Πανεπιστήμιο Πειραιώς

Ημερομηνία: 7 Ιανουαρίου 2024

Σύνταξη από:

Αντώνιος Τσαλμπούρης, Π22272

Δημήτριος Λαζάνας, Π22082

Μεταγλωττιστές - Εργασία 2023-24

Θέμα 1ο

Επιλογή Γλώσσας: C++

C++ Version: 11

Αρχείο Λύσης: `thema1.cpp`

Εντολή Μεταγλώττισης

```
g++ -std=c++11 thema1.cpp -o thema1
```

Ζήτημα: Υλοποίηση ενός Ντετερμινιστικού Αυτόματου Στοίβας που αναγνωρίζει εκφράσεις υπό τους εξής όρους:

- Ίδιος αριθμός χαρακτήρων “x” και “y”.
- Κατά την ανάγνωση από αριστερά προς δεξιά, οι χαρακτήρες “y” δεν πρέπει ποτέ να υπερβαίνουν τους “x”.
- Εκτύπωση της αλληλουχίας βημάτων για αναγνώριση ή απόρριψη της έκφρασης.

Υλοποίηση

Class AutomatoStoivas

- **Περιγραφή:** Η κλάση AutomatoStoivas διαθέτει δομή στοίβας “char” και περιλαμβάνει μεθόδους για επεξεργασία και αναγνώριση (ή απόρριψη) των εκφράσεων, καθώς και για την εκτύπωση των ακολουθηθέντων βημάτων.
- **Συνάρτηση processExpression():** Δέχεται την έκφραση από την `main()`. Για κάθε χαρακτήρα "x" προσθέτει ένα "x" στη στοίβα, ενώ για κάθε "y" ελέγχει τη στοίβα και αφαιρεί ένα "x" εφόσον ικανοποιούνται οι όροι. Σε περίπτωση ασυμφωνίας, τυπώνεται μήνυμα απόρριψης.
- **Συνάρτηση printStack():** Εκτυπώνει τα περιεχόμενα της στοίβας, χρησιμοποιώντας βοηθητική στοίβα για την εμφάνιση των στοιχείων της.

main()

- **Περιγραφή:** Δημιουργεί ένα αντικείμενο της κλάσης AutomatoStoivas, ζητά την έκφραση από τον χρήστη, καλεί την `processExpression()` για τους απαραίτητους ελέγχους και τερματίζει το πρόγραμμα.

Παραδείγματα Εκτέλεσης

1. Παράδειγμα Αναγνώρισης

```
 Type expression: xxyyxyxy
Push X --> Stack: x
Push X --> Stack: xx
Pop X --> Stack:  x
Pop X --> Stack:
Push X --> Stack: x
Pop X --> Stack:
Push X --> Stack: x
Pop X --> Stack:
Expression Accepted!
```

2. Δύο Παραδείγματα Απόρριψης

```
 Type expression: yxxy
Expression Rejected: 'y' does not match to a 'x'
```

```
Type expression: xxyyy
Push X --> Stack: x
Push X --> Stack: xx
Pop X --> Stack:  x
Pop X --> Stack:
Expression Rejected: 'y' does not match to a 'x'
```

Θέμα 2ο

Επιλογή Γλώσσας: C

Αρχείο Λύσης: `randomStringGenerator.c`

Εντολή Μεταγλώττισης

```
gcc randomStringGenerator.c -o randomStringGenerator
```

Μας ζητήθηκε να σχεδιάσουμε και να υλοποιήσουμε μια γεννήτρια συμβολοσειρών για την παρακάτω γραμματική, λαμβάνοντας μέριμνα η διαδικασία να τερματίζεται οπωσδήποτε.

```
<Z>::=(<K>)
<K>::=<G><M>
<G>::=v|<Z>
<M>::=-<K>|+<K>|ε, όπου ε η κενή συμβολοσειρά
```

ΥΛΟΠΟΙΗΣΗ

Εισαγωγή Βιβλιοθηκών

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

Ανάλυση κάθε Βιβλιοθήκης

1. `<stdio.h>`

Η βιβλιοθήκη `<stdio.h>` (Standard Input/Output Header) χρησιμοποιείται για την εκτέλεση εισόδου και εξόδου στο πρόγραμμα. Συγκεκριμένα, αυτή η βιβλιοθήκη παρέχει δυνατότητες για την εκτύπωση στην οθόνη (μέσω της συνάρτησης `printf`) και για την ανάγνωση εισόδου από τον χρήστη αν χρειαστεί. Στο συγκεκριμένο πρόγραμμα, χρησιμοποιείται για την εμφάνιση μηνυμάτων και των παραγόμενων αλφαριθμητικών.

