Απαλλακτική Εργασία στο μάθημα

Γλώσσες Προγραμματισμού και Μεταγλωττιστές

Ακ. Έτος: 2022-2023

Ονοματεπώνυμο: Γεώργιος Δάλλας

ΑΕΜ: 4116

0.Εισαγωγή

Ο σκοπός της εργασίας είναι η δημιουργία ενός μεταγλωττιστή, ο οποίος παράγει κώδικα σε γλώσσα μηχανής και συμβολικής αναπαράστασης MIXAL, για τον φανταστικό υπολογιστή του D.Knuth, γνωστό και ως MIX. Ο πηγαίος κώδικας της γλώσσας, βρίσκεται στα αρχεία **lexicalAnalysis.l**, **yaccFile.y** καθώς και στα αρχεία που βρίσκονται στον φάκελο **filesForYacc**. Η εκτελέσιμη μορφή βρίσκεται στο αρχείο **a.exe**. Για να παραχθεί η εκτελέσιμη μορφή της γλώσσας απαιτείται να εκτελεστούν οι παρακάτω εντολές στο παράθυρο τερματικού μέσα στον φάκελο όπου βρίσκονται τα αρχεία:

> flex lexicalAnalysis.l

> yacc –dv yaccFile.y

> gcc lex.yy.c y.tab.c

Για την παραγωγή του αρχείου με τον κώδικα σε MIXAL απαιτείται η εκτέλεση της παρακάτω εντολής:

> a < “(το αρχειο με τον κωδικα για μεταγλωτιση)”

Το τελικό αρχείο με την γλώσσα στόχο που παράγεται ονομάζεται **assembly.mix** και για την εκτέλεση του χρησιμοποιείται το **MIXBuilder**, όπου αναπαριστά εικονικά έναν υπολογιστή MIX και επιτρέπει την εκτέλεση αρχείων MIXAL.

1. Λεξική Ανάλυση

Το πρώτο από τα στάδια μεταγλώττισης, είναι η λεξική ανάλυση. Για το στάδιο αυτό χρησιμοποιείται το εργαλείο **flex**. Ο κώδικας υλοποίησης για αυτό βρίσκεται στο αρχείο “**lexicalAnalysis.l**”. Η λεξική ανάλυση δέχεται συμβολοσειρές και παράγει κατάλληλες διευκρινιστικές λεξικές μονάδες για την επεξεργασία τους στο επόμενο στάδιο της Συντακτικής Ανάλυσης. Η διαδικασία παραγωγής αυτών, γίνεται μέσω χρήσης κανονικών εκφράσεων. Στην κανονική έκφραση, για την αναγνώριση αριθμών, μπορεί να παρατηρηθεί ότι αναγνωρίζονται μόνο θετικοί αριθμοί. Το χαρακτηριστικό αυτό είναι σκόπιμο. Για λόγους απλοποίησης, αναγνωρίζεται και ενώνεται ο χαρακτήρας “-“ από τους αριθμούς έπειτα, στο στάδιο παραγωγής της γλώσσας στόχου, χωρίς να προστίθεται πολυπλοκότητα. Στο στάδιο αυτό επιπλέον, όταν

εντοπίζεται η δήλωση νέας γραμμής (\n), αυξάνεται κατά ένα η μεταβλητή countn, όπου χρησιμοποιείται κυρίως κατά την εμφάνιση της προβληματικής γραμμής σε ένα σφάλμα.

2. Συντακτική Ανάλυση

Το στάδιο αυτό είναι το πιο περίπλοκο όλου του μεταγλωττιστή και έχει ως σκοπό την παραγωγή ενός αφηρημένου συντακτικού δέντρου. Το κύριο αρχείο που χρησιμοποιείται για την συντακτική ανάλυση είναι το "**yaccFile.y"** καθώς και το εργαλείο **yacc**, το οποίο παράγει την συντακτική ανάλυση. Επιπρόσθετα, κατά την διάρκεια της συντακτικής ανάλυσης, χρησιμοποιείται και το αρχείο "**yaccHeaders.h",** όπου περιέχει διάφορες δηλώσεις για την συντακτική ανάλυση, το αρχείο "**symbolTable.h**", το οποίο παράγει παράλληλα έναν πίνακα συμβόλων χρησιμοποιώντας την συνάρτηση addSymbol για την προσθήκη τους, ιδιαίτερα χρήσιμο για το επόμενο βήμα, την σημασιολογική ανάλυσης και τέλος, το αρχείο "**abstractSyntaxTree.h**" το οποίο περιέχει την δήλωση της δομής για την δημιουργία του αφηρημένου συντακτικού δέντρου καθώς και απαραίτητες συναρτήσεις για την χρήση της. Η κύρια συνάρτηση που διασυνδέει τους κόμβους για την παραγωγή του συντακτικού δέντρου είναι η addNode, όπου δημιουργεί έναν νέο κόμβο στο δέντρο, καθορίζει τον αριστερό και δεξή κόμβους παιδιά του και δέχεται μια ακόμη παράμετρο για το καθοριστικό όνομα του κόμβου αυτού.

