Κεφάλαιο 1

Εισαγωγή

Οι γλώσσες προγραμματισμού είναι σημειογραφίες που περιγράφουν υπολογισμούς στους ανθρώπους και στις μηχανές. Ο κόσμος όπως τον γνωρίζουμε, εξαρτάται από τις γλώσσες προγραμματισμού, διότι όλο το λογισμικό που εκτελείται σε όλους τους υπολογιστές έχει γραφτεί σε κάποια γλώσσα προγραμματισμού. Αλλά, πριν ένα πρόγραμμα είναι έτοιμο για να τρέξει, πρέπει πρώτα να μεταφραστεί σε μια μορφή. η οποία να μπορεί να εκτελεστεί από έναν υπολογιστή.

Τα συστήματα λογισμικού που κάνουν αυτή την μετάφραση ονομάζονται *μεταγλωττιστές*.

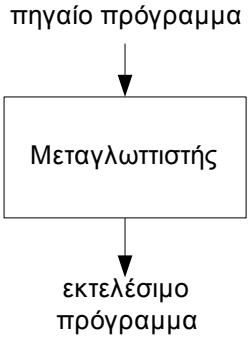
Το βιβλίο αυτό πραγματεύεται το σχεδιασμό και την υλοποίηση μεταγλωττιστών. Θα ανακαλύψουμε ότι μερικές βασικές ιδέες μπορούν να χρησιμοποιηθούν για την κατασκευή μεταφραστών για μια μεγάλη ποικιλία γλωσσών και μηχανών. Εκτός από τους μεταγλωττιστές, οι αρχές και οι τεχνικές για το σχεδιασμό μεταγλωττιστών έχουν εφαρμογή σε πολλά άλλα πεδία εκτός των μεταγλωττιστών, που είναι πιθανό να επαναχρησιμοποιηθούν αρκετές φορές στην καριέρα ενός επιστήμονα υπολογιστών. Η μελέτη για τη συγγραφή ενός μεταγλωττιστή σχετίζεται με γλώσσες προγραμματισμού, αρχιτεκτονικές μηχανών, θεωρία γλωσσών, αλγορίθμους και τεχνολογία λογισμικού.

Σ' αυτό το προκαταρκτικό κεφάλαιο, γίνεται μια εισαγωγή στους διαφορετικούς τύπους γλωσσικών μεταφραστών, δίνεται μια γενική θεώρηση της δομής ενός τυπικού μεταγλωττιστή και συζητούνται οι τάσεις στις γλώσσες προγραμματισμού και στις αρχιτεκτονικές μηχανών που επηρεάζουν τους μεταγλωττιστές. Συμπεριλαμβάνονται μερικές παρατηρήσεις για τη σχέση ανάμεσα στη σχεδίαση μεταγλωττιστών και τη θεωρία της επιστήμης των υπολογιστών και δίνεται μια συνοπτική περιγραφή των εφαρμογών της τεχνολογίας των μεταγλωττιστών εκτός του πεδίου της μεταγλώττισης. Το κεφάλαιο ολοκληρώνεται με μια σύντομη περιγραφή βασικών εννοιών γλωσσών προγραμματισμού που θα χρειαστούν για την μελέτη των μεταγλωττιστών.

1.1 Γλωσσικοί Επεξεργαστές

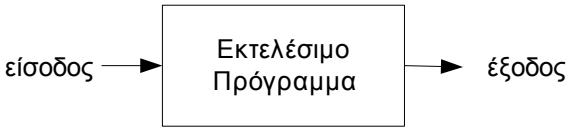
Με απλά λόγια, ένας μεταγλωττιστής είναι ένα πρόγραμμα που μπορεί να διαβάσει ένα πρόγραμμα σε μια γλώσσα – την *πηγαία* γλώσσα – και να το μεταφράσει σε ένα ισοδύναμο πρόγραμμα σε μια άλλη γλώσσα – τη γλώσσα *στόχο*. Δείτε την Εικόνα 1.1. Ένας σημαντικός ρόλος του μεταγλωττιστή είναι η αναφορά λαθών στο πηγαίο

πρόγραμμα τα οποία ανακαλύπτει κατά την διαδικασία της μετάφρασης.



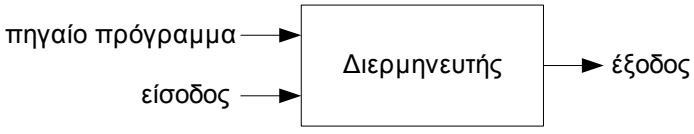
Εικόνα 1.1: Ένας μεταγλωττιστής

Εάν το παραγόμενο πρόγραμμα είναι ένα εκτελέσιμο πρόγραμμα σε γλώσσα μηχανής, μπορεί να εκτελεστεί από το χρήστη για να επεξεργαστεί εισόδους και να παράγει εξόδους. Δείτε την Εικόνα 1.2



Εικόνα 1.2: Τρέχοντας το εκτελέσιμο πρόγραμμα

Ένας *διερμηνευτής* είναι ένας άλλο κοινό είδος γλωσσικού επεξεργαστή. Αντί να παράγει ένα εκτελέσιμο πρόγραμμα ως μετάφραση, ένας διερμηνευτής εμφανίζεται να εκτελεί απ' ευθείας τις λειτουργίες που καθορίζονται από το πηγαίο πρόγραμμα στις εισόδους που δίνονται από το χρήστη, όπως φαίνεται στην Εικόνα 1.3.



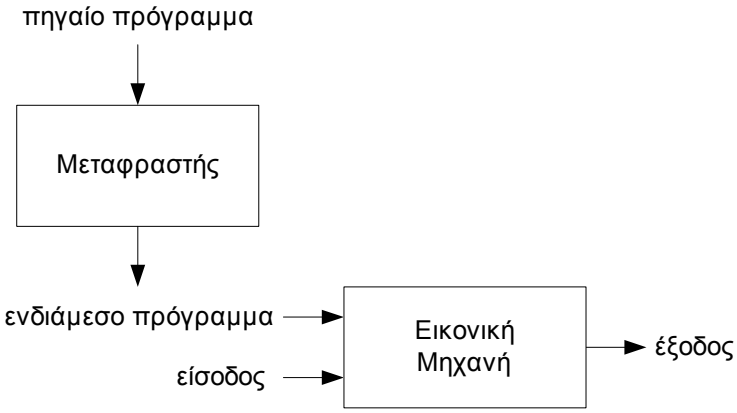
Εικόνα 1.3: Ένας Διερμηνευτής

Η γλώσσα μηχανής του εκτελέσιμου προγράμματος που παράγεται από ένα μεταγλωττιστή, είναι συνήθως πολύ γρηγορότερη από ένα διερμηνευτή σε αντίστοιχες εισόδους και εξόδους. Παρ' όλα αυτά, ένας διερμηνευτής συνήθως μπορεί να δώσει καλύτερα διαγνωστικά λαθών σε σχέση με ένα μεταγλωττιστή, διότι εκτελεί το πηγαίο πρόγραμμα εντολή προς εντολή.

Παράδειγμα 1.1: Οι επεξεργαστές της γλώσσας Java συνδυάζουν μεταγλώττιση και διερμηνεία, όπως φαίνεται στην Εικόνα 1.4. Ένα πηγαίο πρόγραμμα σε Java μπορεί πρώτα να μεταγλωττιστεί σε μια ενδιάμεση μορφή που ονομάζεται *γενική ροή εντολών (bytecodes)*. Η γενική ροή εντολών στη συνέχεια διερμηνεύεται από

μια εικονική μηχανή. Ένα όφελος αυτής της διάταξης είναι ότι η γενική ροή εντολών που μεταγλωττίζεται από μια αρχιτεκτονική μπορεί να διερμηνευθεί από μια άλλη αρχιτεκτονική, που πιθανόν βρίσκεται στην άλλη άκρη του δικτύου.

Προκειμένου να επιτευχθεί γρηγορότερη επεξεργασία των εισόδων σε εξόδους, μερικοί μεταγλωττιστές Java, που ονομάζονται μεταγλωττιστές *δυναμικής μετάφρασης (just-in-time)*, μεταφράζουν πρώτα τη γενική ροή εντολών σε γλώσσα μηχανής, αμέσως πριν τρέξουν το πρόγραμμα για να επεξεργαστούν την είσοδο.

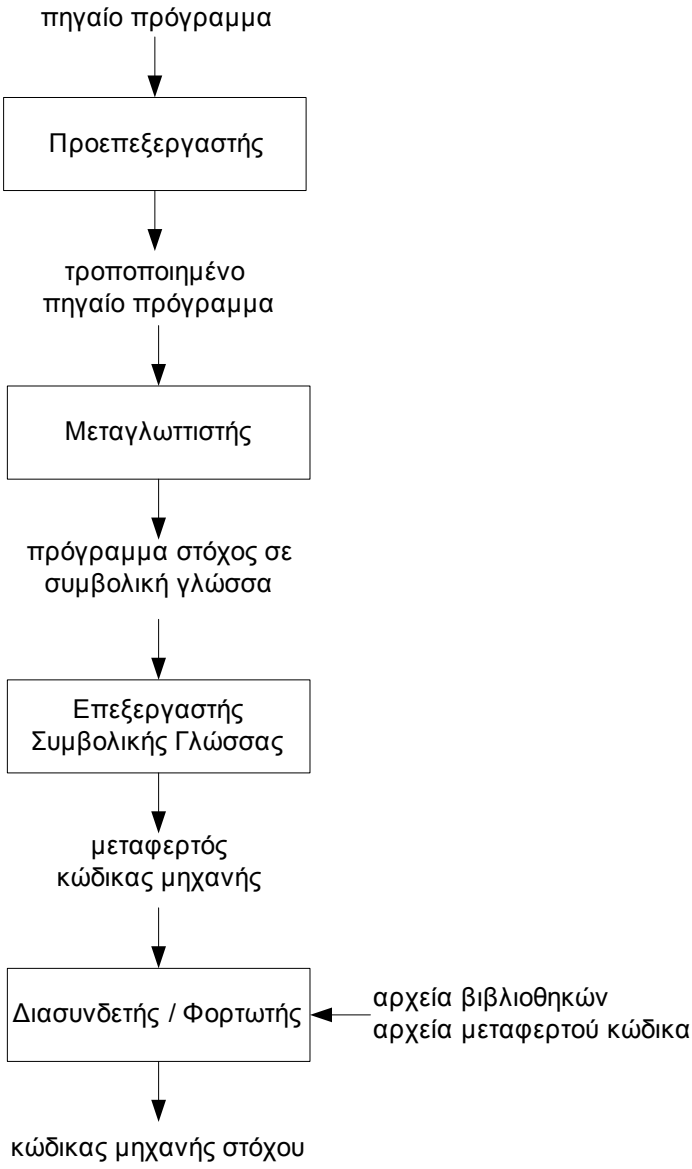


Εικόνα 1.4: Ένας υβριδικός μεταγλωττιστής

Μαζί με το μεταγλωττιστή, μπορεί να απαιτούνται πολλά άλλα προγράμματα για να δημιουργηθεί ένα εκτελέσιμο πρόγραμμα, όπως φαίνεται στην Εικόνα 1.5. Ένα πηγαίο πρόγραμμα μπορεί να διαιρείται σε τμήματα, τα οποία αποθηκεύονται σε ξεχωριστά αρχεία. Η εργασία της σύνθεσης του πηγαίου προγράμματος ανατίθεται μερικές φορές σε ένα ξεχωριστό πρόγραμμα, που ονομάζεται *προεπεξεργαστής*. Ο προεπεξεργαστής μπορεί επίσης να αντικαθιστά συντομογραφίες, που ονομάζονται μακροεντολές, με εντολές της πηγαίας γλώσσας.

Το τροποποιημένο πηγαίο πρόγραμμα στη συνέχεια δίνεται σε ένα μεταγλωττιστή. Ο μεταγλωττιστής μπορεί να παράγει ως έξοδο ένα πρόγραμμα σε συμβολική γλώσσα (assembly-language), διότι η συμβολική γλώσσα και παράγεται ευκολότερα ως έξοδος και αποσφαλματώνεται ευκολότερα σε σχέση με τον κώδικα μηχανής. Η συμβολική γλώσσα στη συνέχεια επεξεργάζεται από ένα πρόγραμμα που ονομάζεται *επεξεργαστής συμβολικής γλώσσας (assembler)* και παράγει ως έξοδο μεταφερτό (relocatable) κώδικα μηχανής.

Τα μεγάλα προγράμματα συχνά μεταγλωττίζονται σε κομμάτια, με αποτέλεσμα ο μεταφερτός κώδικας μηχανής να χρειάζεται να διασυνδεθεί μαζί με άλλα αρχεία μεταφερτού κώδικα (object files) και βιβλιοθηκών για να δημιουργηθεί ο κώδικας που θα τρέξει στην πραγματικότητα στη μηχανή. Ο *διασυνδετής* αντιστοιχίζει διευθύνσεις εξωτερικής μνήμης, σημεία στα οποία ο κώδικας σε ένα αρχείο μπορεί να αναφέρεται σε μια θέση εντός ενός άλλου αρχείου. Κατόπιν ο *φορτωτής* τοποθετεί μαζί όλα τα εκτελέσιμα αρχεία αντικειμένων στη μνήμη για εκτέλεση.



Εικόνα 1.5: Ένα σύστημα επεξεργασίας γλωσσών

1.1.1 Ασκήσεις Ενότητας 1.1

Άσκηση 1.1.1: Ποια είναι η διαφορά μεταξύ ενός μεταγλωττιστή και ενός διερμηνευτή;

Άσκηση 1.1.2: Ποια είναι τα πλεονεκτήματα (α) ενός μεταγλωττιστή έναντι ενός διερμηνευτή (β) ενός διερμηνευτή έναντι ενός μεταγλωττιστή;

Άσκηση 1.1.3: Ποια είναι τα πλεονεκτήματα ενός συστήματος επεξεργασίας γλώσσας στο οποίο ο μεταγλωττιστής παράγει συμβολική γλώσσα αντί για γλώσσα μηχανής;

Άσκηση 1.1.4: Ένας μεταγλωττιστής που μεταφράζει μια υψηλού επιπέδου γλώσσα σε μια άλλη υψηλού επιπέδου γλώσσα ονομάζεται μεταφραστής *πηγαίου-προς-πηγαίου (source-to-source).* Ποια είναι τα πλεονεκτήματα της χρήσης της γλώσσας C ως γλώσσα στόχος ενός μεταγλωττιστή;

Άσκηση 1.1.5: Περιγράψτε μερικές από τις εργασίες που πρέπει να εκτελεί ένας επεξεργαστής συμβολικής γλώσσας (assembler).

1.2 Η δομή ενός Μεταγλωττιστή

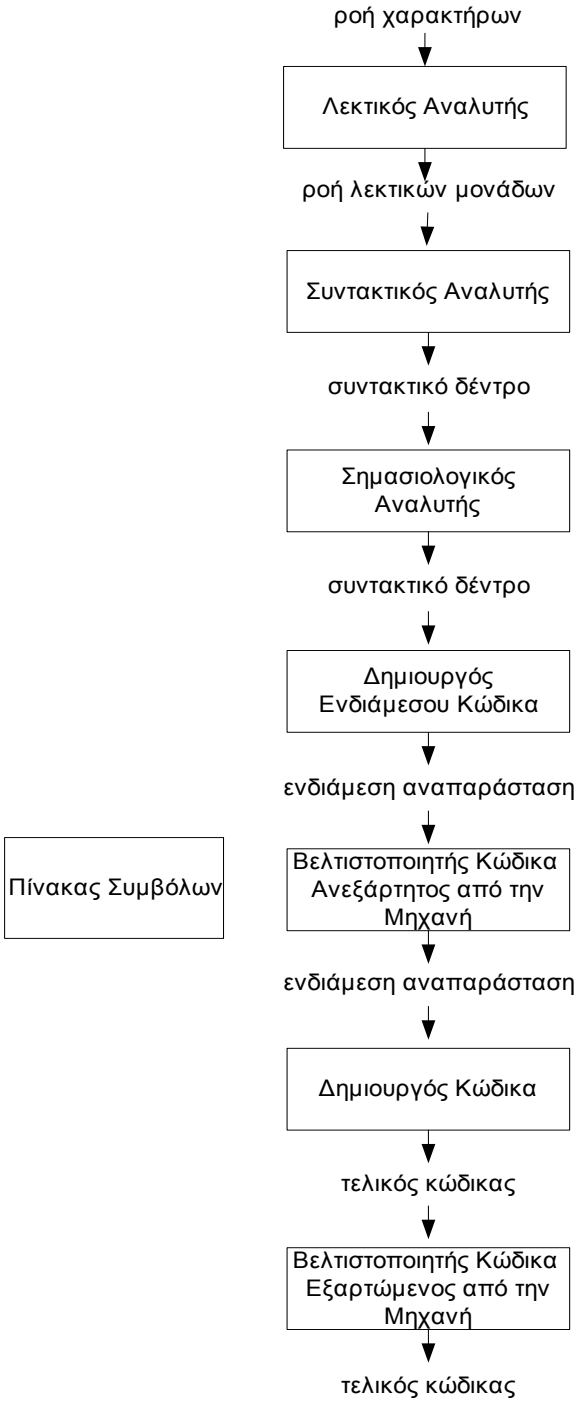
Μέχρι στιγμής έχουμε μεταχειριστεί το μεταγλωττιστή ως ένα κουτί που αντιστοιχίζει ένα πηγαίο πρόγραμμα σε ένα σημασιολογικά ισοδύναμο τελικό πρόγραμμα. Αν ανοίξουμε λίγο αυτό το κουτί, θα δούμε ότι υπάρχουν δύο τμήματα για αυτή την αντιστοίχιση: η ανάλυση και η σύνθεση.

Η διαδικασία της *ανάλυσης* διασπά το πηγαίο πρόγραμμα σε συστατικά κομμάτια και επιβάλλει μια γραμματική δομή σε αυτά. Στη συνέχεια χρησιμοποιεί αυτή τη δομή για να δημιουργήσει μια ενδιάμεση αναπαράσταση του πηγαίου προγράμματος. Εάν η διαδικασία της ανάλυσης εντοπίσει ότι το πηγαίο πρόγραμμα είναι είτε συντακτικά λανθασμένο είτε έχει σημασιολογικά προβλήματα, τότε πρέπει να παρέχει μηνύματα πληροφόρησης, έτσι ώστε ο χρήστης να μπορέσει να κάνει διορθωτικές κινήσεις. Η διαδικασία της ανάλυσης συλλέγει επίσης πληροφορίες για το πηγαίο πρόγραμμα και τις αποθηκεύει σε μια δομή δεδομένων που ονομάζεται *πίνακας συμβόλων*, η οποία διαβιβάζεται μαζί με την ενδιάμεση αναπαράσταση στη διαδικασία της σύνθεσης.

Η διαδικασία της σύνθεσης κατασκευάζει το επιθυμητό τελικό πρόγραμμα χρησιμοποιώντας την ενδιάμεση αναπαράσταση και τις πληροφορίες του πίνακα συμβόλων. Η διαδικασία της ανάλυσης συχνά αποκαλείται *μπροστινή πλευρά (front-end)* του μεταγλωττιστή, ενώ η διαδικασία της σύνθεσης είναι η *πίσω πλευρά (back-end).*

Αν εξετάσουμε τη διαδικασία της μεταγλώττισης με περισσότερη λεπτομέρεια, διαπιστώνουμε ότι λειτουργεί ως μια αλληλουχία *φάσεων*, καθεμιά από τις οποίες μετασχηματίζει μια αναπαράσταση του πηγαίου προγράμματος σε μια άλλη. Μια τυπική διάσπαση ενός μεταγλωττιστή σε φάσεις απεικονίζεται στην Εικόνα 1.6. Στην πράξη, πολλές φάσεις μπορεί να ομαδοποιηθούν και οι ενδιάμεσες αναπαραστάσεις ανάμεσα στις ομαδοποιημένες φάσεις δεν χρειάζεται ρητά να κατασκευάζονται. Ο πίνακας συμβόλων, που αποθηκεύει πληροφορία η οποία αφορά ολόκληρο το πηγαίο πρόγραμμα, χρησιμοποιείται σε όλες τις φάσεις του μεταγλωττιστή.

Μερικοί μεταγλωττιστές έχουν ανάμεσα στην «μπροστινή» και την «πίσω» πλευρά μια φάση βελτιστοποίησης η οποία είναι ανεξάρτητη από τη μηχανή-στόχο.



Εικόνα 1.6: Οι φάσεις ενός μεταγλωττιστή

Ο στόχος αυτής της φάσης βελτιστοποίησης είναι να εκτελέσει μετασχηματισμούς στην ενδιάμεση αναπαράσταση, έτσι ώστε η «πίσω πλευρά» να μπορεί να παράγει ένα καλύτερο τελικό πρόγραμμα συγκριτικά με αυτό που θα παραγόταν από μια μη-βελτιστοποιημένη ενδιάμεση αναπαράσταση. Καθώς η βελτιστοποίηση είναι προαιρετική, κάποια από τις δύο φάσεις βελτιστοποίησης που απεικονίζονται στην Εικόνα 1.6 μπορεί να λείπουν.

