

Αρχές Γλωσσών Προγραμματισμού & Μεταφραστές  
Εργαστηριακή Άσκηση

Σπύρος Καφτάνης (5542) | Αποστόλης Κέμος (5544)  
Χριστίνα Χριστοδούλου (5691) | Ελευθερία Στεφάνου (5940)

## Ενότητα 1 [ Εισαγωγή & BNF ]

Η γλώσσα η οποία αναπτύσσεται σε αυτή την εργασία αποτελεί ένα υποσύνολο του πρωτοκόλλου http(rfc2616) και πιο συγκεκριμένα ένα μήνυμα αίτησης, με τους περιορισμούς της εκφώνησης.

Για τη δημιουργία της BNF καθώς και γενικά για την υλοποίηση του λεξικού και του συντακτικού αναλυτή πραγματοποιήθηκαν οι παρακάτω συμβάσεις (βασισμένες στο πρωτόκολλο) για τη μορφή του αρχείου:

- Η πρώτη και υποχρεωτική γραμμή του αιτήματος θα μπορεί να έχει τη μορφή: Request-Type : Method SP URL SP HTTP1.1 CRLF(όπου SP=κενό και CRLD=νέα γραμμή)
- Τα general headers θα εμφανίζονται με το όνομά τους και έπειτα με τη τιμή τους, έχοντας μία άνω και κάτω τελεία ανάμεσά τους (πχ. Date : 22/09/2014 23:59)
- Τα request headers θα εμφανίζονται με όνομά τους ακολουθούμενο με μία αποδεκτή τιμή με το σύμβολο της άνω και κάτω τελείας ανάμεσά τους. (πχ. User-Agent : CERN-LineMode/2.15),
- Τα entity headers θα εμφανίζονται με το όνομά τους και έπειτα με τη τιμή τους, έχοντας επίσης μία άνω και κάτω τελεία ανάμεσά τους. (πχ. Content-Length : 10)
- Τέλος το message body ορίζεται με το αλφαριθμητικό "Message-Body" ακολουθούμενο από το κείμενο. (πχ. Message-Body: autoEinaio).

Οι επιτρεπόμενες τιμές για κάθε έκφραση ορίζονται στο flex αρχείο (myscanner.l) και ακολουθούν τους κανόνες του πρωτοκόλλου (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html>) .

Ο βασικός κανόνας ο οποίος ακολουθείται σε ένα τέτοιου είδους πρωτόκολλο είναι ότι η πρώτη σειρά (Request Line) και η τελευταία (Message Body) είναι υποχρεωτικές, ενώ ανάμεσά τους μπορούν να υπάρχουν όσες τιμές από general, request και entity headers επιθυμούμε, όπως φαίνεται και στον επίσημο ορισμό.

```
Request      = Request-Line           ; Section 5.1
               *(( general-header      ; Section 4.5
                 | request-header      ; Section 5.3
                 | entity-header ) CRLF ; Section 7.1
               CRLF
               [ message-body ]       ; Section 4.3
```

Ακολουθεί η περιγραφή της γραμματικής της γλώσσας σε BNF.

**expr**::= <Request-Line><expr2>

**Request-Line**::=<VALID\_TYPE\_TITLE>':<VALID\_TYPE\_VALUE>"\n"

**expr2**::= <B>expr2|expr3

**B**=<general\_header>|<request\_header>|<entity\_expires>|<entity\_length>|ε

**General\_Header**::=<VALID\_CONNECTION\_TITLE>":<VALID\_CONNECTION\_VALUE>"\n"  
| <VALID\_DATE\_TITLE> ':<VALID\_DATE\_VALUE>"\n"  
| <VALID\_TRANFER\_TITLE> ':<VALID\_TRANSFER\_VALUE>"\n"

**Request\_Header**::=

<VALID\_ACCEPT\_CHARSET\_TITLE>":<VALID\_ACCEPT\_CHARSET\_VALUE>"\n"  
| <VALID\_REFERERD\_TITLE>":<VALID\_REFERERD\_VALUE>"\n"  
| <VALID\_USER\_AGENT\_TITLE>":<VALID\_USER\_AGENT\_VALUE>"\n"

**Entity\_Header**::=<VALID\_CONTENT\_VALUE>"\n"

**Entity\_Expires**::=<VALID\_EXPIRES\_VALUE>"\n"

**Expr3**::=<VALID\_MESSAGE\_BODY\_TITLE>':<VALID\_MESSAGE\_BODY\_VALUE>"\n"

**VALID\_TYPE\_TITLE**::="Request-Type"

**<VALID\_TYPE\_VALUE>**::=("GET"|"POST"|"HEAD")\ (<Cap> | <Low>){ (<Cap> | <Low> | <digit> | "/" | ".") } ".html" \ "HTTP/1.1"

**<VALID\_TYPE\_VALUE>**::=("GET"|"POST"|"HEAD")\ (<Cap> | <Low>){ (<Cap> | <Low> | <digit> | "/" | ".") } ".html" \ "HTTP/1.1"

**VALID\_CONNECTION\_TITLE**::="Connection"

**VALID\_CONNECTION\_VALUE**::=1#("<Cap> | <Low>){ <Cap> | <Low> | <digit> }

**VALID\_DATE\_TITLE**::="Date"

**VALID\_DATE\_VALUE**::=<digit\_03><digit>"/"<digit\_19><digit>"/"<digit><digit><digit><digit> [0-2]<digit\_03>":<digit\_05><digit>

