Οικονομικό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής
Μεταπτυχιακό Πρόγραμμα Σπουδών

Μεταγλωττιστής κανονικών εκφράσεων σε Java Bytecodes

Υπεύθυνος Καθηγητής: Διομήδης Σπινέλλης

Καρακόιδας Βασίλειος M3020009 bkarak@aueb.gr

Περιεχόμενα

Κατάλογος Πινάκων

Κατάλογος Σχημάτων

Πρόλογος

The investment group eyed the entrepreneur with caution, their expressions flickering from scepticism to intrigue and back again.

"Your bold plan holds promise," their spokesman conceded. "But it is costly and entirely speculative. Our mathematicians mistrust your figures. Why should we entrust our money into your hands? What do you know that we do not?"

"For one thing," he replied, "I know how to balance an egg on its point without outside support. Do you?" And with that, the entrepreneur reached into his satchel and delicately withdrew a fresh hen's egg. He handed over the egg to the financial tycoons, who passed it amongst themselves trying to carry out the simple task. At last they gave up. In exasperation they declared, "What do you ask is impossible! No man can balance an egg on its point."

So the entrepreneur took back the egg from the annoyed businessmen and placed it upon the fine oak table, holding it so that its point faced down. Lightly but firmly, he pushed down on the egg with just enough force to crush in its bottom about half an inch. When he took his hand away, the egg stood there on its own, somewhat messy, but definitely balanced. "Was that impossible?" he asked.

"It's just a trick," cried the businessmen. "Once you know how, anyone can do it."

"True enough," came the retort. "But the same can be said for anything. Before you know how, it seems an impossiblity. Once the way is revealed, it's so simple that you wonder why you never thought of it that way before. Let me show you that easy way, so others may easily follow. Will you trust me?"

Eventually convinced that this entrepreneur might possibly have something to show them, the skeptical venture capitalists funded his project. From the tiny Andalusian port of Palos de Morguer set forth the Niña, the Pinta and the Santa Maria, led by an entreprenuer with a sligthly broken egg and his own ideas: Christopher Columbus.

Many have since followed.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τα ακόλουθα άτομα για την άμεση συνεισφορά τους κατα την διάρκεια εκπόνησης της πτυχιακής μου εργασίας:

Την οικογένεια μου που με στηρίζει όλα τα χρόνια των σπουδών μου και μου παρέχει ένα ήρεμο και ζεστό περιβάλλον.

Του Διομήδη Σπινέλλη (επιβλέποντα καθηγητή) που μου εμπιστεύθηκε το συγκεκριμένο θέμα, που με βοήθησε στα πρώτα βήματα εκπόνησης της πτυχιακής και που μου έδωσε μια απειρία συμβουλών και εργαλείων (βοηθητικών προγραμμάτων, πολλά από αυτά δεν είναι δημοσιευμένα στο ευρύ κοινό :)) κατα την διάρκεια αυτής.

Του Κώστα Σαΐδη που σχολίασε προσεκτικά το κείμενο της πτυχιακής αλλά και τις πρώιμες εκδόσεις του F.i.r.e.

Του Γιώργο Οικονόμου (νουός) όπου βάφτισε του μεταγλώττιστή F.i.r.e., όπως επίσης και για την τεράστια υπομονή του παρά του σχεδόν καθημερινό του βομβαρδισμό από κανονικές εκφράσεις. :)

Του Δημήτριο Λίαππη ο οποίος έθεσε τα καλύτερα επιχειρήματα που έχω ακούσει όταν του έθεσα το θέμα του F.i.r.e. με αποτέλεσμα να οδηγηθώ σε πιο ώριμες αποφάσεις στο σχεδιασμό του, καθώς και στην φιλότιμη προσπάθεια του να μου διορθώσει το παρόν κείμενο (εξαιρούμενο το κείμενο των ευχαριστιών :)).

Του Αντώνιο Πετρόπουλο ο οποίος μου ανέδειξε πολλά σφάλματα στην υλοποίηση και μου έδωσε ιδέες (πολλές φορες μέσω της συζήτησης) για τον τρόπο αντιμετώπισης τους.

Του Γιαννή Οικονόμου για την "κατά λάθος" συνεισφορά του στα περιεχόμενα του πρώτου κεφαλαίου.

Του Χρήστο ΚΚ Λοβέρδο για τις διορθώσεις στις πρώιμες εκδόσεις του κείμενου και για το antlr :).

Τους συμφοιτητές μου στο μεταπτυχιακό (Έτος 2002 - 2003) για τον πολύ όμορφο χρόνο που διανύσαμε μαζί.

Περίληψη

Η καταγωγή των κανονικών εκφράσεων βρίσκεται στην θεωρία αυτομάτων και τυπικών γλωσσών. Αυτά τα δύο πεδία μελετούν υπολογιστικές μεθόδους και τρόπους για να περιγραφούν οι τυπικές γλώσσες. Το 1940 ο Warren McCulloch και ο Walter Pits περιέγραψαν το νευρικό σύστημα μοντελοποιόντας το με την χρήση αυτομάτων. Ο μαθηματικός Stephen Kleene περιέγραψε τα μοντέλα αυτά χρησιμοποιώντας μαθηματικούς συμβολισμούς και τα ονόμασε κανονικά σύνολα (regular sets). Ο Ken Thompson τα υλοποίησε στον κειμενογράφο του qed και αργότερα στο κειμενογράφο ed ο οποίος αποτελούσε μέρος του UNIX. Από τότε οι κανονικές εκφράσεις χρησιμοποιούνται ευρύτατα σε όλα τα UNIX και στις εφαρμογές τους όπως: expr, awk, Emacs, vim, lex και στην Perl [Enc03c].

Οι κανονικές εκφράσεις ανήκουν στον τύπο γραμματικής 3 της ιεραρχίας Chomsky (Chomsky hierarchy). Οι ιεραρχίες Chomsky αποτελούν ένα σύνολο από τάξεις τυπικών γραμματικών (formal grammars) . Με βάση αυτές δημιουργούνται οι τυπικές γλώσσες (formal languages). Η ιεραρχία αυτή περιγράφηκε από τον Chomsky το 1956 [Enc03a].

Η Java ξεκίνησε να αναπτύσσεται από τις αρχές του 1991 από τους Bill Joy, Andy Bechtolsheim, Wayne Rosing, Mike Sheridan, James Gosling και Patrick Naughton. Παρουσιάστηκε επίσημα από την Sun Microsystems το 1995. Από την εμφάνιση της κέρδισε αρκετούς οπαδούς και πλέον είναι μια από τις τρεις επικρατέστερες γλώσσες στην προτίμηση των μηχανικών λογισμικού και των προγραμματιστών.

Με την εμφάνιση της Java άρχισαν να εμφανίζονται οι πρώτες υλοποιήσεις μηχανών κανονικών εκφράσεων. Οι προσπάθειες που έγιναν υλοποιούσαν κυρίως μηχανές που ακολουθούσαν το πρότυπο του POSIX, εκτός από την ORO η οποία υλοποιήθηκε με βάση τις επεκτάσεις τις PERL (παράγραφος 2.1.3). Με την έκδοση 1.4.x η SUN αποφάσισε να συμπεριλάβει μια μηχανή κανονικών εκφράσεων στη κύρια διανομή της πλατφόρμας Java. Η

μηχανή αυτή περιλαμβάνεται στις "βασικές διεπαφές προγραμματισμού" (java.util.regex).

Ο μεταγλωττιστής F.i.r.e. (Fast Implementation of Regular Expressions) είναι κατά βάση μια μηχανή κανονικών εκφράσεων. Σχεδιάστηκε με γνώμονα να πληρεί τις παρακάτω προϋποθέσεις:

1. Μεγαλύτερη αποδοτικότητα εκτέλεσης
2. 100% Συμβατότητα με την αντίστοιχη βιβλιοθήκη της Sun (java.util.regex)
3. Μηχανή βασισμένη σε ντετερμινιστικό αυτόματο (DFA)
4. Υβριδική υλοποίηση
5. 100% υλοποίηση σε Java

Ο μεταγλωττιστής F.i.r.e. υποστηρίζει την ανάπτυξη τμημάτων (modules). Στο κεφάλαιο 5 αναφέρεται ότι μπορούν να προγραμματιστούν δύο κατηγορίες τμημάτων: (1) τα τμήματα δημιουργίας αυτομάτων από κανονική έκφραση και (2) τα τμήματα μεταγλωττιστών σε κώδικα μηχανής.

Τα τμήματα δημιουργίας αυτομάτων είναι υπεύθυνα για την κατασκευή του ντετερμινιστικού αυτόματου από τη κανονική έκφραση.

Τα τμήματα μεταγλωττιστών σε κώδικα μηχανής έχουν ως στόχο την μετατροπή του αυτομάτου στο τελικό περιβάλλον εκτέλεσης πχ. Java Source, JVM bytecode κτλ.

Ο μεταγλωττιστής F.i.r.e. δημιουργήθηκε με κύριο στόχο την συμβατότητα και την ταχύτητα. Για να ελέγξουμε την συμβατότητα του με τα πρότυπα εκτελέσαμε μια σειρά από κανονικές εκφράσεις οι οποίες είχαν ως στόχο να δοκιμάσουν τον μεταγλωττιστή σε ακραίες περιπτώσεις.

Για να συγκρίνουμε την απόδοση της F.i.r.e. επιλέξαμε τις δύο πιο αποδοτικές μηχανές κανονικών εκφράσεων, μέσα από εκείνες που αναλύουμε στο κεφάλαιο 4. Αυτές είναι η μηχανή Automaton και η αντίστοιχη της SUN (java.util.regex).

Executive Summary

Regular expressions are a powerful tool for text parsing and manipulation. Their theoretical background lie with Automata theory and formal languages. These two fields study methods to describe formal languages. In 1940 Warren McCulloch and Walter Pits described the human neural system using automata. The mathematician Stephen Kleene described the proposed models with mathematical notation named "regular sets". Later on Ken Thompson implemented the "regular sets" in his text editor named "qed" and later on "ed" which was part of the UNIX distribution. Since then, regular expressions are widely used in almost all UNIX variants in their standards applications such as expr, awk, Emacs etc.

Regular expressions belong to grammar type 3 of the Chomsky hierarchy. This hierarchy described by Chomsky in 1956 and provided a categorization for grammars that described formal languages.

The development of Java platform began in 1991 from Bill Jony, Andy Bechtolsheim, Wayne Rosing, Mike Sheridan, James Gosling and Patrick Naughton. It was formally introduced to the development community in 1995. Java is now one of the three most popular programming languages (The other two are C and C++).

Since JDK 1.x a lot of implementations of regular expressions variants began to show up. All implementations followed the POSIX standard, except Jakarta ORO which followed the Perl extensions (PCRE). In 2002 SUn microsystems began to distribute a regular expressions engine as part of the standard development kit (J2SDK, J2EE).

Our compiler named F.i.r.e. (Fast Implementation of regular expressions) is a regular expression engine. It conceptual design was made in mind with the following principles:

1. Execution Speed and Faster Search

2. 100% compatibility with SUN regular expression engine

3. DFA engine type

4. Hybrid implementation (DFA capabilities with NFA-like features like grouping parentheses etc.)

5. 100% Java implementation

The F.i.r.e. compiler has modular design. It has two basic module types:

1. Regular expression to DFA modules (F.i.r.e.-DFA)

2. DFA to bytecode (F.i.r.e.-WRITER)

F.i.r.e.-DFA modules are being used to parse a regular expression and create a automato. F.i.r.e.-WRITER uses the produced automato as input and compiles it to bytecode.

As aforementioned, F.i.r.e. designed mainly with compatibility and execution speed in mind. In order to check the compatiblity we run a compatibility test with a standard set of regular expressions designed specifically to test such engines.

To test execution speed we tested F.i.r.e. with two other (the fastest engines in Java) regular expression engine. These are the SUN regular expression engine and Automaton.

Κεφάλαιο 1

Εισαγωγή

"And so it begins."
– Kosh (to Sinclair), "Chrysalis"

1.1 Οριοθέτηση της εργασίας

Οι κανονικές εκφράσεις είναι ένα πάρα πολύ δυνατό εργαλείο για τον χειρισμό δεδομένων. Τα τελευταία χρόνια γνωρίζουν μεγάλη αναγνώριση και πλέον αποτελούν αναπόσπαστο μέρος πολλών γλωσσών προγραμματισμού (Perl, Java, VB.NET, C#, PHP, Python κ.ο.κ) και εφαρμογών (Emacs, vi).

Οι κανονικές εκφράσεις αναπτύσσονται αρκετές δεκαετίες. Εμφανίστηκαν στο λειτουργικό σύστημα UNIX ως λειτουργικό κομμάτι αρκετών εφαρμογών. Σύντομα έτυχαν μεγάλης αποδοχής, με αποτέλεσμα να γραφούν μηχανές κανονικών εκφράσεων σε σχεδόν όλα τα λειτουργικά συστήματα που υπάρχουν.

Ανήκουν σε μια πλειάδα επιστημονικών περιοχών. Η επιστημονική περιοχή στην οποία ανήκουν είναι τα *Διακριτά Μαθηματικά*. Κανείς όμως δεν μπορεί να αμφισβητήσει την σημασία και την ανάπτυξη που έτυχαν λόγω των *μεταγλωττιστών (Compilers)*.

Παρόλο που το υπόβαθρο τους είναι καθαρά θεωρητικό, κανείς δεν μπορεί να αγνοήσει την πρακτική τους σημασία. Ο Thompson το 1968 ανέφερε πρώτος την μεταγλώττιση των κανονικών εκφράσεων σε κώδικα μηχανής για την αναγνώριση κειμένου [Tho68]. Η ανάπτυξη τους συνεχίζεται ακόμα και τώρα, καθώς συνεχώς προτείνονται τρόποι επέκτασης της γραμματικής τους ώστε να γίνουν πιο ευέλικτες και αποδοτικές [CC97].

1.2 Στόχοι

Η παρούσα εργασία αποτελεί την συνέχεια εργασίας εξαμήνου στο μάθημα *Ηλεκτρονικό εμπόριο* , στα πλαίσια του MSc στα πληροφοριακά συστήματα του τμήματος πληροφορικής το ακαδημαϊκό έτος 2002-2003 [ge03]. Η εργασία είχε τίτλο *¨Μεταγλωττιστής κανονικών εκφράσεων σε Java"*, και είχε ως στόχο να αποδείξει αν μπορούσε να υλοποιηθεί μηχανή κανονικών εκφράσεων σε Java που θα είχε καλύτερες επιδόσεις από τις ήδη υπάρχουσες, χρησιμοποιώντας ειδικές τεχνικές της πλατφόρμας Java.

Στα πλαίσια της εργασίας υλοποιήθηκε μια πρότυπη μηχανή κανονικών εκφράσεων (fregex - Fast REGular EXpressions). Οι επιδόσεις της μηχανής κρίθηκαν ικανοποιητικές και δικαιολογούσαν την περαιτέρω έρευνα σε επίπεδο πτυχιακής εργασίας.

Ο στόχος της εργασίας αυτής είναι η υλοποίηση μιας μηχανής κανονικών εκφράσεων που ουσιαστικά θα μεταγλωττίζει την δοσμένη έκφραση σε γλώσσα μηχανής της πλατφόρμας της Java. Επίσης στους στόχους της εργασίας είναι και η σύγκριση με τις υπάρχουσες υλοποιήσεις ώστε να εξαχθούν συμπεράσματα για το εγχείρημα.

1.3 Περίγραμμα εργασίας

Στο **δεύτερο κεφάλαιο** αναλύεται η θεωρητική πλευρά των κανονικών εκφράσεων. Χωρίζεται σε δύο μέρη, στο πρώτο γίνεται μια προσπάθεια ορισμού των κανονικών εκφράσεων και κατάταξη τους στις τυπικές γλώσσες. Στο δεύτερο μέρος του κεφαλαίου αναπτύσσονται μέθοδοι επεξεργασίας των κανονικών εκφράσεων.

Το **κεφάλαιο τρία** αναφέρεται στην γλώσσα προγραμματισμού Java. Πιο συγκεκριμένα προσπαθεί να αναλύσει σε βάθος την αρχιτεκτονική της πλατφόρμας που προτείνει η Java και να εξηγήσει τον τρόπο λειτουργίας της.

Όπως είναι φυσικό η εργασία αυτή δεν είναι η πρώτη που ασχολείται με τις μηχανές κανονικών εκφράσεων σε Java. Στο **κεφάλαιο τέσσερα** γίνεται μια ανασκόπηση των μηχανών κανονικών εκφράσεων που έχουν αναπτυχθεί έως τώρα για την πλατφόρμα της Java.

Στο **κεφάλαιο πέντε** παρατίθεται η αρχιτεκτονική της μηχανής κανονικών εκφράσεων F.i.r.e. (Fast Implementation of Regular Expressions). Στο κε-

φάλαιο αυτό γίνεται μια προσπάθεια να γίνει κατανοητή η φιλοσοφία με βάση την οποία υλοποιήθηκε η μηχανή F.i.r.e.

Το **κεφάλαιο έξι** αναφέρονται θέματα που προέκυψαν κατά την υλοποίηση της μηχανής κανονικών εκφράσεων F.i.r.e.

Στο **κεφάλαιο επτά** αναφέρεται σε θέματα επιδόσεων των μηχανών κανονικών εκφράσεων. Επίσης παρουσιάζεται η πλατφόρμα δοκιμής επιδόσεων που αναπτύχθηκε για να δοκιμαστεί σε πληρότητα η μηχανή F.i.r.e.

Στο **κεφάλαιο οκτώ** συνοψίζονται τα θέματα που αναπτύχθηκαν στα προηγούμενα κεφάλαια και αναφέρονται κάποια συμπεράσματα. Επίσης αναφέρονται τομείς πιθανής βελτίωσης της μηχανής F.i.r.e. καθώς και θέματα για μελλοντική ενασχόληση.

1.4 Μέθοδος γραφής της εργασίας

Για την γραφή του παρόντος παραδοτέου χρησιμοποιήθηκε LaTeX. Πιο συγκεκριμένα χρησιμοποιήθηκαν οι διανομές για Microsoft Windows (MiKTeX-e-TeX 2.4.1398 (2.1) (MiKTeX 2.4)) και Linux (TeX, Version 3.14159 (Web2C 7.4.5). Η γραμματοσειρά που χρησιμοποιήθηκε είναι η kerkis [ST03]. Για την αυτοματοποίηση των διαδικασιών μεταγλώττισης χρησιμοποιήθηκε το make και η Perl. Η οργάνωση της βιβλιογραφίας έγινε με την βοήθεια του bibtex. Για τα σχήματα χρησιμοποιήθηκε το xfig, ενώ για την δημιουργία των αυτομάτων και γράφων το Graphviz. Τα διαγράμματα κλάσης UML έγιναν με την χρήση του UMLGraph [Spi03].

Κεφάλαιο 2

Κανονικές Εκφράσεις

"I am a scientist. Nothing shocks me."
- Indiana Jones, Indiana Jones and the temple of doom

2.1 Ορισμοί

2.1.1 Ιστορική αναδρομή

Η καταγωγή των κανονικών εκφράσεων βρίσκεται στην θεωρία αυτομάτων και τυπικών γλωσσών. Αυτά τα δύο πεδία μελετούν υπολογιστικές μεθόδους και τρόπους για να περιγραφούν οι τυπικές γλώσσες. Το 1940 ο Warren McCulloch και ο Walter Pits περιέγραψαν το νευρικό σύστημα μοντελοποιόντας το με την χρήση αυτομάτων. Ο μαθηματικός Stephen Kleene περιέγραψε τα μοντέλα αυτά χρησιμοποιώντας μαθηματικούς συμβολισμούς και τα ονόμασε κανονικά σύνολα (regular sets). Ο Ken Thompson τα υλοποίησε στον κειμενογράφο του qed και αργότερα στο κειμενογράφο ed ο οποίος αποτελούσε μέρος του UNIX. Από τότε οι κανονικές εκφράσεις χρησιμοποιούνται ευρύτατα σε όλα τα UNIX και στις εφαρμογές τους όπως: expr, awk, Emacs, vim, lex και στην Perl [Enc03c].

2.1.2 Ορισμός των κανονικών εκφράσεων

Οι κανονικές εκφράσεις ανήκουν στον τύπο γραμματικής 3 της ιεραρχίας Chomsky (Chomsky hierarchy). Οι ιεραρχίες Chomsky αποτελούν ένα σύνολο από τάξεις τυπικών γραμματικών (formal grammars) . Με βάση αυτές δημιουργούνται οι τυπικές γλώσσες (formal languages). Η ιεραρχία αυτή περιγράφηκε από τον Chomsky το 1956 [Enc03a].

Ορισμός τυπικών γραμματικών

Μια τυπική γραμματική G αποτελείται από τα ακόλουθα συστατικά στοιχεία [Enc03b]:

1. Ένα πεπερασμένο σύνολο N από μη- τερματικά σύμβολα (nonterminal symbols).
2. Ένα πεπερασμένο σύνολο Σ από τερματικά σύμβολα (terminal symbols) τα οποία δεν ανήκουν στο N.
3. Ένα πεπερασμένο σύνολο από κανόνες παραγωγής (production rules) P.

Παράδειγμα

Ας θεωρήσουμε μια γραμματική G με N = {S,B}, Σ = {a,b,c} και με κανόνες παραγωγής

1. S → aBSc
2. S → abc
3. Ba → aB
4. Bb → bb

και θεωρούμε το σύμβολο S ως σύμβολο εκκίνησης (start symbol). Μερικές από τις συμβολοσείρες που παράγονται είναι:

- S → abc (με την χρήση του $2^{ου}$ κανόνα)
- S → aBSc → aBabcc → aaBbcc → aabbcc (με την χρήση $2^{ου}, 3^{ου}$ και $4^{ου}$ κανόνα).
- S → aBSc → aBaBScc → aBaBabccc → aaBBabccc → aaBaBbccc → aaaBBbccc → aaaBbbccc → aaabbbccc (με την χρήση του $2^{ου}, 3^{ου}$ και $4^{ου}$ κανόνα).

Είναι προφανές ότι οι παραπάνω γραμματική ορίζει την γλώσσα $\{ a^n b^n c^n | n > 0\}$ όπου a^n δηλώνει μια συμβολοσειρά από n a's.

Η ιεραρχία Chomsky

Η ιεραρχία Chomsky αποτελεί μια τυποποίηση των τυπικών γραμματικών οι οποίες δημιουργούν τυπικές γλώσσες [Cho56]. Η ιεραρχία αποτελείται από τα παρακάτω επίπεδα:

- Γραμματικές τύπου-0 (unrestricted grammars): περιλαμβάνουν όλων των τύπων τις τυπικές γραμματικές. Μπορούν να γεννήσουν ακριβώς όλες τις γλώσσες που αναγνωρίζονται από τις μηχανές Turing. Οι παραγόμενες γλώσσες ανήκουν στην κατηγορία των επαναληπτικά αριθμήσιμων (recursively enumerable).

- Γραμματικές τύπου-1 (context-sensitive grammars): Αυτές οι γραμματικές κατασκευάζουν γλώσσες ευπαθούς περικειμένου (context-sensitive environments). Οι γλώσσες που προκύπτουν από αυτές τις γραμματικές μπορούν να αναγνωριστούν από γραμμικά περιορισμένες μη ντετερμινιστικές μηχανές Turing.

- Γραμματικές τύπου-2 (context-free grammars): κατασκευάζουν γλώσσες ελεύθερου περιεχομένου (context-free languages). Αυτές είναι οι γλώσσες που αναγνωρίζονται από μη ντετερμινιστικό pushdown αυτομάτου (non-deterministic pushdown automaton). Οι παραγόμενες γλώσσες αποτελούν την θεωρητική βάση του συντακτικού των περισσοτέρων γλωσσών προγραμματισμού.

- Γραμματικές τύπου-3 (regular grammars): κατασκευάζουν τις κανονικές γλώσσες (regular languages). Αυτές οι γλώσσες μπορούν να αναγνωριστούν από ένα ντετερμινιστικό αυτόματο (finite state automaton). Χρησιμοποιούνται συχνά για να κατασκευάσουν πρότυπα αναζήτησης (search patterns) και την λεκτική δομή των γλωσσών προγραμματισμού.

Οι παραπάνω κανόνες συνοψίζονται στον πίνακα 2.1.

