ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΤΟ ΠΡΟΤΥΠΟ WEBASSEMBLY

Διπλωματική Εργασία

του

Καρέλα Βασιλείου

Πτυχιούχου πληροφορικής ΕΑΠ

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής

Κασκάλης Θεόδωρος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24 Φεβρουαρίου 2021

Κασκάλης Θεόδωρος	Μαργαρίτης Κωνσταντίνος	Ξυνόγαλος Στυλιανός
....	....	....

Καρέλας Βασίλειος

....

Θεσσαλονίκη, 2021

Ευχαριστίες

Θέλω να εκφράσω τις πιο εγκάρδιες ευχαριστίες μου στον επιβλέποντα καθηγητή κ. Κασκάλη Θεόδωρο για τη συμβολή και βοήθειά του καθ' όλη τη διάρκεια εκπόνησης της παρούσας εργασίας. Ο τρόπος σκέψης και το πλήθος γνώσεων του, θα αποτελέσουν οδοδείκτη για την περεταίρω πρόοδο μου. Επίσης, ευχαριστώ την οικογένειά μου, για την αμέριστη συμπαράσταση της στην επίπονη αλλά και δημιουργική αυτή προσπάθεια μου.

Περίληψη

Το πρότυπο WebAssembly επιτρέπει την παραγωγή κώδικα, μέσω της μεταγλώττισης έτοιμων προγραμμάτων που έχουν αναπτυχθεί με άλλες γλώσσες προγραμματισμού, ο οποίος μπορεί να φορτωθεί και να εκτελεστεί από ένα πρόγραμμα πλοήγησης. Βασικός σκοπός του προτύπου είναι η δημιουργία Διαδικτυακών εφαρμογών υψηλών επιδόσεων, αξιοποιώντας έτοιμο κώδικα που έχει ήδη δημιουργηθεί με πιο φιλικές γλώσσες. Με το πρότυπο WebAssembly, μπορούν να ξεπεραστούν κάποιες αδυναμίες που παρουσιάζονται σε κάποιες πολύ διαδεδομένες γλώσσες ανάπτυξης Διαδικτυακών εφαρμογών, όπως η γλώσσα Javascript.

Στην παρούσα εργασία γίνεται μία παρουσίαση του προτύπου WebAssembly και των χαρακτηριστικών του, καθώς επίσης και μία ιστορική αναδρομή της πορείας που ακολουθήθηκε μέχρι την πλήρη ανάπτυξη του προτύπου. Παρουσιάζονται επίσης και οι προγενέστερες προσπάθειες που έγιναν, από τις μεγάλες εταιρείες του χώρου, για την ανάπτυξη γρήγορων εφαρμογών Διαδικτύου. Στη συνέχεια περιγράφονται κάποιες από τις σχεδιαστικές αδυναμίες που εντοπίζονται στην γλώσσα Javascript και τις οποίες καλείται να καλύψει το πρότυπο WebAssembly. Τέλος, γίνεται μία συγκριτική μελέτη των επιδόσεων ενός συνόλου προγραμμάτων που αναπτύχθηκαν, τόσο σε WebAssembly όσο και Javascript, ώστε να συγκριθούν οι αντίστοιχοι χρόνοι εκτέλεσης των προγραμμάτων. Από την μελέτη που πραγματοποιήθηκε, μπορούμε να πούμε ότι η WebAssembly υπερτερεί σε επιδόσεις σε σχέση με τη Javascript, ο βαθμός επιτάχυνσης όμως εξαρτάται σε μεγάλο βαθμό από τον τύπο των προγραμμάτων που εκτελούνται.

Λέξεις Κλειδιά: WebAssembly, Javascript, συγκριτική μελέτη, απόδοση Διαδικτυακών εφαρμογών

Abstract

WebAssembly is a specification that allows the production of code, through the compilation of previously developed applications by different programming languages, that can be loaded and executed in the environment of a Web browser. The basic goal of the specification is the creation of high-performance Internet applications, taking advantage of existing code developed by more human-friendly languages. WebAssembly tries to face some of weaknesses that appear in some popular Web programming languages, like Javascript.

This thesis, first presents the characteristics of the WebAssembly specification and makes an overview of the route that was followed till the present day. It also presents the attempts. made by the big Internet companies. in order to develop a framework for the development of fast applications. Some of the weaknesses of Javascript, that WebAssembly tries to bypass, are also described. Finally, a comparative study of the efficiency of the applications developed by WebAssembly compared to Javascript applications, is also presented. From the study we can deduce that WebAssembly applications are faster than Javascript applications with the same functionality, but the speedup factor depends strongly on the type of the processing that takes place.

Keywords: WebAssembly, Javascript, comparative study, Internet applications performance

Περιεχόμενα

1 Εισαγωγή 9

1.1 Περιγραφή και σημαντικότητα του θέματος 9

1.2 Σκοπός και Στόχοι 11

1.3 Διάρθρωση της εργασίας 11

2 Θεωρητικό υπόβαθρο και σχετική έρευνα 13

2.1 Η γλώσσα JavaScript 13

2.1.1 Χαρακτηριστικά της γλώσσας JavaScript 16

2.1.2 Χρήσεις της JavaScript 18

2.2 Αδυναμίες της γλώσσας JavaScript και αντιμετώπισή τους 20

2.2.1 Στιγμιαία μεταγλώττιση (Just-In-Time compilation) 21

2.3 asm.js 23

2.4 Emscripten και LLVM 24

2.5 Native Client της Google 27

3 Το πρότυπο WebAssembly 29

3.1 Ιστορική αναδρομή και εξάπλωση του προτύπου 29

3.1.1 Υποστήριξη γλωσσών προγραμματισμού 30

3.2 Τρόπος εκτέλεσης κώδικα WebAssembly 32

3.3 Αρχιτεκτονική του προτύπου WebAssembly 33

3.4 Εντολές σωρού 39

3.5 Αριθμητικές εντολές 39

3.6 Τοπικές και καθολικές μεταβλητές 42

3.7 Συναρτήσεις 45

3.8 Διαχείριση μνήμης στην WebAssembly 46

3.9 Εντολές ελέγχου 49

3.10 Μορφή κειμένου (WAT) 52

4 Ανάπτυξη κώδικα WebAssembly και συγκριτική μελέτη 56

4.1 Δημιουργία και εκτέλεση κώδικα WebAssembly 56

4.2 Οργάνωση συγκριτικής μελέτης 60

4.2.1 Μέτρηση του χρόνου φόρτωσης μίας σελίδας 61

4.3 Σύγκριση επιδόσεων WebAssembly και Javascript 63
4.4 Παρουσίαση αποτελεσμάτων 66
5 Επίλογος 70
5.1 Σύνοψη και συμπεράσματα 70
6 Βιβλιογραφία 72

Κατάλογος Εικόνων

Εικόνα 2-1: Κατάταξη γλωσσών προγραμματισμού με βάση το δείκτη RedMonk........ 15
Εικόνα 2-2: Κατάταξη γλωσσών προγραμματισμού με βάση το δείκτη PYPL............. 16
Εικόνα 2-3: Εκδόσεις της JavaScript.. 18
Εικόνα 2-4: Στάδια μεταγλώττισης LLVM 26
Εικόνα 3-1: Μεταγλώττιση κώδικα WebAssembly 34
Εικόνα 3-2: Ανεξαρτησία WebAssembly από πλατφόρμα εκτέλεσης........................... 34
Εικόνα 3-3: Τρόπος εκτέλεσης κώδικα Wasm 36
Εικόνα 3-4: Εντολές της γλώσσας WebAssembly 38
Εικόνα 3-5: Καθολικές μεταβλητές στην WebAssembly.. 43
Εικόνα 3-6: Τοπικές μεταβλητές στην WebAssembly.. 43
Εικόνα 3-7: Παράδειγμα κλήσης συνάρτησης 46
Εικόνα 3-8: Ορισμός περιοχής μνήμης στην WebAssembly 47
Εικόνα 3-9: Η εντολή block.. 49
Εικόνα 3-10: Η εντολή if 50
Εικόνα 3-11: Αλλαγή ροής με την εντολή br 51
Εικόνα 3-12: Δομή επανάληψης με την εντολή br_if 52
Εικόνα 3-13: Το πεδίο type της μορφής WAT.. 53
Εικόνα 3-14: Το πεδίο import ης μορφής WAT.. 54
Εικόνα 3-15: Το πεδίο export της μορφής WAT... 54
Εικόνα 3-16: Το πεδίο memory στη μορφή WAT.. 55
Εικόνα 3-17: Το πεδίο table στη μορφή WAT 55
Εικόνα 4-1: Δημιουργία προγράμματος HelloWorld με το Emscripten 58
Εικόνα 4-2: Το περιβάλλον του WebAssembly Studio.. 59
Εικόνα 4-3: Μετατροπή Wasm σε WAT μορφή 59
Εικόνα 4-4: Μέτρηση χρόνου φόρτωσης μίας ιστοσελίδας με τη Javascript................. 61
Εικόνα 4-5: Φάσεις για τη φόρτωση Διαδικτυακού περιεχομένου 62
Εικόνα 4-6: Κώδικας C για την ακολουθία Fibonacci 63
Εικόνα 4-7: Μορφή WAT του κώδικα Fibonacci 64
Εικόνα 4-8: Κώδικας Javascript για τη φόρτωση του Wasm module 65
Εικόνα 4-9: Κώδικας HTML για την ιστοσελίδα.. 65
Εικόνα 4-10: Κώδικας Javascript για την ακολουθία Fibonacci 66

Εικόνα 4-11: Καταγραφή χρόνου φόρτωσης για τη σελίδα με κώδικα Wasm 67
Εικόνα 4-12: Καταγραφή χρόνου φόρτωσης για τη σελίδα με κώδικα WebAssembly .. 68

Κατάλογος Πινάκων

Πίνακας 1: Τύποι εντολών της WebAssembly.. 37

Πίνακας 2: Εντολές διαχείρισης της μνήμης.. 47

Πίνακας 3: Εμφάνιση συγκριτικών αποτελεσμάτων... 69

1 Εισαγωγή

1.1 Περιγραφή και σημαντικότητα του θέματος

Ο Παγκόσμιος Ιστός και οι τεχνολογίες πάνω στις οποίες βασίζεται, μεταβάλλονται συνεχώς και ταυτόχρονα, οι απαιτήσεις ασφάλειας, αξιοπιστίας και ταχύτητας αυξάνονται με πολύ γρήγορο ρυθμό. Η γλώσσα JavaScript είναι μία γλώσσα προγραμματισμού, η οποία αποτελεί μία πολύ δημοφιλή επιλογή κατά τον προγραμματισμό των ιστοσελίδων. Ενώ αρχικά ξεκίνησε σαν μία γλώσσα για τον έλεγχο της λειτουργικότητας των φορμών εισαγωγής δεδομένων από τον χρήστη και τη διαχείριση του μοντέλου DOM της γλώσσας HTML, έχει καταλήξει πλέον να αποτελεί μία από τις περισσότερο απαιτητικές και διαδεδομένες γλώσσες προγραμματισμού, η οποία χρησιμοποιείται για πολλούς διαφορετικούς σκοπούς.

Η χρήση της γλώσσας JavaScript, με άλλα λόγια, δεν περιορίζεται πλέον μόνο στον προγραμματισμό της λειτουργικότητας στη μεριά του πελάτη (client-side scripting), αλλά χρησιμοποιείται και σε άλλα πεδία και πολύ διαφορετικούς σκοπούς. Για παράδειγμα, με τη βοήθεια της τεχνολογίας *node.js*, είναι δυνατόν να χρησιμοποιηθεί η JavaScript για τον προγραμματισμό στη μεριά του εξυπηρετητή (server-side scripting). Επιπλέον, με την χρήση του προγραμματιστικού πλαισίου (framework) *Electron*, μπορεί να χρησιμοποιηθεί για τη δημιουργία εφαρμογών οι οποίες μπορούν να εκτελεστούν αυτόνομα από το λειτουργικό σύστημα, ανεξάρτητα από την τεχνολογία του Παγκόσμιου Ιστού. Η γλώσσα JavaScript χρησιμοποιείται ακόμη και για την ανάπτυξη εφαρμογών σε "έξυπνες" συσκευές (smart devices), όπως tablets και κινητά, ανεξάρτητα από την πλατφόρμα στην οποία βασίζονται (Android, iOS, κτλ).

Το αποτέλεσμα όλης αυτής της εξέλιξης, ήταν η γλώσσα JavaScript να αποκτήσει ρόλους για τους οποίους δεν είχε σχεδιαστεί από το ξεκίνημά της. Το γεγονός αυτό προκαλεί συχνά ως αποτέλεσμα την εμφάνιση κάποιων προβλημάτων, τα οποία κυρίως εστιάζονται στην ταχύτητα εκτέλεσης των προγραμμάτων που έχουν γραφτεί σε γλώσσα JavaScript, κυρίως όταν αναλαμβάνουν την επεξεργασία μεγάλου όγκου δεδομένων, όπως στην

περίπτωση επεξεργασίας εικόνας και βίντεο ή όταν πρόκειται για εφαρμογές στο χώρο των παιχνιδιών (gaming).

Κατά το παρελθόν, έχουν γίνει διάφορες προσπάθειες για την βελτίωση της απόδοσης των προγραμμάτων που έχουν βασιστεί στη JavaScript. Η χρήση μεταγλωττιστών just-in-time (JIT) και οι τεχνικές βελτιστοποίησης που χρησιμοποιούνται από το προγράμματα πλοήγησης, έχουν βελτιώσει την ταχύτητα εκτέλεσης των προγραμμάτων μέχρι ένα σημείο. Στη συνέχεια, τεχνολογίες όπως η *Google Native Client* και η *asm.js* έχουν επίσης δώσει κάποια καλά αποτελέσματα και η κάθε μία από αυτές έχει τα πλεονεκτήματα και τα μειονεκτήματά της. Οι προσπάθειες αυτές, για την βελτίωση της γλώσσας και την κάλυψη των αδυναμιών της που σχετίζονται με την επίδοση των εφαρμογών, κατά τα τελευταία χρόνια, εστιάζονται στην τεχνολογία WebAssembly που αποτελεί το θέμα της παρούσας εργασίας.

Το πρότυπο *WebAssembly* αποτελεί την νεότερη τεχνολογία η οποία υπόσχεται επιδόσεις, οι οποίες αγγίζουν τις επιδόσεις προγραμμάτων που έχουν γραφτεί σε άλλες “γρηγορότερες” γλώσσες προγραμματισμού και έχουν μεταγλωττιστεί ειδικά για τις συγκεκριμένες πλατφόρμες (native applications). Η προσπάθεια είναι ιδιαίτερα ελπιδοφόρα γιατί αποτελεί ανοικτό πρότυπο και επιπλέον, υποστηρίζεται και αναπτύσσεται συνεργατικά από όλες τις πρωτοπόρες εταιρίες που ανταγωνίζονται στο χώρο. Ως αποτέλεσμα αυτής της σύμπραξης, το πρότυπο υποστηρίζεται από το σύνολο σχεδόν των προγραμμάτων πλοήγησης που χρησιμοποιείται από η μεγάλη πλειοψηφία των τελικών χρηστών. Η συγκεκριμένη τεχνολογία είναι φυσικά αρκετά πρόσφατη και υπάρχουν ακόμη πολλά ανοικτά θέματα, αποτελεί όμως από την άλλη μεριά μία ιδιαίτερα υποσχόμενη τεχνολογία. Το γεγονός αυτό αποτέλεσε και το βασικό κίνητρο για την συγγραφή της παρούσας εργασίας.

1.2 Σκοπός και Στόχοι

Ο σκοπός της παρούσας διπλωματικής εργασίας είναι η διερεύνηση του προτύπου WebAssembly, των πλεονεκτημάτων τα οποία προσφέρει και των προοπτικών που δημιουργούνται με την χρήση της. Αναλυτικότερα, οι στόχοι της εργασίας είναι οι πιο κάτω:

- Παρουσίαση του προτύπου WebAsselmbly και του τρόπου με τον οποίο λειτουργεί, σε συνδυασμό με την χρήσης της γλώσσας JavaScript.
- Αναλυτική περιγραφή της διαδικασίας ανάπτυξης προγραμμάτων που κάνουν χρήση του προτύπου WebAssembly.
- Σύγκριση του προτύπου WebAssembly με άλλες τεχνολογίες που αναπτύχθηκαν για την βελτίωση της επίδοσης της Javascript.
- Μελέτη των επιδόσεων κάποιων προγραμμάτων που θα αναπτυχθούν με το πρότυπο WebAssembly για το σκοπό αυτό, σε σχέση με αντίστοιχα προγράμματα που έχουν δημιουργηθεί με απλή Javascript.
- Διερεύνηση των χρήσεων που μπορεί να έχει η νέα αυτή τεχνολογία και η επίδρασή της στη λειτουργία του Παγκόσμιου Ιστού.

Στις παραγράφους που ακολουθούν, παρουσιάζεται η υλοποίηση των στόχων οι οποίοι τέθηκαν πιο πάνω.

1.3 Διάρθρωση της εργασίας

Στο κεφάλαιο 2 γίνεται μία εισαγωγή και παρουσιάζεται το θεωρητικό υπόβαθρο για τη συνέχεια της εργασίας. Παρουσιάζεται μία επισκόπηση της έρευνας που έχει γίνει στο πεδίο της εξέλιξης της γλώσσας JavaScript σε σχέση με τη βελτίωση των επιδόσεων των εφαρμογών που γράφονται με τη γλώσσα αυτή.

Στο κεφάλαιο 3 γίνεται μία λεπτομερής περιγραφή του προτύπου WebAssembly και του τρόπου με τον οποίο μπορεί να γίνει ανάπτυξη εφαρμογών μέσω της τεχνολογίας αυτής. Παρουσιάζονται επίσης τα πλεονεκτήματα του προτύπου, σε σχέση με άλλες εναλλακτικές τεχνολογίες που αναπτύχθηκαν τα προηγούμενα χρόνια.

Στο κεφάλαιο 4 πραγματοποιείται μία συγκριτική μελέτη κάποιων εφαρμογών που αναπτύσσονται με τη βοήθεια της τεχνολογίας WebAssembly, σε σχέση με παρόμοιες εφαρμογές οι οποίες δεν κάνουν χρήση της τεχνολογίας αυτής. Η σύγκριση αφορά τον χρόνο που απαιτούν οι εφαρμογές αυτές για να ολοκληρώσουν τη λειτουργία τους.

Το κεφάλαιο 5 αποτελεί τον επίλογο της παρούσας εργασίας και παρουσιάζονται τα συμπεράσματα τα οποία προκύπτουν, ως αποτέλεσμα της διερεύνησης του προτύπου WebAssembly και της σύγκρισης των επιδόσεων που προσφέρει, σε σχέση με άλλες τεχνολογίες. Επιπλέον, περιγράφονται κάποιες ιδέες για μελλοντικές χρήσεις του προτύπου, τόσο σε σχέση με τον Παγκόσμιο Ιστό, όσο και πέρα από αυτόν.

2 Θεωρητικό υπόβαθρο και σχετική έρευνα

2.1 Η γλώσσα JavaScript

Η JavaScript είναι μία γλώσσα προγραμματισμού για ηλεκτρονικούς υπολογιστές η οποία κάνει χρήση *διερμηνευτή* (interpreter) και χρησιμοποιείται ευρέως για τη δημιουργία εφαρμογών για τον Παγκόσμιο Ιστό. Η χρήση διερμηνευτή συνεπάγεται ότι δεν έχουμε συνολική μεταγλώττιση του κώδικα πριν την εκτέλεσή του, αλλά οι εντολές αναλύονται, μία προς μία, τη στιγμή της εκτέλεσης του προγράμματος. Συνήθως, προγράμματα τέτοιας μορφής, αναφέρονται και ως *σενάρια* (scripts). Επειδή η JavaScript χρησιμοποιείται συχνά για την ανάπτυξη της διεπαφής μίας εφαρμογής με τον τελικό χρήστη (user interface), πολλές φορές λέμε ότι η γλώσσα χρησιμοποιείται για την ανάπτυξη σεναρίων από την πλευρά του πελάτη (*client-side scripting*). Στην πραγματικότητα, αυτό αλλάζει όλο και περισσότερο, αφού η JavaScript χρησιμοποιείται πλέον και για πολλούς άλλους σκοπούς, ακόμη και για εφαρμογές που δεν συνδέονται άμεσα με τον Παγκόσμιο Ιστό, όπως θα δούμε πιο κάτω.

Η γλώσσα JavaScript αναπτύχθηκε τη δεκαετία του 1990 από τον Brendan Eich της Netscape, αρχικά με την επωνυμία *Mocha*, ενώ αργότερα μετονομάστηκε σε *LiveScript* και τελικά, το 1995, πήρε το τελικό της όνομα [Krill 2008]. Κάποιες φορές συγχέεται με την γλώσσα προγραμματισμού *Java*, όμως η μόνη ομοιότητα που έχουν είναι στην ονομασία τους και στο γεγονός ότι η μορφή των βασικών εντολών προέρχεται από την γλώσσα C. Ο κώδικας ενός σεναρίου σε γλώσσα JavaScript μπορεί να βρίσκεται στο ίδιο αρχείο με τον κώδικα HTML, αλλά μπορεί να βρίσκεται και σε ξεχωριστό αρχείο, συνήθως με επέκταση .js, το οποίο μπορεί να εισαχθεί σε μια ιστοσελίδα με την κατάλληλη δήλωση.