1.2.1 Λεκτική Ανάλυση

Η πρώτη φάση ενός μεταγλωττιστή ονομάζεται *λεκτική ανάλυση* ή *σάρωση*. Ο λεκτικός αναλυτής διαβάζει τη ροή των χαρακτήρων που αποτελούν το πηγαίο πρόγραμμα και ομαδοποιεί τους χαρακτήρες σε ακολουθίες με κάποιο νόημα, που αποκαλούνται *λεξήματα*. Για κάθε λέξημα, ο λεκτικός αναλυτής παράγει ως έξοδο ένα *λεκτικό σύμβολο (token)* της μορφής

<*όνομα-λεκτικού συμβόλου, τιμή-ιδιότητας*>

το οποίο περνά στην επόμενη φάση, τη συντακτική ανάλυση. Στο λεκτικό σύμβολο, το πρώτο συστατικό *όνομα-λεκτικού συμβόλου* είναι ένα αφηρημένο σύμβολο που χρησιμοποιείται κατά τη συντακτική ανάλυση και το δεύτερο συστατικό *τιμή-ιδιότητας* δεικτοδοτεί μια καταχώρηση στον πίνακα συμβόλων γι' αυτό το λεκτικό σύμβολο. Η πληροφορία της καταχώρησης του πίνακα συμβόλων χρειάζεται για τη σημασιολογική ανάλυση και την παραγωγή κώδικα.

Για παράδειγμα, υποθέστε ότι το πηγαίο πρόγραμμα περιέχει τη δήλωση ανάθεσης

```
position = initial + rate * 60          (1.1)
```

Οι χαρακτήρες σε αυτή την ανάθεση θα μπορούν να ομαδοποιηθούν στα ακόλουθα λεξήματα και να αντιστοιχηθούν στα ακόλουθα λεκτικά σύμβολα που θα περάσουν στον συντακτικό αναλυτή:

1. Το `position` είναι ένα λέξημα που θα αντιστοιχιζόταν στο λεκτικό σύμβολο <id,1>, όπου το id είναι ένα αφηρημένο σύμβολο που αντιπροσωπεύει το *προσδιοριστικό* ***(id**entifier)* και το 1 δεικτοδοτεί την καταχώρηση του πίνακα συμβόλων για το `position`. Η καταχώρηση του πίνακα συμβόλων για ένα προσδιοριστικό κρατά πληροφορία σχετική με αυτό, όπως το όνομά του και τον τύπο του.
2. Το σύμβολο ανάθεσης = είναι ένα λέξημα που αντιστοιχίζεται στο λεκτικό σύμβολο <=>. Καθώς αυτό το λεκτικό σύμβολο δεν χρειάζεται τιμή-ιδιότητας, έχει παραληφθεί το δεύτερο συστατικό. Θα μπορούσε να χρησιμοποιηθεί οποιοδήποτε αφηρημένο σύμβολο όπως assign για όνομα-υποκατάστατου, αλλά για σημειογραφική ευκολία επιλέχθηκε να χρησιμοποιηθεί το ίδιο το λέξημα ως το όνομα του αφηρημένου συμβόλου.
3. Το λέξημα `initial` που αντιστοιχίζεται στο λεκτικό σύμβολο <id,2>, όπου το 2 δεικτοδοτεί την καταχώρηση του πίνακα συμβόλων για το `initial`.

4. Το λέξημα + που αντιστοιχίζεται στο υποκατάστατο <+>.
5. Το λέξημα `rate` που αντιστοιχίζεται στο λεκτικό σύμβολο <id,3>, όπου το 3 δεικτοδοτεί την καταχώρηση του πίνακα συμβόλων για το `rate`.
6. Το `*` είναι ένα λέξημα που αντιστοιχίζεται στο λεκτικό σύμβολο <*>.
7. Το `60` είναι ένα λέξημα που αντιστοιχίζεται στο λεκτικό σύμβολο <60>.[1]

Τα κενά που χωρίζουν τα λεξήματα θα απορρίπτονταν από τον λεξικό αναλυτή.

Η Εικόνα 1.7 δείχνει την αναπαράσταση της δήλωσης ανάθεσης (1.1) μετά την λεκτική ανάλυση ως μια ακολουθία από λεκτικά σύμβολα

<id,1> <=> <id,2> <+> <id,3> <*> <60> (1.2)

Σε αυτή την αναπαράσταση, τα ονόματα των υποκατάστατων =, +, και * είναι τα αντίστοιχα λεκτικά σύμβολα για τους τελεστές ανάθεσης, πρόσθεσης και πολλαπλασιασμού.

1.2.2 Συντακτική Ανάλυση

Η δεύτερη φάση ενός μεταγλωττιστή είναι η *συντακτική ανάλυση* ή *γραμματική ανάλυση*. Ο συντακτικός αναλυτής χρησιμοποιεί τα πρώτα συστατικά στοιχεία των λεκτικών συμβόλων που παρήγαγε ο λεκτικός αναλυτής για να δημιουργήσει μια δενδρικού τύπου ενδιάμεση αναπαράσταση που απεικονίζει την γραμματική δομή της ροής των λεκτικών συμβόλων. Μια συνηθισμένη αναπαράσταση είναι το *συντακτικό δέντρο* στο οποίο κάθε εσωτερικός κόμβος αναπαριστά μια πράξη και τα παιδιά αυτού του κόμβου αναπαριστούν τα ορίσματα της πράξης. Ένα συντακτικό δέντρο για την ροή λεκτικών συμβόλων (1.2) απεικονίζεται ως έξοδος του συντακτικού αναλυτή στην Εικόνα 1.7.

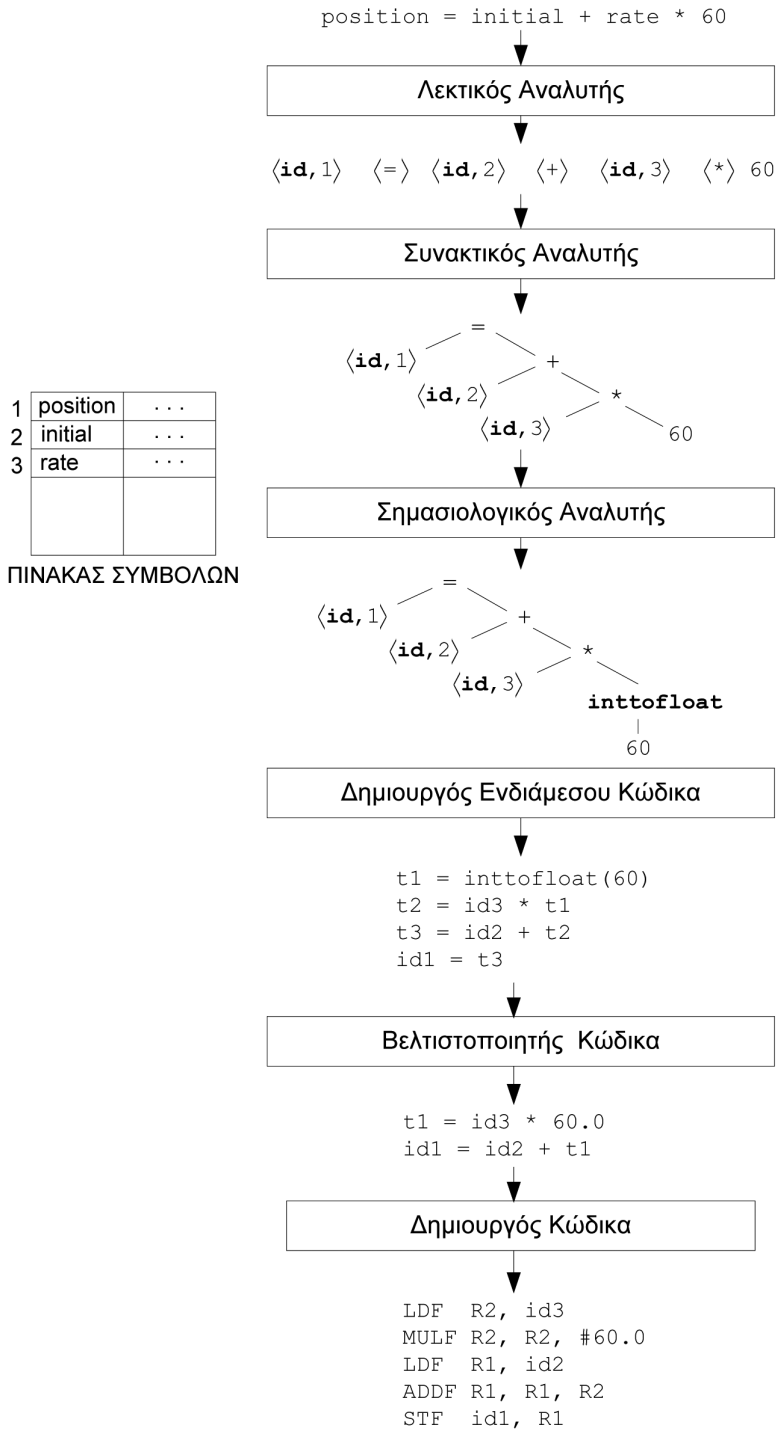
Αυτό το δέντρο δείχνει τη σειρά με την οποία θα εκτελεστούν οι πράξεις στην ανάθεση

```
position = initial + rate * 60
```

Το δέντρο έχει έναν εσωτερικό κόμβο που χαρακτηρίζεται με * με αριστερό παιδί το <id,3> και δεξιό παιδί τον ακέραιο 60. Ο κόμβος <id,3> αναπαριστά το προσδιοριστικό `rate`. Ο κόμβος με το * δηλώνει ρητά ότι πρέπει πρώτα να πολλαπλασιαστεί η τιμή του `rate` με το 60. Ο κόμβος με το + δηλώνει ότι πρέπει να προστεθεί το αποτέλεσμα του πολλαπλασιασμού στην τιμή του `initial`. Η ρίζα του δέντρου, =, δηλώνει ότι το αποτέλεσμα της πρόσθεσης πρέπει να αποθηκευτεί στην τοποθεσία του προσδιοριστικού `position`. Αυτή η διάταξη των πράξεων είναι συνεπής με τις συνήθεις συμβάσεις της αριθμητικής που μας λένε ότι ο πολλαπλασιασμός έχει υψηλότερη προτεραιότητα σε σχέση με την πρόσθεση, άρα πρέπει να εκτελείται πριν από την πρόσθεση.

---

[1] Τεχνικά, για το λέξημα 60 θα έπρεπε να δημιουργηθεί ένα λεκτικό σύμβολο όπως το <number,4>, όπου το 4 δείχνει στον πίνακα συμβόλων για την εσωτερική αναπαράσταση του ακέραιου 60 αλλά η συζήτηση για λεκτικό σύμβολο αριθμών μετατίθεται στο Κεφάλαιο 2. Το Κεφάλαιο 3 πραγματεύεται τεχνικές για την δημιουργία λεκτικών αναλυτών.



Εικόνα 1.7: Μετάφραση μιας δήλωσης ανάθεσης

Οι επόμενες φάσεις του μεταγλωττιστή χρησιμοποιούν τη γραμματική δομή για να βοηθηθούν στην ανάλυση του πηγαίου προγράμματος και να παράγουν το τελικό πρόγραμμα. Στο Κεφάλαιο 4 θα χρησιμοποιήσουμε γραμματικές χωρίς συμφραζόμενα (context-free grammars) για να καθορίσουμε τη γραμματική δομή των γλωσσών προγραμματισμού και θα συζητήσουμε αλγορίθμους για την αυτόματη δημιουργία αποδοτικών συντακτικών αναλυτών για συγκεκριμένες κατηγορίες γραμματικών. Στα Κεφάλαια 2 και 5 θα δούμε ότι ορισμοί καθοδηγούμενοι από συντακτικό (sybtax-directed definitions) μπορεί να βοηθήσουν στον καθορισμό της μετάφρασης των δομών των γλωσσών προγραμματισμού.

1.2.3 Σημασιολογική Ανάλυση

Ο *σημασιολογικός αναλυτής* χρησιμοποιεί το συντακτικό δέντρο και τις πληροφορίες του πίνακα συμβόλων για να ελέγξει αν το πηγαίο πρόγραμμα είναι σημασιολογικά συνεπές με τον ορισμό της γλώσσας. Επίσης συλλέγει πληροφορίες τύπων και τις αποθηκεύει είτε στο συντακτικό δέντρο είτε στον πίνακα συμβόλων, για μελλοντική χρήση κατά τη διάρκεια δημιουργίας του ενδιάμεσου κώδικα.

Ένα σημαντικό μέρος της σημασιολογικής ανάλυσης είναι ο *έλεγχος τύπων*, όπου ο μεταγλωττιστής ελέγχει ότι κάθε τελεστής έχει ταιριαστούς τελεστέους. Για παράδειγμα, πολλοί ορισμοί γλωσσών προγραμματισμού απαιτούν ο δείκτης ενός πίνακα να είναι ακέραιος˙ ο μεταγλωττιστής πρέπει να αναφέρει λάθος, αν ένας αριθμός κινητής υποδιαστολής χρησιμοποιείται ως δείκτης σε έναν πίνακα.

Οι προδιαγραφές της γλώσσας μπορεί να επιτρέπουν μερικές μετατροπές τύπων που ονομάζονται *υποχρεωτικές μετατροπές (coercions).* Για παράδειγμα, ένας δυαδικός αριθμητικός τελεστής μπορεί να εφαρμοστεί είτε σε ένα ζεύγος ακεραίων είτε σε ένα ζεύγος αριθμών κινητής υποδιαστολής. Αν ο τελεστής εφαρμοστεί σε έναν αριθμό κινητής υποδιαστολής και σε έναν ακέραιο, ο μεταγλωττιστής μπορεί να μετατρέψει ή να *μετατρέψει υποχρεωτικά* τον ακέραιο σε έναν αριθμό κινητής υποδιαστολής.

Μια τέτοια υποχρεωτική μετατροπή εμφανίζεται στην Εικόνα 1.7. Υποθέστε ότι τα `position`, `initial`, και `rate` έχουν δηλωθεί ως αριθμοί κινητής υποδιαστολής, και ότι το λέξημα `60` από μόνο του είναι ακέραιος. Ο ελεγκτής τύπων του σημασιολογικού αναλυτή στην Εικόνα 1.7 ανακαλύπτει ότι ο τελεστής * εφαρμόζεται στον αριθμό κινητής υποδιαστολής `rate` και στον ακέραιο `60`. Σ' αυτή την περίπτωση ο ακέραιος μπορεί να μετατραπεί σε έναν αριθμό κινητής υποδιαστολής. Στην Εικόνα 1.7, παρατηρήστε ότι η έξοδος του σημασιολογικού αναλυτή έχει έναν επιπλέον κόμβο για τον τελεστή inttofloat, που μετατρέπει το ακέραιο όρισμα του σε έναν αριθμό κινητής υποδιαστολής. Ο έλεγχος τύπων και η σημασιολογική ανάλυση παρουσιάζονται στο Κεφάλαιο 6.

1.2.4 Παραγωγή Ενδιάμεσου Κώδικα

Κατά τη διαδικασία μετάφρασης ενός πηγαίου προγράμματος σε τελικό κώδικα, ένας μεταγλωττιστής μπορεί να κατασκευάσει μια ή περισσότερες ενδιάμεσες αναπαραστάσεις, οι οποίες μπορεί να έχουν μια ποικιλία μορφών. Τα συντακτικά δέντρα είναι μια μορφή ενδιάμεσης αναπαράστασης τα οποία χρησιμοποιούνται συνήθως κατά τη διάρκεια της συντακτικής και σημασιολογικής ανάλυσης.

Μετά τη συντακτική και τη σημασιολογική ανάλυση, πολλοί μεταγλωττιστές παράγουν μια συγκεκριμένη ενδιάμεση αναπαράσταση χαμηλού-επιπέδου που μπορεί να θεωρηθεί ότι προσομοιάζει έναν τύπο γλώσσας μηχανής, την οποία μπορούμε να φανταστούμε ως ένα πρόγραμμα για μια αφηρημένη μηχανή. Αυτή η ενδιάμεση αναπαράσταση πρέπει να έχει δύο σημαντικές ιδιότητες: πρέπει να είναι εύκολο να παραχθεί και πρέπει να είναι εύκολο να μεταφραστεί στη γλώσσα της μηχανής στόχου.

Στο Κεφάλαιο 6, θα μελετήσουμε μια ενδιάμεση μορφή που ονομάζεται *κώδικας τριών-διευθύνσεων*, η οποία αποτελείται από μια ακολουθία εντολών τύπου συμβολικού κώδικα με τρεις τελεστέους ανά εντολή. Κάθε τελεστέος μπορεί να δρα ως καταχωρητής.

Η έξοδος της γεννήτριας ενδιάμεσου κώδικα της Εικόνας 1.7 αποτελείται από την ακολουθία εντολών τριών-διευθύνσεων

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```
(1.3)

Υπάρχουν αρκετά σημεία που αξίζει να επισημανθούν για τις εντολές τριών-διευθύνσεων. Πρώτον, κάθε εντολή ανάθεσης τριών-διευθύνσεων έχει το πολύ έναν τελεστή στο δεξιό της τμήμα. Έτσι, αυτές οι εντολές καθορίζουν τη σειρά με την οποία πρέπει να γίνουν οι πράξεις˙ ο πολλαπλασιασμός προηγείται της πρόσθεσης στο πηγαίο πρόγραμμα (1.1). Δεύτερον, ο μεταγλωττιστής πρέπει να δημιουργήσει ένα προσωρινό όνομα για να κρατήσει την τιμή που υπολογίζεται από μια εντολή τριών-διευθύνσεων. Τρίτον, μερικές «εντολές τριών διευθύνσεων» όπως η πρώτη και η τελευταία στην παραπάνω ακολουθία (1.3) έχουν λιγότερους από τρεις τελεστέους.

Στο Κεφάλαιο 6, καλύπτονται οι κυριότερες ενδιάμεσες αναπαραστάσεις που χρησιμοποιούνται στους μεταγλωττιστές. Το Κεφάλαιο 5 εισάγει τεχνικές για συντακτικά κατευθυνόμενη μετάφραση οι οποίες εφαρμόζονται στο Κεφάλαιο 6 για έλεγχο τύπων και παραγωγή ενδιάμεσου κώδικα για κλασικές δομές γλωσσών προγραμματισμού όπως οι εκφράσεις, δομές ροής ελέγχου και κλήσεων διαδικασιών.

1.2.5 Βελτιστοποίηση Κώδικα

Η φάση βελτιστοποίησης κώδικα ανεξαρτήτως μηχανής προσπαθεί να βελτιώσει τον ενδιάμεσο κώδικα, έτσι ώστε να προκύψει καλύτερος τελικός κώδικας. Συνήθως καλύτερος σημαίνει γρηγορότερος, αλλά μπορεί να είναι επιθυμητοί και άλλοι στόχοι, όπως μικρότερος σε μέγεθος κώδικας ή τελικός κώδικας που καταναλώνει λιγότερη ενέργεια. Για παράδειγμα, ένας προφανής απλός αλγόριθμος παράγει τον ενδιάμεσο κώδικα (1.3), χρησιμοποιώντας μια εντολή για κάθε τελεστή στη δενδρική αναπαράσταση που προκύπτει από τον σημασιολογικό αναλυτή.

Ένας απλός αλγόριθμος παραγωγής ενδιάμεσου κώδικα ακολουθούμενος από βελτιστοποίηση κώδικα είναι ένας λογικός τρόπος για να παραχθεί καλός τελικός κώδικας. Ο βελτιστοποιητής μπορεί να συμπεράνει ότι η μετατροπή του 60 από

ακέραιο σε αριθμό κινητής υποδιαστολής μπορεί να γίνει μια για πάντα κατά το χρόνο μεταγλώττισης, έτσι η πράξη inttofloat μπορεί να εξαλειφθεί αντικαθιστώντας τον ακέραιο 60 με τον αριθμό κινητής υποδιαστολής 60.0. Επιπλέον, επειδή το `t3` χρησιμοποιείται μόνο μια φορά για να μεταδώσει την τιμή του στο `id1`, ο βελτιστοποιητής μπορεί να μετατρέψει το (1.3) σε μια μικρότερη ακολουθία εντολών

```
t1 = id3 * 60.0
id1 = id2 + t1
```
(1.4)

Υπάρχει μεγάλη ποικιλία στο ποσό της βελτιστοποίησης κώδικα που μπορούν να πραγματοποιήσουν διαφορετικοί μεταγλωττιστές. Σ' αυτούς που κάνουν τα περισσότερα, οι αποκαλούμενοι «*μεταγλωττιστές με δυνατότητα βελτιστοποίησης*», ξοδεύεται σημαντικός χρόνος σ' αυτή τη φάση. Υπάρχουν απλές βελτιστοποιήσεις που μπορεί να μειώσουν σημαντικά το χρόνο εκτέλεσης του τελικού προγράμματος χωρίς να καθυστερούν σημαντικά τη μεταγλώττιση. Τα Κεφάλαια από το 8 και μετά πραγματεύονται με λεπτομέρεια βελτιστοποιήσεις που δεν εξαρτώνται από τη μηχανή και βελτιστοποιήσεις που εξαρτώνται από τη μηχανή.

1.2.6 Παραγωγή Κώδικα

Ο παραγωγός κώδικα λαμβάνει ως είσοδο μια ενδιάμεση αναπαράσταση του πηγαίου προγράμματος και την αντιστοιχίζει στη γλώσσα στόχο. Αν η γλώσσα στόχος είναι κώδικας μηχανής, τότε επιλέγονται καταχωρητές ή τοποθεσίες μνήμης για κάθε μια από τις μεταβλητές που χρησιμοποιούνται από το πρόγραμμα. Στη συνέχεια οι ενδιάμεσες εντολές μεταφράζονται σε ακολουθίες εντολών μηχανής που εκτελούν τη ίδια εργασία. Ένα σημαντικό θέμα της παραγωγής κώδικα είναι η συνετή ανάθεση καταχωρητών που κρατούν μεταβλητές.

