



ΠΑΝΕΠΙΣΤΗΜΙΟ  
ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ  
UNIVERSITY OF WEST ATTICA

*Σχολή Μηχανικών*

*Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών*

*Εργαστήριο «Μεταγλωττιστές»*

*Μέρος Α-2 : Κωδικοποίηση αυτομάτων πεπερασμένων καταστάσεων μέσω FSM*

*Ημερομηνία Αποστολής: 1/5/2024*

*Τμήμα Β2 - Ομάδα 2*

ΚΟΝΤΟΥΛΗΣ ΔΗΜΗΤΡΙΟΣ 21390095

ΜΕΝΤΖΕΛΟΣ ΑΓΓΕΛΟΣ ΚΩΝΣΤΑΝΤΙΝΟΣ 21390132

ΒΑΡΣΟΥ ΕΥΦΡΟΣΥΝΗ 21390021

ΓΚΙΟΖΙ ΕΝΤΕΡΙΣΑ 21390041

ΑΛΕΞΟΠΟΥΛΟΣ ΛΕΩΝΙΔΑΣ 21390006

## Περιεχόμενα

<b>1. Εισαγωγή.....</b>	<b>3</b>
<b>2. Τεκμηρίωση.....</b>	<b>3</b>
2.1 Κανονικές Εκφράσεις .....	3
2.2 Κώδικας FSM.....	5
2.3 Διαγράμματα Πεπερασμένων Καταστάσεων .....	8
2.4 Πίνακες Μετάβασής .....	10
<b>3. Ενιαίο Πεπερασμένο Αυτόματο .....</b>	<b>12</b>
3.2 Διάγραμμα Πεπερασμένων Καταστάσεων .....	14
3.3 Γενικός Πίνακας Μετάβασης.....	14
<b>4. Περιπτώσεις Ελέγχου.....</b>	<b>15</b>
4.1 Κανονικές Εκφράσεις .....	15
4.2 Κώδικες FSM .....	17
<b>5. Ανάλυση αρμοδιοτήτων .....</b>	<b>22</b>
5.1 Γενικές Αρμοδιότητες .....	22
5.2 Υλοποίηση Word .....	23

## 1. Εισαγωγή

Το παρών έγγραφο αποτελεί την κωδικοποίησή και ανάλυση των λεκτικών μονάδων και αναγνωριστικών της γλώσσας Uni-C σε πεπερασμένα αυτόματα. Ειδικότερα ως λεκτικές μονάδες, εννοούμε τις πηγαίες λέξεις που αναγνωρίζονται από ένα πρότυπο αναγνώρισης μέσω ενός λεκτικού αναλυτή. Οι λεκτικές μονάδες χρειάζεται να αναγνωρίζονται για την έπειτα χρήση τους από στον συντακτικό αναλυτή ενώ τα αναγνωριστικά αναγνωρίζονται από τον λεκτικό αναλυτή αλλά παραλείπονται από τον συντακτικό.

Στην γλώσσα Uni-C, ως λεκτικές μονάδες έχουμε τα αναγνωριστικά (identifiers), τις λέξεις κλειδιά (keywords), τα κυριολεκτικά (λεκτικά και αριθμητικά) και τους τελεστές (operators). Επιπλέον, ως αναγνωριστικά έχουμε τους διαχωριστές χαρακτήρων (ακολουθίες κενών και tab) και τα σχόλια που, όπως αναφέραμε, δεν αποτελούν λεκτικές μονάδες αλλά αναγνωρίζονται και αυτά από πρότυπα και επομένως πρέπει ο λεκτικός αναλυτής να τα αναγνωρίζει.

Επομένως, στην τεκμηρίωση της εργασίας μας αναλύουμε αρχικά σε κανονικές εκφράσεις, κωδικοποιούμε και σχεδιάζουμε τα πεπερασμένα αυτόματα και τους πίνακες μεταβάσεις τους των παραπάνω λεκτικών μονάδων (εκτός των keywords) και των σχολίων. Τους διαχωριστές χαρακτήρων τους ενσωματώνουμε στο ενιαίο πεπερασμένο αυτόματο που αναγνωρίζει όλες τις λεκτικές μονάδες και τα σχόλια.

Ειδικότερα, το κεφάλαιο της τεκμηρίωσης αναλύει τα επιμέρους (πέντε) πεπερασμένα αυτόματα παραθέτοντας όλα τους κώδικες, σχήματα και πίνακες μετάβασης. Το κεφάλαιο με το ενιαίο πεπερασμένο αυτόματο, ενσωματώνει όλα τα επιμέρους αυτόματα σε ένα ενιαίο παραθέτοντας αντίστοιχα την κωδικοποίησή του, το σχήμα του και τον γενικό πίνακα μετάβασης του. Στο κεφάλαιο με τις περιπτώσεις ελέγχου παραθέτονται ορισμένοι έλεγχοι λειτουργίας των επιμέρους αυτόματων και του ενιαίου αυτόματου σε εικόνες, επιβεβαιώνοντας την σωστή τους λειτουργία. Στο τελευταίο κεφάλαιο αναλύονται η αρμοδιότητες κάθε μέλους της ομάδας μας στην υλοποίηση της εργασίας.

## 2. Τεκμηρίωση

Ακολουθεί η ανάλυση και κωδικοποίηση κατά σειρά των αναγνωριστικών (identifiers), των λεκτικών κυριολεκτικών (string literals), των αριθμητικών κυριολεκτικών (numerical literals) και των σχολίων (comments).

Αρχικά, για το καθένα παραθέτουμε την κανονική έκφραση που το περιγράφει σύμφωνα με το πρότυπο αναγνώρισης που αναγράφεται στο έγγραφο ανάλυσης της γλώσσας Uni-C. Στη συνέχεια, σύμφωνα με την κάθε κανονική έκφραση την κωδικοποιούμε σε πεπερασμένο αυτόματο, με χρήση του εργαλείου fsm, που αναγνωρίζει την λεκτική μονάδα ή αναγνωριστικό καταλήγοντας δηλαδή σε μια κατάσταση αποδοχής ή όχι. Σχεδιάζουμε το σχήμα σύμφωνα με τον κώδικα του πεπερασμένου αυτόματου (Microsoft Visio) και τέλος σύμφωνα με το σχήμα κατασκευάζουμε τον πίνακα μετάβασης που περιέχει πληροφορίες σχετικά με τις μεταβάσεις που μπορεί να πραγματοποιήσει το αυτόματο από μία κατάσταση σε μία άλλη.

### 2.1 Κανονικές Εκφράσεις

Ακολουθεί η περιγραφή και διατύπωση της κάθε κανονικής έκφρασης των λεκτικών μονάδων και των σχολίων.

#### Αναγνωριστικά/Identifiers

`[a-zA-Z_]+[a-zA-Z_0-9]*`

Το πρώτο σύνολο μέσα στις αγκύλες ταιριάζει κάθε λατινικό χαρακτήρα (μικρό ή κεφάλαιο) και την \_ (κάτω παύλα) και το + περιγράφει ότι πρέπει να εμφανίζεται πάνω 1 ή περισσότερες φορές. Έτσι, υποχρεώνουμε ο πρώτος χαρακτήρας να είναι λατινικό γράμμα ή κάτω παύλα (πρέπει να υπάρξει μια φόρα τουλάχιστον). Το δεύτερο σύνολο στις δεύτερες αγκύλες ταιριάζει το προηγούμενο σύνολο που αναφέραμε με προσθήκη των αριθμητικών ψηφίων (0-9) και το \* περιγράφει ότι πρέπει να εμφανίζεται 0 ή περισσότερες φορές. Επομένως, καταφέραμε να μην μπορέσουν τα αριθμητικά ψηφία να εμφανιστούν ως πρώτοι χαρακτήρες σε ένα αναγνωριστικό, γιατί αναγκαστικά πρέπει να γραφτεί ένας χαρακτήρας ή μια κάτω παύλα για να γραφτεί μετά ένας αριθμός.

## Λεκτικά Κυριολεκτικά/String Literals

```
"([^\\"*(\\[\\n\\"])[^\\"]*))"
```

Τα Strings πρέπει να αρχίζουν και να τελειώνουν με " (quotation marks) οπότε για αυτό τα βάζουμε στην αρχή και στο τέλος του regex. Το πρώτο σύνολο (αγκύλες) μέσα στην παρένθεση περιγράφει όλα τα σύμβολα εκτός (^) από το " (quotation mark) και το \ (backslash) 0 ή περισσότερες φορές (\*). Έτσι, μπορούμε να έχουμε οποιονδήποτε χαρακτήρα, αριθμό ή σύμβολο μέσα στο string ή και κανένα διατυπώνοντας δηλαδή το κενό string. Το δεύτερο σύνολο περιγράφει την δυνατότητα να έχουμε escape sequences μέσα στο string, δηλαδή υποχρεώνοντας το πρώτο \ και μετά μπορεί να ακολουθήσει μόνο \, n ή " και μετά πάλι μπορεί να ακολουθήσει οποιοσδήποτε χαρακτήρας, αριθμός ή σύμβολο όπως προηγουμένως. Επομένως, καταφέραμε να έχουμε empty strings, escape sequences και οποιονδήποτε χαρακτήρα μέσα σε ένα λεκτικό κυριολεκτικό.

## Αριθμητικά Κυριολεκτικά/Numerical Literals

```
(0[xX][\dA-F]+)|(0[0-7]+)|(((1-9)\d*|0)(\.\d+)?([eE][-]?((1-9)\d*|0))?)?)
```

Στην πρώτη περίπτωση βρίσκω τους αριθμούς οι οποίοι είναι σε δεκαεξαδική μορφή δηλαδή που ξεκινάνε από 0x ή 0X περιέχουν αριθμούς από το 0 έως το 9 ή και κεφαλαία λατινικά γράμματα από το A έως το F. Στην συνέχεια ελέγχω τους οκταδικούς αριθμούς που μπορεί ο αριθμός να περιέχει από 0 έως 7. Τέλος βρίσκω όλους τους ακραίους και τους δεκαδικούς. Αρχικά βρίσκω αν ο αριθμός θα είναι μεγαλύτερος του μηδενός ή αν είναι 0. Στην συνέχεια αν ο αριθμός είναι δεκαδικός μετά την υποδιαστολή ταιριάζω όλους τους αριθμούς μια φορά το λιγότερο (\d+). Τέλος, ελέγχω αν ο αριθμός είναι υψωμένος σε δύναμη με το σύμβολο e ή με το E και μετά ή θα περιέχεται μια μόνο φορά το αρνητικό πρόσημο μαζί με κάποιο ακέραιο αριθμό (πρώτα να είναι μεγαλύτερος του μηδενός και μετά να περιέχονται πάνω από 0 φορές όλοι οι αριθμοί \d\*) ή μηδέν. Αν δεν είναι αρνητικό το πρόσημο πάλι υψώνεται σε ακέραιο αριθμό ή μηδέν.

