Τ.Ε.Ι ΗΠΕΙΡΟΥ

Τ.Ε.Ι OF EPIRUS

ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ & ΟΙΚΟΝΟΜΙΑΣ
ΤΜΗΜΑ ΤΗΛΕΠΛΗΡΟΦΟΡΙΚΗΣ & ΔΙΟΙΚΗΣΗΣ

SCHOOL OF MANAGEMENT AND ECONOMICS
DEPARTMENT OF COMMUNICATIONS INFORMATICS AND MANAGEMENT

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

"ΥΛΟΠΟΙΗΣΗ ΕΝΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΟΥ ΚΕΛΥΦΟΥΣ ΕΡΓΑΣΙΑΣ"

Γιαννακίδης Αποστόλης

Επιβλέπων καθηγητής:
Τσούλος Ιωάννης

Άρτα, Σεπτέμβριος 2005

Γιαννακίδης Αποστόλης

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

"Υλοποίηση ενός προγραμματιστικού κελύφους εργασίας"

ΤΕΙ ΗΠΕΙΡΟΥ

Στον πατέρα μου Τάσο,
στην μητέρα μου Σούλα και
στον αδερφό μου Μαρίνο.

Ευχαριστίες

Οφείλω πρώτα απ’ όλα ένα μεγάλο ευχαριστώ σε όλους τους καθηγητές μου που είχανε το κουράγιο και την υπομονή να μου μεταφέρουνε τις γνώσεις τους. Μόνο θετικές αναμνήσεις έχω απ’ όλους τους. Ένα θερμό ευχαριστώ στον επιβλέποντα καθηγητή μου, Γιάννη Τσούλο, για την ουσιαστική καθοδήγησή του και την συνεχή υποστήριξη που οδήγησαν στην ολοκλήρωση αυτής της πτυχιακής εργασίας.

Από καρδιάς θέλω να ευχαριστήσω τον Γιώργο. Τον ευχαριστώ γιατί έτσι και αλλιώς είναι πάντα ο Γιώργος.

Ένα ξεχωριστό ευχαριστώ από καρδιάς θέλω να πω και στην Ελίνα την ξαδέρφη μου για την πρόθυμη και ουσιαστική βοήθειά της.

Ευχαριστώ πολύ τον Μιχάλη, τον ψηλό, τον Άλκη, τον Γιάννη, την Φωτεινή, την Ελένη, την Μαργαρίτα, τον Ανδρέα, την Έμη και την Γεωργία. Να είστε καλά παιδιά.

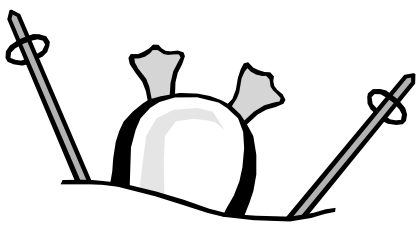
Ευχαριστώ πολύ τον Τίμο για την φιλία του, για τον ενθουσιασμό του, για τις χρήσιμες υποδείξεις του καθώς και για τις τεχνικές συζητήσεις μας που διεύρυναν τους προβληματισμούς μου.

Τέλος, χρωστάω ολόψυχα ένα μεγάλο ευχαριστώ στους γονείς μου, για την συμπαράστασή τους και την κατανόησή τους. Συναισθάνομαι το μέγεθος της ψυχικής και σωματικής κόπωσης που τους προκάλεσα. Η συγκεκριμένη εργασία είναι φυσικά αφιερωμένη και στους δύο καθώς και στον αδερφό μου.

Πίνακας Περιεχομένων

ΚΕΦΑΛΑΙΟ 1ο

Έννοιες λειτουργικών συστημάτων



Εισαγωγή

Σε αυτό το εισαγωγικό κεφάλαιο θα ξεκινήσουμε κάνοντας μία γενική ανασκόπηση σε βασικές έννοιες των Λειτουργικών Συστημάτων καθώς και στον τρόπο υλοποίησής τους. Μία καλή κατανόηση αυτών των εννοιών είναι σημαντική για την κατανόηση των θεμάτων που παρουσιάζονται στα επόμενα κεφάλαια. Περιληπτικά στο κεφάλαιο αυτό θα ασχοληθούμε με τις βασικές λειτουργίες και τους στόχους των Λειτουργικών Συστημάτων, την δομή τους, θα αναλύσουμε και θα επεξηγήσουμε την έννοια των διεργασιών, της διαδιεργασιακής επικοινωνίας, με τα σήματα και τον τρόπο διαχείρισης τους και τέλος θα αναφερθούμε στο παράδειγμα του Linux.

1.1 ■ Ορισμός Λειτουργικού Συστήματος

Προτού περιγράψουμε τα ζητήματα σχετικά με την υλοποίηση ενός κελύφους εργασίας, είναι αναγκαίο να κατανοήσουμε μερικές έννοιες και μερικούς σημαντικούς ορισμούς καθώς και να κατανοήσουμε την σχέση του Λειτουργικού Συστήματος με το κέλυφος εργασίας (**shell**).

Ένα Λειτουργικό Σύστημα είναι ένα πρόγραμμα το οποίο ενεργεί ανάμεσα από το χρήστη (του υπολογιστή) και το hardware. Ο σκοπός του Λειτουργικού Συστήματος είναι να παρέχει το περιβάλλον στο οποίο ο χρήστης θα μπορεί να εκτελεί προγράμματα με ένα βολικό και αποδοτικό τρόπο[1]. Επίσης σκοπός του Λ.Σ. είναι να καθορίζει τον τρόπο λειτουργίας του υπολογιστικού συστήματος, ελέγχοντας και συντονίζοντας τη χρήση των μονάδων του από τα διάφορα προγράμματα εφαρμογής των χρηστών.

Οι στόχοι ενός ΛΣ είναι[2]:

1. Η διευκόλυνση των χρηστών. Τα ΛΣ υπάρχουν επειδή κάνουν πιο εύκολη τη χρήση των υπολογιστικών συστημάτων και δίνουν τη δυνατότητα σε ανθρώπους με περιορισμένες γνώσεις γύρω από τους υπολογιστές να εκτελέσουν πολύπλοκες εργασίες.

2. Η αποδοτική λειτουργία του υπολογιστικού συστήματος, δηλαδή η όσο το δυνατόν καλύτερη χρησιμοποίηση του υλικού, ώστε να κατανέμεται καλύτερα το υπολογιστικό φορτίο. Το ΛΣ διαθέτει τη «γενική εικόνα» όλων των προγραμμάτων που πρέπει να εκτελεστούν, όλων των χρηστών του υπολογιστικού συστήματος και των αναγκών τους· έτσι, μπορεί να ρυθμίσει καλύτερα πότε και ποια προγράμματα θα εκτελεστούν κλπ.

1.2 ■ Υπηρεσίες Λειτουργικών Συστημάτων

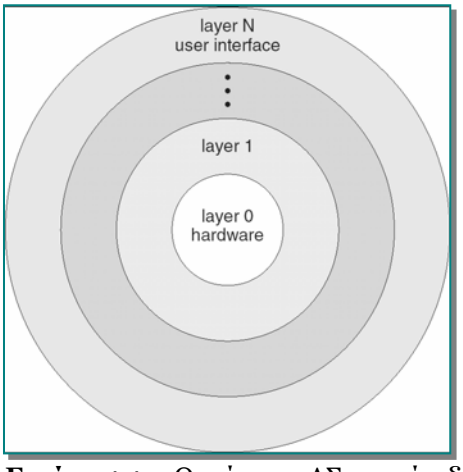
Ένα ΛΣ παρέχει κάποιες σημαντικές υπηρεσίες στον χρήστη και στο υπόλοιπο σύστημα. Οι πιο σημαντικές από αυτές είναι:

Διαχείριση Διεργασιών
- Δημιουργία και καταστολή διεργασιών
- Επικοινωνία και συγχρονισμός διεργασιών
Διαχείριση Μνήμης
- Διαχείριση μνήμης ανάμεσα σε πολλαπλές διεργασίες
- Ανάθεση μνήμης κατόπιν αίτησης από μία διεργασία
Διαχείριση Αρχείων
- Δημιουργία και επεξεργασία αρχείων και καταλόγων
Διαχείριση Συστήματος Εισόδου/Εξόδου
- Διαχείριση συσκευών εισόδου – εξόδου
Διαχείριση Συσκευών Δευτερεύουσας Αποθήκευσης
- Διαχείριση ελεύθερου χώρου σε μονάδες δίσκων
Δικτυακή Υποστήριξη
- Παροχή TCP/IP και σχετικών πρωτοκόλλων (HTTP,FTP,SSL)
Προστασία και Ασφάλεια
- Παροχή πιστοποίησης
- Προστασία από κακόβουλες επιθέσεις μέσω δικτύου.

Πίνακας 1.1 – Υπηρεσίες Λειτουργικών Συστημάτων

1.3 ■ Δομή Λειτουργικών Συστημάτων

Η σχεδίαση ενός ΛΣ μπορεί να γίνει με αρκετούς τρόπους. Η συνηθέστερη δομή ενός Λειτουργικού Συστήματος είναι η οργάνωση σε επίπεδα (**layers**).



Εικόνα 1.1 – Οργάνωση ΛΣ σε επίπεδα

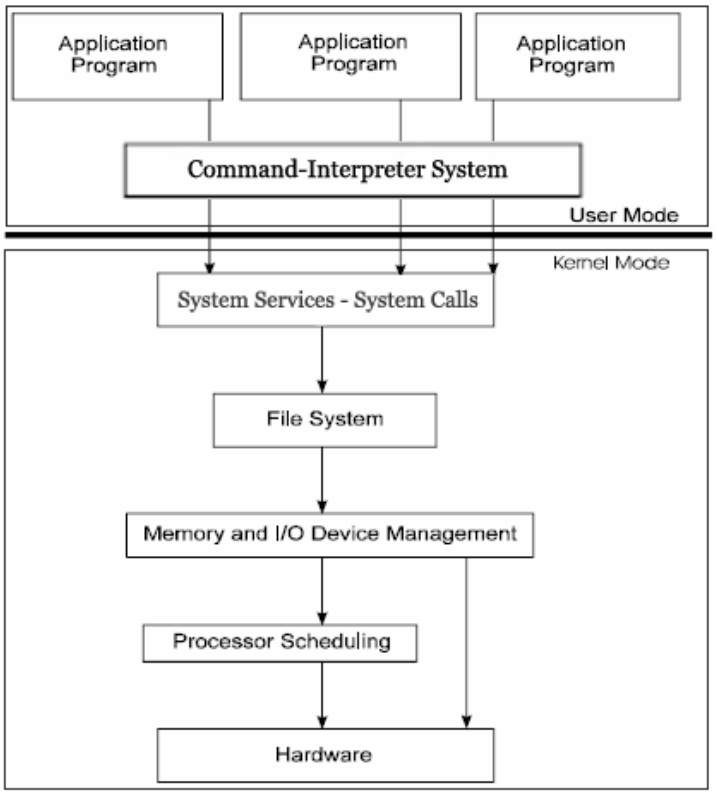
Αυτό σημαίνει ότι κατά τη σχεδίασή τους έχουν διαιρεθεί σε τμήματα, και κάθε τμήμα τους επικοινωνεί μόνο με αυτά που βρίσκονται στο αμέσως ανώτερο ή το αμέσως κατώτερο επίπεδο. Όσα τμήματα χρησιμοποιούν απευθείας το υλικό του υπολογιστή, βρίσκονται στο κατώτερο επίπεδο του ΛΣ. Τα υπόλοιπα τμήματα, που βρίσκονται σε ανώτερα επίπεδα, δεν επικοινωνούν καθόλου με το υλικό, αλλά χρησιμοποιούν τα τμήματα που ανήκουν στο αμέσως κατώτερο επίπεδο. Το κατώτερο επίπεδο (layer 0) είναι το hardware και το ανώτερο επίπεδο (layer N) είναι η **διεπαφή του χρήστη (user interface)**. Την οργάνωση αυτή την βλέπουμε στην εικόνα 1.1.

Τα προγράμματα των χρηστών επικοινωνούν μόνο με το υψηλότερο επίπεδο σε ένα ΛΣ. Η διεπαφή με το χρήστη μπορεί να γίνεται είτε με εντολές,

με το **διερμηνέα εντολών** (**command interpreter, shell**), ή με μία **διεπαφή χρήστη με χρήση γραφικών (Graphical User Interface, GUI)**, είτε με προγράμματα γραμμένα από τον χρήστη.

Το πλεονέκτημα της οργάνωσης σε επίπεδα είναι ο αρθρωτός (**modular**) σχεδιασμός που έχει σαν αποτέλεσμα τον ευκολότερο σχεδιασμό του Λειτουργικού Συστήματος. Η βασική δυσκολία της οργάνωσης σε επίπεδα αφορά το προσεκτικό προσδιορισμό των λειτουργιών του κάθε επιπέδου.

Στο παρακάτω σχήμα φαίνεται ένα παράδειγμα οργάνωσης ΛΣ σε επίπεδα. Η οργάνωση αυτή είναι ενδεικτική γιατί υπάρχουν πολλές παραλλαγές της, αλλά η φιλοσοφία είναι κοινή.



Εικόνα 1. 2 – Παράδειγμα οργάνωσης ενός ΛΣ σε επίπεδα

Πολλά Λειτουργικά Συστήματα περιλαμβάνουν το user interface σαν κομμάτι του ίδιου του ΛΣ. Το Unix είναι κατά κάποιο τρόπο πρωτοποριακό από την άποψη ότι το shell δεν είναι ενσωματωμένο στο ΛΣ (kernel mode) αλλά είναι ένα ξεχωριστό πρόγραμμα στο user mode. Εδώ πρέπει να γίνει η διευκρίνιση ότι το κέλυφος αν και δεν είναι μέρος του Unix (kernel), συμπεριλαμβάνεται στην διανομή του Unix σαν αναπόσπαστο κομμάτι του[3]. Το γεγονός αυτό επέτρεψε στην δημιουργία πολλών ειδών shell για το Unix από διάφορους προγραμματιστές.

Μερικά από τα διαθέσιμα κελύφη του Unix και του Linux είναι τα εξής:

Όνομα Shell	Παρατηρήσεις
sh (Bourne)	**Είναι το πρώτο κέλυφος που δημιουργήθηκε από τις πρώτες εκδόσεις του Unix.**
Bash	**Το bash (Bourne Again Shell) είναι το πιο ευρέως χρησιμοποιούμενο (και πιο δυναμικό) κέλυφος. Είναι συμβατό με το POSIX και δημιουργήθηκε και διανέμεται από το ίδρυμα GNU. Είναι συμβατό με το Bourne Shell. Το bash είναι το εξ' ορισμού κέλυφος σε αρκετές εκδόσεις του Linux.**
ksh, pdksh	**Το κέλυφος Korn και το δωρεάν αδερφικό του shell (pdksh) είναι γραμμένα από τον David Korn και είναι το εξ'ορισμού κέλυφος σε αρκετές εμπορικές εκδόσεις του Unix.**
csh, tcsh, zsh	**Το κέλυφος C αρχικά δημιουργήθηκε από τον Bill Joy στο Μπέρκλεϊ. Το C Shell και τα παράγωγά του (tcsh, zsh) είναι τα πιο διαδεδομένα κελύφη μετά το κέλυφος Korn.**

Πίνακας 1.2 – Μερικά από τα γνωστότερα κελύφη του Unix/Linux

Εκτός από το C shell και μερικά από τα παράγωγά του, όλα τα υπόλοιπα κελύφη είναι αρκετά όμοια και είναι όλα συμβατά με το πρότυπο για τα κελύφη που καθορίζεται το X/Open 4.2 και στο POSIX 1003.2. Το πρότυπο POSIX 1003.2 καθορίζει τις ελάχιστες προδιαγραφές για ένα κέλυφος, αλλά το X/Open 4.2 καθορίζει ένα πιο φιλικό αλλά και δυναμικό shell αλλά είναι πιο απαιτητικό για την δημιουργία τους.

Για να ανακαλύψετε πιο κέλυφος χρησιμοποιείτε δοκιμάστε την ακόλουθη διαταγή:

```
$ echo $SHELL
```

ή

```
$ echo $0
```

Πιθανότατα θα εκτελείτε το κέλυφος bash. Για να αλλάξετε το εξ' ορισμού κέλυφος χρησιμοποιήστε τη διαταγή:

```
$ chsh
```

Πριν κάποιος χρήστης επιλέξει ένα άλλο shell πρέπει το κέλυφος αυτό να είναι εγκατεστημένο και ο διαχειριστής του συστήματος πρέπει να το έχει κάνει διαθέσιμο εισάγοντάς το στο αρχείο /etc/shells.

1.4 ■ Βασικοί παράγοντες υλοποίησης Λειτουργικών Συστημάτων

Στο σχεδιασμό των περισσότερων σύγχρονων Λειτουργικών Συστημάτων χρησιμοποιούνται τρεις βασικοί μηχανισμοί υλοποίησης[4]:

i. **Καταστάσεις του επεξεργαστή**
ii. **Πυρήνας**
iii. **Μέθοδος αίτησης υπηρεσίας του συστήματος**

1.4.1 Καταστάσεις του επεξεργαστή

Στα σύγχρονα Λειτουργικά Συστήματα κάθε διεργασία εκτελείται στο δικό της χώρο μνήμης (**address space**). Χώρος μνήμης χρήστη (**user space)** ονομάζουμε τον χώρο μνήμης στον οποίο εκτελούνται τα προγράμματα του χρήστη και το ίδιο το πρόγραμμα λέμε ότι εκτελείτε σε κατάσταση χρήστη (**user mode)**. Ο χώρος μνήμης στον οποίο εκτελούνται οι υπηρεσίες του πυρήνα μέσω διεργασιών του ίδιου του πυρήνα ονομάζεται χώρος μνήμης πυρήνα (**kernel space)** και οι διεργασίες αυτές λέμε ότι εκτελούνται σε κατάσταση πυρήνα (**kernel mode** - λέγεται επίσης supervisor mode, system mode, monitor mode ή και privileged mode). Η περιοχή μνήμης του πυρήνα (kernel space) είναι μια ιδιαίτερη περιοχή μνήμης στην οποία υπάρχει ο κώδικας του kernel και τα προγράμματα του χρήστη έχουν πρόσβαση μόνο μέσω κλήσεων συστήματος (**System Calls**). Οι δύο καταστάσεις που αναφέραμε παραπάνω (user mode/kernel mode) ονομάζονται καταστάσεις εκτέλεσης της CPU (CPU execution modes ή processor modes). Η κατάσταση εκτέλεσης υποδηλώνεται από ένα bit, το bit κατάστασης (**mode bit**) στο Processor Status Word (ένα καταχωρητή στην CPU). Με το bit κατάστασης μπορούμε να ξεχωρίσουμε αν μία διεργασία (**process**) εκτελείτε εκ μέρους του χρήστη ή του ίδιου του ΛΣ (του πυρήνα). Ιδιαίτερη προσοχή πρέπει να δώσουμε στο γεγονός ότι για να αλλάξει κατάσταση το mode bit, πρέπει να γίνει μέσω **προνομιακών εντολών** (**privileged instruction)**. Διαφορετικά οποιοσδήποτε χρήστης θα μπορούσε να θέσει το mode bit ίσο με 0 και να μπει σε kernel mode!

Όταν το σύστημα ξεκινάει (διαδικασία boot) το hardware ξεκινάει σε system mode (το bit κατάστασης είναι 0). Το Λειτουργικό Σύστημα τότε φορτώνεται στην μνήμη και ξεκινάει να εκτελεί προγράμματα χρηστών σε κατάσταση χρήστη. Το σύστημα πάντα αλλάζει σε κατάσταση χρήστη (αλλάζει το bit κατάστασης σε 1) πριν δώσει τον έλεγχο σε ένα πρόγραμμα χρήστη.

Παλιότεροι επεξεργαστές, όπως ο Intel 8088/8086 δεν διέθεταν bit κατάστασης. Γι' αυτό το λόγο δεν υπήρχε διάκριση μεταξύ προνομιακών εντολών και εντολών χρήστη, με αποτέλεσμα να μην υπάρχει απομόνωση κρίσιμων περιοχών μνήμης. Αυτό έδινε την δυνατότητα σε κάθε εφαρμογή χρήστη να είναι σε θέση να αναφερθεί σε οποιαδήποτε διεύθυνση μνήμης.

Μνήμη που μπορούσε να περιέχει δεδομένα ή και κώδικα του ίδιου του ΛΣ ή άλλης κρίσιμης διεργασίας.

Με την εξέλιξη των επεξεργαστών και των ΛΣ έγινε η χρήση του bit κατάστασης. Έτσι μία διεργασία που εκτελείται σε κατάσταση χρήστη δεν έχει πρόσβαση σε μνήμη πέρα από το δικό της χώρο μνήμης. Αντίθετα μία διεργασία που εκτελείται σε κατάσταση πυρήνα μπορεί να αναφερθεί σε οποιοδήποτε μέρος της μνήμης.

1.4.2 Πυρήνας

Όπως είπαμε παραπάνω, τα κομμάτια του ΛΣ που είναι κρίσιμα για την σωστή λειτουργία τους εκτελούνται σε κατάσταση πυρήνα, ενώ άλλο software εκτελούνται σε κατάσταση χρήστη. Στην ορολογία των Λειτουργικών Συστημάτων το κομμάτι του ΛΣ που εκτελείται σε κατάσταση πυρήνα ονομάζεται **πυρήνας (kernel)** του Λειτουργικού Συστήματος.

Στον πυρήνα είναι υλοποιημένο όλο το βασικό τμήμα του ΛΣ. Ο πυρήνας βρίσκεται φορτωμένος στην μνήμη του υπολογιστή διαρκώς και αναλαμβάνει την διανομή της μνήμης μεταξύ των εφαρμογών (**memory management**) και την διανομή του χρόνου μεταξύ των διεργασιών (**process scheduling**). Επίσης περιλαμβάνει οδηγούς συσκευών (**device drivers**) για δίσκους, κάρτες δικτύου, σειριακές θύρες κλπ, τα διάφορα πρωτόκολλα επικοινωνίας, υποστήριξη συστημάτων αρχείων (**file systems)** και λοιπό κώδικα για την διαχείριση τους. Όλες οι παραπάνω υπηρεσίες που προσφέρει ο πυρήνας είναι στην διάθεση κάθε προγράμματος χρήστη. Με ποιο τρόπο όμως ένα πρόγραμμα ζητάει (κάνει αίτηση) για μία συγκεκριμένη υπηρεσία από τον πυρήνα;

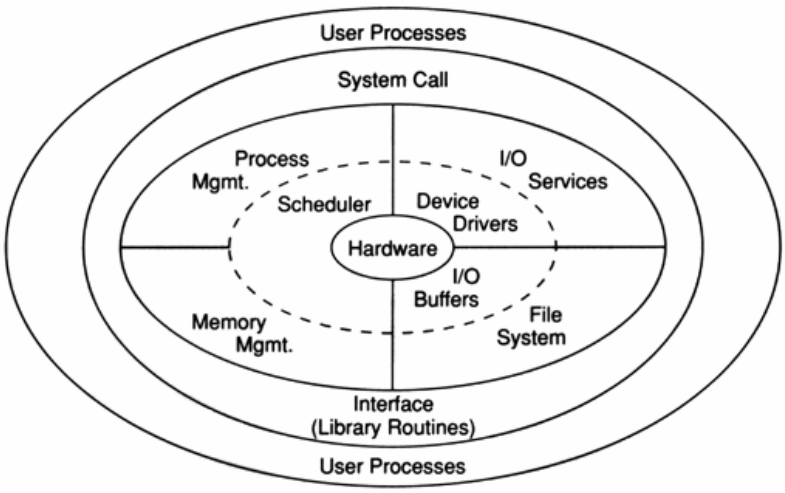
1.4.3 Μέθοδος αίτηση υπηρεσίας του συστήματος

Υπάρχουνε δύο τεχνικές με τις οποίες ένα πρόγραμμα που εκτελείτε σε user mode μπορεί να κάνει χρήση των υπηρεσιών του πυρήνα:

- Κλήσεις συστήματος (System Calls)
- Πέρασμα/ανταλλαγή μηνυμάτων (Message Passing)

1.4.3.1 Κλήσεις Συστήματος

Για να επικοινωνήσει ένα πρόγραμμα χρήστη (το οποίο εκτελείται σε κατάσταση χρήστη) με το ΛΣ χρησιμοποιεί τις διάφορες κλήσεις συστήματος (**System Calls**) τα οποία παρέχει ο ίδιος ο πυρήνας του ΛΣ. Με άλλα λόγια, οι κλήσεις συστήματος είναι η διεπαφή (interface) μεταξύ της κατάστασης χρήστη και της κατάστασης πυρήνα (του ΛΣ στην ουσία). Το σύνολο όλων των κλήσεων συστήματος αποτελούν το **OS API**. Η παρακάτω εικόνα δείχνει την σχέση του ΛΣ, των κλήσεων συστήματος και των προγραμμάτων του χρήστη.



Εικόνα 1. 3 - Παράδειγμα της σχέσης του ΛΣ με τις κλήσεις συστήματος

Ένα πρόγραμμα το οποίο εκτελείται σε κατάσταση μνήμης δεν έχει πρόσβαση στο χώρο μνήμης του πυρήνα και κατά συνέπεια ούτε στις εσωτερικές ρουτίνες του και στις δομές δεδομένων του (data structures). Η πρόσβαση γίνεται ελεγχόμενα μέσω των κλήσεων συστήματος. Αυτή η διαδικασία μέσω των κλήσεων συστήματος γίνετε για λόγους ασφάλειας. Τα προγράμματα των χρηστών δεν πρέπει σε καμία περίπτωση δεν πρέπει να μπορούν να βλάψουν το Λειτουργικό Σύστημα και κατά συνέπεια ούτε να μπορούν να βλάψουν άλλο πρόγραμμα χρήστη.

Ο τρόπος με τον οποίο οι κλήσεις συστήματος επιτυγχάνουν την επικοινωνία με τον πυρήνα και ο τρόπος με τον οποίο κατασκευάζονται είναι εκτός θέματος της συγκεκριμένης εργασίας γι' αυτό το λόγο δεν θα μπούμε σε περισσότερες τεχνικές λεπτομέρειες. Παρόλα αυτά είναι σημαντικό να αναφέρουμε τον τρόπο με τον οποίο δουλεύουμε τις κλήσεις συστήματος από προγραμματιστική μεριά.

Οι κλήσεις συστήματος έχουνε την μορφή απλών συναρτήσεων (functions) και μπορούνε να κληθούν σχεδόν από οποιαδήποτε γλώσσα προγραμματισμού (C/C++/JAVA κλπ). Υπάρχει ένας πολύ μεγάλος αριθμός κλήσεων συστήματος για τις δεκάδες υπηρεσίες του πυρήνα. Κάθε κλήση συστήματος είναι δηλωμένη σε ένα αρχείο επικεφαλίδας (**header file)** το οποίο πρέπει να γίνει include πριν καλέσουμε τη συγκεκριμένη κλήση συστήματος. Για παράδειγμα αν θέλουμε να καλέσουμε τη κλήση συστήματος **read** του Unix θα την καλέσουμε ως εξής[5] :

```
amt = read(fd, buf, numbyte);
```

και στην αρχή του προγράμματός μας θα κάνουμε include το **unistd.h** ως εξής:

```
#include <unistd.h>
```

Σε ένα τυπικό αρχείο επικεφαλίδας είναι δηλωμένες περισσότερες από μία κλήσεις (το unistd.h καθορίζει περίπου 80). Στα Windows οι κλήσεις

συστήματος είναι κομμάτι του **Win32 API** το οποίο είναι διαθέσιμο για χρήση από οποιονδήποτε compiler της Microsoft (για Windows). Στο Unix τις κλήσεις συστήματος καθορίζει το πρότυπο **POSIX** (Portable Operating System-IX). Με την τυποποίηση αυτή του POSIX μία εφαρμογή γραμμένη σε ένα UNIX σύστημα μπορεί εύκολα να μεταφερθεί σε ένα άλλο σύστημα UNIX χωρίς πρόβλημα, αρκεί και τα δύο αυτά συστήματα UNIX να ακολουθούν την τυποποίηση POSIX.

Οι κλήσεις συστήματος μπορούν να ομαδοποιηθούν σε πέντε γενικές κατηγορίες[6]: **1)** έλεγχος διεργασιών (**process control)**, **2)** διαχείριση αρχείων **(file management)**, **3)** διαχείριση συσκευών **(device management)**, **4)** διατήρηση πληροφοριών **(information maintenance)** και **5)** επικοινωνίες (**communications)**.

Ο πίνακας 1.2 δείχνει με συνοπτικό τρόπο τις κατηγορίες των κλήσεων συστήματος που παρέχονται από ένα τυπικό ΛΣ:

1.Έλεγχος Διεργασιών
- δημιουργία, τερματισμός διεργασίας
- φόρτωμα και εκτέλεση
- ανάκτηση και ρύθμιση παραμέτρων των διεργασιών
- χρονική αναμονή
- αναμονή γεγονότος ή σήματος
- κατανομή και απελευθέρωση μνήμης
2. Διαχείριση Αρχείων
- δημιουργία και διαγραφή αρχείων
- άνοιγμα και κλείσιμο αρχείων
- ανάγνωση, εγγραφή και επανατοποθέτηση (reposition)
- ανάκτηση και ρύθμιση παραμέτρων των αρχείων
3. Διαχείριση Συσκευών
- αίτηση και απελευθέρωση συσκευής
- ανάγνωση, εγγραφή και επανατοποθέτηση (reposition)
- ανάκτηση και ρύθμιση παραμέτρων των συσκευών
4. Διατήρηση Πληροφοριών
- Ανάκτηση και ρύθμιση ώρας και ημερομηνίας
- Ανάκτηση και ρύθμιση δεδομένων του συστήματος
- Πληροφορίες για τους χρήστες του συστήματος
5. Επικοινωνία
- Δημιουργία και τερματισμός συνδέσμου επικοινωνίας
- Αποστολή και λήψη μηνυμάτων

Πίνακας 1.3 – Οι κατηγορίες των System Calls

Στην εργασία αυτή και για την υλοποίηση του κελύφους θα κάνουμε εκτενή χρήση διάφορων κλήσεων συστήματος των κατηγοριών ελέγχου διεργασιών και διατήρησης πληροφοριών. Τις κλήσεις συστήματος που θα χρησιμοποιήσουμε θα τις αναλύσουμε σε επόμενο κεφάλαιο.

1.4.3.2 Πέρασμα/ανταλλαγή μηνυμάτων

Η δεύτερη τεχνική για την επικοινωνία ενός προγράμματος με τον πυρήνα του ΛΣ είναι το πέρασμα ή ανταλλαγή μηνυμάτων (Message passing). Όταν μία εφαρμογή του χρήστη θέλει να επικοινωνήσει με τον πυρήνα κατασκευάζει ένα μήνυμα με το οποίο περιγράφει την επιθυμητή υπηρεσία. Έπειτα στέλνει το μήνυμα σε μία συγκεκριμένη διεργασία του Λειτουργικού Συστήματος. Σε αυτό το σημείο η εφαρμογή του χρήστη περιμένει το αποτέλεσμα της υπηρεσίας του πυρήνα και το bit κατάστασης έχει οριστεί σε 0 (κατάσταση πυρήνα). Μόλις ο πυρήνας τελειώσει την υπηρεσία που του ζητήθηκε επιστρέφει στην εφαρμογή του χρήστη ένα μήνυμα και θέτει το bit κατάστασης σε 1 (κατάσταση χρήστη).

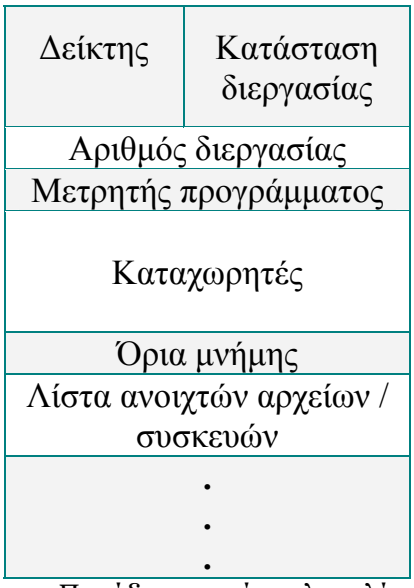
Η τεχνική αυτή της ανταλλαγής μηνυμάτων είναι εκτός σκοπού της εργασίας και δεν θα ασχοληθούμε περαιτέρω.

1.5 ■ Διεργασίες (Processes)

Τα πρώτα λειτουργικά συστήματα επέτρεπαν την εκτέλεση ενός και μόνο προγράμματος μία δεδομένη χρονική στιγμή. Το πρόγραμμα αυτό είχε τον πλήρη έλεγχο του συστήματος και των πόρων του. Τα σημερινά συστήματα επιτρέπουν την ταυτόχρονη εκτέλεση πολλών προγραμμάτων και η τεχνική αυτή ονομάζεται πολυδιεργασία (**multiprogramming)**. Αυτή η εξέλιξη απαίτησε αυστηρότερο έλεγχο και τμηματοποίηση των εκτελούμενων προγραμμάτων. Αυτή την ανάγκη ήρθε να καλύψει η έννοια της **διεργασίας (process)**. Μία διεργασία είναι ένα πρόγραμμα σε εκτέλεση. Όταν ένα πρόγραμμα πρόκειται να εκτελεστεί αυτό πρέπει να γίνει στα πλαίσια κάποιας διεργασίας. Η διεργασία αυτή χρειάζεται συγκεκριμένους **πόρους** από το σύστημα για να ολοκληρωθεί. Πόρους όπως χρόνο από τον επεξεργαστή (CPU), μνήμη, αρχεία και είσοδο-έξοδο συσκευών. Οι πόροι αυτοί δεσμεύονται για την διεργασία από το ΛΣ είτε κατά την δημιουργία της είτε κατά την εκτέλεσή της.

Η έννοια της διεργασίας είναι πολύ σημαντική στα Λειτουργικά Συστήματα και στην εργασία αυτή και έτσι θα πρέπει να την εξηγήσουμε λίγο παραπάνω. Ας υποθέσουμε ότι έχουμε ένα εκτελέσιμο πρόγραμμα στα windows με όνομα αρχείου (filename) winamp.exe. Το αρχείο αυτό αποτελεί το πρόγραμμα το οποίο θέλουμε να το εκτελέσουμε. Στο σημείο αυτό δεν υπάρχει καμία διεργασία που να αφορά το συγκεκριμένο πρόγραμμα. Όταν εκτελέσουμε το αρχείο αυτό (είτε με διπλό κλικ με το mouse είτε με enter), τότε το ΛΣ θα φορτώσει το αρχείο στην μνήμη (συγκεκριμένα τον **κώδικα του αρχείου – text segment**), θα δεσμεύσει ένα **χώρο διευθύνσεων (address space)** στην μνήμη για τις **μεταβλητές του προγράμματος (data segment)** και για την **στοίβα (stack segment)**, θα δημιουργήσει μερικές μεταβλητές που θα αφορούν την κατάσταση του όπως τον **μετρητή προγράμματος (program counter)**, την **λέξη πληροφοριών κατάστασης (process status)** και διάφορους γενικούς **καταχωρητές (registers)**. Αφού το ΛΣ ολοκληρώσει όλη αυτή τη λειτουργία τότε το πρόγραμμά μας είναι έτοιμο προς εκτέλεση.

Η διαδικασία που περιγράψαμε παραπάνω είναι η διαδικασία με την οποία το ΛΣ θα κατασκευάσει το λεγόμενο **Μπλοκ Ελέγχου Διεργασίας (Process Control Block)**. Κάθε διεργασία αναπαρίσταται από το ΛΣ με ένα Μπλοκ Ελέγχου Διεργασίας. Η παρακάτω εικόνα δείχνει ένα μπλοκ ελέγχου διεργασίας και τα τμήματα από τα οποία αποτελείται.

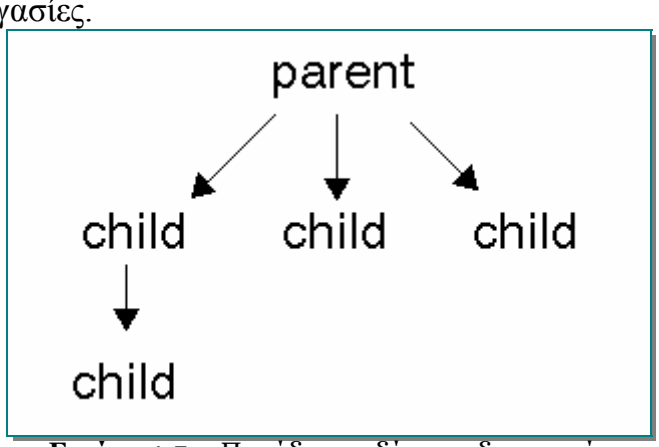


Εικόνα 1.4 – Παράδειγμα ενός μπλοκ ελέγχου διεργασίας

- **Κατάσταση διεργασίας (Process state):** Η κατάσταση μιας διεργασίας μπορεί να είναι – "νέα", "έτοιμη", "εκτελείται", "περιμένει", "τερματίστηκε".
- **Μετρητής προγράμματος (Program counter):** Ο μετρητής αυτός δηλώνει την διεύθυνση της επόμενης εντολής για εκτέλεση για την συγκεκριμένη διεργασία.
- **Καταχωρητές (CPU registers):** Οι καταχωρητές ποικίλουν σε αριθμό και σε τύπο ανάλογα με την αρχιτεκτονική του κάθε συστήματος.
- **Όρια μνήμης (Memory Management Information):** Στο πεδίο αυτό υπάρχουν πληροφορίες όπως οι τιμές των καταχωρητών base και limit, οι πίνακες σελίδων (page tables) κλπ.
- **Λίστα ανοιχτών αρχείων / συσκευών (IO status information):** Οι πληροφορίες που υπάρχουνε εδώ αφορούν την λίστα με τις συσκευές που απασχολεί η συγκεκριμένη διεργασία καθώς και την λίστα με τα ανοιχτά αρχεία κλπ.

Τα περισσότερα σημερινά λειτουργικά συστήματα επιτρέπουν να δημιουργούνται και να τερματίζονται διεργασίες δυναμικά. Για να αξιοποιηθεί πλήρως αυτή η δυνατότητα ώστε να έχουμε παράλληλη επεξεργασία, χρειάζεται μια κλήση συστήματος που να δημιουργεί μια νέα διεργασία. Αυτή η κλήση συστήματος μπορεί απλώς να δημιουργεί έναν κλώνο της καλούσας διεργασίας ή να επιτρέπει στη δημιουργούσα διεργασία **(μητρική – parent)** να καθορίσει την αρχική κατάσταση της νέας διεργασίας **(θυγατρική – child)**, στην οποία περιλαμβάνεται το πρόγραμμα, τα δεδομένα και η διεύθυνση αρχής.[7]

Μία διεργασία (parent) μπορεί να δημιουργήσει αρκετές νέες διεργασίες (children) κατά την διάρκεια της εκτέλεσής της. Κάθε νέα διεργασία μπορεί με την σειρά της να δημιουργήσει νέες διεργασίες κατασκευάζοντας έτσι ένα *δέντρο* από διεργασίες.



Εικόνα 1.5 – Παράδειγμα δέντρου διεργασιών

Σε ορισμένες περιπτώσεις η μητρική διεργασία διατηρεί μερικό ή και πλήρη έλεγχο πάνω στη δημιουργούμενη διεργασία. Σε άλλες περιπτώσεις μια μητρική διεργασία έχει μικρότερο έλεγχο πάνω στις θυγατρικές της: από την στιγμή που θα δημιουργηθεί μία διεργασία, δεν υπάρχει τρόπος η μητρική να τη σταματήσει, να την ξαναξεκινήσει, να την εξετάσει ή να την τερματίσει. Στην περίπτωση αυτή οι δύο διεργασίες εκτελούνται ανεξάρτητα η μία από την άλλη. Όταν μία διεργασία (μητρική) δημιουργεί μία νέα διεργασία (θυγατρική) τότε υπάρχουνε δύο περιπτώσεις από την άποψη της εκτέλεσης:

1. Η μητρική συνεχίζει να εκτελείται ταυτόχρονα με την θυγατρική διεργασία
2. Η μητρική διεργασία περιμένει όλες τις θυγατρικές να τερματιστούν πριν συνεχίσει την εκτέλεσή της.

Για να επεξηγήσουμε αυτές τις περιπτώσεις ας χρησιμοποιήσουμε σαν παράδειγμα το Λειτουργικό Σύστημα Unix. Στο Unix κάθε διεργασία χαρακτηρίζεται από το **περιγραφέα διεργασίας (process identifier - pid)** το οποίο στην ουσία δεν είναι τίποτα άλλο από ένα ακέραιο αριθμό μοναδικό στο σύστημα την συγκεκριμένη χρονική στιγμή. Ο πυρήνας του Unix διατηρεί ένα πίνακα με τους περιγραφείς των διεργασιών ώστε να μπορεί να έχει άμεση πρόσβαση σε αυτές. Για να δείτε την λίστα με τις ενεργές διεργασίες σε ένα σύστημα Unix/Linux χρησιμοποιήστε την εντολή ps –a.

Στο ΛΣ Unix αλλά και στο Linux δημιουργούμε μία νέα διεργασία με την κλήση συστήματος **fork**. Η νέα διεργασία που θα δημιουργηθεί θα είναι ένα αντίγραφο της μητρικής διεργασίας. Οι δύο διεργασίες θα είναι πανομοιότυπες (ίδιο text, data και stack segment) και μοιράζονται ακόμα και τους **περιγράφεις αρχείων (file identifiers)**. Κάθε μία όμως έχει τον δικό της περιγραφέα διεργασίας και το δικό της ξεχωριστό χώρο διευθύνσεων. Κανένα κομμάτι του χώρου διευθύνσεων δεν είναι κοινό για τις δύο διεργασίες, άρα δεν μπορούνε να αναφερθούνε στην ίδια μεταβλητή. Οι δύο πλέον διεργασίες (μητρική και θυγατρική) συνεχίζουνε την εκτέλεσή τους από την εντολή

αμέσως μετά την κλήση της fork με μία διαφορά: Η τιμή επιστροφής της fork θα είναι 0 για την νέα διεργασία και μη μηδενική για την μητρική. Συγκεκριμένα στην μητρική διεργασία η fork θα επιστρέψει το process identifier (pid) της νέας (θυγατρικής) διεργασίας.

Συνήθως αμέσως μετά την κλήση της fork η νέα διεργασία θα θέλουμε να εκτελέσει το δικό της πρόγραμμα και όχι το ίδιο με την μητρική διεργασία. Ας μην ξεχνάμε ότι η θυγατρική είναι ένα αντίγραφο της μητρικής διεργασίας στην φάση αυτή. Για να φορτώσει νέο κώδικα στην μνήμη η νέα διεργασία καλεί την κλήση συστήματος **execlp** (ή κάποια άλλη κλήση συστήματος της οικογενείας **exec**). Η execlp φορτώνει στην μνήμη ένα εκτελέσιμο (δυαδικό – binary) αρχείο, διαγράφοντας τον κώδικα που ήτανε έως εκείνη την στιγμή φορτωμένος στην μνήμη και ξεκινάει την εκτέλεση του αρχείου αυτού από το **entry point** του.

Η μητρική διεργασία μπορεί να περιμένει την θυγατρική να τελειώσει καλώντας την κλήση συστήματος **wait** ή **waitpid**. Αν δεν καλέσει την wait απλά δεν περιμένει και συνεχίζει ταυτόχρονα την εκτέλεσή της. Όταν μία διεργασία τερματίσει τότε ο αποδεσμεύονται οι πόροι του συστήματος (που είχανε δεσμευτεί κατά την δημιουργία της) και διαγράφεται ο περιγραφέας της διεργασίας από τον πίνακα περιγραφέων που κρατάει ο πυρήνας. Στο σημείο αυτό ο πυρήνας ενημερώνει την μητρική διεργασία με ένα σήμα (signal) για τον τερματισμό της θυγατρικής της διεργασίας. Η κλήση συστήματος wait που αναφέραμε είναι αυτή που ενημερώνει την μητρική διεργασία για τον τερματισμό της θυγατρικής. Ο παρακάτω κώδικας C υλοποιεί την διαδικασία που μόλις αναφέραμε σε περιβάλλον Unix/Linux και παρουσιάζει πως χρησιμοποιούνται όλες αυτές οι κλήσεις συστήματος που αναφέραμε.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t pid;
  /* fork another process */
  pid = fork();
  if (pid == -1) {       /* error occured */
    fprintf(stderr,"fork failed\n");
    exit(1);
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls","ls",NULL);
  }
  else {          /* parent process */
    wait(NULL);
    printf("Child Complete");
    exit(0);
  }
}
```

1.6 ■ Διαδιεργασιακή επικοινωνία (Interprocess Communication)

Οι διεργασίες δεν εκτελούνται απομονωμένες από το υπόλοιπο σύστημα ούτε από τις υπόλοιπες διεργασίες. Πολλές φορές χρειάζεται μία διεργασία να επικοινωνήσει με μία άλλη για ανταλλαγή δεδομένων ή για να συγχρονίσουνε την εκτέλεσή τους.. Όλα τα σύγχρονα Λειτουργικά Συστήματα διαθέτουνε διάφορες τεχνικές για να επιτευχθεί αυτή η επικοινωνία. Η επικοινωνία αυτή μεταξύ των διεργασιών ονομάζεται Διαδιεργασική επικοινωνία (**Interprocess Communication – IPC**). Οι τεχνικές αυτής της επικοινωνίας σε ένα ΛΣ όπως το Unix μπορούνε να χωριστούνε σε πέντε κατηγορίες:

- **Διοχέτευση εισόδου/εξόδου (Pipes)**
- **Ουρές μηνυμάτων (Message Queues)**
- **Διαμοιραζόμενη μνήμη (Shared Memory)**
- **Σηματοφόροι (Semaphores)**
- **Υποδοχές (Sockets)**

Με τόσους πολλούς τρόπους για να πετύχουμε περίπου το ίδιο πράγμα είναι φυσικό να αναρωτηθούμε ποιος είναι ο καλύτερος. Υπάρχουνε τουλάχιστον τρεις διαστάσεις που μπορούμε να δώσουμε στον όρο "καλύτερος":

- Κατάλληλος για τις ανάγκες τις εφαρμογής μας
- Μεταφέρσιμος, στην περίπτωση που χρειαστεί κάποια στιγμή να τον μεταφέρουμε σε διαφορετικό σύστημα, για παράδειγμα: HP/UX, Solaris ή από AIX σε Linux.
- Αποδοτικός

Στην συνέχεια θα παρουσιάσουμε συνοπτικά τους μηχανισμούς IPC που αναφέραμε παραπάνω.

1.5.1 Διοχέτευση εισόδου/εξόδου (Pipes)

Η διοχέτευση εισόδου/εξόδου (pipe) είναι ένα είδος προσωρινής αποθήκευσης στην οποία μια διεργασία μπορεί να γράφει ένα ρεύμα δεδομένων και μία άλλη διεργασία να το διαβάζει. Τα bytes ανακτώνται από μία διοχέτευση πάντα με την σειρά που γράφτηκαν. Δεν είναι δυνατή η τυχαία προσπέλαση. Γενικά συνδέουμε μία διοχέτευση στην έξοδο μιας διεργασίας και στην είσοδο της άλλης διεργασίας. Οι χρήστες του Unix και του Linux θα έχουνε συναντήσει πολλές φορές εντολές κελύφους όπως:

```
$ who | sort | more
```

Στην παραπάνω εντολή υπάρχουν 3 διεργασίες συνδεδεμένες με 2 διοχετεύσεις. Τα δεδομένα κατευθύνονται σε μία μόνο κατεύθυνση, από την *who* στην *sort* και μετά στην *more*. Είναι επίσης εφικτό να δημιουργήσουμε

διοχετεύσεις αμφίδρομης επικοινωνίας (από την διεργασία Α στην διεργασία Β και από την διεργασία Β πίσω στην διεργασία Α) καθώς και διοχετεύσεις σε δαχτυλίδι (ring) (από την διεργασία Α στην Β, από την Β στην Γ και από την Γ πίσω στην Α) χρησιμοποιώντας κλήσεις συστήματος. Τα περισσότερα κελύφη όμως δεν παρέχουνε τέτοιες δυνατότητες στους χρήστες. Αυτός είναι και ο λόγος για τον οποίο τέτοιες συνδέσεις είναι άγνωστες στους περισσότερους χρήστες του Unix.[8]

Μία διοχέτευση είναι σαν αρχείο ανοιγμένο για ανάγνωση και για εγγραφή. Οι διοχετεύσεις δημιουργούνται με την κλήση συστήματος **pipe** η οποία επιστρέφει 2 περιγραφείς **(descriptors)** για αυτήν. Ένα για ανάγνωση και ένα για εγγραφή.

```
#include <unistd.h>

int pipe(int file_descriptor[2]);
```

Αφού λοιπόν επιστρέφει περιγραφείς αρχείων ο τρόπος ανάγνωσης και εγγραφής είναι μονόδρομος. Πρέπει να χρησιμοποιήσουμε τις κλήσεις συστήματος **read** και **write**.

Προσοχή πρέπει να δοθεί σε δύο συναρτήσεις που είναι επίσης διαθέσιμες. Είναι οι συναρτήσεις **popen** και **pclose**. Οι συναρτήσεις αυτές δεν είναι κλήσεις συστήματος. Είναι υψηλού επιπέδου συναρτήσεις και για να υλοποιήσουνε διοχέτευση μεταξύ δυο διεργασιών καλούνε το κέλυφος το οποίο κάνει όλη την δουλειά στην ουσία.[9] Το γεγονός αυτό δεν είναι πάντοτε επιθυμητό και έχει αρκετά μειονεκτήματα. Φυσικά το μεγαλύτερο πλεονέκτημά των συναρτήσεων αυτών είναι η ευκολία με την οποία πετυχαίνουμε επικοινωνία μεταξύ δύο διεργασιών. Οι συναρτήσεις αυτές χρησιμοποιούν **ρεύματα αρχείων** (**file streams** - FILE *) για να ανοίξουνε και να κλείσουνε τα αρχεία και για να κάνουμε ανάγνωση και εγγραφή χρησιμοποιούμε τις συναρτήσεις της καθιερωμένης βιβλιοθήκης **stdio** (standard stdio library), **fread** και **fwrite**.

Οι διοχετεύσεις (pipes) χωρίζονται σε δύο γενικές κατηγορίες. **1)** Τις **ανώνυμες διοχετεύσεις (unnamed pipes)** και **2)** τις **επώνυμες διοχετεύσεις (named pipes)** ή αλλιώς **FIFOs**. Όσα αναφέραμε ως τώρα αφορούν τις ανώνυμες διοχετεύσεις (unnamed pipes).

Οι επώνυμες διοχετεύσεις (named pipes – FIFOs) συνδυάζουνε χαρακτηριστικά των συνηθισμένων αρχείων και των ανώνυμων διοχετεύσεων (unnamed pipes). Μία επώνυμη διοχέτευση είναι ένας ειδικός τύπος αρχείου (μην ξεχνάμε ότι τα πάντα στο Unix/Linux είναι αρχεία) το οποίο υπάρχει σαν όνομα στο σύστημα αρχείων αλλά συμπεριφέρεται σαν ανώνυμη διοχέτευση όπως αναφέραμε προηγουμένως. Μπορούμε να δημιουργήσουμε επώνυμες διοχετεύσεις από το κέλυφος αλλά και μέσα από ένα πρόγραμμα (με κλήσεις συστήματος).

Από το κέλυφος μπορούμε να δημιουργήσουμε επώνυμες διοχετεύσεις με δύο εντολές : mknod και mkfifo:

```
$ mknod filename p
```

```
$ mkfifo filename
```

Η εντολή κελύφους mknod είναι ιστορικά η πρώτη που χρησιμοποιήθηκε όμως δεν συμπεριλαμβάνεται στο πρότυπο X/Open με αποτέλεσμα να μην είναι διαθέσιμη σε όλα τα συστήματα Unix. Υπάρχει όμως σχεδόν σε όλα τα συστήματα Linux. Η προτιμητέα όμως εντολή κελύφους είναι η mkfifo η οποία υπάρχει σε όλα τα συστήματα Unix και Linux και συμπεριλαμβάνεται στο πρότυπο X/Open.

Μέσα από κώδικα C μπορούμε να χρησιμοποιήσουμε δύο διαφορετικές κλήσεις:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char* filename, mode_t mode);
int mknod(const char* filename, mode_t mode | S_IFOFO, (dev_t) 0);
```

Μπορούμε να διαγράψουμε το FIFO όπως ακριβώς ένα συνηθισμένο αρχείο με την εντολή κελύφους rm ή μέσα από πρόγραμμα χρησιμοποιώντας την κλήση συστήματος **unlink**.

Η διαφορά μεταξύ ανώνυμων και επώνυμων διοχετεύσεων είναι ότι οι επώνυμες διοχετεύσεις δεν είναι όσο περιοριστικές όσο οι ανώνυμες. Μερικά από τα πλεονεκτήματα των επώνυμων διοχετεύσεων είναι[10]:

- Έχουνε όνομα και οντότητα στο σύστημα αρχείων
- Μπορούνε να χρησιμοποιηθούν σε άσχετες μεταξύ τους διεργασίες
- Υπάρχουνε μέχρι να διαγραφούνε ρητά.

Οι επόμενες τρεις τεχνικές διαδιεργασιακής επικοινωνίας (IPC) παρουσιάστηκαν στην την έκδοση του Unix AT&T System V2 (1983) και έχουν παρόμοιο προγραμματιστικό interface. Γι' αυτούς τους λόγους οι τεχνικές αυτές αποκαλούνται πολύ συχνά σαν **System V IPC**.

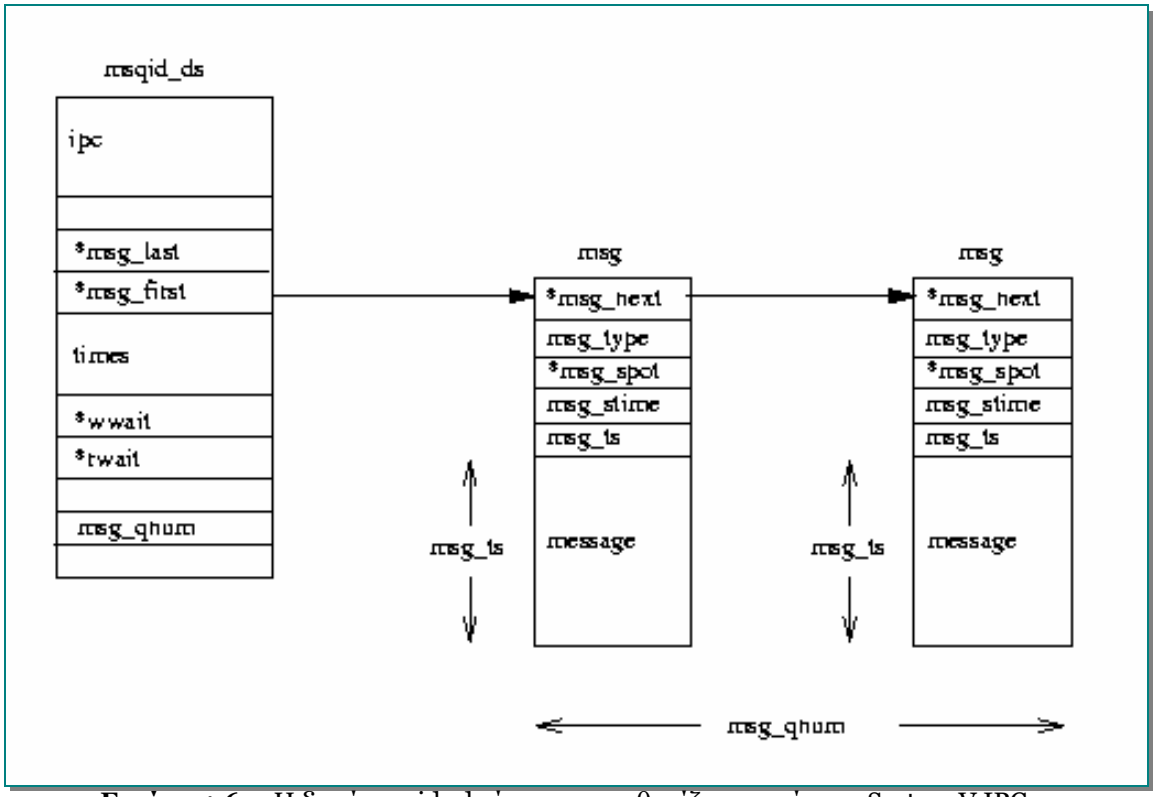
1.5.2 Ουρές μηνυμάτων (Message Queues)

Οι ουρές μηνυμάτων (message queues) μοιάζουνε από πολλές πλευρές τις επώνυμες διοχετεύσεις αλλά χωρίς την πολυπλοκότητα που σχετίζεται με το άνοιγμα και κλείσιμο της διοχέτευσης. Οι ουρές μηνυμάτων προσφέρουν ένα σχετικά εύκολο και αποδοτικό τρόπο για να ανταλλάξουμε δεδομένα μεταξύ δύο διεργασιών. Στην θέση του αρχείου (που χρησιμοποιούν οι επώνυμες

διοχετεύσεις) χρησιμοποιούν ένα **κλειδί**. Το κλειδί αυτό είναι ένας θετικός ακέραιος αριθμός τύπου **key_t** όπως δηλώνεται στο αρχείο επικεφαλίδας sys/types.h.

