

UNIVERSITA’ DEGLI STUDI DI TORINO

Dipartimento di Informatica

Corso di Linguaggi Formali e Traduttori

**Elaborato finale del Laboratorio di Linguaggi Formali e Traduttori**

Autore: Annalisa Sabatelli

Matr. 866879

1. **Implementazione di un DFA in Java**

Introduzione

Con riferimento ai DFA oggetto degli esercizi proposti, verrà utilizzata la seguente notazione nel corso della parte prima di questo elaborato.

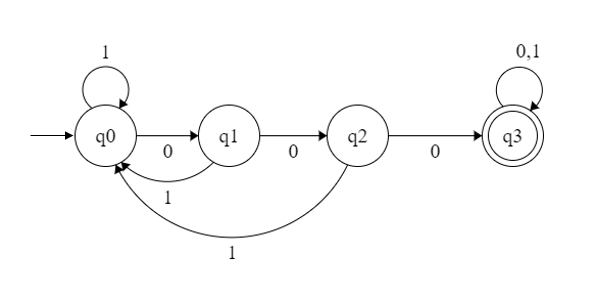
Il generico DFA *D* è così caratterizzato:

dove

* Q è l’insieme di tutti gli stati;
* ∑ è l’insieme di tutti i possibili simboli di input;
* δ è la funzione di transizione;
* F è l’insieme degli stati finali o di accettazione.

Esercizio 1.1

Si consideri il seguente DFA che riconosce stringhe con 3 zeri consecutivi:

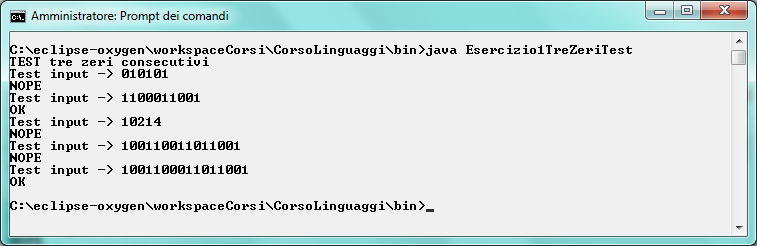


L’automa è implementato dal codice “*Esercizio1TreZeri.java*” fornito in allegato (folder PARTE\_1).

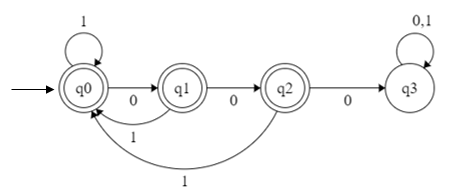
Sono state testate le seguenti stringhe

010101, 1100011001, 10214, 100110011011001, 1001100011011001

Per comodità è stata implementata una classe di test (“*Esercizio1TreZeriTest.java*”) per eseguire in sequenza i test delle stringhe, di seguito sono riportati i risultati:



Si opera, ora, una modifica al DFA precedente con lo scopo di ottenerne uno che riconosce il linguaggio complementare, ossia quello composto da stringhe che non contengono 3 zeri consecutivi.

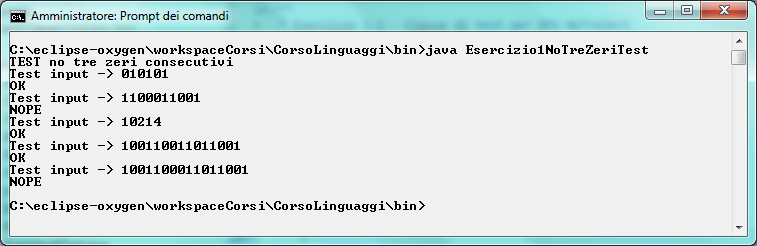


L’automa è implementato dal codice “*Esercizio1NoTreZeri.java*” fornito in allegato (folder PARTE\_1).

Sono state testate le seguenti stringhe

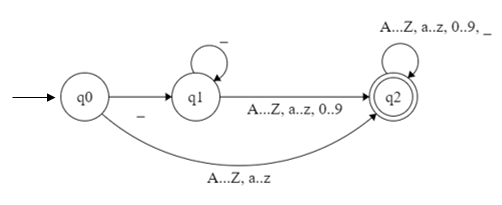
010101, 1100011001, 10214,100110011011001, 1001100011011001

Sono riportati di seguito i risultati dei test (classe di comodo “*Esercizio1NoTreZeriTest.java*”):



Esercizio 1.2

Si definisce un DFA per riconoscere il linguaggio degli identificatori di Java, ossia una sequenza non vuota di lettere, numeri e il simbolo “\_” che non comincia con un numero e non può essere composto solo da “\_”.



L’automa è implementato dal codice “*Esercizio2JavaIdentificatori.java*” fornito in allegato (folder PARTE\_1).

Sono state testate le seguenti stringhe:

\_\_

0oisudf\_

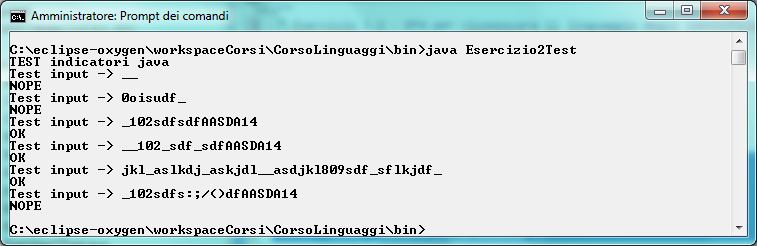
\_102sdfsdfAASDA14

\_\_102\_sdf\_sdfAASDA14

jkl\_aslkdj\_askjdl\_\_asdjkl809sdf\_sflkjdf\_

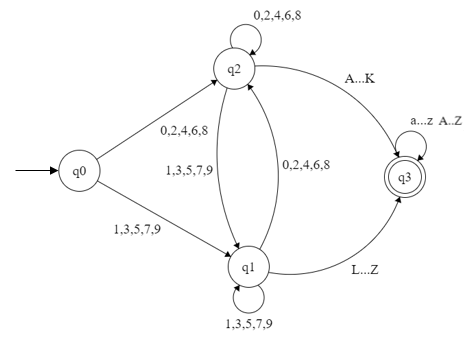
\_102sdfs:;/()dfAASDA14

Sono riportati di seguito i risultati dei test (classe di test “*Esercizio2Test.java*”):



Esercizio 1.3

Si definisce un DFA che riconosce il linguaggio di stringhe che contengono un numero di matricola seguito immediatamente da un cognome, dove la combinazione di matricola-cognome corrisponde a studenti del turno 2 o 3 del laboratorio di Linguaggi Formali e Traduttori.



