Esercizi Grammatiche CF

Esercizio A

Si consideri l'insieme di simboli terminali $T = \{a, b\}$. Per ognuno dei linguaggi seguenti si dia una grammatica che lo generi

1. l'insieme delle stringhe della forma $a^n b^n$ con $n \geq 0$ (n volte a seguite da n volte b, e.g., `""`, `"ab"`, `"aabb"`, `...`)
2. l'insieme delle stringhe con lo stesso numero di a e di b (e.g., `""`, `"abba"`, `"baaabb"`, `...`)
3. l'insieme delle stringhe palindrome con a e b (e.g., `""`, `"a"`, `"b"`, `"abba"`, `"bbaabb"`, `...`)

Soluzione

1. `S ::= ""| aSb`
2. `S ::= ""| aSbS | bSaS`
3. `S ::= ""| a | b | aSa | bSb`

Esercizio B

Si consideri la seguente grammatica CF

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

1. Usando la nozione di derivazione a uno o più passi, mostrare che
   a) La stringa `"0*1+0"` appartiene al linguaggio generato da `Exp`;
   b) 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).

Considera la seguente variante della grammatica precedente

```
Exp1 ::= Exp1 '+' Exp1 | Exp2
Exp2 ::= Exp2 '*' Exp2 | Exp3
Exp3 ::= Num | '(' Exp1 ')'
Num  ::= '0' | '1'
```

4. La stringa `"0+1*0"` quante derivazioni left-most ammette a partire da `Exp1`?
   È derivabile a partire da `Exp2`?

5. La string `"0*1*0"` quante derivazioni left-most ammette a partire da `Exp1`?
   È derivabile a partire da `Exp2`?

6. Quali differenze osservi rispetto alla formulazione precedente della grammatica?

Considera la seguente ulteriore variante

```
Exp1 ::= Exp2 '+' Exp1 | Exp2
Exp2 ::= Exp3 '*' Exp2 | Exp3
Exp3 ::= Num | '(' Exp1 ')'
Num  ::= '0' | '1'
```

7. La stringa `"0+1*0"` quante derivazioni left-most ammette a partire da `Exp1`?
   È derivabile a partire da `Exp2`?

8. La string `"0*1*0"` quante derivazioni left-most ammette a partire da `Exp1`?
   È derivabile a partire da `Exp2`?

9. Quali differenze osservi rispetto alle formulazioni precedenti della grammatica?

Soluzioni

1. consideriamo i due punti separatamente
   a) la seguente è una possibile derivazione left-most

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

b) Supponiamo esista una derivazione e analizziamone la forma. Al primo passo abbiamo che

- la produzione `Exp ::= Num` non si può usare perché da `Num` si possono ottenere solo `"0"` e `"1"` che sono diverse da `"1+("`
- la produzione `Exp ::= Exp '*'Exp` non si può usare perché `"1+("` non contiene `'*'`
- la produzione `Exp ::= '('Exp ')'` non si può usare perché `"1+("` non inizia con `')'`

Quindi il primo passo di derivazione è necessariamente `Exp -> Exp '+'Exp`, ma per continuare la derivazione e ottenere la stringa desiderata, dal simbolo `Exp` a destra di `'+'` bisognerebbe derivare `"("`, che è impossibile (le uniche stringhe di lunghezza 1 derivabili da `Exp` sono `"0"` e `"1"`). Quindi la suddetta derivazione non può esistere.

2. la seguente è una possibile derivazione right-most

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

3. la seguente è un'altra derivazione left-most

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

4. la stringa `"0+1*0"` ammete solo la seguente derivazione left-most a partire da `Exp1` dato che la grammatica forza la precedenza del `'*'` sul `'+'`

```
Exp1 -> Exp1 '+' Exp1 ->+ '0' '+' Exp1
     -> '0' '+' Exp2 -> '0' '+' Exp2 '*' Exp2
     ->+ '0' '+' '1' '*' Exp2 ->+ '0' '+' '1' '*' '0'
```

perché il simbolo `'+'` al di fuori di parantesi è derivabile solo a partire da `Exp1`. Per la stessa ragione, la stringa non è derivabile da `Exp2`.

