ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αυτόματος Έλεγχος Λεκτικών Αναλυτών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αθηνά Ιωακειμίδη

Επιβλέπων: Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αυτόματος Έλεγχος Λεκτικών Αναλυτών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αθηνά Ιωακειμίδη

Επιβλέπων: Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Ιουλίου 2021.

.....	.....	.....
Νικόλαος Παπασπύρου Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021

.....

Αθηνά Ιωακειμίδη
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Σκοπός της παρούσας διπλωματικής εργασίας είναι η δημιουργία ενός εργαλείου που ελέγχει την ορθότητα μεταγλωττιστών και, πιο συγκεκριμένα, των λεκτικών αναλυτών.

Οι μεταγλωττιστές είναι από τα πιο σημαντικά εργαλεία στην επιστήμη των υπολογιστών. Κάθε πρόγραμμα, για να καταλήξει σε μορφή που να εκτελείται από τον υπολογιστή, πρέπει πρώτα να έχει δεχτεί επεξεργασία από κάποιο μεταγλωττιστή. Αυτός ο μεταγλωττιστής είναι η μοναδική σύνδεση μεταξύ του κώδικα, που διαβάζει και γράφει εύκολα ένας προγραμματιστής, και του προγράμματος, που αναγνωρίζει και μπορεί να τρέξει ο υπολογιστής. Συνεπώς, οι μεταγλωττιστές είναι αναγκαίο και αναπόσπαστο κομμάτι του προγραμματισμού, γι' αυτό και η εξασφάλιση της σωστής λειτουργίας τους είναι υψίστης σημασίας.

Στην παρούσα διπλωματική εργασία ασχολούμαστε με το πρώτο μέρος ενός μεταγλωττιστή, το λεκτικό αναλυτή, και υλοποιούμε ένα εργαλείο για τον έλεγχό του. Ο έλεγχος αυτός θα γίνεται με τυχαία παραγωγή δοκιμαστικών εισόδων, «θετικών» και «αρνητικών», οι οποίες θα τροφοδοτούνται στο υπό έλεγχο πρόγραμμα για να μελετηθεί η ανταπόκρισή του. Η υλοποίηση θα γίνει στη γλώσσα Python, με τη χρήση κανονικών εκφράσεων, πεπερασμένων αυτομάτων και property-based testing.

Λέξεις κλειδιά

Έλεγχος ορθότητας, αυτόματος έλεγχος, τυχαίος έλεγχος, test-case generation, λεκτικός αναλυτής, μεταγλωττιστής, κανονικές εκφράσεις, πεπερασμένα αυτόματα, μετασχηματισμοί πεπερασμένων αυτομάτων, έλεγχος ορθότητας με βάση ιδιότητες

Abstract

The purpose of this diploma thesis is the implementation of a testing tool for compilers and, specifically, for lexical analysers.

Compilers are some of the most important tools in computer science. Every program, in order to become executable by a computer, needs to be processed by a compiler at first. This compiler is the only connection between the code, that a programmer can easily read and write, and the program, that a computer can recognise and run. As a result, compilers are an essential and integral part of programming. This is why the assurance of their correctness is of utmost importance.

In this diploma thesis, we focus on the first part of a compiler, the lexical analyser, and build a testing tool for that. This testing will occur by generating random test-cases, positive and negative, which will then be inputted in the program under test, in order to check its response. The implementation will be in Python and will use regular expressions, finite automata and property-based testing.

Keywords

Testing, automated testing, random testing, test-case generation, lexical analyser, lexer, scanner, compiler, regular expressions, finite automata, finite automata conversions, property-based testing

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κ. Νικόλαο Παπασπύρου, για τη βοήθεια, την υπομονή και την καθοδήγησή του καθ' όλη τη διάρκεια εκπόνησης αυτής της διπλωματικής εργασίας καθώς και για όλα τα ενδιαφέροντα μαθήματα που διδάσκει. Ευχαριστώ πολύ την Ελένη και την Κατερίνα για την πολύτιμη συνεισφορά τους. Τέλος, ευχαριστώ θερμά την οικογένειά μου που πιστεύουν σε εμένα και με στηρίζουν σε ό,τι κάνω.

Αθηνά Ιωακειμίδη,
Αθήνα, 14 Ιουλίου 2021

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-3-21, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2021.

URL: http://www.softlab.ntua.gr/techrep/
FTP: ftp://ftp.softlab.ntua.gr/pub/techrep/

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

1. Εισαγωγή

1.1 Μεταγλωττιστής

Ο *μεταγλωττιστής (compiler)* είναι ένα πρόγραμμα το οποίο δέχεται ως είσοδο ένα πρόγραμμα γραμμένο σε μια γλώσσα L_A και παράγει ως έξοδο ένα ισοδύναμο πρόγραμμα σε μια γλώσσα L_T. Η γλώσσα L_A ονομάζεται *αρχική ή πηγαία γλώσσα (source language)* και είναι συνήθως μια γλώσσα προγραμματισμού υψηλού επιπέδου, ενώ η L_T ονομάζεται *τελική γλώσσα (target language)* και είναι η γλώσσα μηχανής κάποιου υπολογιστή.

Οι γλώσσες υψηλού επιπέδου είναι περιγραφικές γλώσσες και είναι άμεσα κατανοητές από τον άνθρωπο. Αντίθετα, οι γλώσσες μηχανής αποτελούνται από ακολουθίες 0 και 1 και είναι άμεσα κατανοητές από τον υπολογιστή. Οι γλώσσες υψηλού επιπέδου είναι, λοιπόν, εύχρηστες κατά τον προγραμματισμό αλλά πολύ περίπλοκες για να τις διαβάσει ο υπολογιστής. Απεναντίας, η γλώσσα μηχανής αναγνωρίζεται και εκτελείται άμεσα από τον υπολογιστή αλλά είναι πολύ δύσκολο για τον άνθρωπο να γράψει και να συντηρήσει τον κώδικα.

Η δουλειά του μεταγλωττιστή είναι, έτσι, η γεφύρωση του χάσματος μεταξύ του κώδικα που διαβάζει και γράφει εύκολα ένας προγραμματιστής και του προγράμματος που αναγνωρίζει και μπορεί να τρέξει ο υπολογιστής. [1]

1.2 Φάσεις της Μεταγλώττισης

Η διαδικασία της μεταγλώττισης αποτελείται από τρεις κύριες φάσεις: τη λεκτική ανάλυση, τη συντακτική ανάλυση και τη σημασιολογική ανάλυση.

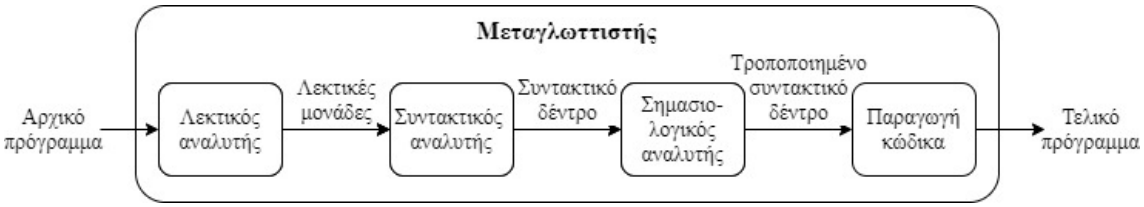
Η *λεκτική ανάλυση (lexical analysis)* είναι η πρώτη φάση της μεταγλώττισης, κατά την οποία το αρχικό πρόγραμμα χωρίζεται σε μια σειρά από λεκτικές μονάδες (tokens). Οι λεκτικές μονάδες κατατάσσονται σε κατηγορίες (όπως αναγνωριστικά, συμβολοσειρές και αριθμητικές σταθερές), κάθε μια από τις οποίες μπορεί να περιγραφεί από μια κανονική έκφραση.

Η *συντακτική ανάλυση (syntax analysis)* είναι η δεύτερη φάση της μεταγλώττισης, κατά την οποία διαβάζονται οι λεκτικές μονάδες και κατασκευάζεται ένα συντακτικό δέντρο (syntax tree) που απεικονίζει τη συντακτική δομή του αρχικού προγράμματος.

Η *σημασιολογική ανάλυση (semantic analysis)* είναι η τρίτη φάση της μεταγλώττισης, κατά την οποία το συντακτικό δέντρο της προηγούμενης φάσης τροποποιείται και συμπληρώνεται με πληροφορίες σημασιολογικής φύσης. Επίσης, στο πρόγραμμα εκτελούνται σημασιολογικοί έλεγχοι, όπως ο έλεγχος τύπων (type checking).

Τα επόμενα βήματα που κάνει ένας μεταγλωττιστής είναι η παραγωγή κώδικα σε μια ενδιάμεση γλώσσα (intermediate language) με βάση το συντακτικό δέντρο. Αυτός ο κώδικας, μετά από βελτιστοποιήσεις, θα αποτελέσει το τελικό εκτελέσιμο πρόγραμμα.

Στο Σχήμα 1.1 δίνεται μια αναπαράσταση της δομής ενός μεταγλωττιστή, όπως περιγράφηκε παραπάνω.



Σχήμα 1.1: Δομή ενός μεταγλωττιστή

Σε αυτή την εργασία θα εστιάσουμε στην πρώτη φάση, τη λεκτική ανάλυση, και πιο συγκεκριμένα στο πρόγραμμα που την εκτελεί, τον *λεκτικό αναλυτή (lexical analyser, lexer ή scanner).* [1]

1.3 Κίνητρο

Η πρώτη και πιο βασική απαίτηση που έχουμε από ένα μεταγλωττιστή είναι να λειτουργεί σωστά, δηλαδή να μεταφράζει σωστά ένα πρόγραμμα από την αρχική στην τελική γλώσσα. Ένας μεταγλωττιστής που περιέχει λάθη, ακόμα και αν αυτά εμφανίζονται πολύ σπάνια, δεν μπορεί να χρησιμοποιηθεί και πρέπει να εγκαταλείπεται. Ένα λάθος στο μεταγλωττιστή θα περαστεί σίγουρα στα προγράμματα που αυτός παράγει. Και τα σφάλματα που προκύπτουν από λανθασμένη μεταγλώττιση είναι πολύ δύσκολο να εντοπιστούν και να αντιμετωπιστούν. Είναι εξαιρετικά σημαντικό, λοιπόν, να εξασφαλίσουμε ότι ένας μεταγλωττιστής δουλεύει όπως είναι αναμενόμενο.

Για να εξετάσουμε αν ένας μεταγλωττιστής λειτουργεί σωστά, αρκεί να ελέγξουμε ότι οι επιμέρους φάσεις του είναι σωστές. Άρα, προκειμένου να εξασφαλίσουμε ότι ένας μεταγλωττιστής είναι σωστός, πρέπει πρώτα να εξασφαλίσουμε ότι ο λεκτικός αναλυτής του είναι σωστός. [1]

Στην παρούσα διπλωματική εργασία θα επιχειρήσουμε να ελέγξουμε αν ο λεκτικός αναλυτής είναι σωστά υλοποιημένος με χρήση των κανονικών εκφράσεων που περιγράφουν τις λεκτικές μονάδες του.

2. Κανονικές Εκφράσεις

Οι κανονικές εκφράσεις είναι ακολουθίες χαρακτήρων που προσδιορίζουν μοτίβα αναζήτησης και είναι ο βασικός τρόπος περιγραφής των λεκτικών μονάδων μιας γλώσσας προγραμματισμού. Είναι ιδιαίτερα σημαντικές κατά την κατασκευή μεταγλωττιστών καθώς συστηματοποιούν την περιγραφή των λέξεων της αρχικής γλώσσας, η οποία είναι απαραίτητη για την κατασκευή του λεκτικού αναλυτή.

Οι κανονικές εκφράσεις έχουν πολλές εφαρμογές στην επιστήμη των υπολογιστών, με κυριότερη την αναζήτηση λέξεων σε κείμενο. Για το λόγο αυτό, πολλές γλώσσες παρέχουν βιβλιοθήκες και συναρτήσεις για τη χρήση κανονικών εκφράσεων, συνήθως με μικρές παραλλαγές στην εφαρμογή τους μεταξύ των διαφορετικών γλωσσών προγραμματισμού.

Τυπικός ορισμός:

Με δεδομένο ένα αλφάβητο Σ, *κανονικές εκφράσεις (regular expressions)* είναι:

1. η κενή συμβολοσειρά ϵ
2. κάθε σύμβολο a του αλφάβητου Σ
3. η παράθεση δύο κανονικών εκφράσεων r και s, η οποία συμβολίζεται με rs
4. η διάζευξη δύο κανονικών εκφράσεων r και s, η οποία συμβολίζεται με $r|s$
5. το αστέρι Kleene μιας κανονικής έκφρασης r, το οποίο συμβολίζεται με $r^\star$
6. η παρένθεση μιας κανονικής έκφρασης r, η οποία συμβολίζεται με (r)

όπου το αστέρι Kleene $r^\star$ ορίζεται ως η παράθεση 0 ή περισσότερων επαναλήψεων της κανονικής έκφρασης r.

Ο τυπικός ορισμός περιλαμβάνει το ελάχιστο δυνατό σύνολο κανόνων που μπορούν να ορίσουν μια κανονική έκφραση. Ωστόσο, υπάρχουν πολλοί ακόμη τελεστές που χρησιμοποιούνται στις κανονικές εκφράσεις, με σκοπό αυτές να γίνουν πιο συνοπτικές και, κατά συνέπεια, πιο εύχρηστες και πρακτικές.

Κάποιοι τέτοιοι επιπλέον τελεστές είναι:

- ο τελεστής $+$, όπου η έκφραση r^+ ορίζεται ως η παράθεση 1 ή περισσότερων επαναλήψεων της κανονικής έκφρασης r, δηλαδή $r^+ = rr^\star$,
- ο τελεστής ?, όπου η έκφραση $r?$ δείχνει ότι η κανονική έκφραση r είναι προαιρετική, δηλαδή $r? = r|\epsilon$, και
- ο τελεστής $\sim$, όπου η έκφραση $\sim r$ ορίζεται ως το συμπλήρωμα (άρνηση) της κανονικής έκφρασης r.

Η σειρά προτεραιότητας των τελεστών που εμφανίζονται στον τυπικό ορισμό είναι ως εξής: ο τελεστής $\star$ έχει τη μεγαλύτερη προτεραιότητα, μετά ακολουθεί η παράθεση και τέλος η διάζευξη. Οι τελεστές εφαρμόζονται αυτόματα στις μικρότερες δυνατές συμβολοσειρές που αποτελούν κανονικές εκφράσεις και στις οποίες εφάπτεται ο κάθε τελεστής.