Για παράδειγμα, χρησιμοποιώντας τους καταχωρητές `R1` και `R2`, ο ενδιάμεσος κώδικας (1.4) θα μπορούσε να μεταφραστεί στον παρακάτω κώδικα μηχανής

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```
(1.5)

Ο πρώτος τελεστέος σε κάθε εντολή καθορίζει έναν προορισμό. Το `F` σε κάθε εντολή μας πληροφορεί ότι χειρίζεται αριθμούς κινητής υποδιαστολής. Ο κώδικας στο (1.5) φορτώνει τα περιεχόμενα της διεύθυνσης `id3` στον καταχωρητή `R2` και στη συνέχεια τον πολλαπλασιάζει με τη σταθερά κινητής υποδιαστολής `60.0`. Το # σημαίνει ότι το 60.0 θα μεταχειριστεί ως άμεση σταθερά. Η τρίτη εντολή μετακινεί το id2 στον καταχωρητή R1 και η τέταρτη προσθέτει σε αυτόν την τιμή που υπολογίστηκε πριν στον καταχωρητή R2. Τέλος, η τιμή του καταχωρητή R1 αποθηκεύεται στη διεύθυνση του id1, έτσι ώστε ο κώδικας να υλοποιεί σωστά την εντολή της ανάθεσης (1.1). Το Κεφάλαιο 8 καλύπτει την παραγωγή κώδικα.

Η συζήτηση αυτή για την παραγωγή κώδικα έχει αγνοήσει το σημαντικό θέμα της κατανομής αποθηκευτικού χώρου για τους προσδιοριστές του πηγαίου προγράμματος. Όπως θα δούμε στο Κεφάλαιο 7, η οργάνωση του αποθηκευτικού χώρου κατά το χρόνο εκτέλεσης εξαρτάται από τη γλώσσα που μεταγλωττίζεται. Οι αποφάσεις για την κατανομή του αποθηκευτικού χώρου λαμβάνονται είτε κατά την παραγωγή ενδιάμεσου κώδικα είτε κατά τη διάρκεια παραγωγής του τελικού κώδικα.

1.2.7 Διαχείριση Πίνακα Συμβόλων

Μια βασική λειτουργία ενός μεταγλωττιστή είναι η καταγραφή των ονομάτων των μεταβλητών που χρησιμοποιούνται στο πηγαίο πρόγραμμα και η συλλογή πληροφοριών σχετικά με τις διάφορες ιδιότητες κάθε ονόματος. Αυτές οι ιδιότητες μπορεί να παρέχουν πληροφορίες σχετικά με τον αποθηκευτικό χώρο που διατίθεται για ένα όνομα, τον τύπο του, την εμβέλεια (που μπορεί να χρησιμοποιηθεί η τιμή του μέσα στο πρόγραμμα), και στην περίπτωση των ονομάτων διαδικασιών, πληροφορίες όπως ο αριθμός και ο τύπος των ορισμάτων τους, η μέθοδος περάσματος κάθε ορίσματος (για παράδειγμα με τιμή ή με αναφορά) και ο τύπος που επιστρέφεται.

Ο πίνακας συμβόλων είναι μια δομή δεδομένων που περιέχει μια εγγραφή για κάθε όνομα μεταβλητής, με πεδία για τις ιδιότητες του ονόματος. Η δομή δεδομένων πρέπει να είναι κατάλληλα σχεδιασμένη ώστε να επιτρέπει στον μεταγλωττιστή να βρίσκει μια εγγραφή για ένα όνομα γρήγορα και να αποθηκεύει ή να ανακτά δεδομένα από αυτή την εγγραφή επίσης γρήγορα. Οι πίνακες συμβόλων παρουσιάζονται στο Κεφάλαιο 2.

1.2.8 Η Ομαδοποίηση των Φάσεων σε Διελεύσεις

Η δόμηση σε φάσεις ασχολείται με τη λογική οργάνωση ενός μεταγλωττιστή. Σε μια υλοποίηση, δραστηριότητες από πολλές φάσεις μπορεί να ομαδοποιηθούν σε μια *διέλευση* που διαβάζει ένα αρχείο εισόδου και γράφει σε ένα αρχείο εξόδου. Για παράδειγμα, οι φάσεις της «μπροστινής πλευράς» λεκτική ανάλυση, συντακτική ανάλυση, σημασιολογική ανάλυση και παραγωγή ενδιάμεσου κώδικα μπορεί να ομαδοποιηθούν σε μια διέλευση. Η βελτιστοποίηση κώδικα θα μπορούσε να είναι μια προαιρετική διέλευση. Κατόπιν θα μπορούσε να υπάρχει μια διέλευση «πίσω πλευράς» που θα αποτελείται από την παραγωγή κώδικα για μια συγκεκριμένη μηχανή στόχο.

Για μερικές κατηγορίες μεταγλωττιστών που έχουν δημιουργηθεί, έχουν σχεδιάσει προσεκτικά ενδιάμεσες αναπαραστάσεις που επιτρέπουν τη σύνδεση της «μπροστινής πλευράς» για μια συγκεκριμένη γλώσσα με την «πίσω πλευρά» για μια καθορισμένη μηχανή στόχο. Με αυτούς τους μεταγλωττιστές, μπορούμε να κατασκευάσουμε μεταγλωττιστές για διαφορετικές πηγαίες γλώσσες και μια μηχανή στόχο συνδυάζοντας τις διαφορετικές «μπροστινές πλευρές» με την «πίσω πλευρά» για τη συγκεκριμένη μηχανή στόχο. Παρόμοια, μπορούμε να κατασκευάσουμε μεταγλωττιστές για διαφορετικές μηχανές στόχους, συνδυάζοντας μια «μπροστινή πλευρά» με «πίσω πλευρές» για διαφορετικές μηχανές στόχους.

1.2.9 Εργαλεία Κατασκευής Μεταγλωττιστών

Ο δημιουργός μεταγλωττιστών, όπως και ο καθένας που αναπτύσσει λογισμικό, μπορεί να χρησιμοποιήσει αποδοτικά μοντέρνα περιβάλλοντα ανάπτυξης λογισμικού που περιέχουν εργαλεία όπως κειμενογράφους γλωσσών, αποσφαλματωτές, διαχειριστές εκδόσεων λογισμικού, εργαλεία ανάλυσης απόδοσης (profilers), αυτοματοποιημένα πλαίσια ελέγχου λογισμικού (test harnesses) και ούτω καθεξής. Επιπρόσθετα, πέρα των γενικών εργαλείων ανάπτυξης λογισμικού, έχουν δημιουργηθεί άλλα πιο εξειδικευμένα εργαλεία για να βοηθήσουν στις διάφορες φάσεις υλοποίησης ενός μεταγλωττιστή.

Αυτά τα εργαλεία χρησιμοποιούν γλώσσες ειδικού σκοπού για τον καθορισμό και την υλοποίηση εξειδικευμένων συστατικών πολλά δε χρησιμοποιούν ιδιαίτερα πολύπλοκους αλγόριθμους. Τα πιο επιτυχημένα εργαλεία είναι εκείνα που κρύβουν τις λεπτομέρειες του αλγόριθμου δημιουργίας και παράγουν συστατικά (components) που μπορούν εύκολα να ενταχθούν μέσα στα υπόλοιπα συστατικά ενός μεταγλωττιστή. Μερικά συχνά χρησιμοποιούμενα εργαλεία για κατασκευή μεταγλωττιστών περιλαμβάνουν

1. *Δημιουργοί γραμματικών/συντακτικών αναλυτών (Parser generators)* που παράγουν αυτόματα συντακτικούς αναλυτές από την περιγραφή της γραμματικής μιας γλώσσας προγραμματισμού.
2. *Δημιουργοί σαρωτών (Scanner generators)* που παράγουν λεκτικούς αναλυτές από την περιγραφή μιας γλώσσας με κανονικές εκφράσεις για την αναγνώριση των λεκτικών συμβόλων της.
3. *Μηχανές μετάφρασης κατευθυνόμενες από την σύνταξη (Syntax-directed translation)* οι οποίες παράγουν συλλογές από ρουτίνες οι οποίες διασχίζουν ένα δέντρο συντακτικής/γραμματικής ανάλυσης ώστε να παράγουν ενδιάμεσο κώδικα.
4. *Δημιουργοί δημιουργών κώδικα(Code-generator generators)* που παράγουν έναν δημιουργό κώδικα από μια συλλογή κανόνων για τη μετάφραση κάθε λειτουργίας της ενδιάμεσης γλώσσας σε γλώσσα μηχανής της μηχανής στόχου.
5. *Μηχανές ανάλυσης ροής δεδομένων (Data-flow analysis engines)* που διευκολύνουν τη συλλογή πληροφοριών σχετικά με το πώς τιμές μεταδίδονται από ένα τμήμα ενός προγράμματος προς καθένα από τα υπόλοιπα τμήματα. Η ανάλυση ροής δεδομένων είναι σημείο κλειδί στην βελτιστοποίηση κώδικα.
6. *Εργαλειοθήκες κατασκευής μεταγλωττιστών (Compiler-construction toolkits)* που παρέχουν μια ολοκληρωμένη συλλογή από ρουτίνες για την κατασκευή των διαφόρων φάσεων ενός μεταγλωττιστή.

Πολλά από αυτά τα εργαλεία θα περιγραφούν στις σελίδες αυτού του βιβλίου.

1.3 Η Εξέλιξη των Γλωσσών Προγραμματισμού

Η πρώτοι ηλεκτρονικοί υπολογιστές εμφανίστηκαν στη δεκαετία του 1940 και προγραμματίζονταν σε γλώσσα μηχανής από ακολουθίες 0 και 1 που δήλωναν ρητά στον υπολογιστή ποιές λειτουργίες να εκτελέσει και με ποιά σειρά. Οι λειτουργίες από μόνες τους ήταν πολύ «χαμηλού» επιπέδου: μετακίνησε δεδομένα από μια θέση σε μια άλλη, πρόσθεσε τα περιεχόμενα δύο καταχωρητών, σύγκρινε δύο τιμές και ούτω καθεξής. Είναι περιττό να εξηγήσουμε γιατί ότι αυτό το είδος προγραμματισμού ήταν αργό, κουραστικό και επιρρεπές σε λάθη. Επιπρόσθετα, τα προγράμματα απ' τη στιγμή που γράφονταν και μετά ήταν δύσκολο να κατανοηθούν και να τροποποιηθούν.

1.3.1 Η Μετακίνηση σε Γλώσσες Υψηλότερου Επιπέδου

Το πρώτο βήμα προς γλώσσες προγραμματισμού περισσότερο φιλικές προς τους ανθρώπους ήταν η ανάπτυξη μνημονικών συμβολικών γλωσσών στις αρχές της δεκαετίας του 1950. Αρχικά, οι εντολές σε μια συμβολική γλώσσα ήταν απλά μνημονικές αναπαραστάσεις των εντολών μηχανής. Αργότερα, προστέθηκαν μακροεντολές στις συμβολικές γλώσσες, έτσι ώστε ένας προγραμματιστής να μπορεί να ορίσει παραμετροποιήσιμες συντομογραφίες για συχνά χρησιμοποιούμενες ακολουθίες εντολών μηχανής.

Ένα σημαντικό βήμα προς τις γλώσσες υψηλού επιπέδου έγινε το δεύτερο μισό της δεκαετίας του 1950 με την ανάπτυξη της γλώσσας Fortran για επιστημονικούς υπολογισμούς, της Cobol για επεξεργασία επιχειρησιακών δεδομένων και της Lisp για υπολογισμούς με συμβολικές εκφράσεις. Η φιλοσοφία πίσω από αυτές τις γλώσσες ήταν να δημιουργηθούν υψηλού επιπέδου σημειογραφίες με τις οποίες οι προγραμματιστές θα μπορούσαν να γράψουν ευκολότερα αριθμητικούς υπολογισμούς, επιχειρησιακές εφαρμογές και προγράμματα με συμβολικές εκφράσεις. Οι γλώσσες αυτές ήταν τόσο επιτυχημένες που χρησιμοποιούνται ακόμη και σήμερα.

Στις επόμενες δεκαετίες, δημιουργήθηκαν πολύ περισσότερες γλώσσες με καινοτόμα χαρακτηριστικά που βοήθησαν στο να γίνει ο προγραμματισμός ευκολότερος, πιο φυσικός και περισσότερο εύρωστος. Αργότερα σ' αυτό το κεφάλαιο, θα αναπτυχθούν μερικά σημαντικά χαρακτηριστικά που είναι κοινά σε πολλές σύγχρονες γλώσσες προγραμματισμού.

Σήμερα, υπάρχουν χιλιάδες γλώσσες προγραμματισμού. Μπορούν να ταξινομηθούν με πολλούς τρόπους. Μια ταξινόμηση είναι με βάση τη γενιά. Οι γλώσσες μηχανής είναι *γλώσσες πρώτης γενιάς*, οι συμβολικές γλώσσες είναι *δεύτερης γενιάς* ενώ *τρίτης γενιάς* είναι οι υψηλότερου επιπέδου γλώσσες όπως η Fortran, Cobol, Lisp, C, C++, C# και Java. *Τέταρτης γενιάς* γλώσσες είναι γλώσσες που έχουν σχεδιαστεί για συγκεκριμένες εφαρμογές όπως η NOMAD για δημιουργία αναφορών, η SQL για ερωτήματα σε βάσεις δεδομένων και η Postscript για μορφοποίηση κειμένου. Ο όρος *πέμπτη γενιά* χρησιμοποιείται για γλώσσες βασισμένες σε λογική και περιορισμούς όπως η Prolog και η OPS5.

Μια άλλη ταξινόμηση των γλωσσών χρησιμοποιεί τον όρο *προστακτικές* για γλώσσες στις οποίες το πρόγραμμα καθορίζει *πως* θα γίνει ένας υπολογισμός και *δηλωτικές* για γλώσσες στις οποίες το πρόγραμμα καθορίζει *τι* υπολογισμός απαιτείται να γίνει. Γλώσσες όπως οι C, C++, C#, και η Java είναι προστακτικές γλώσ-

σες. Στις προστακτικές γλώσσες υπάρχει η έννοια της κατάστασης του προγράμματος και των εντολών που αλλάζουν αυτή την κατάσταση. Οι συναρτησιακές γλώσσες όπως η ML και η Haskell και οι γλώσσες με περιοριστική λογική όπως η Prolog θεωρούνται συχνά δηλωτικές γλώσσες.

Ο όρος *γλώσσα von Neumann* χρησιμοποιείται σε γλώσσες προγραμματισμού των οποίων το υπολογιστικό μοντέλο βασίζεται στην αρχιτεκτονική υπολογιστών von Neumann. Πολλές από τις σημερινές γλώσσες, όπως η Fortran και η C είναι γλώσσες von Neumann.

Μια *αντικειμενοστρεφής γλώσσα* είναι μια γλώσσα η οποία υποστηρίζει αντικειμενοστρεφή προγραμματισμό, έναν τρόπο προγραμματισμού στον οποίο ένα πρόγραμμα αποτελείται από μια συλλογή από αντικείμενα που αλληλεπιδρούν μεταξύ τους. Οι Simula 67 και η Smalltalk ήταν οι σημαντικότερες πρώτες αντικειμενοστρεφείς γλώσσες. Γλώσσες όπως η C++, C#, η Java και η Ruby είναι πιο πρόσφατες αντικειμενοστρεφείς γλώσσες.

Οι *γλώσσες σεναρίου* είναι διερμηνευόμενες γλώσσες με τελεστές υψηλού επιπέδου σχεδιασμένους για να «συνενώνουν» υπολογισμούς. Αυτοί οι υπολογισμοί αρχικά ονομάζονταν «σενάρια». Οι Awk, Javascript, Perl, PHP, Python, Ruby και Tcl είναι δημοφιλή παραδείγματα γλωσσών σεναρίου. Τα προγράμματα που γράφονται σε γλώσσες σεναρίου είναι συχνά πολύ μικρότερα από τα ισοδύναμα προγράμματα που είναι γραμμένα σε γλώσσες όπως η C.

1.3.2 Η Επίδραση στους Μεταγλωττιστές

Καθώς ο σχεδιασμός των γλωσσών προγραμματισμού και των μεταγλωττιστών συσχετίζονται στενά, η εξέλιξη στις γλώσσες προγραμματισμού έθεσαν νέες απαιτήσεις στους δημιουργούς των μεταγλωττιστών. Έπρεπε να επινοήσουν αλγορίθμους και αναπαραστάσεις για να μεταφράσουν και να υποστηρίξουν τα νέα χαρακτηριστικά των γλωσσών. Η αρχιτεκτονική των υπολογιστών έχει επίσης εξελιχθεί από τη δεκαετία του 1940. Οι δημιουργοί των μεταγλωττιστών εκτός από το να παρακολουθούν τα νέα χαρακτηριστικά των γλωσσών έπρεπε να επινοούν αλγορίθμους μετάφρασης που θα εκμεταλλεύονταν στο μέγιστο τις νέες δυνατότητες του υλικού.

Οι μεταγλωττιστές μπορεί προωθήσουν τη χρήση γλωσσών υψηλού επιπέδου με την ελαχιστοποίηση της επιβάρυνσης εκτέλεσης των προγραμμάτων που είναι γραμμένα σε αυτές τις γλώσσες. Οι μεταγλωττιστές είναι επίσης ζωτικοί στο να κάνουν τις αρχιτεκτονικές υψηλής απόδοσης αποδοτικές στις εφαρμογές των χρηστών. Στην πράξη, η απόδοση ενός υπολογιστικού συστήματος εξαρτάται τόσο πολύ από την τεχνολογία των μεταγλωττιστών ώστε οι μεταγλωττιστές να χρησιμοποιούνται ως εργαλείο για την αποτίμηση αρχιτεκτονικών εννοιών πριν ακόμη κατασκευαστεί ο υπολογιστής.

Η δημιουργία μεταγλωττιστών έχει προκλήσεις. Ένας μεταγλωττιστής από μόνος του είναι ένα μεγάλο πρόγραμμα. Επιπλέον, πολλά μοντέρνα συστήματα επεξεργασίας γλωσσών χειρίζονται διάφορες πηγαίες γλώσσες και μηχανές στόχους εντός του ιδίου πλαισίου, αυτό σημαίνει ότι λειτουργούν ως συλλογές μεταγλωττιστών, πιθανότατα αποτελούμενες από εκατομμύρια γραμμές κώδικα. Συνεπώς είναι απαραίτητη η εφαρμογή καλών πρακτικών τεχνολογίας λογισμικού για τη δημιουργία και εξέλιξη των μοντέρνων γλωσσικών επεξεργαστών.

Ένας μεταγλωττιστής πρέπει να μπορεί να μεταφράσει σωστά τα δυνητικά άπειρα σύνολα προγραμμάτων που θα μπορούσαν να γραφτούν στη πηγαία γλώσσα. Το πρόβλημα της παραγωγής βέλτιστου τελικού κώδικα από ένα πηγαίο πρόγραμμα είναι γενικά αμφίρροπο, έτσι οι δημιουργοί μεταγλωττιστών πρέπει να αποφασίσουν πως και ποια προβλήματα θα χειριστούν και ποιες ευρετικές μεθόδους θα χρησιμοποιήσουν για να προσεγγίσουν το πρόβλημα της παραγωγής αποδοτικού κώδικα.

Η μελέτη των μεταγλωττιστών είναι επίσης μια μελέτη για το πώς η θεωρία συναντά την πράξη, όπως θα δούμε στην Ενότητα 1.4

Ο σκοπός αυτού του βιβλίου είναι να διδάξει τη μεθοδολογία και τις θεμελιώδεις ιδέες που χρησιμοποιούνται στο σχεδιασμό των μεταγλωττιστών. Δεν αποτελεί στόχο του βιβλίου να διδάξει όλους τους αλγορίθμους και τις τεχνικές που θα μπορούσαν να χρησιμοποιηθούν για την κατασκευή ενός προηγμένου συστήματος επεξεργασίας γλωσσών. Παρ' όλα αυτά, οι αναγνώστες αυτού του βιβλίου θα αποκτήσουν τη βασική γνώση και αντίληψη για να μάθουν πώς να κατασκευάζουν έναν μεταγλωττιστή σχετικά εύκολα.

1.3.3 Ασκήσεις Ενότητας 1.3

Άσκηση 1.3.1: Υποδείξτε ποιοι από τους παρακάτω όρους:

α) προστακτική	β) δηλωτική	γ) von Neumann
δ) αντικειμενοστρεφείς	ε) συναρτησιακή	στ) τρίτης γενιάς
ζ) τέταρτης γενιάς	η) σεναρίου	

ταιριάζουν με ποιες από τις παρακάτω γλώσσες:

1) C	2) C++	3) Cobol	4) Fortran	5) Java
6) Lisp	7) ML	8) Perl	9) Python	10) VB.

1.4 Η Επιστήμη της Κατασκευής ενός Μεταγλωττιστή

Ο σχεδιασμός μεταγλωττιστών είναι γεμάτος από όμορφα παραδείγματα στα οποία πολύπλοκα προβλήματα του πραγματικού κόσμου λύνονται με μαθηματική αφαίρεση της ουσίας του προβλήματος. Είναι εξαιρετικά παραδείγματα του πως μπορεί η αφαίρεση να χρησιμοποιηθεί για να λύνει προβλήματα: πάρε ένα πρόβλημα, σχημάτισε μια μαθηματική αφαίρεση που συλλαμβάνει τα βασικά του χαρακτηριστικά και επίλυσέ το με χρήση μαθηματικών τεχνικών. Η διατύπωση του προβλήματος πρέπει να θεμελιώνεται σε μια στέρεα αντίληψη των χαρακτηριστικών των προγραμμάτων και η λύση πρέπει να επικυρώνεται και να βελτιώνεται εμπειρικά.