## Τελεστές/Operators

```
([+\-\\*\\/\\=\\!\\<\\>\\=])|([\\*\\/\\=\\!\\<\\>%]|&{1,2}|\\-{1,2}|\\+{1,2}|(\\|\\|))
```

Η πρώτη περίπτωση μέσα σε παρένθεση εξασφαλίζει ότι το παραπάνω σύνολο από τελεστές πρέπει να ακολουθούνται από το ίσων (=). Η δεύτερη περίπτωση (OR |) απλώς εξασφαλίζει να τυπώνονται οι τελεστές που δεν αποτελούν με έναν άλλον τελεστή κάποιο ζευγάρι. Η τρίτη περίπτωση εξασφαλίζει τους τελεστές που μπορούν να εμφανιστούν μία φορά μόνοι τους αλλά και αποτελούν ζευγάρι με τον εαυτό τους. Η τελευταία περίπτωση εξασφαλίζει να ταιριάζει ο || (LOGICAL OR) τελεστής. Με τις παραπάνω περιπτώσεις, εξασφάλισαμε το ταίριασμα όλων των τελεστών που αναγράφονται στο πρότυπο.

## Σχόλια/Comments

```
\\/\\*(\\.|\\n)*?\\/\\*\\/|\\/\\/.
```

Τα σχόλια μπορούν να διατυπωθούν με δύο τρόπους: με δύο slash (//) και έπειτα οποιονδήποτε άλλο χαρακτήρα (εκτός whitespace characters) ή αρχίζοντας και τελειώνοντας αντίστοιχα με /\* \*/ και εμπεριέχοντας οποιονδήποτε χαρακτήρα αναμεσά τους (συμπεριλαμβανομένων των whitespace characters). Η πρώτη OR εξασφαλίζει τον δεύτερο τρόπο που περιγράψαμε με . (τελεία) να εννοούμε οποιονδήποτε χαρακτήρα εκτός του newline (οπότε για αυτό βάλαμε σε OR το \n γιατί τα σχόλια μπορούν υπάρχουν και σε άλλη γραμμή). Η δεύτερη OR εξασφαλίζει τον πρώτο τρόπο που απλώς ξεκινάει με // και έχει έπειτα οποιονδήποτε χαρακτήρα (.).

## 2.2 Κώδικας FSM

Ακολουθεί η κωδικοποίηση των επιμέρους πεπερασμένων αυτομάτων. Ο σχολιασμός σε τι αποσκοπεί κάθε κατάσταση υπάρχουν σαν σχόλια μέσα στα .fsm αρχεία. Γενικά, το \n το χρησιμοποιούμε για να τερματίσει το fsm πρόγραμμα και δεν αντικατροπτρίζεται στις κανονικές εκφράσεις ή στα σχήματα/πίνακες μετάβασης των επιμέρους πεπερασμένων αυτόματων. Οπότε για αυτό στους περισσότερους κώδικες στην κατάσταση GOOD δεχόμαστε το \n και ξαναπάμε στην κατάσταση αποδοχής.

### Αναγνωριστικά/Identifiers

```
START=SZ
// the SZ state is the state that checks for the start of the identifier

SZ: A-Z a-z _      -> S0 // if it starts with any letter, uppercase or lowercase, or _, go to S0
*                 -> BAD // anything else like digits (0-9) for example is not accepted

S0: a-z A-Z _ 0-9 -> S0 // if the start of the name is correct, check the rest of the identifier
    \n            -> GOOD // if its just a single letter, or just one underscore (_), its acceptable
    *            -> BAD // anything else is bad

GOOD(OK):
```

### Λεκτικά Κυριολεκτικά/String Literals

```
START=SZ
SZ: "            -> S0 // string is opened
    *            -> BAD // if its anything else it's not a string

// when we are in this state (S0) we are inside a string

S0: *            -> S1 // anything inside the string is accepted, and we go to S1
    \\           -> S2 // if its a slash go to S2 to check for \, " or n
    "            -> GOOD // string is closed, go to GOOD (Accepting State)

S1: \\           -> S2 // if there is a \ and then go to S2 to check for another one (\\ translates to \ in a string)
    "            -> GOOD // in this case the string is closed, so we go to GOOD (Accepting State)
    *            -> S1 // if you receive anything (inside the string) keep looping to S1

S2: \\ " n -> S1 // check for a second \ (\\), if the string closes ("), or if there is a new line character (\n)
    *            -> BAD // anything else is not accepted

GOOD(OK): \n -> GOOD
```

```

START=S0
S0: 0      -> S1 // if it's a 0, then we have multiple options. Go to S1 to check for more
1-9      -> S2 // digits 1-9 are considered an integer. Go to S2 to check for more
*        -> BAD // anything else is not valid

S1: .      -> S3 // this is the case where we have 0. which is a floating point number
e E      -> S4 // here we have the case of 0e which is the exponential (0e0 is allowed)
X x      -> S6 // 0x or 0X is about Hexadecimal.
0-7      -> S7 // any number starting with zero and followed by any digit 0-7 is Octal
\n       -> GOOD // zero by itself
*        -> BAD // everything else is not accepted

// Integer
S2: 0-9    -> S2 // if we have any digits 1-9, we check for more digits 0-9
.         -> S3 // here we have the left part of a decimal point number (for example 12.)
e E       -> S4 // in this case we have an integer followed by the exponential sign (power)
\n        -> GOOD // in this case an integer has been formed
*         -> BAD // everything else not valid

// Float
S3: 0-9    -> S10 // this regards the digits right of the decimal point
*         -> BAD // everything else not valid

// State regarding decimal point numbers and decimal point exponential numbers
S10: 0-9   -> S10 // keep checking if there are more digits on the right side of the decimal point
e E       -> S4 // here we have the left part of an exponential float (2.5e or 2.5E)
\n        -> GOOD // in this case a floating point number is formed
*         -> BAD // everything else not valid

// Exponential
S4: 1-9    -> S8 // this regards the right part of the exponential after the e/E. Basically it's the power
-         -> S5 // this is the case of a negative exponential ([0-9]e-. Right part of minus is on S5)
0         -> GOOD // number raised to the power of 0
*         -> BAD // anything else not valid

// Negative Exponential
S5: 1-9    -> S8 // check for digits in the negative exponential (5e-10 for example)
*         -> BAD // anything else is not accepted

// Hexadecimal
S6: A-F 0-9 -> S9 // this regards the right part of a hexadecimal number after the x/X.
*         -> BAD // anything else not accepted

S9: A-F 0-9 -> S9 // more than one hex digits after the x/X
\n        -> GOOD // combinations like 0xF and 0xFFFF for example are accepted
*         -> BAD // anything else not accepted

// Octal
S7: 0-7    -> S7 // check for other octal digits
\n        -> GOOD // octal numbers like 0147, 063 etc. are accepted
*         -> BAD // anything else not accepted

S8: 0-9    -> S8 // this checks for a multiple digit power on the exponential
\n        -> GOOD // any exponential that has one or more digits is accepted
*         -> BAD // anything else not accepted

GOOD(OK) : \n ->GOOD

```

```

START=SZ
SZ: \* / \= ! < > -> S1 // These operators can be written alone, or followed by an equal sign (=)
    |          -> S2 // if | is written then it's about an OR logical operation, so we go to S2
+          -> S3 // the plus sign, depending on the following sign, can mean 3 different operations.
-          -> S4 // the minus sign can also mean 3 different operations like the plus sign.
&          -> S5 // if & is written then it's about a logical AND operation, so we go to S5
%          -> GOOD // this is the modulus operation which is only one percentage sign (%)
*          -> BAD // anything else is not accepted because it's not an operator

// state regarding * / = ! < >
S1: \n \=      -> GOOD // if its just a single operator (* / ...), or followed by an equal sign (=)
    *          -> BAD // anything else is not accepted

// state regarding | sign
S2: |          -> GOOD // check if there is another |. If there is then it's an OR operation (||)
    *          -> BAD // if it's only one | and anything else, then it's not valid

// state regarding plus sign(+)
S3: \n \+ \=    -> GOOD // the operations can be addition (+), increment (++) or +=
    *          -> BAD // if it's anything else after the plus sign then it's not accepted

// state regarding minus sign(-)
S4: \n \- \=    -> GOOD // the operations can be subtraction (-), increment (--) or -=
    *          -> BAD // if it's just one minus and anything else after then it's not accepted.

//state regarding ampersand (&)
S5: \n &        -> GOOD // if there is another & sign then it's a logical AND operation (&&)
    *          -> BAD // if it's only one & sign followed by anything else it's not valid

GOOD(OK): \n -> GOOD

```

## Σχόλια/Comments

```

START = SZ
SZ: /          -> S0 // comment is started
    *          -> BAD // if it's anything else is not a comment

S0: /          -> S1 // if there is another bracket, then it's a one line comment
    \*         -> S2 // if there is an asterisk followed, then it's about a multi-line comment
    *          -> BAD // if it's anything else then it's not valid

S1: \n         -> GOOD // in this case we have a single line comment (//)
    *          -> S1 // here we are inside the comment, so pretty much everything is accepted

S2: \*         -> S3 // in this case we have /* which is a multi-line comment and we check if there is an
asterisk
    *          -> S2 // everything else is not accepted

S3: /          -> GOOD // if there is a slash (/) after the asterisk (from S2), the multi-line comment closes
    *          -> BAD // anything else does not regard the comment

GOOD(OK): \n ->GOOD

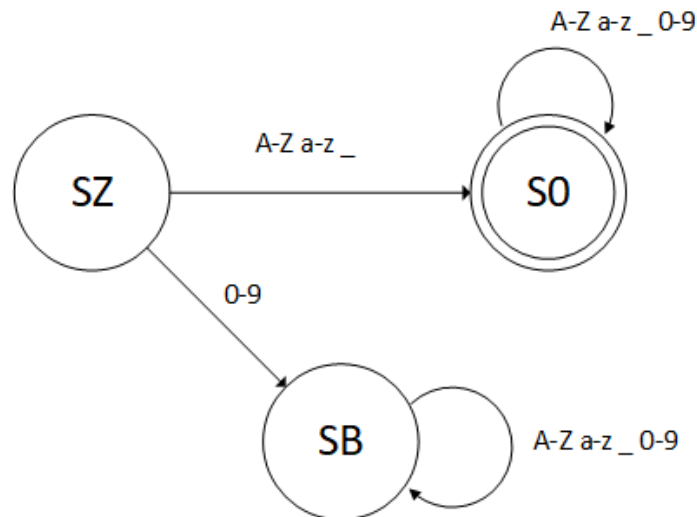
```