L’automa è implementato dal codice “*Esercizio3MatricolaCognome.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe:

0123

Bianchi

987Rossi

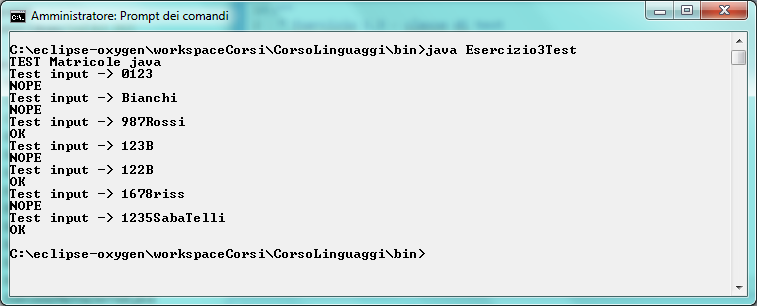
123B

122B

1678riss

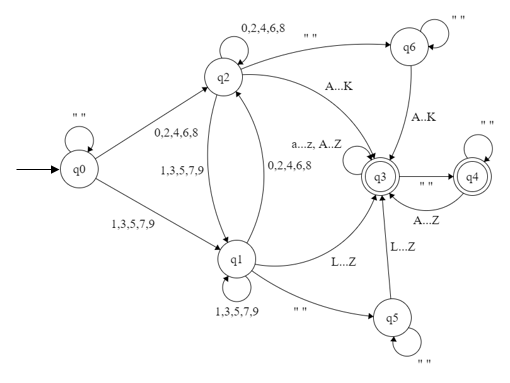
1235SabaTelli

Sono riportati di seguito i risultati dei test (classe di test “*Esercizio3Test.java*”):



Esercizio 1.4

Si definisce un DFA che, partendo da quello elaborato nell’esercizio 1.3, riconosca anche la combinazione di matricola e cognome separati o preceduti o seguiti da una sequenza di spazi.



L’automa è implementato dal codice “*Esercizio4MatricolaCognome.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe (le virgolette sono riportate per delimitare le stringhe, in particolare la seconda stringa inizia con uno spazio):

"654321 Rossi"

" 123456 Bianchi "

"123456Bia nchi"

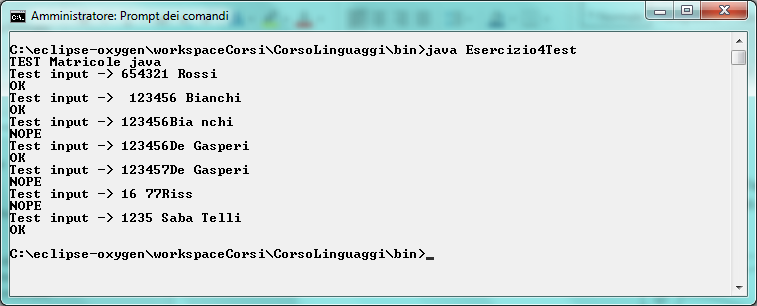
"123456De Gasperi"

"123457De Gasperi"

"16 77Riss"

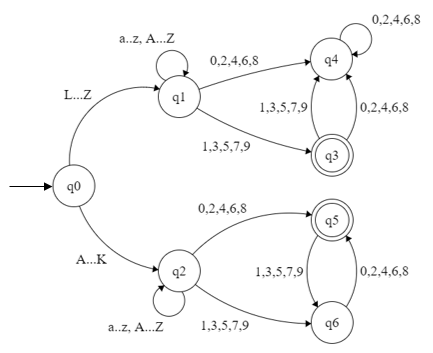
"1235 Saba Telli"

Sono riportati di seguito i risultati dei test (classe di test “*Esercizio4Test.java*”):



Esercizio 1.5

Si definisce un DFA che, partendo da quello elaborato nell’esercizio 1.3, riconosca il linguaggio di stringhe di matricole e cognomi degli studenti del turno 2 e 3 di laboratorio in cui il cognome precede il numero di matricola.



L’automa è implementato dal codice “*Esercizio5MatricolaCognome.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe:

Rossi654321

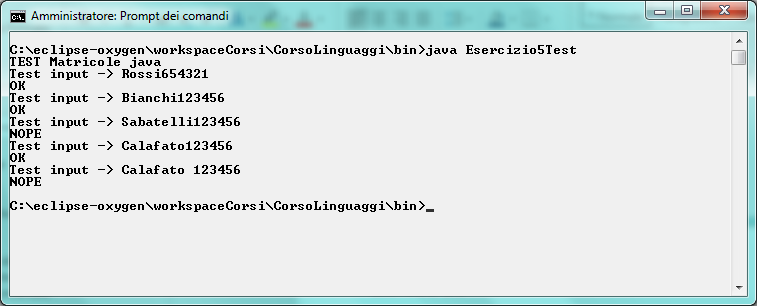
Bianchi123456

Sabatelli123456

Calafato123456

Calafato 123456

Sono riportati di seguito i risultati dei test (classe di test “*Esercizio5Test.java*”):

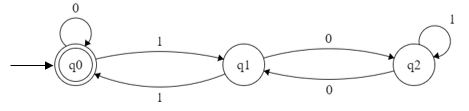


Esercizio 1.6

Si definisce un DFA che riconosce il linguaggio dei numeri binari il cui valore è multiplo di 3.

La soluzione utilizza le proprietà dei numeri naturali e la loro rappresentazione binaria. Se è la stringa di numeri binari letta ﬁno all’istante generico t, la sua rappresentazione in numero decimale sarà . Se all’istante t+1 si aggiunge alla stringa l’input y allora si dimostra facilmente che il corrispondente numero decimale è . Se un numero naturale x è divisibile per 3 allora vuol dire che esiste un altro numero naturale k tale che x è esprimibile come 3k. Si ipotizzi che ad un certo istante l’automa abbia letto la stringa che rappresenta il numero x divisibile per 3 e legga il simbolo y, possono darsi due casi:

Simmetricamente se si parte con un numero non divisibile per tre (quindi con resto di partenza uguale a 1 o a 2), si arriverà ad ottenere un numero divisibile o non divisibile a seconda del valore di y. In pratica, essendo 0 divisibile per 3, sono sufficienti tre stati per rappresentare tutte le sue classi di resto e le transizioni tra queste a seconda degli input ricevuti dal DFA.