3. Σημασιολογική Ανάλυση

Το στάδιο αυτό στον συγκεκριμένο μεταγλωττιστή είναι το πιο απλό καθώς υπάρχει μόνο ένας τύπος δεδομένων, άρα το μόνο που χρειάζεται να ελεγχθεί από την σημασιολογική ανάλυση, είναι εάν τα ονόματα μεταβλητών έχουν δηλωθεί. Εδώ, χρησιμοποιείται ο πίνακας συμβόλων που αναφέρθηκε στο προηγούμενο βήμα, όπου ελέγχονται όλα του τα σύμβολα και αν βρεθεί ομοιότητα τότε επιτρέπεται η χρήση της μεταβλητής σε μια εντολή. Αλλιώς παράγεται σφάλμα και η μετάφραση δεν είναι δυνατή. Η συνάρτηση που εφαρμόζει τον έλεγχο αυτό ονομάζεται checkIfDeclared, καλείται στην χρήση των μεταβλητών σε διάφορες πράξεις και υπάρχει στο αρχείο "**yaccFile.y"**. Σε περίπτωση δήλωσης μεταβλητής με ήδη δηλωμένο όνομα, χρησιμοποιείται η συνάρτηση checkExistance στον πίνακα συμβόλων και αν βρεθεί ομοιότητα τότε παράγει σφάλμα, αλλιώς επιτρέπει την προσθήκη της νέας μεταβλητής στον πίνακα συμβόλων.

4. Παραγωγή ενδιάμεσης αναπαράστασης

Για την ενδιάμεση αναπαράσταση του κώδικα, επιλέχτηκε η υλοποίηση κώδικα τριών διευθύνσεων. Το στάδιο υλοποιείται στο αρχείο "**threeAddressCode.h**" και δέχεται σαν είσοδο το αφηρημένο συντακτικό δέντρο από το δεύτερο στάδιο. Η συνάρτηση που παράγει τον κώδικα αυτόν, είναι η generateThreeAddressCode και ελέγχει το όνομα του κόμβου, παράγει τον κατάλληλο κώδικα για τον κόμβο αυτό, και στην συνέχεια καλεί την ίδια συνάρτηση για την παραγωγή κώδικα για τους κόμβους παιδιά του κάθε κόμβου. Επιπλέον, κατά την παραγωγή του κώδικα αυτού, σε κάποιες περιπτώσεις παράγονται επιπλέον βοηθητικές μεταβλητές που διευκολύνουν κάποιες πράξεις του μεταγλωττιστή, οπου ονομάζονται HELPV. Σε κάθε κόμβο που εξετάζεται από την συνάρτση αυτή, προστίθενται γραμμές με κώδικα ενδιάμεσης αναπαράστασης στο αρχείο "**threeAddressCode.txt**". Το στάδιο είναι απαραίτητο στην

προετοιμασία της γλώσσας για την παραγωγή της assembly, καθώς μετατρέπει περίπλοκα κομμάτια όπως τις λούπες while, for και την εντολη if σε απλοποιημένο κώδικα, όμοιο με αυτό της assembly, χρησιμοποιώντας labels και την εντολή goto για την υλοποίηση αυτών.

5. Βελτιστοποίηση

Το στάδιο βελτιστοποίησης υλοποιείται στο αρχείο “**optimization.h**”, δέχεται σαν είσοδο το αρχείο “**threeAddressCode.txt**” και παράγει το αρχείο “**optimizedCode.txt**”. Στο στάδιο αυτό, αφαιρούνται εντολές που δεν επηρεάζουν το τελικό αποτέλεσμα στο πρόγραμμα (πρόσθεση και αφαίρεση με 0, πολλαπλασιασμός και διαίρεση με το 1), μετατρέπει τον πολλαπλασιασμό με 0, στην απλή εξίσωση της μεταβλητής με το 0, και μετατρέπει τον πολλαπλασιασμό και την διαίρεση με δυνάμεις του 2 σε εντολές shit left και shift right που είναι ταχύτερες. Γενικά ο στόχος της βελτιστοποίησης, είναι η μετατροπή του κώδικα για ταχύτερη εκτέλεση της γλώσσας στόχου. Αυτό επιτυγχάνεται είτε με την απόλυτη απαλοιφή κομματιών κώδικα που δεν γίνονται προσβάσιμα κατά την εκτέλεση του κώδικα ή κατά την μετατροπή των εντολών σε εντολές που απαιτούν λιγότερες κύκλους ρολογιού για την ολοκλήρωση τους, όπως και τις μετατροπές που εφαρμόζονται στον συγκεκριμένο μεταγλωττιστή.