2. `<stdlib.h>`

Η `<stdlib.h>` (Standard Library Header) περιέχει συναρτήσεις για διαχείριση μνήμης, αλλά και για την παραγωγή τυχαίων αριθμών, που είναι βασικό για τη λειτουργία αυτού του προγράμματος. Η συνάρτηση `rand()` χρησιμοποιείται για την παραγωγή τυχαίων αριθμών που καθορίζουν την επιλογή των ακολουθιών κατά τη δημιουργία του αλφαριθμητικού.

3. `<time.h>`

Η βιβλιοθήκη `<time.h>` περιλαμβάνει συναρτήσεις για τη διαχείριση και την απόκτηση πληροφοριών για χρόνο και ημερομηνία. Σε αυτό το πρόγραμμα, χρησιμοποιείται για την αρχικοποίηση του σπόρου της τυχαίας αριθμοποίησης μέσω της συνάρτησης `srand(time(NULL))` . Αυτό εξασφαλίζει ότι οι τυχαίοι αριθμοί που παράγονται είναι διαφορετικοί κάθε φορά που εκτελείται το πρόγραμμα.

Κάθε βιβλιοθήκη που έχει επιλεγεί για αυτό το πρόγραμμα παίζει συγκεκριμένο ρόλο στην υλοποίηση και την αποδοτικότητα του κώδικα, συμβάλλοντας στην ολοκληρωμένη λειτουργία και την αλληλεπίδραση με τον χρήστη.

Για την ικανοποίηση των ζητουμένων (σε C) χρησιμοποιήθηκαν εκτός από την `main()` άλλες 4 συναρτήσεις (μία για κάθε κανόνα παραγωγής). Αναλυτικά, αυτές ήταν οι:

- `generate_Z(char *result, int *pos, int depth)`
- `generate_K(char *result, int *pos, int depth)`
- `generate_G(char *result, int *pos, int depth)`
- `generate_M(char *result, int *pos, int depth)`

Κάθε μία από αυτές, δέχεται ως παράμετρο το buffer `result`, όπου αποθηκεύονται οι χαρακτήρες που δημιουργήθηκαν, την μεταβλητή `pos` που λειτουργεί ως index για το buffer των χαρακτήρων (και αυξάνεται κατά 1 κάθε φορά που γίνεται κάποια καταχώρηση στο `result`) και τέλος την μεταβλητή `depth` που μετράει πόσες φορές έγινε επιλογή μη-τερματικού χαρακτήρα αντί για τερματικού.

Οι 4 αυτές συναρτήσεις δηλώνονται σε global scope προκειμένου να μπορούν να κληθούν από οπουδήποτε στο πρόγραμμα. Βάσει του τρόπου υλοποίησης μας, η μία συνάρτηση καλεί την άλλη μέχρι ότου να φτάσουμε στην τελική συμβολοσειρά. Κατά αυτό το τρόπο, όταν παρακολουθούμε τα βήματα που ακολούθησε η τυχαία γεννήτρια στο τερματικό, φαίνονται σαν η συμβολοσειρά να δημιουργήθηκε από τα αριστερά προς τα δεξιά.

Στους κανόνες αντικατάστασης του `G` και του `M`, υπάρχουν διαφορετικές επιλογές για την αντικατάσταση αυτών των χαρακτήρων. Προκειμένου να διασφαλίσουμε την τυχαιότητα της επιλογής του κανόνα αντικατάστασης, χρησιμοποιούμε μια γεννήτρια τυχαίων αριθμών στις αντίστοιχες συναρτήσεις `generate_G` και `generate_M`.