Γραμματική	**Γλώσσα**	**Αυτόματο**
Τύπος-0	Επαναληπτικά αριθμίσημες	Μηχανή Turing
Τύπος-1	Ευπαθούς περιεχομένου	Γραμμικά περιορισμένες μη ντετερμινιστικές μηχανές Turing
Τύπος-2	Ελεύθερου περιεχομένου	Μη ντετερμινιστικό pushdown αυτόματο
Τύπος-3	Κανονικές γλώσσες	Ντετερμινιστικό αυτόματο

Πίνακας 2.1: Η ιεραρχία Chomsky

Κανονικές εκφράσεις

Οι κανονικές εκφράσεις αποτελούνται από σταθερές (constants) και τελεστές (operators) που δηλώνουν μια σειρά παραγόμενων συμβολοσειρών (Strings) και πράξεις πάνω σε αυτές [CC97]. Δοσμένου ενός πεπερασμένου αλφάβητου (finite alhpabet) μπορούν να οριστούν οι παρακάτω σταθερές:

1. Κενό σύνολο (empty set), $\square$ δηλώνοντας το σύνολο $\square$
2. Κενή συμβολοσειρά (empty Strings), ε δηλώνοντας το σύνολο $\{\varepsilon\}$
3. Χαρακτήρας α, α οριζόμενος στο Σ δηλώνοντας το σύνολο $\{\ddot{}\alpha\ddot{}\}$

και οι ακόλουθες πράξεις:

1. Σύνθεση (concatenation), RS δηλώνοντας το σύνολο $\{\alpha|\beta$ με το $\alpha \in R$ και το $\beta \in S\}$.
2. Ένωση (set union), $R \cup S$ δηλώνοντας την ένωση των R και S.
3. Αστέρι Kleene (Kleene Star), R* δηλώνει το σύνολο όλων των συμβολοσειρών που προκύπτουν με σύνθεση μηδέν ή παραπάνω συμβολοσειρών που ανήκουν στο R.

2.1.3 Συντακτικό κανονικών εκφράσεων

Με την πάροδο του χρόνου αναπτύχθηκαν διάφορα σύνολα συντακτικών κανόνων που επέκτειναν τους θεωρητικούς ορισμούς Τα επικρατέστερα από αυτά είναι τα οι παραδοσιακές κανονικές εκφράσεις του UNIX (Traditional UNIX regular expressions), οι μοντέρνες κανονικές εκφράσεις του προτύπου POSIX

Έκφραση	Ταιριάζει	Παράδειγμα
c	Οποιοσδήποτε χαρακτήρας c	a
"s"	Αλφαριθμητικό s	"**"
.	Οποιοσδήποτε χαρακτήρας εκτός του newline	a.*b
^	Αρχή γραμμής	^abc
$	Τέλος γραμμής	abc$
[s]	Οποιοσδήποτε χαρακτήρας στο s	[abc]
[^s]	Οποιοσδήποτε χαρακτήρας εκτός s	[^abc]
r*	Μηδέν ή παραπάνω από r	a*
r+	Ένα ή παραπάνω από r	a+
r?	Μηδέν ή ένα από r	a?
r{m,n}	Από m έως n εμφανίσεις του r	a{1,5}
r_1r_2	r_1 ακολουθώντας το r_2	ab
$r_1\|r_2$	r_1 ή r_2	$a\|b$
(r)	r	$(a\|b)$
r_1/r_2	r_1 ακολουθούμενο από το r_2	abc/123

Πίνακας 2.2: Συντακτικοί κανόνες κανονικών εκφράσεων

(POSIX modern regular expressions) και οι επεκτάσεις της Perl (Perl Compatible Regular Expressions (PCRE). Μια σύνοψη των συντακτικών κανόνων βρίσκεται στον πίνακα 2.2. Στην συνέχεια θα αναλύσουμε τα προαναφερόμενα είδη συντακτικών κανόνων.

Παραδοσιακές κανονικές εκφράσεις του UNIX (Traditional UNIX regular expressions)

Το βασικό συντακτικό των κανονικών εκφράσεων που χρησιμοποιείται στο UNIX έχει αντικατασταθεί πλέον από το αντίστοιχο του POSIX (το οποίο αποτελεί επέκταση του). Παρόλα αυτά μερικές εφαρμογές χρησιμοποιούν ακόμα αυτό το συντακτικό (sed, grep κτλ). Σε αυτό το συντακτικό οι πιο πολλοί χαρακτήρες ταιριάζουν μόνο με τον ευατό τους (π.χ. a ταιριάζει μόνο με το "a" κ.ο.κ.). Υπάρχουν και μερικοί χαρακτήρες για τις εξαιρέσεις. Αυτοί είναι οι παρακάτω (βλ. πίνακα 2.2):

```
. [] [^] ^ \(\) * {x,y}
```

Στο παραδοσιακό σύνολο συντακτικών κανόνων δεν υπάρχει τελεστής για ένωση.

Μοντέρνες κανονικές εκφράσεις του προτύπου POSIX (POSIX modern regular expressions)

Οι κανονικές εκφράσεις του προτύπου POSIX έχουν παρόμοιο συντακτικό με τις αντίστοιχες που ορίστηκαν αρχικά με την εμφάνιση του UNIX. Στην πράξη ισχύουν όλοι οι παραπάνω τελεστές με την προσθήκη κάποιων επιπλέον. Αυτοί είναι (βλ. πίνακα 2.2):

```
+ ? |
```

Είναι σημαντικό να αναφέρουμε ότι ορίζεται πλέον η ένωση με τον τελεστή '.'. Εκτός όμως από καινούργιους τελεστές ορίζονται και κατηγορίες χαρακτήρων ώστε να απλοποιηθεί το συντακτικό των κανονικών εκφράσεων. Αυτές οι κατηγορίες στην πράξη χρησιμεύουν στο να είναι δυνατή η γραφή γενικού τύπου κανονικών εκφράσεων που να έχουν υπόσταση σε άλλες γλώσσες π.χ. στα αγγλικά όλοι οι κεφαλαίοι χαρακτήρες ορίζονται ως [A-Z] ενώ στα ελληνικά ως [Α-Ω], αυτό και στις δύο περιπτώσεις στο POSIX απεικονίζεται ως [:upper:]. Στον πίνακα 2.3 απεικονίζονται όλες οι διαθέσιμες κλάσεις (με τις αντίστοιχες κανονικές εκφράσεις για την αγγλική γλώσσα).

Κλάση	**Κανονική έκφραση**
[:alnum:]	[a-zA-Z0-9]
[:cntrl:]	[\x00-\x1F\x7F]
[:lower:]	[a-z]
[:space:]	[\t\n\x0B\f\r]
[:alpha:]	[a-zA-Z]
[:digit:]	[0-9]
[:print:]	[a-zA-Z0-9!"#$%&'()*,-./:;<=>?@[\]^_`{|}~]
[:upper:]	[A-Z]
[:blank:]	[\t]
[:graph:]	[a-zA-Z0-9!"#$%&'()*,-./:;<=>?@[\]^_`{|}~]
[:punct:]	[!"#$%&'()*,-./:;<=>?@[\]^_`{|}~]
[:xdigit:]	[0-9a-fA-F]

Πίνακας 2.3: Κλάσεις χαρακτήρων του POSIX

Το πρότυπο του POSIX για τις κανονικές εκφράσεις βρίσκεται πλήρες στο κεφάλαιο 9 του The Single UNIX Specification, Version 3 [IG03]. Για λόγους πληρότητας η γραμματική που ορίζεται στο POSIX παρατίθεται στο παράρτημα Α'.

Επεκτάσεις της Perl για τις κανονικές εκφράσεις (Perl Compatible Regular Expressions - PCRE)

Η Perl είναι μια πολύ διαδεδομένη γλώσσα στα UNIX λειτουργικά συστήματα, καθώς και στο προγραμματισμό δυναμικών ιστοσελίδων στο διαδίκτυο [WCO00]. Προτείνει μια επέκταση για το συντακτικό των κανονικών εκφράσεων, ώστε να γίνει δυνατή και η ενεργοποίηση εντολών ταιριάσματος (matching), αντικατάστασης (substitution) και αντικατάστασης με ξένο αλφάβητο (transliteration). Ο καινούριος τρόπος σύνταξης που προτείνει είναι της μορφής {τρόπος χρήσης}/{έκφραση}/{έκφραση}/{διαφοροποιητές}. Η προσπάθεια αυτή επέκτασης του συντακτικού των κανονικών εκφράσεων δεν είναι ευρέως διαδεδομένη, οπότε ως αποδεκτό πρότυπο θεωρείται το αντίστοιχο του POSIX.

Ταίριασμα (Matching m//)

Ερευνά για την ύπαρξή δοσμένης έκφρασής σε δεδομένα και συντάσσεται με τους ακόλουθους (ισοδύναμους) τρόπους.

m/έκφραση/cgimosx
/έκφραση/cgismosx
?έκφραση?cgismosx

Οι διαφοροποιητές (modifiers) αναλύονται στον πίνακα 2.4. Ακολουθεί παράδειγμα χρήσης (απόσπασμα Perl κώδικα):

```
# search for baggins in $shire
if($shire =~ m/baggins/) { ... }
```

Αντικατάσταση (Substitution s///)

Ο τελεστής s/// αντικαθιστά τους χαρακτήρες που ταιριάζουν με την δοθείσα έκφραση με την συμβολοσειρά. Συντάσσεται με τους ακόλουθο τρόπο:

s/έκφραση/συμβολοσειρά αντικατάστασης/egimosx

Όλοι οι διαφοροποιητές (εκτός του /e) έχουν ακριβώς την ίδια χρήση με τους αντίστοιχους του τελεστή m// (βλ. πίνακα 2.4). Ο διαφοροποιητής /e επιτρέπει στην συμβολοσειρά αντικατάστασης να μεταγλωττιστεί πριν αντικατασταθεί (οπότε είναι δυνατή η γραφή κώδικα Perl για δυναμική αντικατάσταση). Ακολουθεί παράδειγμα χρήσης (σε γλώσσα προγραμματισμού Perl):

Διαφοροποιητής	Χρήση
/i	Αγνοεί την χρήση κεφαλαίων - μικρών
/m	Αφήνει το ^ και το $ να ταιριάζει μετά από το \n
/s	Αφήνει την . να ταιριάζει με τους χαρακτήρες νέας γραμμής και αγνοεί το $*
/x	Αγνοεί τους χαρακτήρες κενού (whitespace) και επιτρέπει σχόλια μέσα στην κανονική έκφραση
/o	Μεταγλωττίζει την έκφραση μόνο μια φορά
/g	Βρίσκει όλα τα δυνατά αποτελέσματα στο σύνολο των δεδομένων
/cg	Επιτρέπει τον έλεγχο μετά από μία αποτυχημένη προσπάθεια ταιριάσματος

Πίνακας 2.4: Διαφοροποιητές του m//

```
# make the sequel of the hobbit en passant
($lotr = $hobbit) =~ s/Bilbo/Frodo/g;
```

Αντικατάσταση με ξένο αλφάβητο (Transliteration tr///)

Ο τελεστής tr/// αν και δεν χρησιμοποιεί κανονικές εκφράσεις για την σύνταξή του θεωρείται μέρος των επεκτάσεων που προτείνει η Perl. Συντάσσεται ακολούθως:

tr/λίστα αναζήτησης/λίστα αντικατάστασης/cds

Η χρήση των διαφοροποιητών του τελεστή tr/// αναλύεται στον πίνακα 2.5.

Διαφοροποιητής	Χρήση
/c	Συμπλήρωση της λίστας αναζήτησης
/d	Διαγραφή των χαρακτήρων που δεν αντικαταστάθηκαν
/s	Διαγραφή των διπλά αντικαθιστούμενων χαρακτήρων

Πίνακας 2.5: Διαφοροποιητές του tr///

Ακολουθεί παράδειγμα χρήσης (σε γλώσσα Perl):

```
# just a simple encryption algorithm
$message =~ tr/A-Za-z/N-ZA-Mn-za-m/;
```

2.1.4 Επεξηγήσεις και παραδείγματα χρήσης

Η φιλοσοφία χρήσης των κανονικών εκφράσεων είναι πάρα πολύ απλή. Καθορίζουμε το πρότυπο σύμφωνα με κάποιους βασικούς κανόνες και μετά το δοκιμάζουμε στα δεδομένα μας. Τα πρότυπα κατασκευάζονται με την χρήση χαρακτήρων και κάποιων ειδικών τελεστών. Στην συνέχεια θα αναλύσουμε τους πιο σημαντικούς τελεστές με παραδείγματα, ώστε να γίνει περισσότερο κατανοητή η χρήση τους.

Καθορισμός ομάδων χαρακτήρων

Οι τελεστές [],[^] επιτρέπουν να ομαδοποιήσουμε τους χαρακτήρες που θέλουμε να αναζητήσουμε.

- [a-z] - αναζητεί όλους τους χαρακτήρες που δεν είναι κεφαλαία γράμματα του αγγλικού αλφαβήτου.
- [0-9] - αναζητεί όλα τους αριθμούς (0123456789).
- [a-f] - αναζητεί το υποσύνολο χαρακτήρων abcdef.
- [^a] - αναζητεί όλους τους χαρακτήρες εκτός του a.
- [A-Z/] - αναζητεί όλους τους κεφαλαίους χαρακτήρες και επιπλέον το χαρακτήρα ′/′.

Οι τελεστές * ? + {m,n}

Σε περίπτωση που θέλουμε να αναζητήσουμε χαρακτήρες οι οποίοι επαναλαμβάνονται μπορούμε να χρησιμοποιήσουμε τους τελεστές *,+,? ή {m,n}.

- a+ - Αναζητεί τις συμβολοσείρες a, aa, aaa, aaaa, aaaaa κ.ο.κ.
- a{0,1} - Αναζητεί τις συμβολοσείρες ε και a.
- (abra)+cadabra - Αναζητεί τις συμβολοσείρες abracadabra, abraabracadabra κ.ο.κ.
- [1-9]{1} poulakia? kath(e|on)tai - Αναζητεί όλες τις συμβολοσείρες ’1 poulaki kathetai’ ... ’9 poulakia kathontai’.

Προσπαθώντας να βρούμε μια IP διεύθυνση

Έστω ότι προσπαθούμε να αναζητήσουμε μια IP διεύθυνση σε ένα πλήθος δεδομένων. Η διεύθυνση είναι της μορφής 0-255.0-255.0-255.0-255. Με μια πρώτη προσέγγιση θα μπορούσαμε να γράψουμε την παρακάτω κανονική έκφραση:

```
[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*
```

Με αυτή την κανονική έκφραση σίγουρα θα βρίσκαμε όλες τις IP διευθύνσεις που υπάρχουν στα δεδομένα μας, αλλά μήπως δεν βρίσκουμε μόνο αυτές; Πράγματι, αν εξετάσουμε προσεκτικά την δοσμένη έκφραση θα δούμε ότι ταιριάζει και με δεδομένα όπως 2398429.203492034.23042304.023494230. Το πρόβλημα εντοπίζεται στο [0-9]* μέρος της έκφρασης, οπότε θα προσπαθήσουμε να το περιορίσουμε λίγο. Θα μπορούσαμε να την ανασκευάσουμε σε μια έκφραση της μορφής:

```
[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
```

Τώρα ταιριάζει με δεδομένα της μορφής 0-999.0-999.0-999.0-999. Ας δοκιμάσουμε να το περιορίσουμε λίγο ακόμα. Πάλι το πρόβλημα είναι η έκφραση [0-9]{1,3}. Σε μια πιο προσεγμένη μορφή της θα μπορούσε να γράφει:

```
[1-2]?[0-9]{0,2}\.[1-2]?[0-9]{0,2}\.
[1-2]?[0-9]{0,2}\.[1-2]?[0-9]{0,2}
```

Με την βελτιωμένη έκφραση ταιριάζει με δεδομένα της μορφής 0-299.0-299.0-299.0-299. Για να τη περιορίσουμε άλλο θα πρέπει να γίνει εξαιρετικά πολύπλοκη. Ως επόμενο βήμα βελτίωσης θα μπορούσαμε να μειώσουμε το μέγεθος της. Εύκολα μπορούμε να παρατηρήσουμε ότι τα 3 πρώτα μέρη επαναλαμβάνονται. Όποτε η παραπάνω έκφραση μπορεί να γραφεί:

```
([1-2]?[0-9]{0,2}\.){3}[1-2]?[0-9]{0,2}
```

Ο Jeffrey E.F. Friedl στο βιβλίο του *Mastering Regular Expressions, 2^{nd} Edition*, αναλύει το ίδιο πρόβλημα και καταλήγει στην παρακάτω κανονική έκφραση [Fri02, p.189]:

```
([01]?[0-9][0-9]?|2[0-4][0-9]|25[0-5])\.
([01]?[0-9][0-9]?|2[0-4][0-9]|25[0-5])\.
([01]?[0-9][0-9]?|2[0-4][0-9]|25[0-5])\.
([01]?[0-9][0-9]?|2[0-4][0-9]|25[0-5])
```

Αυτή η κανονική έκφραση είναι αρκετά πιο πολύπλοκη άλλα ερευνά και το ενδεχόμενο κάποια IP διεύθυνση να είναι λάθος.

2.2 Τρόποι επεξεργασίας των κανονικών εκφράσεων

2.2.1 Γενικά για την διαδικασία επεξεργασίας κανονικών εκφράσεων

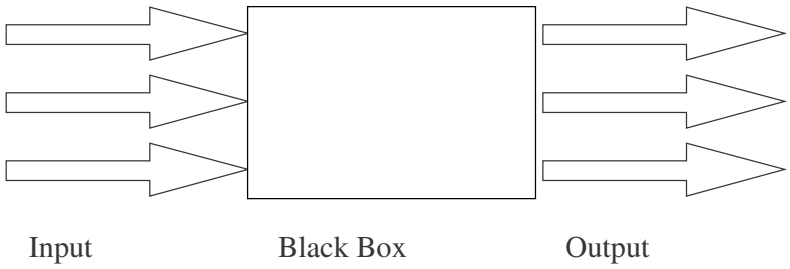
Οι κανονικές εκφράσεις χρησιμοποιούνται κατά κόρον στους λεκτικούς αναλυτές που αποτελούν μέρη των μεταγλωττιστών των γλωσσών προγραμματισμού. Υπάρχουν πολλές στρατηγικές που μπορούμε να ακολουθήσουμε για να κατασκευάσουμε ένα αναγνωριστή κανονικών εκφράσεων. Η πιο συνηθισμένη μέθοδος είναι η κατασκευή ενός αυτομάτου (ντετερμινιστικού ή μη) [ASU85, p.121]. Τι κερδίζουμε όμως με την χρήση των ντετερμινιστικών αυτομάτων και τι με την χρήση των μη ντετερμινιστικών; Η διαφοροποίηση έγκειται στον χρόνο εκτέλεσης και στον χώρο που απαιτείται για την αναπαράσταση του αυτομάτου. Ένα μη ντετερμινιστικό αυτόματο είναι πιο αργό, αλλά απαιτεί λιγότερο χώρο για την αναπαράσταση του. Ένα ντετερμινιστικό αυτόματο απαιτεί ακριβώς το αντίθετο. Οι χρόνοι και ο χώρος που απαιτούνται συνοψίζονται στο πίνακα 2.6. Το |r| είναι το μέγεθος της κανονικής έκφρασης, ενώ το |x| αντιπροσωπεύει το μέγεθος των δεδομένων εισόδου. Η έκφραση O(|r|) είναι γνωστή ως big-oh notation , και χρησιμοποιείται για να προσδιορίσει την αποδοτικότητα ενός αλγορίθμου [Big02].

Αυτόματο	**Χώρος**	**Χρόνος**
Μη- ντετερμινιστικό	O(|r|)	O(|r| $\times$ |x|)
Ντετερμινιστικό	O($2^{\|r\|}$)	O(|x|)

Πίνακας 2.6: Απαιτήσεις χώρου και χρόνου για την επεξεργασία μιας κανονική έκφρασης

2.2.2 Αυτόματα

Σύμφωνα με τον Neumann για τον χρήστη το αυτόματο είναι ένα "μαύρο κουτί" (black box) που διαθέτει διαύλους εισόδου (input) και εξόδου (output) [Anten] (σχήμα 2.1).



Σχήμα 2.1: Αυτόματο

Μη ντετερμινιστικά αυτόματα (NFA)

Ως μη ντετερμινιστικό αυτόματο (Non-deterministic finite automata) ορίζεται κάθε πεντάδα (Q,Σ, δ, q_0, F) όπου:

- Q είναι ένα πεπερασμένο σύνολο καταστάσεων
- Σ είναι ένα πεπερασμένο αλφάβητο από σύμβολα
- $q_0 \in$Q είναι η αρχική κατάσταση
- F$\subseteq$Q είναι το σύνολο το τελικών καταστάσεων
- δ είναι η συνάρτηση με πεδίο ορισμό το Q$\times\Sigma$ και πεδίο τιμών 2^Q (συνάρτηση μετάβασης)

Η συνάρτηση μετάβασης (Transition function) έχει τα παρακάτω χαρακτηριστικά:

- δ είναι μια συνάρτηση με πεδίο ορισμού το Q$\times\Sigma$ και πεδίο τιμών 2^Q
- δ(q,a) = υποσύνολο του Q

Ντετερμινιστικά αυτόματα (DFA)

Ως ντετερμινιστικό αυτόματο (Deterministic finite automata) ή D-αυτόματο ορίζεται κάθε πεντάδα α=(S,E,T,s_0,f) όπου:

- S: Ένα πεπερασμένο σύνολο καταστάσεων
- E: Ένα αλφάβητο εισόδου
- $T \subset S$: Το σύνολο των τελικών καταστάσεων

- $s_0 \in S$: Η αρχική κατάσταση
- f: Μια συνάρτηση με πεδίο ορισμού το SxE και πεδίο τιμών το S

Η συνάρτηση f ονομάζεται συνάρτηση μετάβασης του αυτομάτου και εκφράζει τον εσωτερικό μηχανισμό του. Είναι εκείνη που το χαρακτηρίζει ως ντετερμινιστικό, αφού σε κάθε ζεύγος το SxE αντιστοιχεί ακριβώς ένα στοιχείο του S.

2.2.3 Κατηγοριοποίηση μηχανών κανονικών εκφράσεων

Η κατηγοριοποίηση των μηχανών κανονικών εκφράσεων γίνεται από τον τύπο μηχανής που χρησιμοποιούν. Εκτός από τις παρακάτω βασικές κατηγορίες υπάρχουν και υβριδικές μηχανές που συνδυάζουν χαρακτηριστικά και των δύο κατηγοριών. Αυτές αποτελούν εξαίρεση στον κανόνα και δεν είναι ευρέως διαδεδομένες. Κλασσικά παραδείγματα τέτοιων υλοποιήσεων είναι αυτή της Tcl και της GNU awk [Fri02].

Μηχανές βασισμένες σε ντετερμινιστικά αυτόματα

Οι μηχανές που βασίζονται σε ντετερμινιστικά αυτομάτα χαρακτηρίζονται από την ιδιότητα τους να καθοδηγούνται από τα δεδομένα εισόδου (text-directed)[Fri02]. Τα κύρια χαρακτηριστικά τους είναι:

- Το ταίριασμα γίνεται με μεγάλη ταχύτητα
- Η διαδικασία ταιριάσματος είναι συνεπής (δεν χρειάζεται backtracking)
- Οι τελεστές +,* χαρακτηρίζονται ως lazy. Επιστρέφουν το μεγαλύτερο δυνατό σύνολο χαρακτήρων
- Χρησιμοποιείται παραπάνω μνήμη για την αναπαράσταση του αυτόματου
- Δεν είναι δυνατή η χρήση παρενθέσεων ομαδοποίησης

Μηχανές βασισμένες σε μη-ντετερμινιστικά αυτόματα

Οι υλοποιήσεις που βασίζονται σε μη-ντετερμινιστικά αυτόματα έχουν το χαρακτηριστικό ότι καθοδηγούνται από την ίδια την μηχανή κανονικών εκφράσεων (regex-directed). Αυτό σημαίνει ότι σε κάποιες περιπτώσεις η συνθήκη μετάβασης δεν καθορίζεται από τα δεδομένα.

- Μικρότερη απόδοση

- Η διαδικασία ταιρίασματος δεν είναι συνεπής (χρειάζεται backtracking)
- Οι τελεστές +,* είναι greedy.
- Είναι δυνατή η χρήση παρενθέσεων ομαδοποίησης

Διαφορές μεταξύ NFA και DFA

Οι διαφορές των δύο κατηγοριών συνοψίζονται στον πίνακα 2.7.