L’automa è implementato dal codice “*Esercizio6BinariTre.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe:

110

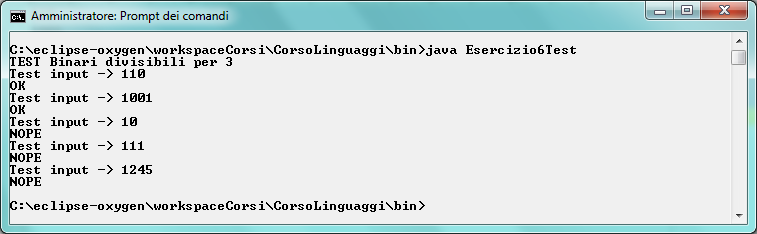
1001

10

111

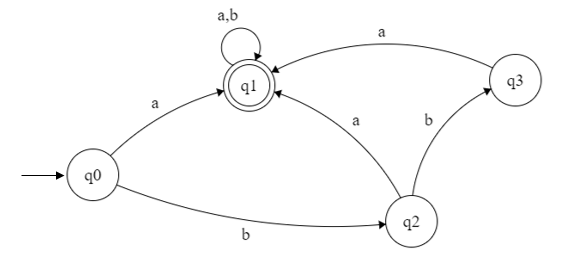
1245

Sono riportati di seguito i risultati dei test (classe di test “*Esercizio6Test.java*”):



Esercizio 1.7

Si definisce un DFA che riconosce il linguaggio di stringhe sull’alfabeto {a, b} tali che “a” occorre almeno una volta nelle prime tre posizioni della stringa anche se la stringa è composta da meno di tre simboli.



L’automa è implementato dal codice “*Esercizio7Linguaggioab.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe:

ababbbb

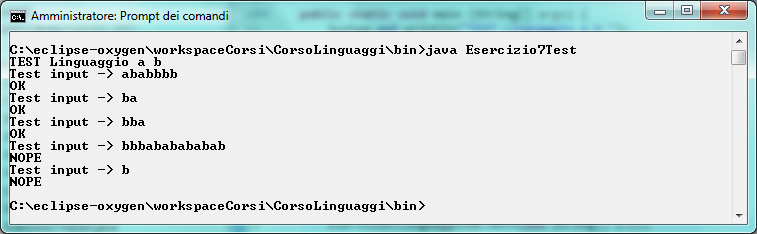
ba

bba

bbbababababab

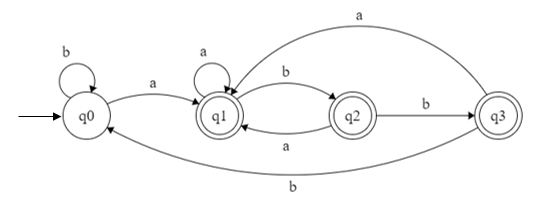
b

Sono riportati di seguito i risultati dei test (classe di test “*Esercizio7Test.java*”):



ESERCIZIO 1.8

Si definisce un DFA che riconosce il linguaggio di stringhe sull’alfabeto {a, b} tali che “a” occorre almeno una volta nelle ultime tre posizioni della stringa anche se la stringa è composta da meno di tre simboli.



L’automa è implementato dal codice “*Esercizio8Linguaggioab.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe:

abb

baaaaaaa

abbbbbb

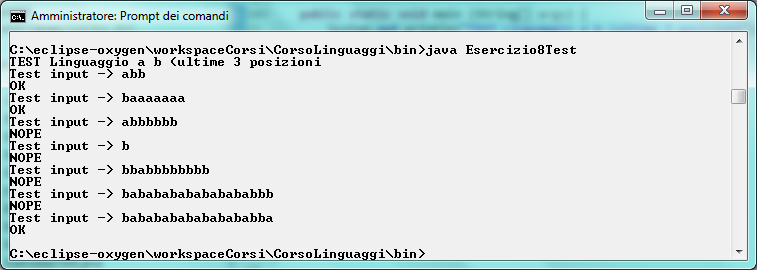
b

bbabbbbbbbb

bababababababababbb

bababababababababba

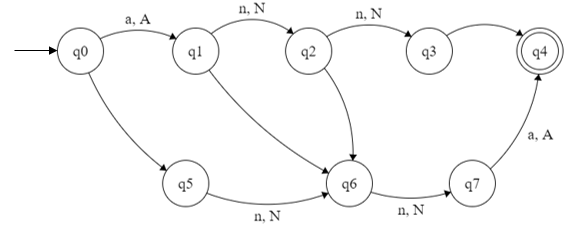
Sono riportati di seguito i risultati dei test (classe di test “*Esercizio8Test.java*”):



NOTA: Una possibile soluzione alternativa è quella di utilizzare il medesimo DFA dell’esercizio 1.7 ma invertire l’ordine di lettura della stringa. In questo modo, i due esercizi avrebbero la medesima rappresentazione grafica del DFA ma due codici implementativi differenti.

ESERCIZIO 1.9

Si definisce un DFA che riconosce il linguaggio delle stringhe che contengono il proprio nome e tutte le stringhe ottenute dopo la sostituzione di un carattere del nome con un altro qualsiasi.



NOTA BENE : Dove non specificato si intende che la transizione tra i due stati connessi avviene per qualsiasi carattere (alfa-numerico e speciale).

L’automa è implementato dal codice “*Esercizio9NomiVari.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe:

ANNA

%NNA

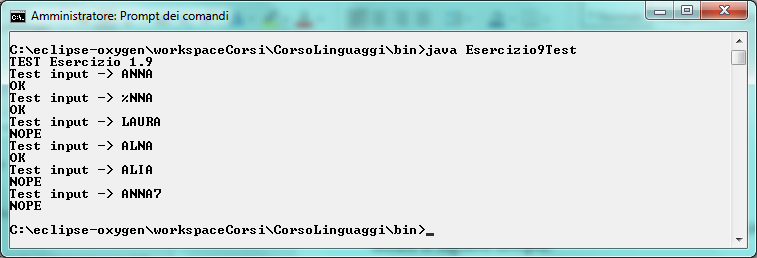
LAURA

ALNA

ALIA

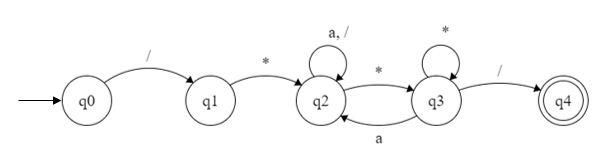
ANNA7

Sono riportati di seguito i risultati dei test (classe di test “*Esercizio9Test.java*”):



ESERCIZIO 1.10

Si definisce un DFA con alfabeto {/,\*,a} che riconosce il linguaggio di commenti delimitati da /\* (inizio) e \*/ (fine).