Ένας μεταγλωττιστής πρέπει να αποδέχεται όλα τα πηγαία προγράμματα που συμφωνούν με τις προδιαγραφές της γλώσσας. Το σύνολο των προγραμμάτων είναι απεριόριστο και οποιοδήποτε πρόγραμμα μπορεί να είναι πολύ μεγάλο, αποτελούμενο πιθανόν από εκατομμύρια γραμμές κώδικα. Κάθε μετατροπή που εκτελείται από το μεταγλωττιστή κατά τη διάρκεια της μετάφρασης ενός πηγαίου προγράμμα-

τος πρέπει να διατηρεί το νόημα του προγράμματος που μεταγλωττίζεται. Έτσι οι δημιουργοί των μεταγλωττιστών έχουν επίδραση όχι μόνο στους μεταγλωττιστές που δημιουργούν αλλά και σε όλα τα προγράμματα που μεταγλωττίζουν οι μεταγλωττιστές τους. Αυτή η ισχύς κάνει το γράψιμο των μεταγλωττιστών ιδιαίτερα ανταποδοτικό, αλλά κάνει επίσης και την ανάπτυξη τους μια πρόκληση.

1.4.1 Η Μοντελοποίηση στη Σχεδίαση και Υλοποίηση Μεταγλωττιστή

Η μελέτη των μεταγλωττιστών είναι κυρίως μια μελέτη του πως σχεδιάζουμε τα σωστά μαθηματικά μοντέλα και επιλέγουμε τους σωστούς αλγορίθμους, εξισορροπώντας την ανάγκη για γενίκευση και ισχύ έναντι της απλότητας και της αποδοτικότητας.

Μερικά από τα πιο θεμελιώδη μοντέλα είναι οι μηχανές πεπερασμένων καταστάσεων και οι κανονικές εκφράσεις, που θα συναντήσουμε στο Κεφάλαιο 3. Αυτά τα μοντέλα είναι χρήσιμα για την περιγραφή των λεκτικών μονάδων των προγραμμάτων (λέξεις κλειδιά, προσδιοριστές και άλλα παρόμοια) και για την περιγραφή των αλγορίθμων που χρησιμοποιούνται από το μεταγλωττιστή για την αναγνώριση αυτών των μονάδων. Επίσης ανάμεσα στα πιο θεμελιώδη μοντέλα είναι οι γραμματικές χωρίς συμφραζόμενα, που χρησιμοποιούνται για την περιγραφή της συντακτικής δομής των γλωσσών προγραμματισμού όπως ο εμφωλιασμός παρενθέσεων ή οι δομές ελέγχου. Οι γραμματικές θα εξεταστούν στο Κεφάλαιο 4. Παρόμοια, όπως θα δούμε στο Κεφάλαιο 5, τα δέντρα αποτελούν ένα σημαντικό μοντέλο για την αναπαράσταση της δομής των προγραμμάτων και τη μετάφρασή τους σε κώδικα μηχανής.

1.4.2 Η Επιστήμη της Βελτιστοποίησης του Κώδικα

Ο όρος «βελτιστοποίηση» στο σχεδιασμό μεταγλωττιστών αναφέρεται στις προσπάθειες που κάνει ένας μεταγλωττιστής για να παράγει κώδικα που είναι πιο αποδοτικός από τον προφανή κώδικα. Η λέξη «βελτιστοποίηση» είναι ακυριολεξία (ακατάλληλη ονομασία) (misnomer), καθώς δεν υπάρχει καμιά μέθοδος που να εγγυάται ότι ο κώδικας που παράγεται από έναν μεταγλωττιστή θα είναι το ίδιο γρήγορος ή γρηγορότερος από οποιονδήποτε άλλο κώδικα που εκτελεί την ίδια εργασία.

Τα τελευταία χρόνια, η βελτιστοποίηση του κώδικα που εκτελεί ένας μεταγλωττιστής έχει γίνει ταυτόχρονα σημαντικότερη και πιο πολύπλοκη. Είναι πιο πολύπλοκη διότι οι αρχιτεκτονικές των επεξεργαστών έχουν γίνει πιο πολύπλοκες, δίνοντας περισσότερες ευκαιρίες για βελτίωση του τρόπου με τον οποίο εκτελείται ο κώδικας. Είναι περισσότερο σημαντική διότι μαζικοί παράλληλοι υπολογιστές απαιτούν ουσιώδη βελτιστοποίηση, διαφορετικά η απόδοσή τους υστερεί κατά τάξεις μεγέθους. Με την πιθανή επικράτηση των πολυπύρηνων μηχανών (υπολογιστές με ολοκληρωμένα κυκλώματα που περιέχουν μεγάλο αριθμό επεξεργαστών), όλοι οι μεταγλωττιστές θα έχουν να αντιμετωπίσουν το πρόβλημα της εκμετάλλευσης των πολυεπεξεργαστικών μηχανών.

Είναι δύσκολο, αν όχι αδύνατο, να κατασκευαστεί ένας στιβαρός μεταγλωττιστής χωρίς «κόλπα». Έτσι, μια εκτεταμένη και χρήσιμη θεωρία έχει χτιστεί γύρω από το πρόβλημα της βελτιστοποίησης του κώδικα. Η χρήση μιας αυστηρής μαθη-

ματικής θεμελίωσης μας επιτρέπει να αποδείξουμε ότι μια βελτιστοποίηση είναι σωστή και ότι παράγει το επιθυμητό αποτέλεσμα για όλες τις πιθανές εισόδους. Θα δούμε, αρχίζοντας από το Κεφάλαιο 9, πως μοντέλα όπως τα γραφήματα, οι πίνακες και τα γραμμικά προγράμματα είναι αναγκαία, αν ο μεταγλωττιστής πρόκειται να παράγει καλά βελτιστοποιημένο κώδικα.

Από την άλλη πλευρά, η θεωρία από μόνη της είναι ανεπαρκής. Όπως σε πολλά προβλήματα του πραγματικού κόσμου, δεν υπάρχουν τέλειες απαντήσεις. Στην πραγματικότητα οι περισσότερες από τις ερωτήσεις που τίθενται στην βελτιστοποίηση των μεταγλωττιστών είναι αμφίρροπες. Μια από τις πιο σημαντικές δεξιότητες στον σχεδιασμό μεταγλωττιστών είναι η ικανότητα της διατύπωσης του σωστού προβλήματος προς επίλυση. Για αρχή, χρειαζόμαστε καλή κατανόηση της συμπεριφοράς των προγραμμάτων και ενδελεχή πειραματισμό και αξιολόγηση για να επιβεβαιωθεί η διαίσθηση μας.

Οι βελτιστοποιήσεις των μεταγλωττιστών πρέπει να ακολουθούν τους παρακάτω σχεδιαστικούς στόχους:

- Η βελτιστοποίηση πρέπει να είναι σωστή, που σημαίνει, διατήρηση του νοήματος του μεταγλωττιζόμενου προγράμματος.
- Η βελτιστοποίηση πρέπει να βελτιώνει την απόδοση πολλών προγραμμάτων.
- Ο χρόνος μεταγλώττισης πρέπει να διατηρείται σε λογικές τιμές, και
- Η απαιτούμενη προσπάθεια ανάπτυξης πρέπει να είναι εφικτή.

Είναι αδύνατο να τονίσουμε υπερβολικά τη σημαντικότητα της ορθότητας. Είναι εύκολο να φτιάξεις ένα μεταγλωττιστή που δημιουργεί γρήγορο κώδικα, αν ο παραγόμενος κώδικας δε χρειάζεται να είναι σωστός! Οι μεταγλωττιστές βελτιστοποίησης είναι τόσο δύσκολο να είναι πάντα ορθοί, που θα μπορούσαμε να πούμε ότι κανένας μεταγλωττιστής βελτιστοποίησης δεν είναι πλήρως απαλλαγμένος από λάθη!

Ο δεύτερος στόχος είναι ότι ο μεταγλωττιστής πρέπει να είναι αποδοτικός στην βελτίωση της απόδοσης πολλών προγραμμάτων. Κανονικά απόδοση σημαίνει την ταχύτητα της εκτέλεσης του προγράμματος. Ειδικά στις ενσωματωμένες εφαρμογές, μπορεί να θέλουμε επίσης να ελαχιστοποιήσουμε το μέγεθος του παραγόμενου κώδικα. Στην περίπτωση των κινητών συσκευών, είναι επίσης επιθυμητό ο κώδικας να ελαχιστοποιεί την κατανάλωση ισχύος. Ουσιαστικά οι ίδιες βελτιστοποιήσεις που επιταχύνουν το χρόνο εκτέλεσης, εξοικονομούν επίσης και ισχύ. Παράλληλα με την απόδοση, θέματα ευχρηστίας όπως η αναφορά λαθών και η αποσφαλμάτωση είναι επίσης σημαντικά.

Τρίτον, χρειάζεται να διατηρούμε το χρόνο μεταγλώττισης μικρό για να υποστηρίζεται ένα γρήγορος κύκλος ανάπτυξης και αποσφαλμάτωσης. Αυτή η απαίτηση καλύπτεται ευκολότερα καθώς οι μηχανές γίνονται πιο γρήγορες. Συχνά ένα πρόγραμμα πρώτα αναπτύσσεται και αποσφαλματώνεται, χωρίς βελτιστοποιήσεις. Ο χρόνος μεταγλώττισης όχι μόνο μειώνεται, αλλά το πιο σημαντικό είναι ότι, τα μη βελτιστοποιημένα προγράμματα είναι ευκολότερο να αποσφαλματωθούν, διότι οι βελτιστοποιήσεις που εισάγονται από ένα μεταγλωττιστή συχνά κάνουν ασαφή τη σχέση ανάμεσα στον πηγαίο κώδικα και τον παραγόμενο κώδικα. Η ενεργοποίηση των βελτιστοποιήσεων στο μεταγλωττιστή μπορεί να εμφανίσει νέα προβλήμα-

τα στο πηγαίο πρόγραμμα, για αυτό το λόγο πρέπει να γίνουν πάλι έλεγχοι ορθής λειτουργίας στο βελτιστοποιημένο κώδικα. Η ανάγκη για επιπλέον ελέγχους αποθαρρύνει τη χρήση των βελτιστοποιήσεων στις εφαρμογές, ιδίως αν η απόδοσή τους δεν είναι τόσο κρίσιμης σημασίας.

Τελικά, ένας μεταγλωττιστής είναι ένα πολύπλοκο σύστημα. Πρέπει να κρατάμε το σύστημα απλό για να διασφαλιστεί ότι το κόστος της ανάπτυξης και της συντήρησης του μεταγλωττιστή είναι εφικτό. Υπάρχει ένας άπειρος αριθμός βελτιστοποιήσεων που θα μπορούσαμε να υλοποιήσουμε και η δημιουργία μιας σωστής και αποδοτικής βελτιστοποίησης θέλει αρκετή προσπάθεια. Πρέπει να θέσουμε προτεραιότητα στις βελτιστοποιήσεις, υλοποιώντας μόνο εκείνες που οδηγούν στα μεγαλύτερα οφέλη των πηγαίων προγραμμάτων και συναντιόνται στην πράξη.

Έτσι, στη μελέτη των μεταγλωττιστών, μαθαίνουμε όχι μόνο πώς να δημιουργήσουμε ένα μεταγλωττιστή αλλά επίσης τη γενική μεθοδολογία επίλυσης πολύπλοκων και ανοιχτών προβλημάτων. Η προσέγγιση που χρησιμοποιείται στην ανάπτυξη ενός μεταγλωττιστή συμπεριλαμβάνει τόσο θεωρία όσο και πειραματισμό. Συνήθως ξεκινάμε διατυπώνοντας το πρόβλημα βασιζόμενοι στη διαίσθησή μας για το ποια είναι τα σημαντικά ζητήματα.

1.5 Εφαρμογές της Τεχνολογίας των Μεταγλωττιστών

Ο σχεδιασμός των μεταγλωττιστών δεν έχει να κάνει μόνο με τους μεταγλωττιστές, και πολλοί άνθρωποι χρησιμοποιούν τις τεχνολογικές γνώσεις που απέκτησαν από τη μελέτη των μεταγλωττιστών, χωρίς να έχουν γράψει, με αυστηρά κριτήρια, ακόμη και τμήμα ενός μεταγλωττιστή για μια γλώσσα προγραμματισμού ευρείας χρήσης. Η τεχνολογία των μεταγλωττιστών έχει και άλλες σημαντικές χρήσεις. Επιπρόσθετα, ο σχεδιασμός μεταγλωττιστών επιδρά σε πολλές άλλες περιοχές της επιστήμης των υπολογιστών. Σε αυτή την ενότητα, κάνουμε μια ανασκόπηση των πιο σημαντικών αλληλεπιδράσεων και εφαρμογών αυτής της τεχνολογίας.

1.5.1 Υλοποίηση Γλωσσών Προγραμματισμού Υψηλού Επιπέδου

Μια γλώσσα προγραμματισμού υψηλού-επιπέδου ορίζει μια προγραμματιστική αφαίρεση: ο προγραμματιστής εκφράζει έναν αλγόριθμο χρησιμοποιώντας τη γλώσσα και ο μεταγλωττιστής πρέπει να μεταφράσει το πρόγραμμα στην τελική γλώσσα. Γενικά είναι ευκολότερο να προγραμματίσεις σε μια γλώσσα υψηλού-επιπέδου, αλλά είναι λιγότερο αποδοτικό, που σημαίνει ότι τα τελικά προγράμματα τρέχουν πιο αργά. Οι προγραμματιστές που χρησιμοποιούν μια χαμηλού-επιπέδου γλώσσα έχουν περισσότερο έλεγχο πάνω σε έναν υπολογισμό και μπορούν, κατά κανόνα, να παράγουν πιο αποδοτικό κώδικα. Δυστυχώς, τα χαμηλού-επιπέδου προγράμματα είναι δυσκολότερο να γραφτούν και – ακόμη χειρότερα – είναι λιγότερο μεταφέρσιμα, περισσότερο επιρρεπή σε λάθη και δυσκολότερα να συντηρηθούν. Οι μεταγλωττιστές βελτιστοποίησης συμπεριλαμβάνουν τεχνικές για τη βελτίωση της απόδοσης του παραγόμενου κώδικα, αντισταθμίζοντας μ' αυτόν τον τρόπο την αναποτελεσματικότητα που εισάγεται από τις υψηλού-επιπέδου αφαιρέσεις.

Παράδειγμα 1.2: Η λέξη κλειδί **register** της γλώσσας προγραμματισμού C είναι ένα από τα πρώτα παραδείγματα της αλληλεπίδρασης μεταξύ τεχνολογίας μεταγλωττιστών και εξέλιξης της γλώσσας. Όταν δημιουργήθηκε η γλώσσα προγραμματισμού C στα μέσα της δεκαετίας του 1970, θεωρούνταν αναγκαίο να αφεθεί στον προγραμματιστή ο έλεγχος του ποιες μεταβλητές του προγράμματος τοποθετούνται σε καταχωρητές. Αυτός ο έλεγχος έγινε περιττός καθώς αναπτύχθηκαν αποδοτικές τεχνικές ανάθεσης καταχωρητών και τα περισσότερα σύγχρονα προγράμματα δεν χρησιμοποιούν πλέον αυτό το χαρακτηριστικό της γλώσσας.

Στην πραγματικότητα, τα προγράμματα που χρησιμοποιούν τη λέξη-κλειδί **register** μπορεί να χάνουν σε αποδοτικότητα, διότι οι προγραμματιστές συχνά δεν είναι ο καλύτερος κριτής σε θέματα πολύ χαμηλού-επιπέδου όπως είναι η ανάθεση καταχωρητών. Η βέλτιστη επιλογή της ανάθεσης καταχωρητών εξαρτάται σε μεγάλο βαθμό από τις ιδιαιτερότητες της αρχιτεκτονικής μιας μηχανής. Η μη αυτοματοποιημένη λήψη χαμηλού-επιπέδου αποφάσεων που σχετίζονται με τη διαχείριση πόρων όπως για παράδειγμα η ανάθεση καταχωρητών μπορεί στην πράξη να βλάψει την απόδοση, ειδικά αν το πρόγραμμα τρέχει σε άλλες μηχανές από εκείνη για την οποία γράφτηκε.

Οι πολλές μεταστροφές στην επιλογή των δημοφιλέστερων γλωσσών προγραμματισμού έχουν γίνει προς την κατεύθυνση της αύξησης του επιπέδου αφαίρεσης. Η C ήταν η κυρίαρχη γλώσσα προγραμματισμού συστημάτων την δεκαετία του 80, όμως πολλά από τα νέα έργα που ξεκίνησαν την δεκαετία του 90 επέλεξαν την C++. Η Java, που εισήχθη το 1995, κέρδισε γρήγορα δημοτικότητα στο τέλος της δεκαετίας του 90. Τα νέα χαρακτηριστικά των γλωσσών προγραμματισμού που εισήχθησαν σε κάθε φάση έγιναν κίνητρο για νέα έρευνα στην βελτιστοποίηση των μεταγλωττιστών. Ακολούθως, δίνουμε μια σφαιρική θεώρηση των κύριων χαρακτηριστικών των γλωσσών που έχουν υποκινήσει σημαντικές προόδους στην τεχνολογία των μεταγλωττιστών.

Ουσιαστικά όλες οι ευρέως διαδεδομένες γλώσσες προγραμματισμού, συμπεριλαμβανομένων των C, Fortran και Cobol υποστηρίζουν οριζόμενους από το χρήστη συγκεντρωτικούς (aggregate) τύπους δεδομένων όπως οι πίνακες (arrays) και οι δομές (structures) και έλεγχο ροής υψηλού επιπέδου, όπως οι βρόχοι και οι κλήσεις διαδικασιών. Αν πάρουμε απλά κάθε υψηλού-επιπέδου δομή ή λειτουργία πρόσβασης-δεδομένων και τη μεταφράσουμε απ' ευθείας σε κώδικα μηχανής, το αποτέλεσμα θα ήταν σε μεγάλο βαθμό μη αποδοτικό. Έχει αναπτυχθεί μια συλλογή βελτιστοποιήσεων μεταγλωττιστή γνωστές ως *βελτιστοποιήσεις ροής δεδομένων* που μας δίνουν τη δυνατότητα ανάλυσης της ροής δεδομένων μέσα σε ένα πρόγραμμα και αφαιρούν πλεονασμούς που συναντώνται σε αυτές τις δομές. Στην πράξη ο παραγόμενος κώδικας μοιάζει με τον κώδικα που έχει γραφτεί από έναν πεπειραμένο προγραμματιστή σε μια γλώσσα κατωτέρου επιπέδου.

Ο προσανατολισμός προς τα αντικείμενα εισήχθη πρώτα στην Simula το 1967 και έχει ενσωματωθεί σε γλώσσες όπως οι Smalltalk, C++, C# και Java. Οι βασικές ιδέες πίσω από την αντικειμενοστρέφεια είναι:

1. Η αφαίρεση δεδομένων και
2. Η κληρονομικότητα των ιδιοτήτων,

οι οποίες κάνουν τα προγράμματα πιο τμηματικά με αποτέλεσμα να είναι ευκολότερο να συντηρηθούν. Τα αντικειμενοστρεφή προγράμματα είναι διαφορετικά από αυτά που είναι γραμμένα σε πολλές άλλες γλώσσες στο ότι αποτελούνται από πολύ περισσότερες, αλλά μικρότερες, διαδικασίες (αποκαλούνται *μέθοδοι* στην αντικειμενοστρεφή ορολογία). Έτσι, οι βελτιστοποιήσεις του μεταγλωττιστή πρέπει να μπορούν να αποδίδουν καλά στα όρια ανάμεσα στις διαδικασίες του πηγαίου προγράμματος. Η ένθεση διαδικασίας (procedure inlining), που είναι η αντικατάσταση της κλήσης μια διαδικασίας με το σώμα της διαδικασίας, είναι ιδιαίτερα χρήσιμη εδώ. Έχουν επίσης αναπτυχθεί βελτιστοποιήσεις που επιταχύνουν την εκτέλεση των εικονικών (virtual) μεθόδων.

Η Java έχει πολλά χαρακτηριστικά που κάνουν τον προγραμματισμό ευκολότερο, πολλά από τα οποία έχουν εισαχθεί νωρίτερα σε άλλες γλώσσες. Η γλώσσα Java υποστηρίζει ασφάλεια τύπου, αυτό σημαίνει ότι ένα αντικείμενο δεν μπορεί να χρησιμοποιηθεί ως ένα αντικείμενο ενός μη σχετιζόμενου τύπου. Όλες οι προσπελάσεις σε πίνακες ελέγχονται για να επιβεβαιωθεί ότι βρίσκονται εντός των ορίων του πίνακα. Η Java δεν έχει δείκτες και δεν επιτρέπει αριθμητική δεικτών. Έχει μια ενσωματωμένη δυνατότητα εκκαθάρισης αχρήστων αντικειμένων που αυτόματα αποδεσμεύει τη μνήμη μεταβλητών που δε χρησιμοποιούνται πλέον. Αν και όλα αυτά τα χαρακτηριστικά κάνουν τον προγραμματισμό ευκολότερο, επιφέρουν επιβάρυνση στο χρόνο εκτέλεσης. Έχουν αναπτυχθεί βελτιστοποιήσεις μεταγλωττιστή για να μειώσουν την επιβάρυνση, για παράδειγμα, περιορίζοντας μη αναγκαίους ελέγχους ορίων και δημιουργώντας αντικείμενα που δεν είναι προσβάσιμα πέρα από μια διαδικασία στη στοίβα (stack) αντί για το σωρό (heap). Έχουν επίσης αναπτυχθεί αποδοτικοί αλγόριθμοι για να ελαχιστοποιήσουν το επιπλέον κόστος της συλλογής αχρήστων αντικειμένων.

