Appunti di Linguaggi e Programmazione orientata agli Oggetti

Roberto Castellotti

November 18, 2020

Contents

1 Introduzione **5**
1.1 Elementi principali di un PL 5
1.1.1 Statically Typed Languages 5
1.1.2 Dinamically Typed Languages 5
1.1.3 Sintassi 6
1.1.4 Stringhe 6
1.1.5 Linguaggi 7
1.2 Espressioni Regolari (regex) 8
1.2.1 Semantica delle espressioni regolari 8
1.2.2 Precedenza e Associatività delle espressioni regolari . . . 9
1.2.3 Operatori derivati e notazione estesa (Java API syntax) . 9
1.2.4 Usi delle espressioni regolari 10
1.3 Analisi lessicale 10
1.4 Linguaggi Regolari 11
1.5 Analisi Sintattica di un programma 11
1.5.1 Esempi di parsing 12

2 Grammatiche Context Free **13**
2.1 Una grammatica BNF (Backus-Naur Form) 13
2.2 Terminologia di una grammatica CF 13
2.3 Grammatiche come definizioni induttive di linguaggi 14
2.4 Derivazioni 14
2.4.1 One-step derivation 14
2.4.2 Definizione formale di derivazione 14
2.4.3 Derivation tree (o parse tree) 14
2.4.4 Esempi di derivation tree (ANTLR) 15
2.4.5 Definizione formale di derivation tree 15
2.5 Grammatiche ambigue 16
2.5.1 Motivi di ambiguita' per una grammatica 16
2.5.2 Possibili soluzioni per risolvere ambiguita' 16
2.5.3 Tecniche Standard per disambiguare le grammatiche . . . 17

3 Il linguaggio OCaml **23**
3.1 Paradigmi di programmazione . . . 23
3.1.1 Paradigma interamente funzionale . . . 23
3.2 Tipi . . . 25
3.3 Funzioni Higher Order . . . 26
3.4 Tuple . . . 27
3.5 Variabili . . . 29
3.5.1 Dichiarazione di variabili locali . . . 29
3.5.2 Dichiarazione di variabili globali . . . 31
3.6 Curried/uncurried functions . . . 31
3.6.1 Applicazione parziale . . . 32
3.6.2 Ugualianza di funzioni . . . 32
3.6.3 Curried functions e generic programming . . . 33
3.7 Booleani . . . 33
3.8 Liste . . . 35
3.8.1 Regole sintattiche per il costruttore :: . . . 36
3.8.2 Liste vs Tuple . . . 36
3.8.3 Type constructor per le liste . . . 36
3.8.4 Semantica statica . . . 37
3.8.5 Polymorhpic types . . . 37
3.8.6 Concatenazione di Liste . . . 37
3.8.7 Semantica statica della concatenazione . . . 38
3.8.8 Pattern Matching . . . 38
3.8.9 Semantica del pattern matching . . . 40
3.9 Ricorsione ed efficienza . . . 43
3.9.1 Accumulatori . . . 44
3.9.2 Tail recursion . . . 44
3.9.3 Accumulatori e Tail Recursion . . . 45
3.10 Stringhe in OCaml . . . 46
3.11 Funzioni generiche sulle liste . . . 47
3.11.1 Funzione `map` . . . 47
3.11.2 Funzione `fold_left` . . . 47
3.12 Exceptions . . . 48
3.12.1 Eccezioni in Ocaml . . . 49
3.12.2 Dichiarazione del costruttore di eccezioni . . . 49
3.13 Variant types . . . 51
3.13.1 Numeri floating point in Ocaml . . . 52
3.13.2 Recursive variant types . . . 53

4 OOP **57**
4.1 Introduzione: alcuni linguaggi OOP . . . 57
4.2 Instance Methods . . . 58
4.3 Le classi . . . 59

Chapter 1

Introduzione

1.1 Elementi principali di un PL

- sintassi
- semantica statica (solo nei linguaggi statically typed)
- semantica dinamica

1.1.1 Statically Typed Languages

Una semantica statica ha regole per controllare che:

- gli operatori/statement siano usati consistentemente con i tipi dei valori
- le variabili siano dichiarate e usate consistentemente con la definizione
- pros: early error detection, efficienza
- esempi: Java, C++

1.1.2 Dinamically Typed Languages

- No semantica statica
- uso inconsistente dei valori → genera errrori a runtime
- esempi: Python, Javascript

Errori di sintassi

```
x=; // syntax error
```

```
int x=0;// Java, statically typed language
if(y<0) x=3; else x="three";
// Static error: incompatible types, String cannot be converted to int
```

Errori Statici

Errori Dinamici (tutti i linguaggi)

E' molto più probabile che si presentino nei linguaggi dinamici

```
x=null;
if(y<0) y=1; else y=x.value;
// Dynamic error if y⩾0:
// in Java: Exception in thread "main" java.lang.NullPointerException
// in JavaScript (dynamic language): cannot read property 'value' ofnull
```

1.1.3 Sintassi

1.1.4 Stringhe

alfabeto: un insieme non vuoto finito di simboli A
stringa: una sequenza $u : [1 \ldots n] \rightarrow A$ dove:

- u è una funzione **totale**
- $n = length(u)$

Essenzialmente possiamo dire che un programma è una stringa su un alfabeto.

Esempi di stringhe

Stringa vuota

- Una stringa $u : [1 \ldots 0] \rightarrow A$
- Esiste una unica funzione $u : \emptyset \rightarrow A$

Stringa non vuota

Consideriamo $A = \{('a', \ldots, 'z')\} \cup \{('A', \ldots, 'Z')\}$, la stringa *"word"* è rappresentata dalla funzione $u : [1 \ldots n] \rightarrow A$ t.c.

- $u(1) = "W"$
- $u(2) = "o"$
- $u(3) = "r"$
- $u(4) = "d"$

Concatenazione di Stringhe

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

```
for all i ∈ [1...length(u) + length(v)]
    (u·v)(i) = if i ≤ length(u) then u(i) else v(i − length(u))
```

- La concatenazione di stringhe è **associativa** ma non **commutativa**
- La stringa vuota è l' elemento neutro della concatenazione
- u^n rappresenta u concatenata con se stessa n volte

Insiemi di stringhe Se A è un alfabeto definiamo:

- A^n : insieme di tutte le stringhe di lunghezza n su A
- $A^+ = \bigcup_{n>0} A^n$: insieme di tutte le stringhe su A con lunghezza > 0
- $A^* = \bigcup_{n\geq 0} A^n = A^0 \cup A^+$: insieme di tutte le stringhe su A
- A^0 : $\{\epsilon\}$

1.1.5 Linguaggi

Identifiers Consideriamo L_{id} insieme di tutti gli **identifiers**

- $A = \{'a', \ldots, 'z'\} \cup \{'A', \ldots, 'Z'\} \cup \{'0', \ldots, '9'\}$
- $L_{id} = \{'a', 'b', \ldots, 'a0', 'a1', \ldots\}$
- **concatenazione:** $L1 \cdot L2 = \{u \cdot w | u \in L_1, w \in L_2\}$
- **unione:** $L_1 \cup L_2$

Unione : $L_1 \cup L_2$: qualsiasi stringa di L è una stringa di L_1 oppure di L_2

Concatenazione : $L = L_1 \cup L_2$: qualsiasi stringa di L è una stringa di L_1 seguita da una stringa di L_2

Redpilled facts about languages

- La concatenazione di linguaggi è **associativa** ma non **commutativa**
- $L^0 = \{\epsilon\}$ La stringa vuota è l' **elemento neutro** della concatenazione di stringhe
- L^n rappresenta L concatenato con se stesso n volte
- L^+ : $\bigcup_{n>0} L^n$ (ogni stringa è ottenuta concatendando zero o più stringhe di L)
- L^* : $\bigcup_{n\geq 0} L^n = L^0 \cup L^+$ (* è detta Kleene Star)

1.2 Espressioni Regolari (regex)

Le **espressioni regolari** sono un formalismo per definire semplici linguaggi, sostanzialmente un **insieme di stringhe**.

Definzione induttiva delle espressioni regolari su un un alfabeto A

- Casi Base
  - **empty set:** $\emptyset$ è una espressione regolare su A
  - **singleton set con empty** ϵ: è una espressione regolare su A (elemento neutro concatenazione)
  - **singleton set con stringa n=1:** $\forall \sigma \in A$ σ è una espressione regolare su A
- Casi Induttivi
  - **unione:** se e_1 e e_2 sono regex su A allora $e_1|e_2$ è una regex su A.
  - **concatenazione:** se e_1 e e_2 sono regex su A allora e_1e_2 è una regex su A (non si usa nessun simbolo)
  - **Kleene Star:** se e è una regex su A allora e^* è una regex su A
  - **parentesi:** se e è una regex su A allora (e) è una regex su A

1.2.1 Semantica delle espressioni regolari

La semantica di una regex su A è un linguaggio su A (un insieme di stringhe su A)

- $\emptyset \rightsquigarrow$ insieme vuoto
- $\epsilon \rightsquigarrow \{\epsilon\}$
- $\sigma \rightsquigarrow \{"\sigma"\}, \forall \sigma \in A$
- $e_1|e_2 \rightsquigarrow$ unione delle semantche di e_1 e e_2
- $e_1e_2 \rightsquigarrow$ concatenazione delle semantche di e_1 e e_2
- $e^* \rightsquigarrow$ la Kleene Star della semantica di e
- $(e) \rightsquigarrow$ la semantica di e

1.2.2 Precedenza e Associatività delle espressioni regolari

1. la Kleene Star ha maggiore precedenza rispetto a concatenazione e unione
2. la concatenazione ha precedenza maggiore rispetto all'unione
3. la concatenazione e l'unione sono associative a sinistra

$$e_1|e_2|e_3 \text{ corrispopnde a } (e_1|e_2)|e_3$$
$$e_1e_2e_3 \text{ corrispopnde a } (e_1e_2)e_3$$

> La associatività a sinistra ha impatto sulla sinstassi ma non sulla semantica, perché unione e concatenazione dei linguaggi sono associative.

4. le parentesi possono forzare le regole di precedenza, ad esempio:

$$ab|c \rightsquigarrow \{ab, c\}$$
$$a(b|c) \rightsquigarrow \{ab, ac\}$$

1.2.3 Operatori derivati e notazione estesa (Java API syntax)

- $e+ \equiv ee^*$ (una o più volte e)
- ϵ è rappresentato dalla stringa vuota ($a|e$ diventa $a|$)
- $e? \equiv \ |e$ (una o zero volte e)
- $[\dots]$ (uno dei caratteri tra parentesi) ($[a4B]$ corrisponde a $a|4|b$)
- $[\dots - \dots]$ (uno dei caratteri nel range tra parentesi) ($[b-d]$ corrisponde a $b|c|d$)
- $[a4b]-[b-d]$ può essere scritta in modo più compatto come $[a4Bb-d]$
- $[\hat{}\dots]$ ogni carettere eccetto il carattere specificato ($[\hat{}a4Bb-d]$ significa ogni carattere eccetto $a, 4, B, b, c, d$)
- `.` significa ogni carattere
- `\` è il carattere di escape, assegna valore standard a caratteri speciali e valore speciale a caratteri standard, ad esempio:
  - `*, \+, \?, \., \\, \[, \-, \^`
  - `\t`: tab
  - `\n`: newline
  - `\s`: ogni carattere che include whitespace

- `\S`: ogni carattere che non include whitespace
- `\d`: ogni digit (`[0-9]`)
- `\D`: ogni non-digit (`[^0-9]`)
- `\w`: ogni word (`[a-zA-Z_0-9]`)
- `\W`: ogni non-word (`[^\w]`)

Esempi

- **identifiers:** `(a|...|z|A|...Z)(a|...|z|A|...Z|0|...|9)`, better `[a-zA-Z][a-zA-Z0-9]*`
- **numeri (base 10):** `0|(1|...|9)(0|...|9)*`, better `0|[1-9][0-9]*`

  La regex precedente permette di evitare il match di 00 e 023

- **numeri (base 8):** `0(0|...|7)(0|...|9)*`, better `0[0-7]*`

1.2.4 Usi delle espressioni regolari

Le regex sono usate per:

- lexer/tokenizer
- data validation (web forms)
- text manipulation (find and replace)

1.3 Analisi lessicale

Una substring di una stringa che è considerata una unità sintattica prende il nome di **lessema**, la decomposizione di stringhe in lessemi prende il nome di **analisi lessicale**, un tool che fa analisi lessicale e riconosce i lessemi si chiama **lexer/scanner**.