**VALID\_TRANFER\_TITLE**::="Tranfer-Encoding"

**VALID\_TRANSFER\_VALUE**::="chunked"|"gzip"|"deflate"

**VALID\_REQUEST\_HEADERS\_TITLE**::="Request-Header"

**VALID\_REQUEST\_HEADER\_VALUE**::="Accept-Charset"|"Refered"|"User-Agent"

**VALID\_CONTENT\_VALUE**::=Content-Length : {<digit> }

```

VALID_EXPIRES_VALUE::=Expires : HTTP-
<digit><digit>"/"<digit><digit>"/"<digit><digit><digit><digit><digit_02><digit_03>:<di
git_05><digit>

VALID_MESSAGE_BODY_TITLE::="Message-Body"
VALID_MESSAGE_BODY_VALUE::=(<Cap> | <Low>){ <Cap> | <Low> | <digit> | " " }
VALID_ACCEPT_CHARSET_TITLE::="Accept-Charset"
VALID_ACCEPT_CHARSET_VALUE::=( [ (<Cap> | <Low>) {<Cap> | <Low> | <digit> | "-"} }
] | "" );q="<digit>

VALID_REFERERED_TITLE::="Refered"
VALID_REFERERED_VALUE::=(<Cap> | <Low>){ (<Cap> | <Low> | <digit> | "/" | "." ) } ".html"

VALID_USER_AGENT_TITLE::="User-Agent"
VALID_USER_AGENT_VALUE::=(<Cap> | <Low>){<Cap> | <Low> | <digit>
| "-") } "/"<digit>".<digit><digit>

<digit>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9
<Cap>::= A | B | C | D | E | F | H | I | G | K | L | M | N | O | P | Q | R | S | T | U | V | W |
Y | Z
<Low>::= a | b | c | d | e | f | h | i | g | k | l | m | n | o | p | q | r | s | t | u | v | w | y |
z
<digit_03>::= 0 | 1 | 2 | 3
<digit_02>::= 0 | 1 | 2
<digit_05>::= 0 | 1 | 2 | 3 | 4 | 5
<digit_19>::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9

```

Η **expr** είναι η αρχική έκφραση, η οποία καθορίζει ότι θα υπάρχει υποχρεωτικά ένα **Request-Line** και μία **expr2**. Το **Request-Line** ορίζεται εύκολα όπως είπαμε νωρίτερα, χρησιμοποιώντας και τα τερματικά σύμβολα **VALID\_TYPE\_TITLE** και **VALID\_TYPE\_VALUE**. Έτσι καθορίζεται ότι θα υπάρχει υποχρεωτικά κάποια **Request-Line** στο αρχείο εισόδου.

Το **expr2** τώρα, αποτελείται από την έκφραση **B**, η οποία περιλαμβάνει σε διάζευξη όλα τα δυνατά headers, μαζί με το **expr2**. Η σημασία αυτής της αναδρομικής κλήσης είναι ότι αν βάλουμε κάποιο header στη συνέχεια θα μπορούμε να βάλουμε και άλλα. Η έκφραση ολοκληρώνεται κάνοντας διάζευξη στο **expr3**, το οποίο είναι το message-body.

Στη συνέχεια, αφού έχει οριστεί ο βασικός κανοντας του πρωτοκόλλου ορίζονται τα headers με τη μορφή που αναλύσαμε αρχικά, και τέλος ορίζονται τα τερματικά σύμβολα με τις όλες τις δυνατές τερματικές τιμές που μπορούν να πάρουν.

