ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

Σχολή Μηχανικών

Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών

Εργαστήριο «Μεταγλωττιστές»

Μέρος Α-3 : Συμπλήρωση πρότυπου κώδικα FLEX

Ημερομηνία Αποστολής: 7/5/2024

Τμήμα Β2 - Ομάδα 2

ΚΟΝΤΟΥΛΗΣ ΔΗΜΗΤΡΙΟΣ 21390095

ΜΕΝΤΖΕΛΟΣ ΑΓΓΕΛΟΣ ΚΩΝΣΤΑΝΤΙΝΟΣ 21390132

ΒΑΡΣΟΥ ΕΥΦΡΟΣΥΝΗ 21390021

ΓΚΙΟΖΙ ΕΝΤΕΡΙΣΑ 21390041

ΑΛΕΞΟΠΟΥΛΟΣ ΛΕΩΝΙΔΑΣ 21390006

Περιεχόμενα

1. Εισαγωγή

Το παρών έγγραφο αναφέρεται στην ανάπτυξη και υλοποίηση του λεκτικού αναλυτή με το εργαλείο FLEX. Ο λεκτικός αναλυτής είναι ένα εργαλείο με το οποίο μπορούμε να εντοπίσουμε τις λεκτικές μονάδες της γλώσσας Uni-C. Οι λεκτικές μονάδες και οι ιδιότητες τους έχουν αναλυθεί ιδιαίτερα στο Μέρος Α-2 (Part A-2) της εργασίας, στο οποίο έγινε εκπόνηση των κανονικών εκφράσεων καθώς και των πεπερασμένων αυτόματων (σχέδια & κώδικας FSM) των λεκτικών μονάδων. Στο κεφάλαιο της τεκμηρίωσης, αναλύεται η λειτουργία του λεκτικού αναλυτή αλλά και τυχόν προβλήματα που προέκυψαν κατά την λεκτική ανάλυση, τα οποία μας έδωσαν λανθασμένα αποτελέσματα και χρειάστηκαν περαιτέρω αντιμετώπιση. Επιπρόσθετα, θα γίνει ανάλυση του αρχείου με τις λεκτικές μονάδες το οποίο δόθηκε ως είσοδος στον λεκτικό αναλυτή, καθώς και το αρχείο εξόδου με τα αποτελέσματα που παράχθηκε από αυτόν.

2. Τεκμηρίωση

Παρακάτω ακολουθεί η ανάλυση τις λειτουργίας του λεκτικού αναλυτή. Για την περιγραφή της ορθής λειτουργίας του λεκτικού αναλυτή, θα γίνει αρχικά εξήγηση του αρχείου εισόδου το οποίο δόθηκε ως είσοδο σε αυτόν και έπειτα θα γίνει περιγραφή της εξόδου καθώς και σχολιασμός των αποτελεσμάτων. Το πρόγραμμα που αναπτύχθηκε ως είσοδος για τον λεκτικό αναλυτή είναι C – like πρόγραμμα, δηλαδή οι εντολές αντιπροσωπεύουν αυτές τις γλώσσας C. Είναι σημαντικό να αναφερθεί πως για την υλοποίηση του λεκτικού αναλυτή είναι απαραίτητο το αρχείο token.h (header file), το οποίο περιέχει την αντίστοιχη τιμή επιστροφής για κάθε token. Τα tokens ουσιαστικά είναι ονόματα τα οποία αντιπροσωπεύουν τις κανονικές εκφράσεις που έχουν τοποθετηθεί ως προς αναγνώριση (εξηγείται και εντός του αρχείου του λεκτικού αναλυτή).

2.1 Ανάλυση εισόδου & εξόδου λεκτικού αναλυτή

Σε αυτό το κεφάλαιο θα γίνει ανάλυση της εισόδου και της εξόδου καθώς η σύγκριση αυτών. Αρχικά θα αναφερθεί το πως κατασκευάστηκε η είσοδος και τι περιέχει μέσα αλλά και τι περιμένουμε από τον λεκτικό αναλυτή. Έπειτα θα υπάρξει ανάλυση της εξόδου στην οποία θα επαληθεύσουμε ότι αυτά που αναφέρθηκαν είναι ορθά και ότι ο λειτουργικός αναλυτής μας λειτουργεί όπως πρέπει.

2.1.1 Ανάλυση εισόδου

Επειδή το αρχείο εισόδου είναι σημαντικού μεγέθους, θα γίνει ανάλυση του σε όσο το δυνατόν καλύτερα χωρισμένα κομμάτια γίνεται. Το αρχείο αυτό βρίσκεται διαθέσιμο και στο τελικό παραδοτέο .zip με όνομα input.txt. Ξεκινήσαμε γράφοντας μία συνάρτηση main, έτσι ώστε να τεστάρουμε για αρχή τον εντοπισμό των identifiers, των keywords, του συμβόλου “,” αλλά και τις παρενθέσεις. Έπειτα ξεκινήσαμε από τη δοκιμή κάθε είδους αριθμού (δεκαδικοί, δεκαεξαδικοί, ακέραιοι, οκταδικοί, εκθετικοί).

```
int main(int argc, char** argv) {

    // testing numbers
    int num  =   5;  // whitespaces are ignored
    int num1 = -10;  // this is not recognized as a number, it's an operator and a number
    int num2 = 0x14FAC2;
    int num3 = 0;
    int num4 = 012442562378; // the analyzer will separate the 8 from the other part of the number
    float result = 3.14e-10;
    float b = 1.1e3.14 // this is not a valid number

    // Arithmetical operators
    result +=  num1 + (num2 * num3);
    result -=  num2 - (num3 / num1);
    #modulo /= num1 % num1;

    char str1[24] = "Compilers Lab"; // testing braces

    // Comparison operators and string literals
    if (num1 >= num2)
        printf("num1 > num2\n");
    else if (num1 <= num2)
        printf("num1 < num2\\");
    else if (num1 !== num2)
        printf("numbers are not equal\"");
```