Κάθε φορά, αφού γίνει η επιλογή του κανόνα βάσει του οποίου θα γίνει αντικατάσταση του στοιχείου, εμφανίζουμε στον χρήστη τον κανόνα αυτό καθώς και το αποτέλεσμα της έκφρασης, ως έχει, εκείνη τη δεδομένη στιγμή. Ο τρόπος που επιλέξαμε να διαχειριστούμε το «ε» (κενό) ήταν απλώς να τυπώσουμε στον χρήστη τον κανόνα που επιλέχθηκε και να του δείξουμε την έκφραση δίχως να κάνουμε κάποια αλλαγή σε αυτή.

Επιπλέον, όπως αναφέραμε και προηγουμένως, έπρεπε να διασφαλίσουμε πως ο αλγόριθμος μας, θα επέλεγε κάποια στιγμή έναν κανόνα που οδηγεί σε τερματικό σύμβολο παρά την τυχαιότητα του. Έτσι, κάθε φορά που επιλεγόταν τυχαία κάποιος μη τερματικός κανόνας αυξάναμε την τιμή της μεταβλητής `depth` κατά 1. Κατ' επέκταση σε κάθε επιλογή κανόνα προβλέψαμε εκτός από την τυχαιότητα, εάν η τιμή της μεταβλητής `depth` είναι μεγαλύτερη από 10, τότε να γίνεται απευθείας η επιλογή του κανόνα που οδηγεί σε τερματικό σύμβολο.

Περιγραφή Συναρτήσεων

Συνάρτηση generate_Z

```
void generate_Z(char *result, int *pos, int depth)
{
    ...
}
```

Η `generate_Z` αρχικοποιεί το αλφαριθμητικό με έναν αριστερό παρενθετικό και καλεί την `generate_K`.

Συνάρτηση generate_K

```
void generate_K(char *result, int *pos, int depth)
{
    ...
}
```

Η `generate_K` καλεί τις `generate_G` και `generate_M` για να συνεχίσει τη δημιουργία του αλφαριθμητικού.

Συνάρτηση generate_G

```
void generate_G(char *result, int *pos, int depth)
{
    ...
}
```

Η `generate_G` προσθέτει είτε έναν χαρακτήρα 'v' εί

τε καλεί αναδρομικά την `generate_Z`.

Συνάρτηση generate_M

```
void generate_M(char *result, int *pos, int depth)
{
    ...
}
```

Η `generate_M` μπορεί να προσθέσει τους χαρακτήρες '-' ή '+' και να καλέσει αναδρομικά την `generate_K` ή να μην προσθέσει τίποτα (επιστρέφοντας το κενό αλφαριθμητικό).

Τέλος, στην κύρια συνάρτηση `main()`, δεν γίνεται τίποτα παραπάνω παρά αρχικοποιήσεις μεταβλητών (`result`, `pos`, γεννήτριας αριθμών) καθώς και τοποθέτηση του σπόρου για την τυχαία παραγωγή αριθμών και κλήση της πρώτης συνάρτησης `generate_Z` που ανταποκρίνεται στον πρώτο κανόνα αντικατάστασης. Αφού ολοκληρωθεί η δημιουργία της συμβολοσειράς και το πρόγραμμα επιστρέψει στην `main()`, τυπώνουμε για την χρήστη το τελικό αποτέλεσμα και το πρόγραμμα τερματίζεται.

Κύρια Συνάρτηση

```
int main()
{
    srand(time(NULL));
    char result[1000] = {0};
    int pos = 0;
    generate_Z(result, &pos, 0);
    printf("Generated string: %s\n", result);
    return 0;
}
```

Παράδειγμα αποτελέσματος εκτέλεσης του προγράμματος:

```
Applied rule `<Z>` ::= `(<K>)`
Generated string: `(`
Applied rule `<K>` ::= `<G><M>`
Applied rule `<G>` ::= `v`
Generated string: `(v`
Applied rule `<M>` ::= `+<K>`
Generated string: `(v+`
Applied rule `<K>` ::= `<G><M>`
Applied rule `<G>` ::= `v`
Generated string: `(v+v`
Applied rule `<M>` ::= `ε`
Generated string: `(v+v`
Generated string: `(v+v)`
```

