Documentazione del progetto.

Laboratorio di Linguaggi Formali e Traduttori

Corso B - Turno T4

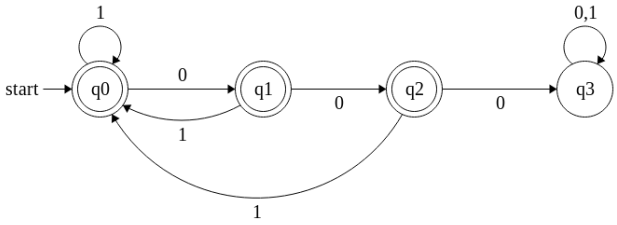
José Manuel Sánchez Aquilué

Indice

Sessione 1 (11/10/19)

Esercizio 1.1.

Per riconoscere il linguaggio complementare all'esempio meramente dobbiamo cambiare gli stati finali in non finali e viceversa.



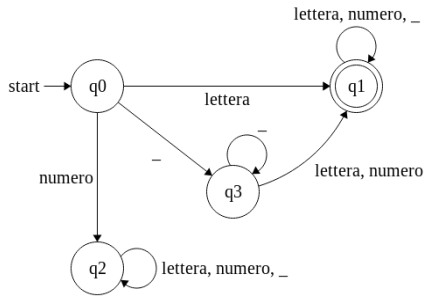
Ho testato il codice con questi esempi:

ε:		OK
0101 :	OK	
0011 :	OK	
11100 :		OK
1111 :	OK	
111000 :		NOPE
000111 :		NOPE
0110100011 :	NOPE	
12345 :		NOPE
ciao :	NOPE	

Come si può osservare, il risultato è il sperato.

Esercizio 1.2.

Visto che se gli stringhe che cominciano con un numero non possono esser un identificatore, li rifiuteremo sempre, indipendentemente dai ulteriori caratteri. Allo stesso modo, sempre saranno accettate gli stringhe che cominciano con una lettera e quelle che cominciano con "_" ma dopo abbiano altri simboli degli alfabeto.

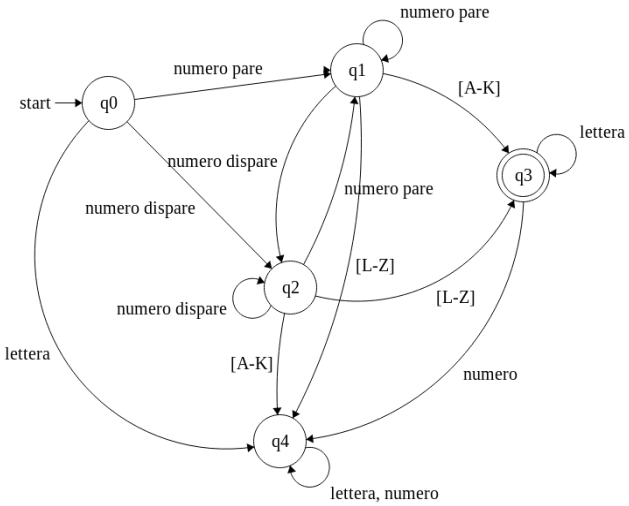


Ho testato il codice con questi esempi:
ε: NOPE
p : OK
ciao1 : OK
x2y2 : OK
x_1 : OK
lft_lab : OK
_temp : OK
x_1_y_2 : OK
x__ : OK
5 : NOPE
221B : NOPE
123 : NOPE
9_to_5 : NOPE
___ : NOPE
x&y : NOPE
as() : NOPE
12? : NOPE
?_jp : NOPE
Come si può osservare, ancora una volta ,il risultato è il sperato.

Esercizio 1.3.

Come dice il enunciato, il primo carattere deve esser un numero. Se il numero è pari, andrà allo stato 1; e se è dispari, al 2. Dopo, quando arrivi la prima lettera, se è nello stato 1 e la lettera è in [A-K], la stringa sarà accettata (sempre che non ce sia un altro numero dopo). Se, al contrario, si trova nello stato 2, la lettera dovrà essere in [L-Z] per potere essere accettata.

Nel mio codice ho usato aritmetica modulare per andare agli stati q1 e q2.



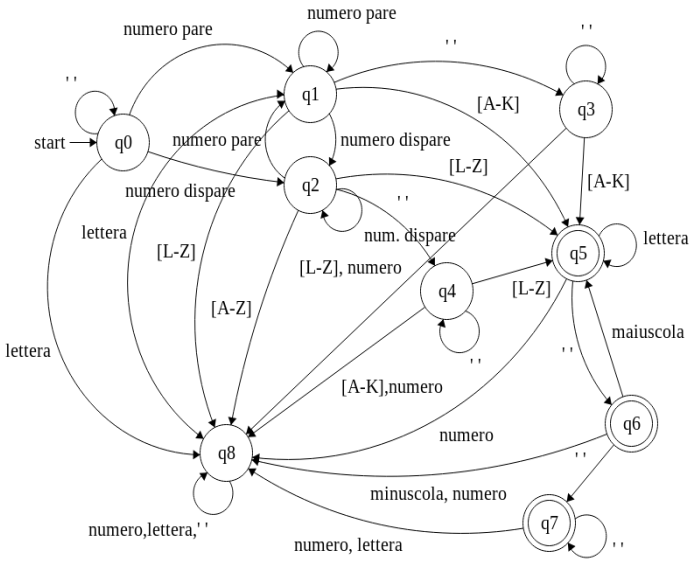
Ho provato il mio codice con questi esempi:
ε: NOPE
1A : NOPE
2K : OK
3L : OK
4Z : NOPE
1234aaaa : NOPE
Jose123 : NOPE
123456Bianchi : OK
654321Rossi : OK
654321Bianchi : NOPE
123456Rossi : NOPE
654321Rossi_ : NOPE
_654321Rossi : NOPE

Esercizio 1.4.

