Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Πειράματα και Μετρήσεις στον Συντακτικό Αναλυτή Της Μηχανής V8

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ Ι. ΣΚΛΗΚΑΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2018

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Πειράματα και Μετρήσεις στον Συντακτικό Αναλυτή Της Μηχανής V8

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ Ι. ΣΚΛΗΚΑΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 6η Ιουλίου 2018.

.....	.....	.....
Νικόλαος Σ. Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Ι. Γκούμας Επίκ. Καθηγητής Ε.Μ.Π.	Αριστείδης Τ. Παγουρτζής Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2018

Νικόλαος Ι. Σκλήκας

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

Copyright © Νικόλαος Ι. Σκλήκας, 2018.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Σκοπός της παρούσας εργασίας είναι η μέτρηση του χρόνου που καταναλώνει η μηχανή V8 για τον εντοπισμό συντακτικών λαθών και η αποφυγή των επαναλαμβανόμενων προσπαθειών εντοπισμού λαθών στα ίδια κομμάτια κώδικα ώστε να μειωθεί ο συνολικός χρόνος που χρειάζεται η ανάλυση.

Τη σημερινή εποχή η γλώσσα JavaScript είναι μια από τις πιο ευρέως χρησιμοποιημένες γλώσσες προγραμματισμού. Μηχανές JavaScript χρησιμοποιούνται σε ένα μεγάλο εύρος εφαρμογών από περιηγητές ιστού μέχρι πλατφόρμες ανάπτυξης εφαρμογών. Ευτυχώς οι σύγχρονες μηχανές JavaScript είναι πολύ πιο γρήγορες από ότι στο παρελθόν και γίνονται πολλές προσπάθειες για να βελτιωθεί ακόμα περισσότερο η απόδοση τους.

Σε αυτή την εργασία ασχοληθήκαμε με τον συντακτικό αναλυτή της μηχανής V8. Ο V8 είναι μία από τις ταχύτερες μοντέρνες μηχανές JavaScript και είναι ένας πάνω στο χρόνο μεταγλωττιστής. Συγκεκριμένα ο συντακτικός αναλυτής του είναι ένας αναλυτής αναδρομικής καθόδου. Το πρόβλημα που προσπαθήσαμε να αντιμετωπίσουμε είναι ότι κάποια κομμάτια κώδικα περνάνε πολλές φορές από τον αναλυτή με συνέπεια να γίνεται πολλές φορές η προσπάθεια εντοπισμού συντακτικών λαθών. Η επανειλημμένη προσπάθεια εντοπισμού συντακτικών λαθών είναι πολύ συχνή όταν ο V8 χρησιμοποιείται από τον περιηγητή Google Chrome, το οποίο είναι η πιο συνήθης χρήση του, καθώς η ανανέωση ή η επαναλαμβανόμενη επίσκεψη κάποιας ιστοσελίδας προκαλεί την ανάλυση του ίδιου κώδικα. Η συχνότητας της επαναλαμβανόμενης επίσκεψης κάποιας ιστοσελίδας γίνεται εύκολα αντιληπτή αν σκεφτεί κανείς το πλήθος των ιστοσελίδων που επισκέπτεται σε σχέση με το χρόνο που περνά στο διαδίκτυο.

Για να αποδείξουμε την πρακτικότητα και τα οφέλη της προσέγγισης μας αφού αφαιρέσαμε τον εντοπισμό λαθών από τον συντακτικό αναλυτή του V8, χτίσαμε τον Chrome χρησιμοποιώντας τον αλλαγμένο V8 και μετρήσαμε την επιτάχυνση που είχαμε όταν επισκεπτόμασταν τις πιο δημοφιλείς ιστοσελίδες, οι οποίες ξέραμε ότι δεν είχαν συντακτικά λάθη στη JavaScript. Τέλος κάνοντας μικρές αλλαγές στον κώδικα προσαρμόσαμε τον αναλυτή για να κάνει έλεγχο για συντακτικά λάθη μόνο την πρώτη φορά που συναντάει κάποιο κομμάτι κώδικα πετυχαίνοντας επιτάχυνση έως και 1,3% στο χρόνο φόρτωσης της σελίδας.

Λέξεις κλειδιά

Γλώσσες προγραμματισμού, συντακτική ανάλυση, συντακτικά λάθη, JavaScript, γλώσσα σεναρίων, μηχανή V8.

Abstract

The purpose of this diploma dissertation is to measure the time spent by the V8 engine to detect syntax errors and to avoid repetitive attempts to detect syntax errors in the same piece of code in order to reduce the total time of parsing.

Today JavaScript is one of the most widespread used programming languages. JavaScript engines are used in a wide range of applications from web browsers to application development platforms. Fortunately, the JavaScript engines are much faster than in the past and many efforts are being made to further improve their performance.

In this we dealt with the V8 engine's parser. The V8 is one of the fastest modern JavaScript engines and it is a just-in-time compiler. Specifically the parser is a recursive descent parser. The problem that we tried to deal with is that some pieces of code are parsed many times and therefore are checked many times for syntax errors many times. Repeated syntax error detection is very common when V8 is used by the Google Chrome browser, which is its most common use, as refreshing or revisiting a webpage causes the re-parsing of the same code. The frequency of a recurring visit of some website is quite large, considering the number of websites someone visits in relation to time he spents on the internet.

To prove the practicality and benefits of our approach, after we removed syntactic error detection from the V8 parser we built Chrome using the altered V8 and we measured the acceleration that we achieved when we visited the most popular websites of the internet, which were error free, using the tool WebPageReplay. Finally, making small changes to the code we adjusted the parser to check for syntax errors only the first time it encounters a piece of code achieving speedup of up to 1.3% on page load time.

Key words

Programming languages, parsing, syntax errors, JavaScript, scripting language, V8 engine.

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διατριβής, κ. Νικόλαο Παπασπύρου, για τη συνεχή καθοδήγηση και εμπιστοσύνη του. Θέλω να ευχαριστήσω ακόμα τον συμφοιτητή και φίλο Πέτρο Πετρόπουλο, ο οποίος με βοήθησε σε διάφορα στάδια αυτής της εργασίας. Θα ήθελα τέλος να ευχαριστήσω την οικογένειά μου και κυρίως τους γονείς μου, οι οποίοι με υποστήριξαν και έκαναν δυνατή την απερίσπαστη ενασχόλησή μου τόσο με την εκπόνηση της διπλωματικής μου, όσο και συνολικά με τις σπουδές μου.

Νικόλαος Ι. Σκλήκας,

Αθήνα, 6η Ιουλίου 2018

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-03-18, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2018.

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

Περιεχόμενα

Κατάλογος πινάκων

Κατάλογος σχημάτων

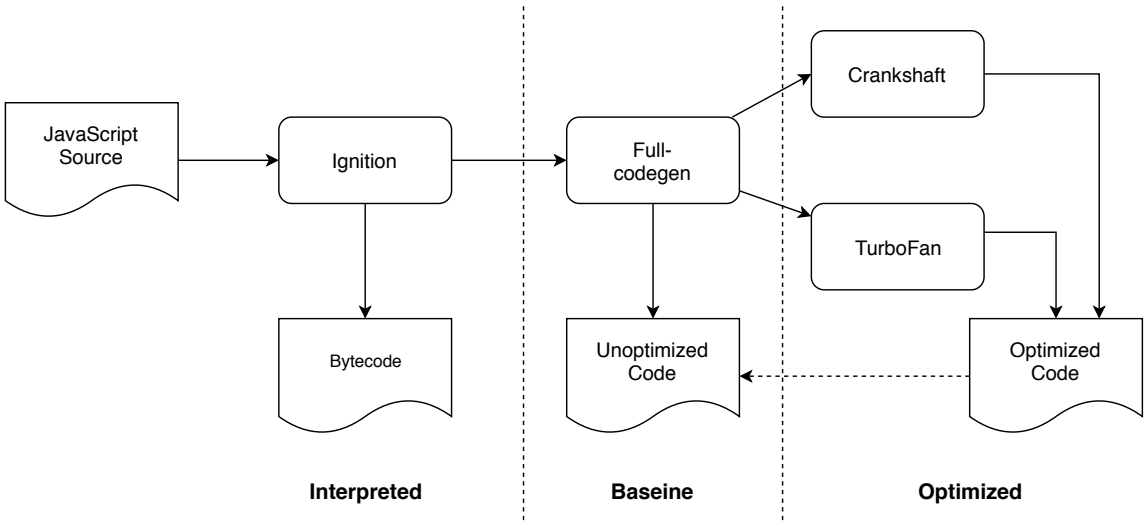
Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Σε αυτή την εργασία ασχοληθήκαμε με τη μηχανή JavaScript V8, η οποία χρησιμοποιείται από τον περιηγητή Google Chrome, συγκεκριμένα ασχοληθήκαμε με το συντακτικό αναλυτή του V8. Σύμφωνα με μετρήσεις της Google που έγιναν το 2017 μια τυπική διαδικτυακή εφαρμογή φορτώνει περίπου 0,4MB JavaScript. Σύμφωνα με την ίδια έρευνα ο V8 χρειαζόταν περίπου 370ms για να κάνει συντακτική ανάλυση, δηλαδή είχε ταχύτητα περίπου 1 MB/s.[Osma17] Σύμφωνα με μετρήσεις που έγιναν χρησιμοποιώντας το εργαλείο WebPageReplay της Google πάνω σε πραγματικές ιστοσελίδες γύρω στο 15-20% του χρόνου που χρειάζεται για να φορτώσουν περνιέται στον συντακτικό αναλυτή του V8. Από τα παραπάνω νούμερα φαίνεται ο συντακτικός αναλυτής είναι μια πολύ σημαντική συνιστώσα του V8.

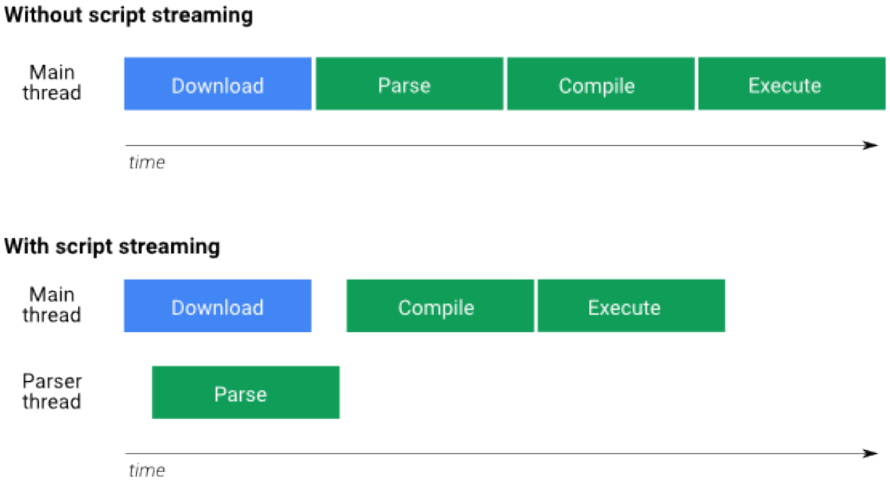
Για αυτό έχουν γίνει πολλές πολλές προσπάθειες για να βελτιώθει ο συντακτικός αναλυτής. Παλαιότερα η δομή του V8 ήταν διαφορετική, η δομή αυτή φαίνεται στο σχήμα 1.1[McIl16], και επειδή χρειαζόταν ο πηγαίος κώδικας τόσο στο Full-codegen όσο και στο Crankshaft ο συντακτικός αναλυτής έκανε ανάλυση στα ίδια κομμάτια κώδικα, όταν έπρεπε να γίνει βελτιστοποίηση/αποβελτιστοποίηση τους, κάτι το οποίο άλλαξε στον V8 5.9[Clif17] και έδωσε σημαντική επιτάχυνση αλλά και βελτίωση επίσης και της ανάγκες του V8 από άποψη μνήμης.



Σχήμα 1.1: Η δομή του V8 μέχρι την έκδοση 5.9

Επιπλέον ξεκινώντας από το Chrome 41 η συντακτική ανάλυση ξεκινάει σε ένα νήμα στο περιθώριο πριν να ληφθεί ολόκληρη η ιστοσελίδα[Höl15] έτσι η συντακτική ανάλυση τελειώνει αισθητά πιο

γρήγορα καθώς εκμεταλλεύεται χρόνο κατά τον οποίο πριν δεν έκανε τίποτα, όπως φαίνεται και στο σχήμα 1.2.



Σχήμα 1.2: Η έναρξη της ανάλυσης σε ένα νήμα στο παρασκήνιο έδωσε σημαντική επιτάχυνση.

Τέλος ο Chrome, όπως όλοι οι σύγχρονοι περιηγητές, κρατάει σε ένα αρχείο στατικά δεδομένα (JavaScript, CSS) από τις ιστοσελίδες που επισκέπτεται για να μην χρειάζεται να τα ξαναζητήσει αν ο χρήστης ξαναεπισκεφτεί κάποια ιστοσελίδα, ξεκινώντας από την έκδοση 4.2 ο V8 δίνει την επιλογή να παράγει και να αποθηκεύει δεδομένα για την μεταγλώττιση τα οποία μπορεί να χρησιμοποιήσει αν του ξαναζητηθεί να μεταγλωττίσει τον ίδιο κώδικα, ώστε να αναπαράγει το αποτέλεσμα της προηγούμενης μεταγλώττισης χωρίς να χρειαστεί να μεταγλωττίσει από το μηδέν [Guo15]. Μάλιστα αυτή η τεχνική έδωσε τόσο καλά αποτελέσματα που στον Chrome 66 αποφάσισαν να αυξήσουν το ποσοστό των δεδομένων που αποθηκεύονται όταν ο χρήστης επισκέπτεται πολλές φορές μια ιστοσελίδα.