Κριτήριο	**Ντετερμινιστική**	**Μη-ντετερμινιστική**
Κόστος σε μνήμη	✓	
Ταχύτητα	✓	
Μεγαλύτερο δυνατό ταίριασμα	✓	διαφορετικό ανά περίπτωση
Ευκολία υλοποίησης		✓
Παρένθεσης ομαδοποίησης		✓
Τελεστές πρόβλεψης		✓
Συνεπές ταίριασμα	✓	
Γραμμές κώδικα	$\sim$ 4500	$\sim$ 9700
Τελεστές greedy	✓	

Πίνακας 2.7: Διαφορές ντετερμινιστικών και μη-ντετερμινιστικών μηχανών

2.2.4 Δημιουργία μη ντετερμινιστικού αυτόματου από κανονική έκφραση

Για την δημιουργία ενός μη-ντετερμινιστικού αυτομάτου ο Thompson είχε διατυπώσει ένα αλγόριθμο (Thompson's construction) [ASU85, p.122].

Αλγόριθμος του Thompson για την κατασκευή μη ντετερμινιστικού αυτόματου από κανονική έκφραση (Thompson's construction)

Είσοδος Μια κανονική έκφραση r ενός αλφάβητου Σ.

Έξοδος Ένα μη-ντετερμινιστικό αυτόματο που δέχεται δεδομένα L(r)

Μέθοδος
Αρχικά διαιρούμε την κανονική έκφραση r στις επιμέρους κανονικές εκφράσεις της. Μετά χρησιμοποιώντας τους κανόνες 1 και 2 κατασκευάζουμε το μη ντετερμινιστικό αυτόματο για καθένα από τα βασικά σύμβολα του r (τα

οποία μπορεί να είναι ε ή σύμβολα του αλφαβήτου). Είναι σημαντικό να καταλάβουμε ότι αν συναντήσουμε ένα σύμβολο a πολλές φορές στο r , τοτε θα κατασκευάσουμε ένα ξεχωριστό αυτόματο για κάθε εμφάνιση.
Μετά χρησιμοποιώντας την συντακτική δομή της κανονικής έκφρασης r , συνδυάζουμε τα αυτόματα που έχουμε παράγει, χρησιμοποιώντας τον κανόνα 3, έως ότου να καταλήξουμε στο αυτόματο της συνολικής έκφρασης.

1. Για το σύμβολο ε κατασκευάζουμε το μη ντετερμινιστικό αυτόματο του σχήματος 2.2. Αυτός ο κανόνας αναγνωρίζει το $\{\varepsilon\}$. Η κατάσταση i είναι η καινούρια αρχική κατάσταση και f η τελική κατάσταση.

2. Για ένα σύμβολο a Σ, κατασκευάζουμε το αυτόματο του σχήματος 2.3. Αυτός ο κανόνας αναγνωρίζει το {a} . Η κατάσταση i είναι η καινούρια αρχική κατάσταση και f η τελική κατάσταση.

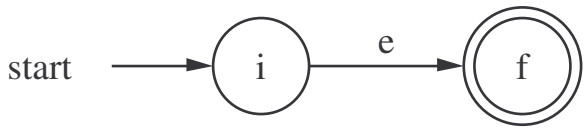
3. Έστω ότι N(s) και N(t) για τις κανονικές εκφράσεις s και t.

   (α′) Για την κανονική έκφραση s | t κατασκευάζουμε το αυτόματο που παράγεται από την σύνθεση αυτών N(s | t). Το παραγόμενο αυτόματο παρίσταται στο σχήμα 2.4.

   (β′) Αντίστοιχα για την κανονική έκφραση st, κατασκευάζουμε το αυτόματο N(st) (σχήμα 2.5).

   (γ′) Για την κανονική έκφραση s* κατασκευάζουμε το N(s*) (σχήμα 2.6).

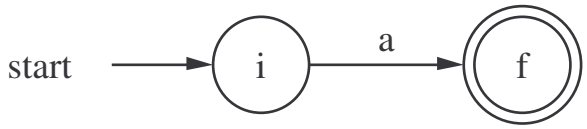
   (δ′) Για τις κανονικές εκφράσεις της μορφής (s) , χρησιμοποιούμε το ίδιο αυτόματο N(s) σαν το παραγόμενο αυτόματο.



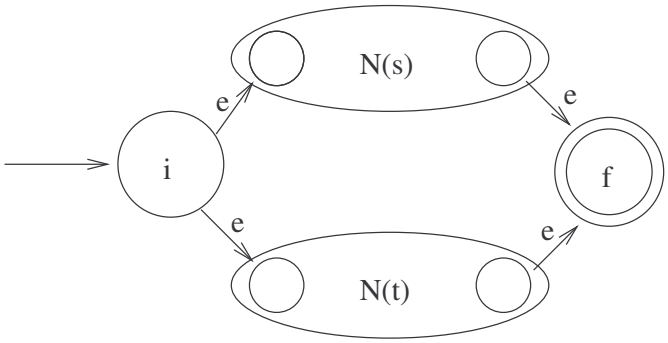
Σχήμα 2.2: Κανόνας 1 του αλγόριθμου Thompson

2.2.5 Δημιουργία ντετερμινιστικού αυτόματου από κανονική έκφραση

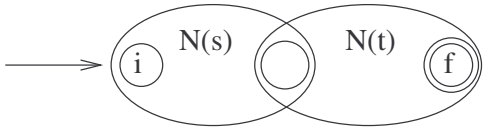
Για την δημιουργία ντετερμινιστικού αυτόματου από κανονική έκφραση ακολουθούνται εν γένει δύο μέθοδοι. Με την πρώτη απλά κατασκευάζουμε το



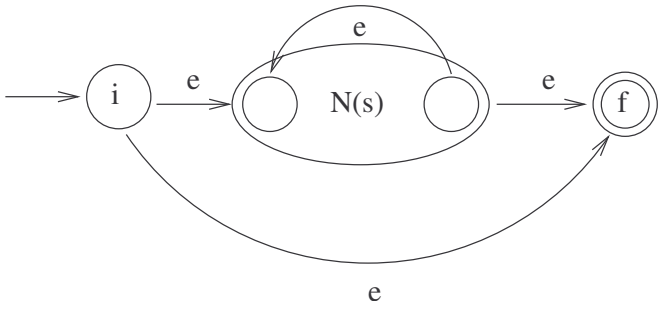
Σχήμα 2.3: Κανόνας 2 του αλγόριθμου Thompson



Σχήμα 2.4: Κανόνας 3-a του αλγόριθμου Thompson



Σχήμα 2.5: Κανόνας 3-b του αλγόριθμου Thompson



Σχήμα 2.6: Κανόνας 3-c του αλγόριθμου Thompson

μη ντετερμινιστικό αυτόματο και μετά το μετατρέπουμε σε ντετερμινιστικό. Η δεύτερη μέθοδος κατασκευάζει το αυτόματο απευθείας από την κανονική έκφραση.

Μετατροπή μη ντετερμινιστικού αυτόματου σε ντετερμινιστικό

Αν κατασκευάσουμε ένα μη ντετερμινιστικο αυτόματο το οποίο αναγνωρίζει μία γλώσσα L, τότε μπορούμε να κατασκευάσουμε ένα αντίστοιχο ντετερμινιστικό αυτόματο που να μπορεί να αναγνωρίζει την ίδια γλώσσα. Ο αλγόριθμος μετατροπής είναι γνωστός και ως αλγόριθμος κατασκευής υποσυνόλου (subset construction) [ASU85, p.118].

Είσοδος Ένα μη ντετερμινιστικό αυτόματο N.

Έξοδος Ένα ντετερμινιστικό αυτόματο D που να δέχεται την ίδια γλώσσα.

Μέθοδος
Ορίζουμε την έννοια του ε-closure. Έστω ότι:

- N = το σύνολο των καταστάσεων ενός μη ντετερμινιστικού αυτόματου.
- R = το σύνολο των καταστάσεων του N οι οποίες είναι ενεργές (υπάρχει κατάσταση και συνθήκη μετάβασης που οδηγεί σε αυτές) και η συνθήκη μετάβασης τους είναι ε.

Τότε ως ε-closure(N) ορίζεται το N$\cup$R.

Ο αλγόριθμος κατασκευής υποσυνόλου έχει τα εξής βήματα:

1. Ορίζουμε ως αρχική κατάσταση του D το ε-closure(S), όπου το S είναι η αρχική κατάσταση του N. Η επιλεγείσα κατάσταση δεν σημειώνεται.
2. Εκτελείται ο ακόλουθος ψευδοκώδικας:

```
Επανάληψη
    Έξοδος όταν όλες οι καταστάσεις του D είναι σημειωμένες
    Επέλεξε μια μη σημειωμένη κατάσταση x στο D
    Για κάθε κατάσταση ς στο Σ
        Κατασκευάζουμε το σύνολο των καταστάσεων T
        στο N του οποίου υπάρχει μια κατάσταση ς από το x
        Θέτουμε y = ε-closure(T)
        Αν y ∉ D τότε
```

Προσθέτουμε το y στο D
Προσθέτουμε την ακμή (edge) (x,ς, y) στο D
Τέλος Για κάθε
σημειώνουμε το x
Τέλος επανάληψης

3. Η τελική κατάσταση S του D είναι όλες αυτές που είναι τελικές καταστάσεις στο N.

Κεφάλαιο 3

Η ιδεατή μηχανή της Java

A programming language is for thinking of programs, not for expressing programs you''ve already thought of. It should be a pencil, not a pen.
- Hackers and Painters, Paul Graham

Σκοπός του κεφαλαίου είναι η εισαγωγή στις βασικές έννοιες της ιδεατής μηχανής της Java. Οι πλήρεις προδιαγραφές για την ιδεατή μηχανή της Java είναι αποτυπωμένες στο Java Virtual Machine Specification [LY97].

3.1 Ιστορική αναδρομή

Η Java ξεκίνησε να αναπτύσσεται από τις αρχές του 1991 από τους Bill Joy, Andy Bechtolsheim, Wayne Rosing, Mike Sheridan, James Gosling και Patrick Naughton. Παρουσιάστηκε επίσημα από την Sun Microsystems το 1995. Από την εμφάνιση της κέρδισε αρκετούς οπαδούς και πλέον είναι μια από τις τρεις επικρατέστερες γλώσσες στην προτίμηση των μηχανικών λογισμικού και των προγραμματιστών.

3.2 Αρχιτεκτονική της Java

Η πλατφόρμα της Java αποτελείται από τρία βασικά μέρη [Inc03]:

1. Το πακέτο ανάπτυξης λογισμικού (SDK, Software Development Kit)
2. Το πακέτο περιβάλλοντος εκτέλεσης (JRE, Java Runtime Environment)
3. Την πλατφόρμα της Java.

Το πακέτο ανάπτυξης λογισμικού χρησιμοποιείται για την ανάπτυξη εφαρμογών σε Java, το πακέτο περιβάλλοντος εκτέλεσης χρησιμοποιείται για την εκτέλεση των εφαρμογών (πρέπει να διανέμεται μαζί με την εφαρμογή) ενώ η πλατφόρμα περιέχει την βάση για την ιδεατή μηχανή και πρέπει να αναπτύσσεται ξεχωριστά για κάθε λειτουργικό σύστημα.

3.2.1 Εργαλεία ανάπτυξης και διεπαφές προγραμματισμού (Development Tools & APIs)

Τα εργαλεία ανάπτυξης περιλαμβάνονται μόνο στο πακέτο ανάπτυξης λογισμικού και χρησιμοποιούνται για την ανάπτυξη εφαρμογών σε Java. Αποτελούνται από τον μεταγλωττιστή (java compiler), τον εκσφαλματωτή (java debugger), τα εγχειρίδια τεκμηρίωσης των διεπαφών προγραμματισμού (javadoc) και τα εργαλεία για ανάπτυξη εφαρμογών για φορητές συσκευές (jPDA).

3.2.2 Τεχνολογίες διανομής (Deployment Technologies)

Οι τεχνολογίες διανομής αποτελούν μια συλλογή εφαρμογών που διαθέτει η πλατφόρμα της Java για την σωστή και εύκολη διανομή εφαρμογών.

3.2.3 Εργαλεία διεπαφών χρηστών (User Interface Toolkits)

Τα εργαλεία διεπαφών χρηστών χρησιμοποιούνται για την ανάπτυξη γραφικών διεπαφών των εφαρμογών. Επίσης περιλαμβάνονται και διεπαφές προγραμματισμού για ήχο (Sound), για γραφικά (Java 2D) καθώς και ειδικές διεπαφές για την ανάπτυξη εφαρμογών για άτομα με ειδικές ανάγκες (Accessibility).

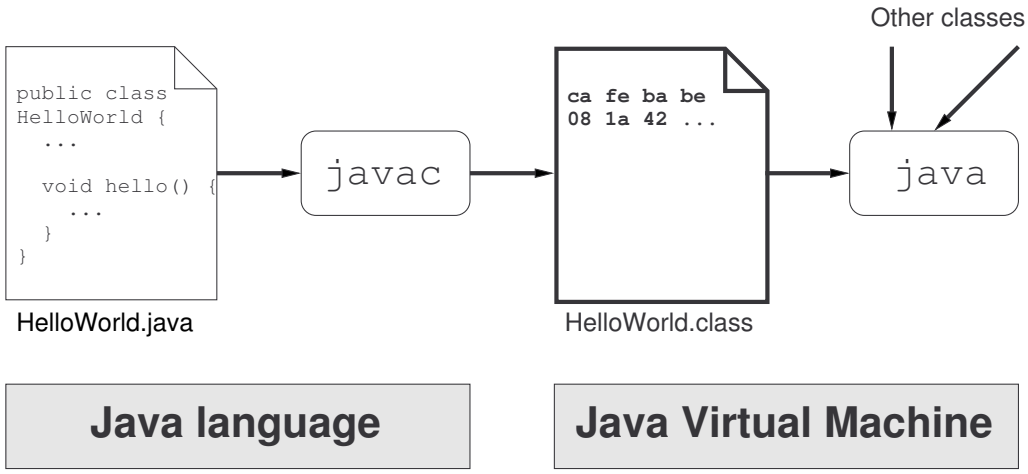
3.2.4 Διεπαφές προγραμματισμού για ενσωμάτωση (Integration APIs)

Οι διεπαφές προγραμματισμού για ενσωμάτωση αποτελούνται από τέσσερις βασικές διεπαφές, το (1) Remote Method Innvocation (RMI), (2) το Java DataBase Connectivity (JDBC), το (3) Java Naming Directory Interface (JNDI) και (4) CORBA. Χρησιμοποιούνται για την ενσωμάτωση εφαρμογών της Java με άλλα συστήματα.

3.2.5 Βασικές διεπαφές προγραμματισμού (Core APIs)

Οι βασικές διεπαφές προγραμματισμού παρέχουν ένα σύνολο προγραμματιστικών πακέτων για την διευκόλυνση της ανάπτυξης εφαρμογών. Οι τεχνολογίες που καλύπτουν οι βασικές διεπαφές προγραμματισμού είναι ποικίλες, όπως, XML (eXtensible Markup Language), Java Native Interface (JNI) κτλ.

3.3 Η ιδεατή μηχανή της Java



Σχήμα 3.1: Μεταγλώττιση και εκτέλεση Java κλάσεων

3.3.1 Εισαγωγή

Η ιδεατή μηχανή εκτελεί τα αρχεία κλάσεων. Αποτελείται από τον Just-In-Time μεταγλωττιστή (Java HotSpot Server/Client Compiler) και την ιδεατή μηχανή (Java HotSpot VM Runtime). Ο just-in-time μεταγλωττιστής χρησιμοποιείται για την μεταγλώττιση κώδικα της ιδεατής μηχανής, σε φυσικό κώδικα (native code) , με σκοπό την αύξηση των επιδόσεων σε ορισμένες περιπτώσεις [Ayc03].

Τα προγράμματα τα οποία είναι γραμμένα σε Java μεταγλωττίζονται σε ένα μεταφέρσιμο δυαδικό κώδικα ο οποίος ονομάζεται *byte code* . Κάθε κλάση αντιστοιχίζεται σε ένα ξεχωριστό αρχείο κλάσης (class file) το οποίο περιέχει δεδομένα και εντολές για την ιδεατή μηχανή (byte code). Τα αρχεία αυτά φορτώνονται δυναμικά στον διερμηνέα (interpretter) και εκτελείται [Dah01].

Στο σχήμα 3.1 εικονογραφείται η διαδικασία μεταγλώττισης και εκτέλεσης μιας κλάσης Java. Το αρχείο Helloworld.java μεταγλωττίζεται στο αρχείο κλάσης Helloworld.class , φορτώνεται από την ιδεατή μηχανή όπου και εκτελείται.

3.3.2 Τύποι δεδομένων

Οι τύποι δεδομένων (data types) που υποστηρίζονται από την ιδεατή μηχανή της Java αναφέρονται στον πίνακα 3.1.

Τύπος	**Περιγραφή**
byte	1-byte προσημασμένος ακέραιος συμπληρώματος 2
short	2-byte προσημασμένος ακέραιος συμπληρώματος 2
int	4-byte προσημασμένος ακέραιος συμπληρώματος 2
long	8-byte προσημασμένος ακέραιος συμπληρώματος 2
float	4-byte IEEE 754 δεκαδικός απλής ακρίβειας
double	8-byte IEEE 754 δεκαδικός διπλής ακρίβειας
char	2-byte μη προσημασμένος χαρακτήρας Unicode
object	4-byte διεύθυνση σε αντικείμενο της Java
returnadress	4-bytes, χρησιμοποιείται με τις εντολές jsr/ret/jsr_w/ret_w

Πίνακας 3.1: Τύποι δεδομένων της Java

Σχεδόν όλοι οι έλεγχοι για τους τύπους δεδομένων γίνονται κατα την διάρκεια της μεταγλώττισης. Οι πίνακες της Java αντιμετωπίζονται ως objects (αντικείμενα).

3.3.3 Καταχωρητές

Η ιδεατή μηχανή έχει ένα αριθμό καταχωρητών που χρησιμοποιούνται κατά την λειτουργία της. Οι καταχωρητές αυτοί έχουν μέγεθος 32-bit.

Ο καταχωρητής pc (program counter) περιέχει την διεύθυνση της επόμενης εντολής προς εκτέλεση. Κάθε μέθοδος χρησιμοποιεί ένα κομμάτι μνήμης στο οποίο καταχωρεί:

- Ένα σύνολο τοπικών μεταβλητών (καταχωρητής register)
- Μια στοίβα εντολών (Καταχωρητής optop)

- Μια δομή περιβάλλοντος εκτέλεσης (καταχωρητής frame)

Η μνήμη αυτή μπορεί να δεσμευθεί συνολικά, εφόσον το μέγεθος που απαιτείται είναι γνωστός κατά την διάρκεια της μεταγλώττισης.

3.3.4 Τοπικές μεταβλητές

Κάθε μέθοδος έχει ένα σταθερό αριθμό τοπικών μεταβλητών. Αυτές διευθυνσιοδοτούνται ως αποστάσεις λέξεων (word offsets) με την χρήση του καταχωρητή vars. Όλες οι τοπικές μεταβλητές έχουν μέγεθος 32-bit.

Οι μεταβλητές τύπου long, double καταλαμβάνουν δύο τοπικές μεταβλητές και διευθυνσιοδοτούνται μέσω της διεύθυνσης της πρώτης.

3.3.5 Στοίβα εντολών

Οι εντολές της ιδεατής μηχανής χρησιμοποιούν την στοίβα εντολών για να λάβουν τα απαραίτητα δεδομένα για την εκτέλεση τους. Τα αποτελέσματα αποθηκεύονται και αυτά στην στοίβα εντολών μετά την εκτέλεση της εντολής. Το μέγεθος των θέσεων της στοίβας εντολών είναι 32-bit.

3.3.6 Περιβάλλον εκτέλεσης

Η πληροφορία που περιέχει το περιβάλλον εκτέλεσης (Execution environment) χρησιμοποιείται για δυναμική διασύνδεση (dynamic linking), κανονική επιστροφή μεθόδων (normal method returns) και για δημοσιοποίηση εξαιρέσεων (exception propagation).

Δυναμική διασύνδεση

Το περιβάλλον εκτέλεσης περιέχει πληροφορίες του πίνακα συμβόλων του διερμηνέα (interpreter symbol table) για την τρέχουσα μέθοδο που εκτελείται καθώς και την τρέχουσα κλάση, για να υποστηρίξει την δυναμική διασύνδεση με τις εντολές της μεθόδου Ο κώδικας που είναι αποθηκευμένος σε κάθε αρχείο κλάσης αναφέρεται σε μεθόδους και μεταβλητές οι οποίες πρόκειται να προσπελασθούν συμβολικά. Με την δυναμική διασύνδεση αυτές οι συμβολικές μέθοδοι μεταφράζονται σε πραγματικές μέθοδοι, φορτώνοντας κλάσεις όποτε χρειαστεί και δίνοντας συγκεκριμένες διευθύνσεις των μεταβλητών οι οποίες είναι αποθηκευμένες στο περιβάλλον εκτέλεσης.

Κανονική επιστροφή μεθόδων

Όταν μια μέθοδο τερματιστεί κανονικά, τότε η τιμή επιστρέφεται στην μέθοδο που την κάλεσε. Για να γίνει αυτό καλείται μια εντολή της ιδεατής μηχανής κατάλληλη για τον αντίστοιχο τύπο των δεδομένων που επιστρέφεται. Το περιβάλλον εκτέλεσης χρησιμοποιείται σε αυτή την περίπτωση για την φόρτωση των καταχωρητών, του μετρητή προγράμματος (pc) ο οποίος έχει αυξηθεί κατάλληλα. Η εκτέλεση τότε συνεχίζεται στο περιβάλλον εκτέλεσης της μεθόδου προκαλέσει την κλήση.

Δημοσιοποίηση εξαιρέσεων

Σε ειδικές περιπτώσεις μπορεί να χρειασθεί η δημοσιοποίηση κάποιας εξαίρεσης. Οι περιπτώσεις αυτές είναι συνήθως λάθη. Η δημοσιοποίηση της εξαίρεσης μπορεί να προκληθεί με ένα από τους παρακάτω τρόπους:

1. Αποτυχία κατά την προσπάθεια να βρεθεί κάποιο αρχείο κλάσης.
2. Ένα λάθος κατά την εκτέλεση (run-time error), όπως π.χ. η προσπέλαση κάποιας μη δεσμευμένης διεύθυνσης (null pointer).
3. Ένα ασύγχρονο γεγονός, π.χ. από την χρήση της Thread.stop από ένα άλλο νήμα (thread).
4. αν το πρόγραμμα χρησιμοποιήσει την κλήση throw.

Ανάλογα με τον κώδικα του προγράμματος (αν προβλέπει τον χειρισμό της εξαίρεσης μέσω εντολών try - catch ή όχι) , γίνεται χειρισμός της εξαίρεσης ελεγχόμενα ή μη (αναγκάζεται την ιδεατή μηχανή να δημιουργήσει μη προβλεπόμενη εξαίρεση).

3.3.7 Διαχείρισή μνήμης

Η διαχείρισή μνήμης στην Java γίνεται αυτόματα μέσω του garbage collection. Δεν δίνεται στον προγραμματιστή η δυνατότητα να αποδεσμεύσει αντικείμενα κατά το δοκούν. Πρέπει να επισημάνουμε ότι ο garbage collector έχει επίδραση μόνο στην περιοχή μνήμης όπου αποθηκεύονται τα αντικείμενα (objects). Η περιοχή αυτή συχνά αναφέρεται ως Java Heap.

3.3.8 Η δομή των αρχείων κλάσης (class files)

Η δομή των αρχείων ενός αρχείου κλάσης Java απεικονίζεται στο σχήμα 3.2 [Dah01]. Κάθε αρχείο κλάσης ξεκινάει με την επικεφαλίδα $0xCAFEBABE$

(magic number) και τον αριθμό έκδοσης (version number). Ακολουθεί η μνήμη σταθερών (constant pool), η δομή που κρατά τα δικαιώματα πρόσβασης για την κλάση (access rights), η λίστα με τις διεπαφές που υλοποιεί (implemented interfaces), η λίστα με τα πεδία (Fields) και τις μεθόδους (Methods) και τέλος η λίστα με τα χαρακτηριστικά (attributes). Επειδή χρησιμοποιείται τόσο πολύ πληροφορία για να βρεθούν δυναμικά οι απαραίτητες κλάσεις, πεδία και μέθοδοι η μνήμη σταθερών (που μπορεί να θεωρηθεί ως ένας μεγάλος πίνακας συμβολοσειρών) αποτελεί περίπου το 60% ενός αρχείου κλάσης. Το 12% αποτελείται από της εντολές της ιδεατής μηχανής (byte code) και το 28% από όλα τα υπόλοιπα δομικά στοιχεία.