Θέμα 3

Επιλογή Γλώσσας: C++

C++ Version: 11

Αρχεία Λύσης: `stringAnalyzer.cpp, node.cpp, node.h`

Εντολή Μεταγλώττισης

```
g++ -std=c++11  stringAnalyzer.cpp node.cpp -o stringAnalyzer
```

Πρόβλημα

Υλοποίηση συντακτικού αναλυτή top-down για την αναγνώριση ή απόρριψη συμβολοσειρών με βάση τη δοσμένη γραμματική. Η γραμματική έχει οριστεί ως εξής:

- G→ (M)
- M→ YZ
- Y→ a | b | G
- Z→ *M | -M | +M| ε (ε = κενή συμβολοσειρά)

Ο αναλυτής επιστρέφει το σχετικό δέντρο και εκτυπώνει την ανάλυση, με παράδειγμα την έκφραση ((a-b)*(a+b)).

Τύπος Γραμματικής

Ας εξετάσουμε αν η ακόλουθη γραμματική είναι LL(1). Για αυτό, χρειάζεται να υπολογίσουμε τα σύνολα FIRST και FOLLOW για κάθε μη τερματικό.

```
G → (M)
M → YZ
Y → a | b | G
Z → *M | -M | +M | ε
```

Σύνολα FIRST

```
FIRST(G) = {(}
FIRST(M) = FIRST(Y) = {a, b, (}
FIRST(Y) = {a, b, (}
FIRST(Z) = {*, -, +, ε}
```

Σύνολα FOLLOW

```
FOLLOW(G) = {$, *, +, -, )}
FOLLOW(M) = FOLLOW(Z) = {)}
FOLLOW(Y) = {*, +, -, )}
```

Συναρτήσεις EMPTY

```
EMPTY(G) = FALSE
EMPTY(M) = FALSE
EMPTY(Y) = FALSE
EMPTY(Z) = TRUE
```

Σύνολα LOOKAHEAD

```
LOOKAHEAD(G → (M)) = FIRST(() = {(}
LOOKAHEAD(M → Y Z) = FIRST(Y) = {a, b, (}

LOOKAHEAD(Y → a) = FIRST(a) = {a}
LOOKAHEAD(Y → b) = FIRST(b) = {b}
LOOKAHEAD(Y → G) = FIRST(G) = {(}

LOOKAHEAD(Z → *M) = FIRST(*) = {*}
LOOKAHEAD(Z → -M) = FIRST(-) = {-}
LOOKAHEAD(Z → +M) = FIRST(+) = {+}
LOOKAHEAD(Z → ε) = FOLLOW(Z) = {)}
```

Για να επιβεβαιώσουμε ότι η γραμματική είναι LL(1), πρέπει να εξασφαλίσουμε ότι τα σύνολα LOOKAHEAD για κάθε παραγωγή ενός μη τερματικού είναι ασυμβίβαστα, δηλαδή το διασταύρωσή τους είναι κενό σύνολο. Αυτό σημαίνει ότι δεν υπάρχουν κοινά στοιχεία στα LOOKAHEAD σύνολα των παραγωγών του ίδιου μη τερματικού, επιτρέποντας στον parser να αποφασίσει αμφισβητήσιμα ποια παραγωγή να χρησιμοποιήσει με βάση το επόμενο είσοδο σύμβολο.

```
LOOKAHEAD(Y → a) ∩ LOOKAHEAD(Y → b) = ∅
LOOKAHEAD(Y → a) ∩ LOOKAHEAD(Y → G) = ∅
LOOKAHEAD(Y → b) ∩ LOOKAHEAD(Y → G) = ∅

LOOKAHEAD(Z → *M) ∩ LOOKAHEAD(Z → -M) = ∅
LOOKAHEAD(Z → *M) ∩ LOOKAHEAD(Z → +M) = ∅
LOOKAHEAD(Z → *M) ∩ LOOKAHEAD(Z → ε) = ∅
LOOKAHEAD(Z → -M) ∩ LOOKAHEAD(Z → +M) = ∅
LOOKAHEAD(Z → -M) ∩ LOOKAHEAD(Z → ε) = ∅
LOOKAHEAD(Z → +M) ∩ LOOKAHEAD(Z → ε) = ∅
```

Καθώς καμία από τις παραγωγές του ίδιου μη τερματικού δεν έχει το ίδιο σύνολο LOOKAHEAD, η γραμματική επιβεβαιώνεται ως LL(1).

