Corso: Fondamenti, Linguaggi e Traduttori

Paola Giannini

Osservazioni Esercizio 6 Dicembre (implementazione Scanner)

Commenti generali

- Lo `Scanner`
  - Alcuni hanno il metodo `nextToken()` che non riflette l'automa che per prima cosa scandisce tutti i caratteri di skip e poi fa le varie scan (che non devono consumare caratteri di skip).
  - Questo vale anche per molte `scanNumber()`, che alle volte riconoscono come float stringhe che hanno più di un punto, oppure non riconoscono come intero `0`. La complicazione, anche qui, arriva quando non si segue la logica dell'automa.
  - L'ipotesi che molti fanno nelle `scanNumber()` e `scanId()` è che il numero o identificatore debba essere seguito da un carattere di skip o un operatore (alcuni non considerano l'operatore!).
  - In generale un metodo NON deve essere troppo lungo, nè contenere troppi annidamenti di **`if-else`**.
  - Se usate cicli NON ci dovrebbero essere uscite dal ciclo o **`break`**!
  - In generale `scanNumber()` TROPPO COMPLICATE e la complicazione del codice è venuta dal non seguire la logica dell'automa riconoscitore.
- Le eccezioni e i relativi test dimostrano che anche qui c'e' un po' di confusione.

Indice

1. Il metodo `nextToken`
2. Il metodo `scanNumber`
3. Il metodo `scanID`
4. I test

Non ritornate `null`

```
1   public Token nextToken() throws IOException, LexicalException {
2     char nextChar = peekChar();
3
4     // skip characters
5     while(this.skpChars.contains(nextChar)) {
6           ......................
7     }
8
9     if(this.digits.contains(nextChar)) return this.scanNumber(readChar());
10
11    if( this.letters.contains(nextChar) ) return this.scanId(readChar());
12
13    if(this.operatorsMap.containsKey(nextChar))
14      return new Token((TokenType) this.operatorsMap.get(readChar()), riga);
15
16    String err = String.format("Lexical error in row %s\n", riga);
17    this.log.append(err);
18    readChar();
19    return null;
20  }
```

- Va bene non sollevare un'eccezione e mettere il messaggio nella stringa di log, ma perchè allora non andare avanti
- alla riga 16

  `return nextToken();`

Test inutile ed eccezione NON appropriata!

```
1   public Token nextToken() throws Exception {
2     char nextChar = peekChar(); // contiene il prossimo carattere dell'input
3     while (this.skpChars.contains(nextChar)) {
4       readChar(); // consumo il carattere
5       if (nextChar == '\n') this.riga++;
6       if (nextChar == EOF) return new Token(TokenType.EOF, this.riga);
7       nextChar = peekChar();
8     }
9     if (nextChar == '.') {  // punto all'inizio di un token (es .123) errore
10      while (!this.skpChars.contains(nextChar)) {
11        readChar(); // consumo il carattere
12      }
13      throw new Exception();
14    }
15    if (this.digits.contains(nextChar)) return scanNumber();
16    if (this.letters.contains(nextChar)) return scanId();
17    if (this.operatorsMap.containsKey(nextChar)) {
18      readChar(); // consumo il carattere
19      return new Token(this.operatorsMap.get(nextChar), this.riga);
20    }
21    throw new Exception();
22  }
```

- Lanciare un `Exception` senza messaggio non fornisce informazione su dove è il problema!
- Codice alle righe 9-14 è INUTILE perchè verrebbe in ogni modo sollevata una eccezione. (Non faceva parte dell'automa!)
- Se, quando avviene un errore, si consumano i caratteri fino ad arrivare ad un carattere di skip questo si dovrebbe fare anche prima di lanciare l'eccezione alla riga 21.

Codice più complicato del necessario ed eccezione non appropriata!

```
1   while(nextChar!=EOF) {
2     if(skipchars.contains(nextChar)) { //Stato 0
3       if(nextChar=='\n') this.riga++;
4     }
5     else {
6       if(digits.contains(nextChar)) {  /*Stato 2:*/ rettoken=scanNumber(nextChar);  return retto
7       }
8       else {
9         if(operatorsMap.containsKey(nextChar)){ /*Stato 4:*/ rettoken=new Token(operatorsMap.get
10         }
11        else {
12          if(letters.contains(nextChar)){ /*Stato 3:*/   rettoken=scanId(nextChar);return rettok
13        }
14        else {
15          consumaErr(nextChar);
16          throw new IllegalArgumentException("Token non valido");  /*Stato 8*/
17        } }  } }
18  nextChar=readChar();
19  }
20  rettoken=new Token(TokenType.EOF, riga); //Stato 1
21  return rettoken;
```

- NON segue la logica dell'automa che consuma i caratteri di skip (ciclo sullo stato iniziale) prima di controllare gli altri e l'annidamento (che NON è dell'automa!) del codice rende il metodo molto meno capibile!
- L'eccezione `IllegalArgumentException` è "unchecked" per cui non deve essere dichiarata e questo NON bene perchè il parser che chiama la nexToken NON sa che c'è stato un errore nello scanner!

