, $L \cdot L^\star$.

Intuizione

- Qualsiasi stringa di L^+ è ottenuta concatenando una o più stringhe di L;
- Qualsiasi stringa di $L^\star$ è ottenuta concatenando 0 o più stringhe di L: *Concatenando zero stringhe si ottiene la stringa vuota.*

Capitolo 3

Espressioni regolari

Le espressioni regolari sono un formalismo comunamente utilizzato per definire linguaggi semplici.

Teorema 7 *La definizione induttiva di un espressione regolare su un alfabeto A:*

Base:

- 0 *è un espressione regolare di* A*;*
- λ *è un espressione regolare di* A*;*
- *per ogni* $\sigma \in A$*,* σ *è un espressione regolare in* A*.*

Passo induttivo:

- *se* e_1 *ed* e_2 *sono espressioni regolare di* A*,*

  allora $e_1|e_2$ *è un espressione regolare di* A*;*
- *se* e_1 *ed* e_2 *sono espressioni regolare di* A*,*

  allora e_1e_2 *è un espressione regolare di* A*;*
- *se* e *è un espressione regolare di* A*,*

  allora $e^\star$ *è un espressione regolarare di* A*.*

Esercizio Scrivere un **REGEX** che esprima i nomi di variabili permessi.

$$L_{id} = (\{'a', \ldots, 'z'\} \cup \{'A', \ldots, 'Z'\}) \cdot \{'a', \ldots, 'z'\} \cup \{'A', \ldots, 'Z'\} \cup \{'0', \ldots, '9'\}*$$

$$e_{id} = (a| \ldots |z|A| \ldots |Z)(a| \ldots |z|A| \ldots |Z|0| \ldots |9)*$$

3.1 Semantica

La semantica di un espressione regolare in A è un liguaggio su A:

- $\emptyset \rightsquigarrow$ insieme vuoto;
- $\epsilon \rightsquigarrow \{\epsilon\}$;
- $\sigma \rightsquigarrow \{"\sigma"\}, \forall \sigma \in A$;
- $e_1 \mid e_2 \rightsquigarrow$ unione delle semantiche di e_1 ed e_2;
- $e_1 e_2 \rightsquigarrow$ concatenzatione delle semantiche di e_1 ed e_2.

3.2 Sintassi concreta delle espressioni regolari

3.2.1 Precedenza ed associatività

- la **stella di Kleene** ha priorità sulla concatenzazione e l'unione;
- la concatenazione ha precedemza sull'unione;
- la concatenazione e l'unione sono associative a sinistra.

3.2.2 Operatori derivati

- $e+ = ee+$: (una o più volte e);
- ϵ: è rappresentata dalla stringa vuota: $a|\epsilon$ diventa $a|$;
- $e? = |e$: (e è opzionale, ovvero uno o nessuno);

- $[a0B]$: uno qualsiasi dei caratteri nella quadre $(a|0|B)$;
- $[b-d]$: uno qualsiasi dei caratteri nel range tra le quadre $(b|c|d)$;
- $[a0B]|[b-d]$: può essere scritto come $[a0Bb-d]$;
- $[\wedge\dots]$: qualsiasi carattere ad eccezioni di $\dots$ (esempio: $[^a0Bb-d]$ qualsiasi carattere ad eccezione di $a, 0, B, b, c, d$).

3.2.3 Caratteri speciali in JAVA

- . rappresenta ogni carattere;
- \ è il carattere di *escape*, per dare un significato speciale ai caratteri regolari, oppure un significato ordinario per caratteri speciali.

Caratteri speciali cui dare un significato ordinario

Esempi:

$$|, *, +, ?, ., , (,), [,], -, \wedge$$

Semantiche:

- $\backslash. \rightsquigarrow \{"."\}$,
- $\backslash\backslash \rightsquigarrow \{"\backslash"\}$,
- $. \rightsquigarrow \{s \mid s \text{ ha lunghezza 1, l'insieme di tutti i caratteri}\}$,
- $\backslash \rightsquigarrow$ non è sintatticamente corretto.
- $n \rightsquigarrow \{"n"\}$,
- $\backslash n \rightsquigarrow \{"linefeed"\}$,

Caratteri cui dare un significato speciale

- $\backslash t$: tab;
- $\backslash n$; newline;
- $\backslash s$: qualsiasi spazio vuoto;
- $\backslash S$: qualsiasi spazio non vuoto;
- $\backslash d$: qualsiasi carattere numerico ($[0-9]$);
- $\backslash D$: qualsiasi carattere non numerico ($[\wedge 0-9]$);
- $\backslash w$: qualsiasi parola ($[[a-zA-Z\_0-9]]$);
- $\backslash W$: qualsiasi carattere che non sia una parola ($[\wedge \backslash w]$).

Teorema 8 *Un linguaggio si dice* regolare *se può essere definito da un' espressione regolare.*

3.3 Analisi lessicale

Teorema 9 *Un* lexeme *è una sottostringa considerata come un'unità sintattica.*

Teorema 10 *L'analisi lessicale affronta il problema della decomposizione di una stringa in un* lexeme.

Teorema 11 *Un* lexer *o* scanner *è un programma che esegue l'analisi lessicale e genera* lexemes.

Esempio in C: La stringa "$x2 = 042;$ è decomposta nei *lexemes* seguenti:

- "$x2$";
- " $=$ ";
- "042";
- ";".

3.3.1 Token

Un *token* è una nozione di *lexeme* pià astratta; ad un *token* corrisponde sempre un *lexeme*.

In alcuni casi un *token* può mantenere informazioni sulla sematica, come i valori dei numeri.

Un *tokenizer* è un programma che esegue l'analisi lessicale e genera *token*.

Esempio in C: La stringa "$x2 = 042;$" è decomposta nei token seguenti:

- *IDENTIFIER*: con il nome "$x2$";
- *ASSIGN_OP*;
- *INT_NUMBER*: con il valore di 34;
- *STATEMENT_TERMINATOR*.

3.4 Linguaggi regolari

Teorema 12 *Un* linguaggio regolare *è un linguaggui definibile con un espressione regolare.*

I linguaggi regolari *possono essere definiti in altre maniere equivalenti:*

- *con una grammatica regolare a destra o sinistra, anche chiamata* lineare*;*
- *con una serie di* automata *non deterministica o deterministica finita (**NFA** o **DFA**).*

3.4.1 Limitazioni

I linguaggi regolari sono linguaggi semplici. Esempio:

- il linguaggio degli identificatori;
- il linguaggio dei numeri.

Le espressioni regolari possono definire le unità che costituiscono la sintassi di un linguaggio di programmazione, ma non possono definire la sintassi dell'intero linguaggio.

3.4.2 Esempi di linguaggi non regolari

Il linguaggio di espressioni con numeri naturali, addizione binaria e moltiplicazione e parentesi **non possono** essere definiti da un espressione regolare: il problema è posto dalle parentesi, per cui se venissero rimosse, allora il linguaggio sarebbe regolare.

Un altro esempio di linguaggio semplice non regolare:

$$\{"a"^n"b"^n \mid n \in \mathbb{N}\} = \{"","ab","aabb","aaabbb",\dots\}.$$

Capitolo 4

Analisi sintattica

Teorema 13 *L' **analisi sintattica** è definita sull'analisi lessicale, risolve:*

- *il riconoscimento di una sequenza di* lexems/tokens *come validi se rispettano alcune regole sintattiche;*
- *la costruzione, in caso di successo, di una rappresentazione astratta della sequenza riconosciuta per eseguire delle operazioni su di essa.*

4.1 Parser

Teorema 14 *Un **parser** è un programma che esegue l'analisi sintattica.*

4.1.1 Parsers per i linguaggi di programmazione

I parser per i linguaggi di programmazione, riconoscono i *token* generati da un *tokenizer*, mentre le regole sintattiche sono definite formalmente da una *grammatica*.

I parser generano:

- un albero **parse/deviation**, una rappresentazione meno astratta della sequenza analizzata;
- un **albero sintattico astratto** (abstract syntax tree, AST), una rappresentazione pi astratta della sequenza analizzata.

Inoltre i parser possono essere scritti a mano, oppure generati automaticamente da una grammatica con specifici software come *ANTLR* o *BISON*.

Primo esempio con sintassi C/Java/C++ Token analizzati:

- *IDENTIFIER*: con il nome "$x2$";
- *ASSIGN_OP*;
- *INT_NUMBER*: con il valore di 34;
- *STATEMENT_TERMINATOR*.

Il parser ritornerà errore dato che la sequenza non è riconosciuta oppure uno o più messaggi di errore sono presenti.

Secondo esempio con sintassi C/Java/C++ Token analizzati:

- *IDENTIFIER*: con il nome "$x2$";
- *ASSIGN_OP*;
- *INT_NUMBER*: con il valore di 34;
- *ADD_OP*;
- *INT_NUMBER*: con il valore di 10;
- *STATEMENT_TERMINATOR*.

Il parser riconosce la sequenza e genera un *AST*.

Capitolo 5

Context free (CF) grammars

Le grammatiche *context free*, sono il formalismo più diffuso per definire le regole sintattiche di un linguaggio di programmazione. Sono più espressive di un espressione regolare e sono basate sulla concatenazione d unzione di più nomi e su definizioni ricorsive.

Listing 5.1: Una grammatica **CF** per espressioni semplici

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Da notare che: *Num* è definito nella grammatica solo per completezza, infatti i token *Num* sono definiti separatamente da un espressione regolare.

Listing 5.2: Esempio rivisitato di una grammatica **CF** per espressioni semplici

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
NUM definito da 0|1
```

Notazione: In *Exp* è maiuscolo solo il primo carattere: è definito nella grammatica. In *NUM* tutte le lettere sono maiuscole; è definito separatamente da un espressione regolare.

5.1 Terminologia delle grammatiche CF

Listing 5.3: Esempio

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

5.1.1 Terminologia per la grammatica G

Per $G = (T, N, P)$:

- $\{'+', '*', '(', ')', '0', '1'\}$ è l'insieme T dei **simboli terminali**;
- $\{Exp, Num\}$ sono l'insieme N di simboli **non terminali**;
- l' insieme P di produzioni:

$$\{(Exp, Num), (Exp, Exp, '+' Exp), (Exp, Exp' *' Exp), (Exp, '(' Exp')'),$$
$$(Num, '0'), (Num, '1')\}$$

Da notare che:

- ogni simbolo non terminale corrisponde ad un linguaggio; i linguaggi sono definiti come unioni di concatenazioni;
- i simboli terminali sono *lexems* dei linguaggi definiti dalla grammatica;
- le produzioni hanno forma (B, α), per $B \in \mathbb{N} \cap \alpha \in (T \cup N)^*$

5.2 Grammatiche come definizione induttiva di linguaggi

5.2.1 Primo esempio

Listing 5.4: Esempio

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Definizione induttiva di linguaggi

$$Exp = Num \cup (Exp \cdot \{"+"\} \cdot Exp) \cup (Exp \cdot \{"*"\} \cdot Exp) \cup (\{"("\} \cdot Exp \cdot \{")"\})$$

$$Num = \{"0"\} \cup \{"1"\}$$

Da notare che:

- $Exp = Num \cup \ldots$ è il caso base per Exp: un numero è un espressione;
- Exp è definito su di Num, Num è definito esclusivamente per casi base.

5.2.2 Secondo esempio

Listing 5.5: Esempio