5. la stringa `"0*1*0"` ammette le seguenti due derivazioni left-most a partire da `Exp1`

```
Exp1 -> Exp2 -> Exp2 '*' Exp2 ->+ '0' '*' Exp2
     -> '0' '*' Exp2 '*' Exp2 ->+ '0' '*' '1' '*' Exp2
     ->+ '0' '*' '1' '*' '0'
```

```
Exp1 -> Exp2 -> Exp2 '*' Exp2 -> Exp2 '*' Exp2 '*' Exp2
     ->+ '0' '*' Exp2 '*' Exp2 ->+ '0' '*' '1' '*' Exp2
     ->+ '0' '*' '1' '*' '0'
```

Inoltre è derivabile a partire da `Exp2` come mostrano le derivazioni precedenti (entrambe come primo passo trasformano `Exp1` in `Exp2`).

6. Rispetto alla grammatica precedente, la stringa `"0+1*0"` ha una sola derivazione left-most, dato che la nuova grammatica forza la precedenza del `'*'` sul `'+'`.

7. la stringa `"0+1*0"` ha una sola derivazione left-most a partire da `Exp1` dato che la grammatica forza la precedenza del `'*'` sul `'+'`

   ```
   Exp1 -> Exp2 '+' Exp1 ->+ '0' '+' Exp1
        -> '0' '+' Exp2 -> '0' '+' Exp3 '*' Exp2
        ->+ '0' '+' '1' '*' Exp2 ->+ '0' '+' '1' '*' '0'
   ```

   Inoltre non è derivabile da `Exp2` per motivi analoghi alla grammatica precedente.

8. La stringa `"0*1*0"` ha una sola derivazione left-most a partire da `Exp1`, dato che la grammatica forza l'associatività a destra per il `'*'`

   ```
   Exp1 -> Exp2 -> Exp3 '*' Exp2 ->+ '0' '*' Exp2
        -> '0' '*' Exp3 '*' Exp2 ->+ '0' '*' '1' '*' Exp2
        ->+ '0' '*' '1' '*' '0'
   ```

   Questa derivazione mostra anche che la stringa è derivabile da `Exp2`.

9. rispetto alla grammatica precedente, anche la stringa `"0*1*0"` ammette una sola derivazione left-most perché la grammatica forza una regola di associatività a destra per il `'*'`.

Esercizio C

1. Data la grammatica

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

   mostrare che esistono due diversi alberi di derivazione per la stringa `"1*1*1"` a partire da `Exp`.

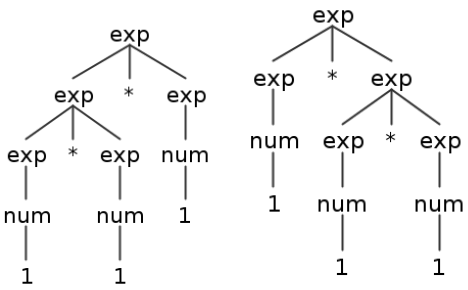
2. Data la grammatica

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

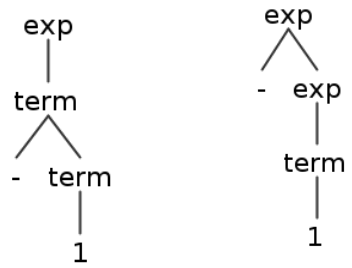
   mostrare che è ambigua per `Exp`.

Soluzioni

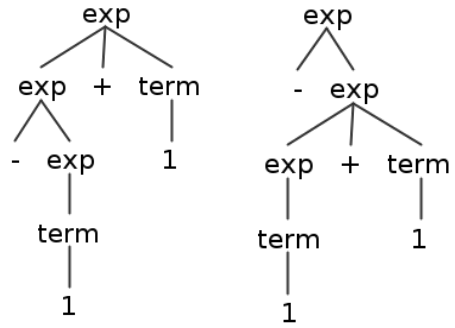
1. i seguenti sono due alberi di derivaizone che corrispondono ad associatività a sinistra e a destra rispettivamente



2. Nella grammatica esistono due sorgenti diverse di ambiguità:
   - Le produzioni `Exp ::= '-'Exp` e `Term ::= '-'Term` sono ridondanti, perciò esistono due diversi alberi di derivazione per semplici espressioni come `"-1"`:



   - La grammatica è ambigua rispetto alle regole di precedenza tra `'-'` unario e `'+'` binario; per esempio, per la stringa `"-1+1"` esistono questi due alberi di derivazione:



Esercizio D

Considera le seguenti grammatiche e per ognuna di queste

- se è ambigua, mostra una stringa con due alberi di derivazione
- se è non ambigua, spiega perché