Η προσέγγιση μας είναι διαφορετική από τα παραπάνω καθώς οι παραπάνω προσεγγίσεις προσπαθούν να αποφύγουν εντελώς την κλήση του αναλυτή, εμείς αντιθέτως προσπαθήσαμε να επιταχύνουμε την ανάλυση. Η ιδέα μας είναι ότι μπορεί ο Chrome να κρατά δεδομένα όταν επισκέπτεται μια ιστοσελίδα πολλές φορές για να αποφεύγει τη συντακτική ανάλυση και να ξεκινά αμέσως τη μεταγλώττιση, όμως αυτό το κάνει για ένα μικρό αριθμό συναρτήσεων, μάλλον επειδή όσα περισσότερα κρατάει τόσο περισσότερη μνήμη χρειάζεται. Για τις υπόλοιπες συναρτήσεις, τις οποίες δεν τις αποθηκεύει στη μνήμη ο V8 είναι αναγκασμένος να ακολουθήσει όλη τη διαδικασία της μεταγλώττισης, όπως περιγράφηκε προηγουμένως, δηλαδή θα πρέπει να ξανακάνει συντακτική ανάλυση και να ψάξει για λάθη στον κώδικα. Όμως αφού έχει ξανά- αναλύσει τον κώδικα θα έπρεπε να ξέρει αν υπάρχουν συντακτικά λάθη ή όχι. Σκοπός της εργασίας είναι να μετρήσουμε την επιτάχυνση που μπορούμε να πετύχουμε αν απενεργοποιήσουμε τον εντοπισμό συντακτικών λαθών για κώδικα (ή κομμάτια κώδικα) που έχουμε ξανασυναντήσει και να παρουσιάσουμε μια βασική υλοποίηση του V8 που να απενεργοποιεί τον εντοπισμό συντακτικών λαθών αν ξέρει πως ο κώδικας δεν έχει λάθη.

1.2 Σύνοψη της εργασίας

Παρακάτω θα παρουσιάσουμε συνοπτικά τη γλώσσα JavaScript, τη μηχανή V8, τις μετατροπές που κάναμε. Συγκεκριμένα η δομή της εργασίας είναι η ακόλουθη:

Κεφάλαιο 2. Περιγραφή της γλώσσας JavaScript και του V8

Κεφάλαιο 3. Περιγραφή του συντακτικού αναλυτή του V8

Κεφάλαιο 4. Περιγραφή της υλοποίησης των αλλαγών που κάναμε

Κεφάλαιο 5. Παρουσίαση των αποτελεσμάτων μας

Κεφάλαιο 6. Συμπεράσματα και προτάσεις μελλοντικής επέκτασης

Κεφάλαιο 2

Η γλώσσα JavaScript και η μηχανή V8

Η JavaScript είναι μια ερμηνευμένη (interpreted) ή πάνω στο χρόνο μεταγλωττισμένη (JIT compiled) γλώσσα προγραμματισμού με συναρτήσεις πρώτης κατηγορίας (first-class functions). Ενώ είναι πιο γνωστή ως η γλώσσα σεναρίου για ιστοσελίδες, χρησιμοποιείται και σε πολλά περιβάλλοντα πέρα από τους περιηγητές όπως node.js και Apache CouchDB. Πρόκειται για μια πρωτότυπη, πολλαπλών υποδειγμάτων scripting γλώσσα προγραμματισμού που είναι δυναμική, και υποστηρίζει αντικειμενοστρεφή, προστακτικό, και δηλωτικό στυλ προγραμματισμού.[MDN c] Το πρότυπο της JavaScript ονομάζεται ECMAScript.

2.1 Ιστορία

Η γλώσσα προγραμματισμού JavaScript δημιουργήθηκε αρχικά από τον Brendan Eich της εταιρείας Netscape με την επωνυμία Mocha. Αργότερα, Mocha μετονομάστηκε σε LiveScript, και τελικά σε JavaScript, κυρίως επειδή η ανάπτυξή της επηρεάστηκε περισσότερο από τη γλώσσα προγραμματισμού Java. LiveScript ήταν το επίσημο όνομα της γλώσσας όταν για πρώτη φορά κυκλοφόρησε στην αγορά σε beta έκδοση με το πρόγραμμα περιήγησης στο Web, Netscape Navigator εκδοχή 2.0 τον Σεπτέμβριο του 1995. LiveScript μετονομάστηκε σε JavaScript σε μια κοινή ανακοίνωση με την εταιρεία Sun Microsystems στις 4 Δεκεμβρίου, 1995, όταν επεκτάθηκε στην έκδοση του προγράμματος περιήγησης στο Web, Netscape εκδοχή 2.0B3.

Η JavaScript απέκτησε μεγάλη επιτυχία ως γλώσσα στην πλευρά του πελάτη για εκτέλεση κώδικα σε ιστοσελίδες, και περιλήφθηκε σε διάφορα προγράμματα περιήγησης στο Web, ώστε τα σενάρια του πελάτη να μπορούν να επικοινωνούν με τον χρήστη, να ανταλλάσσουν δεδομένα ασύγχρονα και να αλλάζουν δυναμικά το περιεχόμενο του εγγράφου που εμφανίζεται.

Τον Νοέμβριο του 1996, η Netscape ανακοίνωσε ότι είχε υποβάλει τη γλώσσα JavaScript στο Ecma International (μια οργάνωση της τυποποίησης των γλωσσών προγραμματισμού) για εξέταση ως βιομηχανικό πρότυπο, ώστε και άλλη περιηγητές να την υλοποιήσουν βασισμένη στο έργο που είχε γίνει στη Netscape. Αυτό οδήγησε στην έκδοση της προδιαγραφής γλώσσας ECMAScript, η οποία δημοσιεύθηκε στην ECMA-262 το 1997, με την JavaScript να είναι η πιο γνωστή υλοποίηση, άλλες υλοποίησεις ήταν η ActionScript και η JScript.

Η εξέλιξη του προτύπου συνεχίστηκε σε κύκλους, με την έκδοση της ECMAScript 2 το 1998 και της ECMAScript 3 το 1999, η οποία είναι η βάση της σύγχρονης JavaScript. Ενώ γινόντουσαν διάφορες προσπάθειες για να βγει η ECMAScript 4 η κοινότητα των προγραμματιστών έθεσε ως στόχο την επανάσταση σχετικά με το τι θα μπορούσε να γίνει με την JavaScript. Αυτή η προσπάθεια της κοινότητας πυροδότησε το 2005 όταν ο Jesse James Garrett κυκλοφόρησε ένα white paper στο οποίο επινόησε τον όρο Ajax και περιέγραψε ένα σύνολο τεχνολογιών, των οποίων η JavaScript ήταν η ραχοκοκαλιά, που χρησιμοποιείται για τη δημιουργία εφαρμογών web όπου τα δεδομένα μπορούν να φορτωθούν στο παρασκήνιο, αποφεύγοντας την ανάγκη για επαναφόρτωση πλήρους σελίδας και οδηγώντας σε πιο δυναμικές εφαρμογές. Αυτό οδήγησε σε μια περίοδο αναγέννησης της χρήσης της JavaScript με

πρωτοπόρους τις βιβλιοθήκες ανοιχτού κώδικα και τις κοινότητες που σχηματίστηκαν γύρω από αυτές, με βιβλιοθήκες όπως Prototype, jQuery, Dojo Toolkit, MooTools και άλλες να κυκλοφορούν.

2.2 Χαρακτηριστικά

Παρακάτω αναφέρουμε τα κύρια χαρακτηριστικά που έχει η JavaScript και τη συγκρίνουμε με άλλες γλώσσες που έχουν κάποια από αυτά.

2.2.1 Προστακτικός και Δομημένος Προγραμματισμός

Η JavaScript υποστηρίζει ένα μεγάλο κομμάτι της συνταξης δομημένου προγραμματισμού από τη C (if δηλώσεις, while βρόχους, switch δηλώσεις, κοκ). Μια μερική εξαίρεση είναι η εμβέλεια, η JavaScript αρχικά είχε μόνο εμβέλεια εντός των συναρτήσεων με τη λέξη κλειδί var. Η ECMAScript 2015 πρόσθεσε τις λέξεις κλειδιά let και const για εμβέλεια παραγράφου, οπότε πλέον η JavaScript υποστηρίζει εμβέλεια στο πλαίσιο συνάρτησης και στο πλαίσιο παραγράφου. Όπως και η C, η JavaScript κάνει διάκριση μεταξύ εκφράσεων και δηλώσεων. Μια διαφορά με τη C είναι ότι η JavaScript έχει αυτόματη εισαγωγή ερωτηματικών που επιτρέπει στον προγραμματιστή να παραλείψει τα ερωτηματικά που σημαίνουν τη λήξη μιας δήλωσης.

2.2.2 Δυναμικότητα

Όπως οι περισσότερες γλώσσες σεναρίου η JavaScript έχει δυναμικό σύστημα τύπων. Δηλαδή η εξακρίβωση της ασφάλειας των τύπων του προγράμματος γίνεται κατά το χρόνο εκτέλεσης, έτσι μπορεί ένα πρόγραμμα να αποτύχει κατά την εκτέλεση επειδή υπάρχει κάποιο σφάλμα τύπου. Ένας τύπος συνδέεται με μια τιμή και όχι με μια μεταβλητή, που σημαίνει ότι μια μεταβλητή μπορεί μια στιγμή να περιέχει έναν ακέραιο και μετά να περιέχει μια συμβολοσειρά. Η JavaScript διαθέτει διάφορους τρόπους να ελέγξει τον τύπο ενός αντικειμένου, συμπεριλαμβανομένου duck typing. Η JavaScript επίσης, με χρήση της συνάρτησεις eval, μπορεί να εκτελέσει δηλώσεις, που της παρέχονται ως συμβολοσειρές, κατά το χρόνο εκτέλεσης.

2.2.3 Αντικειμενοστραφής Προγραμματισμός

Η JavaScript αποτελείται σχεδόν ολόκληρη από αντικείμενα. Ένα αντικείμενο είναι ένας πίνακας συσχέτισης, μαζί με ένα πρωτότυπο (prototype). Κάθε κλειδί-συμβολοσειρά παρέχει το όνομα για μια ιδιότητα (property) του αντικειμένου, υπάρχουν δύο τρόποι να οριστεί ένα τέτοιο όνομα. Μια ιδιότητα μπορεί να προστεθεί, να ανακάμψει ή να διαγραφεί κατά την εκτέλεση. Η JavaScript επίσης έχει ένα μικρό σύνολο από ενσωματωμένα αντικείμενα όπως Function και Date. Παρακάτω φαίνονται μερικές από τις ιδιαιτερότητες της JavaScript που την κάνουν να ξεχωρίζει από άλλες αντικειμενοστραφείς γλώσσες:

Protypes

Η JavaScript χρησιμοποιεί prototypes εκεί που πολλές γλώσσες χρησιμοποιούν κλάσεις για την κληρονομικότητα. Κάθε αντικείμενο έχει μια ιδιωτική ιδιότητα που κρατάει ένα δείκτη σε ένα άλλο αντικείμενο που λέγεται το πρωτότυπο του. Με τη σειρά του και κάθε πρωτότυπο έχει ένα δικό του πρωτότυπο και έτσι σχηματίζεται μια αλυσίδα από αντικείμενα η οποία τελειώνει στο null, το οποίο δεν έχει πρωτότυπο. Παρόλο που αυτό θεωρείται μια από τις αδυναμίες της JavaScript, το πρωτοτυπικό μοντέλο κληρονομικότητας είναι πιο δυνατό από το κλασικό μοντέλο.[MDN b]

Συναρτήσεις ως Κατασκευαστές

Οι συναρτήσεις πέρα από τον τυπικό τους ρόλο, έχουν και το ρόλο του κατασκευαστή αντικειμένων. Προτάσσοντας την κλήση μια συνάρτησης με τη λέξη κλειδί new δημιουργείται ένα πρωτότυπο, κληρονομώντας ιδιότητες και μεθόδους από τον κατασκευαστή.

Συναρτήσεις ως Μέθοδοι

Σε αντίθεση με πολλές αντικειμενοστραφείς γλώσσες δεν υπάρχει διάκριση μεταξύ του ορισμού μιας συνάρτησης και του ορισμού μιας μεθόδου. Η διάκριση γίνεται κατά την κλήση της συνάρτησης, όταν μια συνάρτηση καλείται ως μέθοδος ενός αντικειμένου, η τοπική λέξη κλειδί της συνάρτησης ”this” δένεται με το αντικείμενο για αυτή την κλήση.

2.2.4 Συναρτησιακός

Μια συνάρτηση είναι πρώτης τάξης, δηλαδή μια συνάρτηση θεωρείται αντικείμενο. Ως αντικείμενο μια συνάρτηση έχει ιδιότητες και μεθόδους. Μια εμφωλιασμένη συνάρτηση είναι μια συνάρτηση που ορίζεται μέσα σε μια άλλη και δημιουργείται κάθε φορά που η εξωτερική συνάρτηση καλείται. Επιπλέον κάθε εμφωλιασμένη συνάρτηση υιοθετεί το λεκτική εμβελειοθέτηση της εξωτερικής.[Flan06] Επίσης η JavaScript υποστηρίζει ανώνυμες συναρτήσεις.

2.2.5 Εκχώρηση

Οι πρωτοτυπικές γλώσσες δεν εμφανίζουν κληρονομικότητα με τη συνήθη έννοια, αφού δε διαθέτουν κλάσεις. Ο αντίστοιχος μηχανισμός για αυτές ονομάζεται εκχώρηση. Αν κάποια οντότητα δεχθεί μια κλήση μεθόδου την οποία δεν μπορεί να χειριστεί, είναι δυνατό να την εκχωρήσει σε κάποια άλλη. Αν η εκχώρηση γίνεται προς το πρωτότυπο από το οποίο προήλθε η οντότητα, η όλη κατάσταση καταλήγει να προσομοιάζει κάπως στην κληρονόμηση. Ωστόσο, η εκχώρηση μπορεί να λάβει και πιο περίπλοκες μορφές. Κάποιες πρωτοτυπικές γλώσσες επιτρέπουν σε μια οντότητα να επιλέγει τελείως ελεύθερα σε ποια οντότητα θα εκχωρήσει, καθώς και να μεταβάλει την εκχώρηση κατά την εκτέλεση του προγράμματος.[Webb03] Η JavaScript υποστηρίζει σιωπηρή (implicit) και ρητή (explicit) εκχώρηση. Όπως φαίνεται παρακάτω:

Συναρτήσεις ως Ρόλοι

Η JavaScript υποστηρίζει εγγενώς διάφορες υλοποιήσεις του σχεδιαστικού μοτίβου Ρόλος, που βασίζονται σε συναρτήσεις, όπως Traits και Roles. Μια τέτοια συνάρτηση ορίζει πρόσθετη συμπεριφορά με τουλάχιστον μια μέθοδο που συνδέεται στη λέξη κλειδί this μέσα στο σώμα της. Ένας Ρόλος πρέπει να εκχωρηθεί ρητά μέσω κλήσης ή να εφαρμοστεί σε αντικείμενα που πρέπει να διαθέτουν πρόσθετη συμπεριφορά που δεν μοιράζεται μέσω της αλυσίδας πρωτοτύπου.