Esempio. la stringa `"x2=042;"` è decomposta in: `"x2"`; `"="`; `"x2"`; `"042"`; `";"`

Esempi di decomposizione (C/Java/C++)
La maggior parte degli errori sintattici non viene detectato durante la analisi lessicale (primo step della analisi statica),

- la stringa `"=x2;042"` non è sintatticamente corretta ma viene decomposta nei lessemi `"="`; `"x2"`; `";"`; `"042"` senza alcun problema.
- la stringa `"2x=042;"` non è sintatticamente corretta, (sintassi dell' identifier sbagliata) ma la stringa viene decomposta nei lessemi seguenti senza problemi: `"2"`; `"x"`; `"="`; `"042"`; `";"`, quindi un errore lessicale può introdurre una decomposizione non voluta

- alcuni errori lessicali sono riconosciuti dal lexer perché non permettono di decomporre in lessemi (`"x2='\';"`), questo errore in particolare è dovuto al fatto che la sintassi dei char prevede che ci sia almeno un carattere tra `\` e `'`

Durante l' analisi statica si lavora su un livello di astrazione più alto rispetto al lessema: il **token**, che fornisce una astrazione della sintassi del lessema insieme al **tipo** del token. Alcuni esempi di token sono: identifiers, numeri, operatore di assignment, un token può avere informazioni sulla **sintassi** o sulla **semantica** `(numero,valore)`. Un **tokenizer** è un lexer che riconosce i lessemi e genera i corrispondenti **token**.

Esempio. La stringa `"x2=042;"` è decomposta nei seguenti token:

- `IDENTIFIER` con informazione sintattica: name `"x2"`
- `ASSIGN_OP` senza informazioni aggiuntive
- `INT_NUMBER` con valore semantico: value `"trentaquattro"` (ottale)
- `STATEMENT_TERMINATOR` senza informazioni aggiuntive

1.4 Linguaggi Regolari

Un **linguaggio regolare** è un linguaggio definibile con una **espressione regolare**, questi linguaggi sono piuttosto semplici (Identifier, numeri, stringhe) e possono definire semplici parti di un linguagggio di programmazione, ma non ne possono definire l'intera sintassi.

Il linguagggio delle esepressioni con numeri, somma binaria, moltiplicazione e parentesi non può essere definito da una regex (a causa delle parentesi) (NON DIMOSTRATO), un semplice esempio di linguaggio non lineare è il seguente:

$$\{a^n b^n | n \in \mathbb{N}\} = \{"", "ab", "aabb", "aaabbb", \ldots\}$$

1.5 Analisi Sintattica di un programma

L'**analisi sintattica di un programma**, basata su regole sintattiche definite da una **grammatica**, è svolta da un parser che:

- riconosce le sequenze di token generate dal **tokenizer**
- genera un:
  - **parse/derivation tree:** una rappresentazione concreta del programma riconosciuto
  - **Abstract Syntax Tree (AST):** una rappresentazione più astratta

Un **parser** può essere scritto a mano o generato con tool come ANTLR o Bison.

1.5.1 Esempi di parsing

Failure Token analizzati

- `IDENTIFIER` con informazione sintattica: name `"x2"`
- `INT_NUMBER` con valore semantico: value `"trentaquattro"` (ottale)
- `ASSIGN_OP` senza informazioni aggiuntive
- `STATEMENT_TERMINATOR` senza informazioni aggiuntive

la sequenza non è riconsociuta e sono riportati i messaggi di errore

- `IDENTIFIER` con informazione sintattica: name `"x2"`
- `ASSIGN_OP` senza informazioni aggiuntive
- `INT_NUMBER` con valore semantico: value `"trentaquattro"` (ottale)
- `ADD_OP`
- `INT_NUMBER` con valore semantico: value `"otto"` (ottale)
- `STATEMENT_TERMINATOR` senza informazioni aggiuntive

Success

la sequenza è riconsociuta e viene costruito il seguente AST

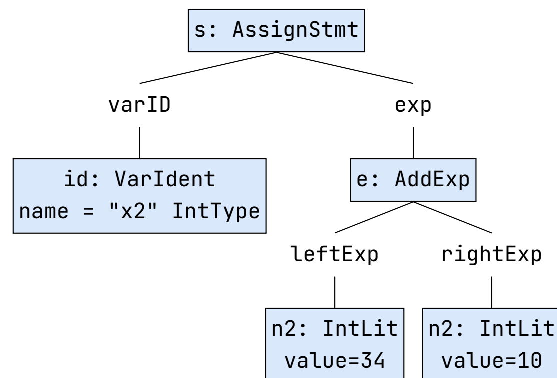


Figure 1.1: AST generato dal parser

Chapter 2

Grammatiche Context Free

Le **grammatiche context free (CF)** sono un formalismo standard per definire la sintassi di un linguaggio di programmazione. Esse sono più espressive delle **espressioni regolari**: hanno gli stessi operatori di base (concatenazione e unione) ma usano la **ricorsione** al posto dell'iterazione (Kleene Star).

2.1 Una grammatica BNF (Backus-Naur Form)

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

`Exp` ha la prima lettera maiuscola (definito nella grammatica)
`NUM` ha tutte le letere maiuscole (definito a parte da una regex)

2.2 Terminologia di una grammatica CF

- `{'+','*','(',')','0','1'}` è l' insieme `T` dei **simboli terminali**
- `{Exp,Num}` è l'insieme `N` dei **simboli non terminali**
- ```
  (Exp, Num),
  (Exp, Exp '+' Exp),
  (Exp, Exp '*' Exp),
  (Exp, '(' Exp ')'),
  (Num, '0'),
  (Num, '1')
  ```
  è l'insieme `P` delle **produzioni**
  ```

Ogni **non-terminale** corrisponde a un linguaggio (definito come unioni o concatenazioni). I simboli terminali sono lessemi del linguaggio definito dalla grammatica, le produzioni sono del tipo (B, α) con $B \in \mathbb{N}, \alpha \in (T \cup N)^*$.

2.3 Grammatiche come definizioni induttive di linguaggi

$$\text{Exp} = \text{Num} \cup (\text{Exp} \cdot \{"+"\} \cdot \text{Exp}) \cup (\text{Exp} \cdot \{"*"\} \cdot \text{Exp}) \cup (\{"("\} \cdot \text{Exp} \cdot \{")"\})$$
$$\text{Num} = \{"0"\} \cup \{"1"\}$$

`Num` è il caso base di `Exp`: un numero è una espressione, `Exp` è definita su `Num` mentre `Num` è definita solo su casi base.

2.4 Derivazioni

Una grammatica genera un linguaggio per ogni simbolo non-terminale.

Ad esempio la grammatica precendente genera L_{Exp} e $L_{Num} = \{"0", "1"\}$

2.4.1 One-step derivation

La derivazione a uno step ci permette di capire se `"1+0"` $\in L_{Exp}$, dimostriamolo:

1. $\underline{\text{Exp}} \to \text{Exp} \times \text{Exp}$ produzione (Exp, Exp × Exp)
2. $\underline{\text{Exp}} \times \text{Exp} \to \text{Num} \times \text{Exp}$ produzione (Exp, Num)
3. $\text{Num} \times \underline{\text{Exp}} \to \text{Num} \times \text{Num}$ produzione (Exp, Num)
4. $\underline{\text{Num}} \times \text{Num} \to 0 \times \text{Num}$ produzione (Num, 0)
5. $0 \times \underline{\text{Num}} \to 0 \times 1$ produzione (Num, 1)

2.4.2 Definizione formale di derivazione

La derivazione one-step per una grammatica $G = (T, N, P)$ è del tipo:

$$\alpha_1 B \alpha_2 \to \alpha_1 \gamma \alpha_2, \text{ con } \alpha_1, \alpha_2 \in (T \cup N)^*, (B, \gamma) \in P$$

2.4.3 Derivation tree (o parse tree)

Un **derivation tree** èuna versione meno astratta di un AST. Ogni volta che facciamo un passo di derivazione questo è determinato da 2 gradi di liberta':

- la produzione che viene usata
- lo specifico non-terminale che vogliamo rimpiazzare

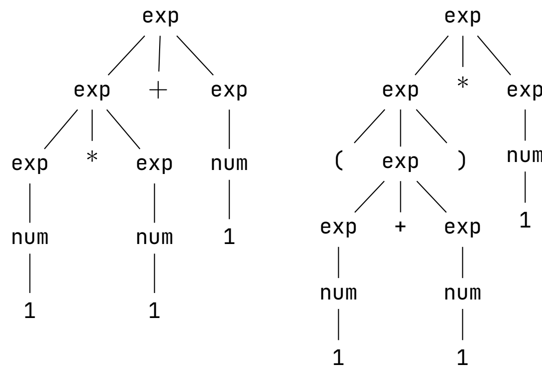


Figure 2.1: Derivation tree di `"1*1+1"` e `"(1+1)*1"`

Un **derivation tree** è una generalizzazione di una derivazione multi-step tale che:

- la stringa che ne deriva contiene solo simboli terminali
- i non-terminali sono rimpiazzati simultaneamente
- la struttura della sequenza di token analizzata è resa esplicita

2.4.4 Esempi di derivation tree (ANTLR)

