LINGUAGGI E PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

RICCARDO CEREGHINO

Appunti

Settembre 2019 – classicthesis v4.6

INDICE

ELENCO DELLE FIGURE

LISTINGS

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*.

Parte I

SINTASSI

2

STRINGHE

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

Definizione 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 : $length(u) = n$.*

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

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

2.1.3 *Concatenazione di stringhe*

Definizione 4

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

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

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

2.1.3.1 *Monoide*

La concatenazione è associativa, ma non commutativa.
La stringa vuota è l'identità dell'elemento.

2.1.3.2 *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*

Definizione 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

Definizione 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:

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

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

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*

2.2.2.1 *Unione*

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

ESEMPIO:

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

2.2.2.2 *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 $L^\star = L^0 \cup L^+$, $L \cdot L^\star$.

2.2.5.1 *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*.

3 ESPRESSIONI REGOLARI

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

Definizione 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;
- $e_1e_2 \rightsquigarrow$ 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.

3.2.3.1 *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.
- $n \rightsquigarrow \{"n"\}$,
- $\backslash n \rightsquigarrow \{"linefeed"\}$,

3.2.3.2 *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 ($[\wedge \backslash w]$).

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

3.3 ANALISI LESSICALE

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

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

Definizione 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

Definizione 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:

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

4 ANALISI SINTATTICA

Definizione 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

Definizione 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*.

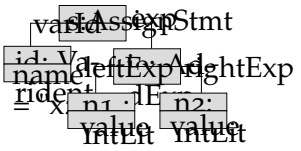


Figura 4.1: L'AST generato della sequenza precedente.

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**;
- $\{(Exp, Num), (Exp, Exp, '+' Exp), (Exp, Exp '*' Exp), (Exp, '('Exp')'), (Num, '0'), (Num, '1')\}$ è l'insieme P di produzioni.

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'
```

5.2.1.1 *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*

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

5.3.3.1 *Derivazione ad un passo*

Definizione 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.*

5.3.3.2 *Derivazione a più passi*

Definizione 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 $\gamma_1 \to^+ \gamma_3$.*

5.3.3.3 *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: $L_B = \{u \mid B \to^+ u\}$.

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'
```

6.1.1.1 *Albero di derivazione per "1*1+1"*

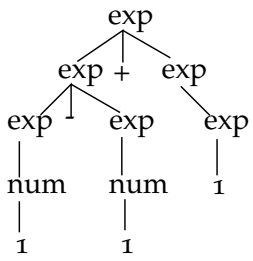


Figura 6.1: Albero di derivazione per "1*1+1""

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

$Exp \to Exp' +' Exp \to^+ Exp \times Exp' +' Num' \to^+ Num' *' Num' +' 1' \to^+ '1' '*' '1' '+' '1'$

ESERCIZIO: Mostrare che $''1 + (\notin L_{\text{Exp}}$:

$Exp \to Exp' *' Exp \to^+ Num' *' Exp' +' Exp \to^+ '1' '*'$ Num $'+'$ Num $\to^+ '1' '*' '1' '+' '1'$

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).

6.1.2.1 *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.

7 GRAMMATICHE AMBIGUE

Definizione 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'.

7.1.6.1 *Esempio 1: + e * con la stessa precedenza*

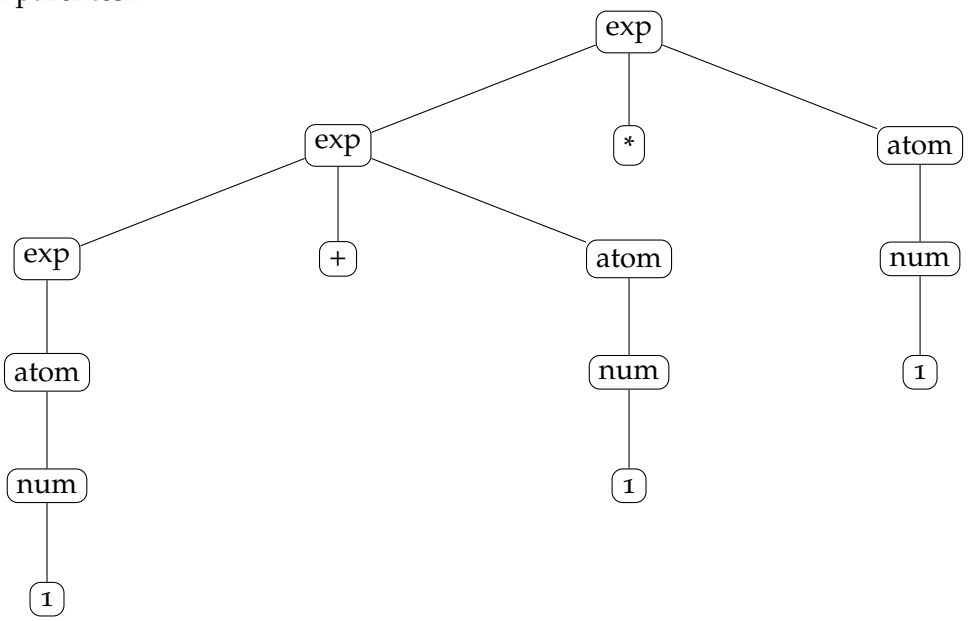
Listing 7.4: Grammatica ambigua

```
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'
```

