A COMPILER FOR SIMPLEPP

DOCUMENTATION

Zagkas Dimosthenis 4359 cs04359

Kalaitsidis Damianos 4370 cs04370

CONTENTS

INTRO

Το εξής πρότζεκτ υλοποιεί έναν μεταφραστή από την γλώσσα simplepp (simple-python-python) σε C. Γίνεται χρήση του εργαλείου ANTLR4 (Another Tool for Language Recognition) για την αυτοματοποίηση της γραμματικής ενώ η υλοποίηση του κώδικα γίνεται με java.

INTRODUCTION TO SIMPLEPP

Η simplepp (simple-python-python) είναι μία απλή μορφή της γλώσσας python, σχεδιασμένη συγκεκριμένα για το παρόν πρότζεκτ. Ακολουθούν λεπτομέρειες της γλώσσας καθώς και ιδιαιτερότητές της.

Όντας απλή μορφή της γλώσσας python, κάθε πρόγραμμα γραμμένο σε simplepp θα πρέπει να περνάει από διερμηνέα της Python.

Η γλώσσα ακολουθεί τα αντικειμενοστραφή πρότυπα των κλάσεων και του κυρίως προγράμματος της Python, όπου το κυρίως πρόγραμμα δεν δηλώνεται ούτε λαμβάνεται ως κλάση.

Κάθε κλάση δηλώνεται με την λέξη κλειδί *class* και ακολουθεί το όνομα της κλάσης, ενώ σε ένα πρόγραμμα υπάρχει τουλάχιστον μία κλάση.

Από την simplepp, υποστηρίζεται η λειτουργία της κληρονομικότητας, όπου η κλάση που κληρονομείται, τοποθετείται μέσα σε παρενθέσεις αμέσως μετά την κλάση που κληρονομεί. Όπως και στην Python, οι κλάσεις που κληρονομούνται, τοποθετούνται πριν τις κλάσεις που τις κληρονομούν.

Η ύπαρξη της *__init__* συνάρτησης ισχύει και στην simplepp, με τον ίδιο ακριβώς τρόπο, όπως και στην Python. Είναι υποχρεωτική για κάθε κλάση και σε αυτήν, γίνεται η δήλωση των παραμέτρων της, εάν υπάρχουν.

Οι μέθοδοι δηλώνονται με την λέξη κλειδί *def,* ακολουθεί το όνομα τους και μέσα σε παρένθεση, οι παράμετροί της.

Όπως και στην Python, η λέξη κλειδί *self* χρησιμοποιείται για την δήλωση των αντικειμένων στο οποίο εφαρμόζεται.

Στην simplepp, δεν υπάρχει δήλωση των πεδίων των κλάσεων και των τοπικών μεταβλητών των μεθόδων.

Η simplepp υποστηρίζει μόνο μεταβλητές τύπου integer (*int*).

Οι μέθοδοι μπορούν να δεχθούν ως παράμετρο αντικείμενα.

Η πρόσβαση στα πεδία των αντικειμένου γίνεται με τον συμβατικό τρόπο της Python, δηλαδή αντικείμενο, τελεία, πεδίο (*αντικείμενο.πεδίο).*

Λόγο της απλούστερης μορφής της γλώσσας που υλοποιούμαι, οι εντολές που υποστηρίζονται είναι περιορισμένες, δηλαδή: if-else, while, print και return, καθώς και η εκχώρηση.

Επίσης υποστηρίζονται λογικές συνθήκες και αριθμητικές εκφράσεις, με τη συνήθη προτεραιότητα τελεστών.

Η χρήση μία μεθόδου γίνεται με το όνομα του αντικειμένου που την περιέχει, ακολουθούμενο από μία τελεία, το όνομα της μεθόδου και τις παραμέτρους της σε παρένθεση.