Σύνθεση αντικειμένου και κληρονομικότητα

Ενώ η ρητή εκχώρηση βασισμένη στις συναρτήσεις καλύπτει τη σύνθεση στη JavaScript, η σιωπηρή εκχώρηση συμβαίνει κάθε φορά που η αλυσίδα πρωτοτύπου διασχίζεται προκειμένου να βρεθεί π. χ. μια μέθοδος που μπορεί να σχετίζεται με ένα αντικείμενο, ενώ δεν ανήκει άμεσα σε αυτό. Μόλις η μέθοδος βρεθεί, καλείται μέσα στο πλαίσιο του αντικειμένου. Έτσι, η κληρονομικότητα στη JavaScript καλύπτεται από έναν αυτοματισμό εκχώρησης που συνδέεται με την πρωτότυπη ιδιότητα των συναρτήσεων-κατασκευαστών.

2.2.6 Περιβάλλον χρόνου εκτέλεσης

Η JavaScript συνήθως βασίζεται σε ένα περιβάλλον χρόνου εκτέλεσης (run-time environment), π. χ. ένα πρόγραμμα περιήγησης, για την παροχή αντικειμένων και μεθόδων βάσει των οποίων τα σενάρια μπορούν να αλληλεπιδρούν με το περιβάλλον, π.χ. ένα DOM ιστοσελίδας. Επίσης βασίζεται στο run-time environment για να παρέχει τη δυνατότητα εισαγωγής και εξαγωγής σεναρίων.

Η JavaScript επεξεργάζεται τα μηνύματα από μια ουρά, ένα τη φορά. Με τη φόρτωση ενός νέου μηνύματος, η JavaScript καλεί μια συνάρτηση που σχετίζεται με αυτό το μήνυμα, η οποία δημιουργεί ένα πλαίσιο στοίβας κλήσεων (από τις παραμέτρους της συνάρτησης και τις τοπικές μεταβλητές). Η στοίβα κλήσεων μικραίνει και μεγαλώνει με βάση τις ανάγκες της συνάρτησης. Με τη συμπλήρωση της συνάρτησης, όταν η στοίβα είναι άδεια, η JavaScript προχωρά στο επόμενο μήνυμα στην ουρά. Αυτό ονομάζεται βρόχος συμβάντων (event loop), που περιγράφεται ως ”εκτέλεση μέχρι ολοκλήρωσης”, επειδή κάθε μήνυμα έχει υποβληθεί σε πλήρη επεξεργασία πριν ληφθεί υπόψη το επόμενο μήνυμα. Ωστόσο, το μοντέλο ταυτοχρονισμού της γλώσσας περιγράφει τον βρόχο συμβάντων ως non-blocking: η είσοδος/έξοδος (input/output) του προγράμματος εκτελείται χρησιμοποιώντας συμβάντα και callback functions. Αυτό σημαίνει, για παράδειγμα, ότι η JavaScript μπορεί να επεξεργαστεί ένα κλικ του ποντικιού, ενώ περιμένει για μια αναζήτηση στη βάση δεδομένων να επιστρέψει.[MDN a]

2.3 V8

Μια μηχανή JavaScript είναι ένα πρόγραμμα ή ένας διερμηνέας που εκτελεί κώδικα JavaScript. Οι πρώτες μηχανές JavaScript ήταν απλοί διερμηνείς οι οποίοι απλά έτρεχαν τον πηγαίο κώδικα, καθώς όμως η JavaScript που υπήρχε στις ιστοσελίδες αύξανε και σε μέγεθος αλλά και σε πολυπλοκότητα οι μηχανές έπρεπε να προσαρμοστούν. Ο περιηγητής που είχε τη πιο γρήγορη μηχανή JavaScript ήταν και ο πιο γρήγορος. Για να αντεπεξέλθουν, οι μηχανές JavaScript άρχισαν να είναι πάνω στο χρόνο (JIT) μεταγλωττιστές. Ένας JIT μεταγλωττιστής μαζεύει στατιστικά για το πρόγραμμα καθώς τρέχει και επιλέγει ποια σημεία του κώδικα και ποιες συναρτήσεις συμφέρει να βελτιστοποιήσει για να τρέξει πιο γρήγορα ο κώδικας, περισσότερα για αυτό θα δούμε παρακάτω συγκεκριμένα για το V8.

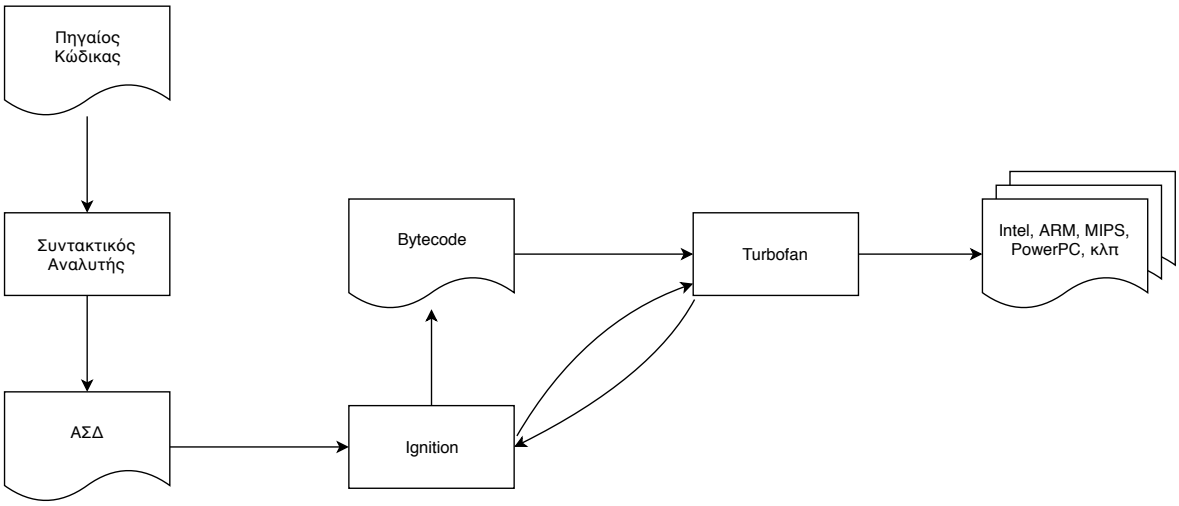
Η μηχανή V8 είναι η υψηλής απόδοσης, ανοιχτού κώδικα, μηχανή JavaScript της Google. Είναι γραμμένη σε C++ και υλοποιεί την ECMAScript όπως είναι ορισμένη στο ECMA-262. Ο V8, όπως και οι περισσότερες μηχανές JavaScript, δεν παρέχει το DOM, το DOM του παρέχεται από το Chrome. Το V8 παρέχει όλους τους τύπους, τους τελεστές, τα αντικείμενα και τις συναρτήσεις που ορίζονται από το πρότυπο της ECMA. Παρακάτω παρουσιάζουμε συνοπτικά τη δομή και ορισμένα χαρακτηριστικά του V8[1], εκτός από τον αναλυτή για τον οποίο θα αναφερθούμε εκτενέστερα στο επόμενο κεφάλαιο.

2.3.1 Δομή

Ο V8 παίρνει τον κώδικα JavaScript και πρώτα, όπως οι περισσότεροι μεταγλωττιστές, κάνει λεξική ανάλυση και μετασχηματίζει τον κώδικα σε λεξιμόρια, αν υπάρχει κάποιο λεξικογραφικό λάθος στον κώδικα θα βρεθεί σε αυτό το στάδιο. Έπειτα δίνει στον αναλυτή (parser) τα λεξιμόρια, ο οποίος κάνει συντακτική ανάλυση και παράγει το Αφηρημένο Συντακτικό Δέντρο(ΑΣΔ), σε αυτό το στάδιο θα βρεθούν τυχόν συντακτικά λάθη που υπάρχουν στον κώδικα, αυτά ονομάζονται early errors.[ECMA] Στη συνέχεια καλείται το Ignition, το οποίο μετασχηματίζει το ΑΣΔ σε οκταδυφιακό κώδικα (bytecode) και τρέχει τον κώδικα. Καθώς τρέχει ο κώδικας ο V8 συλλέγει στατιστικά για τις κλήσεις των συναρτήσεων, για τους τύπους των μεταβλητών, τους τύπους των παραμέτρων τελεστών και γενικότερα συναρτήσεων, κ.α. Αυτά τα στατιστικά δίνονται στο Turbofan, το οποίο είναι ο JIT μεταγλωττιστής

[1] Η έκδοση του V8 που χρησιμοποιήσαμε είναι η 6.4.0 και το commit που πήραμε τον κώδικα από το repository του V8(`https://github.com/v8/v8`) είναι στο 9bf0002176de.

του V8. Το Turbofan με βάση τα στατιστικά αυτά παράγει βελτιστοποιημένο κώδικα μηχανής. Ο κώδικας που παράγει το Turbofan είναι προσαρμοσμένος για τα δεδομένα που περιμένει με βάση τα στατιστικά που έχουν συλλεχθεί, αν συμβεί κάτι το οποίο δεν το περίμενε το Turbofan, π. χ. αν δώσουμε στον τελεστή '+', αντί για αριθμούς, συμβολοσειρές, τότε ο V8 κάνει αποβελτιστοποίηση. Αυτό σημαίνει πως αντικαθιστά ένα κομμάτι κώδικα που παρήχθη από το Turbofan με κώδικα του Ignition, μέχρι να δει πως έχει νόημα να ξανακάνει βελτιστοποιήσεις. Η δομή του V8 φαίνεται και στο 2.1.



Σχήμα 2.1: Η δομή του V8

2.3.2 Ignition

Το Ignition είναι είναι ένας interpreter. Το Ignition παίρνει το ΑΣΔ που παρήγαγε ο αναλυτής και παράγει bytecode. Bytecode είναι ένα σύνολο εντολών, που είναι σχεδιασμένο για την εκτέλεση από έναν interpreter. Η γλώσσα του Ignition είναι μια μηχανή καταχωρητών (register machine), δηλαδή έχει ένα άπειρο πλήθος από καταχωρητές και κάθε bytecode ορίζει ως είσοδο και έξοδο του καταχωρητές. Ένας ειδικός καταχωρητής-συσσωρευτής αποτελεί την είσοδο ή την έξοδο αρκετών bytecodes. Το bytecode αυτό φτιάχτηκε για να είναι εύκολα συμβατό με το API του Turbofan και για να μην χρειάζεται πολύ χώρο. Το Ignition αφού παράγει το bytecode κάνει διάφορες βελτιστοποιήσεις όπως βελτιστοποίηση καταχωρητών, βελτιστοποιήσεις κλειδαρότρυπας και απαλοιφή άχρηστου κώδικα. Τέλος μετατρέπει το bytecode σε χαμηλού επιπέδου, ανεξάρτητες αρχιτεκτονικής, μάκροεντολες συμβολικής γλώσσας μηχανής του Turbofan και τις δίνει στο Turbofan για να παράγει κώδικα μηχανής. Κατά την εκτέλεση του κώδικα το Ignition συλλέγει στατιστικά, όπως κλήσεις, τύπους κλπ., τα οποία τα δίνει στη συνέχεια στο Turbofan για να παράγει βελτιστοποιημένο κώδικα.

2.3.3 Turbofan

Το Turbofan είναι ένας JIT μεταγλωττιστής ο οποίος υποστηρίζει παραγωγή κώδικα για 9 αρχιτεκτονικές. Η ενδιάμεση αναπαράσταση που χρησιμοποιεί το Turbofan είναι μια χαλάρωση της θάλασσας κόμβων (sea of nodes). Υπάρχουν τρία επίπεδα στη γλώσσα του Turbofan το επίπεδο με τους τελεστές της JavaScript, στο οποίο εκφράζεται η σημασιολογία των υπερφορτωμένων τελεστών της JavaScript, το ενδιάμεσο επίπεδο με απλοποιημένους τελεστές, οι οποίοι εκφράζουν τελεστές εικονικής μηχανής, όπως δέσμευση μνήμης και έλεγχος ορίων,και το επίπεδο μηχανής, στο οποίο οι τελεστές αντιστοιχούν αρκετά κοντά σε εντολές μηχανής. Ο γράφος στην αρχή αποτελείται από κόμβους υψηλού επιπέδου (JavaScript) οι οποίοι με επαναλαμβανόμενα περάσματα μετατρέπονται σε κόμβους του

χαμηλότερου επιπέδου (γλώσσα μηχανής). Με κάθε πέρασμα γίνεται ανάλυση τύπων, ανάλυση ευρών, ανάλυση ζωντάνιας, μείωση του γράφου, μείωση των τύπων, βελτιστοποιήσεις κ. α. Τέλος όταν τελειώσει η επεξεργασία του γράφου παράγεται κώδικας μηχανής. Να σημειώσουμε ότι ο κώδικας που δίνει το Ignition στο Turbofan είναι του χαμηλότερου τύπου (κώδικας μηχανής) και σε αυτό δε γίνονται όλες οι βελτιστοποιήσεις του Turbofan.

Το Turbofan χρησιμοποιεί στατιστικά που συλλέχθηκαν από το Ignition για να κάνει βελτιστοποιήσεις στον κώδικα, ένα από αυτά είναι οι τύποι που έχουν οι διάφορες μεταβλητές, αν όμως συμβεί κάτι που δεν περίμενε, όπως μια μεταβλητή να αλλάξει τύπο, τότε ο κώδικας που είχε παράξει δεν είναι πλέον σωστός και πρέπει να γίνει αποβελτιστοποίηση, για αυτό καλείται πάλι το Ignition για να παράγει κώδικα από την αρχή.

2.3.4 Χαρακτηριστικά

Συλλογή σκουπιδιών

Ο ρακοσυλλέκτης του V8 είναι ένας σταματά-το-κόσμο χρονολογικός ρακοσυλλέκτης, δηλαδή ανά τακτά χρονικά διαστήματα η εκτέλεση του προγράμματος σταματά και ο ρακοσυλλέκτης προσπαθεί να καθαρίσει τη μνήμα από αντικείμενα που δε χρησιμοποιούνται. Το καθάρισμα δε γίνεται με ένα πέρασμα αλλά κάθε φορά που καλείται ο ρακοσυλλέκτης, ελέγχει μερικά από τα αντικείμενα που βρίσκονται στο σωρό και έπειτα επιστρέφει, την επόμενη φορά που θα κληθεί θα συνεχίσει από εκεί που έμεινε. Αυτό έχει ως συνέπεια πολύ μικρές διακοπές της λειτουργίας του V8. Ο ρακοσυλλέκτης βρίσκεται σε ξεχωριστό νήμα.