```
Exp ::= Term | Exp '+' Term | Exp '*' Term
Term ::= '(' Exp ')' | Num
Num ::= '0' | '1'
```

Da notare che: Le definizioni di Exp e $Term$ sono ricorsive reciprocamente.

5.3 Derivazioni

Listing 5.6: Esempio

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

5.3.1 Linguaggi generati da una grammatica

- Una grammatica genera un linguaggio per ogni simbolo non terminale;
- la grammatica precedente genera due linguaggi L_{Exp} e L_{Num};
- il linguaggio per Num è relativamente semplice: $L_{\text{Num}} = \{"0", "1"\}$.

Da notare che: per definire L_{Num} e per dimostrare che $"1 + 0" \in L_{\text{Exp}}$ e che $"1 + *(" \notin L_{\text{Exp}}$ si rendono necessarie la **derivazione a passo singolo e la derivazione a passi multipli**.

5.3.2 Derivazione ad un passo

- $Exp \rightarrow Exp' *' Exp$ è usata la produzione $(Exp, Exp' *' Exp)$;
- $Exp' *' Exp \rightarrow Num' *' Exp$ è usata la produzione (Exp, Num);
- $Num' *' Exp \rightarrow Num' *' Num$ è usata la produzione (Exp, Num);
- $Num' *' Num \rightarrow' 0'' *' Num$ è usata la produzione $(Num,' 0')$;
- $'0'' *' Num \rightarrow' 0'' *'' 1'$ è usata la produzione $(Num,' 1')$;

Da notare che: Non esiste alcuna derivazione da $'0'' *'' 1'$ dato che nessuna produzione può essere usata; $'0'' *'' 1'$ è la stringa $0 * 1$ che appartiene a L_{Exp}.

5.3.3 Definizioni di derivazione

Derivazione ad un passo

Teorema 15 *La derivazione ad un passo per la grammatica $G = (T, N, P)$:*

- *possiede una forma* $\alpha_1 B \alpha_2 \to \alpha_1 \gamma \alpha_2$*;*
- $\alpha_1, \alpha_2 \in (T \cup N)^*$*;*
- $(B, \gamma) \in P$ *ovvero* (B, γ) *in produzione.*

Derivazione a più passi

Teorema 16 *La chiusura transitiva di* $\to$*:*

- *il caso base: se* $\gamma_1 \to \gamma_2$*, allora* $\gamma_1 \to^+ \gamma_2$*;*
- *caso induttivo: se* $\gamma_1 \to \gamma_2$*, e* $\gamma_2 \to^+ \gamma_3$*, allora* $\gamma_1 \to^+ \gamma_3$*.*

Linguaggio generato

Il linguaggio L_B generato da $G = (T, N, P)$ per i non terminali $B \in N$:

- tutte le stringhe di terminali che possono essere derivati in uno o più passaggi da B;
- formalmente: $L_B = \{u \mid B \to^+ u\}$.

Capitolo 6

Alberi di derivazione

6.1 Albero di derivazione (parse tree)

Osservazione 1 Le grammatiche CF sono utilizzate per definire linguaggi ed implementare parsers, i parsers dovrebbero generare gli alberi, ma le derivazioni non sono alberi.

Osservazione 2 Un passo di derivazione è determinato da:

- la produzione usata;
- lo specifico simbolo non termitale che rimpiazza.

Quest'ultimo punto non influenza la stringa finale dei terminali ottenuti dalla derivazione.

Intuizione Un albero di derivazione è una generalizzazione di una derivazione a più passaggi in modo che la stringa derivata contenga solo terminali e che i non terminali siano rimpiazzati in parallelo.

6.1.1 Esempi di alberi di derivazione (ANTLR)

Listing 6.1: Grammatica ANTLR

```
grammar SimpleExp;

Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Albero di derivazione per "1*1+1"

Mostrare che "$1 * 1 + 1$" $\in L_{\text{Exp}}$ usando la nozione di derivazione ad uno o più passi.

$$\begin{aligned} Exp \to Exp' +' Exp &\to^+ Exp \times Exp' +' Num' \\ &\to^+ Num' *' Num' +' 1' \\ &\to^+ \text{'1' '*' '1' '+' '1'} \end{aligned}$$

Esercizio: Mostrare che "$1 + ($" $\notin L_{\text{Exp}}$:

$$\begin{aligned} Exp \to Exp' *' Exp &\to^+ Num' *' Exp' +' Exp \to^+ \\ &\text{'1' '*' Num '+' Num} \to^+ \text{'1' '*' '1' '+' '1'} \end{aligned}$$

6.1.2 Definizione di albero di derivazione in G=(T,N,P)

Albero di derivazione per $u \in T^*$ partendo da $B \in N$.

- se un nodo è etichettato da C ed ha n figli $I_1, \ldots, I_n$, allora $(C, I_1, \ldots, I_n) \in P$ (ovvero, $(C, I_1, \ldots, I_n)$ è una produzione di G;
- la radice è etichettata da B;
- u è ottenuto dalla concatenaziona da sinistra a destra dii tutte le etichette terminali (nodi foglia).

Definizione equivalente di un linguaggio generato

Il linguaggio L_B generato da $G = (T, N, P)$ per un non terminale $B \in N$ è composto da tutte le stringhe u di terminali così che esiste un albero di derivazione per u

partendo da B.

Capitolo 7

Grammatiche ambigue

Teorema 17 *Una grammatica $G = (T, N, P)$ è ambigua per $B \in N$ se esitono due differenti alberi di derivazione partendo da B per la stessa stringa.*

Per esempio la grammatica $1 + 1 + 1$ è ambigua a seconda delle parentesi.

7.1 Soluzioni per l'ambiguità

Per risolvere il problema dell'ambiguità si può cambiare la sintassi:

7.1.1 Notazione prefissa

Listing 7.1: Notazione prefissa

```
Exp ::= Num | '+' Exp Exp | '*' Exp Exp
Num ::= '0' | '1'
```

In questo caso esiste un unico albero di derivazione per "$11 + 1*$ e le parentesi non sono più necessarie.

7.1.2 Notazione postfissa

Listing 7.2: Notazione postfissa

```
Exp ::= Num | Exp Exp '+' | Exp Exp '*'
```

```
Num ::= '0' | '1'
```

In questo caso esiste di nuovo un unico albero di derivazione e le parentesi sono di nuovo non necessarie.

7.1.3 Notazione funzionale

Listing 7.3: Notazione funzionale

```
Exp ::= Num | 'add' '(' Exp ',' Exp ')' | 'mul' '(' Exp ',' Exp
    ')'
Num ::= '0' | '1'
```

In questo esempio esiste un unico albero di derivazione per $add(1, mul(1, 1)$.

7.1.4 Notazione infissa

Generalmente la notazione infissa è una soluzione più pratica.

- si definiscono le regole di associatività per gli operatori binari:
  - addizione associativa a sinistra: "$1 + 1 + 1$" diventa "$(1 + 1) + 1$";
  - addizione associativa a destra: "$1 + 1 + 1$" diventa "$1 + (1 + 1)$";
- si definiscono le regole di precedenza per gli operatori, usando le parentesi per sovrascriverlo:
  - la moltiplicazione ha precedenza rispetto l'addizione: "$1 + 1 * 1$" significa "$1 + (1 * 1)$";
  - l'addizione ha precedenza rispetto la moltiplicazione: "$1 * 1 + 1$" significa "$1 * (1 + 1)$".

7.1.5 Operatori con la stessa precedenza

Gli operatori binari possono avere la stessa precedenza, in questo caso condividono la regola associativa:

- addizione e moltiplicazione hanno la stessa precedenza e sono associative a sinistra: "$1 + 1 * 1$" diventa "$1 + (1 * 1)$" e "$1 * 1 + 1$" diventa "$(1 * 1) + 1$";
- addizione e moltiplicazione hanno la stessa precedenza e sono associative a destra: "$1 + 1 * 1$" diventa "$1 + (1 * 1)$" e "$1 * 1 + 1$" diventa "$1 * (1 + 1)$";

Nota che: le regole sull' associatività risolvono ambiguità tra operatori binari con la stessa precedenza, inoltre mischiando operatori con diverse **arità** rende l'eliminazione dell'ambiguità più complessa.

7.1.6 Tecniche per risolvere l'ambiguità

- una grammatica ambigua G è trasformata in una grammatica non ambigua G';
- l' **equivalenza** significa che per tutti i non terminali di B e G, i linguaggi generati da G, G', B sono uguali;
- è possibile per la **trasformazione** di codificare l' associatività e le regole di precedenza nella grammatica non ambigua G'.

Esempio 1: + e * con la stessa precedenza

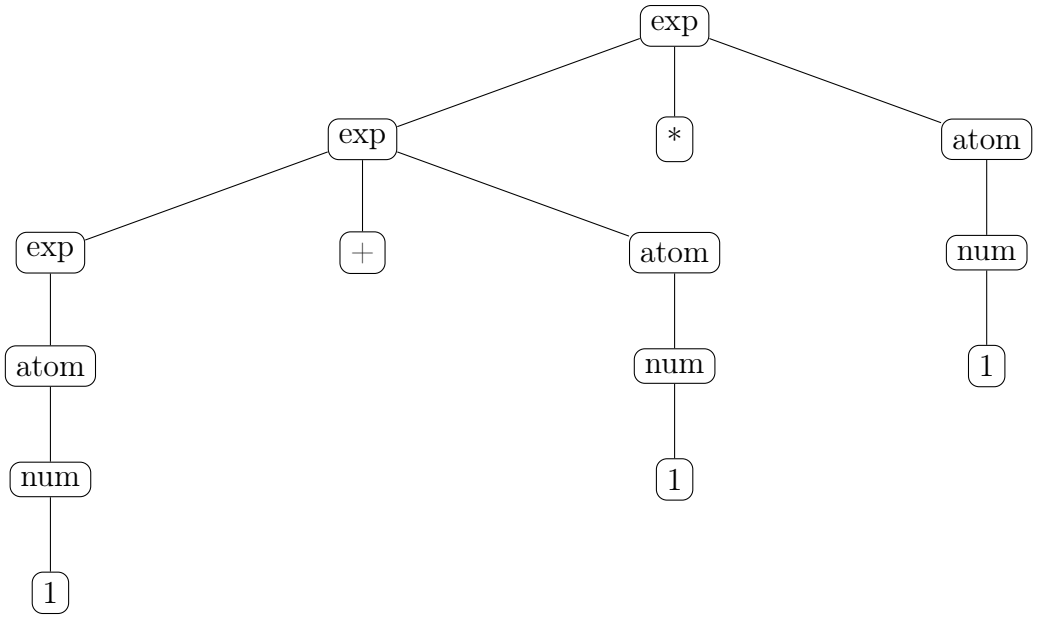
Listing 7.4: Grammatica ambigua

```
Exp ::= Num | Exp '+' Exp | Exp '*' Exp | '(' Exp ')'
Num ::= '0' | '1'
```