L’automa è implementato dal codice “*Esercizio10Commenti.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe:

/\*/

/\*\*/\*\*\*/

/\*\*\*\*/

/\*a\*a\*/

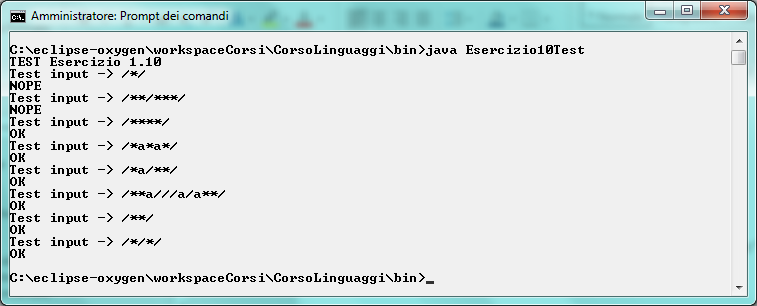
/\*a/\*\*/

/\*\*a///a/a\*\*/

/\*\*/

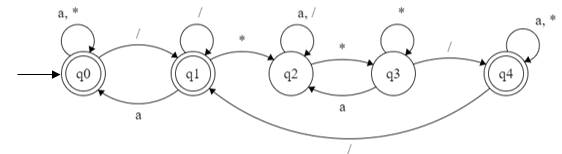
/\*/\*/

Sono riportati di seguito i risultati dei test (classe di test “*Esercizio10Test.java*”):



ESERCIZIO 1.11

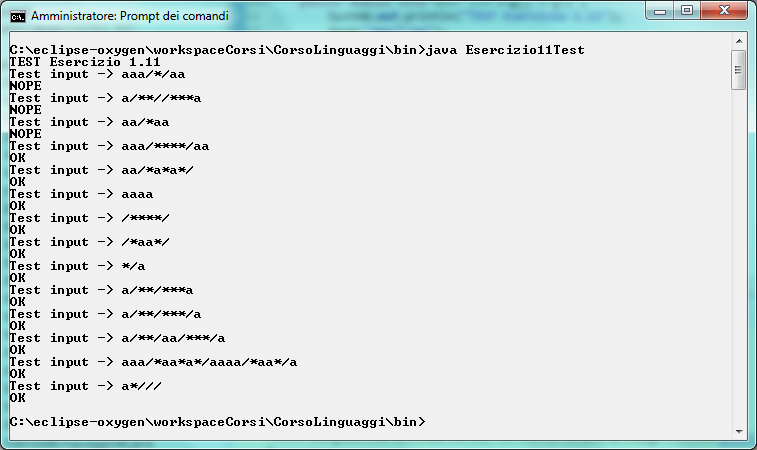
Si definisce un DFA con alfabeto {/,\*,a} che riconosce il linguaggio di commenti delimitati da /\* (inizio) e \*/ (fine) con la possibilità di far precedere o seguire il commento da ulteriori stringhe di caratteri del linguaggio considerato ammettendo anche commenti multipli.



L’automa è implementato dal codice “*Esercizio11Commenti.java*” fornito in allegato (folder PARTE\_1). Sono state testate le seguenti stringhe:

|  |  |
| --- | --- |
| aaa/\*/aa  a/\*\*//\*\*\*a  aa/\*aa  aaa/\*\*\*\*/aa  aa/\*a\*a\*/  aaaa  /\*\*\*\*/ | /\*aa\*/  \*/a  a/\*\*/\*\*\*a  a/\*\*/\*\*\*/a  a/\*\*/aa/\*\*\*/a  aaa/\*aa\*a\*/aaaa/\*aa\*/a  a\*/// |

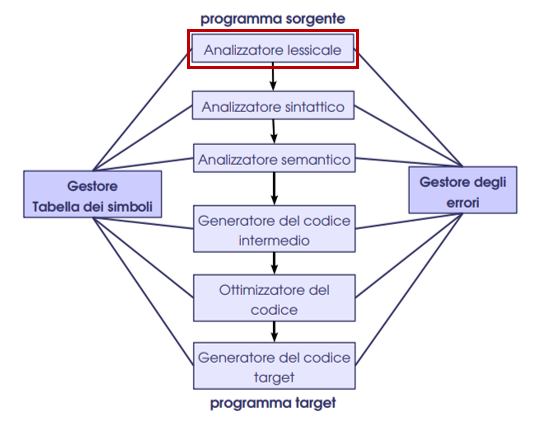
Sono riportati di seguito i risultati dei test (classe di test “*Esercizio11Test.java*”):



1. Analisi lessicale

Introduzione

Le sezioni dalla 2 alla 5 del seguente elaborato affrontano ciascuna una specifica fase del processo di compilazione rappresentato schematicamente nella figura sotto riportata.



L’analisi lessicale è, dunque, la prima fase ed il suo compito è quello di fornire uno stream di token a partire dalla sequenza di caratteri che rappresenta il programma che si vuole compilare. Lo stream di token prodotto dall’analisi lessicale costituisce l’input per le fasi successive (analisi sintattica). L’analizzatore lessicale legge ed analizza i caratteri in input fino al riconoscimento di un nuovo token. Se questo è un identificatore, viene creata una entrata corrispondente nella tabella dei simboli. Nel caso in cui nessun token venga riconosciuto, l’analizzatore lessicale segnala un errore e una possibile implementazione Java lancia un’eccezione apposita. L’analizzatore lessicale provvede anche ad eliminare dal testo gli spazi bianchi, che separano i token, ed i commenti. Questi ultimi, infatti, devono essere del tutto ignorati dal compilatore. Un altro compito importante è quello di associare i giusti numeri di riga con i messaggi di errore, in modo tale da facilitare la rintracciabilità dell’errore nel codice.