Εισαγωγή Βιβλιοθηκών

```
#include <iostream>
#include <map>
#include <string>
#include <set>
#include <vector>
#include <iomanip> // Για τη διαμόρφωση της εξόδου στην κονσόλα.
```

Ανάλυση Βιβλιοθηκών

- `<iostream>` : Χρήση για είσοδο/έξοδο στο πρόγραμμα (π.χ., `std::cout`).
- `<map>` : Δομή δεδομένων map για την αντιστοίχιση κλειδιών με τιμές.

- `<string>` : Διαχείριση συμβολοσειρών στο C++.
- `<set>` : Δομή συνόλου για αποθήκευση μοναδικών τιμών.
- `<vector>` : Δυναμικός πίνακας για αποθήκευση στοιχείων.
- `<iomanip>` : Παρέχει λειτουργίες για τη διαμόρφωση της εξόδου (π.χ., `std::setw` για τον καθορισμό του πλάτους εξόδου).

Κύριες Συναρτήσεις και Δομές

`Node` (`node.h`) Ορισμός Κλάσης

```
#ifndef NODE_H
#define NODE_H

#include <iostream>
#include <vector>

class Node {
public:
    Node(char name);
    ~Node();

    void push(const std::string &production);

    Node *getNextNode();
    char getName();
    std::vector<Node *> getChildren();
    void print

Children();

private:
    char name;
    Node *parent;
    Node *nextNode;
    std::vector<Node *> *children;

    void setParent(Node *parent);
    void addChild(Node *child);
    Node *climbUp();
    Node *searchNextNode(Node *currentNode);
    void setNextNode();
};

#endif // NODE_H
```

Η κλάση `Node` αναπαριστά έναν κόμβο στο δέντρο ανάλυσης. Κάθε κόμβος περιέχει ένα χαρακτήρα που αντιπροσωπεύει ένα μη τερματικό ή τερματικό στοιχείο της γραμματικής, καθώς και δείκτες προς τον γονέα και τα παιδιά του. Η κλάση παρέχει λειτουργίες για την προσθήκη παιδιών, την αναζήτηση επόμενων κόμβων, και την εκτύπωση των παιδιών για την οπτικοποίηση της δομής του δέντρου.

`initializeRulesTable()`

Αρχικοποιεί τον πίνακα κανόνων `rulesTable`, που περιέχει τους κανόνες της γραμματικής.

`initializeTerminalCharacters()`

Αρχικοποιεί το σύνολο `terminalCharacters` με τους τερματικούς χαρακτήρες της γραμματικής.

`isTerminal(char character)`

Ελέγχει αν ένας χαρακτήρας είναι τερματικός σύμφωνα με το σύνολο `terminalCharacters`.

`M(const std::string &nonTerminal, const std::string &character)`

Επιστρέφει την παραγωγή από τον πίνακα κανόνων βάσει ενός μη τερματικού και ενός χαρακτήρα.

`generateIsEmpty(const std::string& production)`

Ελέγχει αν μια παραγωγή είναι κενή μετά τον χαρακτήρα ' `>` '.

`stackPush(std::vector<char>& stack, const std::string& production)`

Εισάγει χαρακτήρες μιας παραγωγής στη στοίβα.

`stackPop(std::vector<char>& stack, bool print)`

Αφαιρεί τον επάνω χαρακτήρα από τη στοίβα και προαιρετικά εκτυπώνει τη στοίβα.

`terminate()`

Τερματίζει την εκτέλεση με μήνυμα απόρριψης της συμβολοσειράς.

`parse(const std::string& input)`

Αναλύει μια δοθείσα συμβολοσειρά και εκτυπώνει αν αυτή αναγνωρίζεται ή όχι.