Listing 7.5: Associatività a sinistra non ambigua

```
Exp ::= Atom | Exp '+' Atom | Exp '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

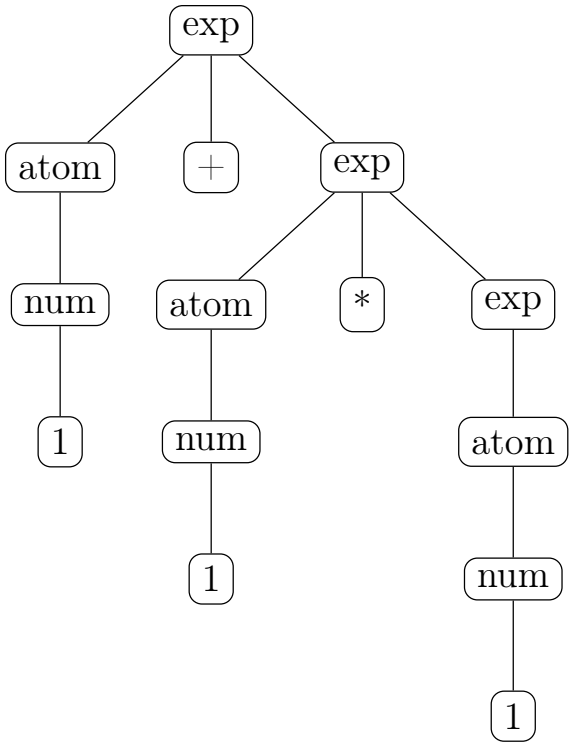
Nota che: $Exp' +' Atom | Exp' *' Atom$ significa che sul lato destro di $+(*)$, le addizione (e moltiplicazioni) sono permesse solo se circondate da parentesi.



Listing 7.6: Associatività a destra non ambigua

```
Exp ::= Atom | Atom '+' Exp | Atom '*' Exp
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Nota che: $Atom' + 'Exp|Atom' * 'Exp$ significa che sul lato sinistro di $+(*)$,le addizione (e moltiplicazioni) sono permesse solo se circondate da parentesi.



Esempio 2: * con la maggiore precedenza

Listing 7.7: Associative a sinistra non ambigue

```
Exp ::= Mul | Exp '+' Mul
Mul ::= Atom | Mul '*' Atom'
Atom ::=  Num | '(' Exp ')'
Num ::= '0' | '1'
```

Nota che: $Mul' * ' Atom$ significa che entrambi i lati delle addizione di $*$ sono permesse solo se circondate da parentesi.

Listing 7.8: Associative a destra non ambigue

```
Exp ::= Mul | Exp '+' Mul
Mul ::= Atom | Atom '*' Mul'
Atom ::=  Num | '(' Exp ')'
Num ::= '0' | '1'
```

Nota che: $Atom' * 'Mul$ significa che entrambi i lati delle addizione di $*$ sono permesse solo se circondate da parentesi.

Esempi rimanenti

- $*$ con precedenza ed associativtà a sinistra, $+$ associatività a destra;
- $*$ con precedenza ed associativtà a destra, $+$ associatività a sinistra;
- $+$ con precedenza ed associativtà a sinistra, $*$ associatività a destra;
- $+$ con precedenza ed associativtà a destra, $+$ associatività a sinsitra;

7.2 Sintassi ambigua per statements

Listing 7.9: Esempio di ambiguità per gli statements

```
Stmt ::= ID '=' Exp | 'if' '(' Exp ')' Stmt | Stmt ';' Stmt | '
   {' Stmt '}'
Exp ::= ID | BOOL // ID e BOOL sono definiti da espressioni
   regolari
```

Figura 7-1: Primo esempio di albero di derivazione per "if(x) x=false;y=true"

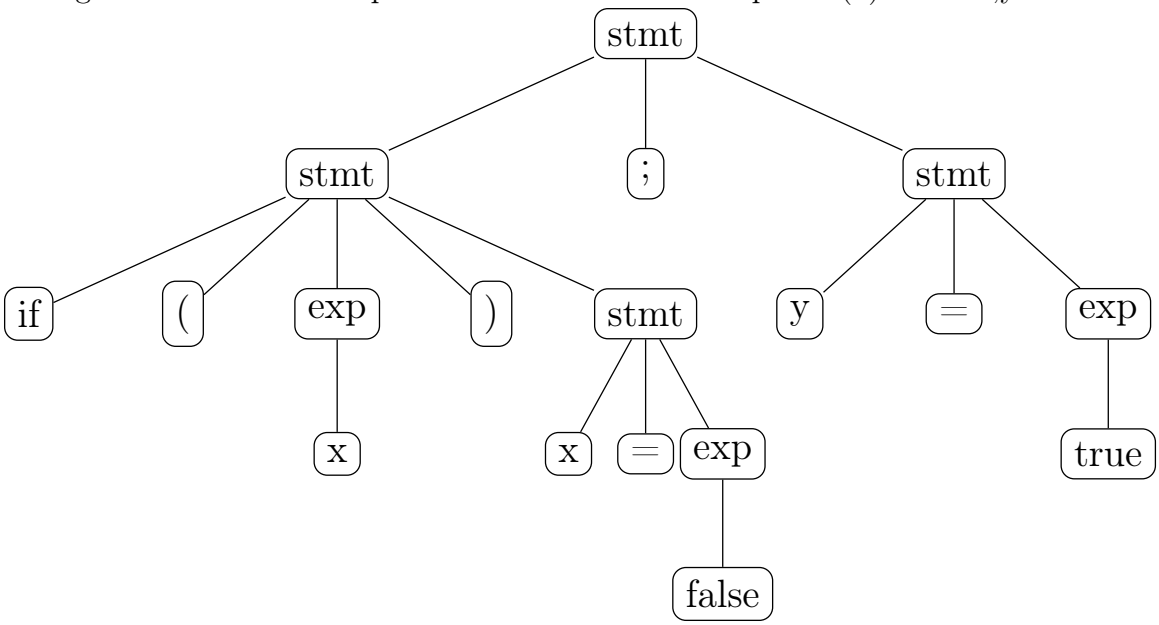
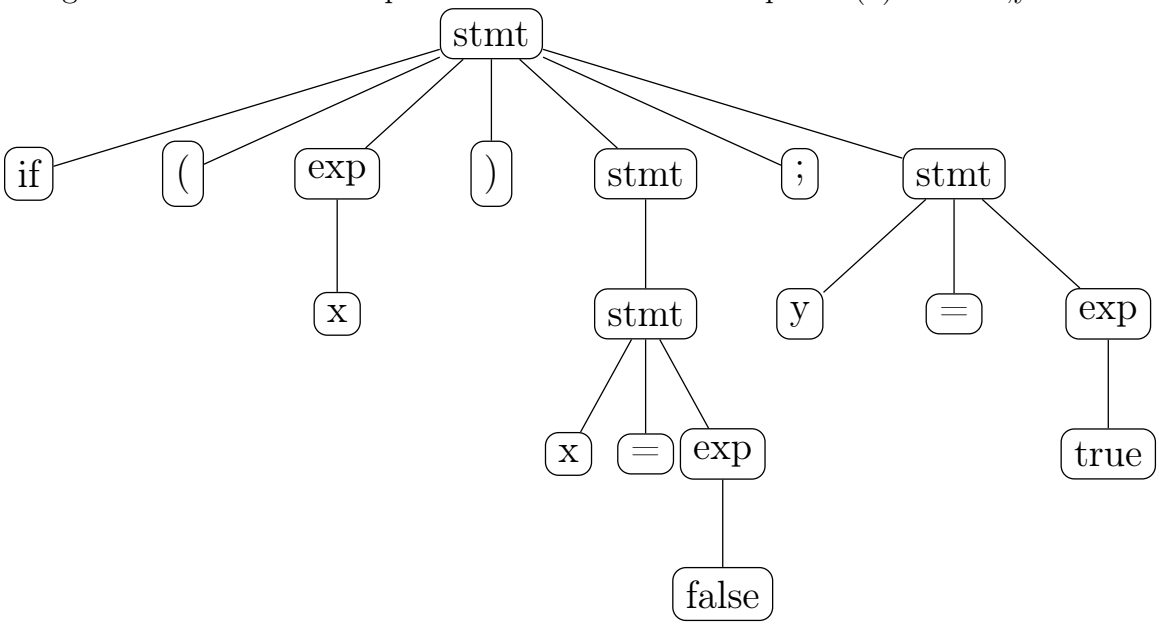


Figura 7-2: Secondo esempio di albero di derivazione per "if(x) x=false;y=true"



Capitolo 8

Costruzione di un parser da una grammatica

Individuiamo due passaggi per costruire un parser da una grammatica:

1. la grammatica non deve essere ambigua;

   Listing 8.1: Una grammatica non ambigua

   ```
   Exp ::= Mul | Exp '+' Mul
   Mul ::= Atom | Mul '*' Atom
   Atom ::= Num | '(' Exp ')'
   Num ::= '0' | '1'
   ```

2. per ogni token di input, il parser dever sceltiere una produzione unica da usare per l'albero *parse* corretto.

Problemi:

- i token sono letti dal parser da sinsitra a destra;
- nella più semplice delle ipotesi, il parser è a conoscenza solo del token successivo (*lookahead token*);

- un parser con un *lookahead token* non è in grado di scegliere la giusta produzione per la grammatica di cui sopra.

8.1 Come costruire un parser da una grammatica

Listing 8.2: Una grammatica non ambigua

```
Exp ::= Mul | Exp '+' Mul
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Quindi un parser con **un** *lookahead token* non può essere costruito per la grammatica di cui sopra.

Controesempio: Se il primo lookahead token è un numero, allora entrambe le produzioni di *Exp* potrebbero funzionare. A seconda del secondo lookahead token t:

- la produzione (Exp,Mul) è usata se t è $'*'$ oppure la fine dello stream di input;
- la produzione (Exp,$Exp\ '+'\ Mul$) è usata se $t = '+'$.

Osservazioni: Per costruire un albero *parse*, la produzione (Exp,Mul) deve essere usata, inoltre quando le produzioni di Exp sono usate consecutivamente, vengono ottenute stringhe della forma seguente: Mul, oppure Mul seguita dalla stringa $'+'\ Mul$ ripetuta una o più volte.

Listing 8.3: Grammatica rivisitata a seguito delle osservazioni precedenti

```
Exp ::= Mul | AddSeq
AddSeq ::= '+' Mul | '+' Mul AddSeq
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

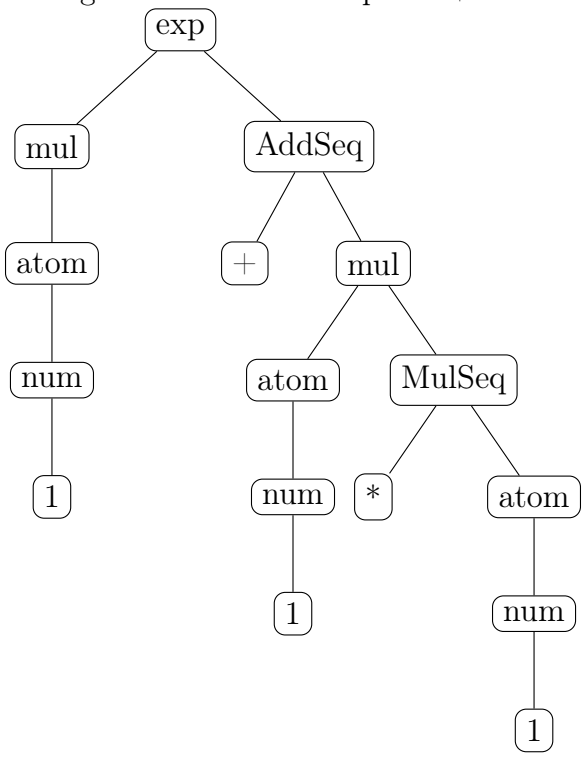
Considerazioni sulla trasformazione della grammatica La grammatica è equivalente alla precedente, ma quando si costruisce un albero *parse* sappiamo che un nodo Exp deve sempre avere un figlio c a sinistra etichettato da Mul dopo che l'albero *parse* con radice c è stato costruito, quindi la produzione corretta è (Exp, Mul $AddSeq$) se il token *lookahead* è $'+'$; altrimenti la produzione è (Exp,Mul): un nodo $AddSeq$ deve sempre avere un figlio a sinistra c_1 etichettato da $'+'$, seguito da un figlio c_2 etichettato da Mul.