Για την εφαρμογή των τελεστών σε μεγαλύτερες κανονικές εκφράσεις χρησιμοποιούμε παρενθέσεις. Οι παρενθέσεις χρησιμοποιούνται επίσης για την αποφυγή ασαφειών και παρανοήσεων, αν και συνήθως παραλείπονται.

Οι κανονικές εκφράσεις μπορούν να χρησιμοποιηθούν για περιγράψουν κανονικές γλώσσες. Η κανονική γλώσσα που περιγράφεται από την κανονική έκφραση r συμβολίζεται με $L(r)$ και σε αυτήν εφαρμόζονται όλοι οι τελεστές που ορίστηκαν παραπάνω.

Οι κανονικές γλώσσες αναγνωρίζονται από τα πεπερασμένα αυτόματα, που είναι και ο τύπος αφηρημένων μηχανών που τους αντιστοιχεί (Θεώρημα του Kleene). Για τα πεπερασμένα αυτόματα καθώς και για τη μετατροπή μεταξύ κανονικής έκφρασης και πεπερασμένου αυτόματου, θα μιλήσουμε στα επόμενα κεφάλαια. [1] [2]

Παράδειγμα:

Έστω η κανονική έκφραση $(01|1)^{\star}$, μερικές λέξεις της γλώσσας που περιγράφει είναι οι εξής:

- ϵ
- 01
- 1
- 0101
- 01101
- 010111011

3. Πεπερασμένα Αυτόματα

Τα πεπερασμένα αυτόματα (finite automata) είναι αφηρημένες μηχανές που αναγνωρίζουν λέξεις για κανονικές γλώσσες. Λέγονται «πεπερασμένα» γιατί ο αριθμός των καταστάσεων που μπορεί να πάρει το κάθε αυτόματο είναι πεπερασμένος.

Τα πεπερασμένα αυτόματα δέχονται ως είσοδο μια λέξη, ένα γράμμα τη φορά, και αναλόγως με το γράμμα-είσοδο μεταβαίνουν στην αντίστοιχη κατάσταση. Όταν ολοκληρωθεί η λέξη, αν το αυτόματο βρίσκεται σε τελική κατάσταση (αλλιώς κατάσταση αποδοχής) τότε αναγνωρίζει τη λέξη εισόδου. Αν δε βρίσκεται σε τελική κατάσταση τότε την απορρίπτει.

Το σύνολο A των λέξεων που μπορεί να αναγνωρίσει το πεπερασμένο αυτόματο M λέγεται *γλώσσα του* M και συμβολίζεται με $L(M) = A$. Κάθε πεπερασμένο αυτόματο μπορεί να αναγνωρίσει πολλές λέξεις, αλλά πάντα μόνο μία γλώσσα.

Κάθε πεπερασμένο αυτόματο ορίζεται από το αλφάβητό του, το σύνολο των καταστάσεων όπου μπορεί να βρεθεί, την αρχική κατάσταση και τις εισόδους που προκαλούν τις μεταβάσεις καθώς και το σύνολο των τελικών καταστάσεών του.

Υπάρχουν δύο τύποι πεπερασμένων αυτομάτων:

- τα ντετερμινιστικά πεπερασμένα αυτόματα (DFA), και
- τα μη-ντετερμινιστικά πεπερασμένα αυτόματα (NFA)

Τα ντετερμινιστικά και τα μη-ντετερμινιστικά πεπερασμένα αυτόματα είναι ισοδύναμα από πλευράς υπολογιστικής ικανότητας. Γι' αυτό και μπορούμε να μετατρέψουμε ένα μη-ντετερμινιστικό σε ένα αντίστοιχο ντετερμινιστικό πεπερασμένο αυτόματο και αντίστροφα. Τον αλγόριθμο του μετασχηματισμού μεταξύ των δύο αυτομάτων θα αναλύσουμε στο επόμενο κεφάλαιο. [1] [3]

3.1 Ντετερμινιστικό Πεπερασμένο Αυτόματο (DFA)

Ορισμός:

Ένα *ντετερμινιστικό πεπερασμένο αυτόματο (deterministic finite automaton, DFA)* ορίζεται ως η πεντάδα $M = (\Sigma, Q, \delta, q_0, F)$, όπου:

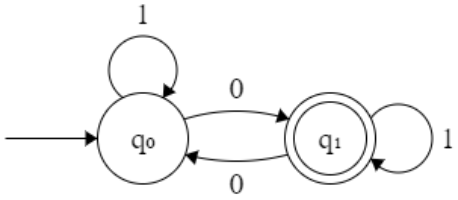
- Σ είναι το αλφάβητο του αυτόματου, ένα μη κενό και πεπερασμένο σύνολο χαρακτήρων
- Q είναι ένα μη κενό και πεπερασμένο σύνολο καταστάσεων
- δ είναι η συνάρτηση μετάβασης για την οποία ισχύει: $\delta : Q \times \Sigma \to Q$
- q_0 είναι η αρχική κατάσταση και ισχύει: $q_0 \in Q$
- F είναι το σύνολο των τελικών καταστάσεων και ισχύει: $F \subseteq Q$

Σε ένα ντετερμινιστικό πεπερασμένο αυτόματο η συνάρτηση μετάβασης δ ορίζεται μονοσήμαντα για κάθε ζευγάρι τρέχουσας κατάστασης και χαρακτήρα εισόδου, δηλαδή $\delta(q, a) \in Q$. Με άλλα λόγια από κάθε κατάσταση ξεκινά πάντα ένα και μόνο ένα βέλος μετάβασης για κάθε σύμβολο του αλφάβητου. Αυτός είναι ο λόγος που αυτά τα πεπερασμένα αυτόματα χαρακτηρίζονται «ντετερμινιστικά».

Έτσι, κατά την αναγνώριση μιας λέξης το ντετερμινιστικό πεπερασμένο αυτόματο ξεκινάει από την κατάσταση q_0. Για κάθε γράμμα a της εισόδου το αυτόματο θα μεταβεί

από την κατάσταση q όπου βρισκόταν, στη μοναδική κατάσταση $\delta(q, a)$. Η διαδικασία αυτή συνεχίζεται για κάθε επόμενο γράμμα μέχρι να ολοκληρωθεί η λέξη. Το αυτόματο ελέγχει τότε αν έχει καταλήξει σε τελική κατάσταση και άρα αναγνωρίζει τη λέξη, ή αν δεν έχει καταλήξει σε τελική κατάσταση και άρα την απορρίπτει. [1] [3]

Παράδειγμα:



Σχήμα 3.1: Το ντετερμινιστικό πεπερασμένο αυτόματο M_1

Το αυτόματο M_1 μπορεί να περιγραφεί σύμφωνα με τον τυπικό ορισμό ως εξής:

- $\Sigma = \{0, 1\}$
- $Q = \{q_0,\ q_1\}$
- συνάρτηση μετάβασης δ σε μορφή πίνακα:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

- q_0 είναι η αρχική κατάσταση
- $F = \{q_1\}$

Έστω η λέξη εισόδου $\lambda_1 = 010011$, το αυτόματο M_1 θα περάσει διαδοχικά από τις εξής καταστάσεις:

$$q_0 \rightarrow\ q_1 \rightarrow q_1 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow \boldsymbol{q_1}$$

Με το πέρας της λέξης το αυτόματο καταλήγει στην κατάσταση q_1 η οποία ανήκει στις τελικές καταστάσεις και άρα η λέξη αναγνωρίζεται.

Έστω η λέξη εισόδου $\lambda_2 = 1101110$, το αυτόματο M_1 θα περάσει διαδοχικά από τις εξής καταστάσεις:

$$q_0 \rightarrow\ q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_1 \rightarrow q_1 \rightarrow \boldsymbol{q_0}$$

Με το πέρας της λέξης το αυτόματο καταλήγει στην κατάσταση q_0 η οποία *δεν* ανήκει στις τελικές καταστάσεις και άρα η λέξη απορρίπτεται.

3.2 Μη-ντετερμινιστικό Πεπερασμένο Αυτόματο (NFA)

Ορισμός:

Ένα *μη-ντετερμινιστικό πεπερασμένο αυτόματο (non-deterministic finite automaton, NFA)* ορίζεται ως η πεντάδα $M = (\Sigma, Q, \delta, q_0, F)$, όπου:

- Σ είναι το αλφάβητο του αυτόματου, ένα μη κενό και πεπερασμένο σύνολο χαρακτήρων
- Q είναι ένα μη κενό και πεπερασμένο σύνολο καταστάσεων
- δ είναι η συνάρτηση μετάβασης για την οποία ισχύει: $\delta : Q \times \Sigma \to \mathcal{P}(Q)$
- q_0 είναι η αρχική κατάσταση και ισχύει: $q_0 \in Q$
- F είναι το σύνολο των τελικών καταστάσεων και ισχύει: $F \subseteq Q$

Σε ένα μη-ντετερμινιστικό πεπερασμένο αυτόματο η συνάρτηση μετάβασης δ δεν ορίζεται μονοσήμαντα για κάθε ζευγάρι τρέχουσας κατάστασης και χαρακτήρα εισόδου, δηλαδή $\delta(q, a) \subseteq Q$. Με άλλα λόγια από κάθε κατάσταση μπορούν να ξεκινούν μηδέν, ένα ή περισσότερα βέλη μετάβασης για κάθε σύμβολο του αλφάβητου.

Έτσι, κατά την αναγνώριση μιας λέξης το ντετερμινιστικό πεπερασμένο αυτόματο ξεκινάει από την κατάσταση q_0. Για κάθε γράμμα a της εισόδου το αυτόματο θα επιλέξει αυθαίρετα μία εκ των καταστάσεων $\delta(q, a)$ για να μεταβεί από την κατάσταση q στην κατάσταση $q' \in \delta(q, a)$. Αν το $\delta(q, a)$ είναι κενό ($\delta(q, a) = \emptyset$), τότε η διαδικασία σταματάει και το αυτόματο απορρίπτει τη λέξη. Αλλιώς η διαδικασία συνεχίζεται για κάθε επόμενο γράμμα μέχρι να ολοκληρωθεί η λέξη. Το αυτόματο ελέγχει τότε αν έχει καταλήξει σε τελική κατάσταση και άρα αναγνωρίζει τη λέξη, ή αν δεν έχει καταλήξει σε τελική κατάσταση και άρα την απορρίπτει.

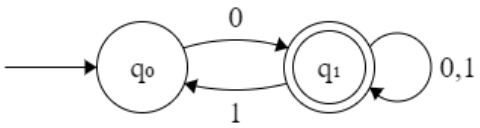
Η αυθαιρεσία στην επιλογή μεταξύ των καταστάσεων $\delta(q, a)$ είναι αυτή που δίνει σε αυτά τα πεπερασμένα αυτόματα το χαρακτηρισμό «μη-ντετερμινιστικά».

Παρακάτω δίνονται μερικοί χρήσιμοι ορισμοί για το μη-ντετερμινιστικό πεπερασμένο αυτόματο που θα χρειαστούν σε μελλοντικά κεφάλαια.

Ως *μη-ντετερμινιστικό πεπερασμένο αυτόματο με μηδενικές μεταβάσεις (NFA-ϵ)* ορίζεται οποιοδήποτε πεπερασμένο αυτόματο που περιέχει μηδενικές "αυθόρμητες" μεταβάσεις, δηλαδή χωρίς να διαβάζονται χαρακτήρες από την είσοδο.

ϵ-closure μιας δεδομένης κατάστασης q είναι το σύνολο των καταστάσεων όπου το αυτόματο NFA-ϵ που την περιέχει μπορεί να φτάσει από την κατάσταση q αποκλειστικά με μηδενικές μεταβάσεις. Μπορούμε να ορίσουμε το ϵ-closure της κατάστασης q και αναδρομικά, ως την ένωση των ϵ-closure των καταστάσεων όπου μπορεί να φτάσει το αυτόματο με μία μόνο μηδενική μετάβαση. [1] [3]

Παράδειγμα:



Σχήμα 3.2: Το μη-ντετερμινιστικό πεπερασμένο αυτόματο M_2

Το αυτόματο M_2 μπορεί να περιγραφεί σύμφωνα με τον τυπικό ορισμό ως εξής:

- $\Sigma = \{0, 1\}$
- $Q = \{q_0,\ q_1\}$
- συνάρτηση μετάβασης δ σε μορφή πίνακα:

	0	1
q_0	q_1	
q_1	q_1	q_0, q_1

- q_0 είναι η αρχική κατάσταση
- $F = \{q_1\}$

Έστω η λέξη εισόδου $\lambda_1 = 001$, το αυτόματο M_2 θα περάσει διαδοχικά είτε από τις καταστάσεις:

$$q_0 \rightarrow\ q_1 \rightarrow q_1 \rightarrow \boldsymbol{q_1}$$

είτε από τις καταστάσεις:

$$q_0 \rightarrow\ q_1 \rightarrow q_1 \rightarrow \boldsymbol{q_0}$$

αφού ισχύει $\delta(q_1, 1) = \{q_0, q_1\}$. Η επιλογή από αυτό το υποσύνολο καταστάσεων γίνεται αυθαίρετα, γι' αυτό και αν εισάγουμε την ίδια λέξη πολλές φορές το αποτέλεσμα του αυτόματου μπορεί να διαφέρει. Δηλαδή, αν το αυτόματο επιλέξει να παραμείνει στο q_1, τότε καταλήγει σε τελική κατάσταση και άρα αναγνωρίζει τη λέξη, ενώ αν επιλέξει να μεταβεί στο q_0 που δεν είναι τελική κατάσταση, τότε την απορρίπτει. Αφού υπάρχει έστω και μία ακολουθία εκτέλεσης που αναγνωρίζει τη λέξη, λέμε ότι το αυτόματο αναγνωρίζει τη λέξη αυτή.

Έστω η λέξη εισόδου $\lambda_2 = 1101$, το αυτόματο M_2 την απορρίπτει αμέσως γιατί στη συνάρτηση δ δεν υπάρχει μετάβαση από αρχική κατάσταση q_0 με είσοδο το χαρακτήρα 1, δηλαδή $\delta(q_0, 1) = \emptyset$.

4. Μετασχηματισμοί Πεπερασμένων Αυτομάτων

Οι κανονικές εκφράσεις και τα πεπερασμένα αυτόματα, τόσο τα ντετερμινιστικά όσο και τα μη-ντετερμινιστικά, έχουν ισοδύναμη εκφραστική ισχύ. Αυτό σημαίνει ότι για κάθε κανονική έκφραση r μπορεί να κατασκευαστεί ένα ντετερμινιστικό πεπερασμένο αυτόματο M τέτοιο ώστε $L(r) = L(M)$, και αντίστροφα. Η απόδειξη της ισοδυναμίας μεταξύ αυτών των μοντέλων γίνεται κατασκευαστικά.