Ακολουθεί εάν απλό παράδειγμα το οποίο η simplepp μεταγλωττίζει.

```
class Person:
    def __init__(self, pid, born) :
        self.pid = pid
        self.born = born
    def getPid(self):
        return self.pid
    def getBorn(self):
        return self.born
    def millenium(self):
```

```
            if self.born < 2000:
                return 1
            else:
                return 2

class Employee(Person):
    def __init__(self, pid, born, afm, department):
        self.pid = pid
        self.born = born
        self.afm = afm
        self.department = department
    def getDepartment(self):
        return self.department
    def setDepartment(self, department):
        self.department = department
    def getPid(self):
        return self.afm

class StupidPrint:
    def __init__(self, employee):
        print(employee.pid)
        print(employee.born)
        print(employee.afm)
        print(employee.department)

if __name__ == '__main__':
    george = Person(200223, 2002)
    john = Person(200055, 2000)
    peter = Employee(200122, 2001, 990122, 1)
    print(george.getBorn())
    print(john.getBorn())
    print(peter.getBorn())
    print(peter.millenium())
```

```
        print(peter.getDepartment())
        peter.setDepartment(2)
        print(peter.getDepartment())
        print(george.getPid())
        print(peter.getPid())
        stupid = StupidPrint(peter)
```

Το οποίο στην C μεταφράζεται ως εξής:

```
#include <stdio.h>

typedef struct {
   int pid;
   int born;
} Person;

void Person_init(Person *self, int pid, int born) {
   self->pid = pid;
   self->born = born;
}

int Person_getPid(Person *self) {
   return self->pid;
}

int Person_getBorn(Person *self) {
   return self->born;
}

int Person_millenium(Person *self) {
   if (self->born < 2000) {
      return 1;
```

```
    }
    return 0;
}
typedef struct {
    Person base;
    int afm;
    int department;
} Employee;

void Employee_init(Employee *self, int pid, int born, int afm, int department) {
    Person_init((Person *)self, pid, born);
    self->afm = afm;
    self->department = department;
}

int Employee_getDepartment(Employee *self) {
    return self->department;
}

void Employee_setDepartment(Employee *self, int department) {
    self->department = department;
}

int Employee_getPid(Employee *self) {
    return self->afm;
}

typedef struct {
    Employee *employee;
} StupidPrint;

void StupidPrint_init(StupidPrint *self, Employee *employee) {
    self->employee = employee;
```

```
    printf("%d\n", self->employee->base.pid);
    printf("%d\n", self->employee->base.born);
    printf("%d\n", self->employee->afm);
    printf("%d\n", self->employee->department);
}

int main() {
    Person george;
    Person_init(&george, 200223, 2002);
    Person john;
    Person_init(&john, 200055, 2000);
    Employee peter;
    Employee_init(&peter, 200122, 2001, 990122, 1);
    printf("%d\n", Person_getBorn(&george));
    printf("%d\n", Person_getBorn(&john));
    printf("%d\n", Person_getBorn((Person *)&peter));
    printf("%d\n", Person_millenium((Person *)&peter));
    printf("%d\n", Employee_getDepartment(&peter));
    Employee_setDepartment(&peter, 2);
    printf("%d\n", Employee_getDepartment(&peter));
    printf("%d\n", Person_getPid(&george));
    printf("%d\n", Employee_getPid(&peter));
    StupidPrint stupid;
    StupidPrint_init(&stupid, &peter);
    return 0;
}
```

IMPLEMENTATION OF THE COMPILER

Σε αυτό το κεφάλαιο θα αναλύσουμε την χρήση και την υλοποίηση της γραμματικής της γλώσσας με το εργαλείο ANTLR.

Το ANTLR είναι ένα εργαλείο που προσφέρει αυτοματοποιημένη λεκτική και συντακτική ανάλυση του κώδικα, του οποίου η μόνη απαίτηση είναι η γραμματική της γλώσσας.

Η γραμματική αποτελείται από πολλούς κανόνες συνδεδεμένους μεταξύ τους, ξεκινώντας από τον κανόνα prog, του οποίου η λειτουργία είναι να καλεί τον επόμενο κανόνα.

prog

: classes;

Ο κανόνας classes χρησιμοποιείται στην αρχή του προγράμματος, καλώντας τον κανόνα class τουλάχιστον μία φορά, όπως απαιτείται και από την γλώσσα, η ύπαρξη δηλαδή τουλάχιστον μία κλάσης στον κώδικα. Το σύμβολο “+” υποχρεώνει τον κανόνα class να λειτουργήσει τουλάχιστον μία φορά. Ο κανόνας classes τελειώνει με κάλεσμα του κανόνα main, στο τέλος του προγράμματος.

classes

:class+ main;

Ο κανόνας class καλείται για κάθε κλάση του προγράμματος python. Στην πρώτη περίπτωση αναγνωρίζει τις κλάσεις χωρίς κληρονομικότητα. Αποδέχεται την λέξη κλειδί *class* και μετά το όνομα της κλάσης ως *ID* ακολουθούμενο από άνω και κάτω τελεία. Μετά βλέπει initFunction για το def __init__ και στην συνέχεια όλες τις υπόλοιπες συναρτήσεις.