## 2.3 Διαγράμματα Πεπερασμένων Καταστάσεων

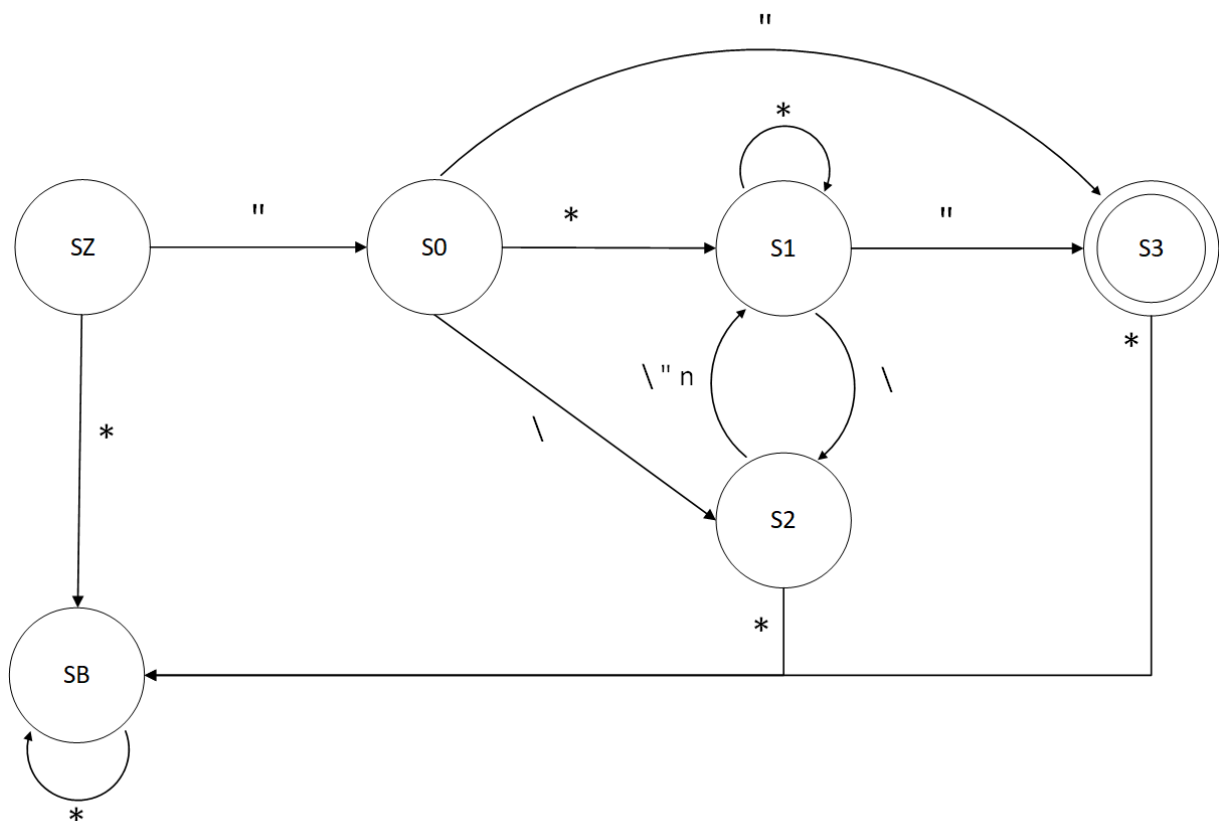
Ακολουθούν τα επιμέρους σχήματα των πεπερασμένων αυτομάτων. Η κατάσταση SB είναι η BAD κατάσταση που περιγράφουμε στον κώδικα και η SG είναι η GOOD κατάσταση. Με την μετάβαση (\*) εννοούμε τον οποιαδήποτε άλλο χαρακτήρα που δεν έχει αναφερθεί μετάβαση. Επιπλέον, στον κώδικα τα έχουμε όλα να καταλήγουν στην κατάσταση αποδοχής GOOD γιατί όπως είπαμε χρειαζόμαστε σαν τερματισμό το \n. Στα σχήματα όμως, ορίζουμε και καταστάσεις αποδοχής αυτές που στον κώδικα με \n πάνε στην κατάσταση GOOD. Αυτό το κάνουμε έτσι ώστε τα σχήματα να φαίνονται πιο περιεκτικά και σύντομα.

Σημείωση: όλα τα επιμέρους σχήματα καταλήγουν στην κατάσταση SB (BAD) εκτός από το σχήμα για τα αριθμητικά κυριολεκτικά. Αυτό έγινε λόγω μεγάλης πολυπλοκότητας του σχήματος αν όλες οι καταστάσεις κατέληγαν εκεί. Εννοείται ότι όποια άλλη μετάβαση υπάρξει καταλήγει στην κατάσταση SB (BAD).