Σε αυτό το κεφάλαιο θα ασχοληθούμε με δύο τέτοιες κατασκευές:

- από κανονική έκφραση σε μη-ντετερμινιστικό πεπερασμένο αυτόματο με μηδενικές μεταβάσεις, και
- από μη ντετερμινιστικό πεπερασμένο αυτόματο με μηδενικές μεταβάσεις σε ντετερμινιστικό πεπερασμένο αυτόματο,

γιατί είναι οι πλέον χρήσιμες για την κατασκευή λεκτικών αναλυτών.

Θα ασχοληθούμε επίσης με έναν αλγόριθμο βελτιστοποίησης ντετερμινιστικών πεπερασμένων αυτομάτων, έτσι ώστε να έχουν όσο γίνεται μικρότερο πλήθος καταστάσεων.

4.1 Μετασχηματισμός RE → NFA-ϵ

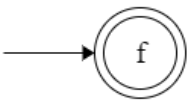
Σε αυτό το κεφάλαιο θα δούμε πώς γίνεται ο μετασχηματισμός μιας κανονικής έκφρασης σε μη-ντετερμινιστικό πεπερασμένο αυτόματο με μηδενικές μεταβάσεις.

Ο αλγόριθμος του Thompson δουλεύει αναδρομικά με βάση τον τυπικό ορισμό των κανονικών εκφράσεων. Έτσι κάθε κανονική έκφραση που δεν είναι στοιχειώδης (κενή συμβολοσειρά ή χαρακτήρας του αλφάβητου) μπορεί να μετασχηματιστεί σε αντίστοιχο NFA-ϵ με τη χρήση της μετατροπής των στοιχειωδών στοιχείων.

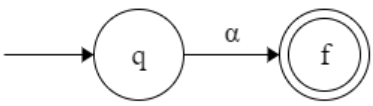
Κάθε κανονική έκφραση r αντιστοιχεί σε ένα NFA-ϵ το οποίο συμβολίζεται με $M(r)$ και θα αναπαρίσταται παρακάτω με το αντίστοιχο διάγραμμα καταστάσεων. Θεωρούμε ότι κάθε αυτόματο έχει μόνο μία τελική κατάσταση και δεν υπάρχουν μεταβάσεις που ξεκινούν από αυτή.

Η κανονική έκφραση R μετασχηματίζεται στο αντίστοιχο μη-ντετερμινιστικό πεπερασμένο αυτόματο $M(R)$ με βάση τον αλγόριθμο του Thompson ως εξής:

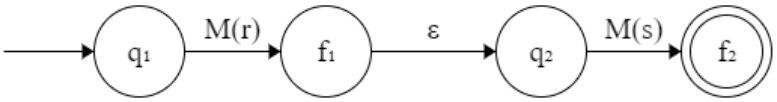
1. η κενή συμβολοσειρά $R = \epsilon$ μετασχηματίζεται στο αυτόματο $M(\epsilon)$ που αποτελείται μόνο από μία κατάσταση η οποία είναι ταυτόχρονα και αρχική και τελική:



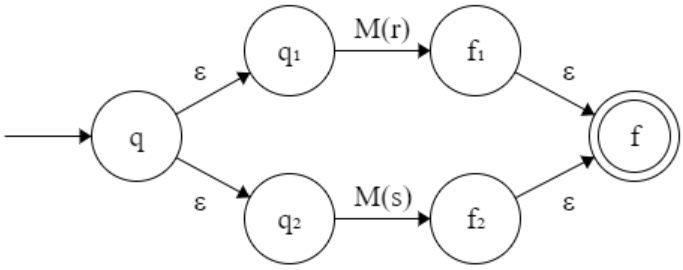
2. το σύμβολο $R = a$ μετασχηματίζεται στο αυτόματο $M(a)$ που αποτελείται μόνο από μία μετάβαση για το σύμβολο a μεταξύ της αρχικής και της τελικής κατάστασης:



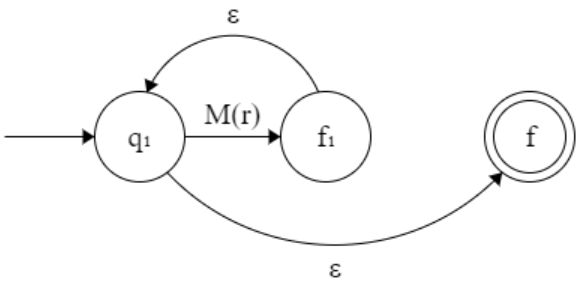
3. η παράθεση $R = rs$ δύο κανονικών εκφράσεων r και s μετασχηματίζεται στο αυτόματο $M(rs)$: Αρχικά κατασκευάζουμε αναδρομικά τα αυτόματα $M(r)$ και $M(s)$ με βάση το 2ο κανόνα. Η τελική κατάσταση του $M(r)$ συνδέεται με μια μηδενική μετάβαση με την αρχική κατάσταση του $M(s)$ και η τελική κατάσταση του $M(r)$ παύει να είναι τελική. Για το τελικό αυτόματο $M(rs)$ ορίζεται ως αρχική κατάσταση η αρχική κατάσταση του $M(r)$ και ως τελική η τελική κατάσταση του $M(s)$.



4. η διάζευξη $R = r|s$ δύο κανονικών εκφράσεων r και s μετασχηματίζεται στο αυτόματο $M(r|s)$: Αρχικά κατασκευάζουμε αναδρομικά τα αυτόματα $M(r)$ και $M(s)$. Προσθέτουμε στο διάγραμμα μια καινούρια αρχική κατάσταση και τη συνδέουμε με μηδενικές μεταβάσεις με τις αρχικές καταστάσεις και των δύο αυτομάτων $M(r)$ και $M(s)$. Επιπλέον προσθέτουμε μια καινούρια τελική κατάσταση και συνδέουμε τις τελικές καταστάσεις των δύο αυτομάτων $M(r)$ και $M(s)$ με αυτήν. Οι τελικές καταστάσεις των $M(r)$ και $M(s)$ παύουν πλέον να είναι τελικές. Το τελικό διάγραμμα καταστάσεων ορίζει το αυτόματο $M(r|s)$.



5. το αστέρι Kleene $R = r^{\star}$ μιας κανονικής έκφρασης r μετασχηματίζεται στο αυτόματο $M(r^{\star})$: Αρχικά κατασκευάζουμε αναδρομικά το αυτόματο $M(r)$. Προσθέτουμε στο διάγραμμα μια καινούρια τελική κατάσταση και δύο μηδενικές μεταβάσεις: η μία συνδέει την τελική κατάσταση του $M(r)$ με την αρχική και η άλλη συνδέει την αρχική κατάσταση του $M(r)$ με την καινούρια τελική κατάσταση. Η τελική κατάσταση του αυτόματου $M(r)$ παύει πλέον να είναι τελική. Το διάγραμμα καταστάσεων που μόλις φτιάξαμε ορίζει το αυτόματο $M(r^{\star})$.

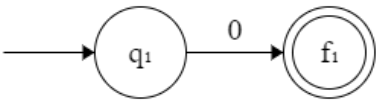


[1]

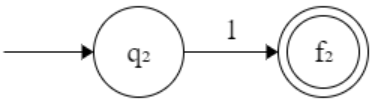
Παράδειγμα:

Έστω η κανονική έκφραση $(01|1)^*$ που μετασχηματίζεται σε NFA-ϵ με τα παρακάτω βήματα:

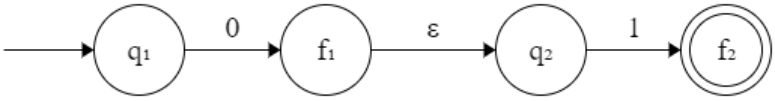
Βήμα 1: $\boldsymbol{M(0)}$



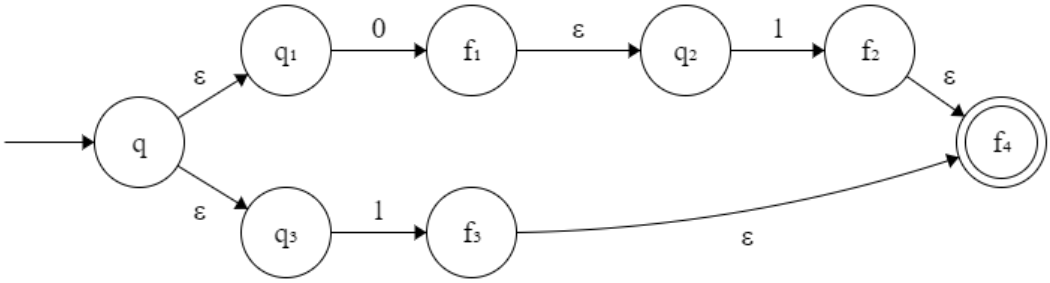
Βήμα 2: $\boldsymbol{M(1)}$



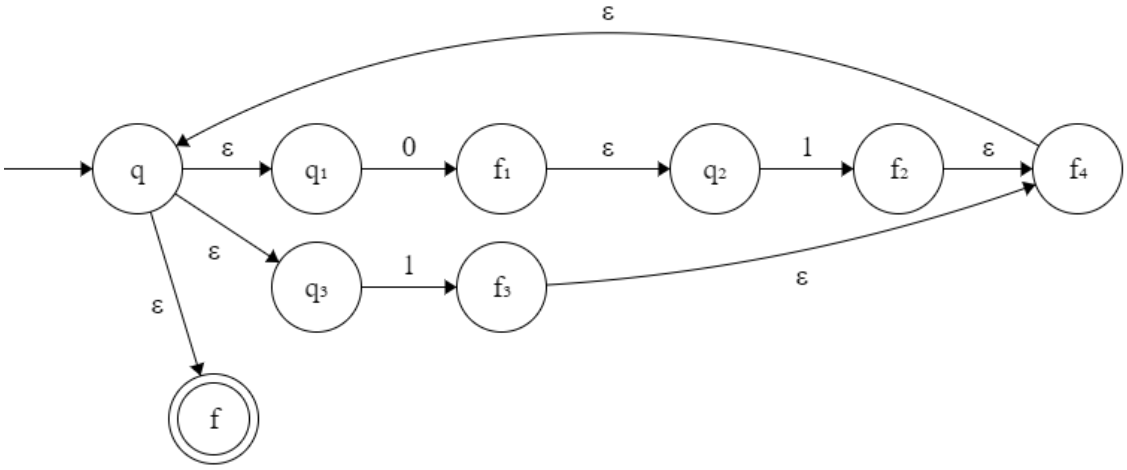
Βήμα 3: $\boldsymbol{M(01)}$



Βήμα 4: $\boldsymbol{M(01|1)}$



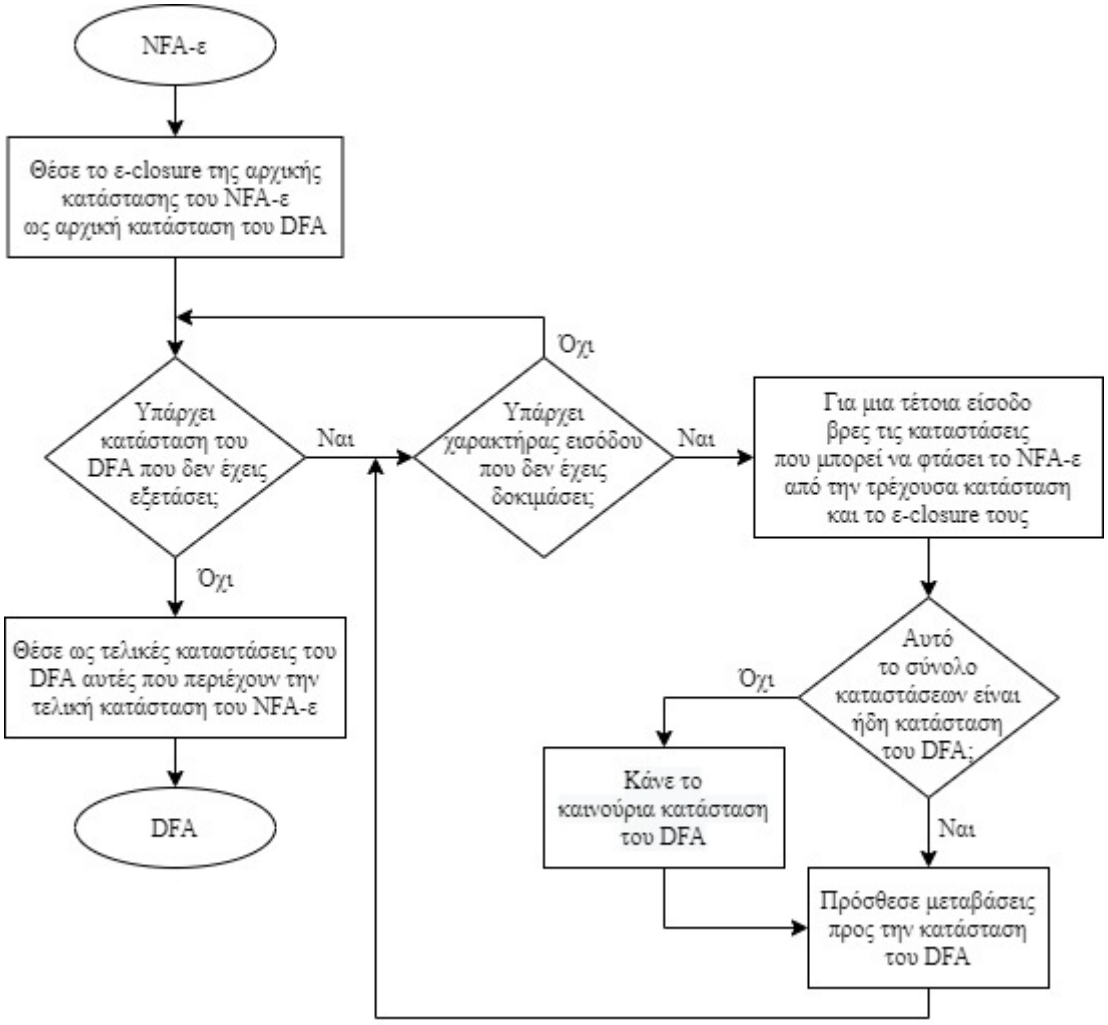
Βήμα 5: $\boldsymbol{M((01|1)^*)}$



Σχήμα 4.1: Το NFA-ϵ που αντιστοιχεί στην κανονική έκφραση (01|1)*

4.2 Μετασχηματισμός NFA-ϵ → DFA

Σε αυτό το κεφάλαιο θα δούμε πώς γίνεται ο μετασχηματισμός ενός μη-ντετερμινιστικού πεπερασμένου αυτόματου με μηδενικές μεταβάσεις σε ντετερμινιστικό πεπερασμένο αυτόματο.