```
grammar SimpleExp;
exp: num|exp'*'exp|exp'+'exp|'('exp')' ;
num: '0'|'1';
```

2.4.5 Definizione formale di derivation tree

Un derivation tree in G per $u \in T^*$ a partire da $B \in N$ è un alberto con nodi etichettati in $T \cup N$ tale che:

- se un nodo è etichettato con $C \in N$ e i suoi n figli con $I_1, \dots I_n$ da sinistra a destra, allora $(C, I_1 \dots I_n) \in P$ ($(C, I_1 \dots I_n)$ è una produzione di G)
- la radice è etichettata con B
- u è ottenuto concatenando left-to-right tutte le etichette terminali (necesariamente foglie)

La creazione di un albero ci permette di definire che un linguaggio L_b generato da $= (T, N, P)$ per i non-terminali $B \in N$: come un insieme di tutte le stringhe u dei terminali tali che esiste un derivation tree per u partendo da B

2.5 Grammatiche ambigue

Una grammatica $G = (T, N, P)$ è ambigua per $B \in N$ se esistono due derivation tree distinti partendo da B per la stessa stringa.

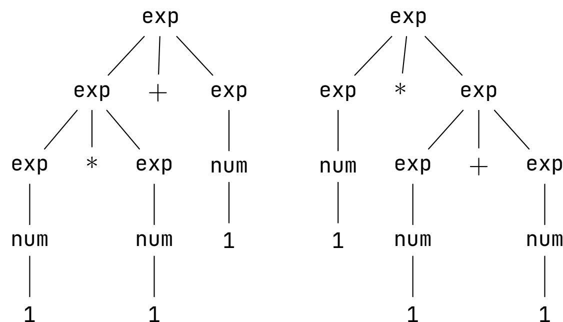


Figure 2.2: Derivation tree di "1*1+1"

2.5.1 Motivi di ambiguita' per una grammatica

- Gli operatori infissi binari sono intrinsecamente ambigui
- **associativita' sintattica di un singolo operatore:** $1 + 1 + 1$ significa $(1 + 1) + 1$ o $1 + (1 + 1)$? L' addizione è **left-associative** o **right-associative**?
- **Precedenza degli operatori** $1 + 1 * 1$ significa $(1 + 1) * 1$ o $1 + (1 * 1)$? Quale operatore ha precedenza maggiore?
- **associativita' sintattica degli operatori con la stessa precedenza:** se addizione e moltiplicazione hanno la stessa precedenza $1+1*1$ significa $(1 + 1) * 1$ o $1 + (1 * 1)$? La moltiplicazione e l' addizione sono **left-associative** o **right-associative**?

2.5.2 Possibili soluzioni per risolvere ambiguita'

Per rimuovere l' ambiguita' possiamo cambiare la sintassi:

Prefix notation

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

- esiste un unico derivation tree per "+1*1 1"
- differenza tra "+1*1 1" e "*+1 1 1"
- le parentesi non sono più necessarie

Postfix notation

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

- esiste un unico derivation tree per `"1 1 1*+"`
- differenza tra `"1 1 1*+"` e `"1 1+1*"`
- le parentesi non sono più necessarie

Functional notation

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

- esiste un unico derivation tree per `"add(1,mul(1,1))"`
- differenza tra `"add(1,mul(1,1))"` e `"mul(add(1,1),1)"`

Soluzioni migliori

La **infix-notation** seppure ambigua è la più intuitiva e pratica, definiamo quindi alcune regole per eliminare la ambiguita' in una grammatica CF:

- definiamo regole di associazione
  - la addizione è **left-associative**: `"1+1+1"` signfica `"(1+1)+1"`
  - la addizione è **right-associative**: `"1+1+1"` signfica `"1+(1+1)"`
- definiamo regole di precedenza per gli operatori e usiamo le parentesi per overridare
  - la moltiplicazione ha precedenza maggiore della addizione: `"1+1*1"` significa `"1+(1*1)`
  - la addizione ha precedenza maggiore della moltiplicazione: `"1*1+1"` significa `"1*(1+1)`
  - addizione e moltiplicazione hanno la stessa precedenza e sono left-associative: `"1*1+1*1"` significa `"((1*1)+1)*1"`
  - addizione e moltiplicazione hanno la stessa precedenza e sono right-associative: `"1*1+1*1"` significa `"1*(1+1)*1)"`

2.5.3 Tecniche Standard per disambiguare le grammatiche

- una grammatica ambigua G viene trasformata in una grammatica G' non ambigua **equivalente**
- **equivalente** significa che per tutti in non-terminali B di G i linguaggi generati da G e G' da B sono uguali
- la trasformazione è in grado di aggiungere regole di associativita' o di precedenza nella grammatica G' non ambigua

+ e * con la stessa precedenza

Una grammatica ambigua

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

Grammatica non ambigua left-associative

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

Grammatica non ambigua right-associative

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

`Exp'+'Atom(Exp'*'Atom)` significa che sul lato destro di `+` o `*` addizioni e moltiplicazioni sono ammesse solamente se dentro le parentesi

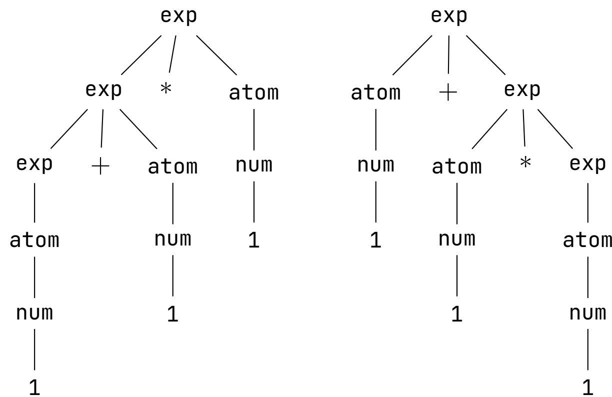


Figure 2.3: Unico derivation tree di `"1*1+1"` left e right-associative

*** con precedenza maggiore**

Grammatica ambigua

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

Grammatica non ambigua left-associative

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

`Mul'*'Atom` significa che su entrambi i lati di `*` le addizioni sono ammesse solamente dentro le parentesi

Grammatica non ambigua right-associative

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

`Atom'*'Mul` significa che su entrambi i lati di `*` le addizioni sono ammesse solamente dentro le parentesi

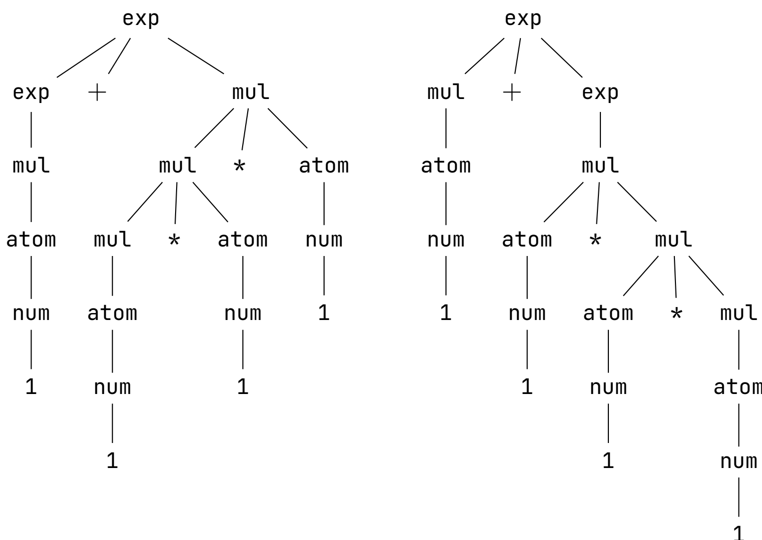


Figure 2.4: Unico derivation tree per `1+1*1*1` left e right associative

Sintassi ambigua per statement

Statement vs Espressione: una espressione è una stringa che possiamo valutare e che ha un valore, uno **statement** richiama l' idea di esecuzione ma non ha un valore

```
Stmt ::= ID'='Exp';'|'if''('Exp')'Stmt|Stmt Stmt|'{'Stmt'}'
Exp ::= ID|BOOL // ID and BOOL defined by regular expressions
```

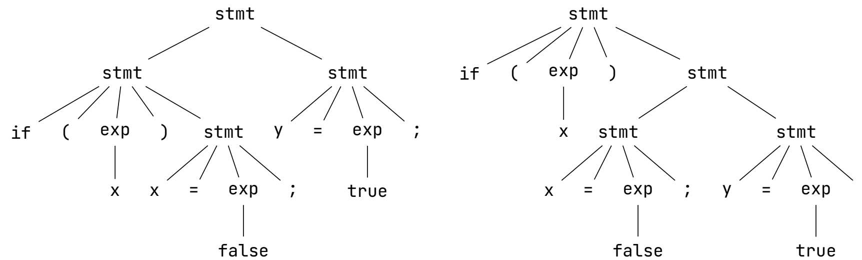


Figure 2.5: Due derivation tree per "if(x)x=false;y=true;"

Questa differenza si riflette negli AST che vengono creati:

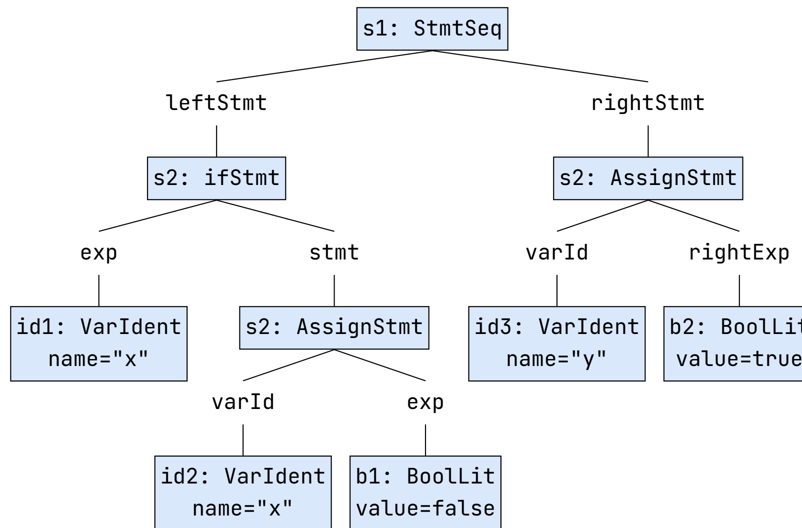


Figure 2.6: AST generato dal parser

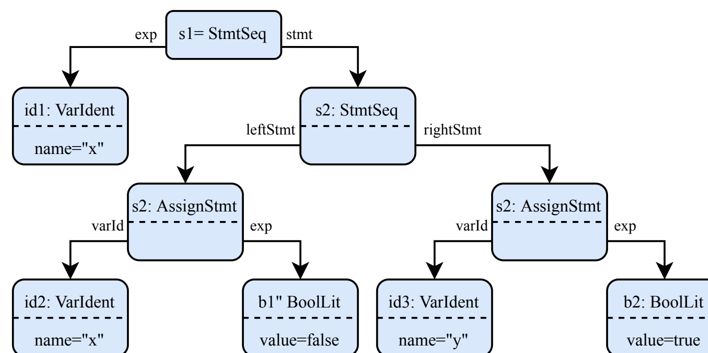


Figure 2.7: Abstract Syntax Tree per `"if(x)x=false;y=true;"` (sequenza di statement ha precedenza maggiore)

Esempio in JavaScript (o qualsiasi altro lang C-like)

```
x=false;
y=false; //wrong indentation
if(x)x=false; y=true;
console.log('$x $y');
```

```
x=false;
y=false;
if(x){x=false; y=true;}
console.log('$x $y');
```

```
Output: false true
```

```
Output: false false
```

Chapter 3

Il linguaggio OCaml

3.1 Paradigmi di programmazione

Un **paradgima di programmazione** è lo stile/approccio usato per programmare in un linguaggio di programmazione basato sul **predominante modello computazionale**.

Principali paradigmi

- **paradigma imperativo** (Von Neumann style) è basato sulle nozioni di **istruzione** e **stato**, si divide in:
  - **procedurale** (C)
  - **object-oriented** (Java)
- **paradigma dichiarativo** è basato su un livello di astrazione più alto
  - **funzionale** (ML) basato sulla nozione di **funzione matematica** e **applicazione di funzione**
  - **logico** (Prolog) basato sulla nozione di **regola logica** e **query**

I linguaggi di programmazione moderni abbracciano diversi paradigmi di programmazione per essere più flessibili; Java, JavaScript, Python3 e mlti altri linguaggi supportano sia il paradigma **imperativo** sia quello **dichiarativo**

3.1.1 Paradigma interamente funzionale

- programma = definizone di funzioni matematiche + 1 espressione (main)
- le call di funzioni del contesto imperativo si chiamano **applicazioni di funzioni**
- **non esiste una nozione di stato**: no assegnazione di variabili, no statement, solo espressioni

```
fun x → x+1 (* funzione incremento *)
fun x y → x+y (* funzione addizione *)
(fun x → x+1) 3 (* applicazione della funzione a 3 *)
```

Listing 3.1: Esempi di funzioni e applicazioni

- le variabili sono parametri di funzioni che contengono valori costanti
- le funzioni sono **valori di prima classe**, vale a dire che una funzione può essere il risultato della valutazione di una espressione.
- le **high order functions** sono funzioni che trattano altre funzioni in input/output
- le **lambda expressions/functions** o **anonymous function** sono funzioni definite da una espressione

In questo corso useremo OCaml, un dialetto francese di ML, è un linguaggio multi paradigma con un nucleo puramente funzionale, **statically typed** e con **type inference**.

Grammatica EBNF

```
Exp::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp |
       Uop Exp | Exp BOP Exp|'('Pat')'
Pat::= ID|'(' Pat ')' ' //patern semplificati
```

- la notazione BNF è estesa con operatori postfissi (`+ * ?`) e parentesi
  - *: la **Kleene Star**
  - +: concatenazione di 1 o più (diverso dal simbolo terminale `'+'`)
  - ?: opzionalita' 0 o 1 volta
- `ID` : identifier di variabili `[a-zA-Z_][\w']*`
- `NUM`: numeri naturali `0[bB][01][01_]*|0[oO][0-7][0-7_]*|0[xX][0-9a-fA-F][0-9a-fA-F_]*|\d[\d_]*`
- `UOP`: operatori aritmetici unari `[+-]`
- `BOP`: operatori aritmetici binari `[+-*/`mod]|
- `|Pat|`:pattern, verra' data una definizione più completa

```
exp1 exp2
```

- `exp1` deve returnare una funzione `f`
- `exp2` deve returnare un argomento valido `a`
- `exp1 exp2` deve ritornare `f(a)`

Regole di precedenza e associativita'

- regole standard per espressioni aritmetiche
- la applicazione di funzione è **left-associative**

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

- la applicazione ha precedenza maggiore degli operatori binari

```
(fun x->x*2) 1+2 (* is ((fun x->x*2) 1)+2 *)
1+(fun x->x*2) 2 (* is 1+((fun x->x*2) 2) *)
```

- le anonymous function hanno precedenza minore di applicazione e operatori binari

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

- casi critici: applicazione e operatori unari

```
f+3 (* addizione *)
f (+3) (* applicazione *)
f-3 (* sottrazione *)
f (-3) (* applicazione *)
+ f 3 (* è +(f 3) *)
- f 3 (* è -(f 3) *)
```

3.2 Tipi

- `int` è un **tipo primitivo** (interi)
- `int -> int` è un **tipo composito**
- `->` è un **costruttore di tipo** e viene usato per costruire tipi compositi a partire da tipi semplici, i tipi costruiti prendono il nome di **arrow types o function types**
- $t_1 \rightarrow t_2$ è il tipo delle funzioni da t_1 a t_2 che:
  - possono esser applicate a un singolo argomento di tipo `t1`
  - restituiscono sempre un valore del tipo `t2`