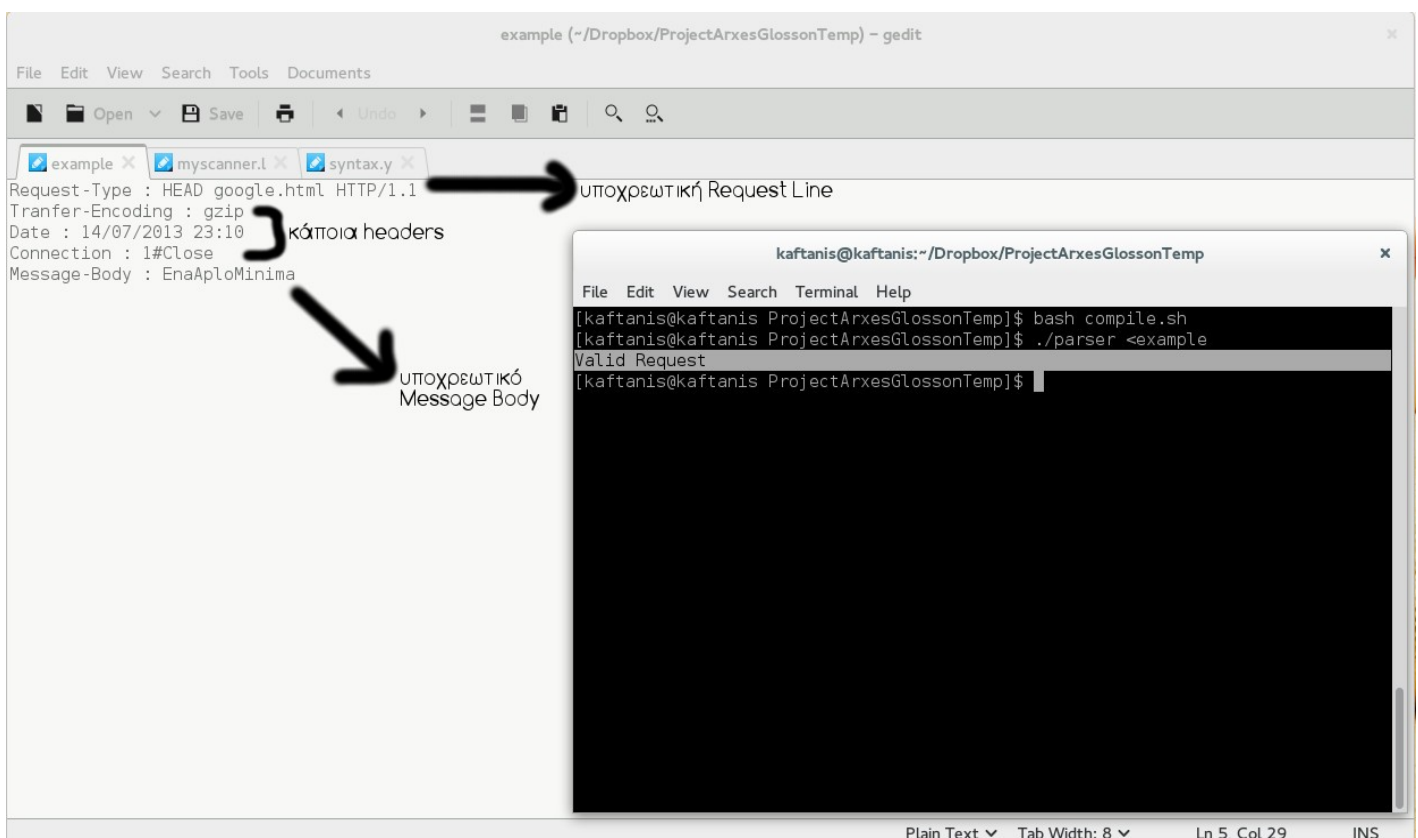
## Ενότητα 2 [ Δεύτερο Ερώτημα ]

Στο **δεύτερο ερώτημα**, δημιουργήθηκε ο λεξικός και ο συντακτικός αναλυτής έτσι ώστε να ελέγχεται αν το αρχείο που δίνεται σαν είσοδος είναι συντακτικά ορθό. Στο αρχείο flex (**myscanner.l**) ορίστηκαν όλοι οι κανόνες για τα tokens που μπορούν να υπάρχουν στο αρχείο. Στον αρχείο bison (**syndax.y**) ορίστηκαν οι συντακτικοί κανόνες, με την ίδια περίπου μορφή που ορίστηκαν στην BNF.

Ο βασικός κανόνες είναι ότι πρέπει να έχουμε υποχρεωτικά Request Line και Message Body και όσα headers θέλουμε χωρίς κανένα περιορισμό. Φυσικά όλα αυτά πρέπει να ακολουθούν τους λεξικούς κανόνες που ορίστηκαν στον λεξικό αναλυτή (flex).

Αφού κάνουμε τρέξουμε το flex και το bison για τον λεξικό και τον συντακτικό αναλυτή αντίστοιχα και κάνουμε compile τα αρχεία σε c τα οποία παράγουν (δείτε πως στην ενότητα 5) εκτελούμε το πρόγραμμα με όρισμα το αρχείο example.

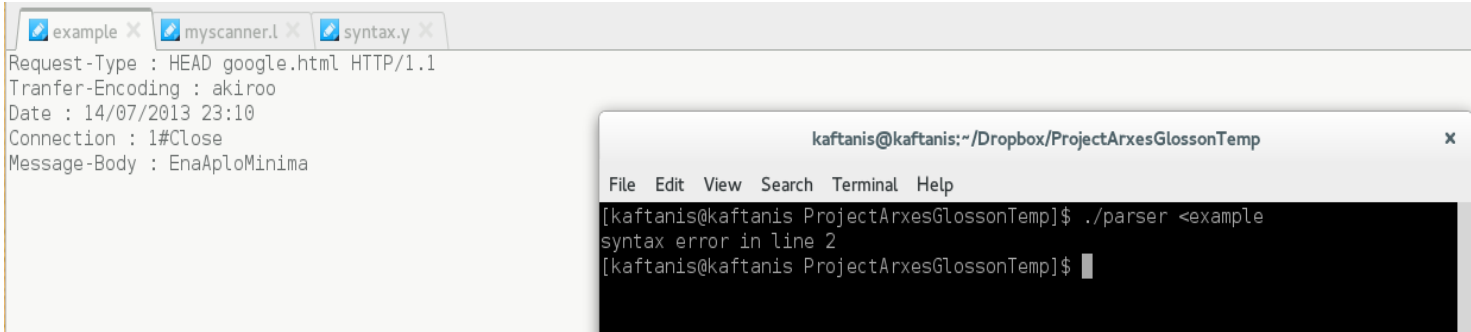
Ας δούμε αρχικά ένα ορθό παράδειγμα:



Όπως είναι φανερό, το παράδειγμα αυτό είναι ορθό γι αυτό και εμφανίζεται το

μήνυμα "Valid Request". Τα headers που εμφανίζονται εδώ είναι και τα τρία general headers, αλλά θα μπορούσαν να είναι οποιαδήποτε από τα διαθέσιμα.