Επιπρόσθετα, η Java έχει σχεδιαστεί για να υποστηρίζει φορητό και μεταφερτό κώδικα. Τα προγράμματα διανέμονται ως ενδιάμεσος κώδικας της εικονικής μηχανής Java, ο οποίος πρέπει είτε να διερμηνευτεί είτε να μεταγλωττιστεί δυναμικά, δηλαδή κατά το χρόνο εκτέλεσης σε κώδικα μηχανής. Έχει επίσης μελετηθεί η εφαρμογή της δυναμικής μεταγλώττισης και σε άλλες χρήσεις όπου εξάγονται δυναμικά πληροφορίες κατά τον χρόνο εκτέλεσης, οι οποίες χρησιμοποιούνται για να παραχθεί βελτιστοποιημένος κώδικας. Στη δυναμική βελτιστοποίηση είναι σημαντικό να ελαχιστοποιηθεί ο χρόνος μεταγλώττισης καθώς είναι μέρος της επιβάρυνσης του χρόνου εκτέλεσης. Μια συνήθης τεχνική που χρησιμοποιείται είναι να μεταγλωττίζονται και να βελτιστοποιούνται μόνο εκείνα τα τμήματα του προγράμματος που θα εκτελεστούν συχνότερα.

1.5.2 Βελτιστοποιήσεις για Αρχιτεκτονικές Υπολογιστών

Η γρήγορη εξέλιξη της αρχιτεκτονικής των υπολογιστών έχει επίσης οδηγήσει σε μια ακόρεστη ζήτηση για νέες τεχνολογίες στους μεταγλωττιστές. Σχεδόν όλα τα συστήματα υψηλής απόδοσης επωφελούνται των ίδιων δύο βασικών τεχνικών: παραλληλισμός και ιεραρχίες μνήμης. Ο παραλληλισμός μπορεί να βρεθεί σε διάφορα επίπεδα: στο *επίπεδο εντολών* όπου πολλαπλές λειτουργίες εκτελούνται ταυτόχρονα και στο *επίπεδο επεξεργαστή* όπου πολλαπλά νήματα (threads) της ίδιας εφαρμογής εκτελούνται σε διαφορετικούς επεξεργαστές. Οι ιεραρχίες μνήμης εί-

ναι η απάντηση στον βασικό περιορισμό ότι μπορούμε να κατασκευάσουμε πολύ γρήγορο ή πολύ μεγάλο χώρο αποθήκευσης, αλλά όχι χώρο αποθήκευσης που είναι ταυτόχρονα γρήγορος και μεγάλος.

Παραλληλισμός

Όλοι οι μοντέρνοι μικροεπεξεργαστές εκμεταλλεύονται τον παραλληλισμό επιπέδου εντολών. Ωστόσο, αυτός ο παραλληλισμός μπορεί να κρυφτεί από τον προγραμματιστή. Τα προγράμματα γράφονται θεωρώντας ότι οι εντολές εκτελούνται ακολουθιακά και το υλικό (hardware) ελέγχει δυναμικά για εξαρτήσεις στην ακολουθιακή ροή των εντολών και τις εκτελεί παράλληλα όποτε είναι δυνατόν. Σε μερικές περιπτώσεις, η μηχανή περιλαμβάνει έναν χρονοπρογραμματιστή υλικού που μπορεί να αλλάξει την σειρά των εντολών για να αυξήσει τον παραλληλισμό στο πρόγραμμα. Ανεξάρτητα από το εάν το υλικό αλλάζει την σειρά των εντολών ή όχι, οι μεταγλωττιστές μπορούν να αναδιατάξουν τις εντολές για να κάνουν περισσότερο αποδοτικό τον παραλληλισμό επιπέδου εντολών.

Ο παραλληλισμός επιπέδου εντολών μπορεί επίσης ξεκάθαρα να εμφανισθεί στο σύνολο εντολών. Οι μηχανές πολύ μεγάλης λέξης εντολής (VLIW – Very Long Instruction Word) έχουν εντολές που μπορούν να εκτελούν παράλληλα πολλαπλές λειτουργίες. Η αρχιτεκτονική Intel IA64 είναι ένα πολύ γνωστό παράδειγμα τέτοιας μηχανής. Όλοι οι υψηλής απόδοσης, γενικού σκοπού μικροεπεξεργαστές συμπεριλαμβάνουν επίσης εντολές που μπορούν να λειτουργούν την ίδια χρονική στιγμή σε ένα πίνακα δεδομένων. Για αυτές τις μηχανές έχουν αναπτυχθεί τεχνικές μεταγλωττιστών που παράγουν αυτόματα τέτοιου είδους κώδικα από ακολουθιακά προγράμματα.

Οι πολυεπεξεργαστές έχουν γίνει επίσης τόσο διαδεδομένοι που ακόμη και προσωπικοί υπολογιστές συχνά έχουν πολλαπλούς επεξεργαστές. Οι προγραμματιστές έχουν τη δυνατότητα να γράψουν πολυνηματικό (multithreaded) κώδικα για τους πολυεπεξεργαστές ή μπορεί να παραχθεί αυτόματα παράλληλος κώδικας από έναν μεταγλωττιστή από συμβατικά ακολουθιακά προγράμματα. Ένας τέτοιος μεταγλωττιστής κρύβει από τους προγραμματιστές τις λεπτομέρειες εύρεσης παραλληλισμού σε ένα πρόγραμμα, κατανέμοντας τον υπολογισμό στη μηχανή και ελαχιστοποιώντας το συγχρονισμό και την επικοινωνία ανάμεσα στους επεξεργαστές. Πολλές εφαρμογές επιστημονικού υπολογισμού και εφαρμοσμένης μηχανικής είναι υπολογιστικά απαιτητικές και μπορούν να επωφεληθούν σημαντικά από την παράλληλη επεξεργασία. Έχουν αναπτυχθεί τεχνικές παραλληλισμού για την αυτοματοποιημένη μετάφραση ακολουθιακών επιστημονικών προγραμμάτων σε κώδικα για πολυεπεξεργαστές.

Ιεραρχίες μνημών

Μια ιεραρχία μνήμης αποτελείται από πολλά επίπεδα αποθήκευσης με διαφορετικές ταχύτητες και μεγέθη, με το επίπεδο που βρίσκεται κοντά στον επεξεργαστή να είναι το γρηγορότερο αλλά ταυτόχρονα το μικρότερο. Ο μέσος χρόνος προσπέλασης μνήμης ενός προγράμματος μειώνεται αν οι περισσότερες από τις προσπελάσεις του ικανοποιούνται από τα γρηγορότερα επίπεδα της ιεραρχίας. Τόσο ο παραλληλισμός όσο και η ύπαρξη μια ιεραρχίας μνήμης βελτιώνουν την δυνητι-

κή απόδοση μιας μηχανής, αλλά πρέπει να αξιοποιηθούν αποδοτικά από τον μεταγλωττιστή για να προσδώσουν πραγματική απόδοση σε μια εφαρμογή.

Οι ιεραρχίες μνήμης βρίσκονται σε όλες τις μηχανές. Ένας επεξεργαστής έχει συνήθως έναν μικρό αριθμό καταχωρητών που αποτελούνται από εκατοντάδες bytes, πολλά επίπεδα κρυφών μνημών (cache) που περιέχουν από kilobytes έως megabytes μνήμης, φυσική μνήμη που μεγέθους από megabytes έως gigabytes και τέλος δευτερεύουσα αποθήκευση που προσφέρει μεγέθη από gigabytes και πάνω. Αντίστοιχα, η ταχύτητα προσπέλασης ανάμεσα σε διαδοχικά επίπεδα μιας ιεραρχίας μπορεί να διαφέρει από δύο ή τρεις τάξεις μεγέθους. Η απόδοση ενός συστήματος συχνά περιορίζεται όχι από την ταχύτητα του επεξεργαστή αλλά από την απόδοση του υποσυστήματος μνήμης. Ενώ παραδοσιακά οι μεταγλωττιστές εστιάζουν στην βελτιστοποίηση της εκτέλεσης στον επεξεργαστή, σήμερα δίνεται περισσότερη έμφαση στο να κάνουμε την ιεραρχία μνήμης πιο αποδοτική.

Η αποδοτική χρήση των καταχωρητών είναι πιθανώς το σημαντικότερο πρόβλημα στη βελτιστοποίηση ενός προγράμματος. Οι καταχωρητές πρέπει σαφώς να διαχειριστούν με λογισμικό, σε αντίθεση με τις κρυφές και φυσικές μνήμες που είναι κρυμμένες από το σύνολο εντολών και διαχειρίζονται από το υλικό. Έχει βρεθεί ότι σε μερικές περιπτώσεις στρατηγικές διαχείρισης κρυφής μνήμης υλοποιούμενες από υλικό δεν είναι αποδοτικές, ειδικά σε κώδικα επιστημονικών υπολογισμών που έχει μεγάλες δομές δεδομένων (κυρίως πίνακες). Είναι εφικτό να βελτιωθεί η απόδοση της ιεραρχίας μνήμης με αλλαγή της διάταξης των δεδομένων ή αλλάζοντας την σειρά των εντολών που προσπελαύνουν τα δεδομένα. Μπορούμε επίσης να αλλάξουμε τη διάταξη του κώδικα για να βελτιώσουμε την αποδοτικότητα των κρυφών μνημών για εντολές (instruction caches).

1.5.3 Σχεδίαση Νέων Αρχιτεκτονικών Υπολογιστών

Στις πρώτες μέρες της σχεδίασης αρχιτεκτονικών υπολογιστών, οι μεταγλωττιστές αναπτύσσονταν αφού είχαν φτιαχτεί οι μηχανές. Αυτό έχει αλλάξει. Καθώς ο προγραμματισμός σε γλώσσες υψηλού επιπέδου είναι ο κανόνας, η απόδοση ενός υπολογιστικού συστήματος αποφασίζεται όχι μόνο από την καθαρή ταχύτητα της μηχανής αλλά επίσης από το πόσο καλά μπορούν οι μεταγλωττιστές να εκμεταλλευτούν τις δυνατότητές της. Έτσι, στην ανάπτυξη σύγχρονων αρχιτεκτονικών υπολογιστών, οι μεταγλωττιστές αναπτύσσονται κατά τη φάση σχεδιασμού του επεξεργαστή και ο μεταγλωττισμένος κώδικας, που τρέχει σε προσομοιωτές, χρησιμοποιείται για την αποτίμηση των χαρακτηριστικών της προτεινόμενης αρχιτεκτονικής.

RISC

Ένα από τα πιο γνωστά παραδείγματα του πως οι μεταγλωττιστές επηρέασαν το σχεδιασμό της αρχιτεκτονικής υπολογιστών, ήταν η επινόηση της αρχιτεκτονικής RISC (Reduced Instruction-Set Computer – Υπολογιστής με Μειωμένο Σύνολο-Εντολών). Πριν από αυτή την εφεύρεση, η τάση ήταν να αναπτύσσονται προοδευτικά όλο και πιο πολύπλοκα σύνολα εντολών με σκοπό να κάνουν ευκολότερο τον προγραμματισμό σε συμβολική γλώσσα (assembly). Αυτές οι αρχιτεκτονικές ήταν γνωστές ως CISC (Complex Intsruction-Set Computer – Υπολογιστής με Πολύπλοκο Σύνολο Εντολών) Για παράδειγμα, τα σύνολα εντολών CISC περιλαμβάνουν πολύπλοκες μεθόδους διευθυνσιοδότησης μνήμης που υποστηρίζουν εντολές

προσπέλασης σε δομές-δεδομένων και κλήσης διαδικασιών που αποθηκεύουν καταχωρητές και περνούν παραμέτρους στη στοίβα.

Οι βελτιστοποιήσεις μεταγλωττιστών μπορούν συχνά να μειώσουν αυτές τις εντολές σε ένα μικρό αριθμό απλούστερων λειτουργιών εξαλείφοντας τους πλεονασμούς στις πολύπλοκες εντολές. Έτσι, είναι προτιμότερο να κατασκευάζονται απλούστερα σύνολα εντολών τα οποία οι μεταγλωττιστές μπορούν να τα χρησιμοποιήσουν αποδοτικά, ενώ ταυτόχρονα και το υλικό είναι ευκολότερο να βελτιστοποιηθεί.

Οι περισσότερες αρχιτεκτονικές επεξεργαστών γενικού-σκοπού, συμπεριλαμβανομένων των PowerPC, SPARC, MIPS, Alpha και PA-RISC βασίζονται στην ιδέα του RISC. Αν και η αρχιτεκτονική x86 – ο πιο δημοφιλής μικροεπεξεργαστής – έχει ένα σύνολο εντολών CISC, πολλές από τις ιδέες που αναπτύχθηκαν για τις μηχανές RISC χρησιμοποιούνται στην υλοποίηση του ίδιου του επεξεργαστή. Επιπλέον, ο πιο αποδοτικός τρόπος να χρησιμοποιήσεις μια μηχανή x86 υψηλής απόδοσης είναι να χρησιμοποιήσεις μόνο τις απλές τις εντολές.

Εξειδικευμένες αρχιτεκτονικές

Στην διάρκεια των τριών τελευταίων δεκαετιών έχουν προταθεί πολλές αρχιτεκτονικές έννοιες. Περιλαμβάνουν τις μηχανές ροής δεδομένων, ανυσματικές μηχανές, μηχανές VLIW (Very Long Instruction Word – Πολύ Μεγάλης Λέξης Εντολής), SIMD (Single Instruction, Multiple Data – Μονής Εντολής, Πολλαπλών Δεδομένων), συστοιχίες επεξεργαστών, συστολικές συστοιχίες, πολυεπεξεργαστές με διαμοιραζόμενη μνήμη και πολυεπεξεργαστές με κατανεμημένη μνήμη. Η ανάπτυξη καθεμιάς από αυτές τις αρχιτεκτονικές έννοιες συνοδεύονταν από έρευνα και ανάπτυξη της αντίστοιχης τεχνολογίας μεταγλωττιστών.

Μερικές από αυτές τις ιδέες έχουν βρει το δρόμο τους στους σχεδιασμούς των ενσωματωμένων μηχανών. Καθώς ολόκληρα συστήματα μπορούν να χωρέσουν σε ένα μόνο ολοκληρωμένο κύκλωμα, οι επεξεργαστές δεν είναι ανάγκη να είναι πλέον προκατασκευασμένα τμήματα προϊόντων, αλλά μπορούν να προσαρμοστούν για να επιτευχθεί καλύτερη αποτελεσματικότητα κόστους για μια συγκεκριμένη εφαρμογή. Έτσι, σε αντίθεση με τους επεξεργαστές γενικού σκοπού, όπου οι οικονομίες κλίμακας έχουν οδηγήσει τις αρχιτεκτονικές των υπολογιστών να συγκλίνουν, οι επεξεργαστές εξειδικευμένων εφαρμογών παρουσιάζουν μια ποικιλία υπολογιστικών αρχιτεκτονικών. Η τεχνολογία μεταγλωττιστών χρειάζεται να υποστηρίζει όχι μόνο τον προγραμματισμό γι' αυτές τις αρχιτεκτονικές, αλλά επίσης να αξιολογεί προτεινόμενα σχέδια αρχιτεκτονικών.

1.5.4 Μεταφράσεις Προγραμμάτων

Ενώ κανονικά σκεφτόμαστε τη μεταγλώττιση ως μια μετάφραση από μια υψηλού επιπέδου γλώσσα σε μία γλώσσα επιπέδου μηχανής, η ίδια τεχνολογία μπορεί να εφαρμοστεί για τη μετάφραση ανάμεσα σε διαφορετικά είδη γλωσσών. Στην συνέχεια παρουσιάζονται μερικές από τις σημαντικές εφαρμογές των τεχνικών μετάφρασης προγραμμάτων.

Δυαδική Μετάφραση

Η τεχνολογία των μεταγλωττιστών μπορεί να χρησιμοποιηθεί για να μεταφράσει

δυαδικό κώδικα μιας μηχανή σε αυτόν μιας άλλης, επιτρέποντας σε μια μηχανή να τρέξει προγράμματα που αρχικά είχαν μεταγλωττιστεί για άλλο σύνολο εντολών. Η δυαδική μετάφραση έχει χρησιμοποιηθεί από διάφορες εταιρείες υπολογιστών για να αυξήσουν τη διαθεσιμότητα λογισμικού για τις μηχανές τους. Συγκεκριμένα, εξαιτίας της κυριαρχίας της αγοράς προσωπικών υπολογιστών x86, οι περισσότεροι τίτλοι λογισμικού είναι διαθέσιμοι ως x86 κώδικας. Έχουν αναπτυχθεί δυαδικοί μεταφραστές για τη μετατροπή κώδικα x86 τόσο σε κώδικα Alpha όσο και σε κώδικα Sparc. Η δυαδική μετάφραση έχει επίσης χρησιμοποιηθεί από την Trasnmeta Inc. στη δική τους υλοποίηση του συνόλου εντολών x86. Αντί να εκτελεί το πολύπλοκο σύνολο εντολών του x86 απ' ευθείας σε υλικό, ο επεξεργαστής Transmeta Crusoe είναι ένας επεξεργαστής VLIW που βασίζεται στη δυαδική μετάφραση για να μετατρέπει κώδικα x86 σε εγγενή κώδικα VLIW.

Η δυαδική μετάφραση μπορεί επίσης να χρησιμοποιηθεί για την παροχή συμβατότητας προς τα πίσω. Όταν ο επεξεργαστής στο Apple Macintosh άλλαξε από Motorolla MC 68040 σε PowerPC to 1994, χρησιμοποιήθηκε δυαδική μετάφραση για να επιτραπεί στους επεξεργαστές PowerPC να τρέχουν παραδοσιακό κώδικα MC 68040.

Σύνθεση υλικού

Δεν γράφεται μόνο το λογισμικό σε γλώσσες υψηλού-επιπέδου. Ακόμη και ο σχεδιασμός υλικού περιγράφεται κυρίως σε υψηλού επιπέδου γλώσσες περιγραφής υλικού όπως η Verilog και η VHDL (Very high-speed integrated circuit Hardware Description Language – Γλώσσα Περιγραφής Υλικού πολύ γρήγορων ολοκληρωμένων κυκλωμάτων). Τα σχέδια υλικού περιγράφονται σε επίπεδο μεταφοράς καταχωρητή (RTL – Register Tranfer Level), όπου οι μεταβλητές αναπαριστούν καταχωρητές και οι εκφράσεις αναπαριστούν συνδυαστική λογική. Τα εργαλεία σύνθεσης υλικού μεταφράζουν αυτόματα περιγραφές RTL σε πύλες, οι οποίες στην συνέχεια αντιστοιχίζονται σε τρανζίστορ και τελικά σε φυσικό σχεδιασμό. Σε αντίθεση με τους μεταγλωττιστές, αυτά τα εργαλεία παίρνουν συχνά ώρες βελτιστοποιώντας το κύκλωμα. Υπάρχουν επίσης τεχνικές για μετάφραση από υψηλότερα επίπεδα όπως το επίπεδο συμπεριφοράς ή λειτουργικότητας.

Διερμηνευτές ερωτήσεων βάσης δεδομένων

Εκτός από τον καθορισμό λογισμικού και υλικού, οι γλώσσες είναι χρήσιμες σε πολλές άλλες εφαρμογές. Για παράδειγμα, γλώσσες ερωτημάτων, ιδιαίτερα η SQL (Structured Query Language – Δομημένη Γλώσσα Ερωτημάτων), χρησιμοποιούνται για αναζήτηση σε βάσεις δεδομένων. Οι ερωτήσεις προς τη βάση δεδομένων αποτελούνται από κατηγορήματα που περιέχουν σχεσιακούς και δυαδικούς τελεστές. Μπορούν να διερμηνευτούν ή να μεταγλωττιστούν σε εντολές για αναζήτηση σε μια βάση δεδομένων για εγγραφές που ικανοποιούν το κατηγόρημα αυτό.

Μεταγλωττισμένη προσομοίωση

Η προσομοίωση είναι μια γενική τεχνική που χρησιμοποιείται σε πολλούς επιστημονικούς κλάδους και κλάδους εφαρμοσμένης μηχανικής για την κατανόηση ενός φαινομένου ή για την επιβεβαίωση μιας σχεδίασης. Οι είσοδοι στον προσομοιωτή

συνήθως περιλαμβάνουν την περιγραφή της σχεδίασης και ειδικές παραμέτρους εισόδου για τη συγκεκριμένη εκτέλεση της προσομοίωσης. Οι προσομοιώσεις μπορεί να είναι υπολογιστικά πολύ ακριβές. Τυπικά χρειάζεται να προσομοιωθούν πολλές πιθανές παραλλαγές της σχεδίασης με πολλά διαφορετικά σετ εισόδου και κάθε πείραμα μπορεί να πάρει μέρες για να ολοκληρωθεί σε μια μηχανή υψηλής απόδοσης. Αντί να γράψουμε έναν προσομοιωτή που διερμηνεύει τη σχεδίαση, είναι ταχύτερο να μεταγλωττίσουμε τη σχεδίαση ώστε να παραχθεί κώδικας μηχανής που εξομοιώνει εγγενώς τη συγκεκριμένη σχεδίαση. Η μεταγλωττισμένη προσομοίωση μπορεί να τρέξει τάξεις μεγέθους γρηγορότερα απ' ότι η προσέγγιση που βασίζεται σε διερμηνευτή. Η μεταγλωττισμένη προσομοίωση χρησιμοποιείται στα καλύτερα εργαλεία που προσομοιώνουν σχεδιάσεις γραμμένες σε Verilog ή VHDL.