```
(* REPL *)
# 42
- : int = 42
# fun x -> x*2;;
- : int -> int = <fun> (* int -> int è il tipo della funzione *)
# (fun x->x+1) 2
- : int = 3
# fun x y -> x+y;;
- : int -> int -> int = <fun>
# (fun x y -> x+y) 3 4;;
- : int = 7
```

Listing 3.2: `REPL:` i tipi sono inferiti dall' interprete

- il costruttore arrow type è **right-associative**

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

- un costruttore costruisce tipi diversi dai tipi semplici di cui è composito

  $t_1 \rightarrow t_2 \neq t_1$ e $t_1 \rightarrow t_2 \neq t_2$

- due arrow ytpe sono uguali se sono costruiti a partire dagli stessi tipi semplici

  `t1 -> t2 = t3 -> t4` sse `t1 = t3 e t2 = t4`

- dagli esempi precedenti deduciamo che:

  `int->int->int` ≠ `(int->int)->int`

Tipi ed espressioni di tipi

- `int -> int -> int` è una espressione di tipi ma viene chiamata tipo perché rappresenta un tipo specifico

- `int -> int -> int` e `(int -> int) -> int` sono due espressioni di tipi diversi che rappresentano lo stesso tipo (stesso AST)

3.3 Funzioni Higher Order

Le **funzioni higher order** sono funzioni che returnano altre funzioni.

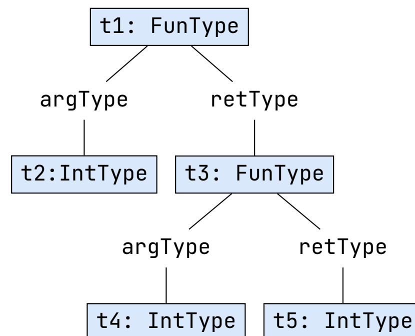


Figure 3.1: AST di `int→int→int` e `(int→int)→int`

```
# fun x y→x+y;; (* input: int, output: una funzione int →int*)
- : int → int → int = <fun>
# fun x→fun y→x+y;; (* versione estesa esempio precedente *)
- : int → int → int = <fun>
# fun x y z→x*y*z;;
- : int → int → int → int = <fun>
# fun x→fun y→fun z→x*y*z;;
- : int → int → int → int = <fun>
```

Listing 3.3: Alcuni esempi di funzioni higher order

Una utile abbreviazione sintattica

```
fun pat1 pat2 ... patn → exp (*è una abbreviazione per:*)
fun pat1 → fun pat2 → fun patn → exp
```

Nei primi due esempi prendiamo la funzione somma e la applichiamo a un `int`, succesivamente il risultato ottenuto (ancora una funzione) lo applichiamo a un altro `int`.

3.4 Tuple

Definiamo una nuova produzione per `Exp` e `Pat`

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

e una nuova produzione per `Type`

```
Type::='unit'|Type('*'Type)+
```

`unit` rappresenta la tupla vuota

```
# fun f -> 1+f 0 (*1*)
- : (int -> int) -> int = <fun>
# (fun f -> 1+ f 0)(fun x ->x+1);;
- : int = 2
# fun x y -> x+y (*2*)
- : int -> int -> int = <fun>
# fun f x -> 1+f (1+x) (*3*)
- : (int -> int) -> int -> int = <fun>
# (fun f x -> 1+f (1+x)) (fun x->x*2);;
- : int -> int = <fun>
# ((fun f x -> 1+f (1+x)) (fun x->x*2))3;;
- : int = 9
```

Listing 3.4:

1. una funzione che prende in input una funzione e returna un intero
2. una funzione che prende in input un intero e returna una funzione (da `int` a `int`)
3. una funzione che prende in input una funzione (da `int` a `int`) e returna una funzione (da `int` a `int`)

```
# () (*1*)
- : unit = ()
# 1,2,3 (*2*)
- : int * int * int = (1, 2, 3)
# (1,2),3 (*3*)
- : (int * int) * int = ((1, 2), 3)
# 1,(2,3) (*4*)
- : int * (int * int) = (1, (2, 3))
```

Listing 3.5:

1. la tupla vuota
2. una tupla di dimensione 3 `int`
3. una tupla di dimensione 2 (un elemento è a sua volta una tupla)

Precendenza e regole di associativita'

- l' operatore `,` ha precedenza maggiore delle funzioni anonime
- l' operatore `,` ha predecenza minore degli altri operatri
- l' operatore `,` non è **ne' left ne right-associative**
- il costruttore `*` ha precedenza maggiore del costruttore `->`
- il costruttore `*` non è **ne' left ne right-associative**

Nelle tuple le parentesi sono necessarie solamente per la tupla vuota o per forzare precedenze.

Adesso che abbiamo esteso la definizione di **pattern** possiamo definire funzioni che hanno come **argomento tuple**.

3.5 Variabili

3.5.1 Dichiarazione di variabili locali

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

Le dichiarazioni nestate oscurano dichiarazioni esterne con lo stesso ID.
In questo esempio `f` continua a fare riferimento al valore della variabile prima della sua dichiarazione.

```
# fun()->3
- : unit -> int = <fun>
# fun (x,y) -> x+y
- : int * int -> int = <fun>
# (fun (x,y) -> x+y) (3,4)
- : int = 7
# fun (x,y,z)->x*y*z
- : int * int * int -> int = <fun>
# (fun (x,y,z)->x*y*z) (2,3,5)
- : int = 30
# fun ((x,y),z)->x*y*z
- : (int * int) * int -> int = <fun>
# (fun ((x,y),z)->x*y*z) (2,3,5)
Characters 23-30:
(fun ((x,y),z)->x*y*z) (2,3,5);;
-----
Error: an expression was expected of type (int * int) * int
```

Listing 3.6:

1. la tupla vuota
2. una tupla di dimensione 3 `int`
3. una tupla di dimensione 2 (un elemento è a sua volta una tupla)

```
# let f x=x+1 and v=41 in f v;; (* f and v can only be used here *)
- : int = 42
# let x=1 in let x=x*2 in x*x (* nested declarations *)
- : int = 4
```

Listing 3.7: Esempio di dichiarazione di variabili locali

```
let v=40;;
let f x = x*v;; (* v refers to the declaration above *)
f 3;; (* evaluates to 120 *)
let v=4;; (* previous declaration of v shadowed *)
(* f still refers to the shadowed variable v *)
f 3;; (* evaluates to 120 *)
```

Listing 3.8: Scope statico delle dichiarazioni

```
let x=2;; (* x has type int and value 2 *)
let y=x+40;; (* y has type int and value 42 *)
x+y;; (* value 44 *)
```

Listing 3.9: Esempi di dichiarazione di variabili gloabli

3.5.2 Dichiarazione di variabili globali

```
Dec ::= 'let' Pat '=' Exp| 'let' ID Pat+ '=' Exp
```

Ricordiamo che:

- le variabili sono **globali** perché dichiarate **top level**
- il valore di una variabile **non può essere cambiato**, è una **costante**
- no assignment di variabili
- una nuova dichiarazione di variabile con lo stesso nome **oscura** quella precedente

Possiamo anche assegnare alle variabili delle funzioni:

```
let inc x = x+1;; (* inc has type int → int *)
let add (x,y) = x+y;; (* add has type int*int → int *)
let add2 x y = x+y;; (* add2 has type int → int → int *)
```

Una abbreviazione sintattica utile:

```
let id pat1 pat2 ... patn = exp
```

è una abbreviazione per:

```
let id = fun pat1 pat2 ... patn → exp
```

che è a sua volta una abbreviazione per:

```
let id = fun pat1 → fun pat2 → ... fun patn → exp
```

3.6 Curried/uncurried functions

Una **carried function** è una **high-order function** che restituisce una catena di funzioni.

```
fun pat1 → fun pat2 → ... fun patn → exp
```

```
(* addition of two integers *)
fun x y->x+y;; (* curried version int->int->int *)
fun (x,y)->x+y;; (* uncurried version int*int->int *)
(* multiplication of three integers *)
fun x y z->x*y*z;; (* curried version int->int->int->int *)
fun (x,y,z)->x*y*z;; (* uncurried version int*int*int->int *)
```

Listing 3.10: Esempi di funzioni curried/uncurried

```
let uncurried_add(x,y)=x+y;;
(* computes 1+2 with the uncurried version *)
uncurried_add(1,2);;

let curried_add x y=x+y;;
(* computes 1+2 by partial application *)
let inc=curried_add 1;; (* passes argument 1 and saves the result *)
inc 2;; (* passes argument 2 and computes the final result *)
```

Listing 3.11: Esempi di funzioni uncurried e curried

Una **uncarried function** è una funzione che prende in input una **tupla** di dimensione n.

```
fun (pat1,pat2,...,patn) -> exp
```

Una **uncurried function** può essere trasformata in una equivalente **curried function** e viceversa.

3.6.1 Applicazione parziale

Le **curried functions** permettono una applicazione **parziale**, vale a dire che possiamo passare gli argomenti uno alla volta, le **uncurried functions** invece richiedono che tutti gli argomenti vengano passati assieme (in una tupla).

La **applicazione parziale** permette una **specializzazione delle funzioni**, ovvero permette di creare funzioni specifiche a partire da funzioni generiche **senza duplicazione del codice**.

3.6.2 Ugualianza di funzioni

Due funzioni sono uguali se e solo se restituiscono gli stesi valori per tutti gli argomenti in input, ma non sempre le funzioni possono essere trattate come altri valori.

```
let curried_add x y=x+y;; (* curried_add : int → int → int *)
let f1=curried_add;; (* f1 : int → int → int *)
let f2 x=curried_add x;; (* f2 : int → int → int *)
let f3 x y=curried_add x y;; (* f3 : int → int → int *)
```

Listing 3.12: `curried_add,f1,f2,f3` definiscono la stessa funzione (non testabile)

```
let rec sumsquare n = if n≤0 then 0 else n*n+sumsquare(n-1);;
let rec sumcube n = if n≤0 then 0 else n*n*n+sumcube(n-1);;
```

Listing 3.13: Addizione di quadrati e addizione di cubi

Nei linguaggi funzionali le funzioni NON possono essere comparate tra di loro a causa di limitazioni teoriche.

3.6.3 Curried functions e generic programming

Le due funzioni sono praticamente uguali, possiamo migliorare il codice usando una **curried function** che prende come argomento una funzione. Questa funzione può essere ancora migliorata:

3.7 Booleani

```
Exp ::= BOOL | 'not' Exp | Exp '&&' Exp | Exp '||' Exp
Type ::= 'bool'
BOOL è definito dall' espressione regolare false|true.
```

Le regole standard per la sintassi sono:

```
(* computes f 0 + f 1 + ... + f n *)
let rec gen_sum f n = if n≤0 then 0 else f n+gen_sum f (n-1);;
(* (int → int) → int → int *)
let der_sumsquare = gen_sum (fun x→x*x);; (* int → int *)
let der_sumcube = gen_sum (fun x→x*x*x);; (* int → int *)
```

Listing 3.14: `gen_sum` è una funzione curried il cui primo argomento è `f` e non `n`

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

Listing 3.15: `aux` non richiede che sia passato come argomento `f`

- `&&` e `||` sono **left-associative**
- `not` ha precedenza maggiore di `&&`
- `&&` ha precedenza maggiore di `||`

Semantica statica

- `false` e `true` sono type-correct e sono di tipo `bool`
- `not e` è type-correct, è di tipo `bool` se e solo se `e` è type-correct ed è di tipo `bool`
- `e1 && e2` e `e1 || e2` sono type-correct e sono di tipo `bool` se e solo se `e1` ed `e2` sono type-correct e sono di tipo `bool`

Semantica dinamica

- gli operandi `&&` e `||` sono valutati left-to-right con **short-circuit** (non sempre il secondo operando è valutato)
- se `e1` è `false` allora `e1 && e2` è `false` e `e2` non è valutato
- se `e1` è `true` allora `e1 && e2` corrisponde al valore di `e2`
- se `e1` è `false` allora `e1 || e2` corrisponde al valore di `e2`
- se `e1` è `true` allora `e1 || e2` è `true` e `e2` non è valutato

Una espressione condizionale è definita come:

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

Le espressioni condizionali hanno precedenza più bassa di ogni altro operatore

Semantica statica `if e then e1 else e2` è type-correct e ha tipo `t` se e solo se:

- `e` è type correct ed è di tipo `bool`
- `e1` e `e2` sono type-correct e hanno tipo `t`