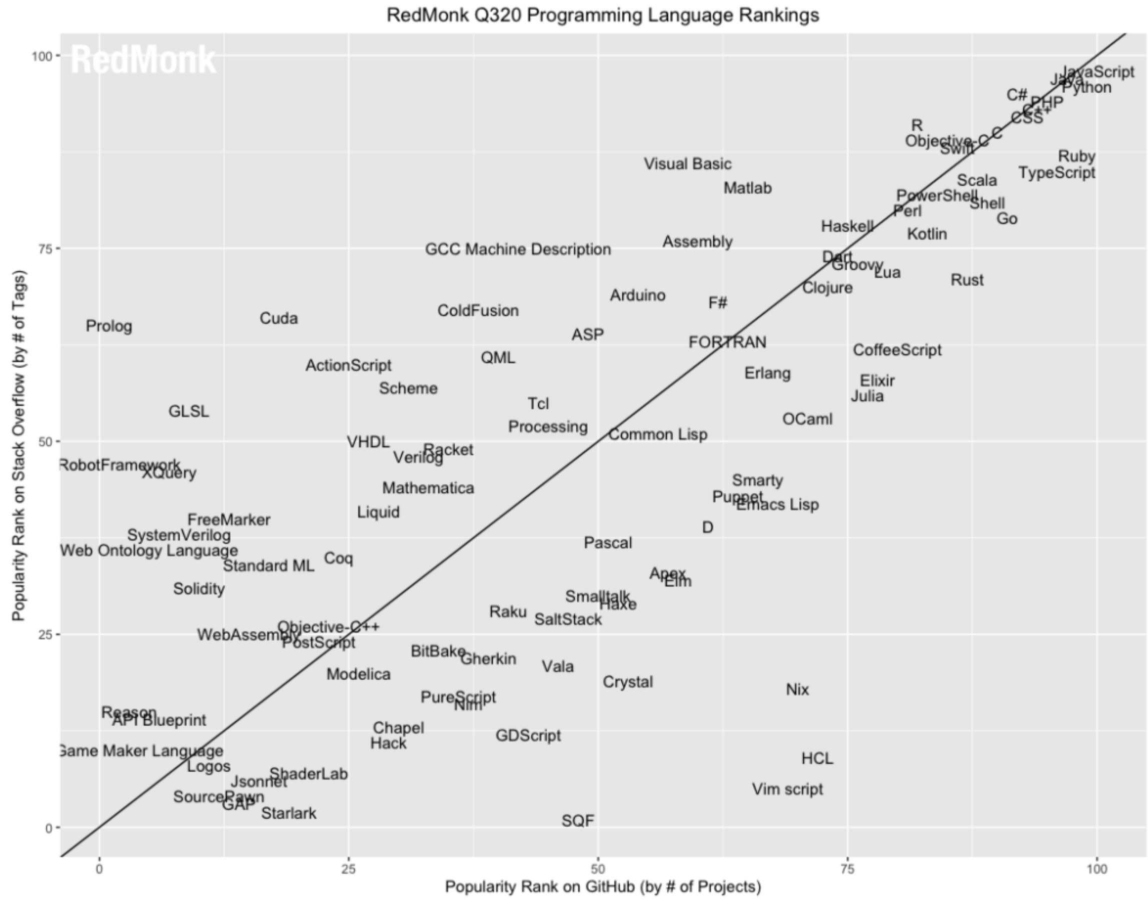
Η Microsoft δημιούργησε τη δική της έκδοση της γλώσσας, την οποία ονόμασε *JScript,* για λόγους εμπορικών σημάτων. Το Νοέμβριο του 1996, η Netscape έκανε υποβολή της γλώσσας για την κατοχύρωσή της ως βιομηχανικό πρότυπο στο Ecma International. Ως

αποτέλεσμα της κατοχύρωσης αυτής, έχουμε την τυποποιημένη μορφή της γλώσσας με την ονομασία *ECMAScript* [ECMAScript 2007].

Η JavaScript είναι μια δυναμική γλώσσα σεναρίων, η οποία κάνει χρήση ασθενών τύπων και η σύνταξή της είναι επηρεασμένη από τη γλώσσα προγραμματισμού C. Μπορεί επίσης να χρησιμοποιηθεί για διαφορετικά προγραμματιστικά παραδείγματα (programming paradigms), δεδομένου ότι υποστηρίζει τόσο διαδικαστικό προγραμματισμό, όσο αντικειμενοστρεφή ή ακόμη και συναρτησιακό προγραμματισμό. Επιπρόσθετα, με τη χρήση ειδικών βιβλιοθηκών, όπως το *Node.js*, μπορεί να χρησιμοποιηθεί ακόμη και για την ανάπτυξη σεναρίων στην πλευρά του εξυπηρετητή (*server-side scripting*). Πέρα από τις χρήσεις όμως που σχετίζονται με τον Παγκόσμιο Ιστό, η JavaScript μπορεί να χρησιμοποιηθεί και για την ανάπτυξη ανεξάρτητων εφαρμογών, με τη βοήθεια βιβλιοθηκών όπως του *Electron framework*. Χρησιμοποιείται επίσης και στον χώρο ανάπτυξης εφαρμογών για "έξυπνες" συσκευές, οι οποίες μπορούν να εκτελεστούν τόσο σε περιβάλλον Android, όσο και σε περιβάλλον iOS.

Η γλώσσα JavaScript είναι μία από τις πιο δημοφιλείς γλώσσες προγραμματισμού. Στην κατάταξη κατά RedMonk, η JavaScript βρίσκεται στην πρώτη θέση ανάμεσα σε όλες τις γλώσσες προγραμματισμού για οποιοδήποτε σκοπό, όπως φαίνεται στην Εικόνα 2-1 [RedMonk 2020]. Η συγκεκριμένη κατάταξη βασίζεται σε δεδομένα που λαμβάνονται από το GitHub και το Stack Overflow. Εξετάζεται δηλαδή ο κώδικας των εφαρμογών που υπάρχουν στο GitHub, ώστε να γίνει μία στατιστική ανάλυση των γλωσσών προγραμματισμού που χρησιμοποιούνται για την ανάπτυξη των εφαρμογών. Επιπλέον, αναλύονται τα θέματα των συζητήσεων που γίνονται ανάμεσα σε προγραμματιστές στη δημοφιλή πλατφόρμα του Stack Overflow, αναλύοντας τη γλώσσα προγραμματισμού που αφορούν οι συζητήσεις αυτές. Η JavaScript καταλαμβάνει την κορυφαία θέση στην κατάταξη που προκύπτει από την ανάλυση αυτή, δείχνοντας έτσι την ευρεία χρήστη της στο χώρο του προγραμματισμού.

Με βάση τον δείκτη PYPL (PopularitY of Programming Language Index), ο οποίος βασίζεται στη συχνότητα των αναζητήσεων που γίνονται στο Διαδίκτυο για την εύρεση οδηγών και υλικού εκμάθησης γλωσσών προγραμματισμού, η JavaScript καταλαμβάνει, για τον Οκτώβρη του 2020, την 3η θέση ανάμεσα στις γλώσσες προγραμματισμού, μετά την Python και τη Java, με αυξητικές μάλιστα τάσεις, όπως φαίνεται στην Εικόνα 2-2 [PYPL 2020].



Εικόνα 2-1*: Κατάταξη γλωσσών προγραμματισμού με βάση το δείκτη RedMonk*

Τα πιο πάνω στοιχεία, αναδεικνύουν τη δημοφιλία της γλώσσας JavaScript και επομένως, τη μεγάλη σημασία που έχουν οι προσπάθειες για την κάλυψη των κενών και μειονεκτημάτων που παρουσιάζει η γλώσσα. Όπως αναλύεται πιο κάτω, η JavaScript καλείται να διαδραματίσει ρόλους για τους οποίους δεν σχεδιάστηκε εξ' αρχής. Στο πλαίσιο της βελτίωσης των αδύναμων σημείων της γλώσσας, εντάσσεται και η ανάπτυξη του προτύπου WebAssembly, όπως θα παρουσιαστεί αναλυτικότερα στις επόμενες παραγράφους.

Worldwide, Oct 2020 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	31.02 %	+2.2 %
2		Java	16.38 %	-2.8 %
3		JavaScript	8.41 %	+0.4 %
4		C#	6.52 %	-0.6 %
5		PHP	5.83 %	-0.4 %
6		C/C++	5.56 %	-0.4 %

Εικόνα 2-2*: Κατάταξη γλωσσών προγραμματισμού με βάση το δείκτη PYPL*

2.1.1 Χαρακτηριστικά της γλώσσας JavaScript

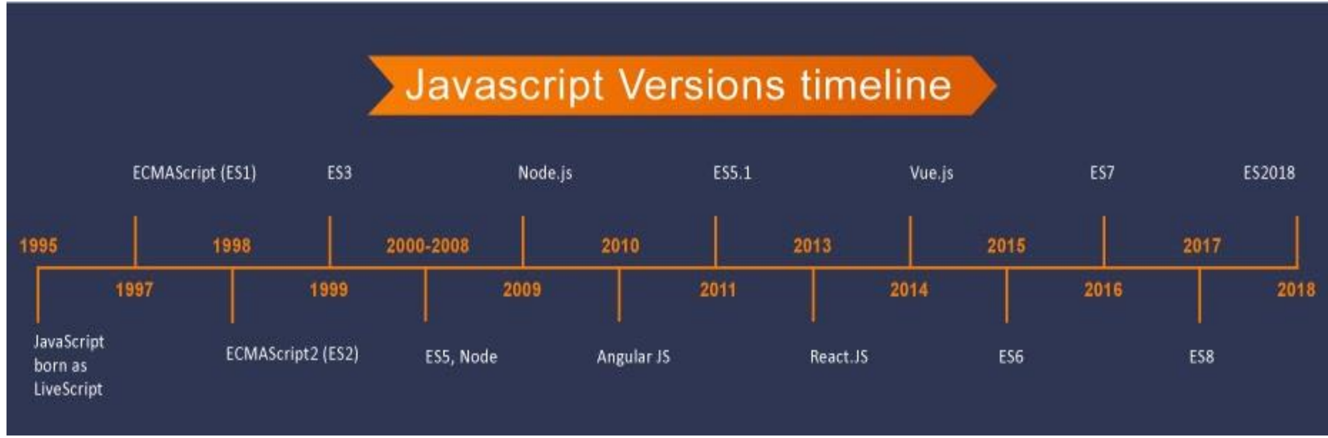
Τα βασικά χαρακτηριστικά της γλώσσας JavaScript εμφανίζονται πιο κάτω:

- Είναι δομημένη γλώσσα. Αποτελεί χαρακτηριστικό που κληρονόμησε από τη C, όπως και τη μορφή των βασικών εντολών της. Κάνει χρήση αγκυλών (curly brackets) για τη δόμηση του κώδικα.
- Είναι γλώσσα υψηλού επιπέδου. Με τη χρήση βιβλιοθηκών και συναρτήσεων, είναι ευκολότερο για τον προγραμματιστή να γράψει ένα πρόγραμμα JavaScript, σε σχέση με άλλες γλώσσες όπως η C, η Assembly κτλ.
- Είναι διερμηνευόμενη γλώσσα. Η JavaScript κάνει χρήση διερμηνευτή (interpreter). Δεν γίνεται επομένως ολοκληρωμένη μεταγλώττιση του προγράμματος, ώστε να παραχθεί η εκτελέσιμη μορφή του. Οι εντολές μεταφράζονται, αναλύονται και εκτελούνται μία προς μία, τη στιγμή της εκτέλεσης του προγράμματος.

- Είναι δυναμική γλώσσα. Οι μεταβλητές και οι συναρτήσεις μπορούν να αλλάξουν ή να δημιουργηθούν νέες οποιαδήποτε στιγμή, κατά την εκτέλεση του προγράμματος.
- Είναι γλώσσα ασθενών τύπων (weakly ή loosely typed). Κατά την εκτέλεση του προγράμματος, δεν γίνεται αυστηρός έλεγχος του τύπου που έχει αποδοθεί σε μία μεταβλητή. Το χαρακτηριστικό αυτό δίνει μία ευελιξία στη γλώσσα, αλλά κάποιες φορές μπορεί να έχει απρόβλεπτα αποτελέσματα.
- Υποστηρίζει αντικειμενοστρεφή προγραμματισμό. Αν και συχνά χρησιμοποιείται για ανάπτυξη εφαρμογών που ακολουθούν το διαδικαστικό μοντέλο, μπορεί να υποστηρίξει τη δημιουργία κλάσεων και αντικειμένων.
- Υποστηρίζει συναρτησιακό προγραμματισμό. Έχουν προστεθεί χαρακτηριστικά στη γλώσσα, έτσι ώστε να είναι δυνατή η ανάπτυξη εφαρμογών που ακολουθούν το συναρτησιακό μοντέλο.
- Είναι γλώσσα ανεξάρτητη από την πλατφόρμα (platform independent). Μπορεί να χρησιμοποιηθεί για τη δημιουργία σεναρίων που εκτελούνται από οποιαδήποτε πλατφόρμα.
- Υποστηρίζει εμφωλευμένες και ανώνυμες συναρτήσεις.
- Κάνει χρήση συναρτήσεων πρώτης-κλάσης (first-class functions). Μπορούμε να έχουμε συναρτήσεις που έχουν σαν όρισμα άλλες συναρτήσεις ή που επιστρέφουν μία συνάρτηση ως αποτέλεσμα.

Πέρα από τα βασικά χαρακτηριστικά της γλώσσας, υπάρχουν νέα χαρακτηριστικά που προστίθενται συνεχώς. Τον Ιούνιο του 2020 βγήκε η έκδοση EcmaScript 2020 (ES2020) ή αλλιώς έκδοση 11 της γλώσσας, η οποία εκτός από την προσθήκη νέων συναρτήσεων, δίνει στη γλώσσα τη δυνατότητα χρήσης πολύ μεγάλων ακεραίων (BigInt). Η JavaScript είναι μία ιδιαίτερα δημοφιλής γλώσσα και είναι λογικό επομένως να προστίθενται συνεχώς νέα χαρακτηριστικά, τα οποία συνήθως υιοθετούνται σε μικρό χρονικό διάστημα από τα σύγχρονα προγράμματα πλοήγησης. Οι εκδόσεις της γλώσσας μέχρι το 2018, εμφανίζονται στην *Εικόνα 2-3* [Navhoax 2018]. Στην εικόνα φαίνονται και οι στιγμές κατά τις οποίες εμφανίστηκαν κάποια διαδεδομένα frameworks της γλώσσας, όπως το Angular, το React και το Vue, καθώς και η έκδοση της τεχνολογίας node.js. Με την έκδοση *ES.next*

αναφερόμαστε στα επιθυμητά εκείνα χαρακτηριστικά της γλώσσας για τα οποία υπάρχει πρόθεση να εισαχθούν μελλοντικά στη γλώσσα.



Εικόνα 2-3: Εκδόσεις της JavaScript

2.1.2 Χρήσεις της JavaScript

Η Javascript είναι μία υψηλού επιπέδου γλώσσα προγραμματισμού που μπορεί να χρησιμοποιηθεί για προγράμματα που ακολουθούν είτε το μοντέλο του διαδικαστικού προγραμματισμού, είτε του αντικειμενοστρεφή ή συναρτησιακού προγραμματισμού. Μαζί με την HTML και τη CSS αποτελεί τον πυρήνα των γλωσσών που χρησιμοποιούνται για την δημιουργία ιστοσελίδων του Παγκόσμιου Ιστού.

Με τη βοήθεια της Javascript μπορούν να δημιουργηθούν διαδραστικές (interactive) ιστοσελίδες και η μία πολύ βασική χρήση της γλώσσας ήταν, και εξακολουθεί να είναι, η ανάπτυξη client-side εφαρμογών. Υπάρχει μάλιστα πληθώρα από δημοφιλείς βιβλιοθήκες και frameworks τα οποία έχουν σκοπό την ευκολότερη και αποδοτικότερη χρήση της γλώσσας για την ανάπτυξη GUI (Graphical User Interfaces) για την επικοινωνία με τον τελικό χρήστη του Παγκόσμιου Ιστού. Η βιβλιοθήκη *jQuery*, ξεκίνησε το 2006 και σχεδιάστηκε για να απλοποιήσει την ανάπτυξη σεναρίων με τη γλώσσα Javascript. Στην πορεία αναπτύχθηκαν ολοκληρωμένα frameworks τα οποία επιτρέπουν την περισσότερο τυποποιημένη και εύκολη ανάπτυξη διεπαφών μέσω της γλώσσας Javascript, όπως για

παράδειγμα τα *React*, *Angular*, *Vue*, *Ember*, *Backbone* κλπ. Αναπτύχθηκαν επίσης τεχνικές, όπως η *Ajax* (Asynchronous JavaScript and XML), οι οποίες επιτρέπουν τη λήψη δεδομένων τα οποία μεταβάλλονται από έναν εξυπηρετητή (server), χωρίς να είναι απαραίτητο να γίνει από την αρχή η λήψη της σελίδας αυτής από τον server.

Η γλώσσα JavaScript όμως, μπορεί να χρησιμοποιηθεί και για τον προγραμματισμό στη μεριά του εξυπηρετητή (server-side scripting), παρόλο που δεν προορίζονταν αρχικά για το σκοπό αυτό. Αυτό μπορεί να γίνει με τη βοήθεια της τεχνολογίας *node.js*, η οποία επιτρέπει την ανάπτυξη εργαλείων, με τη βοήθεια της JavaScript, που μπορούν να εκτελεστούν από την γραμμή εντολών (command-line tools) και την ανάπτυξη server-side σεναρίων για τη δημιουργία δυναμικών ιστοσελίδων [Node.js 2020]. Το node.js αναπτύχθηκε αρχικά από τον Ryan Dahl το 2009, περίπου 13 χρόνια μετά το LiveWire Pro Web της Netscape που προσπάθησε αρχικά τη χρήση της JavaScript ως γλώσσας για server-side scripting. Αναπτύχθηκε μάλιστα, το 2010, ειδικός διαχειριστής πακέτων για το περιβάλλον node.js, με το όνομα *npm*. Από το Σεπτέμβρη του 2016 ακολουθεί το πρότυπο EcmaScript 6 (ES6) και η διαχείριση της τεχνολογίας γίνεται πλέον από τον οργανισμό OpenJS Foundation, που προήλθε το 2019 από την ένωση του JS Foundation και του Node.js Foundation.

Η JavaScript μπορεί να χρησιμοποιηθεί και για την ανάπτυξη αυτόνομων εφαρμογών, που μπορούν να εκτελεστούν σε οποιοδήποτε λειτουργικό σύστημα, παρέχοντας την απαραίτητη γραφική διασύνδεση με τον χρήστη (GUI), χωρίς την ανάγκη χρησιμοποίησης κάποιου προγράμματος πλοήγησης (Web browser). Το *Electron* είναι ένα πλαίσιο ανάπτυξης λογισμικού (framework), ανοικτού κώδικα που βρίσκεται στο GitHub, με τη βοήθεια του οποίου μπορούν να αναπτυχθούν τέτοιου είδους εφαρμογές. Η ανάπτυξη του Electron ξεκίνησε το 2013 ως *Atom Shell* και κάνει χρήση του node.js και του Chromium rendering engine [Betts 2017]. Με τη βοήθεια του Electron, έχουν αναπτυχθεί εφαρμογές όπως οι Skype, Visual Studio Code, Discord, WhatsApp, Atom κλπ. Από τον Οκτώβριο του 2017, αναπτύχθηκε και το *Electron.NET* για τη χρήση και της .NET τεχνολογίας. Οι εφαρμογές Electron έχουν δεχθεί κάποια κριτική σε σχέση με την ταχύτητα εκτέλεσής τους, σε σχέση με τη σύγκρισή τους με άλλες παρόμοιες εφαρμογές που έχουν δημιουργηθεί με άλλες γλώσσες [Beyer 2019].

Η JavaScript μπορεί να χρησιμοποιηθεί και για την ανάπτυξη εφαρμογών για “έξυπνες” συσκευές με λειτουργικό σύστημα Android, iOS κτλ. Το *Apache Cordova* είναι ένα

πλαίσιο ανάπτυξης λογισμικού (framework) ανοικτού κώδικα, που ξεκίνησε το 2009 και παλαιότερα αναφέρονταν ως *PhoneGap*. Στην πραγματικότητα, η εμπορική έκδοση της Adobe, συνεχίζει να αποκαλείται PhoneGap. Με τη βοήθεια του Apache Cordova, μπορούν να δημιουργηθούν υβριδικές εφαρμογές, οι οποίες κάνουν χρήση γλωσσών όπως HTML, CSS και JavaScript, για την ανάπτυξη εφαρμογών που μπορούν να λειτουργήσουν σε κινητές συσκευές, όποια πλατφόρμα και αν έχουν αυτές. Στην πραγματικότητα, έχουν αναπτυχθεί διάφορες βιβλιοθήκες και frameworks που κάνουν χρήση του Apache Cordova και προσφέρουν εργαλεία και επιπλέον λειτουργίες για την ανάπτυξη εφαρμογές για "έξυπνες" συσκευές, όπως οι Ionic, Monaca, VoltBuilder, TACO, Onsen UI κλπ.

Από την πιο πάνω ανάλυση, γίνεται σαφές ότι η JavaScript έχει αρχίσει να χρησιμοποιείται πλέον και σε πεδία για τα οποία δεν είχε σχεδιαστεί εξ' αρχής. Αυτό έχει σαν αποτέλεσμα να εμφανίζει κάποιες αδυναμίες, οι οποίες κυρίως αφορούν τις επιδόσεις των εφαρμογών που δημιουργούνται με τη γλώσσα. Για το λόγο αυτό, έχουν αναπτυχθεί τεχνολογίες οι οποίες προσπαθούν να αντιμετωπίσουν τις αδυναμίες αυτές, όπως αναλύεται στη συνέχεια.

2.2 Αδυναμίες της γλώσσας JavaScript και αντιμετώπισή τους

Η JavaScript, όπως και όλες οι άλλες γλώσσες προγραμματισμού, έχει τα δυνατά της σημεία, αλλά και τις αδυναμίες της. Ειδικά από τη στιγμή που έχει αρχίσει να χρησιμοποιείται για σκοπούς για τους οποίους δεν έχει σχεδιαστεί από την αρχή. Λογικό είναι να εμφανίζει κάποια αδύνατα σημεία. Η βασικότερη αδυναμία της, όταν χρησιμοποιείται για επεξεργασία μεγάλου όγκου δεδομένων ή όταν χρησιμοποιείται έξω από τα όρια του Παγκόσμιου Ιστού, είναι η ταχύτητα εκτέλεσης των εφαρμογών. Αυτό είναι κάτι το αναμενόμενο, από τη στιγμή που όπως αναλύθηκε πιο πάνω, η JavaScript κάνει χρήση διερμηνευτή (interpreter) και όχι μεταγλωττιστή (compiler) όπως συμβαίνει με άλλες γλώσσες προγραμματισμού. Δεν υπάρχει δηλαδή ο μεταγλωττισμένος κώδικας, ο οποίος έχει δημιουργηθεί για κάποια συγκεκριμένη πλατφόρμα, ο οποίος μπορεί να εκτελεστεί ταχύτατα όταν το αποφασίσει ο χρήστης. Θα πρέπει η κάθε εντολή να αναλύεται και μεταφράζεται ξεχωριστά, κάθε φορά που εκτελείται το πρόγραμμα, κάτι που δημιουργεί προβλήματα επιδόσεων, ειδικά όταν υπάρχουν βρόχοι με εντολές που

εκτελούνται επαναλαμβανόμενα. Εξαιτίας αυτών των εν γένει αδυναμιών της Javascript, έχουν γίνει πολλές προσπάθειες για την αντιμετώπισή τους.

Οι πρώτες προσπάθειες για τη δημιουργία εφαρμογών οι οποίες φιλοξενούνται μέσα σε ιστοσελίδες, έγιναν στη γλώσσα προγραμματισμού Java. Τα *Java applets* είχαν μεγάλη διάδοση, την εποχή που εμφανίστηκε η JavaScript, μετά από σχεδιασμό 10 μόλις ημερών [Severance 2012]. Η Microsoft το 1996, έβγαλε την τεχνολογία *ActiveX*, που έδινε παρόμοιες δυνατότητες στους σχεδιαστές ιστοσελίδων. Όμως, μετά από μία δεκαετία, οι εφαρμογές JavaScript άρχισαν να υποστηρίζονται σε πολύ μεγάλο βαθμό από το σύνολο των προγραμμάτων πλοήγησης και άρχισαν να εκτοπίζουν την χρήση των Java applets και των ActiveX στοιχείων [Evans 2015].