Ας δούμε και ένα λανθασμένο:



Εδώ στο Transfer-Encoding έχουμε βάλει μία λανθασμένη τιμή αντί για τις τρεις τις οποίες θεωρεί έγκυρες. Έτσι το πρόγραμμα μας εμφανίζει μήνυμα λάθους (μέσω της yyerror), μαζί με την γραμμή στην οποία εμφανίστηκε αυτό.

Η εύρεση της γραμμής του λάθους γίνεται με τη βοήθεια της extern μεταβλητής yylineno. Με αυτό το τρόπο στη μεταβλητή αυτή αποθηκεύεται ο αριθμός της σειράς που σταμάτησε το πρόγραμμα λόγω σφάλματος. Χρησιμοποιώντας της yyerror εμφανίζουμε και το νούμερο της σειράς:

```
void yyerror(char *s) {  
    fprintf(stderr, "%s in line %d\n", s, yylineno);  
}
```

Για να λειτουργήσει σωστά αυτή η μέθοδος πρέπει να προσθέσουμε επίσης το %option yylineno στο flex αρχείο.

Μπορείτε να δείτε τον κώδικα του ερωτήματος αυτού στον φάκελο "Erotima2" μαζί με κάποια επιπλέον παραδείγματα.

---

## Ενότητα 3 [ Τρίτο Ερώτημα ]

Στο ερώτημα αυτό πρέπει όταν στη Request Line υπάρχει POST (αντί για HEAD ή GET που είναι οι άλλες δύο επιλογές) να υπάρχει υποχρεωτικά message-body (κάτι που είναι ήδη υποχρεωτικό λόγω του ορισμού του πρωτοκόλλου), αλλά και το Content-Length.

Αρχικά θα δούμε πως έχει οριστεί το Content-Length στον flex (ήταν ορισμένο έτσι και στο προηγούμενο ερώτημα). Το Content-Length ανήκει στα entity header και πρέπει να έχει έναν αριθμό ο οποίος δείχνει το πλήθος των χαρακτήρων του message-body.

Κατά τη συνεργασία flex-bison υπάρχει μια global μεταβλητή για κάθε επιστρεφόμενο token με το όνομα `yylval`. Όταν θέσουμε κάτι σε αυτή τη μεταβλητή από το flex μπορούμε να το χρησιμοποιήσουμε αργότερα και στο bison. Έτσι λοιπόν αυτό που πρέπει να γίνει είναι να επιστρέφουμε το μέγεθος του μηνύματος από το message body και την τιμή του Content-Length από το Content-Length.

Το μέγεθος του μηνύματος το θέτουμε στη `yylval` χρησιμοποιώντας τη συνάρτηση `strlen(char*)` της `<string.h>`, η οποία έχει γίνει `include`.

```
[A-Za-z][A-Za-z0-9|\s]* {yylval = strlen(yytext); return VALID_MESSAGE_BODY_VALUE; }
```

Για την τιμή του Content-Length θα χρειαστεί λίγο περισσότερη προσπάθεια. Το Content-Length ανήκει όπως αναφέραμε στα entity headers, οπότε σύμφωνα με τις συμβάσεις της πρώτης ενότητας ορίζεται στο αρχείο με τη παρακάτω μορφή:

```
Content-Length : 10
```

Οι πρώτοι 17 χαρακτήρες δεν αποτελούν χρήσιμη πληροφορία για αυτό που χρειαζόμαστε. Το μέγεθος του αλφαριθμητικού που χρειαζόμαστε θα είναι λοιπόν όσο είναι το μέγεθος όλου μείον 17.

```
int len = strlen(yytext)-17;
```

Στη συνέχεια δημιουργούμε ένα πίνακα χαρακτήρων `subarray` με μέγεθος `len+1` (έτσι ώστε να μπορεί να μπει και το τερματικό σύμβολο `\0`). Χρησιμοποιώντας την `memcpy` εισάγουμε στο `subarray` τους χαρακτήρες που βρίσκονται από τη θέση 17 συν άλλες `len`. Με άλλα λόγια εισάγουμε τη τιμή που έχει το Content-Length στο αρχείο. Εισάγουμε το τερματικό σύμβολο στον `subarray` και θέτουμε στη `yylval` του συγκεκριμένου token το αλφαριθμητικό που αυτό σε μορφή `int` με τη βοήθεια της `atoi`.

```
"Content-Length" \ : [0-9]* {   int len = strlen(yytext)-17;
                                char subarray[len+1];
                                memcpy( subarray, &yytext[17], len );
                                subarray[len+1]='\0';
                                yyval = atoi(subarray); return VALID_CONTENT_VALUE; }
```

Στο bison τώρα ορίζουμε τις ακέραιες μεταβλητές `bodyLength` και `contentLength`, στις οποίες εκχωρούμε το `yylval` κάθε φορά που συναντάται κάτι από τα δύο. Στο Message-Body τώρα (αφού εκχωρηθεί και εκεί η `yylval` στο `bodyLength`), γίνεται ο έλεγχος για το αν οι τιμές των δύο αυτών μεταβλητών είναι ίδιες και αν μιλάμε για την περίπτωση όπου έχουμε POST. Αυτό το ελέγχουμε με τη μεταβλητή `isContentLength` η οποία παίρνει τη τιμή 1 όταν υπάρχει Content-Length, έτσι ώστε αν δεν είναι να μην απαιτείται η ύπαρξη του Content Length. Αν ισχύουν όλα αυτά εμφανίζεται το μήνυμα "Valid Request", διαφορετικά εμφανίζεται μήνυμα λάθους.