Altra eccezione NON appropriata, annidamenti e variabili inutili

```
1   public Token nextToken() throws IOException {
2     // nextChar contiene il prossimo carattere dell'input.
3     char nextChar = peekChar();
4     char skip;
5     // Avanza nel buffer leggendo i carattere in skipChars
6     while (this.skipChars.contains(nextChar)) {
7       skip = this.readChar();
8        if (skip == '\n') { nextChar = peekChar();  riga++; }
9        else if (skip == EOF) { Token eof = new Token(TokenType.EOF, riga);  return eof; }
10         else nextChar = peekChar();
11    }
12          ................
13    // Altrimenti il carattere NON E' UN CARATTERE LEGALE
14    throw new IOException("Errore: Token non riconoscibile");
15  }
```

- Anche l'eccezione `IOException` NON ha niente a che fare con il tipo di errore da segnalare.
- L' annidamento alle righe 7-11e la variabile `skip` non sono necessari. Le righe 4-11 posso essere riscritte in modo più capibile

```
//INV: nextChar contiene il prossimo carattere (non ancora consumato!)
while(skpChars.contains(nextChar)) {
  nextChar = readChar();//consume skip character
  if(nextChar==EOF) // case EOF
    return new Token(TokenType.EOF,this.riga);
  if(nextChar=='\n' ) // case EOL
    this.riga++;
  nextChar = peekChar();//peek new character
}
```

Indice

1. Il metodo `nextToken`
2. Il metodo `scanNumber`
3. Il metodo `scanID`
4. I test

Metodo TROPPO LUNGO, codice ripetuto, controlli non necessari!!!! (1)

```
1   private Token scanNumber() throws IOException {
2     String value = "";
3     if(peekChar() == '0') { //ho uno 0
4       value = value+readChar(); //consuma 0
5       if(peekChar() == '.') { //se trova '.'
6         value = value+readChar(); //consuma '.'
7         int cont = 0;
8         while(!skipChars.contains(peekChar()) && !operatorsMap.containsKey(peekChar())) {
9           if(cont>=5) throw new IOException("....");
10          else if(!digits.contains(peekChar()) && !operatorsMap.containsKey(peekChar())) {
11            throw new IOException(".....");
12          }
13          value = value+readChar();  cont++; // consuma e incrementa
14        }
15        if((skipChars.contains(peekChar()) && cont==0) ||
16              (operatorsMap.containsKey(peekChar()) && cont==0))throw new IOException("....");
17        else {Token tokenFLOAT0=new Token(TokenType.FLOAT, riga, value);return tokenFLOAT0; }
18      }
19      else if(skipChars.contains(peekChar()) || operatorsMap.containsKey(peekChar())){//solo 0
20        Token token0 = new Token(TokenType.INT, riga, value);
21        return token0;
22      }
23      else { //se trova qualsiasi altra cosa dopo lo 0
24        throw new IOException("....");
25      }  }
26    else //ho un numero != 0
27     ........
28    return null; //se riconosce un numero anche senza leggerlo
29  }
```