Όταν δημιουργείται μία ουρά μηνυμάτων, ο πυρήνας του ΛΣ αποθηκεύει τα δεδομένα που θα στέλνονται στην ουρά αυτή. Ο πυρήνας έχει ένα πεπερασμένο ποσό μνήμης το οποίο χρησιμοποιεί για την ουρά μηνυμάτων. Αν η μνήμη αυτή γεμίσει τότε ο πυρήνας μπλοκάρει την είσοδο και καμία διεργασία δεν μπορεί να στείλει άλλα δεδομένα, αν πρώτα μια άλλη διεργασία δεν διαβάσει από την ουρά ώστε να αδειάσει η μνήμη. Μία διεργασία διαβάζει τα δεδομένα με την σειρά που αυτά μπήκαν στην ουρά. Υπάρχει όμως δυνατότητα να μία διεργασία να κοιτάξει και πέρα από το άκρο της ουράς "look ahead" για δεδομένα μεγαλύτερης προτεραιότητας. Σημαντικό είναι να τονίσουμε ότι τα δεδομένα θα διατηρηθούν στην ουρά ακόμα και αν η διεργασία που δημιούργησε την ουρά τερματιστεί. Για το χρονικό διάστημα που καμία άλλη διεργασία δεν έχει διαβάσει τα δεδομένα που βρίσκονται στην ουρά, ο πυρήνας τα κρατάει υποθηκευμένα και δεν επιτρέπει σε καμία άλλη διεργασία να δημιουργήσει άλλη ουρά μηνυμάτων. Αυτό μπορεί να είναι αρκετά επικίνδυνο και πρέπει να δοθεί ιδιαίτερη προσοχή γιατί πολύ εύκολα μπορεί να δημιουργηθεί **άρνηση υπηρεσιών** (**denial of service**).

Το Linux διαθέτει μία λίστα με τις ουρές μηνυμάτων με όνομα msgque vector. Κάθε στοιχείο αυτής της λίστας είναι ένας δείκτης σε μία δομή δεδομένων με όνομα msqid_ds η οποία περιγράφει πλήρως μία ουρά μηνυμάτων. Όταν δημιουργείται μία ουρά μηνυμάτων, το σύστημα δεσμεύει χώρο στην μνήμη για ένα νέο αντικείμενο msqid_ds και ενημερώνεται η λίστα msgque. Η εικόνα 1.6 παρουσιάζει την δομή msqid_ds.



Εικόνα 1.6 – Η δομή msqid_ds όπως την καθορίζει το πρότυπο System V IPC

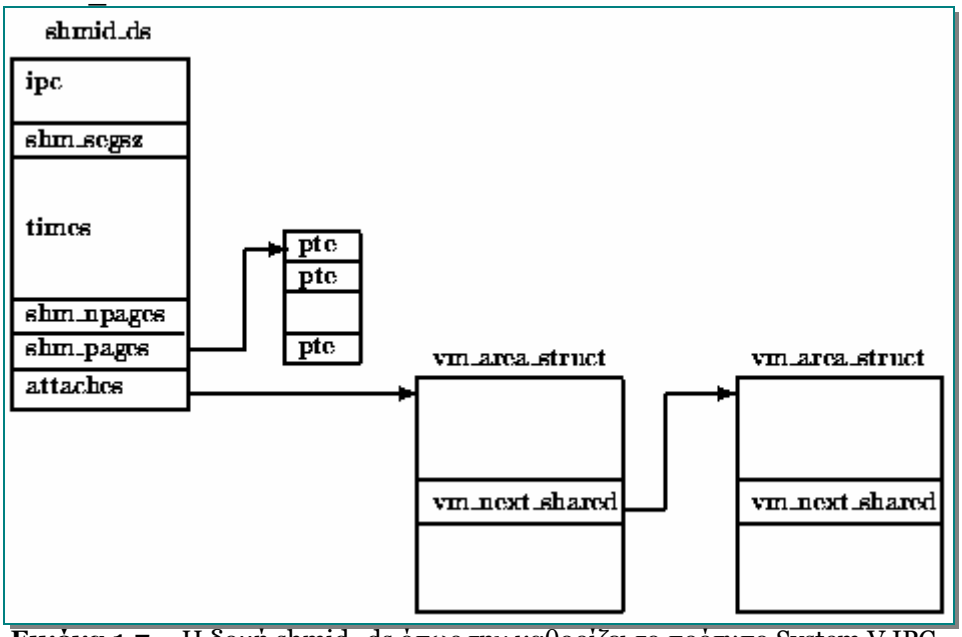
Η δομή msqid_ds είναι δηλωμένη στο αρχείο επικεφαλίδας sys/msg.h και για να την χρησιμοποιήσουμε θα πρέπει να συμπεριλάβουμε στο πρόγραμμά μας και τις επικεφαλίδες sys/types.h καθώς και sys/ipc.h.

1.5.3 Διαμοιραζόμενη μνήμη (Shared Memory)

Η διαμοιραζόμενη μνήμη (shared memory) είναι η δεύτερη τεχνική που παρουσιάζεται στις υπηρεσίες System V IPC. Η διαμοιραζόμενη μνήμη είναι ίσως η πιο εύκολη τεχνική διαδιεργασιακής επικοινωνίας. Όπως λέει το όνομά της δεν είναι τίποτα άλλο παρά ένα κομμάτι μνήμης το οποίο είναι κοινό μεταξύ των διεργασιών. Μία διεργασία δημιουργεί αυτό το κοινό κομμάτι μνήμης και θέτει τα δικαιώματα που θα έχουν οι άλλες διεργασίες στην μνήμη αυτή. Όταν επιτρέπεται η εγγραφή σε περισσότερες από μία διεργασίες τότε ένα εξωτερικό πρωτόκολλο συγχρονισμού ή ένας μηχανισμός όπως οι σηματοφόροι (semaphores) χρησιμοποιούνται για την αποφυγή συγκρούσεων.

Όπως και στις ουρές μηνυμάτων έτσι και στην διαμοιραζόμενη μνήμη η διεργασία που θα δημιουργήσει ένα τμήμα μοιραζόμενες μνήμης χρησιμοποιεί ένα κλειδί. Η διεργασία που θα δημιουργήσει την διαμοιραζόμενη μνήμη ονομάζεται server και οι διεργασίες που θα χρησιμοποιούν και θα έχουνε πρόσβαση στην μνήμη αυτή ονομάζονται clients.

Το Linux διαθέτει μία λίστα με τα μοιραζόμενα τμήματα μνήμης με όνομα shm_segs vector. Κάθε στοιχείο αυτής της λίστας είναι ένας δείκτης σε μία δομή δεδομένων με όνομα shmid_ds η οποία περιγράφει πλήρως μία διαμοιραζόμενη μνήμη. Στην δομή αυτή αποθηκεύονται πληροφορίες όπως πόσο μεγάλη είναι η μνήμη, πόσες διεργασίες την χρησιμοποιούν καθώς και πληροφορίες που αφορούν την χαρτογράφηση της μεριζόμενης μνήμης με την εικονική μνήμη της κάθε διεργασίας. Όταν δημιουργείται μία διαμοιραζόμενη μνήμη, το σύστημα δεσμεύει χώρο στην μνήμη για ένα νέο αντικείμενο shmid_ds και ενημερώνεται η λίστα shm_segs. Η εικόνα 1.7 παρουσιάζει την δομή shmid_ds.



Εικόνα 1.7 – Η δομή shmid_ds όπως την καθορίζει το πρότυπο System V IPC

Πριν προχωρήσουμε στους **σηματοφόρους (semaphores)** πρέπει να αναλύσουμε λίγο μερικές έννοιες που είναι ουσιαστικές ώστε να κατανοήσουμε πλήρως την χρήση και την ανάγκη τους.

Συνθήκες ανταγωνισμού και κρίσιμα τμήματα

Φανταστείτε μία περίπτωση όπου υπάρχουν δύο ανεξάρτητες διεργασίες, η διεργασία 1 και η διεργασία 2, οι οποίες επικοινωνούν μέσω μιας μεριζόμενης περιοχής προσωρινής αποθήκευσης (shared buffer) της κύριας μνήμης. Στην μνήμη αυτή είναι αποθηκευμένος ένας πίνακας ο οποίος κρατάει πληροφορίες σχετικά με τις κρατήσεις εισιτηρίων σε μία παράσταση. Η πρώτη διεργασία είναι το πρόγραμμα ενός ταμία ενός τερματικού και και η δεύτερη διεργασία είναι το πρόγραμμα ενός δεύτερου ταμία σε ένα άλλο τερματικό. Ας υποθέσουμε οι έχει απομείνει για την παράσταση ένα μόνο εισιτήριο και στα 2 ταμεία καταφθάνουν ταυτόχρονα αγοραστές. Ο κάθε ταμίας μέσα από το τερματικό του προσπαθεί να κάνει την κράτηση για αυτό το μοναδικό εισιτήριο. Τελικά ποιος ταμίας θα κάνει την κράτηση; Υπάρχει το ενδεχόμενο να γίνει ταυτόχρονα η κράτηση και για τους δύο αγοραστές;

Βλέπουμε ότι όταν δύο ή περισσότερες διεργασίες εκτελούνται χωρίς συγχρονισμό και έχουνε πρόσβαση σε κοινούς πόρους, οποιαδήποτε πράξη στους κοινούς αυτούς πόρους (στο παράδειγμά μας η διαμοιραζόμενη μνήμη) είναι επισφαλής. Η δυσάρεστη αυτή περίπτωση είναι γνωστή ως **συνθήκη ανταγωνισμού (race condition)** επειδή η επιτυχία της μεθόδου εξαρτάται από το ποιος τα κερδίσει τον αγώνα ταχύτητας για την πρόσβαση στον κοινό πόρο. Έτσι η προσπέλαση των κοινών δεδομένων των διεργασιών θα πρέπει να προφυλάσσεται ρητά με παρέμβαση του προγραμματιστή που υλοποιεί την παράλληλη εφαρμογή. Η διαδικασία διάταξης της εκτέλεσης των νημάτων κατά την προσπέλαση των κοινών δεδομένων τους, ονομάζεται συγχρονισμός και απαιτείται για την διασφάλιση της σωστής ροής του προγράμματος.

Κάθε τμήμα του κώδικα ενός προγράμματος στο οποίο πραγματοποιείται διαχείριση κοινά διαμοιραζόμενων δεδομένων χαρακτηρίζεται **κρίσιμο τμήμα (critical section)**. Τα κρίσιμα τμήματα συχνά ονομάζονται και ακολουθιακές περιοχές, καθώς όταν η ροή του προγράμματος φτάσει στην αρχή ενός τέτοιου τμήματος, τότε ο παραλληλισμός θα πρέπει να συρρικνωθεί και η προσπέλαση του κρίσιμου τμήματος θα πρέπει να γίνει ακολουθιακά.

Πώς μπορούμε ν' αποφύγουμε συνθήκες ανταγωνισμού; Πρέπει να βρούμε ένα τρόπο ούτως ώστε μόνο μία διεργασία να έχει πρόσβαση στα κοινά δεδομένα σε κάθε στιγμή. Αυτή η ιδιότητα λέγεται και **αμοιβαίος αποκλεισμός (mutual exclusion)**. Το πρόβλημα του αμοιβαίου αποκλεισμού είναι κομβικό για ένα Λ.Σ.

Παρόλο που αυτή η προϋπόθεση (του αμοιβαίου αποκλεισμού) αποφεύγει τις συνθήκες ανταγωνισμού δεν είναι επαρκής ώστε παράλληλες διεργασίες να συνεργάζονται σωστά και αποτελεσματικά χρησιμοποιώντας κοινούς πόρους. Χρειαζόμαστε τέσσερις προϋποθέσεις για να έχουμε μία σωστή λύση[11]:

1. Δεν επιτρέπεται σε δύο διεργασίες ταυτόχρονα να βρίσκονται στο κρίσιμο τμήμα.
2. Δεν πρέπει να γίνονται υποθέσεις σχετικά με την ταχύτητα και τον αριθμό των επεξεργαστών ενός υπολογιστή.
3. Καμία διεργασία που εκτελεί κώδικα εκτός του κρίσιμου τμήματος δεν μπορεί να μπλοκάρει άλλες διεργασίες
4. Ο χρόνος αναμονής μίας διεργασίας για να εισέλθει στο κρίσιμο τμήμα πρέπει να είναι πεπερασμένος.

Η συνθήκη ανταγωνισμού μπορεί να επιλυθεί τουλάχιστον με δύο τρόπους. Η μία λύση είναι να εφοδιαστεί η κάθε διεργασία με ένα bit "αναμονής αφύπνισης" (wakeup waiting bit)[12]. Όποτε διαβάζεται ένα σήμα αφύπνισης σε μία διεργασία που εκτελείται ακόμα, το bit αναμονής αφύπνισης της παίρνει τιμή 1. Οπότε η διεργασία πέφτει σε νάρκη ενώ το bit αφύπνισης της έχει την τιμή 1, η διεργασία ξαναξεκινά αμέσως αυτόματα και το bit αναμονής αφύπνισης μηδενίζεται. Το bit αναμονής αφύπνισης αποθηκεύει το πλεονάζον σήμα αφύπνισης για μελλοντική χρήση.

Αν και η μέθοδος αυτή λύνει το πρόβλημα της συνθήκης ανταγωνισμού όταν υπάρχουν μόνο δύο διεργασίες αποτυγχάνει στη γενική περίπτωση επικοινωνίας n διεργασιών, επειδή τότε μπορεί να χρειαστεί να αποθηκευτούν n-1 αφυπνίσεις. Βέβαια, θα μπορούσε η κάθε διεργασία να είναι εφοδιασμένη με n-1 bit αναμονής αφύπνισης ώστε να μπορεί να μετά μέχρι το n-1 το μοναδιαίο σύστημα, αλλά η λύση αυτή είναι μάλλον άκομψη.

1.5.4 Σηματοφόροι (semaphores)

Ο καθηγητής Dijkstra το 1968 στην εργασία του "Cooperating Sequential Processes" (Academic Press) πρότεινε μία πιο γενική λύση για το πρόβλημα του συγχρονισμού των παράλληλων διεργασιών. Στην πρότασή του εισήγαγε μία νέα μεταβλητή την οποία ονόμασε **σηματοφόρο (semaphore)**. Μία μεταβλητή σηματοφόρος μπορεί να πάρει μόνο ακέραιους θετικούς αριθμούς. Το ΛΣ διαθέτει δύο κλήσεις συστήματος οι οποίες επενεργούν στους σηματοφόρους, τις **up** (επάνω) και **down**(κάτω). Η up προσθέτει 1 σε ένα σηματοφόρο και η down αφαιρεί 1.

Ο Dijkstra ονόμασε τις λειτουργίες αυτές wait και signal που στα Ολλανδικά γράφονται passeren και vrijgeven. Από τις λέξεις αυτές έμεινε και η σημειογραφία στην βιβλιογραφία[13]:

- P (σηματοφόρος) για την λειτουργία wait ή up
- V (σηματοφόρος) για την λειτουργία signal ή down

Ο πιο απλός τύπος σηματοφόρου είναι αυτός που παίρνει μόνο τις τιμές 0 και 1 και ονομάζεται **δυαδικός σηματοφόρος (binary semaphore)**. Αυτός είναι και ο πιο γνωστός τύπος σηματοφόρου. Οι σηματοφόροι που μπορούν να

πάρουν πολλές θετικές ακέραιες τιμές ονομάζονται **γενικοί σηματοφόροι (general semaphores)**.

Ο ορισμός των λειτουργιών των σηματοφόρων P και V είναι εκπληκτικά εύκολος. Αν υποθέσουμε ότι έχουμε ένα σηματοφόρο με όνομα sv τότε οι ορισμοί των λειτουργιών περιγράφονται όπως δείχνει ο πίνακας:

Εντολή	Σηματοφόρος = 0	Σηματοφόρος > 0
P(sv)	Σηματοφόρος = σηματοφόρος + 1. Αν άλλη διεργασία είχε σταματήσει καθώς επιχειρούσε να πραγματοποιήσει μια εντολή V(sv) μπορεί τώρα να ολοκληρώσει την V(sv) και να συνεχίσει να εκτελείται.	Σηματοφόρος = σηματοφόρος +1
V(sv)	Η διεργασία σταματά, μέχρι η άλλη διεργασία να αυξήσει αυτόν το σηματοφόρο με μία P(sv).	Σηματοφόρος = σηματοφόρος - 1

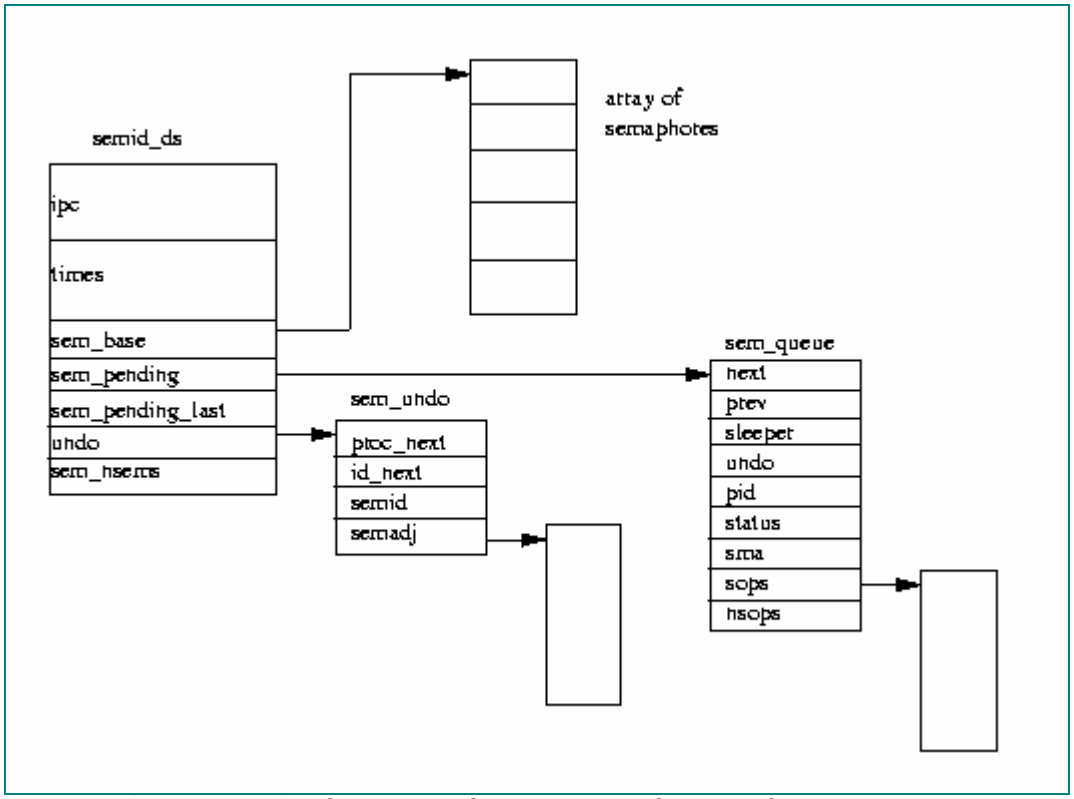
Πίνακας 1.4

Ο έλεγχος της τιμής, η αλλαγή της και η πιθανή αναστολή της εκτέλεσης, γίνονται σαν μία, αδιαίρετη **ατομική ενέργεια (atomic action)**. Είναι εγγυημένο ότι όταν ξεκινήσει μία λειτουργία σε ένα σηματοφόρο καμία άλλη διεργασία μπορεί να αποκτήσει πρόσβαση στον σηματοφόρο αυτό μέχρι η λειτουργία να ολοκληρωθεί. Αυτή η ατομικότητα είναι θεμελιώδης για την επίλυση προβλημάτων συγχρονισμού και αποφυγής συνθηκών ανταγωνισμού. Αυτός είναι και ο λόγος για τον οποίο μία οποιαδήποτε μεταβλητή δεν μπορεί να συμπεριφερθεί με αυτό τον τρόπο. Διότι δεν υπάρχει κανένας τρόπος σχεδόν σε καμία γλώσσα προγραμματισμού όπως η C/C++ ώστε να ελέγξουμε την τιμή μιας μεταβλητής και να την αλλάξουμε με ατομικό τρόπο. Αυτή η τεχνική καθιστά τους σηματοφόρους ξεχωριστούς.

Τώρα που αναλύσαμε την έννοια των σηματοφόρων ας κοιτάξουμε πως λειτουργούν και πως υλοποιούνται από το Linux. Οι σηματοφόροι έχουνε ένα σχετικά πολύπλοκο προγραμματιστικό interface. Το interface τους είναι λεπτομερέστατο και προσφέρουνε πολύ περισσότερες συναρτήσεις απ' ότι χρησιμοποιούνται συνήθως. Η βασική δομή δεδομένων που αναπαριστά ένα σηματοφόρο στο Linux ονομάζεται semid_ds. Η λίστα στην οποία αποθηκεύονται οι δείκτες αυτής της δομής ονομάζεται semary. Η εικόνα 1.8 παρουσιάζει την δομή δεδομένων semid_ds όπως καθορίζει το πρότυπο System V IPC.

Τέλος να αναφέρουμε ότι κάθε πρόγραμμα για να χειριστεί σηματοφόρους θα πρέπει να συμπεριλάβει τα παρακάτω αρχεία:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```



Εικόνα 1.8 – Η δομή semid_ds όπως την καθορίζει το πρότυπο System V IPC

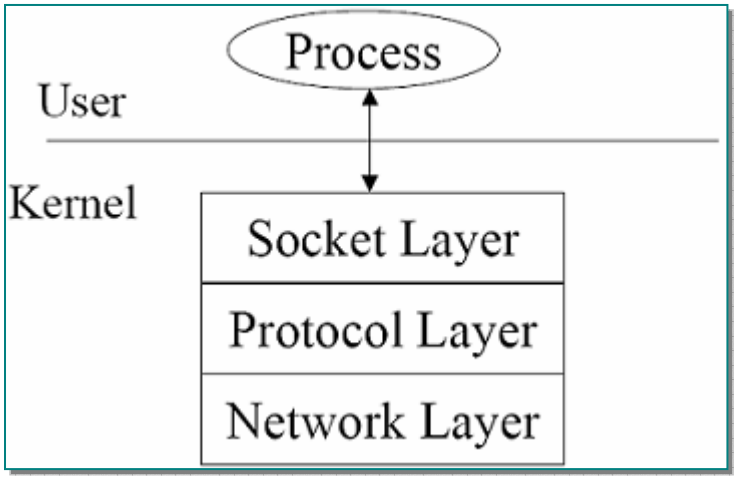
1.5.5 Υποδοχές (Sockets)

Έως τώρα όλες οι τεχνικές διαδιεργασιακής επικοινωνίας που αναλύσαμε βασίζονταν σε πόρους και σε διεργασίες που βρίσκονται στο ίδιο υπολογιστικό σύστημα. Ας φανταστούμε έναν χρήστη που επιθυμεί να αποκτήσει πρόσβαση σε ένα αρχείο που βρίσκεται σε κάποιο server. Για παράδειγμα ο χρήστης θέλει να μάθει πόσες γραμμές κειμένου έχει ένα αρχείο. Η αίτηση χειρίζεται από τον απομακρυσμένο server ο οποίος αποκτά πρόσβαση στο αρχείο, υπολογίζει το επιθυμητό αποτέλεσμα και τελικώς μεταφέρει τα δεδομένα στον χρήστη.

Η έκδοση Berkley (BSD) του Unix παρουσίασε ένα νέο εργαλείο επικοινωνίας το οποίο ονόμασε **διεπαφή υποδοχών (socket interface)**, η οποία στην ουσία είναι μία επέκταση της έννοιας της διοχέτευσης (pipe) που αναφέραμε προηγουμένως. Η διεπαφή socket είναι διαθέσιμη σε όλες τις εκδόσεις Unix και Linux. Επίσης στα Windows έγινε διαθέσιμη με την βιβλιοθήκη Windows Sockets ή **WinSock** την οποία κάνει διαθέσιμη το αρχείο winsock.dll. Με αυτό τον τρόπο εφαρμογές των Windows μπορούν να επικοινωνήσουνε μέσω ενός δικτύου με εφαρμογές που βρίσκονται σε υπολογιστές με Unix ή Linux και το αντίθετο. Να σημειώσουμε ότι αν και η διεπαφή υποδοχών είναι διαθέσιμη και στα Windows το προγραμματιστικό interface διαφέρει πολύ από αυτό του Unix αν και έχει σαν βάση τις υποδοχές.

Η υποδοχή είναι μία αφαίρεση (abstraction) που χρησιμοποιείται ως ένα τερματικό επικοινωνιακό σημείο (communication endpoint). Απαιτείται ένα ζεύγος sockets για την επικοινωνία μέσω δικτύου χρησιμοποιώντας συνήθως το TCP/IP πρωτόκολλο. Μία διεργασία μπορεί να δημιουργήσει μία υποδοχή,

να προσαφθεί σε αυτή και να εγκαινιάσει μια σύνδεση με μία υποδοχή σε ένα απομακρυσμένο υπολογιστικό σύστημα. Φυσικά πρέπει κα τα δύο υπολογιστικά συστήματα να υποστηρίζουνε το ίδιο πρωτόκολλο πχ TCP/IP. Μέσω αυτής της σύνδεσης μπορούν να ανταλλάσσονται δεδομένα και προς τις δύο κατευθύνσεις. Οι διεργασίες για να πετύχουνε την επικοινωνία



Εικόνα 1.9 – Διαδικασία κλήσεων socket

χρησιμοποιούν κατάλληλες κλήσεις συστήματος. Οι κλήσεις αυτές με την σειρά τους καλούνε κατάλληλες ρουτίνες του επιπέδου δικτύου (network layer) της στοίβας πρωτοκόλλων που χρησιμοποιεί το Λειτουργικό Σύστημα. Την διαδικασία αυτή περιγράφει η εικόνα 1.9.

Μία διεργασία (server) δημιουργεί μία υποδοχή χρησιμοποιώντας την κλήση συστήματος **socket**.

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol)
```

Αφού δημιουργηθεί η υποδοχή, η διεργασία δίνει ένα όνομα στην υποδοχή αυτή. Τοπικές υποδοχές παίρνουνε ένα όνομα αρχείου από το σύστημα αρχείων του ΛΣ. Στο Linux αυτό το αρχείο μπορεί να βρεθεί συνήθως στο κατάλογο /tmp ή /usr/tmp. Για δικτυακές υποδοχές το όνομα της θα είναι ένας περιγραφέας (identifier) σχετικός με το δίκτυο στο οποίο οι clients θα συνδεθούν. Ο περιγραφέας αυτός είναι συνήθως ένας ακέραιος αριθμός μοναδικός για κάθε υποδοχή μεγαλύτερος από 1024. Τον περιγραφέα αυτό τον ονομάζουμε **αριθμό θύρας (port number)**. Ο αριθμός αυτός χρησιμοποιείται από το ΛΣ για να μπορέσει να στείλει τα εισερχόμενα δεδομένα στην σωστή διεργασία. Η κλήση συστήματος με την οποία μία διεργασία δίνει ένα όνομα στην υποδοχή είναι η **bind**.

```
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
```

Αμέσως μετά την bind η διεργασία περιμένει ένα client να συνδεθεί στην υποδοχή. Η κλήση συστήματος **listen** δημιουργεί μία ουρά για τις εισερχόμενες συνδέσεις.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog)
```

Η διεργασία server μπορεί να δεχτεί μία σύνδεση χρησιμοποιώντας την κλήση συστήματος **accept**.

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *my_addr, size_t *address_len)
```

Όταν μία διεργασία θέλει να τερματίσει μία υποδοχή χρησιμοποιεί την κλήση συστήματος close ακριβώς όπως την χρησιμοποιούμε για να κλείσουμε περιγραφείς αρχείων.

1.7 ■ Χρονοπρογραμματισμός διεργασιών (Process scheduling)

Όπως αναφέραμε στην ενότητα 1.5 τα σύγχρονα Λειτουργικά Συστήματα είναι πολύδιεργασιακά (multiprogramming). Στόχος των πολυδιεργασιακών Λειτουργικών Συστημάτων είναι να εκτελούνε διεργασίες συνεχώς ώστε να μεγιστοποιηθεί η εκμετάλλευση του επεξεργαστή. Σε ένα σύστημα με ένα μόνο επεξεργαστή μόνο μία διεργασία μπορεί να εκτελείται κάθε στιγμή. Όλες οι υπόλοιπες διεργασίες πρέπει να περιμένουνε έως ότου ελευθερωθεί ο επεξεργαστής. Το πρόβλημα είναι ότι οι διεργασίες πολλές φορές χρειάζονται να διεκπεραιώσουν λειτουργίες εισόδου/εξόδου οι οποίες είναι συνήθως χρονοβόρες και το διάστημα αυτό ο επεξεργαστής μένει ανεκμετάλλευτος.

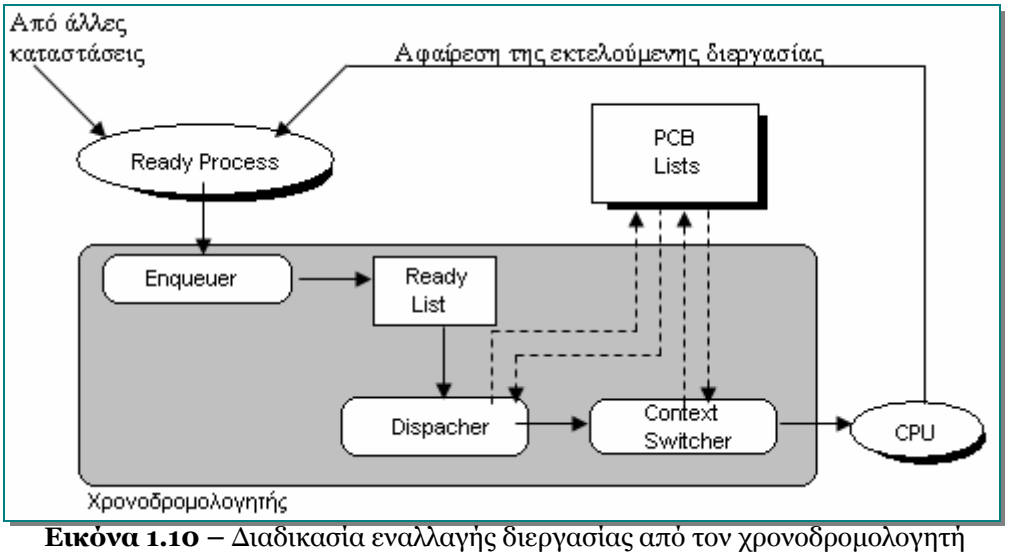
Η ιδέα της πολυδιεργασίας είναι απλή. Μία διεργασία εκτελείται έως ότου πρέπει να περιμένει για να ολοκληρωθεί μία αίτηση εισόδου/εξόδου. Όταν μία διεργασία πρέπει να περιμένει σε ένα πολυδιεργασιακό Λειτουργικό Σύστημα, τότε ο επεξεργαστής παρέχεται σε μία άλλη διεργασία. Το κομμάτι του Λειτουργικού Συστήματος που αναλαμβάνει την εναλλαγή των διεργασιών ονομάζεται **χρονοδρομολογητής** (**scheduler**) και η διαχείριση του επεξεργαστή ονομάζεται **χρονοπρογραμματισμός** (**scheduling**) και ο αλγόριθμος που χρησιμοποιείται ονομάζεται **αλγόριθμος χρονοπρογραμματισμού (scheduling algorithm)**.

Ο χρονοπρογραμματισμός διεργασιών αποτελεί θεμελιώδης λειτουργία των Λειτουργικών Συστημάτων. Πριν αναφέρουμε αλγόριθμους χρονοπρογραμματισμού πρέπει να αντιληφθούμε ποια είναι τα κριτήρια ενός σωστού αλγόριθμου χρονοπρογραμματισμού και τι πρέπει να μας εξασφαλίζει αυτός[14]:

- Δικαιοσύνη (fairness) – Κάθε διεργασία δικαιούται ισόχρονο μερίδιο από τον επεξεργαστή.
- Αποδοτικότητα (efficiency) - Ο επεξεργαστής πρέπει να απασχολείται στο 100% κάθε στιγμή.
- Ελάχιστο χρόνο απόκρισης (response time) – Ελαχιστοποίηση χρόνου απόκρισης για την αλληλεπίδραση των χρηστών
- Ελάχιστο χρόνο διεκπεραίωσης (turnaround time) – Ελαχιστοποίηση του χρόνου για έξοδο (output) στους χρήστες.
- Μέγιστο ρυθμό απόδοσης (throughput) – Μεγιστοποίηση του αριθμού των διεργασιών που επεξεργάζονται ανά ώρα.

Ο αλγόριθμος χρονοπρογραμματισμού αποφασίζει πότε πρέπει μία διεργασία να απομακρυνθεί από τον επεξεργαστή και ποια διεργασία πρέπει να δεσμεύσει τον επεξεργαστή. Οι διεργασίες που δεν εκτελούνται και είναι έτοιμες να εκτελεστούν διατηρούνται σε μία λίστα η οποία ονομάζεται λίστα ετοιμότητας (**ready list)**. Ανάλογα με τον αλγόριθμο χρονοπρογραμματισμού η λίστα αυτή μπορεί να είναι υλοποιημένη σαν ουρά FIFO, ή σαν ουρά προτεραιότητας (priority queue), σαν δέντρο ή ακόμα και σαν απλή συνδεδεμένη λίστα χωρίς προτεραιότητα (unordered linked list). Όπως και να είναι υλοποιημένη η λίστα, οι εγγραφές τις είναι δείκτες σε μπλοκ ελέγχου διεργασιών (PCB)[15].

Όταν ο χρονοδρομολογητής αλλάξει την διεργασία που εκτελεί ο επεξεργαστής, τότε πριν γίνει η εναλλαγή αποθηκεύει όλους τους καταχωρητές του επεξεργαστή (PC, IR, Processor status, ALU status) στο PCB της διεργασίας που θα εγκαταλείψει τον επεξεργαστή. Η εικόνα 1.10 περιγράφει αυτή την διαδικασία.



Εικόνα 1.10 – Διαδικασία εναλλαγής διεργασίας από τον χρονοδρομολογητή

Ο χρονοδρομολογητής μπορεί να ασκήσει δραματική επιρροή στην επίδοση ενός πολυδιεργασιακού Λειτουργικού Συστήματος αφού έχει πλήρη έλεγχο στο πότε μία διεργασία θα δεσμεύσει τον επεξεργαστή.

Οι χρονοδρομολογητές έχουνε μελετηθεί πάρα πολύ τις τελευταίες τρεις δεκαετίες για αρκετούς λόγους όπως[16]:

- Η συμπεριφορά του χρονοδρομολογητή είναι κρίσιμη για την επίδοση κάθε ξεχωριστής διεργασίας.
- Η συμπεριφορά του χρονοδρομολογητή μπορεί να είναι κρίσιμη ακόμα και για ολόκληρη την συμπεριφορά του συστήματος
- Οι μεθοδολογίες που μελετήθηκαν για τους χρονοδρομολογητές ήταν ουσιαστικές για την έρευνα των λειτουργιών ενός ΛΣ
- Τα προβλήματα των χρονοδρομολογητών είναι ένα από τα πιο ενδιαφέροντα πεδία της έρευνας των υπολογιστών.

Οι αποφάσεις του χρονοδρομολογητή μπορούν να συμβούν κάτω από τέσσερις περιπτώσεις[17]:

1. Όταν μία διεργασία αλλάζει από εκτελέσιμη κατάσταση σε κατάσταση αναμονής (για παράδειγμα μία αίτηση εισόδου/εξόδου ή αναμονή για τερματισμό μιας θυγατρικής διεργασίας)
2. Όταν μία διεργασία αλλάζει από εκτελέσιμη κατάσταση σε κατάσταση ετοιμότητας (για παράδειγμα μία διακοπή (interrupt) συνέβη).
3. Όταν μια διεργασία αλλάζει από κατάσταση αναμονής σε κατάσταση ετοιμότητας (για παράδειγμα ολοκλήρωση διαδικασίας εισόδου/εξόδου).
4. Όταν μία διεργασία τερματιστεί.

Στις περιπτώσεις 1 και 4 δεν υπάρχει επιλογή από την άποψη του χρονοπρογραμματισμού. Αν υπάρχει μία διεργασία στην ready list πρέπει να επιλεχθεί για εκτέλεση. Υπάρχει ωστόσο επιλογή στις περιπτώσεις 2 και 3. Όταν ο χρονοπρογραμματισμός συμβαίνει κάτω από τις περιπτώσεις 1 και 4 τότε λέμε ότι έχουμε **μη προεκχωρητικό χρονοπρογραμματισμό (nonpreemptive scheduling)**. Στην άλλη περίπτωση λέμε ότι έχουμε **χρονοπρογραμματισμό προεκχώρισης (preemptive scheduling)**.

Στον μη προεκχωρητικό χρονοπρογραμματισμό όταν μία διεργασία δεσμεύσει τον επεξεργαστή, τον διατηρεί έως ότου τον ελευθερώσει είτε λόγο τερματισμού της είτε λόγο αλλαγής της σε κατάσταση αναμονής. Αυτός ο τύπος χρονοπρογραμματισμού χρησιμοποιήθηκε στα Windows 3.1 και από τα Λειτουργικά Συστήματα του Macintosh. Είναι η μόνη τεχνική που μπορεί να χρησιμοποιηθεί σε συγκεκριμένες πλατφόρμες λόγω του ότι δεν απαιτεί ειδικό hardware (για παράδειγμα, ένα ρολόι (timer)) όπως απαιτείται για τον χρονοπρογραμματισμό προεκχώρισης.

Όπως έχουμε αναφέρει σε αυτή την ενότητα, μία διεργασία μπορεί να αναβάλει την εκτέλεσή της σε τυχαίες χρονικές στιγμές χωρίς προειδοποίηση ώστε μία άλλη διεργασία να εκτελεστεί. Αυτό οδηγεί σε συνθήκες ανταγωνισμού και στην απαραίτητη χρήση σηματοφόρων ή κάποιας άλλης σύνθετης μεθόδου αποφυγής τους. Από την άλλη, μία πολιτική στην οποία θα επιτρέπουμε σε μία διεργασία να εκτελείται όσο χρειάζεται για την ολοκλήρωσή της θα σήμαινε άρνηση υπηρεσιών σε όλες τις άλλες διεργασίες.

Για παράδειγμα ας φανταστούμε μία διεργασία η οποία υπολογίζει δισεκατομμύρια ψηφία του π. Μία τέτοια διεργασία θα σήμαινε άρνηση υπηρεσιών για αόριστο χρονικό διάστημα.

Έτσι ερχόμαστε στην ανάγκη εύρεσης ενός αλγορίθμου που να αποφασίζει ποια από τις διεργασίες που βρίσκονται στην λίστα ετοιμότητας θα πρέπει να δεσμεύσει τον επεξεργαστή. Μερικοί από τους πιο γνωστούς αλγόριθμους είναι:

- First Come, First Served
- Shortest Job First
- Priority Scheduling
- Round Robin
- Multiple Queues
- Lottery

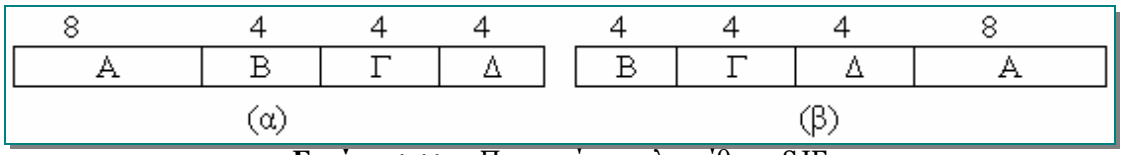
1.7.1 First Come, First Served

Ο αλγόριθμος αυτός είναι ίσως ο πιο εύκολος αλγόριθμος χρονοπρογραμματισμού. Ο αλγόριθμος FCFS ανήκει στο μη προεκχωρητικό χρονοπρογραμματισμό. Με τον αλγόριθμο αυτό, όποια διεργασία ζητήσει πρώτη την CPU αυτή θα είναι η πρώτη που θα την δεσμεύσει. Η υλοποίηση του αλγόριθμου χρονοπρογραμματισμού FCFS βασίζεται όπως είναι φανερό σε μία ουρά FIFO. Όταν μία διεργασία εισέλθει στην λίστα ετοιμότητας, το PCB της συνδέεται στο τέλος της ουράς. Όταν ο επεξεργαστής είναι ελεύθερος, τον δεσμεύει η διεργασία που βρίσκεται στην κορυφή της ουράς και αφαιρείται από την ουρά. Παρόλο που ο αλγόριθμος FCFS είναι εύκολος για να υλοποιηθεί αγνοεί όλα τα κριτήρια τα οποία επηρεάζουν την επίδοση. Το μειονέκτημα αυτού του αλγόριθμου είναι ο σχετικά μεγάλος χρόνος αναμονής καθώς επίσης και τα προβλήματα που δημιουργεί σε συστήματα time-sharing όπου ο κάθε χρήστης χρειάζεται το μερίδιό του από τον επεξεργαστή ανά τακτά χρονικά διαστήματα. Θα είναι καταστροφικό να επιτρέψουμε σε μία διεργασία να κρατήσει δεσμευμένο τον επεξεργαστή για μεγάλο χρονικό διάστημα.

1.7.2 Shortest Job First

Όλοι οι αλγόριθμοι που θα αναφέρουμε αφορούν αλληλεπιδραστικά (interactive) συστήματα. Ο συγκεκριμένος αλγόριθμος αφορά κυρίως εργασίες δέσμης (batch jobs) των οποίων οι χρόνοι εκτέλεσης είναι γνωστοί εκ των προτέρων. Ο αλγόριθμος SJF επιλέγει την διεργασία η οποία έχει τον μικρότερο χρόνο εκτέλεσης. Ο αλγόριθμος αυτός ελαχιστοποιεί το μέσο χρόνο αναμονής γιατί εξυπηρετούνται οι μικρές διεργασίες πριν από τις μεγάλες διεργασίες. Ενώ όμως ελαχιστοποιεί το μέσο χρόνο αναμονής, καταδικάζει τις διεργασίες με μεγάλο χρόνο εκτέλεσης. Για παράδειγμα, αν η λίστα ετοιμότητας είναι κατακερματισμένη τότε οι μεγάλες διεργασίες τείνουν να ξεχνιούνται στην λίστα ενώ οι μικρότερες διεργασίες εκτελούνται. Στην

εξαιρετική περίπτωση που το σύστημα έχει λίγους χρόνους αδράνειας (idle time) οι μεγάλες διεργασίες δεν πρόκειται να εκτελεστούν ποτέ. Η παρακάτω εικόνα παρουσιάζει ένα παράδειγμα του αλγορίθμου SJF σύμφωνα με το διάγραμμα Gantt.



Εικόνα 1.11 - Παρουσίαση αλγορίθμου SJF

1.7.3 Priority Scheduling

Ο αλγόριθμος SJF είναι μία ειδική περίπτωση αλγορίθμου priority – scheduling. Μία προτεραιότητα σχετίζεται με την κάθε διεργασία και ο επεξεργαστής δεσμεύεται από την διεργασία με την μεγαλύτερη προτεραιότητα. Στον αλγόριθμο FCFS μπορούμε να φανταστούμε ότι κάθε διεργασία έχει την ίδια προτεραιότητα με τις υπόλοιπες. Οι προτεραιότητες στον αλγόριθμο priority scheduling μπορούν να ανατεθούν στατικά ή δυναμικά. Για παράδειγμα στο ΛΣ Unix υπάρχει η εντολή nice με την οποία ο χρήστης μπορεί να αλλάξει την προτεραιότητα μιας διεργασίας.

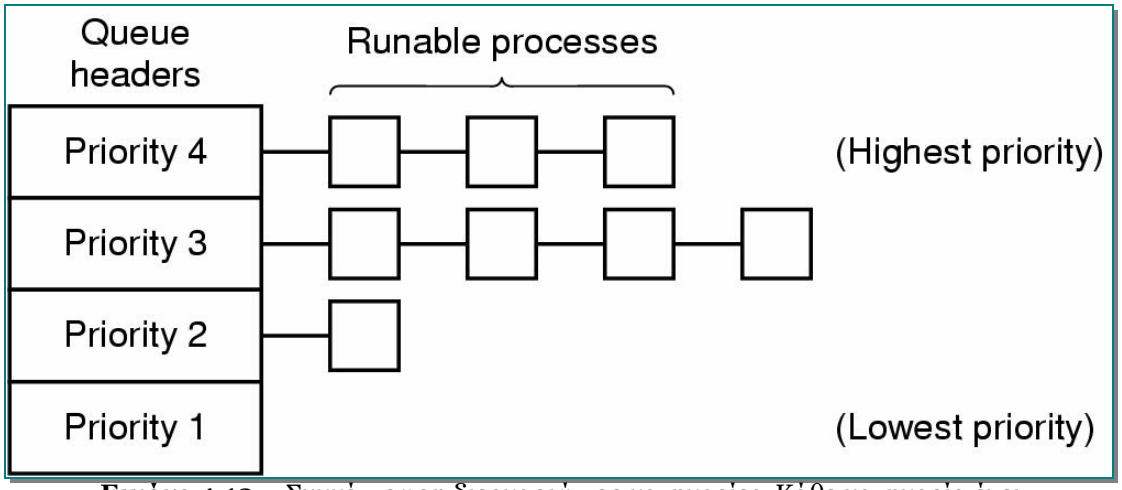
Ακόμα και σε ένα PC με ένα μόνο ιδιοκτήτη, μπορεί να υπάρχουν πολλές διεργασίες, κάποιες πιο σημαντικές από κάποιες άλλες. Για παράδειγμα, ένα daemon που επεξεργάζεται το ηλεκτρονικό ταχυδρομείο στο παρασκήνιο (background) θα του δοθεί χαμηλότερη προτεραιότητα σε σχέση με μία διεργασία που προβάλει ένα video στην οθόνη σε πραγματικό χρόνο (real-time).

Για να εμποδίσει τις διεργασίες με υψηλή προτεραιότητα να εκτελούνται αόριστα, ο χρονοδρομολογητής μπορεί να μειώσει την προτεραιότητα της τρέχουσας εκτελούμενης διεργασίας κάθε αίτηση διακοπής του ρολογιού (clock interrupt). Σε περίπτωση που αυτή η μείωση στην προτεραιότητα της διεργασίας πέσει κάτω από την διεργασία με την αμέσως μεγαλύτερη προτεραιότητα, τότε οι διεργασίες εναλλάσσονται. Διαφορετικά, σε κάθε διεργασία μπορεί να ανατεθεί ένα μέγιστο χρονικό όριο το οποίο θα δεσμεύει τον επεξεργαστή. Όταν ξεπεράσει αυτό το χρονικό όριο, ο χρονοδρομολογητής θα δώσει την ευκαιρία να εκτελεστεί η διεργασία με την αμέσως μεγαλύτερη προτεραιότητα.

Το μειονέκτημα αλγόριθμου Priority Scheduling είναι ότι οι διεργασίες με χαμηλή προτεραιότητα μπορεί να μην εκτελεστούν ποτέ. Το πρόβλημα αυτό που συναντάμε σε αυτού του τύπου αλγορίθμους ονομάζεται **αόριστη διακοπή (indefinite blocking)** ή **λιμοκτονία (starvation)**. Μία διεργασία είναι που έτοιμη για εκτέλεση αλλά στερείται τον επεξεργαστή θεωρείται μπλοκαρισμένη περιμένοντας τον επεξεργαστή. Σε ένα φορτωμένο σύστημα με μία σταθερή ροή διεργασιών με μεγάλη προτεραιότητα υπάρχει το ενδεχόμενο να εμποδίσει διεργασίες με χαμηλή προτεραιότητα να δεσμεύσουν τον επεξεργαστή.

Μία λύση σε αυτό το πρόβλημα είναι η ωρίμανση (aging) στην οποία ο χρονοδρομολογητής ανεβάζει την προτεραιότητα των διεργασιών που βρίσκονται αρκετή ώρα στην λίστα ετοιμότητας.

Πολύ συχνά είναι βολικό να συγκεντρώνουμε διεργασίες σε κατηγορίες και να χρησιμοποιούμε προτεραιότητες ανάμεσα στις κατηγορίες αλλά μέσα στην κατηγορία οι διεργασίες να αλλάζουνε με τον αλγόριθμο Round Robin που θα αναλύσουμε αμέσως μετά. Η εικόνα 1.12 παρουσιάζει αυτή την τεχνική.

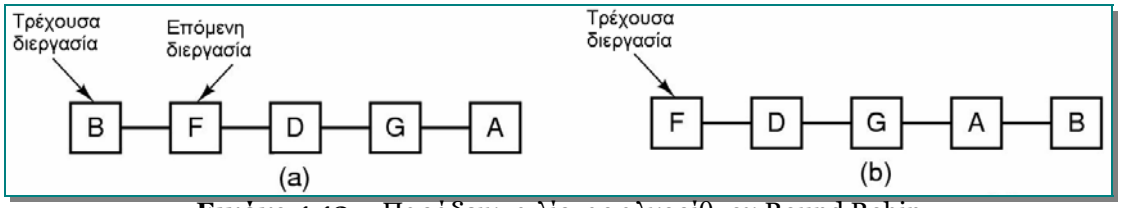


Εικόνα 1.12 – Συγκέντρωση διεργασιών σε κατηγορίες. Κάθε κατηγορία έχει μία προτεραιότητα.

1.7.4 Round Robin

Ο αλγόριθμος Round Robin σχεδιάστηκε ειδικά για τα time-sharing συστήματα. Είναι παρόμοιος με τον αλγόριθμο FCFS αλλά έχει προστεθεί και η στρατηγική της προεκχώρησης στην εναλλαγή των διεργασιών. Ο αλγόριθμος RR είναι ίσως ο πιο διαδεδομένος και χρησιμοποιείται πιο συχνά σε σχέση με τους άλλους αλγόριθμους. Η φιλοσοφία είναι απλή. Αν έχουμε n διεργασίες, κάθε διεργασία θα λάβει το 1/n χρόνο επεξεργασίας. Ο χρόνος αυτός ονομάζεται **κβάντο (quantum)**.

Ο αλγόριθμος RR είναι εύκολος στην υλοποίηση του. Το μόνο που έχει να κάνει ο χρονοδρομολογητής είναι να διατηρήσει μία λίστα με τις εκτελούμενες διεργασίες. Όταν μία διεργασία χρησιμοποιήσει το κβάντο της, μπαίνει στο τέλος της λίστας όπως φαίνεται στην εικόνα 1.13.



Εικόνα 1.13 – Παράδειγμα λίστας αλγορίθμου Round Robin.

1.7.5 Multiple Queues

Ένας από τους πρώτους χρονοδρομολογητές ήτανε αυτός στο ΛΣ CTSS. Το πρόβλημα με το CTSS ήταν ότι η εναλλαγή των διεργασιών γινόταν πολύ αργά γιατί ο υπολογιστής 7094 μπορούσε να διατηρήσει στην μνήμη μία μόνο διεργασία. Κάθε εναλλαγή διεργασίας σήμαινε ανταλλαγή της τρέχουσας διεργασίας από την μνήμη στον δίσκο και ανάγνωση την νέας διεργασίας από τον δίσκο στην μνήμη. Οι σχεδιαστές του CTSS γρήγορα διαπίστωσαν ότι είναι αποδοτικότερο να δίνουνε στις διεργασίες που ζητάνε συχνά τον επεξεργαστή μεγάλα κβάντα πολύ συχνά από το να δίνουνε σε όλες τις διεργασίες μικρά κβάντα με μεγάλη συχνότητα (για να μειώσουνε τις ανταλλαγές με τον σκληρό δίσκο). Από την άλλη, αν δίνουμε μεγάλα κβάντα στις διεργασίες θα σήμαινε μικρό χρόνο απόκρισης. Η λύση που έδωσαν ήταν να δημιουργήσουν κατηγορίες προτεραιότητας (priority classes). Οι διεργασίες στην μεγαλύτερη κατηγορία θα εκτελούνται για ένα κβάντο. Οι διεργασίες στην αμέσως μεγαλύτερη κατηγορία θα εκτελούνται για 2 κβάντα. Οι διεργασίες στην αμέσως μεγαλύτερη κατηγορία θα εκτελούνται για 4 κβάντα και ούτω καθεξής. Όταν μία διεργασία χρησιμοποιούσε όλα τα κβάντα που ήτανε διαθέσιμα για αυτήν κατέβαινε μία κατηγορία.

Για παράδειγμα ας μελετήσουμε μία διεργασία η οποία για να ολοκληρωθεί χρειάζεται 100 κβάντα. Στην αρχή θα της δοθεί ένα κβάντο και ύστερα θα γίνει εναλλαγή. Την επόμενη φορά που θα εκτελεστεί, θα της δοθούν 2 κβάντα πριν γίνει ξανά η εναλλαγή. Στις επόμενες εκτελέσεις θα της δοθούν 4,8,16,32 και 64 κβάντα, αν και θα χρησιμοποιήσει μόνο τα 37 από τα τελευταία 64 που θα της δοθούν για να ολοκληρωθεί. Μόνο 7 εναλλαγές θα χρειαστούν (μαζί με την πρώτη που φορτώθηκε) σε σχέση με τις 100 εναλλαγές ενός αλγόριθμου Round Robin. Επίσης καθώς η διεργασία θα βυθίζεται όλο και πιο βαθιά στις κατηγορίες, θα εκτελείται όλο και λιγότερο συχνά γλιτώνοντας τον επεξεργαστή για άλλες μικρές, αλληλεπιδραστικές διεργασίες.

Ο IBM 7094 και το CTSS

Στα μέσα της δεκαετίας του 1960, ο υπολογιστής της IBM 7094 ήταν ο μεγαλύτερος και πιο γρήγορος υπολογιστής μεγάλης ισχύος (mainframe), ο οποίος μπορούσε να χειρίζεται αριθμούς κινητής υποδιαστολής με μία ταχύτητα της τάξης των 0,32 MIPS (Million Instructions Per Hour – εκατομμύρια εντολές την ώρα). Ήταν ένας υπολογιστής δεύτερης γενιάς και ήταν σχεδιασμένος για επιστημονικές και τεχνολογικές εφαρμογές μεγάλης κλίμακας (large scale).

Το ΛΣ που διέθετε ήταν το CTSS (Compatible Time Sharing System) το οποίο ήτανε γραμμένο από το Project Mac του MIT και ήταν το πρώτο ΛΣ που διέθετε τεχνικές διαμοιρασμού χρόνου (time sharing). Αυτός ήταν και ο λόγος που άσκησε μεγάλη επιρροή σε μεταγενέστερα ΛΣ με δημοφιλέστερο, τον απόγονό του, το Multics το οποίο είναι πρόγονος του Unix. Το Multics είναι και αυτό παιδί του Project Mac.

1.7.6 Lottery

Μία τελείως διαφορετική προσέγγιση παρουσιάζει ο αλγόριθμος χρονοπρογραμματισμού lottery. Η βασική ιδέα είναι η εξής. Κάθε διεργασία έχει λαχεία για διάφορους πόρους του συστήματος όπως ο επεξεργαστής. Κάθε φορά που μία διαδικασία επιλογής χρονοπρογραμματισμού πρέπει να κάνει μία επιλογή, ένας λαχνός επιλέγεται στην τύχη και η διεργασία που κρατάει αυτό το λαχνό δεσμεύει τον πόρο. Όταν η τεχνική αυτή αφορά τον χρονοπρογραμματισμό των διεργασιών, το σύστημα μπορεί να πραγματοποιεί κληρώσεις 50 φορές το δευτερόλεπτο και κάθε νικήτρια διεργασία κερδίζει 20msec από τον επεξεργαστή.

Ο αλγόριθμος lottery παρουσιάζει μερικές πολύ ενδιαφέρουσες ιδιότητες. Για παράδειγμα, όταν μία νέα διεργασία εμφανιστεί και της δοθούν μερικοί λαχνοί στην επόμενη κλήρωση θα υπάρχει το ενδεχόμενο να νικήσει ανάλογα με τον αριθμό των λαχνών που έχει. Με άλλα λόγια, ο αλγόριθμος lottery ανταποκρίνεται σε κάθε διεργασία.

1.8 ■ Σήματα και διαχείριση σημάτων

Παρόλο που οι περισσότερες μορφές διαδιεργασιακής ενδοεπικοινωνίας είναι από πριν σχεδιασμένες, υπάρχουν καταστάσεις στις οποίες απαιτείται μη αναμενόμενη ανάγκη για επικοινωνία. Για παράδειγμα εάν ο χρήστης κατά λάθος ζητήσει από ένα επεξεργαστή κειμένου να εκτυπώσει τα περιεχόμενα ενός τεράστιου αρχείου και αργότερα συνειδητοποιήσει το λάθος του θα πρέπει να υπάρχει ένας τρόπος για να διακόψει τον επεξεργαστή κειμένου. Στο Unix και στο Linux μπορεί να πατήσει τον συνδυασμό πλήκτρων Ctrl-C, οποίος στέλνει ένα σήμα διακοπής στο πρόγραμμα. Το πρόγραμμα λαμβάνει το σήμα και σταματάει την εκτύπωση.

Με άλλα λόγια, ένα **σήμα (signal)** είναι μία ειδοποίηση σε μία διεργασία ότι ένα συνέβη ένα γεγονός. Στο παράδειγμά μας, ο χρήστης ζήτησε μία διακοπή. Υπάρχουνε πολλά σήματα τα οποία δεν παράγονται από τον χρήστη αλλά από το ίδιο το λειτουργικό σύστημα ή από άλλες διεργασίες. Παραδείγματα τέτοιων σημάτων είναι η εξαίρεση (exception) σε υπολογισμό αριθμών κινητής υποδιαστολής, παράνομη (illegal) εντολή, λανθασμένη διεύθυνση μνήμης (memory segment violation) ή ακόμα και από μία διεργασία σε μία άλλη με σκοπό την ανταλλαγή πληροφοριών ή για να αλλάξει συμπεριφορά.

Ένα σήμα μπορεί να ληφθεί είτε συγχρονισμένα είτε ασύγχρονα ανάλογα με την πηγή και τον λόγο για τον οποίο το γεγονός παρήγαγε το σήμα. Παρ'όλα αυτά ότι και αν είναι το σήμα (συγχρονισμένο ή ασύγχρονο), όλα τα σήματα ακολουθούν το ίδιο μοτίβο[18]:

- Ένα σήμα παράγεται μετά από εμφάνιση ενός συγκεκριμένου γεγονότος
- Ένα σήμα προορίζεται για μία διεργασία
- Όταν το σήμα παραληφθεί από την διεργασία πρέπει να το χειριστεί.

Ένα παράδειγμα ενός συγχρονισμένου σήματος είναι όταν μία διεργασία προσπαθήσει να πραγματοποιήσει διαίρεση με το μηδέν. Τότε ένα σήμα παράγεται και στέλνεται στην ίδια την διεργασία. Ένα συγχρονισμένο σήμα παραδίδεται στην ίδια την διεργασία η οποία εκτέλεσε την λειτουργία η οποία παρήγαγε το σήμα.

Όταν ένα σήμα παράγεται από ένα γεγονός έξω από την εκτελούμενη διεργασία τότε λέμε ότι η διεργασία έλαβε το σήμα ασύγχρονα. Ένα παράδειγμα τέτοιου σήματος είναι ο τερματισμός μιας διεργασίας από πλήκτρα όπως Ctrl-C ή λήξης ενός μετρητή (timer expire). Γενικά, ένα ασύγχρονο σήμα στέλνεται σε μία άλλη διεργασία.

Κάθε σήμα μπορεί να χειρισθεί (handled) με έναν από τους δύο πιθανούς **διαχειριστές (handlers):**

1. Τον εξ' ορισμού διαχειριστή
2. Έναν διαχειριστή ορισμένο από τον χρήστη

Κάθε σήμα έχει έναν εξ' ορισμού χειριστή ο οποίος εκτελείται από τον πυρήνα όταν χειρίζεται το σήμα. Η εξ' ορισμού ενέργεια του διαχειριστή μπορεί να παρακαμφθεί από έναν διαχειριστή ορισμένο από τον χρήστη. Στην περίπτωση που ο χρήστης έχει ορίσει δικό του διαχειριστή τότε αυτός είναι που θα κληθεί για την διαχείριση του σήματος και όχι ο εξ' ορισμού διαχειριστής. Ορισμένα σήματα μπορούν να αγνοηθούν ενώ άλλα μπορούν να τερματίσουν την εκτέλεση της διεργασίας.

Όταν ένα σήμα στέλνεται σε μία διεργασία και η διεργασία αυτή δεν έχει ανακοινώσει την επιθυμία της να δεχτεί το σήμα αυτό η διεργασία αυτή σκοτώνεται (killed) χωρίς δεύτερη ευκαιρία. Συνήθως δημιουργείται ένα αρχείο απόρριψης πυρήνα (core dump). Το αρχείο αυτό, με όνομα core τοποθετείται στον τρέχοντα κατάλογο και είναι ένα αντίγραφο (image) της διεργασίας. Το αρχείο αυτό είναι χρήσιμο πολλές φορές κατά την αποσφαλμάτωση του προγράμματος (debugging).

Για να αποφύγουμε αυτή τη μοίρα, μία διεργασία μπορεί να χρησιμοποιήσει την κλήση συστήματος **sigaction** για να ανακοινώσει ότι είναι προετοιμασμένη να δεχτεί κάποιου τύπου σήματα και παρέχει την διεύθυνση της διαδικασίας η οποία θα διαχειριστεί το σήμα.

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

Το πρότυπο SUS2 (Single Unix Specification Version 2) καθορίζει 28 σήματα. Παρακάτω θα δούμε τα ονόματα μερικών σημάτων όπως αυτά ορίζονται στο αρχείο επικεφαλίδας signal.h. Όλα ξεκινάνε με το πρόθεμα SIG.

Όνομα σήματος	Περιγραφή
SIGABORT	Διακοπή διεργασίας
SIGALRM	Ρολόι συναγερμού (alarm clock)
SIGFPE	Εξαίρεση αριθμού κινητής υποδιαστολής
SIGHUP	Προσωρινή διακοπή (hangup)
SIGILL	Παράνομη εντολή
SIGINT	Διακοπή τερματικού
SIGKILL	Σκότωμα (kill)
SIGPIPE	Εγγραφή σε διοχέτευση χωρίς αναγνώστη
SIGQUIT	Τερματισμός τερματικού
SIGSEGV	Παράνομη πρόσβαση σε περιοχή μνήμης
SIGTERM	Τερματισμός
SIGUSR1	Σήμα ορισμένο από τον χρήστη 1
SIGUSR2	Σήμα ορισμένο από τον χρήστη 2

Πίνακας 1.5 – Ονόματα και περιγραφές γνωστών σημάτων

Μπορούμε να παράγουμε όποιο σήμα θέλουμε με την εντολή κελύφους kill και killall. Επίσης μπορούμε να παράγουμε ένα σήμα και μέσα από κώδικα C με τις κλήσεις συστήματος **kill, killpg, abort**.