Για τους ακεραίους δοκιμάσαμε ένα απλό ακέραιο, στον οποίο όμως τοποθετήσαμε και μερικά έξτρα spaces για να ελέγξουμε και τον εντοπισμό των whitespace από τον αναλυτή. Αυτό που περιμένουμε στην έξοδο είναι η αγνόηση των whitespaces από τον αναλυτή και η αναγνώριση του αριθμού. Συνεχίζοντας με τους αριθμούς δοκιμάσαμε να βάλουμε έναν αρνητικό αριθμό. Στην ουσία όμως οι αρνητικοί αριθμοί δεν αναγνωρίζονται ως λεκτικές μονάδες από μόνοι τους, αναγνωρίζονται ως τον συνδυασμό ενός operator συνοδευόμενο από έναν αριθμό. Οπότε περιμένουμε να το δούμε αυτό στην έξοδο μας μετέπειτα. Περαιτέρω δοκιμάσαμε έναν δεκαεξαδικό αριθμό (num2) καθώς και το 0 μόνο του (num3). Στην Πέμπτη περίπτωση (num4), εισάγουμε έναν οκταδικό αριθμό, ο οποίος όμως στο τελευταίο του ψηφίο έχει το 8, το οποίο δεν είναι επιτρεπτό στους οκταδικούς. Οπότε στην έξοδο μας περιμένουμε να δούμε χωρισμένο τον οκταδικό αριθμό από το 8. Και οι δύο όμως αναγνωρίζονται ως ακέραιοι (integers) . Το 3.14e-10 θα αναγνωριστεί κανονικά ως float αριθμός. Το 1.1e3.14 δεν μπορεί να αναγνωριστεί ολόκληρο, επειδή το 1.1e3 είναι όντως αριθμός, αλλά το .14 δεν είναι. Συγκεκριμένα το σύμβολο . (τελεία) επιστρέφεται ως unknown token, και ο αριθμός ως απλά ένα integer.

Κάτω από τους αριθμούς τοποθετήσαμε ένα identifier ακολουθούμε από braces τα οποία περιέχουνε αριθμό μέσα ([24]), δηλαδή σε μορφή πίνακα, έτσι ώστε να ελέγξουμε ότι ο λεκτικός αναλυτής αναγνωρίζει ξεχωριστά τα braces και τον αριθμό. Είναι σημαντικό να αναφερθεί πως στα παραπάνω χρησιμοποιούνται λέξεις που περιγράφουν τύπους δεδομένων όπως int, float.

Μετέπειτα δοκιμάσαμε τους αριθμητικούς τελεστές. Αρχικά βλέπουμε τον τελεστή += ο οποίος αποτελεί έναν τελεστή μόνος του και όχι ως δύο διαφορετικούς τελεστές + και =. Σε αυτή καθώς και στην επόμενη σειρά στον κώδικα, έχουμε τοποθετήσει και παρενθέσεις, οπότε περιμένουμε να δούμε και αυτές στην έξοδο. Στο τρίτο παράδειγμα (#modulo), ο αναλυτής περιμένουμε να μην το αναγνωρίσει ως ένα identifier, αλλά να ξεχωρίσει το σύμβολο # από το identifier modulo. Παρακάτω ελέγχουμε τους συγκριτικούς τελεστές τους οποίους έχουμε τοποθετήσει σε δομές ελέγχου if, εντός των οποίων τεστάρουμε κάποια string literals σε συνδυασμό με Escape Sequence characters. Οπότε σε

αυτό το κομμάτι κώδικα, περιμένουμε να δούμε τους τελεστές αναγνωρισμένους από τον αναλυτή, εκτός από τον τελευταίο (!==), ο οποίος πρέπει να αναγνωρισθεί ξεχωριστά ως '!=' και '=' ξεχωριστά.