Σχήμα 4.2: Διάγραμμα ροής μετασχηματισμού NFA-ϵ → DFA

Ο αλγόριθμος του μετασχηματισμού ενός NFA-ϵ σε αντίστοιχο DFA είναι ο εξής:

Βήμα 1: Θέσε ως αρχική κατάσταση του DFA το ϵ-closure της αρχικής κατάστασης του NFA-ϵ. Δηλαδή: $q'_0 = \epsilon\text{-}closure(q_0)$

Βήμα 2: Για κάθε σύμβολο του αλφαβήτου: Βρες τις καταστάσεις που μπορεί να φτάσει το του NFA-ϵ από την τρέχουσα κατάσταση και το ϵ-closure τους. Αν το αυτό το σύνολο καταστάσεων δεν υπάρχει ήδη σαν κατάσταση στο DFA, φτιάξε καινούρια κατάσταση του DFA. Πρόσθεσε τις μεταβάσεις προς την κατάσταση του DFA.

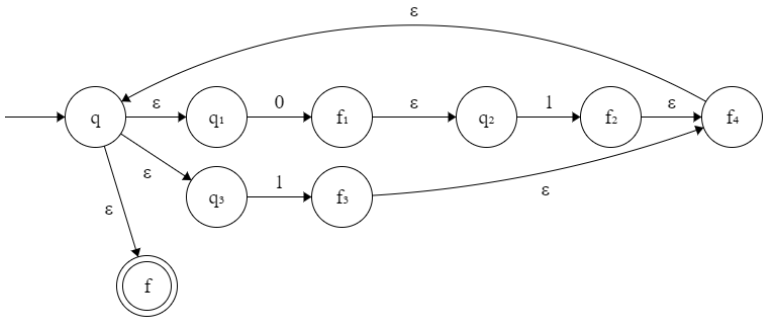
Βήμα 3: Αν υπάρχει καινούρια κατάσταση, θέσε την ως τρέχουσα κατάσταση και επανάλαβε το βήμα 2.

Βήμα 4: Επανάλαβε τα βήματα 2 και 3 μέχρι να μην υπάρχουν άλλες καινούριες καταστάσεις.

Βήμα 5: Θέσε ως τελικές καταστάσεις του DFA τις καταστάσεις που περιέχουν την τελική κατάσταση του NFA-ϵ.

Ένα διάγραμμα ροής του αλγορίθμου αυτού φαίνεται στο Σχήμα 4.2. [1]

Παράδειγμα:



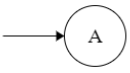
Στο παραπάνω NFA-ϵ το ϵ-closure για κάθε κατάσταση είναι:

$$\epsilon\text{-}closure(q) = \{q, q_1, q_3, f\} \qquad \epsilon\text{-}closure(f) = \{f\}$$
$$\epsilon\text{-}closure(q_1) = \{q_1\} \qquad \epsilon\text{-}closure(f_1) = \{f_1, q_2\}$$
$$\epsilon\text{-}closure(q_2) = \{q_2\} \qquad \epsilon\text{-}closure(f_2) = \{f_2, f_4, q, q_1, q_3, f\}$$
$$\epsilon\text{-}closure(q_3) = \{q_3\} \qquad \epsilon\text{-}closure(f_3) = \{f_3, f_4, q, q_1, q_3, f\}$$
$$\epsilon\text{-}closure(f_4) = \{f_4, q, q_1, q_3, f\}$$

Εφαρμόζουμε τον αλγόριθμο μετασχηματισμού σε DFA και έχουμε:

Βήμα 1: Αρχική κατάσταση

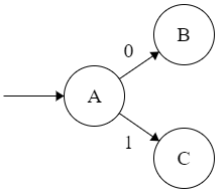
$$A = \epsilon\text{-}closure(q) \Rightarrow \boldsymbol{A = \{q, q_1, q_3, f\}}$$



Βήμα 2:

$$\delta'(A,0) = \epsilon\text{-}closure\big(\delta(q,0) \cup \delta(q_1,0) \cup \delta(q_3,0) \cup \delta(f,0)\big) = \epsilon\text{-}closure(\emptyset \cup \{f_1\} \cup \emptyset \cup \emptyset) = \{f_1, q_2\} \Rightarrow \boldsymbol{B = \{f_1, q_2\}}$$

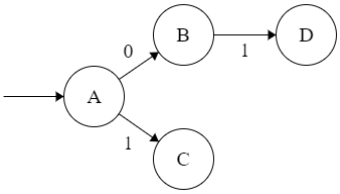
$$\delta'(A,1) = \epsilon\text{-}closure\big(\delta(q,1) \cup \delta(q_1,1) \cup \delta(q_3,1) \cup \delta(f,1)\big) = \epsilon\text{-}closure(\emptyset \cup \emptyset \cup \{f_3\} \cup \emptyset) = \{f_3, f_4, q, q_1, q_3, f\} \Rightarrow \boldsymbol{C = \{f_3, f_4, q, q_1, q_3, f\}}$$



Βήμα 3:

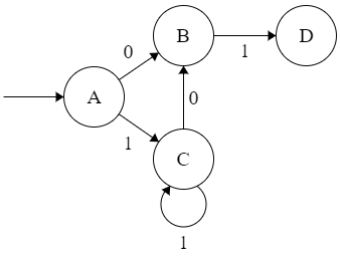
$$\delta'(B,0) = \epsilon\text{-}closure(\emptyset) = \emptyset$$

$$\delta'(B,1) = \epsilon\text{-}closure(\{f_2\}) = \{f_2, f_4, q, q_1, q_3, f\} \Rightarrow \boldsymbol{D = \{f_2, f_4, q, q_1, q_3, f\}}$$



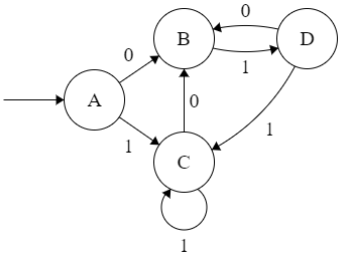
Βήμα 4:

$$\delta'(C,0) = \epsilon\text{-}closure(\{f_1\}) = \{f_1, q_2\} = B$$
$$\delta'(C,1) = \epsilon\text{-}closure(\{f_3\}) = \{f_3, f_4, q, q_1, q_3, f\} = C$$

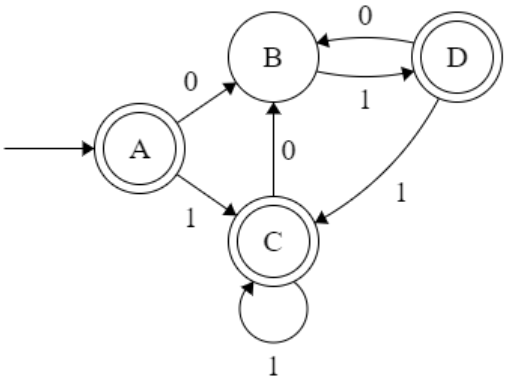


Βήμα 5:

$$\delta'(D,0) = \epsilon\text{-}closure(\{f_1\}) = \{f_1, q_2\} = B$$
$$\delta'(D,1) = \epsilon\text{-}closure(\{f_3\}) = \{f_3, f_4, q, q_1, q_3, f\} = C$$



Βήμα 6: Τελικές καταστάσεις οι A, C, D



Σχήμα 4.3: Το DFA που αντιστοιχεί στην κανονική έκφραση (01|1)*

4.3 Ελαχιστοποίηση DFA

Σε αυτό το κεφάλαιο θα δούμε πώς γίνεται η ελαχιστοποίηση ενός ντετερμινιστικού πεπερασμένου αυτόματου.

Ο αλγόριθμος της ελαχιστοποίησης ενός DFA με βάση το *Θεώρημα Myhill–Nerode* είναι ο εξής:

Βήμα 1: Φτιάξε δύο σύνολα από ζευγάρια καταστάσεων:

$$S = \{ (q_i, q_j) \mid i < j \text{ και } (q_i, q_j \in F \text{ ή } q_i, q_j \notin F) \}$$
$$T = \{ (q_i, q_j) \mid i < j \text{ και } ((q_i \in F \text{ και } q_j \notin F) \text{ ή } (q_i \notin F \text{ και } q_j \in F)) \}$$

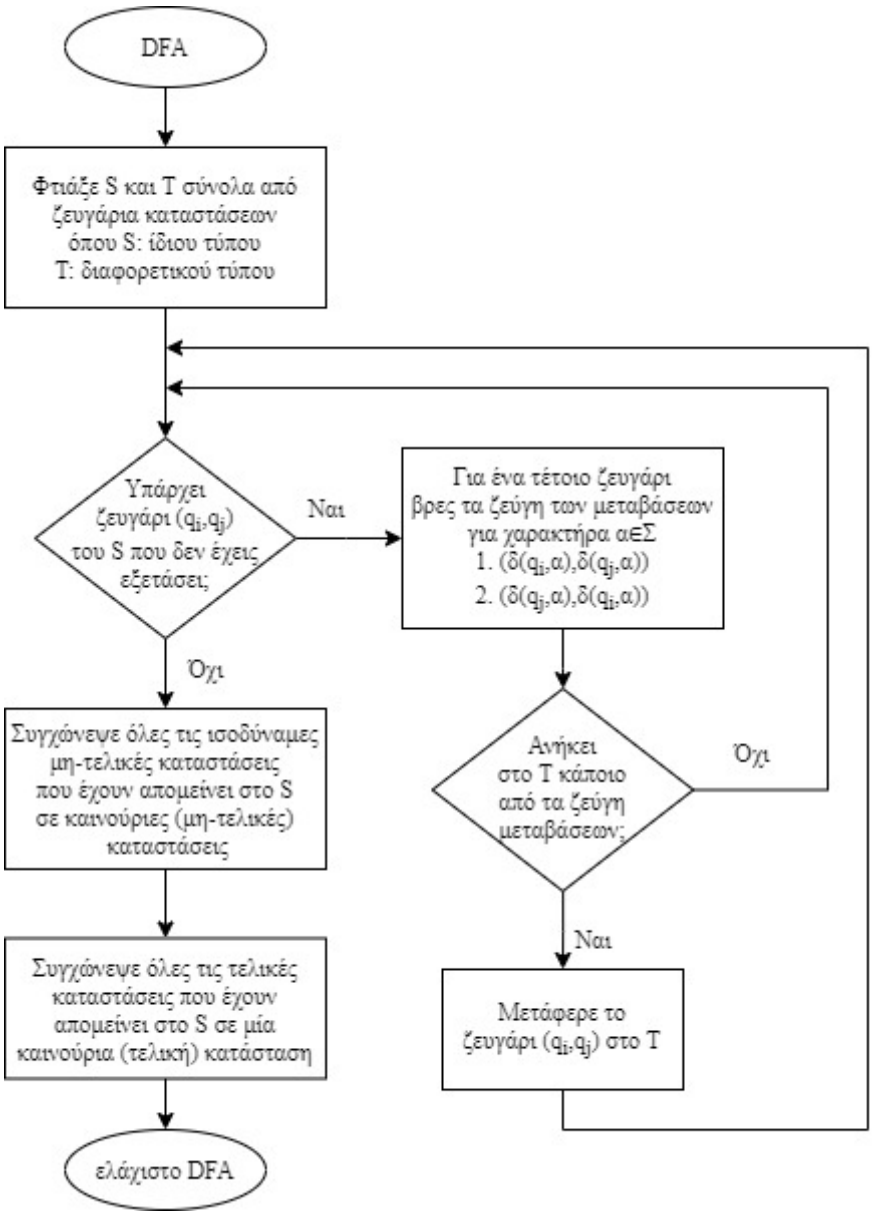
Δηλαδή αν θεωρήσουμε δύο τύπους καταστάσεων, τελικές και μη-τελικές, μπορούμε να ορίσουμε τα σύνολα ως εξής:

$$S = \{ (q_i, q_j) \mid i < j \text{ και } q_i, q_j \text{ ίδιου τύπου} \}$$
$$T = \{ (q_i, q_j) \mid i < j \text{ και } q_i, q_j \text{ διαφορετικού τύπου} \}$$

Βήμα 2: Για κάθε ζευγάρι (q_i, q_j) του συνόλου S, αν το ζευγάρι $\left(\delta(q_i, a), \delta(q_j, a)\right)$ ή $\left(\delta(q_j, a), \delta(q_i, a)\right)$ για κάποιο χαρακτήρα $a \in \Sigma$ ανήκει στο σύνολο T, τότε μετάφερε το ζευγάρι (q_i, q_j) από το σύνολο S στο σύνολο T.

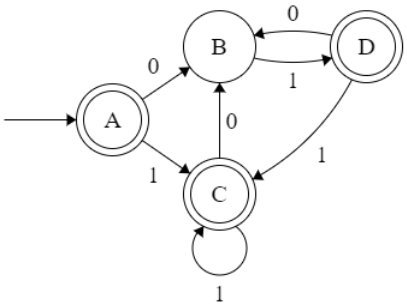
Βήμα 3: Τα ζευγάρια που έχουν απομείνει στο σύνολο S είναι ζευγάρια ισοδύναμων καταστάσεων. Συγχώνεψε όλες τις μη-τελικές καταστάσεις σε μια καινούρια κατάσταση και όλες τις τελικές καταστάσεις σε μια άλλη.

