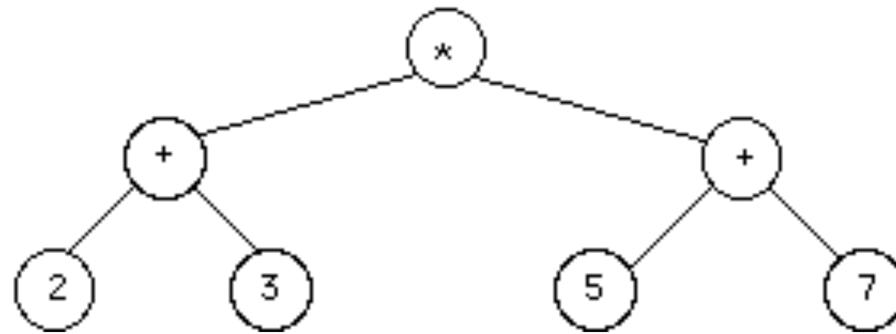
Espressioni Aritmetiche

La scrittura e il calcolo di espressioni aritmetiche sono problemi, a prima vista semplici, che nascondono insospettate complicazioni e si prestano ad una interessante trattazione. Prendiamo come esempio il calcolo della espressione aritmetica rappresentata dall'albero:



Nella notazione algebrica comunemente usata l'espressione si scrive:

(2+3)*(5+7)

il cui valore è 60. Eliminare le parentesi porterebbe all'espressione:

2+3*5+7

che comunemente viene valutata 24.

Tale valutazione deriva dall'avere attribuita precedenza all'operatore *. Senza tale precedenza l'espressione sarebbe ambigua. Nessuna ambiguità invece si introduce omettendo l'operatore *:

(2+3)(5+7)

e quest'ultima è la notazione più usata in algebra. Un ulteriore coppia di parentesi sarebbe ridondante ma renderebbe più omogenea la scrittura:

((2+3)*(5+7))

Le notazioni polacche

Coloro che hanno fatto esperienza di calcolo sulle calcolatrici Hewlett-Packard conosceranno già le cosiddette notazioni polacche,

- diretta: **`* + 2 3 + 5 7`**
- inversa: **`2 3 + 5 7 + *`**

Le notazioni polacche, seppur prive di parentesi, non sono ambigue purché siano chiaramente delimitate le costanti numeriche (qui si è usato il *blank* come separatore altrimenti la sequenza **`2`** seguito da **`3`** poteva essere confusa col valore **`23`**).

Infine, volendo evitare il cosiddetto *zucchero sintattico*, trattando, cioè, la somma e il prodotto come qualunque altra funzione di due variabili, si ha la notazione prefissa:

```
prodotto(somma(2,3), somma(5,7))
```

Nel seguito ci limitiamo per semplicità a trattare le operazioni di addizione e moltiplicazione tra numeri di una cifra. Presentiamo dapprima per ciascuna forma la grammatica generativa.

Le grammatiche delle varie notazioni

- La notazione infissa ha la grammatica:

  <numero> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  <espr> ::= (<espr> + <espr>) | (<espr> * <espr>) | <numero>

Questa grammatica prevede le parentesi ad ogni livello.

- Più complessa è la specifica di una grammatica non ambigua che, assegnando la precedenza all'operatore di moltiplicazione, permetta di eliminare le parentesi superflue:

  <numero> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  <espr> ::= <espr> + <term> | <term>

  <term> ::= <term> * <factor> | <factor>

  <factor> ::= <numero> | (<espr>)

Questa grammatica non sarà più trattata nel seguito.

- La notazione polacca diretta ha la grammatica:

<numero> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<espr> ::= + <espr> <espr> | * <espr> <espr> | <numero>

- Molto simile è la notazione prefissa:

<numero> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<espr> ::= somma(<espr> , <espr>) | prod(<espr> , <espr>) | <numero>

- La notazione polacca inversa ha la grammatica:

<numero> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<espr> ::= <espr> <espr> + | <espr> <espr> * | <numero>