Η διαφοροποίηση σε αυτό το ερώτημα είναι ότι πρέπει να υπάρχει υποχρεωτικά το Content-Length όταν η μέθοδος είναι POST. Για το λόγο αυτό αρχικά στο flex χωρίζουμε τα αποδεκτά token του Request-Line έτσι ώστε όταν υπάρχει HEAD ή GET να επιστρέφεται το token VALID\_TYPE\_VALUE ενώ όταν υπάρχει GET το VALID\_POST\_VALUE.

Έτσι, στο Request-Line του bison υπάρχουν τώρα οι δύο παρακάτω επιλογές:

```
Request_Line:  VALID_TYPE_TITLE ':' VALID_TYPE_VALUE NEWLINE expr2
               | VALID_TYPE_TITLE ':' VALID_POST_VALUE NEWLINE length
               ;
```

Όταν η μέθοδος είναι GET ή HEAD τότε πρέπει να υπάρχει **expr2** το οποίο είναι το ίδιο με το προηγούμενο ερώτημα (βλέπε και BNF).

```
expr2 : general_header expr2
       | request_header expr2
       | entity_length expr2
       | entity_expires expr2
       | expr3
```

Όταν όμως η μέθοδος είναι POST υπάρχει μία διαφορετική έκφραση, η length:

```
length: general_header length
        | request_header length
        | entity_expires length
        | entity_length expr2
        | expr4
        ;
```

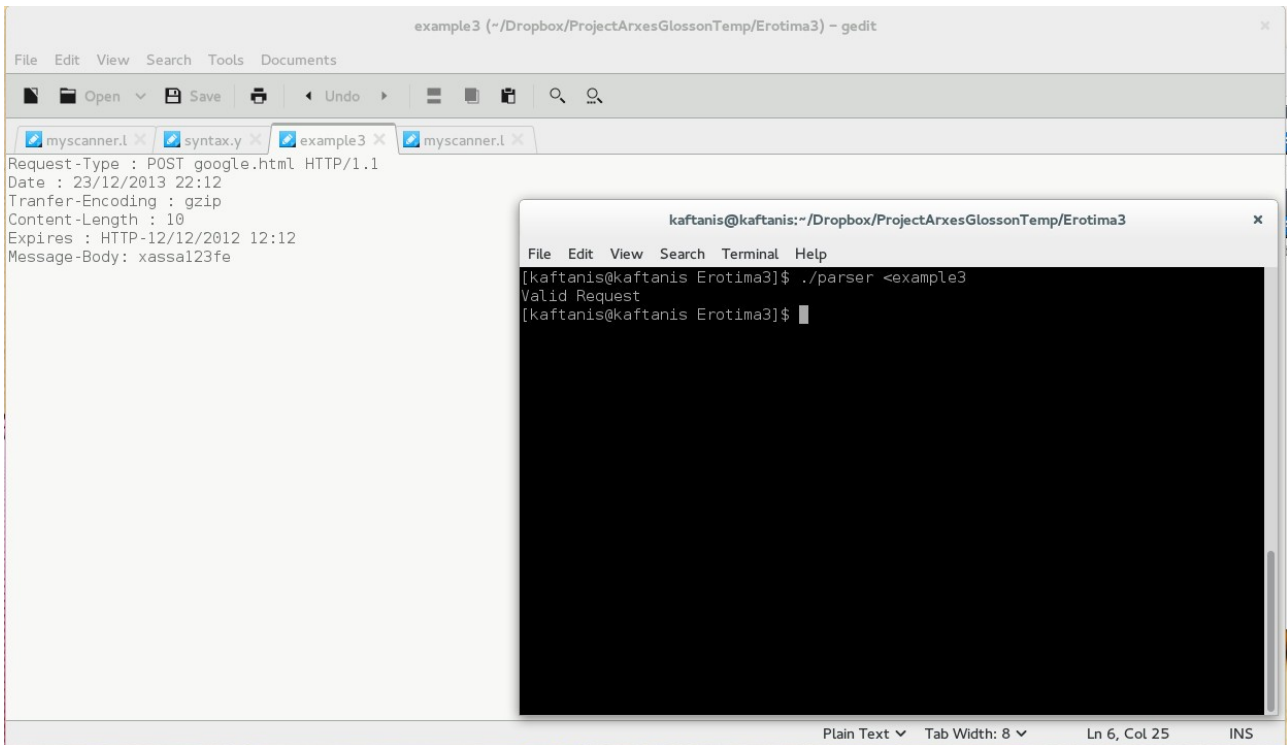
Η μέθοδος αυτή λειτουργεί σαν τη προηγούμενη με τη διαφορά ότι πέρα από τα headers υπάρχει το **expr4** αντί για το **expr3**. Το **expr4** ελέγχει το message-body, αλλά εμφανίζει οπωσδήποτε μήνυμα λάθους.

```
expr4:  VALID_MESSAGE_BODY_TITLE ':' VALID_MESSAGE_BODY_VALUE
        NEWLINE
        { printf("Content Length was not found\n");
          ;}
```

Αυτό συμβαίνει επειδή όπως μπορείτε να δείτε, αν βρεθεί κάτι άλλο πέρα από το entity\_length το πρόγραμμα παραμένει στην έκφραση length (αναδρομικά). Αν λοιπόν η έκφραση length βρει message-body σημαίνει ότι δε βρέθηκε Content-Length, και το entity\_length οδηγεί πίσω στο κανονικό **expr2**.