```
(* addition of square numbers *)
let sumsquare n = if n⩽0 then 0 else n*n+sumsquare(n-1);;
    Error: Unbound value sumsquare
    Hint: If this is a recursive definition,
    you should add the 'rec' keyword on line 1
let rec sumsquare n = if n⩽0 then 0 else n*n+sumsquare(n-1);;

(* an example of mutually recursive functions *)
let rec
f n = if n⩽0 then 1 else f (n-1) + g (n-1)
and
g n = if n⩽0 then 1 else f (n-1) * g (n-1);;
```

Listing 3.16: Dichiarazioni ricorsive di variabili globali

Semantica dinamica

- se `e` è `true` allora `if e then e1 else e2` assume il valore di `e1` e `e2` non è valutata
- se `e` è `false` allora `if e then e1 else e2` assume il valore di `e2` e `e1` non è valutata

Estendiamo adesso quanto detto precedentemente sulle variabili globali:

```
Dec ::= 'let' 'rec'? Pat '=' Exp ('and' Dec)* |
        'let' 'rec'? ID Pat+ '=' Exp ('and' Dec)*
```

- la keyword `'rec'` opzionale indica che la dichiarazione può essere ricorsiva
- più variabili possono essere dichiarate contemporaneamente usando la keyword `'and'` (usato per `mutual recursion`)
- le dichiarazioni ricorsive sono accettate solo per i tupi funzione e altri tipi speciali (noi consideriamo solo funzioni)

3.8 Liste

Sintassi

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

- `[]` rappresenta la lista vuota
- `h::t` rappresenta la lista con **head** `h` e **tail** `t`, `h` è un elemento mentre `t` è una lista

- `[]` ≠ `h::t`
- `h` ≠ `h::t`
- `t` ≠ `h::t`
- `h1::t1`=`h2::t2` se e solo se `h1=h2` e `t1=t2`

3.8.1 Regole sintattiche per il costruttore ::

L' operatore ::

- è **right-associative** (unico modo per garantire type-correctness)

  `h1::h2::t=h1::(h2::t)`

  `(h1::h2)::t` non è **type correct**
- ha precedenza minore di operatori unari e binari infix
- ha precedenza maggiore di:

  costruttore di tupla

  anonymous functions `(fun ... -> ...)`

  espressioni condizionali `(if ... then ... else ...)`
- `[e1;e2;...;en]` è una scrittura abbreviata:

  `[1] = 1::[]`

  `[1;2;3] = 1::2::3::[]`

  `[1,true] = (1,true)::[]`

  `1,[true] = 1,true::[]`

  L' operatore `;` ha precedenza maggiore del costruttore di tupla.
  `[1,true;2,false]=[(1,true);(2,false)]=(1,true)::(2,false)::[]`

3.8.2 Liste vs Tuple

Le liste devono essere `omogenee` (gli elementi devono avere tutti lo stesso tipo)

- `1,2,true` ha tipo `int*int*bool`
- `1;2;true` non è **type correct**

3.8.3 Type constructor per le liste

Il costruttore di tipo per le liste:

- è un operatore unario postfisso (`list`)

  `bool list` rappresenta il tipo "lista di bool"

  `int list list` rappresenta il tipo "lista di liste di interi"

```
# [1;2] (* a list of integers *)
- : int list = [1; 2]
# [true;false;true] (* a list of booleans *)
- : bool list = [true; false; true]
# [1,true] (* a list of pairs int*bool *)
- : (int * bool) list = [(1, true)]
# [1,true;2,false] (* a list of pairs int*bool *)
- : (int * bool) list = [(1, true); (2, false)]
# [[1;2];[0;3;4];[]] (* a list of lists of integers *)
- : int list list = [[1; 2]; [0; 3; 4]; []]
```

Listing 3.17: Esempi sulle liste

- ha precedenza maggiore dei costruttori → e *
- `t` ≠ `t list`, `t1→t2` ≠ `t list`, `t1*t2` ≠ `t list`
- `t1 list = t2 list` se e solo se: `t1=t2`

3.8.4 Semantica statica

- `[]` ha tipo `'a list`
- `e1::e2` è type correct e ha tipo `t list` se e solo se:

  `e1` è type correct e ha tipo `t`

  `e1` è type correct e ha tipo `t list`

3.8.5 Polymorhpic types

- `'a list` è un **polymorhpic type** o **type scheme**, rappresenta l' insieme dei valori composto dalla intersezione infinita di `t list` per tutti i tipi `t` (contiene solo `[]`)

3.8.6 Concatenazione di Liste

`Exp::= Exp '@' Exp`

- **left-associative**
- ha precedenza minore del costruttore `::`
- la notazione `(@)` rappresenta la corrispondente curried function polymorphic type `'a list → 'a list → 'a list`

  `[1]@[2;3]=(@)[1] [2;3]`

  `[1]@[2;3]@[4]=(@)((@)[1] [2;3])[4]`

```
# [1;2]@[3]@[4;5;6]
- : int list = [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 -> 'a list = <fun>
# (@) [1;2]
- : int list -> int list = <fun>
# (@) [1;2] [3;4;5]
- : int list = [1; 2; 3; 4; 5]
```

Listing 3.18: Esempi sulle liste

La concatenazione non è un costruttore!

- `e@[] = []@e = e`
- `[]@[1;2;3] = [1]@[2;3] = [1;2]@[3] = [1;2;3]@[] = [1;2;3]`

3.8.7 Semantica statica della concatenazione

`e1@e2` è type correct e ha tipo `t list` se e solo se `e1` e `e2` sono type correct e hanno tipo `t list`

- la complessita' temporale del costruttore di lista è $\Theta(1)$
- la complessita' temporale della concatenazione è $\Theta(n)$ con n = lunghezza dell' operando a sinistra

3.8.8 Pattern Matching

I pattern di liste sono una nuova produzione per `Pat`

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

- Tutte le variabili in un pattern **devono essere distinte** (per rendere il pattern matching più efficiente)
- I pattern sono costuiti con i costruttori, non con gli operatori, **solamente i costruttori** garantiscono una **decomposizione unica** dei valori

```
x x::y [x;y;z] x,y
```

Listing 3.19: Pattern validi

```
x@y x+y x&&y x,x
```

Listing 3.20: Pattern non validi

```
let add (x,y) = x+y;;
add (3,5);; (* does (3,5) match with pattern (x,y)? *)
```

- `(3,5)` e `x,y` matchano con la sostituzione `x=3,y=5`
- se facciamo la sostituzione `x=3,y=5` a `x+y` otteniamo `3+5`

```
let hd (h::t) = h;; (* returns the head of the list *)
hd [3;5];; (* does [3;5] match with pattern h::t? *)
```

- `[3:5]` e `h::t` matchano con la sostituzione `h=3,t=[5]`
- se facciamo la sostituzione `h=3,t=[5]` a `h` otteniamo `3`
- reminder: `[3;5]` è una abbreviazione sintattica per `3::[5]`

Listing 3.21: Esempi di pattern matching

```
# let hd (h::t) = h;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val hd : 'a list -> 'a = <fun>
# hd [];;
Exception: Match_failure ("//toplevel//", 1, 7).
```

Listing 3.22: Warning riferito al pattern `[]`

```
(* functions defined by two cases *)
let rec length l = match l with
    [] → 0
    | h::t → 1+length t;; (* h and t are local variables *)
let rec sum l = match l with
    [] → 0
    | h::t → h+sum t;; (* h and t are local variables *)
(* function defined by three cases *)
let swap l = match l with
    [] → []
    | [x] → [x] (* x is a local variable *)
    | x::y::t → y::x::t;; (* x, y and t are local variables *)
```

Listing 3.23: Esempi di pattern matching con diversi pattern

Un singolo pattern può non essere sufficiente per tutti i validi argomenti di una funzione. Per esempio in questo caso `[]` e `h::t` non matchano per alcuna sostituzione di `h` e `t`, questo è corretto perché la testa di una lista è undefined per una lista vuota.

La sintassi di una espressione per matchare un value con multipli pattern è:

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

3.8.9 Semantica del pattern matching

`match e with p1 → e1 |...| pn → en`

Semantica statica

- l' espressione `e` e i pattern `p1...pn` devono avere lo stesso tipo
- tutte le espressioni `e1...en` devono avere lo stesso tipo

b

Semantica dinamica

- viene valutata `e`
- vengono testati tutti i pattern `p1...pn` left-to-right e top-to-bottom
- se `p1` è il primo pattern per cui `e` e `p1` sono matchati l' espressione `e1` è valutata, con le variabili definite dalla sostituzione per il match

```
# let rec length l = match l with
hd::tl → 1+length tl
| [x] → 1
^^^
| [] → 0;;
Warning 11: this match case is unused.
```

Listing 3.24: Check di semantica statica, unused case

- se non c'è nessun match viene sollevata una eccezione `Match_failure`

Oltre allo warning ottenuto precedentemente (il pattern non comprendeva un caso) possiamo ottenere un warning se un pattern **non è usato**, ad esempio:

Come detto precedentemente i costruttori di tipo garantiscono che se esiste un match con `p` esiste **una sola sostituzione** per le variabili in `p`, ecco un controesempio:

```
# fun ls → match ls with l1@l2 → l1;; (* @ is not a constructor! *)
Error: Syntax error
```

Se `ls` fosse `[1;2;3]` quale sarebbe il value di `l1`?

- `[]`??
- `[1]`??
- `[1;2]`??
- `[1;2;3]`??

Tutti i **literals** (token che rappresentano valori) sono costruttori costanti (piuttosto inutile), ecco alcuni esempi:

```
let mynot b = match b with false → true | true → false;;
let iszero i = match i with 0 → true | any → false;;
(* a variable is needed for the second pattern *)
```

Una sintassi più compatta prevede:

- uso di una wildcard (`_`) come pattern che matcha tutti i valori quando una variabile non è necessaria
- una sintassi abbreviata `function p1 → e1 |...| pn → en` per rappresentare `fun var → match var with p1 → e1 |...| pn → en`
- `p as id`: un pattern `p` può esser associato con un`id` per riferiri al valore matchato più semplicemente sul lato destro della →

```
let mynot = function false -> true | _ -> false;;
let iszero = function 0 -> true | _ -> false;;
let rec length = function _::tl -> 1+length tl | _ -> 0;;
let rec sum = function hd::tl -> hd+sum tl | _ -> 0;;
let swap = function x::y::l -> y::x::l | other -> other;;
let ord_swap = function
    x::y::tl as l -> if x>y then y
```

Listing 3.25: Esempi di pattern matching

3.9 Ricorsione ed efficienza

Consideriamo la seguente funzione:

```
let rec sum = function
    hd::tl → hd + sum tl (* inductive case *)
    | _ → 0;; (* base case [] *)

let l=List.init 100_000 (fun x→x+1)(* creates [1;2;...;100_000] *)
    in sum l;;
- : int = 5000050000

let l=List.init 1_000_000 (fun x→x+1)(* creates [1;2;...;1_000_000] *)
in sum l;;
```

```
Stack overflow during evaluation (looping recursion?).
```

Listing 3.26: Applicazioni della funzione che somma gli elementi in una lista

```
let rec reverse = function
    hd::tl → reverse tl @ [hd] (* inductive case *)
    | _ → [];; (* base case [] *)

let l=List.init 10_000 (fun x→x+1)(* creates [1;2;...;10_000] *)
    in reverse l;; (* it takes time! *)
```

Listing 3.27: Applicazioni della funzione che reversa una lista

Questa funzione richiede molto tempo per la sua esecuzione, la sua complessita' è infatti $\Theta(n^2)$

```
(* Fibonacci numbers and Binomial coefficients *)
let rec fib n = if n≤1 then n else fib(n-2)+fib(n-1);;
let rec bin n k = if n=k||k=0 then 1 else bin(n-1)(k-1)+bin(n-1) k;;
```

Listing 3.28: Due funzioni con complessita' $\Theta(n!)$

Nella programmazione imperativa questi problemi sono risolti implementando iterativamente le funzioni, tuttavia nella programmazione funzionale non abbiamo a disposizione **statement** quali `for` e `while`, si devono quindi usare tecniche differenti.

3.9.1 Accumulatori

```
(* example with imperative programming, this is not OCaml ! *)
sum(l){
    acc=0; (* initial value of the accumulator *)
    while(l and hd::tl match){
        acc=acc+hd;
        l=tl;
    }
(* if l and hd::tl do not match, l is empty, acc is returned *)
return acc;
}
```

```
let rec aux acc = function (* aux : int -> int list -> int *)
    hd::tl -> aux (acc+hd) tl
    | _ -> acc
    in aux 0;; (* aux 0 : int list -> int *)
```

Listing 3.29: Implementazione in OCaml (sotto) di una procedura iterativa

3.9.2 Tail recursion

Tail recursion significa che nel body della funzione ricorsiva l' ultima operazione che deve essere valutata prima di riuscire a calcolare il valore finale della funzione è proprio la chiamata ricorsiva.

Nella prima funzione l' ultima operazione è l' addizione, nella seconda funzione prima calcoliamo la somma, poi chiamiamo la funzione ricorsiva.