2.2.1 Στιγμιαία μεταγλώττιση (Just-In-Time compilation)

Όπως έχει ήδη αναφερθεί, η JavaScript κάνει χρήση διερμηνευτή (interpreter), κάτι που παρουσιάζει τόσο πλεονεκτήματα, όσο και μειονεκτήματα. Όταν χρησιμοποιείται διερμηνευτής για μία γλώσσα προγραμματισμού, η εκτέλεση των εντολών του προγράμματος είναι πιο άμεση. Δεν είναι υποχρεωτικό να γίνει πρώτα η μεταγλώττιση του προγράμματος και να αποθηκευτεί η μεταγλωττισμένη μορφή του σε ξεχωριστό αρχείο. Η ανάλυση και εκτέλεση των εντολών γίνεται μία προς μία. Μία εντολή μεταφράζεται και εκτελείται, αφού προηγουμένως έχουν μεταφραστεί και εκτελεστεί όλες οι προηγούμενες. Εάν κατά την μετάφραση ή εκτέλεση μίας εντολής εμφανιστεί σφάλμα, τότε το πρόγραμμα σταματάει, αφού όμως έχουν ήδη εκτελεστεί όλες οι προηγούμενες. Επομένως, όταν χρησιμοποιείται διερμηνευτής, παραλείπεται η φάση της ολοκληρωτικής μετάφρασης του προγράμματος, δίνοντας μία αμεσότητα στην εκτέλεση των εντολών.

Στις γλώσσες που χρησιμοποιούν μεταγλωττιστή (compiler) όμως, έχουμε την εξ' ολοκλήρου μετάφραση του προγράμματος, πριν την εκτέλεσή του. Εάν ολόκληρο το πρόγραμμα μεταφραστεί χωρίς να υπάρχουν λάθη, τότε μπορεί να εκτελεστεί μία ή περισσότερες φορές. Δεν χρειάζεται δηλαδή η εκ νέου μετάφρασή του, κάθε φορά που εκτελείται το πρόγραμμα. Το χαρακτηριστικό αυτό, κάνει την εκτέλεση των προγραμμάτων που έχουν μεταγλωττιστεί αρκετά ταχύτερη, δεδομένου ότι δεν είναι απαραίτητη η μετάφραση της κάθε εντολής, αλλά αρκεί μόνο η εκτέλεση του

μεταγλωττισμένου κώδικα. Ειδικότερα σε περιπτώσεις όπου έχουμε επανάληψη μία ομάδας εντολών πολλές φορές, δηλαδή έχουμε τη χρήση βρόχων (loops), το κέρδος από την μεταγλώττιση είναι πολύ μεγάλο.

Οι εντολές JavaScript είναι πολύ συχνά διασκορπισμένες σε διάφορα σημεία ενός αρχείου, όπου εμφανίζονται επιπλέον οδηγίες σε γλώσσες όπως HTML και CSS ή ακόμη και εντολές PHP. Επομένως, η επιλογή διερμηνευτή για τη γλώσσα ήταν μονόδρομος. Αυτό όμως κάνει την εκτέλεση των προγραμμάτων JavaScript αργή. Σε περιπτώσεις όπου για παράδειγμα, έχουμε κώδικα JavaScript ο οποίος ελέγχει την είσοδο των στοιχείων που εισάγει ο τελικός χρήστης σε μία ιστοσελίδα, αυτό δεν αποτελεί πρόβλημα, αφού ο επιπλέον χρόνος που απαιτείται είναι πολύ μικρός. Εάν όμως χρησιμοποιούμε την JavaScript για την εμφάνιση γραφικών για κάποια παιχνίδια ή για επαναλαμβανόμενες αριθμητικές πράξεις, για σκοπούς δηλαδή για τους οποίους δεν σχεδιάστηκε αρχικά η γλώσσα, η επιβάρυνση στον χρόνο είναι πολύ σημαντική και αποτρεπτική για τον τελικό χρήστη.

Για να αντιμετωπίσουν το πρόβλημα αυτό και να βελτιώσουν την απόδοση των εφαρμογών JavaScript, οι κατασκευαστές προγραμμάτων πλοήγησης δημιούργησαν εργαλεία για τη στιγμιαία μεταγλώττιση του κώδικα, που ονομάστηκαν Just-In-Time (JIT) compilers. Τα εργαλεία αυτά είναι διαφορετικά για τον κάθε κατασκευαστή, αλλά όλα έχουν παρόμοια φιλοσοφία [Clark 2017]. Ένας JIT compiler διαθέτει έναν αναλυτή που ονομάζεται *monitor* ή *profiler*, δουλειά του οποίου είναι η εξέταση του κώδικα, έτσι ώστε να εντοπιστούν οι εντολές οι οποίες επαναλαμβάνουν συχνά την εκτέλεσή τους. Τα τμήματα εκείνα του κώδικα τα οποία επαναλαμβάνονται χαρακτηρίζονται ως “θερμά” (warm). Ιδιαίτερα, τα τμήματα εκείνα τα οποία επαναλαμβάνονται πάρα πολλές φορές χαρακτηρίζονται “καυτά” (hot).

Η πιο πάνω ανάλυση, γίνεται κατά την εκτέλεση του προγράμματος. Όταν μία εντολή χαρακτηριστεί ως “θερμή”, τότε γίνεται η μεταγλώττισή της και κατά τις επόμενες εμφανίσεις της γίνεται εκτέλεση του μεταγλωττισμένου κώδικα. Αποφεύγεται δηλαδή η εκ νέου μετάφραση της εντολής, κερδίζοντας έτσι χρόνο. Εάν μία εντολή επαναληφθεί πολλές φορές και χαρακτηριστεί ως “καυτή”, τότε γίνεται επιπλέον βελτιστοποίηση (optimization) στη μεταγλώττισή της, κάνοντας για παράδειγμα υποθέσεις για τους τύπους των δεδομένων που χρησιμοποιεί, με βάση τις προγενέστερες εκτελέσεις της εντολής. Με τον τρόπο αυτό, κερδίζεται ακόμη περισσότερος χρόνος, αφού αποφεύγεται η εξέταση των

τύπων των μεταβλητών που συμμετέχουν στις πράξεις. Η JavaScript όμως, είναι μία γλώσσα ασθενών τύπων (weakly typed). Αυτό σημαίνει ότι ο τύπος μίας μεταβλητής μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος. Κάτι τέτοιο, έχει ως αποτέλεσμα την παύση χρήσης του βελτιστοποιημένου κώδικα που δημιουργήθηκε και αναφέρεται ως deoptimization ή bailling out. Τα περισσότερα προγράμματα πλοήγησης θέτουν όριο στον αριθμό των deoptimizations που μπορούν να γίνουν, διαφορετικά μπορεί να υπάρξουν περιπτώσεις όπου ένα πρόγραμμα εκτελείται γρηγορότερα αν αποφευχθεί τελείως η φάση της βελτιστοποίησης.

Η χρήση των JIT compilers βελτιώνει σε έναν βαθμό την ταχύτητα εκτέλεσης των εφαρμογών JavaScript, αλλά όχι στον επιθυμητό βαθμό, αφού συχνά υπάρχει σημαντική υστέρηση στην απόδοση των εφαρμογών JavaScript σε σχέση με εφαρμογές όπου έχει γίνει μεταγλώττιση. Για τον λόγο αυτό, έχουν γίνει επιπλέον προσπάθειες βελτίωσης της απόδοσης, όπως αναλύεται πιο κάτω.

2.3 asm.js

Παρόλη τη βελτίωση στη ταχύτητα που εμφάνισε η JavaScript, οι επιδόσεις των εφαρμογών που γράφονταν με τη γλώσσα, υστερούσαν ακόμη σε μεγάλο βαθμό σε σχέση με το επιθυμητό, κυρίως για περιπτώσεις όπου οι εφαρμογές έπρεπε να διαχειριστούν μεγάλο όγκο δεδομένων. Η Mozilla προσπάθησε να βελτιώσει την απόδοση των εφαρμογών JavaScript μέσω της *asm.js*. Η asm.js είναι ένα υποσύνολο της γλώσσας JavaScript το οποίο περιλαμβάνει μόνο κάποια επιθυμητά στοιχεία της γλώσσας τα οποία επιδέχονται βελτιστοποιήσεων κατά την μετάφρασή τους. Στο υποσύνολο αυτό της γλώσσας, είναι δυνατόν να μεταγλωττιστεί ένα υπάρχον πρόγραμμα που έχει ήδη αναπτυχθεί σε άλλη γλώσσα προγραμματισμού, η οποία έχει καλύτερες επιδόσεις ως προς την ταχύτητα σε σχέση με την JavaScript, όπως για παράδειγμα η C ή η C++. Με άλλα λόγια, με τον τρόπο αυτόν, η γλώσσα JavaScript έγινε γλώσσα-στόχος (compile-target) για άλλες γρηγορότερες γλώσσες προγραμματισμού [Herman 2014].

Υπάρχουν έτοιμα εργαλεία, όπως το Emscripten, τα οποία μπορούν να μεταφράσουν τον πηγαίο κώδικα ενός προγράμματος που έχει δημιουργηθεί σε μία γλώσσα προγραμματισμού, όπως η C ή η C++, σε μορφή asm.js. Στη συνέχεια, ο παραγόμενος

κώδικας μπορεί να εκτελεστεί από οποιοδήποτε πρόγραμμα πλοήγησης. Με τον τρόπο αυτό, μπορούν να δημιουργηθούν ιστοσελίδες, οι οποίες εμπεριέχουν εφαρμογές, οι οποίες έχουν δημιουργηθεί αρχικά με τη βοήθεια άλλων γλωσσών προγραμματισμού, όπως παιχνίδια για παράδειγμα. Οι εφαρμογές αυτές εκτελούνται ταχύτερα, από ότι αν δημιουργούνταν από την αρχή σε γλώσσα JavaScript.

Η γλώσσα asm.js δεν χρησιμοποιήθηκε μόνο για παιχνίδια, αλλά και για άλλου είδους εφαρμογές. Για παράδειγμα, η Facebook χρησιμοποίησε την τεχνολογία αυτή, για την συμπίεση των εικόνων που χρησιμοποιούν οι χρήστες στο προφίλ τους και η Adobe για την ανάπτυξη του Lightroom, μίας on-line εφαρμογής επεξεργασίας εικόνας [Wagner 2017]. Ένα από τα βασικά προβλήματα της asm.js, σύμφωνα με τον Zakai, ήταν η έλλειψη από πρότυπα τα οποία να υποστηρίζονται από όλους τους κατασκευαστές, με αποτέλεσμα να υπάρχουν θέματα ασυμβατότητας ανάμεσα σε διαφορετικά προγράμματα πλοήγησης. Επιπλέον, υπήρξε αρκετά μεγάλη καθυστέρηση κατά την φόρτωση της εφαρμογής από το πρόγραμμα πλοήγησης, κάτι που ήταν αποτρεπτικό από τη μεριά του τελικού χρήστη [Zakai 2017]. Για τους πιο πάνω λόγους, συνεχίστηκε η αναζήτηση για νέες τεχνολογίες που θα βελτιώσουν ακόμη περισσότερο την ταχύτητα των εφαρμογών JavaScript.

2.4 Emscripten και LLVM

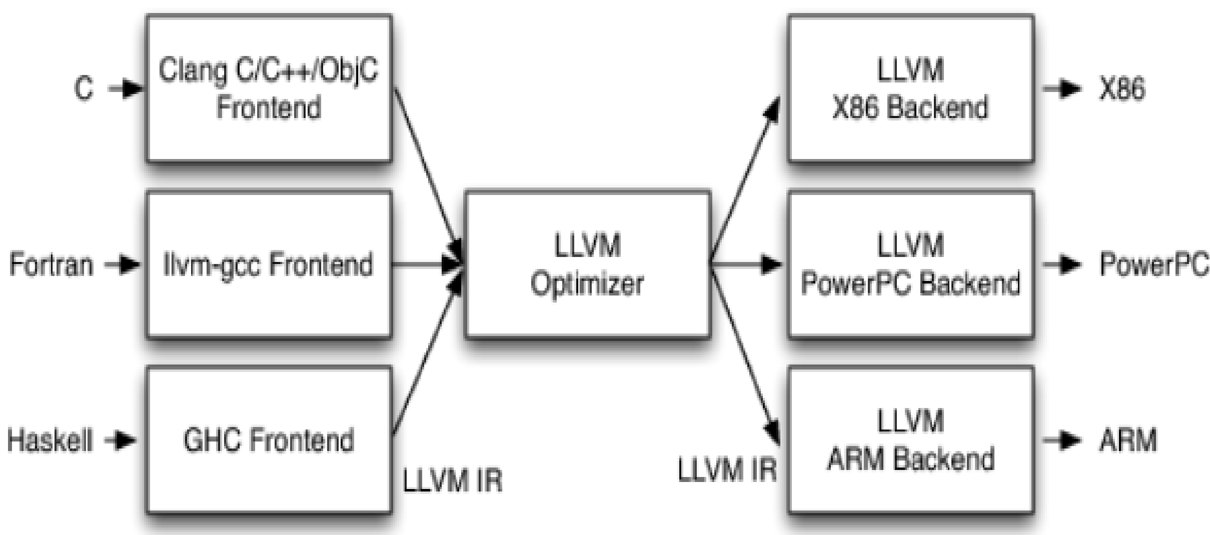
Η ιδέα της μεταγλώττισης ενός έτοιμου προγράμματος που έχει δημιουργηθεί με άλλη γλώσσα, σε τέτοια μορφή ώστε να είναι άμεσα εκτελέσιμο από ένα πρόγραμμα πλοήγησης, ήταν ιδιαίτερα ελκυστική και δεν ήταν εντελώς καινούργια. Με την τεχνολογία *Web Toolkit* της Google, είναι δυνατή η μετατροπή ενός προγράμματος από Java σε JavaScript. Επίσης, με την τεχνολογία *Pyjamas* είναι δυνατή η μετατροπή ενός προγράμματος από γλώσσα Python σε JavaScript. Παρόλα αυτά, μία τέτοιου είδους ένα-προς-ένα μεταγλώττιση από μία άλλη γλώσσα προγραμματισμού, σε μορφή JavaScript, δεν είναι καθόλου εύκολη, αφού η JavaScript δεν έχει σχεδιαστεί για μεγάλες και απαιτητικές εφαρμογές. Το αποτέλεσμα είναι, όταν γίνει η μετατροπή αυτή, να δημιουργούνται συχνά αρκετά προβλήματα κατά την εκτέλεση του προγράμματος. Επιπρόσθετα, πολλές από τις προγραμματιστικές διεπαφές (APIs), όπως για παράδειγμα

το σύστημα του ήχου, λειτουργούν διαφορετικά μέσα από μία εφαρμογή Παγκοσμίου Ιστού [Wagner 2017].

Με τη βοήθεια εργαλείων όπως ο μεταγλωττιστής *Emscripten*, είναι δυνατόν ένας προγραμματιστής να γράψει ένα πρόγραμμα σε χαμηλότερου επιπέδου γλώσσα, όπως η C ή η C++, και στη συνέχεια να μεταγλωττίσει το πρόγραμμα αυτό σε JavaScript, διατηρώντας ως έναν βαθμό τις επιδόσεις του αρχικού προγράμματος. Με τον τρόπο αυτό έχουμε μία βελτίωση της ταχύτητας του προγράμματος, σε σχέση με την περίπτωση να γραφόταν το πρόγραμμα από την αρχή σε JavaScript, διατηρώντας όμως μία υστέρηση, που κάποιες φορές μπορεί να είναι σημαντική, σε σχέση με το πρόγραμμα σε μορφή άλλης "γρηγορότερης" γλώσσας. Στην πραγματικότητα η μετατροπή γίνεται σε γλώσσα asm.js, η οποία όπως αναφέρθηκε είναι ένα υποσύνολο της γλώσσας.

Στην ουσία, με το εργαλείο Emscripten, μεταφράζουμε ένα πρόγραμμα από τη μορφή *LLVM* (Low-Level Virtual Machine), μία ενδιάμεση μορφή κώδικα που είναι πολύ διαδεδομένη στον χώρο των μεταγλωττιστών, σε μορφή asm.js. Η μετατροπή σε μορφή LLVM μπορεί να γίνει από έναν μεταγλωττιστή της γλώσσας στην οποία έχει δημιουργηθεί το αρχικό πρόγραμμα. Με τον τρόπο αυτό, το εργαλείο Emscripten μπορεί να χρησιμοποιηθεί για την υποστήριξη πολλών γλωσσών προγραμματισμού. Με τη βοήθεια του Emscripten, έχει γίνει δυνατή η χρήση μέσω του Παγκοσμίου Ιστού, λογισμικού όπως το Unreal Engine 3, SQLite MeshLab, Bullet Physics, AutoCAD, μέρος της βιβλιοθήκης Qt κλπ. [Emscripten 2020].

Ο LLVM είναι ένα σύνολο από εργαλεία μεταγλώττισης που γράφτηκαν σε C++ και σχεδιάστηκαν κυρίως για την βελτιστοποίηση κατά τον χρόνο μεταγλώττισης (compiling), σύνδεσης (linking) και εκτέλεσης των προγραμμάτων. Ο LLVM μπορεί να υποστηρίξει οποιαδήποτε γλώσσα προγραμματισμού και υπάρχει πολύ μεγάλη ποικιλία από front-ends για γλώσσες όπως Java, Python, C/C++ μέσω του Clang, Objective C, Haskell, Ruby, ActionScript, GLSL κλπ. Η ανάπτυξη του LLVM ξεκίνησε το 2000 από τους Vikram Ade και Chris Lattner στο Πανεπιστήμιο του Illinois at Urbana-Champaign ως ερευνητικό εργαλείο για την διερεύνηση τεχνικών δυναμικής μεταγλώττισης. Το 2005 η Apple προσέλαβε τον Chris Lattner και οργάνωσε μια ομάδα γύρω του, για να δουλέψει πάνω στον LLVM για χρήσεις πάνω στο λογισμικό της εταιρίας [Lattner 2006].



Εικόνα 2-4: Στάδια μεταγλώττισης LLVM

Στο πρώτο στάδιο μεταγλώττισης LLVM, αναλαμβάνει το front-end τον έλεγχο του πηγαίου κώδικα, που μπορεί να είναι γραμμένος σε μία από τις γλώσσες που αναφέρθηκαν, για λεκτικά και συντακτικά λάθη και να την μετατροπή του σε μορφή ενδιάμεσου κώδικα (Intermediate Representation - IR). Ο ενδιάμεσος κώδικας IR στη συνέχεια μπορεί να περάσει από μία σειρά προαιρετικών αναλύσεων και βελτιστοποιήσεων. Τέλος, ο βελτιστοποιημένος αυτός ενδιάμεσος κώδικας περνάει στον code-generator, ο οποίος δημιουργεί ως έξοδο την τελική μορφή του κώδικα (μορφή assembly), που μπορεί να τρέξει στην αρχιτεκτονική του συστήματος που μας ενδιαφέρει, όπως x86, PowerPC, ARM κτλ. [LLVM 2014]. Τα στάδια μεταγλώττισης LLVM εμφανίζονται στην *Εικόνα 2-4*.

Ο LLVM optimizer διαβάζει τον κώδικα IR, τον επεξεργάζεται και στη συνέχεια δίνει σαν έξοδο τον βελτιστοποιημένο LLVM IR, που οδηγεί σε ταχύτερο κώδικα. Ο optimizer εκτελεί μία σειρά από περάσματα (passes) βελτιστοποιήσεων, όπου το κάθε πέρασμα λειτουργεί στην έξοδο του προηγούμενου. Εξαιτίας της μη υποστήριξης δομών ελέγχουν από τη γλώσσα LLVM, απαιτούνται ειδικός αλγόριθμος αναδόμησης βρόχων, που ονομάστηκε relooper, που επιτρέπει τη δημιουργία αποδοτικού κώδικα JavaScript [Zakai 2011].

Μετά τις αρχικά επιτυχημένη προσπάθεια εκτέλεσης απαιτητικών εφαρμογών, όπως παιχνίδια, μέσα από το περιβάλλον ενός προγράμματος πλοήγησης, κατά την προσπάθεια εκτέλεσης παιχνιδιών που βασίζονται στην παιχνιδομηχανή (game-engine) Unity, μέσω της μετατροπής τους σε JavaScript με τη βοήθεια του Emscripten, εμφανίστηκαν κάποια προβλήματα απόδοσης. Μέρος των προβλημάτων αυτών, αντιμετωπίστηκε με επιτυχία, όταν περιόρισαν το σύνολο των παραγόμενων εντολών JavaScript [Wagner 2017]. Έτσι, καταλήξαμε στο υποσύνολο της γλώσσας που ονομάζεται asm.js, όπου περιέχονται τα στοιχεία της γλώσσας τα οποία επιδέχονται πρόωρη μεταγλώττιση (ahead of time – AOT compilation).

Το εργαλείο Emscripten μπορεί να χρησιμοποιηθεί επίσης και για τη γλώσσα WebAssembly, όπως θα δούμε πιο κάτω. Στην πραγματικότητα, η χρήση της asm.js έχει ήδη παραγκωνιστεί, εξαιτίας της ανάπτυξης της WebAssembly, αφού επιτρέπει τη δημιουργία ακόμη γρηγορότερων εφαρμογών. Το Emscripten όμως και ο LLVM εξακολουθούν να αποτελούν πολύτιμα εργαλεία στην προσπάθεια εκτέλεσης απαιτητικών εφαρμογών μέσω του Παγκόσμιου Ιστού.

2.5 Native Client της Google

Η Google προσπάθησε να αυξήσει τις επιδόσεις των εφαρμογών του Παγκοσμίου Ιστού αρχικά με την ανοικτού κώδικα τεχνολογία *Native Client* (NaCl) και αργότερα με την τεχνολογία *Portable Native Client* (PNaCl). Με την τεχνολογία Native Client, είναι δυνατή η εκτέλεση του αντικείμενου κώδικα μίας εφαρμογής, σε μορφή x86, ARM ή MIPS, που έχει γραφτεί σε οποιαδήποτε γλώσσα προγραμματισμού, μέσα από το περιβάλλον ενός προγράμματος πλοήγησης [Donovan 2010]. Κάτι παρόμοιο επιτύγχανε και η τεχνολογία ActiveX της Microsoft, αλλά η διαφορά είναι ότι με την Native Client τεχνολογία, ο κώδικας εκτελείται μέσα σε ένα ελεγχόμενο περιβάλλον (sandbox), χωρίς να έχει πλήρη ελευθερία ώστε να υπάρχουν θέματα ασφάλειας [Yee 2009].