```
#include <signal.h>

int kill(pid_t pid, int signum);
```

Η κλήση συστήματος kill στέλνει οποιοδήποτε σήμα, όχι μόνο το σήμα SIGKILL, σε οποιαδήποτε διεργασία στην οποία έχει το δικαίωμα να στέλνει σήματα. Άδεια έχει όταν εκτελείται από τον διαχειριστή του συστήματος (supervisor) ή αν το πραγματικό (real) ή το λειτουργικό (effective) ID του χρήστη της διεργασίας που στέλνει το σήμα ταιριάζει με το πραγματικό (real) ή αποθηκευμένο (saved) ID του χρήστη της διεργασίας που δέχεται το σήμα[19].

```
#include <signal.h>

int killpg(pid_t pgrp, int signum);
```

Η κλήση συστήματος παράγει ένα σήμα το οποίο στέλνεται στο process group του οποίου το process-group ID καθορίζεται με την παράμετρο pgrp. Η κλήση συστήματος killpg είναι ουσιαστικά μία άχρηστη κλήση διότι μπορούμε να πετύχουμε το ίδιο αποτέλεσμα με την κλήση: kill(-pgrp, signum);

```
#include <stdlib.h>

void abort(void);
```

Η κλήση abort μοιάζει σαν την kill αλλά στέλνει μόνο το σήμα SIGABRT. Με την κλήση αυτή η διεργασία τερματίζει, εκτός και αν η διεργασία έχει ορίσει διαχειριστή για το σήμα αυτό.

ΚΕΦΑΛΑΙΟ 2ο

Διερμηνευτές

Εισαγωγή

Στο δεύτερο κεφάλαιο της εργασίας αυτής θα αναφερθούμε στους διερμηνευτή (interpreters). Ένα κέλυφος (shell – command interpreter) ενός Λειτουργικού Συστήματος είναι μία κατηγορία διερμηνευτή ο οποίος πολλές φορές παρέχει και εργαλεία προγραμματισμού. Ως εκ τούτου θα μελετήσουμε τον τρόπο υλοποίησης τους. Αρχικά θα σχηματίσουμε το απαραίτητο θεωρητικό υπόβαθρο με την μελέτη γραμματικών, γλωσσών και τεχνικών λεκτικής και συντακτικής ανάλυσης. Με το υπόβαθρο αυτό θα μελετήσουμε την λεκτική και συντακτική ανάλυση ενός αρχικού προγράμματος αλλά και θα προγραμματίσουμε λεκτικούς και συντακτικούς αναλυτές με χειρονακτικό τρόπο αλλά και αυτόματα με την βοήθεια εργαλείων υλοποίησης διερμηνευτών και μεταγλωττιστών όπως το Flex και ο Bison.

2.1 ■ Επεξεργαστές γλωσσών προγραμματισμού

Ένας **επεξεργαστής γλώσσας προγραμματισμού (programming language processor)** είναι κάθε σύστημα το οποίο διαχειρίζεται προγράμματα καθορισμένα με κάποια συγκεκριμένη γλώσσα προγραμματισμού. Ο ορισμός των επεξεργαστών γλωσσών προγραμματισμού είναι πολύ γενικός και περιγράφει μία ποικιλία συστημάτων όπως[1]:

- **Διορθωτές κειμένου (editors)**. Ένας διορθωτής κειμένου επιτρέπει την εισαγωγή πηγαίου κώδικα (source code) ενός προγράμματος, την τροποποίησή του και την αποθήκευσή του. Ένας διορθωτής κειμένου (text editor) μας επιτρέπει να επεξεργαστούμε οποιοδήποτε έγγραφο κειμένου (όχι απαραίτητα πηγαίο κώδικα προγράμματος).

- **Μεταφραστές (translators)** και **μεταγλωττιστές (compilers)**. Ένας μεταφραστής μεταφράζει ένα πηγαίο κώδικα από μία γλώσσα προγραμματισμού η οποία ονομάζεται και **γλώσσα προέλευσης (source language)** σε μία άλλη που ονομάζεται **γλώσσα προορισμού (target language)**. Συγκεκριμένα, ο μεταγλωττιστής (compiler) μεταφράζει ένα πρόγραμμα γραμμένο σε μία γλώσσα υψηλού επιπέδου (high-level language) (πχ C/C++) σε μία γλώσσα χαμηλού επιπέδου (low level language), προετοιμάζοντας το πρόγραμμα για εκτέλεση στο συγκεκριμένο μηχάνημα. Μία γλώσσα χαμηλού επιπέδου είναι μία συμβολική γλώσσα της οποίας κάθε εντολή αποτελεί στην ουσία μία αντιστοιχία με μία εντολή μηχανής. Η εντολή μηχανής είναι αυτή που καταλαβαίνει ο υπολογιστής και μπορεί να την εκτελέσει άμεσα. Επίσης, πριν πραγματοποιήσει την μετάφραση, ελέγχει το πηγαίο κώδικα του προγράμματος για συντακτικά και γραμματικά λάθη.

- **Διερμηνευτές (interpreters)**. Ένας διερμηνευτής δέχεται ένα πηγαίο κώδικα ενός προγράμματος μιας συγκεκριμένης γλώσσας προγραμματισμού και είτε το εκτελεί αμέσως, είτε το τοποθετεί σε μία στοίβα και το εκτελεί από εκεί. Αυτό το είδος της εκτέλεσης, παραλείπει την διαδικασία της μετάφρασης του πηγαίου κώδικα σε κώδικα χαμηλού επιπέδου (compilation).

Στην πράξη, όλες οι κατηγορίες επεξεργαστών γλωσσών προγραμματισμού που αναφέραμε χρησιμοποιούνται κατά την διαδικασία της ανάπτυξης λογισμικού. Σε ένα τυπικό σύστημα οι επεξεργαστές γλωσσών προγραμματισμού μπορούν να συναντηθούν με δύο μορφές. Σε συστήματα Unix θα τους συναντήσουμε με την μορφή ξεχωριστών εργαλείων μία μορφή την οποία το Unix ονομάζει **φιλοσοφία των "εργαλείων λογισμικού" ("software tools" philosophy)**. Η άλλη μορφή που μπορούμε να συναντήσουμε τους επεξεργαστές γλωσσών προγραμματισμού είναι τα **ολοκληρωμένα περιβάλλοντα ανάπτυξης (IDE – Integrated Development Environment)**. Στα ολοκληρωμένα αυτά περιβάλλοντα ανάπτυξης συναντάμε όλα τα απαραίτητα εργαλεία για την ανάπτυξη εφαρμογών όπως editors, interpreters, compilers, linkers, debuggers αλλά και υπηρεσίες όπως αναζήτηση και αντικατάσταση κειμένου καθώς και διαχείρισης του project. Ένα καλό παράδειγμα ολοκληρωμένου περιβάλλοντος ανάπτυξης είναι τα IDE της Borland τα οποία κυκλοφορούν σε εκδόσεις για Linux και για Windows.

2.2 ■ Διερμηνευόμενες γλώσσες και εκτελούμενες γλώσσες

Όπως αναφέραμε και προηγουμένως ο μεταγλωττιστής (compiler) μεταφράζει ένα πρόγραμμα γραμμένο σε μία γλώσσα προορισμού σε μία γλώσσα χαμηλού επιπέδου η οποία μπορεί να είναι η γλώσσα assembly ή η γλώσσα μηχανής του συγκεκριμένου υπολογιστή. Στην περίπτωση που η μετάφραση γίνει σε γλώσσα **assembly** θα πρέπει να χρησιμοποιηθεί ένας ακόμα μεταφραστής, ο **assembler**, ο οποίος θα μεταφράσει το πρόγραμμα από assembly σε γλώσσα μηχανής. Στην συνέχεια μπορεί να χρησιμοποιηθεί ένα πρόγραμμα σύνδεσης (linker) το οποίο θα συνδέσει τον κώδικα μας με τον κώδικα από τις βιβλιοθήκες που πιθανότατα θα έχουμε συμπεριλάβει στον κώδικά μας. Σε περίπτωση που το πρόγραμμά μας έχει χωριστεί σε πολλά αρχεία, τότε ο linker θα συνδέσει όλα τα αρχεία ώστε να παράγει το τελικό εκτελέσιμο (executable) αρχείο.

Ο διερμηνευτής (interpreter) από την άλλη, δεν θα κάνει κανενός είδους μετάφραση. Θα αναλύσει τον πηγαίο κώδικα και ταυτόχρονα θα ξεκινήσει την εκτέλεση του προγράμματος χωρίς να παράγει κανένα εκτελέσιμο αρχείο. Για να πραγματοποιήσει την εκτέλεση του προγράμματος όμως θα χρειαστεί να αναλύσει τον κώδικα και πιθανότατα να δημιουργήσει ένα ενδιάμεσο κώδικα (intermediate code). Ο διερμηνευτής ενεργεί όπως θα ενεργούσε ένας άνθρωπος αν ήθελε να καταλάβει τι κάνει το πρόγραμμα χωρίς να χρησιμοποιήσει τον υπολογιστή. Πρώτα ελέγχει τον πηγαίο κώδικα για συντακτικά λάθη. Έπειτα ξεκινάει να εκτελεί μία-μία τις εντολές του

προγράμματος, ξεκινώντας από το entry point του προγράμματος. Επίσης, παρακολουθεί τις τιμές των μεταβλητών και αποθηκεύει τις τιμές τους σε δικές του μεταβλητές ή δομές δεδομένων.

Όπως βλέπουμε οι μεταγλωττιστές και οι διερμηνευτές έχουνε πολλά κοινά μεταξύ τους αλλά ταυτόχρονα έχουνε και πολλές διαφορές. Παρ'όλα αυτά όμως, και οι δύο είναι σημαντικοί. Υπάρχουν περιπτώσεις στις οποίες ο διερμηνευτής ταιριάζει περισσότερο και υπάρχουν άλλες περιπτώσεις στις οποίες ο διερμηνευτής δεν κάνει καθόλου για την δουλειά που θέλουμε.

Οι διερμηνευτές είναι πιο ευέλικτοι από τους μεταγλωττιστές. Για παράδειγμα, ας υποθέσουμε πως έχουμε ένα πρόγραμμα στο οποίο υπάρχει μία εντολή διαίρεσης με το μηδέν ή ένα άλλο οποιοδήποτε λογικό λάθος. Ένας διερμηνευτής θα εκτελέσει εντολή-εντολή το πρόγραμμα και όταν φτάσει στην εντολή διαίρεσης το μηδέν θα μπορέσει να σταματήσει την εκτέλεση, να ενημερώσει τον προγραμματιστή σε ποια γραμμή στον κώδικα βρίσκεται το λάθος και να επιτρέψει στον χρήστη να αλλάξει την λανθασμένη εντολή. Έπειτα ο διερμηνευτής είναι σε θέση να συνεχίσει κανονικά την εκτέλεση του προγράμματος. Ο μεταγλωττιστής όμως δεν είναι σε θέση να δώσει τέτοιες πληροφορίες αλλά ούτε και να συμπεριφερθεί με αντίστοιχο τρόπο. Ο μεταγλωττιστής θα εκτελέσει το εκτελέσιμο αρχείο και όχι τον πηγαίο κώδικα, άρα δεν είναι σε θέση να γνωρίζει σε ποια γραμμή στον πηγαίο κώδικα έγινε το λάθος. Επίσης, όταν φτάσει στην εντολή με το λογικό λάθος θα σταματήσει η εκτέλεση του προγράμματος χωρίς να υπάρχει δυνατότητα συνέχισής της από το σημείο του λάθος. Όταν σταματήσει η εκτέλεση από ένα λογικό λάθος το μόνο πράγμα που μπορεί να ενημερώσει τον προγραμματιστή ο μεταγλωττιστής είναι η διεύθυνση μνήμης στην οποία υπήρχε η λανθασμένη εντολή. Μία πληροφορία σχετικά άχρηστη γιατί κάθε φορά που θα εκτελούμε το εκτελέσιμο αρχείο θα φορτώνεται σε νέες διευθύνσεις μνήμης. Γενικά όμως ο μεταγλωττιστής είναι κατάλληλος για συντακτικά λάθη και επιπλέον υπάρχουν και έξυπνοι μεταγλωττιστές που μπορούν να βρουν σφάλματα κατά το χρόνο εκτελέσεως.

Όπως βλέπουμε όταν πρόκειται για λόγους αποσφαλμάτωσης (debugging) οι διερμηνευτές είναι σίγουρα προτιμότεροι από τους μεταγλωττιστές. Έτσι τα τελευταία χρόνια οι μεταγλωττιστές έχουν εξελιχθεί ώστε να ενσωματώσουν διάφορες τεχνικές αποσφαλμάτωσης. Στην ουσία, προσθέτουν διάφορες πληροφορίες αποσφαλμάτωσης μέσα στο εκτελέσιμο αρχείο που θα παράγουνε. Με τις πληροφορίες αυτές η εκτέλεση του προγράμματος μπορεί να γίνει υπό επιτήρηση και ο προγραμματιστής μπορεί να έχει στην διάθεσή του αρκετές πληροφορίες. Παρ'όλα αυτά η πρόσθεση πληροφοριών αποσφαλμάτωσης στο εκτελέσιμο αρχείο έχει σαν μειονέκτημα την μείωση επίδοσης του εκτελέσιμου προγράμματος. Έτσι, οι προγραμματιστές όταν τελειώσουν με την διαδικασία αποσφαλμάτωσης και επιθυμούν να παράγουν την τελική (final) έκδοση της εφαρμογής τους, ρυθμίζουν ανάλογα τον μεταγλωττιστή ώστε να μην προσθέσει τις επιπρόσθετες πληροφορίες αποσφαλμάτωσης, οι οποίες είναι άχρηστες για τον τελικό χρήστη και ταυτόχρονα μειώνουν την απόδοση της εφαρμογής.

Όσον αφορά την ταχύτητα εκτέλεσης ο μεταγλωττιστής φαίνεται να είναι αυτός που κυριαρχεί. Όπως είπαμε, ο διερμηνευτής αναλύει μία-μία τις εντολές του πηγαίου προγράμματος και τις εκτελεί. Η ανάλυση αυτή κοστίζει σημαντικά στην ταχύτητα εκτέλεσης. Ο μεταγλωττιστής πραγματοποιεί και αυτός ανάλυση του πηγαίου κώδικα, αλλά έπειτα παράγει ένα εκτελέσιμο αρχείο με κώδικα μηχανής. Η εκτέλεση του κώδικα μηχανής σε σχέση με την διερμηνεία του πηγαίου κώδικα από τον διερμηνευτή μπορεί να είναι 10 με 100 φορές ταχύτερη[2].

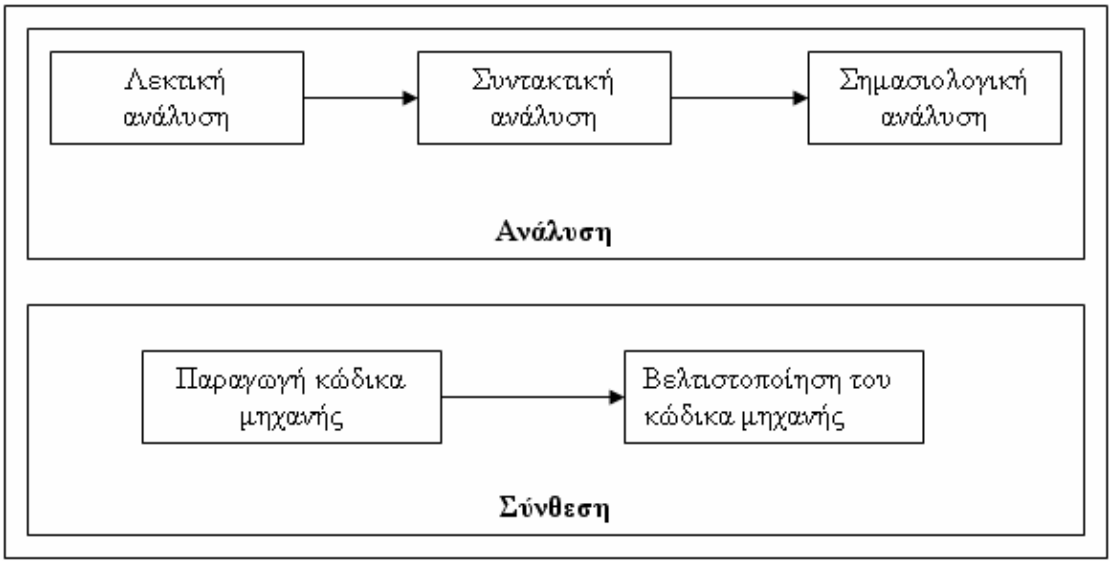
2.3 ■ Φάσεις μεταγλωττίσεως

Στην ενότητα αυτή θα μελετήσουμε την εσωτερική δομή των μεταγλωττιστών και των διερμηνευτών. Στην αρχή του κεφαλαίου αναφέραμε ότι οι μεταγλωττιστές και οι διερμηνευτές πραγματοποιούν ανάλυση του πηγαίου προγράμματος. Η ανάλυση αυτή χωρίζεται σε στάδια και κάθε στάδιο χωρίζεται με την σειρά του σε φάσεις.

Τα στάδια στα οποία χωρίζεται ένας μεταγλωττιστής είναι δύο. Η **ανάλυση** και η **σύνθεση**[3]. Στην ανάλυση ο πηγαίος κώδικας του προγράμματος αναλύεται ώστε να διαπιστωθεί η δομή του και η **σημασιολογία** του **(semantics)**. Στο στάδιο της σύνθεσης παράγεται ο κώδικας μηχανής.

Ένας διερμηνευτής αποτελείται από ένα μόνο στάδιο. Το στάδιο της ανάλυσης. Το στάδιο της σύνθεσης λείπει από τους διερμηνευτές γιατί όπως αναφέραμε, οι διερμηνευτές δεν παράγουνε κανένα εκτελέσιμο αρχείο.

Πολλές φορές στην βιβλιογραφία συναντάμε τα στάδια ανάλυσης και σύνθεσης με το όνομα μπροστά άκρο (front end) πίσω άκρο (back end).



Εικόνα 2.1 – Τα στάδια και οι φάσεις μεταγλώττισης

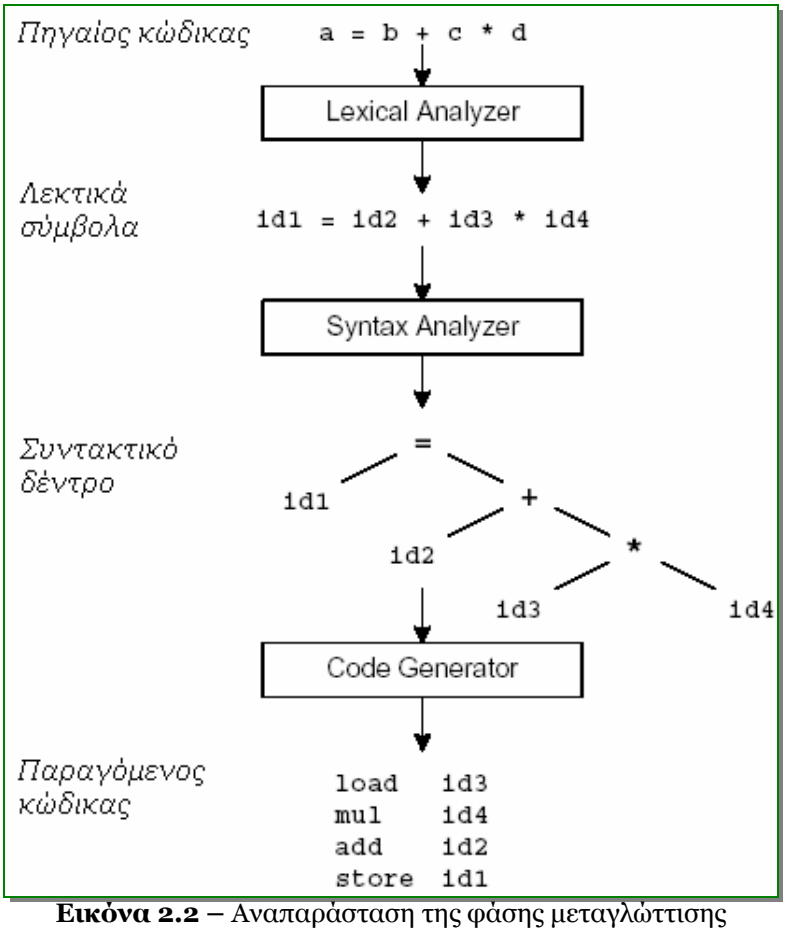
Όπως βλέπουμε και στην εικόνα 2.1, οι φάσεις από τις οποίες αποτελείται το στάδιο της ανάλυσης είναι τρεις:

- Λεκτική ανάλυση
- Συντακτική ανάλυση
- Σημασιολογική ανάλυση

Οι φάσεις της σύνθεσης είναι δύο:

- Παραγωγή κώδικα μηχανής
- Βελτιστοποίηση του κώδικα μηχανής

Στην συνέχεια θα μελετήσουμε την κάθε φάση ξεχωριστά, ξεκινώντας με τις τρεις φάσεις του σταδίου της ανάλυσης και έπειτα με τις δύο φάσεις του σταδίου της σύνθεσης. Πριν προχωρήσουμε πρέπει να σημειώσουμε ότι για να κατανοήσουμε απόλυτα την λειτουργία και τον τρόπο υλοποίηση ενός μεταγλωττιστή πρέπει να έχουμε στο μυαλό μας την εικόνα ενός μεταγλωττιστή όχι σαν ένα ενιαίο πρόγραμμα αλλά την κάθε φάση από την οποία αποτελείται σαν ξεχωριστή μονάδα. Το σύνολο των μονάδων αυτών αποτελούν τον μεταγλωττιστή. Η μονάδα που πραγματοποιεί την λεκτική ανάλυση ονομάζεται λεκτικός αναλυτής (ή σαρωτής – **scanner**) και η μονάδα που πραγματοποιεί συντακτική ανάλυση ονομάζεται συντακτικός αναλυτής (**parser**).



Εικόνα 2.2 – Αναπαράσταση της φάσης μεταγλώττισης

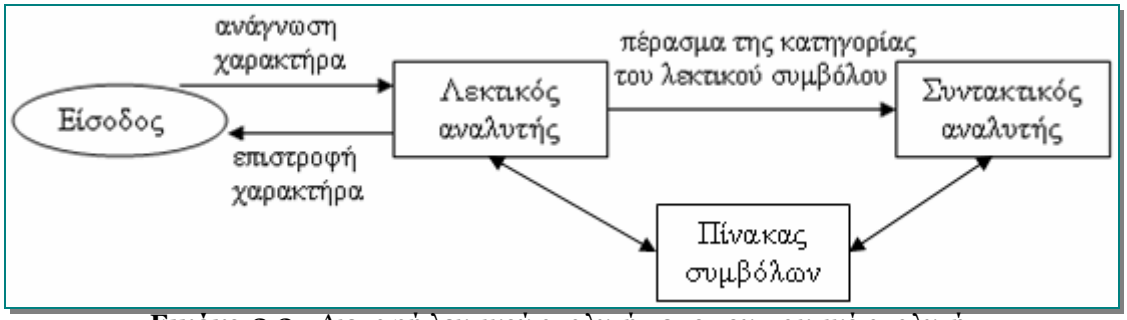
2.3.1 Λεκτική ανάλυση

Η λεκτική ανάλυση (**σάρωση - scanning**) είναι μία σχετικά απλή φάση στην οποία αναγνωρίζονται τα λεκτικά σύμβολα **(tokens)** μιας γλώσσας προγραμματισμού.

2.3.1.1 Λεκτικά σύμβολα

Ένα λεκτικό σύμβολο μπορεί να αποτελείται από πολλούς χαρακτήρες. Οι χαρακτήρες από τους οποίους αποτελείται το λεκτικό σύμβολο δεν έχουνε καμία αξία μόνοι τους. Το σύνολο των χαρακτήρων αποτελεί το σύμβολο. Για παράδειγμα ας υποθέσουμε ότι έχουμε το σύμβολο “integer”. Όπως βλέπουμε το λεκτικό αυτό σύμβολο αποτελείται από τους χαρακτήρες i,n,t,e,g,e,r. Όταν κατά την λεκτική ανάλυση συναντήσουμε τον χαρακτήρα i δεν συμβαίνει τίποτα. Κατά την συνέχεια όμως της λεκτικής ανάλυσης εντοπίζουμε και τους χαρακτήρες n,t,e,g,e,r το σύνολο των οποίων αποτελούν το λεκτικό σύμβολο “integer”. Οι χαρακτήρες έχουν αξία μόνο για να διαχωρίζουν τα σύμβολα μεταξύ τους. Για παράδειγμα, άλλο λεκτικό σύμβολο είναι το “let” και άλλο λεκτικό σύμβολο είναι το “lot”.

Η λεκτική ανάλυση πραγματοποιεί μία σάρωση του πηγαίου κώδικα για να ανακαλύψει τα λεκτικά σύμβολα. Κατά την σάρωση αυτή αγνοεί τις κενές γραμμές, τους κενούς χαρακτήρες (tabs, spaces) και τα σχόλια που πιθανών να υπάρχουν και αναγνωρίζει μόνο τα λεκτικά σύμβολα της γλώσσας. Ωστόσο σε πολλές περιπτώσεις τα σύμβολα αυτά αποτελούν μέρος της γλώσσας. Όταν ανακαλυφτεί ένα λεκτικό σύμβολο τότε κατηγοριοποιείται σε σχέση με το είδος του. Για παράδειγμα η Pascal ορίζει τύπους συμβόλων όπως: αναγνωριστές (**identifiers**), δεσμευμένες λέξεις, αριθμούς, συμβολοσειρές (strings) και ειδικά σύμβολα όπως ερωτηματικά, παρενθέσεις, αριθμητικούς τελεστές κλπ. Όπως βλέπουμε τα λεκτικά σύμβολα χαρακτηρίζονται απολύτως από την κατηγορία τους και από τον σχηματισμό των χαρακτήρων που το αποτελούν (ορθογραφία) η οποία στην βιβλιογραφία συναντάτε με τον όρο **lexeme**. Όταν αναγνωριστεί ένα λεκτικό σύμβολο τότε, στον συντακτικό αναλυτή θα παραδοθεί μόνο η κατηγορία του και όχι η ορθογραφία του. Η ορθογραφία ορισμένων (όχι όλων) λεκτικών συμβόλων θα αναλυθεί από τον σημασιολογικό αναλυτή. Η ορθογραφία ορισμένων λεκτικών σύμβολων (όπως για παράδειγμα το λεκτικό σύμβολο “include” της C) δεν θα αναλυθεί σε καμία φάση (εκτός από την λεκτική ανάλυση φυσικά).



Εικόνα 2.3– Διεπαφή λεκτικού αναλυτή με τον συντακτικό αναλυτή

Όταν ένα lexeme σχηματίσει ένα αναγνωριστή (identifier) πρέπει να υπάρχει ένας μηχανισμός με τον οποίο να προσδιορίζεται αν έχει ξαναεμφανιστεί. Ο μηχανισμός αυτός είναι ένας πίνακας που ονομάζεται **πίνακα συμβόλων (symbol table)**. Στον πίνακα αυτό αποθηκεύεται το lexeme του αναγνωριστή.

Πρέπει να τονίσουμε ξανά ότι η μόνη δουλειά της λεκτικής ανάλυσης είναι ο εντοπισμός και η αναγνώριση των λεκτικών συμβόλων και δεν ασχολείται καθόλου με την σειρά που εμφανίζονται τα λεκτικά σύμβολα. Για παράδειγμα, αν η λεκτική ανάλυση γινόταν σε ένα πρόγραμμα γραμμένο σε C, και το πρόγραμμα ήταν:

```
; number int return do = ++
```

τότε τα όλα τα λεκτικά σύμβολα θα αναγνωριστούν χωρίς κανένα πρόβλημα ασχέτως αν η σύνταξη είναι λανθασμένη.

Υπάρχουν δεκάδες εργαλεία τα οποία παράγουν λεκτικούς αναλυτές για μία γλώσσα με πιο γνωστό εργαλείο το lex και τον διάδοχό του, το flex για το οποίο θα αναφερθούμε σε επόμενο κεφάλαιο.

2.3.1.2 Κανονικές εκφράσεις

Για τις ανάγκες της λεκτικής ανάλυσης χρησιμοποιούμε τις **κανονικές εκφράσεις (Regular Expressions - RE)**. Οι κανονικές εκφράσεις είναι μία βολική μέθοδος για να περιγράφουμε αναγνωριστές (identifiers) και σταθερές.

Τα κύρια χαρακτηριστικά της σημειογραφίας των κανονικών εκφράσεων είναι :

- '|' διαχωρίζει εναλλακτικές εκφράσεις
- '*' δηλώνει ότι το προηγούμενο αντικείμενο μπορεί να επαναληφθεί μηδέν ή περισσότερες φορές.
- '(' και ')' είναι παρενθέσεις για ομαδοποίηση
- 'ε' δηλώνει κενό αλφαριθμητικό

Για παράδειγμα, ένας αναγνωριστής μπορεί να αναπαρασταθεί χρησιμοποιώντας κανονικές εκφράσεις ως:

letter (letter | digit)*

Μερικά έγκυρα αλφαριθμητικά που περιγράφονται με την παραπάνω κανονική έκφραση είναι:

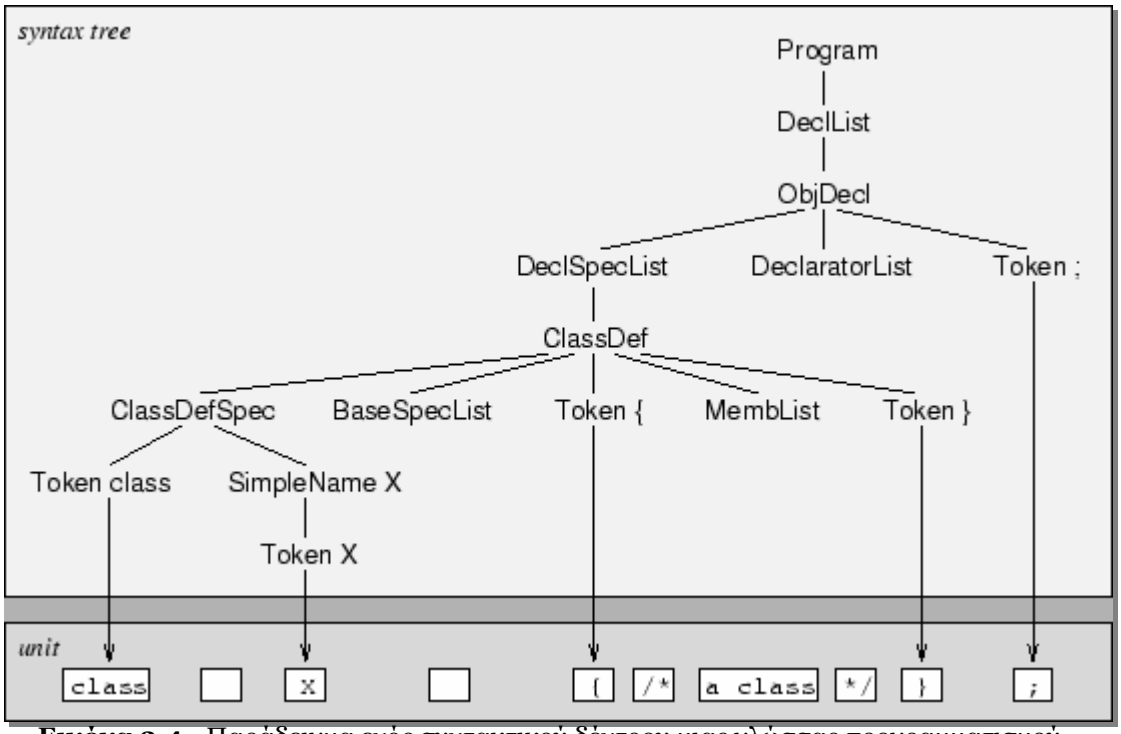
- **a**
- **aa**
- **a3a**
- **a33**
- **ab46c**

Οι κανονικές εκφράσεις παράγουν σύνολα αλφαριθμητικών με άλλα λόγια, παράγουν γλώσσες. Ωστόσο οι κανονικές εκφράσεις είναι ικανές να παράγουν μόνο πολύ απλές γλώσσες που ονομάζονται **κανονικές γλώσσες (regular languages)**.

2.3.2 Συντακτική ανάλυση

Κάθε γλώσσα προγραμματισμού διέπεται από κανόνες οι οποίοι καθορίζουν την συντακτική δομή ενός σωστά σχηματισμένου προγράμματος. Στην Pascal για παράδειγμα ένα πρόγραμμα αποτελείται από ομάδες. Μία ομάδα αποτελείται από εντολές (statements), μία εντολή αποτελείται από παραστάσεις (expressions) και μία παράσταση αποτελείται από λεκτικά σύμβολα κλπ.

Η λειτουργία του συντακτικού αναλυτή είναι να προσδιορίσει την συντακτική δομή ενός προγράμματος καθώς και να κατασκευάσει μία ενδιάμεση αναπαράσταση του πηγαίου κώδικα η οποία θα περιγράφει την συντακτική δομή του. Η ενδιάμεση αναπαράσταση υλοποιείται συνήθως με την δομή δεδομένων δέντρου (**syntax tree** ή **parse tree**) και η πιο συνηθισμένη εκδοχή αυτού του δέντρου είναι το δέντρο αφηρημένης σύνταξης (**Abstract Syntax Tree – AST**).



Εικόνα 2.4– Παράδειγμα ενός συντακτικού δέντρου μιας γλώσσας προγραμματισμού

2.3.2.1 Context-free γραμματικές και BNF

Η σύνταξη μιας γλώσσας μπορεί να περιγραφεί με την χρήση γραμματικών ανεξάρτητου περιεχομένου (**context-free grammars**) ή σημειογραφίας **BNF** (Backus-Naur Form). Η BNF είναι μία μεταγλώσσα, δηλαδή μία γλώσσα με την οποία περιγράφουμε μία άλλη γλώσσα. Μία γραμματική η οποία εκφράζεται με BNF αποτελείται από:

- Ένα πεπερασμένο σύνολο από τερματικά σύμβολα (ή απλά **τερματικά** - **terminals**). Αυτά είναι τα βασικά σύμβολα τα οποία σχηματίζουν αλφαριθμητικά. Ο όρος λεκτικό σύμβολο είναι συνώνυμο με τον όρο τερματικό όταν μιλάμε για γραμματικές γλωσσών προγραμματισμού. Τυπικά παραδείγματα τερματικών είναι: '>=', 'while', 'if', 'then', ';' κλπ.

- Ένα πεπερασμένο σύνολο από μη-τερματικά σύμβολα (ή απλά **μη-τερματικά** – **nonterminals**). Ένα μη-τερματικό σύμβολο αναπαριστά μία συγκεκριμένη κατηγορία φράσεων σε μία γλώσσα προγραμματισμού. Τυπικά παραδείγματα μη-τερματικών είναι: statement, expression, declaration.

- Ένα **σύμβολο αρχής** (**start symbol**). Ένα σύμβολο αρχής είναι ένα μη-τερματικό σύμβολο το οποίο αναπαριστά την κύρια κατηγορία φράσεων στην γλώσσα προγραμματισμού. Για παράδειγμα ένα σύμβολο αρχής θα μπορούσε να είναι ένα μη-τερματικό με όνομα program.

- Ένα πεπερασμένο σύνολο από **κανόνες παραγωγής** (**production rules**). Οι κανόνες παραγωγής καθορίζουν τον τρόπο με τον οποίο τα τερματικά και τα μη-τερματικά μπορούν να συνδυαστούν για να σχηματίσουν αλφαριθμητικά. Κάθε κανόνας αποτελείται από ένα μη-τερματικό, ακολουθούμενο από τα σύμβολα '::=' και δεξιά από αυτά τα σύμβολα μία ακολουθία μη-τερματικών και τερματικών.

Παραδείγματα κανόνων παραγωγής μίας γλώσσας προγραμματισμού θα μπορούσαν να είναι :

Program	::=	single-Command
Command	::=	single-Command | Command **;** single-Command
single-Command	::=	**begin** Command **end** | V-name **:=** Expression
V-name	::=	Identifier
Identifier	::=	Letter | Identifier Letter | Identifier Digit
Expression	::=	Expression Operator Expression
Operator	::=	**+ | - | * | / | =**

Τα μη-τερματικά είναι:

Program *(σύμβολο αρχής)*
Command
single-Command
V-name
Identifier
Expression
Operator

ενώ τα τερματικά σύμβολα είναι:

begin
end
:=
+
-
*
/
=
;

Μία παραλλαγή της σημειογραφίας BNF είναι η επεκτεταμένη BNF (**Extended BNF - EBNF**). Η σημειογραφία EBNF είναι ένας συνδυασμός της BNF και των κανονικών εκφράσεων η οποία συνδυάζει τα πλεονεκτήματα και των BNF και των κανονικών εκφράσεων.. Ένας κανόνας παραγωγής EBNF είναι της μορφής N::=X, όπου το N είναι ένα μη-τερματικό και το X είναι μία επεκτεταμένη κανονική έκφραση (extended Regular Expression) δηλαδή μία κανονική έκφραση η οποία περιλαμβάνει και τερματικά και μη-τερματικά σύμβολα.

Σε αντίθεση με την απλή BNF, το δεξί μέρος του κανόνα της παραγωγής μπορεί να περιλαμβάνει εκτός από '|', το οποίο δηλώνει εναλλακτική έκφραση, σύμβολα όπως '*' καθώς και '(' και ')'.

Σε αντίθεση με τις απλές κανονικές εκφράσεις, στο δεξί τους μέρος μπορεί να υπάρχουν και μη-τερματικά εκτός από τερματικά.

Κάνοντας χρήση της EBNF μπορούμε να γράψουμε κανόνες παραγωγής όπως ο κανόνας Identifier, που παρουσιάσαμε προηγουμένως, με απλούστερο τρόπο.

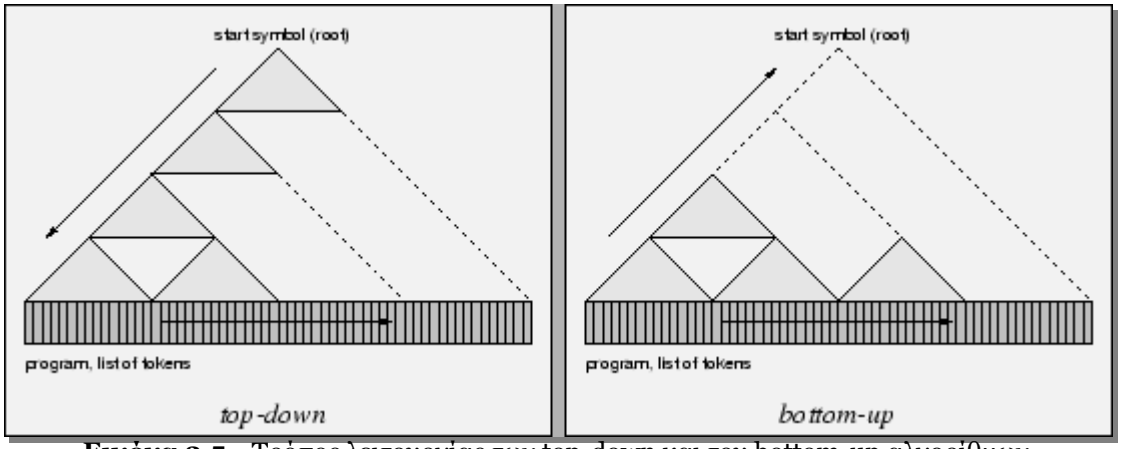
Identifier	::=	Letter | (Letter | Digit)*

2.3.2.2 Αλγόριθμοι συντακτικής ανάλυσης

Οι αλγόριθμοι που εκτελούν την συντακτική ανάλυση ονομάζονται αλγόριθμοι συντακτικής ανάλυσης (**parsing algorithms**). Διάφοροι αλγόριθμοι έχουν αναπτυχθεί αλλά μόνο δύο είναι οι βασικοί. Είναι η

συντακτική ανάλυση από κάτω προς τα πάνω (**bottom-up parsing**) και η συντακτική ανάλυση από πάνω προς τα κάτω (**top-down parsing**). Οι αλγόριθμοι αυτοί χαρακτηρίζονται από τον τρόπο με τον οποίο κατασκευάζουν το συντακτικό δέντρο. Στην ουσία ένας συντακτικός αναλυτής δεν είναι απαραίτητο να κατασκευάσει ρητά ένα συντακτικό δέντρο, αλλά είναι βολικό να εξηγήσουμε τους αλγόριθμους συντακτικής ανάλυσης με όρους κατασκευής ενός συντακτικού δέντρου[4]. Παρ'όλα αυτά ο συντακτικός αναλυτής πρέπει να είναι σε θέση να κατασκευάσει ένα τέτοιο δέντρο γιατί σε αντίθετη περίπτωση η μεταγλώττιση δεν θα είναι εγγυημένα σωστή[5].

Οι αλγόριθμοι top-down είναι πιο δημοφιλείς γεγονός που οφείλεται στο ότι αποδοτικοί συντακτικοί αναλυτές μπορούν να κατασκευαστούν πολύ εύκολα ακόμα και χειροκίνητα (χωρίς αυτόματα εργαλεία ανάπτυξης συντακτικών αναλυτών). Οι αλγόριθμοι bottom-down από την άλλη μπορούν να χειριστούν μεγαλύτερες κατηγορίες γραμματικών έτσι αρκετά εργαλεία αυτόματης ανάπτυξης συντακτικών αναλυτών τείνουν να χρησιμοποιούν αυτούς τους αλγόριθμους. Η εικόνα 2.4 παρουσιάζει τον τρόπο λειτουργίας των δύο αυτών κατηγοριών αλγορίθμων συντακτικής ανάλυσης.



Εικόνα 2.5– Τρόπος λειτουργίας των top-down και του bottom-up αλγορίθμων

2.3.2.2.1 Bottom-up parsing

Ο αλγόριθμος bottom-up εξετάζει τα τερματικά σύμβολα του πηγαίου κώδικα από αριστερά προς τα δεξιά και κατασκευάζει το συντακτικό δέντρο από κάτω προς τα πάνω.

Σαν παράδειγμα ας ορίσουμε μερικούς κανόνες παραγωγής για την γραμματική ενός μικρού κομματιού της αγγλικής γλώσσας.

Sentence ::= Subject Verb Object **.**
Subject ::= **I** | **a** Noun | **the** Noun
Object ::= **me** | **a** Noun | **the** Noun
Noun ::= **cat** | **mat** | **rat**
Verb ::= **like** | **is** | **see** | **sees**

Τα τερματικά σύμβολα είναι αυτά που εμφανίζονται με έντονα γράμματα. Τα μη-τερματικά σύμβολα είναι τα : Sentence (αρχικό σύμβολο), Subject, Object, Noun και Verb.

Να σημειώσουμε ότι η γραμματική αυτή δεν είναι τέλεια και επιτρέπει μερικές προτάσεις που δεν είναι έγκυρες στα Αγγλικά, αλλά είναι μία απλή γραμματική που μας επιτρέπει να μελετήσουμε τους αλγόριθμους συντακτικής ανάλυσης.

Παρακάτω βλέπουμε μερικές προτάσεις που παράγονται από την γραμματική που περιγράψαμε παραπάνω.

the cat sees a rat
I like the mat
the cat likes me

Μερικές μη έγκυρες προτάσεις αυτής της γραμματικής είναι:

me sees the cat
I the mat see

Ας προσπαθήσουμε τώρα να αναλύσουμε συντακτικά την παρακάτω πρόταση με την μέθοδο bottom-up:

the cat sees a rat .

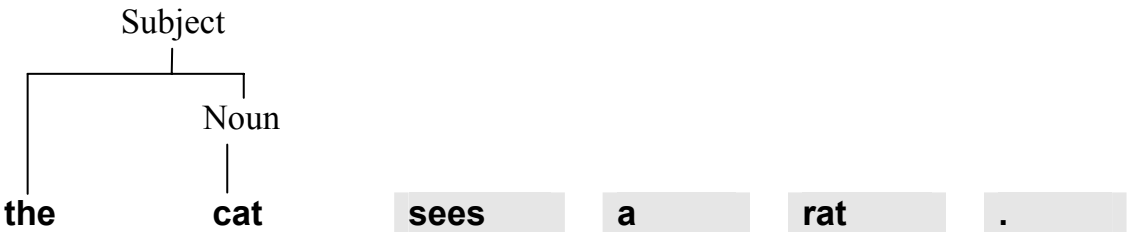
Ο αλγόριθμος bottom-up θα λειτουργήσει ως εξής:

1) Πρώτα θα συναντήσει το τερματικό σύμβολο '**the**'. Ο συντακτικός αναλυτής δεν μπορεί να κάνει τίποτα ακόμα με αυτό το τερματικό σύμβολο και έτσι προχωράει το επόμενο σύμβολο. Συναντάει το τερματικό σύμβολο '**cat**'. Εδώ μπορεί να χρησιμοποιήσει τον κανόνα Noun::=**cat** και να σχηματίσει ένα δέντρο Noun με το τερματικό 'cat' σαν υποδέντρο του:

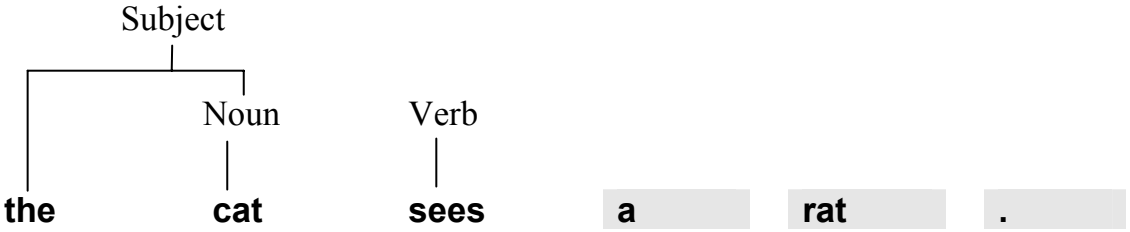
(Τα τερματικά που δεν έχουν εξεταστεί ακόμα από τον συντακτικό αναλυτή, έχουν γκρίζο χρώμα)

2) Τώρα ο συντακτικός αναλυτής μπορεί να χρησιμοποιήσει τον κανόνα Subject::=**the** Noun χρησιμοποιώντας το τερματικό σύμβολο '**the**' που

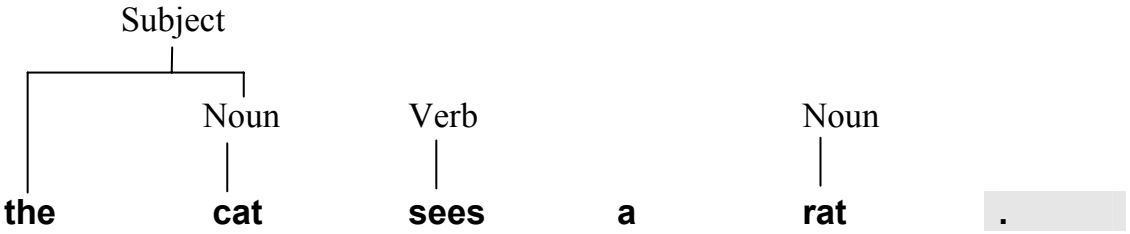
αναγνώρισε προηγουμένως και το δέντρο Noun που δημιούργησε στο προηγούμενο βήμα. Έτσι δημιουργεί ένα δέντρο Subject:



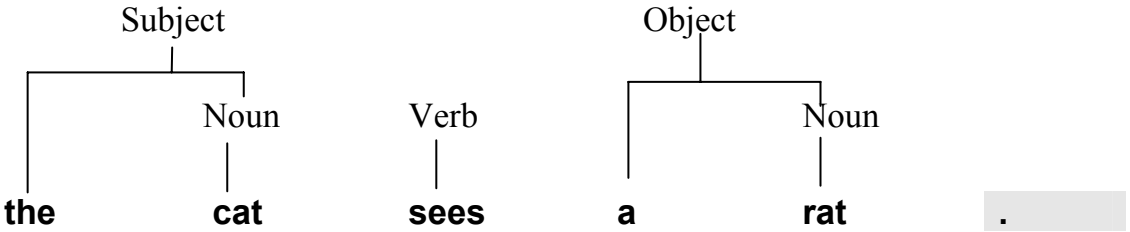
3) Στο σημείο αυτό ο συντακτικός αναλυτής προχωράει το επόμενο τερματικό σύμβολο το οποίο είναι το '**sees**'. Εδώ μπορεί να χρησιμοποιήσει τον κανόνα Verb::=**sees** και να κατασκευάσει ένα δέντρο Verb.



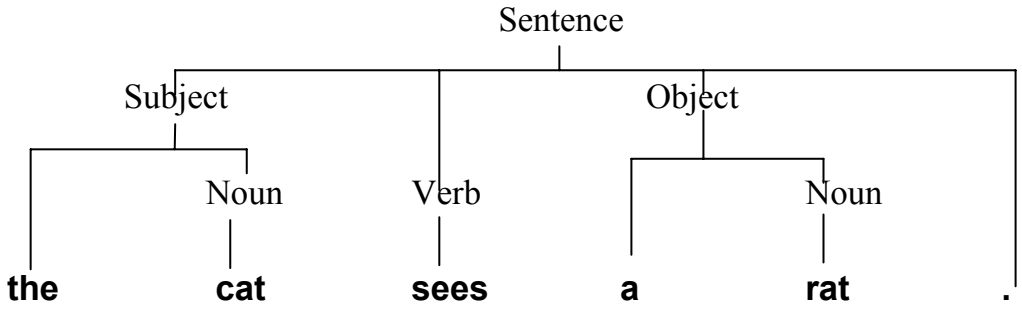
4) Το επόμενο τερματικό σύμβολο είναι το '**a**'. Ο συντακτικός αναλυτής δεν μπορεί να κάνει ακόμα τίποτα με αυτό το σύμβολο, έτσι προχωράει στο επόμενο το οποίο είναι το '**rat**'. Εδώ μπορεί να χρησιμοποιήσει το κανόνα Noun::=**rat** και να κατασκευάσει ένα δέντρο Noun:



5) Τώρα μπορεί να χρησιμοποιήσει το τερματικό σύμβολο που διάβασε στο προηγούμενο βήμα χρησιμοποιώντας τον κανόνα Object::=**a** Noun και να κατασκευάσει ένα δέντρο Object:



6) Το επόμενο και τελευταίο τερματικό σύμβολο που διαβάζει είναι το '**.**'. Με το σύμβολο αυτό και τα δέντρα Subject, Verb και Object χρησιμοποιεί τον κανόνα Sentence::= Subject Verb Object . Κατασκευάζει λοιπόν και ένα δέντρο Sentence:



Ο συντακτικός αναλυτής κατασκεύασε επιτυχώς ένα δέντρο Sentence με τα τερματικά που είχε σαν είσοδο. Με άλλα λόγια ολοκλήρωσε με επιτυχία την συντακτική ανάλυση.

2.3.2.2.2 Top-down parsing

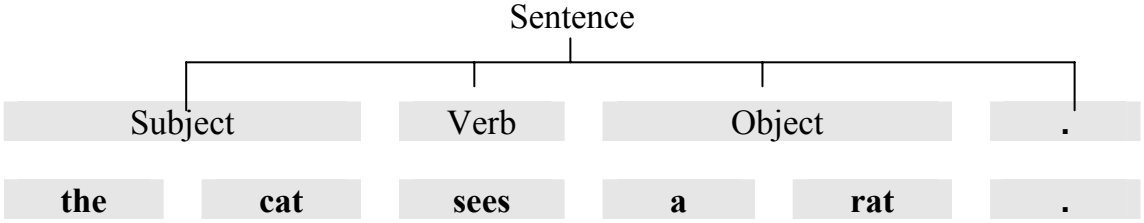
Ο αλγόριθμος αυτός λειτουργεί ως εξής. Ο συντακτικός αναλυτής εξετάζει τα τερματικά σύμβολα από αριστερά προς τα δεξιά και κατασκευάζει το συντακτικό δέντρο από την κορυφή (root node) προς τα κάτω. Για παράδειγμα ας χρησιμοποιήσουμε και πάλι την γραμματική της αγγλικής γλώσσας που δημιουργήσαμε προηγουμένως.

Ας προσπαθήσουμε πάλι να αναλύσουμε συντακτικά την ίδια πρόταση με τον αλγόριθμο top-down αυτή τη φορά.

the cat sees a rat .

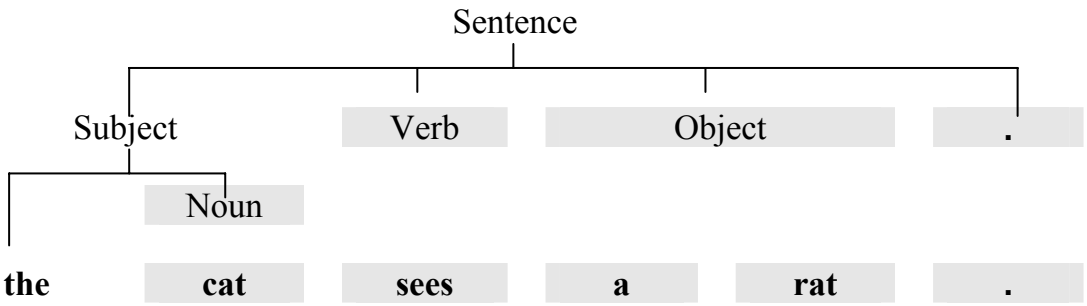
Ο αλγόριθμος ξεκινάει φτιάχνοντας τον ριζικό κόμβο (root node) με όνομα Sentence. Έπειτα προχωράει ως εξής:

1) Ο συντακτικός αναλυτής πρέπει να αποφασίσει ποιον κανόνα παραγωγής πρέπει να χρησιμοποιήσει στον κόμβο Sentence. Με βάση την γραμματική δεν έχει πολλές επιλογές. Έτσι εφαρμόζει τον κανόνα Sentence::= Subject Verb Object.

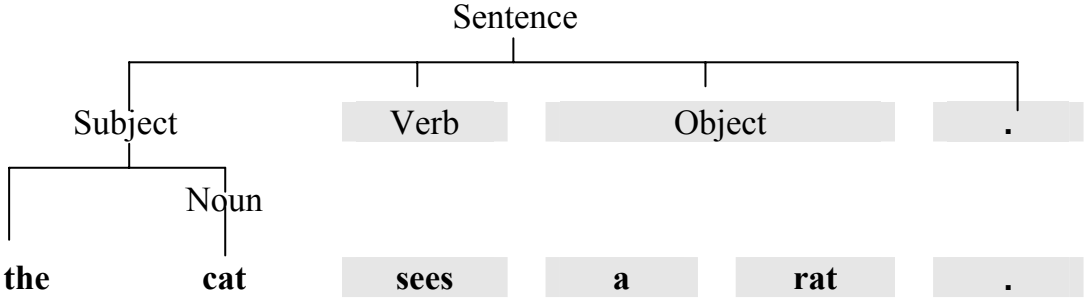


Αυτό το βήμα δημιούργησε τέσσερα ανολοκλήρωτα δέντρα. Τα ανολοκλήρωτα δέντρα και τα τερματικά σύμβολα που δεν έχουν αναλυθεί από τον συντακτικό αναλυτή θα έχουν γκρίζο χρώμα.

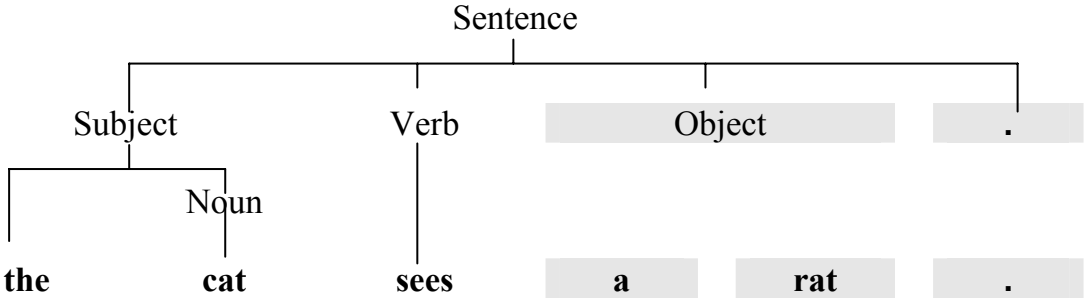
2) Τώρα ο συντακτικός αναλυτής θα ασχοληθεί με το πρώτο ανολοκλήρωτο δέντρο από τα αριστερά του. Συναντάει το ανολοκλήρωτο δέντρο Subject. Πρέπει να επιλέξει ποιον κανόνα να εφαρμόσει. Είναι φανερό ότι θα επιλέξει τον κανόνα Subject. Ο κανόνας Subject όμως έχει τρεις εναλλακτικές. Διαβάζει το πρώτο τερματικό σύμβολο το οποίο είναι το '**the**'. Τώρα πλέον είναι σαφές ότι θα χρησιμοποιήσει τον κανόνα Subject::= **the** Noun. Έτσι συνδέει το τερματικό '**the**' με το δέντρο Subject και δημιουργεί ένα ανολοκλήρωτο υποδέντρο Noun κάτω από τον δέντρο Subject.



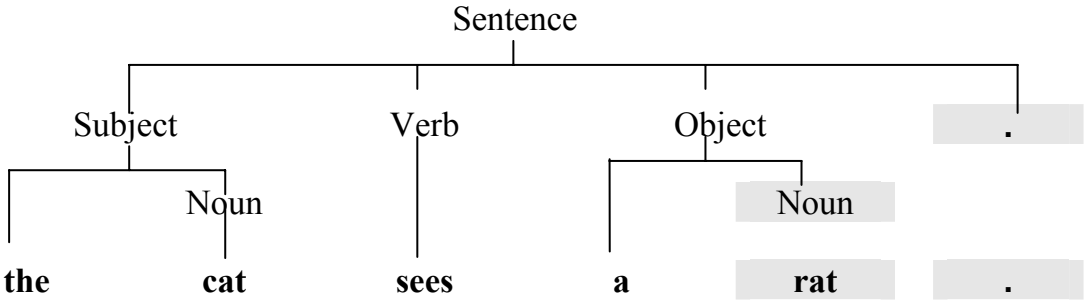
3) Το ανολοκλήρωτο δέντρο που είναι πιο αριστερά είναι τώρα το Noun. Ο συντακτικός αναλυτής διαβάζει το επόμενο τερματικό σύμβολο και βλέπει ότι είναι το '**cat**'. Έτσι χρησιμοποιεί τον κανόνα Noun::=**cat** και ενώνει το τερματικό '**cat**' με το δέντρο Noun.



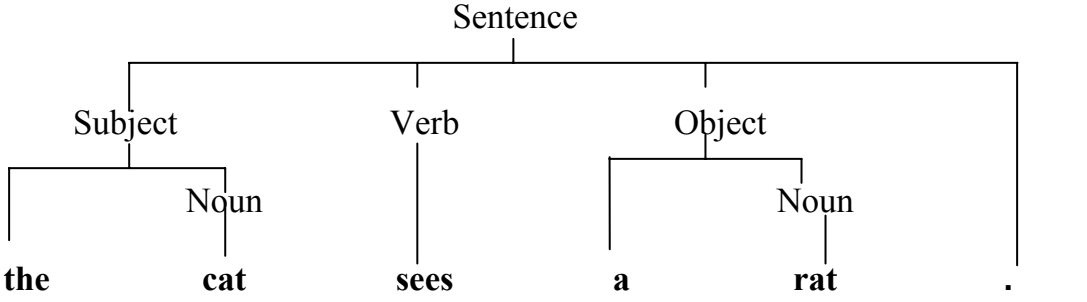
4) Συνεχίζοντας ο συντακτικός αναλυτής, βρίσκει το επόμενο ανολοκλήρωτο δέντρο το οποίο είναι το δέντρο Verb. Διαβάζει το επόμενο τερματικό το οποίο είναι το '**sees**'. Ο μόνος κανόνας παραγωγής που μπορεί να χρησιμοποιήσει είναι ο κανόνας Verb::=**sees**. Έτσι ενώνει το τερματικό '**sees**' με το δέντρο Verb.



5) Το επόμενο ανολοκλήρωτο δέντρο που συναντάει είναι το δέντρο Object. Πολύ απλά χρησιμοποιεί τον κανόνα παραγωγής Object::=**a** Noun. Έτσι ενώνει το τερματικό '**a**' με το δέντρο Object και δημιουργεί ένα νέο ανολοκλήρωτο υποδέντρο κάτω από το δέντρο Object.



6) Το πρώτο ανολοκλήρωτο δέντρο από αριστερά είναι τώρα το δέντρο Noun. Ο συντακτικός αναλυτής διαβάζει το επόμενο τερματικό το οποίο είναι το '**rat**' και ψάχνει να βρει έναν κανόνα παραγωγής για να συνδέσει το δέντρο Noun με το τερματικό '**rat**'. Ο κανόνας αυτός υπάρχει και είναι ο Noun::=**rat**. Αν δεν υπήρχε η συντακτική ανάλυση θα είχε αποτύχει. Έτσι, συνδέει το τερματικό 'rat' με το δέντρο Noun. Το μόνο που έμεινε για την ολοκλήρωση της συντακτικής ανάλυσης είναι να συνδέσει το τερματικό '**.**'. Έχοντας τα ολοκληρωμένα δέντρα Subject, Verb, Object ο συντακτικός αναλυτής κάνει χρήση του κανόνα παραγωγής Sentence::= Subject Verb Object **.** και ενώνει το τερματικό '**.**' με το δέντρο Sentence.



Ο συντακτικός αναλυτής ολοκλήρωσε με επιτυχία την συντακτική ανάλυση χρησιμοποιώντας τον αλγόριθμο top-down.

Ο πιο γνωστός αλγόριθμος της κατηγορίας top-down είναι ο αλγόριθμος αναδρομικής κατάβασης (**recursive descent**). Ο αλγόριθμος αυτός υλοποιείται σχετικά εύκολα. Ο αλγόριθμος αποτελείται από μεθόδους parseN. Μία μέθοδος parseN για κάθε μη-τερματικό σύμβολο όπου N το όνομα του μη-τερματικού. Σε επόμενη ενότητα θα δούμε μία υλοποίηση ενός αλγορίθμου αναδρομικής κατάβασης.

2.3.3 Σημασιολογική ανάλυση

Όπως αναφέραμε προηγούμενος μία από τις δουλειές του συντακτικού αναλυτή είναι η δημιουργία συντακτικών δέντρων. Αν ο συντακτικός αναλυτής κατασκευάσει επιτυχώς το συντακτικό δέντρο αυτό σημαίνει ότι το πρόγραμμα έχει έγκυρη σύνταξη. Ωστόσο, κάθε συντακτικό δέντρο που παράγεται δεν είναι απαραίτητο να είναι και έγκυρο πρόγραμμα της γλώσσας προγραμματισμού. Για να εξηγήσουμε αυτή την περίπτωση ας δούμε δύο προγράμματα γραμμένα σε C.