Dopo che l'albero con radice c_2 è costruito, la produzione corretta se il token *lookahead*$=' +'$ è: ($AddSeq$, $'+'$ Mul $AddSeq$), altrimenti la produzione è ($AddSeq$, $'+'$ Mul).

Listing 8.4: Soluzione completa

```
Exp ::= Mul | AddSeq
AddSeq ::= '+' Mul | '+' Mul AddSeq
Mul ::= Atom | Atom MulSeq
MulSeq ::= '*' Atom | '*' Atom MulSeq
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Figura 8-1: Parse tree per "1+1*1"



8.2 Grammatiche EBNF

La notazione **BNF** estende la notazione postfissa con gli operatori: $*, +, ?$. La grammatica precedente può essere trasformata in una grammatica più semplice utilizzando la notazione **EBNF**.

Listing 8.5: Soluzione completa utilizando la gammatica ENBNF

```
Exp ::= Mul | ('+' Mul)*
Mul ::= Atom ('*' Atom)*
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

8.2.1 Da una grammatica ad un parser top-down

Per semplicità consideriamo solamente il problema del riconoscimento del linguaggio, **top-down** significa che il *parse tree* è costruito dalla radice, l'albero di generazione sarà considerato nei laboratori.

Assunzioni sul tokenizer Il tokenizer definisce le procedure seguenti:

- *nextToken()*: la lettura del successivo *lookahead token*;
- *tokenType()*: il tipo del token corrente;
- *checkTokenType()*: il tipo del token corrente è controllatom un eccezione viene sollevata se il controllo fallisce, altrimenti viene letto il token successivo.

Linee guida Il codice del parser proviene direttamente dalla grammatica, struttura ricorsiva inclusa, il parser consiste in una procedura principale insieme ad una procedura per ogni simbolo non terminale della grammatica.

Listing 8.6: Pseudocodice detivara dalla grammatica EBNF precedente

```
parse(){
  nextToken()
  parseExp()
  checkTokenType(EOS)
}
parseExp(){
  parseMul()
  while(tokenType()==ADD){
    nextToken()
    parseMul()
  }
}
parseMul(){
  parseAtom()
```

```
  while(tokenType()==MUL){
    nextToken()
    parseAtom()
  }
}
parseAtom(){
  if(tokenType()==OPEN_PAR){
    nextToken()
    parseExp()
    checkTokenType(CLOSE_PAR)
  }
  else
    checkTokenType(NUM)
  nextToken()
}
```

Grammatica EBNF per operatori con associatività a destra

Listing 8.7: Grammatica non ambigua

```
  // * con precedenza maggiore, sia + che * sono associativi a
     destra
  Exp ::= Mul | Mul '+' Exp
  Mul ::= Atom | Atom '*' Mul
  Atom ::= Num | '(' Exp ')'
  Num ::= '0' | '1'
```

Listing 8.8: Grammatica EBNF equivalente

```
  Exp ::= Mul | ('+' Exp)?
  Mul ::= Atom ('*' Mul)?
  Atom ::= Num | '(' Exp ')'
  Num ::= '0' | '1'
```

Listing 8.9: Pseudocodice derivato dalla grammatica EBNF precedente

```
parse(){
  nextToken()
  parseExp()
  checkTokenType(EOS)
}
parseExp(){
  parseMul()
  if(tokenType()==ADD){
    nextToken()
    parseExp()
  }
}
parseMul(){
  parseAtom()
  if(tokenType()==MUL){
    nextToken()
    parseMul()
  }
}
parseAtom(){
  if(tokenType()==OPEN_PAR){
    nextToken()
    parseExp()
    checkTokenType(CLOSE_PAR)
  }
  else
    checkTokenType(NUM)
  nextToken()
}
```

Capitolo 9

Paradigmi di programmazione

Teorema 18 *I paradigmi di programmazione sono lo stile/approccio nell'utilizzo di un linguaggio di programmazione.*

Il più delle volte un paradigma di programmazione è basato un un modello computazionale emergente.

Esempi principali di paradigmi

- **Imperativi:** più vicini al modello hardware, quindi basati sulle nozioni di istruzione e stato, dei linguaggi di esempio sono:
  - *proceduali:* C;
  - *orientati agli oggetti:* Java;
- **dichiarativi:** basati su un modello astratto, esempi di linguaggi sono:
  - *funzionali: ML*, basati sulla nozione di *definizione di funzione* e *applicazione di funzione*;
  - *logici: Prolog*, basati sulla nozione di *regola logica* e *query*.

Di solito i linguaggi di programmazione implementano più di un paradigma per favorire la flessibilità, *Java, Javascript, Python* supportano sia i paradigmi imperativi che quelli dichiarativi.

9.1 Paradigma completamente funzionale

Caratteristiche:

- **programma:** le definizioni di funzioni matematiche ed un espressione principale;
- **computazione:** l'applicazione della funzione (*function call*);
- **nessuna nozione di stato:** nessun assegnamento di variabili, più in generale nessuno *statement*, solo *espressioni*;
- **variabili:** parametri di funzioni o variabili locali contenenti valori costanti.

Terminologia:

- **higher order functions:** funzioni che possono accettare funzioni come argomenti e possono ritornare funzioni;
- **lambda expression/fuunction o anonymous functions:** funzioni ottenuto dall'evaluazione di un espressione.

Le funzioni sono valori di prima classe: un evaluazione di espressioni può dare come risultato un altra espressione (first class values).

Capitolo 10

OCaml

OCaml deriva da **ML**, è un linguaggio multi-paradigma con un approccio puramente funzionale, è staticamente tipato con inferenza di tipo:

- gli errori di tipo sono rilevati staticamente;
- i tipi possono essere omessi nei programmi.

Listing 10.1: Sintassi

```
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
   Exp BOP Exp | '(' Exp ')'
Pat ::= ID // pattern semplificato
```

Commenti:

- **ID** rappresenta gli identificatori di variabile $[a-zA-Z\_][\backslash w']*$;
- **NUM** rappresenta i numeri naturali:

$$0[bB][01][01\_]* | 0[oO][0-7][0-7\_]* | 0[xX][0-9a-fA-F][0-9a-fA-F\_]* | \backslash d[\backslash d\_]*$$

;

- **UOP** rappresenta operatori aritmetici unari $[+-]$;
- **BOP** rappresenta operatori aritmetici binari $[+ - *\]|mod$;
- **Pat** rappresenta i patterns, per semplicità si possono associare agli identificatori.

10.1 Funzioni ed applicazioni

- esempi di funzioni anonime:

Listing 10.2: Esempio di funzioni anonime

```
fun x -> x+1 (* funzione di incremento *)
fun x y -> x+y (* funzione di addizione *)
```

- applicazione di funzioni:

Listing 10.3: Esempio di applicazione di funzioni

```
(fun x -> x+1) 3 (* il risultato sara 4 *)
```

Inoltre prendendo in esempio:

Listing 10.4: Esempio di applicazioni

```
exp1 exp2
```

- l'evaluazione di **exp1** si aspetta come valore di ritorno una funzione f;
- l'evaluazione di **exp2** si aspetta come valore di ritorno un argomento valido a;
- l'evaluazione di **exp1 exp2** ritorna $f(a)$ (f applicato ad a).

10.2 Regole di precedenza ed associatività

- Si usano le regole standard per le espressioni aritmetiche;

- l'applicazione è associativa a sinistra:

Listing 10.5: Esempio di associatività

```
(fun x y -> x+y) 3 4 (* equivale a ((fun x y -> x+y) 3)
    4 *)
```

- l'applicazione ha precedenza maggiore rispetto gli operatori binari:

Listing 10.6: Esempio di precedenza

```
(fun x -> x*2) 1+2 (* equivale a ((fun x -> x*2) 1) +2
   *)
1+(fun x -> x*2) 1+2 (* equivale a 1+((fun x -> x*2) 1)
    +2 *)
```

- le funzioni anonime hanno piorità più bassa rispetto le applicazioni e gli operatori binari:

Listing 10.7: Esempio di precedenza

```
fun x -> x*2 (* equivale a fun x -> x*2) *)
fun f a -> f a (* equivale a fun f a -> (f a) *)
```

- i casi limite:

Listing 10.8: Casi limite di precedenza

```
f + 3 (* addizione *)
f (+3) (* applicazione *)
f - 3 (* sottrazione *)
f (-3) (* applicazione *)
+ f 3 (* equivale a +(f 3) *)
- f 3 (* equivale a -(f 3) *)
```

10.3 Una sessione REPL (Read Eval Print Loop)

Listing 10.9: I tipi possono essere inferiti dall'interprete

```
# 42;;
- : int = 42
# fun x->x*2;;
- : int -> int = <fun>
# (fun x->x+1) 2;;
- : int = 3
```

10.4 Sintassi

Listing 10.10: Grammatica BNF

```
Type ::= 'int' | Type '->' Type
```

10.4.1 Terminologia

- **int** è il tipo primitivo degli interi;
- **int** -> **int** è un tipo composito;
- **->** è un tipo *costruttore*, usato per costruire tipi compositi da tipi più semplici;
- i tipi costruiti con il costruttore freccia (->) sono chiamati *arrow types* o *tipi di funzione*.

Significato del tipo freccia $t_1 -> t_2$ identifica il tipo di funzioni da t_1 a t_2 che:

- più essere applicazo ad un singolo argomento di tipo t_1;
- ritorna sempre un valore di tipo t_2.

Osservazioni:

- il costruttore di tipo freccia è associativo a destra.