Η Google ανακοίνωσε το 2011 την υποστήριξη διάφορων παιχνιδιών, γνωστά για τις μεγάλες απαιτήσεις τους σε επεξεργαστική ισχύ, τα οποία ήταν δυνατόν να εκτελεστούν μέσα από το περιβάλλον του Google Chrome, με τη βοήθεια της τεχνολογίας NaCl. Ένας από τους βασικούς στόχους που τέθηκαν, ήταν η ασφάλεια του συστήματος που φιλοξενεί

την εφαρμογή, κάτι που δεν μπόρεσε να πετύχει η τεχνολογία ActiveX [Yee 2009]. Οι μη ασφαλείς κλήσεις συστήματος δεν επιτρέπονταν από το σύστημα και μάλιστα η Google έδινε αμοιβή σε οποιονδήποτε ανακάλυπτε κάποιο κενό ασφαλείας. Ένα από τα μειονεκτήματα του NaCl ήταν ότι έπρεπε να δημιουργούνται διαφορετικά εκτελέσιμα αρχεία (μορφή nexe) για κάθε διαφορετική αρχιτεκτονική. Στην ουσία, το μειονέκτημα αυτό, περιόρισε τη χρήση της τεχνολογίας σε εφαρμογές που μπορούσε να κατεβάσει ο χρήστης από Google Web Store.

Το PNaCl στην ουσία επιλύει κάποια θέματα φορητότητας (portability) ανάμεσα σε διαφορετικές πλατφόρμες (x86, MIPS και ARM), αφού επιτρέπει την μεταγλώττιση του αρχικού πηγαίου κώδικα σε μία ενδιάμεση μορφή (αρχείο pexe), το οποίο μπορεί να διανεμηθεί μαζί με τα αρχεία μίας ιστοσελίδας και στη συνέχεια να εκτελεστεί από το πρόγραμμα πλοήγησης [Donovan 2010]. Τον Μάιο του 2017, η Google ανακοίνωσε την σταδιακή απόσυρση της τεχνολογίας PNaCl, λόγω υποστήριξης της τεχνολογίας WebAssembly, η οποία φαίνεται να είναι ακόμη πιο υποσχόμενη. Ένας λόγος για την μεταστροφή της εταιρείας είναι η περιορισμένα αποδοχή της τεχνολογίας από άλλους κατασκευαστές και η περιορισμένη χρήση της [Nelson 2017].

Οι τεχνολογίες που παρουσιάστηκαν πιο πάνω, asm.js, Emscripten και NaCl, έχουν προσφέρει πάρα πολλά στην ανάπτυξη του προτύπου της WebAssembly, αφού το πρότυπο της WebAssembly χρησιμοποιεί πολλά στοιχεία από τις τεχνολογίες αυτές και αποτελεί τον φυσικό συνεχιστή τους.

3 Το πρότυπο WebAssembly

3.1 Ιστορική αναδρομή και εξάπλωση του προτύπου

Το πρότυπο *WebAssembly*, που συχνά αναφέρεται και *Wasm* ή απλά *WA*, είναι ένα ανοικτό πρότυπο που περιγράφει τη δυαδική κωδικοποίηση εκτελέσιμων προγραμμάτων, με τη βοήθεια του οποίου μπορεί να διασφαλιστεί η φορητότητα, ασφάλεια και η ταχύτητα εκτέλεσης μίας εφαρμογής [WebAssembly 2020]. Το πρότυπο περιλαμβάνει και την αντίστοιχη γλώσσα Assembly, μαζί με ένα σύνολο διεπαφών για την επικοινωνία των WebAssembly εφαρμογών με το περιβάλλον όπου φιλοξενούνται. Ο βασικός στόχος του προτύπου είναι η ενσωμάτωση εφαρμογών υψηλών επιδόσεων μέσα σε ιστοσελίδες, με τη δημιουργία κατάλληλου περιβάλλοντος εκτέλεσης (runtime environment – RE). Όμως, το πρότυπο σχεδιάστηκε με τέτοιο τρόπο, έτσι ώστε να μπορεί να χρησιμοποιηθεί και για την ανάπτυξη αυτόνομων εφαρμογών. Στην ουσία, η παραγωγή του κώδικα γίνεται συνήθως με αυτόματο τρόπο, με εργαλεία όπως το Emscripten που παρουσιάστηκε στο προηγούμενο κεφάλαιο, και δεν απαιτείται η συγγραφή WebAssembly κώδικα “με το χέρι” από κάποιους προγραμματιστές.

Το πρότυπο WebAssembly ανακοινώθηκε για πρώτη φορά το 2015. Η πρώτη επίδειξη του προτύπου έγινε με την εκτέλεση του παιχνιδιού Angry Bots που δημιουργήθηκε με το Unity, τόσο μέσα από το περιβάλλον του Mozilla Firefox, όσο μέσα από το Google Chrome και Microsoft Edge. Τον Μάρτιο του 2017 ανακοινώθηκε η πρώτη MVP (Minimum Viable Product) έκδοση του προϊόντος και τον Σεπτέμβρη του 2017 ανακοινώθηκε η υποστήριξη από το Safari 11 [Krill 2017]. Τον Φεβρουάριο του 2018 ανακοινώθηκαν οι προδιαγραφές του προϊόντος σε μορφή working draft από το WebAssembly Working Group [W3C 2018]. Το WebAssembly Working Group έχει σαν μέλη τους βασικότερους κατασκευαστές προγραμμάτων πλοήγησης, κάτι που αποδίδει ιδιαίτερη βαρύτητα στην προσπάθεια.

Κατά τον Οκτώβρη του 2020 πλέον, ο Mozilla Firefox και ο Google Chrome, που είναι τα πιο διαδεδομένα προγράμματα πλοήγησης, υποστηρίζουν τη μορφή εφαρμογών wasm (της WebAssembly) σε περιβάλλον Linux, MacOS, Windows και Android. Επίσης, η τελευταία έκδοση του Microsoft Edge, όπως και του Safari περιλαμβάνουν πλήρη υποστήριξη για το πρότυπο WebAssembly. Η Autodesk σχεδιάζει την υποστήριξη της μορφής wasm για την παιχνιδομηχανή της Stingray v1.8. Έχει υπολογιστεί ότι το πρότυπο υποστηρίζεται από το 92,93% των σύγχρονων προγραμμάτων πλοήγησης για υπολογιστές γραφείου (desktop browsers) και από το 93,44% των σύγχρονων προγραμμάτων πλοήγησης για φορητές συσκευές, όπως smartphones και tablets [CanIUse 2020]. Για παλαιότερες εκδόσεις των προγραμμάτων πλοήγησης, είναι δυνατή η χρήση ειδικών βιβλιοθηκών για την μετατροπή του wasm κώδικα σε asm.js, έτσι να έχουμε την επιτυχή εκτέλεσή του. Είναι πολύ εύκολο επομένως, να παρατηρήσει κάποιος, ότι το πρότυπο WebAssembly απολαμβάνει την καθολική υποστήριξη και αποδοχή από όλους τους βασικούς "παίκτες" του χώρου, οι οποίοι μάλιστα συνεργάζονται για την βελτίωση και την εξέλιξή του.

3.1.1 Υποστήριξη γλωσσών προγραμματισμού

Αυτή τη στιγμή η WebAssembly υποστηρίζεται κυρίως από τη γλώσσες C και C++. Κώδικας που έχει δημιουργηθεί με κάποια από αυτές τις γλώσσες, μπορεί να μεταγλωττιστεί σε WebAssembly με τη βοήθεια του Emscripten, άλλα και άλλων εργαλείων που βασίζονται στο LLVM. Ωστόσο, υπάρχουν πολλές άλλες δημοφιλείς γλώσσες προγραμματισμού, οι οποίες έχουν ξεκινήσει σε πειραματικό στάδιο την υποστήριξη του προτύπου και ο βαθμός υποστήριξης αυξάνεται συνεχώς. Υπάρχει μία σειρά εργαλείων και μεταγλωττιστών, τα οποία βρίσκονται σε συνεχή εξέλιξη και υπόσχονται τη διαδεδομένη υποστήριξη του προτύπου στο άμεσο μέλλον [Awsome WASM 2020].

Κάποιες από τις γλώσσες που υποστηρίζουν το πρότυπο WebAssembly, εμφανίζονται πιο κάτω:

- **C** και C++. Η υποστήριξη από τις γλώσσες αυτές είναι πολύ καλή με τη βοήθεια του Emscripten.
- **Rust**. Η WebAssembly αποτελεί επίσημα υποστηριζόμενο στόχο μεταγλώττισης, σε αρκετά προχωρημένο σημείο, και μάλιστα υπάρχει ενεργή κοινότητα για την υποστήριξη του προτύπου.
- **Go**. Η γλώσσα Go υποστηρίζει πλέον και επίσημα, αν και σε πειραματικό στάδιο, το πρότυπο της WebAssembly.
- **C#**. Υποστηρίζει πειραματικά την WebAssembly με τη βοήθεια της τεχνολογίας Blazor, που υιοθετήθηκε πειραματικά από τη Microsoft, αφού πρώτα γίνει η ενσωμάτωση κάποιων .NET βιβλιοθηκών στη Wasm.
- **Typescript**. Η TypeScript αποτελεί επέκταση της Javascript υποστηρίζοντας τον ορισμό τύπων δεδομένων. Ένας κώδικας Typescript μετατρέπεται σε κώδικα Javascript πριν την εκτέλεσή του. Η γλώσσα υποστηρίζει, σε αρκετά πειραματικό στάδιο, το πρότυπο WebAssembly, με τη βοήθεια του εργαλείου AssemblyScript, που αποτελεί ανοικτό λογισμικό βασισμένο στον μεταγλωττιστή Binrayen. Ο μεταγλωττιστής Binrayen είναι γραμμένος σε C++ και έχει ως στόχο τη μεταγλώττιση σε WebAssembly, κώδικα που έχει δημιουργηθεί σε άλλες γλώσσες προγραμματισμού.
- **Java**. Υποστηρίζεται η WebAssembly με τη βοήθεια του λογισμικού TeaVM που επιτρέπει την μετατροπή των bytecodes που δημιουργούνται κατά τη μεταγλώττιση ενός προγράμματος Java σε WebAssembly. Το TeaVM μάλιστα μπορεί να χρησιμοποιηθεί και για τη μεταλγώττιση κώδικα Kotlin ή Scala. Ένα παρόμοιο εργαλείο, που μπορεί να χρησιμοποιηθεί για τον ίδιο σκοπό, είναι το ByteCoder.
- **Python**. Υπάρχει το Pyodide που προσφέρει υποστήριξη για WebAssembly και περιλαμβάνει μάλιστα και τις επιστημονικές βιβλιοθήκες της Python, όπως Numpy, Pandas και matplotlib.
- **PHP**. Έχει ήδη αναπτυχθεί πειραματικό πρωτότυπο για την υποστήριξη του προτύπου.

- **Perl**. Το WebPerl υποστηρίζει τη μετατροπή σε WebAsselbmy, κώδικα που έχει αναπτυχθεί σε γλώσσα Perl. Με τον τρόπο αυτό μπορούμε να εκτελέσουμε σενάριο Perl στον Παγκόσμιο Ιστό.

Η πιο πάνω λίστα γλωσσών προγραμματισμού οι οποίες υποστηρίζουν, σε αρχικό έστω στάδιο, το πρότυπο WebAssembly, δείχνει την εξάπλωση του προτύπου και υπόσχεται την ευρεία αποδοχή του στο άμεσο μέλλον.

3.2 Τρόπος εκτέλεσης κώδικα WebAssembly

Ο στόχος της WebAssembly είναι η άμεση εκτέλεση στο περιβάλλον του Παγκόσμιου Ιστού, κώδικα που έχει γραφεί σε άλλη γλώσσα προγραμματισμού, όπως η C και η C++, χωρίς την μετατροπή του αρχικού κώδικα. Ο πηγαίος κώδικας μεταγλωττίζεται με τη βοήθεια των κατάλληλων εργαλείων, έτσι ώστε να παραχθεί ο εκτελέσιμος κώδικας Wasm. Το παραγόμενο Wasm module που προκύπτει, μπορεί να χρησιμοποιηθεί με τρόπο παρόμοιο με αυτό των βιβλιοθηκών των γλωσσών προγραμματισμού, να φορτωθεί δηλαδή από κάποιον άλλον εξωτερικό κώδικα, ο οποίος μπορεί να καλέσει τις παρεχόμενες λειτουργίες (συναρτήσεις).

Το περιβάλλον όπου φιλοξενείται και εκτελείται ο κώδικας WebAssembly είναι συνήθως ένα πρόγραμμα πλοήγησης που έχει τη δυνατότητα εκτέλεσης κώδικα Javascript. Η μεταγλώττιση μπορεί να γίνει είτε με την τεχνική JIT (Just In Time) είτε με την τεχνική AOT (Ahead Of Time), ανάλογα με τις επιθυμητές ρυθμίσεις. Σύμφωνα με την τεχνική JIT, οι συναρτήσεις μεταγλωττίζονται την πρώτη φορά που πρέπει να χρησιμοποιηθούν. Αντίθετα, σύμφωνα με την τεχνική AOT, όλες οι συναρτήσεις του Wasm module μεταγλωττίζονται κατά τη φόρτωση του module. Στη συνέχεια οι μεταγλωττισμένες συναρτήσεις αποθηκεύονται τοπικά στη μνήμη και εκτελούνται κατά την κλήση τους από τον κώδικα του προγράμματος. Στην Εικόνα 3-1 εμφανίζεται η διαδικασία που περιγράφεται πιο πάνω.

Όταν το σύστημα που φιλοξενεί τον κώδικα Wasm θέλει να καλέσει μία συνάρτηση, θα πρέπει να αρχικοποιήσει το Wasm module. Η αρχικοποίηση αυτή περιλαμβάνει τον έλεγχο

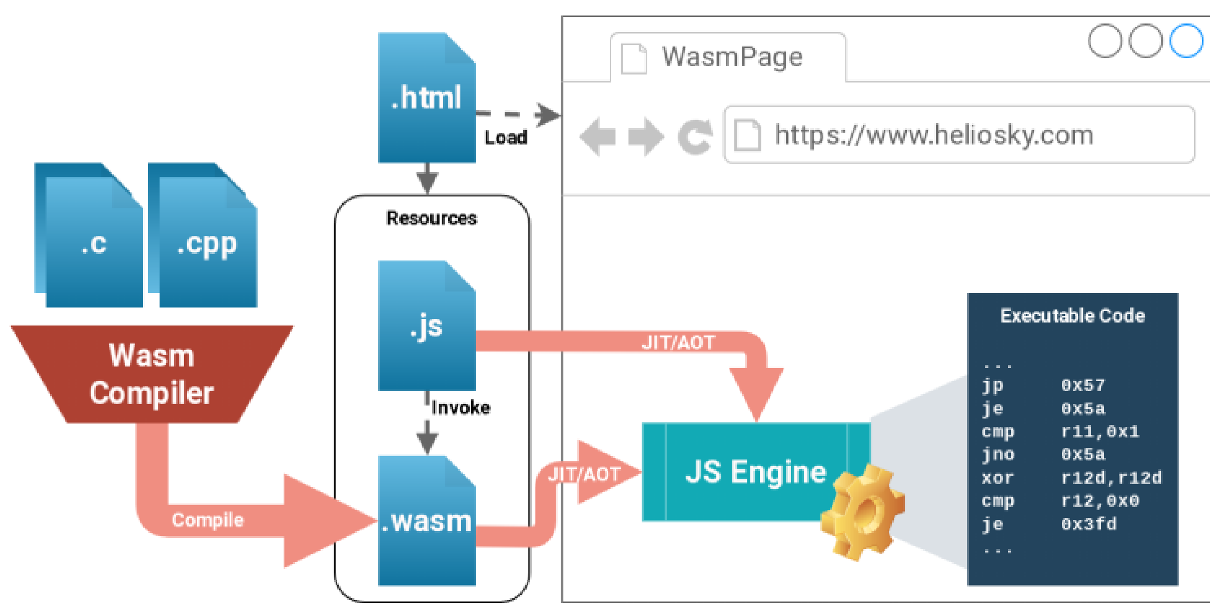
για τυχόν προαπαιτούμενες βιβλιοθήκες που ίσως είναι απαραίτητες για την εκτέλεση του κώδικα και αναλαμβάνει την σύνδεσή τους (linking) με τον κώδικα. Το πρότυπο WebAssembly δεν ορίζει κάποιου είδους βασική βιβλιοθήκη για την εκτέλεση των προγραμμάτων, οπότε θέλει προσοχή ώστε να φορτώνονται κάθε φορά όλες οι απαραίτητες βιβλιοθήκες για την εκτέλεση του κώδικα.

Το World Wide Web Consortium (W3C), που αποτελεί τον βασικό οργανισμό για την ανάπτυξη των προτύπων του Παγκόσμιου Ιστού, έχει υπό την αιγίδα του το WebAssembly Community Group (CG) για την προώθηση και επίβλεψη των θεμάτων που αφορούν το πρότυπο WebAssembly. Μία από τις δράσεις του WebAssembly CG είναι η προτυποποίηση του API που χρησιμοποιείται από τον κώδικα Wasm.

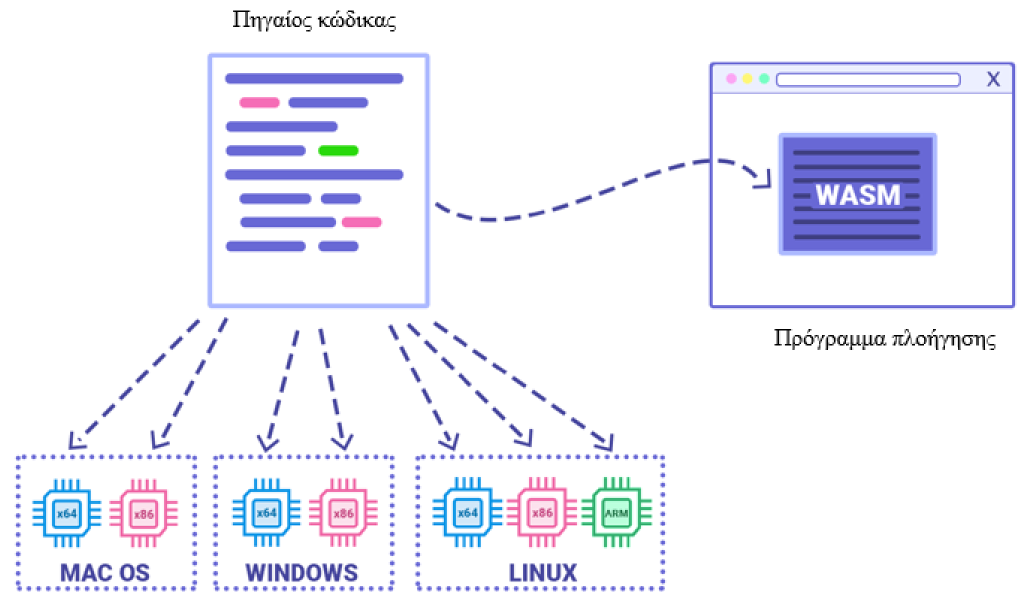
3.3 Αρχιτεκτονική του προτύπου WebAssembly

Το πρότυπο WebAssembly μοντελοποιεί τη δική του εικονική αρχιτεκτονική συνόλου εντολών (Virtual ISA). Μία αρχιτεκτονική συνόλου εντολών (Instruction Set Architecture – ISA) περιγράφει με αφαιρετικό τρόπο τη λειτουργία του συστήματος όπου εκτελείται ένα πρόγραμμα, περιγράφοντας τις μονάδες που είναι απαραίτητες για την σωστή εκτέλεση του προγράμματος, όπως το μοντέλο μνήμης του συστήματος, τον τρόπο κωδικοποίησης των εντολών κτλ.

Η αρχιτεκτονική που χρησιμοποιεί το πρότυπο WebAssembly είναι ανεξάρτητη από κάποια συγκεκριμένη πλατφόρμα (platform-independent) και δεν απαιτεί την εκτέλεση κάποιας ειδικής εικονικής μηχανής από το σύστημα όπου φιλοξενείται. Αντίθετα, μεταφράζει τις εντολές Wasm απευθείας σε κώδικα μηχανής, κάνοντας έτσι τη γλώσσα να είναι στην ουσία ένας ενδιάμεσος στόχος για άλλες γλώσσες προγραμματισμού. Με τον τρόπο αυτό η γλώσσα είναι ανεξάρτητη της πλατφόρμας που χρησιμοποιείται από ένα σύστημα. Στην Εικόνα 3-2 φαίνεται ο τρόπος με τον οποίο επιτυγχάνει το πρότυπο WebAssembly την ανεξαρτησία από το λειτουργικό σύστημα (πλατφόρμα) όπου εκτελείται ο κώδικας.



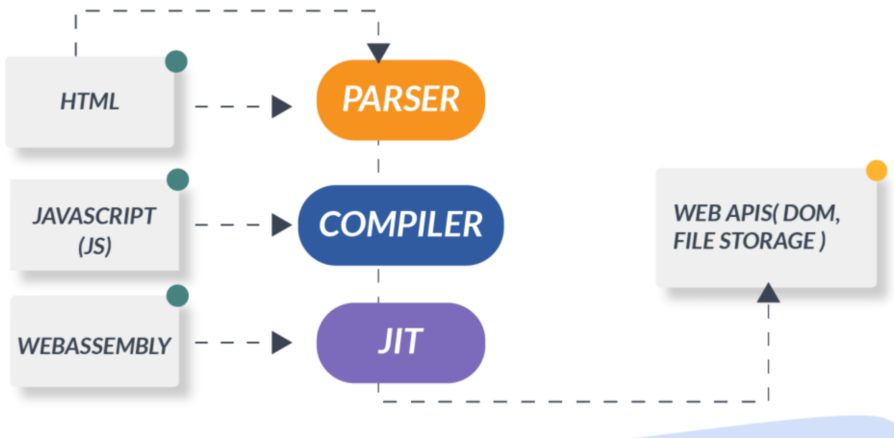
Εικόνα 3-1: *Μεταγλώττιση κώδικα WebAssembly*



Εικόνα 3-2*: Ανεξαρτησία WebAssembly από πλατφόρμα εκτέλεσης*

Το πρότυπο χρησιμοποιεί αρχιτεκτονική σωρού και κάθε εντολή χρησιμοποιεί τα ορίσματα που βρίσκονται αποθηκευμένα στην κορυφή μίας μνήμης σωρού. Με άλλα λόγια, κάθε εντολή παίρνει (λειτουργία pop) τα ορίσματα που χρειάζεται από την κορυφή του σωρού, υπολογίζει το αποτέλεσμα και το αποθηκεύει πίσω στο σωρό (λειτουργία push). Αυτός ο τρόπος λειτουργίας, είναι διαφορετικός από τον τρόπο που περιγράφεται στο μοντέλο που βασίζεται σε καταχωρητές. Το μοντέλο της αρχιτεκτονικής σωρού έχει κάποια πλεονεκτήματα, όπως το γεγονός ότι είναι περισσότερο συμπαγής ο κώδικας που παράγεται και μπορεί επομένως να επιτύχει μικρότερους χρόνους μετάφρασης. Ο αρχικός κώδικας μπορεί να αναπαρασταθεί εύκολα με ένα αφηρημένο συντακτικό δέντρο (Abstract Syntax Tree - AST) και μπορεί να εφαρμοστεί βελτιστοποίηση βασισμένη στην αρχιτεκτονική του συστήματος [WebAssembly CG, 2019]. Για τους λόγους αυτούς χρησιμοποιείται ευρέως πλέον από τις μοντέρνες εικονικές μηχανές και τους JIT μεταγλωττιστές.