```
#include <stdio.h>

int main()
{
   int var;
   var = 5;
   printf("%d",var);
   return 0;
}
```

Πρόγραμμα Α

```
#include <stdio.h>

int main()
{
   var = 5;
   printf("%d",var);
   return 0;
}
```

Πρόγραμμα Β

Ο συντακτικός αναλυτής θα παράγει κανονικά τα συντακτικά δέντρα και των δύο αυτών προγραμμάτων. Ωστόσο, μόνο το πρώτο είναι έγκυρο πρόγραμμα C. Το δεύτερο δεν είναι έγκυρο γιατί η μεταβλητή var δεν έχει δηλωθεί. Έτσι το δεύτερο πρόγραμμα θα μεταγλωττιστεί κανονικά ενώ το πρώτο θα εμφανίσει μήνυμα λάθους κατά την μεταγλώττιση. Ένας τυπικός μεταγλωττιστής θα εμφανίσει ένα μήνυμα λάθους σαν το ακόλουθο.

```
Error: `var' undeclared identifier in function main()
```

Μία άλλη κατηγορία λαθών είναι αυτή που παρουσιάζει ο παρακάτω κώδικας.

```
#include <stdio.h>

int main()
{
int first;
int second[3] = {4,6,8};
first = second;
printf("%d", first);
}
```

Το πρόβλημα είναι προφανώς η ασυμβατότητα των τύπων κατά την εκχώρηση:

```
first = second;
```

Ωστόσο, η C είναι γενικά αρκετά ανεκτική όσον αφορά τα λεγόμενα λάθη τύπων (**type errors**). Για παράδειγμα οι παρακάτω εκχωρήσεις είναι απόλυτα σωστές και δεν θα εμφανίσει λάθη ο μεταγλωττιστής.

```
int p = 4.3;
float x = 2;
int c = 'a';
int k = NULL;
```

Η C για να δεχτεί σαν σωστές εκχωρήσεις όπως τις παραπάνω, πραγματοποιεί μετατροπές τύπων, όπου αυτό είναι εφικτό. Για παράδειγμα στην εκχώρηση: int c = 'a'; η C θα μετατρέψει τον χαρακτήρα 'a' στην αντίστοιχη της τιμή στον πίνακα ASCII και αυτή την (ακέραια) τιμή θα χρησιμοποιήσει στην εκχώρηση.

Ένα πρόγραμμα για να είναι σωστό σημασιολογικά όλες οι μεταβλητές, οι συναρτήσεις, οι κατηγορίες (classes) κλπ πρέπει να είναι σωστά δηλωμένες και οι εκφράσεις και οι μεταβλητές πρέπει να χρησιμοποιούνται κάνοντας χρήση σωστών τύπων. Έτσι λοιπόν ο μεταγλωττιστής πρέπει να πραγματοποιήσει έλεγχο και για όλους αυτούς τους τύπους λαθών πριν προχωρήσει στην παραγωγή κώδικα μηχανής. Η διαδικασία αυτού του ελέγχου ονομάζεται σημασιολογική ανάλυση (**semantic analysis**) και η μονάδα που πραγματοποιεί αυτή την ανάλυση ονομάζεται σημασιολογικός αναλυτής (**semantic analyzer**). Η δουλειά της σημασιολογικής ανάλυσης είναι να

ελέγξει αν ο πηγαίος κώδικας συμμορφώνεται με τους σημασιολογικούς περιορισμούς της γλώσσας.

Οι περιορισμοί αυτοί είναι:

- Κανόνες εμβέλειας (**scope rules**): Οι κανόνες αυτοί καθορίζουν τις δηλώσεις (declaration) και τις εμφανίσεις των αναγνωριστών (identifiers).

- Κανόνες τύπων (**type rules**): Οι κανόνες αυτοί μας επιτρέπουν να βγάζουμε συμπεράσματα σχετικά με τον τύπο μιας έκφρασης (expression) και να κρίνουμε αν μία έκφραση έχει σωστό τύπο.

Η φάση της σημασιολογικής ανάλυσης αποτελείται από δύο υποφάσεις:

- Αναγνώριση (**identification**): Είναι η φάση στην οποία εφαρμόζονται οι κανόνες εμβέλειας σε κάθε αναγνωριστή που εμφανίζεται στον πηγαίο κώδικα καθώς και στην δήλωσή του (αν αυτή υπάρχει).

- Έλεγχος τύπων (**type checking**): Είναι η φάση στην οποία εφαρμόζονται οι κανόνες τύπων σε κάθε έκφραση που εμφανίζεται στον πηγαίο κώδικα.

2.3.3.1 Αναγνώριση

Η πρώτη φάση του σημασιολογικού αναλυτή είναι να ελέγξει αν κάθε αναγνωριστής έχει δηλωθεί κατάλληλα. Αν δεν έχει δηλωθεί τότε το πρόγραμμα δεν είναι σωστό και πρέπει να αναφέρει το λάθος. Η φάση αυτή ονομάζεται αναγνώριση (identification). Εδώ να σημειώσουμε βέβαια ότι υπάρχουν και γλώσσες προγραμματισμού στις οποίες ένας αναγνωριστής δεν είναι απαραίτητο να έχει δηλωθεί. Μία τέτοια γλώσσα είναι η γλώσσα PHP. Στις περιπτώσεις αυτές, οι αναγνωριστές δηλώνονται αυτόματα την πρώτη φορά που τους ανατίθεται κάποια τιμή.

Η μέθοδος που χρησιμοποιείται για τον έλεγχο της αναγνώρισης είναι η χρήση ενός πίνακα αναγνώρισης (identification table). Ο πίνακας αυτός χρησιμοποιείται για την συσχέτιση των αναγνωριστών με τις ιδιότητές τους. Όταν ο σημασιολογικός αναλυτής συναντήσει έναν αναγνωριστή για πρώτη φορά ψάχνει να βρει την δήλωσή του. Αν η δήλωση αυτή υπάρχει και την εντοπίσει ο σημασιολογικός αναλυτής, τότε προσθέτει μία εγγραφή στον πίνακα αναγνώρισης με τρία πεδία. Το όνομα του αναγνωριστή, το επίπεδο της εμβέλειάς του (**scope level**) και τις ιδιότητες του.

Η εμβέλεια ενός αναγνωριστή είναι το κομμάτι του κώδικα στο οποίο υπάρχει δυνατότητα χρήσης του αναγνωριστή αυτού. Η εμβέλεια ενός αναγνωριστή εξαρτάται από το σημείο στο οποίο ο αναγνωριστής δηλώθηκε.

Αφού πληροφορηθεί για την εμβέλεια των αναγνωριστών, ο σημασιολογικός αναλυτής στην συνέχεια ελέγχει ότι κάθε αναγνωριστής έχει δηλωθεί μόνο μία φορά. Δηλαδή, αν είναι ο μοναδικός αναγνωριστής με αυτό το όνομα στο συγκεκριμένο επίπεδο εμβέλειας. Αν ένας αναγνωριστής έχει

δηλωθεί περισσότερες από μία φορές στο ίδιο επίπεδο εμβέλειας, τότε αναφέρει το λάθος.

Σε ορισμένες απλές γλώσσες **scripting** όπως είναι οι γλώσσες των κελυφών δεν υπάρχουν επίπεδα εμβέλειας. Όλες οι μεταβλητές, ασχέτως από το σημείο που δηλώθηκαν, θεωρούνται καθολικές (**global**). Μπορούν να χρησιμοποιηθούν παντού μέσα στον κώδικα. Πράγμα που σημαίνει ότι δεν υπάρχει δυνατότητα να ξαναχρησιμοποιηθεί μία μεταβλητή με το ίδιο όνομα αλλού μέσα στο πρόγραμμα.

2.3.3.2 Έλεγχος τύπων

Η δεύτερη φάση του σημασιολογικού αναλυτή είναι να επιβεβαιώσει ότι ο πηγαίος κώδικας δεν περιέχει λάθη σε σχέση με τους τύπους των αναγνωριστών και των εκφράσεων. Αυτή την διαδικασία την ονομάζουμε έλεγχο τύπων (type checking). Κατά την διαδικασία αυτή, ελέγχονται:

- Τα κυριολεκτικά (literals): Ο τύπος των κυριολεκτικών γίνεται αμέσως γνωστός.

- Οι αναγνωριστές: Ο τύπος ενός αναγνωριστή γίνεται γνωστός από την δήλωσή του.

- Μοναδιαίοι τελεστές (unary operators): Ας υποθέσουμε ότι έχουμε μία έκφραση "Τ Ε" , όπου Τ είναι ο μοναδιαίος τελεστής του οποίου ο τύπος είναι της μορφής $T_1 \rightarrow T_2$. Που σημαίνει ότι ο τελεστής πρέπει να εφαρμοστεί σε ένα τελεστέο τύπου T_1 και το αποτέλεσμα θα είναι τύπου T_2. Το Ε είναι η έκφραση. Στην περίπτωση αυτή ο σημασιολογικός αναλυτής επιβεβαιώνει ότι η έκφραση Ε είναι τύπου T_1 και ότι το αποτέλεσμα της έκφρασης "Τ Ε" είναι τύπου T_2. Σε διαφορετική περίπτωση αναφέρει λάθος.

- Δυαδικοί τελεστές (binary operators): Ας υποθέσουμε ότι έχουμε μία έκφραση "E_1 Τ E_2" , όπου Τ είναι ο δυαδικός τελεστής του οποίου ο τύπος είναι της μορφής $T_1 \times T_2 \rightarrow T_3$. Που σημαίνει ότι ο τελεστής πρέπει να εφαρμοστεί σε ένα τελεστέο τύπου T_2 και T_2 το αποτέλεσμα θα είναι τύπου T_3. Στην περίπτωση αυτή ο σημασιολογικός αναλυτής επιβεβαιώνει ότι η έκφραση E_1 είναι τύπου T_1, η έκφραση E_2 είναι τύπου T_2 και ότι το αποτέλεσμα της έκφρασης "E_1 Τ E_2" είναι τύπου T_3. Σε διαφορετική περίπτωση αναφέρει λάθος.

2.3.4 Παραγωγή κώδικα μηχανής

Οι τρεις υποφάσεις που αναφέραμε παραπάνω (λεκτική ανάλυση, συντακτική ανάλυση και σημασιολογική ανάλυση) αφορούν τη φάση Ανάλυσης του μεταγλωττιστή. Στην συνέχεια θα ασχοληθούμε με την φάση της Σύνθεσης. Η φάση της Ανάλυσης του πηγαίου προγράμματος δεν

ασχολείται καθόλου με την αρχιτεκτονική του υπολογιστή στον οποίο γίνεται. Η Σύνθεση και συγκεκριμένα η φάση της παραγωγής κώδικα μηχανής (**code generation**) από την άλλη λαμβάνει υπ' όψη της την αρχιτεκτονική του υπολογιστή. Αυτό γίνεται διότι η γλώσσα προορισμού που θα παράγει, θα πρέπει να είναι η γλώσσα μηχανής που καταλαβαίνει το συγκεκριμένο μηχάνημα.

Το γεγονός ότι κατά τη φάση αυτή λαμβάνεται υπ' όψη η αρχιτεκτονική του υπολογιστή και το γεγονός ότι υπάρχει μεγάλη ποικιλία αρχιτεκτονικών δημιουργεί αρκετά προβλήματα. Τα κύρια προβλήματα της φάσης της παραγωγής κώδικα μηχανής είναι[6]:

- **Επιλογή κώδικα (code selection)**: Το πρόβλημα αυτό αφορά την απόφαση του πρέπει να ληφθεί σε σχέση με το ποια ακολουθία εντολών μηχανής θα αποτελέσει τον κώδικα για κάθε έκφραση του πηγαίου προγράμματος. Για τον λόγο αυτό χρησιμοποιούμε πρότυπα κώδικα (code templates). Ένα πρότυπο κώδικα είναι ένας γενικός κανόνας ο οποίος καθορίζει τον κώδικα μηχανής που θα χρησιμοποιηθεί για κάθε τύπο έκφρασης (εκχώρηση, κλήση συνάρτησης κλπ). Στην πράξη τα πρότυπα κώδικα σε ορισμένες περιπτώσεις μπορεί να γίνουν πολύ πολύπλοκα.

- **Δέσμευση μνήμης (storage allocation)**: Το πρόβλημα αυτό αφορά την επιλογή της διεύθυνσης μνήμης της κάθε μεταβλητής του πηγαίου προγράμματος. Η επιλογή μπορεί να γίνει είτε θέτοντας μία ακριβής διεύθυνση για τις καθολικές μεταβλητές (static storage allocation), είτε μπορεί να τεθεί μία σχετική διεύθυνση για κάθε τοπική μεταβλητή (stack storage allocation).

- **Δέσμευση καταχωρητών (register allocation)**: Σε περίπτωση που ο υπολογιστής προορισμού διαθέτει καταχωρητές, πρέπει να χρησιμοποιηθούν για την διατήρηση των ενδιάμεσων αποτελεσμάτων κατά τον υπολογισμό μιας έκφρασης. Αρκετές επιπλοκές μπορεί να προκύψουν στην πράξη όπως να μην υπάρχουν αρκετοί καταχωρητές για τον υπολογισμό μιας πολύπλοκης έκφρασης ή συγκεκριμένοι καταχωρητές να έχουν δεσμευτεί για συγκεκριμένες λειτουργίες.

2.3.5 Βελτιστοποίηση του κώδικα μηχανής

Κατά την φάση της παραγωγής κώδικα αναφέραμε ότι ο κώδικας που θα παραχθεί από τον μεταγλωττιστή θα πρέπει να κάνει αποτελεσματική χρήση των πόρων του υπολογιστή προορισμού. Μία ποικιλία αλγορίθμων έχουν αναπτυχθεί, με τους οποίους βελτιστοποιείται ο κώδικας (**code optimization**) που παράγεται από την προηγούμενη φάση. Ωστόσο, η βελτιστοποίηση αυτή δεν είναι ποτέ ισάξια ενός κώδικα που γράφτηκε εξ' αρχής σε assembly (συγκεκριμένα είναι τουλάχιστον δύο φορές πιο αργός). Έτσι λοιπόν ακόμα και με τη βελτιστοποίηση δεν παράγεται κώδικας πραγματικά

βελτιστοποιημένος. Γενικά οι καλύτερες βελτιστοποιήσεις είναι εκείνες οι οποίες αποφέρουν το καλύτερο κέρδος με την λιγότερη προσπάθεια.

Υπάρχουν κάποια κριτήρια τα οποία πρέπει να ληφθούν υπ' όψη κατά την διαδικασία βελτιστοποίησης του κώδικα[7].

- Ο κώδικας μετά την βελτιστοποίηση πρέπει να διατηρήσει το νόημα του. Αυτό σημαίνει ότι δεν πρέπει να αλλάξει την έξοδο του προγράμματος για μία δεδομένη είσοδο αλλά ούτε και να δημιουργήσει σφάλματα όπως διαίρεση με το μηδέν η οποία δεν υπήρχε στο πηγαίο κώδικα.

- Η βελτιστοποίηση πρέπει κατά μέσο όρο να επιταχύνει την εκτέλεση του προγράμματος κατά ένα μετρίσιμο ποσό.

Επειδή η βελτιστοποίηση είναι μία διαδικασία η οποία απαιτεί αρκετό χρόνο, θα πρέπει να είναι πολλές φορές δυνατό να ρυθμίζουμε τον μεταγλωττιστή κατάλληλα ώστε να πραγματοποιεί βελτιστοποίηση μόνο όταν ο το επιθυμεί ο προγραμματιστής. Για παράδειγμα, δεν υπάρχει νόημα να πραγματοποιείται βελτιστοποίηση σε ένα πρόγραμμα το οποίο θα εκτελεστεί μία ή δύο φορές συνολικά. Τέτοιες περιπτώσεις είναι για παράδειγμα τα προγράμματα που γράφουν φοιτητές και κατά την περίοδο εκπαίδευσής τους.

Μερικές από τις πιο συνηθισμένες μεθόδους βελτιστοποίησης είναι[8]:

- Αναδίπλωση σταθερών (**constant folding**): Εάν μία έκφραση βασίζεται μόνο σε γνωστές τιμές (πχ σταθερές), μπορεί να υπολογιστεί κατά την διάρκεια της μεταγλώττισης (compile time) και όχι κατά την διάρκεια της εκτέλεσης (run time). Για παράδειγμα ο κώδικας :

```
limit = 10;
index = limit – 1;
```

μπορεί να αντικατασταθεί από τον κώδικα:

```
limit = 10;
index = 9;
```

- Εξάλειψη κοινών εκφράσεων (**common sub-expression elimination**): Σε περίπτωση που η ίδια έκφραση χρησιμοποιηθεί σε περισσότερα από ένα σημεία στον πηγαίο κώδικα, τότε είναι δυνατόν να υποθηκεύσουμε το αποτέλεσμα της πρώτης έκφρασης και να το χρησιμοποιήσουμε στην θέση της ίδιας έκφρασης όταν ξανασυναντηθεί σε άλλο σημείο. Για παράδειγμα θα μπορούσε να υπάρχει ένας υπολογισμός σαν τον ακόλουθο:

```
test = (x-y) * (x-y+z);
```

Η έκφραση x-y μπορεί να υπολογιστεί μία μόνο φορά και να χρησιμοποιηθεί η τιμής της, την επόμενη φορά που ξανασυναντάμε την ίδια έκφραση.

- Μετακίνηση κώδικα (**code movement**): Σε περίπτωση που μία έκφραση η οποία βρίσκεται μέσα σε ένα βρόγχο (loop) έχει την ίδια τιμή σε κάθε επανάληψη του βρόγχου, τότε η έκφραση μπορεί να βγει έξω από τον βρόγχο, να υπολογιστεί η τιμή της μία φορά και να χρησιμοποιηθεί η τιμή αυτή μέσα στο βρόγχο. Για παράδειγμα ο κώδικας:

```
int v[10];

void f()
{
   int i,a,b;
   a = 5;
   b = 8;
   for(i=0;i<10;i++)
      v[i]=a*b;
}
```

μπορεί να αντικατασταθεί από τον κώδικα:

```
int v[10];

void f()
{
   int i,a,b,t1;
   a = 5;
   b = 8;
   t1 = a * b;
   for(i=0;i<10;i++)
      v[i]=t1;
}
```

2.4 ■ Κατηγορίες γραμματικών

Όπως αναφέραμε και σε προηγούμενη ενότητα, μία γραμματική ορίζεται από τέσσερα στοιχεία. Ένα πεπερασμένο σύνολο τερματικών στοιχείων, ένα πεπερασμένο σύνολο μη-τερματικών στοιχείων, ένα πεπερασμένο σύνολο κανόνων παραγωγής και ένα μη-τερματικό στοιχείο που ονομάζεται σύμβολο αρχής.

Μία γραμματική συμβολίζεται με το τετράπτυχο (V_T, V_N, P, S)

όπου

V_T, είναι το σύνολο των τερματικών
V_N, είναι το σύνολο των μη-τερματικών
P, είναι το σύνολο των κανόνων παραγωγής
S, είναι το σύμβολο αρχής

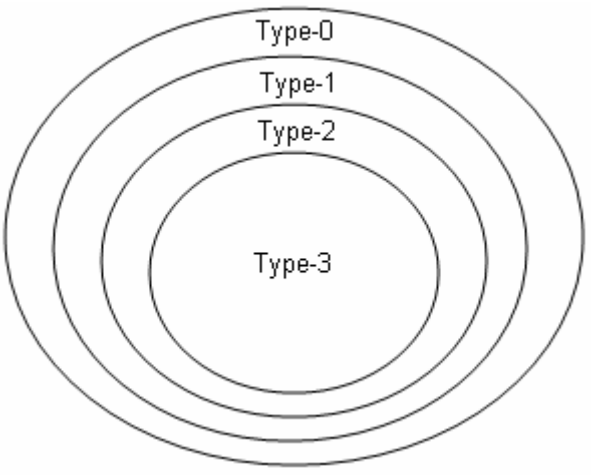
Να σημειώσουμε οτι το V_T και το V_N δεν έχουν κανέναν κοινό σύμβολο μεταξύ τους (δηλαδή $V_T \cap V_N = \varnothing$) και το V ορίζεται σαν $V_T \cup V_N$. Ένας κανόνας παραγωγής αποτελείται από ένα ζεύγος (α,β) όπου α είναι η αριστερή πλευρά του κανόνα, το β είναι η δεξιά πλευρά του κανόνα και ο κανόνας συμβολίζεται ως :

$$\alpha \rightarrow \beta$$

Μία γραμματική χρησιμοποιείται για να παράγει ακολουθίες συμβόλων οι οποίες κατασκευάζουν τα αλφαριθμητικά της γλώσσας, ξεκινώντας από το σύμβολο αρχής και διαδοχικά αντικαθιστά τα μη-τερματικά χρησιμοποιώντας ένα κανόνα παραγωγής σε κάθε στάδιο. Η διαδικασία τελειώνει όταν παραχθεί ένα αλφαριθμητικό μόνο από τερματικά σύμβολα (κανένα μη-τερματικό). Οποιοδήποτε αλφαριθμητικό παράγεται με αυτή τη μέθοδο λέγεται ότι ανήκει στην γλώσσα η οποία ορίστηκε με την συγκεκριμένη γραμματική.

Να σημειώσουμε ότι μία γλώσσα δεν ορίζεται αποκλειστικά από μία μόνο γραμματική. Δύο γραμματικές οι οποίες παράγουν την ίδια γλώσσα τις ονομάζουμε **ισοδύναμες** και, όπως θα δούμε παρακάτω, είναι πολλές φορές χρήσιμο ακόμα και απαραίτητο (από την πλευρά της κατασκευής μεταγλωττιστών) να αντικαθιστούμε μία γραμματική με μία άλλη ισοδύναμή της.

Ο καθηγητής Noam Chomsky στην εργασία του "Language and Mind" (Harcourt Brace Jovanovich, San Diego) το 1968 όρισε μία σημαντική ιεραρχία γραμματικών. Η ιεραρχία αυτή ονομάζεται **ιεραρχία Chomsky**. Η ιεραρχία αυτή καθορίζει τέσσερις κατηγορίες γραμματικών. Η πρώτη κατηγορία ονομάζεται **type-0**, η δεύτερη **type-1**, η τρίτη **type-2** και η τέταρτη **type-3**. Η θέση μιας γραμματικής στην ιεραρχία αυτή καθορίζεται από την μορφή που έχουν οι κανόνες παραγωγής της[9].



Εικόνα 2.6– Η ιεραρχία γραμματικών Chomsky

Η πιο γενική κατηγορία γραμματικής είναι η γραμματική type-0, οι κανόνες παραγωγής της οποίας δεν είναι καθόλου περιοριστικοί. Οι κατηγορίες type 1,2 και 3 ορίζουν όλο και μεγαλύτερους περιορισμούς στους κανόνες παραγωγής της γραμματικής.

Η κατηγορία type-0 είναι ο πιο γενικός τύπος γραμματικής και μπορεί κάποιος να την συναντήσει με την ονομασία αναδρομική απαριθμίσημη γραμματική (**Recursive Enumerable Grammar**). Οι γραμματικές type-0 είναι ισοδύναμες με τις μηχανές Turing με την έννοια ότι με μία δεδομένη γραμματική type-0, υπάρχει μία μηχανή Turing η οποία δέχεται όλα τα αλφαριθμητικά αυτής της γραμματικής. Όταν δεχτεί μία έγκυρη είσοδο η μηχανή Turing θα σταματήσει σε μία κατάσταση απάντησης.

Στην επόμενη κατηγορία γραμματικών εισέρχεται ο εξής περιορισμός. Για κάθε κανόνα παραγωγής της μορφής $\alpha \rightarrow \beta$, το μήκος του αλφαριθμητικού α πρέπει να είναι μικρότερο ή ίσο με το μέγεθος του αλφαριθμητικού β, εννοώντας τον αριθμό των συμβόλων που περιέχουν, δηλαδή:

$$| \alpha | \leq | \beta |$$

Η κατηγορία των γραμματικών, των οποίων όλοι οι κανόνες παραγωγής είναι αυτής της μορφής, είναι γνωστή σαν type-1 ή ευαίσθητου περιεχομένου (**Context Sensitive**). Για παράδειγμα μία γραμματική η οποία περιέχει έναν κανόνα παραγωγής όπως: $Q \rightarrow \varepsilon$ (όπου ε το κενό αλφαριθμητικό) δεν ανήκει στην κατηγορία type-1.

Η επόμενη κατηγορία της ιεραρχίας Chomsky είναι η κατηγορία type-2 ή ανεξάρτητου περιεχομένου γραμματικές (**Context Free Grammar - CFG**), η οποία ορίζει τον εξής περιορισμό: στην αριστερή πλευρά ενός κανόνα παραγωγής πρέπει να υπάρχει μόνο ένα μη-τερματικό στοιχείο. Σχεδόν ολόκληρη η θεωρία των μεταγλωττιστών βασίζεται στην κατηγορία type-2 (αλλά και στην κατηγορία type-3). Επιπροσθέτως, η κατηγορία γραμματικών type-2 είναι πολύ δημοφιλής για να ορίζει την γραμματική φυσικών γλωσσών (natural

languages). Στις γραμματικές ελευθέρου περιεχομένου είναι δυνατόν να ορίσουμε ένα μη-τερματικό σύμβολο (ακόμα και το αρχικό σύμβολο) ίσο με το κενό αλφαριθμητικό (αν και αυτό δεν είναι δεκτό στις γραμματικές ευαίσθητου περιεχομένου) ώστε να μπορέσουμε να συμπεριλαμβάνουμε το κενό αλφαριθμητικό στην γλώσσα. Δηλαδή είναι δυνατόν:

$S \rightarrow \varepsilon$

Επιτρέποντας το αυτό, δεν αυξάνεται η δύναμη των γραμματικών ελευθέρου περιεχομένου, αλλά είναι πολλές φορές πολύ βολικό.

Η τελευταία κατηγορία στην ιεραρχία είναι η κατηγορία type-3 ή συνηθισμένη γραμματική (**Regular Grammar**). Πριν αναφέρουμε τον περιορισμό που ορίζει η κατηγορία αυτή πρέπει να μελετήσουμε τον όρο αριστερή και δεξιά γραμμική γραμματική (**left linear grammar, right linear grammar**).

Στην δεξιά γραμμική γραμματική, κάθε κανόνας παραγωγής είναι μιας από τις δύο μορφές που ακολουθούν:

$A \rightarrow a$

ή

$A \rightarrow bC$

όπου A και C είναι μοναδικά μη-τερματικά σύμβολα και το b είναι ένα μοναδικό τερματικό σύμβολο. Για παράδειγμα η γραμματική με τους ακόλουθους κανόνες παραγωγής:

$S \rightarrow xS$
$S \rightarrow yB$
$S \rightarrow x$
$S \rightarrow y$
$B \rightarrow yB$
$B \rightarrow y$

είναι δεξιά γραμμική γραμματική και αν θέλαμε να συμπεριλάβουμε στην γλώσσα το κενό αλφαριθμητικό θα πρέπει να προσθέσουμε και τον κανόνα:

$S \rightarrow \varepsilon$

Παρ'όλα αυτά η ισοδύναμη γραμματική:

$S \rightarrow XY$
$X \rightarrow xX$
$X \rightarrow \varepsilon$
$Y \rightarrow yY$
$Y \rightarrow \varepsilon$

δεν είναι δεξιά γραμμική γραμματική, επειδή οι κανόνες 1,3,5 δεν είναι της απαραίτητης μορφής. Στην δεξιά γραμμική γραμματική δεν επιτρέπεται ένα μη-τερματικό να παράγει το κενό αλφαριθμητικό εκτός από το σύμβολο αρχής.

Κατ' αντίστοιχο τρόπο στην αριστερή γραμμική γραμματική οι κανόνες παραγωγής πρέπει να είναι της μορφής:

$$A \rightarrow a$$

ή

$$A \rightarrow Bc$$

Η γραμματική η οποία ακολουθεί την μορφή της δεξιάς γραμμικής γραμματικής ή της αριστερής γραμμικής γραμματική ονομάζεται συνηθισμένη (Regular Grammar). Προσοχή πρέπει να δοθεί στην περίπτωση που μία γραμματική περιέχει μερικούς κανόνες της μορφής της δεξιάς γραμμικής αλλά ταυτόχρονα και μερικούς κανόνες της μορφής της αριστερής γραμμικής γραμματικής. Στην περίπτωση αυτή η γραμματική δεν ονομάζεται συνηθισμένη και ανήκει στην κατηγορία type-2.

Οι συνηθισμένες γραμματικές (type-3) μπορούν εύκολα να χρησιμοποιηθούν σαν βάση στην διαδικασία της λεκτικής ανάλυσης, ενώ οι γραμματικές ελεύθερου περιεχομένου (type-2) προτιμούνται γενικά στην διαδικασία της συντακτικής ανάλυσης. Παρ' όλα αυτά, οι συντακτικοί αναλυτές οι οποίοι βασίζονται εντελώς στις γραμματικές ελεύθερου περιεχομένου δεν επαρκούν για να καλύψουν όλες τις πτυχές τις συντακτικής ανάλυσης.

2.5 ■ Χειρονακτική λεκτική και συντακτική ανάλυση

Σε προηγούμενη ενότητα είδαμε πως λειτουργεί η διαδικασία της λεκτικής και της συντακτικής ανάλυσης. Στην ενότητα αυτή θα προσπαθήσουμε να υλοποιήσουμε έναν απλό λεκτικό αναλυτή και ένα απλό συντακτικό αναλυτή. Για την υλοποίησή τους δεν θα χρησιμοποιήσουμε κάποιο εργαλείο αυτόματης παραγωγής αναλυτών. Τα πιο γνωστά εργαλεία για αυτή τη δουλειά είναι το Lex (και ο απόγονός του, το Flex) το οποίο είναι ένα εργαλείο αυτόματης παραγωγής λεκτικών αναλυτών και το Yacc (και ο απόγονός του, το Bison) το οποίο είναι ένα εργαλείο αυτόματης παραγωγής συντακτικών αναλυτών. Η χρήση αυτών των εργαλείων θα παρουσιαστεί σε επόμενη ενότητα.

Η γλώσσα που θα χρησιμοποιήσουμε για την υλοποίηση των αναλυτών είναι η Java ώστε να χρησιμοποιήσουμε τεχνικές αντικειμενοστραφούς προγραμματισμού (OOP). Η γλώσσα που θα δημιουργήσουμε δεν θα είναι μία ολοκληρωμένη γλώσσα αλλά θα είναι μία μικρή εκδοχή μιας γλώσσας προγραμματισμού. Η μικρή αυτή γλώσσα θα μας βοηθήσει να δούμε τα πιο σημαντικά χαρακτηριστικά της υλοποίησης των αναλυτών μιας γλώσσας προγραμματισμού[10].

2.5.1 Χειρονακτική λεκτική ανάλυση

Πρώτα θα ασχοληθούμε με την λεκτική ανάλυση της γλώσσας. Η λεκτική ανάλυση όπως έχουμε αναφέρει, έχει σκοπό την αναγνώριση των λεκτικών συμβόλων της γλώσσας. Ο λεκτικός αναλυτής θα πρέπει να αγνοεί τους κενούς χαρακτήρες καθώς και τα σχόλια που πιθανών να υπάρχουν στον πηγαίο κώδικα. Για την υλοποίηση του λεκτικού αναλυτή θα ακολουθήσουμε τα εξής βήματα:

1. Αναπαράσταση της λεκτικής γραμματικής με την σημειογραφία EBNF.
2. Μεταφορά κάθε κανόνα παραγωγής EBNF *N::=X* σε μία μέθοδο scan*N*, της οποίας το σώμα θα καθορίζεται από το *X*.
3. Ορισμός του λεκτικού αναλυτή ώστε να περιλαμβάνει:
    - μία ιδιωτική μεταβλητή currentChar;
    - βοηθητικές ιδιωτικές μεθόδους take και takeIt;
    - ιδιωτικές μεθόδους λεκτικής ανάλυσης οι οποίες αναπτύχθηκαν στο βήμα 2, οι οποίες θα είναι βελτιωμένες ώστε να καταγράφουν το είδος και την ορθογραφία του κάθε λεκτικού συμβόλου.
    - μία δημόσια μέθοδο scan η οποία θα αναλύει τον πηγαίο κώδικα ως 'Separator* Token'. Δηλαδή θα αγνοεί όλους τους κενούς χαρακτήρες και θα επιστρέφει το πρώτο λεκτικό σύμβολο που βρίσκει.

Η μεταβλητή currentChar θα κρατάει τον πρώτο χαρακτήρα της ακολουθίας χαρακτήρων τύπου *N*. Στην έξοδο η μεταβλητή currentChar θα περιέχει τον αμέσως επόμενο χαρακτήρα από την ακολουθία χαρακτήρων. Η μέθοδοι take και takeIt θα παίρνουν τον επόμενο χαρακτήρα από τον πηγαίο κώδικα και θα τον αποθηκεύουν στην μεταβλητή currentChar. Η διαφορά αυτών των δύο μεθόδων είναι ότι η μέθοδος take θα αποθηκεύσει στην μεταβλητή currentChar τον χαρακτήρα, μόνο αν ο χαρακτήρας είναι ο ίδιος με αυτόν που δέχεται σαν όρισμα.

Όπως αναφέρει το βήμα 1, πρέπει να ορίσουμε τη **λεκτική γραμματική** (**lexical grammar**) η οποία καθορίζει το λεξικό (lexicon) της γλώσσας. Η λεκτική αυτή γραμματική περιλαμβάνει τερματικά σύμβολα τα οποία είναι ξεχωριστοί χαρακτήρες και τα μη-τερματικά σύμβολα Token και Separator.

Token	::=	Letter (Letter | Digit)* | Digit Digit* | **+ | - | * | / | < | > | = | \ |** **; | : (=** | ε) | **~ | (|)** | eot
Separator	::=	**!** Graphic* eol | space | eol

Στους παραπάνω κανόνες παραγωγής:

- το space αντιπροσωπεύει κάθε κενό χαρακτήρα
- το eol αντιπροσωπεύει τον χαρακτήρα τέλους γραμμής (end of line)
- το eot αντιπροσωπεύει τον χαρακτήρα τέλους κειμένου (end of text)
- το Digit αντιπροσωπεύει έναν από τα ψηφία '0','1',….,'9'
- το Letter αντιπροσωπεύει έναν από τα πεζά γράμματα 'a','b',…..,'z'
- το Graphic αντιπροσωπεύει ένα κενό ή εμφανίσιμο χαρακτήρα.

Το βήμα 2 ορίζει ότι πρέπει να δημιουργήσουμε μία μέθοδο scan*N* για κάθε κανόνα παραγωγής. Με βάση την παραπάνω λεκτική γραμματική που ορίσαμε, πρέπει να δημιουργήσουμε 2 μεθόδους. Την scanToken και την scanSeparator.

Η μέθοδος scanToken:

```
private byte scanToken ()  {
   switch (currentChar) {

   case 'a': case 'b': case 'c': case 'd':
   case 'e': case 'f': case 'g': case 'h':
   case 'i': case 'j': case 'k': case 'l':
   case 'm': case 'n': case 'o': case 'p':
   case 'q': case 'r': case 's': case 't':
   case 'u': case 'v': case 'w': case 'x':
   case 'y': case 'z':
   takeIt();
   while (isLetter(currentChar) || isDigit(currentChar))
   takeIt();
return Token.IDENTIFIER;

case '0' : case '1' : case '2' :
case '3' : case '4' : case '5' :
case '6' : case '7' : case '8' :
case '9' :
    takeIt();
    while (isDigit(currentChar))
      takeIt();
return Token.INTLITERAL;

case '+' : case '-' : case'*' :
case '/' : case '<' : case'>' :
case '=' : case '\\' :
    takeIt();
    return Token.OPERATOR;
```

```
case ';' :
   takeIt();
   return Token.SEMICOLON;

case ':':
   takeIt();
   if (currentChar == '=') {
      takeIt();
      return Token.BECOMES;
   }
   else
      return Token.COLON;

case '~':
   takeIt();
   return Token.IS;

case '(':
   takeIt();
   return Token.LPAREN;

case '\000':
   return Token.EOT;

default:
        // Εμφάνιση λεκτικού λάθους
   }
}
```

- Η μέθοδος scanToken αναγνωρίζει τα λεκτικά σύμβολα -

και η μέθοδος scanSeparator:

```
private void scanSeparator () {
   switch (currentChar)  {
   case '!': {
         takeIt();
         while (isGraphic(currentchar))
            takeIt();
            take('\n');
   }
break;

case '  ' :
case '\n' :
  takeIt();
  break;
   }
```

```
}
```

- Η μέθοδος scanSeparator αναγνωρίζει τα σχόλια και τους κενούς χαρακτήρες -

Το βήμα 3 ορίζει ότι πρέπει να δημιουργήσουμε μία ολοκληρωμένη κλάση που θα αναπαριστά τον λεκτικό αναλυτή. Ο αναλυτής αυτός πρέπει να περιέχει μία μεταβλητή CurrentChar, δύο μεθόδους take και takeIt και τέλος πρέπει να περιλαμβάνει τις μεθόδους scanN που δημιουργήσαμε στο προηγούμενο βήμα.

Ο κώδικας για τον λεκτικό αναλυτή είναι:

```
public class Scanner {
  private char currentChar = // ο πρώτος χαρακτήρας από το πηγαίο κώδικα
  private byte currentKind;
  private StringBuffer currentSpelling

  private void take(char expectedChar) {
  if (currentChar == expectedChar) {
    currentSpelling.append(currentChar);
    currentChar = // ο επόμενος χαρακτήρας από το πηγαίο κώδικα
  }
  else
    // Εμφάνιση λεκτικού λάθους
  }

  private void takeIt() {
    currentSpelling.append(currentChar);
    currentChar = // ο επόμενος χαρακτήρας από το πηγαίο κώδικα
  }

  private boolean isDigit(char c) {
    // Επιστρέφει true αν ο χαρακτήρας c είναι ψηφίο
}

  private boolean isLetter(char c) {
    // Επιστρέφει true αν ο χαρακτήρας c είναι γράμμα
  }

  private boolean isGraphic(char c) {
    // Επιστρέφει true αν ο χαρακτήρας c είναι εμφανίσιμος
  }

  private scanToken() {
    // Η υλοποίηση αυτής της μεθόδου είναι η ίδια με αυτή που αναφέραμε
    // προηγουμένως
  }
```

```
  private scanSeparator() {
    // Η υλοποίηση αυτής της μεθόδου είναι η ίδια με αυτή που αναφέραμε
    //προηγουμένως
  }

  public Token scan() {
    while (currentChar == '!' || currentChar == ' ' || currentChar == '\n')
    scanSeparator();
    currentSpelling = new StringBuffer("");
    currentKind = scanToken();
    return new Token(currentKind, currentSpelling.toString());
  }
  }
}
```

- Η κατηγορία Scanner αποτελεί τον λεκτικό αναλυτή -

Τέλος πρέπει να ορίσουμε μία κλάση η οποία θα αναπαριστά τα λεκτικά σύμβολα, την **ορθογραφία** τους και τον **τύπο** τους. Ο κώδικας για την κλάση αυτή είναι:

```
public class Token{
public byte kind;
public String spelling;

public Token(byte kind, String spelling) {
  this.kind = kind;
  this.spelling = spelling;
  if (kind == IDENTIFIER)
    for(int k= BEGIN; k<=WHILE;k++)
      if(spelling.equals(spellings[k])) {
        this.kind = k;
        break;
      }
}

public final static byte IDENTIFIER = 0, INTLITERAL = 1,
          OPERATOR = 2, BEGIN = 3, CONST = 4, DO = 5,
          ELSE = 6, END = 7, IF = 8, IN = 9,
          LET = 10, THEN = 11, VAR = 12, WHILE = 13,
          SEMICOLON = 14, COLON 15, BECOMES = 16,
          IS = 17, LPAREN = 18, RPAREN = 19, EOT = 20;

private final static String[] spellings = { "<identifier>", "<integer-literal",
      "<operator>", "begin", "const", "do", "else", "end", "if", "in",
      "let", "then", "var", "while", ";", ":", ":=", "~", "(", ")","<eot>" }
```

```
}
```

- Η κατηγορία Token περιγράφει την δομή των λεκτικών συμβόλων -

2.5.2 Χειρονακτική συντακτική ανάλυση

Στο σημείο αυτό θα ασχοληθούμε με την συντακτική ανάλυση. Η μέθοδος συντακτικής ανάλυσης του συντακτικού αναλυτή θα είναι η μέθοδος bottom-up και ο αλγόριθμος που θα χρησιμοποιήσουμε θα είναι ο αλγόριθμος αναδρομικής κατάβασης. Για την υλοποίηση του συντακτικού αναλυτή από μία γραμματική ελεύθερου περιεχομένου, θα ακολουθήσουμε τα εξής βήματα:

1. Αναπαράσταση της γραμματικής με την σημειογραφία EBNF.
2. Μεταφορά κάθε κανόνα παραγωγής EBNF *N::=X* σε μία μέθοδο parse*N*, της οποίας το σώμα θα καθορίζεται από το *X*.
3. Ορισμός του συντακτικού αναλυτή ώστε να περιλαμβάνει:
   - μία ιδιωτική μεταβλητή currentToken;
   - βοηθητικές ιδιωτικές μεθόδους accept και acceptIt;
   - ιδιωτικές μεθόδους συντακτικής ανάλυσης οι οποίες αναπτύχθηκαν στο βήμα 2.
   - μία δημόσια μέθοδο parse η οποία θα καλεί την μέθοδο parseS όπου S είναι το αρχικό σύμβολο της γραμματικής.

Όπως αναφέρει το πρώτο βήμα, το πρώτο πράγμα που κάνουμε είναι να ορίσουμε την γραμματική της γλώσσας. Την γραμματική θα την αναπαραστήσουμε με την σημειογραφία EBNF.