All'inizio, possiamo leggere spazi fino che il primo numero arrivi (se arriva una lettera, la respingeremo). Come nell'automa anteriore, se il numero è pari, andiamo al stato 1; e, se è dispari, al 2. Se il numero è pari, la stringa sarà accettata se il cognome comincia per una lettera in [A-K]. Se è dispari, deve cominciare per una lettera in [L-Z].

Però ora, tra il numero e il cognome posiamo trovare spazi. Per questo motivo, ho aggiunto gli stati 3 e 4, che riconoscano gli spazi fino che appaia la prima lettera.

Un'altra difficoltà sono i cognomi composti. Se, dopo di leggere una stringa di lettere, c'è un **unico** spazio seguito di un'altra stringa di lettere che comincia per **maiuscola**, il automa lo interpreterà come la seconda parte di un cognome composto. Di fatto, un cognome può essere composto di più di due parole.



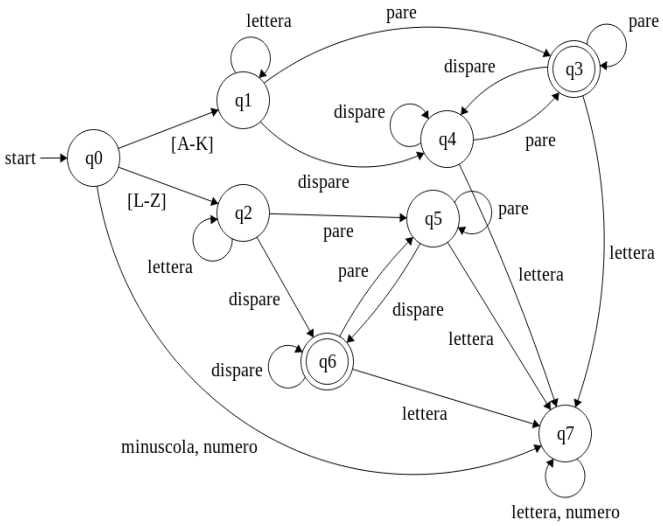
Per ultimo, dopo de la lettura del cognome possiamo trovare una stringa di spazi.

Ho testato il codice con questi stringhe:
ε: NOPE
: NOPE
1A : NOPE
2K : OK
3L : OK
4Z : NOPE
1234aaaa : NOPE
Jose123 : NOPE
123456Bianchi : OK
654321Rossi : OK
654321Bianchi : NOPE
123456Rossi : NOPE
1234 Rossi : NOPE
123 Rossi : OK
1234 Rossi : NOPE
123 Rossi : OK
1234 Bianchi : OK
123 Bianchi : NOPE
1234 Rossi : NOPE
123 Rossi : OK
1452 Da Vinci : OK
1452 Da Vinci : NOPE
1519 Da Vinci : NOPE
1519 Tra Vinci : OK
1519 Tra Vinci : NOPE
1452 da Vinci : NOPE

1519 Tra vinci : NOPE
1519 Tra Vinci : OK
654321Rossi? : NOPE
?654321Rossi : NOPE

Esercizio 1.5.

In questo caso, il primo carattere deve essere una maiuscola. Se la prima lettera è in [A-K], andiamo allo stato 1; se è in [L-Z] al 2. Da q1, dopo leggere il cognome, possiamo andar allo stato 3 se il numero è pari; o al stato 4, se è dispari. Da q5, ci sono altri stati per i numeri pari e dispari (q5 e q6, rispettivamente). Naturalmente, gli stati 3 e 6 saranno stati di accettazioni.



Lo ho provato con questi esempi:
ε: NOPE
Bianchi123456 : OK
Rossi654321 : OK
Bianchi654321 : NOPE
Rossi123456 : NOPE
A1 : NOPE
K2 : OK
L3 : OK
Z4 : NOPE
aaaa1234 : NOPE
Jose123 : NOPE
Rossi 654321 : NOPE
Rossi654321_ : NOPE
_Rossi654321 : NOPE

Esercizio 1.6.

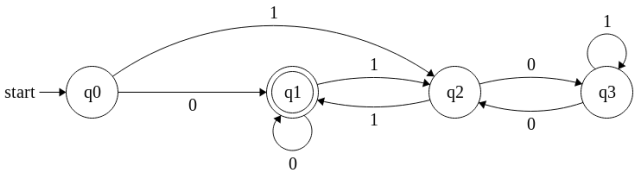
Ci sono tre casi: N mod 3 = 0, N mod 3 = 1 e N mod 3 = 2. Questi casi saranno i stati del nostro automa.