Metodo TROPPO LUNGO, codice ripetuto, controlli non necessari!!!! (2)

```
1       .........
2     else if(digits.contains(peekChar()) && peekChar() != '0') { //ho un numero != 0
3       int cont=0;
4       while(!skipChars.contains(peekChar())&&!operatorsMap.containsKey(peekChar())&&peekChar() !
5         if(!digits.contains(peekChar())) throw new IOException(".....");
6         value = value+readChar();
7       }
8       if(peekChar() == '.') {value = value+readChar(); //consuma '.'
9         while(!skipChars.contains(peekChar()) && !operatorsMap.containsKey(peekChar())) {
10          if(cont>=5)  throw new IOException(".....");
11          else if(!digits.contains(peekChar())) {
12            throw new IOException("Numero non valido in riga: "+riga+". char illegale dopo x.");
13          }
14          value = value+readChar(); cont++;
15        }
16        if((skipChars.contains(peekChar()) && cont==0) ||
17            (operatorsMap.containsKey(peekChar()) && cont==0)) { //se ho solo x.
18          throw new IOException(".....");
19        }
20        else if(operatorsMap.containsKey(peekChar())) { //se ho un float con un op che segue
21          Token tokenFLOAT = new Token(TokenType.FLOAT, riga, value);
22          return tokenFLOAT;
23        } }
24      if(cont == 0) { //se non ho decimali
25        Token tokenINT = new Token(TokenType.INT, riga, value);
26        return tokenINT;
27      }
28      else {
29        Token tokenFLOAT = new Token(TokenType.FLOAT, riga, value);
30        return tokenFLOAT;
31      }  }
32    return null; //se riconosce un numero anche senza leggerlo
33  }
```

Problemi del codice precedente

- Dopo aver letto il punto (riga 5 di (1) e riga 8 di (2)) siamo nello stesso stato, per cui si deve fare esattamente la stessa cosa, l'unica cosa diversa è la stringa che è stata letta. Per cui definire:

```
private Token floatDopoPunto(String primaPunto) {.......}
```

- Di nuovo le eccezioni di IO non sono appropriate, anche se hanno messaggi che vanno bene.
- Fate il controllo `cont>0` e `cont<=5` insieme dopo aver letto tutti i numeri (come nell'automa!)
- Se il primo carattere è 0 l'unica cosa che dovete controllare se il carattere successivo non è punto è se il carattere successivo è una cifra o una lettera e sollevare un'eccezione,
- Se il primo carattere è una cifra l'unica cosa che dovete controllare è che il numero non abbia alla fine una lettera e sollevare un'eccezione,

COMPLICATO: riconosce anche FLOAT con 2 punti, `while` con troppi punti di uscita

```
private Token scanNumber() throws Exception {
    char nextChar = peekChar(); // contiene il prossimo carattere dell'input
    String parola = "";
    boolean uscito = false;
    boolean primoGiro = true;

    while (true) {
        if (this.operatorsMap.containsKey(nextChar) || this.letters.contains(nextChar)) {
            uscito = true;
            break;
        }

        if (this.skpChars.contains(nextChar))
            break;

        if (this.digits.contains(nextChar) || nextChar == '.') {
            parola = parola + nextChar;
            readChar(); // consumo carattere
        }

        nextChar = peekChar();
    }
    int indicePunto = parola.indexOf('.');
    int indiceDopoPunto = indicePunto + 1;
    int indicePiuSei = indicePunto + 6; // per controllare numeri decimali

    if (uscito == false) {
        if (parola.charAt(0) == '0') {
            if (parola.length() == 1) {
                return new Token(TokenType.INT, this.riga, parola);
            } else {
                if (parola.charAt(1) == '.') {
                    if (parola.length() <= indicePiuSei) {
                        return new Token(TokenType.FLOAT, this.riga, parola);
                    }
                }
            }
        } else {
            if (indicePunto < 0) { // se il punto NON è presente
                return new Token(TokenType.INT, this.riga, parola);
            } else {
                if (this.digits.contains(parola.charAt(indiceDopoPunto))) { // se il carattere dopo il punto è un
                                                                            // num
                    if (parola.length() <= indicePiuSei) {
                        return new Token(TokenType.FLOAT, this.riga, parola);
                    }
                }
            }
        }
    } else {
        while (!this.skpChars.contains(nextChar)) {
            readChar(); // consumo il carattere
            nextChar = peekChar();
        }
        throw new Exception();
    }
    throw new Exception();
```