Listing 10.11: Associatività a destra dell'operatore freccia

```
int->int->int = int->(int->int)
```

- un tipo costruttore costruisce sempre un tipo diverso rispetto a quello dei propri componenti:

$$t_1\text{->}t_2 \neq t_1 \qquad t_1\text{->}t_2 \neq t_2$$

- due tipi freccia sono uguali se sono costruiti dallo stesso tipo di componenti:

$$t_1\text{->}t_2 = t_3\text{->}t_4 \text{ se e solo se } t_1 = t_3 \cap t_2 = t_4$$

- dai punti prima deduciamo:

```
int->int->int ≠ int->(int->int)
```

10.4.2 Funzioni di alto ordine

$$\textbf{fun}\ pat_1 pat_2 \ldots pat_n\text{->}exp$$

è un abbreviazione di:

$$\textbf{fun}\ pat_1\text{->}\textbf{fun}\ pat_2\text{->} \ldots \textbf{fun}\ pat_n\text{->}exp$$

10.5 Tuple

Listing 10.12: Nuove produzioni per Exp e Pat

```
Exp ::= '(' ')' | Exp ',' Exp
Pat ::= '(' ')' | '(' Pat (',' Pat)* ')'
```

Listing 10.13: Nuova produzione per Type

```
Type ::= 'unit' | Type '*' Type
```

10.5.1 Precedenza ed associatività

- l'operatore *tuple* ha precedenza più bassa rispetto gli altri operatori;
- l'operatore *tuple* non è associativo nè a destra nè a sinistra;
- il costruttore $*$ ha precedenza maggiore del costruttore $->$;
- il costruttore $*$ non è associativo nè a destra nè a sinistra.

Listing 10.14: Esempi di tuple

```
# ()
- : unit = ()

# 1,2,3
- : int * int * int = (1,2,3)

# (1,2),3
- : (int * int) * int = ((1,2),3)

# 1,(2,3)
- : int * (int * int) = (1,(2,3))

# fun() -> 3
- : unit -> int = <fun>

# fun ((x,y),z)->x*y*z
- : (int * int) * int -> int = <fun>

#fun (x. (y,z))->x*y*z
```

```
- : int * (int * int) -> int 0 <fun>
```

10.6 Funzioni curry

Teorema 19 *Una funzione curry (da* Haskell Curry*), è una funzione di alto ordine con un singolo argomento che ritorna una catena di funzioni con un singolo argomento.*

Una funzione non curry è una funzione con argomenti multipli.

Una funzione curry può essere trasformata in una funzione non curry e viceversa.

Listing 10.15: Esempi di funzioni curry

```
(* addizione di due interi *)
fun x y -> x+y;; (* versione curry int->int->int *)
fun (x y) -> x+y;; (* versione non curry int->int->int *)
(* moltiplicazione di tre interi *)
fun x y z -> x*y*z;; (* versione curry int->int->int->int *)
fun (x y z) -> x*y*z;; (* versione non curry int->int->int->int
    *)
```

10.6.1 Applicazione parziale

Le funzioni curry permettono l'applicazione parziale, ovvero gli argomenti possono essere passati uno alla volta.

Le funzioni non curry non permettono un applicazione parziale, tutti gli argomenti devono essere passai.

Listing 10.16: Esempi di applicazione parziale di funzioni curry

```
let curried_add x y=x+y;;
let uncurried_add(x,y)=x+y;;
(* computa 1+2 con la versione non curry *)
```

```
uncarried_add(1,2);;
(* computa 1+2 con l'applicazione parziale *)
let inc=curried_add 1;; (* passa l'argomento 1 e salva il
   risultato *)
inc 2;; (* passa l'argomento 2 e computa il risultato finale *)
```

L'applicazione parziale permette la specializzazione di funzioni: da una funzione generica è possibile generarne di più specifiche senza duplicazione di codice, quindi il riutilizzo e la mantenibilità sono favoriti.

10.7 Valori booleani

Per $BOOL = false|true$.

Listing 10.17: Sintassi

```
Exp ::= BOOL | 'not' Exp | Exp '&&' Exp | Exp '||' Exp
Type ::= 'bool'
```

10.7.1 Regole sintattiche standard

- && e || sono associativi a sinistra;
- **not** ha precedenza maggiore di &&;
- && ha precedenza maggiore di ||.

10.7.2 Semantica statica

- **false** e **true** sono di tipo *bool*;
- **not** e è di tipo *bool* se e solo se e è di tipo *bool*;
- il tipo di **not** e non è corretto se e non è di tipo *bool* oppure se il tipo di e non è corretto;

- $e_1 \&\& e_2$ e $e_1 || e_2$ sono di tipo *bool* se e solo se e_1 ed e_2 sono di tipo *bool*;
- il tipo di **not** $e_1 \&\& e_2$ e $e_1 || e_2$ non è corretto se e_1 o e_2 non sono di tipo *bool* oppure se il tipo di e_1 o e_2 non è corretto.

Inoltre:

- gli operatori && e || sono risolti sinistra a destra;
- se e_1 ritorna *false* allora $e_1 \&\& e_2$ ritorna *false*, altrimenti ritorna il valore di e_2;
- se e_1 ritorna *true* allora $e_1 || e_2$ ritorna *true*, altrimenti ritorna il valore di e_2.

10.7.3 Espressioni condizionali

le operazioni condizionali hanno precedenza minore di tutte le altre operazioni.

Listing 10.18: Espressioni condizionali

```
Exp ::= 'if' Exp 'then Exp 'else' Exp
```

10.7.4 Variabili globali

Listing 10.19: Grammatica delle variabili globali

```
Dec ::= 'let' Def (' and' Def)*
    | 'let' 'rec' FunDef (' and' FunDef)*
Def ::= Pat '=' Exp | FunDef
FunDef ::= ID Pat* '=' Exp
```

Esempio di variabili globali e funzioni curry Consideriamo i seguenti due esempi.

Listing 10.20: Addizione di quadrati

```
let rec sumsquare n = (* sumsquare si usa anche con associativo
    a destra*)
```

```
  if n<=0 then 0 else n*n+sumsquare(n-1);;
```

Listing 10.21: Addizione di cubi

```
let rec sumcube n = (* sumcube si usa anche con associativo a
   destra*)
  if n<=0 then 0 else n*n+sumcube(n-1);;
```

Si nota che sono quasi identici, quindi si può usare una funzione curry.

Listing 10.22: Soluzione con funzione curry

```
let rec gen_sum f n = (* (int -> int) -> int -> int *)
if n<=0 then 0 else f n+gen_su, f (n-1);;


let der_sumsquare = gen_sum (fun x->x*x);; (* int -> int *)
let der_sumcube = gen_sum (fun x->x*x*x);; (* int -> int *)
```

Notiamo che *gen_sum* può essere specializzato dato che è funzione curry ed il primo argomento è f piuttosto che n.

10.7.5 Dichiarazione di variabili locali

Listing 10.23: Sintassi delle variabili locali

```
Dec ::= 'let' Def (' and' Def)* 'in' Exp
    | 'let' 'rec' FunDef (' and' FunDef)* 'in' Exp
Def ::= Pat '=' Exp | FunDef
FunDef ::= ID Pat* '=' Exp
```

Listing 10.24: Esempio di variabili locali

```
let f x=x+1 and v=41 in f v;; (* f e v possono essere usati
   solo qui *)
- : int = 42
```

```
let x=1 in let x=x*2 in x*x (* dichiarazioni annidate *)
- : int = 4
```

Da notare che le dichiarazioni annidate sovrascrivono le dichiarazioni con lo stessi ID.

10.7.6 Scopo delle dichiarazioni statiche

Listing 10.25: Esempio di dicharazione statica

```
let v=40;;

let f x = x*v;; (* v riferisce alla dichiarazione precedente *)

f 3;; (* ritorna 120 *)

let v=4;; (* dichiarazione di v sovrascritta *)

f 3;; (* ritorna 120 *)
```

Listing 10.26: Miglioramento dell'esempio della somma di quadrati e cubi

```
let gen_sum f = (* (int ->) -> int -> int *)
    let rec aux n = if n<=0 then 0 else f n+aux (n-1) (* int ->
        int )
    in aux;;
```

Non dobbiamo passare l'argomento f alla funzione ricorsiva aux.

10.8 Liste

Listing 10.27: Sintassi delle liste

```
Exp ::= '[' ']' | Exp '::'Exp
```

- la lista vuota è rappresentata da [];
- $hd :: ts$ è la lista con la testa (hd) e la coda (tl);
- $[] \neq t_1 :: t_2$ e $t_1 \neq t_1 :: t_2$ e $t_2 \neq t_1 :: t_2$;
- $t_1 :: t_2 = t_1' :: t_2'$ se e solo se $t_1 = t_1'$ e $t_2 = t_2'$;
- $[e_1; e_2; \ldots; e_n]$ è l'abbreviazione per $e_1 :: e_2 :: \cdots :: e_n$.

10.8.1 Regole sintattiche

- Associatività a destra;
- minore precedenza degli operatori unari e binari con notazione infissa;
- maggiore precedenza de:
  - il costruttore di tupla;
  - espressioni di funzioni anonime (**fun** $\cdots ->$ ...);
  - espressioni condizionali (**if** ... **then** ... **else** ...);
- si può usare la notazione $[e_1; e_2; \ldots e_n]$ come abbreviazione.

Attenzione L'operatore ; all'interno delle quadre ha le proprie regole di precedenza, per esempio l'operatore di tupla ha maggiore precedenza.

10.8.2 Tipi di costruttori per le liste

Le liste devono essere omogenee, tutti gli elementi devono essere dello stesso tipo.

Il costruttore unario postfisso è *list*, ha precedenza maggiore dei costruttori $->$ e $*$.

Listing 10.28: Esempi di liste

```
[1;2] (* una lista di interi *)
- : int list = [1;2]

[true;false;true] (* una lista di booleani *)
- : bool list = [true;false;true]

[1,true] (* una lista di coppie int*bool *)
- : (int * bool) list = [(1, true)]

[[1;2];[0;3;3];[]] (* una lista di liste di interi *)
- : int list list = [[1;2];[0;3;3];[]]
```

Semantica statica

Nella sintassi concreta di *OCaml*, [] è di tipo α list o $'$a *list*.

$e_1 :: e_2$ ha tipo t *list* se e solo se e_1 ha tipo t ed e_2 ha tipo t *list*.

$e_1 :: e_2$ non è di tipo corretto se non esiste tipo t.

Polimorfismo

Il tipo α list si dice di tipo polimordico o di scema, dato che α è di tipo variabile.

10.8.3 Concatenazione

L'operatore binario infisso è associativo a sinistra ed ha minore precedenza del costruttore ::; la concatenazione **non** è un costruttore.

Listing 10.29: Operatore binario infisso di lista

```
Exp ::= Exp '@' Exp
```

Semantica statica

Una lista concatenata $e_1@e_2$ sarà di tipo lista se e solo se entrambi gli e sono liste, altrimenti non sarà di tipo corretto.

10.8.4 Esempi

Listing 10.30: Esempi di liste

```
[1;2]@[3]@[4;5;6];;
- : int list 0 [1;2;3;4;5;6]

[[1]]@[2]::[[3]];;
- : int list list = [[1]; [2]; [3]]

([1]@[2])::[[3]];;
- : int list list = [[1; 2]; [3]]

(@)
- : 'a list -> -> 'a list = <fun>

(@) [1;2]
- : int list -> int list = <fun>

(@) [1;2] [3;4;5]
- : int list = [1; 2; 3; 4; 5]
```

10.9 Pattern matching

Le funzioni che non possono essere definite con un singolo pattern sono:

- la lunghezza di una lista;
- la somma di tutti gli elementi di una lista;
- la lista con i primi due elementi scambiati.