Se N è multiplo di 3, e li aggiungiamo un 0 per la destra; N' = 2*N, che è multiplo di 3. Se li aggiungiamo 1, N' = 2*N +1.

Andiamo a esaminare tutti i casi:

N mod 3 =	N' = 2*N + 0	N' mod 3	N'' = 2*N + 1	N'' mod 3
0	0	0	1	1
1	2	2	3	0
2	4	1	5	2

A partire da questa tabella possiamo costruire il automa.

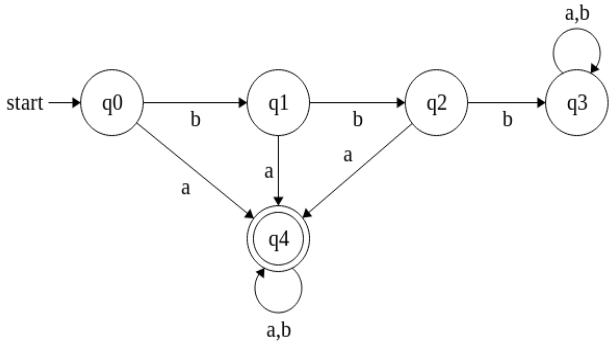


Ho testato il codice con questi esempi:
ε: NOPE
0 : OK
1 : NOPE
10 : NOPE
11 : OK
100 : NOPE
101 : NOPE
1111 : OK
10000 : NOPE
0011 : OK
0010 : NOPE
21 : NOPE

Sessione 2 (18/10/19)

Esercizio 1.7.

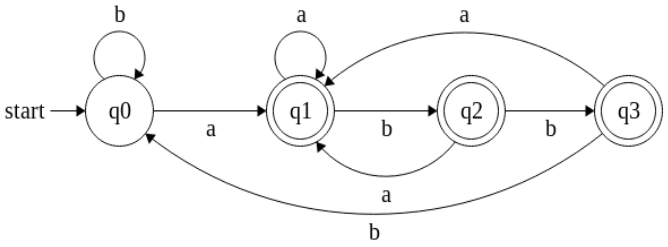
Da q0, q1 e q2 si cambierà di stato, a q4; se riconosce una ‘a’. Però da q3 (arriviamo a q3 dopo aver letto tre b all’inizio della stringa), la stringa sarà sempre rifiutata. Mentre che da q4 sarà sempre accettata.



Ho testato il codice con questi esempi:
ε: NOPE
abb : OK
abbbbbb : OK
bbaba :OK
baaaaaaa : OK
aaaaaaa : OK
a : OK
ba : OK
bba : OK
aa : OK
bbabbbbbbbb :OK
bbbababab : NOPE
b : NOPE
bbb : NOPE
ppp : NOPE

Esercizio 1.8.

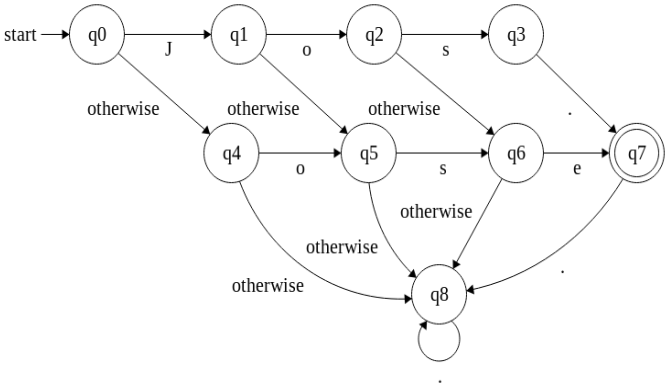
Lo stato q1 rappresenta le stringhe che finiscono in “a”, q2 quelle che finiscono in “ab”, q3 quelle che finiscono in “abb”. Pertanto sono stati di accettazione.



Lo ho provato con questi esempi:
ε: NOPE
abb : OK
bbaba :OK
baaaaaaa : OK
aaaaaaa : OK
a : OK
ba : OK
bba : OK
babbb :NOPE
aa : OK
bbbababab : OK
abbbbbb : NOPE
bbabbbbbbbb :NOPE
b : NOPE
r : NOPE

Esercizio 1.9.

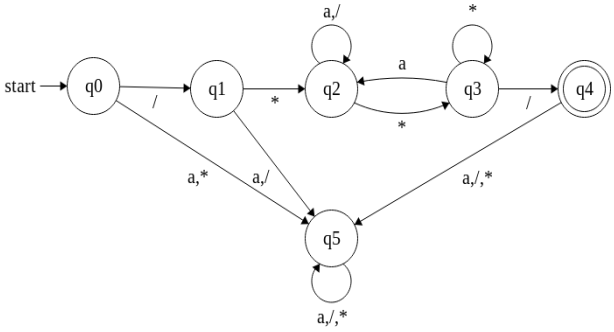
Prima di tutto, schiarire che in questa annotazione “otherwise” significa qualsiasi carattere tranne quello dell'altra transizione, e “.” significa qualsiasi carattere. Q0, q1, q2 e q3 rapresentano che quello che abbiamo letto corrisponde a un prefisso della stringa “Jose”. Mentre che in q4, q5 o q6 che quello che abbiamo letto è un prefisso della stringa “Jose” con una sola sostituzione. Poiché accettiamo solo sostituzioni, una stringa con meno o più di quattro caratteri sarà mai accettata.