1.5.5 Εργαλεία Ανάπτυξης Λογισμικού

Βάσιμα τα προγράμματα είναι τα πιο πολύπλοκα τεχνουργήματα της εφαρμοσμένης μηχανικής που έχουν ποτέ παραχθεί. Αποτελούνται από πολλές λεπτομέρειες, καθεμιά από τις οποίες πρέπει να είναι σωστή πριν το πρόγραμμα λειτουργήσει πλήρως. Ως αποτέλεσμα, τα λάθη είναι ευρέως διαδεδομένα στα προγράμματα. Τα προγράμματα μπορεί να σταματήσουν απότομα ένα σύστημα, μπορεί να παράγουν λάθος αποτελέσματα, να κάνουν ένα σύστημα επιρρεπές σε επιθέσεις ασφαλείας ή ακόμη να οδηγήσουν σε καταστροφικές αποτυχίες κρίσιμα συστήματα. Ο συστηματικός έλεγχος είναι η βασική τεχνική για τον εντοπισμό λαθών στα προγράμματα.

Μια ενδιαφέρουσα και πολλά υποσχόμενη συμπληρωματική προσέγγιση είναι η χρήση ανάλυσης ροής δεδομένων για το στατικό εντοπισμό λαθών (δηλαδή, πριν τρέξει το πρόγραμμα). Η ανάλυση ροής δεδομένων μπορεί να βρει λάθη σε όλα τα πιθανά μονοπάτια εκτέλεσης και όχι μόνο σε εκείνα που εξετάζονται από τα σύνολα πιθανών δεδομένων εισόδου όπως στην περίπτωση του συνήθους συστηματικού ελέγχου (testing) του προγράμματος. Πολλές από τις τεχνικές ανάλυσης ροής δεδομένων, που αναπτύχθηκαν αρχικά για βελτιστοποιήσεις μεταγλωττιστών, μπορούν να χρησιμοποιηθούν για δημιουργία εργαλείων που βοηθούν τους προγραμματιστές σε έργα εφαρμοσμένης μηχανικής λογισμικού.

Το πρόβλημα της εύρεσης όλων των λαθών ενός προγράμματος είναι στη γενίκευσή του άλυτο. Μια ανάλυση ροής δεδομένων μπορεί να σχεδιαστεί να προειδοποιεί τους προγραμματιστές για όλες τις πιθανές εντολές με μια συγκεκριμένη κατηγορία λαθών. Αλλά αν οι περισσότερες από αυτές τις προειδοποιήσεις είναι λανθασμένες, οι χρήστες δεν θα χρησιμοποιήσουν το εργαλείο. Έτσι, οι πραγματικοί ανιχνευτές λαθών δεν είναι συχνά ούτε τέλειοι ούτε πλήρεις. Αυτό σημαίνει, ότι μπορεί να μην βρουν όλα τα λάθη στο πρόγραμμα και για όλα τα λάθη που αναφέρονται να μην υπάρχει εγγύηση ότι είναι πραγματικά λάθη. Παρ' όλα αυτά, έχουν αναπτυχθεί πολλές στατικές αναλύσεις που είναι αποδοτικές σε πραγματικά προγράμματα στην εύρεση λαθών, όπως η ανάκτηση τιμής από κενή αναφορά (null) ή από απελευθερωμένο δείκτη. Το γεγονός ότι οι ανιχνευτές λαθών μπορεί να είναι ατελείς, τους κάνει σημαντικά διαφορετικούς από τις βελτιστοποιήσεις των μεταγλωττιστών. Οι βελτιστοποιητές πρέπει να είναι συντηρητικοί και δεν μπορούν σε καμιά περίπτωση να αλλάξουν την σημασιολογία του προγράμματος.

Συνολικά σ' αυτή την ενότητα, θα αναφέρουμε πολλούς τρόπους με τους οποίους η ανάλυση του προγράμματος, βασιζόμενη σε τεχνικές που αρχικά αναπτύ-

χθηκαν για τη βελτιστοποίηση κώδικα σε μεταγλωττιστές, έχουν βελτιώσει την ανάπτυξη λογισμικού. Ιδιαίτερης σημασίας είναι τεχνικές που ανιχνεύουν στατικά πότε ένα πρόγραμμα θα μπορούσε να είναι εκτεθειμένο σε κίνδυνο ασφαλείας.

Έλεγχος τύπων

Ο έλεγχος τύπων είναι μια αποδοτική και καλά εδραιωμένη τεχνική για τον εντοπισμό ασυνεπειών στα προγράμματα. Μπορεί να χρησιμοποιηθεί για να εντοπίσει, για παράδειγμα, λάθη που εμφανίζονται όταν μια λειτουργία εφαρμόζεται στον λάθος τύπο αντικειμένου ή αν οι παράμετροι που περνιούνται σε μια διαδικασία δεν ταιριάζουν με την υπογραφή αυτής της διαδικασίας. Η ανάλυση του προγράμματος μπορεί να πάει πέρα από την εύρεση λαθών τύπων αναλύοντας την ροή των δεδομένων στο πρόγραμμα. Για παράδειγμα, αν σε ένα δείκτη αποδίδεται η μηδενική τιμή (null) και κατόπιν γίνεται προσπάθεια ανάκτησης της τιμής από την θέση μνήμης που δείχνει ο δείκτης τότε το πρόγραμμα βρίσκεται ξεκάθαρα σε μια λανθασμένη κατάσταση.

Η ίδια τεχνολογία μπορεί να χρησιμοποιηθεί για να εντοπιστεί μια ποικιλία από κενά ασφαλείας, στα οποία ένας επιτιθέμενος μπορεί να παρέχει μια συμβολοσειρά ή άλλα δεδομένα που χρησιμοποιούνται απρόσεκτα από το πρόγραμμα. Μια συμβολοσειρά που παρέχεται από το χρήστη μπορεί να χαρακτηριστεί με τον τύπο «επικίνδυνη». Αν μια συμβολοσειρά αυτού του τύπου μπορεί να επηρεάσει τη ροή ελέγχου του κώδικα σε κάποιο σημείο του προγράμματος, τότε υπάρχει ένα πιθανό ασθενές σημείο από άποψη ασφαλείας.

Έλεγχος ορίων

Είναι ευκολότερο να γίνουν λάθη όταν προγραμματίζουμε σε μια γλώσσα χαμηλότερου επιπέδου σε σχέση με μια γλώσσα υψηλότερου επιπέδου. Για παράδειγμα, πολλές παραβιάσεις ασφαλείας σε συστήματα προκαλούνται από υπερχειλίσεις προσωρινής μνήμης σε προγράμματα γραμμένα σε C. Επειδή η C δεν έχει έλεγχο ορίων πίνακα, η διασφάλιση ότι οι πίνακες δεν προσπελαύνονται εκτός ορίων εξαρτάται από το χρήστη. Το πρόγραμμα που αποτυγχάνει να ελέγξει ότι τα δεδομένα που δίνονται από το χρήστη μπορεί να υπερχειλίσουν την προσωρινή μνήμη, μπορεί να ξεγελαστεί στο να αποθηκεύσει δεδομένα ενός χρήστη εκτός μνήμης. Ένας επιτιθέμενος μπορεί να μεταβάλλει τα δεδομένα εισόδου προκαλώντας την δυσλειτουργία του προγράμματος και διακινδυνεύοντας την ασφάλεια του συστήματος. Έχουν αναπτυχθεί τεχνικές, με περιορισμένη επιτυχία όμως, για την εύρεση υπερχειλίσεων μνήμης στα προγράμματα.

Αν το πρόγραμμα είχε γραφτεί σε μια ασφαλή γλώσσα που συμπεριλαμβάνει αυτόματο έλεγχο εύρους, αυτό το πρόβλημα δε θα υπήρχε. Η ανάλυση ροής δεδομένων που χρησιμοποιείται για να περιοριστούν πλεονάζοντες έλεγχοι ορίων μπορεί επίσης να χρησιμοποιηθεί για τον εντοπισμό υπερχειλίσεων προσωρινής μνήμης. Η σημαντική διαφορά, παρ' όλα αυτά, είναι ότι ενώ η αποτυχία περιορισμού ενός ελέγχου ορίων θα είχε ως αποτέλεσμα ένα μικρό επιπλέον κόστος στο χρόνο εκτέλεσης, η αποτυχία προσδιορισμού μιας υπερχείλισης προσωρινής μνήμης μπορεί να διακινδυνεύσει την ασφάλεια του συστήματος. Έτσι, ενώ η χρήση απλών τεχνικών είναι κατάλληλη για τη βελτιστοποίηση ελέγχων ορίων, χρειάζονται σύνθετες

και περίπλοκες τεχνικές ανάλυσης για να έχει κανείς αποτελέσματα υψηλής ποιότητας στα εργαλεία ανίχνευσης λαθών, όπως η παρακολούθηση των τιμών των δεικτών που χρησιμοποιούνται σε περισσότερες από μία διαδικασίες.

Εργαλεία Διαχείρισης-Μνήμης

Η συλλογή αχρήστων αντικειμένων (garbage collection) είναι άλλο ένα εξαίρετο παράδειγμα ισορροπίας ανάμεσα στην αποδοτικότητα και σε έναν συνδυασμό ευκολίας προγραμματισμού και αξιοπιστίας λογισμικού. Η αυτόματη διαχείριση μνήμης εξαλείφει όλα τα λάθη διαχείρισης μνήμης (π.χ. διαρροή μνήμης), που είναι μια σημαντική πηγή προβλημάτων στα προγράμματα σε C και C++. Έχουν αναπτυχθεί διάφορα εργαλεία για να βοηθήσουν τους προγραμματιστές να βρουν λάθη διαχείρισης μνήμης. Για παράδειγμα, το Purify είναι ένα ευρέως χρησιμοποιούμενο εργαλείο που πιάνει δυναμικά λάθη διαχείρισης μνήμης την στιγμή που συμβαίνουν. Έχουν επίσης αναπτυχθεί και εργαλεία που βοηθούν στον προσδιορισμό μερικών από αυτά τα προβλήματα στατικά.

1.6 Βασικές Γνώσεις Γλωσσών Προγραμματισμού

Σε αυτή την ενότητα, θα καλύψουμε τη σημαντικότερη ορολογία και χαρακτηριστικά που εμφανίζονται στη μελέτη των γλωσσών προγραμματισμού. Ο σκοπός μας δεν είναι να καλύψουμε όλες τις έννοιες των δημοφιλών γλωσσών προγραμματισμού. Υποθέτουμε ότι ο αναγνώστης είναι εξοικειωμένος με τουλάχιστον μια από τις C, C++, C#, ή Java και ότι μπορεί να έχει συναντήσει επιπλέον και άλλες γλώσσες.

1.6.1 Η Διάκριση Στατικού / Δυναμικού Μέρους

Ανάμεσα στα πιο σημαντικά θέματα που αντιμετωπίζουμε όταν σχεδιάζουμε ένα μεταγλωττιστή για μια γλώσσα, είναι ποιες αποφάσεις μπορεί να πάρει ένας μεταγλωττιστής για ένα πρόγραμμα. Αν μια γλώσσα χρησιμοποιεί μια πολιτική που επιτρέπει στο μεταγλωττιστή να αποφασίσει για ένα θέμα, τότε λέμε ότι η γλώσσα χρησιμοποιεί μια *στατική* πολιτική ή ότι το θέμα μπορεί να αποφασιστεί κατά το *χρόνο μεταγλώττισης*. Από την άλλη πλευρά, μια πολιτική που επιτρέπει μια απόφαση να παρθεί μόνο όταν εκτελούμε το πρόγραμμα, λέγεται *δυναμική πολιτική* ή ότι απαιτεί μια απόφαση κατά το *χρόνο εκτέλεσης*.

Ένα θέμα στο οποίο θα εστιάσουμε είναι η εμβέλεια των δηλώσεων. Η *εμβέλεια* μιας δήλωσης του x είναι η περιοχή του προγράμματος στην οποία χρήσεις του x αναφέρονται σε αυτή τη δήλωση. Μια γλώσσα χρησιμοποιεί *στατική εμβέλεια* ή *λεξικογραφική εμβέλεια* εάν απλά κοιτώντας το πρόγραμμα είναι δυνατόν να αποφασιστεί η εμβέλεια μιας δήλωσης. Διαφορετικά, η γλώσσα χρησιμοποιεί *δυναμική εμβέλεια*. Με τη δυναμική εμβέλεια, καθώς το πρόγραμμα εκτελείται, η ίδια χρήση του x θα μπορούσε να αναφέρεται σε οποιαδήποτε από αρκετές διαφορετικές δηλώσεις του x.

Οι περισσότερες γλώσσες, όπως η C και η Java, χρησιμοποιούν στατική εμβέλεια. Η στατική εμβέλεια θα συζητηθεί στην Ενότητα 1.6.3.

Παράδειγμα 1.3: Ως ένα άλλο παράδειγμα του χαρακτηρισμού στατικό/δυναμικό, θεωρήστε τη χρήση του όρου «static» όπως εφαρμόζεται στα δεδομένα σε μια δήλωση κλάσης στην Java. Στην Java, μια μεταβλητή είναι το όνομα για μια θέση στη μνήμη που χρησιμοποιείται για να κρατά μια τιμή δεδομένων. Εδώ, το «static» αναφέρεται όχι στην εμβέλεια της μεταβλητής, αλλά μάλλον στη δυνατότητα του μεταγλωττιστή να αποφασίσει τη θέση στην μνήμη όπου μπορεί να βρεθεί η δηλωμένη μεταβλητή. Μια δήλωση όπως

```
public static int x;
```

κάνει το *x μια μεταβλητή κλάσης* και δηλώνει ότι υπάρχει μόνο ένα αντίγραφο του *x*, ανεξάρτητα από το πόσα αντικείμενα αυτής της κλάσης δημιουργούνται. Επιπλέον, ο μεταγλωττιστής μπορεί να προσδιορίσει μια θέση στη μνήμη όπου θα κρατηθεί ο ακέραιος *x*. Αντίθετα αν το «static» είχε παραληφθεί από αυτή τη δήλωση, τότε κάθε αντικείμενο της κλάσης θα είχε τη δική του θέση στη μνήμη όπου θα κρατούσε το *x* και ο μεταγλωττιστής δεν θα μπορούσε να προσδιορίσει όλες αυτές τις θέσεις πριν από το τρέξιμο του προγράμματος.

1.6.2 Περιβάλλοντα και Καταστάσεις

Άλλη μια σημαντική διάκριση που πρέπει να κάνουμε όταν συζητάμε για γλώσσες προγραμματισμού είναι αν αλλαγές που συμβαίνουν καθώς το πρόγραμμα εκτελείτε επηρεάζουν τις τιμές των δεδομένων ή επηρεάζουν τη διερμηνεία των ονομάτων για αυτά τα δεδομένα. Για παράδειγμα, η εκτέλεση μιας ανάθεσης όπως η x = y + 1 αλλάζει την τιμή που αναφέρεται από το όνομα *x*. Πιο συγκεκριμένα, η ανάθεση αλλάζει την τιμή σε οποιαδήποτε θέση δηλώνεται από το *x*.

Μπορεί να είναι λιγότερο ξεκάθαρο ότι η θέση που δηλώνεται από το *x* μπορεί να αλλάζει κατά το χρόνο εκτέλεσης. Για παράδειγμα, όπως συζητήσαμε στο Παράδειγμα 1.3, αν το *x* δεν είναι στατική (ή «κλάσης») μεταβλητή, τότε κάθε αντικείμενο της κλάσης έχει τη δική του θέση για ένα στιγμιότυπο της μεταβλητής *x*. Σε αυτή την περίπτωση, η ανάθεση στο *x* μπορεί να αλλάξει οποιοδήποτε από τα «στιγμιότυπα» της μεταβλητής, ανάλογα με το αντικείμενο στο οποίο χρησιμοποιείται μια μέθοδος που περιέχει την ανάθεση.

Εικόνα 1.8: Αντιστοίχηση δύο-βημάτων από ονόματα σε τιμές

Η συσχέτιση των ονομάτων με θέσεις στην μνήμη (την *αποθήκη*) και κατόπιν με τιμές μπορεί να περιγραφεί από δύο αντιστοιχίσεις που αλλάζουν καθώς το πρόγραμμα τρέχει (βλέπε Εικόνα 1.8):

1. Το *περιβάλλον* είναι μια αντιστοίχιση από ονόματα σε θέσεις στη μνήμη.

Αφού οι μεταβλητές αναφέρονται σε θέσεις («l-values» στην ορολογία της C), θα μπορούσαμε εναλλακτικά να ορίσουμε ένα περιβάλλον ως μια αντιστοίχιση από ονόματα σε μεταβλητές.

2. Η *κατάσταση* είναι μια αντιστοίχιση από θέσεις στη μνήμη στις τιμές τους. Αυτό σημαίνει, ότι η κατάσταση αντιστοιχίζει «l-values» στις αντίστοιχες «r-values», στην ορολογία της C.

Τα περιβάλλοντα αλλάζουν σύμφωνα με τους κανόνες εμβέλειας μιας γλώσσας.

Παράδειγμα 1.4: Θεωρήστε το απόσπασμα του προγράμματος σε C στην Εικόνα 1.9. Ο ακέραιος *i* δηλώνεται ως καθολική μεταβλητή και δηλώνεται επίσης ως μια τοπική μεταβλητή της συνάρτησης *f*. Όταν εκτελείται η *f*, το περιβάλλον προσαρμόζεται έτσι ώστε το όνομα *i* να αναφέρεται στη θέση που είναι δεσμευμένη για το *i* το οποίο είναι τοπικό στην *f* και κάθε χρήση του *i*, όπως η ανάθεση που φαίνεται i = 3, αναφέρονται σ' αυτή τη θέση. Τυπικά, στην τοπική μεταβλητή *i* δίνεται ένας χώρος στη στοίβα εκτέλεσης.

```
...
int i;                  /* global i        */
...
void f(···) {
      int i;            /* local i         */
      ...
      i = 3;            /* use of local i  */
      ...
}
...
      x = i + 1;        /* use of global i */
```

Εικόνα 1.9.: Δύο δηλώσεις του ονόματος *i*

Όποτε εκτελείται μια συνάρτηση *g* διαφορετική από την *f*, οι χρήσεις του *i* δεν μπορεί να αναφέρονται στο *i* που είναι τοπικό στην *f*. Οι χρήσεις του ονόματος *i* στην *g* πρέπει να βρίσκονται εντός της εμβέλειας μια άλλης δήλωσης του *i*. Ένα παράδειγμα είναι η δήλωση x = i + 1 και βρίσκεται εντός μιας διαδικασίας της οποίας ο ορισμός δεν εμφανίζεται. Το *i* στο *i+1* υποτίθεται ότι αναφέρεται στο καθολικό *i*. Όπως στις περισσότερες γλώσσες, οι δηλώσεις στην C πρέπει να προηγούνται της χρήσης τους, έτσι μια συνάρτηση που εμφανίζεται πριν από την καθολική μεταβλητή *i* δεν μπορεί να αναφέρεται σ' αυτήν.

Το περιβάλλον και οι αντιστοιχίσεις κατάστασης στην Εικόνα 1.8 είναι δυναμικά, αλλά υπάρχουν μερικές εξαιρέσεις:

1. *Η στατική έναντι της δυναμικής σύνδεσης* ονομάτων σε θέσεις. Οι περισσότερες συνδέσεις ονομάτων με θέσεις είναι δυναμικές και παρουσιάζονται διάφορες προσεγγίσεις σε αυτόν τον τύπο σύνδεσης σε αυτή τη ενότητα. Σε μερικές δηλώσεις, όπως το global *i* στην Εικόνα 1.9, μπορεί να δοθεί μια

Ονόματα, Προσδιοριστές και Μεταβλητές

Αν και οι όροι «όνομα» και «μεταβλητή» συχνά αναφέρονται στο ίδιο πράγμα, τους χρησιμοποιούμε με προσοχή για τη διάκριση ανάμεσα στα ονόματα κατά το χρόνο μεταγλώττισης και τις θέσεις μνήμης κατά το χρόνο εκτέλεσης που υποδηλώνονται από τα ονόματα.

Ένας *προσδιοριστής,* είναι μια συμβολοσειρά χαρακτήρων, κυρίως γράμματα ή ψηφία, που αναφέρεται σε (προσδιορίζει) μια οντότητα, όπως ένα αντικείμενο, μια διαδικασία, μια κλάση ή έναν τύπο. Όλοι οι προσδιοριστές είναι ονόματα αλλά όλα τα ονόματα δεν είναι προσδιοριστές. Τα ονόματα μπορεί επίσης να είναι εκφράσεις. Για παράδειγμα το όνομα *x.y* μπορεί να υποδηλώνει το πεδίο *y* μιας δομής που υποδηλώνεται από το *x*. Εδώ τα *x* και *y* είναι προσδιοριστές, ενώ το *x.y* είναι ένα όνομα, αλλά όχι ένας προσδιοριστής. Σύνθετα ονόματα όπως το *x.y* ονομάζονται προσδιοριζόμενα ονόματα.

Μια *μεταβλητή* αναφέρεται σε μια συγκεκριμένη θέση στη μνήμη. Είναι κοινό ο ίδιος προσδιοριστής να δηλώνεται πάνω από μια φορά. Κάθε τέτοια δήλωση εισάγει μια νέα μεταβλητή. Ακόμη και αν κάθε προσδιοριστής δηλώνεται μόνο μια φορά, ένας προσδιοριστής που είναι τοπικός σε μια αναδρομική διαδικασία θα αναφέρεται σε διαφορετικές θέσεις στη μνήμη σε διαφορετικές χρονικές στιγμές.