Αυτές οι λέξεις αναγνωρίζονται από τον λεκτικό αναλυτή ως keywords. Η λίστα των keywords βρίσκεται αναλυτικά στο .l αρχείο στο οποίο περιέχεται ο κώδικας του λεκτικού αναλυτή. Συνεχίζουμε λοιπόν την ανάλυση του αρχείου εισόδου.

```
    // Logical operators and arithmetic operators
    if (num1 > 0 && num2 > 0) {
        printf("Both num1 and num2 are positive\n");
    }
    else if (num1 < 0 | num2 != 0 || num3 == 0) {
        do {
             --result;
             result++;

             if(result == 0) break;
        } while(!num4 == 0 || num3 +- = 0);
    }

    /* the combination as a whole won't be recognized as an identifier.
       ~ is an unknown token and @ is a symbol. */
    ~var@ = 5;

    /* This is an unterminated multi-line comment that will cause the analyzer to break
       because of EOF. Because of this, in the output of the lexical analyzer (output.txt)
       we will see that the brackets of the main function never close.

    return 0;
}
```

Στην αρχική δομή if, ο συνδυασμός &&& δεν αποτελεί κάποιο είδος τελεστή, οπότε πρέπει να αναγνωριστεί ξεχωριστά ως '&&' και '&'. Αντίστοιχα στην δομή else if βλέπουμε το σύμβολο '|' το οποίο πρέπει δεν αποτελεί κάποιο σύμβολο της αλφαβήτου της γλώσσας (unknown token), ενώ ο δεύτερος είναι όντως σωστός και στην ουσία παριστά την λογική πράξη OR. Μέσα στην επανάληψη do while βρίσκονται κάποιοι αριθμητικοί τελεστές που δεν έχουν προαναφερθεί. Οι τελεστές – και ++ είναι τελεστές που πρέπει να εντοπισθούν ως μία λεκτική μονάδα και όχι ξεχωριστά. Στα παραπάνω αναφέρονται δομές ελέγχους και δομές επανάληψης όπως if … else if και do {…} while και στη συνέχεια εμφανίζεται η εντολή break. Οι λέξεις αυτές εντοπίζονται από τον λεκτικό αναλυτή ως keywords. Συνεχίζοντας παρατηρούμε την ύπαρξη του τελεστή +- ο οποίος δεν αποτελεί τελεστή. Βγαίνοντας από την δομή ελέγχου if … else if, τοποθετήσαμε τον χαρακτήρα ~ ο οποίος δεν αποτελεί έγκυρο τελεστή και αναγνωρίζεται από τον αναλυτή ως UNKNOWN_TOKEN. Το var είναι ένα identifier και το @ είναι σύμβολο. Ο συνδυασμός τους όμως δεν αποτελούν identifier ως σύνολο.

Τέλος έχουμε περιλάβει μία περίπτωση η οποία μας προκάλεσε κάποια προβλήματα. Έχουμε τοποθετήσει σχόλια πολλαπλών γραμμών, τα οποία δεν κλείνουν (*/). Μαζί με το return 0; και τον χαρακτήρα '}' που σηματοδοτεί το κλείσιμο της main συνάρτησης, υπάρχει και το EOF (End Of File). Ένα σχόλια δεν μπορεί να περιέχει μέσα το end of file καθώς το end of file είναι το τέλος του αρχείου, πράγμα το οποίο σημαίνει ότι το σχόλιο δεν θα μπορέσει να τερματίσει ποτέ. Παρακάτω, σε επόμενες υποενότητες, γίνεται πλήρης ανάλυση του προβλήματος που αναφέρθηκε, καθώς και ενός γενικότερου προβλήματος που αντιμετωπίσαμε με τα σχόλια πολλαπλών γραμμών. Ας περάσουμε τώρα στην ανάλυση εξόδου του λεκτικού αναλυτή.

2.1.2 Ανάλυση εξόδου

Στην ενότητα αυτή θα πραγματοποιηθεί ανάλυση της εξόδου του λεκτικού αναλυτή. Το αρχείο εξόδου είναι ιδιαίτερα μεγάλο σε μέγεθος, οπότε θα γίνει μία προσπάθεια να οργανωθεί σε όσο το δυνατόν καλύτερα πλαίσια ώστε να είναι κατανοητό. Επιπλέον, οτιδήποτε αναφέρεται παρακάτω αφορά το αρχείο εισόδου που έχει αναλυθεί στην προηγούμενη υποενότητα 2.1.1. Συνεπώς όλα τα αποτελέσματα και οι γραμμές που θα παρουσιαστούν παρακάτω βασίζονται εκεί, οπότε δεν χρειάζεται να το επαναπεριλάβουμε.

Αναφέρεται πως το format της γραμμής εξόδου του λεκτικού αναλυτή δίνεται ως εξής

```
Line= αριθμός γραμμής, UNKNOWN TOKEN, value="τιμή λανθασμένης συμβολοσειράς"
```