```
let rec sum = function
    hd::tl -> hd + sum tl (* last operation: addition *)
    | _ -> 0;;
```

```
let rec aux acc = function
    hd::tl -> aux (acc+hd) tl (* last operation: recursion *)
    | _ -> acc
in aux 0;;
```

Listing 3.30: Funzione non tail recursive (sopra) e tail recursive (sotto)

3.9.3 Accumulatori e Tail Recursion

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

let l=List.init 1000000 (fun x->x+1) (* creates [1;2;...;1_000_000] *)
in acc_sum l;;
- : int = 500000500000
```

Listing 3.31: Implementazione più efficiente della funzione sum

Osserviamo che:

- **aux** e' tail recursive con un accumulatore ed e' neccesaria per definire `acc_sum`
- `acc_sum` fa una applicazione della funzione `aux` e decide il valore iniziale di `acc` (0)

Funzione polimorfa

```
let acc_rev l = (* parameter l needed to get a polymorphic function *)
    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)(* creates list [1;2;...;10_000] *)
in acc_rev l;;

acc_rev [1;2;3];;
- : int list = [3; 2; 1]
acc_rev [true;true;false];;
- : bool list = [false; true; true]
```

Listing 3.32: Implementazione più efficiente della funzione reverse (complessita' lineare)

La funzione `acc_rev` è polimorfa, la possiamo applicare a tutti i tipi di liste quindi, se dimentichiamo di passare il paramentro non funziona l' inferenza dei

tipi. Tecnicamente ha tipo `'_weak1 list -> '_weak1 list` , ovvero assume un tipo specifico in base al tipo della lista alla prima chiamata.

Funzione non polimorfa

```
let no_poly_rev = (* no_poly_rev : '_weak1 list -> '_weak1 list *)
    let rec aux acc = function
        hd::tl -> aux (hd::acc) tl
        | _ -> acc
    in aux [] ;;
    no_poly_rev [1;2;3]
    - : int list = [3; 2; 1]
    no_poly_rev [true;true;false]
```

Listing 3.33: Funzione non polimorfa

3.10 Stringhe in OCaml

Le stringhe sono un tipo primitivo in OCaml, `""` rappresenta la stringa vuota, `"i love bears"` è una stringa non vuota. Sulle stringhe è definita la concatenazione `^`, che ha precedenza minore della applicazione di funzione, per approfondire: https://caml.inria.fr/pub/docs/manual-ocaml/libref/String.html

```
let s="hello"^" "^"world";;
val s : string = "hello world"
(^);;
- : string -> string -> string = <fun>
String.length s;;
- : int = 11
String.uppercase_ascii s;;
- : string = "HELLO WORLD"
String.lowercase_ascii "HELLO WORLD";;
- : string = "hello world
```

Listing 3.34: Esempi sulle stringhe

3.11 Funzioni generiche sulle liste

3.11.1 Funzione `map`

`List.map: ('a → 'b) → 'a list → 'b list`, ad esempio
`List.map f [x1,...,xn]=[fx1,... fxn]`

```
let map f =
    let rec aux acc = function (* acc contains a list *)
        hd::tl → aux (f hd::acc) tl (* put f hd on the head of acc*)
        | _ → List.rev acc (* reverses the list *)
    in aux [];;
```

Listing 3.35: Una implementazione efficiente usando la ricorsione di coda

- :: determina una complessita' lineare nella lunghezza della lista
- la lista deve essere reversata prima o dopo la applicazione di `aux`

```
map ((+)1) [1;2;3];;
- : int list = [2; 3; 4]
map String.length ["apple"; "orange" ];;
- : int list = [5; 6]
map String.uppercase_ascii ["apple"; "orange" ];;
- : string list = ["APPLE"; "ORANGE"]
```

Listing 3.36: Esempi di applicazione della funzione `map`

3.11.2 Funzione `fold_left`

La funzione `fold_left` e' un pattern generico per funzioni che operano sulle liste con un accumulatore.
`List.fold_left: ('a → 'b → 'a) → 'a →'b list →'a`, ad esempio
`List.fold_left f a0 [x1;...;xn] = an` dove:

- `a0` e' il valore iniziale dell' accumulatore
- `a1=f ao x1`
- `an=f an-1 xn`

`f: 'a -> 'b -> 'a` e' usato per combinare:

- il valore corrente dell' accumulatore `acc` di tipo `'a`
- l' elemento corrente della lista `hd` di tipo `'b`

per ottenere il nuovo valore dell' accumulatore di tipo `'a`

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

Listing 3.37: Una possibile implementazione efficiente usando la ricorsione di coda

```
let sum_list = fold_left (+) 0;; (* (+):int -> int -> int *)
val sum_list : int list -> int = <fun>
sum_list [1;2;3;4];;
- : int = 10
let prod_list = fold_left ( * ) 1;; (* ( * ):int -> int -> int *)
val prod_list : int list -> int = <fun>
prod_list [1;2;3;4];;
- : int = 24
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
```

Listing 3.38: Esempi di uso della funzione `fold_left`

3.12 Exceptions

Principi importanti di ingengeria del software:

- Fault e comportamenti imprevedibili **devono essere segnalati il prima possibile** e **devono essere gestiti al momento giusto**
- I crash **devono essere evitati**, se possibile

Una corretta implementazione del sistema delle eccezioni permette di distinguere chiaramente il comportamento corretto del codice da quello errato, la normale esecuzione infatti **deve essere interrotta** non appena un errore viene riscontrato.

Valori ed eccezioni:

- un valore deve essere returnato solo quando una procedura e' completata correttamente
- le eccezioni sono **raised/thrown** quando una procedura non e' completata normalmente.
- quando una eccezione e' **throwata/raisata** nessun return value e' atteso

Esistono due meccanismi relativi alle eccezioni:

- **exception generation** con un meccanismo di propagazione di queste
- **exception handling**

3.12.1 Eccezioni in Ocaml

In Ocaml le eccezioni hanno tipo **exn** e sono create con costruttori.

- **exception generation:** `raise: exn → 'a`
- **exception handling:** `try e with p1 →e1 | ...| pn →en`

3.12.2 Dichiarazione del costruttore di eccezioni

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

`CONS_ID` deve iniziare con una lettera maisucola

```
exception Fault;; (* constant constructor *)
exception Fault1 of string;; (* a unary constructor *)
exception Fault2 of string*exn;; (* a binary constructor *)
let exc=Fault;;
let exc1=Fault1 "error message";;
let exc2=Fault2 ("msg",exc);;
```

Listing 3.39: Esempi di dichiarazione di costruttori di eccezioni

```
(* self-explanatory, no additional info *)
exception Division_by_zero;;
(* general exception with an error message *)
exception Failure of string;;
(* self-explanatory, associated info: function name *)
exception Invalid_argument of string;;
(* self-explanatory, associated info: file name, code line and column *)
exception Match_failure of string*int*int;;
let failwith msg = raise (Failure msg);; (* failwith : string -> 'a *)
```

Listing 3.40: Alcune funzioni predefinite e eccezioni

```
let hd = function
    hd::_ -> hd
    | _ -> failwith "hd";;

    (* control flow is changed: x+1 is not evaluated *)
let x=hd [] in x+1;;
Exception: Failure "hd".

(* element at index 4 not found *)
List.nth [1;2;3] 4;;
Exception: Failure "nth".

(* index -1 is not valid *)
List.nth [1;2;3] (-1);;
Exception: Invalid argument "List.nth".
```

Listing 3.41: Esempi di eccezioni in Ocaml

3.13 Variant types

I **variant types** permettono all' utente di definire nuovi tipi con i loro costruttori

```
type color = Red | Green | Blue;; (* just constant constructors *)

let to_string = function (* to_string : color → string *)
    Red → "red"
    | Green → "green"
    | Blue → "blue";;

List.map to_string [Red; Blue; Green; Blue];;
- : string list = ["red"; "blue"; "green"; "blue"]
```

Listing 3.42: Esempi di variant types solo con costruttori costanti

Gli identificatori di tipo devono iniziare con una **lettera minuscola**, mentre gli identificatori dei costruttori devono iniziare con una **lettera maiuscola**, inoltre i costruttori non possono essere **curried**.

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

    let perimeter = function (* perimeter : shape → float *)
        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 3.43: Esempi di variant types solo con costruttori non costanti

3.13.1 Numeri floating point in Ocaml

I numeri in virgola mobile in Ocaml sono rappresentati dal tipo `float`, sono definiti:

- gli operatori standard `+. -. *. /. **`
- le variabili globali `nan, infinity,neg_infinity`
- altre features nella `Stdlib` (importata implicitamente)
- altri features nel modulo `Float`

`int` e `float` non sono compatibili, Ocaml non fa alcuna conversione implicita:

```
(+) :  int → int → int
(+.) :  float → float → float

3.14 * 2;;
^^^^
Error: This expression has type float but an expression was
expected of type int
```

3.13.2 Recursive variant types

```
(*
    abstract syntax trees (ast) for expressions with
    -) integer literals
    -) unary -
    -) binary + and *
*)
type exp_ast =
  NumLit of int (* integer literals *)
  | Sign of exp_ast (* unary - *)
  | Add of exp_ast * exp_ast (* binary + *)
  | Mul of exp_ast * exp_ast (* binary * *);;
```

Listing 3.44: Implementazione di un abstract syntax tree

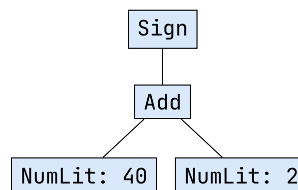


Figure 3.2: `let ast = Sign(Add(NumLit 40,NumLit 2));;`

```
type exp_ast =
  NumLit of int (* integer literals *)
  | Sign of exp_ast (* unary - *)
  | Add of exp_ast * exp_ast (* binary + *)
  | Mul of exp_ast * exp_ast (* binary * *);;

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

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

Listing 3.45: Evaluation degli abstract syntax tree

```
type 'a option = None | Some of 'a;;

(* get : 'a option -> 'a, see module Option *)
let get = function
    Some v -> v
    | _ -> raise (Invalid_argument "get");;

(* find : ('a -> bool) -> 'a list -> 'a option *)
let find p =
    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 3.46: Tipi varianti polimorfi