Κρυμμένα αντικείμενα

Η JavaScript είναι μια δυναμική γλώσσα και ιδιότητες μπορούν να προστεθούν και να αφαιρεθούν κατά την εκτέλεση από τα αντικείμενα. Αυτό κάνει αρκετά δύσκολη τη βελτιστοποιήση του χρόνου φόρτωσης μιας ιδιότητας. Για αυτό ο V8 υλοποιεί κρυμμένα αντικείμενα (hidden objects). Κάθε φορά που μια ιδιότητα προστίθεται σε ένα αντικείμενο ο V8 φτιάχνει ένα καινούριο κρυμμένο αντικείμενο, κάτι που κάνει την πρόσβαση των ιδιοτήτων του πολύ γρήγορη. Επίσης αυτή η προσέγγιση έχει το πλεονέκτημα ότι αντικείμενα που έχουν δεχτεί τις ίδιες αλλαγές μπορούν να μοιράζονται το ίδιο κρυμμένο αντικείμενο.

2.3.5 Ο V8 στον Chrome

Όπως αναφέραμε ο V8 αρχικά σχεδιάστηκε ως ο μεταγλωττιστής JavaScript του Chrome. Τα DOM αντικείμενα που χρησιμοποιούνται για τον έλεγχο των γραφικών της ιστοσελίδας δεν είναι μέρος της JavaScript, αυτά παρέχονται από τον Chrome στο V8. Για τη σύνδεση του V8 με το Chrome χρησιμοποιείται το API του V8. Ο Chrome για κάθε καρτέλα που είναι ανοιχτή χρησιμοποιεί ένα διαφορετικό στιγμιότυπο του V8. Τα στιγμιότυπα αυτά δεν επικοινωνούν και υπάρχουν ανεξάρτητα.

2.3.6 Blink

Το Blink δεν είναι μέρος του V8, αλλά παρουσιάζουμε συνοπτικά τη λειτουργία του για να γίνει πιο κατανοητή στον αναγνώστη η λειτουργία του V8. Το Blink είναι ένα browser engine βασισμένο στον WebCore και χρησιμοποιείται στον Chrome.

Browser engine είναι λογισμικό που χρησιμοποιείται για λειτουργήσει ένας περιηγητής, συγκεκριμένα είναι υπεύθυνος για να διαχειρίζεται τη διάταξη και την ερμηνεία των ιστοσελίδων. Μια browser

engine μετατρέπει το HTML αρχείο και τους σχετικούς πόρους, όπως εικόνες και φύλλα στυλ, σε μια ιστοσελίδα η οποία μπορεί να αντιληφθεί και αλληλεπιδράσει με τον τελικό χρήστη. Έτσι και το Blink είναι υπεύθυνο για να διαχειρίζεται τα HTML και CSS στοιχεία μιας ιστοσελίδας, επίσης είναι το ενδιάμεσο στρώμα που "μιλάει" με τον V8.

2.3.7 Μεταγλώττιση

Για τη μεταγλώττιση του V8 χρησιμοποιείται ο μεταγλωττιστής Clang. Για το χειρισμό των κανόνων μεταγλώττισης και τον ορισμό των εξαρτήσεων που έχουν τα διάφορα αρχεία χρησιμοποιείται το εργαλείο Ninja. Το Ninja είναι ένα σύστημα δόμησης, παίρνει ως είσοδο τις αλληλεξαρτήσεις των διαφόρων αρχείων (συνήθως πηγαίο κώδικα και εκτελέσιμα) και ενορχηστρώνει την κατασκευή τους. Χρησιμοποιείται το Ninja για το χτίσιμο του V8 γιατί είναι πιο γρήγορο από άλλα εργαλεία όπως το GNU Make. Την ταχύτητα του την οφείλει στην πολύ απλή γλώσσα του, η οποία όμως το κάνει πιο αδύναμο από άποψη εκφραστικότητας από άλλα συστήματα. Για την παραγωγή των αρχείων δόμησης του Ninja χρησιμοποιείται το εργαλείο GN, το οποίο είναι ένα εργαλείο meta-build system και ανήκει και αυτό,όπως και το Ninja, στο έργο Chromium.

Κεφάλαιο 3

Συντακτικός Αναλυτής

Η συντακτική ανάλυση και παραγωγή ΑΣΔ της JavaScript παρουσιάζει αρκετές δυσκολίες οι οποίες προκύπτουν τόσο από τη γραμματική της γλώσσας όσο και από τη δυναμικότητα της. Για παράδειγμα ένα μεγάλο πρόβλημα που πρέπει να αντιμετωπίσει ο αναλυτής είναι ότι ένα μεγάλο μέρος του κώδικα JavaScript που περιέχει μια ιστοσελίδα είναι πολύ πιθανό να μην τρέξει ποτέ αν δεν κληθούν οι συναρτήσεις, παρόλα αυτά πρέπει να αναλυθούν οι συναρτήσεις γιατί όπως ορίζει το πρότυπο της ECMAScript αν υπάρχει ένα ”early error” στον κώδικά τότε πρέπει να επιστραφεί σφάλμα.

Παρακάτω θα αναλύσουμε κάποια από αυτό το πρόβλημα, και κάποια άλλα, και τους μηχανισμούς που έχουν υλοποιηθεί στο V8 για να τα ξεπεράσουν. Επίσης θα αναλύσουμε ποια είναι τα λάθη τα οποία εντοπίζει ο συντακτικός αναλυτής.

3.1 Προβλήματα

Σε αυτή τη παράγραφο θα παρουσιάσουμε συνοπτικά μερικά από τα προβλήματα που συναντά ο συντακτικός αναλυτής του V8.

Μια βελτιστοποίηση που προσπαθεί να κάνει ο V8 είναι να μην παράγει ΑΣΔ για συναρτήσεις που πιστεύει πως δε θα κληθούν, με αυτό τον τρόπο καταφέρνει όχι μόνο να πετύχει επιτάχυνση χρόνο εκτέλεσης, άλλα και να μειώσει σημαντικά τη μνήμη που χρησιμοποιεί. Σε αυτές τις συναρτήσεις ο V8 κάνει “οκνηρή” συντακτική ανάλυση, μια πιο απλή συντακτική ανάλυση από ότι στις άλλες η οποία είναι και πιο γρήγορη. Το πρόβλημα είναι ότι αν χρειαστεί κάποια στιγμή να κληθεί μια από αυτές τις συναρτήσεις τότε ο V8 αναγκάζεται να την αναλύσει κανονικά, σπαταλώντας έτσι παραπάνω χρόνο από ότι αν δεν είχε προσπαθήσει να κάνει τη βελτιστοποίηση. Δυστυχώς είναι αδύνατο να ξέρουμε από πριν αν μια συνάρτηση θα κληθεί ή όχι. Επιπλέον ο V8 κάνει οκνηρή ανάλυση σε εμφωλευμένες συναρτήσεις, που μπορεί να οδηγήσει σε πολλαπλή ανάλυση τους. Αυτό μπορεί να προκαλέσει μεγάλα προβλήματα καθώς η σύγχρονη JavaScript είναι βαθιά εμφωλευμένη. Δηλαδή στον κώδικα που φαίνεται παρακάτω, πρώτα θα γίνει οκνηρή ανάλυση όλων των συναρτήσεων, αν μετά κληθεί η συνάρτηση lazy_outer τότε οι εσωτερικές συναρτήσεις θα ξανα-αναλυθούν οκνηρά και στο τέλος αν κληθεί η inner τότε θα αναλυθεί οκνηρά η inner2 για τρίτη φορά.

```
function lazy_outer() { // Lazy parse this
    function inner() {
        function inner2() { ... }
    }
}

... lazy_outer(); // Οκνηρήανάλυση inner & inner2
... inner(); // Οκνηρήανάλυση inner2 η(3 φορά!)
```

Ένα άλλο πρόβλημα έχει σχέση με τη γραμματική της JavaScript. Η γραμματική της JavaScript είναι ασαφής, όπως φαίνεται και από τα παραδείγματα κώδικα που βρίσκονται παρακάτω υπάρχουν

περιπτώσεις που δεν μπορείς να ξέρεις από πριν τι αναλύεις.

```
(a, b, c) => { return a; } // Σωστό: arrow function
(a, b, c)                  // Σωστό: comma expression
(a, 1, 2)                  // Σωστό: comma expression
(a, 1, 2) => { return a; } // Λάθος
(a, ...b) => { return b; } // Σωστό: arrow func + rest param
(a, ...b)                  // Λάθος
```

3.2 Συντακτικά Λάθη

Όπως αναφέραμε και στο προηγούμενο κεφάλαιο τα λάθη της JavaScript που πρέπει να εντοπίζει ένας μεταγλωττιστής κατά τη στατική συντακτική ανάλυση του προγράμματος ονομάζονται early errors. Αυτά όπως ορίζονται από το πρότυπο της JavaScript είναι εξής[ECMA]:

> An implementation must report most errors at the time the relevant ECMAScript language construct is evaluated. An early error is an error that can be detected and reported prior to the evaluation of any construct in the Program containing the error. An implementation must report early errors in a Program prior to the first evaluation of that Program. Early errors in eval code are reported at the time eval is called but prior to evaluation of any construct within the eval code. All errors that are not early errors are runtime errors.
> An implementation must treat any instance of the following kinds of errors as an early error:
>
> - Any syntax error.
> - Attempts to define an ObjectLiteral that has multiple get property assignments with the same name or multiple set property assignments with the same name.
> - Attempts to define an ObjectLiteral that has both a data property assignment and a get or set property assignment with the same name.
> - Errors in regular expression literals that are not implementation-defined syntax extensions.
> - Attempts in strict mode code to define an ObjectLiteral that has multiple data property assignments with the same name.
> - The occurrence of a WithStatement in strict mode code.
> - The occurrence of an Identifier value appearing more than once within a FormalParameterList of an individual strict mode FunctionDeclaration or FunctionExpression.
> - Improper uses of return, break, and continue.
> - Attempts to call PutValue on any value for which an early determination can be made that the value is not a Reference (for example, executing the assignment statement 3=4).

Ο V8 δεν πληρεί το πρότυπο της JavaScript καθώς δε πιάνει όλα αυτά τα λάθη πριν να αρχίσει να εκτελεί το πρόγραμμα. Μάλιστα αν υπάρχει ένα τέτοιο σφάλμα σε μια συνάρτηση η οποία δε καλείται μπορεί να μην εντοπιστεί ποτέ το σφάλμα. Δηλαδή αν δηλώσουμε τη συνάρτηση:

```
function foo(){let a=1; let a=2;}
```

Κανονικά ο V8 θα έπρεπε να μας επιστρέψει σφάλμα, καθώς η δήλωση με let μιας μεταβλητής δύο φορές στην ίδια εμβέλεια είναι early error. Στο V8 αντιθέτως αυτό που θα συμβεί είναι πως θα πάρουμε σφάλμα μόνο αν κάποια στιγμή αργότερα κληθεί η παραπάνω συνάρτηση:

```
foo();
```
Uncaught SyntaxError: Identifier 'a' has already been declared at <anonymous>:1:1

3.3 Ο Συντακτικός Αναλυτής του V8

Ο συντακτικός αναλυτής του V8 είναι ένας αναλυτής αναδρομικής κατάβασης. Ο αναλυτής είναι χωρισμένος σε δύο κύριες κλάσεις, τον Parser και τον PreParser, οι οποίες κληρονομούν τις βασικές τους συναρτήσεις από μία κλάση-βάση υλοποιώντας το μοντέλο Curiously Recurring Template Pattern (CRTP). Παρακάτω παρουσιάζουμε τις βασικές τους διαφορές και τις βασικές τους λειτουργίες, επίσης παρουσιάζουμε το πως συνδέεται ο αναλυτής με το υπόλοιπο πρόγραμμα.

3.3.1 PreParser

Όπως αναφέραμε παραπάνω πολλές φορές ένα κομμάτι κώδικα JavaScript που περιέχεται σε μια σελίδα δεν εκτελείται ποτέ, παρόλα αυτά (όπως είναι λογικό) ο αναλυτής πρέπει να κάνει κάποιους έλεγχους ακόμα και σε αυτά τα κομμάτια. Ο PreParser έχει υλοποιηθεί για ακριβώς αυτό το λόγο. Ο PreParser είναι ο "οκνηρός" αναλυτής του V8, χρησιμοποιείται για την ανάλυση συναρτήσεων που δε θέλει ο V8 να μεταγλωττίσει, συλλέγει δεδομένα για τον κώδικα όπως το που ξεκινά και που τελειώνει μια συνάρτηση και εντοπίζει μερικά early errors χωρίς να παράγει το ΑΣΔ. Ο V8 χρησιμοποιεί διάφορες ευριστικές που του παρέχονται από το χρήστη για να αποφασίσει αν μια συνάρτηση θα κληθεί ή όχι και κατά συνέπεια αν πρέπει να κάνει οκνηρή ή πρόθυμη ανάλυση. Δύο από αυτές τις ευριστικές είναι ο εγκλεισμός της δήλωσης της συνάρτησης σε παρενθέσεις () ή η τοποθέτηση ενός θαυμαστικού (!) πριν από τη δήλωση.

Το πλεονέκτημα του PreParser είναι ότι η ανάλυση που κάνει σε μια συνάρτηση είναι πολύ πιο γρήγορη από την ανάλυση που κάνει ο Parser (περίπου 2 φορές πιο γρήγορη). Αυτό συμβαίνει γιατί ο PreParser δεν παράγει ΑΣΔ και επίσης ενώ δημιουργεί εμβέλειες δεν τις γεμίζει με μεταβλητές και δηλώσεις. Επίσης ο PreParser βρίσκει ένα υποσύνολο των σφαλμάτων που θα έπρεπε να βρίσκει (δεν πληρεί το πρότυπο της JavaScript).

3.3.2 Parser

Ο Parser είναι ο πρόθυμος συντακτικός αναλυτής του V8. Χρησιμοποιείται για την ανάλυση συναρτήσεων που ξέρουμε πως θα χρειαστεί να μεταγλωττιστούν, παράγει το ΑΣΔ και τις εμβέλειες. Επίσης εντοπίζει όλα τα συντακτικά λάθη, επειδή όμως δεν καλείται για όλες τις συναρτήσεις, μπορεί να υπάρχει κάποιο early error στον κώδικα και να μην το εντοπίσει.