`prettyPrintTree(Node* node, std::string prefix = "", bool isLast = true)`

Εκτυπώνει αναδρομικά ένα δέντρο ξεκινώντας από τον κόμβο `node`, δημιουργώντας μια δομημένη αναπαράσταση του δέντρου στην κονσόλα. Χρησιμοποιεί το `prefix` για την διαμόρφωση των επιπέδων του δέντρου και το `isLast` για να ελέγξει αν ο κόμβος είναι ο τελευταίος του γονέα του, προσθέτοντας το κατάλληλο σύμβολο σύνδεσης. Η συνάρτηση χρωματίζει τα ονόματα των κόμβων ανάλογα με το αν είναι τερματικά ή μη, χρησιμοποιώντας τη συνάρτηση `isTerminal`.

`main(int argc, char* argv[])`

Η κύρια συνάρτηση του προγράμματος εκκίνησης. Ακολουθούνται τα εξής βήματα:

1. **Αρχικοποίηση Κανόνων και Χαρακτήρων:** Καλεί τις `initializeRulesTable()` και `initializeTerminalCharacters()` για την αρχικοποίηση των κανόνων της γραμματικής και του συνόλου τερματικών χαρακτήρων αντίστοιχα.
2. **Χειρισμός Ορισμάτων Γραμμής Εντολών:** Έλεγχος αν έχει δοθεί ένα όρισμα μέσω της γραμμής εντολών (π.χ., `./stringAnalyzer "(a-b)*(a+b)"`). Αν ναι, χρησιμοποιεί αυτό το όρισμα ως την είσοδο για την ανάλυση.
3. **Λήψη Εισόδου Από Χρήστη:** Εάν δεν έχει δοθεί όρισμα γραμμής εντολών, ζητείται από τον χρήστη να εισάγει μια συμβολοσειρά.
4. **Χρήση Προεπιλεγμένης Συμβολοσειράς:** Αν ο χρήστης δεν παρέχει είσοδο, τότε το πρόγραμμα χρησιμοποιεί μια προεπιλεγμένη συμβολοσειρά για την ανάλυση, η οποία είναι `"((a-b)*(a+b))"`.
5. **Κλήση της `parse()`**: Η συμβολοσειρά (είτε από το όρισμα εντολής, είτε από τον χρήστη, είτε η προεπιλεγμένη) δίνεται στην `parse()` για ανάλυση.

Η `main()` είναι σχεδιασμένη για να είναι ευέλικτη στην επεξεργασία διαφορετικών συμβολοσειρών, επιτρέποντας τόσο τη διαδραστική λειτουργία με τον χρήστη όσο και την αυτοματοποιημένη δοκιμή μέσω ορισμάτων γραμμής εντολών.

Παράδειγμα αποτελέσματος εκτέλεσης του προγράμματος:

```
 Enter string to parse (leave empty for default):
Parsing: ((a-b)*(a+b))
$G                  ((a-b)*(a+b))$
$)M(                ((a-b)*(a+b))$
$)M                 (a-b)*(a+b))$
$)ZY                (a-b)*(a+b))$
$)ZG                (a-b)*(a+b))$
$)Z)M(              (a-b)*(a+b))$
$)Z)M               a-b)*(a+b))$
$)Z)ZY              a-b)*(a+b))$
$)Z)Za              a-b)*(a+b))$
$)Z)Z               -b)*(a+b))$
$)Z)M-              -b)*(a+b))$
$)Z)M               b)*(a+b))$
$)Z)ZY              b)*(a+b))$
$)Z)Zb              b)*(a+b))$
$)Z)Z               )*(a+b))$
$)Z)                )*(a+b))$
$)Z                 *(a+b))$
$)M*                *(a+b))$
$)M                 (a+b))$
$)ZY                (a+b))$
$)ZG                (a+b))$
$)Z)M(              (a+b))$
$)Z)M               a+b))$
$)Z)ZY              a+b))$
$)Z)Za              a+b))$
$)Z)Z               +b))$
```

```
$)Z)M+              +b))$
$)Z)M               b))$
$)Z)ZY              b))$
$)Z)Zb              b))$
$)Z)Z               ))$
$)Z)                ))$
$)Z                 )$
$)                  )$
$                   $
└── G
    ├── )
    ├── M
    │   ├── Z
    │   │   ├── M
    │   │   │   ├── Z
    │   │   │   │   └── e
    │   │   │   └── Y
    │   │   │       └── G
    │   │   │           ├── )
    │   │   │           ├── M
    │   │   │           │   ├── Z
    │   │   │           │   │   ├── M
    │   │   │           │   │   │   ├── Z
    │   │   │           │   │   │   │   └── e
    │   │   │           │   │   │
    │   │   │           │   │   │   └── Y
    │   │   │           │   │   │       └── b
    │   │   │           │   │   └── +
    │   │   │           │   └── Y
    │   │   │           │       └── a
    │   │   │           └── (
    │   │   └── *
    │   └── Y
    │       └── G
    │           ├── )
    │           ├── M
    │           │   ├── Z
    │           │   │   ├── M
    │           │   │   │   ├── Z
    │           │   │   │   │   └── e
    │           │   │   │   └── Y
    │           │   │   │       └── b
    │           │   │   └── -
    │           │   └── Y
    │           │       └── a
    │           └── (
    └── (
String is recognized
```