Ένα διάγραμμα ροής του αλγορίθμου αυτού φαίνεται στο Σχήμα 4.4. [1]



Σχήμα 4.4: Διάγραμμα ροής ελαχιστοποίησης DFA

Παράδειγμα:



Για αυτό το DFA βρίσκουμε:

$S = \{(A, C), (A, D), (C, D)\}$

$T = \{(A, B), (B, C), (B, D)\}$

$$(A, C) \rightarrow \begin{cases} \delta(A, 0) = B \\ \delta(C, 0) = B \end{cases} \rightarrow (B, B) \notin T$$

$$\rightarrow \begin{cases} \delta(A, 1) = C \\ \delta(C, 1) = C \end{cases} \rightarrow (C, C) \notin T$$

$$(A, D) \rightarrow \begin{cases} \delta(A, 0) = B \\ \delta(D, 0) = B \end{cases} \rightarrow (B, B) \notin T$$

$$\rightarrow \begin{cases} \delta(A, 1) = C \\ \delta(D, 1) = C \end{cases} \rightarrow (C, C) \notin T$$

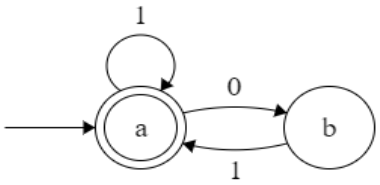
$$(C, D) \rightarrow \begin{cases} \delta(C, 0) = B \\ \delta(D, 0) = B \end{cases} \rightarrow (B, B) \notin T$$

$$\rightarrow \begin{cases} \delta(C, 1) = C \\ \delta(D, 1) = C \end{cases} \rightarrow (C, C) \notin T$$

Τελικό σύνολο: $S = \{(A, C), (A, D), (C, D)\}$

Άρα οι τελικές καταστάσεις A, C, D είναι ισοδύναμες και συγχωνεύονται στην κατάσταση a.

Έτσι, το ελάχιστο DFA έχει τις καταστάσεις $a = \{A, C, D\}$ και $b = \{B\}$ και είναι το εξής:



Σχήμα 4.5: Το ελάχιστο DFA που αντιστοιχεί στην κανονική έκφραση (01|1)*

5. Έλεγχος Ορθότητας

Έλεγχος ορθότητας (testing) είναι η διαδικασία εκτέλεσης ενός προγράμματος με σκοπό να βρεθούν και να διορθωθούν τυχόν λάθη στον κώδικα. Αυτό γίνεται με ένα σύνολο από *δοκιμαστικές εισόδους (test-cases)* που εισάγονται στο πρόγραμμα για να μελετηθεί η συμπεριφορά του.

Ο έλεγχος ορθότητας των προγραμμάτων συντελεί σημαντικό ρόλο στον προγραμματισμό λογισμικού, καθώς εξασφαλίζει την ορθή λειτουργία του προγράμματος με βάση τις απαιτήσεις του προγραμματιστή. Ειδικότερα ο έλεγχος μπορεί να δείξει σε ποιο βαθμό το υπό-έλεγχο πρόγραμμα:

- ικανοποιεί τις προδιαγραφές σύμφωνα με τις οποίες σχεδιάστηκε,
- ανταποκρίνεται σωστά σε κάθε είδος εισόδου,
- εκτελεί τις λειτουργίες του σε αποδεκτό χρονικό διάστημα,
- είναι επαρκώς λειτουργικό,
- μπορεί να εγκατασταθεί και να τρέξει στο προοριζόμενο για αυτό περιβάλλον, και
- πετυχαίνει το γενικό επιθυμητό αποτέλεσμα.

Υπάρχουν δυο βασικοί μέθοδοι ελέγχου ορθότητας: «άσπρο κουτί» (white-box testing) και «μαύρο κουτί» (black-box testing). Στη μέθοδο «άσπρο κουτί» ο ελεγκτής του προγράμματος έχει πρόσβαση στον κώδικα και συνεπώς φτιάχνει τις δοκιμαστικές εισόδους έχοντας γνώση της δομής του προγράμματος και των σχεδιαστικών επιλογών που χρησιμοποιήθηκαν σε αυτό. Αντίθετα, στη μέθοδο «μαύρο κουτί» ο ελεγκτής δεν έχει πρόσβαση στον κώδικα αλλά μόνο στις προδιαγραφές του προγράμματος και συνεπώς ελέγχει κατά κύριο λόγο τη λειτουργικότητα του προγράμματος. Στον έλεγχο ορθότητας χρησιμοποιείται και ένας συνδυασμός των δύο μεθόδων που μπορούμε να αποκαλέσουμε «γκρι κουτί» (grey-box testing), ο οποίος τονίζει τα θετικά σημεία της κάθε μεθόδου ενώ περιορίζει τα μειονεκτήματά τους, προσφέροντας έτσι μια καλή ισορροπία μεταξύ των δύο άκρων.

Ανεξάρτητα από την μέθοδο ελέγχου που θα χρησιμοποιηθεί, όμως, η επιλογή των test-cases είναι εξαιρετικά σημαντικό στάδιο για την αποτελεσματικότητα του ελέγχου ορθότητας. Είτε ο ελεγκτής γνωρίζει ή όχι τον υπό έλεγχο κώδικα, ο έλεγχος δεν μπορεί να είναι επιτυχημένος στην εντόπιση σφαλμάτων αν έχει γίνει κακή επιλογή των test-cases.

Τα test-cases θα πρέπει να είναι τέτοια ώστε να γίνεται εμφανές αν το υπό έλεγχο πρόγραμμα δουλεύει όπως είναι αναμενόμενο. Για να γίνει αυτό θα πρέπει να ελέγχονται όλες οι προδιαγραφές του εκάστοτε ελεγχόμενου προγράμματος, καθώς και όλες οι γνωστές σχεδιαστικές επιλογές που έγιναν στον κώδικα. Θα πρέπει, δηλαδή, όσο αυτό είναι εφικτό, να ελέγχεται κάθε γραμμή του κώδικα.

Επιπλέον, για να γίνει ουσιαστικός έλεγχος ορθότητας θα πρέπει να υπάρχουν και «θετικά» και «αρνητικά» test-cases. *«Θετικές» (positive)* λέγονται οι δοκιμαστικές είσοδοι για τις οποίες αναμένουμε το πρόγραμμα να τρέξει επιτυχώς και να δώσει αντίστοιχη έξοδο. *«Αρνητικές» (negative)* λέγονται οι δοκιμαστικές είσοδοι για τις οποίες αναμένουμε το πρόγραμμα να εμφανίσει μήνυμα σφάλματος, δηλαδή οι δοκιμαστικές είσοδοι που σκοπό έχουν να «σπάσουν» το υπό έλεγχο πρόγραμμα. Άρα για κάθε δοκιμαστική είσοδο που χρησιμοποιούμε στον έλεγχο θα πρέπει να είναι γνωστή και η αντίστοιχη αναμενόμενη συμπεριφορά του προγράμματος: η έξοδος του προγράμματος, αν είναι «θετική» δοκιμαστική είσοδος, ή το μήνυμα σφάλματος, αν είναι «αρνητική».

Μερικοί γενικοί τύποι «θετικών» και «αρνητικών» test-cases φαίνονται στον Πίνακα 5.1.

«Θετικά» test-cases:	«Αρνητικά» test-cases:
• Μέσες καταστάσεις, αναμενόμενες τιμές • Ελάχιστη κανονική κατάσταση • Μέγιστη κανονική κατάσταση • Συμβατότητα με παλιά δεδομένα	• Πολύ λίγα ή καθόλου δεδομένα • Πάρα πολλά δεδομένα • Λάθος τύπου (άτοπα) δεδομένα • Δεδομένα με λάθος μέγεθος • Μη αρχικοποιημένα δεδομένα

Πίνακας 5.1: Παραδείγματα «θετικών» και «αρνητικών» test-cases

Παρ' όλα αυτά είναι δύσκολο να αποδειχθεί ότι κάποιο κομμάτι κώδικα δεν περιέχει καθόλου λάθη, αφού και μετά από τον πιο εξονυχιστικό έλεγχο παραμένει πάντα η πιθανότητα ο κώδικας να περιέχει λάθη που δεν έχουν εντοπιστεί ακόμα.

Πιο συγκεκριμένα, αν για κάποια είσοδο το υπό έλεγχο πρόγραμμα δεν συμπεριφέρεται όπως θα περιμέναμε, τότε ξέρουμε σίγουρα ότι το πρόγραμμα έχει κάποιο λάθος. Από την άλλη, αν για κάποια είσοδο το πρόγραμμα συμπεριφέρεται σωστά, τότε ξέρουμε με βεβαιότητα μόνο ότι δεν εμφανίζει κανένα σφάλμα για τη συγκεκριμένη είσοδο.

Ο μόνος τρόπος να εξασφαλιστεί ότι κάποιο πρόγραμμα δεν περιέχει καθόλου λάθη είναι να γίνει εξαντλητικός έλεγχος, δηλαδή να τρέξει το πρόγραμμα ελέγχου για κάθε πιθανή τιμή εισόδου και για κάθε πιθανό συνδυασμό αυτών. Γρήγορα γίνεται αντιληπτό ότι ακόμα και για τα μικρότερα προγράμματα κάτι τέτοιο θα ήταν απαγορευτικό λόγω του απολύτου μεγέθους της πληροφορίας. Για το λόγο αυτό είναι απαραίτητη μια κάποια στρατηγική επιλογή των test-cases που να βοηθάει στην εύρεση όσο το δυνατόν περισσότερων σφαλμάτων.

Ιδανικά ο έλεγχος ορθότητας ενός προγράμματος θα πρέπει να γίνεται συχνά μεταξύ των εκδοχών ή/και επεκτάσεών του ώστε τα λάθη να εντοπιστούν και να διορθωθούν όσο το δυνατόν νωρίτερα. Όμως ο έλεγχος είναι συνήθως χρονοβόρα και επίπονη διαδικασία όταν γίνεται με το χέρι. Η συγγραφή νέων test-cases καθώς και η διαδοχική είσοδός τους στο υπό-έλεγχο πρόγραμμα μπορεί να αποτελέσει σημαντική προσπάθεια και να αποθαρρύνει τον ελεγκτή από το συχνό ή/και εκτενή έλεγχο.

Για να αποφευχθεί η προσπάθεια του χειροκίνητου ελέγχου και να προτραπεί ο συχνότερος έλεγχος, είναι απαραίτητη η αυτοματοποίηση του ελέγχου. Τα πλεονεκτήματα που μας παρέχει η αυτοματοποίηση είναι:

- Μικρότερη πιθανότητα σφαλμάτων κατά τον έλεγχο, αφού για παράδειγμα τα test-cases δεν χρειάζεται να συντάσσονται από την αρχή με το χέρι κάθε φορά που θέλουμε να τα τρέξουμε.
- Εύκολη διαθεσιμότητα για να τρέξει ο έλεγχος ξανά και ξανά, μεταξύ εκδοχών του προγράμματος.
- Σύγκριση μεταξύ των αποτελεσμάτων του τρέχοντος ελέγχου με τα αποτελέσματα προηγούμενων ελέγχων, αν χρησιμοποιήσουμε τα ίδια test-cases.
- Πιο γρήγορη διαδικασία ελέγχου και πιθανώς χωρίς την ανάγκη ανθρώπινης επιτήρησης.
- Εύκολη προσαρμογή σε νέα δεδομένα με την αυτοματοποίηση της δημιουργίας καινούριων test-cases.

Η αυτοματοποίηση της δημιουργίας test-cases μπορεί να γίνει με *τυχαίο έλεγχο (random testing),* που είναι μια τεχνική ελέγχου κατά την οποία γίνεται τυχαία επιλογή των test-cases. Ο τυχαίος έλεγχος είναι μια τεχνική συχνά εύκολη στην υλοποίηση και στην

κατανόηση και έχει αποδειχθεί ότι είναι αποτελεσματικός τρόπος ανίχνευσης σφαλμάτων. Γίνεται συνήθως με τη χρήση γεννητριών τυχαίων δεδομένων (random-data generator), για τις οποίες υπάρχουν έτοιμες βιβλιοθήκες σε πολλές γλώσσες προγραμματισμού. Μια τέτοια γεννήτρια είναι σε θέση να παράγει πολλά και τυχαία test-cases γρηγορότερα από έναν άνθρωπο, αλλά και ασυνήθιστα test-cases με συνδυασμούς δεδομένων που πιθανόν ένας άνθρωπος να μη σκεφτόταν.

Ο τυχαίος έλεγχος είναι πολύ χρήσιμος γιατί μπορεί να είναι η μόνη πρακτική επιλογή για περιπτώσεις ελέγχου ορθότητας «μαύρο κουτί», κατά τις οποίες ενδέχεται να μην είναι γνωστές όλες οι προδιαγραφές του υπό έλεγχο προγράμματος. Από την άλλη, συχνά υποστηρίζεται ότι ο τυχαίος έλεγχος είναι μη αποδοτικός, καθώς δεν αξιοποιεί όσες διαθέσιμες πληροφορίες υπάρχουν για το υπό έλεγχο πρόγραμμα. Ωστόσο, ο τυχαίος έλεγχος παραμένει μια πολύ διαδεδομένη τεχνική, αφού μπορεί να χρησιμοποιηθεί μόνος του ή να αποτελεί μέρος κάποιας άλλης τεχνικής ελέγχου. [4] [5] [6] [7] [8]

5.1 Έλεγχος Ορθότητας για Μεταγλωττιστές

Ο έλεγχος ορθότητας των μεταγλωττιστών είναι ιδιαίτερα σημαντικός, αφού από αυτούς εξαρτώνται όλα τα μελλοντικά προγράμματα που θα γραφτούν στην γλώσσα που μεταγλωττίζει.

Ένας μεταγλωττιστής είναι η μοναδική σύνδεση μεταξύ της γλώσσας υψηλού επιπέδου, που χρησιμοποιεί ο άνθρωπος, και της γλώσσας μηχανής, που διαβάζει και τρέχει ο υπολογιστής. Για να μπορεί, λοιπόν, να χρησιμοποιηθεί με κάποιο νόημα αυτή η γλώσσα υψηλού επιπέδου πρέπει να εξασφαλιστεί η ορθότητα του μεταγλωττιστή.

Ο έλεγχος ενός μεταγλωττιστή γίνεται με τη χρήση μιας σειράς προγραμμάτων στην πηγαία γλώσσα η οποία ονομάζεται *σύνολο προγραμμάτων ελέγχου (test suite)*. Αυτά τα προγράμματα τροφοδοτούνται στο μεταγλωττιστή για να μελετηθεί η συμπεριφορά του. Όπως και με τα test-cases παραπάνω, για να γίνει ουσιαστικός έλεγχος ενός μεταγλωττιστή η test suite θα πρέπει να περιλαμβάνει και «θετικά» και «αρνητικά» προγράμματα.

Αν το πρόγραμμα ελέγχου είναι «θετικό», τότε αναμένεται από το μεταγλωττιστή η δημιουργία εκτελέσιμου προγράμματος, το οποίο στη συνέχεια εκτελείται και για το οποίο αξιολογείται η λογική του ορθότητα. Αν είναι «αρνητικό» τότε αναμένεται η εμφάνιση των αντίστοιχων μηνυμάτων σφάλματος. Στην περίπτωση που για κάποια είσοδο η συμπεριφορά του μεταγλωττιστή αποκλίνει από την προβλεπόμενη, τότε θεωρείται ότι ο μεταγλωττιστής έχει κάποιο σφάλμα στον κώδικά του.