1. ```
   Exp1 ::= Exp1 '+' Exp1 | Exp2
   Exp2 ::= Exp2 '*' Exp2 | Exp3
   Exp3 ::= Num | '(' Exp1 ')'
   Num  ::= '0' | '1'
   ```

2. ```
   Exp1 ::= Exp1 '+' Exp2 | Exp2
   Exp2 ::= Exp2 '*' Exp2 | Exp3
   Exp3 ::= Num | '(' Exp1 ')'
   Num  ::= '0' | '1'
   ```

3. ```
   Exp1 ::= Exp2 '+' Exp1 | Exp2
   Exp2 ::= Exp3 '*' Exp2 | Exp3
   Exp3 ::= Num | '(' Exp1 ')'
   Num  ::= '0' | '1'
   ```

4. ```
   Stmt ::= ID '=' Exp ';' | 'if' '(' Exp ')' Stmt | Stmt Stmt |
     '{' Stmt '}'
   Exp  ::= ID | NUM | BOOL
   ```

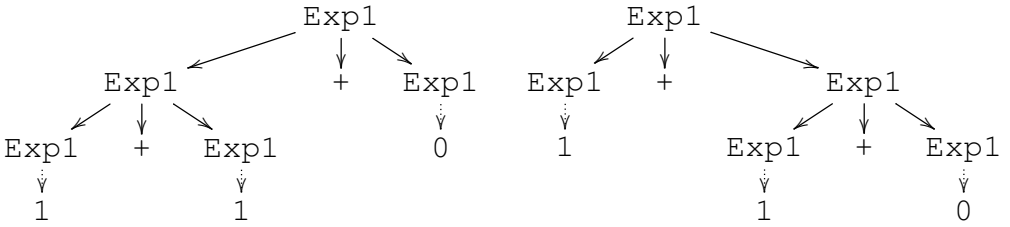
   (dove `ID`, `NUM`, `BOOL` descrivono i linguaggi, rispettivamente, degli identificatori, dei literal numerici e dei literal booleani)

5. ```
   Stmt1 ::= Stmt2 Stmt1 | Stmt2
   Stmt2 ::= 'if' '(' Exp ')' Stmt3 | Stmt3
   Stmt3 ::= ID '=' Exp | '{' Stmt1 '}'
   Exp   ::= ID | NUM | BOOL
   ```

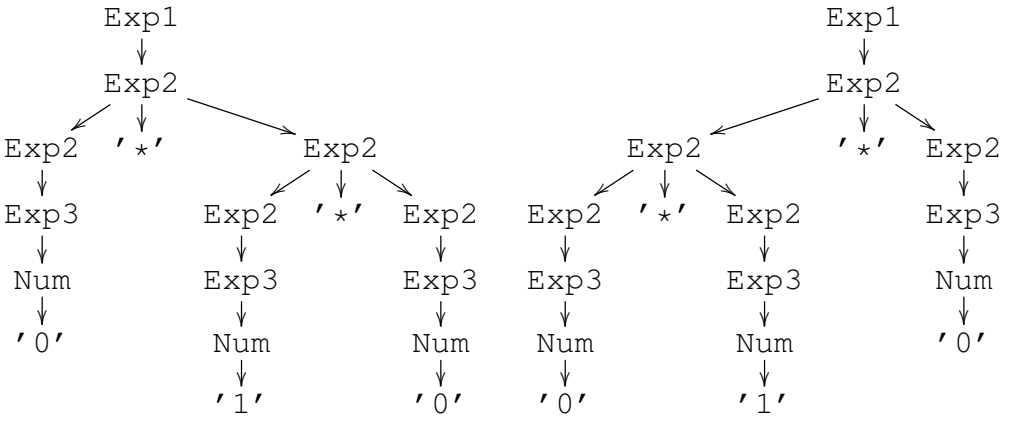
6. ```
   Exp1 ::= Exp2 | Exp1 '||' Exp2
   Exp2 ::= Exp3 | Exp2 '&&' Exp3 | '!' Exp2
   Exp3 ::= Bool | '(' Exp1 ')'
   Bool ::= 'true' | 'false'
   ```

Soluzione

1. la grammatica è ambigua: la stringa `"1+1+0"` ammette i seguenti alberi di derivazione