Το αριστερό κομμάτι του output παρακάτω περιέχει τις γραμμές 2, 5 και 6 που αφορούν τον εντοπισμό της main καθώς και μερικών αριθμών. Το δεξί κομμάτι αντίστοιχα περιέχει αριθμός προς ανάλυση. Τα αποτελέσματα αριστερά είναι σωστά και ο αναλυτής εντοπίζει τα πάντα σωστά. Αντιθέτως στα αποτελέσματα δεξιά υπάρχουν δύο παρατηρήσεις οι οποίες αναφέρθηκαν και στην ανάλυση εισόδου. Στη γραμμή 8 παρατηρείται ο διαχωρισμός μεταξύ του οκταδικού αριθμού και του ακεραίου 8 επειδή το 8 δεν ανήκει στους οκταδικούς αριθμούς, οπότε ο αναλυτής των ξεχώρισε ως ακέραιο. Άλλη μία παρατήρηση είναι στη γραμμή 10. Η γλώσσα δεν μας επιτρέπει να έχουμε δύναμη πραγματικούς αριθμούς. Συνεπώς 1.1e3 είναι σωστό, αλλά το υπόλοιπο κομμάτι δεν συνεπάγεται σε κάτι, για αυτό και το έχει εντοπίσει την . (τελεία) ως UNKNOWN TOKEN, και το 14 ως αριθμό

```
Line=2, token=KEYWORD, value="int"
Line=2, token=IDENTIFIERS, value="main"
Line=2, token=OPEN_PARENTHESIS, value="("
Line=2, token=KEYWORD, value="int"
Line=2, token=IDENTIFIERS, value="argc"
Line=2, token=SYMBOLS, value=","
Line=2, token=IDENTIFIERS, value="char"
Line=2, token=OPERATORS, value="*"
Line=2, token=OPERATORS, value="*"
Line=2, token=IDENTIFIERS, value="argv"
Line=2, token=CLOSE_PARENTHESIS, value=")"
Line=2, token=OPEN_BRACKET, value="{"
Line=5, token=IDENTIFIERS, value="num1"
Line=5, token=OPERATORS, value="="
Line=5, token=OPERATORS, value="-"
Line=5, token=INTEGER, value="10"
Line=5, token=DELIMITER, value=";"
Line=6, token=IDENTIFIERS, value="num2"
Line=6, token=OPERATORS, value="="
Line=6, token=INTEGER, value="0x14FAC2"
Line=6, token=DELIMITER, value=";"
```

```
Line=7, token=KEYWORD, value="int"
Line=7, token=IDENTIFIERS, value="num3"
Line=7, token=OPERATORS, value="="
Line=7, token=INTEGER, value="0"
Line=7, token=DELIMITER, value=";"
Line=8, token=KEYWORD, value="int"
Line=8, token=IDENTIFIERS, value="num4"
Line=8, token=OPERATORS, value="="
Line=8, token=INTEGER, value="01244256237"
Line=8, token=INTEGER, value="8"
Line=8, token=DELIMITER, value=";"
Line=9, token=KEYWORD, value="float"
Line=9, token=IDENTIFIERS, value="result"
Line=9, token=OPERATORS, value="="
Line=9, token=FLOAT, value="3.14e-10"
Line=9, token=DELIMITER, value=";"
Line=10, token=KEYWORD, value="float"
Line=10, token=IDENTIFIERS, value="b"
Line=10, token=OPERATORS, value="="
Line=10, token=FLOAT, value="1.1e3"
Line=10, token=UNKNOWN TOKEN, value="."
Line=10, token=INTEGER, value="14"
```

Έπειτα από τη δοκιμή των αριθμών, έγιναν δοκιμές σχετικά με τους αριθμητικούς τελεστές. Δοκιμάστηκαν απλοί συντελεστές όπως +, -, *, /, αλλά και πιο σύνθετοι όπως +=, -+, /=. Επιπλέον έγινε έλεγχος του συμβόλου # πάνω σε ένα identifier. Ο λεκτικός αναλυτής πρέπει να το ξεχωρίσει αυτό, καθώς ένας identifier δεν μπορεί να ξεκινάει με το σύμβολο #. Παρακάτω δίνονται τα κομμάτια της εξόδου που μας εμφάνισε ο λεκτικός αναλυτής.

```
Line=13, token=IDENTIFIERS, value="result"
Line=13, token=OPERATORS, value="+="
Line=13, token=IDENTIFIERS, value="num1"
Line=13, token=OPERATORS, value="+"
Line=13, token=OPEN_PARENTHESIS, value="("
Line=13, token=IDENTIFIERS, value="num2"
Line=13, token=OPERATORS, value="*"
Line=13, token=IDENTIFIERS, value="num3"
Line=13, token=CLOSE_PARENTHESIS, value=")"
Line=13, token=DELIMITER, value=";"
Line=14, token=IDENTIFIERS, value="result"
Line=14, token=OPERATORS, value="-="
Line=14, token=IDENTIFIERS, value="num2"
Line=14, token=OPERATORS, value="-"
Line=14, token=OPEN_PARENTHESIS, value="("
Line=14, token=IDENTIFIERS, value="num3"
Line=14, token=OPERATORS, value="/"
Line=14, token=IDENTIFIERS, value="num1"
Line=14, token=CLOSE_PARENTHESIS, value=")"
Line=14, token=DELIMITER, value=";"
```

```
Line=15, token=UNKNOWN TOKEN, value="#"
Line=15, token=IDENTIFIERS, value="modulo"
Line=15, token=OPERATORS, value="/="
Line=15, token=IDENTIFIERS, value="num1"
Line=15, token=OPERATORS, value="%"
Line=15, token=IDENTIFIERS, value="num1"
Line=15, token=DELIMITER, value=";"
Line=17, token=IDENTIFIERS, value="char"
Line=17, token=IDENTIFIERS, value="str1"
Line=17, token=OPEN_BRACE, value="["
Line=17, token=INTEGER, value="24"
Line=17, token=CLOSE_BRACE, value="]"
Line=17, token=OPERATORS, value="="
Line=17, token=STRINGS, value=""Compilers Lab""
Line=17, token=DELIMITER, value=";"
```