ESERCIZIO 2.1

Negli esercizi di questa sezione l’analizzatore lessicale viene realizzato tramite un programma Java organizzato sulle seguenti classi :

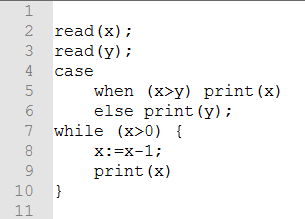
* ***Token***: simboli di parentizzazione, negazione ed operazioni fondamentali vengono trasformati in istanze di classe e ad ognuna di esse è associato un tag come attributo di istanza pari al codice ASCII del simbolo.
* ***Word***: estende la classe Token. Per tutte le parole chiave del linguaggio Java e per i simboli logici, di assegnazione e di confronto viene creata una variabile di istanza caratterizzata da due attributi di istanza: lessema e tag.
* ***Tag***: ad ogni parola chiave assegna un valore di tag. È costituita da un elenco di variabili di istanza costanti final.
* ***NumberTok***: estende la classe Token. Viene utilizzata per associare un token alle costanti numeriche che possono essere presenti nel codice. Tutte le variabili di istanza di questa classe sono caratterizzate da due attributi di istanza : number e tag.
* ***Lexer***: classe contenente il metodo main. È dotata di un insieme di funzioni di istanza che permettono di leggere il codice un carattere per volta ed associare il token corrispondente a seconda dei casi o segnalare errori. Tutti gli identificatori sono letti e riconosciuti, altrimenti, se nuovi, vengono memorizzati in una tabella di hash.

Le classi così impostate permettono di leggere un input da file e stampare la sequenza di token corrispondente.

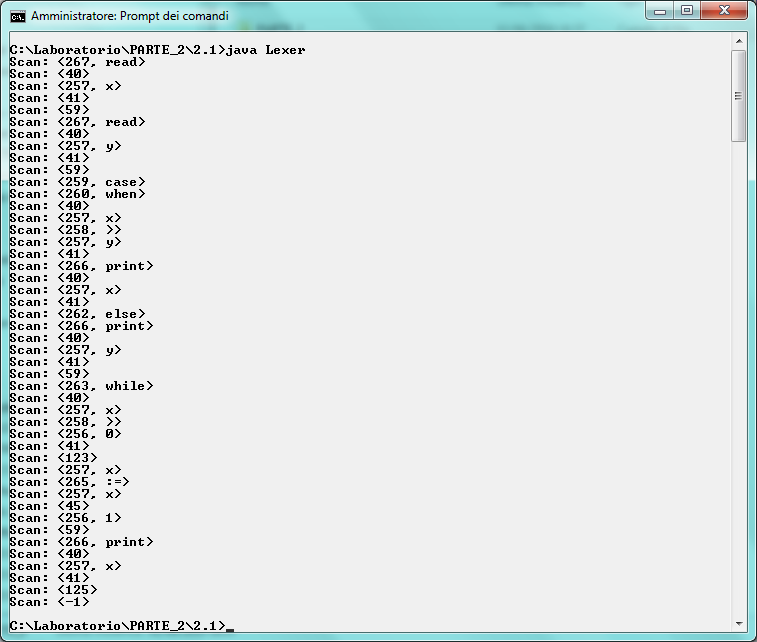
L’analizzatore lessicale è implementato dai codici *Tag.java, NumberTok.java, Word.java, Token.java, Lexer.java* forniti in allegato nel folder *PARTE\_2/2.1*.

Si riporta di seguito, a titolo di esempio, il risultato ottenuto con il file “*Esempio\_semplice.txt*”:

**Esempio\_semplice.txt**



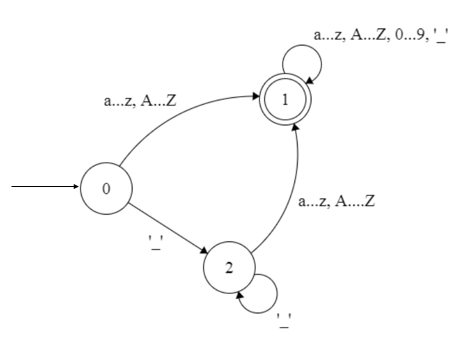
**Output**



ESERCIZIO 2.2

L’analizzatore lessicale realizzato in questo esercizio utilizza la medesima struttura e classi dell’esercizio precedente. Viene esteso, invece, il metodo *lexical\_scan* della classe *Lexer* in modo da gestire identificatori che corrispondono alla seguente espressione regolare:

In pratica, l’analizzatore lessicale deve riconoscere come identificatori ed associare, quindi, il relativo token oltre ad aggiungerli progressivamente nella tabella dei simboli se non già presenti, tutte le sequenze non vuote di lettere, numeri e il simbolo underscore, tale che non siano vuote e che non comincino con un numero o con underscore.

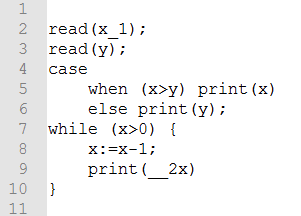


Dal punto di vista implementativo, il codice che si occupa della gestione del riconoscimento degli identificatori e delle parole chiave viene risolto da uno switch/case che implementa un automa a stati finiti sotto riportato di cui è stato omesso lo stato di errore.

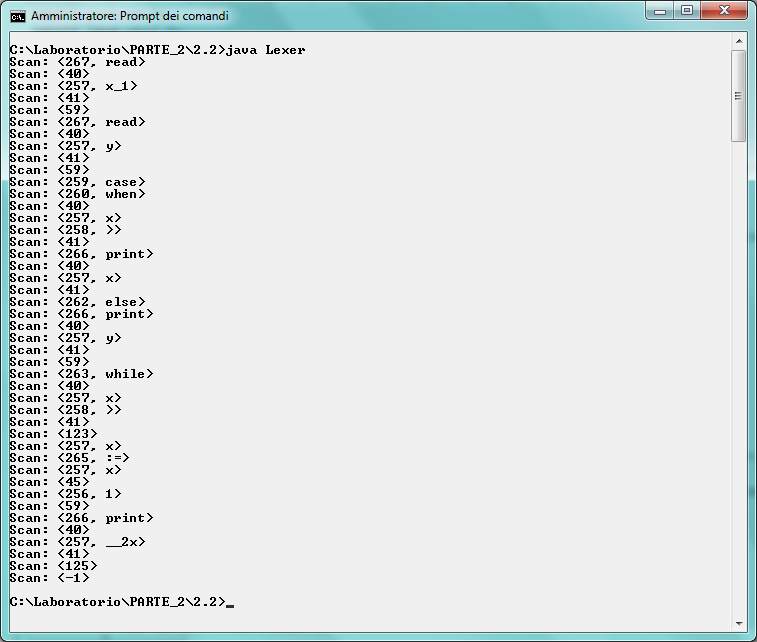
L’analizzatore lessicale è implementato dai codici *Tag.java, NumberTok.java, Word.java, Token.java, Lexer.java* forniti in allegato nel folder *PARTE\_2/2.2*.