COMPLICATO: non riconosce `0` come `TokenType.INT`

```
private Token scanNumber(char nextChar) throws IOException
{
    Token rettoken=new Token(TokenType.ASSIGN, 1);
    boolean puntotrovato=false;
    int primociclo=1;
    int count=0;
    String value="";
    while(digits.contains(nextChar) || nextChar=='.') {
        if(puntotrovato==true && nextChar=='.') {
            consumaErr(nextChar);
            throw new IllegalArgumentException("Token non valido");
        }
        if(nextChar=='0' && primociclo==1)  { //Stato 7
            if(!(peekChar()=='.' || skipchars.contains(peekChar()))) {
                consumaErr(nextChar);
                throw new IllegalArgumentException("Token non valido");
            }
        }
        if(nextChar=='.') { //Stato 5
            if(puntotrovato==false)
            {
                puntotrovato=true;
                if(!digits.contains(peekChar()))
                {
                    consumaErr(nextChar);
                    throw new IllegalArgumentException("Token non valido");
                }
            }
            else //stato 8. ES: 2..
            {
                consumaErr(nextChar);
                throw new IllegalArgumentException("Token non valido");
            }
        }
        if (puntotrovato==true) count++;
        if(count>5) {
            consumaErr(nextChar);
            throw new IllegalArgumentException("Token non valido");
        }
        value=value+nextChar;
        nextChar=readChar();
        primociclo=0;
    }
    if(!(skipchars.contains(nextChar) || nextChar==';')) {
        consumaErr(nextChar);
        throw new IllegalArgumentException("Token non valido");
    }
    else {
        if(nextChar==';')
        {
            buffer.unread(nextChar);
        }
    }
   if (puntotrovato==true) {
        rettoken=new Token(TokenType.FLOAT, riga, value);
    }
    else
    {
        rettoken=new Token(TokenType.INT, riga, value);
    }

    return rettoken;
}
```

Non riconosce 0

```
1   private Token scanNumber() throws IOException, LexicalException {
2     StringBuilder val = new StringBuilder(String.valueOf(peekChar()));
3     boolean iniziaConZero = Character.valueOf(readChar()).equals('0');
4     if (iniziaConZero && !Character.valueOf(peekChar()).equals('.'))
5       lanciaErrore("Un numero che inizia con zero DEVE esser seguito da un punto", val);
6     boolean presenzaUnPunto = false;
7     int numeriDopoPunto = 0;
8     char nextChar;
9     while (!(skpChars.contains(nextChar = peekChar()) || operatorsMap.containsKey(nextChar))) {
10      if (letters.contains(nextChar))
11        lanciaErrore("Il numero contiene delle lettere", val);
12      if (!Character.valueOf(nextChar).equals('.') && !this.digits.contains(nextChar))
13        lanciaErrore("Un numero non puo' contenere il seguente carattere: " + nextChar, val);
14      if (presenzaUnPunto && Character.valueOf(nextChar).equals('.'))
15        lanciaErrore("Il numero contiene piu' punti", val);
16      if (Character.valueOf(nextChar).equals('.')) presenzaUnPunto = true;
17      if (presenzaUnPunto && !Character.valueOf(nextChar).equals('.'))
18        numeriDopoPunto++;
19      if (numeriDopoPunto > 5)
20        lanciaErrore("Il numero contiene piu' di 5 cifre dopo il punto", val);
21      val.append(readChar());
22    }
23    if (presenzaUnPunto && numeriDopoPunto == 0)
24      lanciaErrore("Il numero non contiene cifre dopo il punto", val);
25    return new Token(presenzaUnPunto ? TokenType.FLOAT : TokenType.INT, this.riga, val.toString(
26  }
```

- La logica non è chiara, ma si capisce perchè non riconosce 0 come intero!

Non usare pattern matching e eccezioni "unchecked"

```
1   private Token scanNumber() throws Exception {
2     StringBuilder val = new StringBuilder();
3     StringBuilder temp = new StringBuilder();
4     boolean dot = false;
5     TokenType type = TokenType.INT;
6     while (true) {
7       char currentChar = peekChar();
8       if ((skpChars.contains(currentChar) || operatorsMap.containsKey(currentChar)) &&
9       (temp.toString().matches("0|[1-9][1-9]*") || temp.toString().matches("(0|[1-9][1-9]*).[0-9
10      break;
11      temp.append(currentChar);
12      if (temp.toString().matches("0|[1-9][1-9]*")) {
13        val.append(readChar());
14      } else
15        if ((currentChar == '.' && !dot)  || temp.toString().matches("(0|[1-9][1-9]*).[0-9]{1,5}
16            type = TokenType.FLOAT; val.append(readChar()); dot = true;
17        } else {
18          while(!skpChars.contains(currentChar)) val.append(readChar());currentChar = peekChar()
19          throw new NumberFormatException("Il formato del numero \""+val+"\" alla riga "+riga+ "
20    } }
21    return new Token(type, riga, val.toString());
22  }
```

- Uso delle espressioni regolari e patten matching che è quello che la scanNumber DEVE fare. Ad esempio per `temp%` non viene emesso un token ID e poi dato un errore alla `nextToken` successiva!
- L'eccezione sollevata `NumberFormatException` è "unchecked" per cui può essere ignorata
- La clausola **`throws`** `Exception` è generale e non dice quello che viene sollevato!