Είναι προφανές ότι πρέπει να δοκιμαστεί μεγάλο εύρος προγραμμάτων για να ελεγχθούν όλα τα χαρακτηριστικά μιας πηγαίας γλώσσας, δηλαδή όλοι οι λεκτικοί, συντακτικοί και σημασιολογικοί κανόνες, καθώς και όλα τα μηνύματα σφάλματος που μπορούν να εμφανιστούν κατά τη λανθασμένη χρήση της.

Συνεπώς για να γίνει λίγο πιο εύκολη και σύντομη η διαδικασία του ελέγχου μεταγλωττιστών, είναι χρήσιμο να υπάρχει ένα πρόγραμμα που να παράγει αυτόματα τυχαία test-cases με βάση τις προδιαγραφές της πηγαίας γλώσσας. Με τη δημιουργία ενός τέτοιου προγράμματος ασχολούμαστε σε αυτή την εργασία. [1] [4]

6. Property-based Testing και Βιβλιοθήκη Hypothesis

Ο *έλεγχος ορθότητας με βάση ιδιότητες (property-based testing)* είναι μια τεχνική τυχαίου ελέγχου ορθότητας, η οποία έγινε γνωστή από τη βιβλιοθήκη QuickCheck της γλώσσας Haskell. Όπως προδίδει και το όνομά του, εξετάζει αν για ένα υπό έλεγχο πρόγραμμα ισχύουν κάποιες ιδιότητες που σχετίζονται με την αποτελεσματικότητα και τη χρησιμότητα αυτού του προγράμματος.

Στον παραδοσιακό έλεγχο ορθότητας η συμπεριφορά ενός προγράμματος ορίζεται με παραδείγματα, που ο προγραμματιστής-ελεγκτής πρέπει να συγγράψει με το χέρι. Κάθε παράδειγμα αποτελείται από την είσοδο που θα δοθεί στο πρόγραμμα καθώς και την αναμενομένη συμπεριφορά του προγράμματος για αυτή την είσοδο.

Αντίθετα, στον έλεγχο με βάση τις ιδιότητες αρκεί να προσδιοριστούν τα κοινά χαρακτηριστικά όλων των έγκυρων εισόδων που μπορούν να δοθούν στο πρόγραμμα, καθώς και οι ιδιότητες (εγγυήσεις) που πρέπει να ισχύουν για κάθε έξοδο του προγράμματος όταν σε αυτό δίνονται έγκυρες είσοδοι. Στη συνέχεια, παράγονται τυχαία δεδομένα εισόδου με αυξανομένη πολυπλοκότητα, τα οποία τροφοδοτούνται στο πρόγραμμα, και ελέγχεται αν για τις αντίστοιχες εξόδους ισχύουν οι ιδιότητες-εγγυήσεις.

Είναι προφανές ότι η δουλειά του προγραμματιστή-ελεγκτή μειώνεται δραματικά στη δεύτερη περίπτωση, αφού μεγάλο μέρος του ελέγχου γίνεται αυτοματοποιημένα από το εργαλείο ελέγχου με βάση τις ιδιότητες. [9] [10]

Ένα εργαλείο τέτοιου ελέγχου θα χρησιμοποιήσουμε σε αυτή την εργασία για τον αυτόματο έλεγχο κανονικών εκφράσεων.

6.1 Εισαγωγή στη Βιβλιοθήκη Hypothesis

Η βιβλιοθήκη Hypothesis είναι μια βιβλιοθήκη ελέγχου ορθότητας με βάση ιδιότητες για τη γλώσσα Python.

Πιο συγκεκριμένα, η βιβλιοθήκη Hypothesis παράγει τυχαία δεδομένα εισόδου με τη βοήθεια στρατηγικών (strategies). Τα δεδομένα αυτά περνούν στο πρόγραμμα ελέγχου μέσω του διακοσμητή `@given`, ο οποίος τοποθετείται πριν από τη δήλωσή του. Στη συνέχεια το πρόγραμμα ελέγχου τροφοδοτεί τα δεδομένα στο υπό έλεγχο πρόγραμμα. Με τη χρήση της εντολής “assert” (επίβαλε) επιβάλλονται οι ιδιότητες-εγγυήσεις που πρέπει να ισχύουν για την έξοδο του υπό έλεγχο προγράμματος.

Κατά την εκτέλεση του ελέγχου η βιβλιοθήκη Hypothesis προσπαθεί να βρει παραδείγματα έγκυρων εισόδων, στα οποία οι αντίστοιχες έξοδοι δεν ικανοποιούν τις ιδιότητες-εγγυήσεις. Αν βρει ένα τέτοιο παράδειγμα, η βιβλιοθήκη μειώνει το μέγεθός του, απλοποιώντας το μέχρι να βρει ένα πολύ μικρότερο παράδειγμα που να προκαλεί το ίδιο πρόβλημα. Με αυτό τον τρόπο είναι πιο εύκολο για τον προγραμματιστή-ελεγκτή να διαβάσει το παράδειγμα και να βρει το αντίστοιχο σφάλμα στον κώδικα.

Επιπλέον, στην περίπτωση πολλαπλών ελέγχων στο ίδιο κομμάτι κώδικα, η βιβλιοθήκη Hypothesis αποθηκεύει και χρησιμοποιεί ξανά τα παραδείγματα που εμφάνισαν πρόβλημα στο παρελθόν για να ελέγξει ότι τα αντίστοιχα σφάλματα στον κώδικα έχουν σίγουρα διορθωθεί. [11]

Παράδειγμα:

Έστω η συνάρτηση *power* που βρίσκει τη y-οστή δύναμη του x:

```
def power(x, y):
    p = x
    for i in range(2, y):
        p *= x
    return p
```

Και η συνάρτηση ελέγχου ορθότητας για τη συνάρτηση *power*:

```
@given(x=st.integers(), y=st.integers())
def test_power(x, y):
    a = power(x, y)
    assert x**y == a
```

Σε αυτή τη συνάρτηση φαίνεται ότι θεωρούμε ότι οι είσοδοι x και y πρέπει να είναι ακέραιοι και το αποτέλεσμα της συνάρτησης *power* να είναι x^y.

Όταν τρέξουμε τη συνάρτηση ελέγχου θα λάβουμε το παρακάτω μήνυμα από τη βιβλιοθήκη Hypothesis:

```
Falsifying example: test_power(x=0, y=0)
```

Παρατηρούμε έτσι ότι η συνάρτηση *power* δεν δουλεύει για ύψωση στη μηδενική δύναμη. Επεξεργαζόμαστε τη συνάρτηση:

```
def power(x, y):
    p = 1
    for i in range(y):
        p *= x
    return p
```

Όταν τρέξουμε τη καινούρια συνάρτηση *power* δεν εμφανίζεται μήνυμα από τη βιβλιοθήκη Hypothesis. Αυτό σημαίνει ότι η συνάρτηση δεν εμφάνισε σφάλμα.

6.2 Σύνθετες Στρατηγικές

Στη βιβλιοθήκη Hypothesis υπάρχουν πολλές έτοιμες γεννήτριες δεδομένων για τους βασικούς τύπους δεδομένων, όπως αριθμούς (ακέραιους, δεκαδικούς, δυαδικούς κλπ), χαρακτήρες, λίστες, email και ημερομηνίες. Αυτές οι γεννήτριες ονομάζονται βασικές στρατηγικές (core strategies) και είναι χρήσιμες για την παραγωγή των αντίστοιχων βασικών δεδομένων με μερικές παραμετροποιήσεις.

Κάποια προγράμματα, όμως, απαιτούν πιο περίπλοκα δεδομένα ως είσοδο. Για ένα τέτοιο πρόγραμμα ο προγραμματιστής-ελεγκτής μπορεί να συγγράψει μια συνάρτηση, η οποία να συνδυάζει τις βασικές στρατηγικές έτσι ώστε να παράγει πολύπλοκα, προσαρμοσμένα δεδομένα.

Σε αυτή τη συνάρτηση χρησιμοποιείται ο διακοσμητής `@composite`, ο οποίος τη μετατρέπει σε σύνθετη στρατηγική (composite strategy). Αυτή τη σύνθετη στρατηγική περνάει μετά ο προγραμματιστής-ελεγκτής στο διακοσμητή `@given`. [11]

Παράδειγμα:

Έστω ότι θέλουμε να παράγουμε ταξινομημένες λίστες με μη-αρνητικούς ακεραίους. Γράφουμε την παρακάτω συνάρτηση-γεννήτρια:

```
@st.composite
def sorted_list(draw, elements=st.integers(min_value=0)):
    l = draw(st.lists(elements, max_size=20))
    l.sort()
    return l
```

Τη συνάρτηση-γεννήτρια χρησιμοποιούμε μετά για τον έλεγχο ως εξής:

```
@given(a_list=sorted_list())
def test_list_code(a_list):
    ...
```

7. Υλοποίηση

Σε αυτή τη διπλωματική εργασία κατασκευάζεται ένα εργαλείο για τον έλεγχο ορθότητας ενός λεκτικού αναλυτή. Πιο συγκεκριμένα, το εργαλείο αυτό δέχεται σαν είσοδο μια κανονική έκφραση, την οποία μετατρέπει σε NFA-ϵ, μετά σε DFA και τέλος σε ελάχιστο DFA. Στη συνέχεια, συνδυάζει αυτό το ελάχιστο DFA και τη βιβλιοθήκη property-based testing Hypothesis για να παράγει «θετικά» και «αρνητικά» test-cases για αυτή την κανονική έκφραση. Δηλαδή για δεδομένη κανονική έκφραση παράγονται λέξεις που η κανονική έκφραση αναγνωρίζει και λέξεις που δεν αναγνωρίζει.

Η υλοποίηση γίνεται στη γλώσσα Python, με τη χρήση της βιβλιοθήκης Hypothesis (property-based testing) ως γεννήτρια τυχαίων δεδομένων.

Ολόκληρος ο κώδικας της υλοποίησης διατίθεται ανοικτά στο GitHub, στο σύνδεσμο: https://github.com/athioak7/diplomatiki

7.1 Κλάση Lexer

Ο κώδικάς μας ξεκινάει διαβάζοντας από την είσοδο κανονικές εκφράσεις. Αυτή τη δουλειά κάνει η κλάση `Lexer`, η οποία με τη χρήση της μεθόδου `Lexer.make_tokens()` αναγνωρίζει την κανονική έκφραση χαρακτήρα-χαρακτήρα, και τη μετατρέπει σε μια λίστα από `Tokens`.

Τα `Tokens` που αντιστοιχούν στις κανονικές εκφράσεις φαίνονται στον Πίνακα 7.1.

Κανονική έκφραση	Αντίστοιχο Token	Επεξήγηση
ϵ	`TT_EOF`	κενή συμβολοσειρά / τέλος εισόδου
a	`TT_CHAR`	σύμβολο
rs	`TT_CONCAT`	παράθεση
$r\|s$	`TT_ALT`	διάζευξη
r^*	`TT_STAR`	αστέρι Kleene (0 ή περισσότερες επαναλήψεις)
r^+	`TT_PLUS`	συν (1 ή περισσότερες επαναλήψεις)
(r)	`TT_LPAREN`, `TT_RPAREN`	παρένθεση

Πίνακας 7.1: Tokens σε αντιστοιχία με κανονικές εκφράσεις

Έπειτα, με τη χρήση της μεθόδου `Lexer.get_ab(tokens)` μπορούμε να βρούμε το αλφάβητο της κανονικής έκφρασης, δηλαδή όλους τους χαρακτήρες που αυτή χρησιμοποιεί.

Παράδειγμα:

```
>>> lexer = Lexer("(01|1)*")
>>> tokens = lexer.make_tokens()
>>> print(tokens)
[LPAREN, CHAR:0, CHAR:1, ALT, CHAR:1, RPAREN, STAR]
```

7.2 Κλάση Parser

Ο `Parser` δέχεται ως είσοδο μια λίστα από `Tokens`, την οποία μετατρέπει σε `Nodes` με τη χρήση των κλάσεων `CharNode`, `BinOpNode` και `UnaryOpNode`.

Η γραμματική με βάση την οποία γίνεται αυτή η μετατροπή φαίνεται στον Πίνακα 7.2.

```
expr   : expr ALT term
       | term

term   : term CONCAT factor
       | factor

factor : factor STAR
       | factor PLUS
       | id

id     : CHAR
       | LPAREN expr RPAREN
```

Πίνακας 7.2: Γραμματική κανονικών εκφράσεων

Για κάθε έναν από τους όρους της γραμματικής υπάρχει μέθοδος της κλάσης `Parser` που την αναγνωρίζει και κατασκευάζει τα `Nodes`.

Οι κλάσεις των Nodes είναι οι εξής:

- `CharNode` : για χαρακτήρες
- `BinOpNode` : για δυαδικές πράξεις (alt, concat)
- `UnaryOpNode` : για μοναδιαίες πράξεις (star, plus)

Η μετατροπή από λίστα από Tokens σε εμφωλευμένα Nodes γίνεται με την χρήση της μεθόδου `Parser.parse()`.

Παράδειγμα:

Για την κανονική έκφραση: `(01|1)*`

```
>>> parser = Parser(tokens)
>>> nodes = parser.parse()
>>> print(nodes)
(((0, CONCAT, 1), ALT, 1), STAR)
```

7.3 Κλάση NodeVisitor και Μετασχηματισμός σε NFA-ϵ

Η κλάση `NodeVisitor` επισκέπτεται τα εμφωλευμένα `Nodes` και με βάση αυτά κατασκευάζει ένα NFA-ϵ.

Για την κατασκευή του NFA-ϵ χρησιμοποιεί δύο κλάσεις:

- την κλάση `NFAState`, κλάση παιδί της κλάσης State, που δημιουργεί καταστάσεις, αποθηκεύει τις μεταβάσεις που ξεκινούν από αυτήν και το αν η κατάσταση αυτή είναι τελική, και
- την κλάση `NFAbuilder` που αποθηκεύει την αρχική και την τελική κατάσταση ενός NFA-ϵ.

Για κάθε κλάση `Nodes` υπάρχει και αντίστοιχη μέθοδος `visit`, κάθε μια από τις οποίες δέχεται σαν είσοδο ένα `Node`, μια στοίβα από `NFAbuilder` και μια λίστα με `NFAStates`. Η στοίβα χρησιμοποιείται για την αποθήκευση των επιμέρους `NFAb` μέχρι να χρησιμοποιηθούν για την κατασκευή μεγαλύτερου `NFAb`. Η λίστα χρησιμοποιείται για την καταμέτρηση των έως τώρα κατασκευασμένων `NFAStates`.