Si riporta di seguito, a titolo di esempio, il risultato ottenuto con il file “*esempio\_semplice.txt*”:

**esempio\_semplice.txt**



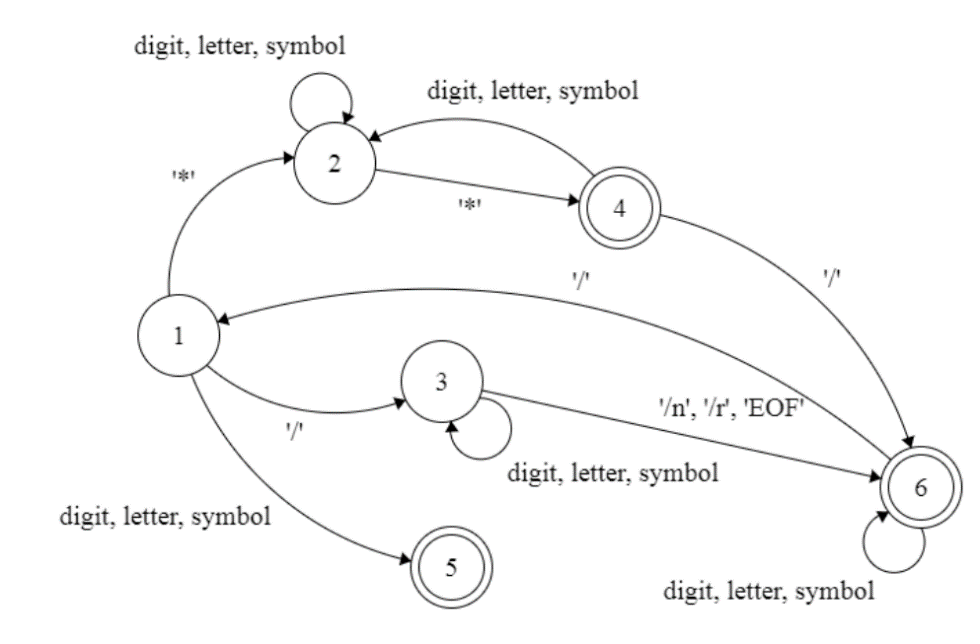
**Output**



ESERCIZIO 2.3

L’analizzatore lessicale utilizzato in questo esercizio utilizza la medesima struttura e classi dell’esercizio precedente. Viene esteso, invece, il metodo *lexical\_scan* della classe *Lexer* in modo che il programma riconosca ed ignori i commenti singoli o concatenati in tutte le loro modalità di scrittura previste da Java o, alternativamente, segnali un errore se un commento non è chiuso prima della fine del file.

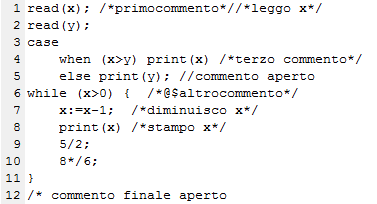
Dal punto di vista implementativo, il codice che si occupa della gestione del riconoscimento dei simboli di parentesi, operazione, confronto, assegnamento, negazione viene risolto da uno switch/case che implementa un automa a stati finiti sotto riportato.



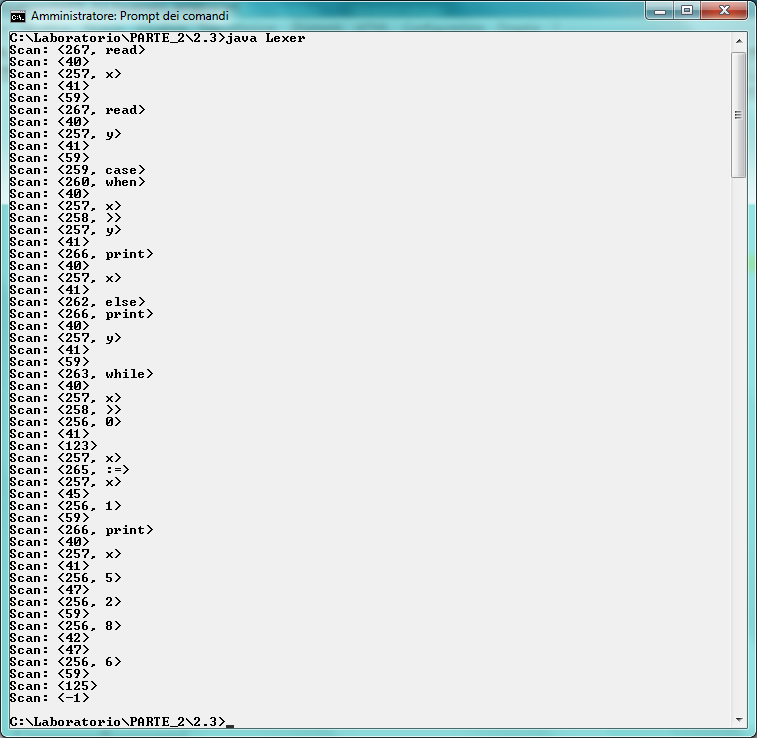
L’analizzatore lessicale è implementato dai codici *Tag.java, NumberTok.java, Word.java, Token.java, Lexer.java* forniti in allegato nel folder *PARTE\_2/2.3*.

Si riporta di seguito, a titolo di esempio, il risultato ottenuto con il file “Esempio\_semplice.txt”:

**Esempio\_semplice.txt**

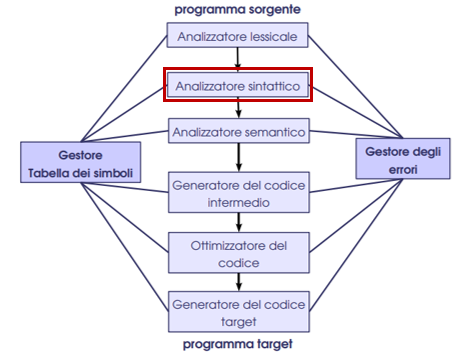


**Output**



1. Analisi sintattica

Introduzione



L’analisi sintattica, chiamata anche *parsing*, è la seconda fase del processo di compilazione. Prende in ingresso i token generati dalla fase precedente e ne verifica la correttezza sulla base di regole sintattiche tipicamente espresse tramite una grammatica libera dal contesto. In pratica, verifica che la stringa di token generata in precedenza appartiene al linguaggio generato dalla grammatica. L’output di questa fase, se correttamente eseguita, è l’albero di derivazione delle frasi grammaticali di cui il programma si compone. Nel caso, invece, in cui la successione di identificatori e simboli non sia coerente con le regole previste dalla grammatica, il *Parser*  segnala un errore ed interrompe il processo di derivazione (parsificazione). Si possono individuare due macrocategorie di *Parser*:

* ***Parser top-down***: parte dallo start symbol della grammatica e cerca di produrre tramite derivazioni progressive, una stringa di terminali che è identica al testo del programma sorgente.
* ***Parser bottom- up***: costruisce un albero sintattico partendo dalle foglie ed arrivando alal radice operando in verso contrario rispetto ad una derivazione destra.