6. Παραγωγή γλώσσας στόχου

Η παραγωγή της τελικής γλώσσας στόχου, δηλαδή η παραγωγή της MIXAL assembly, υλοποιείται στο αρχείο “**genMIXAL.h**”, δέχεται σαν είσοδο το αρχείο που παράγεται απο το προηγούμενο στάδιο “**optimizedCode.txt**” και παράγει το αρχείο “**assembly.mix**”, όπου είναι και το τελικό αρχείο που τροφοδοτείται έπειτα στο εργαλείο **MIXBuilder**, δηλαδή στον εικονικό υπολογιστή MIX για εκτέλεση. Στο στάδιο αυτό, επιλέγεται ο κατάλληλος συνδυασμός από το instruction set του υπολογιστή για την παραγωγή των εντολών και εφαρμόζονται για την μετατροπή της βελτιστοποιημένης ενδιάμεσης αναπαράστασης στην γλώσσα αυτή. Λόγω της περιορισμένης μνήμης που διατίθεται στον υπολογιστή MIX, χρησιμοποιούνται υπορουτίνες για την εκτέλεση κάποιων εντολών, έτσι ώστε να αποφευχθεί η επανάληψη ομοιότυπου κώδικα. Οι υπορουτίνες που υλοποιήθηκαν είναι για τον έλεγχο υπερχείλισης, για την εντολή “!”, για την εντολή εκτύπωσης, για την εντολή “&&: (and) και “||” (or) καθώς και βοηθητικές ρουτίνες για την επιστροφή λογικού true ή false. Για την εντολή εκτύπωσης, η γλώσσα εκτυπώνει + αν ο αριθμός είναι θετικός και – αν είναι αρνητικός και έπειτα ανεξαρτήτως του μεγέθους του αριθμού 10 νούμερα, όπου είναι και το μέγιστο πλήθος αριθμών που υποστηρίζει ο υπολογιστής για έναν ακέραιο. Δηλαδή, ο υπολογιστής εάν ο ακέρεος είναι μικρότερος από δεκα νούμερα εκτυπώνει μυδενικά και μετά ακολουθεί ο αριθμός. Μια ακόμη ιδιαιτερότητα της μηχανής είναι ότι ο μέγιστος ακέραιος που υποστηρίζεται είναι 31 bit, άρα οι επιτρεπτοί αριθμοί βρίσκονται στο εξείς εύρος: -1.073.741.823 <= n <= 1.073.741.823. Σε περιπτώσεις υπερχείλισης του αριθμού αυτού κατά την διάρκεια εκτέλεσης του προγράμματος, επιλέχθηκε η υλοποίηση ενημερωτικού μηνύματος-σφάλματος και έπειτα ο τερματισμός του προγράμματος αυτού. Το ίδιο μύνημα-σφάλμα εμφανίζεται σε περιπτώσεις διαίρεσης με το 0. Το στάδιο αυτό, θα μπορούσε να παράγει διάφορα αποτελέσματα που να θεωρούνταν σωστά. Παρ’ όλα αυτά δόθηκε προσοχή ώστε να μην παράγει περιττές εντολές λόγω της περιορισμένης μνήμης. Έτσι σε διάφορα μέρη γίνεται έλεγχος για κυριολεκτικές τιμές ακεραίων, και γίνεται άμεση εφαρμογή τους με χρήση του “=literalValue=” που προσφέρει η MIXAL, αντί για χρήση βοηθητικών μεταβλητών. Τέλος κατά

τον έλεγχο αυτό για χρήση κυριολεκτικών ακεραίων τιμών, είναι που εφαρμόζονται και οι αρνητικοί αριθμοί χωρίς περιττές εντολές ή πολυπλοκότητα, συνδυάζοντας τους άμεσα με το αρνητικό πρόσημο, σαν να διαβάζονταν έτσι και από τον Λεξικό Αναλυτή. Η δήλωση των μεταβλητών γίνεται ξεχωριστά, μετά από το πεδίο εφαρμογής εντολών του κώδικα MIXAL και πριν το πεδίο δήλωσης των υπορουτινών.