Listing 10.31: Nuove produzioni per Pat

```
Pat ::= '[' ']' | Pat '::' Pat | '[' Pat (';' Pat)* ']'
```

Osservazione Tutte le variabili in un pattern devono essere distinte (questo rende il controllo sui pattern più efficiente). Inoltre i *patterns* sono costruiti con costruttori, non con altri operatori: $x :: y$ è un pattern valido, $x@y$ o $x+y$ non lo sono; i costruttori garantiscono un unica decomposizione dei valori.

10.9.1 Esempi di pattern matching

Listing 10.32: Primo esempio di pattern matching

```
let add (x,y) = x+y;;
add (r,5);;
```

- $(3, 5)$ combacia con il pattern (x, y) se e solo se $x = 3 \cap y = 5$;
- se sostituiamo x, y in (x, y) con 3 e 5, rispettivamente, allora otteniamo il valore $(3, 5)$.

Listing 10.33: Secondo esempio di pattern matching

```
let hd (h::t) = h;; (* ritorna la testa della lista *)
hd [3;5];;
```

- $[3;5]$ combacia con il pattern $(h :: t)$ con 3 e $[5]$ rispettivamente, quindi otteniamo il valore $(3 :: [5]) = [3;5]$.

Listing 10.34: Terzo esempio di pattern matching

```
let hd (h::t) = h;;


hd[];;
```

- $[]$ non combacia con $(h :: t)$ per qualunque valore associato a h e t;
- quindi $[] \neq (h :: t)$ per tutti i possibili valori associati con h e t;
- il comportamento è corretto, dato che la testa di una lista non è definita per la lista vuota.

10.9.2 Matching di patterns multipli

Listing 10.35: Espressione per eseguire un match con multipli patterns

```
Exp ::= 'match' Exo 'with' Pat '->' Exp ('|' Pat '->' Exp)*
```

Esempi

Listing 10.36: Esempio di match multipli

```
let rec length 1 = match 1 with
    [] -> 0
  | hd::tl -> 1+length t1;;


let rec sum 1 = match 1 with
    [] -> 0
  | hd::tl -> hd+sum tl;;
```

```
let swap 1 = match 1 with
    [] -> []
  | [x] -> [x]
  | x::y::1 -> y::x::1;;
```

Sintassi

Listing 10.37: Sintassi di matching di patterns multipli

```
match e with p1 -> e1 | ... pn -> en
```

Semantica statica L'espressione e e tutti i patterns $p_1 \dots p_n$ devono essere dello stesso tipo, così come tutte le espressioni $e_1 \dots e_n$.

Viene riportato un warning se i patterns non sono esaustivi, per esempio se sono mancanti, oppure se un pattern non è utilizzato.

Semantica dinamica In ordine vengono calcolati:

1. e;
2. tutti i pattern $p_1 \dots p_n$ testati da sinistra a destra, dalla cima in fondo;
3. al primo *match* con p_i, l'espressione e_i è calcolata, con le variabili definite dal match con p_i;
4. se non viene trovato un match, allora l'errore *Match_failure* è sollevato.

10.9.3 Decomposizione unica

I costruttori assicurano che se esiste un march per p, allora esiste un unica sostituzione per le variabili in p.

10.9.4 Costruttori per i tipi primitivi

Tutti i litterali (tokens che rappresentano valori) sono costruttori costanti.

10.9.5 Notazione abbreviata

- la carattere *wildcard* _ è il pattern che matcha tutti i valori quando nessuna variabile è necessaria;
- **function** $p_1 -> e_1 | \ldots | p_n -> e_n$ abbrevia la notazione:

  `fun var -> match var with` p_1 `->` e_1 `|` ... `|` $p_n e_n$

- p **as** *id*: un pattern (o sotto-pattern) p può essere associato con un *id* per fare riferimento al valore trovato direttamente.

10.9.6 Esempi di pattern matching funzionanti

Listing 10.38: Esempi di pattern matching funzionanti

```
let mynot = function false -> true | _ -> false;;

let iszero = function 0 -> true | _ -> false;;

let rec length = function _::tl -> \+length tl | _ -> 0;;

let rec sum = function hd::tl -> hd+sum tl | _ -> 0;;

let swap = function x::y::1 -> y::x::1 | other -> other;;

let ord_swap = function
    x::Y::tl as 1 -> if x>y then y::x::tl else 1
  |other -> other;;
```

10.10 Ricorsione ed efficienza

Listing 10.39: Primo esempio di ricorsione

```
let rec sum = function
```

```
    hd::tl -> hd + sum tl (* caso induttivo *)
  | _ -> 0;; (* caso base *)

let l=List.init 100_000 (fun x->x+1) (* crea una lista da 1 a
   10000 *)
in sum l;;
- : int = 5000050000

let l=List.init 1_000_000 (fun x->x+1)
in sum l;;
Stack overflow during evaluation.
```

Listing 10.40: Secondo esempio di ricorsione

```
let rec reverse = function
    hd::tl -> reverse tl @ [hd]
  | _ -> [];;

let l=List.init 10_000 (fun x->x+1)
in reverse l;;
```

Il primo esempio con *tl @ [hd]* ha una complessita lineare rispetto ad l, *reverse* l invece ha una complessità quadratica.

10.10.1 Modulo List

Il modulo *List* è un modulo di OCaml predefinito, *List.init* è una funzione definita in *List*.

10.11 Accumulatori

In OCaml il metodo utilizzato per effettuare dei cicli sono attraverso l'utilizzo degli accumulatori.

Listing 10.41: Accumulatori

```
let rec loop acc l = match l with (* loop : int -> int list ->
   int *)
(* if l=hd::tl allora incremento acc di hd e provo al giro
   successivo tl *)
    hd::tl -> loop (acc+hd) tl
(* se l=[] ritorno acc *)
  | _ -> acc
(* loop chiamato con il valore iniziale di acc *)
in loop 0;; (* loop 0 : int list -> int *)
```

10.12 Ricorsione in coda

Nella ricorsione in coda, l'applicazione viene sempre eseguita per ultima, può essere implementata con un vero e proprio loop.

Listing 10.42: Ricorsione in coda

```
let rec loop acc = function
    hd::tl -> loop (acc+hd) tl
  | _ -> acc
in loop 0;;
```

10.12.1 Ricorsione in coda con accumulatori

Listing 10.43: Esempio di ricorsione in coda con accumulatori

```
let acc_rev l = (* parametro l necessario per una funzione
   polimorfica *)
  let rec aux acc = function
      hd::tl -> aux (hd::acc) tl
```

```
    | _ -> acc
  in aux [] l;;

let l=List.init 10_000 (fun x->x+1)
in acc_rev l;;
```

10.13 Funzioni polimorfiche

Listing 10.44: Esempio di funzione polimorfica

```
let acc_rev l =
  let rec aux acc = function
      hd::tl -> aux (hd::acc) tl
    | _ -> acc
  in aux [] 1;;

acc_rev [1;2;3];;
- : int list = [3;2;1]

acc_rev [true;true;false];;
- : bool list = [false; true; true]
```

10.14 Stringhe

Il tipo primitivo di *string* è supportato, il costruttore è "", la concatenazione $\wedge$ è associativa a sinistra; è un modulo predefinito.

10.15 Funzioni generiche in List

10.15.1 map

Si usa per applicare una funzione agli elementi di una lista.

Listing 10.45: Definizione di map con ricorsione in coda

```
let map f l=
  let rec aux acc = function
      hd::tl -> aux (f hd::acc) tl
    | _ -> acc
  in aux [] (List.rev l);;
```

10.15.2 fold_left

Raprresentazione di un patter generico di liste con accumulatori:

$$f \quad a_0 \quad [x_1; \ldots; x_n] = a_n$$

Dove a_0 è il valore iniziale dell'accumulatore e $a_n = f a_{n-1} x_n$.

Listing 10.46: Definizione di fold_left con ricorsione in coda

```
let fold\_left f =
  let rec aux acc = function
      hd::tl -> aux (f acc hd) tl
    | \_ -> acc
  in aux;;
```

Listing 10.47: Esempio di utilizzo di fold_left

```
let square\_list = fold\_left (fun acc hd -> acc+hd+hd) 0
val square\_list : int list -> int = <fun>

square list [1;2;3;4]
- : int = 30
```

10.16 Eccezioni

La gestione di problemi e comportamenti non voluti deve essere eseguita il prima possibile, al momento giusto e, un crash del software deve essere evitato se possibile.

10.16.1 Benefici

Una chiara separazione del comportamento normale e non:

- **esecuzione normale ed anormale:** l'esecuzione normale del programma deve essere interrotta non appena un errore viene rilevato;
- **valori ed eccezioni:** i valori sono tornati solo se una computazione si risolve normalmente, un eccezione viene lanciata, senza ritorno di valori, se la computazione non si risolve.

10.16.2 Costruttori e sintassi

Le eccezioni hanno tipo *exn* e sono create con costruttori, per generarle e propagarle si può usare la funzione predefinita:

Listing 10.48: Funzione predefinita la propagazione degli errori

```
raise : exn -> 'a
```

La gestione delle eccezioni invece:

Listing 10.49: Gestione delle eccezioni

```
try e with p1 -> e1 | ... | pn -> en
```

raise non ritorna nessun valore, il tipo *'a* può quindi essere utilizzato in qualsiasi contesto.

Listing 10.50: Dichiarazione dei costruttori di eccezioni, sintassi

```
Dec ::= 'exception' CONS_ID ('of' Type)?
```

CONS_ID deve cominciare con una lettera maiuscola.

Listing 10.51: Esempi di eccezioni

```
exception Fault;; (* costante *)
exception Fault1 of string;; (* unario *)
exception Fault2 of string*exn;; (* binario *)
let exc=Fault;;
let exc1=Fault1 "error message";;
let exc2=Fault2 ("msg",exc);;
```

Listing 10.52: Eccezioni predefinite e funzioni

```
exception Division_by_zero;;

exception Failure of string;;

exception Invalid_argument;;

exception Match_failure of string+int+int;;

int failwith msg = raise (Failure.msg);;
```

10.17 Tipi varianti

Permettono all'utente di definire nuovi tipi con dei costruttori.