Δομή του αρχείου κλάσης

Κάθε αρχείο κλάσης περιέχει μια κλάση Java ή μια διεπαφή (interface). Κάθε κλάση Java αποτελείται από μια ροή 8-bit bytes. Όλα τα 16-bit και τα 32-bit δημιουργούνται από δύο ή τέσσερα bytes αντίστοιχα. Τα bytes είναι αποθηκευμένα σε σειρά δικτύου (network order, big endian) [Ste98, p.34]. Οι τύποι δεδομένων u1, u2 και u4 σημαίνουν ένα, δύο ή τέσσερα bytes μή προσημασμένου ακεραίου. Η δομή του αρχείου κλάσης ακολουθεί:

```
Class File {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count - 1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    u2 attributes_count;
    attribute_info attribute[attribute_count];
}
```

magic

Το πεδίο αυτό έχει πάντα την τιμή $0xCAFEBABE$.

minor_version, major_version

Αυτά τα πεδία περιέχουν την έκδοση του μεταγλωττιστή της Java που δημιούργησε το αρχείο κλάσης.

constant_pool_count

Το πεδίο αυτό περιέχει των αριθμό των εγγραφών που έχει η μνήμη σταθερών της κλάσης.

constant_pool

Η μνήμη σταθερών είναι ένας πίνακας από δεδομένα. Αυτά μπορεί να είναι διάφορες μη μεταβαλλόμενες συμβολοσειρές (string constants), ονόματα κλάσεων, ονόματα πεδίων ή οτιδήποτε άλλο σχετίζεται από την δομή της κλάσης ή από τον κώδικα των μεθόδων.

access_flags

Αυτό το πεδίο περιέχει μια μάσκα δεκαέξι διαφοροποιητών (modifiers) που χρησιμοποιούνται στις δηλώσεις κλάσεων, μεθόδων και πεδίων. Η λίστα των διαθεσίμων διαφοροποιητών παρατίθεται στον πίνακα 3.2. Η ίδια μορφή κωδικοποίησης χρησιμοποιείται από τις δομές field_info (ορισμός μεταβλητής) και method_info (ορισμός μεθόδου).

this_class

Αυτό το πεδίο είναι δείκτης στην μνήμη σταθερών (constant pool). Το constant_pool[this_class] πρέπει να δείχνει σε δομή σταθερής κλάσης (CONSTANT_class).

super_class

Το πεδίο αυτό είναι δείκτης στην μνήμη σταθερών. Η τιμή του constant_pool[super_class] πρέπει να είναι κλάση γονέας. Αν ο δείκτης είναι μηδέν τότε γονέας θεωρείται η κλάση java.lang.Object

interfaces_count

Η τιμή του πεδίου αυτού δηλώνει των αριθμό των διεπαφών που υλοποιεί αυτή η κλάση.

interfaces

Κάθε τιμή είναι δείκτης στην μνήμη σταθερών. Οι θέσεις στην μνήμη σταθερών πρέπει να δείχνουν σε διεπαφές που υλοποιούνται από την κλάση.

methods_count

Η τιμή του πεδίου αυτού δηλώνει των αριθμό των μεθόδων που έχουν οριστεί σε αυτή η κλάση.

methods

Κάθε τιμή είναι δείκτης στην μνήμη σταθερών. Οι θέσεις στην μνήμη σταθερών πρέπει να δείχνουν σε μεθόδους της κλάσης.

attributes_count

Η τιμή του πεδίου αυτού δηλώνει των αριθμό των χαρακτηριστικών που έχουν οριστεί σε αυτή η κλάση.

attributes

Κάθε τιμή είναι δείκτης στην μνήμη σταθερών. Οι θέσεις στην μνήμη σταθερών πρέπει να δείχνουν σε χαρακτηριστικά της κλάσης.

3.3.9 Υπογραφές (Signatures)

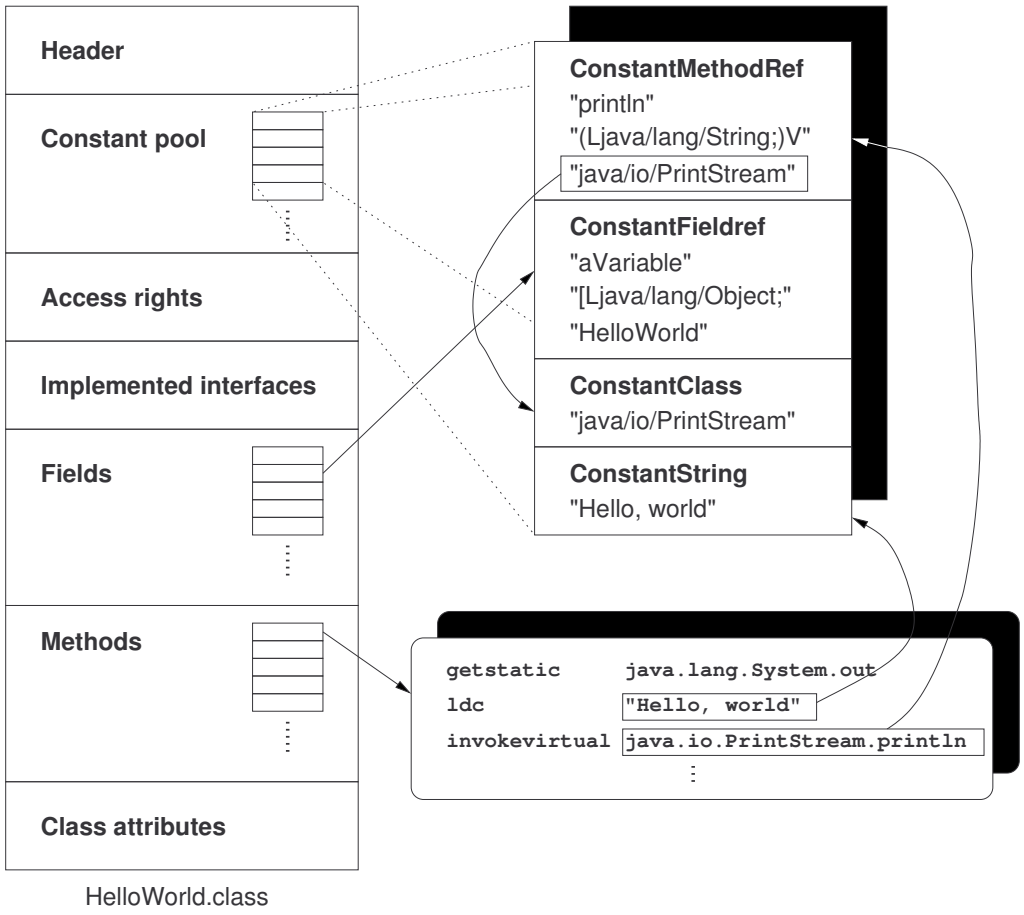
Μια υπογραφή (signature) αντιπροσωπεύει ένα τύπο μεθόδου, πεδίου ή πίνακα. Η υπογραφή πεδίου αντιπροσωπεύει μια παράμετρο σε μια μέθοδο ή τον τύπο μιας μεταβλητής. Δημιουργούνται με βάση την ακόλουθη γραμματική:

```
<field_signature> ::= <field_type>
<field_type>      ::= <base_type>|<object_type>|<array_type>
<base_type>       ::= B|C|D|F|I|J|S|Z
<object_type>     ::= L<fullclassname>;
<array_type>      ::= [<optional_size><field_type>
<optional_size>   ::= [0-9]*
```

Η σημειολογία των βασικών τύπων (base types) εξηγείται στον πίνακα 3.3.

Όνομα σήματος	Τιμή	Περιγραφή	Χρησιμοποιείται
ACC_PUBLIC	0x0001	Ορατή από όλους	Κλάση, Μέθοδος, Χαρακτηριστικό
ACC_PRIVATE	0x0002	Ορατή μόνο από την οριζόμενη κλάση	Μέθοδος, Χαρακτηριστικό
ACC_PROTECTED	0x0004	Ορατή από τις υποκλάσεις	Μέθοδος, χαρακτηριστικό
ACC_STATIC	0x0008	Δηλώνει την μεταβλητή ή την μέθοδο στατική	Μέθοδος, χαρακτηριστικό
ACC_FINAL	0x0010	Απαγορεύει την δημιουργία υποκλάσεων (subclassing), τον επαναπροσδιορισμό (overriding) και την ανάθεση τιμής μετά την αρχικοποίηση	Κλάση, μέθοδος, Χαρακτηριστικό
ACC_SYNCHRONIZED	0x0020	Χρήση στην διαδικασία κλείδωμα της κλάσης	Μέθοδος
ACC_VOLATILE	0x0040	Δεν επιτρέπει το caching της μεταβλητής	Χαρακτηριστικό
ACC_TRANSIENT	0x0080	Δεν επιτρέπεται η χρήση από τον διαχειριστή αντικειμένων (persistent object manager)	Χαρακτηριστικό
ACC_NATIVE	0x0100	Η μέθοδος είναι υλοποιημένη σε άλλη γλώσσα εκτός της Java	Μέθοδος
ACC_INTERFACE	0x0200	Η κλάση ορίζεται ως διεπαφή (interface)	Κλάση
ACC_ABSTRACT	0x0400	Η κλάση/μέθοδος δεν είναι πλήρως ορισμένες	Κλάση, Μέθοδος

Πίνακας 3.2: Ειδικά σήματα πρόσβασης κλάσεων (Access Flags)



Σχήμα 3.2: Δομή ενός αρχείου κλάσης της Java

Συμβολισμός	Τύπος	Περιγραφή
B	byte	προσημασμένος ακέραιος (signed byte)
C	char	χαρακτήρας
D	double	Δεκαδικός κινητής υποδιαστολής διπλής ακρίβειας IEEE
F	float	Δεκαδικός κινητής υποδιαστολής μονής ακρίβειας IEEE
I	int	Ακέραιος (integer)
J	long	Ακέραιος (long integer)
L<classname>;	...	Το αντικείμενο της δωσμένης κλάσης
S	short	προσημασμένος ακέραιος (singed short)
Z	boolean	Λογική τιμή (αληθής ή ψευδής)
[<field sig>	...	Πίνακας

Πίνακας 3.3: Σημειολογία βασικών τύπων υπογραφών της Java

Μια υπογραφή τύπου επιστροφής αντιπροσωπεύει τον τύπο που επιστρέφεται από μία μέθοδο. Δημιουργείτε από την προαναφερθείσα γραμματική προσθέτοντας τον παρακάτω κανόνα:

```
<return_signature> ::= <field_type> | V
```

Το V συμβολίζει ότι η μέθοδος δεν επιστρέφει τίποτα (void).

Παρομοίως η γραμματική που συμβολίζει την υπογραφή των παραμέτρων των μεθόδων, καθώς και τους τύπους δεδομένων που επιστρέφονται από αυτές, περιγράφεται από τους παρακάτω κανόνες.

```
<argument_signature> ::= <field_type>
<method_signature>   ::=
           (<arguments_signature>) <return_signature>
<arguments_signatre> ::= <argument_signature>
```

Παραδείγματα υπογραφών

- public static void main(String[] argv) → ([java/lang/String;)V
- String classname → Ljava/lang/String;

3.3.10 Σύνολο εντολών ιδεατής μηχανής (Byte code instruction set)

Η ιδεατή μηχανή της Java (JVM) δημιουργεί μια τοπική στοίβα συγκεκριμένου μεγέθους για κάθε κλήση μεθόδου. Το μέγεθος της στοίβας υπολογίζεται από τον μεταγλωττιστή. Οι τιμές που αποθηκευόνται είναι τοπικές μεταβλητές οι οποίες χρησιμοποιούνται ως καταχωρητές. Οι μεταβλητές αυτές έχουν μέγιστο πλήθος 65535.

Το σύνολο εντολών της ιδεατής μηχανής αποτελείται από 212 εντολές (instructions), ενώ 44 επιπλέον εντολές (opcodes) είναι προσημειωμένες για μελλοντικές χρήσεις ή βελτιώσεις. Τα σύνολα μπορούν να διαχωριστούν στις παρακάτω ομάδες εντολών:

Χειρισμός στοίβας (Stack operations)

Οι σταθερές (constants) αποθηκεύονται στην στοίβα φορτώνοντας τις από την μνήμη σταθερών χρησιμοποιώντας την ldc εντολή ή με την ειδική συντόμευση iconst_0, bipush κτλ.

Αριθμητικές λειτουργίες (Arithmetic operations)

Η ιδεατή μηχανή της Java χρησιμοποιεί διαφορετικές εντολές για τις αριθμητικές πράξεις ανάλογα με τον τύπο δεδομένων π.χ. Οι εντολές για τους ακεραίους ξεκινούν από i. Για παράδειγμα η εντολή iadd προσθέτει δύο ακεραίους και αποθηκεύει το αποτέλεσμα στην στοίβα. Οι τύποι δεδομένων που εκτελούν αριθμητικές λειτουργίες είναι οι boolean, byte, short, int και char.

Έλεγχος ροής (Control flow)

Ο έλεγχος ροής γίνεται μέσω εντολών διακλάδωσης (branch instructions) όπως goto και if_icmpeq (σύγκριση δύο ακεραίων για ισότητα) κτλ. Οι εντολές try-catch και finally υλοποιούνται από τις εντολές jsp και ret, ενώ εξαιρέσεις (exceptions) δημιουργούνται με την athrow. Οι εντολές στις οποίες καταλήγει μια εντολή διακλάδωσης είναι ακέραιοι που δείχνουν απόσταση σε bytes από την θέση που βρίσκεται εκείνη την στιγμή η εκτέλεση του προγράμματος.

Λειτουργίες φόρτωσης και αποθήκευσης (Load and store operations)

Οι λειτουργίες φόρτωσης και αποθήκευσης χρησιμοποιούνται στις μεθόδους για την διαχείριση των τοπικών μεταβλητών. Μερικά παραδείγματα αυτών

είναι η iload και istore. Υπάρχουν και εντολές όπως η iastore που αποθηκεύει ένα ακέραιο σε ένα πίνακα.

Πρόσβαση πεδίων (Field access)

Η πρόσβαση στα πεδία (χαρακτηριστικά) των κλάσεων γίνεται μέσω τεσσάρων εντολών. Η putfield και getfield χρησιμοποιούνται για τα χαρακτηριστικά των στιγμιοτύπων των κλάσεων, ενώ για τα στατικά οι εντολές getstatic και putstatic.

Κλήση μεθόδων (Method invocation)

Οι μέθοδοι σε επίπεδο ιδεατής μηχανής καλούνται με την χρήση τριών εντολέων. Η invokestatic χρησιμοποιείται για τήν κλήση στατικών μεθόδων, η invokevirtual καλύπτει τις υπόλοιπες περιπτώσεις. Τέλος η εντολή invokespecial καλεί μεθόδους από κλάσεις γονείς (super class) ή ιδιωτικές (private).

Προσδιορισμός αντικειμένων (Object allocation)

Οι κλάσεις δημιουργούνται από την εντολή new. Για την δήλωση πινάκων χρησιμοποιούνται οι εντολές newarray (για μονοδιάστατο πίνακα π.χ. int[]) και anewarray ή multinewarray (για πολυδιάστατους πίνακες π.χ. String[][]).

Μετατροπή και έλεγχος τύπων δεδομένων (Conversion and type checking)

Για τους βασικούς τύπους (int, float, double κτλ) υπάρχουν εντολές μετατροπής όπως π.χ. η f2i που μετατρέπει μια μεταβλητή τύπου float σε integer (int). Η εγκυρότητα της μετατροπής ελέγχεται με την checkcast και την instanceof.

3.3.11 Κώδικας μεθόδων

Οι μέθοδοι που δεν είναι δηλωμένες ως αφαιρετικές (abstract,σχήμα 3.2) περιέχουν τα ακόλουθα δεδομένα: (1) το μέγιστο μέγεθος της στοίβας, (2) των αριθμό των τοπικών μεταβλητών και (3) ένα πίνακα που περιέχει κωδικοποιημένες τις εντολές της ιδεατής μηχανής (byte code instructions).

Κεφάλαιο 4

Υλοποιήσεις μηχανών κανονικών εκφράσεων

"Commander, you know everything about your stone garden. But clearly, you have not spent nearly enough time looking at it."
- Delenn, Babylon 5, "The Gathering"

4.1 Κανονικές εκφράσεις και Java

Με την εμφάνιση της Java άρχισαν να εμφανίζονται οι πρώτες υλοποιήσεις μηχανών κανονικών εκφράσεων. Οι προσπάθειες που έγιναν υλοποιούσαν κυρίως μηχανές που ακολουθούσαν το πρότυπο του POSIX, εκτός από την ORO η οποία υλοποιήθηκε με βάση τις επεκτάσεις τις PERL (παράγραφος 2.1.3). Με την έκδοση 1.4.x η SUN αποφάσισε να συμπεριλάβει μια μηχανή κανονικών εκφράσεων στη κύρια διανομή της πλατφόρμας Java. Η μηχανή αυτή περιλαμβάνεται στις "βασικές διεπαφές προγραμματισμού" (java.util.regex) στις οποίες έχουμε ήδη αναφερθεί στην παράγραφο 3.2.5 (σελίδα 38).

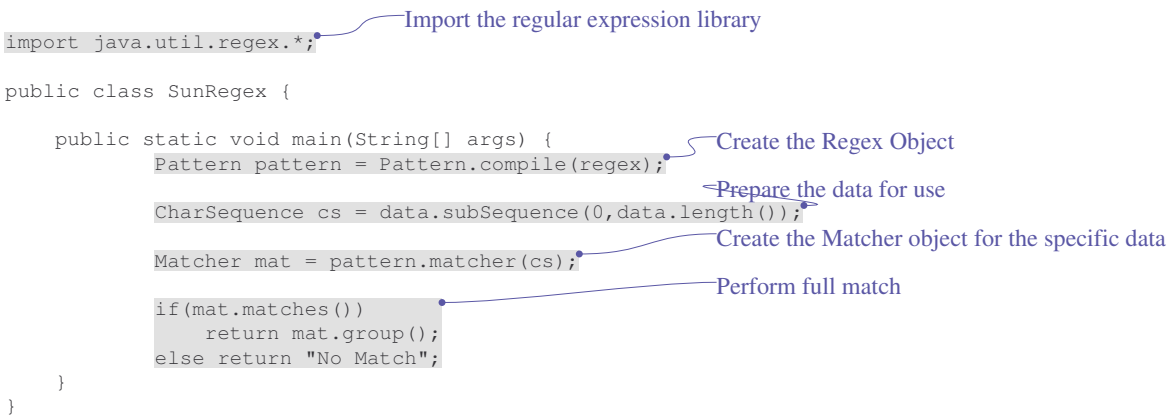
4.2 Μηχανές κανονικών εκφράσεων σε Java

Όλοι οι μεγάλοι οργανισμοί ανάπτυξης λογισμικού έχουν αναπτύξει κάποια μηχανή κανονικών εκφράσεων για Java και την διανέμουν ως βιβλιοθήκη. Είναι σημαντικό να αναφέρουμε ότι οι πιο πολλές από αυτές διανέμονται δωρεάν και αποτελούν λογισμικό ανοιχτού κώδικα (Open Source). Οι κύριοι οργανισμοί που έχουν αναπτύξει μηχανές κανονικών εκφράσεων είναι η Sun Micosystems, η IBM, το Free Software Foundation (GNU) και το Apache software foundation (Jakarta).

4.2.1 Sun Regular Expression Engine

Το πακέτο αυτό διανέμεται μαζί με την πλατφόρμα της Java (από την έκδοση 1.4). Είναι από τις καλύτερες μηχανές κανονικών εκφράσεων για Java όσον αφορά τον συνδυασμό χαρακτηριστικών και απόδοσης. Υλοποιεί όλες τις βασικές λειτουργίες και σαν αρνητικό θα μπορούσαμε να αναφέρουμε ότι δέχεται μόνο αντικείμενα CharSequence ως είσοδο. Επίσης θα μπορούσαμε να αναφέρουμε ότι υποστηρίζει σε μεγάλο βαθμό Unicode. Ακολουθεί παράδειγμα κώδικα χρήσης της μηχανής. Λόγω ομοίοτητας δεν θα παρατεθεί παράδειγμά για τις υπόλοιπες υλοποιήσεις.

Παράδειγμα



4.2.2 IBM

Το πακέτο της IBM είναι εμπορικό και είναι αρκετά πλήρες όσον αφορά τις λειτουργίες του. Έχει αρκετά καλή υποστήριξη για Unicode αλλά όπως θα δούμε αργότερα έχει αρκετά χαμηλή απόδοση.

4.2.3 GNU

Από τις πρώτες μηχανές κανονικών εκφράσεων που υλοποιήθηκαν για την πλατφόρμα της Java. Το πακέτο της GNU έχει πλέον εγκαταλειφθεί. Παρόλα αυτά είναι αρκετά πλήρες όσον αφορά τις προσφερόμενες λειτουργίες Υποστηρίζει Unicode αλλά όχι πλήρως. Η απόδοση του είναι αρκετά χαμηλή σε σχέση με αντίστοιχες μηχανές.

4.2.4 Jakarta Regexp & ORO

Η μηχανή της Jakarta αποτελεί μια καλή προσπάθεια υλοποίησης. Υποστηρίζει Unicode, είναι πλήρης και έχει ικανοποιητική απόδοση. Το ORO είναι μια μηχανή που υποστηρίζει τις επεκτάσεις της PERL (βλέπε παράγραφο 2.1.3). Έχει αρκετά καλές επιδόσεις και δίνει αισθητά καλύτερη λειτουργικότητα σε σχέση με τα άλλα πακέτα.

4.2.5 Automaton

Η μηχανή Automaton είναι μια πειραματική προσπάθεια του Anders Moeller για την κατασκευή μιας γρήγορης μηχανής κανονικών εκφράσεων [Moe03]. Από πλευράς απόδοσης είναι η πιο γρήγορη από τις άλλες μηχανές, ενώ από πλευράς λειτουργικότητας θα μπορούσε να χαρακτηριστεί ελλιπής, •καλύπτοντας μόνο τις βασικές περιπτώσεις. Η υποστήριξη για Unicode κυμαίνεται και αυτή σε βασικό επίπεδο. Πρόκειται για μια βιβλιοθήκη που δεν πρόκειται να χρησιμοποιηθεί ως μέρος μιας εμπορικής εφαρμογής

4.3 Σύγκριση μηχανών κανονικών εκφράσεων

4.3.1 Κριτήρια σύγκρισης

Για να μπορέσουμε να κατατάξουμε τις παραπάνω βιβλιοθήκες θα πρέπει να θέσουμε κάποια κριτήρια. Αυτά απορρέουν μέσα από διάφορα τεχνικά θέματα τα οποία αφορούν τις μηχανές κανονικών εκφράσεων. Αυτά είναι [Fri02]:

- **Τύπος μηχανής**
  χρησιμοποιείται ντετερμινιστικό (DFA) ή μη-ντετερμινιστικό αυτόματο (NFA)?
- **Πρότυπο**
  Ποιο πρότυπο ακολουθεί (POSIX, PCRE) ; Πόσο καλά είναι υλοποιημένο ;
- **Unicode**
  Υποστηρίζει Unicode ;
- **Σχεδιασμός και ευελιξία**
  Δουλεύει με αντικείμενα String ή μόνο με CharSequence ; Χρησιμοποιεί το NIO (New I/O) ; Δουλεύει καλά σε πολυνηματικό περιβάλλον (multi thread);

- **Απαιτήσεις σε πλατφόρμα Java**
  Ποια έκδοση του περιβάλλοντος εκτέλεσης (Java Runtime environment) χρειάζεται για να εκτελεσθεί;

- **Ταχύτητα εκτέλεσης και αποδοτικότητα**
  Είναι ικανοποιητικός ο χρόνος εκτέλεσης;

- **Αξιοπιστία**
  Είναι αξιόπιστα υλοποιημένο; Αν είναι ανοιχτού κώδικα, ο τρόπος γραφής ακολουθεί τα πρότυπα γραφής που έχουν τεθεί από την SUN για την πλατφόρμα της Java ;

4.3.2 Δοκιμή απόδοσης

Για να κατανοήσουμε την ποιότητα των υλοποιημένων βιβλιοθηκών, θα πρέπει να μετρήσουμε τις αποδόσεις τους. Μια τέτοια προσπάθεια έχει πραγματοποιηθεί ήδη από τον ερευνητή Damien Mascord [Mas02]. Ο υπολογιστής στον οποίο εκτελέστηκε το πείραμα είναι ένας Pentium III 650 MHz, με 288 Mb RAM χρησιμοποιώντας το j2sdk 1.4.0. Η κανονική έκφραση που χρησιμοποιήθηκε για τις δοκιμές είναι:
`^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)`
Ο αριθμός επαναλήψεων είχε οριστεί στις 10000. Τα αποτελέσματα παρατίθενται στον πίνακα 4.1. Τα πλήρη αποτελέσματα της δοκιμής απόδοσης βρίσκονται στο παράρτημα Β'.