Αριστερά παρατηρούμε ότι τα εντοπίζει σωστά χωρίς κάποιο πρόβλημα. Στο δεξί όμως υπάρχουν κάποιες παρατηρήσεις. Εντοπίζει σωστά το ότι η δίεση (#) είναι Unknown token και όχι μέρος του identifier. Επιπλέον, στη γραμμή 17 παρατηρούμε πως εντοπίζει σωστά τα braces και τον αριθμό εντός αυτών, διαχειρίζοντας επίσης σωστά το string. Προχωρώντας βλέπουμε τον εντοπισμό συγκριτικών τελεστών.

```
Line=20, token=KEYWORD, value="if"
Line=20, token=OPEN_PARENTHESIS, value="("
Line=20, token=IDENTIFIERS, value="num1"
Line=20, token=OPERATORS, value=">="
Line=20, token=IDENTIFIERS, value="num2"
Line=20, token=CLOSE_PARENTHESIS, value=")"
Line=21, token=IDENTIFIERS, value="printf"
Line=21, token=OPEN_PARENTHESIS, value="("
Line=21, token=STRINGS, value=""num1 > num2\n""
Line=21, token=CLOSE_PARENTHESIS, value=")"
Line=21, token=DELIMITER, value=";"
Line=22, token=KEYWORD, value="else"
Line=22, token=KEYWORD, value="if"
Line=22, token=OPEN_PARENTHESIS, value="("
Line=22, token=IDENTIFIERS, value="num1"
Line=22, token=OPERATORS, value="<="
Line=22, token=IDENTIFIERS, value="num2"
Line=22, token=CLOSE_PARENTHESIS, value=")"
```

```
Line=23, token=IDENTIFIERS, value="printf"
Line=23, token=OPEN_PARENTHESIS, value="("
Line=23, token=STRINGS, value=""num1 < num2\\""
Line=23, token=CLOSE_PARENTHESIS, value=")"
Line=23, token=DELIMITER, value=";"
Line=24, token=KEYWORD, value="else"
Line=24, token=KEYWORD, value="if"
Line=24, token=OPEN_PARENTHESIS, value="("
Line=24, token=IDENTIFIERS, value="num1"
Line=24, token=OPERATORS, value="!="
Line=24, token=OPERATORS, value="="
Line=24, token=IDENTIFIERS, value="num2"
Line=24, token=CLOSE_PARENTHESIS, value=")"
Line=25, token=IDENTIFIERS, value="printf"
Line=25, token=OPEN_PARENTHESIS, value="("
Line=25, token=STRINGS, value=""numbers are not equal\""
Line=25, token=CLOSE_PARENTHESIS, value=")"
Line=25, token=DELIMITER, value=";"
```

Στα αριστερά όλοι οι συγκριτικοί τελεστές είναι έγκυροι. Στα δεξιά όμως υπάρχει ο τελεστής !==, ο οποίος δεν είναι έγκυρος και ο αναλυτής το εντοπίζει ως δύο ξεχωριστούς τελεστές, το διάφορο (!=) και την ανάθεση τιμής = (ίσων). Παρακάτω θα αναλύσουμε τους λογικούς τελεστές, δηλαδή AND, OR, καθώς και λοιπούς αριθμητικούς τελεστές. Το κομμάτι αυτό περιέχει μία δομή if … else if και μία δομή do … while, οπότε η έξοδος του αναλυτή θα βγεί σχετικά μεγάλη σε μέγεθος. Οπότε θα αναλύσουμε πρώτα την δομή if ... else if μόνη της ώστε να είναι πιο ευανάγνωστη η ανάλυση.

```
Line=28, token=KEYWORD, value="if"
Line=28, token=OPEN_PARENTHESIS, value="("
Line=28, token=IDENTIFIERS, value="num1"
Line=28, token=OPERATORS, value=">"
Line=28, token=INTEGER, value="0"
Line=28, token=OPERATORS, value="&&"
Line=28, token=OPERATORS, value="&"
Line=28, token=IDENTIFIERS, value="num2"
Line=28, token=OPERATORS, value=">"
Line=28, token=INTEGER, value="0"
Line=28, token=CLOSE_PARENTHESIS, value=")"
Line=28, token=OPEN_BRACKET, value="{"
Line=29, token=IDENTIFIERS, value="printf"
Line=29, token=OPEN_PARENTHESIS, value="("
Line=29, token=STRINGS, value=""num1 and num2 > 0\n""
Line=29, token=CLOSE_PARENTHESIS, value=")"
Line=29, token=DELIMITER, value=";"
```

```
Line=30, token=CLOSE_BRACKET, value="}"
Line=31, token=KEYWORD, value="else"
Line=31, token=KEYWORD, value="if"
Line=31, token=OPEN_PARENTHESIS, value="("
Line=31, token=IDENTIFIERS, value="num1"
Line=31, token=OPERATORS, value="<"
Line=31, token=INTEGER, value="0"
Line=31, token=UNKNOWN TOKEN, value="|"
Line=31, token=IDENTIFIERS, value="num2"
Line=31, token=OPERATORS, value="!="
Line=31, token=INTEGER, value="0"
Line=31, token=CLOSE_PARENTHESIS, value=")"
Line=31, token=OPEN_BRACKET, value="{"
```