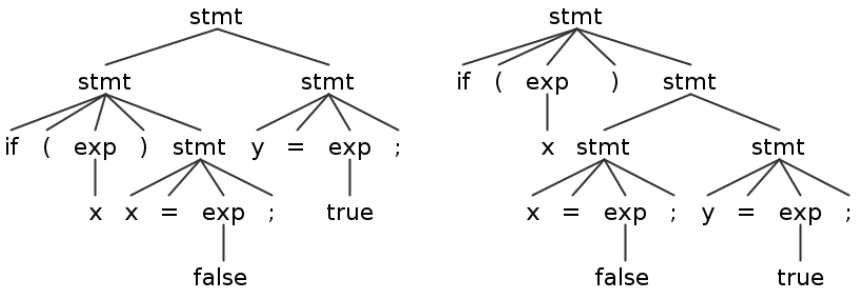


2. la grammatica è ambigua: la stringa `"0*1*0"` ammette due alberi di derivazione



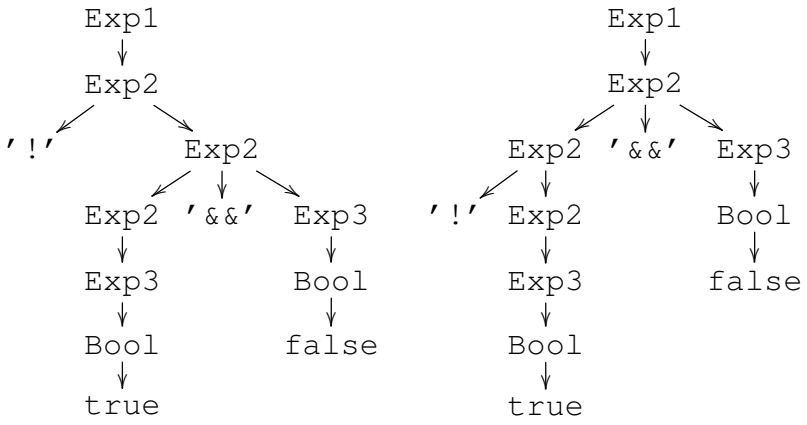
3. la grammatica non è ambigua: `'*'` ha precedenza su `'+'` ed entrambi sono associativi a destra.

4. la grammatica è ambigua: la stringa `"if (x)x = false; y = true;"` ammette i seguenti alberi di derivazione



5. la grammatica non è ambigua: la sequenza di statement associa a destra e l'**`if`** statement ha precedenza sulla sequenza.

6. la grammatica è ambigua: la stringa `"! true && false"` ammette i seguenti alberi di derivaizone



Esercizio E

1. Data la grammatica ambigua

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

   trasformarla in equivalenti grammatiche non ambigue che definiscono le seguenti regole sintattiche:

   a) `'*'` ha precedenza su + e associa da sinistra, `'+'` associa da destra

   b) `'*'` ha precedenza su `'+'` e associa da destra, `'+'` associa da sinistra

   c) `'+'` ha precedenza su `'*'` e associa da sinistra, `'*'` associa da sinistra

   d) `'+'` ha precedenza su `'*'` e associa da sinistra, `'*'` associa da destra

   e) `'+'` ha precedenza su `'*'` e associa da destra, `'*'` associa da sinistra

   f) `'+'` ha precedenza su `'*'` e associa da destra, `'*'` associa da destra

2. Trasformare la seguente grammatica ambigua in una equivalente non ambigua dove lo statement condizionale ha la precedenza sulla sequenza di statement `Stmt Stmt`.

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

3. Trasformare la seguente grammatica in una equivalente non ambigua dove le priorità sono date da
   `'!'> '&&'> '||'> '->'`
   e si abbia che `'&&'` e `'||'` associano a sinistra e `'->'` associa a destra

```
Exp  ::= Bool | Exp '->' Exp | Exp '||' Exp
       | Exp '&&' Exp | '!' Exp | '(' Exp ')'
Bool ::= 'true' | 'false'
```

Soluzione

1. elenchiamo i vari punti

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

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

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

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

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

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

2.
```
Stmt ::=  IfAssign |  IfAssign Stmt   // sequence of statements
IfAsign ::= ID '=' Exp ';' | 'if' '(' Exp ')' IfAssign |
        '{' Stmt '}'  // if or assignment statements or block
Exp ::= ID | BOOL
```

3.
```
Exp ::= Or | Or '->' Exp
Or  ::= And | Or '||' And
And ::= Not | And '&&' Not
Not ::= Atom | '!' Not
Atom ::= Bool | '(' Exp ')'
Bool ::= 'true' | 'false'
```