Με το πέρας όλων των visit, το `NFAb` που έχει απομείνει στη στοίβα είναι το τελικό αποτέλεσμα. Το αποτέλεσμα αυτό δεν είναι εκτυπώσιμο, γι' αυτό και στο επόμενο κεφάλαιο θα παρουσιάσουμε την κλάση `NFA` για την εκτύπωσή του.

Παράδειγμα:

Για την κανονική έκφραση: `(01|1)*`

```
>>> nfa_stack = []
>>> state_list = []
>>> NodeVisitor().visit(nodes, nfa_stack, state_list)
>>> nfab = nfa_stack.pop()
>>> alphabet = lexer.get_ab(tokens)
>>> final = [str(s) for s in state_list if s.is_end]
```

7.4 Κλάση NFA και Μετασχηματισμός σε DFA

Η κλάση `NFA` χρησιμοποιείται για την αναπαράσταση ενός NFA-ϵ μέσω της οργάνωσης των πληροφοριών που λάβαμε από την κλάση `NodeVisitor` στο προηγούμενο κεφάλαιο, για την ευανάγνωστη εκτύπωσή του, καθώς και για το μετασχηματισμό NFA-ϵ σε DFA.

Τα στοιχεία που χρειάζονται για τον ορισμό ενός στιγμιότυπου της κλάσης `NFA` είναι: οι καταστάσεις με τη μορφή λίστας από `NFAStates`, ένα αλφάβητο με τη μορφή λίστας χαρακτήρων, η αρχική κατάσταση και η λίστα με τις τελικές καταστάσεις.

Ο μετασχηματισμός NFA-ϵ σε DFA γίνεται με τη μέθοδο `NFA.NFAtoDFA()`, με βάση τον αλγόριθμο που περιγράφηκε στο κεφάλαιο 4.2. Για το μετασχηματισμό χρησιμοποιείται η μέθοδος `NFA.calcEclosure()`, η οποία υπολογίζει το ϵ-closure για κάθε `NFAState`. Το αποτέλεσμα του μετασχηματισμού είναι φυσικά στιγμιότυπο της κλάσης `DFA` που θα παρουσιάσουμε στο επόμενο κεφάλαιο.

Παράδειγμα:

Για την κανονική έκφραση: `(01|1)*`

```
>>> nfa = NFA(state_list, alphabet, nfab.start, final)
>>> print(nfa)
        0       1       ε       -|      |-
s0      s1              []
s1                      [s2]
s2              s3      []
s3                      [s7]
s4              s5      []
s5                      [s7]
```

```
s6                      [s0, s4]
s7                      [s9, s6]
s8                      [s6, s9]      yes
s9                      []      yes
```

7.5 Κλάση DFA και Ελαχιστοποίηση

Η κλάση `DFA` χρησιμοποιείται για την αναπαράσταση ενός DFA, για την ευανάγνωστη εκτύπωσή του, καθώς και για την παραγωγή test-cases με βάση αυτό το DFA.

Για την κλάση αυτή χρησιμοποιείται η κλάση `DFAState`, κλάση παιδί της κλάσης `State`. Αντίστοιχα με την κλάση `NFAState` που είδαμε νωρίτερα, η κλάση `DFAState`: δημιουργεί καταστάσεις, αποθηκεύει τις μεταβάσεις που ξεκινούν από αυτήν και το αν η κατάσταση αυτή είναι τελική, καθώς επίσης και μια λίστα με τα ονόματα των States με βάση τα οποία δημιουργήθηκε.

Τα στοιχεία που χρειάζονται για τον ορισμό ενός στιγμιότυπου της κλάσης `DFA` είναι: οι καταστάσεις με τη μορφή λίστας από `DFAStates`, ένα αλφάβητο με τη μορφή λίστας χαρακτήρων, η αρχική κατάσταση και η λίστα με τις τελικές καταστάσεις. Επίσης κατά τη δημιουργία ενός στιγμιότυπου της κλάσης `DFA` κατασκευάζονται και τρία λεξικά (dictionary): ένα για όλες τις μεταβάσεις του DFA, ένα για όλες τις έγκυρες εισόδους για κάθε μια από τις καταστάσεις του και ένα για όλες τις άκυρες εισόδους για κάθε μια από τις καταστάσεις του.

Η ελαχιστοποίηση του DFA γίνεται με τη μέθοδο `DFA.minimize()`, με βάση τον αλγόριθμο που περιγράφηκε στο κεφάλαιο 4.3. Το αποτέλεσμα αυτού του μετασχηματισμού είναι ένα καινούριο στιγμιότυπο της κλάσης `DFA`.

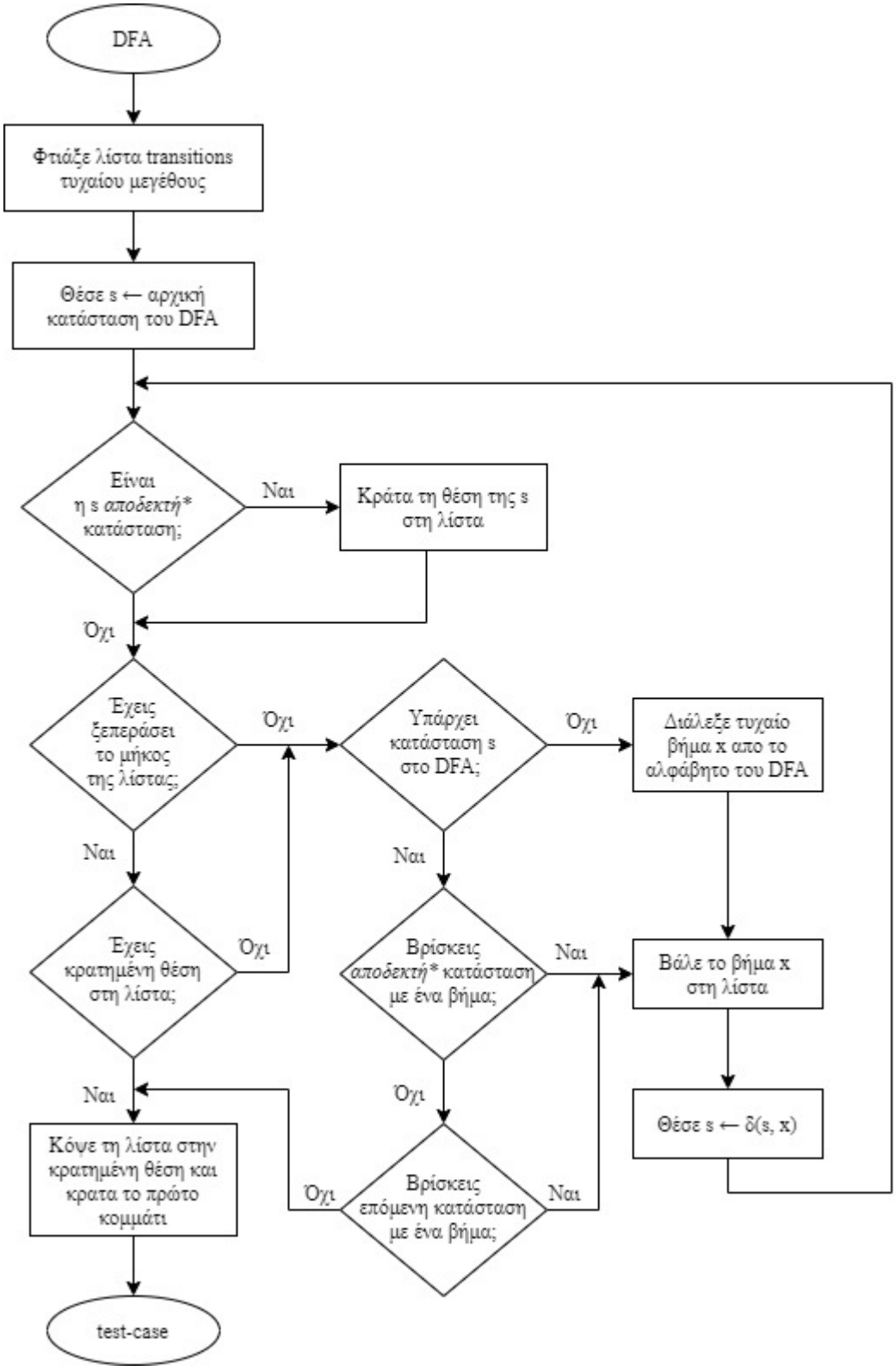
Η κλάση DFA έχει δύο ακόμη μεθόδους, τη μέθοδο `DFA.accept()` και τη μέθοδο `DFA.generate()`, οι οποίες θα αναλυθούν στο επόμενο κεφάλαιο.

Παράδειγμα:

Για την κανονική έκφραση: `(01|1)*`

```
>>> dfa = nfa.NFAtoDFA()
>>> print(dfa)
        0       1       -|      |-
t0      t1      t2      yes     yes
t1              t3
t2      t1      t2      yes
t3      t1      t2      yes
>>> dfa_min = dfa.minimize()
>>> print(dfa_min)
        0       1       -|      |-
q0      q1      q0      yes     yes
q1              q0
```

7.6 Παραγωγή Τυχαίων Test-cases



Σχήμα 7.1: Διάγραμμα ροής της μεθόδου DFA.generate()

* «Αποδεκτές» καταστάσεις για την παραγωγή «θετικών» test-cases είναι οι τελικές καταστάσεις, ενώ για την παραγωγή «αρνητικών» test-cases είναι οι μη-τελικές καταστάσεις.

Για την παραγωγή των test-cases χρησιμοποιείται η μέθοδος `DFA.generate()`, η οποία μπορεί να παράγει ένα «θετικό» ή ένα «αρνητικό» test-case.

Η παραγωγή του test-case γίνεται ξεκινώντας από την αρχική κατάσταση του DFA και διατρέχοντας τις καταστάσεις με τυχαίες μεταβάσεις. Η διαδικασία αυτή αποθηκεύεται σε μια λίστα, για την οποία η μέθοδος διατηρεί έναν δείκτη που δείχνει στην τελευταία θέση ενός αποδεκτού test-case.

Ένα διάγραμμα ροής της μεθόδου `DFA.generate()` φαίνεται στο Σχήμα 7.1.

Η επιλογή μεταξύ παραγωγής «θετικού» ή «αρνητικού» test-case γίνεται με τη χρήση της μεταβλητής boolean `valid`, την οποία δέχεται ως παράμετρο. Δηλαδή για `valid = True` παράγει ένα «θετικό» test-case, ενώ για `valid = False` παράγει ένα «αρνητικό» test-case.

Αυτή η μεταβλητή χρησιμοποιείται για να καθοριστούν ποιες καταστάσεις του DFA θεωρούνται αποδεκτές καταστάσεις, δηλαδή καταστάσεις για τις οποίες ο αλγόριθμος θεωρεί ότι έχει βρει αποτέλεσμα και σταματάει. Πιο συγκεκριμένα, για την παραγωγή «θετικού» test-case, αποδεκτές καταστάσεις είναι οι τελικές καταστάσεις του DFA, ενώ για παραγωγή «αρνητικό» test-case, αποδεκτές καταστάσεις είναι οι μη-τελικές. Είναι εύκολα κατανοητό ότι, στη δεύτερη περίπτωση, αν το DFA κατά την εκτέλεσή του καταλήξει σε μη τελική κατάσταση, η λέξη που παράχθηκε δεν θα αναγνωρίζεται από αυτό και άρα θα είναι «αρνητικό» test-case.

Η μέθοδος `DFA.generate()` καλείται από τις συναρτήσεις `positive()` και `negative()`, οι οποίες παράγουν «θετικά» και «αρνητικά» test-cases αντίστοιχα, και συνεπώς καλούν την `generate()` με τις ανάλογες τιμές για το `valid`, `True` και `False`. Έπειτα, με τη χρήση της βιβλιοθήκης Hypothesis, οι συναρτήσεις αυτές μετατρέπονται σε σύνθετες στρατηγικές, όπως περιγράφεται στο κεφάλαιο 6.2.

Οι στρατηγικές αυτές μετά καλούνται από τη συνάρτηση `test()`, η οποία επιβεβαιώνει το αποτέλεσμα με τη βοήθεια της μεθόδου `DFA.accepts()`. Αυτή ελέγχει αν το αποτέλεσμα αναγνωρίζεται από το DFA και κατά συνέπεια από την αρχική κανονική έκφραση. Τέλος, τυπώνονται τα αποτελέσματα και υπολογίζονται μερικά σχετικά στατιστικά.

Παράδειγμα:

Για την κανονική έκφραση: `(01|1)*`

```
>>> test(dfa_min)
Some positive test-cases:
: 2 times
01: 4 times
0101: 2 times
010101: 2 times
01010101: 1 times
010101011: 1 times
01010101101: 1 times
010101101110101: 1 times
010101111: 1 times
01011: 2 times
0101101: 2 times
01011010111: 1 times
```

```
0101101101: 2 times
010111: 1 times
011: 2 times

...

11101: 1 times
1110101: 1 times
111011: 1 times
1111: 2 times
111101: 1 times
11110101: 1 times
111101011011: 1 times
11111: 1 times
111111: 1 times
100 test-case(s) generated
73 unique test-case(s)
26 max length
6.72 avg length

Some negative test-cases:
#: 2 times
0: 8 times
0#: 1 times
00: 2 times
01#: 1 times
010: 2 times
010#: 1 times
0100: 1 times
0101#: 2 times
01010: 1 times
010100: 2 times
010101#: 1 times

...

11010100: 1 times
11010110101101011010110110 10: 1 times
111#: 1 times
1110: 1 times
1110#: 1 times
11100: 1 times
1110101#: 1 times
11101101#: 1 times
11110111101101111111011#: 1 times
100 test-case(s) generated
76 unique test-case(s)
28 max length
6.23 avg length
```

7.7 Συνολικό Πρόγραμμα

Το συνολικό πρόγραμμα τρέχει εκτελώντας το αρχείο `generate.py`.

Το πρόγραμμα περιμένει είσοδο από το χρήστη, η οποία πρέπει να είναι μια έγκυρη κανονική έκφραση. Σε αυτή την υλοποίηση «έγκυρες» λέγονται οι κανονικές εκφράσεις που χρησιμοποιούν τους αριθμούς 0 έως 9, τα μικρά γράμματα του λατινικού αλφάβητου και τους τελεστές με τη σωστή χρήση τους, όπως παρουσιάζονται στο κεφάλαιο 7.2.