Στην δεύτερη περίπτωση, αναγνωρίζει τις κλάσεις με κληρονομικότητα, για αυτό μετά το πρώτο ID, (το όνομα της κλάσης), βλέπει και “(ID)”, για το όνομα της κλάσης που κληρονομείται.

class

: ‘class’ ID ‘:’ initFunction functions

| ‘class’ ID ‘(‘ ID ‘)’ ‘:’ initFunction functions

;

Ο κανόνας main καλείται στο τέλος του προγράμματος python, μετά από όλες τις κλάσεις. Η ύπαρξη του είναι απαραίτητη από την simplepp. Αναγνωρίζει την συνθήκη της main κλάσης και συνέχεια βλέπει όλα τα statements μέσα της.

main

: 'if' '__name__' '==' '\'__main__\'' ':'

Statements

;

Ο κανόνας *initFunction* καλείται για την πρώτη συνάρτηση κάθε κλάσης, που πάντα πρέπει να είναι η init. Αποδέχεται τις λέξεις κλειδί *def και __init__* ακολουθούμενο από παρενθέσεις και το formalparlist. Στην συνέχεια αναγνωρίζει τα statements της συνάρτησης.

initFunction

: 'def' '__init__' '(' formalparlist ')' ':'

statements

;

Ο κανόνας *functions* καλείται για κάθε κλάση, μετά τον κανόνα για την init function. Σκοπός της είναι να καλεί τον κανόνα function, καμία, μία ή περισσότερες φορές

functions

:function (functions)*

|

;

Ο κανόνας *function* καλείται για κάθε κλάση και αποδέχεται μία συνάρτηση της κλάσης. Αναγνωρίζει την λέξη κλειδί *def,* ακολουθούμενη από το *ID* που είναι το όνομα της συνάρτησης. Στην συνέχεια αποδέχεται τις παρενθέσεις και καλεί τον κανόνα formalparlist για να διαχειριστεί τις παραμέτρους της συνάρτησης. Μετά την άνω και κάτω τελεία, καλεί τον κανόνα statements ή απλά αναγνωρίζει την λέξη *pass,* όταν δεν υπάρχει κανένα statements στην συνάρτηση.

function

: 'def' ID '(' formalparlist ')' ':' (statements | 'pass')

;

Ο κανόνας *statements* καλείται κάθε φορά που χρειάζονται ένα ή περισσότερα statements. Με την σειρά του, καλεί τον κανόνα *statement.*

statements

: (statement) +

|

;

Ο κανόνας *statement* καλεί οποιονδήποτε από τους κανόνες τύπου "statement" ή αποδέχεται την λέξη *pass,* στην περίπτωση που δεν υπάρχει κάποιο statement.

statement

: assignmentStat

: ifStat

: whileStat

: printStat

: returnStat

: callStat

: 'pass'

;

Ο κανόνας *assignmentStat* καλείται όποτε χρειάζεται να γίνει κάποια ανάθεση στην Python. Για παράδειγμα στην περίπτωση: self.pid = pid. Αναγνωρίζει πρώτα ένα ID ή ένα obj (θα εξηγηθεί στην συνέχεια, αλλά το obj είναι της μορφής ID.ID, όπως στο παράδειγμα self.pid). Στην συνέχεια αποδέχεται το σύμβολο *'='* ακολουθούμενο από τους κανόνες *callstat* ή *expression (*οι οποίοι επίσης θα εξηγηθούν στην συνέχεια).

assignmentStat

: (ID | obj) '=' (callstat | expression)

;

Ο κανόνας *ifStat* καλείται όποτε χρειάζεται να γίνει κάποιος έλεγχος της μορφής if. Αναγνωρίζει την λέξη κλειδί *if,* ακολουθούμενη από τον κανόνα condition, το σύμβολο ‘:’ και τέλος τον κανόνα *statements* που εξηγήθηκε παραπάνω. Ο κανόνας *elsepart* καλύπτει την περίπτωση που στο if της Python υπάρχει else.