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;
(* member : 'a -> 'a btree -> bool *)
let rec member el = function
    Node(label,left,right) ->
        el=label || if el<label then member el left
                    else member el right
        | _ -> false;;
(* insert : 'a -> 'a btree -> 'a btree *)
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);;
```

Listing 3.47: Tipi varianti polimorfi e ricorsivi (BST)

Chapter 4

OOP

4.1 Introduzione: alcuni linguaggi OOP

- Simula (1965)
- Smalltalk (1972), sviluppato a Xerox PARC
- Eiffel (1986)

Tra i **linguaggi object oriented mainstream** ci sono:

- C++ (1985), Java (1995), C# (200) **statically typed**
- Python (1990), JavaScript/ECMAScript (1995) **dynamically typed**
- Scala (2004) ha una forte integrazione tra i paradigmi funzionale e O-O
- Kotlin (2011) e' un estensione di Java per Android

Sia Scala che Kotlin sono **statically typed** e basati sulla **JVM**

Il paradigma O-O si basa sulla nozione di **oggetto**, per esempio un oggetto `Timer` lo possiamo rappresentare in questo modo:

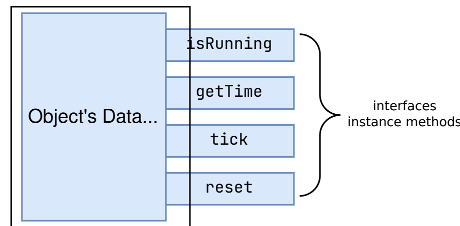


Figure 4.1: Rappresentazione dell' oggetto `Timer`

Questo rende chiaro che un oggetto e' separato dal contesto principale del programma (i dati interni sono nascosti), e' possibile accedere a questi attraverso delle interfacce che si chiamano **Instance Methods**.

4.2 Instance Methods

Per interagire con un oggetto facciamo delle **call** agli **instance methods** dell' oggetto (anche **method invocation/ message sending**), l' oggetto in questione e' il **target** o **receiver** degli **instance methods**. L' esecuzione di un instance method puo':

- modificare i dati (internal state) dell' oggetto
- passare alcuni argomenti
- returnare un valore

La sinstassi di una **method call** (basata sulla dot notation) e':

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

`MID` e' l' identifier dell' instance method, per esempio `timer.reset(42)` rappresenta la chiamata dell' instance method `reset` sul target `timer` con argomento `42`.

Nell' esempio di prima sono definiti 4 instance methods:

- `boolean isRunning()` che controlla se il countdown e' finito o meno, returna `true` se e solo se il time non ha raggiunto `time=0`
- `int getTime()` che returna il tempo rimanente espresso in secondi
- `void tick()` che decrementa di 1 secondo il tempo rimanente sul timer se e' maggiore di zero, non fa nulla altrimenti.

- `int reset(int minutes)` che resetta a `minutes` il time del timer, returna il time precedente in secondi, throwa un eccezzione se `minutes<0` o `minutes>60`

Possiamo distinguere i metodi precedenti in:

- **query methods:** `isRunning` e `getTiem` ispezionano lo stato interno dell' oggetto timer ma **non lo possono modificare**
- **modify method:** `tick` puo' modificare lo stato interno di `timer`
- `reset` ispeziona lo stato interno e lo puo' modificare

Lo stato interno di un oggetto:

- e' generalmente nascosto al mondo esterno
- consiste di **fields/instance variables/attributes**
- ha delle **instance variables** che contengono i dati di un oggetto
- ha dei **fields** che possono essere aggiornati.

Gli oggetti in Java (linguaggio che studiamo) sono definiti tramite delle **classi:**

4.3 Le classi

Le **classi** forniscono una implementazione per oggetti dello stesso tipo, gli oggetti infatti possono essere creati **dinamicamente** dadlle classi.

- gli oggetti creati dalla classe **C** sono chiamati **istanze** di `C`
- le istanze sono **deallocate** manualmente (C++) o automaticamente (garbage collection di Java, C#, Javascript, Python,...)
- tutte le istanze della stessa classe condividono gli stessi instance methods
- tutte le istanze della stessa classe hanno il loro internal state, le variabili di istanza **non sono condivise**
- gli oggetti creati dalle classi **C** hanno **dynamic type** C
- nei linguaggi **statically typed** una classe definisce anche uno **static type**

Le keyword `this` negli instance methods rappresenta l' oggetto target su cui la chiamata del metodo e' fatta.

```java
public class TimerClass {
    // private instance variables
    private int time;
    // public instance
    // "this" is the target object of the method
    public boolean isRunning() {
    return this.time > 0;
    }
    public int getTime() {
        return this.time;
    }
    public void tick() {
        if (this.isRunning())
        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 4.1: Rappresentazione in Java dell' oggetto `Timer`

```
// creates a new object of type TimerClass
TimerClass t1 = new TimerClass();
// calls instance method reset on the object in t1 with argument 1
t1.reset(1);
// creates a new object of type TimerClass
TimerClass t2 = new TimerClass();
// calls instance method reset on the object in t1 with argument 2
t2.reset(2);
```

Listing 4.2: Usi della `TimerClass`

`t1` e `t2` hanno **static type** `TimerClass`.
La creazione degli e' determinata da queste regole:

- **Sintassi:** `'new' CID '(' (Exp ( ',' Exp)*)?' )'`
- **Semantica:** una nuova classe `CID` e' creata dinamicamente, gli argomenti sono usanti per inizializzare i field dell' oggetto appena creato.

Chapter 5

Esercizi

5.1 Espressioni Regolari

Definire le espressioni regolari per i seguenti linguaggi:

5.1.1 Esercizio 1

1. numeri in base 8: iniziano con ’0’, seguito da zero o più cifre ottali;
2. numeri in base 16: iniziano con ’0’, seguito da ’x’ oppure ’X’, seguito da una o più cifre esadecimali (ossia le cifre decimali e le lettere minuscole o maiuscole da ’a’ a ’f’);
3. numeri in base 10 in formato normale o lungo: iniziano con ’0’ solo se significativo, altrimenti iniziano con una cifra decimale diversa da ’0’ seguita da zero o più cifre decimali; opzionalmente terminano con ’L’ o ’l’ (lettera elle maiuscola o minuscola).

1. `0[0-7]*`
2. `0[xX][0-9a-fA-F]+`
3. `(0|[1-9][0-9]*)[Ll]`

5.1.2 Esercizio 2

1. numeri floating-point in base 10: hanno lunghezza maggiore di 1 e devono contenere il carattere ’.’ ma una volta sola. Le stringhe prima e dopo il carattere ’.’ sono stringhe di cifre decimali di lunghezza arbitraria, anche zero.
2. costanti di tipo stringa: hanno lunghezza maggiore di 1, iniziano e terminano con il carattere doppio apice ’"’, la stringa in mezzo al doppio apice iniziale e finale è la concatenazione di un numero arbitrario (anche zero)

di stringhe che possono essere le stringhe di lunghezza due ”” o ”
” oppure qualsiasi stringa di lunghezza uno che non contiene i caratteri backslash ’é doppio apice ’”’.

1. Dividiamo in due casi:

   inizio stringa carattere: `[0-9]+\.[0-9]*`

   inizio stringa punto: `\.[0-9]+`

   Il risultato finale e’ `[0-9]+\.[0-9]*|\.[0-9]+`

2. Nelle stringhe le double-quote sono i delimitatori, se le vogliamo all’ interno della stringa le dobbiamo escapare, `"""` non e’ una stringa valida, in una stringa di un carattere le sintassi accettate sono `"\""` oppure `"\\"`

   Il risultato finale e’ `"([^"\\]|\\["\])*"`

5.2 Grammatiche Context Free

Data la seguente grammatica CF

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

5.2.1 Esercizio

1. Usando la nozione di derivazione a uno o più passi, mostrare che:

   La stringa "0*1+0" appartiene al linguaggio generato da Exp;

   La stringa "1+(" non appartiene al linguaggio generato da Exp.

2. Mostrare che esistono diversi modi per derivare da Exp in uno o più passi la stringa "0*1+0";

3. Mostrare che esistono diversi modi per derivare da Exp in uno o più passi la stringa "0*1+0" anche se a ogni passo si rimpiazza sempre il simbolo non-terminale che compare più a sinistra (left-most strategy).

1.1

```
Exp ->
Exp + Exp ->
Exp * Exp + Exp ->
Num * Exp + Exp ->
Num * Num + Exp ->
Num * Num + Num ->
'0' * '1' + '0'
```

Abbiamo dimostrato che la stringa deriva da questa grammatica.

1.2 Per dimostrare che una stringa non deriva da un linguaggio dobbiamo generalizzare

```
Exp ->
Exp + Exp ->
Num + Exp ->
1 + Exp ->
1 + (Exp)
```

A questo punto abbiamo dimostrato che dopo avere aperto una tonda la dobbiamo necessariamente chiudere.

2 Per dimostrare che esistono diversi modi per rappresentare una stringa e' sufficiente fare una derivazione left-most e una right-most.

3 Il grado di liberta' piu' interessante nelle derivazioni one-step non e' quale non-terminale sostituire, ma la produzione che si sceglie

```
Exp ->                      Exp ->
Exp * Exp ->                Exp + Exp ->
Num * Exp ->                Exp * Exp + Exp ->
0 * Exp ->                  Num * Exp + Exp ->
0 * Exp + Exp ->            0 * Exp + Exp ->
0 * Num + Exp ->            0 * Num + Exp ->
0 * 1 + Exp ->              0 * 1 + Exp ->
0 * 1 + Num ->              0 * 1 + Num ->
0 * 1 + 0                   0 * 1 + 0
```

Abbiamo dimostrato che esistono due diverse derivazioni per la stessa stringa, quindi la grammatica e' ambigua.

5.2.2 Esercizio 2

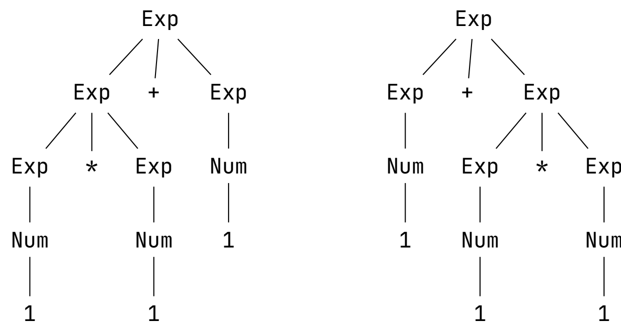
1. Data la seguente grammatica dimostrare che esistono due diversi alberi di derivazione per la stringa `"1*1*1"` a partire da `Exp`

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

2. Data la seguente grammatica dimostrare che e' ambigua per `Exp`

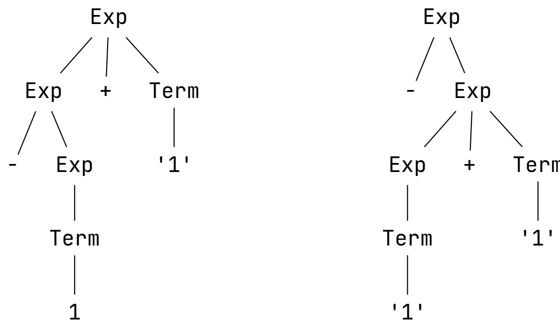
```
Exp::= Term | Exp '+' Term | '-' Exp | '(' Exp ')'
Term::= '0' | '1' | '-' Term
```

1



2

Consideriamo la stringa `-1+1`



Il primo albero da la precedenza al cambio di segno (piu' in basso nell' albero), mentre il secondo da precedenza alla addizione.

5.3 Disambiguazione delle grammatiche

5.3.1 Esercizio 1

Disambiguare la seguente grammatica ambigua trasformandola in equivalenti non ambigue che codifichino le seguenti regole sintattiche:

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

1. * ha precedenza su + e associa da sinistra, + associa da destra
2. * ha precedenza su + e associa da destra, + associa da sinistra
3. + ha precedenza su * e associa da sinistra, * associa da sinistra
4. + ha precedenza su * e associa da sinistra, * associa da destra

L' idea di base e' quella di aggiungere un livello alla grammatica per eliminare le ambiguita'. Quando definiamo un non terminale se lo mettiamo a sinistra significa che forziamo la associativita' dell' operazione a sinistra, inoltre come standard introduciamo i non terminali con precedenza piu' bassa piu' in alto.

1

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

2

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

3

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

4

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

5.3.2 Esercizio 2

Disambiguare la seguente grammatica ambigua trasformandola in una equivalente non ambigua in modo che lo statement `if` abbia precedenza sulla sequenza di statement.

```
Stmt::= ID'='Exp';'|'if''('Exp')'Stmt|Stmt Stmt|'{'Stmt'}'
Exp::= ID | BOOL
```

Consideriamo la stringa `if(true)x=true; y=false`

```
Stmt::=NoSeq | Stmt NoSeq
NoSeq::= 'if' 'Exp' NoSeq | '{' Stmt '}'
Exp::= ID|BOOL
```

`NoSeq` significa tutte le produzioni ad eccezione della seequenza di statement.

5.4 Esercizi su sintassi di funzioni anonime e applicazione in OCaml

Considerare la seguente grammatica ambigua EBNF che definisce la sintassi semplificata di OCaml:

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

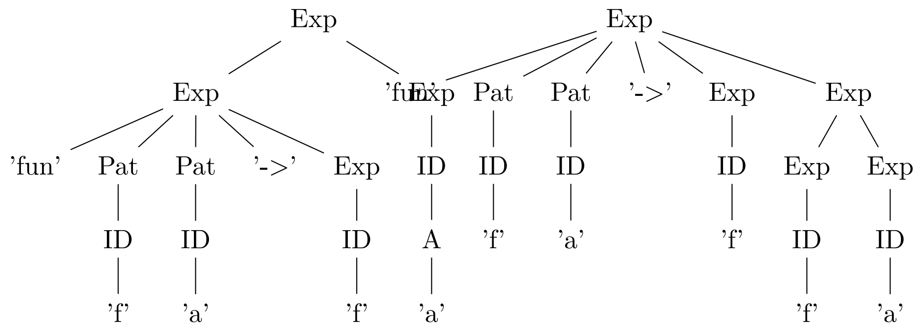
`Exp Exp` corrisponde alla applicazione di funzione.

I literal `ID, NUM, UOP e BOP` sono definiti tramite espressioni regolari convenzionali.