3.3.3 Σημεία Εισόδου

Υπάρχουν δύο σημεία εισόδου στον αναλυτή από το αρχείο src/parsing/parsing.cc και από το αρχείο src/parsing/background-parsing-task.cc. Ξεκινώντας με τον Chrome 41 η συντακτική ανάλυση μιας ιστοσελίδας ξεκινά πριν από την πλήρη λήψη της, δηλαδή όσο ο Chrome κατεβάζει τη σελίδα ο αναλυτής του V8 κάνει ανάλυση σε ένα νήμα που τρέχει στο βάθος,[Höl15] αυτό γίνεται από το αρχείο src/parsing/background-parsing-task.cc. Το src/parsing/parsing.cc έχει δύο συναρτήσεις για να καλεί τον αναλυτή, η μια τον καλεί για να κάνει ανάλυση και να παράγει ΑΣΔ από ένα ολόκληρο πρόγραμμα JavaScript (ParseProgram) και η άλλη για μία μόνο συνάρτηση (ParseFunction).

3.4 Ανάλυση Αναδρομικής Κατάβασης

Ο συντακτικός αναλυτής του V8 είναι ένας αναλυτής αναδρομικής κατάβασης. Η καθοδική συντακτική ανάλυση μπορεί να θεωρηθεί ως το πρόβλημα της κατασκευής ενός συντακτικού δέντρου για μια συμβολοσειρά εισόδου, ξεκινώντας από τη ρίζα και δημιουργώντας τους κόμβους του συντακτικού δέντρου με προδιάταξη (πρώτα-κατά-βάθος). Ισοδύναμα, η καθοδική ανάλυση μπορεί να θεωρηθεί ως η εύρεση ενός αριστερότερου σχηματισμού παραγώγου για μια συμβολοσειρά εισόδου. Σε κάθε βήμα μιας καθοδικής συντακτικής ανάλυσης, το βασικό πρόβλημα είναι εκείνο της απόφασης για την που θα εφαρμοστεί σε ένα μη-τερματικό.

Ένα πρόγραμμα συντακτικής ανάλυσης αναδρομικής κατάβασης αποτελείται από ένα σύνολο διαδικασιών, μια για κάθε μη-τερματικό. Η εκτέλεση ξεκινά με τη διαδικασία για το αρχικό σύμβολο, που σταματά και ανακοινώνει επιτυχία αν το σώμα της διαδικασίας του σαρώσει ολόκληρη τη συμβολοσειρά εισόδου. Ψευδοκώδικας για ένα τυπικό μη-τερματικό εμφανίζεται παρακάτω. Σημειώστε ότι ο κώδικας είναι μη-προσδιοριστικός, καθώς ξεκινά επιλέγοντας την Α-παραγωγή που εφαρμοστεί με έναν τρόπο που δεν καθορίζεται. [Aho14]

Algorithm 1 Μια τυπική διαδικασία για ένα μη-τερματικό σε έναν καθοδικό συντακτικό αναλυτή

1: **function** A
2: Επέλεξε μια Α-παραγωγή, $A \rightarrow X_1 X_2 \ldots X_k$;
3: **for** $i = 1$ έως k **do**
4: **if** το X_i είναι μη-τερματικό **then** κάλεσε τη διαδικασία $X_i()$;
5: **else if** το X_i ισούται με το τρέχον σύμβολο εισόδου α **then** προχώρησε την είσοδο στο επόμενο σύμβολο;
6: **else**/* έχει συμβεί ένα σφάλμα */
7: **end if**
8: **end for**
9: **end function**

Γενικά η αναδρομική κατάβαση μπορεί να απαιτεί υπαναχώρηση δηλαδή μπορεί να απαιτεί επαναλαμβανόμενες σαρώσεις πάνω στην είσοδο. Ο αναλυτής του V8 δεν χρησιμοποιεί καθόλου υπαναχώρηση, αυτό το επιτυγχάνει κρατώντας πίνακες με υποψήφια συντακτικά λάθη και όταν φτάσει η στιγμή που μπορεί να πάρει απόφαση αν πραγματικά είναι συντακτικό λάθος ή όχι τότε δρα ανάλογα.

3.5 Το Σχεδιαστικό Μοτίβο CRTP

Το Curiously recurring template pattern (CRTP) είναι μια τεχνική μεταπρογραμματισμού της C++ στο οποίο μια κλάση X κληρονομεί από μια πρότυπη (template) κλάση χρησιμοποιώντας την ίδια τη X ως παράμετρο του template. Η τεχνική είναι επίσης γνωστή ως F-bound polymorphism.

Η τεχνική πήρε το όνομα της από τον Jim Coplen το 1995[Copl95]. Η γενική μορφή της τεχνικής φαίνεται παρακάτω.

```
// The Curiously Recurring Template Pattern (CRTP)
template<class T>
class Base {
    // methods within Base can use template to access members of Derived
};

class Derived : public Base<Derived> {
    ...
};
```

Το πλεονέκτημα της τεχνικής αυτής είναι ότι η Base έχει αναφορά σε συναρτήσεις της Derived, όμως αυτές μπορούν να υπολογιστούν κατά τη διάρκεια της μεταγλώττισης, καθώς ο πολυμορφισμός είναι στατικός. Σε αντίθεση στον δυναμικό όπου ο μεταγλωττιστής δεν θα μπορούσε να ξέρει σε ποια συνάρτηση αναφέρεται η Base μέχρι να τρέξει το πρόγραμμα. Στο δυναμικό πολυμορφισμό κάθε φορά που η Base θα ήθελε να καλέσει μια συνάρτηση της Derived θα έπρεπε να διαβάσει έναν πίνακα με δείκτες σε συναρτήσεις της άλλης κλάσης, ενώ στο στατικό με τη χρήση του CRTP αυτό θα γινόταν κατά τη μεταγλώττιση και γλιτώνουμε την πρόσβαση στον πίνακα.

Συνήθως, η πρότυπη συνάρτηση κλάση θα εκμεταλλευτεί το ότι τα σώματα των συναρτήσεων μελών αρχικοποιούνται αρκετά αργότερα από όταν δηλώνονται και χρησιμοποιεί μέλη της υποκλάσης μέσα στις δικές της συναρτήσεις, χρησιμοποιώντας cast, όπως φαίνεται παρακάτω:

```
template <class T> class Base {
    void interface() {
        // ...
        static_cast<T*>(this)->implementation();
        // ...
    }

    static void static_func(){
        // ...
        T::static_sub_func();
        // ...
    }
};

class Derived : Base<Derived> {
    void implementation();
    static void static_sub_func();
};
```

Στο παραπάνω παράδειγμα, συγκεκριμένα η συνάρτηση Base<Derived>::interface() παρόλο που δηλώνεται πριν η ύπαρξη της κλάσης Derived γίνει γνωστή από τον μεταγλωττιστή (δηλαδή πριν να δηλωθεί), δεν αρχικοποιείται από τον μεταγλωττιστή μέχρι να κληθεί από κάποιο κώδικα, το οποίο θα γίνει μετά τη δήλωση της Derived, οπότε όταν αρχικοποιηθεί η συνάρτηση “implementation” η δήλωση της Derived::implementation είναι γνωστή.

Αυτή η τεχνική επιτυγχάνει παρόμοιο αποτέλεσμα με τις virtual functions, χωρίς το πρόσθετο κόστος (και κάποια ευελιξία) του δυναμικού πολυμορφισμού. Η συγκεκριμένη χρήση του CRTP λέγεται simulated dynamic binding.

Για να γίνει καλύτερα αντιληπτό το παραπάνω παράδειγμα, ας υποθέσουμε πως έχουμε μια κλάση-βάση χωρίς εικονικές συναρτήσεις. Όταν η βάση καλεί μια άλλη συνάρτηση μέλος, θα καλεί πάντα της

δικές τις συναρτήσεις. Όταν παράγουμε μια κλάση από την κλάση-βάση, τότε η νέα κλάση κληρονομεί όλες τις μεταβλητές και τις συναρτήσεις της βάσης που δεν υπερκαλύφθηκαν. Αν η νέα κλάση καλέσει μια κληρονομημένη συνάρτηση η οποία μετά καλεί μια άλλη συνάρτηση μέλος, εκείνη η συνάρτηση ποτέ δεν θα καλέσει καμία κληρονομημένη ή υπερκαλυμμένη συνάρτηση μέλος της υποκλάσης.

Αν όμως η κλάση-βάση χρησιμοποιεί CRTP για όλες τις συναρτήσεις μέλη της, οι υπερκαλυμμένες συναρτήσεις θα επιλεχθούν κατά την ώρα της μεταγλώττισης. Αυτό πρακτικά μιμείται το σύστημα κλήσεων των virtual functions κατά τη διάρκεια της μεταγλώττισης χωρίς το κόστος σε μέγεθος και σε χρόνο από τις κλήσεις συναρτήσεων (από δομές virtual method tables, αναζητήσεις μεθόδων, μηχανισμούς πολλαπλής κληρονομιάς VTBL) με το μειονέκτημα ότι δεν μπορεί να κάνει αυτή την επιλογή κατά το χρόνο εκτέλεσης. Στην περίπτωση του συντακτικού αναλυτή όμως δε χρειάζεται η ευελιξία κατά το χρόνο εκτέλεσης, άρα η χρήση του CRTP δίνει σημαντικά οφέλη από άποψη χρόνου (και μνήμης).

Κεφάλαιο 4

Η Υλοποίηση μας

Οι αλλαγές που κάναμε στον κώδικα περιορίζονται κυρίως στα αρχεία που σχετίζονται με την συντακτική ανάλυση και βρίσκονται στο φάκελο src/parsing.

4.1 Μια πρώτη προσέγγιση

Αρχικά θέλαμε να δούμε το κέρδος που θα μπορούσαμε να έχουμε στο χρόνο αν βγάζαμε εντελώς τον εντοπισμό σφαλμάτων από τον κώδικα. Στην αρχή χρησιμοποιήσαμε την εντολή στον προεπεξεργαστή ifndef γύρω από κομμάτια κώδικα σε συνδυασμό με τη δήλωση της σημαίας NO_ERROR για να τα μην μεταγλωττίζονται, για παράδειγμα ο κώδικας:

```
if(function->nargs != -1 \&\& function->nargs != args->length())) {
    this->ReportMessage(MessageTemplate::kRuntimeWrongNumArgs);
    *ok = false;
    return nullptr;
}
```

Μετά τη μετατροπή έγινε:

```
#ifndef NO_ERROR
if(function->nargs != -1 && function->nargs != args->length())) {
    this->ReportMessage(MessageTemplate::kRuntimeWrongNumArgs);
    *ok = false;
    return nullptr;
}
#endif
```

Συγκεκριμένα αφαιρέσαμε κομμάτια από τα αρχεία /src/parsing/{parser.cc, preparser.cc, parser-base.h} και src/ast/{scopes.cc, modules.cc}. Συνολικά αφαιρέσαμε 235 κομμάτια κώδικα, 1000 γραμμές, από αυτά τα αρχεία όπως φαίνεται και από τον πίνακα4.1.

Αρχείο	Block κώδικα	Γραμμές κώδικα
src/parsing/parser-base.h	188	717
src/parsing/parser.cc	35	170
src/parsing/preparser.cc	10	48
src/ast/scopes.cc	1	43
src/ast/modules.cc	1	22

Πίνακας 4.1: Αλλαγές ανά αρχείο

Έπειτα για να σιγουρευτούμε ότι δεν χάλασε η λειτουργικότητα του αναλυτή χρησιμοποιήσαμε τα υπάρχοντα τεστ του V8, χρησιμοποιώντας το script /tools/run-tests.py. Ο V8 διαθέτει ένα μεγάλο εύρος τεστ, πάνω από 13000 περιπτώσεις, που καλύπτουν τις διάφορες συνιστώσες του και τις διάφορες λειτουργίες του. Προφανώς μετά τις αλλαγές μας υπήρχαν αρκετά τεστ που δεν πέρναγαν. Για αυτό έπρεπε να σιγουρευτούμε πως τα μόνα τεστ που δεν πέρναγαν ήταν αυτά που σχετίζονταν με early errors. Κατά κύριο λόγο τα τεστ που αφορούν early errors είτε απευθύνονται στον preparser είτε στα μηνύματα και βρίσκονται στον αντίστοιχο φάκελο. Επίσης πολλά τεστ αποτύγχαναν, αλλά αποτύγχαναν σε λάθος σημείο καθώς πέρναγαν από τον συντακτικό αναλυτή χωρίς να εντοπιστεί το λάθος, άλλα το δέντρο που παραγόταν δεν έβγαζε νόημα και έσκαγε είτε η παραγωγή bytecode είτε η παραγωγή τελικού κώδικα.

Στη συνέχεια χτίσαμε το Chrome με τον αλλαγμένο V8[1]. Τα αποτελέσματα και ο τρόπος συλλογής τους παρουσιάζονται στο επόμενο κεφάλαιο.