Ξεκινόντας από τα αριστερά βλέπουμε πως όλα εντοπίζονται σωστά. Μάλιστα ο αναλυτής, εντοπίζει τον τελεστή &&& ως διαφορετικούς τελεστές && (Logical AND Operation) και το σύμβολο της διεύθυνσης &. Βλέποντας στο δεξί μέρος, βλέπουμε ότι περίπου στη μέση της γραμμής 31, ο αναλυτής έχει αναγνωρίσει το σύμβολο | ως UNKNOWN_TOKEN καθώς αυτό δεν αποτελεί σύμβολο της αλφαβήτου της γλώσσας. Αν ήταν συνοδευμένο από άλλο ένα τέτοιο σύμβολο, σχηματίζοντας την λογική πράξη OR (||), τότε ο αναλυτής θα το εντόπιζε ως operator. Τώρα θα γίνει η ανάλυση της εξόδου που αφορά την δομή επανάληψης do … while.

```
Line=32, token=KEYWORD, value="do"
Line=32, token=OPEN_BRACKET, value="{"
Line=33, token=OPERATORS, value="--"
Line=33, token=IDENTIFIERS, value="result"
Line=33, token=DELIMITER, value=";"
Line=34, token=IDENTIFIERS, value="result"
Line=34, token=OPERATORS, value="++"
Line=34, token=DELIMITER, value=";"
Line=36, token=KEYWORD, value="if"
Line=36, token=OPEN_PARENTHESIS, value="("
Line=36, token=IDENTIFIERS, value="result"
Line=36, token=OPERATORS, value="=="
Line=36, token=INTEGER, value="0"
Line=36, token=CLOSE_PARENTHESIS, value=")"
Line=36, token=KEYWORD, value="break"
Line=36, token=DELIMITER, value=";"
```

```
Line=37, token=CLOSE_BRACKET, value="}"
Line=37, token=KEYWORD, value="while"
Line=37, token=OPEN_PARENTHESIS, value="("
Line=37, token=OPERATORS, value="!"
Line=37, token=IDENTIFIERS, value="num4"
Line=37, token=OPERATORS, value="=="
Line=37, token=INTEGER, value="0"
Line=37, token=OPERATORS, value="||"
Line=37, token=IDENTIFIERS, value="num3"
Line=37, token=OPERATORS, value="+"
Line=37, token=OPERATORS, value="-"
Line=37, token=OPERATORS, value="="
Line=37, token=INTEGER, value="0"
Line=37, token=CLOSE_PARENTHESIS, value=")"
Line=37, token=DELIMITER, value=";"
Line=38, token=CLOSE_BRACKET, value="}"
```

Στα αριστερά φαίνεται ολόκληρο το σώμα της do ... while στην οποία ο αναλυτής εντοπίζει όλες τις λεκτικές μονάδες (keywords, brackets, operators κλπ.) σωστά. Στο δεξί μέρος παρατηρούμε την συνθήκη της while, η οποία περιέχει αρκετούς τελεστές, τόσο συγκριτικούς, όσο αριθμητικούς και λογικούς. Μετά από τον τελεστή || παρατηρείται ένας μη έγκυρος τελεστής +- =. Ο αναλυτής ξεχωρίζει ότι αυτό δεν αποτελεί τελεστή, εμφανίζοντας τους τρείς αυτούς τελεστές ως διαφορετικούς. Ο αναλυτής επίσης, δεν αναγνωρίζει το -= ως τελεστή, διότι ανάμεσα στο – και το = υπάρχει κενό, το οποίο λέει στον αναλυτή ότι αυτοί οι τελεστές δεν είναι μαζί, ο ένας μετά τον άλλο. Τέλος έχουμε ένα μικρό κομματάκι κώδικα στο οποίο ελέγχουμε unknown tokens μαζί με identifiers. Ο αναλυτής σωστά καταλαβαίνει πως η τίλντα (~) δεν αποτελεί επιτρεπτό σύμβολο ως αρχή ενός identifier. Επίσης, πολύ σωστά εντοπίζει ότι το σύμβολο @ μετά από το αναγνωριστικό δεν αποτελεί μέρος αυτού.

```
Line=42, token=UNKNOWN TOKEN, value="~"
Line=42, token=IDENTIFIERS, value="var"
Line=42, token=UNKNOWN TOKEN, value="@"
Line=42, token=OPERATORS, value="="
Line=42, token=INTEGER, value="5"
Line=42, token=DELIMITER, value=";"
```

Περαιτέρω στο πρόγραμμα, υπάρχει το κομμάτι με το μη τερματισμένο σχόλιο το οποίο μέσα του περιέχει το EOF. Ο αναλυτής δεν το παίρνει καθόλου υπόψη, όμως στον κώδικα του έχουμε βάλει να τυπώνει ένα μήνυμα στην οθόνη το οποίο απλώς λέει “ERROR: EOF in comment”, υποδεικνύοντας το πρόβλημα του EOF εντός του σχολίου. Το συγκεκριμένο θέμα έχει καλυφθεί με μεγαλύτερη ακρίβεια παρακάτω στην ενότητα 2.2.2 Πρόβλημα ύπαρξης EOF εντός multi-line comment.