Ha stato testato con queste input:
ε: NOPE
JOSE : NOPE
Jose : OK
jose : OK
Josea : NOPE
Jos : NOPE
Jxse : OK
Aose : OK
Jo&e : OK
J%%e : NOPE
Jos^ : OK

Esercizio 1.10.

Cominciamo rifiutando tutto che non comincia per “/*”. Dopo si legge tutti i caratteri fino che appare la stringa “*/”. Se la parola ha più caratteri dopo “*/”, andremo a q5 e sarà rifiutata. Inoltre, se non leggiamo “*/”, mai andremo a q4 e quindi la stringa non sarà accettata.



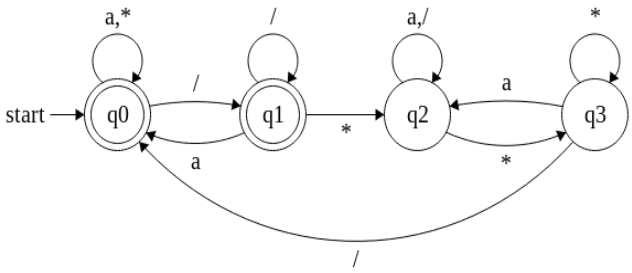
Lo ho provato con questi esempi e il risultato è stato come previsto.

ε: NOPE
/****/ : OK
/*a*a*/ : OK
/*a/**/ : OK
/**a///a/a**/ : OK
/**/ : OK
/*/*/ : OK
/*/ : NOPE
/*aaaa*//*aa*/ : NOPE
/**/***/ : NOPE
/*ciao*/ : NOPE

Esercizio 1.11.

Si accetta tutto che non contenga la stringa “/*”. Quando si legge “/*”, andiamo allo stato q2. Rimaniamo lì (o in q3) fino leggere “*/”. Se non lo leggiamo mai, non andremo a q0, e quindi rifiuteremo la parola.

In questo caso, è ciclico, può avere più di una sottostringa fra “/*” e “*/” .



Ho testato il mio automa con questi esempi:

: OK
aaa/****/aa : OK
aa/*a*a*/ : OK
aaaa : OK
/****/ : OK
/*aa*/ :OK
*/a : OK
a/**/***a : OK
a/**/***/a : OK
a/**/aa/***/a :OK
aaa/*/aa : NOPE
/* : NOPE

aa/* : NOPE
/*aa : NOPE
aas/*aa*/aa/* :NOPE
a/**//***a : NOPE
aa/*aa : NOPE

Sessione 3 (25/10/19)

Esercizio 2.1.

Il caso triviale consiste in leggere un carattere e restituire il token corrispondente. Ma ci sono altri casi in cui una parola comincia per lo stesso carattere che un token, ad esempio la parola “==” e il token “=”. In questi situazioni dobbiamo leggere il carattere ulteriori per determinare se è un token o una parola. Inoltre, c’è il caso di ‘&’ o ‘|’, che sempre devono appare in doppi dello stesso simbolo.

Se lo letto non comincia per una simbolo speciale, ma è una sequenza di lettere (e numeri), lo interpretiamo come una parola. Se è una parola il primo carattere deve esser una lettera, dopo può avere qualunque numero, ma mai un simbolo speciale. Una volta che abbiamo letto la stringa, mediante comparazioni, verifichiamo se è una parola riservata. Se non è una parola riservata, sarà un identificatore.

Può essere che il primo carattere letto sia un numero. Quindi si leggere tutto fino che troviamo qualcosa di diverso da un numero. Questo programma non legge gli zeri a sinistra.

La differenza tra la classe parola e la classe numero è che una salva l'informazione come stringa e l’altra come intero.

In primo luogo, ho testato il mio codice con un file che non genera errori.

Input:

```
do{
        ciao = 100;
        moltiNumeri = 000000212
        cond ( moltiNumeri == ciao)
        then
                num = 12+56-90*34/12
        else
                bool = ((1<2)&&(0>2))||ciao<=12

        when (!bool)
                print(num >= moltiNumeri)
} while( read(file) )
```

```
Scan: <264, do>
Scan: <123>
Scan: <257, ciao>
Scan: <61>
Scan: <256, 100>
Scan: <59>
```