Η δεύτερη δυνατότητα του κανόνα *ifStat* είναι να αποδέχεται την περίπτωση όπου το condition της if βρίσκεται μέσα σε παρενθέσεις, καθώς η python επιτρέπει και τις δύο εκδοχές (στην περίπτωση που το condition αποτελείται μόνο από έναν παράγοντα, χωρίς κόμμα).

ifStat

: ‘if’ condition ‘:’ statements elsepart

| ‘if’ ‘(‘ condition ‘)’ ‘:’ statements elsepart

;

Ο κανόνας *whileStat* λειτουργεί όπως και ο *ifStat,* με την διαφορά ότι δεν υπάρχει η δυνατότητα του else.

whileStat

: ‘while’ condition ‘:’ statements

| ‘while’ ‘(‘ condition ‘)’ ‘:’ statements

;

Ο κανόνας *printStat* καλείται όποτε χρειάζεται να γίνει κάποιο return στην Python. Αποδέχεται την λέξη κλειδί *return* ακολουθούμενη από τον κανόνα *expression.*

printStat

: ‘return’ expression

| ‘return’ ‘(‘ expression ‘)’

;

Ο κανόνας *callStat* καλείται όποτε καλείται κάποια συνάρτηση στην Python. Αποδέχεται ένα obj ή ένα ID και στην συνέχεια, μέσα σε παρενθέσεις, καλεί τον κανόνα *actualparlist.*

TODO το obj χρειάζεται εδώ???

callStat

: (obj | ID) '(' actualparlist ')'

;

Ο κανόνας *elsepart* καλείται μέσα από τον κανόνα *ifStat,* εάν υπάρχει else στην python. Αποδέχεται την λέξη κλειδί *else,* και καλεί τον κανόνα *statements* μετά την άνω και κάτω τελεία.

elsepart

: 'else' ':' statements

;

Σκοπός του formalparlist είναι να καλείτε για έλεγχο παραμέτρων. Ο formalparlist καλεί τον formalparitem τουλάχιστον μία φορά. Ελέγχει αν μετά από αυτόν υπάρχει κόμμα και αν ναι ξανά καλεί τον formalparitem. Αυτό μπορεί να συμβεί για μια, καμία ή και περισσότερες φορές.

formalparlist

:formalparitem (', 'formalparaitem)*

;

Η δουλειά του formalparitem είναι να ελέγχει αν έχουμε μεταβλητή ή αντικείμενο formalparitem. Καλεί τον ID ή τον obj.

formalparitem

: ID

|obj

;

Η actualparlist σκοπό έχει τον έλεγχο ύπαρξης ορισμάτων, για την δουλεία αυτή καλή την actualparitem τουλάχιστον μία φορά και στην συνέχεια ακολουθεί το κόμμα και actualparitem για την ύπαρξη περισσότερων από ένα ορίσματα. Η διαδικασία των περισσότερων από ένα ορίσματα επιτυγχάνετε με το αστεράκι. Την διαδικασία όπου μπορεί να μην υπάρξουν ορίσματα την έχουμε με τον δεύτερο κανόνα της actualparlist |(pipe) .

actualparlist

:actualparitem (***',' actualparitem)****

|

;

Ο κανόνας actualparitem ελέγχει αν έχουμε έκφραση για όρισμα ή αντικείμενο ή μεταβλητή.

actualparitem

:expression

|obj

|ID

;

Ο κανόνας condition σκοπό έχει τον έλεγχο συνθήκης γι' αυτό καλεί τον κανόνα boolterm και ακολουθεί ο λογικός τελεστής or με τον κανόνα boolterm για καθόλου μια φορά ή και περισσότερες εμφανίσεις.

condition

:boolterm

('or' boolterm)*

;

Ο κανόνας boolterm σκοπό έχει τον έλεγχο έκφρασης με λογικό τελεστή and. Ο boolterm καλεί boolfactor όπου θα εκτελεστεί μία φορά, μετά and και τον κανόνα boolfactor όπου μπορουν να εμφανιστούν καθόλου μια φορά η και περισσότερο

boolterm

:boolfactor

('and' boolfactor)*

;