```
Program             ::=  single-Command
Command             ::=  single-Command (; single-Command)*
single-Command      ::=  Identifier (:= Expression | (Expression) )
                       |  if Expression then single-Command
                       |  else single-Command
                       |  while Expression do single-Command
                       |  let Declaration in single-Command
                       |  begin Command end
Expression          ::=  primary-Expression (Operator primary-Expression)*
primary-Expression  ::=  Integer-Literal
                       |  Identifier
                       |  Operator primary-Expression
                       |  ( Expression )
Declaration            |  single-Declaration (; single-Declaration)
single-Declaration  ::=  const Identifier ~ Expression
                       |  var Identifier : Type-denoter
Type-denoter        ::=  Identifier
```

Τα μη-τερματικά Identifier, Operator και Integer-Literal έχουν οριστεί στον λεκτικό αναλυτή και έτσι δεν τα ορίζουμε και εδώ.

Το βήμα 2 αναφέρει ότι πρέπει να μετατρέψουμε κάθε κανόνα παραγωγής της γραμματικής σε μεθόδους parseN. Έτσι έχουμε:

```
private void parseProgram();
private void parseCommand();
private void parseSingleCommand();
private void parseExpression();
private void parsePrimaryExpression();
private void parseDeclaration();
private void parseSingleDeclaration();
private void parseTypeDenoter();
private void parseIdentifier();
private void parseIntegerLiteral();
private void parseOperator();

private void parseSingleDecleration () {
  switch (currentToken.kind)  {

  case Token.CONST:
    {

      acceptIt ();
      parseIdentifier ();
      accept(Token.IS);
      parseExpression();
    }
    break;
case Token.VAR:
   {
     acceptIt ();
     parseIdentifier ();
     accept(Token.COLON);
     parseTypeDenoter();
   }
default:
   // Εμφάνιση συντακτικού λάθους
  }
}

private void parseCommand() {
  parseSingleCommand();
  while (currentToken.kind == Token.SEMICOLON)
  {
    acceptIt();
    parseSingleCommand();
  }
```

```
}

private void parseProgram ()  {
parseSingleCommand();
}

private void parseSingleCommand() {
switch (currentToken.kind) {

case Token.IDENTIFIER:
   {
     parseIdentifier();
     switch (currentToken.kind) {
     case Token.BECOMES:
       {
          acceptIt();
          parseExpression();
       }
       break;

case Token.LPAREN:
   {
       acceptIt();
       parseExpression();
       accept(Token.LPAREN);
   }
    break;
 default:
  // Εμφάνιση συντακτικού λάθους
  }
}
break;

case Token.IF:
  {
    acceptIt();
    parseExpression();
    accept(Token.THEN);
    parseSingleCommand();
    accept(Token.ELSE);
    parseSingleCommand();
  }
   break;

case Token.WHILE:
  {
```

```
      acceptIt();
      parseExpression();
      accept(Token.DO);
      parseSingleCommand();
   }
   break;

case Token.LET:
   {
     acceptIt();
     parseDeclaration();
     accept(Token.IN);
     parseSingleCommand();
   }
   break;

case Token.BEGIN:
   {
     acceptIt();
     parseCommand();
     accept(Token.END);
   }
    break;
 default:
    // Εμφάνιση συντακτικού λάθους
   }
}

private void parseExpression () {
parsePrimaryExpression();
while (currentToken.kind == Token.OPERATOR) {
parseOreraton ();
parsePrimaryExpression();
   }
}

private void parsePrimaryExpression ()  {
  switch (currentToken.kind)  {
  case Token.INTLITERAL:
     parseIntegerLITERAL();
     break;
  case Token.IDENTIFIER:
     parseIdentifier();
     break;
  case Token.OPERATOR:
     {
```

```
        parseOperator();
        parsePrimaryExpression();
    }
    break;
  case Token.LPAREN:
    {
      acceptIt();
      parseExpression();
      accept(Token.RPAREN);
    }
    break;
  default:
    // Εμφάνιση συντακτικού λάθους
}

private void parseTypeDenoter () {
   parseIdentifier() ;
}

private void parseIdentifier () {
  if (currentToken.kind == Token.IDENTIFIER)
     currentToken = scanner.scan();
  else
   // Εμφάνιση συντακτικού λάθους
}
```

- Οι μέθοδοι parseN για την συντακτική ανάλυση των μη-τερματικών συμβόλων -

Το βήμα 3 ορίζει ότι πρέπει να δημιουργήσουμε μία ολοκληρωμένη κλάση που θα αναπαριστά τον συντακτικό αναλυτή. Ο αναλυτής αυτός πρέπει να περιέχει μία μεταβλητή currentToken, δύο μεθόδους accept και acceptIt και τέλος πρέπει να περιλαμβάνει τις μεθόδους parse*N* που δημιουργήσαμε στο προηγούμενο βήμα.

Ο κώδικας για τον συντακτικό αναλυτή είναι:

```
public class Parser {

private Token currentToken;

private void accept (byte expectedKind)  {
   if  (currentToken.kind == expectedKind)
         currentToken = scanner.scan();
   else
    // Εμφάνιση συντακτικού λάθους
}
```

```
private void acceptIt ()   {
   current Token = scanner.scan();
}

//... Εδώ θα μπουν οι μέθοδοι parseN που δημιουργήσαμε προηγουμένως

public void parse ()  {
   currentToken = scanner.scan();
   parseProgram();
   if (currentToken.kind !=Token.EOT)
    // Εμφάνιση συντακτικού λάθους
  }
}
```

- Η κατηγορία Parser αποτελεί τον συντακτικό αναλυτή -

Αξίζει να σημειώσουμε μερικά χαρακτηριστικά του συντακτικού αναλυτή:

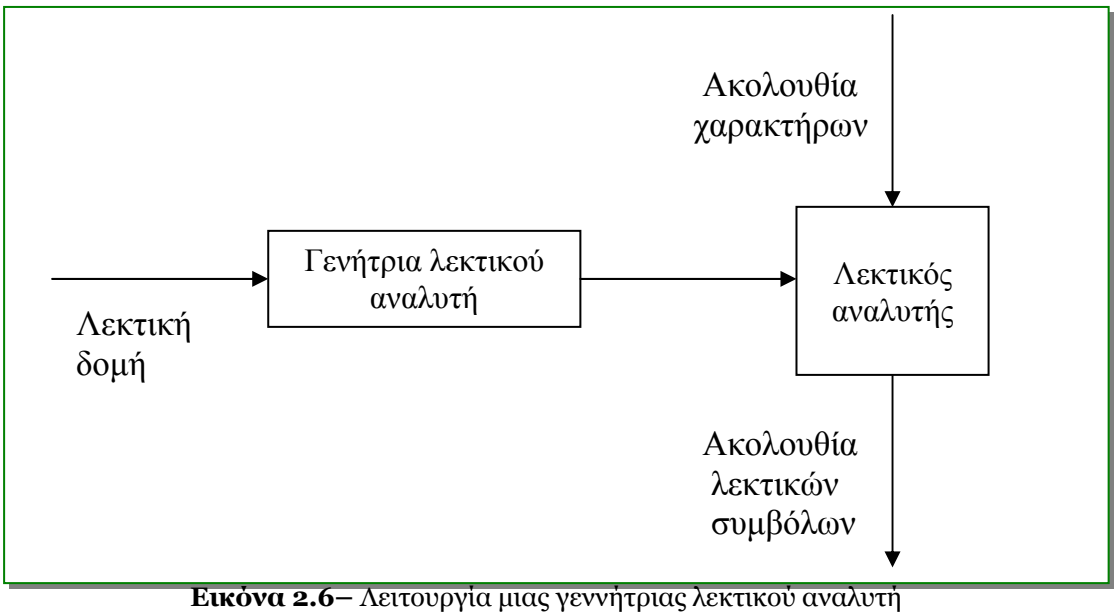
- Ο συντακτικός αναλυτής αγνοεί την ορθογραφία των λεκτικών συμβόλων. Το μόνο που εξετάζει είναι το είδος τους.

- Μετά από την συντακτική ανάλυση, η μέθοδος parse ελέγχει αν το λεκτικό σύμβολο που ακολουθεί το πρόγραμμα είναι το σύμβολο τέλους κειμένου (eot).

- Οι μέθοδοι συντακτικής ανάλυσης (parseN) είναι αμοιβαίως αναδρομικές όπως είναι και οι κανόνες παραγωγής. Για παράδειγμα, η μέθοδος parseCommand καλεί την parseSingleCommand η οποία μπορεί να καλέσει αναδρομικά την parseCommand.

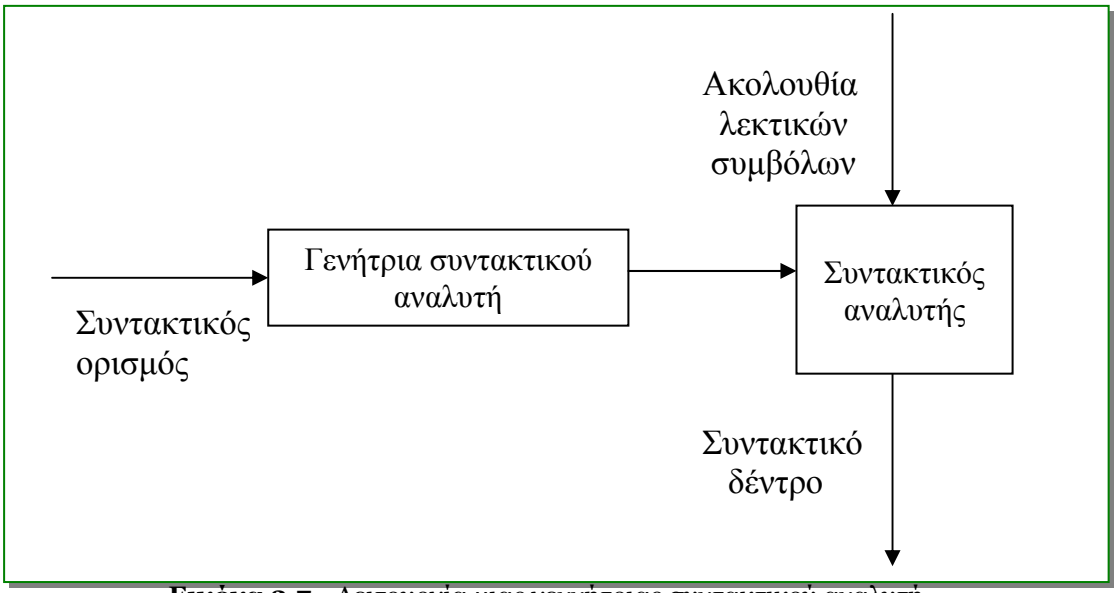
2.6 ■ Αυτοματοποιημένη παραγωγή λεκτικών και συντακτικών αναλυτών

Η κατασκευή λεκτικών και συντακτικών αναλυτών είναι τόσο απλή που με την χρήση κατάλληλων προγραμμάτων μπορεί να αυτοματοποιηθεί. Τα προγράμματα τα ονομάζουμε γεννήτριες αναλυτών (**analyzer generators**). Μία γεννήτρια λεκτικού αναλυτή δέχεται σαν είσοδό του την λεκτική δομή μιας γλώσσας, η οποία ορίζει πως τα λεκτικά σύμβολα δημιουργούνται από χαρακτήρες, και παράγει ένα λεκτικό αναλυτή (ένα πρόγραμμα C για παράδειγμα) για την γλώσσα. Η διαδικασία αυτή παρουσιάζεται στην εικόνα 2.6.



Εικόνα 2.6– Λειτουργία μιας γεννήτριας λεκτικού αναλυτή

Μία γεννήτρια συντακτικού αναλυτή δέχεται σαν είσοδό του τον συντακτικό ορισμό της γλώσσας και παράγει ένα λεκτικό αναλυτή (ένα πρόγραμμα C για παράδειγμα) της γλώσσας. Η διαδικασία αυτή παρουσιάζεται στην εικόνα 2.7.



Εικόνα 2.7– Λειτουργία μιας γεννήτριας συντακτικού αναλυτή

Η πιο γνωστή γεννήτρια λεκτικού αναλυτή είναι το εργαλείο **Lex** και η πιο γνωστή γεννήτρια συντακτικού αναλυτή είναι το εργαλείο **Yacc**. Τα εργαλεία αυτά αρχικά αναπτύχθηκαν για κοινή χρήση σε περιβάλλον Unix. Ωστόσο υπάρχουν εκδόσεις και για περιβάλλον Windows αλλά και για MacOs. Οι εκδόσεις αυτές είναι συνήθως το **Flex** (γεννήτρια λεκτικού αναλυτή) και το **Bison** (γεννήτρια συντακτικού αναλυτή). Υπάρχουν και άλλες εκδόσεις των εργαλείων αυτών όπως το Lex++ και το Yacc++ οι οποίες είναι εκδόσεις για αντικειμενοστραφές προγραμματισμό. Όλες είναι παρόμοιες στην λειτουργία τους αλλά υπάρχουν και διαφορές που είναι μοναδικές σε κάθε έκδοση και

χρειάζεται προσοχή. Στην εργασία αυτή θα αναφερθούμε με τα εργαλεία Flex και Bison.

Υπάρχουν αρκετοί λόγοι για να χρησιμοποιήσει κάποιος μια γεννήτρια λεκτικού ή συντακτικού αναλυτή.

- Διευκολύνει την προσπάθεια παραγωγής ενός διερμηνευτή ή ενός μεταγλωττιστή.
- Μεγαλύτερη εμπιστοσύνη στην λειτουργία του μεταγλωττιστή, βασισμένη στην εμπιστοσύνη που υπάρχει στις γεννήτριες.
- Συμβατότητα μεταξύ των ποικιλιών των μεταγλωττιστών
- Η πιθανότητα ενσωμάτωσης διαφορετικών τύπων ανάλυσης στο ίδιο το εργαλείο.
- Διευκόλυνση της συντήρησης του μεταγλωττιστή

2.6.1 Το εργαλείο Flex

Το πιο γνωστό εργαλείο αυτόματης κατασκευής λεκτικών αναλυτών είναι το Flex (Fast Lexical analyzer generator), το οποίο αρχικά αναπτύχθηκε για χρήση σε συνδυασμό με το Bison. Για να καθορίσουμε τα λεκτικά σύμβολα της γλώσσας χρησιμοποιούμε μία σημειογραφία παρόμοια με την σημειογραφία των κανονικών εκφράσεων. Η σημειογραφία που χρησιμοποιούμε στο Flex διαφέρει από την σημειογραφία των κανονικών εκφράσεων για δύο σημαντικούς λόγους:

1. Επιτρέπει την αποδοτικότερη αναπαράσταση, σε σχέση με τον αριθμό των χαρακτήρων που χρησιμοποιούνται στην αναπαράσταση ορισμένων συμβόλων
2. Επεκτείνει την δύναμη της σημειογραφίας των κανονικών εκφράσεων σε ορισμένες περιπτώσεις.

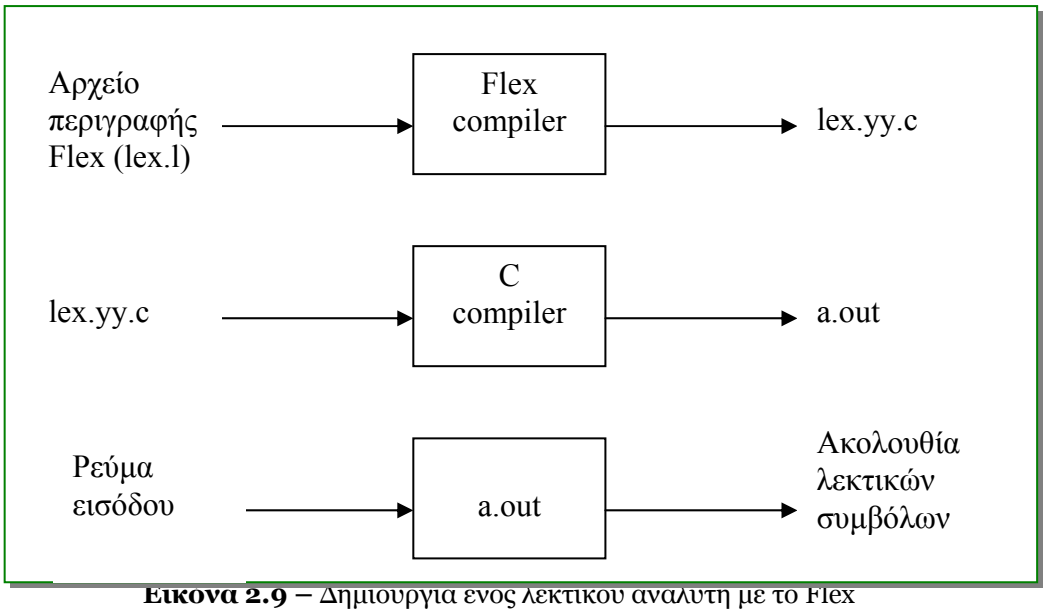
Σαν παράδειγμα του πρώτου λόγου που αναφέραμε είναι ότι η σημειογραφία των κανονικών εκφράσεων δεν μπορεί να αναπαραστήσει την έννοια "όλοι οι χαρακτήρες του αλφαβήτου εκτός από έναν" χωρίς να καταγράψουμε όλους τους χαρακτήρες στο αλφάβητο.

Σαν παράδειγμα του δεύτερου λόγου που αναφέραμε είναι ότι η σημειογραφία του Flex επιτρέπει την αναπαράσταση της έννοιας ενός συμβόλου το οποίο εμφανίζεται σε συγκεκριμένο μόνο περιεχόμενο.

Τη σημειογραφία που χρησιμοποιούμε για να καθορίσουμε τα λεκτικά σύμβολα στο εργαλείο Flex την ονομάζουμε γλώσσα Flex. Πριν προχωρήσουμε στις λεπτομέρειες της γλώσσας Flex ας δούμε πως λειτουργεί γενικά το εργαλείο Flex σε σχέση με έναν μεταγλωττιστή C.

Ο προγραμματιστής πρώτα δημιουργεί ένα αρχείο με όνομα lex.l το οποίο είναι μία περιγραφή του λεκτικού αναλυτή. Η περιγραφή αυτή γίνεται με την γλώσσα Flex. Έπειτα, χρησιμοποιώντας το εργαλείο flex με είσοδο το αρχείο αυτό, παράγεται ένα πρόγραμμα C με όνομα lex.yy.c. Στο αρχείο αυτό υλοποιείται μία συνάρτηση C με όνομα yylex(). Η συνάρτηση αυτή όποτε

καλείται επιστρέφει το επόμενο λεκτικό σύμβολο. Τέλος μεταγλωττίζουμε το αρχείο lex.yy.c με ένα μεταγλωττιστή C ώστε να παραχθεί το τελικό εκτελέσιμο αρχείο, το οποίο είναι ο λεκτικός αναλυτής ο οποίος μετατρέπει ένα ρεύμα εισόδου σε ακολουθία λεκτικών συμβόλων. Η διαδικασία αυτή παρουσιάζεται στην εικόνα 2.9.



Εικόνα 2.9 – Δημιουργία ενός λεκτικού αναλυτή με το Flex

Ένα πρόγραμμα flex (αρχείο περιγραφής) αποτελείται από τρία τμήματα:

```
Δηλώσεις
%%
κανόνες μετάφρασης
%%
βοηθητικές συναρτήσεις
```

Στο τμήμα των δηλώσεων περιλαμβάνονται οι δηλώσεις μεταβλητών και των σταθερών καθώς και δηλώσεις του Flex οι οποίες χρησιμοποιούνται στις κανονικές εκφράσεις οι οποίες δηλώνονται στο επόμενο τμήμα του αρχείου flex το οποίο είναι το τμήμα των κανόνων μετάφρασης. Για παράδειγμα, πριν ορίσουμε μία κανονική έκφραση η οποία θα αναγνωρίζει αναγνωριστές (identifiers) πρέπει πρώτα να δηλώσουμε τις έννοιες των γραμμάτων και των ψηφίων. Οι έννοιες αυτές θα πρέπει να δηλωθούν στο πρώτο τμήμα του αρχείου flex (τμήμα δηλώσεων) και μπορούν να δηλωθούν ως εξής:

```
letter[a-z]
digit [0-9]
```

Παρατηρούμε ότι δεν χρειάζεται να καταγράψουμε κάθε χαρακτήρα στα σύνολα a-z και 0-9. Αφού κάνουμε αυτές τις δηλώσεις μπορούμε να τις

χρησιμοποιήσουμε στην κανονική έκφραση η οποία θα αναγνωρίζει τους αναγνωριστές. Αυτό θα γίνει ως εξής:

identifier {letter}({letter} | {digit})*

Η παραπάνω δήλωση θα πρέπει να περιληφθεί και αυτή στο τμήμα των δηλώσεων του αρχείου flex.

Στο δεύτερο τμήμα του αρχείο flex τοποθετούνται οι κανόνες μετάφρασης. Οι κανόνες μετάφρασης ενός προγράμματος Flex είναι δηλώσεις της μορφής :

p_1	{ action $_1$ }
p_2	{ action $_2$ }
...	
p_n	{ action$_n$ }

όπου p_i είναι μία κανονική έκφραση και action$_i$ είναι ένα κομμάτι κώδικα ο οποίος περιγράφει ποια ενέργεια πρέπει να εκτελέσει ο λεκτικός αναλυτής όταν αναγνωρίσει το σύμβολο που περιγράφει το p_i. Ο κώδικας αυτός είναι συνήθως C αλλά γενικά μπορεί να χρησιμοποιηθεί οποιαδήποτε γλώσσα υλοποίησης[11]. Για παράδειγμα, ο ακόλουθος κανόνας τυπώνει το μήνυμα "identifier recognized" κάθε φορά που εμφανίζεται ένας αναγνωριστής.

```
{identifier}  {printf("identifier recognized\n");}
```

Στο τρίτο και τελευταίο κομμάτι του αρχείου flex δηλώνονται όλες οι βοηθητικές συναρτήσεις τις οποίες μπορεί να καλούν οι ενέργειες στο τμήμα των κανόνων μετάφρασης.

Το εργαλείο Flex διαθέτει μερικές μεταβλητές οι οποίες διευκολύνουν την λεκτική ανάλυση. Οι πιο γνωστές μεταβλητές είναι οι yytext και yylineno. Στην μεταβλητή yytext αποθηκεύεται το αλφαριθμητικό από το οποίο αποτελείται το λεκτικό σύμβολο που αναγνωρίστηκε τελευταίο ενώ η μεταβλητή yylineno αποθηκεύει τον αριθμό της γραμμής του πηγαίου κώδικα στην οποία αναγνωρίστηκε το τελευταίο λεκτικό σύμβολο. Παρακάτω παρουσιάζεται ένα ολοκληρωμένο πρόγραμμα Flex το οποίο κάθε φορά που εμφανίζεται ένας αναγνωριστής τυπώνει το αλφαριθμητικό από το οποίο αποτελείται καθώς και την γραμμή στην οποία βρέθηκε. Το πρόγραμμα αυτό είναι αρκετά απλό και δεν χρησιμοποιούμε καμία βοηθητική συνάρτηση. Παρ'όλα αυτά είναι ενδεικτικό για τον τρόπο συγγραφής ενός προγράμματος Flex.

```
letter          [a-z]
digit           [0-9]
identifier      {letter} ({letter} | {digit})*
%%
{identifier}    {printf("identifier %s on line %d\n",yytext,yylineno);}
%%
```

Αν υποθέσουμε ότι το παραπάνω πρόγραμμα είναι αποθηκευμένο σε ένα αρχείο με όνομα firstlex.l, τότε ο λεκτικός αναλυτής θα παραχθεί με την εντολή:

```
flex firstlex.l
```

Η εντολή αυτή θα παράγει ένα λεκτικό αναλυτή σε C και το όνομα του αρχείου θα είναι lex.yy.c. Έπειτα, μπορούμε να μεταγλωττίσουμε το αρχείο lex.yy.c με ένα μεταγλωττιστή της C και να παράγουμε ένα εκτελέσιμο πρόγραμμα. Η εντολή με την οποία θα γίνει αυτό είναι:

```
gcc –o firstlex lex.yy.c
```

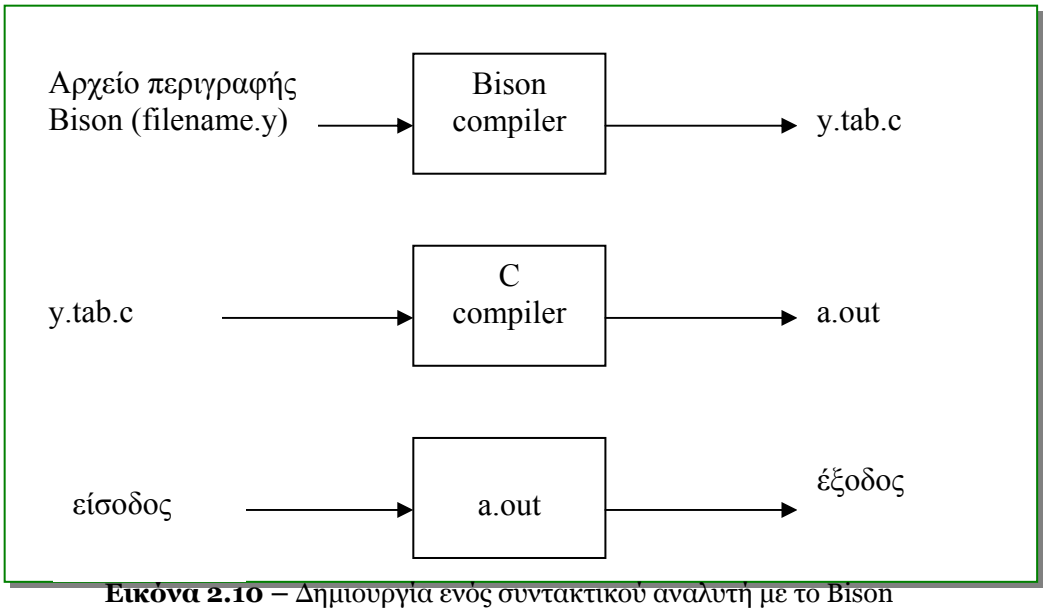
Το όνομα του εκτελέσιμου αρχείου θα είναι firstlex αφού έτσι ορίσαμε με την παράμετρο – ο. Αν δώσουμε σαν είσοδο στο firstlex ένα πηγαίο πρόγραμμα C τότε αυτό θα βρει όλους τους αναγνωριστές στο πρόγραμμα αυτό. Για παράδειγμα αν έχουμε ένα πηγαίο πρόγραμμα C με όνομα cprog, η εκτέλεση του firstlex με είσοδό του, το πρόγραμμα cprog θα γίνει με την εντολή:

```
firstlex < cprog
```

2.6.2 Το εργαλείο Bison

Το εργαλείο Bison είναι ίσος το πιο γνωστό εργαλείο παραγωγής συντακτικών αναλυτών. Ο συντακτικός αναλυτής που παράγει, χρησιμοποιεί μία μέθοδο γενικού τύπου bottom-up συντακτικής ανάλυσης και επιτρέπει την εισαγωγή ενεργειών γραμμένες σε C. Πριν προχωρήσουμε στις λεπτομέρειες του εργαλείου Bison ας δούμε πως λειτουργεί γενικά σε σχέση με έναν μεταγλωττιστή C.

Ο προγραμματιστής πρώτα δημιουργεί ένα αρχείο με όνομα filename.y το οποίο είναι μία περιγραφή του συντακτικού αναλυτή. Έπειτα, χρησιμοποιώντας το εργαλείο Bison με είσοδο το αρχείο αυτό, παράγεται ένα πρόγραμμα C με όνομα y.tab.c καθώς και ένα αρχείο επικεφαλίδας με όνομα y.tab.h. Το αρχείο y.tab.c είναι ο συντακτικός αναλυτής γραμμένος σε γλώσσα C ενώ το αρχείο y.tab.h καθορίζει τα λεκτικά σύμβολα χρησιμοποιώντας τα κατάλληλα defines ώστε να μπορούν να χρησιμοποιηθούν σε κάποιον λεκτικό αναλυτή ή από το flex. Τέλος μεταγλωττίζουμε το αρχείο y.tab.c με ένα μεταγλωττιστή C ώστε να παραχθεί το τελικό εκτελέσιμο αρχείο, το οποίο είναι ο συντακτικός αναλυτής. Η διαδικασία αυτή παρουσιάζεται στην εικόνα 2.10.



Εικόνα 2.10 – Δημιουργία ενός συντακτικού αναλυτή με το Bison

Ένα πρόγραμμα bison (αρχείο περιγραφής) αποτελείται από τέσσερα τμήματα:

```
%{
Δηλώσεις C
%}
Δηλώσεις Bison
%%
Γραμματικοί κανόνες
%%
Βοηθητικές συναρτήσεις C
```

Στο τμήμα δηλώσεων C μπορούμε να δηλώσουμε μεταβλητές και σταθερές τις οποίες θα χρησιμοποιήσουμε αργότερα στις ενέργειες των κανόνων. Επίσης, μπορούμε να χρησιμοποιήσουμε τον προεπεξεργαστή της C για να συμπεριλάβουμε αρχεία επικεφαλίδων με την εντολή #include.

Στο τμήμα δηλώσεων Bison δηλώνουμε τα ονόματα των τερματικών και των μη-τερματικών. Επίσης, μπορούμε να δηλώσουμε την προτεραιότητα των τελεστών καθώς και τα ονόματα των λεκτικών συμβόλων. Για παράδειγμα η πρόταση :

```
%token INTEGER
```

δηλώνει οτι το INTEGER είναι ένα λεκτικό σύμβολο. Τα λεκτικά σύμβολα που δηλώνονται στο σημείο αυτό μπορούν να χρησιμοποιηθούν και στο τρίτο και τέταρτο μέρος του αρχείου περιγραφής Bison.

Στο τμήμα των γραμματικών κανόνων υπάρχουν δηλώσεις οι οποίες καθορίζουν πως κατασκευάζονται τα μη-τερματικά σύμβολα. Οι δηλώσεις αυτές αποτελούνται από ένα κανόνα γραμματικής και μία συσχετισμένη σημασιολογική ενέργεια. Στην ουσία οι κανόνες γραμματικής είναι οι κανόνες

παραγωγής της γραμματικής που έχουμε αναπτύξει σε σημειογραφία EBNF. Ένας κανόνας παραγωγής, όπως έχουμε δει και σε προηγούμενη ενότητα έχει την μορφή:

```
<left side> → <alt 1> | <alt 2> | <alt 3> | ... | <alt n>
```

Ένας κανόνας παραγωγής μιας γραμματικής στο αρχείο bison θα έχει την μορφή:

```
<left side>  :      <alt 1>      { σημασιολογική ενέργεια 1 }
             |      <alt 2>      { σημασιολογική ενέργεια 2 }
             |      <alt 3>      { σημασιολογική ενέργεια 3 }
           . . .
             |      <alt n>      { σημασιολογική ενέργεια n }
             ;
```

Στο Bison ένας χαρακτήρας σε αποστρόφους πχ 'c' θεωρείται το τερματικό σύμβολο c ενώ αλφαριθμητικά που δεν βρίσκονται σε αποστρόφους θεωρούνται μη-τερματικά σύμβολα. Ο πρώτος κανόνας στο αρχείο περιγραφής bison θεωρείται το αρχικό σύμβολο. Μπορούμε όμως να ρυθμίσουμε εμείς το αρχικό σύμβολο με την εντολή %start. Για παράδειγμα:

```
%start S
```

όπου S κάποιο μη τερματικό σύμβολο της γραμματικής.

Σε μία σημασιολογική ενέργεια το σύμβολο $$ αναφέρεται στην τιμή που σχετίζεται με το μη-τερματικό στο αριστερό μέρος του κανόνα, ενώ το σύμβολο $i αναφέρεται στην τιμή που σχετίζεται με το i σύμβολο (τερματικό ή μη-τερματικό) στο δεξί μέρος του κανόνα. Η τιμή του $$ υπολογίζεται σε σχέση με τους όρους των $i. Το πρώτο σύμβολο είναι πάντα το σύμβολο 1.
Για παράδειγμα ο κανόνας:

```
expr: expr '+' expr {$$ = $1 + $3; }
```

προσθέτει την αριθμητική τιμή του πρώτου όρου με τον τρίτο όρο και αποθηκεύει το αποτέλεσμα στο αριστερό μέρος του κανόνα. Αν δεν καθοριστεί σημασιολογική ενέργεια, η προκαθορισμένη είναι η { $$ = $1; }. Αξίζει να σημειωθεί ότι η παραπάνω προκαθορισμένη ενέργεια δεν έχει κανένα αποτέλεσμα, αν δεν έχουν δοθεί αρχικές τιμές στα ορίσματα των τερματικών συμβόλων της συμβολοσειράς εισόδου, οπότε ο συντακτικός αναλυτής λειτουργεί πολύ απλά και χρησιμοποιείται για την αναγνώριση ή απόρριψή της.

Το τελευταίο τμήμα του αρχείου περιγραφής Bison αποτελείται από βοηθητικές συναρτήσεις C. Στο τμήμα αυτό πρέπει να υλοποιηθεί μία συνάρτηση με όνομα yylex() η οποία θα πραγματοποιεί στην ουσία την

λεκτική ανάλυση. Εναλλακτικά μπορούμε να συνδέσουμε το bison με το flex και να χρησιμοποιήσουμε το flex για την λεκτική ανάλυση και όχι μια δική μας συνάρτηση. Για να πετύχουμε την σύνδεση αυτή πρέπει να συμπεριλάβουμε το αρχείο lex.yy.c στην αρχή του τελευταίου τμήματος τους αρχείου bison, δηλαδή:

```
#include "lex.yy.c"
```

Με τον τρόπο αυτό δεν χρειάζεται να υλοποιήσουμε εμείς ένα λεκτικό αναλυτή με όνομα yylex(). Στην περίπτωση αυτή το αρχείο εξόδου του Flex πρέπει να μεταγλωττιστεί σαν τμήμα του αρχείου εξόδου του Bison y.tab.c. Έτσι η συνάρτηση yylex(), η οποία πλέον παρέχεται από το αρχείο εξόδου του Flex, έχει πρόσβαση στα ονόματα των λεκτικών συμβόλων. Για παράδειγμα, σε ένα σύστημα Unix, αν το όνομα του αρχείου περιγραφής του Flex έχει όνομα first.l και το όνομα αρχείου περιγραφής του Bison έχει όνομα second.y τότε για να παράγουμε τον συντακτικό αναλυτή μπορούμε να δώσουμε στην γραμμή εντολών:

```
flex first.l
bison second.y
gcc y.tab.c –ly -ll
```

Στην περίπτωση λογικού λάθους στις σημασιολογικές ενέργειες, καλείται η συνάρτηση yyerror(). Αν κληθεί η yyerror() τερματίζει την λειτουργία του συντακτικού αναλυτή. Για τον λόγο αυτό ο προγραμματιστής μπορεί να υλοποιήσει μία δική του εκδοχή της yyerror().

Πρέπει να αναφέρουμε ότι κύρια συνάρτηση C που κατασκευάζεται από το Bison είναι η yyparse(). Η συνάρτηση αυτή εκτελεί στην ουσία τον συντακτικό έλεγχο. Η συνάρτηση yyparse() επιστρέφει έναν ακέραιο με τιμή 0, όταν αναγνωριστεί η συμβολοσειρά εισόδου, όταν δηλαδή γίνει ελάττωση στο αρχικό σύμβολο της γραμματικής και η συμβολοσειρά έχει διαβαστεί πλήρως. Σε περίπτωση σφάλματος, η yyparse() επιστρέφει τιμή 1.

Ολόκληρη η διαδικασία παραγωγής ενός συντακτικού αναλυτή με την χρήση του Bison από μία γραμματική αποτελείται από τα εξής τέσσερα τμήματα:

1. Καθορισμός της γραμματικής χρησιμοποιώντας την σύνταξη του Bison. Για κάθε γραμματικό κανόνα της γλώσσας πρέπει να καθοριστεί και μία σημασιολογική ενέργεια η οποία θα εκτελεστεί όταν αναγνωριστεί ο κανόνας. Η ενέργεια αυτή περιγράφεται με ακολουθίες εντολών C.

2. Υλοποίηση ενός λεκτικού αναλυτή ο οποίος θα επιστρέφει τα λεκτικά σύνολα στον συντακτικό αναλυτή. Ο λεκτικός αναλυτής μπορεί να γραφτεί χειροκίνητα ή μπορεί να χρησιμοποιηθεί ο λεκτικός αναλυτής που παράγεται από το εργαλείο Flex.

3. Υλοποίηση μιας συνάρτησης ελέγχου η οποία καλεί την συνάρτηση yyparse() του Bison η οποία πραγματοποιεί την συντακτική ανάλυση.

4. Υλοποίηση συναρτήσεων για την αναφορά λαθών.

ΚΕΦΑΛΑΙΟ 3ο

Υλοποιημένη Εργασία

3.1 ■ Συνοπτική περιγραφή

Στο πρώτο κεφάλαιο της εργασίας αναφέραμε την έννοια του κελύφους εργασίας καθώς και την λειτουργία του σε σχέση με το Λειτουργικό Σύστημα. Στο κεφάλαιο αυτό θα ασχοληθούμε εκτενώς με τα κελύφη εργασίας και θα γίνει ανάλυση της υλοποιημένης εργασίας η οποία αποτελεί το πειραματικό κομμάτι αυτής της πτυχιακής εργασίας. Το κέλυφος το οποίο έχει υλοποιηθεί πραγματοποιεί όλες τις βασικές λειτουργίες ενός κελύφους εργασίας του Unix και έχει βασίσει την λειτουργία του στο πιο γνωστό κέλυφος του Unix/Linux, το κέλυφος bash. Για την υλοποίηση ενός κελύφους εργασίας είναι απαραίτητη η εσωτερική γνώση των Λειτουργικών Συστημάτων καθώς και βασικές γνώσεις υλοποίησης των μεταγλωττιστών. Για το λόγο αυτό, στα προηγούμενα δύο κεφάλαια αναλύσαμε έννοιες οι οποίες θα μας δώσουν ένα καλό ξεκίνημα για την κατανόηση του τρόπου υλοποίησης ενός κελύφους εργασίας. Στις ενότητες που ακολουθούν θα αναφέρουμε όλα τα βασικά χαρακτηριστικά ενός κελύφους εργασίας και στο τέλος σχεδόν κάθε ενότητας θα αναφέρουμε τον τρόπο υλοποίησής τους στο πειραματικό κέλυφος της εργασίας το οποίο έχει όνομα wish (Windows Interactive Shell). Τέλος να αναφέρουμε ότι το πειραματικό κέλυφος αναπτύχθηκε σε C++ κάνοντας χρήση αρκετών βιβλιοθηκών και για το λόγο αυτό υποθέτουμε πως ο αναγνώστης διαθέτει βασικές γνώσεις χρήσης της γλώσσας C++ καθώς και βασικές γνώσεις χρήσης του Λειτουργικού Συστήματος Linux.

3.2 ■ Οι βιβλιοθήκες readline, ncurses και η STL της C++

Μία κοινή αρχή των προγραμματιστών είναι να μην ανακαλύπτουν τον τροχό κάθε φορά που θέλουν να υλοποιήσουν μία εφαρμογή. Για το λόγο αυτό έχουν αναπτυχθεί διάφορες βιβλιοθήκες οι οποίες προσφέρουν στους προγραμματιστές έτοιμες συναρτήσεις και κατηγορίες (classes) οι οποίες πραγματοποιούν κοινές εργασίες τις οποίες δεν χρειάζεται να υλοποιήσει ξανά από την αρχή ο κάθε προγραμματιστής. Ένας πολύ σημαντικός λόγος για τον οποίο οι γλώσσες C και C++ είναι τόσο δημοφιλείς είναι η δύναμη τους να χρησιμοποιούν έτοιμες βιβλιοθήκες γραμμένες από τρίτους προγραμματιστές. Για την υλοποίηση του κελύφους αυτής της πτυχιακής εργασίας χρησιμοποιήθηκαν οι βιβλιοθήκες readline, ncurses και η STL της C++. Οι βιβλιοθήκες Readline και ncurses είναι διαθέσιμες σε κάθε χρήστη Unix ή Linux και η καθιερωμένη βιβλιοθήκη STL αποτελεί πλέον αναπόσπαστο τμήμα κάθε μεταγλωττιστή της C++ εφόσον ακολουθεί το πρότυπο **ANSI C++**.

3.2.1 Η βιβλιοθήκη Readline

Η βιβλιοθήκη **Readline** παρέχει ένα σύνολο συναρτήσεων για την επεξεργασία της γραμμής εντολών και αρχικά κατασκευάστηκε για να παρέχει την δυνατότητα επεξεργασίας της γραμμής εντολών του κελύφους bash. Δημιουργήθηκε και συντηρείται από το έργο GNU. Την βιβλιοθήκη μπορεί να την προμηθευτεί κάποιος από το http://ftp.gnu.org/pub/gnu/readline/. Η βιβλιοθήκη προσφέρει την δυνατότητα επεξεργασίας του κειμένου που πληκτρολογεί ο χρήστης στην γραμμή εντολών καθώς το πληκτρολογεί. Για παράδειγμα σε μία εφαρμογή η οποία χρησιμοποιεί την Readline, ο χρήστης μπορεί να προχωρήσει τον κέρσορα μία θέση πίσω πατώντας τον συνδυασμό πλήκτρων Control – b, με τον συνδυασμό Control – f προχωράει τον κέρσορα μία θέση μπροστά και με τον συνδυασμό Alt – f προχωράει τον κέρσορα μία λέξη μπροστά. Η Readline παρέχει και μία ποικιλία εξελιγμένων χαρακτηριστικών όπως το δαχτυλίδι αποκοπής (**kill ring**) και την συμπλήρωση με το πλήκτρο tab (**tab completion**).

Το δαχτυλίδι αποκοπής είναι μία δυνατότητα με την οποία ένας χρήστης μπορεί να πραγματοποιήσει αποκοπή και επικόλληση κειμένου. Αποκοπή κειμένου ονομάζουμε την διαδικασία με την οποία ενώ σβήνουμε ένα κείμενο από την γραμμή εντολών, το κείμενο αυτό αποθηκεύεται για μετέπειτα χρήση. Όταν ο χρήστης αποκόψει ένα κείμενο, αυτό διαγράφεται από την γραμμή εντολών αλλά αποθηκεύεται σε μία δομή η οποία ονομάζεται δαχτυλίδι αποκοπής. Ο χρήστης μπορεί να πραγματοποιήσει συνεχόμενες αποκοπές κειμένου και το κείμενο που αποκόβει κάθε φορά να εισέρχεται στο δαχτυλίδι αποκοπής. Όταν ο χρήστης αποφασίσει να επικολλήσει το κείμενο που απόκοψε τελευταίο, αυτό θα εμφανιστεί στην γραμμή εντολών, ξεκινώντας από την θέση του κέρσορα.

Η συμπλήρωση με το πλήκτρο tab είναι μία δυνατότητα που παρέχει η Readline με την οποία ο χρήστης πατώντας το πλήκτρο tab του πληκτρολογίου του, πραγματοποιεί αυτόματη συμπλήρωση του κειμένου. Ο προγραμματιστής της Readline μπορεί να ρυθμίσει τον τύπο της συμπλήρωσης αλλά ο προκαθορισμένος τύπος συμπλήρωσης είναι η συμπλήρωση των ονομάτων των αρχείων.

Μία πολύ χρήσιμη δυνατότητα που προσφέρει η βιβλιοθήκη Readline είναι η διατήρηση μιας λίστας με κείμενο που είχε πληκτρολογηθεί σε προηγούμενη στιγμή. Η λίστα αυτή χρησιμοποιείται για την ανάκτηση προηγούμενων κειμένων που πληκτρολογήθηκαν αλλά και για την επεξεργασία τους. Αυτή η δυνατότητα ονομάζεται History και έχει τοποθετηθεί και σε ξεχωριστή βιβλιοθήκη, την βιβλιοθήκη History. Η βιβλιοθήκη History μπορεί να χρησιμοποιηθεί σε εφαρμογές και χωρίς την χρήση της Readline.

Οι εφαρμογές που θέλουν να κάνουν χρήση της βιβλιοθήκης Readline πρέπει να συμπεριλάβουν το αρχείο επικεφαλίδας <readline.h>. Παρομοίως, για να γίνει χρήση της βιβλιοθήκης History σε μία εφαρμογή πρέπει να συμπεριληφθεί το αρχείο <history.h>. Πολλές φορές τα αρχεία αυτά

βρίσκονται στο υποκαταλογο readline. Για την μεταγλώττιση του πηγαίου κώδικα πρέπει να χρησιμοποιηθεί η επιλογή –lreadline.

3.2.2 Η βιβλιοθήκη ncurses

Η βιβλιοθήκη **ncurses** χρησιμοποιήθηκε για την παραθυρική έκδοση του κελύφους. Η βιβλιοθήκη ncurses είναι μία πολύ σημαντική βιβλιοθήκη η οποία προσφέρει την δυνατότητα να δημιουργήσουμε γραφικές εφαρμογές μέσα από περιβάλλον γραμμής εντολών. Στην ουσία είναι η μέση λύση ανάπτυξης εφαρμογών γραμμής εντολών και πλήρως γραφικών εφαρμογών όπως είναι οι εφαρμογές X Windows System, **GTK**/GNOME και **Qt**/KDE. Η βιβλιοθήκη ncurses είναι στην ουσία μία νέα έκδοση της βιβλιοθήκης curses η οποία εμφανίστηκε για πρώτη φορά στο **BSD Unix** και αργότερα ενσωματώθηκε στις εκδόσεις System V του Unix λίγο πριν αποτελέσει επίσημα μέρος της τυποποίησης X/Open. Το Linux χρησιμοποιεί την έκδοση ncurses (new curses) η οποία είναι πλήρως μεταφέρσιμη σε άλλες εκδόσεις του Unix αν και τελευταία υπήρξαν κάποιες προσθήκες οι οποίες δεν είναι μεταφέρσιμες. Υπάρχουν επίσης εκδόσεις της βιβλιοθήκης curses και για DOS και Windows. Η βιβλιοθήκη ncurses δημιουργήθηκε και συντηρείται από το έργο GNU και κάποιος μπορεί να την προμηθευτεί από το http://ftp.gnu.org/pub/gnu/ncurses/.

Η τυποποίηση X/Open καθορίζει δύο επίπεδα για την βιβλιοθήκη curses. Το βασικό και το επεκτεταμένο επίπεδο. Η επεκτεταμένη έκδοση της βιβλιοθήκης περιέχει ένα σύνολο συμπληρωματικών συναρτήσεων για την διαχείριση χαρακτήρων που καταλαμβάνουν πολλαπλές στήλες καθώς και για την διαχείριση χρωμάτων. Η βασική έκδοση περιέχει συναρτήσεις για την διαχείριση της οθόνης, την δημιουργία παραθύρων και υποπαραθύρων, την διαχείριση εισόδου από το πληκτρολόγιο κλπ.

Για να μπορέσουμε να χρησιμοποιήσουμε την βιβλιοθήκη ncurses πρέπει να συμπεριλάβουμε στον πηγαίο κώδικα του προγράμματος μας το αρχείο επικεφαλίδας <curses.h> ή το αρχείο επικεφαλίδας <ncurses.h>. Στο Linux το αρχείο curses.h είναι συνήθως σύνδεσμος για το αρχείο ncurses.h. Τέλος για την επιτυχημένη μεταγλώττιση πρέπει να συμπεριληφθεί και η επιλογή –lnurses. Για παράδειγμα αν έχουμε ένα πηγαίο πρόγραμμα με όνομα program.cpp το οποίο κάνει χρήση της βιβλιοθήκης ncurses και θέλουμε να το μεταγλωττίσουμε με τον μεταγλωττιστή g++ πρέπει να δώσουμε την εντολή:

```
$ g++ program.cpp –o program -lncurses
```

3.2.3 Η βιβλιοθήκη STL

Η βιβλιοθήκη **STL** (Standard Template Library) της C++ είναι η καθιερωμένη βιβλιοθήκη της γλώσσας η οποία δεν λείπει πλέον από καμία ANSI υλοποίηση του μεταγλωττιστή. Η βιβλιοθήκη STL είναι ένα σύνολο από έτοιμες κλάσεις και συναρτήσεις γενικής χρήσης που γλιτώνουν σημαντικό χρόνο από το προγραμματιστή καθώς προσφέρουν έναν εύκολο τρόπο να

αντιμετωπιστούν τα περισσότερα θέματα γενικής φύσεως σε ένα πρόγραμμα. Έτσι ο προγραμματιστής έχει πλέον τη δυνατότητα να ασχοληθεί με την ουσία του προβλήματός του, παρά με το ποια ρουτίνα για παράδειγμα θα χρησιμοποιήσει για να ταξινομήσει ένα πίνακα από strings. Η βιβλιοθήκη STL είναι κατά το μεγαλύτερο τμήμα της δημιούργημα ενός μόνο ανθρώπου, του Alexander Stepanov ωστόσο οι κατασκευαστές μεταγλωττιστών της C++ περιλαμβάνουν δικές τους εκδόσεις της βιβλιοθήκης οι οποίες όμως έχουν σαν βάση την αρχική υλοποίηση της από τον Stepanov.

Γενικά η STL αποτελείται από τρεις κατηγορίες κλάσεων και μεθόδων: τους **containers**, τους **αλγόριθμους** και τους **iterators**. Αυτά συνδέονται μεταξύ τους με τέτοιο τρόπο ώστε να επιτρέπεται η εύκολη αντιμετώπιση μιας πληθώρας προβλημάτων των οποίων η επίλυση είναι απαραίτητη για τον προγραμματιστή αλλά είναι απλώς λεπτομέρειες στην συνολική εικόνα ενός προγράμματος (π.χ. ταξινόμηση στοιχείων).Οι κατηγορίες της βιβλιοθήκης ορίζονται σε επικεφαλίδες χωρίς το επίθεμα .h. Για να μην υπάρχει πιθανότητα τα ονόματα που ορίζονται στις επικεφαλίδες να ταυτίζονται με ονόματα που ορίζει ο χρήστης, τα ονόματα που ορίζουν οι επικεφαλίδες αυτές βρίσκονται απομονωμένα σε έναν ξεχωριστό χώρο ονομάτων (**namespace**) με το πρόθεμα "std::". Για να τα χρησιμοποιήσουμε στο πρόγραμμά μας χωρίς το πρόθεμα αυτό μπορούμε μετά τις επικεφαλίδες να δηλώσουμε:

```
using namespace std;
```

Οι containers (περιέχοντες) είναι δομές με κατάλληλα χαρακτηριστικά για την αποθήκευση και διαχείριση δεδομένων, η κάθε μία με διαφορετικές ιδιότητες. Υποκαθιστούν τους ενσωματωμένους πίνακες και επεκτείνουν σημαντικά τις περιορισμένες δυνατότητες που έχουν αυτοί. Μεταξύ άλλων περιλαμβάνονται περιέχοντες που παρέχουν αυτόματη ταξινόμηση (π.χ. set , map) και ταχύτατη ανάκτηση δεδομένων είτε με ακέραιο αριθμητικό δείκτη (π.χ. vector , deque) είτε με δείκτη οποιουδήποτε τύπου (π.χ. map). Παρακάτω παραθέτουμε όλους τους περιέχοντες που υποστηρίζει η βιβλιοθήκη STL:

Όνομα περιέχοντα	Επεξήγηση
list	διπλά συνδεδεμένη λίστα
queue	ουρά
deque	ουρά με πρόσβαση και στις δύο άκρες
map	απεικόνιση (π.χ. από συμβολοσειρές σε ακέραιους)
set	απεικόνιση με μοναδικά στοιχεία στο πεδίο τιμών
stack	στοίβα
vector	πίνακας

Πίνακας 3.1 – Οι περιέχοντες κατηγορίες της βιβλιοθήκης STL

Οι iterators (επαναλήπτες) είναι ένα είδος δείκτη (pointer) σε θέσεις στοιχείων ενός περιέχοντα. Οι επαναλήπτες έχουν την ίδια μορφή για όλους τους περιέχοντες με αποτέλεσμα να παρέχουν συγκεκριμένο, ενιαίο τρόπο για

τη διαχείρισή τους. Μπορούμε να διατρέξουμε ένα περιέχοντα ή να προσπελάσουμε ένα στοιχείο του ανεξάρτητα από το πώς γίνεται σε χαμηλό επίπεδο η οργάνωση των δεδομένων σε αυτόν. Οι επαναλήπτες, ανάλογα με τη δομή των δεδομένων του περιέχοντα, μπορούν να επιτρέπουν την παρακάτω ιεραρχία κινήσεων:

- Τυχαία προσπέλαση
- Κίνηση προς τις δύο κατευθύνσεις
- Κίνηση προς τα εμπρός, ή ανάποδη κίνηση

Επίσης, μπορούν να επιτρέπουν την παρακάτω ιεραρχία πρόσβασης στα δεδομένα που δείχνουν:

- Είσοδο και έξοδο
- Μόνο είσοδο, ή μόνο έξοδο

Οι περιέχοντες από μόνοι τους δεν θα ήταν ιδιαίτερα χρήσιμοι, όχι περισσότερο από την ξεχωριστή υλοποίηση κάποιου προγραμματιστή. Αυτό που κάνει την STL να αποδεικνύεται αξεπέραστο εργαλείο είναι ο συνδυασμός των περιεχόντων με τους αλγόριθμους (algorithms). Οι αλγόριθμοι υλοποιούν με πολύ αποτελεσματικό τρόπο συνήθη τμήματα κώδικα όπως ταξινόμηση και εύρεση ή αντικατάσταση στοιχείου με συγκεκριμένη τιμή, ανεξάρτητα από τον περιέχοντα που χρησιμοποιείται για την αποθήκευση. Αυτό επιτυγχάνεται με τη χρήση των επαναληπτών. Η πλειοψηφία των αλγορίθμων περιλαμβάνονται στο αρχείο επικεφαλίδας <algorithm>. Μερικοί αλγόριθμοι που υποστηρίζονται από την βιβλιοθήκη STL είναι:

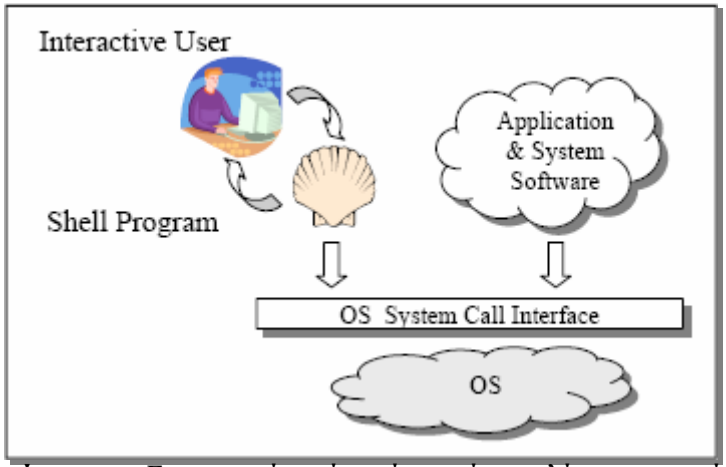
Όνομα αλγόριθμου	Επεξήγηση
adjacent_find	βρίσκει δύο ίσα στοιχεία σε διπλανές θέσεις
binary_search	δυαδική αναζήτηση
copy	αντιγραφή περιοχής
copy_backward	αντίστροφη αντιγραφή περιοχής
count	μέτρημα
count_if	μέτρημα υπό συνθήκη
equal	σύγκριση περιοχών
find	εύρεση στοιχείου
find_end	εύρεση στοιχείου από το τέλος
unique	αφαιρεί τα όμοια στοιχεία από μια περιοχή

Πίνακας 3.2 – Ορισμένοι αλγόριθμοι της βιβλιοθήκης STL

3.3 ■ Η απλή υλοποίηση του κελύφους (Βασική λειτουργία)

Όλες οι εφαρμογές χρήστη χρειάζονται πολλές φορές να κάνουν χρήση μιας υπηρεσίας του ΛΣ. Ο τρόπος για να επικοινωνήσουν με το ΛΣ όπως αναφέραμε εκτενώς και στο πρώτο κεφάλαιο είναι μέσω κλήσεων συστήματος όπως η fork(). Παρομοίως οι χρήστες ενός υπολογιστικού συστήματος

χρειάζονται ένα τρόπο για να επικοινωνήσουν με το ΛΣ ώστε να εκτελέσουν ένα πρόγραμμα, να περιεργαστούν τα αρχεία τους κλπ. Ένας βασικός τρόπος που παρέχει το ΛΣ για την επικοινωνία αυτή είναι το κέλυφος (**shell**) ή διερμηνέας γραμμής εντολών (**command line interpreter**).



Εικόνα 3.1 – Επικοινωνία ενός χρήστη μέσω κελύφους εργασίας

Ο διερμηνέας γραμμής εντολών (κέλυφος εργασίας) είναι μία εφαρμογή γραμμής εντολών και αποτελεί την διεπαφή του χρήστη με το Λειτουργικό Σύστημα. Μέσω του διερμηνέα γραμμής εντολών, ο χρήστης μπορεί να εκτελέσει προγράμματα δίνοντας στην γραμμή εντολών το όνομα του προγράμματος ή της εντολής. Το κέλυφος πρέπει να είναι έτσι υλοποιημένο που να εκτελεί οποιοδήποτε πρόγραμμα, ακόμα και αν το πρόγραμμα που εκτελείται είναι προβληματικό (buggy). Στην περίπτωση αυτή η εκτέλεση του κελύφους δεν πρέπει να διακοπεί ακόμα και αν το Λειτουργικό Σύστημα αποστείλει σήμα τερματισμού το πρόγραμμα που εκτελείται. Αφού το Λειτουργικό Σύστημα τερματίσει το πρόγραμμα, το κέλυφος πρέπει να είναι σε θέση να εκτελέσει το νέο πρόγραμμα που θα δώσει ο χρήστης. Με άλλα λόγια, το κέλυφος απομονώνει την εκτέλεση του από την εκτέλεση των θυγατρικών διεργασιών του. Επί πλέον το κέλυφος θα πρέπει να προστατεύει την εκτέλεσή του από πιθανή προσπάθεια του χρήστη να τερματίσει κατά λάθος το κέλυφος με χρήση σημάτων τερματισμού.

Οι κατασκευαστές του Unix (D.Richie και K.Thompson, 1974) ήταν οι πρώτοι που υιοθέτησαν την τεχνική αυτή για την υλοποίηση ενός διερμηνέα γραμμής εντολών. Αυτοί ήταν που έδωσαν στον διερμηνέα γραμμής εντολών το όνομα κέλυφος (**shell**). Το όνομα αυτό έμεινε στον χρόνο και στο Unix χρησιμοποιείται πολύ συχνά στην θέση του διερμηνέα γραμμής εντολών. Στο ΛΣ Windows πολύ σπάνια αναφέρεται σαν κέλυφος. Η έμπνευση για το όνομα "κέλυφος" ήταν το γεγονός ότι ο διερμηνέας γραμμής εντολών παρέχει προστασία στο ΛΣ όπως το κέλυφος παρέχει προστασία στο στρείδι. Το πρώτο κέλυφος του Unix το έγραψε ο ίδιος ο Thompson και ήταν ένα σχετικά απλό κέλυφος το οποίο έδινε την δυνατότητα εκτέλεσης απλών εντολών και δεν ενσωμάτωνε δυνατότητες προγραμματισμού στο κέλυφος. Την ίδια χρονική περίοδο ο Steve Bourne στο ίδιο εργαστήριο (Bell Laboratories) έγραψε ένα ακόμα κέλυφος για το Unix το οποίο ήταν αρκετά πιο εξελιγμένο. Επέτρεπε

την δυνατότητα προγραμματισμού και εκτέλεσης αρχείων δέσμης. Τα δύο αυτά κελύφη δεν ήταν συμβατά μεταξύ τους και για ένα μεγάλο χρονικό διάστημα συνυπήρχαν στο ίδιο εργαστήριο. Το γεγονός ότι υπήρξαν δύο κελύφη ταυτόχρονα οδήγησε στην ανάγκη καθιέρωσης ενός κελύφους εργασίας. Τελικά το κέλυφος που υπερίσχυσε και καθιερώθηκε ήταν το κέλυφος Bourne. Το κέλυφος αυτό είναι δημοφιλές ακόμα και σήμερα και είναι γνωστό και απλά ως sh. Αργότερα υλοποιήθηκαν και διάφορα άλλα κελύφη από πολλά πανεπιστήμια και εργαστήρια.

Ένα τυπικό κέλυφος εργασίας λειτουργεί ως εξής. Ο χρήστης αλληλεπιδρά με το ΛΣ πληκτρολογώντας αλφαριθμητικά στο κέλυφος. Η εντολή επεξεργάζεται από το κέλυφος όταν ο χρήστης πατήσει το πλήκτρο Enter. Το κέλυφος βασίζει την λειτουργία του σε μία σημαντική σύμβαση για την πραγματοποίηση αυτής της διαδικασίας. Το πρώτο λεκτικό σύμβολο της εντολής είναι συνήθως το όνομα του αρχείου που περιέχει το εκτελέσιμο πρόγραμμα. Για παράδειγμα ls και gcc είναι ονόματα αρχείων. Στην περίπτωση αυτή λέμε ότι η εντολή είναι **εξωτερική**. Υπάρχουν και περιπτώσεις που το πρώτο λεκτικό σύμβολο δεν είναι όνομα αρχείου αλλά είναι **εσωτερική** ή **ενσωματωμένη** εντολή του κελύφους όπως για παράδειγμα η εντολή cd (change directory) η οποία αλλάζει το τρέχοντα κατάλογο. Για τις εσωτερικές και τις εξωτερικές εντολές θα ασχοληθούμε σε επόμενη ενότητα. Στην περίπτωση που η εντολή είναι εξωτερική, το κέλυφος ψάχνει να εντοπίσει το αρχείο στους καταλόγους που έχουν καθοριστεί την μεταβλητή περιβάλλοντος **PATH**. Αν το αρχείο δεν βρεθεί στους καταλόγους που καθορίζει η μεταβλητή PATH, ορισμένα κελύφη ψάχνουν και στον τρέχοντα κατάλογο. Αν το αρχείο δεν βρεθεί και πάλι τότε εμφανίζει ένα μήνυμα λάθους. Η μόνη περίπτωση στην οποία το κέλυφος δεν θα ψάξει στο καταλόγους που καθορίζονται στην μεταβλητή PATH είναι η περίπτωση στην οποία ο χρήστης παρέχει την πλήρη διαδρομή του αρχείου εκτός από το όνομά του.

Το κέλυφος πρέπει να είναι σε θέση να εκτελεί απλές εντολές όπως:

```
ls
```

η οποία εμφανίζει στην οθόνη την λίστα με τα αρχεία του τρέχοντος καταλόγου, αλλά και πιο σύνθετες εντολές όπως:

```
ls -1 | sort > output
```

Η σύνταξη μιας γραμμής εντολών είναι απλή. Όταν το κέλυφος διαβάσει μία εντολή όπως:

```
a.out foo 100
```

θα πραγματοποιήσει λεκτική ανάλυση ώστε να βρει τα λεκτικά σύμβολα της εντολής και ύστερα συντακτική ανάλυση ώστε να διαπιστώσει την συντακτική δομή της εντολής. Για παράδειγμα η παραπάνω εντολή έχει την μορφή:

```
filename argument argument
```

η οποία αποτελεί τη συνηθέστερη μορφή εντολής. Ένα άλλο παράδειγμα είναι η εντολή:

```
ls -1 | sort > output
```

η οποία έχει την μορφή:

```
filename argument pipe filename redirection filename
```

Αφού αναλύσει την γραμμή εντολών, το κέλυφος εκτελεί την εντολή σύμφωνα με την δομή της. Στο τελευταίο παράδειγμα θα πρέπει πρώτα να δημιουργήσει μια θυγατρική διεργασία στην οποία θα εκτελεστεί το πρόγραμμα ls με παράμετρο -1. Έπειτα θα διοχετεύσει την έξοδο της στην είσοδο του προγράμματος sort για το οποίο θα πρέπει να το θέσει υπό εκτέλεση δημιουργώντας μία ακόμα διεργασία. Τέλος θα πρέπει να ανακατευθύνει την έξοδο του προγράμματος sort στο αρχείο με όνομα output. Αν το αρχείο output υπάρχει θα πρέπει να σβηστούν τα περιεχόμενά του και να θέσει σαν νέα περιεχόμενα του αρχείου, την έξοδο του προγράμματος sort. Αν το αρχείο output δεν υπάρχει θα πρέπει το κέλυφος να το δημιουργήσει.

Το κέλυφος ειδοποιεί τον χρήστη ότι είναι έτοιμο να δεχτεί είσοδο εμφανίζοντας μία προτροπή (**prompt**) στην πρώτες αριστερές θέσεις της γραμμής. Η προτροπή είναι ένα σύμβολο συνήθως το $ ή το % για τον απλό χρήστη και το # για τον χρήστη root. Η προτροπή συνήθως μπορεί να ρυθμιστεί κατάλληλα ώστε να εμφανίζει και το όνομα του χρήστη, το όνομα του μηχανήματος αλλά και το όνομα του τρέχοντος καταλόγου. Η προτροπή εμφανίζεται αμέσως αφού φορτωθεί το κέλυφος.

Επίσης το κέλυφος επιτρέπει την διαχείριση των διεργασιών. Επιτρέπει την εκτέλεση διεργασιών στο παρασκήνιο και ο χρήστης έχει την δυνατότητα να φέρει στο προσκήνιο ανά πάσα στιγμή όποια διεργασία παρασκηνίου επιθυμεί. Ωστόσο ανά μία δεδομένη στιγμή, μόνο μία διεργασία μπορεί να είναι στο προσκήνιο. Μία διεργασία λέμε ότι βρίσκεται στο παρασκήνιο όταν το κέλυφος δεν περιμένει να τερματιστεί αλλά αντιθέτως αφίνει την διεργασία να εκτελείται και δίνει στον χρήστη την δυνατότητα να εκτελέσει άλλα προγράμματα, ενώ λέμε ότι μία διεργασία εκτελείται στο προσκήνιο όταν το κέλυφος δεν συνεχίζει την εκτέλεσή του όσο η θυγατρική διεργασία εκτελείται. Έτσι το κέλυφος δεν δίνει την δυνατότητα να εκτελέσει άλλο πρόγραμμα όσο η διεργασία εκτελείται. Επί πλέον ο χρήστης έχει την δυνατότητα να τερματίσει οποιαδήποτε διεργασία έχει εκκινήσει ο ίδιος. Για να θέσει μία διεργασία στο παρασκήνιο ο χρήστης πληκτρολογεί το χαρακτήρα & στο τέλος της εντολής. Για παράδειγμα η εντολή:

```
ls -1 | sort > output &
```

θα εκτελεστεί στο παρασκήνιο. Όταν τελειώσει η εκτέλεσή της, ο χρήστης θα ειδοποιηθεί από το κέλυφος.

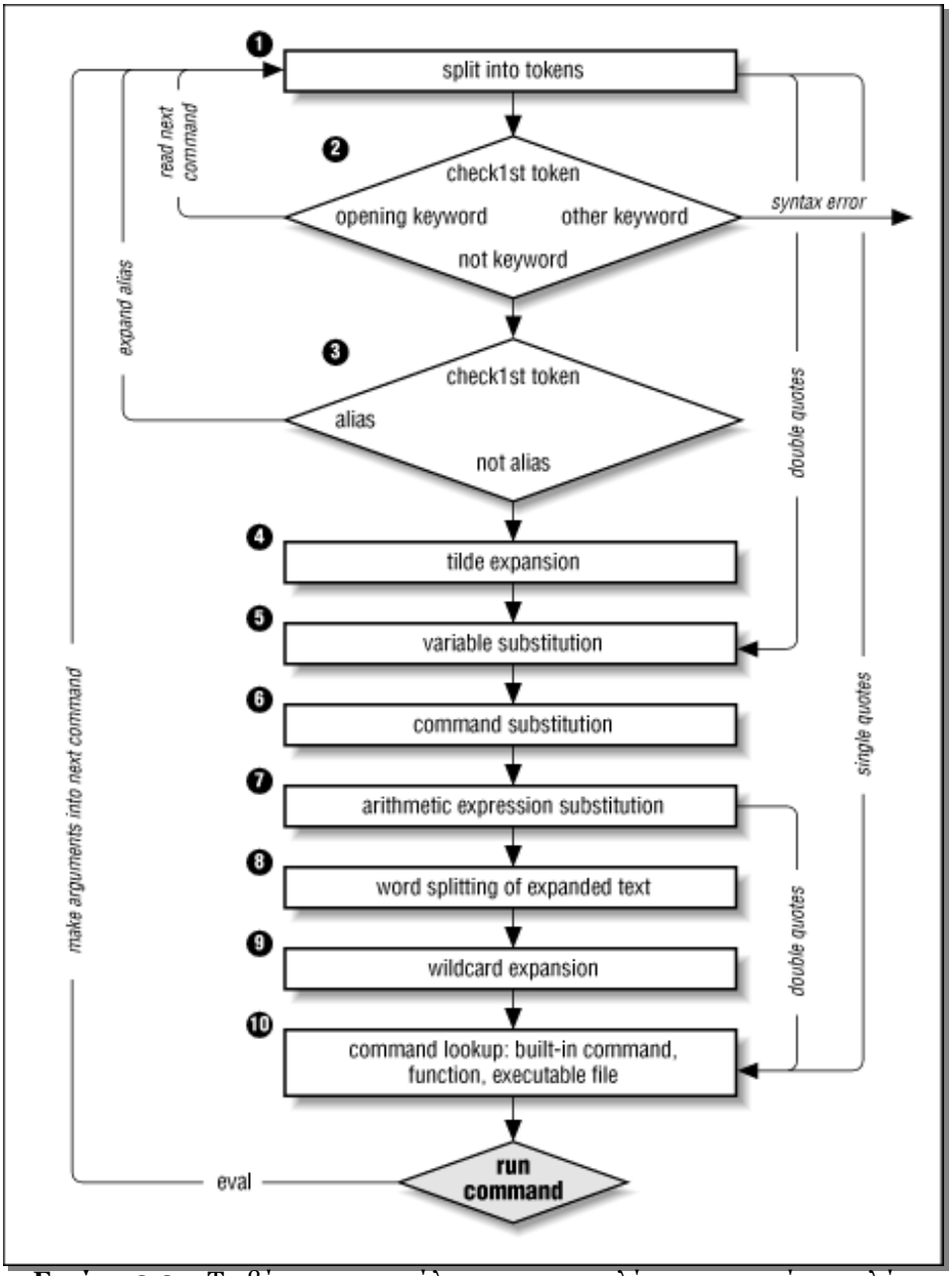
Μία ακόμη δυνατότητα που διαθέτουν αρκετά κελύφη είναι η δυνατότητα να κρατάνε ιστορικό των εντολών που εισάγει ο χρήστης και να προσφέρουν την δυνατότητα επεξεργασίας τους και άμεσης εκτέλεσής τους χρησιμοποιώντας τα πλήκτρα βελάκια (arrow keys) του πληκτρολογίου.

Πολλές φορές είναι χρήσιμο ο χρήστης να πληκτρολογεί ένα ψευδώνυμο (**alias**) μιας εντολής αντί για την εντολή την ίδια. Με τον τρόπο αυτό για παράδειγμα επιταχύνεται η εκτέλεση μίας εντολής η οποία αποτελείται από πολλούς χαρακτήρες και χρησιμοποιείται συχνά.

Τέλος θα τα πιο εξελιγμένα κελύφη εργασίας επιτρέπουν την δυνατότητα δημιουργίας μεταβλητών οι οποίες σε συνδυασμό με τις κατάλληλες δομές ελέγχου (if) και δομές επανάληψης (for, while) δίνουν τη δυνατότητα στον χρήστη για την δημιουργία αρχείων δέσμης με τα οποία αυτοματοποιεί αρκετές διαδικασίες. Η συγγραφή αυτών των αρχείων ονομάζεται προγραμματισμός κελύφους (**shell programming**).

Όπως αναφέραμε και προηγουμένως, πριν εκτελεστεί μία εντολή που δίνει ο χρήστης στην γραμμή εντολών, πρέπει να αναλυθεί κατάλληλα ώστε να διαπιστωθούν τα λεκτικά σύμβολα που την αποτελούν αλλά και η συντακτική δομή της. Σε περίπτωση που ανακαλυφθεί μία μεταβλητή του κελύφους, θα πρέπει να αντικατασταθεί με την τιμή της. Επίσης, αν ανακαλυφθεί ένα ψευδώνυμο μιας εντολής πρέπει να αντικατασταθεί με την αρχική εντολή.

Γενικά η όλη διαδικασία που υπόκειται μία εντολή πριν την εκτέλεσή της, περιγράφεται αναλυτικά από το διάγραμμα ροής της εικόνας 3.2.



Εικόνα 3.2 – Τα βήματα της ανάλυσης μιας εντολής της γραμμής εντολών

3.4 ■ Αρχεία εκκινήσεως του κελύφους

Για να κατανοήσουμε τα αρχεία εκκινήσεως ενός κελύφους πρέπει γνωρίζουμε την διάκριση ενός **κελύφους εκκίνησης (login shell)** και ενός κελύφους μη-εκκίνησης (non-login shell). Όταν κάνουμε σύνδεση σε ένα σύστημα Unix το πρόγραμμα login εκτελεί ένα κέλυφος για εμάς. Κάθε χρήστης μπορεί να επιλέξει το κέλυφος που θέλει να εκτελείται κατά την διαδικασία της σύνδεσης. Από το σημείο αυτό το κέλυφος είναι έτοιμο για να δεχτεί εντολές από τον χρήστη. Ο χρήστης μπορεί να εκκινήσει και νέα κελύφη τα οποία όμως δεν θα είναι κελύφη εκκίνησης. Για παράδειγμα, αν χρησιμοποιήσει το πρόγραμμα ftp θα εκκινηθεί ένα κέλυφος το οποίο όμως δεν

θα είναι κέλυφος εκκίνησης. Το κέλυφος αυτό ονομάζεται **κέλυφος μη-εκκίνησης (non-login shell)** ή **υποκέλυφος (subshell)**.

Ένα κέλυφος εκκίνησης πραγματοποιεί κάποιες ενέργειες τις οποίες δεν εκτελεί ένα υποκέλυφος. Οι ενέργειες αυτές αφορούν συνήθως την ρύθμιση του τερματικού και δεν χρειάζεται να γίνονται κάθε φορά που ένα κέλυφος εκκινείται στο ίδιο τερματικό. Για αυτό το λόγο γίνονται μόνο μία φορά κατά την φόρτωση του κελύφους εκκίνησης. Το ποιες ενέργειες θα εκτελεστούν κατά την φόρτωση του κελύφους εκκίνησης βρίσκονται σε ένα αρχείο το οποίο διαβάζεται από το κέλυφος κατά την εκκίνησή του. Το αρχείο αυτό δεν έχει κάποιο συγκεκριμένο όνομα δηλαδή ποικίλει ανάλογα με το κέλυφος που χρησιμοποιούμε. Επιπλέον υπάρχουν και άλλα αρχεία τα οποία τα διαβάζει το κέλυφος κατά την εκκίνησή του. Αυτά τα αρχεία συνήθως διαβάζονται κάθε φορά που εκκινείται ένα κέλυφος και δεν έχει σχέση με το αν το κέλυφος είναι κέλυφος εκκίνησης ή είναι υποκέλυφος. Παρομοίως τα αρχεία έχουν συνήθως διαφορετικό όνομα σε κάθε κέλυφος.

3.4.1 Γνωστά κελύφη και τα αρχεία εκκινήσεώς τους

Παρακάτω θα δούμε μερικά από τα πιο γνωστά κελύφη και τα αρχεία εκκίνησης που χρησιμοποιούν.

Κέλυφος Bourne

Το κέλυφος Bourne χρησιμοποιεί ένα μόνο αρχείο εκκίνησης. Το αρχείο αυτό έχει όνομα *.profile* και βρίσκεται στο βασικό κατάλογο του χρήστη (home directory). Το κέλυφος Bourne διαβάζει αυτό το αρχείο μόνο όταν το κέλυφος εκτελεστεί σαν κέλυφος εκκίνησης. Σε αντίθετη περίπτωση το κέλυφος ξεκινά χωρίς να διαβάσει κάποιο αρχείο εκκίνησης. Σε αυτή την περίπτωση, οι ρυθμίσεις για το κέλυφος προέρχονται από τις μεταβλητές περιβάλλοντος. Ο δημιουργός του κελύφους Bourne είναι ο Steve Bourne.

Κέλυφος Korn

Το κέλυφος Korn μοιάζει αρκετά με το κέλυφος Bourne. Αν το κέλυφος Korn εκτελεστεί σαν κέλυφος εκκίνησης τότε διαβάζεται το αρχείο *.profile*. Το αρχείο αυτό μπορεί να θέσει την μεταβλητή περιβάλλοντος ENV και να την αποθηκεύσει στο αρχείο *.kshrc* το οποίο το βρίσκεται στο βασικό φάκελο του χρήστη. Έπειτα, κάθε υποκέλυφος Korn θα διαβάζει το αρχείο *.kshrc* κατά την εκκίνησή του. Το κέλυφος Korn δημιουργήθηκε από τον David Korn στα εργαστήρια AT&T και είναι διαθέσιμο από το http://www.kornshell.com.

Κέλυφος Bash

Το κέλυφος Bash διαθέτει δύο αρχεία εκκίνησης αλλά διαβάζει μόνο ένα ανάλογα με το αν είναι κέλυφος εκκίνησης. Επιπλέον διαθέτει και ένα αρχείο το οποίο το διαβάζει όταν ο χρήστης αποσυνδεθεί από το σύστημα. Κάθε φορά που εκκινείται ένα κέλυφος εκκίνησης Bash, το αρχείο *.profile* ή το αρχείο *.bash_profile* ή το αρχείο *.bash_login*. Όταν ο χρήστης αποσυνδεθεί από το σύστημα, το κέλυφος Bash διαβάζει το αρχείο *.bash_logout*. Σε περίπτωση που

το κέλυφος δεν είναι εκκίνησης, τότε διαβάζεται το αρχείο *.bashrc* και όταν ο χρήστης αποσυνδεθεί δεν διαβάζεται το αρχείο *.bash_logout*. Το κέλυφος bash δημιουργήθηκε από το έργο GNU και είναι διαθέσιμο από το http://ftp.gnu.org/pub/gnu/bash/.

Να σημειώσουμε ότι τα αρχεία που αρχίζουν με μία τελεία, στο Unix είναι κρυφά αρχεία και πρέπει να χρησιμοποιήσουμε την εντολή ls –a για να τα δούμε. Επίσης, το επίθεμα που χρησιμοποιείται στα αρχεία .bashrc και .kshrc σημαίνει run commands (εκτέλεση εντολών).

3.4.2 Τα περιεχόμενα των αρχείων εκκινήσεως

Τα αρχεία εκκινήσεως όπως τα .login και .profile συνήθως περιλαμβάνουν τα εξής:

- Θέτουν την μεταβλητή PATH
- Θέτουν τον τύπο του τερματικού και πραγματοποιούν διάφορες ρυθμίσεις σχετικά με το τερματικό
- Θέτουν διάφορες μεταβλητές περιβάλλοντος που είναι χρήσιμες για διάφορα αρχεία δέσμης που εκτελούνται συχνά.
- Εκτελούν ένα ή περισσότερα προγράμματα που θέλει ο χρήστης να εκτελούνται αυτόματα κάθε φορά που συνδέεται στο σύστημα. Για παράδειγμα, ορισμένοι χρήστες θέλουν να εμφανίζεται, κάθε φορά που συνδέονται στο σύστημα, ένα διαφορετικό μήνυμα το οποίο είναι συνήθως διασκεδαστικό.
- Θέτουν την μορφή της προτροπής (prompt)

Τα αρχεία σαν το .bashrc χρησιμοποιούνται συνήθως για να τον καθορισμό των aliases και ρουτίνων του χρήστη.

3.4.3 Τα αρχεία εκκινήσεως του WISH

Το πειραματικό κέλυφος wish διαθέτει δύο αρχεία εκκινήσεως. Το πρώτο και πιο βασικό αρχείο είναι το αρχείο *.wish_profile* το οποίο ορίζει τις βασικές μεταβλητές περιβάλλοντος οι οποίες πρέπει να διαβαστούν από το κέλυφος για να ρυθμίσει κατάλληλα την λειτουργία του. Αν για κάποιο λόγο το αρχείο σβηστεί από τον δίσκο, το κέλυφος είναι κατάλληλα υλοποιημένο ώστε να το ξαναδημιουργήσει θέτοντας τις βασικές μεταβλητές που είναι απαραίτητες για την λειτουργία του. Ο χρήστης μπορεί να προσθέσει και τις δικές του μεταβλητές περιβάλλοντος στο αρχείο αυτό. Οι μεταβλητές αυτές θα δημιουργηθούν και θα είναι διαθέσιμες στο περιβάλλον ανά πάσα στιγμή.

Το δεύτερο αρχείο εκκίνησης του κελύφους wish είναι το αρχείο *aliases*. Το αρχείο αυτό θέτει ορισμένα aliases και ο χρήστης έχει την δυνατότητα να θέσει και τα δικά του αν θέλει. Το αρχείο αυτό φορτώνεται κατά την εκκίνηση και για αυτό το λόγο αν ο χρήστης προσθέσει στο αρχείο *aliases* ορισμένα aliases κατά την διάρκεια της εκτέλεσης του κελύφους, τότε δεν θα είναι

διαθέσιμα παρά μόνο όταν επανεκκινηθεί το κέλυφος. Παρακάτω παραθέτουμε τα περιεχόμενα του αρχείου *.wish_profile* και του αρχείου *aliases*.

```
# .wish_profile
# User specific environment variables
# format : variable value
DIR on
PS1 %
FULLPATH off
GREET off
```

Τα περιεχόμενα του αρχείου .wish_profile

```
# format for aliases file
# command : replacement
cd..  : cd ..
ls    : ls –CF
lsort : ls | sort -f
```

Τα περιεχόμενα του αρχείου aliases

3.5 ■ Μεταβλητές περιβάλλοντος

Σε προηγούμενες ενότητες έχουμε αναφερθεί συχνά στον όρο "μεταβλητή περιβάλλοντος". Για να κατανοήσουμε τις μεταβλητές περιβάλλοντος πρέπει πρώτα να κατανοήσουμε τον όρο "**περιβάλλον**", ένας όρος που δεν εξηγείται συχνά παρόλο που δεν είναι αυτονόητος για τους αρχάριους χρήστες. Η έννοια του περιβάλλοντος πρέπει να γίνει αρκετά κατανοητή γιατί αποτελεί βασικό μηχανισμό του Unix. Επίσης πρέπει να αναφέρουμε ότι διαφορετικές εκδόσεις του Unix ακόμα και διαφορετικές εκδόσεις των κελυφών συμπεριφέρονται διαφορετικά στο περιβάλλον. Ο όρος "περιβάλλον" του Unix δεν έχει καμία σχέση με άλλους όρους που συναντάμε συχνά όπως περιβάλλον προγραμματισμού "programming environment". Το περιβάλλον του Unix είναι μία λίστα με μεταβλητές και αντίστοιχες τιμές, τις οποίες χρησιμοποιούν οι θυγατρικές διεργασίες. Αυτή η λίστα έχει εγγραφές της μορφής:

```
varname=value
```

όπου varname είναι το όνομα της μεταβλητής και value είναι η τιμή της μεταβλητής. Οι μεταβλητές αυτές ονομάζονται **μεταβλητές περιβάλλοντος (environmental variables)** και μπορεί να καθορίσουν σχεδόν τα πάντα – από την μορφή της προτροπής έως το πόσες φορές να ελέγχει το κέλυφος για τα νέα mail του χρήστη. Το κέλυφος κατά την εκκίνησή του ορίζει αρκετές μεταβλητές αλλά και ο χρήστης μπορεί να ορίσει τις δικές του μεταβλητές. Οι μεταβλητές που ορίζει το ίδιο το κέλυφος ονομάζονται **ενσωματωμένες μεταβλητές (built-in variables)** και από σύμβαση είναι με κεφαλαία γράμματα ενώ οι μεταβλητές που ορίζει ο χρήστης ονομάζονται **μεταβλητές**

κελύφους (shell variables) και από σύμβαση είναι με πεζά. Το Unix και συνεπώς και το Linux κάνουν διάκριση στα πεζά και στα κεφαλαία (case sensitive) ως εκ τούτου και το κέλυφος θεωρεί τις μεταβλητές foo, Foo και FOO διαφορετικές.

Τις μεταβλητές περιβάλλοντος δεν τις δηλώνουμε όπως κάνουμε με τις μεταβλητές σε μία γλώσσα προγραμματισμού όπως η C++. Αντιθέτως τις δημιουργούμε απλώς με το να τις αναθέτουμε τιμή. Για παράδειγμα αν γράψουμε στο κέλυφος:

```
name=Nikos
```

τότε, αν υπάρχει η μεταβλητή name θα χαθεί η τιμή της και θα έχει νέα τιμή το αλφαριθμητικό Nikos. Αν δεν υπάρχει μεταβλητή με όνομα name, τότε θα δημιουργηθεί και θα της ανατεθεί η τιμή Nikos. Για την ανάθεση μιας τιμής σε μία μεταβλητή δεν πρέπει να υπάρχουν κενά αριστερά και δεξιά από το σύμβολο =. Αν η τιμή που θέλουμε να θέσουμε σε μία μεταβλητή περιέχει κενά πρέπει να εσωκλείσουμε την τιμή σε αποστρόφους.

Για να προσπελάσουμε την τιμή μιας μεταβλητής μέσα από μία εντολή, πρέπει στην αρχή της μεταβλητής να χρησιμοποιήσουμε το σύμβολο $. Για παράδειγμα η εντολή:

```
echo $name
```

θα τυπώσει στην οθόνη την τιμή της μεταβλητής name, δηλαδή Nikos. Αν η μεταβλητή δεν υπάρχει τότε θα τυπώσει στην οθόνη μια κενή γραμμή. Μία μεταβλητή μπορούμε να τη διαγράψουμε με την εντολή unset.

Με την εντολή env και την ενσωματωμένη εντολή set (χωρίς παράμετρο) μπορούμε να δούμε ολόκληρη την λίστα με τις μεταβλητές περιβάλλοντος. Η εντολή set υπάρχει και στο Λειτουργικό Σύστημα Windows και έχει την ίδια λειτουργία.

3.5.1 Ενσωματωμένες μεταβλητές

Όπως αναφέραμε και προηγουμένως, το κέλυφος κατά την εκκίνησή του ορίζει αρκετές μεταβλητές περιβάλλοντος. Οι μεταβλητές αυτές ονομάζονται ενσωματωμένες μεταβλητές και είναι σημαντικές για την λειτουργία του κελύφους. Οι πιο σημαντικές από αυτές είναι:

Όνομα μεταβλητής	Επεξήγηση
$HOME	Ο βασικός κατάλογος του χρήστη
$PATH	Η λίστα καταλόγων η οποία χρησιμοποιείται για την εύρεση μιας εντολής.
$PS1	Είναι το σύμβολο της προτροπής (prompt)
$TERM	Ο τύπος του τερματικού
$SHELL	Το όνομα του κελύφους που εκτελείται

Πίνακας 3.3 – Ορισμένες μεταβλητές περιβάλλοντος ενός κελύφους

3.5.2 Μεταβλητές περιβάλλοντος στο WISH

Το πειραματικό κέλυφος wish διαχειρίζεται τις μεταβλητές περιβάλλοντος με παρόμοιο τρόπο. Για να ορίσουμε μία μεταβλητή απλά της αναθέτουμε τιμή με την διαφορά ότι δεν χρειάζεται να μην υπάρχουν κενά δεξιά και αριστερά από το = όπως επιβάλουν άλλα κελύφη. Για παράδειγμα η εντολή:

```
name = Nikos
```

είναι μία έγκυρη εντολή για δημιουργία μιας μεταβλητής και ανάθεσης τιμής σε αυτήν.

Τέλος το κέλυφος wish κατά την εκκίνησή του ορίζει ορισμένες μεταβλητές περιβάλλοντος οι οποίες βρίσκονται στο αρχείο .wish_profile. Οι ενσωματωμένες αυτές μεταβλητές είναι:

Όνομα μεταβλητής	Επεξήγηση
$DIR	Ορίζουμε την τιμή on στην μεταβλητή αυτή αν θέλουμε να εμφανίζεται στην προτροπή ο τρέχων κατάλογος
$FULLPATH	Ορίζουμε την τιμή on στην μεταβλητή αυτή αν θέλουμε να εμφανίζεται στην προτροπή η πλήρης διαδρομή του τρέχοντος καταλόγου
$PS1	Είναι το σύμβολο της προτροπής (prompt)
$GREET	Ορίζουμε την τιμή on στην μεταβλητή αυτή αν θέλουμε να εμφανίζει το όνομα και την έκδοση του κελύφους αμέσως μετά την εκκίνησή του.
$SHELL	Το όνομα του κελύφους, δηλαδή wish.

Πίνακας 3.4 – Ορισμένες μεταβλητές περιβάλλοντος του κελύφους wish

3.6 ■ Ενσωματωμένες και εξωτερικές εντολές

Όπως αναφέραμε και σε προηγούμενη ενότητα, όταν ο χρήστης δίνει μία εντολή στο κέλυφος και πατάει το πλήκτρο Enter, το κέλυφος ψάχνει να εντοπίσει το αρχείο του οποίου το όνομα είναι το ίδιο με το πρώτο λεκτικό σύμβολο της εντολής. Οι κατάλογοι που ψάχνει καθορίζονται από την μεταβλητή περιβάλλοντος PATH. Οι πιο συνηθισμένοι κατάλογοι στους οποίους ψάχνει το κέλυφος είναι :

- /bin: Στον κατάλογο αυτό υπάρχουν τα εκτελέσιμα προγράμματα που χρησιμοποιούνται από το σύστημα καθώς και τα προγράμματα που αφορούν την εκκίνηση του συστήματος.

- /usr/bin: Στον κατάλογο αυτό υπάρχουν τα πιο συνηθισμένα εκτελέσιμα προγράμματα όλων των χρηστών.

- /usr/local/bin: Στον κατάλογο αυτό υπάρχουν τα τοπικά προγράμματα που αφορούν συγκεκριμένες εγκαταστάσεις

Η τεχνική αυτή της χρήσης της μεταβλητής PATH χρησιμοποιείται εκτός από το ΛΣ Unix και στο ΛΣ Windows με την διαφορά ότι οι κατάλογοι διαχωρίζονται μεταξύ τους με `;` και όχι με `:` όπως είναι στο Unix. Για παράδειγμα μία τιμή της μεταβλητής PATH σε ένα σύστημα Unix θα μπορούσε να είναι:

```
/usr/local/bin:/bin:/usr/bin:.:/home/nikos/bin:/usr/X11R6/bin
```

Αφού εξετάσει όλες τις διαδρομές που καθορίζει η μεταβλητή PATH και δεν εντοπίσει το αρχείο τότε το κέλυφος ελέγχει την περίπτωση να είναι **ενσωματωμένη εντολή (built-in command)**. Οι ενσωματωμένες εντολές δεν είναι ξεχωριστά αρχεία του συστήματος αλλά είναι υλοποιημένες στο ίδιο το κέλυφος. Μία εντολή μπορεί να είναι ενσωματωμένη στο κέλυφος γιατί η λειτουργία της έχει άμεση σχέση με τη λειτουργία του κελύφους (πχ jobs, list) ή γιατί είναι πολύ απλή στην υλοποίησή της και χρησιμοποιείται πολύ συχνά από τους χρήστες (πχ cd). Συνηθισμένες ενσωματωμένες εντολές ενός κελύφους είναι:

Όνομα εντολής	Επεξήγηση
set	Εμφάνιση όλων των μεταβλητών περιβάλλοντος
cd	Αλλαγή τρέχοντος καταλόγου
echo	Εμφανίζει στην οθόνη το αλφαριθμητικό που δέχεται σαν παράμετρο
history	Εμφανίζει μία λίστα με προηγούμενες εντολές που έχουν δοθεί από τον χρήστη
jobs	Εμφανίζει μία λίστα με τις διεργασίες που εκτελούνται ταυτόχρονα στο παρασκήνιο
fg	Φέρνει μία διεργασία στο προσκήνιο
bg	Πάει μία διεργασία στο παρασκήνιο
alias	Θέτει ένα ψευδώνυμο σε μία εντολή
unalias	Διαγράφει το ψευδώνυμο μιας εντολής
aliases	Εμφανίζει μία λίστα με όλα τα ψευδώνυμα που έχουν οριστεί
popd	Προσθέτει τον τρέχων κατάλογο στην στοίβα καταλόγων
pushd	Αφαιρεί από την στοίβα καταλόγων τον κατάλογο που βρίσκεται στην κορυφή και θέτει τον τρέχων κατάλογο.
dirs	Εμφανίζει μία λίστα με όλους τους καταλόγους που βρίσκονται στην στοίβα καταλόγων.
exec	Εκτελεί ένα πρόγραμμα στην ίδια διεργασία του κελύφους. Δηλαδή δεν δημιουργεί νέα διεργασία με αποτέλεσμα να σταματάει η εκτέλεση του κελύφους μετά το τέλος της εκτέλεσης του προγράμματος.

Πίνακας 3.5 – Ορισμένες ενσωματωμένες εντολές ενός κελύφους

Τα πιο εξελιγμένα κελύφη τα οποία διαθέτουν δυνατότητες προγραμματισμού θεωρούν τις διάφορες δομές ελέγχου και επανάληψης ως ενσωματωμένες εντολές. Συνεπώς ενσωματωμένες εντολές είναι και οι: if, for, break, while, until κλπ.

3.6.1 Ενσωματωμένες εντολές στο WISH

Το πειραματικό κέλυφος wish διαθέτει αρκετές από τις πιο βασικές ενσωματωμένες εντολές αλλά δεν διαθέτει εντολές προγραμματισμού όπως: if, for κλπ. Τα ονόματα των ενσωματωμένων εντολών είναι παρόμοια με αυτών του bash εκτός από ελάχιστες εξαιρέσεις. Παρακάτω παραθέτουμε μία λίστα με τις ενσωματωμένες εντολές που υποστηρίζει το κέλυφος wish.

Όνομα εντολής	Επεξήγηση
set	Εμφάνιση όλων των μεταβλητών περιβάλλοντος
cd	Αλλαγή τρέχοντος καταλόγου
list	Εμφανίζει μία λίστα με προηγούμενες εντολές που έχουν δοθεί από τον χρήστη
jobs	Εμφανίζει μία λίστα με τις διεργασίες που εκτελούνται ταυτόχρονα στο παρασκήνιο
fg	Φέρνει μία διεργασία στο προσκήνιο
bg	Πάει μία διεργασία στο παρασκήνιο
alias	Θέτει ένα ψευδώνυμο σε μία εντολή
unalias	Διαγράφει το ψευδώνυμο μιας εντολής
aliases	Εμφανίζει μία λίστα με όλα τα ψευδώνυμα που έχουν οριστεί
popd	Προσθέτει τον τρέχων κατάλογο στην στοίβα καταλόγων
pushd	Αφαιρεί από την στοίβα καταλόγων τον κατάλογο που βρίσκεται στην κορυφή και θέτει τον τρέχων κατάλογο.
dirstack	Εμφανίζει μία λίστα με όλους τους καταλόγους που βρίσκονται στην στοίβα καταλόγων.
!!	Εκτελεί την τελευταία εντολή της λίστας με τις προηγούμενες εντολές
!x	Εκτελεί την εντολή που βρίσκεται στην x θέση της λίστας με τις προηγούμενες εντολές.
exit, quit	Τερματίζει την λειτουργία του κελύφους

Πίνακας 3.6 – Οι ενσωματωμένες εντολές του κελύφους wish

3.7 ■ Aliases

Πολλοί χρήστες δεν νιώθουν άνετα με τα ονόματα ορισμένων εντολών και πολλές φορές δεν είναι βολικό να πληκτρολογούμε μία μεγάλη εντολή για τις εργασίες που θέλουμε να κάνουμε συχνά. Ένας απλός τρόπος για να αλλάξουμε την συμπεριφορά του κελύφους ώστε να το προσαρμόσουμε στις ανάγκες μας είναι να ορίσουμε ψευδώνυμα (aliases). Για παράδειγμα η εντολή

rm διαγράφει ένα αρχείο. Πολλοί χρήστες δεν αισθάνονται άνετα με την εντολή αυτή γιατί δεν ζητάει επιβεβαίωση της διαγραφής με αποτέλεσμα πολλές φορές να διαγράφονται κατά λάθος αρχεία. Ο μόνος τρόπος για να ζητάει επιβεβαίωση διαγραφής η εντολή rm είναι να την καλούμε με την επιλογή –i κάθε φορά που θέλουμε να σβήσουμε ένα αρχείο. Αυτό είναι αρκετά κουραστικό αν γίνετε συχνή χρήση της εντολής. Μία λύση σε αυτό το πρόβλημα είναι να ορίσουμε ένα ψευδώνυμο στην εντολή αυτή ούτως ώστε να λειτουργεί όπως θέλουμε εμείς. Για να ορίσουμε ένα ψευδώνυμο χρησιμοποιούμε την εντολή alias. Για το παραπάνω παράδειγμα δηλαδή, στο κέλυφος bash θα πρέπει να πληκτρολογήσουμε:

```
$ alias rm=’rm –i’
```

Έτσι κάθε φορά που θα δίνουμε στο κέλυφος την εντολή rm, το κέλυφος θα εκτελεί στην ουσία την εντολή rm -i και έτσι θα ζητάει επιβεβαίωση της διαγραφής. Για να διαγράψουμε ένα ψευδώνυμο χρησιμοποιούμε την εντολή unalias. Για παράδειγμα για να διαγράψουμε το ψευδώνυμο rm που φτιάξαμε παραπάνω θα χρησιμοποιήσουμε την εντολή:

```
$ unalias rm
```

Οι παραπάνω εντολές είναι αυτές που χρησιμοποιεί το κέλυφος bash και διαφορετικά κελύφη έχουν πιθανότατα διαφορετικό τρόπο για την δημιουργία και διαγραφή ψευδώνυμων. Ένας περιορισμός που θέτουν τα κελύφη για τα ψευδώνυμα είναι να μην βρίσκονται μέσα σε αποστρόφους και να είναι η πρώτη λέξη στην εντολή που θα δώσει ο χρήστης.

3.7.1 Aliases στο κέλυφος WISH

Το πειραματικό κέλυφος wish υποστηρίζει τα ψευδώνυμα όπως ακριβώς και το κέλυφος bash αλλά χωρίς τον περιορισμό του να είναι η πρώτη λέξη της εντολής. Το κέλυφος wish αναγνωρίζει ένα ψευδώνυμο σε οποιαδήποτε θέση και αν βρίσκεται αυτό και το αντικαθιστά. Τα ονόματα των εντολών είναι τα ίδια με το bash δηλαδή: alias και unalias και επιπροσθέτως υπάρχει και η εντολή aliases με την οποία ο χρήστης βλέπει την λίστα με τα ψευδώνυμα που αναγνωρίζει το κέλυφος. Τέλος ο χρήστης έχει την δυνατότητα να προσθέσει ψευδώνυμα στο αρχείο *aliases* που βρίσκεται στον βασικό κατάλογο του χρήστη. Τα ψευδώνυμα που βρίσκονται στο αρχείο αυτό θα φορτώνονται κάθε φορά που εκκινήται το κέλυφος και θα είναι ανά πάσα στιγμή διαθέσιμα για χρήση.

3.8 ■ Shell Programming

Το κέλυφος χρησιμοποιεί μία διερμηνευόμενη γλώσσα που σημαίνει ότι ο χρήστης μπορεί να δημιουργήσει προγράμματα χρησιμοποιώντας την γλώσσα

του κελύφους και το κέλυφος μπορεί να εκτελέσει τα προγράμματα αυτά χωρίς να τα μεταγλωττίσει.

Ένας λόγος που οι χρήστες προτιμούνε να φτιάχνουν προγράμματα με το κέλυφος είναι γιατί το κέλυφος προσφέρει μεγάλη ευκολία και ταχύτητα για ανάπτυξη μικρών πρακτικών εφαρμογών. Επίσης ένα κέλυφος υπάρχει σε κάθε έκδοση του Unix και διανομή του Linux ενώ δεν είναι σίγουρο ότι θα υπάρχει πάντα μεταγλωττιστής της γλώσσας που θέλουμε.

Τα προγράμματα που δημιουργεί ένας χρήστης με το κέλυφος ονομάζονται **σενάρια κελύφους (shell scripts)**. Ένα σενάριο κελύφους είναι στην ουσία ένα σύνολο εντολών κελύφους οι οποίες εκτελούνται διαδοχικά σαν μία εντολή. Υπάρχουν δύο τρόποι συγγραφής ενός σεναρίου. Ο χρήστης μπορεί να πληκτρολογήσει στο κέλυφος μία σειρά από εντολές τις οποίες το κέλυφος θα εκτελέσει αλληλεπιδραστικά ή μπορεί να αποθηκεύσει τις εντολές σε ένα αρχείο κειμένου και έπειτα να εκτελέσει το αρχείο αυτό μέσα από το κέλυφος όπως ένα εκτελέσιμο αρχείο μόνο που το κέλυφος για να εκτελέσει το σενάριο ξεκινάει ένα υποκέλυφος (subshell) σε μία νέα διεργασία. Το υποκέλυφος είναι αυτό που θα διερμηνεύσει το σενάριο και όταν τελειώσει θα επιστρέψει τον έλεγχο πίσω στο πατρικό κέλυφος. Επίσης να αναφέρουμε οτι ένα σενάριο μπορεί να εκτελεστεί στο παρασκήνιο όπως ακριβώς και ένα πρόγραμμα.

Πριν εκτελέσουμε το σενάριο κελύφους με το όνομά του, πρέπει να του παραχωρήσουμε άδεια εκτέλεσης (**execute permission**). Όπως είναι γνωστό το σύστημα αρχείων του Unix διαθέτει τρεις τύπους αδειών (ανάγνωση, εγγραφή και εκτέλεση) και αυτές οι άδειες εφαρμόζονται σε τρεις κατηγορίες χρηστών (τον ιδιοκτήτη του αρχείου, την ομάδα των χρηστών και όλους του άλλους). Τυπικά, όταν ένας χρήστης δημιουργήσει ένα αρχείο με ένα κειμενογράφο, το αρχείο αυτό ρυθμίζεται έτσι ώστε να έχει άδεια για ανάγνωση και εγγραφή για τον ίδιο τον χρήστη και άδεια μόνο για ανάγνωση από όλους τους άλλους. Συνεπώς πρέπει να δώσουμε στο σενάριο ρητή άδεια για εκτέλεση χρησιμοποιώντας την εντολή chmod. Ο πιο απλός τρόπος για να γίνει αυτό είναι:

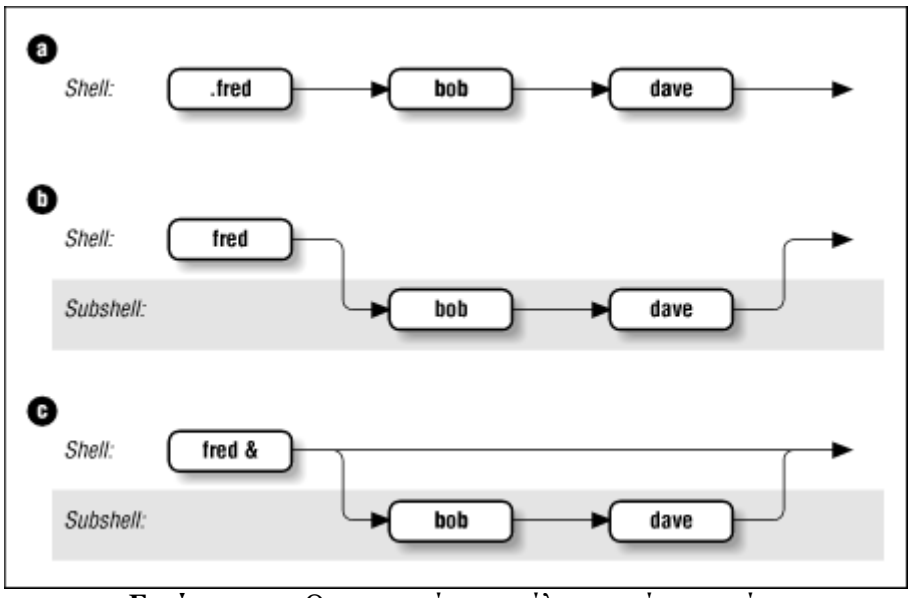
```
$ chmod +x scriptname
```

Το αρχείο θα διατηρήσει αυτή την άδεια ακόμα και αν αλλάξουμε τα περιεχόμενα του αρχείου με ένα κειμενογράφο. Αν δεν προσθέσουμε άδεια εκτέλεσης στο σενάριο και προσπαθήσουμε να το εκτελέσουμε, το κέλυφος θα εμφανίσει ένα μήνυμα σαν:

```
scriptname: cannot execute.
```

Ένας άλλος τρόπος για να εκτελέσουμε ένα σενάριο είναι να πληκτρολογήσουμε στην γραμμή εντολών μία τελεία (**.**) πριν το όνομα του σεναρίου. Αυτό προκαλεί το κέλυφος να διαβάσει τα περιεχόμενα του κελύφους σαν να τα είχε πληκτρολογήσει ο χρήστης στην γραμμή εντολών.

Οι παραπάνω τρόποι εκτέλεσης παρουσιάζονται στην εικόνα 3.3. Στο παράδειγμα που παρουσιάζει η εικόνα, το σενάριο ονομάζεται fred και περιέχει τις εντολές bob και dave.



Εικόνα 3.3 – Οι τρεις τρόποι εκτέλεσης ενός σεναρίου

Σε ένα σενάριο κελύφους μπορούμε να προσθέσουμε σχόλια βάζοντας στην αρχή της γραμμής το σύμβολο #. Υπάρχει επίσης ένα είδος σχολίου που ξεκινάει με τα σύμβολα #! και η σημασία του είναι ειδική. Δεξιά από τα σύμβολα αυτά πρέπει να μπει σαν παράμετρος το κέλυφος το οποίο θα εκτελέσει το σενάριο αυτό. Τυπικά είναι το /bin/sh αλλά μπορεί να είναι όποιο άλλο κέλυφος επιθυμεί ο χρήστης. Στο τέλος ενός σεναρίου θα πρέπει να προσθέτουμε την εντολή exit 0 με την οποία το ΛΣ λαμβάνει ένα κωδικό εξόδου με τον οποίο βεβαιώνεται οτι το σενάριο τερμάτισε σωστά.

3.8.1 Δομές ελέγχου και επανάληψης

Τα εξελιγμένα κελύφη εργασίας του Unix και του Linux όπως έχουμε αναφέρει και προηγουμένως διαθέτουν προγραμματιστικές δομές οι οποίες μπορούν να χρησιμοποιηθούν σε σχέση με τις μεταβλητές περιβάλλοντος ούτως ώστε να μπορούν να δημιουργηθούν πιο σύνθετα σενάρια. Παρακάτω θα παρουσιάσουμε περιληπτικά την χρήση ορισμένων δομών.

Η δομή if

Η δομή ελέγχου if είναι πολύ απλή στην χρήση της και η δομή της είναι παρόμοια με αυτή άλλων γλωσσών προγραμματισμού. Η δομή if ελέγχει το αποτέλεσμα μιας εντολής και αναλόγως εκτελεί μία ομάδα εντολών.

Η δομή της είναι:

```
if condition
then
   statements
else
   statements
fi
```

Ένα παράδειγμα χρήσης της δομής ελέγχου if είναι το παρακάτω, το οποίο κάνει μία ερώτηση στον χρήστη και έπειτα με βάση την απάντηση (yes/no) ενεργεί αναλόγως:

```
#!/bin/sh
echo "Is it morning? (yes/no)"
read timeofday
if [ $timeofday = "yes" ]
then
   echo "Good morning"
else
  echo "Good afternoon"
fi
exit 0
```

Το παραπάνω σενάριο χρησιμοποιεί την εντολή [ ] με την οποία ελέγχουμε το περιεχόμενο της μεταβλητής timeofday. Το αποτέλεσμα εκτιμάται από την εντολή if η οποία ανάλογα με την τιμή της (true/false) επιτρέπει την εκτέλεση διαφορετικής ομάδας εντολών. Επίσης χρησιμοποιήσαμε την εντολή read με η οποία διαβάζει την είσοδο του χρήστη από το πληκτρολόγιο και την αποθηκεύει σε μία μεταβλητή.

Η δομή for

Χρησιμοποιούμε την δομή for για να υλοποιήσουμε ένα βρόγχο μέσα σε ένα σύνολο τιμών που μπορεί να είναι οποιοδήποτε σύνολο αλφαριθμητικών.

Η δομή της είναι:

```
for variable in values
do
  statements
done
```

Ένα απλό παράδειγμα για να καταλάβουμε την λειτουργία της for είναι το παρακάτω:

```
for foo in bar mpar 43
do
  echo $foo
done
exit 0
```

Το παραπάνω σενάριο θα έχει σαν έξοδο:

```
bar
mpar
43
```

Η λειτουργία του είναι απλή. Το κέλυφος δημιούργησε την μεταβλητή foo και της αναθέτει διαφορετική τιμή σε κάθε επανάληψη του βρόγχου. Οι τιμές που θα πάρει θα είναι οι : bar, mpar, 43. Δηλαδή στην πρώτη επανάληψη θα πάρει την τιμή bar, στην δεύτερη επανάληψη θα πάρει την τιμή mpar και στην τρίτη επανάληψη θα πάρει την τιμή 43. Στην λίστα με τις τιμές που μπορεί να πάρει η μεταβλητή foo μπορούμε να βάλουμε οποιοδήποτε αλφαριθμητικό. Αξίζει να σημειώσουμε ότι η τιμή 43 που θα πάρει η τιμή foo στην τρίτη επανάληψη είναι αλφαριθμητικού τύπου (string) και όχι αριθμητικού τύπου.

3.8.2 Χρήση σεναρίων στο κέλυφος WISH

Το κέλυφος wish επιτρέπει την χρήση σεναρίων στην πιο απλή τους όμως μορφή. Δηλαδή δεν διαθέτει δομές ελέγχου ούτε επανάληψης. Ο χρήστης έχει την δυνατότητα να συγκεντρώσει σε ένα αρχείο κειμένου μία ομάδα από εντολές (εσωτερικές ή εξωτερικές) καθώς να κάνει χρήση των μεταβλητών περιβάλλοντος. Ένα σενάριο μπορεί να εκτελεστεί στο κέλυφος απλώς δίνοντας στην γραμμή εντολών το όνομα του σεναρίου. Επίσης υπάρχει ένας ακόμα τρόπος να εκτελεστεί ένα σενάριο. Το κέλυφος wish μπορεί να ξεκινήσει την λειτουργία του με την παράμετρο –f <script name> όπου script name είναι το όνομα του σεναρίου. Για παράδειγμα, ένας χρήστης θα μπορούσε να εκτελέσει το κέλυφος wish ως:

```
$ wish –f myscript
```

Με αυτό τον τρόπο, το κέλυφος wish θα εκτελέσει το σενάριο με όνομα myscript και όταν τελειώσει η εκτέλεση του θα τερματίσει αυτόματα την λειτουργία του.

3.9 ■ Εγκατάσταση και παραδείγματα λειτουργίας του WISH

Η εγκατάσταση του πειραματικού κελύφους wish (window interactive shell) είναι πολύ απλή. Έχει υλοποιηθεί ένα αρχείο Makefile το οποίο απλοποιεί πολύ την μεταγλώττιση του πηγαίου κώδικα. Ο χρήστης για να μεταγλωττίσει το κέλυφος (εφόσον διαθέτει το εργαλείο make) μπορεί να δώσει την απλή εντολή:

```
$ make
```

Αν για κάποιο λόγο η εντολή make δεν καταφέρει να εντοπίσει το αρχείο Makefile, ο χρήστης μπορεί να δώσει την εντολή :

```
$ make -f Makefile
```

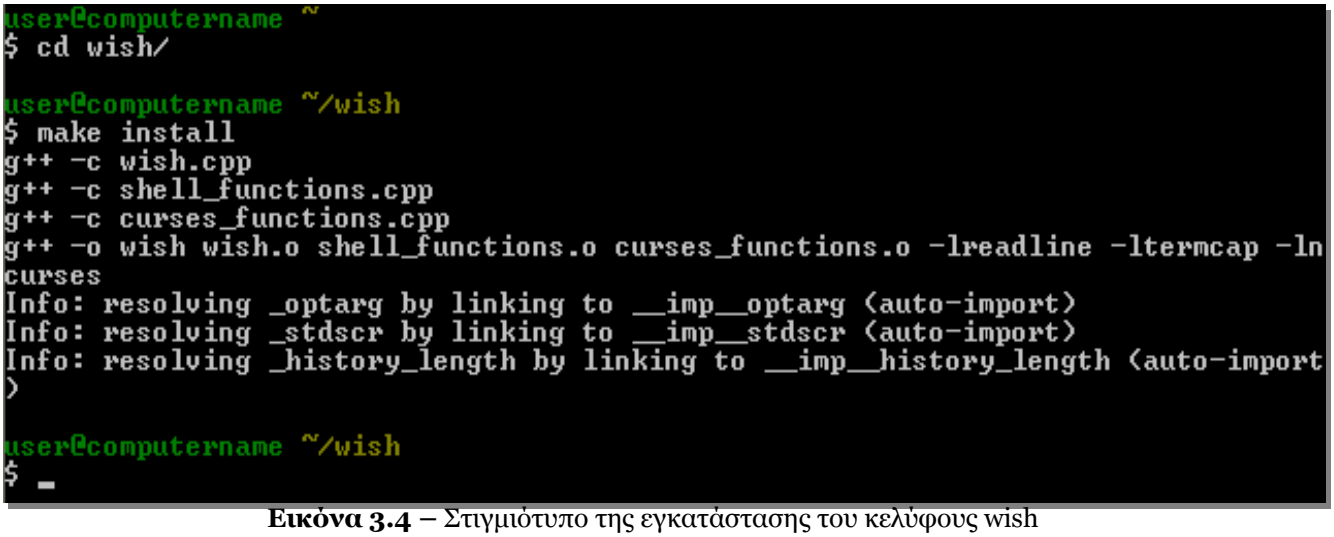
Επίσης αν ο χρήστης επιθυμεί την μεταγλώττιση του κελύφους αλλά και την εγκατάστασή του στον φάκελο /usr/local/bin/wish μπορεί να δώσει την εντολή:

```
$ make install
```

Σε κάθε περίπτωση η εντολή:

```
$ make clean
```

καθαρίζει τον φάκελο από τα προσωρινά αρχεία που ήταν απαραίτητα για την μεταγλώττιση της εφαρμογής. Πρέπει να αναφέρουμε οτι για να γίνει επιτυχημένη μεταγλώττισή του κελύφους πρέπει στο σύστημα του χρήστη να υπάρχουν οι βιβλιοθήκες Readline, ncurses καθώς και ο μεταγλωττιστής g++. Η εικόνα 3.4 παρουσιάζει ένα στιγμιότυπο (screenshot) της εγκατάστασης του κελύφους με την εντολή make install.



Εικόνα 3.4 – Στιγμιότυπο της εγκατάστασης του κελύφους wish

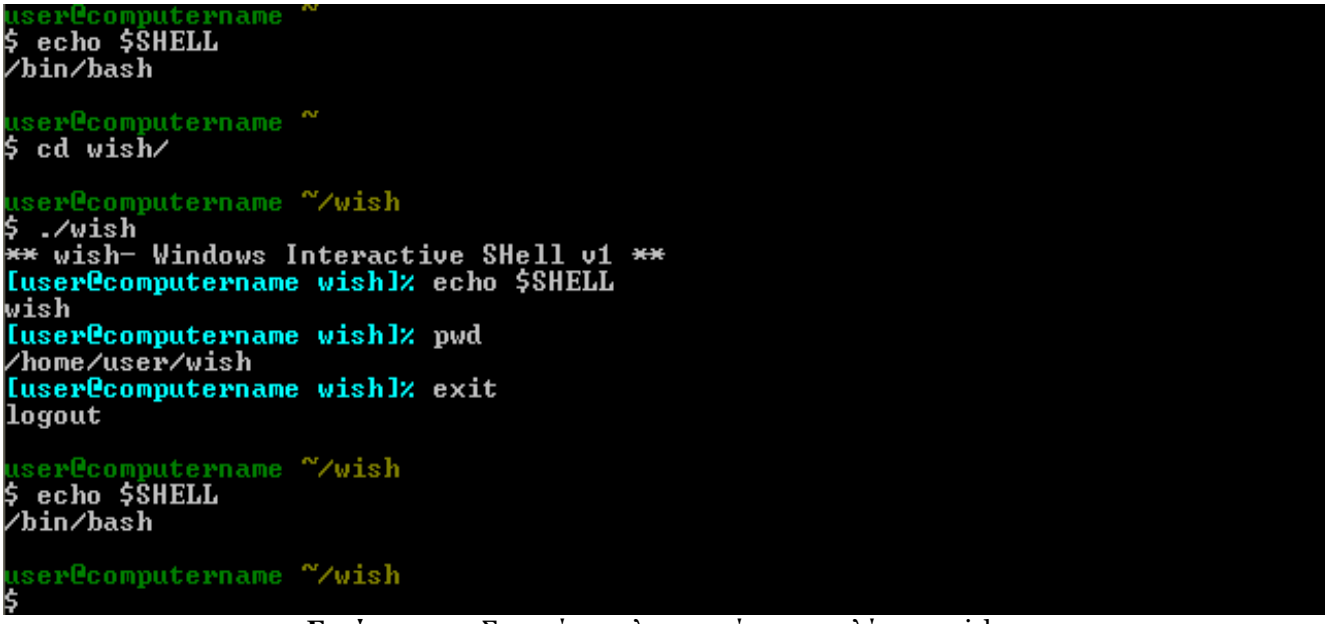
Μετά την εγκατάσταση του κελύφους, ο χρήστης μπορεί να εκκινήσει την απλή έκδοση γραμμής εντολών με την εντολή:

```
$ wish
```

ενώ με την εντολή:

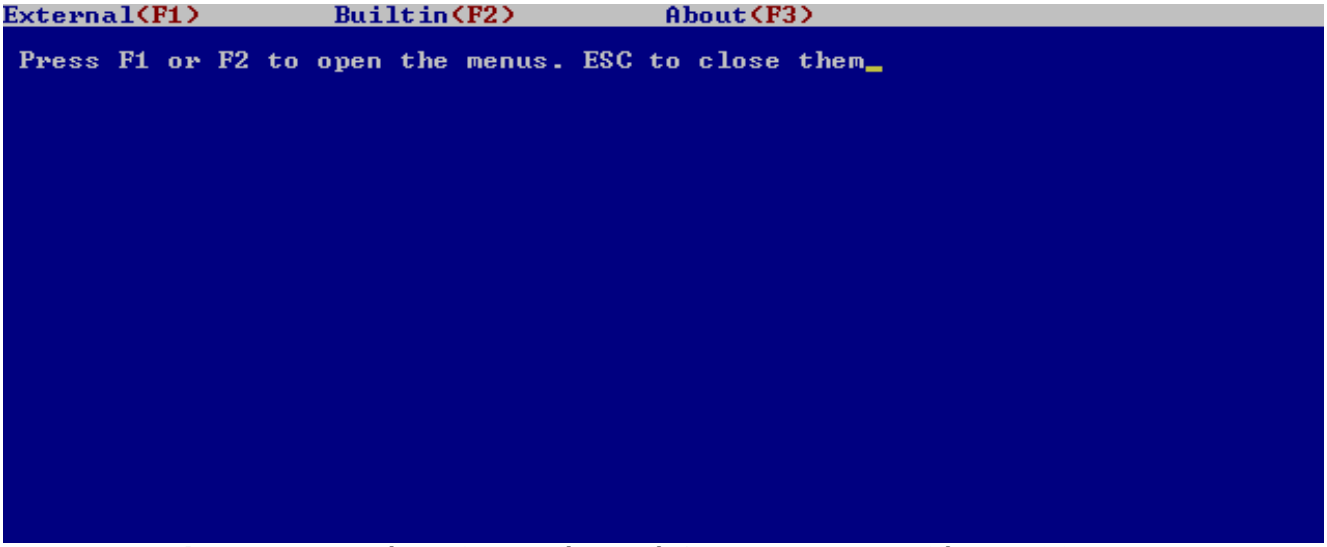
```
$ wish -n
```

εκκινεί την έκδοση ncurses του κελύφους wish. Για την έκδοση ncurses θα αναφερθούμε αναλυτικότερα στην επόμενη ενότητα. Η εικόνα 3.5 παρουσιάζει ένα στιγμιότυπο του κελύφους σε κατάσταση γραμμής εντολών. Όπως φαίνεται και στην εικόνα ο χρήστης πριν εκτελέσει το κέλυφος wish έδωσε την εντολή echo $SHELL για να διαπιστώσει οτι εκείνη τη στιγμή εκτελεί το κέλυφος bash. Έπειτα, εκτελεί το κέλυφος wish και μέσα από το κέλυφος wish δίνει πάλι την εντολή echo $SHELL. Τότε το κέλυφος απαντάει οτι το κέλυφος που εκτελείται είναι πλέον το wish. Μετά την έξοδο από το wish το ενεργό κέλυφος είναι πάλι το κέλυφος bash.



Εικόνα 3.5 – Στιγμιότυπο λειτουργίας του κελύφους wish

Η εικόνα 3.6 παρουσιάζει ένα στιγμιότυπο του κελύφους σε κατάσταση ncurses. Για την ncurses εκδοχή του κελύφους wish θα αναφερθούμε στην επόμενη ενότητα.



Εικόνα 3.6 – Στιγμιότυπο λειτουργίας της έκδοσης ncurses του κελύφους wish

Στο κεφάλαιο αυτό έχουμε αναφέρει τις σημαντικότερες λειτουργίες και δυνατότητες του πειραματικού κελύφους wish. Ο πίνακας 3.3 παρουσιάζει συνοπτικά τις διαφορές του κελύφους wish από μερικά γνωστά κελύφη όπως το bash, το korn shell, το bourne shell κλπ.

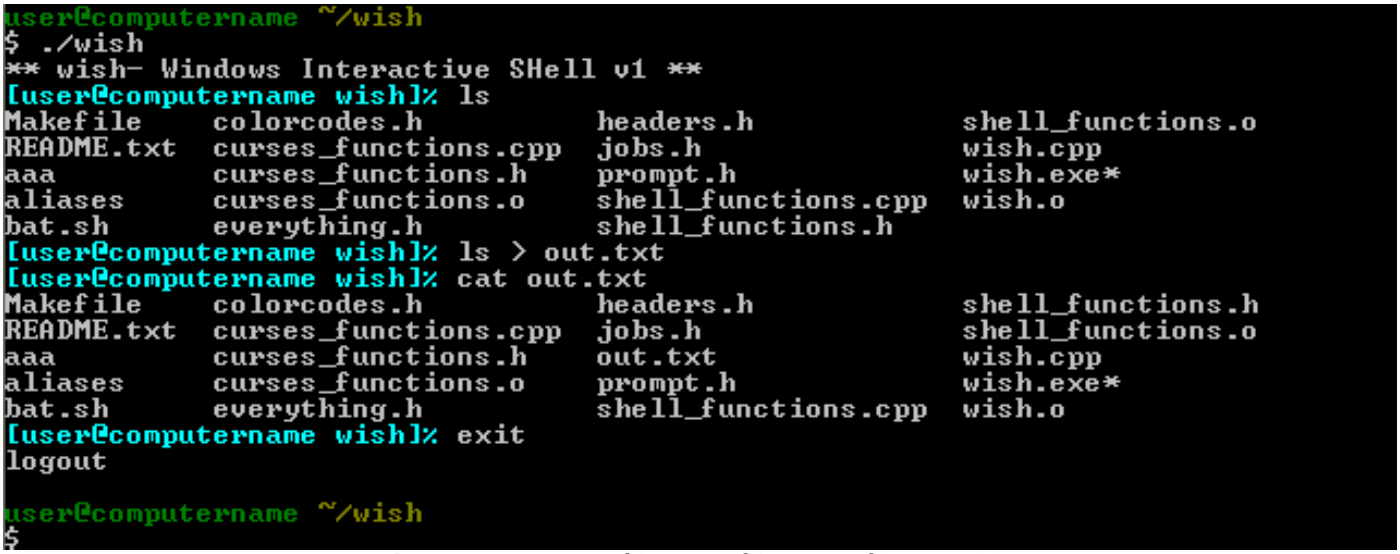
	bourne	csh	ksh	bash	tcsh	zsh	wish
Διαχείριση διεργασιών	-	Y	Y	Y	Y	Y	Y
Ψευδώνυμα (aliases)	-	Y	Y	Y	Y	Y	Y
Συναρτήσεις κελύφους	Y	-	Y	Y	-	Y	-
Δομές προγραμματισμού	sh	csh	sh	sh	csh	sh	-
Σενάρια κελύφους	Y	Y	Y	Y	Y	Y	Y
Ανακατεύθυνση της εισόδου/εξόδου (redirection)	Y	-	Y	Y	-	Y	Y
Διοχέτευση εξόδου (pipes)	Y	Y	Y	Y	Y	Y	Y
Στοίβα καταλόγων	-	Y	Y	Y	Y	Y	Y
Ιστορικό εντολών	-	Y	Y	Y	Y	Y	Y
Επεξεργασία της γραμμής εντολών	-	-	Y	Y	Y	Y	Y
Αυτόματη συμπλήρωση του ονόματος των αρχείων	-	Y	Y	Y	Y	Y	Y
Αυτόματη συμπλήρωση του ονόματος του χρήστη	-	Y	Y	Y	Y	Y	-
Αυτόματη συμπλήρωση του ιστορικού των εντολών	-	-	-	Y	Y	Y	-
Ενσωματομένη δυνατότητα αριθμιτικών υπολογισμών	-	Y	Y	Y	Y	Y	-
Προσωποποίηση της προτροπής	-	-	Y	Y	Y	Y	Y
Διατήθεται ελεύθερα	-	-	-	Y	Y	Y	Y
Ελέγχει το Mailbox	-	Y	Y	Y	Y	Y	-
Διαθέτει αρχείο εκκίνησης	Y	Y	Y	Y	Y	Y	Y
Μπορεί να αγνοήσει το αρχείο εκκίνησης	-	Y	-	Y	-	Y	-
Μπορεί να καθοριστεί το όνομα του αρχείου εκκίνησης	-	-	Y	Y	-	-	-
Καταγραφή μεταβλητών	-	Y	Y	-	Y	Y	-
Τοπικές μεταβλητές	-	-	Y	Y	-	Y	-
Γραφικό περιβάλλον	-	-	-	-	-	-	Y
Συμβατό με Cygwin	Y	Y	Y	Y	Y	Y	Y

Πίνακας 3.7 – Διαφορές του κελύφους wish με άλλα γνωστά κελύφη

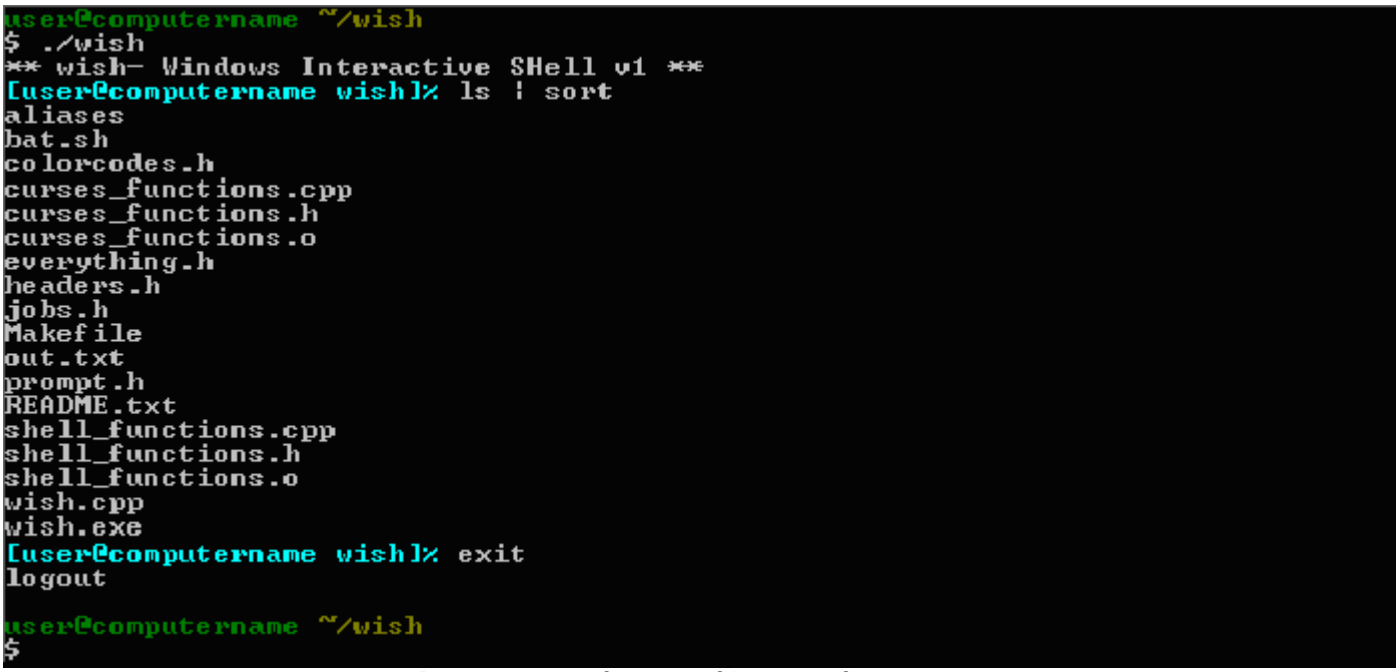
Όπως βλέπουμε και στον παραπάνω πίνακα, το πειραματικό κέλυφος wish διαθέτει αρκετά από τα πιο βασικά χαρακτηριστικά που πρέπει να διαθέτει ένα κέλυφος. Χαρακτηριστικά όπως η ανακατεύθυνση της εισόδου/εξόδου, διοχέτευση εξόδου, σενάρια κελύφους, διαχείριση διεργασιών, ψευδώνυμα, στοίβα καταλόγων, ιστορικό εντολών, αυτόματη συμπλήρωση ονομάτων αρχείων, αρχεία εκκίνησης και προσωποποίηση της προτροπής. Αναφορικά να πούμε ότι το κέλυφος wish μπορεί να λειτουργήσει

και σε περιβάλλον Cygwin εκτός από Unin/Linux. Το Cygwin είναι κατά κάποιο τρόπο, ένας εξομοιωτής περιβάλλοντος Unix και διαθέτει αρκετά από τα εργαλεία του. Τέλος, το κέλυφος wish διαθέτει ένα χαρακτηριστικό που το καθιστά μοναδικό σε σύγκριση με τα υπόλοιπα κελύφη – το γραφικό περιβάλλον ncurses.

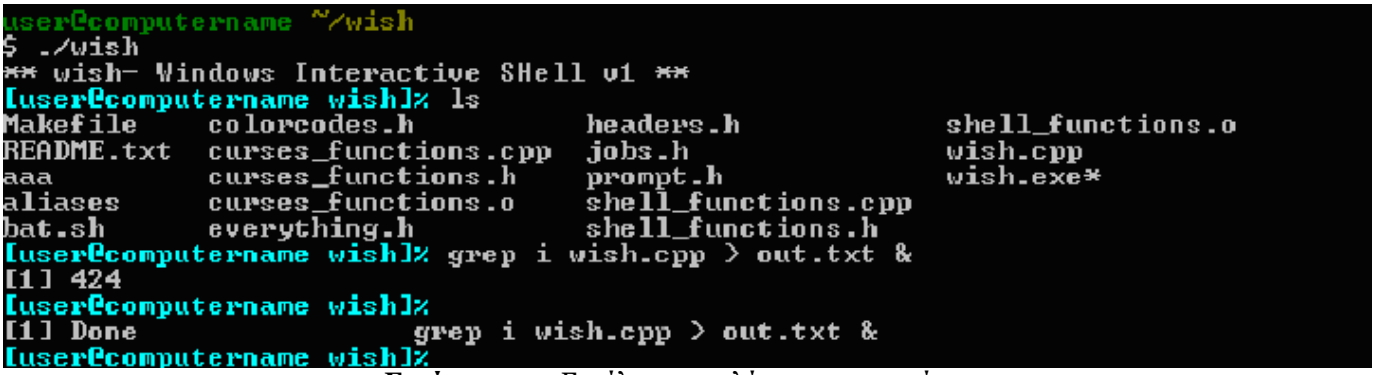
Παρακάτω θα δούμε μερικά στιγμιότυπα του κελύφους wish που παρουσιάζουν αρκετά από τα χαρακτηριστικά που αναφέραμε.



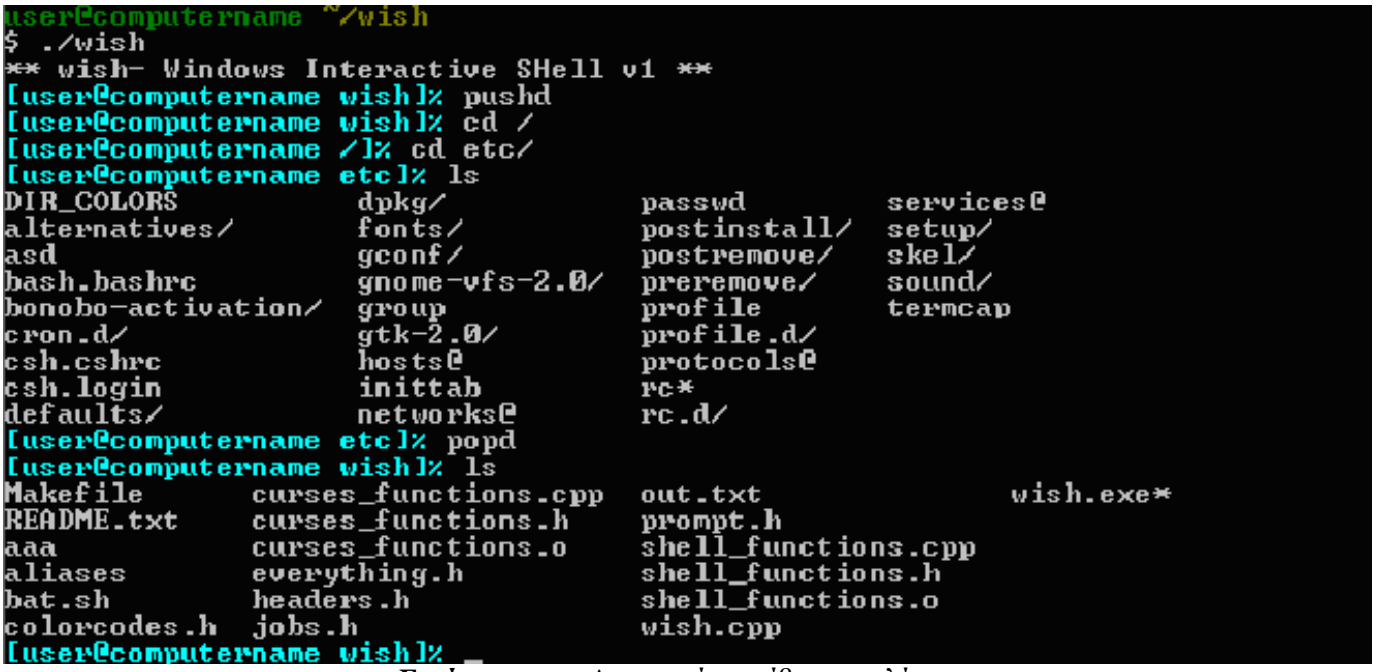
Εικόνα 3.7 – Ανακατεύθυνση εξόδου στο κέλυφος wish



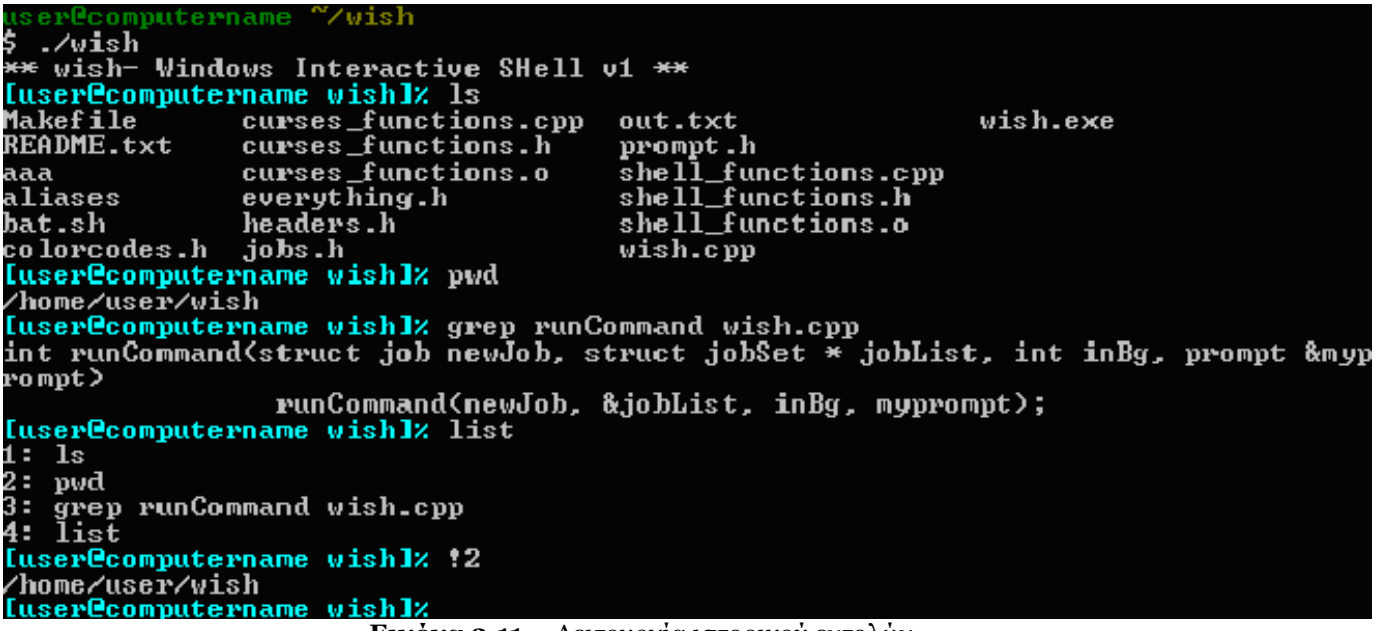
Εικόνα 3.8 – Διοχέτευση εξόδου στο κέλυφος wish



Εικόνα 3.9 – Εκτέλεση εντολής στο παρασκήνιο



Εικόνα 3.10 – Λειτουργία στοίβας καταλόγου

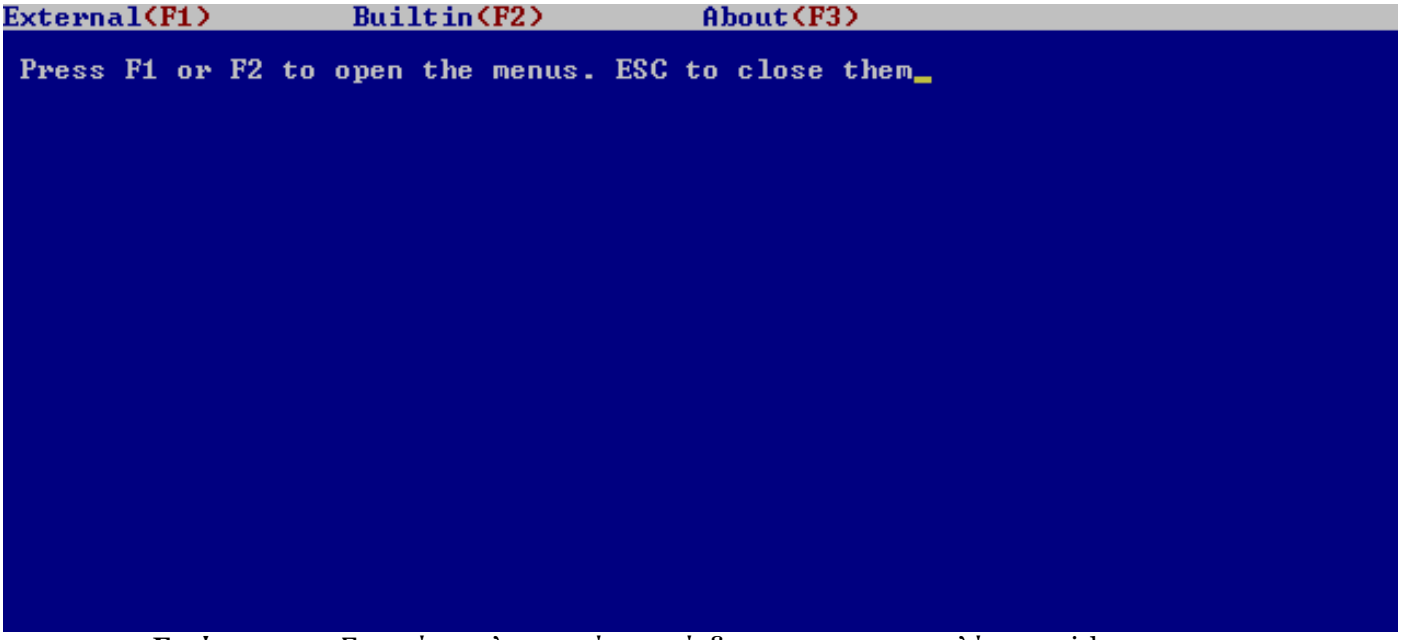


Εικόνα 3.11 – Λειτουργία ιστορικού εντολών

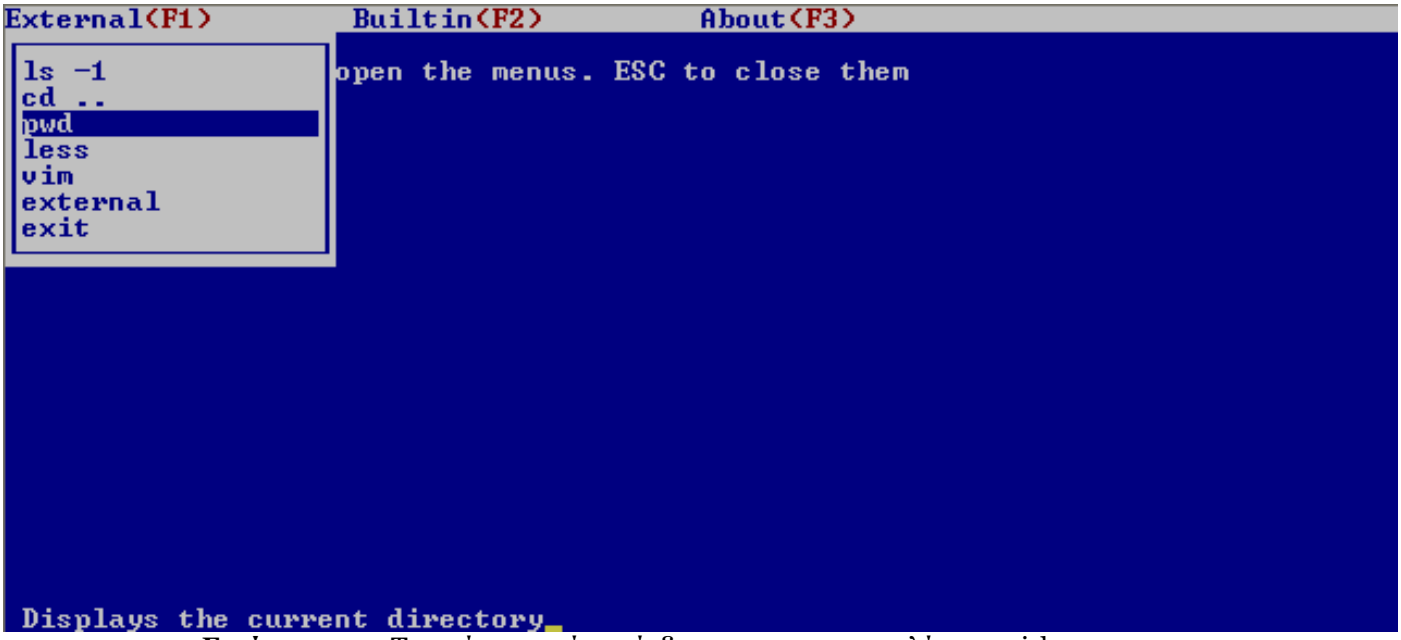
3.10 ■ Η διεπαφή ncurses του WISH

Η διεπαφή ncurses του κελύφους wish αποτελεί ίσος την μεγαλύτερη διαφοροποίησή του από τα υπόλοιπα κελύφη. Ο χρήστης δεν δίνει τις εντολές μέσα από μία γραμμή εντολών, αλλά χρησιμοποιώντας μενού και παράθυρα τα οποία τα χειρίζεται μέσω του πληκτρολογίου του. Υπάρχουν τρία μενού. Το πρώτο μενού έχει όνομα External (εξωτερικές) και οι επιλογές που εμφανίζει είναι μερικές από τις πιο συνηθισμένες εξωτερικές εντολές που χρησιμοποιεί ένας χρήστης. Για παράδειγμα ls, cd .., pwd κλπ. Το δεύτερο μενού έχει όνομα Builtin και εμφανίζει αρκετές από τις ενσωματωμένες εντολές του κελύφους wish. Για παράδειγμα pushd, alias, list κλπ. Το τρίτο μενού έχει όνομα About και περιέχει επιλογές οι οποίες εμφανίζουν πληροφορίες σχετικές με το κέλυφος όπως την έκδοσή του κλπ. Για να πλοηγηθεί ο χρήστης στα μενού χρησιμοποιεί τα πλήκτρα F1, F2, F3 καθώς και τα βελάκια (arrow keys). Με το πλήκτρο ESC κλείνει ένα μενού.

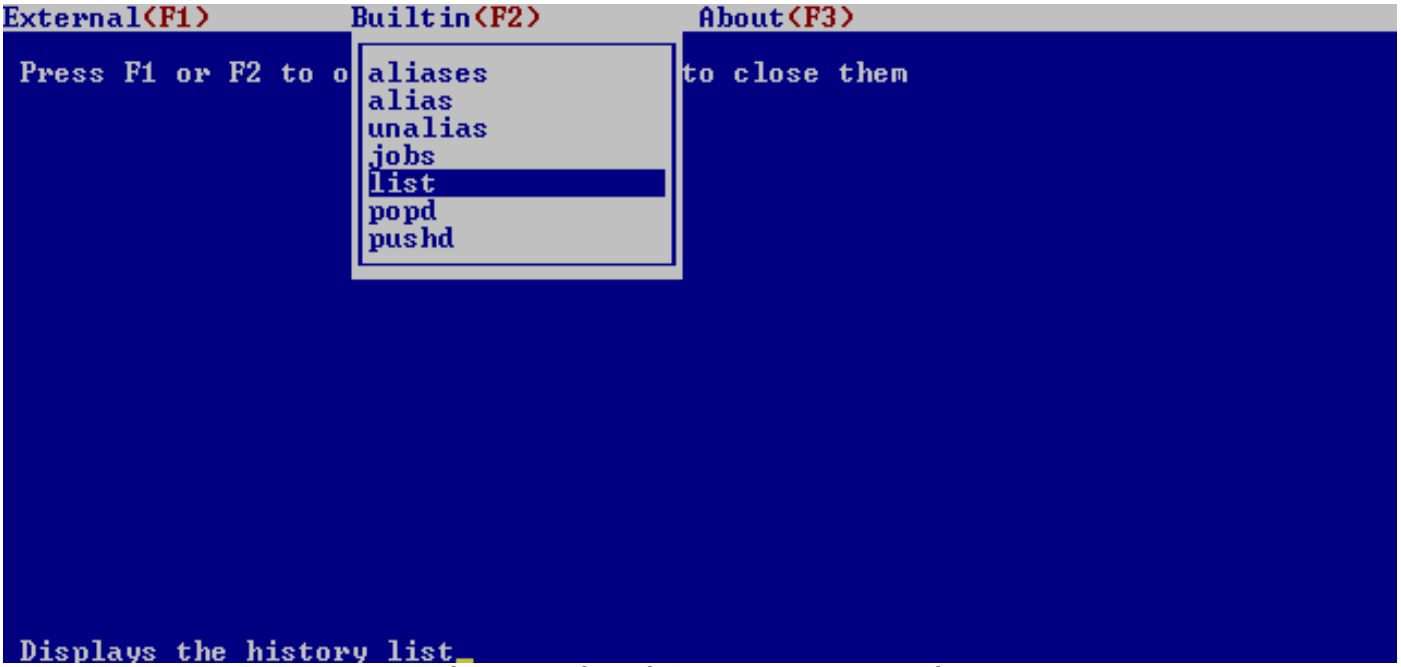
Όταν ο χρήστης επιλέξει μία εντολή από ένα μενού το κέλυφος επιστρέφει σε κατάσταση γραμμής εντολών και την εκτελεί. Μετά το τέλος της εντολής το κέλυφος επιστρέφει στην παραθυρική κατάσταση. Παρακάτω θα δούμε μερικά στιγμιότυπα της διεπαφής ncurses του κελύφους wish



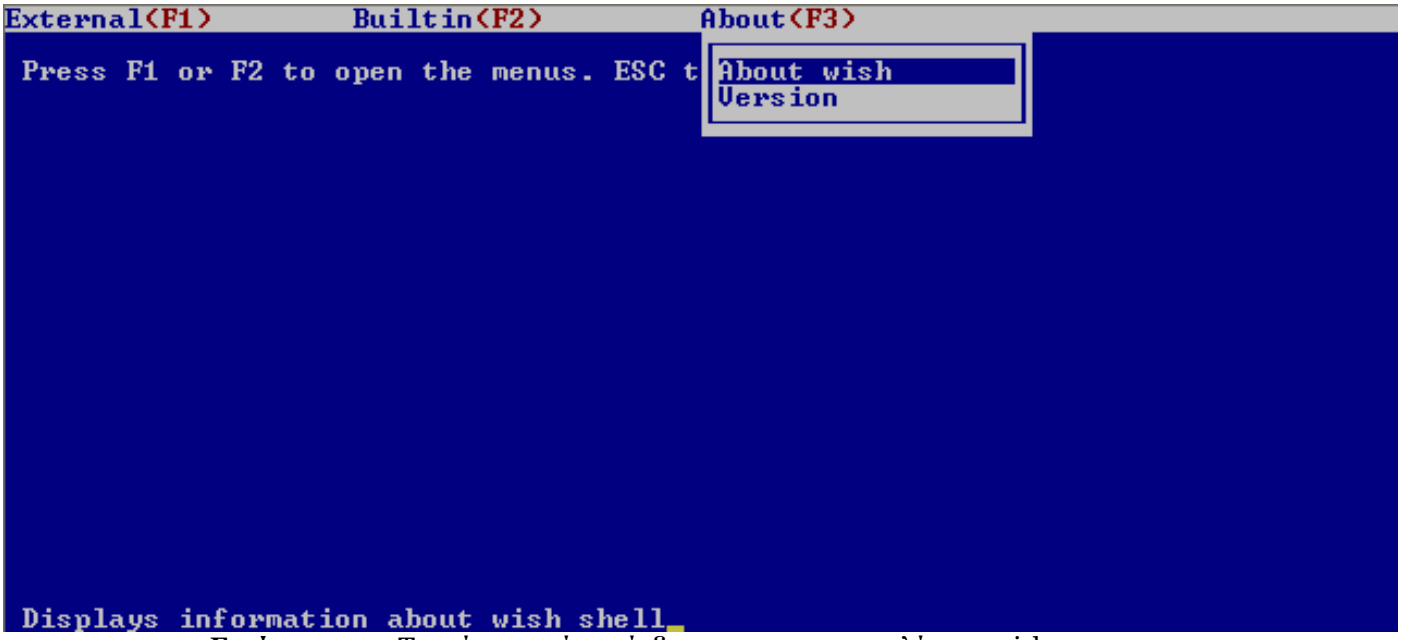
Εικόνα 3.12 – Στιγμιότυπο λειτουργίας της έκδοσης ncurses του κελύφους wish



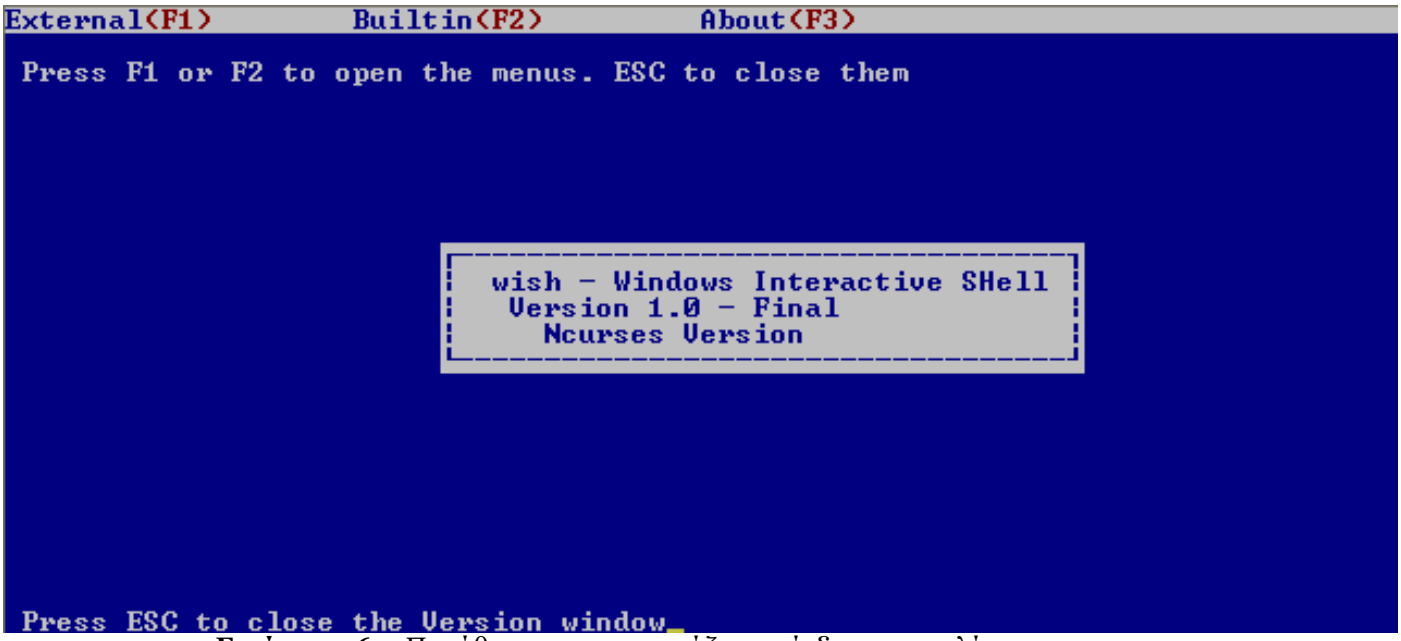
Εικόνα 3.13 – Το πρώτο μενού της έκδοσης ncurses του κελύφους wish



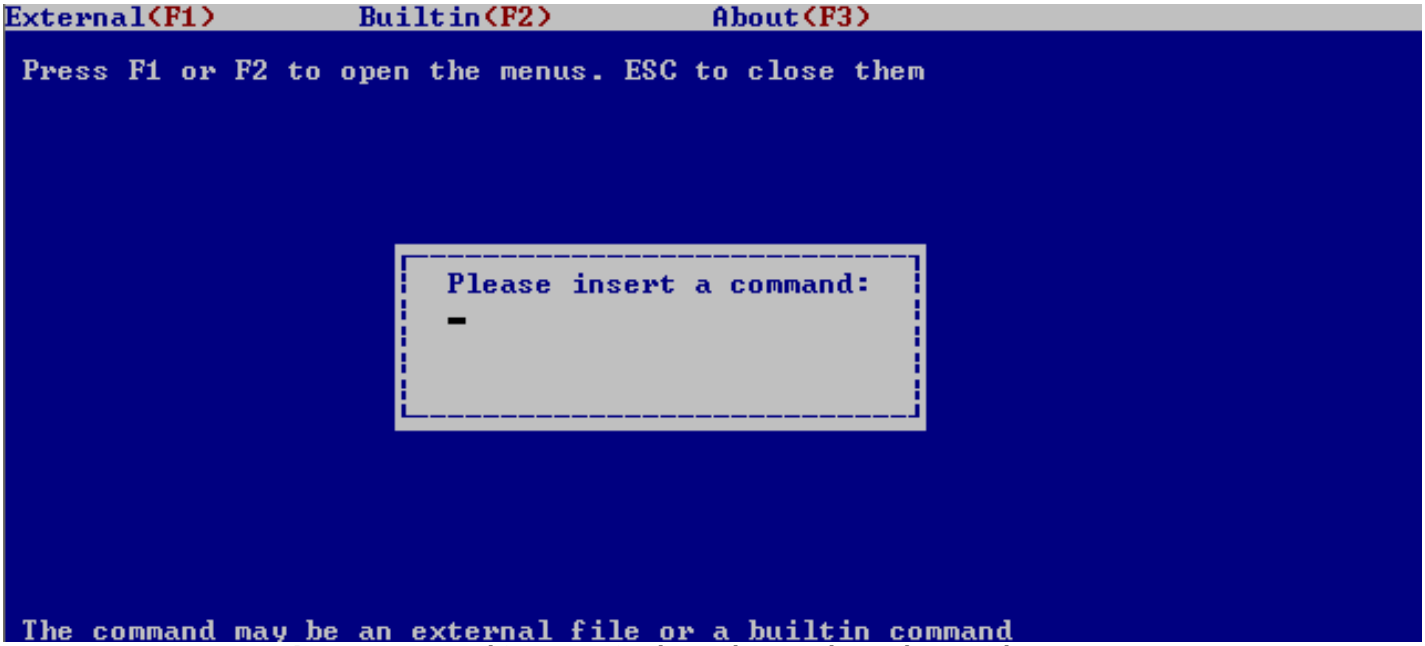
Εικόνα 3.14 – Το δεύτερο μενού της έκδοσης ncurses του κελύφους wish



Εικόνα 3.15 – Το τρίτο μενού της έκδοσης ncurses του κελύφους wish



Εικόνα 3.16 – Παράθυρο που παρουσιάζει την έκδοση του κελύφους



Εικόνα 3.17 – Παράθυρο που ζητάει από τον χρήστη μία εντολή

Παράρτημα

Ο υλοποιημένος κώδικας (source code)

/***/

Το αρχείο everything.h

/***/

```
/****************************************
        Wish Unix Shell - Version 1.0
        Author: Giannakidis Apostolis
        Id: everything.h
****************************************/

#include "headers.h"
#include <readline/readline.h>
#include <readline/history.h>

extern HIST_ENTRY **history_list ();

struct shell_variables shell_vars;

extern char* optarg;

extern int optind, optopt;

static struct sigaction entry_int, entry_quit;

static string last_command;

stack<string> dir_stack;

bool ncurses_ver;
```

/***/

Το αρχείο headers.h

/***/

```
/*****************************************
       Wish Unix Shell - Version 1.0
       Author: Giannakidis Apostolis
       Id: headers.h
*****************************************/

/*    Unix/Linux specific headers */
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <dirent.h>
#include <glob.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/wait.h>
#include <unistd.h>
#include <curses.h>

/* New C++ headers */
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <stack>
#include <sstream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <cctype>

/* My headers */
#include "jobs.h"
#include "prompt.h"
#include "colorcodes.h"
#include "curses_functions.h"
```

/***/

To αρχείο wish.cpp

/***/

```
/****************************************
        Wish Unix Shell - Version 1.0
        Author: Giannakidis Apostolis
        Id: wish.cpp
****************************************/

#include "everything.h"

WINDOW *menubar,*messagebar;
WINDOW **menu_items,*popup_window;

static long get_max_pathname(const char *path)          //Mia sinartisi me i opoia
mou epistrefei to megisto pathname
{
        // pou mporei na iparxei. Tin sinartisi afti ti xrisimopoiei i get_cwd
        long max_path;                                                  // i
opoia epistrefei to current direcotry.

        errno = 0;
        max_path = pathconf(path,_PC_PATH_MAX);
        if(max_path == -1)
        {
          if (errno == 0)
                max_path = 4096;            //guess
        }
        return max_path+1;
}

static char *get_cwd(bool cleanup)          //Epistrefei to current direcotry. Tin
xrisimopoiw gia na mathw to trexw directory
{
        static char* cwd = NULL;                    //to opoio to emfanizw sto
prompt.
        static long max_path;

        if(cleanup)
        {
          free(cwd);
          cwd = NULL;
        }
        else
        {
          if (cwd == NULL)
          {
                max_path = get_max_pathname(".");
                cwd = (char*)malloc((size_t)max_path);
```

```
            }
            getcwd(cwd,max_path);
            return cwd;
        }

        return NULL;
}

void freeJob(struct job * cmd)
{
   for (int i = 0; i < cmd->numProgs; i++)
   {
      free(cmd->progs[i].argv);
      if (cmd->progs[i].redirections) free(cmd->progs[i].redirections);
      if (cmd->progs[i].freeGlob) globfree(&cmd->progs[i].globResult);
   }
   free(cmd->progs);
   if (cmd->text) free(cmd->text);
   free(cmd->cmdBuf);
}

int getCommand(istream *source, string &command, prompt myprompt)
{
   command.clear();
   string buff;
   char *buffer = (char *)NULL;

  textcolor(BRIGHT, CYAN, BLACK);
  string pr = myprompt.get_prompt();
  char color[13];
  sprintf(color, "%c[%d;%d;%dm", 0x1B, 0, 7 + 30, 0 + 40);
  pr+=color;

   if (*source == cin)
   {
       buffer = readline(pr.c_str());      //Diavazw apo to cin (stdout) me tin readline

       if (buffer==(char *)NULL)
       {
         command="";
         logout();
       }
       else
         command=buffer;
   }
   else
   {
       textcolor(RESET, WHITE, BLACK);
       if(source->eof())//Epistrefw 1 an teliwse to arxeio (diavase EOF).
        return 1;
```

```
        getline(*source,buff,'\n');          //Diavazw apo to arxeio me tin geline

        command = buff;
    }

    if (*source == cin)
    {
      if ((command.length()!=0) && (command[0]!='!'))
      {
        if (last_command!=command)  //An i proigoumeni entoli itane idia
        {                           //me tin trexousa entoli den tin prosthetw stin lista
          last_command=command;
                add_history (command.c_str());
        }
      }
    }

    ReplaceAliases(command);

    if(command.find("$")!=string::npos)
      ReplaceVariables(command);

    if ((*source == cin) && (command[0]=='!'))
      quick_command(command);

    return 0;
}

/* i sinartisi afti kanei to parsing tis gramming entolwn.
   Epistrefei to cmd->numProgs. An epistrepsei 0 o xristis den
   edwse kamia entoli. Apla patise to enter. An exei dwsei o xristis
   entoles, o diktis commandPtr dixnei stin epomeni entoli (oxi stin prwti).
   An den iparxei epomeni entoli pernei tin timi NULL
*/

int parseCommand(char **commandPtr, struct job *job, int *isBg)
{
    char *command;
    char *returnCommand = NULL;
    char *src, *buf, *chptr;
    int argc = 0;
    int done = 0;
    int argvAlloced;
    int i;
    char quote = '\0';
    int count;
    struct childProgram * prog;

    while (**commandPtr && isspace(**commandPtr))
```

```
    (*commandPtr)++;        // Agnow ta kena mprosta apo tin entoli

// Agnow kenes grammes kai grammes pou ksekinane me # (sxolia)

 if (!**commandPtr || (**commandPtr=='#'))
 {
   job->numProgs = 0;
   *commandPtr = NULL;
   return 0;
 }

*isBg = 0;
job->numProgs = 1;
job->progs = (childProgram*)malloc(sizeof(*job->progs));

job->cmdBuf = command = (char*)calloc(1, strlen(*commandPtr) + 1);
job->text = NULL;

prog = job->progs;
prog->numRedirections = 0;
prog->redirections = NULL;
prog->freeGlob = 0;
prog->isStopped = 0;

argvAlloced = 5;
prog->argv = (char**)malloc(sizeof(*prog->argv) * argvAlloced);
prog->argv[0] = job->cmdBuf;

buf = command;
src = *commandPtr;
while (*src && !done)
{
   if (quote == *src)
      quote = '\0';
   else if (quote)
    {
      if (*src == '\\')
       {
         src++;
         if (!*src)
      {
            cerr<<"Error: character expected after \\\n";
            freeJob(job);
            return 1;
         }

         if (*src != quote) *buf = '\\';
      } else if (*src == '*' || *src == '?' || *src == '[' ||  *src == ']')
         *buf++ = '\\';
      *buf++ = *src;
```

```
        }

        else if (isspace(*src))
        {
          if (*prog->argv[argc])
           {
             buf++, argc++;

             if ((argc + 1) == argvAlloced)
          {
                argvAlloced += 5;
                prog->argv = (char**)realloc(prog->argv, sizeof(*prog->argv) *
argvAlloced);
             }
             prog->argv[argc] = buf;
          }
        }
        else
        {
         switch (*src)
         {
         case "":
         case '\'':
           quote = *src;
           break;

         case '#':               /* sxolio */
           done = 1;
           break;

         case '>':               /* redirections */
         case '<':
           i = prog->numRedirections++;
           prog->redirections = (redirectionSpecifier*)realloc(prog->redirections,
sizeof(*prog->redirections) * (i + 1));

           prog->redirections[i].fd = -1;
           if (buf != prog->argv[argc])
            {

             prog->redirections[i].fd = strtol(prog->argv[argc], &chptr, 10);

             if (*chptr && *prog->argv[argc])
          {
                buf++;
             argc++;
             }
          }

          if (prog->redirections[i].fd == -1)
```

```
          {
            if (*src == '>')
               prog->redirections[i].fd = 1;
            else
               prog->redirections[i].fd = 0;
          }

          if (*src++ == '>')
          {
            if (*src == '>')
               prog->redirections[i].type = REDIRECT_APPEND, src++;
            else
               prog->redirections[i].type = REDIRECT_OVERWRITE;
          }
          else
            prog->redirections[i].type = REDIRECT_INPUT;

          chptr = src;
          while (isspace(*chptr))
          chptr++;

          if (!*chptr) {
                   cerr<<"Error: file name expected after "<<*src<<endl;
            freeJob(job);
            return 1;
          }

          prog->redirections[i].filename = buf;
          while (*chptr && !isspace(*chptr))
            *buf++ = *chptr++;

          src = chptr - 1;
          prog->argv[argc] = ++buf;
          break;

        case '|':                       /* pipe */
          if (*prog->argv[argc])
          argc++;

          if (!argc)
          {
            cerr<<"Error: empty command in pipe"<<endl;
            freeJob(job);
            return 1;
          }
          prog->argv[argc] = NULL;

          job->numProgs++;
          job->progs = (childProgram*)realloc(job->progs, sizeof(*job->progs) * job-
>numProgs);
```

```
        prog = job->progs + (job->numProgs - 1);
        prog->numRedirections = 0;
        prog->redirections = NULL;
        argc = 0;

        argvAlloced = 5;
        prog->argv = (char**)malloc(sizeof(*prog->argv) * argvAlloced);
        prog->argv[0] = ++buf;

        src++;
        while (*src && isspace(*src))
        src++;

        if (!*src)
         {
           cerr<<"Error: empty command in pipe"<<endl;
           return 1;
        }
        src--;

        break;

      case '&':                    /* background */
        *isBg = 1;
      case ';':                   /* pollaples entoles */
        done = 1;
        returnCommand = *commandPtr + (src - *commandPtr) + 1;
        break;
      default:
        *buf++ = *src;
     }
     }

    src++;
}

if (*prog->argv[argc])
    argc++;

if (!argc)
{
    freeJob(job);
    return 0;
}

prog->argv[argc] = NULL;

if (!returnCommand)
{
    job->text = (char*)malloc(strlen(*commandPtr) + 1);
```

```
      strcpy(job->text, *commandPtr);
   }
   else
   {
      count = returnCommand - *commandPtr;
      job->text = (char*)malloc(count + 1);
      strncpy(job->text, *commandPtr, count);
      job->text[count] = '\0';    /* Afinw ta kena sto telos afou kserw oti
                                     tin epomeni fora pou tha kanei parse tin command
                                     tha agnoisei ta kena */
   }

   *commandPtr = returnCommand;

   return 0;
}

int setupRedirections(struct childProgram * prog)
{
   int i;
   int openfd;
   int mode;
   struct redirectionSpecifier *redir = prog->redirections;

   for (i = 0; i < prog->numRedirections; i++, redir++)
   {
      switch (redir->type)
       {
       case REDIRECT_INPUT:
         mode = O_RDONLY;
         break;
       case REDIRECT_OVERWRITE:
         mode = O_RDWR | O_CREAT | O_TRUNC;
         break;
       case REDIRECT_APPEND:
         mode = O_RDWR | O_CREAT | O_APPEND;
         break;
      }

      openfd = open(redir->filename, mode, 0666);   // Xrisimopoiw to system call tou
Unix open()
                                                    // giati epistrefei to FD tou neou arxeio.

      if (openfd < 0)
       {
         cerr<<"Error opening "<<redir->filename<<" : "<<strerror(errno)<<endl;
         return 1;
      }

      if (openfd != redir->fd)   //To FD tou neou arxeiou to antigrafw
```

```
        {                               //sto redir->fd.
          dup2(openfd, redir->fd);
          close(openfd);
        }
    }

    return 0;
}

/*
Arxikopoiisi tou prompt wste na exei tin morfi : [user_name@machine_name
current_dir]
*/

prompt::prompt()
{
        name = cuserid(NULL);

        name_check();

        node = get_node_name();

        dir = get_cwd(false);      //den xrisimopoiw tin getcwd gia logous efxristias
        get_cwd(true);

        path_check();

        string val;
        val = Find_From_Map("DIR");
        if(val=="on")
          dir_on = true;
        else
          dir_on = false;

        val = Find_From_Map("PS1");
        if(!val.empty())
          ps1=val;
        else
        {
          ps1="%";
          putenv("PS1=%");
        }
}

void prompt::name_check()
{
        if (name.length()==0)
        {
          cerr<<"Error: Could not get name information\n";
          exit(2);
```

```
        }
}

string prompt::get_node_name()
{
        char computer_name[256];

        if(gethostname(computer_name,255)!=0)
        {
         cerr<<"Error: Could not get host information\n";
         exit(2);
        }

        string name = computer_name;
        return name;
}

void prompt::path_check()
{
        if (dir.length()==0)
        {
          cerr<<"Error: Could not get path information (current directory)\n";
          exit(2);
        }

        if (Find_From_Map("FULLPATH")=="off")
        {
         if (dir.length()!=1 && dir!="/")
         {
          int i = dir.find_last_of("/");
          dir = dir.substr(i+1,dir.length()-i);     //den thelw to full path, alla mono to
current directory name
         }
        }
}

/*
Emfanizei to prompt ston xristi
*/

string prompt::get_prompt()
{
        string temp = "["+name+"@"+node;

        if (dir_on)
          temp+=" "+dir;

        temp+="]";

        if(getuid() != 0)
```

```
        temp+=ps1+" ";
      else
        temp+="# ";    //Emfanizw # an einai root

      return temp;
}

/*
Na emfanizei to current directory sto prompt h oxi?
*/

void prompt::display_dir(bool status)
{
dir_on=status;
}

void prompt::refresh_prompt()
{
      dir = get_cwd(false);

      get_cwd(true);

      path_check();
}

/*  Elenxw an ena string periexei grammata */

bool isNumerical(string tmp)
{
      for (int i=0;i<=tmp.length();i++)
      {
        if ( isalpha(tmp[i]) )
          return false;
      }
      return true;
}

static bool ignore_sig()
{
      static bool first = true;
      struct sigaction act_ignore;
      signal(SIGTTOU, SIG_IGN);
      signal(SIGSEGV,SIG_IGN);
      memset(&act_ignore,0,sizeof(act_ignore));
      act_ignore.sa_handler = SIG_IGN;
      if (first)
      {
        first=false;
        sigaction(SIGINT, &act_ignore, &entry_int);
        sigaction(SIGQUIT, &act_ignore, &entry_int);
```

```
        }
        else
        {
          sigaction(SIGINT, &act_ignore, NULL);
          sigaction(SIGINT, &act_ignore, NULL);
        }

        return true;
}

bool isAssignment(char** argv)
{

int i=0;

while (argv[i])
{

  if(strchr(argv[i],'=')!=NULL)
        return true;

  i++;
}

return false;
}

/*  I sinartisi afti ektelei tin entoli  */
int runCommand(struct job newJob, struct jobSet * jobList, int inBg, prompt
&myprompt)
{
    struct job * job;
    char * newdir, * buf;
    int i, len;
    int nextin, nextout;
    int pipefds[2];           /* pipefd[0] is for reading */
    char * statusString;
    int jobNum;

        /*******************************************************/

    /* Edw ilopoiw tis BUILT-IN entoles. Den kanw fork/exec se aftes giati den exei
noima na
        dimiourgithei nea diergasia gia na ektelestoun. */

    if (!strcmp(newJob.progs[0].argv[0], "exit"))        // exit: vgenei apo to shell
    {
        if (newJob.progs[0].argv[1]!=NULL)
        {
          if (!isNumerical(newJob.progs[0].argv[1]))
```

```
        {
          cerr<<"exit: <num> Numeric value expected"<<endl;
          return 0;
        }
        else
        {
           if(ncurses_ver)
           {
                clear();
                delwin(menubar);
                delwin(messagebar);
                endwin();
           }

          checkBackground(jobList);
          cout<<"logout"<<endl;
          exit(atoi(newJob.progs[0].argv[1]));
        }
      }
      else
      {
           if(ncurses_ver)
           {
                clear();
                delwin(menubar);
                delwin(messagebar);
                endwin();
           }

       checkBackground(jobList);
       cout<<"logout"<<endl;
      exit(0);
      }
}
else if (!strcmp(newJob.progs[0].argv[0], "quit")) //quit: vgenei apo to shell
{
    if (newJob.progs[0].argv[1]!=NULL)
    {
      if (!isNumerical(newJob.progs[0].argv[1]))
      {
        cerr<<"quit: <num> Numeric value expected"<<endl;
        return 0;
      }
      else
      {
           if(ncurses_ver)
           {
                clear();
                delwin(menubar);
                delwin(messagebar);
```

```
                endwin();
            }

            checkBackground(jobList);
            cout<<"logout"<<endl;
            exit(atoi(newJob.progs[0].argv[1]));
          }
        }
        else
        {
            if(ncurses_ver)
            {
                clear();
                delwin(menubar);
                delwin(messagebar);
                endwin();
            }

         checkBackground(jobList);
         cout<<"logout"<<endl;
        exit(0);
        }
    }
    else if (!strcmp(newJob.progs[0].argv[0], "jobs"))//jobs: tipwnei oles tis diergasies
pou ektelounde parallila.
    {                                                 // Tipwnei to status tis diergasias, to ID
tis kai to onoma tis diergasias

        for (job = jobList->head; job; job = job->next)
        {
          if (job->runningProgs == job->stoppedProgs)
             statusString = "Stopped";
          else
             statusString = "Running";

          cout<<"["<<job->jobId<<"]"<<statusString<<"\t"<<job->text<<endl;
        }
        return 0;
    }
    else if (!strcmp(newJob.progs[0].argv[0], "cd"))   //cd: Allazei directory.
    {
        char *path;
        if (newJob.progs[0].argv[1]!=NULL)             //An o xristis dwsei directory
meta tin entoli cd, tote tha allaksei se afto to directory
          path = newJob.progs[0].argv[1];
        else if ((path = getenv("HOME")) == NULL)  //An o xristis den dwsei directory
meta tin entoli cd (diladi dwsei sketi tin entoli cd)
          path = ".";                                  //tote allazoume sto HOME
directory tou xristi. An den exei oristei tetoio directory
```

```
                                        //den allazoume directory katholou. Etsi
leitourgei kai to bash.

        if (chdir(path) == -1)                          //Otan epistrefei -1 den katafere
na allaksei directory. Tipwnoume minima lathous
          cout<<path<<" : "<<strerror(errno)<<endl;
        else
        myprompt.refresh_prompt();              //Afou egine allagi sto current directory,
allazoume kai to prompt antistixa.

        return 0;
    }
    else if (!strcmp(newJob.progs[0].argv[0], "list"))  //list: Tipwnei mia lista me oles
tis eggrafes tou history
    {
        if(newJob.progs[0].argv[1])
        {
          if(!strcmp(newJob.progs[0].argv[1],"-c"))
          {
                clear_history();
                return 0;
          }
          else if(!strcmp(newJob.progs[0].argv[1],"-m"))
          {
                if (newJob.progs[0].argv[2])
                {
                        int max = atoi(newJob.progs[0].argv[2]);

                        if((max>=0) && (max<=300))
                                stifle_history(max);
                        else
                                cerr<<"Error: Please insert a positive integer number
less than 300!"<<endl;
                        return 0;
                }
                else
                {
                        cerr<<"Error: You must enter a value!\n"<<endl;
                        cerr<<"list\n\t[-c]\t\tClears history list\n\t[-m VALUE]\tDefine
the maximum entries in history list"<<endl;
                        return 0;
                }
          }
          else
          {
                cerr<<"Error: Unknown option:
"<<newJob.progs[0].argv[1]<<"\n"<<endl;
                cerr<<"list\n\t[-c]\t\tClears history list\n\t[-m VALUE]\tDefine the
maximum entries in history list"<<endl;
                return 0;
```