Nell’ambito di questo laboratorio e di questa sezione sarà considerato il *Parser top-down a discesa ricorsiva*. In sintesi, si basa sulla scrittura di procedure ricorsive ricavate direttamente dalle regole grammaticali. Ogni produzione viene interpretata come una procedura in cui la parte sinistra rappresenta il nome e la parte destra (corpo della produzione) diventa, invece, la sua definizione.

ESERCIZIO 3.1

L’analizzatore sintattico di questo esercizio è un parser a discesa ricorsiva che utilizza l’analizzatore lessicale realizzato nella sezione precedente. Alle classi già viste nella sezione 2 si aggiunge la classe *Parser* che contiene il main e le funzioni associate a ciascun non terminale che verranno, quindi, richiamate ricorsivamente durante l’operazione di parsificazione.

La grammatica libera dal contesto di riferimento genera le espressioni aritmetiche molto semplici composte da numeri non negativi, operatori di somma, sottrazione, divisione e parentesi tonde.

Si riportano di seguito le produzioni che costituiscono la grammatica con associati i relativi insiemi guida.

|  |  |
| --- | --- |
| **PRODUZIONE** | **INSIEME GUIDA** |
|  |  |
| *<start> := <espr> EOF* |  |
| *<expr> := <term> <exprp>* |  |
| *<exprp> := + <term> <exprp>* |  |
| *<exprp> := - <term> <exprp>* |  |
| *<exprp> := ε* |  |
| *<term> := <fact> <termp>* |  |
| *<termp> := \* <fact> <termp>* |  |
| *<termp> := / <fact> <termp>* |  |
| *<termp> := ε* |  |
| *<fact> := (<expr>)* |  |
| *<fact> := NUM* |  |

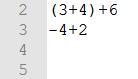
L’analizzatore sintattico è implementato dai codici *Tag.java, NumberTok.java, Word.java, Token.java, Lexer.java, Parser.java* forniti in allegato nel folder *PARTE\_3/3.1*.

Si riporta di seguito, a titolo di esempio, i risultati ottenuti con i file “*esempio\_op.txt*” e “*esempio\_op2.txt*:

**Esempio\_op.txt**



**Esempio\_op2.txt**

****

ESERCIZIO 3.2

Come avveniva per l’esercizio precedente, l’analizzatore sintattico di questo esercizio è un parser a discesa ricorsiva che utilizza l’analizzatore lessicale realizzato nella sezione precedente. Alle classi già viste nella sezione 2 si aggiunge la classe *Parser* che contiene il main e le funzioni associate a ciascun non terminale che verranno, quindi, richiamate ricorsivamente durante l’operazione di parsificazione.

La grammatica libera dal contesto di riferimento è quella di un semplice linguaggio di programmazione.

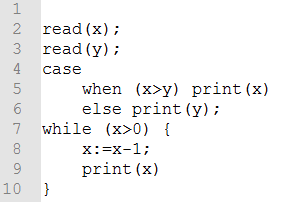
Si riportano di seguito le produzioni che costituiscono la grammatica con associati i relativi insiemi guida.

|  |  |
| --- | --- |
| **PRODUZIONE** | **INSIEME GUIDA** |
|  |  |
| *<prog> := <statlist> EOF* |  |
| *<statlist> := <stat> <statlist>* |  |
| *<statlistp>:= ;<stat> <statlistp>* |  |
| *<statlistp>:= ε* |  |
| *<stat> := ID := <expr>* |  |
| *<stat> := print (<expr>)* |  |
| *<stat> := read (ID)* |  |
| *<stat> := case <whenlist> else <stat>* |  |
| *<stat> := while (<bexpr>) <stat>* |  |
| *<stat> := {<statlist>}* |  |
| *<whenlist> := <whenitem> <whenlistp>* |  |
| *<whenlistp>:= <whenitem> <whenlistp>* |  |
| *<whenlistp>:= ε* |  |
| *<whenitem> := when (<bexpr>) <stat>* |  |
| *<bexpr> := <expr>RELOP <expr>* |  |
| *<exprp> := +<term> <exprp>* |  |
| *<exprp> := -<term> <exprp>* |  |
| *<exprp> := ε* |  |
| *<term> := <fact> <termp>* |  |
| *<termp>:= \*<fact> <termp>* |  |
| *<termp>:= /<fact> <termp>* |  |
| *<termp>:= ε* |  |
| *<fact>:=(<expr>)* |  |
| *<fact>:=NUM* |  |
| *<fact>:=ID* |  |

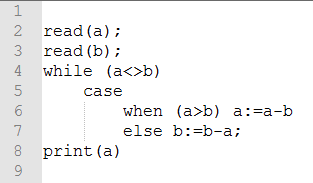
L’analizzatore sintattico è implementato dai codici *Tag.java, NumberTok.java, Word.java, Token.java, Lexer.java, Parser.java* forniti in allegato nel folder *PARTE\_3/3.2*.

Si riporta di seguito, a titolo di esempio, i risultati ottenuti con i file “*esempio\_semplice.txt*”, “*euclid.txt*”, “*factorial.txt*”, “*max\_tre\_num.txt*”:

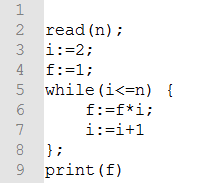
**esempio\_semplice.txt**



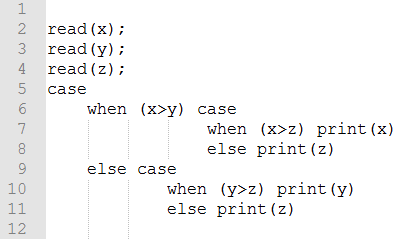
**euclid.txt**



**factorial.txt**

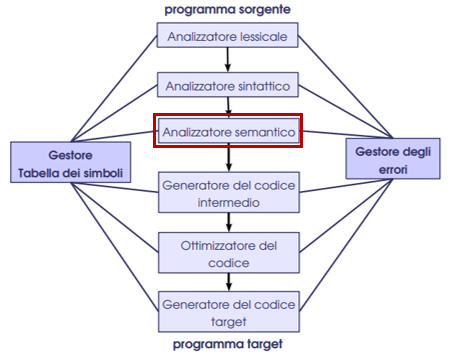


**max\_tre\_num.txt**



1. Analisi semantica o Traduzione diretta dalla sintassi

Introduzione



Definire la semantica di un linguaggio significa definire il valore di ogni espressione costruita applicando le produzioni della grammatica. Ad ogni non terminale della grammatica (e quindi ad ogni nodo dell’albero di derivazione per una data stringa in input) vengono associati uno o più attributi ed a ciascuna produzione una regola, detta regola semantica, che dice come calcolare il valore di una espressione a partire da quello delle sue sotto espressioni e così avanti ricorsivamente fino a giungere alle foglie dell’albero di derivazione dove il valore dell’attributo del terminale NUM è fornito dall’analizzatore lessicale.