Ο κανόνας boolfactor μπορεί να αποδεχτεί τρεις εκδοχές. Πρώτη την ύπαρξη άρνηση ‘not’ με αναγνώριση ανοίγματος παρένθεσης και κάλεσμα condition και μετά κλείσιμο παρένθεσης, ως δεύτερη λειτουργία έχει μόνο την ύπαρξη συνθήκης όπου για να γίνει αποδεκτή πρέπει να έχει άνοιγμα παρένθεσης condition και κλείσιμο παρένθεσης. Τέλος, ο boolfactor καλεί expression(για έλεγχο έκφρασης) Rel_OP(για έλεγχο ύπαρξης συγκριτικών τελεστών)και ξανα expression.

boolfactor

: 'not' '(' condition ')'

| '(' condition ')'

| expression REL_OP expression

;

Ο κανόνας obj χρησιμοποιείται για έλεγχο αντικειμένου, ο οποίος αποδέχεται δυο εκδοχές. Έχουμε την περίπτωση όπου το αντικείμενο θα αποτελείται από δύο ID με την τελεία να τα διαχωρίζει. Η δεύτερη εκδοχή καλεί ID μετα τελεία και callStat(για τη περίπτωση όπου έχουμε αντικείμενο με κάλεσμα συνάρτησης).

obj

:ID '.'ID

|ID '.' callStat

;

Ο κανόνας expression σκοπό έχει τον έλεγχο ορθότητας και ύπαρξης μίας έκφρασης. Για την λειτουργία αυτή καλεί την optionalSign μετα term μία φορά και στην συνέχει αφού έχει περάσει από την term ελέγχει για την ύπαρξη ADD_OP term από καθόλου μια ή περισσότερες φορές

expression

:optionalSign term (ADD_OP term)*

;

Ο term ελέγχει την ύπαρξη ενός όρου και αν υπάρχει τότε για να είναι ορθή καλεί μία φορά factor και μετά από καθόλου μία και περισσότερο MUL_OP factor .

```
term
        :factor (MUL_OP factor)*
;
```

Ο κανόνας factor σκοπό έχει τον έλεγχο ύπαρξης ενός συντελεστή ο οποίος μπορεί να είναι ακέραιος ή μεταβλητή ή αντικείμενο ή έκφραση.

```
factor
        :INT
        |ID
        |obj
        |'('expression')'
;
```

Ο κανόνας optionalSign ελέγχει την ύπαρξη add operator, γι' αυτό καλεί μια φορά ADD_OP ή καθόλου.

```
optionalSign
        :ADD_OP
        |
;
```

Ακολουθεί ο λεκτικός αναλυτής, τον οποίο υλοποιήσαμε μέσα στο ίδιο αρχείο ANTLR.

Ο κανόνας *ID* ακολουθεί τους κανόνες της python και επιτρέπει στα ονόματα μεταβλητών, συναρτήσεων και κλάσεων να ξεκινούν από μικρό ή κεφαλαία λατινικό γράμμα,

συμπεριλαμβανομένης και της κάτω παύλας. Στην συνέχεια, το όνομα μπορεί να έχει οσαδήποτε γράμματα, κάτω παύλες ή και αριθμούς.

ID: [a-zA-Z_]+ [a-ZA-Z0-9_]* ;

Ο κανόνας *INT* χρησιμοποιείται για να αποδεχτεί integers ή floats της python.

INT: [0-9]+ ([0-9]+)?;

Ο κανόνας *REL_OP* αναγνωρίζει τους τελεστές σύγκρισης της python.

REL_OP: '=='|'<='|'>='|'!='|'<'|'>';

Ο κανόνας *ADD_OP* αναγνωρίζει τα σύμβολα της πρόσθεσης και της αφαίρεσης της python.

ADD_OP: '+'|'-';

Ο κανόνας *MUL_OP* αναγνωρίζει τα σύμβολα του πολλαπλασιασμού και της διαίρεσης της python.