Indice

1. Il metodo `nextToken`
2. Il metodo `scanNumber`
3. Il metodo `scanID`
4. I test

Deve ritornare anche la stringa letta e .....

```
1   private Token scanId() throws Exception {
2     char nextChar = peekChar(); // contiene il prossimo carattere dell'input
3     boolean uscito = false;
4     String parola = "";
5     while (true) {
6       if (this.operatorsMap.containsKey(nextChar) || this.digits.contains(nextChar)
7           || nextChar == '.') {
8         uscito = true; break;
9       }
10      if (this.skpChars.contains(nextChar)) break;
11      if (this.letters.contains(nextChar)) {
12        parola = parola + nextChar; readChar(); // consumo carattere
13      }
14      nextChar = peekChar();
15    }
16    if (uscito == false) {
17      if (keyWordsMap.containsKey(parola)) {
18        return new Token(keyWordsMap.get(parola), this.riga);
19      } else {
20        return new Token(TokenType.ID, this.riga); // ritorna un id se sono solo lettere o numer
21    } } else {
22      while (!this.skpChars.contains(nextChar)) {
23        readChar(); nextChar = peekChar();
24      }
25      throw new Exception();
26   }}
```

- Perchè usare `while (true)` e non scorrere le lettere (come fa l'automa!)
- Se un ID è seguito da un operatore da' errore e questo è scorretto perchè posso scrivere x+3

Un identificatore non deve essere seguito solo da cose che ci sono

```
1   private Token scanId() throws IOException, LexicalException {
2     StringBuilder val = new StringBuilder(String.valueOf(readChar()));
3     char nextChar;
4     while (!(skpChars.contains(nextChar = peekChar()) || operatorsMap.containsKey(nextChar))) {
5       if (digits.contains(nextChar))
6         lanciaErrore("Un identificatore o una keyword non ......");
7       if (!letters.contains(nextChar))
8         lanciaErrore("Un identificatore o una keyword non .......");
9       val.append(readChar());
10    }
11    return
12      new Token(keyWordsMap.getOrDefault(val.toString(), TokenType.ID), this.riga,
13                                  keyWordsMap.containsKey(val.toString()) ? null : val.toString(
14  }
```

- Perchè scorrere finchè non si trova un operatore o un carattere di skip? Se aggiungiamo le parentesi al lessico, non si riconoscono più correttamente gli identificatori!

Indice

1. Il metodo `nextToken`
2. Il metodo `scanNumber`
3. Il metodo `scanID`
4. I test

Non si testano così le eccezioni

```
1   @Test
2   public void testScanGenerale() throws IOException {
3     String path ="src/test/data/testScanner/testGenerale.txt";
4     Scanner scanner = new Scanner(path);
5     Token t;
6     try {
7       t = scanner.nextToken(); //test dei token prima di quello che provoca l'eccezione
8         ..........
9       t = scanner.nextToken();
10      assertEquals(2, t.getRiga());
11      assertEquals(TokenType.ASSIGN,t.getTipo());//Il token successivo provocherebbe una eccezi
12    } catch (IOException e) {
13      Assertions.assertTrue(true);
14      try {
15        t = scanner.nextToken(); // questo codice NON viene mai eseguito
16        assertEquals(2, t.getRiga());
17        assertEquals(TokenType.SEMI, t.getTipo());
18        t = scanner.nextToken();
19          ............
20       } catch (IOException e1) {e1.printStackTrace();}
21     } catch (Exception e){ e.printStackTrace();}
22  }
```

- A cosa serve questo test?

Non catturate eccezioni in un test

```
1   @Test
2   public void testINT() {
3     try {
4       Scanner S = new Scanner("src/test/data/testScanner/testINT.txt");
5       try {
6         Token t = S.nextToken();
7         assertEquals("698", t.getVal());
8         assertEquals(2,t.getRiga());
9         assertEquals(TokenType.INT,t.getTipo());
10      } catch (IOException e) {e.printStackTrace();}
11        } catch (FileNotFoundException e) {e.printStackTrace();}
12  }
```

Ci sono diverse opzioni:

- Non usate il **`try-catch`** nei test. Usate la clausola **`throws`**.
- Fate la creazione degli scanner in `@BeforeEach` invece che nel test.