### Αναγνωριστικά/Identifiers

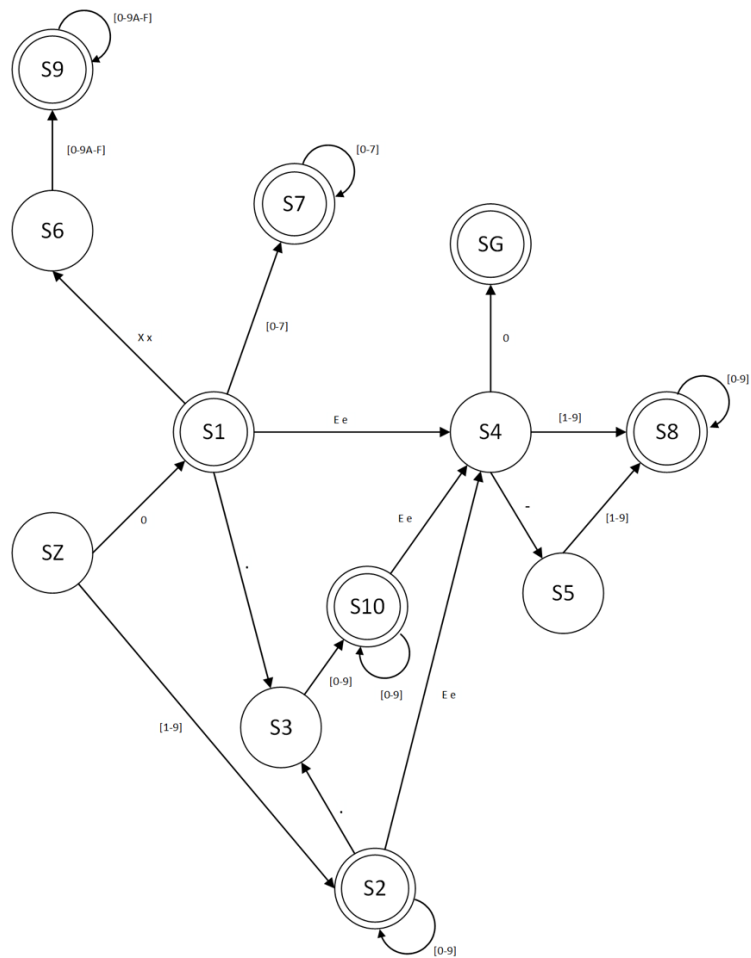


### Λεκτικά Κυριολεκτικά/String Literals

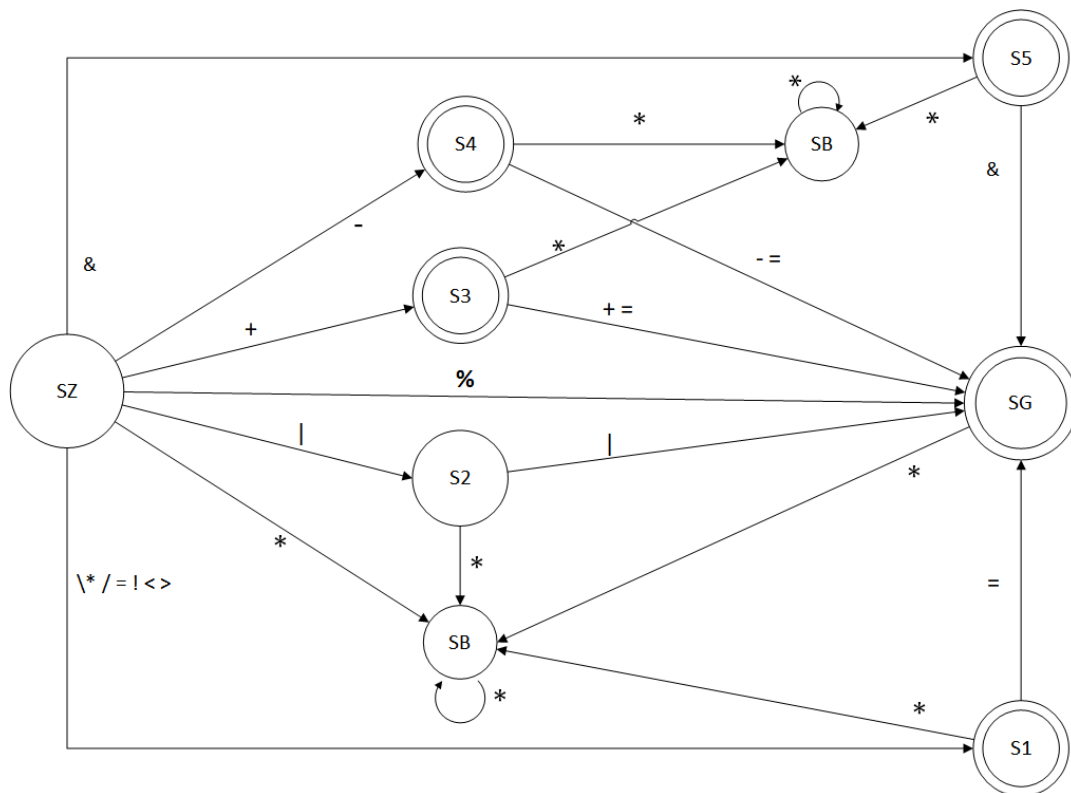


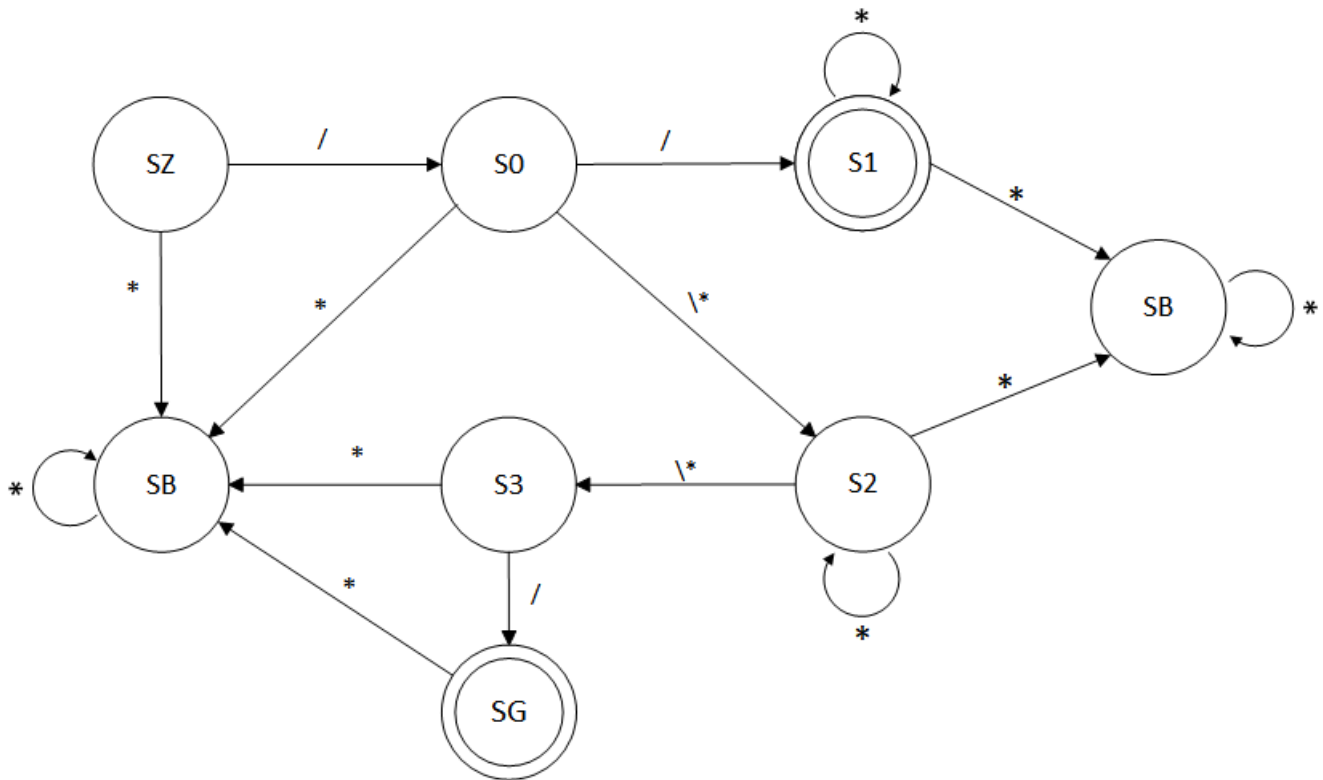


## Αριθμητικά Κυριολεκτικά/Numerical Literals



## Τελεστές/Operators





## 2.4 Πίνακες Μετάβασής

Ακολουθούν οι πίνακες μεταβάσεις για κάθε επιμέρους πεπερασμένο αυτόματο. Με το (\*) εννοούμε οποιαδήποτε άλλη μετάβαση δεν έχει αναφερθεί από τις υπάρχουσες μεταβάσεις.

### Αναγνωριστικά/Identifiers

	A-Z	a-z	_	0-9	* / OTHER
SZ	S0	S0	S0	BAD	BAD
S0	S0	S0	S0	S0	BAD
SG	BAD	BAD	BAD	BAD	BAD
SB / BAD	BAD	BAD	BAD	BAD	BAD

### Λεκτικά Κυριολεκτικά/String Literals

	"	\	n	* / OTHER
SZ	S0	BAD	BAD	BAD
S0	S3	S2	S1	S1
S1	S3	S2	S1	S1
S2	S1	S1	S1	BAD
S3	BAD	BAD	BAD	BAD
SG	BAD	BAD	BAD	BAD
SB / BAD	BAD	BAD	BAD	BAD

## Αριθμητικά Κυριολεκτικά/Numerical Literals

	0	1-9	X x	1-7	E e	.	-	A-F	* / OTHER
SZ	S1	S2	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S1	S7	BAD	S6	S7	S4	S3	BAD	BAD	BAD
S2	S2	S2	BAD	S2	S4	S3	BAD	BAD	BAD
S3	S10	S10	BAD	S10	BAD	BAD	BAD	BAD	BAD
S4	SG	S8	BAD	S8	BAD	BAD	S5	BAD	BAD
S5	BAD	S8	BAD	S8	BAD	BAD	BAD	BAD	BAD
S6	S9	S9	BAD	S9	BAD	BAD	BAD	S9	BAD
S7	S7	BAD	BAD	S7	BAD	BAD	BAD	BAD	BAD
S8	S8	S8	BAD	S8	BAD	BAD	BAD	BAD	BAD
S9	S9	S9	BAD	S9	BAD	BAD	BAD	S9	BAD
S10	S10	S10	BAD	S10	BAD	BAD	BAD	BAD	BAD

## Τελεστές/Operators

	&	-	+	%		\	*	/	=	!	<	>	* / OTHER
SZ	S5	S4	S3	SG	S2	S1	S1	S1	S1	S1	S1	S1	BAD
S1	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	SG	BAD	BAD	BAD	BAD
S2	BAD	BAD	BAD	BAD	SG	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
S3	BAD	BAD	SG	BAD	BAD	BAD	BAD	BAD	SG	BAD	BAD	BAD	BAD
S4	BAD	SG	BAD	BAD	BAD	BAD	BAD	BAD	SG	BAD	BAD	BAD	BAD
S5	SG	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
SG	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD
SB / BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD	BAD

## Σχόλια/Comments

	/	*	\*
SZ	S0	SB	BAD
S0	S1	SB	S2
S1	BAD	S1	BAD
S2	BAD	S2	S3
S3	SG	SB	BAD
SG	BAD	BAD	BAD
SB / BAD	BAD	BAD	BAD

### 3. Ενιαίο Πεπερασμένο Αυτόματο

Σε αυτό το κεφάλαιο, ακολουθεί η ανάλυση και κωδικοποίηση του ενιαίου πεπερασμένου αυτόματου, το σχήμα/διάγραμμα και ο γενικός πίνακας μετάβασης. Για την επίτευξη του κώδικα του ενιαίου ενσωματώνουμε όλους τους κώδικες μετονομάζοντας τις καταστάσεις S0, S1 κτλ., που είχαμε προηγουμένως σε κάθε επιμέρους αυτόματο, και συμπεριλαμβάνουμε σε ποιο επιμέρους πεπερασμένο ανήκει (π.χ το S0\_IDENT είναι η S0 κατάσταση του πεπερασμένου αυτόματου για τα αναγνωριστικά).

Ως αρχική κατάσταση του ενιαίου ορίζουμε την SZ που ανάλογα με τι αρχίζει μεταβιβάζεται στο κάθε επιμέρους αυτόματο που αναλύσαμε και πριν. Για παράδειγμα, αν σαν πρώτη μετάβαση δοθεί το " (quotation mark) ξέρουμε σίγουρα ότι η λεκτική μονάδα που ξεκινάει είναι το string. Παρομοίως, αν σαν πρώτη μετάβαση δοθούν αριθμητικά ψηφία γνωρίζουμε σίγουρα ότι η λεκτική μονάδα που ξεκινάει είναι αριθμητικό κυριολεκτικό (αφού στα αναγνωριστικά δεν γίνεται να αρχίσει με αριθμητικό ψηφίο).

Τώρα η μόνη σημαντική διαφορά στο ενιαίο, είναι στην περίπτωση που δοθεί ο χαρακτήρας / ως αρχική μετάβαση γιατί έχουμε δύο περιπτώσεις για το πρότυπο που ακολουθεί, είτε αρχίζει σχόλιο είτε είναι ο τελεστής /. Οπότε για αυτό μεταβαίνουμε στην μετάβαση S0\_COMM\_OP που συνδυάζει την περίπτωση για τον τελεστή να είναι μόνος του ή να ακολουθεί ίσων (=) και την περίπτωση να ακολουθεί single line ή multi line comment. Επιπλέον, για το ενιαίο αυτόματο αποδεχόμαστε στην αρχική κατάσταση SZ και όποιο white space character (space, tab, new line) ως πρότυπο αναγνώρισης.