7. Παραδείγματα χρήσης μεταγλωτιστή

- Παραδειγμα με υπερχήληση:

testOverflow

Desktop > c-- > C--Compiler > testOverflow

```
1  {
2      var i : int;
3      var j : int;
4      j=2;
5      for(i=2; i<1000; i=i+1){
6          j*=i;
7          print j;
8      }
9  }
```

Devices

Card Reader (16) | Card Punch (17) | Line Printer (18) | Typewriter (19) | Paper Tape (20)

```
   +0000000004
   +0000000012
   +0000000048
   +0000000240
   +0000001440
   +0000010080
   +0000080640
   +0000725760
   +0007257600
   +0079833600
   +0958003200
AN OVERFLOW OCCURED OR DIVISION BY ZERO.PLEASE CHECK YOUR CODE AND TRY AGAIN.
```

- Παράδειγμα με διαίρεση με 0:

testDivisionByZero U

Desktop > c-- > C--Compiler > testDivisionByZero

```
1  {
2      var i : int;
3      var j : int;
4      i=100;
5      print i;
6      print j;
7      i= i/j;
8      print i;
9  }
```

Devices

Card Reader (16) | Card Punch (17) | Line Printer (18) | Typewriter (19) | Paper Tape (20)

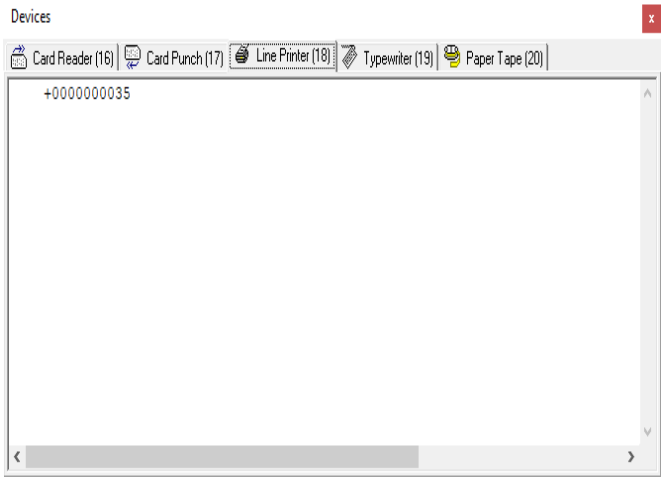
```
   +0000000100
   +0000000000
AN OVERFLOW OCCURED OR DIVISION BY ZERO.PLEASE CHECK YOUR CODE AND TRY AGAIN.
```

- Υπολογισμός περίπλοκης πράξης:

testComplexExp M

Desktop > c-- > C--Compiler > testComplexExp

```
1   {
2       var num1 : int;
3       var num2 : int;
4       var result : int;
5       var waste : int;
6       num1 = -10;
7       num2 = 3;
8       result = -(num1 + (num2 * -128) / 16 - 1);
9       print result;
10  }
11
```

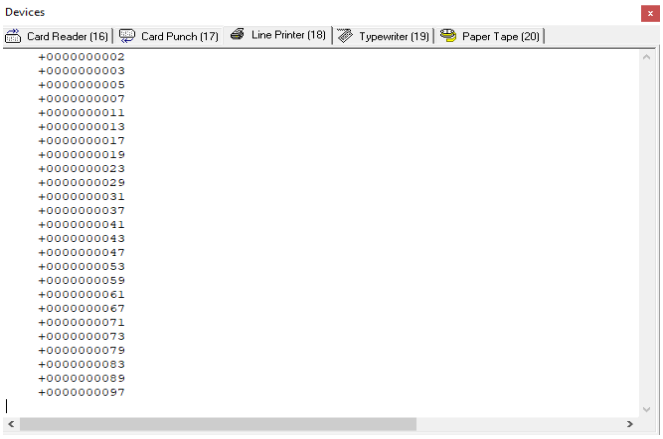


- Παράδειγμα εύρεση πρώτων αριθμών:

testPrimeNumbers

Desktop > c-- > C--Compiler > testPrimeNumbers

```
1   {
2       var n : int;
3       var i : int;
4       var j : int;
5       var isPrime : int;
6       n = 100;
7       i = 2;
8       while (i <= n) {
9           isPrime = 1;
10
11          j = 2;
12          while (j <= i / 2) {
13              if (i % j == 0) {
14                  isPrime = 0;
15                  break;
16              }
17              j = j + 1;
18          }
19          if (isPrime == 1) {
20              print i;
21          }
22          i = i + 1;
23      }
24  }
```



- Παράδειγμα αθροίσματος περιττών αριθμών:

testSumOfEven

Desktop > c-- > C--Compiler > testSumOfEven

```
1   {
2       var limit : int;
3       var sum : int;
4       var numb : int;
5
6       limit = 10;
7       sum = 0;
8       numb = 1;
9
10      while (numb <= limit) {
11          if (numb % 2 == 0) {
12              print(numb);
13              sum = sum + numb;
14          }
15          numb = numb + 1;
16      }
17      print(sum);
18  }
```