4.2 Μια λειτουργική προσέγγιση

4.2.1 Χρήση του CRTP

Η παραπάνω υλοποίηση προφανώς δουλεύει καλά μόνο αν ξέρουμε πως τα αρχεία JavaScript που θα πάρει ο V8 δεν περιέχουν early errors. Στα πειράματα αυτό δεν ήταν θέμα, αφού ξέραμε πως οι κώδικες δεν περιέχουν τέτοια λάθη, αλλά θέλαμε να παράγουμε μια λειτουργική υλοποίηση. Για να το πετύχουμε αυτό παραμετροποιήσαμε τον Parser και τον PreParser σύμφωνα με την τεχνική CRTP και δημιουργήσαμε τέσσερις νέες κλάσεις τις ParserNE και ParserWE, οι οποίες κληρονομούν από τον Parser, και τις PreParserNE και PreParserWE, οι οποίες κληρονομούν από τον PreParser. Ενδεικτικά οι υλοποιήσεις των δύο πρώτων είναι:

```
class ParserWE : public Parser<ParserWE, PreParserWE> {
  public:
    explicit ParserWE(ParseInfo* info) :
             Parser<ParserWE, PreParserWE>(info){};
    static constexpr bool CheckForErrors() { return true; }
};

class ParserNE : public Parser<ParserNE, PreParserNE> {
  public:
    explicit ParserNE(ParseInfo* info) :
             Parser<ParserNE, PreParserNE>(info){};
    static constexpr bool CheckForErrors() { return false; }
};
```

Η λέξη κλειδί constexpr μπορεί να χρησιμοποιηθεί μόνο με απλές συναρτήσεις και δηλώνει στον compiler ότι είναι δυνατό να γίνει αποτίμηση της συνάρτησης κατά τη διάρκεια της μεταγλώττισης. Έτσι αντικαταστήσαμε όλα τα ifndef με if το οποίο έχει ως συνθήκη την κλήση στη συνάρτηση CheckForErrors της αντίστοιχης κλάσης, για παράδειγμα ο κώδικας:

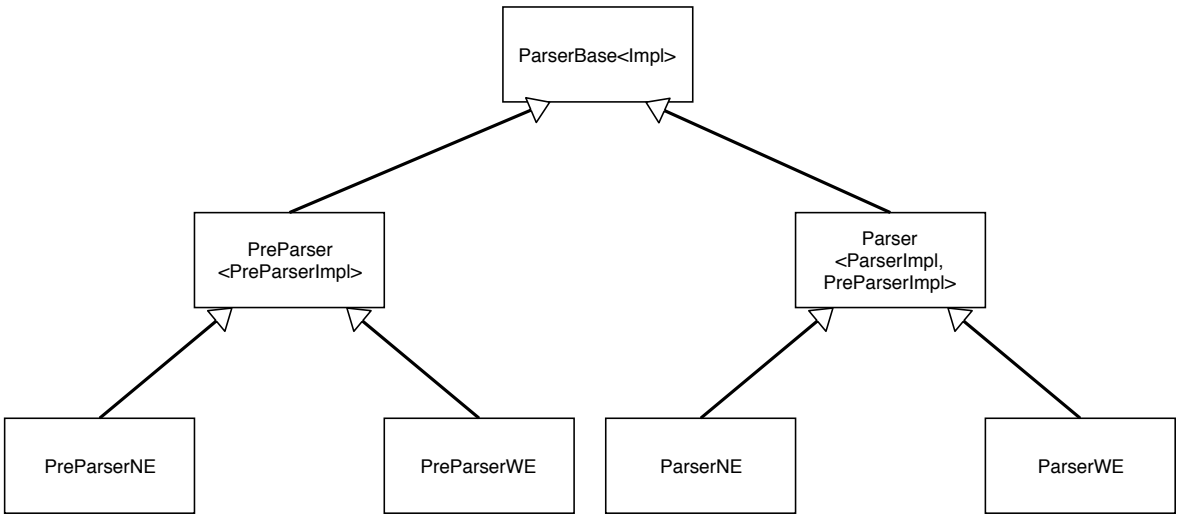
[1] Η έκδοση του Chrome που χρησιμοποιήσαμε είναι η 64.0.3277.0 (Developer Build) (64-bit), την πήραμε από το commit e9f4c7e4beb

```
#ifndef NO_ERROR
    if(function->nargs != -1 && function->nargs != args->length())) {
        this->ReportMessage(MessageTemplate::kRuntimeWrongNumArgs);
        *ok = false;
        return nullptr;
    }
#endif
```

Μετά τη μετατροπή έγινε:

```
if (this->impl()->CheckForErrors() &&
    (function->nargs != -1 && function->nargs != args->length())) {
    this->ReportMessage(MessageTemplate::kRuntimeWrongNumArgs);
    *ok = false;
    return nullptr;
}
```

Με αυτή την αλλαγή καταφέραμε να επιτύχουμε τους ίδιους χρόνους όπως με το ifndef, καθώς γινόταν αποτίμηση της συνάρτησης κατά τη μεταγλώττιση, ενώ ταυτόχρονα μπορούμε να επιλέξουμε αν θα γίνει ανάλυση με ή χωρίς εντοπισμό λαθών χρησιμοποιώντας την κατάλληλη κλάση (η ParserNE είναι επί της ουσίας ο κανονικός Parser και η ParserWE είναι ο αλλαγμένος, αντίστοιχα για τον PreParser). Η νέα δομή του Parser φαίνεται και στο σχήμα 4.1.



Σχήμα 4.1: Η δομή του συντακτικού αναλυτή

Έχοντας τις παραπάνω κλάσεις έπρεπε να ξέρουμε πότε πρέπει να χρησιμοποιήσουμε την κάθε κλάση, δηλαδή αν πρέπει ο αναλυτής να ψάξει για λάθη ή όχι το οποίο είναι ισοδύναμο με το αν έχει αναλυθεί ο κώδικας επιτυχώς στο παρελθόν. Για να ελέγχουμε αν έχουμε ξανά-αναλύσει κάποιο κώδικα χρησιμοποιήσαμε την υπάρχουσα στο V8 συνάρτηση κατακερματισμού, τη συνάρτηση κατακερματισμού του Jenkins με αρχικό seed τη θέση μνήμης του αντικειμένου που δίνουμε ως είσοδο στη συνάρτηση.

4.2.2 Η συνάρτηση κατακερματισμού του Jenkins

Η συνάρτηση κατακερματισμού του Jenkins παίρνει ως είσοδο μια συμβολοσειρά από byte, οποιουδήποτε μήκους και παράγει έξοδο 4 byte. Η συνάρτηση, όπως φαίνεται και στο 2, παίρνει ένα κλειδί

και στη συνέχεια εκτελεί διάφορες πράξεις πάνω σε κάθε χαρακτήρα του κείμενου εισόδου. Η συνάρτηση του Jenkins συνήθως χρησιμοποιείται για αναζήτηση σε πίνακες κατακερματισμού, για αυτό ακριβώς το λόγο χρησιμοποιείται και στο V8. Ο V8 τη χρησιμοποιεί για να κρατάει συναρτήσεις και κώδικες που έχει μεταφράσει δυναμικά (είτε από eval είτε από Function). Όπως αναφέραμε και πριν το κλειδί της συνάρτησης κατακερματισμού είναι η θέση μνήμης του αντικειμένου.

Algorithm 2 Jenkins one at a time hash function

```
function jenkins_one_at_a_time_hash(key,length)
    i ← 0;
    hash ← 0;
    while i ≠ length do
        hash ← hash + key[i];
        hash ← hash + hash << 10;
        hash ← hash ⊕ hash >> 6;
        i ← i + 1
    end while
    hash ← hash + hash << 3;
    hash ← hash ⊕ hash >> 11;
    hash ← hash + hash << 15;
return hash
end function
```

4.2.3 Μνημόνευση στην υλοποίηση μας

Είναι σημαντικό να θυμίσουμε στον αναγνώστη πως, όπως αναφέραμε και στο κεφάλαιο 3.3.3, ο αναλυτής δεν είναι αναγκαίο να πάρει ως είσοδο όλο το πρόγραμμα, μπορεί να αναλύσει και μόνο μια συνάρτηση. Για αυτό για να ξέρουμε ποιους κώδικες έχουμε ξανά-αναλύσει χρησιμοποιήσαμε ένα unordered_map από την stl της C++ και αποθηκεύουμε έναν 64-bit unsigned int ο οποίος, αποτελούνταν στα πρώτα 32 ψηφία από το hash του κώδικα, στα επόμενα 16 ψηφία από το σημείο του κειμένου όπου άρχιζε η συνάρτηση προς ανάλυση (αν δίναμε όλο το πρόγραμμα για ανάλυση τότε αυτό ήταν 0) και μετά το σημείο του κειμένου που τέλειωνε η συνάρτηση προς ανάλυση (αν δίναμε όλο το πρόγραμμα για ανάλυση τότε αυτό ήταν 0). Η μορφή του φαίνεται καλύτερα παρακάτω:

$$z = \overbrace{00111100111100011001100010100000}^{\text{32-bit / Το hash του προγράμματος}} \mid \overbrace{0001001110010000}^{\text{16-bit / Η αρχή της συνάρτησης}} \mid \overbrace{0001101110000111}^{\text{16-bit / Το τέλος της συνάρτησης}}$$

Η παραπάνω στρατηγική hashing έχει κάποιες αδυναμίες. Πρώτα από όλα αν η συνάρτηση τις οποίας θέλουμε να υπολογίσουμε το hash αν βρίσκεται μετά τον 2^{16}-οστό χαρακτήρα του προγράμματος τότε, θα μπορούσε να υπάρχει σύγκρουση με συνέπεια να μη γίνει εντοπισμός για λάθη σε κάτι που έχει λάθη. Αυτό θα μπορούσε να το εκμεταλλευτεί κάποιος κακόβουλος. Μια άλλη αδυναμία του παραπάνω είναι ότι το seed που χρησιμοποιείται για να παραχθεί το hash του προγράμματος εξαρτάται από τη θέση μνήμης στην οποία είναι αποθηκευμένη η συμβολοσειρά του προγράμματος, κάτι το οποίο αλλάζει από τη μια εκτέλεση του V8 σε μια άλλη. Κανένα από τα δύο δε μας προκαλεί πρόβλημα στα πλαίσια της έρευνας μας όμως, αφού και τα δύο παραπάνω προβλήματα θα μπορούσαν να αποφευχθούν αν γινόταν μια πιο προσεκτική επιλογή του τρόπου κατακερματισμού κάτι το οποίο ξεπερνά τα πλαίσια της παρούσας εργασίας.

Εδώ είναι σημαντικό να σημειώσουμε πως αφού ο V8 ξεκινάει να αναλύει ένα πρόγραμμα πριν να κατέβει όλο μας είναι αδύνατο να ξέρουμε αν το έχουμε ξανασυναντήσει. Για αυτό όταν ο αναλυτής καλείται από το background-parsing-task.cc πάντα χρησιμοποιούμε τον ParserNE, ενώ όταν καλείται από το parsing.cc τότε κοιτάμε αν το κλειδί υπάρχει στο unordered_map, αν υπάρχει δεν υπάρχει τότε

πρέπει να ελέγξουμε αν έχει λάθη ο κώδικας και καλούμε τον ParserNE, αν ο ParserNE επιστρέψει χωρίς να εντοπίσει κάποιο λάθος τότε παράγουμε το κλειδί σύμφωνα με το προηγούμενο σχήμα και το αποθηκεύουμε στο unordered_map. Έτσι την επόμενη φορά θα δούμε ότι το κλειδί είναι μέσα στο unordered_map και θα καλέσει τον ParserWE.

Επίσης όπως έχουμε αναφέρει ο Chrome σηκώνει ένα διαφορετικό στιγμιότυπο του V8 για κάθε καρτέλα του. Το κάθε στιγμιότυπο είναι ανεξάρτητο και χρησιμοποιεί διαφορετικό unordered_map, οπότε αν κάτι αναλυθεί σε μια καρτέλα τα άλλα στιγμιότυπα του V8 δεν το γνωρίζουν. Τέλος δεν αποθηκεύουμε κάπου τα στοιχεία του unordered_map, οπότε όταν πεθάνει κάποιο στιγμιότυπο του V8 όλα τα κλειδιά των προγραμμάτων που έχουν αναλυθεί χάνονται. Θα μπορούσαμε να αποθηκεύουμε το κλειδί που παράγουμε στο αρχείο του Chrome στο δίσκο, αλλά πιστεύουμε πως αυτό ξεφεύγει από τα πλαίσια της παρούσας εργασίας καθώς η υλοποίηση μας αποτελεί περισσότερο μια απόδειξη της πρακτικότητας της ιδέας παρά μια υλοποίηση για τον χρήστη.

Κεφάλαιο 5

Αποτελέσματα

5.1 Μετρήσεις

Για να πάρουμε μετρήσεις χρησιμοποιήσαμε το script του V8 tools/callstats.py, το οποίο καλεί το web-page-replay(WPR) (`https://github.com/chromium/web-page-replay`). Το script callstats χρησιμοποιόντας το WPR επισκέπτεται τις ιστοσελίδες που έχουμε επιλέξει και μετράει το χρόνο που καταναλώνεται σε κάθε συνιστώσα του V8 και σε κάθε συνάρτηση. Μας δίνει επίσης την επιλογή να επιλέξουμε πόσες φορές θα επισκεφτεί την κάθε σελίδα, για να μειώσουμε τον θόρυβο, και αν θέλουμε να κάνουμε ανανεώσεις σε κάθε ιστοσελίδα. Ένα νέο στιγμιότυπο του Chrome σηκώνεται για κάθε επίσκεψη το οποίο κλείνει όταν τελειώσει η φόρτωση.

Επίσης χρησιμοποιήσαμε το εργαλείο cpufrequtil των linux, για να μειώσουμε όσο γίνεται την επιρροή από εξωτερικούς παράγοντες, πχ εκτέλεση άλλων προγραμμάτων. Οι μετρήσεις για τα αποτελέσματα που παρουσιάζονται και παρακάτω έγιναν σε τετραπύρηνο επεξεργαστή Intel(R) Core(TM) i5-7200U CPU, με 8GB RAM και Ubuntu 17.10. Με το cpufrequtil εξασφαλίσαμε ότι ο επεξεργαστής έτρεχε καθ‘ όλη τη διάρκεια των πειραμάτων με μέγιστη συχνότητα (3,10 GH).

Web Page Replay

Το web-page-replay είναι ένα εργαλείο του Chromium. Μπορεί να χρησιμοποιηθεί για τη συλλογή των πακέτων που στέλνει ο υπολογιστής μας ή των πακέτων που δέχεται ο υπολογιστής μας. Αυτό το καταφέρνει χρησιμοποιόντας τοπικό DNS και HTTP proxy, επί της ουσίας όλη η κίνηση του υπολογιστή περνά από το web-page-replay και πριν να αναμεταδώσει τα πακέτα στον πραγματικό τους προορισμό, τα καταγράφει. Το web-page-replay έχει τη δυνατότητα να μιμείται την κίνηση που έχει καταγράψει πλήρως, στέλνει τα πακέτα με την ίδια σειρά και με τον ίδιο ρυθμό, αυτό εξασφαλίζει κάποιο επίπεδο συνέπειας μεταξύ των αποτελεσμάτων κάτι το οποίο είναι αδύνατο να έχει κάποιος στο συγκεκριμένο πρόβλημα καθώς υπάρχουν πολύ παράγοντες που επηρεάζουν το χρόνο εκτέλεσης του προγράμματος. Το web-page-replay πλέον έχει αντικατασταθεί (δεν χρησιμοποιείται πλέον) από τη Google με το WebPageReplayGO `https://github.com/catapult-project/catapult/blob/master/web_page_replay_go/README.md`, παρόλα αυτά χρησιμοποιήσαμε το web-page-replay γιατί για αυτό είχαμε δεδομένα.