#### 3.1 Κώδικας FSM

```
// SZ is the starting state that checks for the first input and
// transitions to the appropriate state accordingly.
// So we basically have this starting state, and depending on the starting value
// it transitions to the appropriate state that may have some code combined.
START=SZ
SZ:      A-Z a-z _      -> S0_IDENT // transition to identifier code
        /              -> S0_COMM_OP // transition to comments as well as the division operator
        "              -> S0_STR   // transition to code regarding strings
        0              -> S1_NUM   // transition to one of two states regarding integers
        1-9            -> S2_NUM   // transition to second state regarding integers
        \* \= ! < >    -> S1_OP   // transition to state regarding operators
        |              -> S2_OP   // transition to state regarding operators (OR Operation)
        +              -> S3_OP   // transition to state regarding operators (plus)
        -              -> S4_OP   // transition to state regarding operators (minus)
        &              -> S5_OP   // transition to state regarding operator (&)
        %              -> GOOD    // Modulus Operation (Accepted State)
        \n \s \t       -> SZ      // Whitespaces just loop to SZ till another character is read
        *              -> BAD     // anything else is not accepted

// Identifiers
S0_IDENT: a-z A-Z _ 0-9 -> S0_IDENT // covers the case for numerical numbers
        \n          -> GOOD    // if it just starts with A-Z a-z or underscore (_) then it's accepted
        *           -> BAD     // anything else is not valid

// For comments and operator /
S0_COMM_OP: /          -> S1_COMM // single line comment (//)
        \*            -> S2_COMM // multiple line comment start (/*)
        \= \n         -> GOOD    // /= or / are accepted
        *             -> BAD     // anything else is invalid

// State for single line comment
S1_COMM: *             -> S1_COMM // any text inside the single line comment
        \n            -> GOOD    // covers the case of an empty comment

// S2_COMM and S3_COMM states are about multi-line comments
S2_COMM: *             -> S2_COMM // any text inside the comment
        \*            -> S3_COMM // if asterisk first indicator for multi-line comment end

S3_COMM: /             -> GOOD    // we have the combination */ which is a multi-line comment end
        *             -> BAD     // anything else is not accepted

// For String literals
S0_STR: *              -> S1_STR  // if any text inside the string go to S1_STR
        \\            -> S2_STR  // check if there is a backslash in the string
        "             -> GOOD    // string is closed

S1_STR: \\             -> S2_STR  // escape sequences for \, ", \n
        "             -> GOOD    // string is closed
        *             -> S1_STR  // if you get any text, keep looping on S1
```

```

S2_STR:  \\ " n          -> S1_STR // only valid escape sequences
          *              -> BAD    // anything else is not valid

// For numerical literals (regarding zero)
S1_NUM:  .              -> S3_NUM // beginning of fractional part
          e E           -> S4_NUM // base of the exponential number including e/E
          X x           -> S6_NUM // with the x/X we identify start (0x or 0X) of the hexadecimal number
          0-7           -> S7_NUM // octal number (it has already read a 0)
          \n            -> GOOD   // this covers the case of number 0 alone
          *              -> BAD    // anything else invalid

// Integer
S2_NUM:  0-9            -> S2_NUM // the machine has already read a digit 1-9, so this case covers multi
digit numbers
          .             -> S3_NUM // provided it has read a digit 1-9, check if there is a decimal point
          e E           -> S4_NUM // given a digit 1-9 (base), check for character e/E
          \n            -> GOOD   // single digit integer
          *              -> BAD    // anything else invalid

// Float
S3_NUM:  0-9            -> S10_NUM // numbers with a single digit on the fractional part are accepted
          *              -> BAD    // anything else not accepted

S10_NUM: 0-9            -> S10_NUM // float numbers with fractional part are accepted
          e E           -> S4_NUM // exponential float number (e.g 1.5e10)
          \n            -> GOOD   // floating point number with at least one digit on fractional part
          *              -> BAD    // anything else invalid

// Exponential
S4_NUM:  1-9            -> S8_NUM // this is the power of the number that is expressed after the e/E
          -             -> S5_NUM // in this case we have a negative power
          0              -> GOOD   // this allows 0e0 to be valid
          *              -> BAD    // anything else not accepted

S5_NUM:  1-9            -> S8_NUM // any digit is accepted (e.g 5e-5)
          *              -> BAD    // anything else not valid

S8_NUM:  0-9            -> S8_NUM // in this case we have an integer as the power (multiple digits)
          \n            -> GOOD   // single digit integer power
          *              -> BAD    // anything else invalid

// Hexadecimal
S6_NUM:  A-F 0-9        -> S9_NUM // right part of hexadecimal number (After x/X. Also single digit)
          *              -> BAD    // anything else not valid

S9_NUM:  A-F 0-9        -> S9_NUM // covers multiple digits on the right part of hexadecimal number
          \n            -> GOOD   // hex numbers with at least one digit on the right part are accepted
          *              -> BAD    // anything else invalid

// Octal
S7_NUM:  0-7            -> S7_NUM // check for multiple octal digits
          \n            -> GOOD   // Numbers with multi octal digits are accepted in this case
          *              -> BAD    // anything else invalid

// For operators
S1_OP:  \n \=          -> GOOD   // here we check if it's just a single operator, or followed by an equal
sign (=)
          *              -> BAD    // anything else is not accepted

S2_OP:  |              -> GOOD   // check if there is another |. If there is then it's an OR operation
(|)
          *              -> BAD    // if it's only one | and anything else, then it's not valid

S3_OP:  \n \+ \=        -> GOOD   // the operations can be addition (+), increment (++) or +=
          *              -> BAD    // if it's anything else after the plus sign then it's not accepted

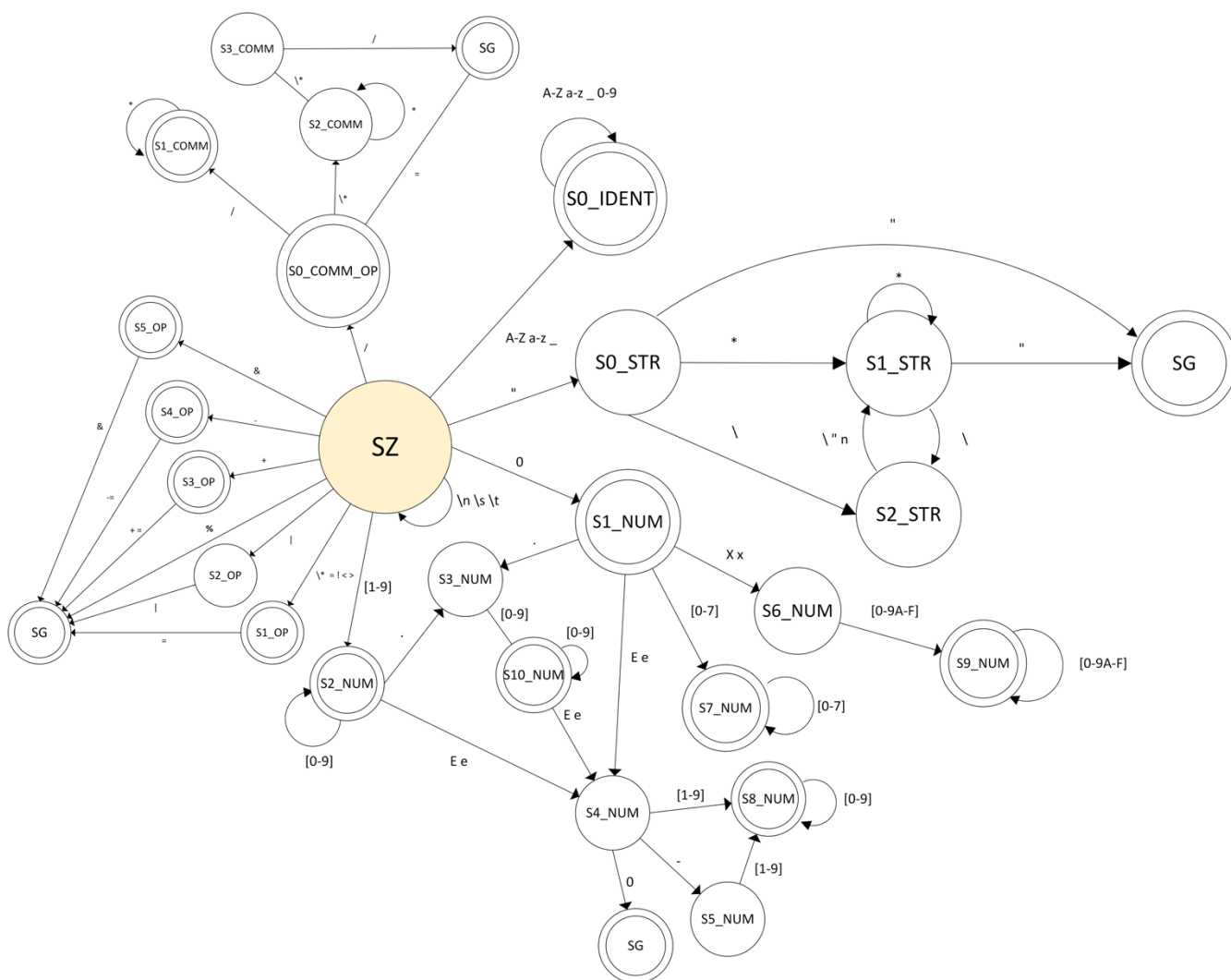
S4_OP:  \n \- \=        -> GOOD   // if it's only one minus (-) it's the difference, -- (decrement) and -=
          *              -> BAD    // if it's just one minus and anything else after then it's not valid.

S5_OP:  \n &            -> GOOD   // if there is another & sign then it's a logical AND operation (&&)
          *              -> BAD    // if it's only one & sign followed by anything else it's not valid

GOOD(OK): \n -> GOOD

```

Η κατάσταση SG είναι η κατάσταση GOOD και γενικά έχει χρησιμοποιηθεί για κατάσταση αποδοχής για κάθε επιμέρους αυτόματο που ενσωματώθηκε. Η κατάσταση BAD δεν έχει συμπεριληφθεί γιατί αυτό θα καθιστούσε το σχήμα αρκετά πολύπλοκο. Εννοείται ότι όποια άλλη μετάβαση δοθεί που δεν έχει γραφτεί μεταβαίνει στο BAD.




Ακολουθεί ο γενικός πίνακας μετάβασης για το ενιαίο πεπερασμένο αυτόματο. Με το (\*) εννοούμε οποιαδήποτε άλλη μετάβαση δεν έχει αναφερθεί από τις υπάρχουσες μεταβάσεις.

[illegible]



## Αριθμητικά Κυριολεκτικά/Numerical Literals

Regular Expression

JavaScript 

`/([0-9A-Fa-f]+)|([0-7]+)|(([-9]\d*|0)(\.\d+)?([eE][-]?([1-9][0-9]*|0))?)/g`

29 matches

Test String

```
0x91AFp1 0000121378 12 132 1.3 3.14 31.1 313.4 3.14e10 3.14e-10 0e0 3.e5 3.1E15 3.1E0
09 0E0 0.13 0XABC987 x01 .0129 000.000 0.01
```

## Τελεστές/Operators

Regular Expression

`/([\+\-\*\//\=\!\<\>=)|([\*\//\=\!\<\>%]|&{1,2}|\-{1,2}|\+{1,2}|(\| \|))/g`

Test String

```
+ = - = * = / = ! = < = > = * / = ! < > % & + - && ++ -- ||
&+ +& -& &- ++ -- +* *- */ ** // !! %% | << >> <| >| !* *| /| +| -| !- !+
```

## Σχόλια/Comments

Regular Expression

`/\/*(\.|\n)*?*\//\//\/*/g`

Test String

```
// ahaffbajgbb aj
NFDJBG*/
afjafjbadgjba
/* bdsjbgjsbgsjdbg
dababdzejgbajbg */
daaudg//
/HGJSDBGBSD
/* NSGNSGNSDNG
/* jad */
/* adb /
KADHDGK/
```



## 4.2 Κώδικες FSM

Για ευκολία, σε όλους τους κώδικες η αρίθμηση των δοκιμαστικών εκτελέσεων, ξεκινάει από την πρώτη σειρά από αριστερά προς τα δεξιά.