ESERCIZIO 4.1

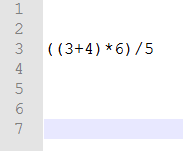
La grammatica annotata di riferimento per l’esercizio 4.1 è quella che permette di valutare espressioni aritmetiche semplici già analizzata nell’esercizio 3.1 al quale si rimanda per la tabella degli insiemi guida associati a ciascuna produzione.

Il valutatore di espressioni in questo esercizio è un programma ricorsivo che utilizza l’analizzatore lessicale realizzato nelle sezioni precedenti . Alle classi già viste nella sezione 2 si aggiunge la classe *Valutatore* che contiene il metodo main e le funzioni associate a ciascun non terminale che verranno, quindi, richiamate ricorsivamente durante l’operazione di valutazione dei valori degli attributi.

L’analizzatore semantico è implementato dai codici *Tag.java, NumberTok.java, Word.java, Token.java, Valutatore.java,* forniti in allegato nel folder *PARTE\_4/4.1*.

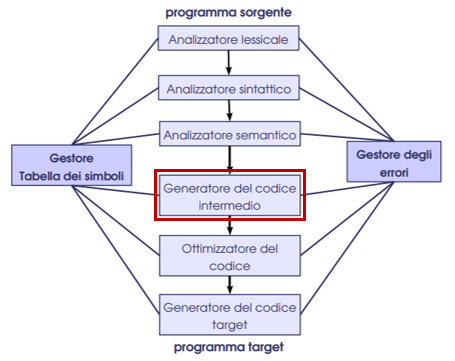
Si riporta di seguito, a titolo di esempio, i risultati ottenuti con i file “*esempio\_op.txt*”.

**esempio\_op.txt**

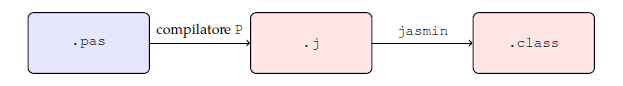


1. Generazione del codice intermedio

Introduzione



L’obiettivo di questa ultima fase è quello di implementare un traduttore per i programmi scritti in linguaggi di programmazione semplice di alto livello al fine di generare bytecode direttamente eseguibile dalla Java Virtual Machine. La JVM è una macchina immaginaria (astratta) la cui implementazione può essere effettuata attraverso l’utilizzo di un software di emulazione che venga eseguito su una macchina reale. I programmi che una JVM sono in grado di eseguire devono essere scritti in appositi file con estensione *.class*. Dato che per problemi di complessità non è semplice generare direttamente bytecode eseguibile dalla JVM, si passa attraverso il linguaggio assembler, un linguaggio mnemonico intermedio, che successivamente verrà tradotto nel formato *.class*.

Il file sorgente viene tradotto dal compilatore (oggetto della realizzazione) nel linguaggio assembler per la JVM. Questo file è poi trasformato in un file .class dal programma assembler Jasmin.

ESERCIZIO 5.1

Il traduttore realizzato in questa sezione fa riferimento alla grammatica dell’esercizio 3.2, che, però, deve essere preventivamente e correttamente annotata per la traduzione. Essa è basata sui seguenti principi:

* bisogna distinguere la sintassi delle espressioni booleane da quella delle espressioni aritmetiche;
* ogni espressione booleana è associata a due attributi ereditati ***.****true* e *.false* che rappresentano gli indirizzi cui viene passato il controllo nei cui l’espressione sia vera o falsa;
* ogni istruzione è associata a un attributo ereditato *.next* che rappresentano il label dell’istruzione successiva.
* l’attributo *.code* che contiene il codice è sintetizzato.

In pratica, ad ogni non terminale della grammatica si associano gli attributi *.next* e *.code* se questo non è coinvolto in espressioni booleane; in caso contrario si associano gli attributi *.true*, *.false* e *.code*.

In generale, gli attributi ereditati *.next, .true* e *.false* aumentano progressivamente e sono label che identificano il gruppo di istruzioni che il programma dovrà eseguire in sequenza a seconda dei salti condizionati o incondizionati che incontra. L’attributo sintetizzato *.code*, invece, contiene proprio il codice da eseguire, ossia l’elenco di istruzioni.

Di seguito si riporta lo schema di traduzione sviluppato per la grammatica dell’esercizio 3.2.

|  |  |
| --- | --- |
| **PRODUZIONE** | **INSIEME GUIDA** |
|  |  |
| *<prog> := <statlist> EOF* |  |
| *<statlist> := <stat> <statlist>* |  |
| *<statlistp>:= ;<stat> <statlistp>* |  |
| *<statlistp>:= ε* |  |
| *<stat> := ID := <expr>* |  |
| *<stat> := print (<expr>)* |  |
| *<stat> := read (ID)* |  |
| *<stat> := case <whenlist> else <stat>* |  |
| *<stat> := while (<bexpr>) <stat>* |  |
| *<stat> := {<statlist>}* |  |
| *<whenlist> := <whenitem> <whenlistp>* |  |
| *<whenlistp>:= <whenitem> <whenlistp>* |  |
| *<whenlistp>:= ε* |  |
| *<whenitem> := when (<bexpr>) <stat>* |  |
| *<bexpr> := <expr>RELOP <expr>* |  |
| *<exprp> := +<term> <exprp>* |  |
| *<exprp> := -<term> <exprp>* |  |
| *<exprp> := ε* |  |
| *<term> := <fact> <termp>* |  |
| *<termp>:= \*<fact> <termp>* |  |
| *<termp>:= /<fact> <termp>* |  |
| *<termp>:= ε* |  |
| *<fact>:=(<expr>)* |  |
| *<fact>:=NUM* |  |
| *<fact>:=ID* |  |