Ιστοσελίδες

Τα δεδομένα που χρησιμοποιήσαμε για να τρέξουμε το WPR αποτελούνται από επισκέψεις στις 25 πιο διάσημες σελίδες του διαδικτύου, η καταγραφή έγινε τον Ιούλιο του 2015. Οι ιστοσελίδες που χρησιμοποιήθηκαν είναι οι ίδιες που χρησιμοποιούνται από τους μηχανικούς του V8 για να κάνουν μετρήσεις και φαίνονται στον πίνακα 5.1

https://www.google.de/search?q=v8
https://www.youtube.com
https://www.facebook.com/shakira
http://www.baidu.com/s?wd=v8
http://www.yahoo.co.jp
http://www.amazon.com/s/?field-keywords=v8
https://hi.wikipedia.org/wiki/%E0%A4%AE%E0%A5%81%E0%A4%96%E0%A4%AA%E0%A5%83%E0%A4%B7%E0%A5%8D%E0%A4%A0
http://www.qq.com
https://www.twitter.com/taylorswift13
https://www.reddit.com
http://www.ebay.fr/sch/i.html?_nkw=v8
http://edition.cnn.com
http://world.taobao.com
http://www.instagram.com/archdigest
https://www.linkedin.com/m/
http://www.msn.com/ar-ae
http://www.bing.com/search?q=v8+engine
http://www.pinterest.com/categories/popular
http://www.sina.com.cn
http://weibo.com
http://yandex.ru/search/?text=v8
http://www.wikiwand.com/en/hill
http://meta.discourse.org
http://reddit.musicplayer.io
https://inbox.google.com
http://maps.google.co.jp/maps/search/restaurant+tokyo
https://adwords.google.com

Πίνακας 5.1: Οι 25 κορυφαίες ιστοσελίδες

5.2 Αποτελέσματα

Ακόμα και με το web-page-replay υπήρχε θόρυβος στις μετρήσεις μας. Για αυτό για να πάρουμε μετρήσεις χρησιμοποιήσαμε την επιλογή του script callstats.py ώστε να επισκεφτεί κάθε ιστοσελίδα 20 φόρες. Επίσης επειδή σε κάθε επίσκεψη χρησιμοποιείται διαφορετικό στιγμιότυπο το Chrome επιλέξαμε να κάνει 10 ανανεώσεις σε κάθε επίσκεψη, αλλιώς δε θα είχαν νόημα οι μετρήσεις μας καθώς διαφορετικό στιγμιότυπο του Chrome σημαίνει διαφορετικό στιγμιότυπο του V8, άρα δεν θα βλέπαμε καμία διαφορά αν δεν κάναμε ανανεώσεις. Στους πίνακες 5.2 και 5.3 φαίνονται τα αποτελέσματα μας. Τα αποτελέσματα των μετρήσεων είναι χωρισμένα σε διάφορες ομάδες, όπως τα παρήγαγε από το callstats, επίσης για κάθε site φαίνεται ο χρόνος χρησιμοποιόντας το κανονικό (vanilla) V8, χωρίς τις αλλαγές μας, και χρησιμοποιόντας το αλλαγμένο (custom) V8. Το Parse και το Parse-Background βρίσκονται σε διαφορετικές ομάδες. Επίσης στον πίνακα 5.4 παρατηρούμε τη διαφορά μεταξύ στον δύο υλοποιήσεων καλύτερα, καθώς είναι εκφρασμένοι σε ποσοστά. Το total υπολογίστηκε χωρίς να λάβουμε υπόψη το adwords καθώς ο χρόνος του adwords είναι τόσο μεγαλύτερος που επικρατεί επί των άλλων. Στην εικόνα 5.1 φαίνονται οι χρόνοι του συντακτικού αναλυτή για κάθε ιστοσελίδα εκτός από το adwords. Επίσης παρουσιάζουμε τα σχήματα 5.2 τα οποία παράχθηκαν από το callstats.py και δείχνουν τη διαφορά των δύο αναλυτών επί εκτέλεσης όλου του προγράμματος για διάφορες ιστοσελίδες. Στα διαγράμματα που ακολουθούν custom είναι η δική μας υλοποίηση και vanilla είναι ο V8 χωρίς καμία αλλαγή και όλοι οι χρόνοι είναι σε ms.

		Total	Blink C++	JavaScript	V8 C++	Compile	Parse	IC	Parse Background	Optimize	GC	API
total	custom	156239.1	52686.0	38031.0	17534.7	10808.8	7657.7	7590.9	6479.2	6319.6	5542.0	3583.7
	vanilla	156402.3	52329.1	38092.0	17472.7	10795.7	8230.5	7609.2	6448.4	6329.0	5523.2	3566.1
adwords	custom	130270.6	31468.9	41602.2	9532.8	9632.9	12561.6	4096.2	1414.3	12463.7	6371.9	1126.4
	vanilla	131612.8	31195.6	41387.9	9508.5	9876.1	14270.3	4068.5	1407.2	12372.2	6407.2	1118.6
amazon	custom	4273.0	1268.8	777.8	450.2	402.9	332.4	296.7	342.2	26.9	165.8	188.2
	vanilla	4251.5	1259.6	779.5	444.4	389.5	346.3	297.2	354.8	26.8	164.0	188.2
baidu	custom	1337.2	687.6	197.9	108.4	66.0	65.1	115.4	27.8	0.0	33.4	35.0
	vanilla	1310.0	653.5	198.2	107.5	66.0	70.0	116.2	30.0	0.0	33.3	34.4
bing	custom	362.0	154.8	65.8	26.1	20.3	20.6	28.6	0.0	6.0	12.7	26.4
	vanilla	370.3	161.3	66.4	26.3	20.5	21.8	29.3	0.0	6.0	11.7	27.0
cnn	custom	14282.0	5438.4	4236.1	1558.3	686.4	677.9	338.7	277.7	450.5	342.0	275.4
	vanilla	14434.8	5522.0	4284.6	1535.3	680.1	718.6	339.0	280.6	456.2	341.5	275.8
discourse	custom	5283.1	570.9	1728.8	748.0	260.1	341.0	680.7	61.5	541.2	312.8	37.3
	vanilla	5301.6	573.9	1727.2	745.9	261.0	361.5	679.3	62.2	541.3	311.5	37.2
ebay	custom	9260.3	2679.0	2026.8	1182.1	534.5	423.1	700.6	355.8	221.8	352.4	784.4
	vanilla	9264.8	2655.2	2023.6	1177.8	535.0	460.2	700.5	348.3	225.2	351.8	785.9
facebook	custom	9339.5	1470.3	2035.2	1187.5	567.5	457.1	509.6	1572.2	1147.6	335.7	56.2
	vanilla	9399.0	1477.0	2046.4	1183.0	571.5	501.4	510.9	1571.2	1140.8	340.9	56.1
google	custom	3029.0	664.7	268.8	252.5	527.7	368.4	213.3	549.4	0.9	113.8	68.7
	vanilla	3076.7	662.4	268.7	254.6	530.0	401.5	214.4	560.4	0.8	115.1	68.8
inbox	custom	446.5	131.9	51.1	65.0	17.1	27.4	56.9	0.0	0.0	70.1	26.9
	vanilla	447.5	132.7	50.9	64.3	16.6	27.6	55.7	0.0	0.0	70.8	27.7
instagram	custom	2806.2	837.8	546.5	353.3	242.7	224.4	245.1	142.1	3.5	109.6	99.8
	vanilla	2822.2	835.8	547.6	355.1	241.8	240.9	247.5	140.3	3.8	109.1	99.6
linkedin	custom	10881.8	1111.0	4392.4	1192.5	259.0	376.3	1140.6	221.4	1403.4	720.4	63.7
	vanilla	10892.0	1111.4	4399.6	1188.2	259.5	387.0	1141.3	225.1	1413.7	701.0	64.1
maps	custom	2268.0	593.9	443.0	98.0	219.2	228.0	81.6	85.6	437.2	69.1	14.0
	vanilla	2260.5	588.2	445.4	98.7	217.7	232.2	81.6	85.2	428.4	70.6	14.2
msn	custom	5028.4	2387.3	669.3	356.8	359.1	277.3	234.5	322.3	36.9	177.4	207.1
	vanilla	4912.3	2222.2	673.5	357.9	364.6	302.2	236.6	326.9	37.0	183.9	207.2

Πίνακας 5.2: Οι χρόνοι των δύο υλοποιήσεων

		Total	Blink C++	JavaScript	V8 C++	Compile	Parse	IC	Parse Background	Optimize	GC	API
pinterest	custom	6744.2	2774.3	880.1	621.2	368.0	631.9	356.6	303.2	449.5	274.9	83.5
	vanilla	6599.6	2648.1	879.7	619.8	367.5	670.2	357.2	245.7	449.2	277.4	83.8
qq	custom	42803.2	17155.3	13005.5	5895.9	3642.3	939.4	347.4	41.7	346.2	813.5	616.6
	vanilla	42727.0	17101.9	12971.9	5873.5	3609.5	1026.6	348.0	43.3	348.0	811.0	593.9
reddit	custom	1685.6	551.5	216.5	178.7	168.3	142.3	143.5	87.0	0.4	87.7	110.6
	vanilla	1709.9	558.9	217.5	178.3	174.1	152.3	145.1	89.0	0.4	83.8	110.7
reddit musicplayer	custom	4516.1	2126.0	637.9	365.0	209.0	215.9	250.4	211.6	224.8	185.1	91.1
	vanilla	4410.7	2000.6	638.0	364.3	207.1	227.3	251.1	223.1	225.9	181.8	92.1
sina	custom	6994.3	3452.3	1245.2	353.3	523.2	376.2	218.9	199.2	51.8	371.8	201.3
	vanilla	7051.4	3467.5	1251.5	354.7	523.0	409.9	220.6	196.7	52.0	372.6	201.5
taobao	custom	2990.9	1134.5	577.9	311.3	200.1	193.8	187.4	160.0	42.3	130.3	54.0
	vanilla	3075.5	1196.9	580.4	313.9	199.8	207.5	189.2	159.0	39.5	134.5	54.7
twitter	custom	4524.4	1393.9	875.3	499.3	363.6	281.2	299.0	438.1	134.7	186.7	52.1
	vanilla	4591.8	1404.9	876.5	497.7	374.4	311.2	298.7	447.7	136.8	190.4	53.2
weibo	custom	5012.7	1818.8	1049.5	420.0	259.2	262.7	263.2	234.1	467.5	126.1	111.3
	vanilla	5042.2	1818.8	1054.1	418.1	260.8	293.6	263.6	233.8	466.6	122.3	110.9
wikipedia	custom	1983.4	401.3	317.0	280.4	168.2	179.4	96.6	179.1	209.7	111.6	38.5
	vanilla	2014.7	414.8	317.5	280.6	169.8	195.7	97.1	179.2	211.0	108.6	39.3
wikiwand	custom	3255.7	1374.0	778.5	270.9	173.6	149.0	167.1	87.0	110.0	91.6	54.6
	vanilla	3274.9	1379.4	783.4	270.7	172.8	159.0	167.3	84.6	112.4	91.0	54.2
yahoo	custom	865.6	378.1	72.9	71.5	71.8	64.1	73.6	27.4	0.0	48.5	56.9
	vanilla	849.9	356.8	72.3	71.5	73.0	68.6	73.9	29.0	0.0	47.9	56.3
yandex	custom	2240.9	872.2	436.4	291.9	123.5	102.3	234.8	57.5	2.2	72.0	48.6
	vanilla	2231.9	855.6	436.1	291.6	124.0	105.9	236.1	59.4	2.2	72.5	48.3
youtube	custom	4025.2	1256.5	498.8	392.9	377.0	299.2	309.2	481.1	5.2	227.2	179.1
	vanilla	4079.7	1269.1	501.8	394.7	388.5	330.4	310.0	478.2	5.2	224.1	179.0

Πίνακας 5.3: Οι χρόνοι των δύο υλοποιήσεων(συνέχεια)

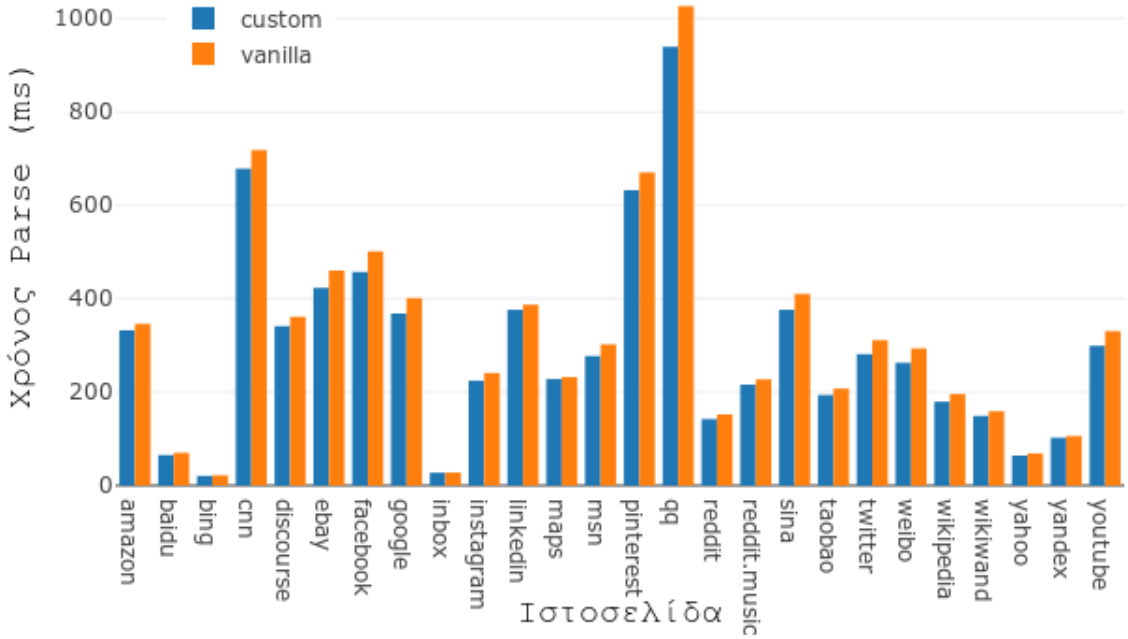
Site	Parser Speedup (%)	Total Speedup (%)	Difference (ms)
total	6.96	0.37	572.8
adwords	11.97	1.30	1708.7
amazon	4.01	0.33	13.9
baidu	7.00	0.37	4.9
bing	5.50	0.32	1.2
cnn	5.66	0.28	40.7
discourse	5.67	0.39	20.5
ebay	8.06	0.40	37.1
facebook	8.84	0.47	44.3
google	8.24	1.08	33.1
inbox	0.72	0.04	0.2
instagram	6.85	0.58	16.5
linkedin	2.76	0.10	10.7
maps	1.81	0.19	4.2
msn	8.24	0.51	24.9
pinterest	5.71	0.58	38.3
qq	8.49	0.20	87.2
reddit	6.57	0.58	10.0
reddit musicplayer	5.02	0.26	11.4
sina	8.22	0.48	33.7
taobao	6.60	0.45	13.7
twitter	9.64	0.65	30.0
weibo	10.52	0.61	30.9
wikipedia	8.33	0.81	16.3
wikiwand	6.29	0.31	10.0
yahoo	6.56	0.53	4.5
yandex	3.40	0.16	3.6
youtube	9.44	0.76	31.2