Όταν το πρόγραμμα λάβει έγκυρη κανονική έκφραση, τυπώνει το ισοδύναμο ελάχιστο DFA και τα «θετικά» και «αρνητικά» test-cases, όπως φαίνεται στο παράδειγμα του προηγούμενου κεφαλαίου.

Για να σταματήσει να τρέχει το πρόγραμμα αρκεί ο χρήστης να δώσει σαν είσοδο τη λέξη `"EXIT"`.

7.8 Έλεγχος του Κώδικα

Για τον έλεγχο του κώδικα χρησιμοποιείται η συνάρτηση `gen_regex()` που παράγει μια τυχαία κανονική έκφραση. Αυτή η συνάρτηση μετατρέπεται σε σύνθετη στρατηγική με τη χρήση της βιβλιοθήκης Hypothesis, όπως περιγράφεται στο κεφάλαιο 6.2.

Στη συνέχεια, η συνάρτηση `test_test()` καλεί αυτή τη στρατηγική και οι παραγόμενες κανονικές εκφράσεις τροφοδοτούνται στις συναρτήσεις `positive()` και `negative()`. Τα παραγόμενα test-cases ελέγχονται με τη χρήση της μεθόδου `re.match()`, η οποία υπολογίζει αν μια δεδομένη κανονική έκφραση αναγνωρίζει μία λέξη. Αυτή η μέθοδος ανήκει στη βιβλιοθήκη `re`, που είναι η ένθετη βιβλιοθήκη για κανονικές εκφράσεις στη γλώσσα Python.

Η συνάρτηση `test_test()` τυπώνει τις κανονικές εκφράσεις που χρησιμοποιούνται για τον έλεγχο και, αν δεν εμφανιστεί μήνυμα σφάλματος, στο τέλος τυπώνει μήνυμα «επιτυχίας».

Ο κώδικας ελέγχου τρέχει εκτελώντας το αρχείο `testing.py`.

Με τη βοήθεια του κώδικα ελέγχου διορθώθηκαν λάθη στο πρόγραμμα που δεν εντοπίστηκαν με παραδοσιακό έλεγχο. Το γεγονός αυτό αποδεικνύει την σημασία και την αναγκαιότητα του ελέγχου ορθότητας, και ιδίως του ελέγχου ορθότητας με βάση τις ιδιότητες.

Παράδειγμα:

```
>>> test_test()
0
0
00000000
wrxzh+2w
0
00000000
0000
(0c)*7
```

```
65co
r0000000
r0z*|4y2
0z*|4y2c
stux
j67
6j6
666

...

7|l6d
(l)+d
l|(l)
l|lf1
lf|lf
l*|lf
70|q88
80|q88
p*yx9
x*yx9
x*xx9
xxxx9
Passed all tests!
```

8. Σχετική Δουλειά

Ο έλεγχος ορθότητας είναι ένα πολύ σημαντικό κομμάτι του προγραμματισμού για να εξασφαλισθεί ότι ένα πρόγραμμα –και πιο συγκεκριμένα ένας μεταγλωττιστής– δουλεύει όπως είναι αναμενόμενο σε ικανοποιητικό βαθμό. Ως εκ τούτου είναι κατανοητό ότι το θέμα της παρούσας εργασίας έχει απασχολήσει και άλλους ακαδημαϊκούς. Στο κεφάλαιο αυτό θα ασχοληθούμε με μελέτες που έχουν γίνει στο παρελθόν πάνω στο θέμα της παραγωγής λέξεων για δεδομένη κανονική έκφραση.

Στην εργασία του, ο McIlroy [12] ασχολείται με την απαρίθμηση όλων των λέξεων που αναγνωρίζονται από μια κανονική έκφραση. Αυτό το υλοποιεί στη γλώσσα Haskell με δύο πολύ διαφορετικούς τρόπους: άμεσα, γράφοντας αλγορίθμους για κάθε λειτουργία της κανονικής έκφρασης, και έμμεσα, μετατρέποντας την κανονική έκφραση σε ένα ισοδύναμο μη-ντετερμινιστικό πεπερασμένο αυτόματο και εκτελώντας το. Στο τέλος της εργασίας, ο McIlroy ελέγχει πόσο αποδοτικός είναι ο κάθε τρόπος και παρατηρεί ότι ο άμεσος τρόπος συχνά παράγει διπλότυπα που απορρίπτει, ενώ ο έμμεσος τρόπος παράγει κάθε λέξη μονάχα μία φορά. Καταλήγει έτσι στο συμπέρασμα ότι, παρότι σε κάποιες περιπτώσεις ο άμεσος τρόπος είναι πιο γρήγορος, ο έμμεσος τρόπος είναι συνολικά πιο αποδοτικός.

Με το θέμα της απαρίθμησης μιας κανονικής έκφρασης ασχολούνται και οι Ackerman και Shallit [13] στην εργασία τους, στην οποία παίρνουν τον αλγόριθμο του Mäkinen [14], τον τροποποιούν και συγκρίνουν την αποδοτικότητα του έναντι άλλων αλγορίθμων.

Οι Radanne και Thiemann [15] παρατήρησαν ότι, ενώ έχουν γίνει πολλές μελέτες σχετικά με την απαρίθμηση των λέξεων που αναγνωρίζονται από μια κανονική έκφραση, δεν υπάρχει μελέτη για την παραγωγή λέξεων που να απορρίπτονται από την κανονική έκφραση. Στην εργασία τους φτιάχνουν ένα εργαλείο, το Regenerate, που παράγει και «θετικές» και «αρνητικές» λέξεις με σκοπό τη χρήση του στον έλεγχο ορθότητας της εντολής `match` για κανονικές εκφράσεις. Η υλοποίηση γίνεται σε δύο γλώσσες, Haskell και OCaml, για εκτεταμένες κανονικές εκφράσεις (με την προσθήκη της τομής και του συμπληρώματος) και με τη χρήση και επέκταση του αλγορίθμου του McIlroy [12]. Το εργαλείο Regenerate, αν και υλοποιήθηκε διαφορετικά, έχει τελικά την ίδια λειτουργία με το εργαλείο φτιάξαμε στην παρούσα εργασία.

Ένα όμοιο εργαλείο φτιάχνουν και οι Larson και Kirk [16] στην εργασία τους. Το εργαλείο αυτό, υλοποιημένο σε Python, αρχικά μετατρέπει την κανονική έκφραση σε μη-ντετερμινιστικό πεπερασμένο αυτόματο και με βάση αυτό παράγει τις λέξεις που αναγνωρίζει η κανονική έκφραση. Έπειτα, παίρνει αυτές τις «θετικές» λέξεις και τις τροποποιεί, είτε προσθέτοντας επαναλήψεις χαρακτήρων είτε αλλάζοντας χαρακτήρες, με σκοπό τη δημιουργία «κακών (evil)» λέξεων που να απορρίπτονται από την κανονική έκφραση. Με αυτό τον τρόπο, αποφεύγει να παραγάγει «αρνητικές» λέξεις από το μηδέν.

Συνοψίζοντας, οι πρώτες ερευνητικές προσπάθειες για την παραγωγή των λέξεων μιας κανονικής έκφρασης έγιναν με τη συγγραφή αλγορίθμων, τους οποίους άλλοι ερευνητές εξέλιξαν και βελτίωσαν. Όμως οι Radanne και Thiemann πήγαν την ιδέα ένα βήμα παρακάτω, εισάγοντας την παραγωγή και «αρνητικών» λέξεων. Την ίδια ιδέα υλοποίησαν διαφορετικά οι Larson και Kirk, χρησιμοποιώντας ένα ισοδύναμο μη-πεπερασμένο αυτόματο για την παραγωγή, όμως, μόνο των «θετικών» λέξεων.

9. Συμπεράσματα

Η παρούσα διπλωματική εργασία εξετάζει τη γενική ιδέα με την οποία ασχολήθηκαν οι έρευνες που παρουσιάστηκαν στο προηγούμενο κεφάλαιο, δηλαδή την τυχαία παραγωγή λέξεων δεδομένης μιας κανονικής έκφρασης. Ωστόσο, εξελίσσει την ιδέα αυτή και αξιοποιεί την αποδοτικότητα των πεπερασμένων αυτομάτων για την παραγωγή και «αρνητικών» λέξεων, κάτι που οι προηγούμενοι ερευνητές δεν έλαβαν υπόψιν.

Ειδικότερα, η κανονική έκφραση μετατρέπεται αρχικά σε μη-ντετερμινιστικό πεπερασμένο αυτόματο, έπειτα σε ντετερμινιστικό πεπερασμένο αυτόματο και τέλος σε ελάχιστο ντετερμινιστικό πεπερασμένο αυτόματο. Αυτό σημαίνει ότι το τελικό αυτόματο που χρησιμοποιείται για την παραγωγή των λέξεων είναι μικρότερο σε μέγεθος, δηλαδή με λιγότερες καταστάσεις και μεταβάσεις.

Επιπλέον, στην παρούσα εργασία, το πεπερασμένο αυτόματο χρησιμοποιείται και για την παραγωγή τόσο των «θετικών» όσο και των «αρνητικών» λέξεων, χωρίς την ανάγκη επιπρόσθετου κώδικα. Αυτό σημαίνει ότι, αφού και στις δύο περιπτώσεις χρησιμοποιείται ο ίδιος κώδικας, ο χρόνος που χρειάζεται για την παραγωγή των «θετικών» και των «αρνητικών» λέξεων είναι ίσος.

Συνεπώς, το πρόγραμμα της παρούσας εργασίας είναι συνολικά πιο αποδοτικό από τις προαναφερθείσες εργασίες.

Η συγκεκριμένη ιδέα της παρούσας διπλωματικής εργασίας δε φαίνεται να υπάρχει στη διαθέσιμη βιβλιογραφία. Επομένως, το πρόγραμμά μας συντελεί μια καινοτομία στην εξέταση της παραγωγής λέξεων για δεδομένη κανονική έκφραση.

9.1 Μελλοντικές Επεκτάσεις

Η αρχική ιδέα αυτής της εργασίας ήταν η υλοποίηση ενός εργαλείου που να παράγει τυχαία test-cases για τον έλεγχο ορθότητας μεταγλωττιστών. Ωστόσο, κατά την συγγραφή της εργασίας παρατηρήθηκε ότι κάτι τέτοιο θα ξεπερνούσε τα όρια μιας διπλωματικής εργασίας. Γι' αυτό και επιλέχθηκε να υλοποιηθεί μόνο το πρώτο κομμάτι αυτού του εργαλείου, δηλαδή ο αυτόματος έλεγχος ενός λεκτικού αναλυτή, με τυχαία παραγωγή λέξεων για δεδομένη κανονική έκφραση.

Μια μελλοντική επέκταση, λοιπόν, της παρούσας διπλωματικής εργασίας είναι η συνέχιση του έργου αυτού, δηλαδή η δημιουργία εργαλείου για τον αυτόματο έλεγχο ενός συντακτικού αναλυτή, ενός σημασιολογικού αναλυτή και τελικά ενός μεταγλωττιστή. Το εργαλείο αυτό θα μπορεί να δέχεται σαν είσοδο όλα τα χαρακτηριστικά μιας πηγαίας γλώσσας, δηλαδή όλους τους λεκτικούς, συντακτικούς και σημασιολογικούς κανόνες, και σαν έξοδο να παράγει «θετικά» και «αρνητικά» προγράμματα στην πηγαία γλώσσα για τον έλεγχο ορθότητας του αντίστοιχου μεταγλωττιστή.

Τέλος, παρότι το πρόγραμμα της παρούσας εργασίας σχεδιάστηκε για τον αυτόματο έλεγχο ορθότητας ενός λεκτικού αναλυτή, η χρήση του μπορεί να γενικευτεί και στην παραγωγή τυχαίων λέξεων κανονικών εκφράσεων για άλλες χρήσεις

Βιβλιογραφία

[1] Ν. Παπασπύρου and Ε. Σκορδαλάκης, Μεταγλωττιστές, Αθήνα: Εκδόσεις Συμμετρία, 2002.

[2] J. Hopcroft, R. Motwani and J. Ullman, Introduction to Automata Theory, Languages, and Computation, Boston: Pearson, 2007.

[3] M. Sipser, Εισαγωγή στη θεωρία υπολογισμού, Ηράκλειο: Πανεπιστημιακές Εκδόσεις Κρήτης, 2015.

[4] S. Boujarwah and K. Saleh, "Compiler test case generation methods: a survey and assessment," *Information and Software Technology,* vol. 39, pp. 617-625, 1997.

[5] C. Rosenthal, L. Hochstein, A. Blohowiak, N. Jones and A. Basiri, Chaos Engineering, O'Reilly, 2017.

[6] S. McConnell, Code Complete, Second Edition, Microsoft Press, 2004.

[7] T. Y. Chen, F.-C. Kuo, R. Merkel and T. H. Tse, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software,* vol. 83, no. 1, pp. 60-66, 2010.

[8] "Software testing," Wikipedia, The Free Encyclopedia, [Online]. Available: https://en.wikipedia.org/w/index.php?title=Software_testing&oldid=1012956455. [Accessed 31 March 2021].

[9] D. MacIver, "In praise of property-based testing," *Increment,* no. 10, August 2019.

[10] A. Löscher and K. Sagonas, "Targeted property-based testing," *ISSTA 2017,* pp. 46-56, 2017.

[11] D. MacIver, "Hypothesis Documentation (Release 6.12.0)," 2021. [Online]. Available: https://hypothesis.readthedocs.io/_/downloads/en/latest/pdf/.

[12] M. McIlroy, "Enumerating the strings of regular languages," *Journal of Functional Programming,* vol. 14, no. 5, pp. 503-518, 2004.

[13] M. Ackerman and J. Shallit, "Efficient enumeration of words in regular languages," *Theoretical Computer Science,* vol. 410, no. 37, pp. 3461-3470, 2009.

[14] E. Mäkinen, "On Lexicographic Enumeration Of Regular And Context-Free Langugages," *Acta Cybernetica,* vol. 13, no. 1, pp. 55-61, 1997.

[15] G. Radanne and P. Thiemann, "Regenerate: A Language Generator for Extended Regular Expressions: with an application to test case generation," *SIGPLAN Not.,* vol. 53, no. 9, pp. 202-214, 2018.

[16] E. Larson and A. Kirk, "Generating Evil Test Strings for Regular Expressions," *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST),* pp. 309-319, 2016.

[17] "py-myopl-code," [Online]. Available: https://github.com/davidcallanan/py-myopl-code. [Accessed 5 July 2021].

[18] "regex," [Online]. Available: https://github.com/xysun/regex. [Accessed 6 July 2021].