Σε αυτό το παράδειγμα εκτέλεσης, παρουσιάζεται η διαδικασία ανάλυσης προεπιλεγμένης συμβολοσειράς χρησιμοποιώντας τον αλγόριθμο. Η είσοδος της συμβολοσειράς και η αντίστοιχη ανάλυση απεικονίζεται βήμα προς βήμα, με την τελική διαπίστωση ότι η συμβολοσειρά αναγνωρίζεται σωστά.

Θέμα 4ο

Αρχείο Λύσης: `thema4.l`

Εντολές Μεταγλώττισης

- Γλώσσα Λύσης: FLEX σε συνδυασμό με C
- Για να μεταγλωττίσετε το πρόγραμμα, χρησιμοποιήστε τις ακόλουθες εντολές:

```
flex thema4.l
gcc lex.yy.c -o thema4
```

Μας ζητήθηκε να υλοποιήσουμε ένα πρόγραμμα FLEX (σε συνδυασμό με C) το οποίο θα αναγνωρίζει τα ονόματα σημείων ως την παράθεση ενός συμβόλου, τα ονόματα τριγώνων ως τη παράθεση 3 συμβόλων κ.ο.κ, έως και την περίπτωση οκταγώνων. Σημαντική παράμετρος επίσης είναι ότι δεν επιτρέπονται οι

επαναλήψεις συμβόλων όταν δίνεται ένα σχήμα.

ΥΛΟΠΟΙΗΣΗ

Για την υλοποίηση των ζητουμένων, και για την αποφυγή προβλημάτων encoding αποφασίσαμε το πρόγραμμα μας να χρησιμοποιεί μόνο λατινικούς χαρακτήρες. Το λεξιλόγιο που αναγνωρίζει το πρόγραμμα μας περιγράφεται αναλυτικά στο χρήστη με την εκκίνηση του προγράμματος, για πιο εύκολη εμπειρία χρήσης.

```
^shmeio[ \t]+[A-Z]{1}$ {
    printf("%s: Valid shmeio.\nEnter another input or enter 0 to exit.\n", yytext);
}
```

Στους κανόνες μετάφρασης του προγράμματος έχουμε δηλώσει όλα τα αποδεκτά μοτίβα προτάσεων. Κάθε φορά που το πρόγραμμα αναγνωρίζει το μοτίβο μίας έκφρασης, όπως αυτή περιεγράφηκε στους κανόνες, προβαίνει στις αντίστοιχες ενέργειες που προγραμματίστηκαν. Για παράδειγμα, στην πιο εύκολη υλοποίηση, στο «σημείο», κάθε φορά που αναγνωρίζεται είσοδος λέξης «σημείο» ακολουθούμενη από κενό και ένα κεφαλαίο γράμμα της λατινικής αλφαβήτου, εκτυπώνεται μήνυμα αποδοχής της έκφρασης. Δίνεται επίσης η επιλογή ο χρήστης να δώσει νέα είσοδο ή να εισάγει το '0' προκειμένου να τερματιστεί το πρόγραμμα (υπάρχει αντίστοιχος κανόνας μετάφρασης του '0' σε εντολή `exit(0);`)

```
^:αρχή γραμμής, $:τέλος γραμμής
```