θέση για αποθήκευση μια για πάντα, καθώς ο μεταγλωττιστής δημιουργεί κώδικα μηχανής.[2]

2. *Η στατική έναντι της δυναμικής σύνδεσης* θέσεων σε τιμές. Η σύνδεση των θέσεων με τιμές (το δεύτερο βήμα στην Εικόνα 1.8), είναι γενικά επίσης δυναμική καθώς δεν μπορούμε να ξέρουμε την τιμή σε μια θέση μέχρι να τρέξουμε το πρόγραμμα. Οι σταθερές είναι μια εξαίρεση. Για παράδειγμα ο ορισμός σε C

```
#define ARRAYSIZE 1000
```

συνδέει το όνομα `ARRAYSIZE` με την τιμή `1000` στατικά. Μπορούμε να προσδιορίσουμε πλήρως αυτή τη σύνδεση κοιτάζοντας την δήλωση, αφού γνωρίζουμε ότι είναι αδύνατο να αλλάξει αυτή η σύνδεση όταν εκτελείται το πρόγραμμα.

1.6.3 Στατική Εμβέλεια και Δομή Τμημάτων

Οι περισσότερες γλώσσες, συμπεριλαμβανομένης της C και της οικογένειάς της, χρησιμοποιούν στατικές εμβέλειες. Οι κανόνες εμβέλειας για τη C βασίζονται

[2]Τεχνικά, ο μεταγλωττιστής της C θα παραχωρήσει μια θέση στην εικονική μνήμη για το global *i*, αφήνοντας στον φορτωτή και το λειτουργικό σύστημα να αποφασίσουν που ακριβώς θα τοποθετηθεί το *i* στην φυσική μνήμη του υπολογιστή. Παρ' όλα αυτά δεν θα μας ενοχλήσουν θέματα «επανατοποθέτησης» όπως αυτά τα οποία δεν έχουν καμιά επίπτωση στην μεταγλώττιση. Αντ' αυτού, αντιμετωπίζουμε τον χώρο διευθύνσεων που χρησιμοποιεί ο μεταγλωττιστής για την παραγωγή κώδικα σαν να εκχωρούσε θέσεις στην φυσική μνήμη.

Διαδικασίες, Συναρτήσεις, και Μέθοδοι

Για να αποφύγουμε να λέμε «διαδικασίες, συναρτήσεις, ή μέθοδοι» κάθε φορά που θέλουμε να μιλήσουμε για ένα υποπρόγραμμα που μπορεί να κληθεί, θα αναφερόμαστε συνήθως σε όλα αυτά σας «διαδικασίες». Εξαίρεση αποτελεί όταν μιλάμε ρητά για προγράμματα σε συγκεκριμένες γλώσσες όπως η C που έχουν μόνο συναρτήσεις. Στην περίπτωση αυτή θα αναφερόμαστε σε αυτά ως «συναρτήσεις». Αν πραγματευόμαστε μια γλώσσα όπως η Java που έχει μόνο μεθόδους, θα χρησιμοποιούμε τον όρο αυτό.

Μια συνάρτηση γενικά επιστρέφει μια τιμή κάποιου τύπου (τον «τύπο επιστροφής») ενώ μια διαδικασία δεν επιστρέφει καμιά τιμή. Η C και οι παρόμοιες με αυτή γλώσσες, που έχουν μόνο συναρτήσεις, χειρίζονται τις διαδικασίες ως συναρτήσεις που έχουν έναν ειδικό τύπο επιστροφής «void» που σημαίνει ότι δεν υπάρχει τιμή επιστροφής. Οι αντικειμενοστρεφείς γλώσσες όπως η Java και η C++ χρησιμοποιούν τον όρο «μέθοδοι». Αυτές μπορεί να συμπεριφέρονται είτε ως συναρτήσεις είτε ως διαδικασίες, αλλά συσχετίζονται μια συγκεκριμένη κλάση.

στη δομή του προγράμματος. Η εμβέλεια μια δήλωσης αποφασίζεται έμμεσα από το που εμφανίζεται η δήλωση μέσα στο πρόγραμμα. Μεταγενέστερες γλώσσες, όπως η C++, η Java και η C#, παρέχουν επίσης έλεγχο των εμβελειών μέσω της χρήσης λέξεων κλειδιά όπως public, private και protected.

Σε αυτή την ενότητα μελετάμε κανόνες στατικής εμβέλειας για γλώσσες με μπλοκ όπου ένα *μπλοκ* είναι μια ομαδοποίηση δηλώσεων και εντολών. Η C χρησιμοποιεί τα άγκιστρα { και } για να οριοθετήσει ένα μπλοκ. Η εναλλακτική χρήση του begin και end για τον ίδιο σκοπό χρονολογείται στην Algol.

Παράδειγμα 1.5: Σε μια πρώτη προσέγγιση, η πολιτική στατικής εμβέλειας της C έχει ως εξής:

1. Ένα πρόγραμμα σε C αποτελείται από μια ακολουθία δηλώσεων μεταβλητών και συναρτήσεων κορυφαίου επιπέδου.
2. Οι συναρτήσεις μπορεί να εμπεριέχουν δηλώσεις μεταβλητών, όπου ως μεταβλητές συμπεριλαμβάνουμε τις τοπικές μεταβλητές και τις παραμέτρους της συνάρτησης. Η εμβέλεια κάθε τέτοιας δήλωσης περιορίζεται στη συνάρτηση στην οποία εμφανίζεται.
3. Η εμβέλεια μιας κορυφαίου επιπέδου δήλωσης ενός ονόματος x ισχύει σε όλο το πρόγραμμα που ακολουθεί, με εξαίρεση εκείνων των εντολών που βρίσκονται εντός μιας συνάρτησης που επίσης έχει μια δήλωση του x.

Η πρόσθετη λεπτομέρεια που αφορά την πολιτική στατικής εμβέλειας στη C ασχολείται με τις δηλώσεις μεταβλητών εντός εντολών. Εξετάζουμε τέτοιες δηλώσεις αμέσως μετά, στο Παράδειγμα 1.6.

Στη C το συντακτικό των μπλοκ δίνεται από:

1. Ένας τύπος εντολής είναι το μπλοκ. Τα μπλοκ μπορούν να εμφανιστούν

οπουδήποτε μπορούν να εμφανιστούν άλλοι τύποι εντολών, όπως οι εντολές ανάθεσης.

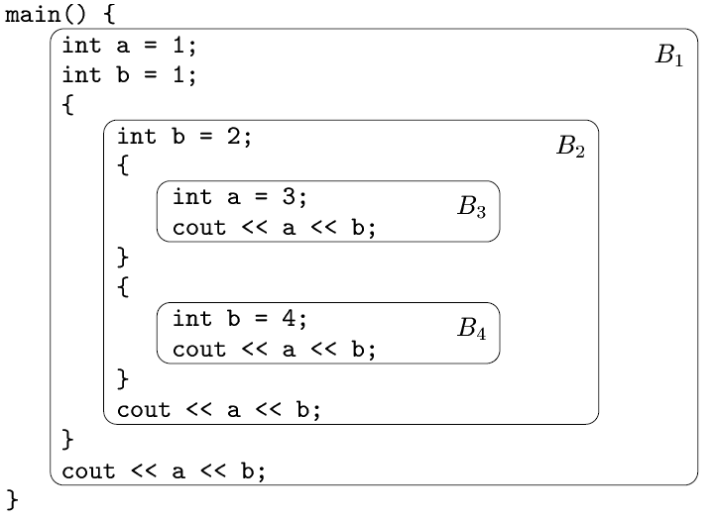
2. Ένα μπλοκ είναι μια ακολουθία δηλώσεων ακολουθούμενη από μια ακολουθία εντολών τα οποία περιβάλλονται μέσα σε άγκιστρα.

Σημειώστε ότι αυτή η σύνταξη επιτρέπει στα μπλοκ να εμφωλιάζονται το ένα μέσα στο άλλο. Αυτή η ιδιότητα εμφωλιάσματος αναφέρεται ως *δομή μπλοκ*. Η οικογένεια γλωσσών της C έχει δομή μπλοκ, με εξαίρεση ότι μια συνάρτηση δεν μπορεί να οριστεί εντός άλλης συνάρτησης.

Λέμε ότι η δήλωση D «ανήκει» σε ένα μπλοκ B αν το B είναι το πιο κοντινά εμφωλιασμένο μπλοκ που περιέχει το D. Αυτό σημαίνει ότι το D βρίσκεται εντός του B, αλλά όχι εντός οποιουδήποτε μπλοκ που είναι εμφωλιασμένο εντός του B.

Ο κανόνας στατικής εμβέλειας για δηλώσεις μεταβλητών σε γλώσσες δομημένες με μπλοκ έχει ως ακολούθως. Αν η δήλωση D του ονόματος x ανήκει στο μπλοκ B, τότε η εμβέλεια του D είναι όλο το B, με εξαίρεση οποιαδήποτε μπλοκ B' τα οποία είναι εμφωλιασμένα σε οποιοδήποτε βάθος εντός του B στα οποία το x δηλώνεται ξανά. Το x δηλώνεται ξανά στο B' αν κάποια άλλη δήλωση D' του ιδίου ονόματος x ανήκει στο B'.

Ένας ισοδύναμος τρόπος να εκφραστεί αυτός ο κανόνας είναι να εστιάσουμε στην χρήση ενός ονόματος x. Ας υποθέσουμε ότι τα B_1, B_2,..., B_k είναι όλα τα μπλοκ που περιβάλλουν τη χρήση του x, με το B_k να είναι το μικρότερο, το οποίο είναι εμφωλιασμένο εντός του B_{k-1}, το οποίο είναι εμφωλιασμένο εντός του B_{k-2} και ούτω καθεξής. Ψάχνουμε για το μεγαλύτερο i, τέτοιο ώστε να υπάρχει μια δήλωση του x που να ανήκει στο B_i. Αυτή η χρήση του x θα ανήκει στην δήλωση του μέσα στο B_i. Εναλλακτικά, αυτή η χρήση του x είναι εντός της εμβέλειας της δήλωσης που βρίσκεται μέσα στο B_i.



Εικόνα 1.10: Μπλοκ σε ένα πρόγραμμα C++

Παράδειγμα 1.6: Το C++ πρόγραμμα της Εικόνας 1.10 έχει τέσσερα μπλοκ, με διάφορους ορισμούς των μεταβλητών a και b. Ως βοήθημα μνήμης, κάθε δήλωση

αρχικοποιεί τη μεταβλητή της στον αριθμό του μπλοκ στο οποίο ανήκει.

Για παράδειγμα, θεωρήστε τη δήλωση `int a = 1` στο μπλοκ B_1 . Η εμβέλειά της είναι όλο το B_1, εκτός από εκείνα τα εμφωλιασμένα (μερικά βαθιά) μπλοκ εντός του B_1 που έχουν την δική τους δήλωση του a. Το B_2 που εμφωλιάζεται αμέσως εντός του B_1 , δεν έχει μια δήλωση του a, αλλά το B_3 έχει. Το B_4 δεν έχει μια δήλωση του a, έτσι το μπλοκ B_3 είναι το μόνο μέρος σε ολόκληρο το πρόγραμμα που είναι εκτός της εμβέλειας της δήλωσης του ονόματος a που ανήκει στο B_1. Αυτό σημαίνει ότι η εμβέλεια συμπεριλαμβάνει το B_4 και όλο το B_2 εκτός από το τμήμα του B_2 που είναι εντός του B_3 . Οι εμβέλειες και των πέντε δηλώσεων συνοψίζονται στην Εικόνα 1.11.

Από μια άλλη οπτική γωνία, ας εξετάσουμε την εντολή εξόδου στο μπλοκ B_4 και ας συνδέσουμε τις μεταβλητές a και b που χρησιμοποιούνται εκεί με τις κατάλληλες δηλώσεις. Η λίστα των περιστοιχιζόμενων μπλοκ, με σειρά αυξανόμενου μεγέθους, είναι η B_4, B_2, B_1. Σημειώστε ότι το B_3 δεν περιβάλλει το εξεταζόμενο σημείο. Το B_4 έχει μια δήλωση του b, έτσι η χρήση του b αναφέρεται σε αυτή την δήλωση και η τιμή του b που τυπώνεται είναι το 4. Παρ' όλα αυτά, το B_4 δεν έχει μια δήλωση του a και έτσι εξετάζουμε το B_2. Ούτε αυτό το μπλοκ έχει μια δήλωση του a, και έτσι προχωράμε στο B_1.

ΔΗΛΩΣΗ	ΕΜΒΕΛΕΙΑ
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

Εικόνα 1.11: Εμβέλειες των δηλώσεων του Παραδείγματος 1.6

Ευτυχώς, υπάρχει μια δήλωση `int a = 1` που ανήκει σε αυτό το μπλοκ, έτσι η τιμή του a που τυπώνεται είναι το 1. Αν δεν υπήρχε αυτή η δήλωση το πρόγραμμα θα ήταν λανθασμένο.

1.6.4 Αποκλειστικός Έλεγχος Προσπέλασης

Οι κλάσεις και οι δομές εισάγουν μια νέα εμβέλεια για τα μέλη τους. Αν το p είναι ένα αντικείμενο μιας κλάσης με ένα πεδίο (μέλος) x, τότε η χρήση του x στο $p.x$ αναφέρεται στο πεδίο x στον ορισμό της κλάσης. Σε αναλογία με τη δομή μπλοκ, η εμβέλεια μιας δήλωσης x σε μια κλάση C επεκτείνεται σε οποιαδήποτε υποκλάση C' εκτός αν η C' έχει μια τοπική δήλωση του ιδίου ονόματος x.

Με τη χρήση λέξεων κλειδιά όπως public, private και protected οι αντικειμενοστρεφείς γλώσσες όπως η C++ ή η Java παρέχουν ρητό έλεγχο στην προσπέλαση ονομάτων μελών σε μια υπερκλάση (superclass). Αυτές οι λέξεις κλειδιά υποστηρίζουν *ενθυλάκωση* περιορίζοντας την πρόσβαση. Έτσι, στα ιδιωτικά (private) ονόματα δίνεται σκοπίμως μια εμβέλεια που συμπεριλαμβάνει μόνο τις δηλώσεις μεθόδων και τους ορισμούς που συσχετίζονται με την κλάση αυτή και κάθε «φίλη»

Δηλώσεις και Ορισμοί

Οι φαινομενικά παρόμοιοι όροι «δήλωση» και «ορισμός» σε έννοιες που σχετίζονται με γλώσσες προγραμματισμού είναι στην πραγματικότητα αρκετά διαφορετικοί. Οι δηλώσεις μας πληροφορούν για τους τύπους των πραγμάτων, ενώ οι ορισμοί για τις τιμές τους. Έτσι το `int i` είναι μια δήλωση του *i*, ενώ το `i = 1` είναι ένας ορισμός του *i*.

Η διαφορά είναι πιο σημαντική όταν ασχολούμαστε με μεθόδους ή άλλες διαδικασίες. Στη C++, μια μέθοδος δηλώνεται στον ορισμό της κλάσης, δίνοντας τους τύπους των ορισμάτων και το αποτέλεσμα της μεθόδου (συχνά αποκαλείται η *υπογραφή* της μεθόδου). Η μέθοδος κατόπιν ορίζεται, δηλ. ο κώδικας που εκτελείται, σε άλλο μέρος. Παρόμοια είναι σύνηθες να ορίζεται μια συνάρτηση C σε ένα αρχείο και να δηλώνεται σε άλλα αρχεία όπου η συνάρτηση χρησιμοποιείται.

(friend) κλάση (ο όρος της C++). Τα προστατευμένα (protected) ονόματα είναι προσπελάσιμα από τις υποκλάσεις. Τα δημόσια (public) ονόματα είναι προσπελάσιμα και από όλο τον κώδικα που βρίσκεται εκτός της κλάσης.

Στη C++, ένας ορισμός κλάσης μπορεί να διαχωρίζεται από τους ορισμούς μερικών ή όλων των μεθόδων της. Έτσι, ένα όνομα *x* που σχετίζεται με την κλάση *C* μπορεί να έχει μια περιοχή κώδικα που είναι εκτός της εμβέλειάς του, ακολουθούμενη από μια άλλη περιοχή (έναν ορισμό μεθόδου) που είναι εντός της εμβέλειάς του. Στην πραγματικότητα, οι περιοχές μέσα και έξω από την εμβέλεια μπορεί να εναλλάσσονται, μέχρι όλες οι μέθοδοι να έχουν οριστεί.

1.6.5 Δυναμική Εμβέλεια

Από τεχνικής άποψης κάθε πολιτική εμβέλειας είναι δυναμική αν βασίζεται σε παράγοντα(ες) που είναι γνωστός(οί) μόνο όταν το πρόγραμμα εκτελείται. Ο όρος *δυναμική εμβέλεια*, παρ' όλα αυτά συνήθως αναφέρεται στην ακόλουθη πολιτική: μια χρήση ενός ονόματος *x* αναφέρεται στη δήλωση του *x* στην πιο πρόσφατα κληθείσα μη-τερματισμένη διαδικασία που διαθέτει μια τέτοια δήλωση. Δυναμική εμβέλεια αυτού του τύπου συμβαίνει μόνο σε ειδικές καταστάσεις. Θα εξετάσουμε δύο παραδείγματα δυναμικών πολιτικών: επέκταση μακροεντολών σε έναν προεπεξεργαστή C και αναζήτηση μεθόδου στον αντικειμενοστρεφή προγραμματισμό.

Παράδειγμα 1.7: Στο πρόγραμμα C της Εικόνας 1.12, ο προσδιοριστής *a* είναι μια μακροεντολή που αντιπροσωπεύει την έκφραση (*x* + 1). Αλλά τι είναι το *x;* Δεν μπορούμε να αποφασίσουμε για το *x* στατικά, δηλαδή σε όρους πηγαίου προγράμματος.

Στην πραγματικότητα, προκειμένου να ερμηνεύσουμε το *x*, πρέπει να χρησιμοποιήσουμε το συνηθισμένο κανόνα δυναμικής εμβέλειας. Εξετάζουμε όλες τις κλήσεις συναρτήσεων που είναι ενεργές εκείνη τη στιγμή και παίρνουμε την πιο πρόσφατα κληθείσα συνάρτηση που έχει μια δήλωση του *x*. Η χρήση του *x* αναφέρεται σε αυτή τη δήλωση.

Αναλογία Ανάμεσα σε Στατική και Δυναμική Εμβέλεια

Ενώ θα μπορούσε να υπάρχει οποιοσδήποτε αριθμός στατικών και δυναμικών πολιτικών για προσδιορισμό της εμβέλειας, υπάρχει μια ενδιαφέρουσα σχέση ανάμεσα στον κανονικό (δομημένο με μπλοκ) κανόνα στατικής εμβέλειας και την κανονική δυναμική πολιτική. Υπό μια έννοια ο δυναμικός κανόνας είναι για το χρόνο ότι ο στατικός κανόνας για το χώρο. Ενώ ο στατικός κανόνας μας ζητά να βρούμε τη δήλωση της οποίας η μονάδα (μπλοκ) περιβάλλει όσο το δυνατόν πιο στενά τη φυσική θέση της χρήσης, ο δυναμικός κανόνας μας ζητά να βρούμε τη δήλωση της οποίας η μονάδα (κλήση διαδικασίας) περιβάλλει όσο το δυνατόν πιο στενά το χρόνο της χρήσης.

Στο παράδειγμα της Εικόνας 1.12, η συνάρτηση *main()* καλεί τη συνάρτηση *b*. Καθώς η *b* εκτελείται, τυπώνει την τιμή της μακροεντολής *a*. Καθώς το (*x* + 1) πρέπει να αντικαταστήσει το *a*, αντιστοιχούμε αυτή τη χρήση του *x* στη δήλωση `int x=1` printf στην *b* αναφέρεται σε αυτό το *x*. Έτσι η τιμή που τυπώνεται είναι το 2.

```
#define a (x+1)

int x = 2;

void b() { int x = 1; printf("%d\n", a); }

void c() { printf("%d\n", a); }

void main() { b(); c(); }
```

Εικόνα 1.12: Μια μακροεντολή της οποίας οι εμβέλειες των ονομάτων πρέπει να υπολογιστούν δυναμικά

Αφού τελειώσει η *b* και κληθεί η *c*, χρειάζεται πάλι να τυπώσουμε την τιμή της μακροεντολής *a*. Παρ' όλα αυτά, το μόνο *x* που είναι προσβάσιμο στη *c* είναι το καθολικό (global) *x*. Η εντολή printf στην *c* αναφέρεται σε αυτή τη δήλωση του *x* και τυπώνεται η τιμή 3.

Η ανάλυση δυναμικής εμβέλειας είναι επίσης ουσιαστική για πολυμορφικές διαδικασίες, δηλαδή εκείνες που έχουν δύο ή περισσότερους ορισμούς για το ίδιο όνομα και η απόφαση κλήσης εξαρτάται μόνο από τους τύπους των ορισμάτων. Σε μερικές γλώσσες όπως η ML (δείτε ενότητα 7.3.3), είναι δυνατόν να αποφασιστούν στατικά τύποι για όλες τις χρήσεις των ονομάτων. Σε αυτή την περίπτωση, ο μεταγλωττιστής μπορεί να αντικαταστήσει κάθε χρήση ενός ονόματος μιας διαδικασίας *p* με μια αναφορά στον κώδικα που αντιστοιχεί στην κατάλληλη μορφή της διαδικασίας. Παρ' όλα αυτά, σε άλλες γλώσσες, όπως η Java και η C++, υπάρχουν φορές που ο μεταγλωττιστής δεν μπορεί να πάρει αυτή την απόφαση.