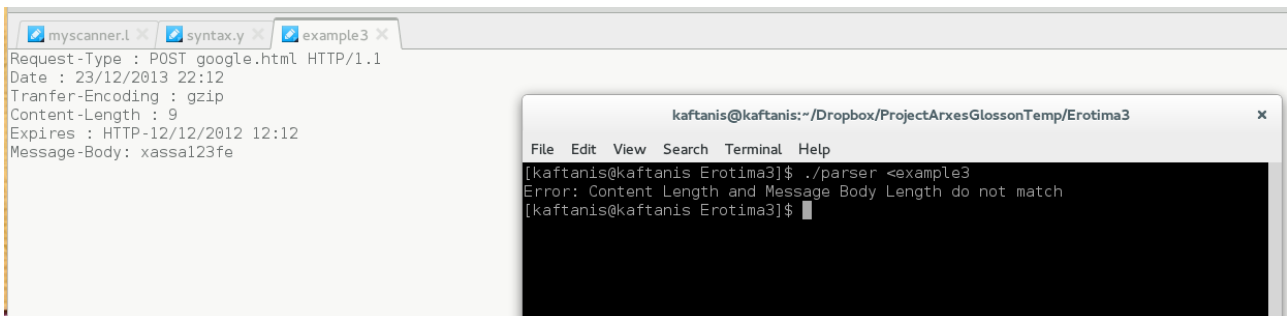
Ακολουθεί ένα παράδειγμα σωστής εκτέλεσης.



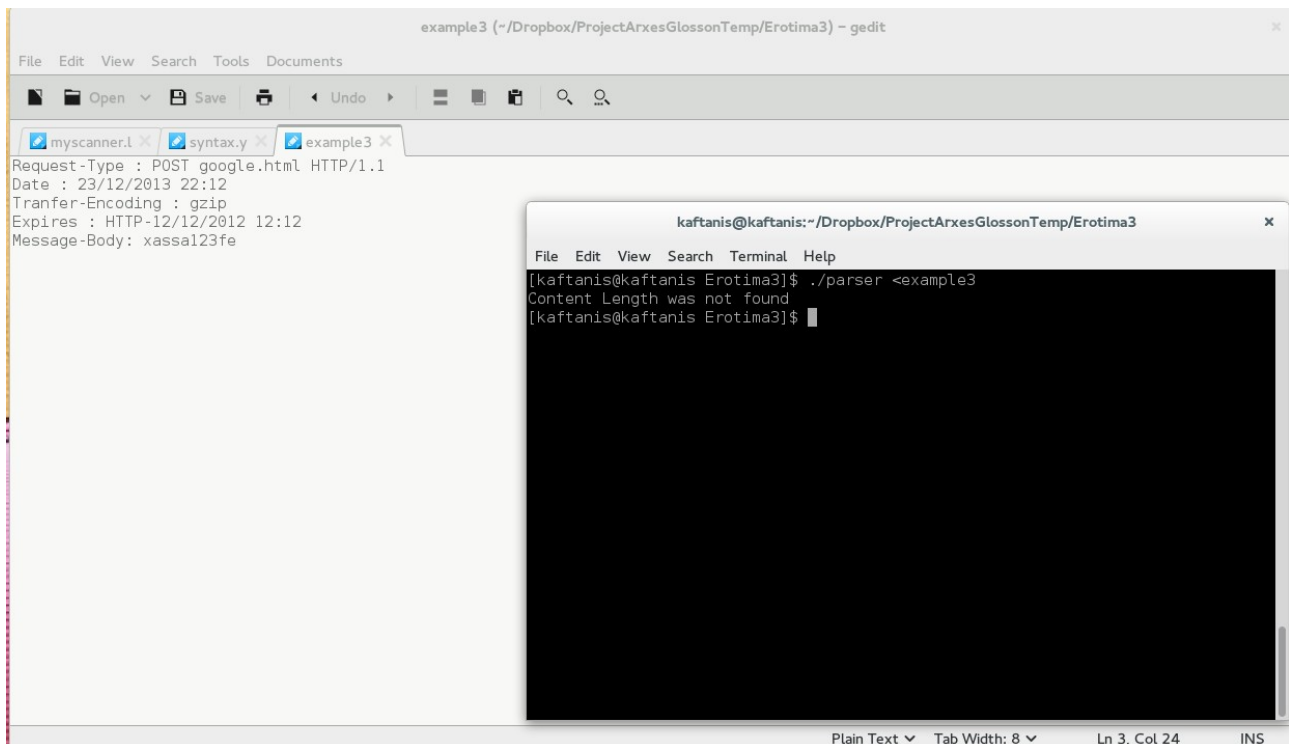


Όπως φαίνεται στο παραπάνω screenshot η μέθοδος είναι POST, το Content-Length υπάρχει και έχει τη σωστή τιμή.

Αν το Content-Length δεν έχει τη σωστή τιμή το πρόγραμμα συμπεριφέρεται με τον αναμενόμενο τρόπο:



Όπως επίσης και αν το Content-Length απουσιάζει όταν έχουμε μέθοδο POST:



Μπορείτε να δείτε τον κώδικα του ερωτήματος αυτού στον φάκελο "Erotima3" μαζί με κάποια επιπλέον παραδείγματα.