Όνομα βιβλιοθήκης	**Χρόνος (miliseconds)**
Sun JDK	1122
Automaton	511
GNU Regex	36032
IBM	3946
Jakarta Regexp	6059
ORO	2263

Πίνακας 4.1: Αποτελέσματα δοκιμής επιδόσεων

4.3.3 Αποτελέσματα

Οι δυνατότητες των μηχανών κανονικών εκφράσεων υπό δοκιμή παρατίθενται στον πίνακα 4.2 [Fri02, p.373]. Η βιβλιοθήκη Automaton παραλείπεται λόγο μη πλήρους υλοποίησης.

Χαρακτηριστικό	Sun	IBM	ORO	GNU	Jakarta
Τύπος μηχανής	NFA	NFA	NFA	NFA	NFA
"." δεν ταιριάζει	Διάφορα	Διάφορα	\n	\n, \r	\n
Εμφωλευμένες Παρενθέσεις	✓	✓	✓	✓	
Unicode	✓	✓	✓	✓	✓
Ουδέτερες παρενθέσεις	✓	✓	✓	✓	
Υποστήριξη οκταδικού	✓		✓	✓	

Πίνακας 4.2: Δυνατότητες μηχανών κανονικών εκφράσεων

Από τις διαθέσιμες υλοποιήσεις μηχανών κανονικών εκφράσεων που είδαμε, οι επικρατέστερες είναι αυτή της SUN και της Jakarta (ORO). Αυτές αποτελούν τις πιο πλήρεις υλοποιήσεις (από πλευράς δυνατοτήτων) αλλά και τις πιο αποδοτικές. Ο Jeffrey E.F. Friedl στο βιβλίο "Mastering Regular Expressions 2^{nd} edition" καταλήγει στο συμπέρασμα ότι η καλύτερη υλοποίηση είναι η αντίστοιχη της SUN [Fri02, p.378]. Η μηχανή που διανέμει η SUN είναι όντος από τις πιο πλήρης, αλλά αυτό δεν την καθιστά ως την μοναδική επιλογή. Οι αντίστοιχες μηχανές της Jakarta είναι προσεγμένες ,πλήρεις και δεν υστερούν σε τίποτα από την βιβλιοθήκη της SUN.

Στην παρούσα εργασία το κύριο κριτήριο μας θα είναι η απόδοση, οπότε ια επιλέξουμε τις δύο πιο αποδοτικές μηχανές για τις δοκιμές μας. Σύμφωνα με τον πίνακα 4.1 αυτές είναι: (1) η μηχανή της SUN και η (2) Automaton.

4.4 Υλοποιήσεις σε άλλες πλατφόρμες

Εκτός από τις υλοποιήσεις για την πλατφόρμα της Java, οι κανονικές εκφράσεις έχουν υπάρχουν για μια πλειάδα γλωσσών και λειτουργικών συστημάτων. Ακολουθούν οι πιο σημαντικές υλοποιήσεις

4.4.1 Visual Basic/C# .NET

Η Miscrosoft διαθέτει και αυτή μια μηχανή κανονικών εκφράσεων με την πλατφόρμα του .NET. Η μηχανή αυτή μπορεί να χρησιμοποιηθεί εισάγοντας τον ορισμό "System.Text.RegularExpressions ". Η υλοποίηση παρουσιάζει κάποιες ιδιαιτερότητες ιδιαίτερά στο συντακτικό της, καθώς δεν ακολουθεί κανένα γνωστό πρότυπο. Επίσης δεν έχουν δοθεί χαρακτηριστικά δυνατοτήτων της μηχανής.

Αυτό που παρουσιάζει ενδιαφέρον είναι η επιλογή μεταγλώττισης σε κώδικα για την πλατφόρμα του .NET. Ενεργοποιώντας την επιλογή αυτή η μηχανή κανονικών εκφράσεων μεταγλωττίζει την έκφραση σε κώδικα του CLI (Common Language Infrastructure) [WHA02]. Μετά την αρχική μεταγλώττιση χτίζεται μια βιβλιοθήκη (DLL - Dynamic Linked Library) που περιέχει τον μεταγλωττισμένο κώδικα ο οποίος εκτελείται από εκεί πλέον ως βιβλιοθήκη του προγράμματος. Η μεταγλωττισμένη έκφραση έχει αυξημένη απόδοση που φτάνει στα επίπεδα κώδικα μηχανής του επεξεργαστή (native code).

Παράδειγμα

```
Imports System.Text.RegularExpressions

Public Class regexUser

    Public Sub New()
        Dim rg As Regex = New Regex("[a-zA-Z]*")

        Dim mat As Match

        mat = rg.Match(data)

        If mat.Length <> 0 Then
            resultsText.AppendText("Ok!!")
        Else
            resultsText.AppendText("Failed!!")
        End If
    End Sub

End Class
```

Import the regular expression library

Create the regex instance

Define the match object variable

Perform match

Check if match is succesfull

4.4.2 Perl

Η perl είναι μια από τις γλώσσες προγραμματισμού που χρησιμοποιεί τις κανονικές εκφράσεις ως βασικό στοιχείο του συντακτικού της. Η μηχανή χρησιμοποιεί αυτόματο τύπου NFA για την αναπαράσταση των κανονικών εκφράσεων, ενώ το συντακτικό ανήκει στην κατηγορία PCRE (παρ. 2.1.3).

Παράδειγμα

```
#!/usr/bin/perl

#example data - perl-bin-10.20.30pre8-1.i386.rpm

while(</cdrom/*rpm>) {
    if(/(.*-[0-9a-zA-Z\.]*)-[0-9]*.*\.i386\.rpm/){

        print $1." // ".$_."\n";

    }
}
```

perl program location (Unix only)

Example data in comment

for all rpm contents in /cdrom do

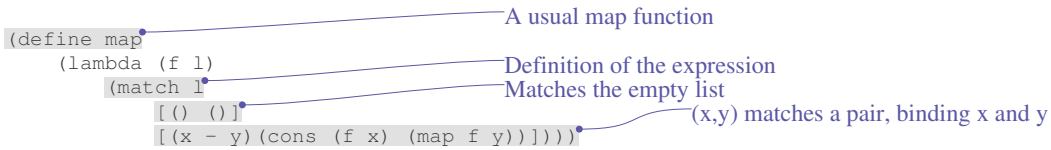
perform tha match

Get the first group ($1)

4.4.3 GNU Kawa

Η GNU εκτός από την μηχανή κανονικών εκφράσεων που αναφέρθηκε νωρίτερα έχει υλοποιήσει και ένα μεταγλωττιστή κανονικών εκφράσεων σε κώδικα Java για την γλώσσα Scheme με την χρήση της πλατφόρμας KAWA [Fou]. Η υλοποίηση αυτή δεν ακολουθεί κανένα γνωστό συντακτικό, αλλά προτείνει ένα δικό της το οποίο βασίζεται στην γλώσσα Scheme.

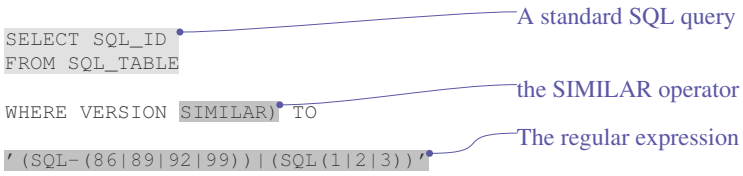
Παράδειγμα



4.4.4 SQL:1999

Στην SQL:1999 προτάθηκε να χρησιμοποιηθούν οι κανονικές εκφράσεις για την αναζήτηση σε κείμενο με την χρήση του τελεστή SIMILAR [EM] [BLSS]. Το συντακτικό των κανονικών εκφράσεων ακολουθούσε το POSIX (παρ. 2.1.3).

Παράδειγμα



Κεφάλαιο 5

Σχεδιασμός του μεταγλωττιστή F.i.r.e.

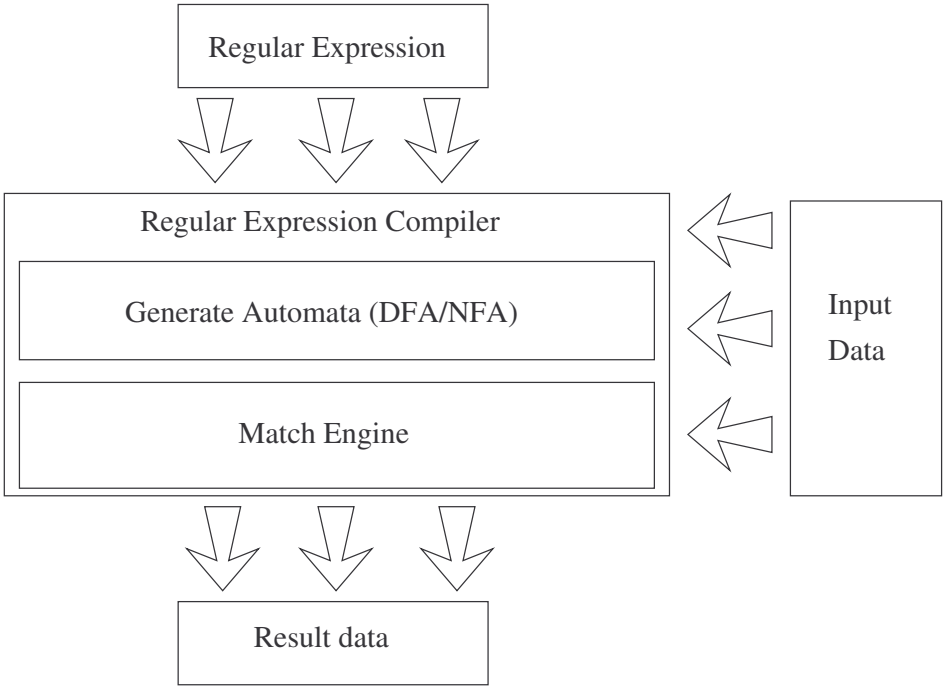
"Do not quench your inspiration and your imagination; do not become the slave of your model."
- Vincent van Gogh

5.1 Βασική αρχιτεκτονική του μεταγλωττιστή

Ο μεταγλωττιστής F.i.r.e. (Fast Implementation of Regular Expressions) είναι κατά βάση μια μηχανή κανονικών εκφράσεων. Σχεδιάστηκε με γνώμονα να πληρεί τις παρακάτω προϋποθέσεις:

1. Μεγαλύτερη αποδοτικότητα εκτέλεσης
2. 100% Συμβατότητα με την αντίστοιχη βιβλιοθήκη της Sun (java.util.regex)
3. Μηχανή βασισμένη σε ντετερμινιστικό αυτόματο (DFA)
4. Υβριδική υλοποίηση (παρ. 2.2.3)
5. 100% υλοποίηση σε Java

Αρχικά πρέπει να αναλύσουμε ποια είναι η διαδικασία επεξεργασίας μιας κανονικής έκφρασης από μία μηχανή κανονικών εκφράσεων. Η κανονική έκφραση τροφοδοτεί τον μεταγλωττιστή (Regular Expression Compiler) που δημιουργεί το αυτόματο (ντετερμινιστικό ή μη ανάλογα με την υλοποίηση). Στην συνέχεια το παραγόμενο αυτόματο χρησιμοποιείται από τον μηχανισμό εύρεσης προτύπων (Matcher Engine) όπου και σε συνδυασμό με τα δεδομένα εισόδου (Input data) επιστρέφονται τα αναμενόμενα αποτελέσματα. Η παραπάνω διαδικασία αποτυπώνεται σχηματικά στο σχήμα 5.1.



Σχήμα 5.1: Μέθοδος επεξεργασίας μιας κανονικής έκφρασης

Ο σχεδιασμός του F.i.r.e. βασίστηκε στην παραπάνω ακολουθία βημάτων. Όμως η αρχιτεκτονική του δεν είναι μονολιθική όπως σε αντίστοιχες υλοποιήσεις. Αποτελείται λοιπόν από τρία σαφώς διαχωρισμένα μέρη:

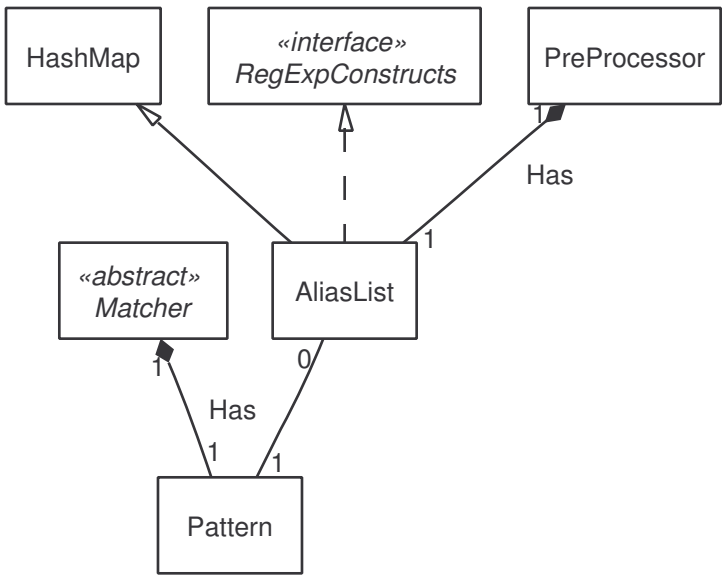
1. Τον πυρήνα του μεταγλωττιστή (F.i.r.e. - CORE)
2. Το τμήμα δημιουργίας αυτομάτου (F.i.r.e. - DFA)
3. Τον μεταγλωττιστή αυτομάτου (F.i.r.e. - WRITER)

Πρέπει να τονίσουμε ότι τα τμήματα δημιουργίας αυτομάτου και ο μεταγλωττιστής αυτομάτου, λειτουργούν ως αποσπώμενα μέρη (plug-ins) του κυρίως μεταγλωττιστή και ενεργοποιούνται κατά βούληση. Ο σχεδιασμός του μεταγλωττιστή F.i.r.e. επηρεάστηκε από διάφορα πρότυπα σχεδίων όπως Singleton κτλ. [GHJV94].

5.2 Πυρήνας του μεταγλωττιστή

Στον πυρήνα του μεταγλωττιστή καθορίζεται η βασική λειτουργικότητα, καθώς και ο τρόπος που επικοινωνούν τα τμήματα μεταξύ τους. Οι βασικές

οντότητες που το αποτελούν ακολουθούν τον τρόπο μοντελοποίησης της μηχανής κανονικών εκφράσεων της Sun (java.uti.regex). Ακολουθεί το διάγραμμα κλάσεων σε UML (σχήμα 5.2).



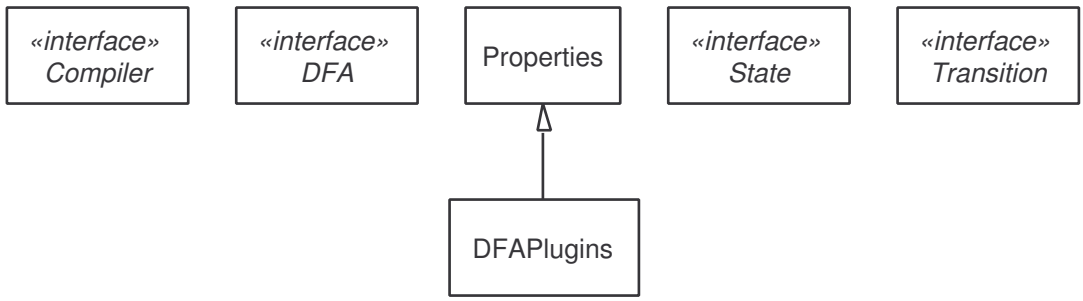
Σχήμα 5.2: Διάγραμμα UML του F.i.r.e.

Οι βασικές κλάσεις είναι η Matcher και η Pattern. Η κλάση Pattern αποτελεί την διεπαφή με τον τμήμα δημιουργίας αυτομάτου και τον μεταγλωττιστή, ενώ αντίθετα η Matcher αποτελεί τον μηχανισμό εύρεσης προτύπων (σχήμα 5.1). Για κάθε κανονική έκφραση ο μεταγλωττιστής κατασκευάζει δυναμικά μια διαφορετική "εκδοχή" της κλάσης Matcher. Αυτό γίνεται ορίζοντας την κλάση Matcher ως abstract, άρα έχουμε την δυνατότητα να δημιουργήσουμε κλάσεις απογόνους αυτής και να υλοποιήσουμε τις σαφώς ορισμένες μεθόδους της, όμως με διαφορετική λειτουργικότητα κάθε φορά.

Η κλάση Preprocessor χρησιμοποιείται ουσιαστικά για την υποστήριξη των κλάσεων χαρακτήρων (παρ. 2.1.3). Πριν από κάθε μεταγλώττιση η κανονική έκφραση μεταφράζεται στην απλή της μορφή και μετά την επεξεργάζεται το τμήμα δημιουργίας αυτομάτου.

5.3 Τμήματα δημιουργίας αυτομάτου

Τα τμήματα δημιουργίας αυτομάτου είναι υπεύθυνα για την δημιουργία του ντετερμινιστικού αυτόματου από την δοθείσα κανονική έκφραση. Το επίπεδο αυτό είναι διαφανές στον τελικό χρήστη. Το διάγραμμα κλάσεων αποτυπώνεται στο σχήμα 5.3.



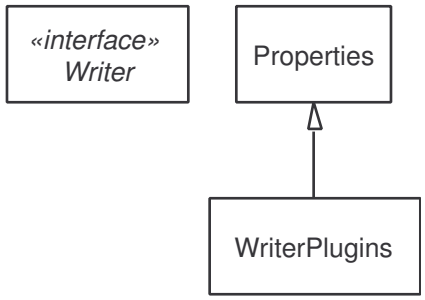
Σχήμα 5.3: Διάγραμμα UML του τμήματος δημιουργίας κλάσεων

Στην ουσία οι διεπαφές (interfaces) Compiler, DFA, State και Transition καθορίζουν την συμπεριφορά που περιμένει ο πυρήνας του μεταγλωττιστή από κάθε ξεχωριστό τμήμα δημιουργίας αυτομάτου. Περισσότερες λεπτομέρειες για τον προγραμματισμό διεπαφών θα δούμε στην παράγραφο 6.2. Η κλάση DFAPlugins είναι στατική και περιέχει όλες τις διαθέσιμες υλοποιήσεις τμημάτων δημιουργίας κώδικα που έχει στην διάθεση του ο μεταγλωττιστής.

5.4 Τμήμα μεταγλώττισης αυτομάτου

Ο μεταγλωττιστής αυτομάτου είναι υπεύθυνος για την μεταγλώττιση του αυτόματου στον τελικό κώδικα εκτέλεσης. Ουσιαστικά σε αυτό το τμήμα κατασκευάζεται η καινούρια εκδοχή της κλάσης Matcher με την μέθοδο που είχε αναφερθεί στην παράγραφο 5.2. Το διάγραμμα των κυριών κλάσεων απεικονίζεται στο σχήμα 5.4.

Η διεπαφή Writer περιέχει τις κλήσεις που θα κάνει ο πυρήνας του F.i.r.e. για να καλέσει τον επιλεγμένο μεταγλωττιστή. Η κλάση WriterPlugins περιέχει όλα τα διαθέσιμα τμήματα μεταγλώττισης που έχουν υλοποιηθεί και είναι διαθέσιμα στον πυρήνα. Η ενεργοποίηση κάθε τμήματος γίνεται μέσω



Σχήμα 5.4: Διάγραμμα UML τμήματος μεταγλώττισης αυτομάτου

της κλάσης αυτής. Περαιτέρω ανάλυση της διαδικασίας προγραμματισμού γίνεται στο κεφάλαιο 6 (παρ. 6.3).

Κεφάλαιο 6

Υλοποίηση του μεταγλωττιστή F.i.r.e.

" Η διαφορά της θεωρίας από την πράξη είναι πολύ μικρή στην θεωρία, αλλά πολύ μεγάλη στην πράξη."
- Ανώνυμος

6.1 Τμηματικός προγραμματισμός στον μεταγλωττιστή F.i.r.e.

Ο μεταγλωττιστής F.i.r.e. υποστηρίζει την ανάπτυξη τμημάτων (modules). Όπως έχει ήδη αναφερθεί στο κεφάλαιο 5 μπορούν να προγραμματιστούν δύο κατηγορίες τμημάτων (1) τα τμήματα δημιουργίας αυτομάτων από κανονική έκφραση και (2) τα τμήματα μεταγλωττιστών σε κώδικα μηχανής.

6.2 Τμήματα δημιουργίας αυτομάτων

Τα τμήματα δημιουργίας αυτομάτων είναι υπεύθυνα για την κατασκευή του ντετερμινιστικού αυτόματου από τη κανονική έκφραση.

6.2.1 Μέθοδος προγραμματισμού

Η μέθοδος προγραμματισμού που χρειάζεται να ακολουθηθεί για την ανάπτυξη ενός τμήματος δημιουργίας αυτομάτου είναι εξαιρετικά απλή και αποτελείται από τα παρακάτω βήματα.

1. Υλοποίηση των διεπαφών Compiler, DFA, State, Transition

2. Καταχώρηση στην κλάση DFAPlugins

3. Προσθήκη στο CLASSPATH του μεταγλωττιστή των απαιτούμενων κλάσεων

Η διεπαφή Compiler

Η διεπαφή Compiler αποτελεί το κεντρικό σημείο υλοποίησης του τμήματος δημιουργίας αυτομάτου. Ο πυρήνας καλεί την συνάρτηση getDFA(String,int) και περιμένει να λάβει το αυτόματο το οποίο αναπαρίσταται από μια κλάση που υλοποιεί την διεπαφή DFA. Ο πηγαίος κώδικας της διεπαφής παρατίθεται στο σχήμα 6.1.

```
package org.fire.regexp.dfa;

public interface Compiler {
    public DFA getDFA(String regex, int flags);
    public String getAuthor();
    public String getName();
}
```

Σχήμα 6.1: Πηγαίος κώδικας της διεπαφής Compiler

Στον πίνακα που ακολουθεί αναλύονται όλα τα συστατικά στοιχεία της διεπαφής.

Όνομα μεθόδου	**Περιγραφή**
getDFA(String,int)	Η μέθοδος που εκκινεί την διαδικασία μεταγλώττισης
getAuthor()	Επιστρέφει τον συγγραφέα του τμήματος δημιουργίας αυτομάτου
getName()	Επιστρέφει το όνομα (μοναδικά ανά τμήμα) του τμήματος δημιουργίας αυτομάτου

Η διεπαφή DFA

Η διεπαφή DFA αποτελεί την δομική οντότητα αναπαράστασης του παραγόμενου ντετερμινιστικού αυτόματου (σχήμα 6.2). Το αυτόματο αποτελείται από κόμβους (state) και συναρτήσεις μετάβασης (Transitions).

```
package org.fire.regexp.dfa;

import java.util.Set;

public interface DFA {
    public Set getStates();
    public void determinize();
    public void removeDeadTransitions();
    public String toDot();
    public void saveDot(String filename);
    public State getInitialState();
}
```

Σχήμα 6.2: Πηγαίος κώδικας της διεπαφής DFA

Ακολουθεί η ανάλυση των μεθόδων της διεπαφής.

Όνομα μεθόδου	**Περιγραφή**
getStates()	Επιστρέφει το σύνολο των κόμβων του αυτόματου
determinize()	Μετατρέπει το αυτόματο σε ντετερμινιστικό
removeDeadTransitions()	Ελέγχει την ορθότητα των συνθηκών μετάβασης και διαγράφει όλες τις μη απαραίτητες
toDot()	Επιστρέφει το παραγόμενο αυτόματο σε μορφή γραφήματος σε μορφή εισόδου για το πρόγραμμα Graphviz
saveDot()	Αντίστοιχα με την μέθοδο toDot. Συμπληρωματικά το αποθηκεύει σε αρχείο.
getInitialState()	Επιστρέφει τον κόμβο εισόδου του αυτόματου

Η διεπαφή State

Κάθε κόμβος του παραγόμενου αυτόματου αναπαρίσταται μέσω κλάσεων που υλοποιούν την διεπαφή State. ακολουθεί το σχήμα 6.3 που περιέχει τον πηγαίο κώδικα, καθώς και ανάλυση των μεθόδων της διεπαφής.

```
package org.fire.regexp.dfa;

import java.util.Set;

public interface State {
    public void addTransition(Transition t);
    public int compareTo(Object o);
    public Set getTransitions();
    public boolean isAccept();
    public void setAccept(boolean accept);
    public State step(char c);
    public String toString();
    public int getStateNumber();
}
```

Σχήμα 6.3: Πηγαίος κώδικας της διεπαφής State

Όνομα μεθόδου	**Περιγραφή**
addTransition(Transition)	Προσθέτει μια συνθήκη μετάβασης στον κόμβο
compareTo(Object)	Συνάρτηση που συγκρίνει τους κόμβους των αυτομάτων
getTransitions()	Επιστρέφει λίστα με τις συναρτήσεις μετάβασης
isAccept()	Πληροφορεί για το αν ο κόμβος είναι τερματικός
setAccept(boolean)	Θέτει την τερματική κατάσταση του κόμβου
step(char)	Ελέγχει τον χαρακτήρα εισόδου με τις συναρτήσεις μετάβασής του κόμβου και επιστρέφει το επόμενο κόμβο στο αυτόματο
toString()	Επιστρέφει μια αλφαριθμητική απεικόνιση του κόμβου
getStateNumber()	Επιστρέφει τον κωδικό αριθμό ανά κόμβο (μοναδικός ανά κόμβο στο αυτόματο)

Η διεπαφή Transition

Η απεικόνιση των συνθηκών μετάβασης υλοποιείται μέσω κλάσεων που ακολουθούν το πρότυπο της διεπαφής Transition. Η διεπαφή περιέχει χαρακτήρες που λειτουργούν ως άνω και κάτω φράγμα για την συνθήκη μετάβασης πχ. αν min = 'a' και max = 'z' τότε η συνθήκη μετάβασης ενεργοποιείται μόνο αν ο χαρακτήρας εισόδου είναι στο διάστημα $'a' \leq input \leq' z'$. Στην περίπτωση που τα όρια είναι ίσα τότε ο χαρακτήρας εισόδου που ενεργοποιεί την συνθήκη είναι ένας.