```
        }
      }
      HIST_ENTRY **list;
      int j;

      list = history_list ();
      if (list)
      {
         for (j = 0; list[j]; j++)
        cout<<j+1<<": "<<list[j]->line<<endl;
      }

      return 0;
   }
   else if (!strcmp(newJob.progs[0].argv[0], "fg") || !strcmp(newJob.progs[0].argv[0],
"bg"))
   {
      if (!newJob.progs[0].argv[1] || newJob.progs[0].argv[2]) {
        cerr<<"Error: "<<newJob.progs[0].argv[0]<<" : exactly one argument is
expected"<<endl;
        return 1;
      }

      if (sscanf(newJob.progs[0].argv[1], "%%%d", &jobNum) != 1)
       {

        cerr<<"Error: "<<newJob.progs[0].argv[0]<<",
"<<newJob.progs[0].argv[1]<<" : bad argument"<<endl;

        return 1;
      }

      for (job = jobList->head; job; job = job->next)
        if (job->jobId == jobNum)
        break;

      if (!job)
       {
        cerr<<"Error: "<<newJob.progs[0].argv[0]<<": unknown job
"<<jobNum<<endl;
        return 1;
      }

      if (*newJob.progs[0].argv[0] == 'f')
       {
        /* Make this job the foreground job */
        if (tcsetpgrp(0, job->pgrp))
          perror("tcsetpgrp");
        jobList->fg = job;
      }
```

```
        /* Restart the processes in the job */
        for (i = 0; i < job->numProgs; i++)
           job->progs[i].isStopped = 0;

        kill(-job->pgrp, SIGCONT);

        job->stoppedProgs = 0;

        return 0;
    }
    else if (!strcmp(newJob.progs[0].argv[0], "set"))
    {
        if (newJob.progs[0].argv[1] != NULL)
            cerr<<"Error: No argument expected"<<endl;
        else
            for (int g = 0; environ[g] != NULL; g++)
               cout<<environ[g]<<endl;

        return 0;
    }
    else if ( (isAssignment(newJob.progs[0].argv)) &&
(strcmp(newJob.progs[0].argv[0], "alias")))
    {
        asg(newJob.progs[0].argv,0);

        return 0;
    }
    else if (!strcmp(newJob.progs[0].argv[0], "alias"))
    {
        asg(newJob.progs[0].argv,1);
        return 0;
    }
    else if (!strcmp(newJob.progs[0].argv[0], "unalias"))
    {
        if((newJob.progs[0].argv[1]==NULL) || (newJob.progs[0].argv[2]!=NULL))
          cout<<"Error: One argument expected"<<endl;
        else
          Unalias(newJob.progs[0].argv[1]);

        return 0;
    }
    else if (!strcmp(newJob.progs[0].argv[0], "aliases"))
    {
        DisplayAliases();
        return 0;
    }
    else if (!strcmp(newJob.progs[0].argv[0], "pushd"))
    {
        if(newJob.progs[0].argv[1])
```

```
    {
       cerr<<"Error: No argument expected"<<endl;
      return 0;
    }

    dir_stack.push(get_cwd(false));

    get_cwd(true);

    return 0;
}
else if (!strcmp(newJob.progs[0].argv[0], "popd"))
{
    if(newJob.progs[0].argv[1])
    {
       cerr<<"Error: No argument expected"<<endl;
      return 0;
    }

    if(!dir_stack.empty())
    {
      char *path = new char[dir_stack.top().length()+1];

      if (path==NULL)
      {
              cerr<<"Error: Could not get enough memory for path"<<endl;
              exit(2);
      }

      path = (char*)dir_stack.top().c_str();

      path[dir_stack.top().length()]='\0';

      if (chdir(path) == -1)
        cout<<path<<" : "<<strerror(errno)<<endl;
      else
        myprompt.refresh_prompt();

      delete path;

      dir_stack.pop();
    }
    else
      cerr<<"Error: Directory stack empty"<<endl;

    return 0;
}
else if (!strcmp(newJob.progs[0].argv[0], "dirstack"))
{
    if(dir_stack.empty())
```

```
        {
          cout<<"The Directory stack is empty"<<endl;
          return 0;
        }
        else
        {
          int p=0;
          stack<string> temp = dir_stack;

          bool numbers = false;

          if(newJob.progs[0].argv[1])
          {
          if(newJob.progs[0].argv[1][0]=='-')
          {
                  if(newJob.progs[0].argv[1][1]=='v')
                          numbers=true;
                  else
                  {
                          cerr<<"Error: Unknown option:
"<<newJob.progs[0].argv[1]<<endl;
                          return 0;
                  }
          }
          }
          while(!temp.empty())
          {
                  if(numbers)
                          cout<<p++<<"\t";

                  cout<<temp.top()<<endl;
                  temp.pop();
          }
        }
        return 0;
    }
        /*************************************************************/

    /*  Afou ftasame edw, ara den einai built-in entoli. Prepei na einai ekswteriko
arxeio. */



    nextin = 0, nextout = 1;
    for (i = 0; i < newJob.numProgs; i++)
    {
       if ((i + 1) < newJob.numProgs)
        {
          pipe(pipefds);
          nextout = pipefds[1];
```

```
    }
     else
        nextout = 1;

    if (!(newJob.progs[i].pid = fork()))
     {
        signal(SIGTTOU, SIG_DFL);

        if (nextin != 0)
         {
            dup2(nextin, 0);
            close(nextin);
        }

        if (nextout != 1)
         {
            dup2(nextout, 1);
            close(nextout);
        }

        setupRedirections(newJob.progs + i);

        execvp(newJob.progs[i].argv[0], newJob.progs[i].argv);

     /*Afou ftasame ws edw, to execvp apetixe.
      Tipwnw ena minima lathous kai termatizw tin diergasia. Oxi to shell.*/

        cerr<<newJob.progs[i].argv[0]<<": "<<strerror(errno)<<endl;
        exit(1);
    }

    setpgid(newJob.progs[i].pid, newJob.progs[0].pid);

    if (nextin != 0)
        close(nextin);
    if (nextout != 1)
        close(nextout);

    nextin = pipefds[0];
}
newJob.pgrp = newJob.progs[0].pid;

newJob.jobId = 1;
for (job = jobList->head; job; job = job->next)
    if (job->jobId >= newJob.jobId)
        newJob.jobId = job->jobId + 1;

if (!jobList->head) {
    job = jobList->head = (struct job*)malloc(sizeof(*job));
}
```

```
    else
    {
        for (job = jobList->head; job->next; job = job->next);
        job->next = (struct job*)malloc(sizeof(*job));
        job = job->next;
    }

    *job = newJob;
    job->next = NULL;
    job->runningProgs = job->numProgs;
    job->stoppedProgs = 0;

    if (inBg)
        cout<<"["<<job->jobId<<"] "<<newJob.progs[newJob.numProgs -
1].pid<<endl;
    else
    {
        jobList->fg = job;
        if (tcsetpgrp(0, newJob.pgrp))
            perror("tcsetpgrp");
    }

    return 0;
}

void removeJob(struct jobSet * jobList, struct job * job)
{
    struct job * prevJob;

    freeJob(job);
    if (job == jobList->head)
        jobList->head = job->next;
    else
    {
        prevJob = jobList->head;
        while (prevJob->next != job)
            prevJob = prevJob->next;
        prevJob->next = job->next;
    }

    free(job);
}

void checkJobs(struct jobSet * jobList) {
    struct job * job;
    pid_t childpid;
    int status;
    int progNum;

    while ((childpid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
```

```
    {
      for (job = jobList->head; job; job = job->next)
       {
         progNum = 0;
         while (progNum < job->numProgs && job->progs[progNum].pid != childpid)
            progNum++;
         if (progNum < job->numProgs)
         break;
       }

       if (WIFEXITED(status) || WIFSIGNALED(status))
        {
         job->runningProgs--;
         job->progs[progNum].pid = 0;

         if (!job->runningProgs)
          {
            cout<<"["<<job->jobId<<"] Done\t\t"<<job->text<<endl;
            removeJob(jobList, job);
         }
       }
        else
        {
         job->stoppedProgs++;
         job->progs[progNum].isStopped = 1;

         if (job->stoppedProgs == job->numProgs)
            cout<<"["<<job->jobId<<"] Stopped\t\t"<<job->text<<endl;
       }
    }

    if (childpid == -1 && errno != ECHILD)
       perror("waitpid");
}

void init_and_check()
{
       if(!isatty(fileno(stdout)))
       {
         cerr<<"Error: You are not a terminal"<<endl;
         exit(2);
       }

       if (getenv("PATH") == NULL)
         putenv("PATH=/usr/X11R6/bin:/usr/local/bin:/bin:/usr/bin");

       putenv("SHELL=wish");

       /* Pernw to pid tou shell kai to ppid. Den xriazete na elenksw gia lathos, giati
          oi sinartiseis aftes den epistrefoune pote error*/
```

```
        shell_vars.pid = getpid();

        shell_vars.ppid = getppid();

        shell_vars.shell_name = "wish- Windows Interactive SHell";

        shell_vars.version = 1.0;

        shell_vars.version_state = "beta";

        checkProfile();

        Read_ConfigFile(".wish_profile");       //Diavazw to profile arxeio, kai fortwnw
tis environmental metavlites

        Read_ConfigFile("aliases");      //Diavazw to profile arxeio, kai fortwnw tis
environmental metavlites

        if (Find_From_Map("GREET")=="on")
          cout<<"** "<<shell_vars.shell_name<<" v"<<shell_vars.version<<"
**"<<endl;
}

int main(int argc, char ** argv)
{
   string command;
   char * nextCommand = NULL;
   struct jobSet jobList = { NULL, NULL };
   struct job newJob;
   istream *input = &cin;
   int i;
   int status;
   int inBg;
   int opt;
   int key;
   int selected_item;
   extern map<int,string> Commands_Menu_String;
   extern map<int,string> Builtin_Menu_String;

   extern int menu_number;
   int isbreak=0;

   if (argv[1])
   {
       if(argv[1][0]!='-')
       {
         cerr<<"Error: Bad syntax!\n"<<endl;
         init_and_check();
         usage(shell_vars);
```

```
      exit(0);
    }
}
ncurses_ver = false;

ignore_sig();

    while((opt=getopt(argc,argv,"hf:vn"))!=-1)
    {
      switch(opt)
      {
              case 'h':
                      init_and_check();
                      usage(shell_vars);
                      exit(0);
              case 'f':
                      input = new ifstream(optarg);

                      if (! *input)
                      {
                              cerr<<"Error: No such file or directory.\n";
                              exit (1);
                      }
                      break;
              case 'v':
                      init_and_check();
                      cout<<shell_vars.shell_name<<" - version: ";
                      cout.setf(ios_base::showpoint);
                      cout.precision(2);
                      cout<<shell_vars.version;
                      cout.unsetf(ios_base::showpoint);
                      cout.precision();
                      cout<<" "<<shell_vars.version_state<<endl;
                      cout<<"Giannakidis Apostolis - 2005"<<endl;
                      exit(0);
              case 'n':
                      ncurses_ver = true;
                      break;
              case ':':
              case '?':
                      exit(0);
              default :
                      break;
      }
    }

init_and_check();

prompt myprompt;
```

```
    if(ncurses_ver)
    {
        init_curses();
        bkgd(COLOR_PAIR(1));
        menubar=subwin(stdscr,1,80,0,0);
        messagebar=subwin(stdscr,1,79,23,1);
        draw_menubar(menubar);
        move(2,1);
        printw("Press F1 or F2 to open the menus. ESC to close them");
        refresh();
    }

    while (1)
    {
       if (!jobList.fg)
        {
          checkJobs(&jobList);

          if (!nextCommand)
           {
          if(!ncurses_ver)
          {
              if (getCommand(input, command,myprompt))
                  break;
          }
          else
          {
                  if(isbreak)
                  {
                          isbreak=0;
                          textcolor(RESET, CYAN, BLACK);
                          cout<<"\nPress any key to return . . ."<<endl;
                          textcolor(RESET, WHITE, BLACK);
                          reset_prog_mode();     // return to previous tty curses mode
before we call getch()
                  }

              do
              {
          key=getch();
                werase(messagebar);
                wrefresh(messagebar);
                if ((key==KEY_F(1)) || (menu_number==1))
                   {
                   menu_items=draw_first_menu(0,messagebar);
                      selected_item=scroll_first_menu(menu_items,7,0, messagebar);
                              delete_menu(menu_items,8);
                      free(menu_items);
                              touchwin(stdscr);
                              refresh();
```

```
}
  else if ((key==KEY_F(2)) || (menu_number==2))
  {
                menu_items=draw_second_menu(20,messagebar);
     selected_item=scroll_second_menu(menu_items,7,20, messagebar);
     delete_menu(menu_items,8);
        free(menu_items);
         touchwin(stdscr);
                refresh();
  }
  else if ((key==KEY_F(3)) || (menu_number==3))
  {
                menu_items=draw_third_menu(40,messagebar);
     selected_item=scroll_third_menu(menu_items,2,40, messagebar);
     delete_menu(menu_items,3);
        free(menu_items);
         touchwin(stdscr);
                refresh();
  }

}while (selected_item<0);

if(menu_number==1)
{
        bool selected=false;
                if(selected_item+1==4)
                {
                        int max_width=0, heigth=11;
                        popup_window = newwin(5,30,9,22);
                        box(popup_window,'|','-');
                        mvwprintw(popup_window,1,2,"%s","Please specify a
text file: ");
                        wbkgd(popup_window,COLOR_PAIR(2));
                        werase(messagebar);
                        wprintw(messagebar,"A text file for less to output");
                        wrefresh(popup_window);
                        wrefresh(messagebar);
                        echo();
                      init_pair(2,COLOR_BLUE,COLOR_WHITE);
                        attrset(COLOR_PAIR(2));
                        move(heigth,25);
                        command="less ";
                        do
                        {
                                if((heigth<12)&&(max_width<23))
                                {
                                        key=getch();
                                        command+=key;
                                        max_width++;
                                        if(max_width==22)
```

```
                                        {
                                                heigth++;
                                                move(heigth,25);
                                                max_width=0;
                                        }
                                }
                                else
                                {
                                        key=getch();
                                        command[command.length()]=key;
                                        move(14,25);
                                }
                        }while((key!=10) && (key!=27));
                        if(key!=27)
                                selected=true;
                        noecho();
                       werase(messagebar);
                       wrefresh(messagebar);
                        werase(popup_window);
                       wrefresh(popup_window);
                        delwin(popup_window);
                }
                else if(selected_item+1==6)
                {
                        int max_width=0, heigth=11;
                        popup_window = newwin(7,30,9,22);
                        box(popup_window,'|','-');
                        mvwprintw(popup_window,1,2,"%s"," Please insert a
command: ");
                        wbkgd(popup_window,COLOR_PAIR(2));
                        werase(messagebar);
                        wprintw(messagebar,"The command may be an external
file or a builtin command");
                        wrefresh(popup_window);
                        wrefresh(messagebar);
                        echo();
                      init_pair(2,COLOR_BLUE,COLOR_WHITE);
                        attrset(COLOR_PAIR(2));
                        move(heigth,25);
                        command="";
                        do
                        {
                                if((heigth<14)&&(max_width<23))
                                {
                                        key=getch();
                                        command+=key;
                                        max_width++;
                                        if(max_width==22)
                                        {
                                                heigth++;
```

```
                                        move(heigth,25);
                                        max_width=0;
                                }
                        }
                        else
                        {
                                key=getch();
                                command[command.length()]=key;
                                move(14,25);
                        }
                }while((key!=10) && (key!=27));
                if(key!=27)
                        selected=true;
                noecho();
            werase(messagebar);
            wrefresh(messagebar);
                werase(popup_window);
            wrefresh(popup_window);
                delwin(popup_window);
    }
    else
    {
            command=Commands_Menu_String[selected_item+1];
            selected=true;
    }

attrset(COLOR_PAIR(1));
move(11,24);
printw("%s","                                        ");
move(12,24);
printw("%s","                                        ");
move(13,24);
printw("%s","                                        ");
touchwin(stdscr);
        refresh();

if(selected)
{
  def_prog_mode();
  endwin();
  isbreak=1;
  if ((command.length()!=0) && (command[0]!='!'))
  {
        if (last_command!=command)
        {
                last_command=command;
                add_history (command.c_str());
        }
  }
  ReplaceAliases(command);
```

```
                if(command.find("$")!=string::npos)
                        ReplaceVariables(command);

                if (command[0]=='!')
                        quick_command(command);

                textcolor(BRIGHT, CYAN, BLACK);
                cout<<"\n"<<myprompt.get_prompt();
                textcolor(RESET, WHITE, BLACK);
                cout<<command<<endl;
              }
              else
              {
                        command="";
                        isbreak=0;
              }

              selected_item=-1;
              menu_number=0;
            }
            else if(menu_number==2)
            {
                 bool selected=false;
                        if(selected_item+1==2)
                 {
                              int max_width=0;
                              popup_window = newwin(5,42,9,22);
                              box(popup_window,'|','-');
                              mvwprintw(popup_window,1,2,"%s","Insert a
command and its' alias: ");
                              wbkgd(popup_window,COLOR_PAIR(2));
                              werase(messagebar);
                              wprintw(messagebar,"Please take care of the alias
format: command=alias");
                              wrefresh(popup_window);
                              wrefresh(messagebar);
                              echo();
                            init_pair(2,COLOR_BLUE,COLOR_WHITE);
                              attrset(COLOR_PAIR(2));
                              move(11,24);
                              command="alias ";
                              do
                              {
                                        if(max_width<35)
                                        {
                                                  key=getch();
                                                  command+=key;
                                                  max_width++;
                                        }
```

```
                                        else
                                        {
                                                key=getch();
                                                if ((key!=10) && (key!=27))
                                                command[command.length()-1]=key;
                                                move(11,59);
                                        }
                                    }while((key!=10) && (key!=27));
                                    if(key!=27)
                                            selected=true;
                                    noecho();
                                  werase(messagebar);
                                  wrefresh(messagebar);
                                    werase(popup_window);
                                  wrefresh(popup_window);
                                    delwin(popup_window);
                            }
                            else if(selected_item+1==3)
                            {
                                    int max_width=0;
                                    popup_window = newwin(5,25,9,22);
                                    box(popup_window,'|','-');
                                    mvwprintw(popup_window,1,2,"%s"," Alias to delete:
");
                                    wbkgd(popup_window,COLOR_PAIR(2));
                                    werase(messagebar);
                                    wprintw(messagebar,"You only need to insert the
command and not its' alias");
                                    wrefresh(popup_window);
                                    wrefresh(messagebar);
                                    echo();
                                 init_pair(2,COLOR_BLUE,COLOR_WHITE);
                                    attrset(COLOR_PAIR(2));
                                    move(11,25);
                                    command="unalias ";
                                    do
                                    {
                                            if(max_width<19)
                                            {
                                                    key=getch();
                                                    command+=key;
                                                    max_width++;
                                            }
                                            else
                                            {
                                                    key=getch();
                                                    if ((key!=10) && (key!=27))
                                                    command[command.length()-1]=key;
                                                    move(11,44);
                                            }
```

```
                }while((key!=10) && (key!=27));
                if(key!=27)
                        selected=true;
                noecho();
              werase(messagebar);
              wrefresh(messagebar);
                werase(popup_window);
              wrefresh(popup_window);
                delwin(popup_window);
   }
   else
   {
          command=Builtin_Menu_String[selected_item+1];
          selected=true;
   }

attrset(COLOR_PAIR(1));
move(11,24);
printw("%s","                                        ");
move(12,24);
printw("%s","                                        ");
move(13,24);
printw("%s","                                        ");
touchwin(stdscr);
        refresh();

if(selected)
{
  def_prog_mode();
  endwin();
  isbreak=1;
  if ((command.length()!=0) && (command[0]!='!'))
  {
        if (last_command!=command)
        {
                last_command=command;
                add_history (command.c_str());
        }
  }
  ReplaceAliases(command);

  if(command.find("$")!=string::npos)
        ReplaceVariables(command);

  if (command[0]=='!')
        quick_command(command);

  textcolor(BRIGHT, CYAN, BLACK);
  cout<<"\n"<<myprompt.get_prompt();
  textcolor(RESET, WHITE, BLACK);
```

```
                cout<<command<<endl;
            }
            else
            {
                    command="";
                    isbreak=0;
            }

            selected_item=-1;
            menu_number=0;

        }
        else if(menu_number==3)
        {

                    if(selected_item+1==1)
                    {
                        popup_window = newwin(7,37,9,25);
                        box(popup_window,'|','-');
                        mvwprintw(popup_window,2,2,"%s"," wish - Windows
Interactive SHell");
                        mvwprintw(popup_window,3,2,"%s","  Giannakidis
Apostolis");
                        mvwprintw(popup_window,4,2,"%s","       (c) 2005");
                        wbkgd(popup_window,COLOR_PAIR(2));
                        werase(messagebar);
                        wrefresh(messagebar);
                        wprintw(messagebar,"Press ESC to close the About
window");
                        wrefresh(popup_window);
                        wrefresh(messagebar);
                        do{
                        key=getch();
                        } while((key!=27) && (key!=10));
                      werase(messagebar);
                      wrefresh(messagebar);
                        delwin(popup_window);
                    }
                    else if (selected_item+1==2)
                    {
                        popup_window = newwin(5,37,9,25);
                        box(popup_window,'|','-');
                        mvwprintw(popup_window,1,2,"%s"," wish - Windows
Interactive SHell");
                        mvwprintw(popup_window,2,2,"%s","  Version 1.0 -
Final");
                        mvwprintw(popup_window,3,2,"%s","    Ncurses
Version");
                        wbkgd(popup_window,COLOR_PAIR(2));
                        werase(messagebar);
```

```
                                wrefresh(messagebar);
                                wprintw(messagebar,"Press ESC to close the Version
window");
                                wrefresh(popup_window);
                                wrefresh(messagebar);
                                do{
                                key=getch();
                                } while((key!=27) && (key!=10));
                              werase(messagebar);
                              wrefresh(messagebar);
                                delwin(popup_window);
                          }

                           touchwin(stdscr);
                                  refresh();
                         isbreak=0;
                         command="";
                         selected_item=-1;
                         menu_number=0;
            }
          }

            nextCommand = (char*)command.c_str();
          }

          if (!parseCommand(&nextCommand, &newJob, &inBg) &&
newJob.numProgs)
            runCommand(newJob, &jobList, inBg, myprompt);
        }
        else
        {
          i = 0;
          while (!jobList.fg->progs[i].pid || jobList.fg->progs[i].isStopped)
          i++;

          waitpid(jobList.fg->progs[i].pid, &status, WUNTRACED);

          if (WIFEXITED(status) || WIFSIGNALED(status))
           {
            jobList.fg->runningProgs--;
            jobList.fg->progs[i].pid = 0;

            if (!jobList.fg->runningProgs)
          {
              removeJob(&jobList, jobList.fg);
              jobList.fg = NULL;

              if (tcsetpgrp(0, getpid()))
                 perror("tcsetpgrp");
            }
```

```
            }
             else
             {
               jobList.fg->stoppedProgs++;
               jobList.fg->progs[i].isStopped = 1;

               if (jobList.fg->stoppedProgs == jobList.fg->runningProgs)
            {
                  cout<<"["<<jobList.fg->jobId<<"] Stopped\t\t"<<jobList.fg->text<<endl;
                  jobList.fg = NULL;
               }
            }

            if (!jobList.fg)
             {
               if (tcsetpgrp(0, getpid()))
                  perror("tcsetpgrp");
            }
         }

      }

         if(ncurses_ver)
         {
           clear();
           delwin(menubar);
           delwin(messagebar);
           endwin();
         }

      exit(EXIT_SUCCESS);
}
```

/***/

Το αρχείο shell_functions.h

/***/

```
/****************************************
        Wish Unix Shell - Version 1.0
        Author: Giannakidis Apostolis
        Id: shell_functions.h
****************************************/

#include <string>
#include <sys/types.h>

using namespace std;

struct jobSet;

struct shell_variables              //To struct afto krataei merikes metavlites oi opoies
exoun sxesi me tin taftotita tou shell
{
        int pid;
        int ppid;
        string shell_name;
        string shell_path;
        double version;
        string version_state;
};

void textcolor(int attr, int fg, int bg);

void checkProfile();

void usage(struct shell_variables);

void Alias(char **argv);

void Unalias(string name);

void DisplayAliases();

void ReplaceVariables(string & command);

void ReplaceAliases(string & command);

void Read_ConfigFile(char *);

string Find_From_Map(string);

void quick_command(string &command);
```

```
void asg(char *argv[], int);

void checkBackground(struct jobSet * jobList);

void logout();
```

/***/

Το αρχείο shell_functions.cpp

/***/

```
/*****************************************
        Wish Unix Shell - Version 1.0
        Author: Giannakidis Apostolis
        Id: shell_functions.cpp
*****************************************/

#include <string>
#include <cstring>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <readline/readline.h>
#include <readline/history.h>
#include "shell_functions.h"
#include <unistd.h>
#include <sys/types.h>
#include <algorithm>
#include <vector>
#include <signal.h>
#include "jobs.h"

using namespace std;

extern HIST_ENTRY **history_list ();

map<string,string> Prof_vars;
map<string,string> Aliases;

void textcolor(int attr, int fg, int bg)
{       char command[13];

        /* Command is the control command to the terminal */
        sprintf(command, "%c[%d;%d;%dm", 0x1B, attr, fg + 30, bg + 40);
        printf("%s", command);
}

void checkProfile()
{
        ifstream infile(".wish_profile");

        if (!infile)
        {
          ofstream outfile(".wish_profile");
```

```
            outfile<<"# .wish_profile\n";
            outfile<<"# User specific environment variables\n";
            outfile<<"# format : VARIABLE value\n";
            outfile<<"DIR on\n";
            outfile<<"PS1 %\n";
            outfile<<"FULLPATH off\n";
            outfile<<"GREET off\n";

            outfile.close();
        }
        else
            infile.close();

}

vector<string> TokenizeVariables(string command)
{
        vector<string> vars;
        size_t pos = command.find("$");
        string temp = command;

        while (pos!=string::npos)
        {
            temp = command.substr(pos);
            command.erase(pos,1);
            pos = temp.find(" ");
            if (pos!=string::npos)
            {
                    temp = temp.substr(1,pos-1);
                    pos = command.find("$",pos);
            }
            else
              temp.erase(0,1);

            if(!isalnum(temp[temp.length()-1]) )
              temp = temp.erase(temp.length()-1,1);
            vars.push_back(temp);
        }

        return vars;
}

void replaceAll( string & source, const string & find, const string & replace )
{
        size_t j;
        for (;(j = source.find( find )) != string::npos;)
        {
        source.replace( j, find.length(), replace );
        }
}
```

```
void ReplaceVariables(string & command)
{
	vector<string> vars;

	vars = TokenizeVariables(command);

	int i=0;

	vector<string>::iterator p;

	for(p=vars.begin();p!=vars.end();++p)
	{
	  char *name = (char*)p->c_str();

	      char *value = getenv(name);
	  if(value==NULL)
	        command.replace(command.find("$"+*p),p->length()+1,"");
	  else
	        command.replace(command.find("$"+*p),p->length()+1,value);
	}

	replaceAll(command," \\+ ","");

}

void usage(struct shell_variables shell_vars)
{
	cout<<shell_vars.shell_name<<" - version: ";
	cout.setf(ios_base::showpoint);
	cout.precision(2);
	cout<<shell_vars.version;
	cout.unsetf(ios_base::showpoint);
	cout.precision();
	cout<<" "<<shell_vars.version_state<<endl;
	cout<<"Giannakidis Apostolis - 2005"<<endl;

	cout<<"\nUsage: wish [<OPTIONS>] [<ARGUMENTS>] ..."<<endl;
	cout<<endl;
	cout<<"Shell options:"<<endl;
	cout<<"  -h\t\tshow this help message, then exit"<<endl;
	cout<<"  -v\t\tshow wish version number, then exit"<<endl;
	cout<<"  -f FILENAME\tSpecifies a filename to execute, then exits"<<endl;
	cout<<"  -n\t\tNcurses version of the shell"<<endl;
	cout<<"\nTip: Using the Ncurses version you can use menus to execute
commands"<<endl;
}

/* I sinartisi afti dimiourgei ena alias  kai tou dinei timi h  allazei tin timi enos
iparxondos alias */
```

```
void Alias(string name, string value)
{
        name.erase(0,5);

        ifstream infile("aliases",ios::in);

        if (!infile)
        {
          cerr<<"Error: can't open aliases file: "<<endl;
          exit (2);
        }

        string line,text;
        bool replace=false;

        while(!infile.eof())
        {
          getline(infile,line,'\n');

          string alias_name = line.substr(0,line.find_first_of(":"));

          size_t n = alias_name.find_first_not_of(" \t");
          size_t k = alias_name.find_last_not_of(" \t");

          if(n!=string::npos)
          {
                  if(k!=string::npos)
                    alias_name = alias_name.substr(n, k-n+1);
                  else
                    alias_name = alias_name.substr(n);
          }
          else
          {
                  if(k!=string::npos)
                    alias_name = alias_name.substr(0, k+1);
          }

          if(alias_name!=name)
                  text+=line+"\n";
          else
          {
                  text.erase(text.length()-1,1);
                  text+=alias_name+" : "+value+"\n";
                  replace=true;
          }
        }
        if (!replace)
        {
          cout<<"New alias <"<<name<<" : "<<value<<"> entered"<<endl;
```

```
            text.erase(text.length()-1,1);
            text+=name+" : "+value+"\n";
            Aliases.insert(make_pair(name,value));
        }
        else
        {
            cout<<"Alias <"<<name<<"> changed value to "<<value<<endl;
            Aliases[name]=value;
        }

        infile.close();

        ofstream outfile("aliases",ios::out);

        if (!outfile)
        {
            cerr<<"Error: can't open aliases file"<<endl;
            exit (2);
        }

        outfile<<text;
        outfile.close();
}

/* I sinartisi afti diagrafei ena alias apo to arxeio aliases */

void Unalias(string name)
{
        ifstream infile("aliases",ios::in);

        if (!infile)
        {
            cerr<<"Error: can't open aliases file: "<<endl;
            exit (2);
        }

        string line,text;
        bool erased=false;
        string alias_name;

        while(!infile.eof())
        {
            getline(infile,line,'\n');

            if (line[0]=='#')
            {
                    text+=line+"\n";
                    continue;
            }
```

```
  alias_name = line.substr(0,line.find_first_of(":"));

  size_t n = alias_name.find_first_not_of(" \t");
  size_t k = alias_name.find_last_not_of(" \t");

  if(n!=string::npos)
  {
          if(k!=string::npos)
            alias_name = alias_name.substr(n, k-n+1);
          else
            alias_name = alias_name.substr(n);
  }
  else
  {
          if(k!=string::npos)
            alias_name = alias_name.substr(0, k+1); //keep n-k+1 chars
  }

  if(alias_name!=name)
          text+=line+"\n";
  else
  {
          Aliases.erase(name);
          erased=true;
          cout<<"Alias: <"<<name<<"> erased!"<<endl;
          break;
  }
}

while(!infile.eof())
{
  getline(infile,line,'\n');
  text+=line+"\n";
}

infile.close();

if(!erased)
  cout<<"Alias <"<<name<<"> not found. No changes were made"<<endl;

ofstream outfile("aliases",ios::out);

if (!outfile)
{
  cerr<<"Error: can't open aliases file: "<<endl;
  exit (2);
}

outfile<<text.substr(0,text.find_last_not_of("\n")+1);
outfile<<"\n";
```

```
        outfile.close();
}

void DisplayAliases()
{
        cout<<"Aliases:"<<endl;

        map<string, string>::iterator p;

        for(p=Aliases.begin();p!=Aliases.end();++p)
          cout<<p->first<<" : "<<p->second<<endl;
}

/* I sinaritisi afti kanei antikatastasi otan entopisei ena alias. */

void ReplaceAliases(string &command)
{
        istringstream iss(command);
        string first_command;

        iss>>first_command;

        if ((first_command=="alias") || (first_command=="unalias"))
          return;

        map<string, string>::iterator p;

        vector<string> match_array;

        size_t pos;

        for(p=Aliases.begin();p!=Aliases.end();++p)
        {
          pos=command.find(p->first);
          if(pos!=string::npos)
                match_array.push_back(p->first);
        }

        if (match_array.size()>0)
        {
          vector<string>::iterator i,max;
          int len=0;
          for(i=match_array.begin();i!=match_array.end();++i)
          {
                int tmp=i->length();
                if(tmp>len)
                {
                        len=tmp;
                        max=i;
                }
```

```
        }
        string from = max->c_str();
        string to = Aliases[from];

        command.replace(command.find(from),from.length(),to);
      }
}

void Read_ConfigFile(char *source) // I sinartisi afti diavazei .wish_profile kai to aliases kai fortwnei tis metavlites
{                                  // pou vriskei mesa sto arxeio.
      /*
      Diavazw to arxeio profile, gia na parw tis times pou edwse o xristis se times orismenwn metavlitwn
      */

      if (!((!strcmp(source,".wish_profile")) || (!strcmp(source,"aliases"))))
        return;

      ifstream file(source);

       if (!file)
       {
        cerr<<"Error: can't open config file: "<<source<<endl;
        exit (2);
       }

      int numTokens;
      string name,value;
      string str,token;
      istringstream iss;

      while(!file.eof())
      {
        getline(file,str,'\n');

        if(str[0]=='\n' || str.length()<=2 || str[0]=='#')
        {
                str.clear();
                continue;
        }

        iss.str(str);

        numTokens=0;
        name.clear();
        value.clear();
        if (!strcmp(source,"aliases"))
        {
                if(str.find(":")==string::npos)
```

```
                continue;

        char rep[50]="";
        iss>>token;
        while(token!=":")
        {
                name+=token;
                if(isspace(iss.peek()))
                        name+=" ";
                iss>>token;
        }

        if(isspace(name[name.length()-1]))
          name = name.substr(0,name.find_last_of(" "));

        while(iss>>token)
        {
                value+=token;
                if(isspace(iss.peek()))
                        value+=" ";
        }
        if(isspace(value[value.length()-1]))
          value = value.substr(0,value.find_last_of(" "));

        Aliases.insert(make_pair(name,value));

}
else if (!strcmp(source,".wish_profile"))
{
 while(iss.good())
 {
                iss>>token;

                if(token.length()==0)
                        continue;

                ++numTokens;
                if(numTokens==1)
                        name=token;
                else if(numTokens==2)
                        value=token;

                token.clear();
 }

 if(numTokens==2)
 {
        if (!strcmp(source,".wish_profile"))
        {
                Prof_vars.insert(make_pair(name,value));
```

```
                        char *env_var = new char[name.length()+value.length()+2];
                        strcpy(env_var,name.c_str());
                        strcat(env_var,"=");
                        strcat(env_var,value.c_str());
                        putenv(env_var);
                }
            }
          }
          iss.clear();
        }
        file.close();
}

string Find_From_Map(string s)
{
        map<string, string>::iterator it = Prof_vars.find(s);
        if(it!=Prof_vars.end())
          return it->second;

        return NULL;
}

void quick_command(string &command)
{
        HIST_ENTRY **list;
        list = history_list ();

        istringstream iss(command);
        int i=0;

        string quick;
        string test;

        iss>>quick;
        iss>>test;

        if(quick=="!!")
        {
          if(test.length()>0)
          {
                command=" ";
                cerr<<"Error: No arguments are expected!"<<endl;
                return;
          }

          while(list[i]) i++;
                i--;
                command=list[i]->line;
        }
        else if((quick[0]=='!') && (isdigit(quick[1])))
```

```
        {
          if(test.length()>0)
          {
                  command=" ";
                  cerr<<"Error: No arguments are expected!"<<endl;
                  return;
          }

          quick.erase(0,1);
          int num=atoi(quick.c_str());
          num--;

          if(history_length>=(num+1))
            command=list[num]->line;
          else
          {
                  command=" ";
                  cout<<"There is no such history entry!"<<endl;
          }
        }
}

/* H sinaritisi afti kanei ekxwrisi timis se mia metavliti perivalondos h se ena alias */

void asg(char *argv[], int mode)
{
        string name, value, command;

        int i=0;

        while (argv[i])
        {
         command+=argv[i];
         i++;
        }

        int name_index = command.find_first_of('=');

        if (name_index==string::npos)
        {
          cerr<<"Error: Bad syntax"<<endl;
          return;
        }

        name = command.substr(0,name_index);
        value = command.substr(name_index+1, command.length());

        if (name.length()==0 || value.length()==0)
          cerr<<"Error: Bad syntax"<<endl;
        else
```

```
        {
          if(mode==0)
                  setenv(name.c_str(), value.c_str(), true);
          else if (mode==1)
          Alias(name,value);
        }
}

void killJobs(struct jobSet * jobList)
{
        struct job * job;
        int i;
        for (job = jobList->head; job; job = job->next)
        {
          for(i=0;i<job->runningProgs;i++)
                  kill(job->progs[i].pid,SIGKILL);
        }
}

void checkBackground(struct jobSet * jobList)
{
        struct job * job;
        int running=0;
        for (job = jobList->head; job; job = job->next)
          running++;

        if(running>0)
        {
          char *buffer = (char*)NULL;

           while(1)
           {
            buffer=(char*)NULL;
            buffer = readline("\nThere are jobs running at the background!Do you wish
to kill them? [Y/N]: ");

             if (buffer != (char*)NULL)
             {
          if (buffer[0]=='Y' || buffer[0]=='y')
          {
                    killJobs(jobList);
           break;
          }
          else if (buffer[0]=='N' || buffer[0]=='n')
                    break;
            }
           }
        }
}
```

```
void logout()
{
        char *buffer = (char*)NULL;

        while(1)
        {
          buffer=(char*)NULL;
          buffer = readline("\nTerminate shell? (Y/N): ");

          if (buffer != (char*)NULL)
          {
           if (buffer[0]=='Y' || buffer[0]=='y')
                        exit(0);
           else if (buffer[0]=='N' || buffer[0]=='n')
                        break;
          }
        }
}
```

/***/

Το αρχείο prompt.h

/***/

```
/****************************************
        Wish Unix Shell - Version 1.0
        Author: Giannakidis Apostolis
        Id: prompt.h
****************************************/

#include <string>
#include "shell_functions.h"

class prompt                    //Mia klasi i opoia diaxeirizete to prompt tou shell
{
public:
        prompt();
        std::string get_prompt();
        void display_dir(bool status);
        void refresh_prompt();
private:
        void path_check();
        void name_check();
        std::string get_node_name();

        std::string name;       //To onoma tou user
        std::string node;       // To onoma tou mixanimatos
        std::string dir;                // O trexon katalogos

        // I timi pou pernoun oi parakatw metavlites eksartonte apo tis metavlites pou
vriskonde sto .yash_profile.
        // Kai o xristis mporei na tis allaksei.

        std::string ps1;
        bool dir_on;            // An einai true, tote tha emfanizete to directory sto
prompt. Allios den tha emfanizete.
};                              // I timi pou pernei eksartate apo tin metavliti DIR.
```

/***/

Το αρχείο jobs.h

/***/

```
/*****************************************
        Wish Unix Shell - Version 1.0
        Author: Giannakidis Apostolis
        Id: jobs.h
*****************************************/

#include <glob.h>

enum redirectionType { REDIRECT_INPUT, REDIRECT_OVERWRITE,
REDIRECT_APPEND };

struct jobSet {
   struct job * head;
   struct job * fg;
};

struct redirectionSpecifier {
   enum redirectionType type;
   int fd;
   char * filename;
};

struct childProgram {
   pid_t pid;
   char ** argv;
   int numRedirections;
   struct redirectionSpecifier * redirections;
   glob_t globResult;
   int freeGlob;
   int isStopped;
};

struct job {
   int jobId;
   int numProgs;
   int runningProgs;
   char * text;
   char * cmdBuf;
   pid_t pgrp;
   struct childProgram * progs;
   struct job * next;
   int stoppedProgs;
};
```

/***/

Το αρχείο curses_functions.h

/***/

```
/*****************************************
      Wish Unix Shell - Version 1.0
      Author: Giannakidis Apostolis
      Id: curses_functions.h
*****************************************/

#include <curses.h>
#include <map>
#include <string>
using namespace std;

void init_curses();
void draw_menubar(WINDOW *menubar);

WINDOW **draw_first_menu(int start_col, WINDOW *messagebar);

WINDOW **draw_second_menu(int start_col, WINDOW *messagebar);

WINDOW **draw_third_menu(int start_col, WINDOW *messagebar);

void delete_menu(WINDOW **items,int count);

int scroll_first_menu(WINDOW **items,int count,int menu_start_col, WINDOW
*messagebar);

int scroll_second_menu(WINDOW **items,int count,int menu_start_col, WINDOW
*messagebar);

int scroll_third_menu(WINDOW **items,int count,int menu_start_col, WINDOW
*messagebar);
```

/***/

Το αρχείο curses_functions.cpp

/***/

```
/****************************************
        Wish Unix Shell - Version 1.0
        Author: Giannakidis Apostolis
        Id: curses_functions.cpp
****************************************/

#include <curses.h>
#include <cstdlib>
#include <iostream>
#include <map>
#include <string>
#include "curses_functions.h"

using namespace std;

const int ENTER=10;
const int ESCAPE=27;

map<int,string> Commands_Menu_String;
map<int,string> Builtin_Menu_String;
map<int,string> About_Menu_String;

char *Command_Infos[]={"Directory file listing","Goes back one
directory","Displays the current directory",
            "Display the contents of a text file", "Executes the vim editor","Executes an
external command",
            "Exit wish shell"};

char *Builtin_Infos[]={"Displays the list with command aliases","Sets a new
alias","Deletes an alias",
            "Displays a list with the background jobs","Displays the history list",
            "Pops the directory from the top of the stack and sets the current directory",
            "Pushes the current directory to the stack"};

char *About_Infos[]={"Displays information about wish shell","Displays the version
of the wish shell"};

int menu_number;

void init_curses()
{
      initscr();
        if(!has_colors())
        {
         endwin();
         cerr<<"Error: no color support on this terminal!"<<endl;
```

```
        exit(1);
       }
       if(start_color() != OK)
       {
        endwin();
        cerr<<"Error: could not initialize colors!"<<endl;
        exit(1);
       }
      init_pair(1,COLOR_WHITE,COLOR_BLUE);
      init_pair(2,COLOR_BLUE,COLOR_WHITE);
      init_pair(3,COLOR_RED,COLOR_WHITE);
      curs_set(2);
      noecho();
      keypad(stdscr,TRUE);

       Commands_Menu_String.insert(make_pair(1,"ls -1"));
       Commands_Menu_String.insert(make_pair(2,"cd .."));
       Commands_Menu_String.insert(make_pair(3,"pwd"));
       Commands_Menu_String.insert(make_pair(4,"less"));
       Commands_Menu_String.insert(make_pair(5,"vim"));
       Commands_Menu_String.insert(make_pair(6,"external"));
       Commands_Menu_String.insert(make_pair(7,"exit"));

       Builtin_Menu_String.insert(make_pair(1,"aliases"));
       Builtin_Menu_String.insert(make_pair(2,"alias"));
       Builtin_Menu_String.insert(make_pair(3,"unalias"));
       Builtin_Menu_String.insert(make_pair(4,"jobs"));
       Builtin_Menu_String.insert(make_pair(5,"list"));
       Builtin_Menu_String.insert(make_pair(6,"popd"));
       Builtin_Menu_String.insert(make_pair(7,"pushd"));

       About_Menu_String.insert(make_pair(1,"About wish"));
       About_Menu_String.insert(make_pair(2,"Version"));

}

void draw_menubar(WINDOW *menubar)
{
      wbkgd(menubar,COLOR_PAIR(2));
      waddstr(menubar,"External");
      wattron(menubar,COLOR_PAIR(3));
      waddstr(menubar,"(F1)");
      wattroff(menubar,COLOR_PAIR(3));
      wmove(menubar,0,20);
      waddstr(menubar,"Builtin");
      wattron(menubar,COLOR_PAIR(3));
      waddstr(menubar,"(F2)");
      wattroff(menubar,COLOR_PAIR(3));
      wmove(menubar,0,40);
      waddstr(menubar,"About");
```

```
	wattron(menubar,COLOR_PAIR(3));
	waddstr(menubar,"(F3)");
	wattroff(menubar,COLOR_PAIR(3));
}

WINDOW **draw_first_menu(int start_col, WINDOW *messagebar)
{
	int i;
	WINDOW **items;
	 menu_number=1;
	items=(WINDOW **)malloc(8*sizeof(WINDOW *));

	items[0]=newwin(9,19,1,start_col);
	wbkgd(items[0],COLOR_PAIR(2));
	box(items[0],ACS_VLINE,ACS_HLINE);
	items[1]=subwin(items[0],1,17,2,start_col+1);
	items[2]=subwin(items[0],1,17,3,start_col+1);
	items[3]=subwin(items[0],1,17,4,start_col+1);
	items[4]=subwin(items[0],1,17,5,start_col+1);
	items[5]=subwin(items[0],1,17,6,start_col+1);
	items[6]=subwin(items[0],1,17,7,start_col+1);
	items[7]=subwin(items[0],1,17,8,start_col+1);
	for (i=1;i<8;i++)
		wprintw(items[i],Commands_Menu_String[i].c_str());
	wbkgd(items[1],COLOR_PAIR(1));
	wrefresh(items[0]);
	 werase(messagebar);
	 wprintw(messagebar,Command_Infos[0]);
	 wrefresh(messagebar);
	return items;
}

WINDOW **draw_second_menu(int start_col, WINDOW *messagebar)
{
	int i;
	WINDOW **items;
	 menu_number=2;
	items=(WINDOW **)malloc(8*sizeof(WINDOW *));

	items[0]=newwin(9,19,1,start_col);
	wbkgd(items[0],COLOR_PAIR(2));
	box(items[0],ACS_VLINE,ACS_HLINE);
	items[1]=subwin(items[0],1,17,2,start_col+1);
	items[2]=subwin(items[0],1,17,3,start_col+1);
	items[3]=subwin(items[0],1,17,4,start_col+1);
	items[4]=subwin(items[0],1,17,5,start_col+1);
	items[5]=subwin(items[0],1,17,6,start_col+1);
	items[6]=subwin(items[0],1,17,7,start_col+1);
	items[7]=subwin(items[0],1,17,8,start_col+1);
	for (i=1;i<8;i++)
```

```
            wprintw(items[i],Builtin_Menu_String[i].c_str());
        wbkgd(items[1],COLOR_PAIR(1));
        wrefresh(items[0]);
         werase(messagebar);
         wprintw(messagebar,Builtin_Infos[0]);
         wrefresh(messagebar);
        return items;
}

WINDOW **draw_third_menu(int start_col, WINDOW *messagebar)
{
        int i;
        WINDOW **items;
         menu_number=3;
        items=(WINDOW **)malloc(3*sizeof(WINDOW *));

        items[0]=newwin(4,19,1,start_col);
        wbkgd(items[0],COLOR_PAIR(2));
        box(items[0],ACS_VLINE,ACS_HLINE);
        items[1]=subwin(items[0],1,17,2,start_col+1);
        items[2]=subwin(items[0],1,17,3,start_col+1);
        for (i=1;i<3;i++)
            wprintw(items[i],About_Menu_String[i].c_str());
        wbkgd(items[1],COLOR_PAIR(1));
        wrefresh(items[0]);
         werase(messagebar);
         wprintw(messagebar,About_Infos[0]);
         wrefresh(messagebar);
        return items;
}

void delete_menu(WINDOW **items,int count)
{
        int i;
        for (i=0;i<count;i++)
         {
            delwin(items[i]);
         }
}

int scroll_first_menu(WINDOW **items,int count,int menu_start_col, WINDOW
*messagebar)
{
        int key;
        int selected=0;
        while (1) {
            key=getch();
            if (key==KEY_DOWN || key==KEY_UP) {
                wbkgd(items[selected+1],COLOR_PAIR(2));
                wnoutrefresh(items[selected+1]);
```

```
                if (key==KEY_DOWN) {
                        selected=(selected+1) % count;
                } else {
                        selected=(selected+count-1) % count;
                }
                wbkgd(items[selected+1],COLOR_PAIR(1));
                wnoutrefresh(items[selected+1]);
                doupdate();
                werase(messagebar);
                wprintw(messagebar,Command_Infos[selected]);
                wrefresh(messagebar);
        }else if (key==KEY_RIGHT) {
                delete_menu(items,count+1);
                touchwin(stdscr);
                refresh();
                items=draw_second_menu(20,messagebar);
                return scroll_second_menu(items,7,20,messagebar);
        }else if (key==KEY_LEFT) {
                delete_menu(items,count+1);
                touchwin(stdscr);
                refresh();
                items=draw_third_menu(40,messagebar);
                return scroll_third_menu(items,2,40,messagebar);
        }  else if (key==ESCAPE) {
                return -1;
        } else if (key==ENTER) {
                return selected;
        }
    }
}

int scroll_second_menu(WINDOW **items,int count,int menu_start_col, WINDOW
*messagebar)
{
    int key;
    int selected=0;
    while (1) {
        key=getch();
        if (key==KEY_DOWN || key==KEY_UP) {
                wbkgd(items[selected+1],COLOR_PAIR(2));
                wnoutrefresh(items[selected+1]);
                if (key==KEY_DOWN) {
                        selected=(selected+1) % count;
                } else {
                        selected=(selected+count-1) % count;
                }
                wbkgd(items[selected+1],COLOR_PAIR(1));
                wnoutrefresh(items[selected+1]);
                doupdate();
                werase(messagebar);
```

```
                wprintw(messagebar,Builtin_Infos[selected]);
                wrefresh(messagebar);
            }else if (key==KEY_RIGHT) {
                delete_menu(items,count+1);
                touchwin(stdscr);
                refresh();
                items=draw_third_menu(40,messagebar);
                return scroll_third_menu(items,2,40,messagebar);
            }else if (key==KEY_LEFT) {
                delete_menu(items,count+1);
                touchwin(stdscr);
                refresh();
                items=draw_first_menu(0,messagebar);
                return scroll_first_menu(items,7,0,messagebar);
            } else if (key==ESCAPE) {
                return -1;
            } else if (key==ENTER) {
                return selected;
            }
        }
}

int scroll_third_menu(WINDOW **items,int count,int menu_start_col, WINDOW
*messagebar)
{
        int key;
        int selected=0;
        while (1) {
            key=getch();
            if (key==KEY_DOWN || key==KEY_UP) {
                wbkgd(items[selected+1],COLOR_PAIR(2));
                wnoutrefresh(items[selected+1]);
                if (key==KEY_DOWN) {
                    selected=(selected+1) % count;
                } else {
                    selected=(selected+count-1) % count;
                }
                wbkgd(items[selected+1],COLOR_PAIR(1));
                wnoutrefresh(items[selected+1]);
                doupdate();
                werase(messagebar);
                wprintw(messagebar,About_Infos[selected]);
                wrefresh(messagebar);
            }else if (key==KEY_RIGHT) {
                delete_menu(items,count+1);
                touchwin(stdscr);
                refresh();
                items=draw_first_menu(0,messagebar);
                return scroll_first_menu(items,7,0,messagebar);
            }else if (key==KEY_LEFT) {
```

```
                delete_menu(items,count+1);
                touchwin(stdscr);
                refresh();
                items=draw_second_menu(20,messagebar);
                return scroll_second_menu(items,7,20,messagebar);
            }  else if (key==ESCAPE) {
                return -1;
            } else if (key==ENTER) {
                return selected;
            }
        }
}
```

/***/

Το αρχείο colorcodes.h

/***/

```
/****************************************
      Wish Unix Shell - Version 1.0
      Author: Giannakidis Apostolis
      Id: colorcodes.h
****************************************/

#define RESET           0
#define BRIGHT          1
#define DIM             2
#define UNDERLINE       3
#define BLINK           4
#define REVERSE         7
#define HIDDEN          8

#define BLACK           0
#define RED             1
#define GREEN           2
#define YELLOW          3
#define BLUE            4
#define MAGENTA         5
#define CYAN            6
#defineWHITE            7
```

Περιεχόμενα εικόνων – πινάκων

Εικόνες

Κεφάλαιο 1

Κεφάλαιο 2

Κεφάλαιο 3

Πίνακες

Κεφάλαιο 1

Κεφάλαιο 3

Βιβλιογραφικές αναφορές

Κεφάλαιο 1

1. **Silberschatz, Galvin, Gagne**: Operating System Concepts, 6th edition, John Wiley and Sons, 2002, σελ: 1-7
2. **Παπακωνσταντίνου Γ., Τσανάκας Π., Κοζύρης Ν., Μανουσοπούλου Α., Ματζάκος Π**: Τεχνολογία Υπολογιστικών Συστημάτων και Λειτουργικά Συστήματα, 1999, σελ: 175 - 180
3. **Andrew S. Tanenbaum, Albert S.Woodhull:** Operating Systems Design and Implementation 2nd edition, σελ: 3
4. **Garry Nutt:** Operating Systems a Modern Perspective, 2nd edition Lab Update, Addison-Wesley 2001, σελ: 80 - 83
5. **Rochkind Marc:** Advanced Unix Programming 2nd edition, Addison-Wesley, 2004, σελ : 19 - 23
6. **Silberschatz, Galvin, Gagne**: Operating System Concepts, 6th edition, John Wiley and Sons, 2002, σελ: 63-74
7. **Andrew S. Tanenbaum**: Structured Computer Organization 4th edition, Prentice Hall, 1995, σελ: 548-549
8. **Rochkind Marc:** Advanced Unix Programming 2nd edition, Addison-Wesley, 2004, σελ : 361
9. **Neil Matthew, Richard Stones:** Beginning Linux Programming, 3rd edition, Wrox, 2004, σελ: 505-514
10. **Glass, G.** : UNIX for Programmers and Users : A Complete Guide. Englewood Cliffs, NJ: Prentice-Hall, 1993.
11. **Andrew S. Tanenbaum, Albert S.Woodhull:** Operating Systems Design and Implementation 2nd edition, σελ: 59
12. **Andrew S. Tanenbaum**: Structured Computer Organization 4th edition, Prentice Hall, 1995, σελ: 553-554
13. **Neil Matthew, Richard Stones:** Beginning Linux Programming, 3rd edition, Wrox, 2004, σελ: 557-559
14. **Andrew S. Tanenbaum, Albert S.Woodhull:** Operating Systems Design and Implementation 2nd edition, σελ: 82-83
15. **Silberschatz, Galvin, Gagne**: Operating System Concepts, 6th edition, John Wiley and Sons, 2002, σελ:151-153
16. **Garry Nutt:** Operating Systems a Modern Perspective, 2nd edition Lab Update, Addison-Wesley 2001, σελ: 195-196
17. **Silberschatz, Galvin, Gagne**: Operating System Concepts, 6th edition, John Wiley and Sons, 2002, σελ:154-155
18. **Silberschatz, Galvin, Gagne**: Operating System Concepts, 6th edition, John Wiley and Sons, 2002, σελ: 136-139
19. **Rochkind Marc:** Advanced Unix Programming 2nd edition, Addison-Wesley, 2004, σελ : 621-623

Κεφάλαιο 2

1. **David A.Watt & Deryck F. Brown:** Programming Language Processors in Java, Compilers and Interpreters, 1st edition, Pearson Education, 2000, σελ: 4-6
2. **Ronald Mak:** Writing Compilers and Interpreters, 2nd edition, Wiley, 1996, σελ 9
3. **Robin Hunter:** The essence of compilers, 1st edition, Pearson Education, 1999, σελ: 5 – 11

4. **David A.Watt & Deryck F. Brown:** Programming Language Processors in Java, Compilers and Interpreters, 1st edition, Pearson Education, 2000, σελ: 83-84
5. **A.Aho & R.Sethi & J.Ullman:** Compilers, principles, techniques and tools, 1st edition, Addison-Wesley 1986, επανέκδοση με διορθώσεις το 1988, σελ: 40-41
6. **David A.Watt & Deryck F. Brown:** Programming Language Processors in Java, Compilers and Interpreters, 1st edition, Pearson Education, 2000, σελ: 250-251
7. **A.Aho & R.Sethi & J.Ullman:** Compilers, principles, techniques and tools, 1st edition, Addison-Wesley 1986, επανέκδοση με διορθώσεις το 1988, σελ: 585-587
8. **David A.Watt & Deryck F. Brown:** Programming Language Processors in Java, Compilers and Interpreters, 1st edition, Pearson Education, 2000, σελ: 349-352
9. **Robin Hunter:** The essence of compilers, 1st edition, Pearson Education, 1999, σελ: 21 – 31
10. **David A.Watt & Deryck F. Brown:** Programming Language Processors in Java, Compilers and Interpreters, 1st edition, Pearson Education, 2000, σελ: 93-124
11. **A.Aho & R.Sethi & J.Ullman:** Compilers, principles, techniques and tools, 1st edition, Addison-Wesley 1986, επανέκδοση με διορθώσεις το 1988, σελ: 107-111

Βιβλιογραφία

1. **Silberschatz, Galvin, Gagne**: Operating System Concepts, 6th edition, John Wiley and Sons, 2002
2. **Παπακωνσταντίνου Γ., Τσανάκας Π., Κοζύρης Ν., Μανουσοπούλου Α., Ματζάκος Π**: Τεχνολογία Υπολογιστικών Συστημάτων και Λειτουργικά Συστήματα, 1999
3. **Andrew S. Tanenbaum, Albert S.Woodhull:** Operating Systems Design and Implementation 2nd edition
4. **Garry Nutt:** Operating Systems a Modern Perspective, 2nd edition Lab Update, Addison-Wesley 2001
5. **Rochkind Marc:** Advanced Unix Programming 2nd edition, Addison-Wesley, 2004
6. **Andrew S. Tanenbaum**: Structured Computer Organization 4th edition, Prentice Hall, 1995
7. **Neil Matthew, Richard Stones:** Beginning Linux Programming, 3rd edition, Wrox, 2004
8. **Glass, G.** : UNIX for Programmers and Users : A Complete Guide. Englewood Cliffs, NJ: Prentice-Hall, 1993
9. **David A.Watt & Deryck F. Brown:** Programming Language Processors in Java, Compilers and Interpreters, 1st edition, Pearson Education, 2000
10. **Robin Hunter:** The essence of compilers, 1st edition, Pearson Education, 1999
11. **A.Aho & R.Sethi & J.Ullman:** Compilers, principles, techniques and tools, 1st edition, Addison-Wesley 1986, επανέκδοση με διορθώσεις το 1988