Η διαδικασία αναγνώρισης έκφρασης περιπλέκεται περισσότερο όταν ο χρήστης εισάγει μεγαλύτερα σχήματα. Αυτό συμβαίνει διότι, πρέπει να κάνουμε τους απαραίτητους ελέγχους προκειμένου να διαπιστώσουμε εάν έχει επαναληφθεί κάποιο σύμβολο. Έτσι, για παράδειγμα όταν ο χρήστης εισάγει το σωστό πρότυπο έκφρασης για ένα τετράγωνο (δηλαδή η λέξη τετράγωνο ακολουθούμενη από κενό και πέντε κεφαλαίους χαρακτήρες), τότε εμείς γνωρίζοντας σε ποια θέση βρίσκεται ο πρώτος και ο τελευταίος χαρακτήρας ξεκινάμε μία διαδικασία 2 επαναλήψεων, όπου ελέγχεται κάθε κεφαλαίο σύμβολο εάν είναι ίδιο με κάποιο από τα επόμενα του. Σε περίπτωση που είναι, μέσω μιας μεταβλητής flag τυπώνουμε κατάλληλο μήνυμα (η flag μεταβλητή αρχικοποιείται ως αληθής και αν προκύψει ταύτιση συμβόλου με κάποιο άλλο αλλάζει σε ψευδής).

Ακολουθεί παράδειγμα κανόνα μετάφρασης για την περίπτωση του του τετραγώνου:

```
^tetragwno[ \t]+[A-Z]{4}$ {
    int firstLetter=10;
    int lastLetter=13;
    int flag=1;
    for(int i=firstLetter;i<lastLetter;i++){
        for(int j=i; j<lastLetter;j++){
            if(yytext[i]==yytext[j+1]){
                flag=0;
            }
        }
    }
    if(flag==1){
        printf("%s: Valid tetragwno.\nEnter another input or enter 0 to exit.\n", yytext);
    }else{
        printf("Wrong input: %s!\nEnter another input or enter 0 to exit.\n", yytext);
    }
}
```

Αντιστοίχως λειτουργούν και οι υπόλοιποι κανόνες μετάφρασης για τα άλλα σχήματα με μόνες διαφοροποιήσεις στο όνομα του σχήματος, στον αριθμό κεφαλαίων χαρακτήρων που αναμένονται, και στις μεταβλητές 'firstLetter' και 'lastLetter' που μαρτυρούν σε ποια θέση του πίνακα 'yytext' βρίσκονται το πρώτο και το τελευταίο σύμβολο κάθε σχήματος.

Συμπληρωματικά, έχουμε προσθέσει έναν κανόνα μετάφρασης ".+" για να εκτυπώνει μόνο 1 μήνυμα απόρριψης αντί για περισσότερα κάθε φορά που βλέπει μια «αλλόκοτη» έκφραση.

Τέλος, στις «βοηθητικές διαδικασίες», έχουμε τη συνάρτηση "yywrap" που χρειάζεται ο flex για να καταλάβει το τέλος μιας εισόδου, καθώς και την `main()` , η οποία τυπώνει στον χρήστη τα αρχικά μηνύματα (μεταξύ των οποίων και το «λεξικό»), καλεί την απαραίτητη συνάρτηση `yylex()` και τερματίζεται.

Παρακάτω δίνεται παράδειγμα εκτέλεσης του προγράμματος:

```
 Available phrases are:
1.shmeio
2.eutheia
3.trigwno
4.tetragwno
5.pentagwno
6.eksagwno
7.eptagwno
8.oktagwno
Please enter your input (or enter '0' to exit): simeio AB
Wrong input: simeio AB!
Enter another input or enter 0 to exit.

eutheia AB
eutheia AB: Valid eutheia.
Enter another input or enter 0 to exit.

trigwno ABC
trigwno ABC: Valid trigwno.
Enter another input or enter 0 to exit.

eptagwno ABCDEFG
eptagwno ABCDEFG: Valid eptagwno.
Enter another input or enter 0 to exit.

0
Exiting the program.
```