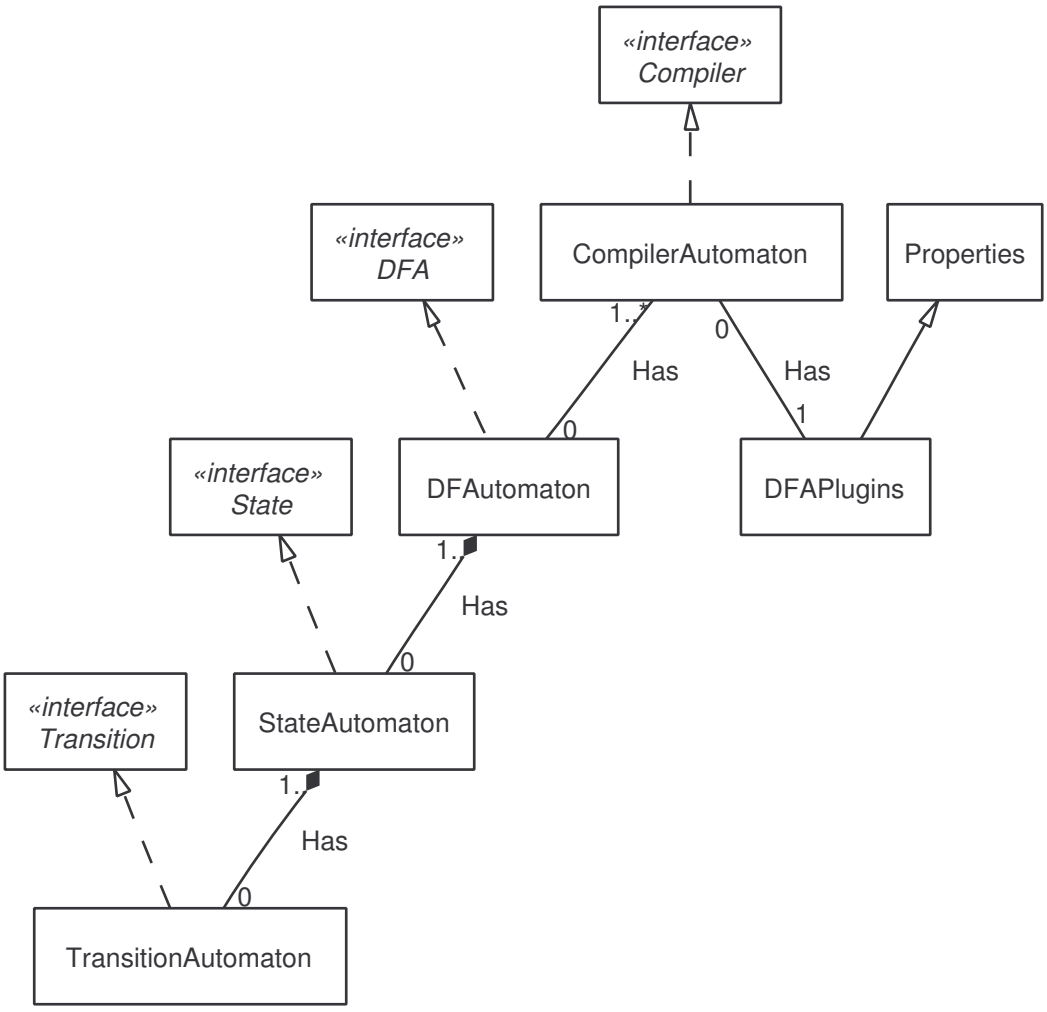
```
package org.fire.regexp.dfa;

public interface Transition {
    public State getDest();
    public char getMax();
    public char getMin();
    public String toString();
}
```

Σχήμα 6.4: Πηγαίος κώδικας της διεπαφής Transition

Όνομα μεθόδου	**Περιγραφή**
getDest()	Επιστρέφει τον κόμβο προορισμού της συνάρτησης μετάβασης
getMax()	Επιστρέφει το χαρακτήρα που λειτουργεί ως άνω φράγμα για την συνθήκη μετάβασης
getMin()	Επιστρέφει το χαρακτήρα που λειτουργεί ως κάτω φράγμα για την συνθήκη μετάβασης
toString()	Επιστρέφει μια αλφαριθμητική απεικόνιση του κόμβου

6.2.2 DFAAutomaton

Στα πλαίσια της πτυχιακής εργασίας υλοποιήθηκε ένα τμήμα δημιουργίας αυτομάτου το οποίο βασίστηκε στην βιβλιοθήκη του Automaton [Moe03]. Το διάγραμμα κλάσεων του τμήματος απεικονίζεται στο σχήμα 6.5.



Σχήμα 6.5: Διάγραμμα κλάσεων του τμήματος DFAAutomaton

Παράδειγμα Χρήσης

Εφόσον υπάρχει υλοποίηση του τμήματος δημιουργίας αυτομάτου (η οποία ακολουθεί την μέθοδο προγραμματισμού που έχουμε ήδη αναλύσει) πρέπει αρχικά να ενεργοποιήσουμε το συγκεκριμένο τμήμα. Στο απόσπασμα κώδικα που ακολουθεί περιγράφεται αυτή η διαδικασία.

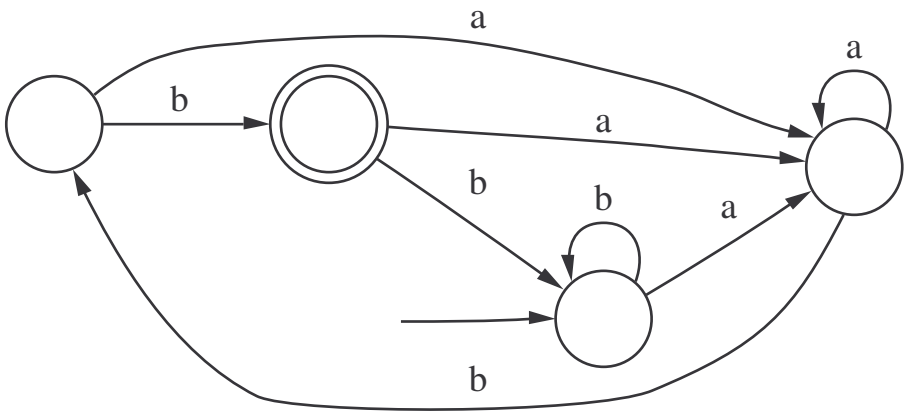
```
import org.fire.regexp.dfa.*;
import org.fire.regexp.*;

public class TestDFAutomaton {

    public static void main(String[] args) {

        DFAPlugins defInstance = DFAPLugins.getDefaultInstance();
        defInstance.setProperty((new CompilerAutomaton()).getName(),
                        "org.fire.regexp.dfa.automaton.CompilerAutomaton","");

        Pattern pat = Pattern.getInstance();
        pat.setCompiler((new CompilerAutomaton()).getName());

    }
```

Import the regex libraries

Register the new compiler plug-in

Initialize the new compiler plug-in

Αν πχ έχουμε την κανονική έκφραση `(a|b)*abb` το παραγόμενο αυτόματο είναι (η γραφική του αναπαράσταση προήλθε μέσω χρήσης της συνάρτησης toDot()):



Με δύο ομόκεντρους κύκλους συμβολίζονται οι τερματικοί κόμβοι, ενώ με βέλος οι συνθήκες μετάβασης.

6.3 Τμήματα μεταγλωττιστών σε κώδικα μηχανής

6.3.1 Μέθοδος προγραμματισμού

Τα τμήματα μεταγλωττιστών σε κώδικα μηχανής έχουν ως στόχο την μετατροπή του αυτομάτου στο τελικό περιβάλλον εκτέλεσης πχ. Java Source, JVM bytecode κτλ.

Η διεπαφή Writer

Για να κατασκευασθεί ένα τμήμα μεταγλωττιστή για κώδικα μηχανής αρκεί να υλοποιηθεί η διεπαφή Writer. Η υλοποίηση των μεθόδων compile(...) θα πρέπει να επιστρέφει ένα instance της κλάσης Matcher , το οποίο είναι μοναδικό για κάθε κανονική έκφραση. Στο σχήμα 6.6 παρατίθεται ο πηγαίος κώδικας της διεπαφής Writer.

```
public interface Writer {
    public String getName();
    public String getDescription();
    public Matcher compile(DFA dfa, String regex);
    public Matcher compile(DFA dfa,
                           String regex,
                           String filename);
}
```

Σχήμα 6.6: Πηγαίος κώδικας της διεπαφής Writer

Ακολουθεί ανάλυση των μεθόδων της διεπαφής Writer.

Όνομα μεθόδου	**Περιγραφή**
getName()	Επιστρέφει το κωδικό όνομα
getDescription()	Επιστρέφει αλφαριθμητικό που περιέχει πληροφορίες για το τμήμα του μεταγλωττιστή
compile(DFA,String)	Μεταγλωττίζει το αυτόματο σε κώδικα μηχανής
compile(DFA,String,String)	Μεταγλωττίζει το αυτόματο σε κώδικα μηχανής και παράγει κλάση με συγκεκριμένο όνομα

Η κλάση Matcher

Το αποτέλεσμα του τμήματος μεταγλώττισης είναι μια εκδοχή της κλάσης Matcher. Κατά την δημιουργία της κλάσης, δύο ζητήματα προκύπτουν και πρέπει να αντιμετωπιστούν. Η υλοποίηση των μεθόδων find() και matches() καθώς και ο τρόπος χειρισμού των δεδομένων εισόδου που γίνεται από τις υπόλοιπες μεθόδους.

Οι μέθοδοι find() και matches() εκτελούν την έρευνα και περιέχουν το κωδικοποιημένο αυτόματο.

Ο χειρισμός δεδομένων εισόδου έχει μεγάλη σημασία γιατί επηρεάζει δραματικά την απόδοση του τμήματος. Για αυτό τον λόγο την υλοποίηση την αφήνουμε στον προγραμματιστή του κάθε τμήματος. Στα πλαίσια της πτυχιακής εργασίας αναπτύχθηκαν δύο μέθοδοι για τον χειρισμό δεδομένων, οι οποίοι αναλύονται στην παράγραφο 6.3.2. Ακολουθεί παράθεση αποσπάσματος κώδικα της κλάσης Matcher και ανάλυση των πιο σημαντικών μεθόδων της.

```
package org.fire.regexp;

public abstract class Matcher implements Serializable{
    [...]
    public abstract boolean find();
    public abstract boolean matches();
    protected abstract void initBuffer(CharSequence);
    protected abstract int getBufferSize();
    protected abstract int getBufferPosition();
    protected abstract void setBufferPosition(int);
    protected abstract void rewindBuffer();
    protected abstract String substring(int,int);
}
```

Σχήμα 6.7: Πηγαίος κώδικας της κλάσης Matcher

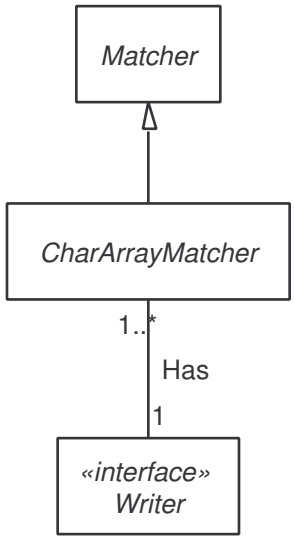
Όνομα μεθόδου	**Περιγραφή**
find()	Επιστρέφει το επόμενο αλφαριθμητικό που ικανοποιεί την κανονική έκφραση
matches()	Εκτελεί καθολικό έλεγχο για ταίριασμα
initBuffer(CharSequence)	Αρχικοποιεί τα δεδομένα για την μηχανή ελέγχου προτύπων (Matching Engine)
getBufferSize()	Επιστρέφει το μέγεθος των δεδομένων εισόδου (σε bytes)
getBufferPosition()	Επιστρέφει την θέση που ερευνά η μηχανή ελέγχου προτύπων
setBufferPosition(int)	Υποδεικνύει συγκεκριμένη θέση στην μηχανή ελέγχου προτύπων για να ξεκινήσει έρευνα
rewindBuffer()	Αρχικοποιεί τα δεδομένα
substring(int,int)	Επιστρέφει αλφαριθμητικό με τα δεδομένα που βρίσκονται σε συγκεκριμένο σύνολο θέσεων

6.3.2 Μέθοδοι χειρισμού των δεδομένων

Με τον χειρισμό των δεδομένων θέλουμε να απλοποιήσουμε και να βελτιστοποιήσουμε τον τρόπο διαχείρισης των δεδομένων εισόδου. Η προσπάθεια όμως αυτή εγκυμονεί κινδύνους, αφού αν ακολουθήσουμε λάθος προσέγγιση μπορούμε να έχουμε καταστροφικά αποτελέσματα. Για την διαχείριση των δεδομένων εισόδου πρέπει να υλοποιήσουμε στην κλάση Matcher της μεθόδους (1) initBuffer(CharSequence), (2) getBufferSize(), (3) getBufferPosition(), (4) setBufferPosition(int), (5) rewindBuffer() και (6) substring(int,int).
Στα πλαίσια της πτυχιακής υλοποιήθηκαν δύο μέθοδοι για τον χειρισμό των δεδομένων. Διαχείριση μέσω του παραδοσιακού I/O της Java και με την χρήση του NIO (New I/O) το οποίο υπόσχεται μεγαλύτερη ταχύτητα στην προσπέλαση και ανάκτηση των δεδομένων.

CharArrayMatcher

Η πρώτη προσέγγιση για τον χειρισμό των δεδομένων εισόδου έγινε μέσω του παραδοσιακού I/O που παρείχε η Java. Τα δεδομένα εισόδου φυλάσσονταν σε ένα πίνακα χαρακτήρων και η προσπέλαση γίνεται μέσω μετρητών. Το διάγραμμα UML απεικονίζεται στο σχήμα 6.8.

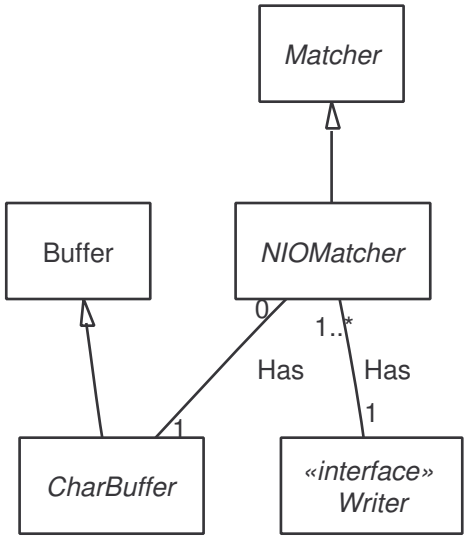


Σχήμα 6.8: Διάγραμμα κλάσεων UML για χειρισμό δεδομένων με παραδοσιακό I/O

Στην πράξη αποδείχτηκε ότι η προσέγγιση αυτή είναι πιο αποδοτική συγκριτικά με το NIO. Αυτό συμβαίνει επειδή είναι δυνατή η απευθείας προσπέλαση στον πίνακα με τα δεδομένα, ενώ μέσω του NIO οι κλήσεις γίνονται μέσω συναρτήσεων, οι οποίες αποτελούν σημαντική επιβάρυνση. Αυτό δεν σημαίνει φυσικά ότι το NIO δεν πραγματοποιεί αυτά που υπόσχεται, απλά είναι πιο αποδοτικό όταν γίνεται αντιγραφή ή ανάκτηση μεγάλων ποσοτήτων δεδομένων.

NIOMatcher

Παρόλο που με την χρήση του παραδοσιακού I/O τα αποτελέσματα απόδοσης ήταν ικανοποιητικά, έπρεπε να δοκιμάσουμε και το New I/O το οποίο υποσχόταν και καλύτερες επιδόσεις. Ακολουθεί το αντίστοιχο διάγραμμα κλάσεων στο σχήμα 6.9.



Σχήμα 6.9: Διάγραμμα κλάσεων UML για χειρισμό δεδομένων με παραδοσιακό I/O

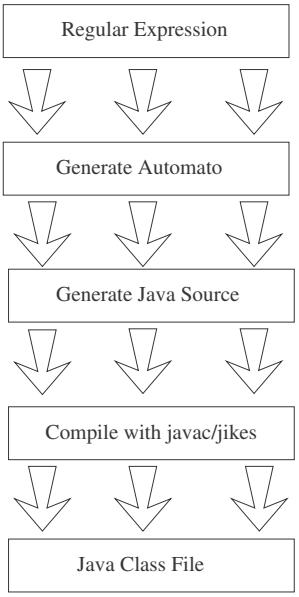
6.3.3 JavaSourceWriter

Το πρώτο τμήμα μεταγλωττιστή σε κώδικα μηχανής είχε ως στόχο την μετατροπή του αυτομάτου σε κώδικα Java. Η προσέγγιση αυτή ήταν το πρώτο λογικό βήμα στον δρόμο της βελτιστοποίησης διότι:

1. Το αποτέλεσμα ήταν ένα αρχείο κώδικα Java, το οποίο ήταν πολύ εύκολο στην επεξεργασία και την βελτίωση.

2. Δίνει την δυνατότητα να μεταγλωττιστεί το αυτόματο σε συγκεκριμένο πρόγραμμα, εξειδικευμένο για κάθε κανονική έκφραση.

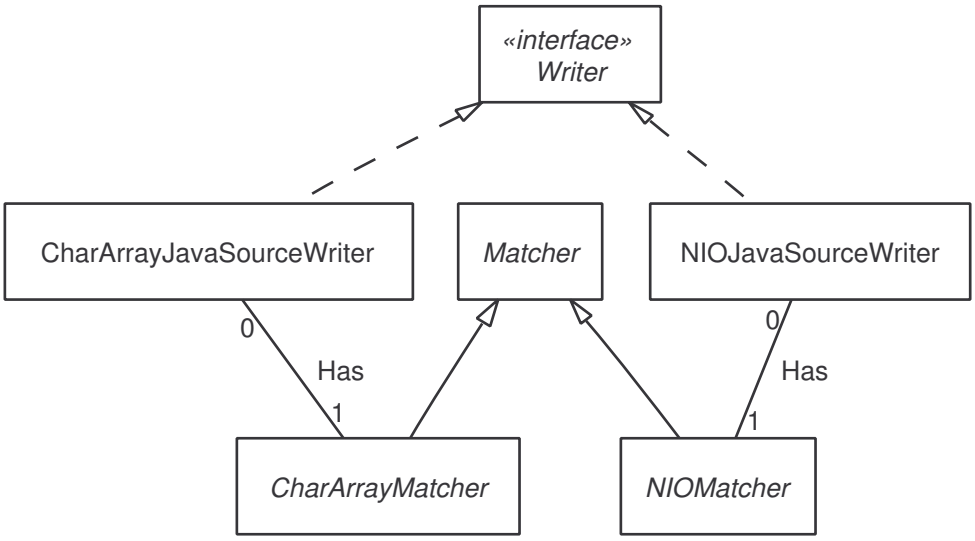
Η διαδικασία λειτουργίας του τμήματος μεταγλώττισης JavaSourceWriter απεικονίζεται στο σχήμα 6.10.



Σχήμα 6.10: Διαδικασία λειτουργίας του JavaSourceWriter

Η κανονική έκφραση μετασχηματίζεται σε αυτόματο, το οποίο με την σειρά του μέσω του τμήματος μεταγλώττισης σε κώδικα Java. Τον παραγόμενο κώδικα το μεταγλωττίζει ο javac ή ο jikes (τυπικοί μεταγλωττιστές για Java). Το παραγόμενο αρχείο είναι μια κλάση που υλοποιεί τις μεθόδους της Matcher. Ακολουθεί το UML διάγραμμα του τμήματος μεταγλώττισης JavaSourceWriter (σχήμα 6.11).

Στη ουσία υλοποιήθηκαν δύο τμήματα μεταγλώττισης που παρήγαγαν Java. Ένα που χειριζόταν τα δεδομένα εισόδου με παραδοσιακό I/O και άλλο ένα που χρησιμοποιούσε NIO. Ο κώδικας που παράγουν τα τμήματα ακολουθεί τις τεχνικές που χρησιμοποιούνται στην λεκτική ανάλυση όπως π.χ. lex [ASU85, p.106]. Παράδειγμα αρχείου κώδικα java παρατίθεται στο παράρτημα Γ.

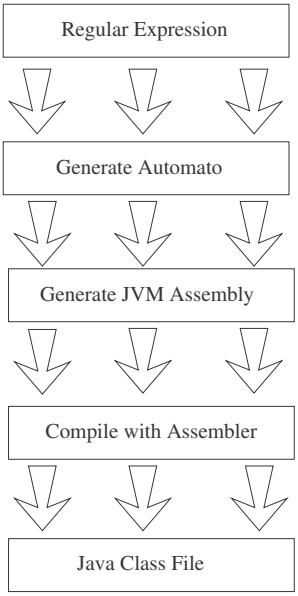


Σχήμα 6.11: Διάγραμμα κλάσεων UML για το τμήμα μεταγλωττιστή JavaSourceWriter

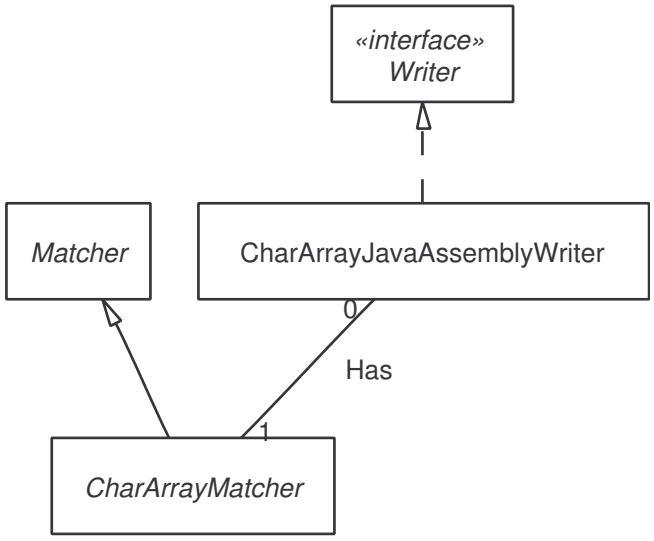
6.3.4 JavaAssemblyWriter

Το επόμενο λογικό βήμα για να βελτιωθεί η απόδοση του παραγόμενου κώδικα είναι η παράκαμψη του μεταγλωττιστή και η υλοποίηση της κλάσης Matcher σε επίπεδο εντολών ιδεατής μηχανής. Η πλατφόρμα της Java έχει ένα είδος assembly την οποία χρησιμοποιεί η ιδεατή μηχανή για να εκτελέσει τα αρχεία κλάσης. Οι μεταγλωττιστές που διανέμονται με την πλατφόρμα της Java κάνουν αρκετές βελτιστοποιήσεις στο εκτελέσιμο κώδικα που παράγουν, αλλά σε εξειδικευμένα προβλήματα μόνο ο άνθρωπος μπορεί να δώσει την αρτιότερη λύση. Παρακάμπτοντας τον μεταγλωττιστή (javac,jikes) η διαδικασία λειτουργίας του τμήματος μεταβάλλεται σύμφωνα με το σχήμα 6.12.