Mostrare che per ognuna delle seguenti stringhe esistono, a partire da `Exp`, diversi alberi di derivazione e indicare quali corrispondono alle regole di precedenza di OCaml:

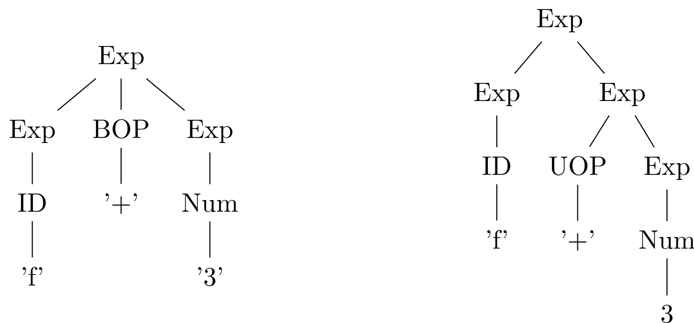
1. `"fun f a->f a"`
2. `"f+3"`
3. `"+f 3"`
4. `"(fun x->x*2) 1+2"`
5. `"(fun x y->x+y)3 4"`

1



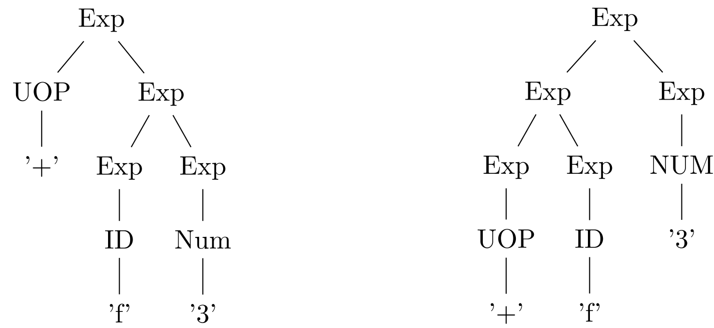
Il secondo albero e' quello corrispondente alle regole sintattiche di Ocaml

2



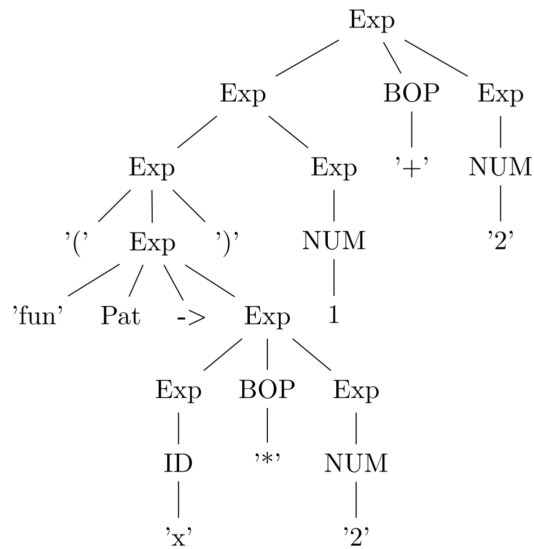
Il primo albero e' quello corrispondente alle regole sintattiche di Ocaml.

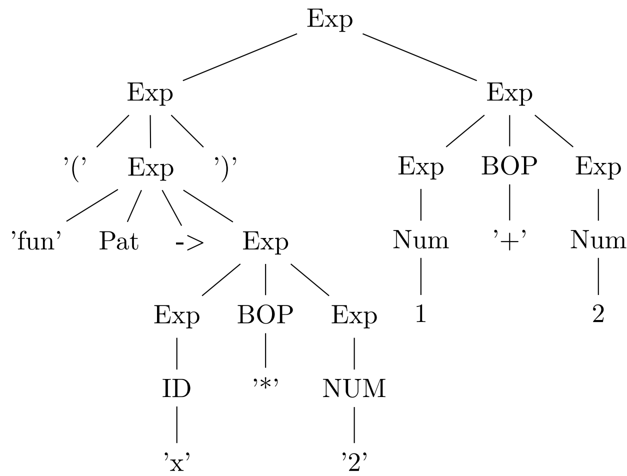
3



Il primo albero e' quello corrispondente alle regole sintattiche di Ocaml.

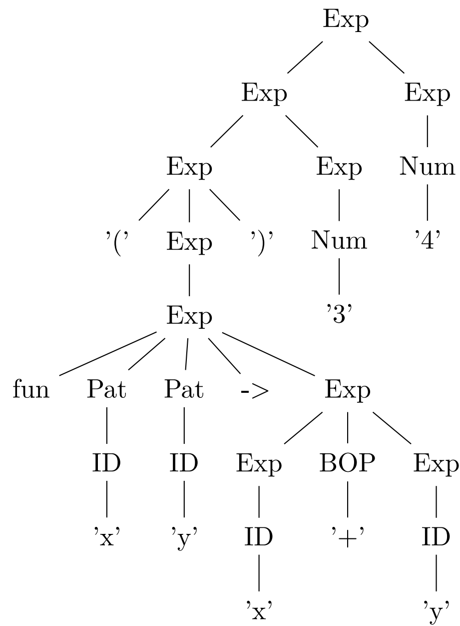
4

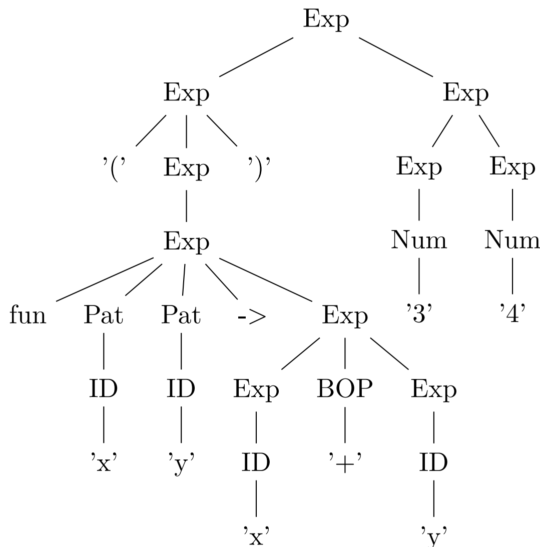




Il primo albero e' quello corrispondente alle regole sintattiche di Ocaml.

5





Il primo albero e' quello corrispondente alle regole sintattiche di Ocaml(associativita' applicazione)

5.5 Esercizi su funzioni higher order

Definire in OCaml funzioni anonime che abbiano i seguenti tipi:

1. `(int→int)→int`
2. `int→int→int`
3. `(int→int)→(int→int)`

Ricordiamo che l' operatore → associa da destra.

1. l' argomento di queste funzioni sono a loro volta funzioni (argomento: intero e tipo di ritorno intero) e restituiscono un intero

   `fun f→ f 0 + 1`, dove `f 0` rappresenta l' applicazione di f all' argomento `0`, questo ci dice che `f` deve essere una funzione e che ritorna un intero (vogliamo sommare questo valore ad 1)

   `fun f→f(f 0)`, dal momento che `f` prende in input `0` e che il valore restituito da `f` diventa input per `f` nuovamente necessariamente il valore restituito da `f` deve essere `int`

2. classico esempio di funzione curried

   `fun x y → x+y` abbreviazione per `fun x → fun y → x+y`

3. `fun f → fun x → f(x+1) - f x` abbreviabile come `fun f x → f(x+1)- f x`

5.6 Esercizi su funzioni higher-order ricorsive

Considerare la funzione `gen_sum` presentata a lezione:

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

`gen_sum f n` restituisce `f 0 + f 1 + ... + f n`

1. Aggiungere un parametro `min` in modo che `gen_sum f min n` restituisca `f min+f (min+1)+...+ f n.`
2. Ottenere come specializzazione della nuova funzione `gen_sum` la funzione `sumsquare` che calcola la somma dei primi n quadrati a partire da 1.
3. Aggiungere l'ulteriore parametro `step` in modo che `gen_sum f min step num` restituisca `f min+f (min+step)+...+f (min+(num-1)*step)`.

1. `gen_sum f 3 5 = f3 f4 f5`

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

2.
```
let sumsquare = gen_sum (fun x-> x*x) 1
```

3.
```
let gen_sum f min step num =
    let rec aux n = if n<min then 0
                    else f n+aux (n-step)
    in aux;;
```

esericizi di oggi

- usiamo il costrutto esplicito `match`, al posto di un **function pattern matching** per potere valutare due liste insieme, usando la scrittura con `function`
- **caso #1** entrambe le liste sono non-vuote
- **caso #2** almeno una lista e' non vuota (negazione del caso precedente)
- **caso #3** dal momento che non sono stati valutati i casi precedenti possiamo evitare la scrittura `l, []` che sarebbe ridondante.

```
let rec zip l1 l2 =
    match l1,l2 with
        h1::t1, h2::t2 -> h1::h2::zip t1 t2 (* caso #1 *)
        |[], l -> l (* caso #2 *)
        |l, _ -> l (* caso #3 *)
```

5.6.1 Esercizi sulla ricorsione di coda

Definire in OCaml le seguenti funzioni mediante ricorsione di coda e uso del parametro di accumulazione.

1. Funzione `gen_sum_list : ('a → int) → 'a list → int`,versione per liste di `gen_sum` vista a lezione: ,ad esempio`gen_sum_list f [v1;...;vn]=f v1+...+f vn`

2. Funzione `append : 'a list → 'a list → 'a list` che concatena due liste senza usare @, ad esempio `append [1;2;3] [4;5;6] = [1;2;3;4;5;6]`

1.
```
let gen_sum_list f =
    let rec aux acc = function
        hd::tl → aux(f hd+acc) tl
        |_ → acc
    in aux 0 ;;
```

   `gen_sum_list (fun x→ x+1) [1;2;3] = [9]`

2.
```
let append l1 l2 =
    let rec aux acc = function
        hd::tl → aux (hd::acc) tl
        |_ → acc
        in List.rev(aux (aux [] l1) l2)
```

   Osservazione: inserire in cima alla lista ha una complessita' costante, lo facciamo per questo motivo e poi facciamo il reverse (anch' esso lineare)

5.6.2 Esercizi su funzioni generiche su liste

1. Definire tramite `List.map` la funzione `is_pos_list : int list → bool list` che restituisce la lista di booleani ottenuti controllando se è positivo ogni elemento della lista passata come argomento, ad esempio `is_pos_list [-1;0;2;3;0]=[false;false;t`

2. Definire con `List.fold_left` la funzione `gen_sum_list : ('a → int) → 'a list → int`, ad esempio `en_sum_list f [v1;...;vn]`f v1+...+f vn=

3. Definire `List.map` usando `List.fold_left` e `List.rev`.

1. `let is_pos_list=List.map(fun x → if x >0 then true else false)`, o meglio:
   `let is_pos_list=List.map(fun x → x >0 )`

2. `let gen_sum_list f =List.fold_left (fun acc hd → acc + f hd)`

3. `let map f l =List.fold_left( fun acc hd → f hd::acc)[](List.rev l)`

5.7 Esercizi sui tipi varianti

Definire la funzione `area:shape →float` che calcola l' area di una figura geometrica, dove il tipo variante `shape` e' cosi' definito:

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

```
let area=function
    Square side → side ** 2.0
    |Circle radius → Float.pi *. radius **2.0
    |Rectangle (width, height) → width *. height (* e' una tupla *)
```

Definire la funzione `eval : bool_exp_ast → bool` che valuta l'albero della sintassi astratta di un'espressione booleana, dove il tipo variante `bool_exp_ast` è così definito:

```
type bool_exp_ast = BoolLit of bool | Not of bool_exp_ast |
                    And of bool_exp_ast * bool_exp_ast |
                    Or of bool_exp_ast * bool_exp_ast;;
```

Esempio:

```
(* ast di !(true && (false || false)) *)
let ast = Not(And(BoolLit true,Or(BoolLit false,BoolLit false)));;
eval ast = true;;
```

```
let rec eval = function
    BoolLit b → b
    |Not ast → not (eval ast)
    |And (ast1, ast2) → (eval ast1) && (eval ast2)
    |Or (ast1, ast2) → (eval ast1) || (eval ast2)
```

Ovviamente la valutazione di `&&` e `||` avviene in short-circuit, se vogliamo imporre alla funzione di valutare sempre entrambi i termini la cosa piu' semplice da fare e' assegnare i valori delle operazioni a delle variabili ausiliarie, ovvero:

```
let rec eval = function
    BoolLit b → b
    |Not ast → not (eval ast)
    |And (ast1, ast2) → let b1=eval ast1 and b2=eval ast2 in b1&&b2
    |Or (ast1, ast2) → let b1=eval ast1 and b2=eval ast2 in b1||b2
```