### Αναγνωριστικά/Identifiers

```
./fsm -trace 01_identifiers.fsm
Counter
sz C -> s0
s0 o -> s0
s0 u -> s0
s0 n -> s0
s0 t -> s0
s0 e -> s0
s0 r -> s0
s0 \n -> good
YES
```

```
./fsm -trace 01_identifiers.fsm
index
sz i -> s0
s0 n -> s0
s0 d -> s0
s0 e -> s0
s0 x -> s0
s0 \n -> good
YES
```

```
./fsm -trace 01_identifiers.fsm
_object
sz _ -> s0
s0 o -> s0
s0 b -> s0
s0 j -> s0
s0 e -> s0
s0 c -> s0
s0 t -> s0
s0 \n -> good
YES
```

```
./fsm -trace 01_identifiers.fsm
item1
sz i -> s0
s0 t -> s0
s0 e -> s0
s0 m -> s0
s0 l -> s0
s0 \n -> good
YES
```

```
./fsm -trace 01_identifiers.fsm
litem
sz l -> bad
fsm: in 01_identifiers.fsm,
state 'bad' input l not
accepted
```

```
./fsm -trace 01_identifiers.fsm
l_item
sz l -> bad
fsm: in 01_identifiers.fsm,
state 'bad' input _ not
accepted
```

Όπως βλέπουμε ο κώδικας που υλοποιεί το αυτόματο για τα αναγνωριστικά τρέχει και παράγει αποτελέσματα. Στην πρώτη εκτέλεση, δηλώνουμε αναγνωριστικό με κεφαλαίο πρώτο χαρακτήρα και βλέπουμε ότι οδηγείται από την αρχική κατάσταση στο s0, κάνει loop στον εαυτό του για όσα γράμματα ακολουθούν μέχρι να οδηγηθεί στην κατάσταση αποδοχής. Στην δεύτερη εκτέλεση, παρομοίως βλέπουμε ότι και για μικρό πρώτο χαρακτήρα λειτουργεί ακριβώς με τον ίδιο τρόπο όπως πριν. Στην τρίτη εκτέλεση, βλέπουμε ότι και για \_ (κάτω παύλα) σαν πρώτο χαρακτήρα πάλι λειτουργεί και αναγνωρίζεται. Στην τέταρτη εκτέλεση, βλέπουμε ότι μπορούμε να βάλουμε αριθμό μετά τον πρώτο χαρακτήρα και μεταβαίνουμε στην κατάσταση αποδοχής. Στην πέμπτη εκτέλεση, βλέπουμε ότι αν βάλουμε αριθμητικό ψηφίο σαν πρώτο χαρακτήρα και μετά λατινικό χαρακτήρα στο αναγνωριστικό μεταβαίνουμε κατευθείαν στην κατάσταση BAD. Στην έκτη εκτέλεση, βλέπουμε ομοίως αν βάλουμε αριθμητικό ψηφίο και έπειτα \_ (κάτω παύλα) πάλι μεταβαίνουμε κατευθείαν στην BAD κατάσταση.

### Λεκτικά Κυριολεκτικά/String Literals

```
./fsm -trace 02_string_literals.fsm
""
sz " -> s0
s0 " -> good
good \n -> good
YES
```

```
./fsm -trace 02_string_literals.fsm
"Test"
sz " -> s0
s0 T -> s1
s1 e -> s1
s1 s -> s1
s1 t -> s1
s1 " -> good
good \n -> good
YES
```

```
./fsm -trace 02_string_literals.fsm
test text"
sz t -> bad

fsm: in 02_string_literals.fsm,
state 'bad' input e not accepted
```

```
./fsm -trace 02_string_literals.fsm
"\test"
sz " -> s0
s0 \ -> s2
s2 t -> bad
fsm: in 02_string_literals.fsm,
state 'bad' input e not accepted
```

```
./fsm -trace 02_string_literals.fsm
"\Test\\"\n"
sz " -> s0
s0 \ -> s2
s2 " -> s1
s1 T -> s1
s1 e -> s1
s1 s -> s1
s1 t -> s1
s1 \ -> s2
s2 " -> s1
s1 \ -> s2
s2 n -> s1
s1 " -> good
good \n -> good
YES
```

```
./fsm -trace 02_string_literals.fsm
"test text
sz " -> s0
s0 t -> s1
s1 e -> s1
s1 s -> s1
s1 t -> s1
s1 \s -> s1
s1 t -> s1
s1 e -> s1
s1 x -> s1
s1 t -> s1
s1 \n -> s1
s1 EOF -> s1
NO
```

Όπως βλέπουμε ο κώδικας που υλοποιεί το αυτόματο για τα λεκτικά κυριολεκτικά τρέχει και παράγει αποτελέσματα. Στην πρώτη εκτέλεση, δηλώνουμε το κενό string και βλέπουμε ότι δέχεται το πρώτο “ μεταβαίνοντας στην κατάσταση s0 και μετά με το τερματικό “ μεταβαίνει στην κατάσταση αποδοχής. Στην δεύτερη εκτέλεση, βλέπουμε ότι με “ μεταβαίνει στη s0, με τον λατινικό χαρακτήρα μεταβαίνει στο s1 και κάνει loop στον εαυτό του μέχρι να δώσουμε το τερματικό “ και πάει στην κατάσταση αποδοχής. Στην τρίτη εκτέλεση, βλέπουμε ότι επειδή δώσαμε ως αρχική μετάβαση έναν λατινικό χαρακτήρα και όχι “ μεταβήκαμε κατευθείαν στην κατάσταση BAD. Στην τέταρτη εκτέλεση, βλέπουμε ότι προχωράει με “ στο s0, μεταβαίνει με \ στο s2 για να δει άμα υπάρχει κάποιο escape sequence αλλά τερματίζεται το string με “ και για αυτό μεταβαίνουμε στο bad. Στην πέμπτη εκτέλεση, βλέπουμε ότι λειτουργούν τα escape sequences δηλαδή κάθε φορά που λαμβάνει \ μεταβαίνει στο s2 και λαμβάνει το « ή το η και ξανά επιστρέφει στο s1 όπου λαμβάνει και λατινικούς χαρακτήρες και τελικώς οδηγείται στην κατάσταση αποδοχής. Στην έκτη εκτέλεση, ξεκινάμε με αρχικό “ μεταβαίνοντας στην κατάσταση s0 και μετά με λατινικό χαρακτήρα στο s1 αλλά επειδή δεν δίνουμε τερματικό “ κάνει loop στον εαυτό του μέχρι να μεταβεί στην κατάσταση BAD.

#### Αριθμητικά Κυριολεκτικά/Numerical Literals

```
./fsm -trace 03_numbers.fsm
3.14e-10
s0 3 -> s2
s2 . -> s3
s3 1 -> s10
s10 4 -> s10
s10 e -> s4
s4 - -> s5
s5 1 -> s8
s8 0 -> s8
s8 \n -> good
YES
```

```
./fsm -trace 03_numbers.fsm
213748
s0 2 -> s2
s2 1 -> s2
s2 3 -> s2
s2 7 -> s2
s2 4 -> s2
s2 8 -> s2
s2 \n -> good
YES
```

```
./fsm -trace 03_numbers.fsm
0XFF42
s0 0 -> s1
s1 X -> s6
s6 F -> s9
s9 F -> s9
s9 4 -> s9
s9 2 -> s9
s9 \n -> good
YES
```

```
./fsm -trace 03_numbers.fsm
063
s0 0 -> s1
s1 6 -> s7
s7 3 -> s7
s7 \n -> good
YES
```

```
./fsm -trace 03_numbers.fsm
3.14
s0 3 -> s2
s2 . -> s3
s3 1 -> s10
s10 4 -> s10
s10 \n -> good
YES
```

```
./fsm -trace 03_numbers.fsm
1
s0 1 -> s2
s2 \n -> good
YES
```

```
./fsm -trace 03_numbers.fsm
X01
s0 X -> bad
fsm: in 03_numbers.fsm,
state 'bad' input 0 not
accepted
```

```
./fsm -trace 03_numbers.fsm
09
s0 0 -> s1
s1 9 -> bad
fsm: in 03_numbers.fsm,
state 'bad' input \n not
accepted
```

```
./fsm -trace 03_numbers.fsm
.0129
s0 . -> bad
fsm: in 03_numbers.fsm,
state 'bad' input 0 not
accepted
```

Όπως βλέπουμε ο κώδικας που υλοποιεί το αυτόματο για τα αριθμητικά κυριολεκτικά τρέχει και παράγει αποτελέσματα. Στην πρώτη εκτέλεση, μεταβαίνει στην καταστάσεις που αναγνωρίζουν ότι είναι δεκαδικός αριθμός και έπειτα στην κατάσταση S10 δέχεται την ύψωση σε δύναμη και μεταβαίνει στην κατάσταση S4 όπου εκεί δέχεται το αρνητικό πρόσημο και μεταβαίνει στην S5 όπου δέχεται έναν θετικό ακέραιο. Από την κατάσταση S5 πηγαίνει στην S8 και διαβάζει ακέραιο μέχρις ότου να δεχθεί \n και να μεταβεί στην κατάσταση αποδοχής. Στην δεύτερη εκτέλεση, δηλώνουμε τον αριθμό 213748 και βλέπουμε ότι δέχεται τον αριθμό 2 μεταβαίνοντας στην κατάσταση S2 και μετά επαναληπτικά μέχρι να δεχθεί \n δέχεται τους αριθμούς το S2. Μόλις δεχθεί \n μεταβαίνει στην κατάσταση αποδοχής. Στην τρίτη εκτέλεση, δηλώνουμε τον δεκαεξαδικό αριθμό 0XFF42 και βλέπουμε ότι δέχεται τον αριθμό 0 μεταβαίνοντας στην κατάσταση S1 , μετά δέχεται το X μεταβαίνοντας στο S6 ,στο S6 δέχεται το F και μεταβαίνει στο S9 δέχοντας επαναληπτικά αριθμούς από 0-9 και λατινικούς χαρακτήρες από A-F μέχρις ότου να δεχθεί \n και να μεταβεί στην κατάσταση αποδοχής.