Όπως είναι προφανές πλέον απλά τροφοδοτούμε τον κώδικα στον Assembler ο οποίος απλά δημιουργεί το αρχείο κλάσης. Παράδειγμα αρχείου κώδικα Assembly που δημιουργήθηκε από το τμήμα μεταγλώττισης JavaAssemblyWriter παρατίθεται στο παράρτημα Δ'. Οι εντολές ιδεατής μηχανής της Java είναι καταγεγραμμένες στο "Java Virtual Machine Specification" [LY97]. Το διάγραμμα UML του τμήματος JavaAssemblyWriter απεικονίζεται στο σχήμα 6.13.



Σχήμα 6.12: Διαδικασία λειτουργίας του JavaAssemblyWriter

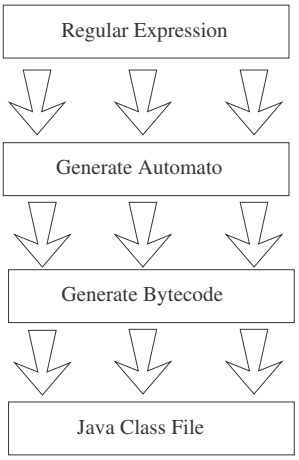


Σχήμα 6.13: Διάγραμμα κλάσεων UML του τμήματος μεταγλώττισης JavaAssemblyWriter

6.3.5 JavaClassWriter

το τελικό βήμα είναι η παράκαμψη του Assembler και η δημιουργία του αρχείου κλάσης απευθείας από το τμήμα μεταγλωττιστή. Υλοποιώντας αυτό

το τμήμα δεν βελτιώνουμε την απόδοση, απλά αυξάνουμε τον βαθμό αυτονομίας. Η διαδικασία λειτουργίας αλλάζει σύμφωνα με αυτή του σχήματος 6.14.



Σχήμα 6.14: Διαδικασία λειτουργίας του JavaClassWriter

Κεφάλαιο 7

Δοκιμές απόδοσης

"Win, lose or draw, this thing's going to know it was in a fight."
- Garibaldi, "Infection", Babylon 5

7.1 Πειραματικά αποτελέσματα

Για να μπορέσουμε να αξιολογήσουμε την μηχανή μεταγλώττισης F.i.r.e θα πρέπει να πραγματοποιήσουμε πειράματα και να την συγκρίνουμε με αντίστοιχες μηχανές σε Java. Επίσης θα πρέπει να ικανοποιεί τα κριτήρια που θέσαμε στο κεφάλαιο 4 όπου συζητήσαμε για τις μηχανές κανονικών εκφράσεων (παρ. 4.3.1).

7.1.1 Έλεγχος συμβατότητας με αντίστοιχες μηχανές

Ο μεταγλωττιστής F.i.r.e. δημιουργήθηκε με κύριο στόχο την συμβατότητα και την ταχύτητα. Για να ελέγξουμε την συμβατότητα του με τα πρότυπα εκτελέσαμε μια σειρά από κανονικές εκφράσεις οι οποίες είχαν ως στόχο να δοκιμάσουν τον μεταγλωττιστή σε ακραίες περιπτώσεις. Ενδεικτικά στο παραδοτέο αυτό υπάρχουν περίπου οι εκατό πρώτες (το τεστ απαρτίζεται από ένα σύνολο 700 εκφράσεων). Το τεστ έχει υλοποιηθεί για την δοκιμή της μηχανή κανονικών εκφράσεων της Perl. Πρέπει να τονίσουμε ότι πολλές κανονικές εκφράσεις είναι λάθος και επίτηδες κατασκευασμένες για να δοκιμάσουν την συμπεριφορά του μεταγλωττιστή σε τέτοιες καταστάσεις. Για κάθε σετ κανονικής έκφρασης και δεδομένων απαιτείται και πλήρες ταίριασμα (full match). Ακολουθεί πίνακας με αποτελέσματα.

Κανονική έκφραση	**Δεδομένα**	**Αποτέλεσμα**
abc	abc	✓
abc	xbc	✗
abc	axc	✗
abc	abx	✗
abc	xabcy	✗
abc	ababc	✗
ab*c	abc	✓
ab*bc	abc	✓
ab*bc	abbc	✓
ab*bc	abbbbc	✓
.{1}	abbbbc	✗
.{3,4}	abbbbc	✗
ab{0,}bc	abbbbc	✓
ab+bc	abbc	✓
ab+bc	abc	✗
ab+bc	abq	✗
ab{1,}bc	abq	✗
ab+bc	abbbbc	✓
ab{1,}bc	abbbbc	✓
ab{1,3}bc	abbbbc	✓
ab{3,4}bc	abbbbc	✓
ab{4,5}bc	abbbbc	✗
ab?bc	abbc	✓
ab?bc	abc	✓
ab{0,1}bc	abc	✓
ab?bc	abbbbc	✗
ab?c	abc	✓
ab{0,1}c	abc	✓
^abc$	abc	✗
^abc$	abcc	✗
^abc	abcc	✓
^abc$	aabc	✗
abc$	aabc	✓
abc$	aabcd	✓
^	abc	✓
$	abc	✗
a.c	abc	✗
a.c	axc	✗
a.*c	axyzc	✗
a.*c	axyzd	✗
a[bc]d	abc	✗
a[bc]d	abd	✓
a[b-d]e	abd	✗
a[b-d]e	ace	✓
a[b-d]	aac	✗
a[-b]	a-	✓

Κανονική έκφραση	**Δεδομένα**	**Αποτέλεσμα**
a]	a]	✗
a[]]b	a]b	✓
a[^bc]d	aed	✗
a[^bc]d	abd	✗
a[^-b]c	adc	✗
a[^-b]c	a-c	✗
\ba\b	a-	✗
\ba\b	-a	✗
\ba\b	-a-	✗
\by\b	xy	✗
\by\b	yz	✗
\by\b	xyz	✗
\Ba\B	a-	✓
\Ba\B	-a	✓
\Ba\B	-a-	✓
\By\b	xy	✗
\by\B	yz	✗
\By\B	xyz	✗
\w	a	✓
\w	-	✗
\W	a	✗
\W	-	✓
a\sb	a b	✗
a\sb	a-b	✗
a\Sb	a b	✓
a\Sb	a-b	✓
\d	1	✗
\d	-	✗
\D	1	✗
\D	-	✓
[\w]	a	✗
[\w]	-	✗
[\W]	a	✗
[\W]	-	✗
a[\s]b	a b	✗
a[\s]b	a-b	✗
a[\S]b	a b	✗
a[\S]b	a-b	✗
[\d]	1	✗
[\d]	-	✗
[\D]	7	✗
[\D]	-	✗
ab|cd	abc	✓
ab|cd	abcd	✓
()ef	def	✗
*a	-	✗

Κανονική έκφραση	**Δεδομένα**	**Αποτέλεσμα**
(*)b	-	✗
$b	b	✓
a\(b	a(b	✗
a\(*b	ab	✓
a\(*b	a((b	✓
a\\b	a\b	✓
((a))	abc	✓
(a)b(c)	abc	✗
a+b+c	aabbabc	✗
a{1,}b{1,}c	aabbabc	✗
a**	-	✓
a.+?c	abcabc	✓
(a+|b)*	ab	✓
(a+|b){0,}	ab	✓
(a+|b)+	ab	✓
(a+|b){1,}	ab	✓
(a+|b)?	ab	✓
(a+|b){0,1}	ab	✓
[^ab]*	cde	✓
abc		✗
a*		✓
([abc])*d	abbbcd	✓
([abc])*bcd	abcd	✓
a|b|c|d|e	e	✓
(a|b|c|d|e)f	ef	✓
abcd*efg	abcdefg	✓
ab*	xabyabbbz	✗
ab*	xayabbbz	✗
(ab|cd)e	abcde	✗
[abhgefdc]ij	hij	✓
^(ab|cd)e	abcde	✗
(a|b)c*d	abcd	✗
(ab|ab*)bc	abc	✓
a([bc]*)c*	abc	✓
a([bc]*)(c*d)	abcd	✓
a([bc]+)(c*d)	abcd	✓
a([bc]*)(c+d)	abcd	✓
a[bcd]*dcdcde	adcdcde	✓
a[bcd]+dcdcde	adcdcde	✗
(ab|a)b*c	abc	✓
((a)(b)c)(d)	abcd	✗
[a-zA-Z_][a-zA-Z0-9_]*	alpha	✓

Κανονική έκφραση	**Δεδομένα**	**Αποτέλεσμα**
^a(bc+|b[eh])g|.h$	abh	✗
(bc+d$|ef*g.|h?i(j|k))	effgz	✗
(bc+d$|ef*g.|h?i(j|k))	ij	✓
(bc+d$|ef*g.|h?i(j|k))	effg	✗
(bc+d$|ef*g.|h?i(j|k))	bcdd	✗
(bc+d$|ef*g.|h?i(j|k))	reffgz	✗
((((((((((a))))))))))	a	✓
((((((((((a))))))))))\01	aa	✗
(((((((((a)))))))))	a	✓
multiple words of text	uh-uh	✗
multiple words	multiple words, yeah	✗
(.*)c(.*)	abcde	✓
\((.*), (.*)\)	(a, b)	✗
[k]	ab	✗
abcd	abcd	✗
a(bc)d	abcd	✓
a[-]?c	ac	✓
(abc)\1	abcabc	✗
([a-c]*)\1	abcabc	✗
\1	-	✗
\2	-	✓
(a)|\1	a	✓
(a)|\1	x	✗
(a)|\2	-	✗
(([a-c])b*?\2)*	ababbbcbc	✗
(([a-c])b*?\2){3}	ababbbcbc	✗
((\3|b)\2(a)x)+	aaxabxbaxbbx	✗
((\3|b)\2(a)x)+	aaaxabaxbaaxbbax	✗
((\3|b)\2(a)){2,}	bbaababbabaaaaabbaaaabba	✗
'abc'i	ABC	✗

Σε μια προσπάθεια σχολιασμού των αποτελεσμάτων, θα μπορούσαμε να πούμε ότι ο μεταγλωττιστής F.i.r.e. καλύπτει αρκετές από τις ακραίες περιπτώσεις τις οποίες δοκιμάσαμε. Έχει ακόμα αρκετά προβλήματα, όσον αφορά τις εμφωλευμένες παρενθέσεις, ενώ αγνοεί κάποιους τελεστές όπως το $ και το ^.

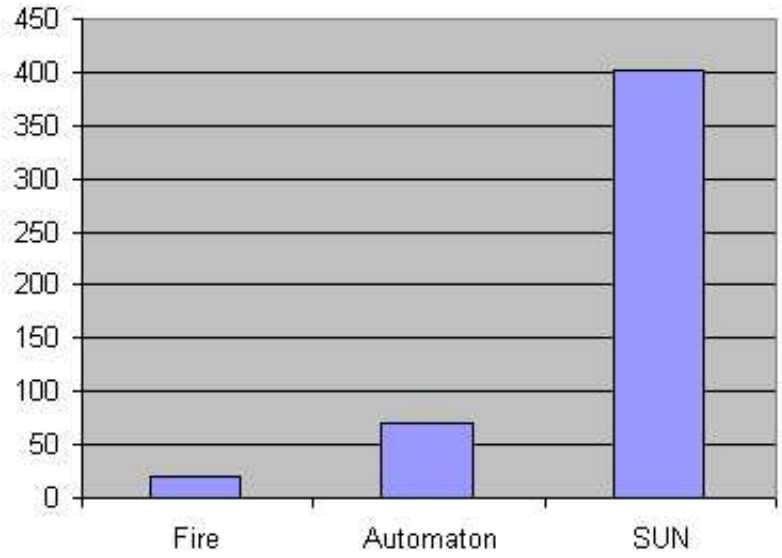
7.1.2 Σύγκριση της απόδοσης του μεταγλωττιστή F.i.r.e. με τις μηχανές κανονικών εκφράσεων σε Java

Για να συγκρίνουμε την απόδοση της F.i.r.e. επιλέξαμε τις δύο πιο αποδοτικές μηχανές κανονικών εκφράσεων, μέσα από εκείνες που αναλύσαμε στο κεφάλαιο 4. Αυτές είναι η μηχανή Automaton και η αντίστοιχη της SUN (java.util.regex).
Ο υπολογιστής στον οποίο εκτελούνται οι δοκιμές είναι ένας Pentium 4 2.53 GHz με 1Gb μνήμη.

Δοκιμή πρώτη

Κανονική έκφραση: $(a|b) * abb$
Δεδομένα: aaaabb
Επαναλήψεις: 300000

Όνομα βιβλιοθήκης	Χρόνος (msecs)
Fire	20
Automaton	71
Sun (java.util.regex)	430

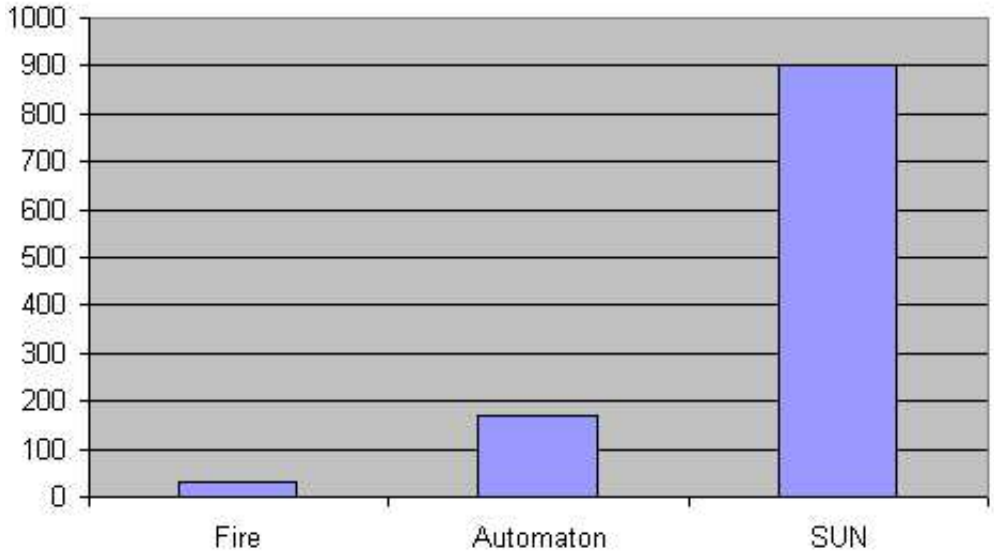


Σχήμα 7.1: Αποτέλεσμα πρώτης δοκιμής

Δοκιμή δεύτερη

Κανονική έκφραση: $(a|b) * abb$
Δεδομένα: aaaaaaaaaaaaaaabb
Επαναλήψεις: 300000

Όνομα βιβλιοθήκης	Χρόνος (msecs)
Fire	30
Automaton	170
Sun (java.util.regex)	901

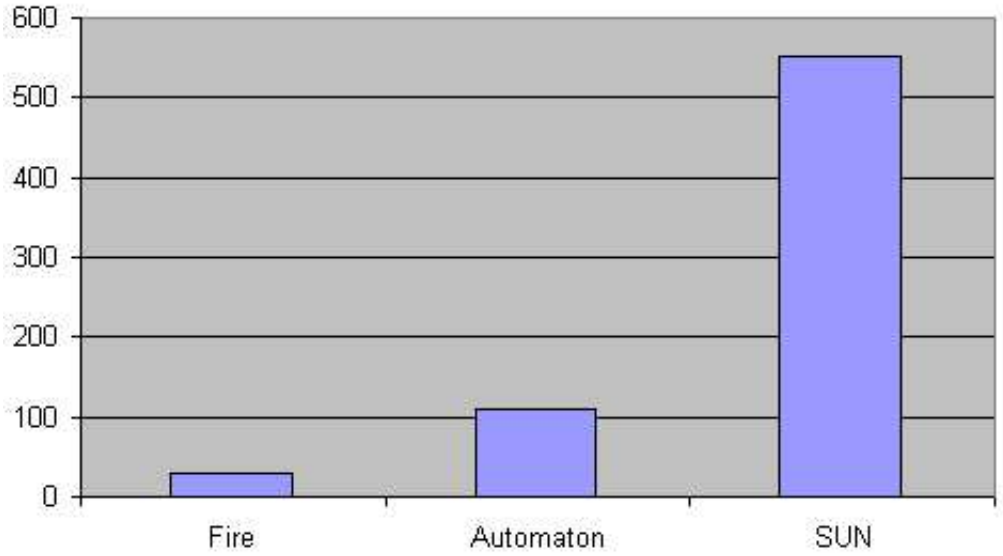


Σχήμα 7.2: Αποτέλεσμα δεύτερης δοκιμής

Δοκιμή τρίτη

Κανονική έκφραση: $(.*)abc(.*)$
Δεδομένα: aaaabcaaaa
Επαναλήψεις: 300000

Όνομα βιβλιοθήκης	Χρόνος (msecs)
Fire	30
Automaton	110
Sun (java.util.regex)	551

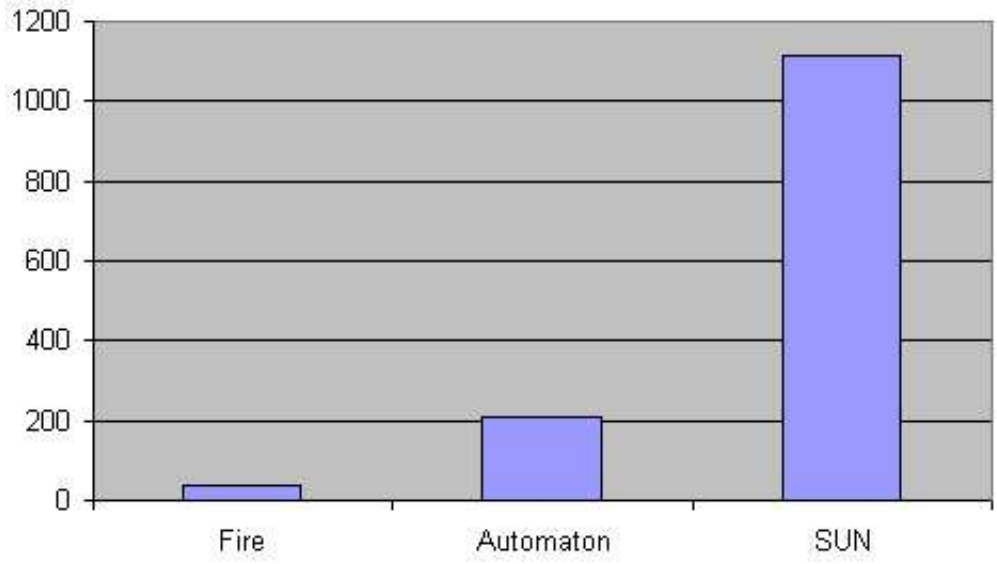


Σχήμα 7.3: Αποτέλεσμα τρίτης δοκιμής

Δοκιμή τέταρτη

Κανονική έκφραση: $(.*)abc(.*)$
Δεδομένα: aaaabcaaaa
Επαναλήψεις: 600000

Όνομα βιβλιοθήκης	Χρόνος (msecs)
Fire	40
Automaton	210
Sun (java.util.regex)	1112



Σχήμα 7.4: Αποτέλεσμα τέταρτης δοκιμής

Σχολιασμός δοκιμών

Στις δοκιμές ένα και δύο δοκιμάστηκε μια κανονική έκφραση με τελεστή greedy για σταθερό αριθμό επαναλήψεων. Ο στόχος ήταν να δούμε την διαφορά σε απόδοση στην περίπτωση που τα δεδομένα μεγαλώνουν. Στην επανάληψη δύο, παρόλο που τα δεδομένα ήταν διπλάσια η απόδοση του F.i.r.e. ανέβηκε μόλις 10 msecs ενώ στις άλλες μηχανές περίπου διπλασιάστηκε.

Στην δεύτερη ομάδα δοκιμών (τρία και τέσσερα) ανεβάσαμε την πολυπλοκότητα της κανονικής έκφρασης και των αριθμό των επαναλήψεων. Παρόλο που πλέον είχαμε δύο greedy τελεστές, ο μεταγλωττιστής F.i.r.e έκανε σχεδόν τον ίδιο χρόνο με την δοκιμή ένα. Στην δοκιμή τέσσερα διπλασιάσαμε τις επαναλήψεις και ο χρόνος ανέβηκε μόλις 10 msecs ενώ στις άλλες δύο μηχανές σχεδόν διπλασιάστηκε.

Οι βασικοί λόγοι που παρουσιάζεται τέτοια μεγάλη διαφορά σε ταχύτητα είναι οι παρακάτω:

1. Ο F.i.r.e λειτουργεί σαν Just-In time μεταγλωττιστής.
2. Το αυτόματο που χρησιμοποιείται είναι ντετερμινιστικό. Είναι σίγουρα πιο γρήγορο από το μη-ντετερμινιστικό της SUN, αλλά το ίδιο γρήγορο με το αυτόματο του Automaton.

3. Ο εκτελέσιμος κώδικας είναι assembly για την ιδεατή μηχανή της Java κατάλληλα βελτιστοποιημένος για την κάθε κανονική έκφραση.

7.2 Αποτελέσματα σύγκρισης

7.2.1 Τύπος μηχανής

Ο τύπος μηχανής του F.i.r.e. είναι υβριδικό DFA. Συνδυάζει χαρακτηριστικά και από τις δύο κατηγορίες.

7.2.2 Πρότυπο

Ακολουθείται το πρότυπο του POSIX. Αλλά έχει γίνει προεργασία για να υποστηριχθούν και χαρακτηριστικά του PCRE.

7.2.3 Unicode

Ο μεταγλωττιστής F.i.r.e. υποστηρίζει Unicode. Οι χαρακτήρες που χρησιμοποιεί είναι Unicode όπως ορίζονται στο Java Virtual Machine Specification.

7.2.4 Σχεδιασμός και ευελιξία

Ο μεταγλωττιστής είναι τμηματικά προγραμματιζόμενος. Έχει σχεδιαστεί να είναι ευέλικτος και επεκτάσιμος. Η ευελιξία του όμως δεν του στερεί επιδόσεις.

7.2.5 Απαιτήσεις σε πλατφόρμα Java

Ο μεταγλωττιστής αναπτύχθηκε στο πακέτο ανάπτυξης λογισμικού της Java έκδοση 1.4.2 (JDK). Δεν έχει προβλήματα με χαμηλότερες εκδόσεις της Java εφόσον δεν υπάρχει κάποια εξάρτηση με καμία έκδοση της Java.

7.2.6 Αξιοπιστία

Η υλοποίηση έγινε ακολουθώντας τα πρότυπα γραφής κώδικα που έχει ορίσει η SUN για την γλώσσα Java. Παρόλο που οι δοκιμές δείχνουν το αντίθετο, ο μεταγλωττιστής θέλει αρκετή δουλειά ακόμα. Αυτή την στιγμή θεωρείται καλή λύση για κάποιον που θέλει να τον χρησιμοποιήσει για κανονική χρήση ή για ερευνητικούς σκοπούς.

Κεφάλαιο 8

Συμπεράσματα

"Our situation has not improved."
- Henry Jones, Indiada Jones and the last crusade

8.1 Μελετώντας το παρελθόν

Οι κανονικές εκφράσεις έδειξαν την χρησιμότητα τους από την εμφάνιση του λειτουργικού συστήματος Unix. Μεγάλες μορφές του χώρου, όπως ο Ken Thompson από πολύ νωρίς άρχισαν να δουλεύουν πάνω στην χρήση των κανονικών εκφράσεων για την απλοποίηση πολύπλοκων προβλημάτων [Tho68]. Στην συνέχεια η ανάπτυξη κλασσικών εργαλείων που βασίζονταν στις κανονικές εκφράσεις όπως το grep, egrep, awk κτλ. έδωσαν τεράστια ώθηση στην ανάπτυξη τους και την βελτιστοποίηση των μεθόδων υλοποίησης τους.

Πολλές φορές όμως έγιναν και αντικείμενο αντιπαράθεσης για πολλούς επιστήμονες Πάρα πολλοί από αυτούς έχουν υλοποιήσει βιβλιοθήκες κανονικών εκφράσεων για λόγους αντιπαραθέσεις σε θέματα ταχύτητας και βελτιστοποίησης.

8.2 Αντιμετωπίζοντας το παρόν

Έχοντας το παρελθόν κατά νου, αντιμετωπίζουμε την παρούσα κατάσταση. Οι κανονικές εκφράσεις υπάρχουν πλέον πίσω σχεδόν από οποιαδήποτε εφαρμογή επεξεργάζεται κείμενο. Βιβλιοθήκες υπάρχουν για όλες τις γλώσσες προγραμματισμού και θεωρείται πλέον συνηθισμένη η χρήση τους.