Listing 10.53: Esempio di tipi varianti

```
type color = Red | Green | Blue;;

let to_string = function
    Red -> "red"
  | Green -> "green"
```

```
  | Blue -> "blue";;

List.map to_string [Red;Blue;Green;Blue];;
```

Listing 10.54: Esempio con costruttori di arità maggiore di 0

```
type shape = Square of float | Circle of float
    | Rectangle of float * float;;

let perimeter = function
    Square side -> 4.0 *. side
  | Circle ray -> 2.0 *. Float,pi *. ray
  | Rectangle (width,height) -> 2.0 *. (width *. height);;

perimeter (Square 4.0);;
- : float = 16
```

Listing 10.55: Esempio di costruttore con ricorsione

```
type exp_ast = NumLit of int | Sign of exp_ast
    | Mul of exp_ast * exp_ast | Add of exp_asd * exp_ast;;

let rec eval = function
    NumLit n -> n
  | Sign t -> - eval t
  | Mul (t1,t2) -> (eval tl) * eval t2
  | Add (t1,t2) -> (eval tl) + eval t2;;

let ast = Sign(Add(NumLit 40, NumLit 2));;
eval ast;;
- : int = -42
```

Listing 10.56: Esempio di costruttore polimorfico

```
type 'a option = None | Some of 'a;;
let get = function (* get : 'a option 'a *)
    Some v -> v
  | _ -> raise (Invalid_argument "get");;

let find p = (* find : ('a -> bool) -> 'a list -> 'a option *)
  let rec aux = function
      hd::tl -> if p hd then Some hd else aux tl
    | _ -> None
  in aux;;

let v=find ((<) 0) [-1;-2;-3];;
val v : int option = Some 3

get v;;
- : int = 3

let v=find ((<) 0) [-1;-2;-3];;
val v : int option = None
get v;;
Exception: Invalid_argument "get".
```

Listing 10.57: Esempio con polimorfismo e ricorsione di costruttore

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;

let rec member el = function
    Node(label,left,right) ->
        el=label || el<label && member el left || member el
           right
  | _ -> false;;
```

```
let rec insert el = function
    Node(label,left,right) as t ->
        if el=label then t
        else if el < label then Node(label,insert el left,right
           )
        else Node(label, left, insert el right)
  | _ -> Node(el,Empty,Empty);;
```

10.18 Numeri a virgola mobile

Il tipo predefinito è **float**, hanno costruttori costanti e gli operatori sono gli stessi dei numeri interi, ma seguiti da un punto. Le variabili globali sono:

- nan;
- infinity;
- neg_infinity.

Altre funzionalità si trovano nei moduli *Stdlib* e *Float*.

Capitolo 11

Programmazione orientata agli oggetti

Il modello su cui si basa il paradigma è l'invio di messaggi attraverso gli oggetti, quando si lancia un programma si invia un messaggio ad un oggetto.

11.1 Oggetti

Un oggetto è identificato da un nome, mantiene degli stati interni che sono solitamente nascosti ed espone i cosiddetti *metodi di istanza*, l' interfaccia con cui si può interagire con l'oggetto.

L'invocazione di un metodo di istanza implica che:

- gli stati interni dell'oggetto possono venire modificati;
- può essere necessario passare degli argomenti alle funzioni;
- la funzione può ritornare un valore.

Listing 11.1: Sintassi di un oggetto

```
Exp ::= Exp '.' MID '(' (Exp ( ',' Exp)*)? ')'
```

Dove *MID* è il nome del metodo di istanza.

I metodi di istanza possono essere di ispezione, di modifica, o entrambi.

11.1.1 Stato interno

Solitamente lo stato interno di un oggetto non è esposto, consiste di *campi*: solitamente chiamati *variabili di istanza* o *attributi*, i quali salvano i dati dell'oggetto, sono solitamente modificabili.

11.2 Classi

Una classe fornisce un implementazione per gli oggetti dello stesso tipo, gli oggetti possono essere creati dinamicamente dalle classi, ogni oggetto creato da una classe C sono detti *istanza* di C, il numero di istanze di un oggetto a runtime può crescere o diminuire, perchè possono essere deallocate automaticamente o manualmente.

Tutte le istanze condividono gli stessi metodi di istanza, ma ogni istanza ha i propri stati interni.

Oggetti creati da una classe C hanno *tipo dinamico* C, in un linguaggio staticamente tipato si dice anche *tipo statico*.

Listing 11.2: Un esempio di classe in Java

```
public class TimerClass {
  private int time; // variabile di istanza

  // metodo di istanza
  public boolean isRunning() {
    return this.time > 0;
  }

  public int getTime() {
    return this.time;
  }
```

```
  public void tick() {
    if (this.time > 0)
      this.time--;
  }

  public int reset (int minutes) {
    if (minutes < 0 || minutes > 60)
      throw new IllegalArgumentException();
    int prevTime = this.time;
    this.time = minutes * 60;
    return prevTime;
  }
}
```

Listing 11.3: Utilizzo della classe

```
// Creo un nuovo oggetto e lo assegno a t1
TimerClass t1 = new TimerClass();
// Chiamo il metodo di istanza reset
t1.reset(1);
// Creo un nuovo oggetto e lo assegno a t2
TimerClass t2 = new TimerClass();
// Chiamo il metodo di istanza reset
t2.reset(2)
```

11.2.1 Parola chiave: this

La parola chiave **this** fa riferimento all'oggetto su cui vengono chiamati i metodi.

Listing 11.4: Esempio di utilizzo di this

```
public int reset(int minutes) {
```

```
  if (minutes < 0 || minutes > 60)
    throw new IllegalArgumentException();
  int prevTime = this.time;
  this.time = minutes * 60;
  return prevTime;
}
```

11.2.2 Information Hiding

La visibilità di metodi o campi di un oggetto può essere definita:

- **private** il metodo o il campo è visibile solo all'interno della classe;
- **public** il metodo o il campo è visibile fuori dalla classe;
- **public class** la dichiarazione della classe è visibile in tutto il programma.

11.2.3 Eccezioni

La dichiarazione **throw** si utilizza quando deve essere segnalato un errore, la sintassi è $'throw' Exp$, l'argomento di throw deve essere un eccezione, la quale è un oggetto particolare.

Listing 11.5: Esempio di eccezione in Java

```
Throwable ex;
...
throw 42; // errore
throw new NullPointerException();
throw ex;
```

11.2.4 Asserzioni

La sintassi delle asserzioni è $'assert' Exp$, dove Exp deve essere un booleano, si usano per documentare, testare e validare gli oggetti.

Listing 11.6: Esempio di asserzione in Java

```
TimerClass tl = new TimerClass();
t1.reset(1);
int seconds = 0;
while (t1.isRunning()) {
  t2.tick();
  seconds++;
}
assert seconds == 60;
assert !t1.isRunning();
```

11.2.5 Oggetti come valori

Nei linguaggi di programmazione orientati agli oggetti, gli oggetti sono valori di primo ordine, ovvero:

- possono essere passati a variabili;
- possono essere passati come argomenti;
- possono essere ritornati come valori.

Gli oggetti sono rappresentati dalla loro **identità**, ovvero l'indirizzo di memoria dello heap in cui l'oggetto è salvato. Motivo per cui gli oggetti sono passati per referenza.

Listing 11.7: Esempio di oggetti come valori

```
TimerClass t1 = new TimerClass();
TimerClass t2 = t1;
TimerClass t3 = null;
assert t1 == t2 && t1 != t3; // true
```

11.2.6 Tipi e sottotipi

Le classi definiscono anche tipi statici, chiamati in Java *reference type* ed in altri contesti *object type*.

Ciò significa che se un espressione e è di tipo statico *TimerClass*, allora e potraà tornare o un istanza di *TimerClass*, o un istanza di sottotipo di *TimerClass* o *null*.

11.2.7 Design by contract

Pre-condizioni, post-condizioni e invarianti su di un metodo:

- **requires p**: è necessario che il predicato p sia valido prima dell'esecuzione del metodo;
- **ensures p**: è necessario che il predicato p sia valido dopo che il metodo è stato eseguito;
- **invariant per la classe C**: è un predicato che deve essere valido alla creazione di ogni istanza di C, e prima e dopo l'esecuzione di ogni metodo di C.

Listing 11.8: Esempio di Design By Contract

```
public class TimerClass {
  private int time;
  /* invariant 0 <= time && time <= 3600; */

  public int reset(int minutes)
  /*  requires 0 <= minutes && minutes <= 60;
      ensures  result == old(this.time)
               && this.time == minutes * 60; */
  {
    if (minutes < 0 || minutes > 60)
      throw new IllegalArgumentException();
    int prevTime = this.time;
```

```
    this.time = minutes * 60;
    return prevTime;
  }

  public boolean isRunning()
  /* ensures result == this.time > 0
             && this.time == old(this.time); */
  {
    return this.time > 0;
  }
  ...
}
```

Per assicurare il funzionamento del **DBC**, si utilizza l' *information hiding* (lo stato dell'oggetto è modificabile solo tramite l' utilizzo di metodi), la presenza di *invariants* in tutti i metodi.

11.3 Costruttori

I costruttori in Java possono essere *overloaded*.

Listing 11.9: Esempio di costruttori Java

```
public class TimerClass (
  private int time;

  public TimerClass() {
  }

  public TimerClass(int minutes) {
    if (minutes < 0 || minutes > 60)
      throw new IllegalArgumentException();
    this.time = minutes * 60;
```

```
  }

  public TimerClass(TimerClass otherTimer) {
    this.time = otherTimer.time;
  }
...
}
```

Listing 11.10: Esempio di uso dei costruttori

```
TimerClass t1 = new TimerClass();
TimerClass t2 = new TimerClass(42);
TimerClass t3 = new TimerClass(t2);

assert t1.getTime()==0 && t2.getTime()==42*60 &&
       t2.getTime() == t2.getTime();
```

11.4 Creazione ed inizializzazione degli oggetti

Immediatamente dopo la creazione di un oggetto dei valori di default vengono assegnati ad ogni variabile di istanza dell'oggetto. Il valore di default è determinato dal tipo dichiarato per quella variabile, se presente un inizializzatore di variabile di istanza invece viene eseguito.

Listing 11.11: Esempio di iinizializzatore di variabile in Java

```
public class TimerClass {
  private int time = 60;

  public TimerClass() {
  }

  public TimerClass(int minutes) {
```

```
    if (minutes < 0 || minutes > 60)
      throw new IllegalArgumentException();
    this.time = minutes * 60;
  }
  public TimerClass(TimerClass otherTimer) {
    this.time = otherTimer.time;
  }
...
}
```

Listing 11.12: Esempio di creazione ed inizializzazione di oggetti in Java

```
TimerClass timer1 = new TimerClass();
TimerClass timer2 = new TimerClass(1);

assert timer1.getTime() == timer2.getTime();
```

11.4.1 Costruttori

Multipli costruttori possono essere definiti, gli stessi devono differire nel numero o nel tipo di parametri, un costruttore di default è definito se nessun costruttore è presente, non ha parametri ed è vuoto.

Un costruttore può essere invocato esplicitamente in un altro costruttore, l'invocazione può essere eseguita **solo nella prima linea**.

'this' '(' (Exp (',' Exp))?')'*

Da notare che i campi degli oggetti non possono essere aggiunti o rimossi a *runtime*, i campi non opzionali devono avere sempre un valore definito mentre i campi opzionali possono avere un valore non definito, in Java per valore indefinito si intende **null**.

Listing 11.13: (1) Esempio di invocazione dei costruttori esplicita

```
public class Person {
  private String name;
```

```
  private String address;

  public Person(String name) {
    if(name == null)
      throw new NullPointerException();
    this.name = name;
  }

  public Person (String name, String address) {
    this(name);
    this.address = address;
  }

  public String getName() {
    return name;
  }

  public String getAddress() {
    return address;
  }
}
```

Listing 11.14: (2) Esempio di invocazione dei costruttori esplicita

```
Person sam = new Person("Samuele");
Person sim = new Person("Simone","Genova");
assert sam.getAddress()=null && sim.getAddress()!=null;
```

11.5 Variabili di classe

In java ci sono le variabili di istanza, che sono gli attributi degli oggetti e le variabili di classe che sono gli attrivuti della classe. La sintassi (*CID*:class identifier, *FID*:field

identifier):

- **field read:** *CID *.*FID*
- **field update:** *CID '.'FID '='Exp*

Listing 11.15: (1) Esempio di variabili di classe