2.2 Αναφορά προβλημάτων κατά την υλοποίηση του αρχείου εισόδου

Και οι δύο υποενότητες που αναλύονται παρακάτω αφορούν προβλήματα σχετικά με τα σχόλια πολλαπλών γραμμών. Η επίλυση των προβλημάτων αυτών υλοποιήθηκε με την χρήση προγράμματος το οποίο αναφέρετε εντός των υποενοτήτων παρακάτω.

2.2.1 Πρόβλημα εντόπισης κειμένου αναμεσα σε multi-line comment

Υπήρξαν κάποια προβλήματα κατά τη υλοποίηση του κώδικα για την εύρεση των σχολίων. Συγκεκριμένα το πρόβλημα αφορούσε τη σωστή εύρεση και διαχείριση των σχολίων πολλαπλής γραμμής. Αν είχε τοποθετηθεί σχόλιο πολλαπλών γραμμών, έπειτα ακολουθούσε μία αλληλουχία από λεκτικές μονάδες (κώδικα) και μετά υπήρχε ξανά σχόλιο πολλαπλών γραμμών, τότε ο λεκτικός αναλυτής δεν εντόπιζε τον κώδικα που ήταν ανάμεσα στα σχόλια. Αυτό το πρόβλημα με τη σειρά του μας δημιούργησε το πρόβλημα της λανθασμένης μέτρησης του αριθμού των γραμμών που όντως αποτελούν κομμάτι κώδικα. Για την καλύτερη κατανόηση του προβλήματος, παρακάτω παρατίθεται ένα απλό παράδειγμα το οποίο μας προκαλούσε το πρόβλημα που προαναφέρθηκε.

```
/* sample
   multiple line
   comment */

int main(int argc, char** argv) {
   // sample code that will be ignored
   // due to the problem described
   int a = 5;
   int b = 10;

   /* another sample
      multiple line
      comment */
   printf("Sum: %d", a+b); // This will be the first lexical unit recognized by the analyzer
   return 0;
}
```

Οπότε το πρόβλημα ήταν ότι ο λεκτικός αναλυτής δεν έβρισκε τα σχόλια πολλαπλών γραμμών ως μία μονάδα. Όπως φαίνεται και παραπάνω, κανονικά θα έπρεπε το πρώτο σχόλιο να ήταν μία μονάδα μόνο του, και το δεύτερο άλλη. Αλλά στο πρόβλημα αυτά θεωρούνταν ένα σχόλιο, επειδή απλώς βρέθηκε το closing tag '*/' των σχολίων. Για αυτό και παρακάτω αναπτύχθηκε η συνάρτηση handle_comment η οποία επιλύει το παραπάνω πρόβλημα. Ο κώδικας της συνάρτησης αντλήθηκε από το αρχείο "Παραδείγματα FLEX αρχείων" το οποίο υπάρχει στο E-Class στα έγγραφα του εργαστηρίου.

Ο κώδικας που δίνεται κάνει διαχείριση των σχολίων χωρίς όμως να διαχειρίζεται σωστά και τον αριθμό των γραμμών. Για αυτό αλλάξαμε τον κώδικα προσθέτοντας μερικές εντολές οι οποίες

διασφαλίζουν ότι η μέτρηση των γραμμών γίνεται σωστά. Ο τροποποιημένος κώδικας φαίνεται παρακάτω συνοδευμένος με σχόλια όπου χρήζει εξήγηση.

```
// Implementation of function that handles multi-line comments
void handle_comment()
{
   /* this function handles multi-line comments
      It detects them, but it does not take them into account
      and handles the line counting accordingly */

   register int c;

   for (;;)
   {
      /* while character read is not '*' (indicating comment end) or EOF
         this loop basically eats up the text inside the comment. */

      while ((c = input()) != '*' && c != 0)
         if (c == '\n') line++;

      // if there is a '*' found, indicating (possibly) the end of the comment
      if (c == '*')
      {
         /* the loop below detects if there are many
            '*' symbols in a row, basically skipping them
            and not regarding them as the end of the comment. */

         while ((c = input()) == '*');

         if (c == '/') break; // found the end of the comment.
      }

      // if there is EOF contained in the multiple line comment
      if (c == 0)
      {
         printf ("Error: EOF in comment.\n");
         break;
      }
   }
}
```

Το πρόγραμμα που βρίσκεται παραπάνω είναι μια τροποποιημένη εκδοχή του προγράμματος που βρίσκεται στο E-Class. Ο κώδικας αυτός καλείται εντός των brackets του comment token (“/*”). Τα return values και γενικότερα οι ιδιότητες των tokens έχουν την εξής δομή στον κώδικα του λεκτικού αναλυτή. Για αυτό τον λόγο τον τοποθετήσαμε τον κώδικα που επιλύει το πρόβλημα μας σε μία συνάρτηση που ονομάσαμε handle_comment, ώστε να είναι πιο ευανάγνωστο.

```
"/*"                { handle_comment(); }
{LINE_COMMENT}      { /* ignore line comments */}
{DELIMITER}         { return DELIMITER; }
{INTEGER}           { return INTEGER; }
{FLOAT}             { return FLOAT; }
{STRINGS}           { return STRINGS; }
```

2.2.2 Πρόβλημα ύπαρξης EOF εντός multi-line comment

Επιπλέον, διακρίναμε θέματα κατά την λεκτική ανάλυση του input.txt αρχείου σχετικά με τον εντοπισμό του EOF (End Of File), συνεπώς τα αποτελέσματα μας ήταν λανθασμένα. Αυτό μας οδήγησε στην άντληση πληροφοριών από εξωτερικές πηγές και συγκεκριμένα από την πηγής που αναφέρεται παρακάτω.