Στην τέταρτη εκτέλεση, δηλώνουμε τον αριθμό 0 και μεταβαίνει στην S1 όπου εκεί δέχεται αριθμούς 0-7 μεταβαίνοντας στην κατάσταση S7 όπου δέχεται τους οκταδικούς αριθμούς μέχρις ότου δεχθεί το \n μεταβαίνοντας έτσι στην κατάσταση αποδοχής. Στην πέμπτη εκτέλεση, δηλώνουμε τον αριθμό 3 και βλέπουμε ότι δέχεται τον αριθμό μεταβαίνοντας στην S2 όπου με την τελεία (.) καταλαβαίνει ότι είναι δεκαδικός αριθμός μεταβαίνοντας στην κατάσταση S3 , όπου εκεί δέχεται ακέραιο πηγαίνοντας στην S10 και εκεί με \n μεταβαίνει

στην κατάσταση αποδοχής. Στην έκτη εκτέλεση, δηλώνουμε τον αριθμό 1 και βλέπουμε ότι δέχεται τον αριθμό μεταβαίνοντας στην κατάσταση S2 και μετά με το \n μεταβαίνει στην κατάσταση αποδοχής. Στην έβδομη εκτέλεση βλέπουμε ότι δέχεται το X και όχι 0 ή κάποιο θετικό ακέραιο μεταβήκαμε απευθείας στην κατάσταση BAD. Στην όγδοη εκτέλεση, δηλώνουμε τον αριθμό 0 και βλέπουμε ότι δέχεται το 0 μεταβαίνοντας στην κατάσταση S1 όπου εκεί δέχεται ακεραίους από 0 έως 7 οπότε όταν δέχθηκε 9 μεταβήκαμε απευθείας στην κατάσταση BAD. Στην ένατη εκτέλεση όπως και στην έκτη εκτέλεση η κατάσταση S0 δέχεται 0 ή 1-9 και όχι τελεία (.) οπότε αφού δεχθήκαμε την τελεία (.) μεταβήκαμε απευθείας στην κατάσταση BAD.

### Τελεστές/Operators

```
./fsm -trace 04_operators.fsm
*
sz * -> s1
s1 \n -> good
YES
```

```
./fsm -trace 04_operators.fsm
+=
sz + -> s3
s3 = -> good
good \n -> good
YES
```

```
./fsm -trace 04_operators.fsm
++
sz + -> s3
s3 + -> good
good \n -> good
YES
```

```
./fsm -trace 04_operators.fsm
!
sz ! -> s1
s1 \n -> good
YES
```

```
./fsm -trace 04_operators.fsm
&
sz & -> s5
s5 \n -> good
YES
```

```
./fsm -trace 04_operators.fsm
&&
sz & -> s5
s5 & -> good
good \n -> good
YES
```

```
./fsm -trace 04_operators.fsm
-+
sz - -> s4
s4 + -> bad
fsm: in 04_operators.fsm,
state 'bad' input \n not
accepted
```

```
./fsm -trace 04_operators.fsm
|
sz | -> s2
s2 \n -> bad
NO
```

```
./fsm -trace 04_operators.fsm
>>
sz > -> s1
s1 > -> bad
fsm: in 04_operators.fsm,
state 'bad' input \n not
accepted
```

Όπως βλέπουμε ο κώδικας που υλοποιεί το αυτόματο για τους τελεστές τρέχει και παράγει αποτελέσματα. Στην πρώτη εκτέλεση, βλέπουμε ότι το \* αναγνωρίζεται και περνάει στην κατάσταση s1 και μετά στην κατάσταση αποδοχής. Στην δεύτερη εκτέλεση, το + περνάει στην κατάσταση s3 που δέχεται τις περιπτώσεις για τον τελεστή + οπότε λαμβάνοντας το = πάει κατευθείαν στην κατάσταση αποδοχής. Στην τρίτη εκτέλεση, παρομοίως το + περνάει στην κατάσταση s3 και με το + πάλι πάει στην κατάσταση αποδοχής. Στην τέταρτη εκτέλεση, το ! περνάει στην κατάσταση s1 που δέχεται τις περιπτώσεις τελεστών που μπορούν να εμφανιστούν μόνοι τους ή με το ίσων (=) και μετά πάει κατευθείαν στην κατάσταση αποδοχής. Στην πέμπτη εκτέλεση, το & περνάει στην κατάσταση s5 που περιγράφει ότι το & μπορεί να εμφανιστεί από μία έως δύο φορές και μετά πάει στην κατάσταση αποδοχής. Στην έκτη εκτέλεση, ικανοποιεί την περίπτωση του & από την s5 να εμφανιστεί δύο φορές και να μεταβεί στην κατάσταση αποδοχής. Στην έβδομη εκτέλεση, βλέπουμε ότι μεταβαίνοντας στην κατάσταση s4 που περιγράφει τις περιπτώσεις για το -, δεν αποτελεί έγκυρη μετάβαση το - οπότε καταλήγει στην κατάσταση BAD. Στην όγδοη εκτέλεση, ο τελεστής | πρέπει πάντα να εμφανίζεται σαν ζευγάρι με τον εαυτό του οπότε για αυτό μεταβαίνει κατευθείαν στην κατάσταση BAD. Στην ένατη εκτέλεση, με τον τελεστή > μεταβαίνουμε στην κατάσταση s1 που μπορεί να δεχτεί μόνο = οπότε για αυτό με την επανάληψη της μετάβασης > μεταβαίνουμε στην κατάσταση BAD.

### Σχόλια/Comments

```
./fsm -trace 06_comments.fsm
// test text
sz / -> s0
s0 / -> s1
s1 \s -> s1
s1 t -> s1
s1 e -> s1
s1 s -> s1
s1 t -> s1
s1 \s -> s1
s1 t -> s1
s1 e -> s1
s1 x -> s1
s1 t -> s1
s1 \n -> good
YES
```

```
./fsm -trace 06_comments.fsm
/* test
sz / -> s0
s0 * -> s2
s2 \s -> s2
s2 t -> s2
s2 e -> s2
s2 s -> s2
s2 t -> s2
s2 \n -> s2
text */
s2 t -> s2
s2 e -> s2
s2 x -> s2
s2 t -> s2
s2 \s -> s2
s2 * -> s3
s3 / -> good
good \n -> good
YES
```

```
./fsm -trace 06_comments.fsm
/* test text
sz / -> s0
s0 * -> s2
s2 \s -> s2
s2 t -> s2
s2 e -> s2
s2 s -> s2
s2 t -> s2
s2 \s -> s2
s2 t -> s2
s2 e -> s2
s2 x -> s2
s2 t -> s2
s2 \n -> s2
s2 EOF -> s2
NO
```

```
./fsm -trace 06_comments.fsm
/ test text
sz / -> s0
s0 \s -> bad
fsm: in 06_comments.fsm,
state 'bad' input t not
accepted
```

```
./fsm -trace 06_comments.fsm
test text */
sz t -> bad
fsm: in 06_comments.fsm,
state 'bad' input e not
accepted
```

```
./fsm -trace 06_comments.fsm
comment //
sz c -> bad
fsm: in 06_comments.fsm,
state 'bad' input o not
accepted
```

Όπως βλέπουμε ο κώδικας που υλοποιεί το αυτόματο για τα σχόλια τρέχει και παράγει αποτελέσματα. Στην πρώτη εκτέλεση, δοκιμάζουμε ένα single line comment και παρατηρούμε ότι ξεκινάει αναγνωρίζοντας τα δύο forward slashes (//), το κείμενο εντός του comment και τέλος τον χαρακτήρα της νέας γραμμής (\n). Στην δεύτερη εκτέλεση, έχουμε εισάγει στο FSM ένα σχόλιο πολλαπλών γραμμών. Αρχίζει αναγνωρίζοντας την έναρξη του σχολίου (/\*) καθώς και το κείμενο εντός του σχολίου, ολοκληρώνοντας την πρώτη γραμμή. Παρόμοια λειτουργεί στο δεύτερο κομμάτι αναγνωρίζοντας το κείμενο εντός του σχολίου και κλείνοντας το σχόλιο με το σύμβολο \*/. Στην τρίτη εκτέλεση, ανοίγουμε ένα σχόλιο πολλαπλών γραμμών χωρίς να το κλείσουμε, καταλήγοντας σε μη επιθυμητή κατάσταση. Στην τέταρτη εκτέλεση, δοκιμάζουμε το μονό forward slash (/) ως είσοδο στο FSM, κάτι το οποίο δεν θεωρείται σχόλιο, συνεπώς η είσοδος δεν είναι αποδεκτή. Η πέμπτη εκτέλεση, είναι ιδιαίτερα παρόμοια με την δεύτερη εκτέλεση, απλώς το σχόλιο εδώ κλείνει, χωρίς να έχει ανοίξει προηγουμένως. Στην έκτη και τελευταία εκτέλεση, εισάγουμε πρώτα το κείμενο και έπειτα δηλώνουμε την αρχή σχολίου, πράγμα το οποίο δεν είναι επιτρεπτό καθώς δεν αποτελεί σχόλιο.

### Ενιαίο Πεπερασμένο Αυτόματο

```
./fsm -trace 05_uniform.fsm
_var_
sz _ -> s0_ident
s0_ident v -> s0_ident
s0_ident a -> s0_ident
s0_ident r -> s0_ident
s0_ident _ -> s0_ident
s0_ident \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
// test
sz / -> s0_comm_op
s0_comm_op / -> s1_comm
s1_comm \s -> s1_comm
s1_comm t -> s1_comm
s1_comm e -> s1_comm
s1_comm s -> s1_comm
s1_comm t -> s1_comm
s1_comm \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
/
sz / -> s0_comm_op
s0_comm_op \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
0
sz 0 -> s1_num
s1_num \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
5
sz 5 -> s2_num
s2_num \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
!=
sz ! -> s1_op
s1_op = -> good
good \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
++
sz + -> s3_op
s3_op + -> good
good \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
-
sz - -> s4_op
s4_op \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
&
sz & -> s5_op
s5_op \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
%
sz % -> good
good \n -> good
YES
```

```
./fsm -trace 05_uniform.fsm
|
sz | -> s2_op
s2_op \n -> bad
NO
```

```
./fsm -trace 05_uniform.fsm
/*
sz / -> s0_comm_op
s0_comm_op * -> s2_comm
s2_comm / -> s2_comm
s2_comm \n -> s2_comm
s2_comm EOF -> s2_comm
NO
```

```
./fsm -trace 05_uniform.fsm
2e--3
sz 2 -> s2_num
s2_num e -> s4_num
s4_num - -> s5_num
s5_num - -> bad
fsm: in 05_uniform.fsm, state
'bad' input 3 not accepted
```

```
./fsm -trace 05_uniform.fsm
var@
sz v -> s0_ident
s0_ident a -> s0_ident
s0_ident r -> s0_ident
s0_ident @ -> bad
fsm: in 05_uniform.fsm, state
'bad' input \n not accepted
```

```
./fsm -trace 05_uniform.fsm
*==
sz * -> s1_op
s1_op = -> good
fsm: in 05_uniform.fsm, state
'good' input = not accepted
```

Όπως βλέπουμε ο κώδικας που υλοποιεί το ενιαίο αυτόματο τρέχει και παράγει αποτελέσματα. Γενικά, ελέγχουμε άμα από την αρχική κατάσταση SZ μεταβιβαζόμαστε «σωστά» στα επιμέρους αυτόματα αφού μετά ισχύουν οι σχολιασμοί που έγιναν για το κάθε επιμέρους αυτόματο. Επομένως, θα αναλύσουμε γενικά άμα αναγνωρίζονται όλες οι λεκτικές μονάδες (κάποιο παράδειγμα για τον καθένα) και κάποιες περιπτώσεις που η είσοδος δεν αποτελεί κάποια λεκτική μονάδα.