---

## Ενότητα 4 [ Τέταρτο Ερώτημα ]

Στο ερώτημα αυτό το πρόγραμμα σε περίπτωση σφάλματος δε πρέπει να σταματάει, αλλά πρέπει να συνεχίζει και να εντοπίζει όλα τα πιθανά σφάλματα μέχρι το τέλος του αρχείου.

Για να το πετύχουμε αυτό χρησιμοποιούμε μια έτοιμη έκφραση η οποία έχει το όνομα `error`. Στις αρχικές μας εκφράσεις λοιπόν, στο αρχείο του συντακτικού αναλυτή, προσθέτουμε την περίπτωση που βρεθεί σφάλμα και το χειριζόμαστε κατάλληλα. Αυτό βέβαια δημιουργεί επιπλέον προβλήματα που καλούμαστε να λύσουμε.

Αρχικά, στο αρχικό `expr` όταν το πρόγραμμα κοιτάει για να βρει την Request-Line, τώρα θα κοιτάει να βρει και σφάλμα. Αν βρει σφάλμα τότε θα οδηγηθεί στην `expr2`, η οποία είναι η έκφραση που καθορίζει τα headers.

```
expr : Request_Line
      | error NEWLINE expr2
      ;
```

Στο `expr2` τώρα, υπάρχουν οι κλασικές περιπτώσεις που δημιουργήθηκαν στα προηγούμενα ερωτήματα, αλλά οι υπάρχουν και άλλες δύο για τους χειρισμούς των σφαλμάτων. Το δεύτερο σφάλμα είναι για τη περίπτωση κατά την οποία δε βρεθεί ποτέ κάποιο valid message-body, οπότε και δε θα ενεργοποιηθεί ποτέ η `expr3`. Σε αυτή τη περίπτωση εμφανίζουμε εδώ με δικό μας μήνυμα το σφάλμα που προέκυψε με το message-body.

```
expr2 : general_header expr2
      | request_header expr2
      | entity_length expr2
      | entity_expires expr2
      | expr3 //to message-body
      | error NEWLINE exprError
      | error NEWLINE {printf("\nMissing or Wrong Message Body\n"); }
      ;
```

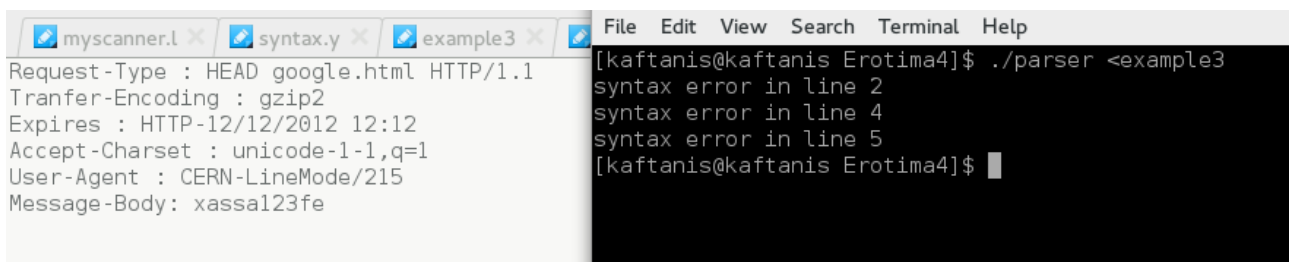
Αν τώρα μετά το `error` εντοπιστεί κάποιο άλλο header χειριζόμαστε τη κατάσταση μέσω της έκφρασης `exprError`.

Μιας και το μήνυμα "Valid Request" εμφανίζεται στην `expr3` δε μπορούμε να αφήσουμε το πρόγραμμα να πάει σε αυτή αν έχει βρει κάποιο λάθος. Αυτός είναι

και ο λόγος ύπαρξης της **exprError**. Εδώ, ανιχνεύονται αναδρομικά με τον ίδιο τρόπο τα headers και να βρεθεί το Message-Body πηγαίνουμε στην MessageError η οποία δεν εμφανίζει τίποτα.

```
exprError: general_header exprError
  | request_header exprError
  | entity_length exprError
  | entity_expires exprError
  | MessageError
;
```

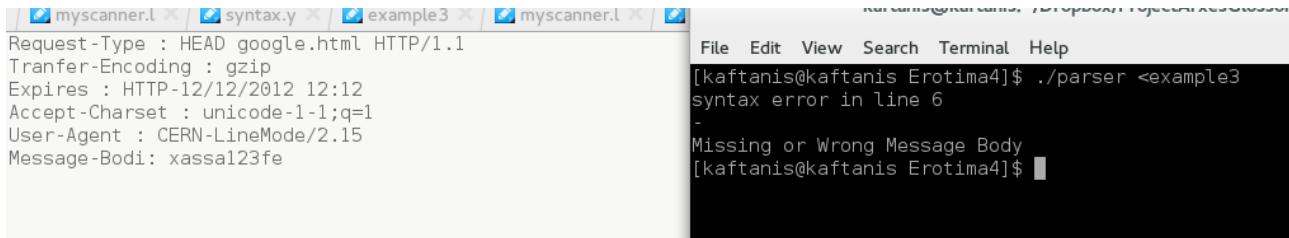
Παράδειγμα εύρεσης λαθών:



The screenshot shows a web browser window with the address bar displaying 'myscanner.l' and tabs for 'myscanner.l', 'syntax.y', and 'example3'. The browser's status bar shows the following HTTP headers: Request-Type : HEAD google.html HTTP/1.1, Transfer-Encoding : gzip2, Expires : HTTP-12/12/2012 12:12, Accept-Charset : unicode-1-1;q=1, User-Agent : CERN-LineMode/215, Message-Body: xassa123fe. To the right, a terminal window shows the command './parser <example3' being executed, resulting in three syntax errors: 'syntax error in line 2', 'syntax error in line 4', and 'syntax error in line 5'.