Μετά από αναζητήσεις στο διαδίκτυο καταφέραμε να αντλήσουμε πληροφορίες από την εξής πηγή:

https://stackoverflow.com/questions/73767676/flex-cant-handle-eof-in-action-or-how-to

στην οποία αναφέρεται πως σε παλαιότερες εκδόσεις (έως και την 2.6.0), η συνάρτηση input() του εργαλείου FLEX, επέστρεφε EOF κάθε φορά που εντόπιζε τέλος του αρχείου (end of file). Από την 2.6.1 και μετά, η συνάρτηση input() υπέστη αλλαγές και πλέον οι συνάρτηση επιστρέφει 0 όταν εντοπίζει end of input. Οπότε αυτό ήταν ένα workaround του προβλήματος στο οποίο απαιτήθηκε η χρήση εξωτερικής πηγής για την άντληση πληροφοριών. Το κομμάτι που προκαλεί το πρόβλημα βρίσκεται στο τέλος του αρχείο εισόδου το οποίο βρίσκεται και παραπάνω και είναι το εξής:

```
/* This is an unterminated multi-line comment
       that will cause the analyzer to break because of EOF.
       Because of this, in the output of the lexical analyzer (output.txt)
       we will see that the brackets of the main function never close.

    return 0;
}
```

2.3 Ελλείψεις και ορθή ή μη μεταγλώττιση και εκτέλεση

Κατά την γνώμη μας δεν υπάρχει κάποια έλλειψη σε σχέση με τα ζητούμενα που δόθηκαν στην εκφώνηση της άσκησης. Η εκτέλεση και η μεταγλώττιση πραγματοποιήθηκαν με επιτυχία χρησιμοποιώντας το αρχείο Makefile, στο οποίο τοποθετήσαμε τις εντολές που παρουσιάζονται παρακάτω, ώστε να τρέξουμε τον λεκτικό μας αναλυτή. Αρχικά, με την βοήθεια της εντολής flex, μπορούμε να παράξουμε το .c αρχείο αναλυτή από το αντίστοιχο .l αρχείο. Αφού παραχθεί το .c αρχείο το κάνουμε compile με τη χρήση της εντολής gcc, παράγοντας έτσι το τελικό executable αρχείο. Τέλος, εκτελούμε το αρχείο αυτό με τα ορίσματα που επιθυμούμε. Τα αρχεία αυτά μπορεί να είναι οποιαδήποτε αρχεία εισόδου τα οποία έχουν ουσία να τεθούν προς ανάλυση από τον αναλυτή. Αν δοθεί ένα αρχείο ως όρισμα (π.χ input.txt) τότε όταν το πρόγραμμα τερματίσει , θα εμφανίσει τα αποτελέσματα στο τερματικό. Αν δώσουμε και αρχείο εξόδου ως δεύτερο όρισμα (π.χ output.txt), τότε τα αποτελέσματα που παράγονται από τον λεκτικό αναλυτή τοποθετούνται στο αντίστοιχο αρχείο εξόδου που ορίσθηκε.

```
all:
   flex -o lexical_analyzer.c lexical_analyzer.l
   gcc -o lexical_analyzer lexical_analyzer.c
   ./lexical_analyzer input.txt output.txt
```

3. Ανάλυση αρμοδιοτήτων

3.1 Αρμοδιότητες στον κώδικα και στον έλεγχο

	Κοντούλης Δημήτριος (21390095)	**Μεντζέλος Άγγελος Κωνσταντίνος (21390132)**	**Βάρσου Ευφροσύνη (21390021)**	**Γκιόζι Εντερίσα (21390041)**	**Αλεξόπουλος Λεωνίδας (2139006)**
Υλοποίηση κώδικα		√			
Διόρθωση κώδικα	√				
Έλεγχος εξόδου			√		
Δημιουργία εισόδου	√				√ Συμπλήρωση
Σχολιασμός κώδικα		√		√ Συμπλήρωση	

3.2 Υλοποίηση word

		Κοντούλης Δημήτριος (21390095)	**Μεντζέλος Άγγελος Κωνσταντίνος (21390132)**	**Βάρσου Ευφροσύνη (21390021)**	**Γκιόζι Εντερίσα (21390041)**	**Αλεξόπουλος Λεωνίδας (2139006)**
1. Εισαγωγή				√		
2. Τεκμηρίωση	2.1.1 Ανάλυση εισόδου	√				
	2.1.2 Ανάλυση εξόδου	√		√		
	2.2.1 Πρόβλημα στα multi-line comment		√			
	2.2.2 Πρόβλημα EOF σε multi-line comment	**√ Αντιμετώπιση**	**√ Εύρεση**			
	2.3 Ελλείψεις και ορθή ή μη μεταγλώττιση και εκτέλεση		√			
3. Ανάλυση αρμοδιοτήτων			√			