Scan: <257, moltiNumeri>
Scan: <61>
Scan: <256, 0>
Scan: <256, 0>
Scan: <256, 0>
Scan: <256, 0>
Scan: <256, 0>
Scan: <256, 0>
Scan: <256, 212>
Scan: <259, cond>
Scan: <40>
Scan: <257, moltiNumeri>
Scan: <258, ==>
Scan: <257, ciao>
Scan: <41>
Scan: <261, then>
Scan: <257, num>
Scan: <61>
Scan: <256, 12>
Scan: <43>
Scan: <256, 56>
Scan: <45>
Scan: <256, 90>
Scan: <42>
Scan: <256, 34>
Scan: <47>
Scan: <256, 12>
Scan: <262, else>
Scan: <257, bool>
Scan: <61>
Scan: <40>
Scan: <40>
Scan: <256, 1>
Scan: <258, <>
Scan: <256, 2>
Scan: <41>
Scan: <269, &&>
Scan: <40>
Scan: <256, 0>
Scan: <258, >>
Scan: <256, 2>
Scan: <41>
Scan: <41>
Scan: <268, ||>
Scan: <257, ciao>

Scan: <258, <=>
Scan: <256, 12>
Scan: <260, when>
Scan: <40>
Scan: <33>
Scan: <257, bool>
Scan: <41>
Scan: <266, print>
Scan: <40>
Scan: <257, num>
Scan: <258, >=>
Scan: <257, moltiNumeri>
Scan: <41>
Scan: <125>
Scan: <263, while>
Scan: <40>
Scan: <267, read>
Scan: <40>
Scan: <257, file>
Scan: <41>
Scan: <41>
Scan: < -1>

Dopo, lo ho testato con casi più critici nei quali dovrebbe avere dei errori.

Input: _id
Output: Erroneous character: _

Input: while (a | b) {}
Output: Scan: <263, while>
Scan: <40>
Scan: <257, a>
Erroneous character after | :
Scan: null

Sessione 4 (8/11/19)

Esercizio 2.2.

Adesso durante la lettura degli identificatori, consideriamo che possono contenere il carattere '_'. Però abbiamo bisogno di utilizzare un flag che prenda il valore true quando leggiamo un numero o una lettera. Di questa maniera, il flag terrà il valore falso quando il identificatore sia una stringa composta solo da '_'.

Al fine di verificare, ho usato lo steso input che nell'esercizio 1.2.

Input.
p
ciao1
x2y2
x_1
lft_lab
_temp
x_1_y_2
x__
5
221B
123
9_to_5
___
x&y
as()
12?
?_jp

Output:
Scan: <257, p>
Scan: <257, ciao1>
Scan: <257, x2y2>
Scan: <257, x_1>
Scan: <257, lft_lab>
Scan: <257, _temp>
Scan: <257, x_1_y_2>
Scan: <257, x__>
Scan: <256, 5>
Scan: <256, 221>
Scan: <257, B>
Scan: <256, 123>

Scan: <256, 9>
Scan: <257, _to_5>
Erroneous identifier___
Erroneous character after & : y

Scan: <257, as>
Scan: <40>
Scan: <41>

Scan: <256, 12>
Erroneous character: ?

Erroneous character: ?

Esercizio 2.3.

Quando leggiamo un simbolo ‘/’, dobbiamo anche leggere il seguente. Se è un altro ‘/’, si ha bisogno ignorare tutti fino trovare un a capo o un EOF. Se al contrario è un ‘*’, si ignora tutto tipi di caratteri fino trovare una stringa “*/”. Quando un commentari delimitati non è chiuso, lanceremo un errore.

Esempio 1:
Input:
/*calcolare la velocita*/
(= d 300) // distanza
(= t 10) // tempo
(print(*d t))

Output: <40> <61> <257, d> <256, 300> <41> <40> <61> <257, t> <256, 10> <41> <40> <266, print> <40> <42> <257, d> <257, t> <41> <41> < -1>

Esempio 2:
Input: (= d 300) /*distanza*//*da Torino a Lione*/
Output: <40> <61> <257, d> <256, 300> <41> < -1>

Esempio 3:
Input: x*/y
Output: <257, x> <42> <47> <257, y> ←1>

Esempio 4:
Input:
(print(*d t))
// prima EOF

Output: <40> <266, print> <40> <42> <257, d> <257, t> <41> <41> ←1>

Esempio 5:
Input:
(print(*d t))
/*
senza chiudere
*

Output: <40> <266, print> <40> <42> <257, d> <257, t> <41> <41>
Error: comment not closed.

Sessione 5 (15/11/19)

Esercizio 3.1.

Dalla grammatica dell'esercizio calcoliamo gli insiemi guida di ogni regola.

Espressioni.	GUIDA( )
⟨start⟩ → ⟨expr⟩ EOF	{(, num}
⟨expr⟩ → ⟨term⟩ ⟨exprp⟩	{(, num}
⟨exprp⟩ → + ⟨term⟩ ⟨exprp⟩	{+}
⟨exprp⟩ → - ⟨term⟩	{-}
⟨exprp⟩ → ε	{), EOF}
⟨term⟩ → ⟨fact⟩ ⟨termp⟩	{(, num}
⟨termp⟩ → * ⟨fact⟩ ⟨termp⟩	{*}
⟨termp⟩ → / ⟨fact⟩ ⟨termp⟩	{/}
⟨termp⟩ → ε	{+, -,), EOF}
⟨fact⟩ → (⟨expr⟩)	{(}
⟨fact⟩ → NUM	NUM

Con questi insiemi possiamo fare il controllo di errori di nostro programma. Ora, dobbiamo riscrivere la grammatica in Java. Per ogni simboli non-terminali si invoca un metodo e per ogni simboli terminale si chiama al metodo Match con il simboli come parametro.