Το λάθος στη δεύτερη γραμμή είναι ότι το gzip2 δεν είναι αποδεκτό token, στη τέταρτη ότι στο AcceptCharser πριν το q υπάρχει ";" και όχι "," ενώ στη πέμπτη ότι μετά τον πρώτο αριθμό πρέπει να υπάρχει ".", έτσι όπως έχουν οριστεί στο flex.

Αν τώρα υπάρχει λάθος και στο message-body εμφανίζεται και το μήνυμα που δημιουργήσαμε:



The screenshot shows the same web browser window as before, but the status bar now displays 'Message-Bodi: xassa123fe' (note the typo 'Bodi' instead of 'Body'). The terminal window on the right shows the command './parser <example3' being executed, resulting in a 'syntax error in line 6' and a 'Missing or Wrong Message Body' error.

Μπορείτε να δείτε τον κώδικα του ερωτήματος αυτού στον φάκελο "Erotima4" μαζί με κάποια επιπλέον παραδείγματα.

## Ενότητα 5 [ Building Staff ]

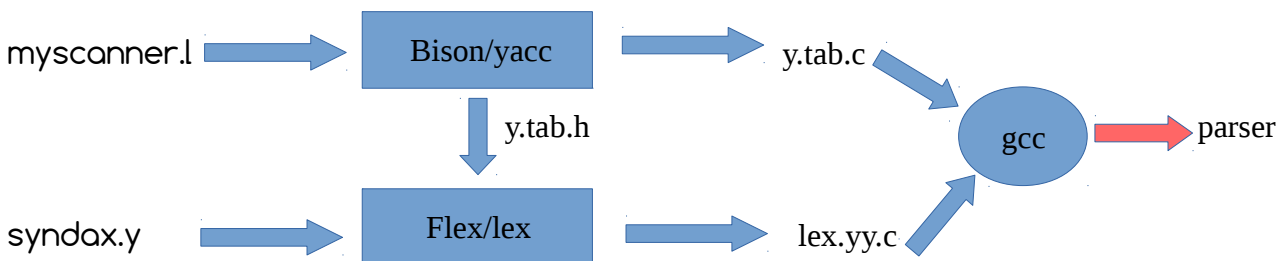
Προκειμένου να δημιουργήσουμε το τελικό εκτελέσιμο αρχείο έχουν πρώτα κληθεί τα προγράμματα flex, bison και gcc.

Όταν δημιουργήσουμε τον συντακτικό αναλυτή (αρχείο με κατάληξη .y), τρέχουμε το bison με όρισμα το αρχείο αυτό και έχουμε σαν έξοδο ένα C κώδικα με το όνομα y.tab.c και το header file του με το όνομα y.tab.h. Στη πραγματικότητα το .y αρχείο είναι μια γραμματική χωρίς συμφραζόμενα και το παραγόμενο αρχείο από το bison είναι το συντακτικός αναλυτής σε C. Υπάρχουν και άλλα τέτοια εργαλεία τα οποία παράγουν συντακτικούς αναλυτές σε άλλες γλώσσες.

Όταν δημιουργήσουμε τον λεξικό αναλυτή (αρχείο με κατάληξη .l) τρέχουμε το flex το οποίο παράγει τα tokens ακολουθώντας τους κανόνες που έχουμε ορίσει σε ένα πρόγραμμα C με το όνομα lex.yy.c. Στο αρχείο του flex πρέπει να έχουμε κάνει include το y.tab.h.

Τέλος χρησιμοποιώντας τον GNU Compiler Collection (GCC) μεταγλωττίζουμε μαζί το παραγόμενο πρόγραμμα σε C του bison (y.tab.c) και αυτό του flex (lex.yy.c) και έχουμε το εκτελέσιμο πρόγραμμα.

Η διαδικασία αυτή φαίνεται σχηματικά παρακάτω.



Για να κάνουμε όλα αυτά τα βήματα σε ένα θα χρειαστούμε ένα bash σκριπτάκι (για διανομές Gnu/Linux), το οποίο τρέχει σωστά το flex, το bison και το gcc. Το σκριπτάκι αυτό βρίσκεται σε κάθε ερώτημα με το όνομα "compile.sh" και για να το εκτελέσετε αρκεί να δώσετε

```
bash compile.sh
```

ή

```
./compile.sh
```

εάν το έχετε κάνει εκτελέσιμο (chmod +x compile.sh).

Το σκριπτάκι αυτό περιέχει τις παρακάτω εντολές:

```
bison -y -d syntax.y
flex myscanner.l
```

```
gcc -c y.tab.c lex.yy.c  
gcc y.tab.o lex.yy.o -o parser
```