Στην Εικόνα 3-3 εμφανίζεται ο τρόπος με τον οποίο χρησιμοποιείται ένας μεταγλωττιστής JIT για την παραγωγή κώδικα που μπορεί στη συνέχεια να κάνει χρήση των επιθυμητών Web APIs. Η ανάλυση του κώδικα HTML γίνεται από τον parser του προγράμματος πλοήγησης. Στη συνέχεια η μετάφραση του κώδικα Javascript γίνεται από το αντίστοιχο πρόγραμμα μεταγλώττισης, ενώ ο μεταγλωττιστής JIT αναλαμβάνει την μετάφραση του κώδικα Wasm [XenonStack, 2020]



Εικόνα 3-3: *Τρόπος εκτέλεσης κώδικα Wasm*

Η WebAssembly ορίζει ένα περιορισμένο σύνολο τύπων δεδομένων. Αυτή τη στιγμή, υποστηρίζει μόνο ακεραίους 32 και 64 δυαδικών ψηφίων, καθώς και αριθμούς κινητής υποδιαστολής. Άλλες γλώσσες assembly, όπως αυτές που απευθύνονται σε αρχιτεκτονική x86 και ARM, υποστηρίζουν διάφορα μεγέθη ακέραιων αριθμών από 8 έως 64 δυαδικά ψηφία. Το ίδιο ισχύει για τις γλώσσες C και C++, όπως και για τις άλλες γλώσσες για τις οποίες έχει ξεκινήσει η υποστήριξη Wasm, όπως περιγράφεται στις προηγούμενες παραγράφους. Επομένως, οι τύποι δεδομένων που απαιτούν μικρότερο χώρο αποθήκευσης, όπως οι τύποι short και char της C/C++, θα πρέπει να αναβαθμιστούν σε ακεραίους των 32 δυαδικών ψηφίων. Αυτό το γεγονός, δεν έρχεται σε πραγματική αντίθεση με τις προδιαγραφές της γλώσσας C/C++, δεδομένου ότι οι γλώσσες αυτές δεν ορίζουν ένα μέγιστο μέγεθος για τον χώρο αποθηκεύονται τα δεδομένα των τύπων αυτών, αλλά προσδιορίζουν μόνο τον τρόπο αποθήκευσης και το ελάχιστο χώρο για την αποθήκευσή τους.

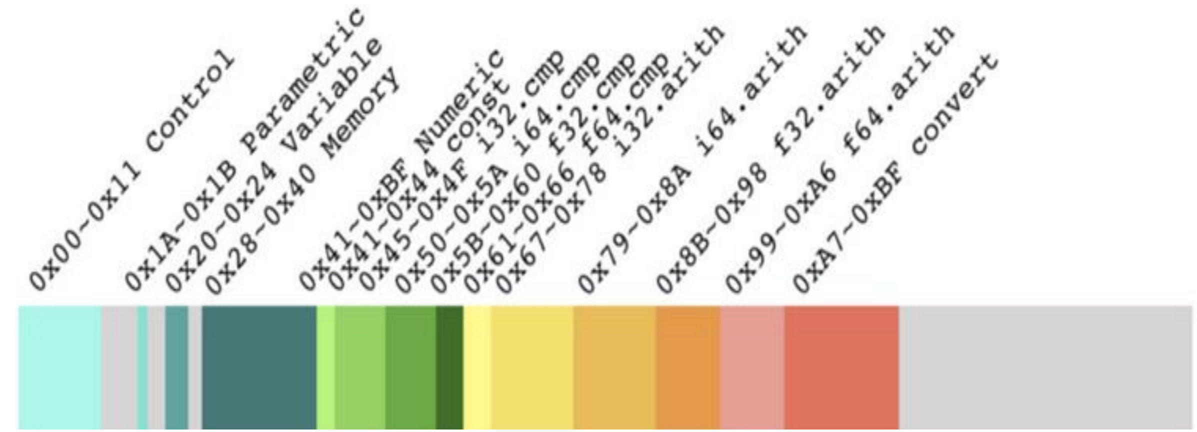
Η WebAssembly χρησιμοποιεί διάφορους τύπους εντολών. Ο **Πίνακας 1** περιλαμβάνει τους τύπους εντολών που χρησιμοποιούνται από τη γλώσσα. Οι περισσότερες εντολές λαμβάνουν κάποια ορίσματα ώστε να εκτελέσουν κάποια λειτουργία με αυτά. Τα ορίσματα που χρησιμοποιούνται, βρίσκονται αποθηκευμένα στο σωρό και ο τύπος τους θα πρέπει να ταιριάζει με τον τύπο δεδομένων που αναμένει η αντίστοιχη εντολή WebAssembly. Σε αντίθετη περίπτωση θα δημιουργηθεί λάθος εκτέλεσης του προγράμματος.

Πίνακας 1*: Τύποι εντολών της WebAssembly*

Τύπος εντολής	**Περιγραφή**
Αριθμητικές εντολές	Εκτελούν αριθμητικές και λογικές πράξεις στους αριθμούς που περιέχονται στη λίστα ορισμάτων τους
Εντολές χειρισμού σωρού	Χειρίζονται τα περιεχόμενα του σωρού που χρησιμοποιείται για την αποθήκευση των ορισμάτων
Εντολές χειρισμού μεταβλητών	Διαβάζουν ή τροποποιούν τα περιεχόμενα των τοπικών και καθολικών μεταβλητών που χρησιμοποιεί το πρόγραμμα
Εντολές χειρισμού μνήμης	Διαβάζουν δεδομένα από τη μνήμη ή αποθηκεύουν αποτελέσματα σε αυτή. Γίνεται χρήση γραμμικής μνήμης.
Εντολές ελέγχου	Εντολές που ελέγχουν τη ροή του προγράμματος.

Το μήκος για τον κωδικό λειτουργίας (opcode) μίας εντολής WebAssembly είναι ίσο με 1 Byte, δηλαδή 8 bits, και επομένως, μπορούν θεωρητικά να υπάρξουν μέχρι $2^8 = 356$ διαφορετικές εντολές. Το μήκος αυτό είναι ίσο με αυτό που χρησιμοποιείται και κατά την μεταγλώττιση σε Java bytecodes. Το πρότυπο Wasm1.0 ορίζει 172 συνολικά εντολές, που χωρίζονται στις κατηγορίες που εμφανίζει ο Πίνακας 1. Η γλώσσα ορίζει 13 εντολές ελέγχου, 5 εντολές χειρισμού μεταβλητών, 25 εντολές διαχείρισης μνήμης, 127

αριθμητικές εντολές και 2 εντολές σωρού. Περισσότερες από τα 2 τρίτα των εντολών επομένως είναι αριθμητικές. Στην Εικόνα 3-4 εμφανίζονται οι κωδικοί των εντολών της WebAssembly ανά κατηγορία εντολών. Για κάθε μία από τις εντολές, υπάρχει μία μνημονική ονομασία, ώστε να μπορεί να χρησιμοποιηθεί εύκολα από κάποιον προγραμματιστή. Για παράδειγμα, η μνημονική ονομασία i32.const χρησιμοποιείται για την εντολή με κωδικό 41 του δεκαεξαδικού συστήματος (0x41) για να δηλώσει μία ακέραια σταθερά μήκους 32 δυαδικών ψηφίων. Κάποιες από τις εντολές της γλώσσας, όπως η i32.const, λαμβάνουν άμεσα ένα στατικό όρισμα το οποίο χρησιμοποιείται από την εντολή. Άλλες εντολές λαμβάνουν δυναμικά το όρισμά τους από το σωρό, όπως ακριβώς συμβαίνει και με το περιβάλλον JVM της εκτέλεσης προγραμμάτων Java.



***Εικόνα 3-4**: Εντολές της γλώσσας WebAssembly*

3.4 Εντολές σωρού

Ένα από τα βασικότερα συστατικά της αρχιτεκτονικής της γλώσσας WebAssembly είναι ο σωρός, όπου μπορούν να αποθηκευτούν (push) και να διαβαστούν (pop) τα ορίσματα των εντολών. Η γλώσσα περιλαμβάνει 2 εντολές για τη διαχείριση του σωρού, η λειτουργία των οποίων περιγράφεται πιο κάτω.

Με την εντολή drop μπορεί να διαβαστεί το περιεχόμενο που υπάρχει στο επάνω μέρος του σωρού (λειτουργία pop). Η εντολή δεν λαμβάνει κάποια παράμετρο, απλά λαμβάνει το περιεχόμενο της κορυφής του σωρού, ανεξάρτητα του τύπου του. Η εντολή select από την άλλη μεριά, λαμβάνει 3 αριθμούς από το σωρό. Αν το πρώτο είναι 0 τότε τοποθετεί τον δεύτερο πίσω στο σωρό, διαφορετικά τοποθετεί τον τρίτο. Είναι χρήσιμη επομένως για περιπτώσεις όπου εκτελούνται εντολές διακλάδωσης (επιλογής). Ο πρώτος από τους αριθμούς αυτούς θα πρέπει να είναι ακέραιος 32 δυαδικών ψηφίων.

Η εισαγωγή δεδομένων στο σωρό, γίνεται συνήθως μέσω των αριθμητικών εντολών που θα περιγραφούν στη συνέχεια. Μία τέτοια εντολή συνήθως εκτελεί κάποια πράξει με τα δεδομένα που βρίσκονται στο επάνω μέρος του σωρού και στη συνέχεια τοποθετεί το αποτέλεσμα της πράξης πίσω στο σωρό (λειτουργία push).

3.5 Αριθμητικές εντολές

Οι μεγάλη πλειοψηφία από τις εντολές που περιλαμβάνει το σύνολο εντολών της WebAssembly είναι αριθμητικές εντολές. Οι αριθμητικές εντολές παίρνουν κάποια ορίσματα και στη συνέχεια εκτελούν κάποια αριθμητική ή λογική πράξη με τα ορίσματα αυτά, αποθηκεύοντας στη συνέχεια το αποτέλεσμα στη στοίβα.

Ανάλογα με τον τύπο των ορισμάτων, οι αριθμητικές εντολές κατανέμονται σε τέσσερις κατηγορίες:

- i32, αν κάνουν πράξεις με ακεραίους αριθμούς των 32 δυαδικών ψηφίων
- i64, αν κάνουν πράξεις με ακεραίους αριθμούς των 64 δυαδικών ψηφίων

- f32, αν κάνουν πράξεις με αριθμούς κινητής υποδιαστολής των 32 δυαδικών ψηφίων
- f64, αν κάνουν πράξεις με αριθμούς κινητής υποδιαστολής των 64 δυαδικών ψηφίων

Κάθε μία από αυτές τις κατηγορίες εντολών, διαιρείται σε πέντε μικρότερες κατηγορίες, ανάλογα με τη λειτουργία που επιτελεί η εντολή:

- Εντολές διαχείρισης σταθερών τιμών (constant instructions) για την αποθήκευση σταθερών στο σωρό. Οι εντολές αυτού του τύπου, όπως π.χ. η i32.const, παίρνουν το μοναδικό όρισμά τους και το τοποθετούν στην κορυφή του σωρού του συστήματος (λειτουργία push).
- Εντολές ελέγχου (test instructions) για τον έλεγχο μίας συνθήκης. Μία τέτοια εντολή, παίρνει τον αριθμό που είναι στο επάνω μέρος του σωρού. Εάν είναι 0, τότε τοποθετεί το 1 σε μορφή i32, διαφορετικά τοποθετεί το 0 σε μορφή i32. Υπάρχουν διαθέσιμες μόνο δύο εντολές αυτού του τύπου, η εντολή i32.eqz και η εντολή i64.eqz. Στην ουσία το 0 και το 1 που τοποθετεί στην κορυφή του σωρού, παίζουν το ρόλο του false και του true αντίστοιχα, αφού δεν υπάρχει στη γλώσσα τύπος boolean, και αποθηκεύονται σε μορφή ακεραίου των 32 δυαδικών ψηφίων (δηλαδή i32).
- Εντολές σύγκρισης (comparison instructions) για τη σύγκριση 2 αριθμών. Οι εντολές αυτές παίρνουν 2 ορίσματα από την κορυφή του σωρού και στη συνέχεια βάζουν στο σωρό το αποτέλεσμα της σύγκρισης που είναι 0 ή 1 σε μορφή i32 που αντιστοιχεί στο false και το true. Για τη σύγκριση των ορισμάτων μπορούν να χρησιμοποιηθούν τελεστές σύγκρισης όπως:
    - eq για την ισότητα
    - ne για διάφορο
    - lt για μικρότερο
    - le για μικρότερο ή ίσο
    - gt για μεγαλύτερο
    - ge για μεγαλύτερο ή ίσο

  Στο τέλος αυτών των τελεστών μπορεί να ακολουθήσει το σύμβολο s αν πρόκειται για προσημασμένους αριθμούς ή το σύμβολο u για μη προσημασμένους. Για παράδειγμα με την εντολή i64.ge_s συγκρίνουμε 2 προσημασμένους ακεραίους

των 64 δυαδικών ψηφίων και ελέγχουμε αν ο πρώτος είναι μεγαλύτερος ή ίσος από τον δεύτερο. Αν πράγματι είναι τότε αποθηκεύεται το 1 στο σωρό, διαφορετικά αποθηκεύεται το 0.

- Εντολές ενός ορίσματος (unary arithmetic instructions) για την εκτέλεση πράξεων με έναν αριθμό. Οι εντολές αυτές παίρνουν ένα όρισμα από την κορυφή του σωρού, εκτελούν μία πράξη και στη συνέχεια τοποθετούν πίσω στο σωρό το αποτέλεσμα. Για παράδειγμα, η εντολή f64.neg παίρνει έναν αριθμό κινητής υποδιαστολής 64 δυαδικών ψηφίων από το σωρό, αλλάζει το πρόσημό του και γράφει το αποτέλεσμα στο σωρό.
- Εντολές δύο ορισμάτων (binary arithmetic instructions) για την εκτέλεση αριθμητικών πράξεων ανάμεσα σε 2 αριθμούς. Οι εντολές αυτές παίρνουν 2 ορίσματα από την κορυφή του σωρού, εκτελούν κάποια πράξη ανάμεσά τους και γράφουν πίσω το αποτέλεσμα. Για παράδειγμα, η εντολή f64.sub εκτελεί αφαίρεση ανάμεσα σε 2 αριθμούς κινητής υποδιαστολής μήκους 64 δυαδικών ψηφίων ο καθένας και γράφει τη διαφορά τους πίσω στο σωρό.
- Εντολές μετατροπής (conversion instructions) για την μετατροπή ενός αριθμού σε άλλη μορφή. Οι εντολές αυτές παίρνουν ένα όρισμα από την κορυφή του σωρού, κάνουν την μετατροπή και αποθηκεύουν το αποτέλεσμα και πάλι στο σωρό. Για παράδειγμα, η εντολή i32.wrap_i64 μετατρέπει τον τύπο ενός αριθμού, από ακέραιο των 64 δυαδικών ψηφίων, σε ακέραιο των 32 δυαδικών ψηφίων.

3.6 Τοπικές και καθολικές μεταβλητές

Η γλώσσα WebAssembly επιτρέπει τον ορισμό τόσο τοπικών μεταβλητών, όσο και καθολικών μεταβλητών. Μία τοπική μεταβλητή αποθηκεύεται σε χώρο ανεξάρτητο από τον σωρό αποθήκευσης των ορισμάτων της συνάρτησης. Με τον τρόπο αυτό, μπορούμε να έχουμε τη μόνιμη αποθήκευση κάποιων δεδομένων, που δεν εξαρτάται από τις προσθαφαιρέσεις (λειτουργίες push και pop) που συμβαίνουν στο σωρό. Μοιάζει στη λειτουργία του με τις αρχιτεκτονικές όπου γίνεται χρήση καταχωρητών, με τη διαφορά ότι ο χώρος που μπορεί να χρησιμοποιηθεί είναι κατά πολύ μεγαλύτερος.

Εκτός από τον ορισμό τοπικών μεταβλητών, η γλώσσα επιτρέπει και την δημιουργία ή εισαγωγή καθολικών μεταβλητών, οι οποίες μπορεί να είναι ορατές σε ολόκληρο το module. Οι καθολικές αυτές μεταβλητές, λειτουργούν με τρόπο παρόμοιο με τις καθολικές μεταβλητές στις γλώσσες C και C++ και είναι ορατές σε κάθε συνάρτηση του συγκεκριμένου module. Η πρόσβαση στις μεταβλητές γίνεται μέσω του κατάλληλου δείκτη, ανάλογα με το σημείο δήλωσης της μεταβλητής. Στην Εικόνα 3-5 εμφανίζεται ένα παράδειγμα χρήσης καθολικών μεταβλητών. Στο παράδειγμα αυτό, έχουμε 2 καθολικές μεταβλητές, της g1 και την g2 οι οποίες εισάγονται, με τη βοήθεια της κατάλληλης δήλωσης import, από το module env. Επίσης, δηλώνονται και άλλες 4 καθολικές μεταβλητές ονόματα g3, g4, g5 και g6 αντίστοιχα, για τις οποίες μάλιστα ορίζεται ο τύπος και ορίζεται μία αρχική τιμή. Η μεταβλητή g5 για παράδειγμα, δηλώνεται ότι είναι τύπου κινητής υποδιαστολής με μήκος 32 δυαδικών ψηφίων (τύπος f32) και παίρνει αρχική τιμή ίση με 1,5. Για κάποιες από τις μεταβλητές επιτρέπεται η αλλαγή της τιμής τους (mutable), κάτι που δηλώνεται με το πρόθεμα mut πριν από τον τύπο της μεταβλητής. Οι υπόλοιπες, για τις οποίες δεν χρησιμοποιείται το πρόθεμα mut, θεωρούνται immutable, δηλαδή δεν μπορεί να μεταβληθεί η τιμή τους. Σημειώνεται ότι με τη χρήση 2 διαδοχικών Ελληνικών ερωτηματικών, μπορούμε να προσθέσουμε σχόλια στο πρόγραμμα, όπως φαίνεται και στην Εικόνα 3-5. Για τις τοπικές μεταβλητές, αρκεί η δήλωση του τύπου τους, όπως φαίνεται και στην Εικόνα 3-6.

```
(module
  (import "env" "g1" (global $g1 i32))        ;; immutable
  (import "env" "g2" (global $g2 (mut f32))) ;; mutable

  (global $g3 (mut i32) (i32.const 123)) ;; mutable
  (global $g4 (mut i64) (i64.const 456)) ;; mutable
  (global $g5 f32 (f32.const 1.5))       ;; immutable
  (global $g6 f64 (f64.const 2.5))       ;; immutable

  (func $main
    ;; $g3 = $g1
    (global.get $g1)
    (global.set $g3)
  )
)
```

Εικόνα 3-5: *Καθολικές μεταβλητές στην WebAssembly*

```
(module
  (func $main (param $a i32) (param $b f32)
    (local $c i32)
    (local $d i64)
    (local $e f32)
    (local $f f64)

    ;; $c = $a
    (local.get $a)
    (local.set $c)
  )
)
```

Εικόνα 3-6: *Τοπικές μεταβλητές στην WebAssembly*

Η WebAssembly διαθέτει 5 συνολικά εντολές για τον χειρισμό των μεταβλητών, από τις οποίες οι 2 αναφέρονται στις καθολικές μεταβλητές και οι άλλες 3 σε τοπικές μεταβλητές, όπως φαίνεται πιο κάτω:

- global.get για τη λήψη της τιμής μίας καθολικής μεταβλητής. Η εντολή αποθηκεύει στο σωρό την τιμή που έχει η καθολική μεταβλητή που της δίνεται ως όρισμα. Για παράδειγμα στην Εικόνα 3-5 με την εντολή global.get $g1 αποθηκεύεται στο σωρό η τιμή της καθολικής μεταβλητής g1.
- global.set για τη μεταβολή της τιμής μίας καθολικής μεταβλητής. Η εντολή παίρνει το περιεχόμενο της τιμής στην κορυφή του σωρού και το αποθηκεύει στην καθολική μεταβλητή που της δίνεται ως όρισμα. Για παράδειγμα, στην Εικόνα 3-5 με την εντολή global.set $g3, αποθηκεύεται το περιεχόμενο της κορυφής του σωρού, που προηγουμένως είχε πάρει την τιμή της μεταβλητής g1, στην καθολική μεταβλητή g3. Το περιεχόμενο του σωρού μεταβάλλεται μετά την εκτέλεση της εντολής, αφού κατά την ανάγνωση της κορυφής του σωρού, η αντίστοιχη τιμή αφαιρείται από αυτόν (λειτουργία pop).
- local.get για τη λήψη της τιμής μίας τοπικής μεταβλητής. Λειτουργεί παρόμοια με τη μεταβλητή global.get.
- local.set για τη μεταβολή της τιμής μίας τοπικής μεταβλητής. Λειτουργεί παρόμοια με τη μεταβλητή global.set. Για παράδειγμα, με το ζεύγος των εντολών local.get $a και local.set $c, όπως εμφανίζεται στην Εικόνα 3-6, ανατίθεται στην μεταβλητή c η τιμή της μεταβλητής a, κάτι δηλαδή που σε μία γλώσσα προγραμματισμού υψηλότερου επιπέδου θα γραφόταν με μία μόνο εντολή, ως $c=$a (εντολή ανάθεσης τιμής).
- local.tee για τη μεταβολή της τιμής μίας τοπικής μεταβλητής. Λειτουργεί παρόμοια με την εντολή local.set, με τη διαφορά ότι το περιεχόμενο του σωρού δεν μεταβάλλεται. Δηλαδή, απλά διαβάζεται η τιμή από την κορυφή του σωρού, χωρίς να αφαιρείται από αυτόν.

3.7 Συναρτήσεις

Η γλώσσα WebAssembly επιτρέπει τον ορισμό συναρτήσεων. Μία συνάρτηση έχει ένα όνομα και μπορεί να παίρνει ένα ή περισσότερα ορίσματα. Στην Εικόνα 3-6 έχουμε μία συνάρτηση με το όνομα main, η οποία παίρνει 2 ορίσματα, το a και το b, τύπου i32 και f32 αντίστοιχα.