Test:

Esempio 1:
Input: 1+1

Output: token = <256, 1>
token = <43>
token = <256, 1>
token = <-1>
Input OK

Esempio 2:
Input: (8)-3
Output: <40> <256, 8> <41> <45> <256, 3> <-1>
Input OK

Esempio 3:
Input: 1+2-3
Output: <256, 1> <43> <256, 2> <45> <256, 3> <-1>
Input OK

Esempio 4:
Input: 12/2+2
Output: <256, 12> <47> <256, 2> <43> <256, 2> <-1>
Input OK

Esempio 5:
Input: 12*2/2
Output: <256, 12> <42> <256, 2> <47> <256, 2> <-1>
Input OK

Esempio 6:
Input: (90*2)-3
Output: <40> <256, 90> <42> <256, 2> <41> <45> <256, 3> <-1>
Input OK

Esempio 7:
Input: 90*(2-3)
Output: <256, 90> <42> <40> <256, 2> <45> <256, 3> <41> <-1>
Input OK

Esempio 8:
Input: (90-2*4)
Output: <40> <256, 90> <45> <256, 2> <42> <256, 4> <41> <-1>
Input OK

Esempio 9:
Input: 100+(12*2)
Output: <256, 100> <43> <40> <256, 12> <42> <256, 2> <41> <-1>
Input OK

Esempio 10:
Input: (100+12)*2
Output: <40> <256, 100> <43> <256, 12> <41> <42> <256, 2> <-1>
Input OK

Esempio 11:
Input: 100/2/2/2
Output: <256, 100> <47> <256, 2> <47> <256, 2> <47> <256, 2> <-1>
Input OK

Esempio 12:
Input: 100/2/(2/2)
Output: <256, 100> <47> <256, 2> <47> <40> <256, 2> <47> <256, 2> <41> <-1>
Input OK

Esempio 13:
Input: 1+1+1+1+1+1+1+1+1+1+1+1
Output: <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <-1> Input OK

Esempio 14:
Input: 1000/2/2/2/2/2/2
Output: <256, 1000> <47> <256, 2> <47> <256, 2> <47> <256, 2> <47> <256, 2> <47> <256, 2> <47> <256, 2> <47> <256, 2> <-1>
Input OK

Esempio 15:
Input: 12+a
Output: <256, 12> <43> <257, a>
Exception in thread "main" java.lang.Error: near line 1: error in term

Sessione 6 (22/11/19)

Esercizio 3.2.

All'eguali che nell'esercizio precedente, devo cominciare calcolando l'insieme GUIDA. In questa occasione, il calcolo (first, follow e guida) è stato più tedioso, perché la grammatica era più lunga.

Espressioni.	GUIDA( )
⟨prog⟩ → ⟨stat⟩ EOF	{(}
⟨statlist⟩ → ⟨stat⟩ ⟨statlistp⟩	{(}
⟨statlistp⟩ → ⟨stat⟩ ⟨statlistp⟩	{(}
⟨statlistp⟩ → ε	{)}
⟨stat⟩ → (⟨statp⟩)	{(}
⟨statp⟩ → = ID ⟨expr⟩	{=}
⟨statp⟩ → cond ⟨bexpr⟩ ⟨stat⟩ ⟨elseopt⟩	{cond}
⟨statp⟩ → while ⟨bexpr⟩ ⟨stat⟩	{while}
⟨statp⟩ → do ⟨statlist⟩	{do}
⟨statp⟩ → print ⟨exprlist⟩	{print}
⟨statp⟩ → read ID	{read}
⟨elseopt⟩ → (else ⟨stat⟩)	{(}
⟨elseopt⟩ → ε	{)}
⟨bexpr⟩ → (⟨bexprp⟩)	{(}
⟨bexprp⟩ → RELOP ⟨expr⟩ ⟨expr⟩	{RELOP}
⟨expr⟩ → NUM	{NUM}
⟨expr⟩ → ID	{ID}
⟨expr⟩ → (⟨exprp⟩)	{(}
⟨exprp⟩ → + ⟨exprlist⟩	{+}
⟨exprp⟩ → - ⟨expr⟩ ⟨expr⟩	{-}
⟨exprp⟩ → * ⟨exprlist⟩	{*}
⟨exprp⟩ → / ⟨expr⟩ ⟨expr⟩	{/}
⟨exprlist⟩ → ⟨expr⟩ ⟨exprlistp⟩	{num, id, (}
⟨exprlistp⟩ → ⟨expr⟩ ⟨exprlistp⟩	{num, id, (}
⟨exprlistp⟩ → ε	{)}

Adesso facciamo quello dell'esercizio anteriore, riscrivere la grammatica in Java. E finalmente, lo testiamo con il codice di moodle.