```
public class Item {
  private static long nextSN;
  private int price;
  private long SerialNumber;

  public Item(int price) {
    if(price < 0)
      throw new IllegalArgumentException();
    this.price = price;
    this.serialNumber = Item.nextSN++;
  }

  public int getPrice() {
    return this.price;
  }

  public long getSerialNumber() {
    return this.SerialNumber;
  }
}
```

Listing 11.16: (1) Esempio di variabili di classe

```
Item item1 = new Item(61_50);
Item item2 = new Item(14_00);
```

```
assert item1.getPrice() == 61_50 && item1.getSerialNumber() ==
   0;
assert item2.getPrice() == 14_00 && item2.getSerialNumber() ==
   0;
```

11.5.1 Inizializzazione

Una variabile di classe è inizializzata dopo che la classe è stata caricata e linkata, vi è assegnato un valore di default, così come per le variabili di istanza; gli inizializzatori sono eseguiti in ordine testuale.

11.6 Metodi di classe

Mentre i metodi di istanza sono invocati su di un oggetto (**this**), i metodi di classe sono invocati sulle classi.

La sintassi: *CID '.' MID '(' (Exp (',' Exp)*)? ')'*

Listing 11.17: Esempio di metodi di classe in Java

```
public class Rectangle {
  private static int defaultSize = 1; //inizializzatore di
     variabile di classe
  private int width = Rectangle.defaultSize;
  private int height = Rectangle.defaultSize;
  // invariand width > 0 && height > 0
  private static void checkSize(int size) {
    if (size <= 0)
      throw new IllegalArgumentException();
  }

  public Rectangle(int width, int height) {
    Rectangle.checkSize(width);
```

```
    Rectangle.checkSize(height);
    this.width = width;
    this.height = height;
  }

  // static factory method
  public static Rectangle ofWidthHeight(int width, int height)
     {
    return new Rectangle(width, height);
  }
  ...
  ...
  Rectangle r1 = new Rectangle(3,5);
  Rectangle r2 = Rectangle.ofWidthHeight(3,5);
```

11.7 Classe Object

Object è una classe predefinita speciale: ogni altra classe è *sottotipo* di *Object*.

Se un espressione ha tipo statico Object, allora risulta in:

- un istanza di una sottoclasse di Object (qualsiasi classe);
- **null**;
- un *array*.

11.7.1 Subtyping

I sottotipi definiscono una relazione tra tipi in maniera gerarchica.

Ogni tipo oggetto è sottotipo di *Object*, ma l'oggetto ed i suoi primitivi non sono comparabili.

Le proprietà che variano a seconda della struttura gerarchica sono:

- **riflessiva:** $T \leq T$;

- **antisimmetrica:** $T_1 \leq T_2 \cap T_2 \leq T_1 \rightarrow T_1 = T_2$;
- **transitiva:** $T_1 \leq T_2 \cap T_2 \leq T_3 \rightarrow T_1 \leq T_3$;

Se è richiesto un tipo T, qualsiasi sottotipo di T è valido.
Gli oggetti hanno due tipi diversi di uguaglianza:

- **forte:** *person1 == person2* se riferiscono alla stessa persona;
- **debole:** *person1.equals(person2)* compara gli attributi, quindi possono essere uguali, ma essere due oggetti diversi.

11.8 Stringhe

Le stringhe sono oggetti immutabili, ovvero che le variabili di istanza non possono essere cambiate dopo l'inizializzazione.

Listing 11.18: Esempio di ereditarietà in Java

```
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public Point(Point p) {
    this(p.x, p.y);
  }
  public int getX() {
    return this.x;
  }
  public int getY() {
    return this.y;
```

```
  }
  public void move(int dx, dy) {
    this.x += dx;
    this.y += dy;
  }
  public boolean overlaps(Point p) {
    return this.x == p.x && this.y == p.y;
  }
}

public class Line {
  private Point a;
  private Point b;

  public Line(Point a, Point b) {
    if(a.overlaps(b))
      throw new IllegalArgumentException();
    this.a = a;
    this.b = b;
  }
  public void move(int dx, int dy) {
    this.a.move(dx, dy);
    this.b.move(dx, dy);
  }
  public boolean overlaps(Line l) {
    return this.a.overlaps(l.a) && this.b.overlaps(l.b)
      || this.a.overlaps(l.b) && this.b.overlaps(l.a);
  }
}
```

Da notare che i *Point* che vengono utilizzati in questo esempio da *Line*, sono condivisi tra varie istanze di *Line*, per cui se si vogliono creare linee che non dipendono

tra loro, bisognerà specificare la possessione dei punti tramite un **private**.

11.9 Shallow e deep copy

Listing 11.19: Shallow copy in Java

```
public Line(line l) {
  this.a = l.a;
  this.b = l.b;
}
```

Listing 11.20: Deep copy in Java

```
public Line(line l) {
  this.a = new Point(l.a);
  this.b = new Point(l.b);
}
```

11.10 Variabili final

Le variabili di istanza, di classe o locali possono essere dichiarate di tipo **final**, in sola lettura.

Da notare che se **final** è applicato ad una variabile di tipo oggetto, nonostante si farà riferimento allo stesso oggetto, lo stesso potrà essere modificato.

11.10.1 Oggetti immutabili

Un oggetto si dice immutabile se tutte le variabili di istanza sono **final** ed ogni variabile di istanza contiene o un valore primitivo, o un oggetto immutabile.

11.11 Interfacce

Per effettuare operazioni tra oggetti diversi si possono adoperare le **interfacce**, vengono definiti dei *supertipi* per associare più oggetti tra loro.

Una classe può implementare più interfacce, tutti i metodi di un interfaccia sono implicitamente **public** e **abstract** (senza corpo).

Listing 11.21: Esempio di interfaccia in OCaml

```
type shape = Square of float | Circle of float
  | Rectangle of float * float;;

let perimeter = function
    Square side -> 4.0 *. side
  | Circle ray -> 2.0 *. (width +. height)
  | Rectangle (width,height) -> 2.0 *. (width +. height);;

let area = function
    Square side -> side *. side
  | Circle ray -> Float.pi *. ray *. ray
  | Rectangle (width,height) -> width *. height;;
```

Listing 11.22: Esempio di interfaccia in Java

```
public interface Shape {
  double perimeter();
  double area();
}

public class Square implements Shape {
  private double side;
  ...
  public double perimeter() {return 4 * this.side; }
```

```
  public double area() { return this.side * this.side; }
}

public class Rectangle implements Shape {
  private double width;
  private double height;
  ...
  public double perimeter() {return 2 * (this.width + this.
     height); }
  public double area() { return this.width * this.height; }
}

public class Circle implements Shape {
  private double radius;
  ...
  public double perimeter() {return 2 * Math.PI * this.radius;
     }
  public double area() { return Math.PI * this.radius^2; }
}
```

Gli approcci sono diversi tra linguaggi funzionali ed ad oggetti, i primi strutturano il codice dividendolo per operazione ed usano il pattern matching per gestire i vari tipi di dato, nei linguaggi ad oggetti invece il codice identifica i dati per tipo ed ogni classe implementa le operazioni.

11.12 Array in Java

Gli array in Java sono oggetti modificabili speciali, hanno una componente chiamata *indice*, la quale è una variabile di istanza senza nome.

Gli array possono essere creati solo dinamicamente, alla loro crazione la lunghezza deve essere specificata e la stessa non può cambiare a run time, difatti *length* è di tipo *final*.

I componenti sono inizializzati con valori di default e sono referenziati con indici da 0 a $length - 1$; $T[]$ è il tipo di array con tipo di componente T.

Listing 11.23: Esempio di inizializzazione di array in Java

```
public class ArrayUtils {
  public static int sum(int[] a) {
    int sum = 0;
    for (int el : a)
      sum += el;
    return sum;
  }
}

int [] a = {1, 2, 3};
assert ArrayUtils.sum(a) == 6;
assert ArrayUtils.sum(new int[] {1,2,3,4}) == 10;
```

Listing 11.24: Esempio di array multidimensionali in Java

```
int [][] mat1 = new int[3][2];
assert mat1.length == 3;
for (int[] row : mat1) {
  assert row.length == 2;
  for (int el : row)
    assert el == 0;
```

```
}


int[][] mat2 = new int[3][];
assert mat2.length == 3;
for (int [] row : mat2)
  assert row == null;


int[][] mat3 = { (1,1}, {1,2,1}, {1,3,3,1} };
```

11.13 Main methods in Java

L'esecuzione di un programma Java può cominciare solamente da una classe con un metodo main, della forma:

public static void main(String[] args){...}

11.14 Standard output in Java

Lo standard output si ottiene dalla chiamata a *System.out*, *System* è una classe predefinita (di tipo *String*), *System.out* è una variabile di classe **final** ed è di tipo *PrintStream*, la quale è una classe predefinita della libreria Java.

11.15 Modularità su larga scala

In Java si possono utilizzare moudli o pacchetti e sottopacchetti per contenere ed esportare logicamente classi.

11.16 Packages

Le classi pubbliche definite nei pacchetti possono essere accedute dall' esterno, mentre le classi private no. Sottopacchetti possono essere contenuti nei pacchetti, in una struttura gerarchica e sono definite diverse **unità di compilazione**.

Un unità di compilazione è composta da 3 parti:

- dichiarazione del **package**, se non presente l'unità sarà parte di un pacchetto senza nome;
- dichiarazioni di **import**;
- dichiarazioni di classi di primo livello.

Listing 11.25: Esempio di un unità di compilazione in Java

```
package shapes;

import java.awt.Color;

class Point {
...
}

public class ColoredLine {
  private Point a;
  private Point b;
  private Color color = Color.BLACK;
  ...
  public COlor getColor() { return this.color; }
  public void setColor ( Color color ) { this.color = color; }
)
```

11.16.1 Pacchetti e sottopacchetti

I pacchetti hanno namespaces gerarchici, dichiarazioni nello stesso pacchetto devono avere nomi diversi, ma in pacchetti diversi possono essere uguali. Pacchetti e sottopacchetti possono contenere solo sottopacchetti, i sottopacchetti devono avere un nome che rifletta il percorso di sistema.

Imports

Si possono usare *type imports*, per fare riferimento ad una classe dichiarata in un altro pacchetto con il suo nome.

Listing 11.26: Import in java

```
import java.util.Scanner;
import java.lang.*;
```

Inoltre si possono importare staticamente metodi così da abbreviarne l'utilizzo

Listing 11.27: Esempio di static import in Java

```
import static java.lang.System.out;


...
out("Posso usare System.out scrivendo solo out");
```

11.17 Oggetti e tipi primitivi

Assegnare tipi agli oggetti permette che i valori possano essere gestiti uniformamente attraverso referenze. Le conversioni tra primitivi a valori di tipo oggetto si dicono **boxing** mentre quelle da oggetto a primitivo si dicono **unboxing**.

Listing 11.28: Esempi di oggetti e tipi primitivi in Java

```
assert 5 / 2 == 2;
assert 5 / 2. == 2.5;
Integer i = 5;
assert i * 2 == 10;
assert i * i == 25;
assert i / 2. == 2.5;
```