Η κλήση μίας συνάρτησης μπορεί να γίνει με τη βοήθεια της εντολής call. Η εντολή call παίρνει μία παράμετρο 32 δυαδικών ψηφίων που προσδιορίζει τη συνάρτηση η οποία θα κληθεί. Πριν την εκτέλεση του κώδικα της συνάρτησης, οι τιμές των ορισμάτων της τοποθετούνται στο σωρό, έχοντας χαμηλότερα τις τιμές για τα ορίσματα που εμφανίζονται πρώτα στη σειρά κατά τη δήλωση της συνάρτησης. Μετά το πέρας τα εκτέλεσης του κώδικα της συνάρτησης, τοποθετείται στο σωρό η τιμή που επιστρέφει η συνάρτηση, αν πράγματι επιστρέφει κάτι. Σύμφωνα με τις προδιαγραφές Wasm1.0 δεν μπορεί μία συνάρτηση να επιστρέφει περισσότερες από μία τιμές, κάτι που ενδέχεται να αλλάξει στις επόμενες εκδόσεις της γλώσσας. Στην Εικόνα 3-7 εμφανίζεται ένα παράδειγμα κλήσης μία συνάρτησης με το όνομα max. Η συνάρτηση αυτή παίρνει 2 παραμέτρους τύπου i32 με ονόματα a και b αντίστοιχα. Οι τιμές των ορισμάτων αυτών τοποθετούνται στο σωρό, με την τιμή του b να βρίσκεται στο πάνω μέρος του. Η συνάρτηση επιστρέφει επίσης έναν αριθμό τύπου i32, όπως φαίνεται στη δήλωσή της. Στη συνέχεια η συνάρτηση συγκρίνει τις τιμές των μεταβλητών a και b, θεωρώντας τις μεταβλητές ως προσημασμένες και αν η πρώτη είναι μεγαλύτερη τα δεύτερης, τότε τοποθετεί το 0 στο σωρό (εντολή i32.gt_s). Τέλος, με την εντολή select, διαβάζει 3 τιμές από την κορυφή του σωρού, όπου η πρώτη από αυτές είναι το αποτέλεσμα της σύγκρισης (0 ή 1) και οι άλλες δύο είναι οι τιμές του b και του a αντίστοιχα (με τη σειρά αυτή). Αν στην κορυφή έχουμε 0, τότε επιστρέφεται η τιμή του b, ενώ αν έχουμε 1 επιστρέφεται η τιμή του a. Άρα η συνάρτηση επιστρέφει το b αν το a δεν είναι μεγαλύτερο του b και το a διαφορετικά. Με άλλα λόγια, η συνάρτηση επιστρέφει το μέγιστο των 2 αριθμών, τοποθετώντας το αποτέλεσμα στην κορυφή του σωρού. Το αποτέλεσμα αυτό, δηλαδή ο μεγαλύτερος των 2 αριθμών, επιστρέφεται και από τη συνάρτηση main, αφού όπως φαίνεται στη δήλωσή της, επιστρέφει έναν ακέραιο μεγέθους 32 δυαδικών ψηφίων.

```
(module
  (func $main (export "main") (result i32)
    (call $max (i32.const 20) (i32.const 80))
  )
  (func $max (param $a i32) (param $b i32) (result i32)
    (local.get $a)
    (local.get $b)
    (i32.gt_s (local.get $a) (local.get $b))
    (select)
  )
)
```

Εικόνα 3-7*: Παράδειγμα κλήσης συνάρτησης*

3.8 Διαχείριση μνήμης στην WebAssembly

Το πρότυπο της WebAssembly εκτός από τη χρήση τοπικών και καθολικών μεταβλητών, επιτρέπει τη δυναμική αποθήκευση δεδομένων σε ένα χώρο γραμμικής μνήμης που χρησιμοποιεί. Η γραμμική μνήμη αυτή, λειτουργεί με τρόπο παρόμοιο με την μνήμη άμεσης (τυχαίας) προσπέλασης (μνήμη RAM) που χρησιμοποιούν οι υπολογιστές. Η πρόσβαση στη γραμμική μνήμη γίνεται με τη βοήθεια ειδικών εντολών ανάγνωσης και αποθήκευσης των δεδομένων. Το μήκος της λέξης για τη μνήμη αυτή, δηλαδή η ελάχιστη ποσότητα ανάγνωσης και αποθήκευσης, είναι ίσο με 8 bits, δηλαδή 1 Byte, παρόλο που η γλώσσα υποστηρίζει ορίσματα μήκους 32 και 64 δυαδικών ψηφίων. Μπορεί με άλλα λόγια, να διαχειριστεί και δεδομένα μικρότερου μήκους, μέχρι 8 bits. Το πλήθος των bits που διαβάζονται ή αποθηκεύονται στη μνήμη καθορίζεται ως παράμετρος στις εντολές ανάγνωσης ή εγγραφής.

Σε κάθε module της WebAssembly, μπορεί να οριστεί ή να εισαχθεί (import) ένας χώρος μνήμης. Ο χώρος μνήμης χωρίζεται σε σελίδες (pages), όπου η κάθε σελίδα έχει μέγεθος ίσο με 64 Kbytes. Όταν ορίζεται ένας χώρος μνήμης, θα πρέπει να οριστεί το ελάχιστο πλήθος σελίδων από τις οποίες αποτελείται η μνήμη αυτή. Ο μέγιστος πλήθος σελίδων μπορεί να οριστεί επίσης, αλλά αυτό είναι προαιρετικό. Μπορούν να οριστούν επίσης, τα αρχικά περιεχόμενα της μνήμης. Στην Εικόνα 3-8 εμφανίζεται ένα παράδειγμα ορισμού του χώρου μνήμης από ένα module της WebAssembly. Η μνήμη αυτή αποτελείται από 1

σελίδα κατ' ελάχιστο, ενώ το μέγιστο πλήθος σελίδων είναι ίσο με 8. Στη μνήμη αυτή περιέχεται η συμβολοσειρά "hello", στη θέση που καθορίζεται από την παράμετρο *offset* του ορισμού.

```
(module
  (memory 1 8) ;; { min: 1, max: 8 }
  (data 0 (offset (i32.const 100)) "hello")
  ;; ...
)
```

Εικόνα 3-8: *Ορισμός περιοχής μνήμης στην WebAssembly*

Υπάρχουν 25 διαφορετικές εντολές στη γλώσσα WebAssembly οι οποίες διαχειρίζονται τη μνήμη, χωρισμένες ανά κατηγορίες. Ο Πίνακας 2 εμφανίζει τις κατηγορίες αυτές, μαζί με μία πολύ σύντομη περιγραφή τους.

Πίνακας 2: *Εντολές διαχείρισης της μνήμης*

Εντολή	**Περιγραφή**
memory.size	Τοποθετεί τον αριθμό της μνήμης στην κορυφή του σωρού σαν ακέραιο των 32 bits (i32). Από τη στιγμή που στις προδιαγραφές Wasm1.0 επιτρέπεται να έχουμε μόνο μία μνήμη σε κάθε module, η μοναδική τιμή που μπορεί να έχει το όρισμα της εντολής είναι το 0.
memory.grow	Παίρνει σαν παράμετρο έναν ακέραιο n, που προσδιορίζει το πλήθος των σελίδων κατά το οποίο θα αυξηθεί η διαθέσιμη μνήμη.
load	Με την εντολή αυτή, μπορούμε να φορτώσουμε τα περιεχόμενα από μία θέση της μνήμης στο σωρό του προγράμματος. Η εντολή παίρνει σαν παράμετρο ένα

	offset, το οποίο προστίθεται στη βάση η οποία εξάγεται από το σωρό, δίνοντας έτσι την τελική θέση από όπου διαβάζονται δεδομένα. Ουσιαστικά υπάρχουν 14 παραλλαγές της εντολής load, ανάλογα με τον τύπο και το μέγεθος των δεδομένων που θα διαβαστούν, όπως π,χ, i32.load, i64.load, f32.load κτλ. Τα δεδομένα θεωρούνται ότι είναι αποθηκευμένα σε μορφή little-endian.
store	Η εντολή store αποθηκεύει δεδομένα στη μνήμη και λειτουργεί με τρόπο ανάλογο με την εντολή load. Υπάρχουν συνολικά εννέα παραλλαγές της εντολής store.

Σε ένα τυπικό περιβάλλον εκτέλεσης κώδικα που έχει δημιουργεί κατά την μεταγλώττιση από μία γλώσσα προγραμματισμού, τα δεδομένα και ο κώδικας μοιράζονται τον ίδιο χώρο διευθύνσεων. Έτσι όμως, ένας δείκτης δεδομένων θα μπορούσε να δείξει προς την τον χώρο του κώδικα ή το αντίθετο. Αυτό θα μπορούσε να δημιουργήσει κινδύνους σχετικά με την ασφάλεια του συστήματος. Για το λόγο αυτό το πρότυπο της WebAssembly ορίζει ξεχωριστό χώρο για την αποθήκευση διευθύνσεων που αντιστοιχούν σε κώδικα και ένας τέτοιος χώρος ονομάζεται πίνακας (table). Ένας πίνακας περιέχει τις διευθύνσεις όπου βρίσκονται κάποιες συναρτήσεις, καθώς και τις υπογραφές τους (function signatures), έτσι ώστε να μπορούν να εκτελεστούν με έμμεσο τρόπο. Η WebAssembly κάνει επιβεβαίωση της υπογραφής της συνάρτησης προτού προχωρήσει στην εκτέλεσή της, προσπαθώντας με τον τρόπο αυτό να ανιχνεύσει ή να εμποδίσει μη έγκυρες κλήσεις που συμβαίνουν εξαιτίας κάποιου σφάλματος ή εσκεμμένα για την παραβίαση της ασφάλειας του συστήματος.

Ο χώρος διευθύνσεων, τόσο για τα δεδομένα όσο και για τον κώδικα, δεν ταυτίζεται με τις πραγματικές φυσικές διευθύνσεις που χρησιμοποιούνται. Η πρώτη θέση της μνήμης στην WebAssembly έχει τη διεύθυνση 0 και φτάνει μέχρι την μέγιστη επιτρεπτή τιμή, ανάλογα με το χώρο που έχει κρατηθεί. Η μνήμη χωρίζεται σε σελίδες (pages) των 64 KBytes και το μέγεθος της μνήμης μπορεί να μεταβληθεί δυναμικά. Από την άλλη μεριά, το μέγεθος των πινάκων είναι στατικό, χωρίς να έχουν τα προγράμματα τη δυνατότητα της άμεσης

τροποποίησης του περιεχομένου των πινάκων, έτσι ώστε να μην παραβιάζεται η ασφάλεια του συστήματος.

3.9 Εντολές ελέγχου

Υπάρχουν συνολικά 11 εντολές ελέγχου στη γλώσσα WebAssembly, κάποιες από τις οποίες παρουσιάζονται στην παράγραφο αυτή. Οι εντολές ελέγχου ορίζουν την ροή του προγράμματος που εκτελείται, δίνοντας τη δυνατότητα να εκτελεστούν εντολές επιλογής και εντολές επανάληψης.

Η εντολή block λειτουργεί όπως μία συνάρτηση η οποία δεν έχει ορίσματα. Όπως και οι συναρτήσεις, στο πρότυπο 1.0 μπορεί να επιστρέψει το πολύ ένα αποτέλεσμα. Στην Εικόνα 3-9 εμφανίζεται ένα παράδειγμα χρήσης της εντολής block. Η εντολή αυτή επιτρέπει την ομαδοποίηση των εντολών του προγράμματος σε λογικές ενότητες. Πριν από την εκτέλεση της ομάδας των εντολών που περιέχονται σε μία block, δεν γίνεται τοποθέτηση κάποιων ορισμάτων στο σωρό, όπως γίνεται στις συναρτήσεις. Μετά το τέλος της εκτέλεσης της ομάδας εντολών, τοποθετείται το αποτέλεσμα στην κορυφή του σωρού, εάν επιστρέφεται αποτέλεσμα.

```
(module
  (func (result i32)
    ;; ... other instructions
    (block (result i32)
      (i32.const 100)
    )
  )
)
```

Εικόνα 3-9*: Η εντολή block*

Η WebAssembly δίνει και τη δυνατότητα δημιουργίας εντολών επιλογής, με τη βοήθεια της εντολής if. Ένα παράδειγμα χρήσης της εντολής if φαίνεται στην Εικόνα 3-10. Η

εντολή αυτή παίρνει το περιεχόμενο της κορυφής του σωρού. Εάν αυτό είναι διάφορο του μηδενός, τότε εκτελεί τις εντολές που υπάρχουν στο τμήμα then της εντολής. Αντίθετα, εάν είναι 0 τότε εκτελεί τις εντολές που υπάρχουν στο τμήμα else της εντολές. Η εντολή μπορεί να επιστρέφει και ένα αποτέλεσμα, το οποίο τοποθετείται στην κορυφή του σωρού. Εάν έχουμε μία εντολή if στην οποία δεν υπάρχει τμήμα else, τότε η εντολή δεν μπορεί να επιστρέφει κάτι.

Επιπλέον, η γλώσσα δίνει τη δυνατότητα επαναληπτικής εκτέλεσης μίας ομάδας εντολών, με τη χρήση της εντολής loop. Η εντολή loop λειτουργεί παρόμοια με την εντολή block, με τη διαφορά ότι με τη βοήθεια της εντολής br, που περιγράφεται στη συνέχεια, μπορεί να οριστεί η ροή του προγράμματος.

Με την εντολή br επιτρέπεται η έξοδος της ροής του προγράμματος από μία ομάδα εντολών. Η εντολή λειτουργεί, ανάλογα με τη χρήση της, παρόμοια με την εντολή break ή την εντολή branch, άλλων γλωσσών προγραμματισμού. Η εντολή παίρνει σαν όρισμα έναν μη προσημασμένο ακέραιο αριθμό μήκους 32 δυαδικών ψηφίων (τύπος u32), ο οποίος καθορίζει το πλήθος των εμφωλευμένων ομάδων από τις οποίες θα “δραπετεύσει” η ροή του προγράμματος. Στο παράδειγμα στην Εικόνα 3-11 εμφανίζεται μία εντολή br με παράμετρο 2. Αυτό σημαίνει ότι η ροή του προγράμματος θα “δραπετεύσει” από 2 εμφωλευμένες ομάδες εντολών. Επομένως, ενώ πριν εκτέλεση της εντολής br η στοίβα περιέχει, από επάνω προς τα κάτω, τις σταθερές 123, 300, 200 και 100, μετά την εκτέλεση της εντολής θα περιέχει μόνο τις σταθερές 123 και 300, πήγε δηλαδή 2 βήματα προς τα πίσω.

```
(module
  (func $max (param $a i32) (param $b i32) (result i32)
    (i32.gt_s (local.get $a) (local.get $b))
    (if (result i32)
      (then (local.get $a))
      (else (local.get $b))
    )
  )
)
```

***Εικόνα 3-10**: Η εντολή if*

```
(module
  (func (export "main") (result i32)
    (i32.const 100) (block (result i32)
      (i32.const 200) (block (result i32)
        (i32.const 300) (block (result i32)
          (i32.const 123) (br 2) ;; <---
        ) (i32.add)
      ) (i32.add)
    ) (i32.add)
  )
)
```

Εικόνα 3-11*: Αλλαγή ροής με την εντολή br*

Η εντολή br_if παίρνει από τη στοίβα την τιμή που βρίσκεται στην κορυφή της και στην περίπτωση που αυτή είναι διαφορετική από 0, τότε μεταβαίνει σε ένα άλλο σημείο του προγράμματος. Διαφορετικά, αν η τιμή της είναι 0, τότε η ροή συνεχίζεται κανονικά. Με τον τρόπο αυτό μπορούμε να επιτύχουμε μία δομή επανάληψης, όπως φαίνεται στην Εικόνα 3-12. Η εντολή loop δεν επιτυγχάνει από μόνη της την επανάληψη της εκτέλεσης κάποιων εντολών. Η εντολή br_if όμως του παραδείγματος, οδηγεί την ροή του προγράμματος και πάλι στην αρχή της ομάδας εντολών του loop, υλοποιώντας έτσι έναν βρόχο (loop). Στο παράδειγμα της εικόνας, έχουμε μία συνάρτηση με τι όνομα sum, η οποία υπολογίζεται το άθροισμα των αριθμών από μία αρχική τιμή from μέχρι μία τελική τιμή to και επιστρέφεται το αποτέλεσμα στο σωρό.

```
(module
  (func $sum (param $from i32) (param $to i32) (result i32)
    (local $n i32)

    (loop $l
      ;; $n += $from
      (local.set $n (i32.add (local.get $n) (local.get $from)))
      ;; $from++
      (local.set $from (i32.add (local.get $from) (i32.const 1)))
      ;; if $from <= $to { continue }
      (br_if $l (i32.le_s (local.get $from) (local.get $to)))
    )

    ;; return $n
    (local.get $n)
  )
)
```

Εικόνα 3-12: *Δομή επανάληψης με την εντολή br_if*

3.10 Μορφή κειμένου (WAT)

Όπως αναφέρθηκε στο προηγούμενο κεφάλαιο, ο κώδικας WebAssembly παράγεται αυτόματα με εργαλεία όπως το Emscripten, τα οποία βοηθούν στη μεταγλώττιση του κώδικα που έχει γραφτεί σε κάποια άλλη γλώσσα προγραμματισμού, σε μορφή Wasm. Η μορφή Wasm όμως, είναι σε δυαδική μορφή, βρίσκεται πιο κοντά στο υλικό του υπολογιστή και επομένως, δεν είναι καθόλου "φιλική" προς τον άνθρωπο (προγραμματιστή). Για το λόγο αυτό, υπάρχει η δυνατότητα μορφοποίησης του κώδικα σε μορφή απλού κειμένου, η οποία αναφέρεται και ως μορφή WAT (WebAssembly Text format).

Στην μορφή Wasm οι πληροφορίες οργανώνονται σε τμήματα (sections), τα οποία μπορούν να εμφανίζονται το πολύ μία φορά, σε αύξουσα σειρά ως προς το αναγνωριστικό τους (section id). Αντίθετα, στη μορφή WAT οι πληροφορίες είναι οργανωμένες σε πεδία (fileds), στα οποία αποδίδεται κάποια τιμή. Δεν υπάρχει κάποιος περιορισμός ως προς τη σειρά εμφάνισης των πεδίων, εκτός του ότι τα πεδία import θα πρέπει να εμφανίζονται

πριν από τα πεδία της μορφής function, table, memery και global. Στη συνέχεια, θα παρουσιαστούν κάποια από τα πιο σημαντικά πεδία που μπορούν να χρησιμοποιηθούν στη μορφή WAT.

Με το πεδίο type μπορεί να οριστεί ο τύπος μία συνάρτησης. Για παράδειγμα, στην Εικόνα 3-13, εμφανίζεται ο ορισμός μία συνάρτησης, στην οποία αποδίδεται το αναγνωριστικό $ft1, η οποία λαμβάνει 2 παραμέτρους τύπου i32 και επιστρέφει ως αποτέλεσμα επίσης έναν ακέραιο τύπου i32. Έχουμε επίσης μία δεύτερη συνάρτηση, με αναγνωριστικό $ft2, η οποία παίρνει σαν παράμετρο έναν αριθμό κινητής υποδιαστολής τύπου f64.

```
(module
  (type $ft1 (func (param i32 i32) (result i32)))
  (type $ft2 (func (param f64)))
)
```

Εικόνα 3-13: *Το πεδίο type της μορφής WAT*

Με το πεδίο import μπορούνε εισάγουμε συναρτήσεις, πίνακες, μνήμες και καθολικές μεταβλητές στο συγκεκριμένο module από το περιβάλλον του. Στην Εικόνα 3-14 εμφανίζεται ένα παράδειγμα όπου εισάγεται μία συνάρτηση, ένας πίνακας, μία μνήμη και 2 καθολικές μεταβλητές. Κατά την εισαγωγή ενός στοιχείου, θα πρέπει να δοθεί κάποιο όνομα στο στοιχείο αυτό, μέσα σε διπλά εισαγωγικά. Εκτός από την εισαγωγή, μπορούμε να έχουμε και την εξαγωγή κάποιου στοιχείου, με τη βοήθεια του πεδίου export, όπως φαίνεται στην Εικόνα 3-15. Και πάλι θα πρέπει να αποδίδεται μία ονομασία στο στοιχείο το οποίο εξάγεται, όπως γίνεται και με το πεδίο import.

```
(module
  (type $ft1 (func (param i32 i32) (result i32)))
  (import "env" "f1" (func   $f1 (type $ft1)))
  (import "env" "t1" (table $t 1 8 funcref))
  (import "env" "m1" (memory $m 4 16))
  (import "env" "g1" (global $g1 i32))        ;; immutable
  (import "env" "g2" (global $g2 (mut i32))) (;; mutable ;;)
)
```

Εικόνα 3-14*: Το πεδίο import ης μορφής WAT*

```
(module
  ;; ...
  (export "f1" (func   $f1))
  (export "f2" (func   $f2))
  (export "t1" (table  $t ))
  (export "m1" (memory $m ))
  (export "g1" (global $g1))
  (export "g2" (global $g2))
)
```

Εικόνα 3-15*: Το πεδίο export της μορφής WAT*

Με το πεδίο function μπορεί να προσδιοριστεί μία συνάρτηση, ορίζοντας τις παραμέτρους που παίρνει και τον τύπο τους, καθώς και τον τύπο της τιμής που πιθανόν επιστρέφει. Υπάρχει πεδίο με το όνομα start το οποίο καθορίζει τη συνάρτηση η οποία καλείται πρώτη κατά την εκτέλεση του κώδικα. Με τα πεδία global και local περιγράφουν κάποιες καθολικές ή τοπικές μεταβλητές που ορίζονται στο module. Τα συγκεκριμένα πεδία περιγράφτηκαν ήδη στις προηγούμενες παραγράφους.
Με το πεδίο memory ορίζεται ένας χώρος μνήμης όπου αποθηκεύονται κάποια δεδομένα. Στην Εικόνα 3-16 εμφανίζεται ο ορισμός μίας μνήμης σε ένα module, όπου προστίθεται και κάποια δεδομένα στις θέσεις 100_{16} και 108_{16} της μνήμης αυτής, όπως ορίζεται από την τιμή του offset στο πεδίο data. Επίσης, το πεδίο table επιτρέπει τον ορισμό ενός πίνακα. Στην Εικόνα 3-17 εμφανίζεται ο ορισμός ενός πίνακα που περιέχει δείκτες προς συναρτήσεις (τύπος funcref). Με το πεδίο elem προστίθενται 3 συναρτήσεις στον πίνακα

αυτόν από τη θέση 5 και μετά, αφού η τιμή του offset είναι ίση με 5. Η πρώτη θέση του πίνακα, θεωρείται ότι έχει offset ίσο με 0.

```
(module
  (memory 4 16)
  (data (offset (i32.const 100)) "Hello, ")
  (data (offset (i32.const 108)) "World!\n")
)
```

Εικόνα 3-16*: Το πεδίο memory στη μορφή WAT*

```
(module
  (func $f1) (func $f2) (func $f3)
  (table 10 20 funcref)
  (elem (offset (i32.const 5)) $f1 $f2 $f3)
)
```

Εικόνα 3-17*: Το πεδίο table στη μορφή WAT*

Στο επόμενο κεφάλαιο θα περιγραφεί ο τρόπος εκτέλεσης ενός προγράμματος WebAssembly και θα γίνει μία συγκριτική μελέτη της ταχύτητας φόρτωσης μία σελίδας που έχει κώδικα αυτής της μορφής, σε σχέση με παρόμοιες σελίδες που έχουν κώδικα Javascript.

4 Ανάπτυξη κώδικα WebAssembly και συγκριτική μελέτη

4.1 Δημιουργία και εκτέλεση κώδικα WebAssembly

Η ανάπτυξη του κώδικα WebAssembly έγινε με τη βοήθεια του εργαλείου EmScripten. Το Emscripten, όπως περιγράφτηκε στις προηγούμενες παραγράφους, είναι ένα εργαλείο που επιτρέπει τη μεταγλώττιση κώδικα που έχει δημιουργηθεί σε άλλες γλώσσες προγραμματισμού, όπως η C/C++ και η Rust, σε μορφή Wasm.