MUL_OP: '*'|'/';

Ο κανόνας *BLOCK_COMMENT* αναγνωρίζει την ύπαρξη block comment της python, είτε στην μορφή «’’’ … ’’’», είτε στην «”””…”””». Ότι υπάρχει μέσα στο block comment, στέλνεται σε ένα hidden channel και δεν χρησιμοποιείται ως token από τον parser.

BLOCK_COMMENT : ('\'\'\'' .*? '\'\'\'' | '"""' .*? '"""') ->channel(HIDDEN);

Ο κανόνας *COMMENT* αναγνωρίζει την ύπαρξη σχολίου της python, όταν βλέπει το σύμβολο ‘#’. Ό,τι περιέχεται μετά από αυτό το σύμβολο, στην ίδια γραμμή, στέλνεται σε ένα hidden channel και δεν χρησιμοποιείται ως token από τον parser.

COMMENT : '#' ~[\r\n]* -> channel(HIDDEN);

Ο κανόνας *WS* αναγνωρίζει τους λευκούς χαρακτήρες (κενό, carrier return, new line και tab), και τους προσπερνά, χωρίς να παράγει token για τον parser.

WS: [\r\n\t]+ -> skip;

DESIGN OF THE COMPILER IN ANTLR

INFORMATION EXTRACTION AND STORAGE FROM PYTHON CODE

Τόσο για τον συντακτικό έλεγχο όσο και για την παραγωγή του τελικό κώδικα σε C, χρειάζεται η εξαγωγή κάποιων πληροφοριών από το αρχικό πρόγραμμα σε python και η αποθήκευσή τους σε κάποιες δομές δεδομένων για μετέπειτα χρήση. Συγκεκριμένα:

Κλάσεις:

Η αποθήκευση των κλάσεων του αρχικού προγράμματος είναι απαραίτητη. Χρησιμοποιείται ένα hashmap (classesAndFunctions), με κλειδιά τα ονόματα των κλάσεων και τιμή μία λίστα από strings που περιέχει τις συναρτήσεις της κάθε κλάσης. Το hashmap λαμβάνει τιμές μέσα στον κανόνα *function,* αφού διαβαστεί το όνομα του.

Επίσης, για τις κλάσεις διατηρείται και μία λίστα (arrayList) με όνομα classesList. Σκοπός αυτής της λίστας είναι η πρόβλεψη κλάσεων με το ίδιο όνομα. Παίρνει τιμές στην αρχή του κανόνα *class,* αφού πρώτα ελεγχθεί πως το όνομα είναι μοναδικό στην λίστα.

Σημείωση: Ίσως να είναι εφικτή η παράλειψη αυτής της λίστας, αφού υπάρχει διαθέσιμο το hashmap που περιέχει τα ονόματα των κλάσεων. Καθώς όμως αυτό το hashmap παίρνει τιμές σε κανόνα μετέπειτα του *class,* είναι πιο εύκολη η χρήση μία λίστας αφιερωμένη μόνο στον έλεγχο των ονομάτων.

Συναρτήσεις:

Όπως και οι κλάσεις, έτσι και οι συναρτήσεις είναι απαραίτητο να διατηρηθούν. Συγκεκριμένα, στο ίδιο hashmap με όνομα classesAndFunctions αποθηκεύονται και οι συναρτήσεις, αντιστοιχισμένες στην κατάλληλη κλάση τους.

Παράμετροι συναρτήσεων:

Οι παράμετροι είναι απαραίτητο να αποθηκεύονται για τον ίδιο πάλι λόγο, την πρόβλεψη παραμέτρων με το ίδιο όνομα. Η συγκεκριμένη περίπτωση όμως καταλήγει να είναι αρκετά περίπλοκη στην υλοποίηση, για λόγους που αναλύονται στην παράγραφο *Implementation Difficulties With ANTLR Actions.*

Η λύση βασίζεται στον σχεδιασμό του πίνακα συμβόλων (λεπτομέρειες στην παράγραφο Symbol Table). Η κλάση *Functions* περιέχει ως πεδίο μία λίστα *formal_par_list* η οποία λαμβάνει τιμές μέσα από την συνάρτηση *add_parameter*. Η συνάρτηση αυτή καλείται στον κανόνα *formalparitem.* Έτσι, συγκεντρώνεται μία λίστα από αντικείμενα τύπου *Function* (οι συναρτήσεις), τα οποία περιέχουν λίστες από *formalparitem* (παραμέτρους).