Implementazione Java

Vediamo ora esempi di programmi di scrittura che trasformano la stringa in albero secondo la grammatica corrispondente. Questi programmi usano la classe `Espressione` che estende la classe `BinNode` che definisce gli alberi binari.

```
public class BinNode {
  public Object key;        // Campo chiave
  public BinNode left;      // Riferimento al figlio sinistro
  public BinNode right;     // Riferimento al figlio destro

  public BinNode() {}

  public BinNode(Object k, BinNode l, BinNode r) {
     if(k==null)throw new IllegalArgumentException();
     key = k;
     left = l;
     right = r;
  }

  public BinNode(Object k) {this(k, null, null);}

  public boolean isLeaf()  {return left==null && right==null;}
  public boolean twoSons() {return left!=null && right!=null;}
}
```

L'idea di base consiste nel rappresentare un'espressione con un albero binario i cui nodi contengono operatori ("**+**" oppure "**-**" oppure "*****") e le cui foglie contengono valori interi.

La classe `Espressione` prevede sia costruttori diretti (privati) che un costruttore pubblico che legge, da un qualunque `Reader` l'espressione in forma infissa, polacca diretta o polacca inversa.

Ricordiamo le specifiche del metodo `read()` dell'interfaccia `Reader`

`public int read() throws IOException`

Read a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.

Returns: The character read, as an integer in the range **`0`** to **`65535`**, or **`-1`** if the end of the stream has been reached

Throws: **`IOException`** - If an I/O error occurs

Vediamo ora l'intestazione e i costruttori della classe `Espressione` .

```
public class Espressione extends BinNode {
  public static final int DIRETTA=-1, INVERSA=-2, INFISSA=-3;

// Costruttore: legge l'espressione da x nel formato "tipo"
 public Espressione(Reader x, int tipo) {
  switch(tipo){
      case(DIRETTA): leggiDiretta(x); break;
      case(INVERSA): leggiInversa(x); break;
      case(INFISSA): leggiInfissa(x); break;
      default: throw new IllegalArgumentException("Illegal Type");
   }
 }
```

```java
//Costruttore privato crea l'espressione "l op r"
  private Espressione(Operatore op, Espressione l, Espressione r){
     super(op,l,r);
  }

// Costruttore privato crea l'espressione  "x"
  private Espressione(Integer x){
     super(x);
  }

// converte un carattere in un Object, il parametro e' intero e deve essere positivo
  private Object charToElement(int i){
   if(i==-1)throw new IllegalArgumentException("Syntax error");
   switch((char) i){
        case('1'): return new Integer(1);
        case('2'): return new Integer(2);
        case('3'): return new Integer(3);
        case('4'): return new Integer(4);
        case('5'): return new Integer(5);
        case('6'): return new Integer(6);
        case('7'): return new Integer(7);
        case('8'): return new Integer(8);
        case('9'): return new Integer(9);
        case('0'): return new Integer(0);
     }
   return new Operatore((char) i);
   }
```