Για τη δημιουργία του κώδικα, αρχικά έγινε εγκατάσταση του emsdk (Emscript SDK), που αποτελεί έναν διαχειριστή πακέτων για τα εργαλεία που συνδέονται με το Emscript. Η εγκατάσταση έγινε σε περιβάλλον Linux Mint 19 από τη γραμμή εντολών, δίνοντας την εντολή που ακολουθεί, αλλά η διαδικασία είναι ανάλογη και σε περιβάλλον Windows ή Macintosh:

```
git clone https://github.com/emscripten-core/emsdk.git
```

Στη συνέχεια, μέσα από το emsdk, έγινε εγκατάσταση του Emscripten, με τις πιο κάτω εντολές:

```
cd emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```

Στη συνέχεια είναι εύκολο να δημιουργηθεί κώδικας Wasm, μαζί με την ιστοσελίδα που περιέχει αυτόν τον κώδικα, με αυτόματο τρόπο, μέσω της εντολής:

```
emcc source.c -s WASM=1 -o source.html
```

όπου ως "source.c" βάζουμε το όνομα του αρχείου όπου περιέχεται ο κώδικας C που θέλουμε να μεταγλωττίσουμε και "source.html" η ιστοσελίδα που περιέχει τον κώδικα αυτόν. Με την παράμετρο WASM ορίζουμε ότι επιθυμούμε τη δημιουργία κώδικα WebAssembly που είναι συμβατή με την έκδοση 1 της γλώσσας. Φυσικά αντί για αρχείο

με κώδικα C, μπορούμε να βάλουμε αρχείο με πηγαίο κώδικα σε κάποιες από τις υποστηριζόμενες γλώσσες προγραμματισμού.

Το αποτέλεσμα της εκτέλεσης της εντολής αυτής είναι η δημιουργία των πιο κάτω αρχείων:

- Ένα δυαδικό αρχείο με κώδικα WebAssemly. Το αρχείο έχει το όνομα που προσδιορίζουμε εμείς, με επέκταση αρχείου το .wasm και περιέχει τον συμβολικό κώδικα του προγράμματος σε μορφή Wasm,
- Ένα αρχείο ιστοσελίδας με κώδικα HTML, όπου εμφανίζεται το αποτέλεσμα της εκτέλεσης του κώδικα Wasm.
- Ένα αρχείο javascript με το όνομα που ορίσαμε και επέκταση js. Το αρχείο περιέχει τον κώδικα για τη φόρτωση του Wasm κώδικα στην ιστοσελίδα.

Εάν επιθυμούμε κατά τη μεταγλώττιση να δημιουργηθεί μόνο το αρχείο Wasm με τον κώδικα σε δυαδική μορφή, τότε αρκεί να δώσουμε την εντολή:

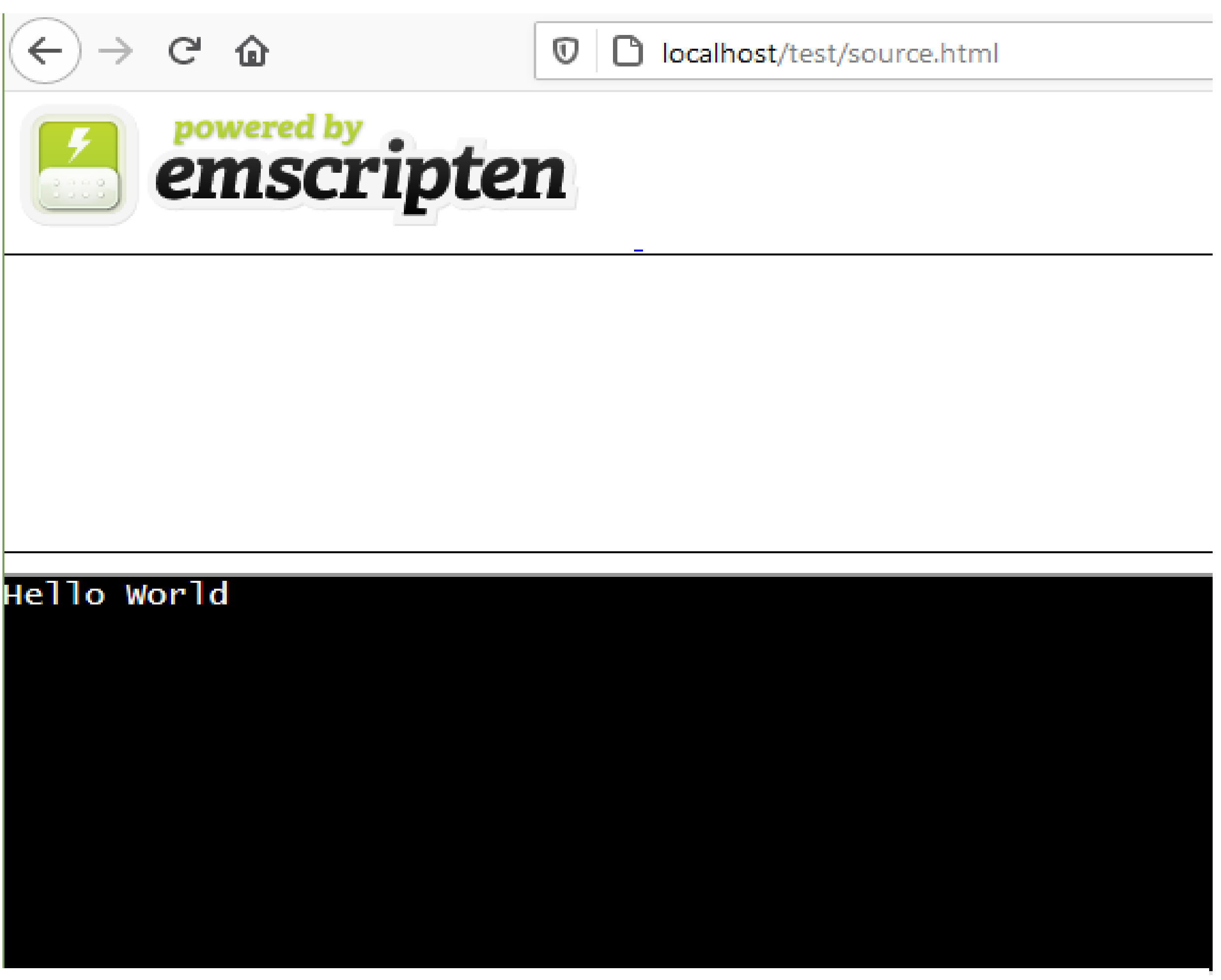
```
emcc source.c or source.cpp -s STANDALONE_WASM
```

Στη συνέχεια θα πρέπει εμείς να φορτώσουμε τον κώδικα σε κάποια ιστοσελίδα με τον δικό μας τρόπο, αντί με τον αυτόματο που δημιουργεί το Emscripten.

Στην *Εικόνα 4-1* εμφανίζεται μία εικόνα από την εμφάνιση μίας ιστοσελίδας που περιέχει τον κώδικα που δημιουργήθηκε με το Emscripten, μετά τη μεταγλώττιση ενός απλού HelloWorld προγράμματος της C, στο οποίο περιέχεται μόνο η εντολή (printf) εμφάνισης του συγκεκριμένου κειμένου. Η σελίδα εμφανίστηκε μέσω ενός Apache web server στον τοπικό υπολογιστή (localhost). Ο κώδικας WebAssembly δεν μπορεί, στην παρούσα έκδοση τουλάχιστον, να φορτωθεί άμεσα στην HTML και αυτό γίνεται με κλήση της συνάρτησης fetch, για τη λήψη του Wasm αρχείου από τον Web server. Αυτός είναι ο λόγος που χρησιμοποιήθηκε ο Apache Web Server. Συνήθως στον κώδικα javascript περιέχονται εντολές της μορφής που φαίνεται πιο κάτω:

```
fetch('source.wasm').then(response =>
  response.arrayBuffer()
).then(bytes => WebAssembly.instantiate(bytes)).then(results => {
  instance = results.instance;
  document.getElementById("container").textContent =
instance.exports.main();
}).catch(console.error);
```

όπου αντί για το όνομα “source.wasm” θα πρέπει να βάλουμε το όνομα του αρχείου που περιέχει τον κώδικα Wasm.



***Εικόνα 4-1**: Δημιουργία προγράμματος HelloWorld με το Emscripten*

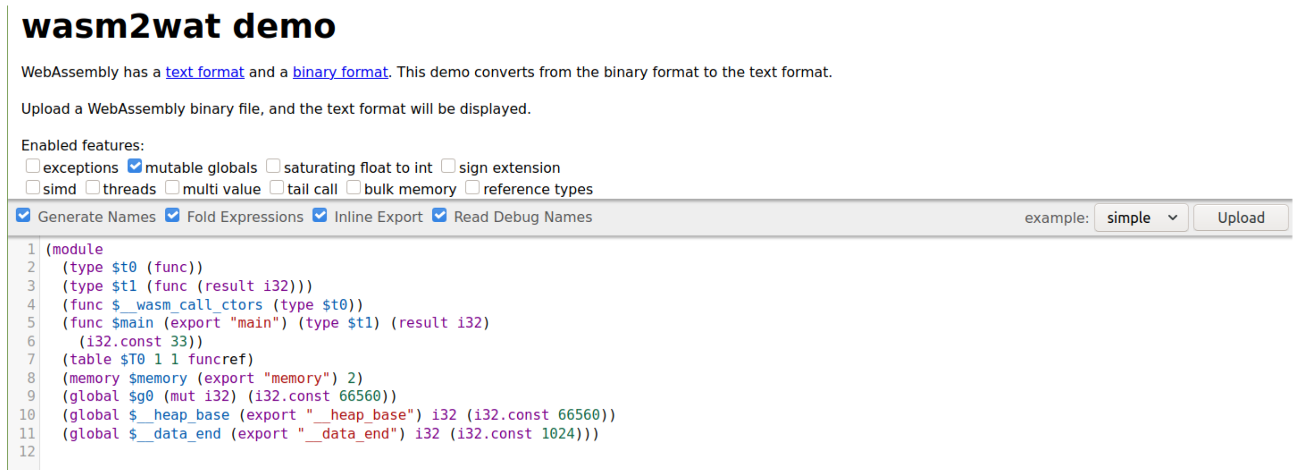
Η δημιουργία και η εκτέλεση κώδικα WebAssembly, μπορεί να γίνει και με online εργαλεία, όπως το WebAssembly Studio που εμφανίζεται στην Εικόνα 4-2. Με το περιβάλλον αυτό, μπορεί να δημιουργηθεί έναν νέο project C ή Rust και στη συνέχεια να γίνει μεταγλώττισή του αυτόματα σε μορφή WebAssembly. Δημιουργούνται μάλιστα αυτόματα και αρχεία Javascript και html, όπου φορτώνεται ο κώδικας και εμφανίζεται το αποτέλεσμά του. Μπορεί επίσης να αποθηκευτούν και τοπικά όλα τα αρχεία που περιέχονται στο project, κάνοντας χρήση της αντίστοιχης λειτουργίας (κουμπάκι “Download”).

Επιπρόσθετα, αν επιθυμούμε να δούμε τη μορφή WAT για τον παραγόμενο δυαδικό κώδικα WebAssembly, υπάρχουν εργαλεία, όπως το εργαλείο wasm2wat που φαίνεται στην Εικόνα 4-3. Το εργαλείο αυτό, επιτρέπει τη φόρτωση ενός αρχείου με κώδικα Wasm και δημιουργεί αυτόματα τη μορφή WAT για τον αντίστοιχο πρόγραμμα.

```
#define WASM_EXPORT __attribute__((visibility("default")))

WASM_EXPORT
int main() {
  int x;
  x=11;
  int y;
  y=22;
  return x+y;
}
```

Εικόνα 4-2: Το περιβάλλον του WebAssembly Studio



Εικόνα 4-3: Μετατροπή Wasm σε WAT μορφή

4.2 Οργάνωση συγκριτικής μελέτης

Στη συνέχεια της έρευνας, έγιναν κάποιες δοκιμές για τον χρόνο που απαιτείται για τη φόρτωση και εμφάνιση των αποτελεσμάτων μίας σελίδας που περιέχει κώδικα Wasm, σε σύγκριση της αντίστοιχης σελίδας που επιτυγχάνει παρόμοια λειτουργικότητα με κώδικα Javascript. Βασικός σκοπός των συγκρίσεων είναι να διαπιστωθεί εάν υπάρχει βελτίωση στο χρόνο εμφάνισης της σελίδας, όταν γίνεται χρήση γλώσσας WebAssembly, και σε τι βαθμό είναι η βελτίωση αυτή.

Για τη σύγκριση των χρόνων φόρτωσης των αντίστοιχων ιστοσελίδων, δημιουργήθηκε μία σειρά προγραμμάτων στη γλώσσα προγραμματισμού C και στη συνέχεια έγινε αυτόματη παραγωγή κώδικα Wasm με το εργαλείο Emscripten και δημιουργήθηκαν οι αντίστοιχες ιστοσελίδες που φορτώνουν τον κώδικα Wasm. Για κάθε μία από τις σελίδες, δημιουργήθηκε έπειτα μία άλλη ιστοσελίδα η οποία περιέχει κώδικα Javascript, με τον οποίο επιτυγχάνονται ακριβώς τα ίδια αποτελέσματα. Με άλλα λόγια, ο κώδικας Javascript είναι ισοδύναμος με τον κώδικα C, μέσω του οποίου δημιουργήθηκε ο κώδικας Wasm της αντίστοιχης ιστοσελίδας. Στη συνέχεια, έγινε μέτρηση των χρόνων που απαιτούνται για τη φόρτωση των αντίστοιχων ιστοσελίδων. Η μέτρηση αυτή του χρόνου έγινε με τη βοήθεια ενός πρόσθετου του προγράμματος πλοήγησης Chrome, με το όνομα Page load time. Το πρόσθετο αυτό μετράει το χρόνο που κάνει να εμφανιστεί η αντίστοιχη σελίδα που εμφανίζεται στο πρόγραμμα πλοήγησης. Η φόρτωση όλων των σελίδων γίνεται τοπικά, από τον Apache Web server που έτρεχε στον τοπικό υπολογιστή. Εξασφαλίζεται επομένως η ισονομία για τη σύγκριση των χρόνων φόρτωσης των αντίστοιχων ιστοσελίδων, ώστε η σύγκριση να είναι δίκαιη.

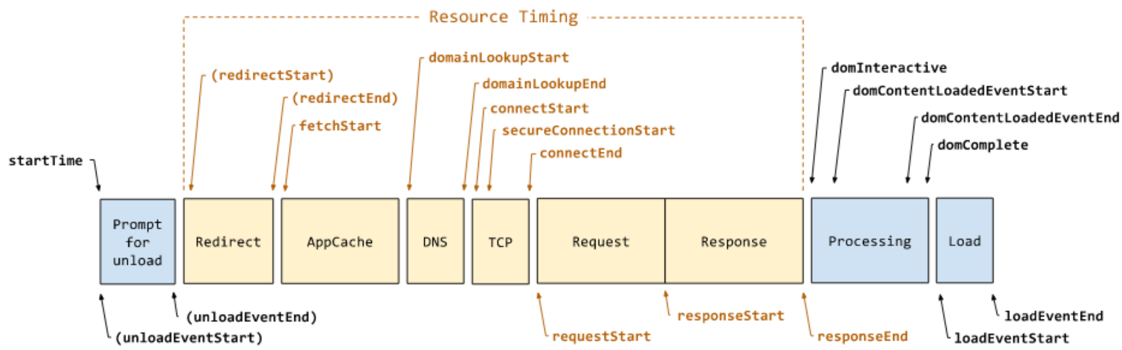
4.2.1 Μέτρηση του χρόνου φόρτωσης μίας σελίδας

Η μέτρηση για τον χρόνο που απαιτείται για τη φόρτωση και εμφάνιση μίας ιστοσελίδας από ένα πρόγραμμα πλοήγησης ή γενικότερα μίας Διαδικτυακής εφαρμογής, αποτελεί ένα πολύ σημαντικό χαρακτηριστικό, επειδή μπορεί να βοηθήσει στη βελτίωση της απόδοσής τους ώστε να γίνουν ταχύτερες. Η Javascript περιλαμβάνει κάποια εργαλεία μέτρησης του χρόνου που απαιτείται για τη φόρτωση των περιεχομένων μίας ιστοσελίδας. Κάνοντας χρήση του αντικειμένου Date της γλώσσας και της μεθόδου getTime() που προσφέρει, μπορεί να μετρηθεί εύκολα ο χρόνος για την φόρτωση μίας ιστοσελίδας, γράφοντας κώδικα της μορφής που φαίνεται στην Εικόνα 4-4. Ο τρόπος βέβαια είναι αρκετά απλοϊκός και δεν μπορεί να δώσει λεπτομέρειες για τις διάφορες φάσεις που είναι απαραίτητες από την αρχή του αιτήματος φόρτωσης μίας ιστοσελίδας, μέχρι να εμφανιστεί το τελικό περιεχόμενο στο πρόγραμμα πλοήγησης.

```
<script type="text/javascript">
var start = new Date().getTime();
function onLoad() {
  var now = new Date().getTime();
  var latency = now - start;
  alert("page loading time: " + latency);
}
```

Εικόνα 4-4*: Μέτρηση χρόνου φόρτωσης μίας ιστοσελίδας με τη Javascript*

Το World Wide Web Consortium (W3C), η κοινότητα που εργάζεται για την ανάπτυξη των προτύπων του Παγκόσμιου Ιστού, έχει εκδώσει από το Σεπτέμβρη του 2020 το Navigation Timing Level 2 editor's draft, που αποτελεί μία διεπαφή για την καταμέτρηση των χρόνων που απαιτούνται στις διάφορες φάσεις φόρτωσης Διαδικτυακού περιεχομένου και αντικαθιστά την προηγούμενη έκδοση, με την ονομασία Navigation-timing (W3C 2020).



***Εικόνα 4-5**: Φάσεις για τη φόρτωση Διαδικτυακού περιεχομένου*

Στην Εικόνα 4-5 εμφανίζονται οι φάσεις για την εμφάνιση Διαδικτυακού περιεχομένου, όπως αυτές ορίζονται από το W3C. Για τις φάσεις όπου ο χρόνος που απαιτείται εξαρτάται από τον server όπου βρίσκεται το περιεχόμενο αυτό, το όνομά τους εμφανίζεται μέσα σε παρένθεση. Παρατηρούμε ότι στάδια που εμφανίζονται εδώ, περιλαμβάνεται ο χρόνος για τυχόν απαλοιφή (unload) του περιεχόμενου της προηγούμενης σελίδας, ο χρόνος για την ανακατεύθυνση (redirect), ο χρόνος για τον έλεγχο στην μνήμη cache της εφαρμογής, όπως και ο χρόνος για την εύρεση της διεύθυνσης IP της σελίδας (DNS). Στη συνέχεια εμφανίζονται οι αντίστοιχοι χρόνοι για την αποστολή του TCP τμήματος και την αποστολή του αιτήματος HTTP και τη λήψη της απάντησης. Στο τέλος εμφανίζεται ο χρόνος για την επεξεργασία του περιεχομένου και τη φόρτωσή του στο πρόγραμμα που είναι υπεύθυνο για την προβολή του περιεχομένου στον τελικό χρήστη.

Το εργαλείο Page load time που επιλέχτηκε, για την καταμέτρηση του χρόνου που απαιτείται για την φόρτωση περιεχομένου που έχει δημιουργηθεί με WebAssembly ή Javascript, κάνει χρήση της διεπαφής Navigation Timing Level 2 και αποτελεί έναν αρκετά αξιόπιστο τρόπο για τη διενέργεια των μετρήσεων.

4.3 Σύγκριση επιδόσεων WebAssembly και Javascript

Για τη σύγκριση των επιδόσεων της γλώσσας WebAssembly, σε σχέση με τις αντίστοιχες επιδόσεις της γλώσσας Javascript, η οποία χρησιμοποιείται ευρέως κατά τον προγραμματισμό ιστοσελίδων, δημιουργήθηκαν κάποια προγράμματα παρόμοιας λειτουργίας και στις 2 γλώσσες και στη συνέχεια καταμετρήθηκαν οι αντίστοιχοι χρόνοι φόρτωσης. Η καταμέτρηση του χρόνου έγινε με το πρόσθετο που παρουσιάστηκε στις προηγούμενη παράγραφο. Ο κώδικας Wasm για κάθε πρόγραμμα, δημιουργήθηκε με την αυτόματη μεταγλώττιση του αρχικού πηγαίου προγράμματος που δημιουργήθηκε σε γλώσσα C. Ένα παράδειγμα τέτοιου προγράμματος φαίνεται στην Εικόνα 4-6, όπου παρουσιάζεται ένα πρόγραμμα υπολογισμού των όρων της ακολουθίας Fibonacci. Κάθε όρος στην ακολουθία αυτή, προκύπτει ως το αποτέλεσμα του αθροίσματος των 2 προηγούμενων όρων. Στο παράδειγμα, υπολογίζεται και επιστρέφεται ο 40^{ος} όρος της ακολουθίας.