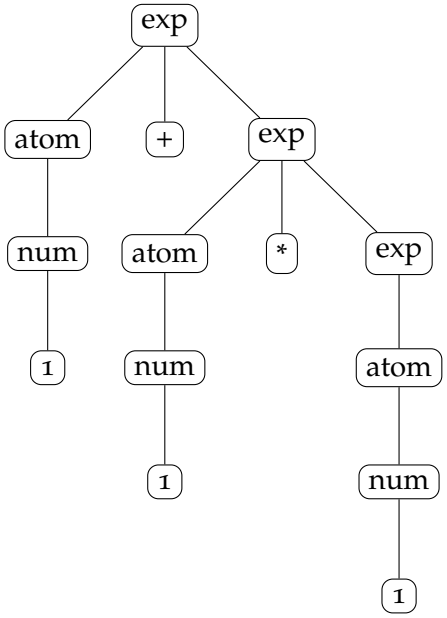
Nota che: $Exp' +' Atom | Exp' *' Atom$ significa che sul lato destro di $+(*)$, le addizione (e moltiplicazioni) sono permesse solo se circondate da parentesi.



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.



7.1.6.2 *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.

7.1.6.3 *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"

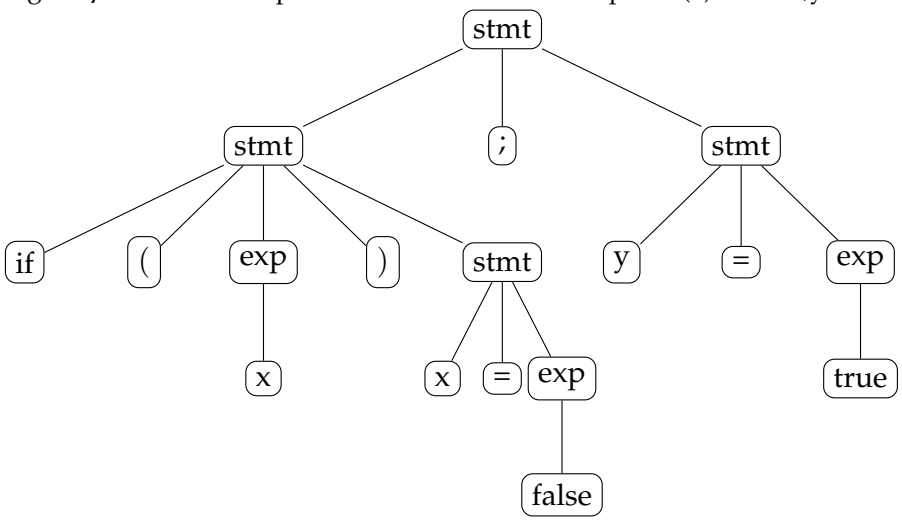
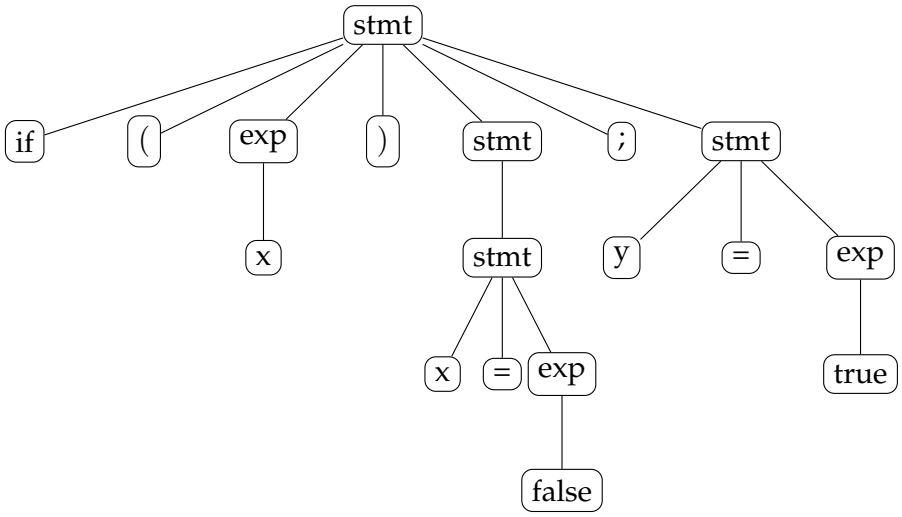