Le letture sono demandate a tre appositi metodi interni che ora consideriamo uno ad uno.

```
private void leggiInversa(Reader x) {
  Stack stack = new Stack();
   try {
     int i = x.read();
     while (i!=-1) {
       Object o = charToElement(i);
       if(o instanceof Integer) stack.push(new Espressione((Integer) o));
       else {
         Espressione r = (Espressione) stack.pop();
         Espressione l = (Espressione) stack.pop();
         stack.push(new Espressione((Operatore) o,l,r));
       }
       i = x.read();
     }
     Espressione esp = (Espressione) stack.pop();
     this.key   = esp.key;
     this.left  = esp.left;
     this.right = esp.right;
   } catch (IOException e) {
       throw new IllegalArgumentException("IOException");
   } catch (EmptyStackException e){
       throw new IllegalArgumentException("Syntax error");
   }
}
```

```java
private void leggiInfissa(Reader x){
   try {
      int i = x.read();
      if(i==-1)throw new IllegalArgumentException("Syntax error");
      char c = (char) i;
      if (c=='('){
         this.left = new Espressione(x, INFISSA);
         this.key  = charToElement(x.read());
         if(!(this.key instanceof Operatore))
            throw new IllegalArgumentException("Syntax error");
         this.right = new Espressione(x, INFISSA);
         if (x.read()!=')')
            throw new IllegalArgumentException("Syntax error");
         } else {
            Object o = charToElement(i);
            if(o instanceof Integer) this.key = o;
            else throw
              new IllegalArgumentException("Syntax error");
         }
   } catch(IOException e){
      throw new IllegalArgumentException("Syntax error");
   }
}
```

```java
private void leggiDiretta(Reader x)  {
  try {
      this.key=charToElement(x.read());
      if(this.key instanceof Operatore){
          this.left =  new Espressione(x,DIRETTA);
          this.right = new Espressione(x,DIRETTA);
      }
  } catch(IOException e){throw new IllegalArgumentException("Syntax error");}
}

// valutazione di un espressione
public int valuta() {
  if (isLeaf())  return ((Integer) this.key).intValue();
  if (twoSons()) return
       ((Operatore) this.key). valuta((Espressione) this.left,(Espressione) this.right);
  throw new IllegalArgumentException("Syntax error");
}

//stampa un espressione nel formato specificato da Job;
public void stampa(int tipo) {
  switch(tipo){
       case(DIRETTA): stampaDiretta(); break;
       case(INVERSA): stampaInversa(); break;
       case(INFISSA): stampaInfissa(); break;
       default: throw new IllegalArgumentException("Illegal Type");
  }
}
```

```java
// stampa una espressione in forma polacca diretta
private void stampaDiretta(){
   if (isLeaf())System.out.print(((Integer) this.key).intValue());
   else if (twoSons()) {
       System.out.print((Operatore) this.key);
       ((Espressione) left).stampaDiretta();
       ((Espressione) right).stampaDiretta();
   } else throw new IllegalArgumentException("Syntax error");
}


// stampa una espressione in forma polacca inversa
private void stampaInversa(){
   if (isLeaf())System.out.print(((Integer) this.key).intValue());
   else if (twoSons()) {
        ((Espressione) left).stampaInversa();
        ((Espressione) right).stampaInversa();
        System.out.print((Operatore)this.key);
   } else throw new IllegalArgumentException("Syntax error");
}
```

```java
// stampa una espressione in forma infissa
private void stampaInfissa(){
   if (isLeaf())System.out.print(((Integer) this.key).intValue());
   else if (twoSons()) {
        System.out.print("(");
        ((Espressione) left).stampaInfissa();
        System.out.print((Operatore)this.key);
        ((Espressione) right).stampaInfissa();
        System.out.print(")");
   } else throw new IllegalArgumentException("illegal expression");
}
```

La classe Operatore

```
//Implementa un operatore aritmetico
public class Operatore {
  public char op;

public Operatore(char op) {
    if (!(op=='+' || op=='*' || op=='-')) throw
      new IllegalArgumentException("illegal operator");
    this.op = op;
  }

  public int valuta(Espressione left, Espressione right) {
    switch(op){
     case('+'): return left.valuta() + right.valuta();
     case('-'): return left.valuta() - right.valuta();
     case('*'): return left.valuta() * right.valuta();
     default:   throw new IllegalArgumentException("Illegal operator");
    }
  }

  public String toString(){return op+"";}
}
```

Il programma di prova

```
import java.io.*;
public class Main{
 static void provaEspressione (Espressione e, int valoreCorretto) {
    try {
         System.out.print(" forma diretta: ");
         e.stampa(Espressione.DIRETTA);
         System.out.println();
         System.out.print(" forma inversa: ");
         e.stampa(Espressione.INVERSA);
         System.out.println();
         System.out.print(" forma infissa: ");
         e.stampa(Espressione.INFISSA);
         System.out.println();
         System.out.print(" valore:        " );
         int res1 = e.valuta();
         System.out.print(res1);
         if (res1==valoreCorretto) System.out.println("   OK");
         else System.out.println("   NO");
     } catch (IllegalArgumentException ex) {
         System.out.println(" ERRORE in provaEspressione: "+ex);
     }
     System.out.println(); System.out.println();
  }
```

```java
  public static void main(String[] args) {
      Espressione e;
      try {
         System.out.println(" lettura Polacca Diretta:  *-+35*479");
         e = new Espressione(new StringReader("*-+35*479"),
            Espressione.DIRETTA);
         System.out.println(" lettura corretta");
         provaEspressione(e, -180);
         System.out.println(" lettura Polacca Inversa: 35+47*-9*");
         e = new Espressione(new StringReader("35+47*-9*"),
            Espressione.INVERSA);
         System.out.println(" lettura corretta");
      provaEspressione(e, -180);
         System.out.println(" lettura infissa: (((3+5)-(4*7))*9)");
         e = new Espressione(new StringReader("(((3+5)-(4*7))*9)"),
            Espressione.INFISSA);
         System.out.println(" lettura corretta");
         provaEspressione(e, -180);
      } catch (RuntimeException ex) {
            System.out.println(" ERRORE  "+ex);
      }
 }
}
```