```c
int main() {
  return fibonacci(40);
}

int fibonacci(int n) {
  if (n == 0)
    return 1;
  if (n == 1)
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Εικόνα 4-6: *Κώδικας C για την ακολουθία Fibonacci*

```
(module
  (type $t0 (func))
  (type $t1 (func (result i32)))
  (type $t2 (func (param i32) (result i32)))
  (func $__wasm_call_ctors (type $t0))
  (func $main (export "main") (type $t1) (result i32)
    i32.const 40
    call $fibonacci)
  (func $fibonacci (type $t2) (param $p0 i32) (result i32)
    (local $l0 i32)
    i32.const 1
    set_local $l0
    block $B0
      get_local $p0
      i32.const 2
      i32.lt_u
      br_if $B0
      i32.const 1
      set_local $l0
      loop $L1
        get_local $p0
        i32.const -1
```

Εικόνα 4-7*: Μορφή WAT του κώδικα Fibonacci*

Ο κώδικας Webassembly που παράγεται με τη βοήθεια του μεταγλωττιστή Emscripten, εμφανίζεται στην Εικόνα 4-7 σε μορφή κειμένου (WAT). Για τη φόρτωση της δυαδικής μορφής του κώδικα (μορφή Wasm) σε μία ιστοσελίδα, χρησιμοποιήθηκε ο κώδικας Javascript που φαίνεται στην Εικόνα 4-8. Η ιστοσελίδα για την εμφάνιση των αποτελεσμάτων έχει πολύ απλή μορφή και εμφανίζει μόνο το αποτέλεσμα που επιστρέφεται, ώστε να μην χάνεται επιπλέον χρόνος κατά τη φόρτωση της ιστοσελίδας. Ο Κώδικας της ιστοσελίδας αυτής φαίνεται στην Εικόνα 4-9. Τέλος, στην Εικόνα 4-10 εμφανίζεται ο αντίστοιχος κώδικας Javascript για τον υπολογισμό του ίδιου όρους της ακολουθίας Fibonacci.

```javascript
fetch('../out/main.wasm').then(response =>
  response.arrayBuffer()
).then(bytes => WebAssembly.instantiate(bytes)).then(results => {
  instance = results.instance;
  document.getElementById("container").textContent = instance.exports.main();
}).catch(console.error);
```

Εικόνα 4-8*: Κώδικας Javascript για τη φόρτωση του Wasm module*

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <style>
    body {
      background-color: rgb(255, 255, 255);
    }
  </style>
</head>
<body>
  <span id="container"></span>
  <script src="./main.js"></script>
</body>
</html>
```

Εικόνα 4-9*: Κώδικας HTML για την ιστοσελίδα*

```
function fibonacci(n) {
  if (n == 0)
    return 1;
  if (n == 1)
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Εικόνα 4-10: *Κώδικας Javascript για την ακολουθία Fibonacci*

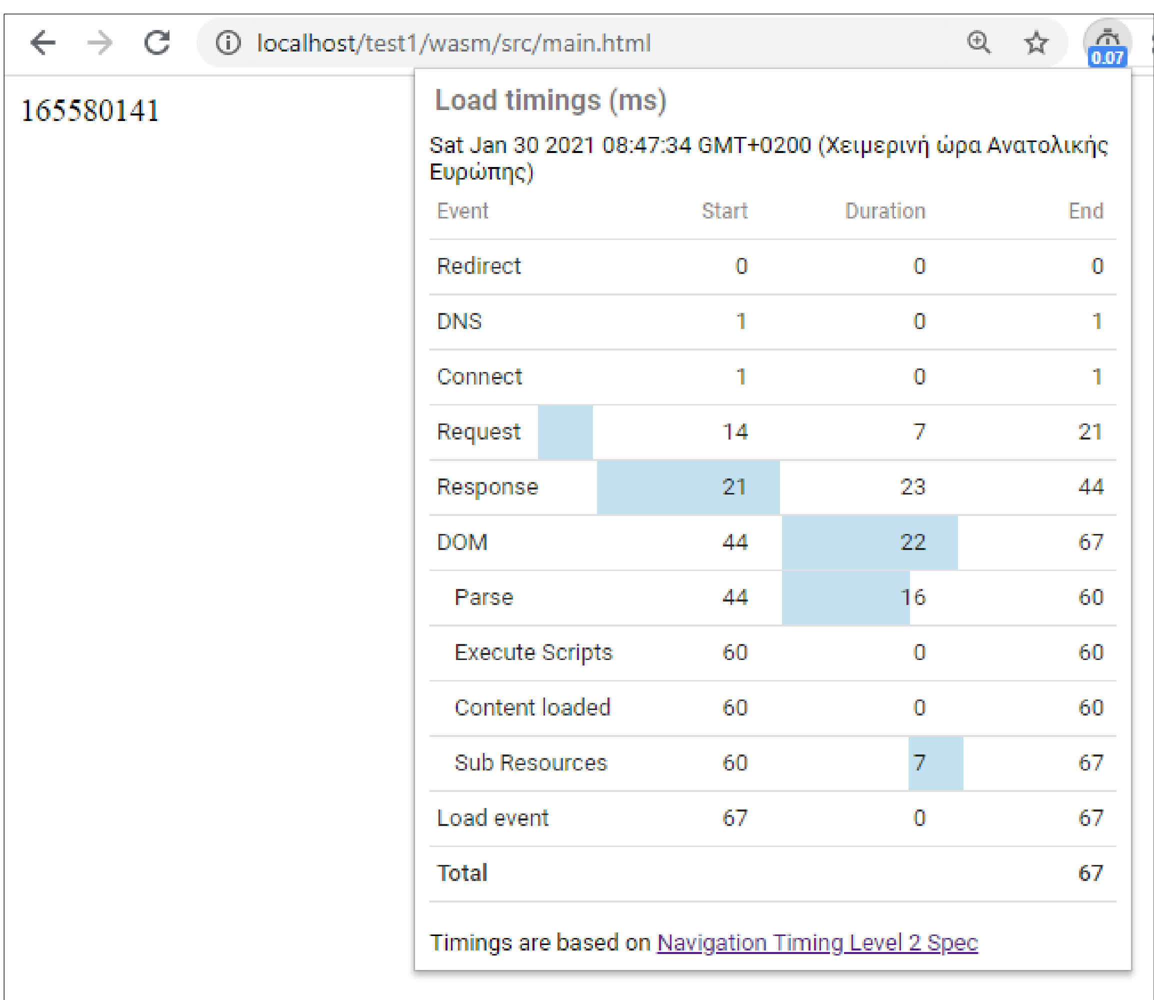
Δημιουργήθηκαν επομένως 2 παρόμοιες ιστοσελίδες, όπου η πρώτη κάνει κάποιους υπολογισμούς σε γλώσσα WebAssembly και η άλλη κάνει τους ίδιους υπολογισμούς σε γλώσσα Javascript. Στη συνέχεια έγινε μέτρηση του χρόνου για τη φόρτωση των 2 ιστοσελίδων που δημιουργήθηκαν. Παρόμοια διαδικασία ακολουθήθηκε και για τα άλλα προγράμματα που δοκιμάστηκαν. Η φόρτωση των ιστοσελίδων έγινε από τον ίδιο τοπικό Apache Web server. Τα αποτελέσματα των μετρήσεων εμφανίζονται στην επόμενη παράγραφο.

4.4 Παρουσίαση αποτελεσμάτων

Στην παράγραφο αυτή θα παρουσιαστούν τα αποτελέσματα των μετρήσεων που αφορούν τους χρόνους φόρτωσης περιεχομένου που έχει αναπτυχθεί με WebAssembly ή Javascript και γίνεται σύγκριση των χρόνων αυτών. Για τον σκοπό αυτό, αναπτύχθηκαν 10 διαφορετικά προγράμματα, τόσο σε WebAssembly όσο και σε Javascript, με παρόμοια λειτουργία, ώστε να γίνει η σύγκριση των επιδόσεών τους. Ο κώδικας WebAssembly, δημιουργήθηκε αυτόματα μετά τη μεταγλώττιση από γλώσσα C με τη βοήθεια του μεταγλωττιστή Emscripten.

Το πρώτο από τα προγράμματα αυτά αφορά τον υπολογισμό του 40^{ου} όρου της ακολουθίας Fibonacci. Στην Εικόνα 4-11 εμφανίζονται τα αποτελέσματα για την μέτρηση των χρόνων

που απαιτούνται για τη φόρτωση της ιστοσελίδας που κάνει χρήση WebAssembly. Όπως φαίνεται και στην εικόνα, η ιστοσελίδα αυτή φορτώθηκα από τον τοπικό Apache Web server. Αντίστοιχα, στην Εικόνα 4-12 εμφανίζονται οι αντίστοιχοι χρόνοι για τη φόρτωση μίας ιστοσελίδας με παρόμοια λειτουργία, η οποία επιτυγχάνεται με την χρήση του κώδικα Javascript (Εικόνα 4-10).

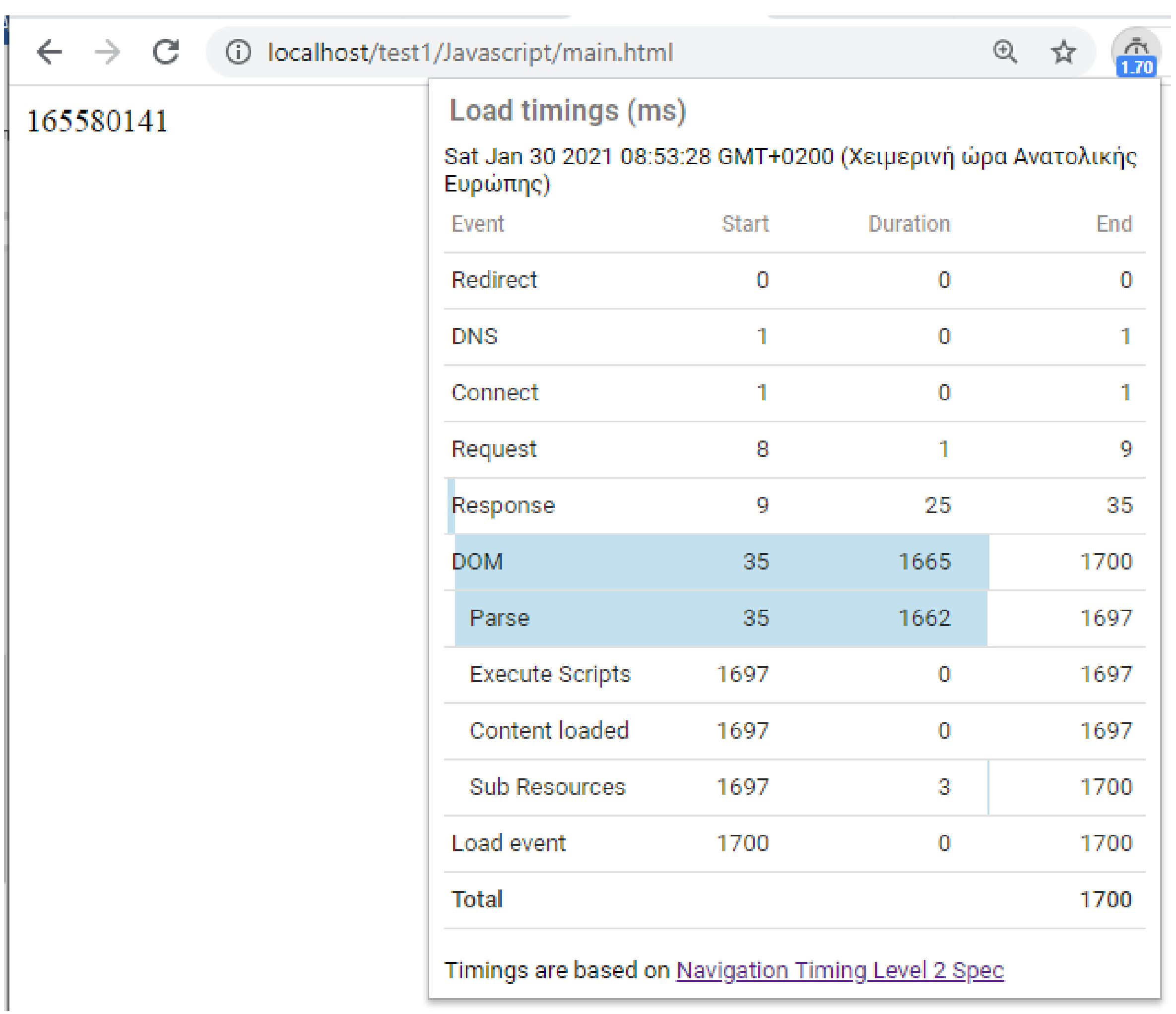


Event	Start	Duration	End
Redirect	0	0	0
DNS	1	0	1
Connect	1	0	1
Request	14	7	21
Response	21	23	44
DOM	44	22	67
Parse	44	16	60
Execute Scripts	60	0	60
Content loaded	60	0	60
Sub Resources	60	7	67
Load event	67	0	67
Total			67

Timings are based on Navigation Timing Level 2 Spec

Εικόνα 4-11: *Καταγραφή χρόνου φόρτωσης για τη σελίδα με κώδικα Wasm*

Όπως φαίνεται και από τις εικόνες, ο χρόνος που απαιτείται στην περίπτωση της WebAssembly είναι αρκετά μικρότερος σε σχέση με τον αντίστοιχο χρόνο της Javascript. Αυτό βέβαια εξαρτάται από το είδος των υπολογισμών που εκτελούνται. Ο Πίνακας 3 εμφανίζει τα συγκριτικά αποτελέσματα, όπως προέκυψαν από τις μετρήσεις αυτές. Στην τελευταία στήλη του πίνακα, εμφανίζεται η επιτάχυνση (speedup), όπως προκύπτει από το λόγο του χρόνου όταν γίνεται χρήση της Javascript, προς τον αντίστοιχο χρόνο για την WebAssembly. Η επιτάχυνση αυτή κυμαίνεται από 1,09 μέχρι 3,68 και η μεγάλη αυτή διακύμανση δείχνει ότι παίζει πολύ μεγάλο ρόλο ο τύπος του περιεχομένου που εμφανίζεται στην ιστοσελίδα και το είδος των υπολογισμών που γίνονται.



***Εικόνα 4-12**: Καταγραφή χρόνου φόρτωσης για τη σελίδα με κώδικα WebAssembly*

***Πίνακας 3**: Εμφάνιση συγκριτικών αποτελεσμάτων*

	Πρόγραμμα	**Javascript**	**Wasm**	**Speedup**
1	Ακολουθία Fibonacci	1,70	0,67	2,54
2	Παραγοντικό	1,46	0,56	2,61
3	Πρόσθεση ακεραίων	0,25	0,23	1,09
4	Πρόσθεση πραγματικών	0,33	0,29	1,14
5	Πολλαπλασιασμός ακεραίων	1,25	0,34	3,68
6	Πολλαπλασιασμός δεκαδικών	1,78	0,64	2,78
7	Ταξινόμηση ακεραίων	1,29	1,02	1,26
8	Ταξινόμηση δεκαδικών	1,78	1,35	1,31
9	Πολλαπλασιασμός πινάκων ακεραίων	2,15	1,43	1,50
10	Πολλαπλασιασμός πινάκων πραγματικών	2,59	1,78	1,46

5 Επίλογος

5.1 Σύνοψη και συμπεράσματα

Η *WebAssembly* αποτελεί μία γλώσσα προγραμματισμού η οποία επιτρέπει την ανάπτυξη κώδικα ο οποίος μπορεί να φορτωθεί και να εκτελεστεί από ένα πρόγραμμα πλοήγησης. Το πρότυπο υποστηρίζεται πλέον σε πάρα πολύ μεγάλο βαθμό από τα πιο διαδεδομένα προγράμματα πλοήγησης του Παγκόσμιου Ιστού. Ο κώδικας WebAssembly παράγεται κατά κανόνα αυτόματα, από τον κώδικα που έχει γραφτεί με τη βοήθεια μίας άλλης γλώσσας προγραμματισμού, με τη βοήθεια ειδικών εργαλείων, όπως ο EmScripten. Με τον τρόπο αυτό μπορεί να αξιοποιηθεί έτοιμος κώδικας, ο οποίος έχει δημιουργηθεί με περισσότερο φιλικές γλώσσες προγραμματισμού και να μεταγλωττιστεί σε μορφή WebAssembly.

Βασικός στόχος του προτύπου είναι η ενσωμάτωση εφαρμογών υψηλών επιδόσεων μέσα σε ιστοσελίδες, με τη δημιουργία κατάλληλου περιβάλλοντος εκτέλεσης (runtime environment – RE). Όμως, το πρότυπο έχει σχεδιαστεί με τέτοιο τρόπο, έτσι ώστε να μπορεί να χρησιμοποιηθεί και για την ανάπτυξη αυτόνομων εφαρμογών. Με το πρότυπο WebAssembly μπορούν να ξεπεραστούν κάποιες εν γένει αδυναμίες που παρουσιάζονται σε κάποιες πολύ διαδεδομένες γλώσσες ανάπτυξης Διαδικτυακών εφαρμογών, όπως η γλώσσα Javascript.

Στην παρούσα εργασία έγινε παρουσίαση του προτύπου WebAssembly και των χαρακτηριστικών του. Έγινε μία ιστορική αναδρομή της πορείας που ακολουθήθηκε μέχρι την πλήρη ανάπτυξη του προτύπου και των προγενέστερων προσπαθειών που έγιναν, από τις μεγάλες εταιρείες του χώρου, για την ανάπτυξη γρήγορων εφαρμογών Διαδικτύου. Περιγράφηκαν επίσης κάποιες από τις σχεδιαστικές αδυναμίες που εντοπίζονται στην γλώσσα Javascript και καλείται να καλύψει το πρότυπο WebAssembly. Επιπλέον, έγινε μία συγκριτική μελέτη των επιδόσεων κάποιου συνόλου προγραμμάτων που αναπτύχθηκαν, τόσο σε WebAssembly, όσο και Javascript, ώστε να συγκριθούν οι αντίστοιχοι χρόνοι.

Από την μελέτη που πραγματοποιήθηκε, μπορούμε να πούμε ότι η WebAssembly μπορεί να αποτελέσει ένα πολύ σημαντικό εργαλείο για την ανάπτυξη γρήγορων Διαδικτυακών εφαρμογών, οι οποίες δεν υστερούν σε ταχύτητα σε σχέση με τις αντίστοιχες native εφαρμογές που εκτελούνται σε ένα υπολογιστικό περιβάλλον. Η εκτέλεση του περιεχομένου που έχει αναπτυχθεί σε WebAssembly, μέσω της μεταγλώττισης από μία άλλη γλώσσα προγραμματισμού, δείχνει να είναι αρκετά πιο γρήγορη σε σχέση με τον αντίστοιχο κώδικα σε Javascript. Το γεγονός αυτό είναι φανερό από τις μετρήσεις που έγιναν και τη συγκριτική μελέτη που πραγματοποιήθηκε στα πλαίσια της παρούσας εργασίας. Η επιτάχυνση που παρατηρείται, φαίνεται να εξαρτάται από ότι φαίνεται, σε πολύ μεγάλο βαθμό από το είδος των προγραμμάτων που αναπτύσσονται και τις επεξεργαστικές απαιτήσεις που έχουν.

6 Βιβλιογραφία

Awsome WASM, https://github.com/mbasso/awesome-wasm#compilers, retrieved 27th October 2020.

Betts A., 2017. *Building Hybrid Applications with Electron*, Slack Engineering, https://slack.engineering/building-hybrid-applications-with-electron/, retrieved 7th October 2020.

Beyer C., 2019. *Electron is Cancer*, https://medium.com/commitlog/electron-is-cancer-b066108e6c32, retrieved 7th October 2020.

CanIUse, 2020. *WebAssembly*, Available: https://caniuse.com/wasm, retrieved: 14th October 2020.

Clark L., 2017. *A crash course in just-in-time (JIT) compilers*, Available: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/, retrieved: 12th October 2020.

Donovan A., Muth R., Chen B. and Sehr D., 2010. *PNaCl : Portable Native Client Executables*, Tech. Rep., 22nd February 2010. Available: https://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf

ECMAScript, 2007. *Proposed ECMAScript 4th Edition – Language Overview*, Available: http://www.ecmascript.org/es4/spec/overview.pdf, retrieved: 8th October 2020.

Emscripten, 2020. *Porting Examples and Demos*, Emscripten GitHub wiki, Available: https://github.com/emscripten-core/emscripten/wiki/Porting-Examples-and-Demos, retrieved: 13th October 2020.

Evans B., 2015. *Java: The Legend*, first edit ed., N. Barber, Ed. O'Reilly Media, Inc.

Grigorik Ilya, 2019. Performance Timeline Level 2. W3C. W3C Working Draft, Available: https://www.w3.org/TR/performance-timeline-2/

Haas A., Rossberg A., Schuff D., Titzer B. L., Holman M., Gohman D., Wagner L., Zakai, A. and Bastien J.F., 2017. *Bringing the Web Up to Speed with WebAssembly,* SIGPLAN Notices. 52 (6): 185–200.

Herman D., Wagner L. and Zakai A., 2014. *asm.js*, Available: http://asmjs.org/spec/latest/, retrieved 10th October 2020.

Krill P., 2008. *JavaScript creator ponders past, future*, InfoWorld, Available: https://www.infoworld.com/article/2653798/javascript-creator-ponders-past--future.html, Retrieved 8th October 2020.

Krill P., 2017. *WebAssembly is now ready for browsers to use*, InfoWorld, 6 March 2017, Avalilable: https://www.infoworld.com/article/3176681/webassembly-is-now-ready-for-browsers-to-use.html, retrieved: 14th October 2020.

Lattner C., 2006. *A cool use of LLVM at Apple: the OpenGL stack*, Available: http://lists.llvm.org/pipermail/llvm-dev/2006-August/006497.html, retrieved: 12th October 2020.

LLVM, 2014. *The LLVM Foundation*, The LLVM Project Blog, LLVM Project News and Details from the Trenches, Available: https://blog.llvm.org/posts/2014-04-03-the-llvm-foundation/, retrieved: 11th October 2020.

Navhoax, 2018. *Introduction to JavaScript*, Available: https://navhoax.com/javascript-tutorial/, Retrieved 12th October 2020

Nelson B., 2017. *Goodbye PNaCl, Hello WebAssembly!* Available: https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html, retrieved 8th October 2020.

Node.js, 2020. *Node.js v13.14.0 Documentation*, https://nodejs.org/docs/latest-v13.x/api/, retrieved 6th October 2020

PYPL, 2020. *PYPL PopularitY of Programming Language: October 2020,* http://pypl.github.io/PYPL.html, retrieved 8th October 2020.

RedMonk, 2020. *The RedMonk Programming Language Rankings: June 2020,* https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/, retrieved 8th October 2020.

Severance C., 2012. *JavaScript: Designing a Language in 10 Days*, *Computer*, vol. 45, no. 2, pp. 7–8, feb 2012. [Online]. Available: https://www.computer.org/csdl/magazine/co/2012/02/mco2012020007/13rRUy08MzA

W3C, 2018. *WebAssembly First Public Working Drafts*, W3C, 15 February 2018. Available: https://www.w3.org/blog/news/archives/6838, retrieved: 14th October 2020.

Wagner L., 2017. *Turbocharging the web*. In: *IEEE Spectrum* 54.12, pp. 48–53. issn: 0018-9235. doi: 10.1109/MSPEC.2017.8118483.

WebAssembly, 2020. *WebAssembly*, Available: https://webassembly.org/, retrieved: 14th October 2020.

WebAssembly Community Group, 2019. Webassembly core specification. *W3C*, Dec 2019. Available: https://www.w3.org/TR/wasm-core-1/) retrieved 29th October 2020

W3C, 2020. *Navigation Timing Level 2 Editor's Draft, Available: https://w3c.github.io/navigation-timing*, retrieved 9th December 2020.

World Wide Web Consortium, 2020. *WebAssembly Core Specification*, World Wide Web Consortium (W3), retrieved 9th June 2020.

XenonStack, 2020. A Beginner's guide to WebAssembly, Mar 26 2020, Available: https://www.xenonstack.com/insights/guide-webassembly/, retrieved on 29th October 2020.

Yee B., Sehr D., Dardyk G., Chen J. B., Muth R., Ormandy T., Okasaka S., Narula N., and Fullagar N., 2009. *Native Client: A Sandbox for Portable, Untrusted x86 Native Code*. In: *2009 30th IEEE Symposium on Security and Privacy*, pp. 79–93. doi: 10.1109/SP.2009.25.

Zakai A., 2011. *Emscripten: an LLVM-to-JavaScript compiler*. *In:* Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 301–312. doi: 10.1145/2048147.2048224.*

Zakai A., 2017. *Why WebAssembly is Faster Than asm.js*. Available: https:// hacks. mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/, retrieved 11th October 2020.