Πίνακας 5.4: Επιτάχυνση

5.2.1 Σχόλια

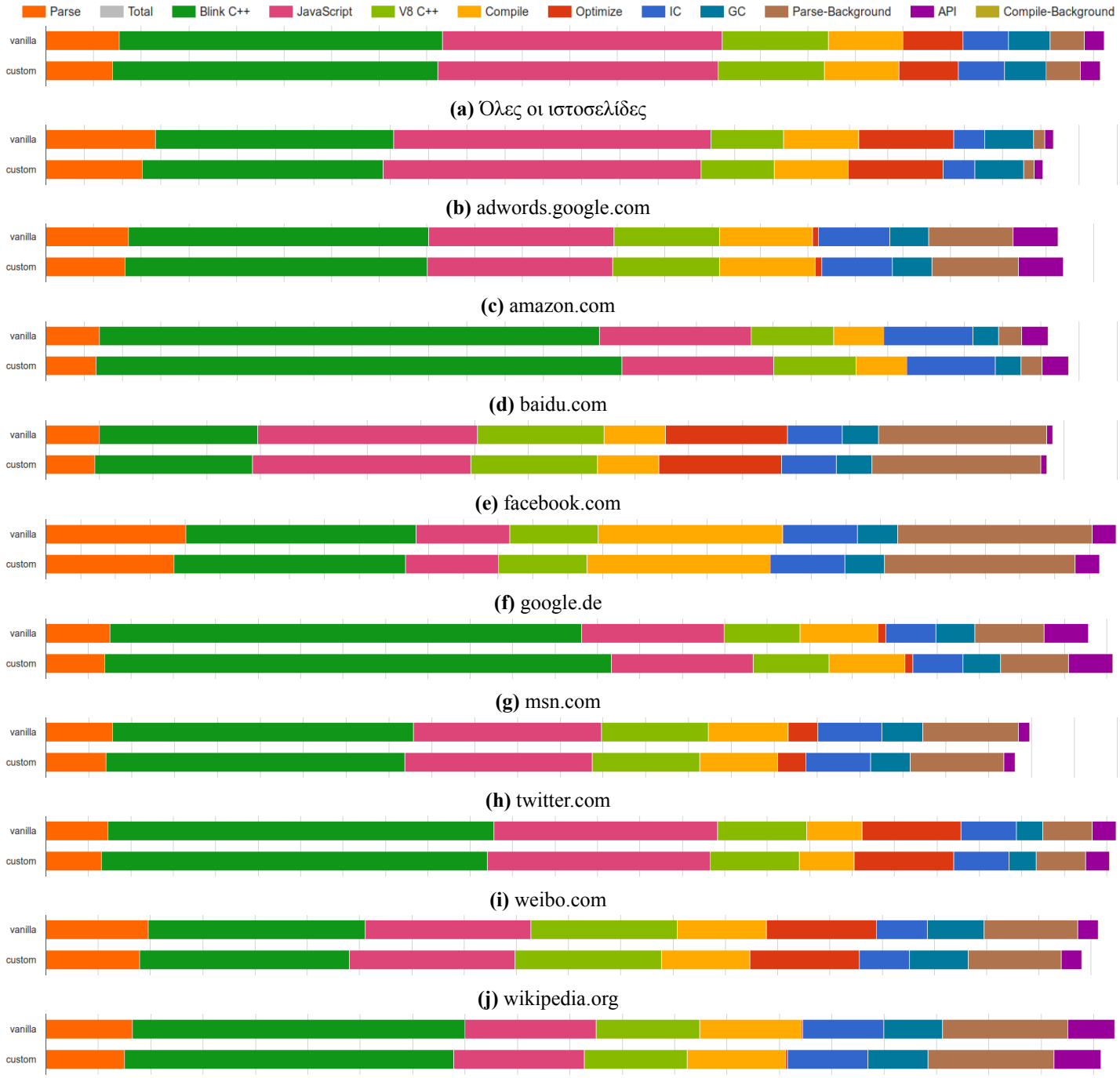
Παρατηρούμε ότι υπάρχει διακύμανση μεταξύ ομάδων που δεν θα έπρεπε, κανονικά θα έπρεπε να βλέπουμε διαφορά μόνο στο Parse, αφού μόνο εκεί υπάρχει διαφορά και όλα τα άλλα είναι ο ακριβώς ίδιος κώδικας. Παρόλα τα όσα προσπαθήσαμε για να μειώσουμε το θόρυβο ακόμα είναι αισθητός, το οποίο είναι λογικό καθώς το σύστημα το οποίο προσπαθούμε να μετρήσουμε είναι πολύ σύνθετο (ο V8 είναι ένα κομμάτι του Chrome και ο συντακτικός αναλυτής είναι ένα κομμάτι του V8) και το περιβάλλον στο οποίο εκτελείται το πρόγραμμα μας είναι επίσης πολύ σύνθετο, αφού πρέπει να περάσει από το δίκτυο του υπολογιστή και στη συνέχεια χρησιμοποιεί πλήρως τον επεξεργαστή. Παρόλο όμως που υπάρχει αυτή η διακύμανση στα αποτελέσματα το Parse στο custom είναι σταθερά πιο γρήγορο, βέβαια ακόμα και σε αυτό η διαφορά δεν είναι σταθερή.

Από τον πίνακα των επιταχύνσεων 5.4 παρατηρούμε ότι η επιτάχυνση που επιτύχαμε διαφέρει πολύ από ιστοσελίδα σε ιστοσελίδα, καθώς το εύρος των επιταχύνσεων μόνο στον Parser είναι από 0,72% έως 11,97%. Όπως περιμέναμε η επιτάχυνση είναι ανάλογη του χρόνου που χρειάζεται για να αναλυθεί η ιστοσελίδα, κάτι το οποίο είναι λογικό. Επίσης σε κάποιες ιστοσελίδες, όπως το bing και το inbox, ο χρόνος που χρειάζεται για τη συντακτική ανάλυση είναι τόσο μικρός που η επιτάχυνση που πετυχαίνουμε είναι στην τάξη μεγέθους του θορύβου.



Σχήμα 5.1: Οι χρόνοι του αναλυτή για τις δύο υλοποιήσεις

Τέλος να σημειώσουμε πως η επιτάχυνση σχετίζεται και με την ταχύτητα της σύνδεσης, καθώς όσο πιο αργή είναι μια σύνδεση τόσο θα αργήσει η λήψη της ιστοσελίδας και τόσο περισσότερο χρόνο θα περάσει στο Parse-Background και λιγότερο στο Parse. Στα πειράματα μας το WPR προσομοιώνει την ταχύτητα της σύνδεσης από όταν έκανε τη λήψη καθώς στέλνει τα πακέτα με τον ίδιο ρυθμό που τα έλαβε.



(a) Όλες οι ιστοσελίδες

(b) adwords.google.com

(c) amazon.com

(d) baidu.com

(e) facebook.com

(f) google.de

(g) msn.com

(h) twitter.com

(i) weibo.com

(j) wikipedia.org

(k) youtube.com

Σχήμα 5.2: Χρόνος ανά συνιστώσα

Κεφάλαιο 6

Συμπεράσματα

6.1 Συνεισφορά

Η συνεισφορά και τα συμπεράσματα της παρούσας εργασίας συνοψίζονται στα παρακάτω:

- Καταφέραμε να αποφεύγουμε τον εντοπισμό λαθών κατά τη συντακτική ανάλυση του V8 όταν έχει αναλυθεί στο παρελθόν το ίδιο πρόγραμμα. Για να το υλοποιήσουμε χρησιμοποιήσαμε το μοντέλο CRTP και το unordered_map της stl της C++.
- Μετρήσαμε την επιτάχυνση της υλοποίησης μας και παρόλο που δεν καταφέραμε να έχουμε αποτελέσματα χωρίς θόρυβο, μπορούμε να δούμε ότι καταφέραμε μια επιτάχυνση της τάξης του 0.5% επί του συνολικού χρόνου και του 7% επί του χρόνου στον Parser, ενώ σε ορισμένες περιπτώσεις η επιτάχυνση έφτασε το 1,3% επί του συνολικού και 12% στον Parser.
- Η συνεισφορά και η αξία αυτής της εργασίας στον πραγματικό κόσμο φαίνεται από σπάνια κάποιος επισκέπτεται μια σελίδα μόνο μια φορά. Επίσης οι σελίδες που ανήκουν στον ίδιο τομέα μοιράζονται μεταξύ τους μεγάλα κομμάτια κώδικα JavaScript. Τέλος οι πιο πολλές ιστοσελίδες χρησιμοποιούν κάποιες βιβλιοθήκες JavaScript (jQuery, Angular, κ.λ.π.). Σε όλες τις παραπάνω περιπτώσεις δε χρειάζεται να γίνει ανάλυση για συντακτικά λάθη πάνω από μια φορά. Άρα παρόλο που η μνήμη της υλοποίησης μας περιορίζεται σε μια καρτέλα του Chrome, μπορεί να δώσει επιτάχυνση στο χρήστη.

6.2 Μελλοντική δουλειά

Υπάρχουν διάφοροι τομείς στους οποίους μπορεί να επεκταθεί και να βελτιωθεί η δουλειά που έγινε σε αυτή την εργασία. Συνοπτικά παρουσιάζονται παρακάτω διάφορες ιδέες για μελλοντική δουλειά:

- Χρησιμοποιήσαμε τα υπάρχοντα τεστ του V8 για να ελέγξουμε ότι όλα δουλεύουν όπως πρέπει μετά τις αλλαγές μας, αλλά δεν υλοποιήσαμε τεστ για να μπορούμε αυτόματα να επικυρώσουμε την ορθότητα του κώδικα μας (η βεβαίωση ότι ο κώδικας μας δουλεύει όπως πρέπει έγινε με ad hoc τρόπο). Το να γραφτούν τεστ είναι πολύ σημαντικό για να ολοκληρωθεί η υλοποίηση μας, άλλα ξέφευγε από τα πλαίσια από τα πλαίσια αυτής της εργασίας.
- Ο τρόπος που γίνεται ο έλεγχος για το αν έχει αναλυθεί στο παρελθόν ένα πρόγραμμα μας περιορίζει καθώς όταν κλείσει η καρτέλα του Chrome το map χάνεται. Η καλύτερη και πιο λειτουργική λύση είναι να προστίθενται τα κλειδιά στο cache αρχείο του Chrome και κάθε φορά που ανοίγει μια καρτέλα να φορτώνονται στο V8.
- Η συνάρτηση κατακερματισμού που χρησιμοποιούμε είναι η συνάρτηση κατακερματισμού του V8, το πρόβλημα με αυτό είναι ότι η συνάρτηση κατακερματισμού του V8 χρησιμοποιεί ως

αρχικό σπόρο τη θέση μνήμης στην οποία είναι αποθηκευμένο το πρόγραμμα. Για να μπορέσει η υλοποίηση που περιγράψαμε παραπάνω να δουλέψει θα έπρεπε η συνάρτηση κατακερματισμού να επιστρέφει την ίδια τιμή μεταξύ διαφορετικών στιγμιοτύπων του V8, για να το καταφέρουμε αυτό θα έπρεπε να αλλάξει η συνάρτηση κατακερματισμού.

- Ο τρόπος που παράγουμε τα κλειδιά για το map θα έπρεπε επίσης να αλλάξει. Στην υλοποίηση μας αν το κείμενο είναι μεγάλο μπορεί να υπάρχει σύγκρουση κλειδιών αν δύο συναρτήσεις βρίσκονται (ξεκινάνε/τελειώνουν) σε κατάλληλα σημεία του κειμένου. Αυτό θα μπορούσε αν το εκμεταλλευτεί κάποιος κακόβουλος. Για να φτιαχτεί αυτό θα έπρεπε να υλοποιηθεί ένας πιο έξυπνος παραγωγής κλειδιών.

Βιβλιογραφία

[Aho14] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, *Μεταγλωττιστές: Αρχες, Τεχνικές, & Εργαλεία*, vol. 2, Εκδόσεις Νέων Τεχνολογιών, 2014.

[Clif17] Daniel Clifford, "Launching Ignition and TurboFan", 2017.

[Copl95] James O. Coplien, "Curiously Recurring Template Patterns", Technical report, AT&T Bell Laboratories, 1995.

[ECMA] ECMAScript, https://www.ecma-international.org/ecma-262/5.1/#sec-16, *Early errors in JavaScript*.

[Flan06] David Flanagan, *Nested functions*, O'Reilly & Associates, 5 edition, 2006.

[Guo15] Yang Guo, "Code Caching", 2015.

[Höl15] Marja Hölttä and Daniel Vogelheim, "New JavaScript techniques for rapid page loads", 2015.

[McIl16] Ross McIlroy, "Firing up the Ignition Interpreter", 2016.

[MDN a] MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop, *Concurrency model and Event Loop*.

[MDN b] MDN Web Docs, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototyp *Inheritance and the prototype chain*.

[MDN c] MDN Web Docs, https://developer.mozilla.org/bm/docs/Web/JavaScript, *Javascript*.

[Osma17] Addy Osmani, "Production Web Apps Performance Study", Technical report, Google, https://github.com/GoogleChromeLabs/discovery/issues/1, 2017.

[Webb03] Adam Brooks Webber, *JavaScript delegation*, Franklin, Beedle & Associates Inc., 2003.