Input:

```
(do
   (print (+ (/ 10 5) 2 56 3 2 3) 34 35)
   (read x)
   (cond (> x 10)
      (do
         (read y)
      )
   (else (print x 12))
   )
   (while (== y 2)
      (= x (+ x 1))
   )
)
```

Output:

```
token = <40>
token = <264, do>
token = <40>
token = <266, print>
token = <40>
token = <43>
token = <40>
token = <47>
token = <256, 10>
token = <256, 5>
token = <41>
token = <256, 2>
token = <256, 56>
token = <256, 3>
token = <256, 2>
token = <256, 3>
token = <41>
token = <256, 34>
token = <256, 35>
token = <41>
token = <40>
token = <267, read>
token = <257, x>
token = <41>
```

token = <40>
token = <259, cond>
token = <40>
token = <258, >>
token = <257, x>
token = <256, 10>
token = <41>
token = <40>
token = <264, do>
token = <40>
token = <267, read>
token = <257, y>
token = <41>
token = <41>
token = <40>
token = <262, else>
token = <40>
token = <266, print>
token = <257, x>
token = <256, 12>
token = <41>
token = <41>
token = <41>
token = <40>
token = <263, while>
token = <40>
token = <258, ==>
token = <257, y>
token = <256, 2>
token = <41>
token = <40>
token = <61>
token = <257, x>
token = <40>
token = <43>
token = <257, x>
token = <256, 1>
token = <41>
token = <41>
token = <41>
token = <41>
token = <-1>
Input OK

Sessione 7 (25/11/19)

Esercizio 4.1.

Prima di tutto, ho preso il codice della sessione 5. In questi casi, come è un SDT, i metodi ritornano un intero e hanno degli variabili locali e argomenti. Se al codice della grammatica della sezione 5 gli aggiungiamo questi attributi e con loro facciamo i calcoli e le assegnazioni indicati nell'enunciato, otteniamo una calcolatrice che mostra sullo screen il risultato dell'espressione aritmetica dell'input.

Una difficoltà di questa pratica era prendere il valore numerico del token. Abbiamo bisogno di fare un cast di Token a NumberTok, così:

```
((NumberTok)look).numero;
```

Ho testato il mio codice con gli stessi esempi che nella sessione 5.

Esempio 1:
Input: 1+1

Output: token = <256, 1>
token = <43>
token = <256, 1>
token = <-1>
2

Esempio 2:
Input: (8)-3
Output: <40> <256, 8> <41> <45> <256, 3> <-1>
5

Esempio 3:
Input: 1+2-3
Output: <256, 1> <43> <256, 2> <45> <256, 3> <-1>
0

Esempio 4:
Input: 12/2+2
Output: <256, 12> <47> <256, 2> <43> <256, 2> <-1>
8

Esempio 5:
Input: 12*2/2
Output: <256, 12> <42> <256, 2> <47> <256, 2> <-1>
12

Esempio 6:
Input: (90*2)-3
Output: <40> <256, 90> <42> <256, 2> <41> <45> <256, 3> <-1>
177

Esempio 7:
Input: 90*(2-3)
Output: <256, 90> <42> <40> <256, 2> <45> <256, 3> <41> <-1>
-90

Esempio 8:
Input: (90-2*4)
Output: <40> <256, 90> <45> <256, 2> <42> <256, 4> <41> <-1>
82

Esempio 9:
Input: 100+(12*2)
Output: <256, 100> <43> <40> <256, 12> <42> <256, 2> <41> <-1>
124

Esempio 10:
Input: (100+12)*2
Output: <40> <256, 100> <43> <256, 12> <41> <42> <256, 2> <-1>
224

Esempio 11:
Input: 100/2/2/2
Output: <256, 100> <47> <256, 2> <47> <256, 2> <47> <256, 2> <-1>
12

Esempio 12:
Input: 100/2/(2/2)
Output: <256, 100> <47> <256, 2> <47> <40> <256, 2> <47> <256, 2> <41> <-1>
50

Esempio 13:
Input: 1+1+1+1+1+1+1+1+1+1+1+1
Output: <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43>
<256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> <43> <256, 1> ←1>
12

Esempio 14:
Input: 1000/2/2/2/2/2/2

Output: <256, 1000> <47> <256, 2> <47> <256, 2> <47> <256, 2> <47> <256, 2> <47> <256, 2> <47> <256, 2> <-1>
15

Esempio 15:
Input: 12+a
Output: <256, 12> <43> <257, a>
Exception in thread "main" java.lang.Error: near line 1: error in term

Sessione 8 (25/11/19)

Esercizio 5.1.

Prima di tutto prendiamo la grammatica del esercizio 3.2, e per ogni variabile della grammatica che riconosce le stringhe generate da A viene creato un metodo. Il insieme guida viene usato per sapere la procedura corrispondente (come negli esercizi precedenti).

Nei metodi prog, statlist e statlistp si crea una nuova label che attua come argomento di stat. Quando stat finisce, questa label è emessa.

Quando troviamo una assegnazione, cerchiamo la variabile; e se non esiste, si crea. Dopo si chiama a expr. Se è una istruzione read, facciamo lo stesso ma invece di chiamare a expr, emitiamo invokestatic, con il flag 0, e istore.