Risultati

```
lettura Polacca Diretta:  *-+35*479
lettura corretta
forma diretta:  *  -  + 35 * 479
forma inversa: 35 + 47 *  - 9 *
forma infissa: (((3 + 5) - (4 * 7)) * 9)
valore:        -180   OK


lettura Polacca Inversa: 35+47*-9*
lettura corretta
forma diretta:  *  -  + 35 * 479
forma inversa: 35 + 47 *  - 9 *
forma infissa: (((3 + 5) - (4 * 7)) * 9)
valore:        -180   OK


lettura infissa: (((3+5)-(4*7))*9)
lettura corretta
forma diretta:  *  -  + 35 * 479
forma inversa: 35 + 47 *  - 9 *
forma infissa: (((3 + 5) - (4 * 7)) * 9)
valore:        -180   OK
```

Aggiunta di una funzionalità

Compilazione di Espressioni Aritmetiche

La funzionalità da aggiungere è la traduzione di una espressione aritmetica, del tipo

((2+3)*(5+7))

in un programma a basso livello che la calcoli:

```
STO V1 2
STO V2 3
SUM V1 V2 V3
STO V1 5
STO V2 7
SUM V1 V2 V4
MUL V3 V4 V1
PRI V1
```

Le istruzioni disponibili sono

```
STO <var> <const>
```

che memorizza la costante <const> nella variabile <var>.

```
PRI <var>
```

che stampa la variabile <var>.

```
SUM <var1> <var2> <var3>
```

che memorizza la somma <var1> + <var2> nella variabile <var3>.

```
DIF <var1> <var2> <var3>
```

che memorizza la differenza var1> - <var2> nella variabile <var3>.

```
MUL <var1> <var2> <var3>
```

che memorizza il prodotto <var1> - <var2> nella variabile <var3>.

NB le variabili vanno usate in modo ottimo ovvero si devono riutilizzare i nomi di variabile già definite e non più utilizzabili.

Aggiunta di una Classe: Var

```
import java.util.BitSet;

public class Var {
  BitSet v = new BitSet();

  public int get() {
      int i=v.nextClearBit(0);
      v.set(i);
      return i;
  }

  public void free(int v1) {
      v.clear(v1);
  }

}
```

Modifiche alle classi esistenti : in Espressione

```java
static PrintStream w;
static Var var;

public void compila(PrintStream w) {
    var = new Var();
    this.w = w;
    int r = this.compila();
    w.println("PRI V" + r);
}

private int compila() {
    if (isLeaf()) {
        int v3 = var.get();
        w.println("STO V" + v3+" "+(Integer) this.key);
        return v3;
    } else {
        int v1 = ((Espressione) left).compila();
        int v2 = ((Espressione) right).compila();
        w.println(((Operatore) key).code() + " V" + v1 + " V" + v2 + " V" + v1);
        var.free(v2);
        return v1;
    }
}
```

Modifiche alle classi esistenti : in Operatore

```
public String code() {
  switch(op){
   case('+'): return "SUM";
   case('-'): return "DIF";
   case('*'): return "MUL";
   default: throw new RuntimeException("Illegal operator");
  }
}
```

Modifiche alle classi esistenti : nel Main

```
public static void main(String[] args) {
    Espressione e;
    try {
              . . .
        System.out.println();
        System.out.println(" CODICE COMPILATO ");

        e.compila(System.out);

    } catch (Exception ex) {
        System.out.println(" ERRORE " + ex);
    }
}
```

Risultati

```
lettura Polacca Diretta:  *-+35*479
 lettura corretta
 forma diretta:  *  -  + 35 * 479
 forma inversa: 35 + 47 *  - 9 *
 forma infissa: (((3 + 5) - (4 * 7)) * 9)
 valore:        -180   OK

             . . .

CODICE COMPILATO
STO V0 3
STO V1 5
SUM V0 V1 V0
STO V1 4
STO V2 7
MUL V1 V2 V1
DIF V0 V1 V0
STO V1 9
MUL V0 V1 V0
PRI V0
```