Στην πρώτη εκτέλεση, ελέγχουμε αν απο την αρχική κατάσταση SZ, οδηγούμαστε σωστά στο κομμάτι του κώδικα που εκτελεί το πεπερασμένο αυτόματο για τα αναγνωριστικά. Βλέπουμε ότι όντως έχοντας ως αρχική μετάβαση την \_ (κάτω παύλα) μεταβιβαζόμαστε σωστά στο S0\_IDENT που είναι η πρώτη κατάσταση που χειρίζεται τα αναγνωριστικά, οπότε έπειτα λειτουργεί το fsm αναλόγως. Στην δεύτερη εκτέλεση, ελέγχουμε αν απο την SZ οδηγούμαστε σωστά στο κομμάτι του κώδικα που χειρίζεται τα σχόλια και τον τελεστή / (διαίρεση). Βλέπουμε ότι όντως μεταβαίνουμε σωστά στο S0\_COMM\_OP που είναι η πρώτη κατάσταση για τα σχόλια και τον τελεστή οδηγείται σωστά στην κατάσταση S1\_COMM που διαχειρίζεται το μονό σχόλιο. Στην τρίτη εκτέλεση, βλέπουμε ότι πάλι οδηγούμαστε στην S0\_COMM\_OP αλλά δεν μεταβαίνουμε στην κατάσταση που διαχειρίζεται το μονό σχόλιο αλλά πάμε στην κατάσταση αποδοχής (αφού δεχτήκαμε τον τελεστή διαίρεσης).

Στην τέταρτη και στην πέμπτη εκτέλεση, ελέγχουμε αν από την αρχική κατάσταση SZ οδηγούμαστε σωστά στο κομμάτι του κώδικα που εκτελεί το πεπερασμένο αυτόματο για τα αριθμητικά κυριολεκτικά. Ειδικότερα, στην τέταρτη εκτέλεση βλέπουμε ότι με το 0 μεταβαίνουμε σωστά στην κατάσταση S1\_NUM που διαχειρίζεται, όπως στο επιμέρους, τις περιπτώσεις του 0 οπότε μεταβαίνουμε σωστά στην κατάσταση αποδοχής. Παρομοίως, στην πέμπτη εκτέλεση βλέπουμε ότι με το 5 μεταβαίνουμε σωστά στην κατάσταση S2\_NUM που διαχειρίζεται, όπως στο επιμέρους, τις περιπτώσεις των αριθμών 1-9 οπότε μεταβαίνουμε σωστά στην κατάσταση αποδοχής.

Στις εκτελέσεις έξι έως έντεκα ελέγχουμε, αν από την αρχική κατάσταση SZ οδηγούμαστε σωστά στο κομμάτι του κώδικα που εκτελεί το πεπερασμένο αυτόματο για τους τελεστές. Ειδικότερα, στην έκτη εκτέλεση βλέπουμε ότι με το ! μεταβαίνουμε σωστά στην κατάσταση S1\_OP που διαχειρίζεται, όπως στο επιμέρους, τις περιπτώσεις των τελεστών που μπορούν να εμφανιστούν μόνοι τους ή με συνδυασμό με το ίσων (=) οπότε για αυτό μεταβαίνουμε σωστά στην κατάσταση αποδοχής. Στην έβδομη εκτέλεση, βλέπουμε ότι με το + μεταβαίνουμε σωστά στην S3\_OP που διαχειρίζεται τις περιπτώσεις του τελεστή πρόσθεσης, όπως στο επιμέρους, οπότε για αυτό σωστά πάμε στην κατάσταση αποδοχής. Στην όγδοη εκτέλεση, από την αρχική κατάσταση μεταβαίνουμε σωστά στην S4\_OP που διαχειρίζεται τις περιπτώσεις του -, όπως στο επιμέρους και για αυτό πάμε κανονικά στην κατάσταση αποδοχής. Στην ένατη περίπτωση, από την SZ πάμε σωστά στην S5\_OP που διαχειρίζεται τις περιπτώσεις του τελεστή &, όπως στο επιμέρους και για αυτό μεταβαίνουμε σωστά στην κατάσταση αποδοχής. Στην δέκατη εκτέλεση, με τον τελεστή % πάμε κατευθείαν στην κατάσταση αποδοχής αφού δεν αποτελεί κάποιο ζευγάρι με κάποιον άλλον τελεστή, όπως ξέρουμε και απο τον επιμέρους κώδικα. Στην ενδέκατη εκτέλεση, από την SZ πάμε σωστά στην S2\_OP που διαχειρίζεται τις περιπτώσεις του τελεστή | (μπορεί να είναι μόνο ζευγάρι) οπότε για αυτό μεταβιβαζόμαστε σωστά στην κατάσταση BAD.

Στις εκτελέσεις δώδεκα έως δεκαπέντε, αναλύουμε τις περιπτώσεις που δεν αναγνωρίζονται απο κανένα πρότυπο είτε απο το ενιαίο είτε από τα επιμέρους. Ειδικότερα, στην δωδέκατη εκτέλεση βλέπουμε ότι μεταβαίνει σωστά στην S0\_COMM\_OP που διαχειρίζεται τα σχόλια και τον τελεστή / αλλά δεν υπάρχει \* για να «κλείσει» σωστά το σχόλιο οπότε μεταβαίνει στην κατάσταση BAD. Στην δέκατη τρίτη εκτέλεση, βλέπουμε ότι δεν αναγνωρίζεται απο το κομμάτι του κώδικα που εκτελεί το επιμέρους πεπερασμένο για τους αριθμούς (λόγω της διπλής παύλας). Στην δέκατη τέταρτη εκτέλεση, βλέπουμε ότι το var αναγνωρίζεται ως αναγνωριστικό αλλά το @ δεν μεταβαίνει σε επιτρεπτή κατάσταση λόγω του προτύπου οπότε οδηγείται στην κατάσταση BAD. Τέλος, στην δέκατη πέμπτη κατάσταση, βλέπουμε ότι με το \* και το = οδηγούμαστε σωστά στο πεπερασμένο για τους τελεστές αλλά δεν υπάρχει επιτρεπτή μετάβαση απο την S1\_OP με το = οπότε οδηγούμαστε στην κατάσταση BAD.

## 5. Ανάλυση αρμοδιοτήτων

### 5.1 Γενικές Αρμοδιότητες

		Κοντούλης Δημήτριος (21390095)	Μεντζέλος Άγγελος Κωνσταντίνος (21390132)	Βάρσου Ευφροσύνη (21390021)	Γκιόζι Εντερία (21390041)	Αλεξόπουλος Λεωνίδας (2139006)
Κανονική Έκφραση	Identifiers			✓		
	Strings			✓		
	Numerical		✓			
	Operators	✓ Υλοποίηση	✓ Συμπλήρωση	✓ Υλοποίηση		
	Comments	✓ Διόρθωση	✓ Συμπλήρωση			✓ Υλοποίηση
FSM Κώδικας	Identifiers	✓				
	Strings			✓		
	Numerical		✓			
	Operators		✓ Υλοποίηση	✓ Συμπλήρωση		
	Comments		✓ Συμπλήρωση			✓ Υλοποίηση
Διάγραμμα Μεταβάσεων	Identifiers	✓				
	Strings	✓				
	Numerical		✓ Υλοποίηση	✓ Διόρθωση		
	Operators	✓				
	Comments	✓ Διόρθωση				✓ Υλοποίηση
Πίνακας Μετάβασης	Identifiers				✓	
	Strings				✓	
	Numerical				✓	
	Operators				✓	
	Comments					✓
Ενιαίο Αυτόματο	FSM Κώδικας	✓ Διόρθωση		✓ Υλοποίηση		
	Διάγραμμα Μετάβασης		✓ Συμπλήρωση	✓ Υλοποίηση		
	Γενικός Πίνακας Μετάβασης			✓		
Σχολιασμός Κώδικα (επιμέρους & ενιαίου)		✓				

## 5.2 Υλοποίηση Word

		Κοντούλης Δημήτριος (21390095)	Μεντζέλος Άγγελος Κωνσταντίνος (21390132)	Βάρσου Ευφροσύνη (21390021)	Γκιόζι Εντερίσα (21390041)	Αλεξόπουλος Λεωνίδας (2139006)
1. Εισαγωγή				✓		
2. Τεκμηρίωση	2.1 Κανονικές Εκφράσεις	✓ Λεκτικά Κυριολεκτικά	✓ Αριθμητικά Κυριολεκτικά	✓ Τελεστές	✓ Αναγνωριστικά	✓ Σχόλια
	2.2 Κώδικας FSM		✓			
	2.3 Διαγράμματα Μεταβάσεων					✓
	2.4 Πίνακες Μεταβάσεων				✓	
3. Ενιαίο Πεπερασμένο Αυτόματο	3.1 Κώδικας FSM	✓ Διόρθωση		✓ Υλοποίηση		
	3.2 Διάγραμμα Μεταβάσεων		✓			
	3.3 Γενικός Πίνακας Μετάβασης				✓	
4. Περιπτώσεις Ελέγχου	4.1 Κανονικές Εκφράσεις		✓			
	4.2 Πεπερασμένα Αυτόματα	✓ Εκτέλεση παραδειγμάτων, Σχολιασμός εκτελέσεων για σχόλια	✓ Σχολιασμός εκτελέσεων για αριθμούς	✓ Σχολιασμός υπόλοιπων εκτελέσεων		
5. Ανάλυση αρμοδιοτήτων				✓		