Nei condizionali, prima si manda come argomento di bexpr una nuova label. Successivamente la istruzione goto viene emessa con una nuova label (corrispondente al else statement). Alla fine questa label sarà utilizzata come argomento di elseopt, ma prima dobbiamo processare gli istruzioni successivi (invocare stat con il argomento lnext). Il metodo elseopt comprova si c'è un else statement.

Con i loop while, si crea una etichetta inic e subito dopo inviare come argomento una nuova label a bexpr. Questa label sarà emessa in una istruzione goto. Finalmente si processa il codice ulteriore.

Quando arrivi un print, exprlist sarà chiamada e invokestatic verrà emessa con il flag 1. Se è un do, meramente si chiamarà a statlist.

Il metodo bexprp comprova che tipo di comparazione ha letto e emette la ordine corrispondente. Exprp fa lo stesso ma con gli espressioni aritmetiche.

Il codice ha stato testato con gli esempi e i risultati hanno stato gli sperati.

Input:

```
(do
   (read x)
   (read y)
   (cond (> x y) (print x) (else (print y)))
   (while (> x 0)
      (do
         (= x (- x 1))
         (print x)
      )
   )
)
```

Output.j:

```
.method public static run()V
 .limit stack 1024
 .limit locals 256
 invokestatic Output/read()I
 istore 0
L1:
 invokestatic Output/read()I
 istore 1
L2:
 iload 0
 iload 1
 if_icmpgt L4
 goto L5
L4:
 iload 0
 invokestatic Output/print(I)V
 goto L3
L5:
 iload 1
 invokestatic Output/print(I)V
L3:
L7:
 iload 0
 ldc 0
 if_icmpgt L8
 goto L6
L8:
 iload 0
 ldc 1
 isub
 istore 0
L9:
 iload 0
 invokestatic Output/print(I)V
L10:
 goto L7
L6:
L0:
 return
.end method
```

Input:

```
(do
        (= x 10)
        (= y 0)
        (while (> x 0) (do
                (= y (+ x y))
                (= x  (- x 1))
        ))
)
```

Output.j:

```
.method public static run()V
 .limit stack 1024
 .limit locals 256
 ldc 10
 istore 0
L1:
 ldc 0
 istore 1
L2:
L4:
 iload 0
 ldc 0
 if_icmpgt L5
 goto L3
L5:
 iload 0
 iload 1
 iadd
 istore 1
L6:
 iload 0
 ldc 1
 isub
 istore 0
L7:
 goto L4
L3:
L0:
 return
.end method
```

Input:

```
(do
    (read a)
    (read b)
    (while (<> a b)
        (cond (> a b) (= a (- a b)) (else (= b (- b a))))
    )
    (print a)
)
```

Output.j:

```
.method public static run()V
 .limit stack 1024
 .limit locals 256
 invokestatic Output/read()I
 istore 0
L1:
 invokestatic Output/read()I
 istore 1
L2:
L4:
 iload 0
 iload 1
 if_icmpne L5
 goto L3
L5:
 iload 0
 iload 1
 if_icmpgt L6
 goto L7
L6:
 iload 0
 iload 1
 isub
 istore 0
 goto L3
L7:
 iload 1
 iload 0
 isub
 istore 1
 goto L4
L3:
 iload 0
```

```
 invokestatic Output/print(I)V
L8:
L0:
 return
.end method
```

Input:

```
(do
   (read n)
   (= i 2)
   (= f 1)
   (while (<= i n)
      (do
         (= f (* f i))
         (= i (+ i 1))
      )
   )
   (print f)
)
```

Output.j:

```
.method public static run()V
 .limit stack 1024
 .limit locals 256
 invokestatic Output/read()I
 istore 0
L1:
 ldc 2
 istore 1
L2:
 ldc 1
 istore 2
L3:
L5:
 iload 1
 iload 0
 if_icmple L6
 goto L4
L6:
 iload 2
 iload 1
 imul
 istore 2
L7:
```

```
 iload 1
 ldc 1
 iadd
 istore 1
L8:
 goto L5
L4:
 iload 2
 invokestatic Output/print(I)V
L9:
L0:
 return
.end method
Input:
```

```
(do
    (read x)
    (read y)
    (read z)
    (cond (> x y)
        (cond (> x z) (print x) (else (print z)))
        (else
            (cond (> y z) (print y) (else (print z)))
        )
    )
)
```

Output.j:

```
.method public static run()V
 .limit stack 1024
 .limit locals 256
 invokestatic Output/read()I
 istore 0
L1:
 invokestatic Output/read()I
 istore 1
L2:
 invokestatic Output/read()I
 istore 2
L3:
 iload 0
 iload 1
 if_icmpgt L5
 goto L6
```

```
L5:
 iload 0
 iload 2
 if_icmpgt L7
 goto L8
L7:
 iload 0
 invokestatic Output/print(I)V
 goto L4
L8:
 iload 2
 invokestatic Output/print(I)V
 goto L4
L6:
 iload 1
 iload 2
 if_icmpgt L9
 goto L10
L9:
 iload 1
 invokestatic Output/print(I)V
 goto L6
L10:
 iload 2
 invokestatic Output/print(I)V
L4:
L0:
 return
.end method
```