Figura 7.2: Secondo esempio di albero di derivazione per "if(x) x=false;y=true"



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

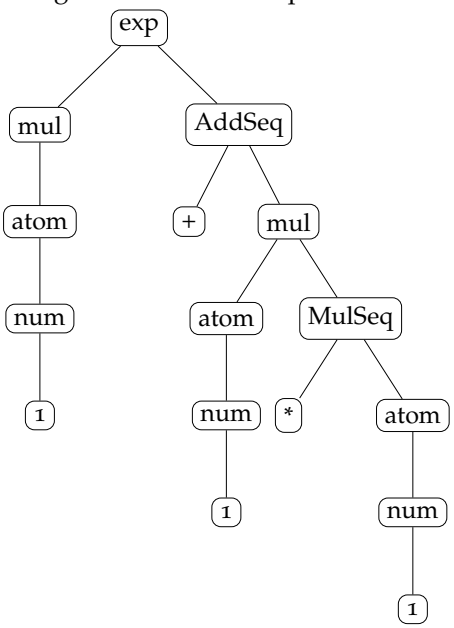
La grammatica è equivalente alla precedente, ma quando si costruisce un albero *parse* sappiamo che un nodo Exp deve sempre avere un figlio c a sinistra etichettato da Mul dopo che l'albero *parse* con radice c è stato costruito, quindi la produzione corretta è (Exp, Mul $AddSeq$) se il token *lookahead* 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 utilizzando 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 direvvamente 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()
}
```

8.2.1.1 *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()
}
```

9 PARADIGMI DI PROGRAMMAZIONE

Definizione 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).

Parte II

OCAML

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 $[+-*\backslash]|mod$;
- **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:

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

- dai punti prima deduciamo:

  `int->int->int` $\neq$ `int->(int->int)`

10.4.2 *Funzioni di alto ordine*

fun $pat_1 pat_2 \ldots pat_n$->exp

è un abbreviazione di:

fun pat_1->**fun** pat_2-> $\ldots$ **fun** pat_n->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

Definizione 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
```

10.8 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.

10.9 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.

10.9.1 *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.

10.10 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 $e_1 :: e_2 :: \cdots :: e_n$.

REGOLE SINTATTICHE

- Associatività a destra;
- minore precedenza degli operatori unari e binari con notazione infissa;
- maggiore precedenza del costruttore di tupla;

  ...

  ... .

10.10.1 *Tipi di costruttori per le liste*

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

-

10.11 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 10.28: 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.

10.11.1 *Esempi di pattern matching*

Listing 10.29: 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 10.30: 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 10.31: 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.

10.11.2 *Matching di patterns multipli*

Listing 10.32: Espressione per eseguire un match con multipli patterns

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

10.11.2.1 *Esempi*

Listing 10.33: Esempio di match multipli

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

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

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

10.11.2.2 *Sintassi*

Listing 10.34: 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.

10.11.3 *Decomposizione unica*

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

10.11.4 *Costruttori per i tipi primitivi*

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

10.11.5 *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 `|` $\ldots$ `|` $p_n e_n$

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

10.11.6 *Esempi di pattern matching funzionanti*

Listing 10.35: 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::l -> y::x::l | other -> other;;

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