Παράδειγμα 1.8: Ένα ξεχωριστό χαρακτηριστικό του αντικειμενοστρεφούς προγραμματισμού είναι η δυνατότητα κάθε αντικειμένου να καλεί την κατάλληλη μέ-

θοδο ως απάντηση σε ένα μήνυμα. Με άλλα λόγια, η διαδικασία που καλείται όταν εκτελείται το *x.m()* εξαρτάται από την κλάση του αντικειμένου που υποδηλώνεται από το *x* την χρονική αυτή στιγμή. Ένα χαρακτηριστικό παράδειγμα είναι το επόμενο:

1. Υπάρχει μια κλάση *C* με μια μέθοδο που ονομάζεται *m()*.
2. Η *D* είναι μια υποκλάση της *C* και η *D* έχει τη δική της μέθοδο που ονομάζεται *m()*.
3. Υπάρχει μια χρήση του *m* στην μορφή *x.m()*, όπου το *x* είναι ένα αντικείμενο της κλάσης *C*.

Κανονικά, είναι αδύνατο διακρίνουμε σε χρόνο μεταγλώττισης, αν το *x* θα είναι της κλάσης *C* ή της υποκλάσης *D*. Αν η εφαρμογή της μεθόδου συμβαίνει πολλές φορές, είναι πολύ πιθανό ότι μερικές θα είναι σε αντικείμενα που υποδηλώνονται από το *x* που είναι της κλάσης *C*, ενώ άλλα θα είναι της κλάσης *D*. Μέχρι και το χρόνο εκτέλεσης δεν μπορεί να αποφασιστεί ποιος είναι ο σωστός ορισμός του *m*. Έτσι ο κώδικας που παράγεται από το μεταγλωττιστή πρέπει να αποφασίσει την κλάση του αντικειμένου *x* και να καλέσει τη μια ή την άλλη μέθοδο που ονομάζεται *m*.

1.6.6 Μηχανισμοί Περάσματος Παραμέτρων

Όλες οι γλώσσες προγραμματισμού έχουν την έννοια της διαδικασίας μπορεί όμως να διαφέρουν στον τρόπο με τον οποίο αυτές οι διαδικασίες παίρνουν τα ορίσματά τους. Σε αυτή την ενότητα θα εξετάσουμε πως οι *πραγματικές παράμετροι* (οι παράμετροι που χρησιμοποιούνται στην κλήση της διαδικασίας) συσχετίζονται με τις *τυπικές παραμέτρους* (εκείνες που χρησιμοποιούνται στον ορισμό της συνάρτησης). Ανάλογα με το μηχανισμό που χρησιμοποιείται, προσδιορίζεται πως ο κώδικας κλήσεων (calling-sequence code) μεταχειρίζεται τις παραμέτρους. Η πλειοψηφία των γλωσσών προγραμματισμού χρησιμοποιούν είτε «πέρασμα με τιμή» είτε «πέρασμα με αναφορά» ή και τα δύο. Θα εξηγήσουμε αυτούς τους όρους καθώς και μια άλλη μέθοδο που είναι γνωστή ως «πέρασμα με όνομα», κυρίως για ιστορικούς λόγους.

Πέρασμα με τιμή

Στο *πέρασμα με τιμή* η πραγματική παράμετρος αποτιμάται (αν είναι μια έκφραση) ή αντιγράφεται (αν είναι μια μεταβλητή). Η τιμή τοποθετείται στη θέση που ανήκει στην αντίστοιχη τυπική παράμετρο της κληθείσας διαδικασίας. Αυτός ο τρόπος χρησιμοποιείται στη C και την Java, ενώ υπάρχει αντίστοιχη επιλογή στη C++ όπως επίσης στις περισσότερες άλλες γλώσσες. Το πέρασμα με τιμή έχει ως αποτέλεσμα ότι όλοι οι υπολογισμοί που συμπεριλαμβάνουν τις τυπικές παραμέτρους και γίνονται από την κληθείσα διαδικασία, είναι τοπικοί σε αυτή την διαδικασία με αποτέλεσμα οι πραγματικές παράμετροι δεν μπορούν να αλλάξουν.

Σημειώστε ότι παρ' όλα αυτά στη C μπορούμε να περάσουμε ένα δείκτη σε μια μεταβλητή, για να επιτρέψουμε σε αυτή την μεταβλητή να μπορεί να αλλαχτεί από τον καλούμενο. Παρομοίως, ονόματα πινάκων που περνιούνται ως παράμετροι

στη C, τη C++ ή την Java δίνουν στην κληθείσα διαδικασία κάτι που ουσιαστικά είναι ένας δείκτης ή αναφορά στον ίδιο τον πίνακα. Έτσι, αν *a* είναι το όνομα ενός πίνακα της καλούμενης διαδικασίας και περνιέται με τιμή στην αντίστοιχη τυπική παράμετρο *x,* τότε μια ανάθεση όπως η `x[i] = 2` αλλάζει πραγματικά το στοιχείο του πίνακα *a*[i]. Ο λόγος είναι ότι παρόλο που το *x* παίρνει ένα αντίγραφο της τιμής του *a*, αυτή η τιμή είναι στην πραγματικότητα ένας δείκτης στην αρχή της περιοχής αποθήκευσης όπου είναι αποθηκευμένος ο πίνακας με όνομα *a*.

Παρομοίως, στην Java, πολλές μεταβλητές είναι στην πραγματικότητα αναφορές ή δείκτες προς τα πράγματα που αντιπροσωπεύουν. Αυτή η παρατήρηση εφαρμόζεται σε πίνακες, συμβολοσειρές και αντικείμενα όλων των κλάσεων. Αν και η Java χρησιμοποιεί αποκλειστικά πέρασμα με τιμή, όποτε περνάμε το όνομα ενός αντικειμένου στην κληθείσα διαδικασία, η τιμή που λαμβάνεται από αυτή την διαδικασία είναι στην πραγματικότητα ένας δείκτης προς το αντικείμενο. Έτσι, η κληθείσα διαδικασία μπορεί να επηρεάσει την τιμή του ίδιου του αντικειμένου.

Πέρασμα με αναφορά

Στο *πέρασμα με αναφορά* η διεύθυνση της πραγματικής παραμέτρου περνιέται στην καλούμενη διαδικασία ως η τιμή της αντίστοιχης τυπικής παραμέτρου. Χρήσεις της τυπικής παραμέτρου μέσα στον κώδικα της καλούμενης διαδικασίας υλοποιούνται ακολουθώντας το δείκτη προς τη θέση μνήμης που δηλώνεται από τον καλούντα. Έτσι οι αλλαγές στην τυπική παράμετρο εμφανίζονται ως αλλαγές στην πραγματική παράμετρο.

Αν παρ' όλα αυτά, η πραγματική παράμετρος είναι μια έκφραση, τότε η έκφραση αποτιμάται πριν την κλήση και η τιμή της αποθηκεύεται σε μια δική της θέση μνήμης. Οι αλλαγές στην τυπική παράμετρο αλλάζουν την τιμή αυτής της θέσης μνήμης, αλλά δεν έχουν καμιά επίδραση στα δεδομένα του καλούντα.

Το πέρασμα με αναφορά χρησιμοποιείται για τις παραμέτρους «ref» στη C++ και είναι μια επιλογή σε πολλές άλλες γλώσσες. Είναι σχεδόν αναγκαία όταν η τυπική παράμετρος είναι ένα μεγάλο αντικείμενο, πίνακας ή δομή. Ο λόγος είναι ότι το πέρασμα με τιμή απαιτεί από τον καλούντα να αντιγράψει ολόκληρη την πραγματική παράμετρο στη θέση μνήμης της αντίστοιχης τυπικής παραμέτρου. Όταν η παράμετρος είναι μεγάλη, η αντιγραφή αυτή κοστίζει. Όπως σημειώθηκε όταν εξετάσαμε το πέρασμα με τιμή, γλώσσες όπως η Java λύνουν το πρόβλημα του περάσματος πινάκων, συμβολοσειρών ή άλλων αντικειμένων αντιγράφοντας μόνο την αναφορά προς αυτά τα αντικείμενα. Το αποτέλεσμα είναι ότι η Java συμπεριφέρεται σαν να χρησιμοποιεί πέρασμα με αναφορά για οτιδήποτε άλλο εκτός από βασικούς τύπους όπως ένας ακέραιος ή ένας πραγματικός.

Πέρασμα με όνομα

Ένας τρίτος μηχανισμός – το πέρασμα με όνομα – χρησιμοποιούνταν στην Algol 60, μία από τις πρώτες γλώσσες προγραμματισμού. Απαιτεί η καλούμενη διαδικασία να εκτελεί σαν η πραγματική παράμετρος να υποκαθιστούσε στην κυριολεξία την τυπική παράμετρο στον κώδικα της καλούμενης διαδικασίας. Δηλαδή, σαν η τυπική παράμετρος να ήταν μια μακροεντολή που αντιπροσωπεύει την πραγματική παράμετρο (με μετονομασία των τοπικών ονομάτων στην καλούμενη διαδι-

κασία, για να τα διατηρήσει διακριτά). Όταν η πραγματική παράμετρος είναι μια έκφραση αντί για μία μεταβλητή, μπορεί να συμβαίνουν μερικές απρόβλεπτες συμπεριφορές, που είναι και ένας λόγος που ο μηχανισμός αυτός δεν προτιμάται σήμερα.

1.6.7 Ψευδωνυμία

Υπάρχει μια ενδιαφέρουσα συνέπεια του περάσματος παραμέτρου με αναφορά ή της προσομοίωσής της, όπως στην Java, όπου αναφορές σε αντικείμενα περνιούνται με τιμή. Είναι πιθανόν δύο τυπικές παράμετροι να αναφέρονται στην ίδια θέση μνήμης. Αυτές οι μεταβλητές λέγεται ότι είναι *ψευδώνυμα* η μια της άλλης. Ως αποτέλεσμα, οποιεσδήποτε δύο μεταβλητές που μπορεί να εμφανίζονται να παίρνουν τις τιμές τους από δύο διακριτές τυπικές παραμέτρους, μπορούν επίσης να γίνουν ψευδώνυμα η μια της άλλης.

Παράδειγμα 1.9: Υποθέστε ότι το *a* είναι ένας πίνακας που ανήκει σε μια διαδικασία *p* και η *p* καλεί μια άλλη διαδικασία *q(x,y)* με μια κλήση *q(a,a)*. Υποθέστε επίσης ότι οι παράμετροι περνιούνται με τιμή, όμως τα ονόματα των πινάκων είναι στην πραγματικότητα αναφορές στη θέση μνήμης όπου είναι αποθηκευμένος ο πίνακας, όπως στη C ή σε παρόμοιες γλώσσες. Τώρα τα *x* και *y* έχουν γίνει ψευδώνυμο το ένα του άλλου. Το σημαντικό σημείο είναι ότι αν εντός της *q* υπάρχει μια ανάθεση `x[10] = 2`, τότε η τιμή `y[10]` γίνεται επίσης 2.

Αποδεικνύεται ότι η κατανόηση της ψευδωνυμίας και των μηχανισμών που τη δημιουργούν είναι ουσιώδεις αν ένας μεταγλωττιστής πρόκειται να βελτιστοποιήσει ένα πηγαίο πρόγραμμα. Όπως θα δούμε ξεκινώντας στο Κεφάλαιο 9, υπάρχουν πολλές περιπτώσεις όπου μπορούμε να βελτιστοποιήσουμε κώδικα εάν είμαστε βέβαιοι ότι συγκεκριμένες μεταβλητές δεν έχουν ψευδώνυμα. Για παράδειγμα, θα μπορούσαμε να καθορίσουμε ότι το `x = 2` είναι το μοναδικό μέρος όπου η μεταβλητή *x* παίρνει τιμή. Αν είναι έτσι, τότε μπορούμε να αντικαταστήσουμε τη χρήση του *x* με τη χρήση του 2. Για παράδειγμα, να αντικαταστήσουμε το `a = x+3` από το απλούστερο `a = 5`. Τώρα υποθέστε ότι υπήρχε μια άλλη μεταβλητή *y* που ήταν ψευδώνυμο του *x*. Τότε μια ανάθεση `y = 4` θα μπορούσε να έχει την απρόσμενη επίδραση αλλαγής του *x*. Θα μπορούσε επίσης να σημαίνει ότι αντικαθιστώντας το `a = x+3` από το `a = 5` θα ήταν λάθος. Σε αυτή την περίπτωση η κατάλληλη τιμή του *a* θα μπορούσε να είναι 7.

1.6.8 Ασκήσεις Ενότητας 1.6

Άσκηση 1.6.1: Για το δομημένο σε μπλοκ κώδικα C της Εικόνας 1.13(α), προσδιορίστε τις τιμές που τίθενται στο *w*, *x*, *y*, και *z*.

Άσκηση 1.6.2: Επαναλάβετε την Άσκηση 1.6.1 για τον κώδικα της Εικόνας 1.13(β).

Άσκηση 1.6.3: Για το δομημένο σε μπλοκ κώδικα της Εικόνας 1.14, υποθέτοντας τη συνήθη στατική εμβέλεια των δηλώσεων, δώστε την εμβέλεια για κάθε μια από τις δώδεκα δηλώσεις.

```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j
{
    int i = 8;
    y = i + j;
}
z = i + j;
```

(α) Κώδικας Άσκησης 1.6.1

```
int w, x, y, z;
int i = 3; int j = 4;
{   int i = 5;
    w = i + j;
}
x = i + j
{
    int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;
```

(β) Κώδικας Άσκησης 1.6.2

Εικόνα 1.13: Κώδικας δομημένος σε μπλοκ

```
{  int w, x, y, z;     /* Μπλοκ B1 */
   {   int x, z;       /* Μπλοκ B2 */
       {   int w, x;   /* Μπλοκ B3 */ }
   }
   {
       int w, x;       /* Μπλοκ B4 */
       {   int y, z;   /* Μπλοκ B5 */ }
   }
}
```

Εικόνα 1.14: Κώδικας δομημένος σε μπλοκ για την Άσκηση 1.6.3

Άσκηση 1.6.4: Τι τυπώνεται από τον ακόλουθο κώδικα σε C;

```
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() {b(); c();}
```

1.7 Περίληψη Κεφαλαίου 1

- *Επεξεργαστές γλωσσών.* Ένα ολοκληρωμένο περιβάλλον ανάπτυξης λογισμικού που περιλαμβάνει πολλά διαφορετικά είδη επεξεργαστών γλωσσών όπως οι μεταγλωττιστές, οι διερμηνευτές, επεξεργαστές συμβολικής γλώσσας, διασυνδετές, αποσφαλματωτές, εργαλεία ανάλυσης απόδοσης
- *Φάσεις Μεταγλωττιστή.* Ένας μεταγλωττιστής λειτουργεί ως μια ακολουθία από φάσεις, καθεμία από τις οποίες μετασχηματίζει το πηγαίο πρόγραμμα από μια ενδιάμεση αναπαράσταση σε μια άλλη.

- *Γλώσσες μηχανής και Συμβολικές γλώσσες.* Οι γλώσσες μηχανής ανήκουν στην πρώτη γενιά γλωσσών προγραμματισμού, ακολουθούμενες από τις συμβολικές γλώσσες. Ο προγραμματισμός σε αυτές τις γλώσσες ήταν χρονοβόρος και επιρρεπής σε λάθη.
- *Μοντελοποίηση στον Σχεδιασμό Μεταγλωττιστή.* Ο σχεδιασμός μεταγλωττιστών είναι μια από τις περιπτώσεις όπου η θεωρία έχει τη μεγαλύτερη δυνατή πρακτική επίδραση. Θεωρητικά μοντέλα που έχουν βρεθεί χρήσιμα στην πράξη συμπεριλαμβάνουν αυτόματα, γραμματικές, κανονικές εκφράσεις, δέντρα και πολλά άλλα.
- *Βελτιστοποίηση κώδικα.* Αν και ο κώδικας δεν μπορεί να «βελτιστοποιηθεί» επακριβώς, η επιστήμη της βελτίωσης της αποδοτικότητας του κώδικα είναι και πολύπλοκη και πολύ σημαντική. Είναι ένα μεγάλο τμήμα της μελέτης της μεταγλώττισης.
- *Γλώσσες Υψηλότερου-Επιπέδου.* Με το πέρασμα του χρόνου, οι γλώσσες προγραμματισμού αναλαμβάνουν σταδιακά τις περισσότερες από τις εργασίες που παλαιότερα αφήνονταν στον προγραμματιστή, όπως η διαχείριση μνήμης, έλεγχος συνέπειας τύπων ή παράλληλη εκτέλεση κώδικα.
- *Μεταγλωττιστές και Αρχιτεκτονικές Υπολογιστών.* Η τεχνολογία των μεταγλωττιστών επηρεάζει την αρχιτεκτονική των υπολογιστών ενώ ταυτόχρονα επηρεάζεται από την πρόοδο στην αρχιτεκτονική. Πολλές σύγχρονες καινοτομίες στην αρχιτεκτονική εξαρτώνται από την ικανότητα των μεταγλωττιστών να εξάγουν από τα πηγαία προγράμματα τις ευκαιρίες να χρησιμοποιήσουν αποδοτικά τις δυνατότητες του υλικού.
- *Παραγωγικότητα Ανάπτυξης και Ασφάλεια Λογισμικού.* Η ίδια τεχνολογία που επιτρέπει στους μεταγλωττιστές να βελτιστοποιούν κώδικα μπορεί να χρησιμοποιηθεί για μια ποικιλία εργασιών ανάλυσης προγραμμάτων, από τον εντοπισμό συνηθισμένων σφαλμάτων προγράμματος έως την ανακάλυψη ότι ένα πρόγραμμα είναι τρωτό σε ένα από τα πολλά είδη εισβολών που οι «χάκερς» έχουν ανακαλύψει.
- *Κανόνες Εμβέλειας.* Η εμβέλεια μιας δήλωσης του x είναι το απόσπασμα του πηγαίου προγράμματος στο οποίο χρήσεις του x αναφέρονται σε αυτή τη δήλωση. Μια γλώσσα χρησιμοποιεί *στατική εμβέλεια* ή *λεξικογραφική εμβέλεια* αν είναι εφικτό να αποφασίσει την εμβέλεια μιας δήλωσης κοιτάζοντας μόνο το πρόγραμμα. Διαφορετικά, η γλώσσα χρησιμοποιεί *δυναμική εμβέλεια.*
- *Περιβάλλοντα.* Ο συσχετισμός των ονομάτων με θέσεις μνήμης και κατόπιν με τιμές μπορεί να περιγραφεί σε όρους *περιβαλλόντων*, που αντιστοιχίζουν ονόματα σε θέσεις μνήμης και *καταστάσεων* που αντιστοιχίζουν θέσεις μνήμης στις τιμές τους.
- *Δομή Μπλοκ.* Γλώσσες που επιτρέπουν των εμφωλιασμό μπλοκ λέγεται ότι έχουν *δομή μπλοκ*. Ένα όνομα x σε ένα εμφωλιασμένο μπλοκ B είναι στην εμβέλεια μιας δήλωσης του x στο D σε ένα περιβάλλον μπλοκ αν δεν υπάρχει καμία άλλη δήλωση του x στο ενδιάμεσο εσωτερικό μπλοκ.

- *Πέρασμα Παραμέτρων.* Οι παράμετροι περνιόνται από την καλούσα διαδικασία στην καλούμενη διαδικασία είτε με τιμή είτε με αναφορά. Όταν περνιόνται μεγάλα αντικείμενα με τιμή, οι τιμές που περνιόνται είναι στην πραγματικότητα αναφορές προς τα ίδια τα αντικείμενα, έχοντας ως αποτέλεσμα ένα αποδοτικό πέρασμα με αναφορά.
- *Ψευδωνυμία.* Όταν οι παράμετροι (στην πραγματικότητα) περνιόνται με αναφορά, δύο τυπικές παράμετροι μπορεί να αναφέρονται στο ίδιο αντικείμενο. Αυτή η πιθανότητα επιτρέπει μια αλλαγή σε μια μεταβλητή να αλλάζει την άλλη.

1.8 Αναφορές Κεφαλαίου 1

Για την ανάπτυξη των γλωσσών προγραμματισμού που έχουν δημιουργηθεί και είναι σε χρήση από το 1967, συμπεριλαμβανομένων των Fortran, Algol, Lisp και Simula, δείτε το [7]. Για γλώσσες που δημιουργήθηκαν μετά από το 1982, συμπεριλαμβανομένων των C, C++, Pascal και Smalltalk, δείτε το [1].

Η Συλλογή Μεταγλωττιστών GNU, gcc, είναι μια δημοφιλής πηγή μεταγλωττιστών ανοιχτού λογισμικού για τη C, την C++, τη Fortran, την Java και άλλες γλώσσες [2]. Το Phoenix είναι μια εργαλειοθήκη κατασκευής μεταγλωττιστών που παρέχει ένα ολοκληρωμένο πλαίσιο για τη δημιουργία των φάσεων της ανάλυσης προγράμματος, παραγωγής κώδικα και βελτιστοποίησης κώδικα των μεταγλωττιστών που συζητούνται σε αυτό το βιβλίο [3].

Για περισσότερες πληροφορίες σχετικά με έννοιες γλωσσών προγραμματισμού, συστήνουμε τα [5,6]. Για περισσότερα σχετικά με την αρχιτεκτονική υπολογιστών και πως επηρεάζει την μεταγλώττιση, προτείνουμε το [4].

1. Bergin, T.J. and R.G. Gibson, *History of Programming Languages*, ACM Press, New York, 1996.
2. http://gcc.gnu.org/ .
3. http://research.microsoft.com/phoenix/default.aspx .
4. Hennesy, J. L. and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface,* Morgan-Kaufmann, San Francisco, CA, 2004.
5. Scott, M. L., *Programming Language Pragmatics, second edition,* Morgan-Kaufmann, San Francisco, CA, 2006.
6. Sethi, R., *Programming Languages: Concepts and Constructs, Addison-Wesley, 1996.*
7. Wexeblat, R. L., *History of Programming Languages,* Academic Press, New York, 1981.