θα μπορούσε κάποιος να επισημάνει ότι η επιστημονική περιοχή έχει επέλθει σε κορεσμό μετά από τόσα χρόνια έρευνας και ανάπτυξης. Η SUN δημιουργώντας την Java και υλοποιώντας μια πρότυπη βιβλιοθήκη κανονικών εκφράσεων για αυτή, έδωσε έδαφος για νέες αντιπαραθέσεις και βελτιστοποιήσεις.

Σε αυτή την αντιπαράθεση έλαβε μέρος και η παρούσα πτυχιακή δημιουργώντας τον F.i.r.e. , ένα Just-in time μεταγλωττιστή για την πλατφόρμα της Java. Στόχος της δημιουργίας του, η κάλυψη των κενών σε απόδοση που έχουν οι μέχρι τώρα βιβλιοθήκες, καθώς και η εμμονή όλων των υλοποιήσεων να χρησιμοποιούν μη-ντετερμινιστικά αυτόματα σαν πυρήνα τους.

Ο F.i.r.e. βασίζεται σε ντετερμινιστικό αυτόματο , γεγονός που του δίνει πολλά πλεονεκτήματα άλλα και μειονεκτήματα. Είναι ταχύτερος και 100% συμβατός με την αντίστοιχη μηχανή κανονικών εκφράσεων της SUN. Η υλοποίηση όμως έχει μεγαλύτερη πολυπλοκότητα από μια αντίστοιχη σε NFA (πχ για τη ανάπτυξη εμφωλευμένων παρενθέσεων ομαδοποίησης απαιτείται η επινόηση διαφόρων προγραμματιστικών τεχνάσματων, ενω με NFA είναι σχετικά πιο απλή η υλοποίηση).
Οι δοκιμές που έγιναν δικαιώνουν τις υποθέσεις μας, αλλά επίσης μας υποδηλώνουν ότι η έρευνα μόλις ξεκίνησε προς αυτή την κατεύθυνση.

8.3 Προσδοκώντας το μέλλον

Ο F.i.r.e είναι ακόμα πολύ ανώριμος για να χαρακτηριστεί τελικό "προϊόν". Θέλει αρκετές δοκιμές για να εξασφαλιστεί 100% η συμβατότητα με τα σύγχρονα πρότυπα, ενώ κάθε επιπλέον χαρακτηριστικό που θα αναπτύσσεται δεν θα πρέπει να επιβαρύνει την απόδοση του.

Με την επιτυχία του μεταγλωττιστή F.i.r.e ανοίγουν θέματα και για άλλες γλώσσες προγραμματισμού και πλατφόρμες, στις οποίες θα μπορούσαν να γίνουν παρόμοιες βελτιστοποιήσεις.

Παράρτημα Α΄

Γραμματική κανονικών εκφράσεων του προτύπου POSIX

```
%token    ORD_CHAR QUOTED_CHAR DUP_COUNT

%token    BACKREF L_ANCHOR R_ANCHOR

%token    Back_open_paren  Back_close_paren
/*          '\('             '\)'        */

%token    Back_open_brace  Back_close_brace
/*          '\{'             '\}'         */

/* The following tokens are for the Bracket Expression
   grammar common to both REs and EREs. */

%token    COLL_ELEM_SINGLE COLL_ELEM_MULTI META_CHAR

%token    Open_equal Equal_close Open_dot
/*           '[='       '=]'        '[.' */
%token Dot_close Open_colon Colon_close
/*            '.]'      '[:'       ':]'  */

%token    class_name
/* class_name is a keyword to the LC_CTYPE locale category */
/* (representing a character class) in the current locale */
/* and is only recognized between [: and :] */

%start    basic_reg_exp
```

```
%%

/* ------------------------------------------
   Basic Regular Expression
   ------------------------------------------
*/
basic_reg_exp  :          RE_expression
               | L_ANCHOR
               |                        R_ANCHOR
               | L_ANCHOR               R_ANCHOR
               | L_ANCHOR RE_expression
               |          RE_expression R_ANCHOR
               | L_ANCHOR RE_expression R_ANCHOR
               ;
RE_expression  :              simple_RE
               | RE_expression simple_RE
               ;
simple_RE      : nondupl_RE
               | nondupl_RE RE_dupl_symbol
               ;
nondupl_RE     : one_char_or_coll_elem_RE
               | Back_open_paren RE_expression Back_close_paren
               | BACKREF
               ;
one_char_or_coll_elem_RE : ORD_CHAR
               | QUOTED_CHAR
               | '.'
               | bracket_expression
               ;
RE_dupl_symbol : '*'
               | Back_open_brace DUP_COUNT Back_close_brace
               | Back_open_brace DUP_COUNT ',' Back_close_brace
               | Back_open_brace DUP_COUNT ',' DUP_COUNT
                 Back_close_brace
               ;

/* ------------------------------------------
   Bracket Expression
   ------------------------------------------
*/
bracket_expression : '[' matching_list ']'
```

```
               | '[' nonmatching_list ']'
               ;
matching_list  : bracket_list
               ;
nonmatching_list : '^' bracket_list
               ;
bracket_list   : follow_list
               | follow_list '-'
               ;
follow_list    :             expression_term
               | follow_list expression_term
               ;
expression_term : single_expression
               | range_expression
               ;
single_expression : end_range
               | character_class
               | equivalence_class
               ;
range_expression : start_range end_range
               | start_range '-'
               ;
start_range    : end_range '-'
               ;
end_range      : COLL_ELEM_SINGLE
               | collating_symbol
               ;
collating_symbol : Open_dot COLL_ELEM_SINGLE Dot_close
               | Open_dot COLL_ELEM_MULTI Dot_close
               | Open_dot META_CHAR Dot_close
               ;
equivalence_class : Open_equal COLL_ELEM_SINGLE Equal_close
               | Open_equal COLL_ELEM_MULTI Equal_close
               ;
character_class : Open_colon class_name Colon_close
               ;

/* ------------------------------------------
   Extended Regular Expression
   ------------------------------------------
*/
```

```
extended_reg_exp   :                        ERE_branch
                   | extended_reg_exp '|' ERE_branch
                   ;
ERE_branch         :            ERE_expression
                   | ERE_branch ERE_expression
                   ;
ERE_expression     : one_char_or_coll_elem_ERE
                   | '^'
                   | '$'
                   | '(' extended_reg_exp ')'
                   | ERE_expression ERE_dupl_symbol
                   ;
one_char_or_coll_elem_ERE  : ORD_CHAR
                   | QUOTED_CHAR
                   | '.'
                   | bracket_expression
                   ;
ERE_dupl_symbol    : '*'
                   | '+'
                   | '?'
                   | '{' DUP_COUNT               '}'
                   | '{' DUP_COUNT ','           '}'
                   | '{' DUP_COUNT ',' DUP_COUNT '}'
                   ;
```

Παράρτημα Β΄

Αποτελέσματα δοκιμών απόδοσης Damien Mascord

Τα αποτελέσματα υπάρχουν δημοσιευμένα στην διεύθυνση http://tusker.disorder.com.au/.

```
Regular expression library: org.apache.regexp.*
RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)

MS MAX AVG MIN DEV INPUT
191 40 0.0191 0 0 http://www.linux.com/
481 70 0.0481 0 1 http://www.thelinuxshow.com/main.php3
842 70 0.0842 0 1 usd 1234.00
1702 70 0.1702 0 1 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
40 10 0.0040 0 0 http://www.linux.com/
80 10 0.0080 0 0 http://www.thelinuxshow.com/main.php3
171 31 0.0171 0 0 usd 1234.00
191 31 0.0191 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
990 10 0.099 0 0 http://www.linux.com/
2843 11 0.2843 0 1 http://www.thelinuxshow.com/main.php3
3264 11 0.3264 0 1 usd 1234.00
4416 20 0.4416 0 2 he said she said he said no
```

```
Total time taken: 6349

Regular expression library: com.stevesoft.pat.Regex

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
311 20 0.0311 0 0 http://www.linux.com/
631 20 0.0631 0 0 http://www.thelinuxshow.com/main.php3
1182 20 0.1182 0 1 usd 1234.00
2624 20 0.2624 0 1 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
60 10 0.0060 0 0 http://www.linux.com/
171 11 0.0171 0 0 http://www.thelinuxshow.com/main.php3
211 11 0.0211 0 0 usd 1234.00
271 11 0.0271 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
812 11 0.0812 0 0 http://www.linux.com/
2274 20 0.2274 0 1 http://www.thelinuxshow.com/main.php3
2694 30 0.2694 0 1 usd 1234.00
3855 30 0.3855 0 1 he said she said he said no

Total time taken: 6930

Regular expression library: com.ibm.regex.RegularExpression

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
100 10 0.01 0 0 http://www.linux.com/
370 200 0.037 0 2 http://www.thelinuxshow.com/main.php3
500 200 0.05 0 2 usd 1234.00
961 200 0.0961 0 2 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
20 10 0.0020 0 0 http://www.linux.com/
20 10 0.0020 0 0 http://www.thelinuxshow.com/main.php3
60 10 0.0060 0 0 usd 1234.00
```

```
100 10 0.01 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
260 20 0.026 0 0 http://www.linux.com/
681 20 0.0681 0 0 http://www.thelinuxshow.com/main.php3
842 20 0.0842 0 0 usd 1234.00
1302 20 0.1302 0 1 he said she said he said no

Total time taken: 2614

Regular expression library: gnu.regexp.RE

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
6712 20 0.6712 0 2 http://www.linux.com/
15789 30 1.5789 0 3 http://www.thelinuxshow.com/main.php3
21119 30 2.1119 0 4 usd 1234.00
33398 30 3.3398 0 4 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
10 10 0.0010 0 0 http://www.linux.com/
20 10 0.0020 0 0 http://www.thelinuxshow.com/main.php3
1242 11 0.1242 0 1 usd 1234.00
1262 11 0.1262 0 1 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
861 11 0.0861 0 0 http://www.linux.com/
1662 11 0.1662 0 1 http://www.thelinuxshow.com/main.php3
2643 20 0.2643 0 1 usd 1234.00
3155 20 0.3155 0 1 he said she said he said no

Total time taken: 37965

Regular expression library: kmy.regex.util.Regex

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
300 90 0.03 0 1 http://www.linux.com/
```

```
660 90 0.066 0 1 http://www.thelinuxshow.com/main.php3
1560 90 0.156 0 1 usd 1234.00
3976 90 0.3976 0 2 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
10 10 0.0010 0 0 http://www.linux.com/
80 10 0.0080 0 0 http://www.thelinuxshow.com/main.php3
180 10 0.018 0 0 usd 1234.00
220 10 0.022 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
291 20 0.0291 0 0 http://www.linux.com/
501 20 0.0501 0 0 http://www.thelinuxshow.com/main.php3
671 20 0.0671 0 0 usd 1234.00
831 20 0.0831 0 0 he said she said he said no

Total time taken: 5347

Regular expression library: java.util.regex.Pattern

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
151 11 0.0151 0 0 http://www.linux.com/
301 11 0.0301 0 0 http://www.thelinuxshow.com/main.php3
441 11 0.0441 0 0 usd 1234.00
751 11 0.0751 0 0 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
30 10 0.0030 0 0 http://www.linux.com/
80 10 0.0080 0 0 http://www.thelinuxshow.com/main.php3
121 11 0.0121 0 0 usd 1234.00
141 11 0.0141 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
60 10 0.0060 0 0 http://www.linux.com/
130 10 0.013 0 0 http://www.thelinuxshow.com/main.php3
200 10 0.02 0 0 usd 1234.00
```

```
270 10 0.027 0 0 he said she said he said no

Total time taken: 1172

Regular expression library: jregex.Pattern

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
312 151 0.0312 0 1 http://www.linux.com/
452 151 0.0452 0 1 http://www.thelinuxshow.com/main.php3
612 151 0.0612 0 1 usd 1234.00
942 151 0.0942 0 1 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
30 10 0.0030 0 0 http://www.linux.com/
110 10 0.011 0 0 http://www.thelinuxshow.com/main.php3
150 10 0.015 0 0 usd 1234.00
170 10 0.017 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
50 10 0.0050 0 0 http://www.linux.com/
120 10 0.012 0 0 http://www.thelinuxshow.com/main.php3
190 10 0.019 0 0 usd 1234.00
270 10 0.027 0 0 he said she said he said no

Total time taken: 1552

Regular expression library:
                org.apache.oro.text.regex.Perl5Matcher

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
130 10 0.013 0 0 http://www.linux.com/
220 10 0.022 0 0 http://www.thelinuxshow.com/main.php3
731 20 0.0731 0 0 usd 1234.00
1912 80 0.1912 0 1 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
```

```
30 10 0.0030 0 0 http://www.linux.com/
40 10 0.0040 0 0 http://www.thelinuxshow.com/main.php3
70 10 0.0070 0 0 usd 1234.00
110 10 0.011 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
71 11 0.0071 0 0 http://www.linux.com/
201 11 0.0201 0 0 http://www.thelinuxshow.com/main.php3
251 11 0.0251 0 0 usd 1234.00
321 11 0.0321 0 0 he said she said he said no
Total time taken: 2404

Regular expression library: RegularExpression.RE

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
201 11 0.0201 0 0 http://www.linux.com/
441 20 0.0441 0 0 http://www.thelinuxshow.com/main.php3
661 20 0.0661 0 0 usd 1234.00
841 20 0.0841 0 0 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
190 10 0.019 0 0 http://www.linux.com/
351 11 0.0351 0 0 http://www.thelinuxshow.com/main.php3
3424 20 0.3424 0 1 usd 1234.00
3585 20 0.3585 0 1 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
110 10 0.011 0 0 http://www.linux.com/
240 10 0.024 0 0 http://www.thelinuxshow.com/main.php3
381 11 0.0381 0 0 usd 1234.00
491 11 0.0491 0 0 he said she said he said no

Total time taken: 4967

Regular expression library: dk.brics.automaton.RegExp

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
```

```
MS MAX AVG MIN DEV INPUT
0 0 0.0 0 0 http://www.linux.com/
0 0 0.0 0 0 http://www.thelinuxshow.com/main.php3
40 10 0.0040 0 0 usd 1234.00
50 10 0.0050 0 0 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
0 0 0.0 0 0 http://www.linux.com/
10 10 0.0010 0 0 http://www.thelinuxshow.com/main.php3
110 10 0.011 0 0 usd 1234.00
130 10 0.013 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
10 10 0.0010 0 0 http://www.linux.com/
40 10 0.0040 0 0 http://www.thelinuxshow.com/main.php3
40 10 0.0040 0 0 usd 1234.00
50 10 0.0050 0 0 he said she said he said no

Total time taken: 541

Regular expression library:
             com.karneim.util.collection.regex.Pattern

RE: ^(([^:]+)://)?([^:/]+)(:([0-9]+))?(/.*)
MS MAX AVG MIN DEV INPUT
20 10 0.0020 0 0 http://www.linux.com/
40 10 0.0040 0 0 http://www.thelinuxshow.com/main.php3
50 10 0.0050 0 0 usd 1234.00
80 10 0.0080 0 0 he said she said he said no

RE: usd [+-]?[0-9]+.[0-9][0-9]
MS MAX AVG MIN DEV INPUT
10 10 0.0010 0 0 http://www.linux.com/
10 10 0.0010 0 0 http://www.thelinuxshow.com/main.php3
101 11 0.0101 0 0 usd 1234.00
111 11 0.0111 0 0 he said she said he said no

RE: \b(\w+)(\s+\1)+\b
MS MAX AVG MIN DEV INPUT
```

```
10 10 0.0010 0 0 http://www.linux.com/
30 10 0.0030 0 0 http://www.thelinuxshow.com/main.php3
50 10 0.0050 0 0 usd 1234.00
70 10 0.0070 0 0 he said she said he said no

Total time taken: 481
```

Παράρτημα Γ΄

Παραγόμενο Java αρχείο από το τμήμα μεταγλώττισης JavaSourceWriter

```
import org.fire.regexp.*;
import org.fire.regexp.writers.abstractMatcher.*;

public class lnatmapr extends CharArrayMatcher{
private final boolean[] state_arr = {false,false,true,false};

public lnatmapr(){
super("(a|b)*abb");
this.regex = "(a|b)*abb";
this.state = 3;
}

public boolean find(){
if(current + 1 == length){ return false; }
char nextChar = '0';
start = current;
lastMatch = -1;
state = 3;
for(current = start;current < length;current++){
nextChar = arrayBuffer[current];
switch(state){
case 0:
 if(nextChar == 'a'){
state = 1; break;
```

```
}else if(nextChar == 'b'){
state = 2; break;
}else { if(current < length - 1){
 if(lastMatch == -1) { current++; find(); }
 else{ return true; } }else{
 return (lastMatch != -1); }
}
case 1:
 if(nextChar == 'b'){
state = 0; break;
}else if(nextChar == 'a'){
state = 1; break;
}else { if(current < length - 1){
 if(lastMatch == -1) { current++; find(); }
 else{ return true; } }else{
 return (lastMatch != -1); }
}
case 2:
lastMatch = current;
 if(nextChar == 'a'){
state = 1; break;
}else if(nextChar == 'b'){
state = 3; break;
}else { return true;}
case 3:
 if(nextChar == 'a'){
state = 1; break;
}else if(nextChar == 'b'){
state = 3; break;
}else { if(current < length - 1){
 if(lastMatch == -1) { current++; find(); }
 else{ return true; } }else{
 return (lastMatch != -1); }
}
}
}

if(state_arr[state]){ lastMatch = current;
return true; }else{ return false;}
}
public boolean matches(){
```

```
char nextChar = '0';
state = 3;
start = 0;
for(current = 0;current < length;current++){
nextChar = arrayBuffer[current];
switch(state){
case 0:
 if(nextChar == 'a'){
 state = 1; break;
}else if(nextChar == 'b'){
 state = 2; break;
}else { return false; }
case 1:
 if(nextChar == 'a'){
 state = 1; break;
}else if(nextChar == 'b'){
 state = 0; break;
}else { return false; }
case 2:
 if(nextChar == 'a'){
 state = 1; break;
}else if(nextChar == 'b'){
 state = 3; break;
}else { break; }
case 3:
 if(nextChar == 'b'){
 state = 3; break;
}else if(nextChar == 'a'){
 state = 1; break;
}else { return false; }
}
}

return state_arr[state];
}
}
```

Παράρτημα Δ΄

Παραγόμενο Java αρχείο από το τμήμα μεταγλώττισης JavaAssemblyWriter

```
.class public lnatmapr
.super org/fire/regexp/writers/abstractMatcher/CharArrayMatcher

; constructor
.method public <init>()V
.limit stack 5
aload_0
ldc "(a|b)*abb"
invokenonvirtual
org/fire/regexp/writers/abstractMatcher/
CharArrayMatcher/<init>(Ljava/lang/String;)V
aload_0
ldc "(a|b)*abb"
putfield org/fire/regexp/Matcher/regex Ljava/lang/String;
aload_0
bipush 3
putfield org/fire/regexp/Matcher/start I

return
.end method


; find
.method public find()Z
```

```
.limit stack 10
.limit locals 5

aload_0
bipush 3
putfield org/fire/regexp/Matcher/state I
aload_0
getfield
org/fire/regexp/writers/abstractMatcher/CharArrayMatcher/current I
istore_1
iload_1
bipush 0
if_icmpeq CONTINUE
aload_0
iload_1
putfield org/fire/regexp/Matcher/start I

CONTINUE:
aload_0
getfield
org/fire/regexp/writers/abstractMatcher/CharArrayMatcher/length I
istore 4
iinc 4 -1
iload 4
iload_1
if_icmpeq STATE_ERROR
aload_0
getfield org/fire/regexp/Matcher/start I
aload_0
swap
putfield
org/fire/regexp/writers/abstractMatcher/CharArrayMatcher/current I
aload_0
getfield
org/fire/regexp/writers/abstractMatcher/CharArrayMatcher/current I
istore_1
aload_0
getfield
org/fire/regexp/writers/abstractMatcher
/CharArrayMatcher/arrayBuffer [C
astore_2
```

```
goto STATE_3

STATE_0:
aload_2
iload_1
caload
istore_3
iinc 1 1
iload_1
iload 4
if_icmpge STATE_ERROR
bipush 97
iload_3
if_icmpeq STATE_1
bipush 98
iload_3
if_icmpeq STATE_2
goto STATE_ERROR

STATE_1:
aload_2
iload_1
caload
istore_3
iinc 1 1
iload_1
iload 4
if_icmpge STATE_ERROR
bipush 98
iload_3
if_icmpeq STATE_0
bipush 97
iload_3
if_icmpeq STATE_1
goto STATE_ERROR

STATE_2:
aload_2
iload_1
caload
istore_3
```

```
iinc 1 1
iload_1
iload 4
if_icmpge STATE_ERROR
bipush 97
iload_3
if_icmpeq STATE_1
bipush 98
iload_3
if_icmpeq STATE_3
goto STATE_OK

STATE_3:
aload_2
iload_1
caload
istore_3
iinc 1 1
iload_1
iload 4
if_icmpge STATE_ERROR
bipush 97
iload_3
if_icmpeq STATE_1
bipush 98
iload_3
if_icmpeq STATE_3
goto STATE_ERROR

STATE_OK:
aload_0
iload_1
putfield org/fire/regexp/writers/
abstractMatcher/CharArrayMatcher/current I
iconst_1
ireturn

STATE_ERROR:
iconst_0
ireturn
.end method
```

```
; matches
.method public matches()Z
iconst_0
ireturn
.end method
```

Βιβλιογραφία

[Anten] Αντώνιος Παναγιωτόπουλος. *Διακριτά Μαθηματικά*. Εκδόσεις Σταμουλης, Αθήνα, 1992.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1985.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[Big02] Norman L. Biggs. *Discrete Mathematics, second edition*. Oxford Universty Press, Great Clarendon Street, Oxford OX2 6DP, 2002.

[BLSS] Michael Benedikt, Leonid Libkin, Thomas Schwentick, and Luc Segoufin. Definable relations and first-order query languages over strings. http://citeseer.nj.nec.com/585309.html.

[CC97] Charles L. A. Clarke and Gordon V. Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19(3):413–426, May 1997.

[Cho56] Noam Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956.

[Dah01] Markus Dahn. Byte code engineering library (bcel), description and usage manual, version 1.0. www, October 2001. http://jakarta.apache.org/bcel/.

[EM] Andrew Eisenberg and Jim Melton. Sql:1999, formerly known as sql3. http://dbs.uni-leipzig.de/en/lokal/standards.pdf.

[Enc03a] Wikipedia The Free Encyclopedia. Chomsky hierarchy. www, 2003. http://en2.wikipedia.org/wiki/Chomsky_hierarchy.

[Enc03b] Wikipedia The Free Encyclopedia. Formal grammar. www, 2003. http://en2.wikipedia.org/wiki/Formal_grammar.

[Enc03c] Wikipedia The Free Encyclopedia. Regular expressions. www, 2003. http://en2.wikipedia.org/wiki/Regular_expressions.

[Fou] GNU Free Software Foundation. Kawa, the java-based scheme system. http://www.gnu.org/software/kawa/.

[Fri02] Jeffrey E.F. Friedl. *Mastering Regular Expressions 2nd Edition*. O'Reilly and Associates, Inc., Sebastopol, CA, 2002.

[ge03] Βασίλειος Καρακόιδας. Μεταγλωττιστής κανονικών εκφράσεων σε java. *MSc στα πληροφοριακά συστήματα - Παραδοτέο εργασίας, τμήμα πληροφορικής, Ηλεκτρονικό εμπόριο* , 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Indianapolis, IN 46920, 1994.

[IG03] IEEE and The Open Group. The single unix specification, version 3. www, 2003. http://www.unix.org/version3/.

[Inc03] Sun Microsystems Inc. Java 2 sdk, standard edition documentation. WWW, June 2003. http://java.sun.com/j2se/1.4.2/docs/index.html.

[LY97] Tim Lindhorn and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, 1997.

[Mas02] Damien Mascord. Java regular expression library benchmarks. WWW, 2002. http://tusker.disorder.com.au/.

[Moe03] Anders Moeller. Automaton library generated javadocs. WWW, May 2003. http://www.brics.dk/automaton/doc/index.html.

[Spi03] Diomidis Spinellis. On the declarative specification of models. *IEEE Software*, 20(2):71–77, January 2003.

[ST03] A. Syropoulos and A. Tsolomitis. The kerkis font family, version 2.0. www, January 2003. http://iris.math.aegean.gr/kerkis/.

[Ste98] W. Richard Stevens. *TCP/IP Illustrated Volume 1 - The Protocols*. Addison-Wesley, One Jacob Way - Reading, 1998.

[Tho68] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, Sebastopol, CA, 2000.

[WHA02] Damien Watkins, Mark Hammond, and Brad Abrams. *Programming in the .NET Environment*. Addison-Wesley, Reading, MA, 2002.