Εκτός από αυτές τις τρεις βασικές δομές, εξάγονται κάποιες άλλες πληροφορίες από τον αρχικό κώδικα σε python, που αποθηκεύονται για μετέπειτα χρήση.

Συγκεκριμένα:

inheritedClass: Μεταβλητή που αποθηκεύει το όνομα της κλάσης που κληρονομείται

classNameForDowncasting: Σε περίπτωση κληρονομικότητας, μπορεί να χρειαστεί downcasting στο κάλεσμα συναρτήσεων (παράγραφος TODO)

classesStructsAndLinesMap: Το hashmap περιέχει τα ονόματα των κλάσεων ως κλειδιά, και λίστες από τις γραμμές που ξεκινούν και τελειώνουν τα struct της κάθε κλάσης, για τιμές.

classesAndFieldsMap: Το hashmap περιέχει τα ονόματα των κλάσεων ως κλειδιά και λίστες από τα όλα τα πεδία κάθε κλάσης, ως τιμές.

classesAndObjectsMap: Το hashmap περιέχει τα ονόματα των κλάσεων ως κλειδιά και λίστες από όλα τα αντικείμενα τύπου ίδιου με την κλάση, ως τιμές.

childAndParentMap: Το hashmap περιέχει τα ονόματα των κλάσεων που κληρονομούν άλλη κλάση ως κλειδιά, και τα ονόματα των κλάσεων που κληρονομούνται, ως τιμές.

FINAL CODE GENERATION

Η γενική λογική εξαγωγής του τελικού κώδικα είναι η εισαγωγή των δεδομένων που παίρνουμε από τους κανόνες της γραμματικής. Η εισαγωγή τους γίνετε σε προσωρινά αρχεία ώστε να έχουμε την δυνατότητα αλλαγής σε προηγούμενα σημεία τα οποία άλλαξαν κατά την διάρκεια εκτέλεσης των κανόνων της γραμματικής, η συνεχής ένωσή τους θα γίνετε μέχρι να παραχθεί το τελικό αρχείο. Η εξαγωγή του τελικού κώδικα σε γλώσσα C έγινε με την βοήθεια της κλάσης WriteToFile, των actions που χρησιμοποιήσαμε μέσα στους κανόνες αλλά και των τριών κύριων μεθόδων που φτιάξαμε στο members.

Η κλάση WriteToFile χρησιμοποιήθηκε για την βελτιστοποίηση του κώδικα ώστε να μην υπάρχει επανάληψη δηλαδή ύπαρξη κλώνων. Η κλάση μας αποτελείται από τις εξής μεθόδους: openFile, closeFile, merge, writeFile, seekInFile, deleteFile.

openFile μέθοδος η οποία παίρνει δυο παραμέτρους το όνομα αρχείου που θέλουμε να δημιουργήσουμε και μια flag, ώστε να αναγνωρίζει αν θα δημιουργήσει από την αρχή το αρχείο ή αν θα κρατήσει το περιεχόμενο.

closeFile μέθοδος η οποία κλείνει το αρχείο.

merge μέθοδος η οποία φτιάχτηκε για την ένωση δύο αρχείων του αρχικού και του προσωρινού ώστε να δημιουργηθεί το τελικό. Η μέθοδος αυτή παίρνει ως παράμετρο το όνομα του αρχείου που θέλουμε να ενσωματώσουμε με το αρχικό το διαβάζει γραμμή γραμμή και το αντιγράφει στο αρχικό.

Η writeFile είναι μέθοδος η οποία παίρνει μια παράμετρο String το οποίο το γράφει στο αρχείο που της έχει δοθεί από το private πεδίο myWriter.

seekInFile μέθοδος η οποία δέχεται δυο παραμέτρους length και ένα String. Η δουλεία της είναι να ψάχνει μέσα στο αρχείο σύμφωνα με το μήκος που της έχει δοθεί για να εισάγει στην κατάλληλη θέση το String.

deleteFile δημιουργήσαμε αυτή την μέθοδο ώστε να διαγράφει το προσωρινό αρχείο για να μην υπάρχει στο τέλος.

Οι τρείς μέθοδοι που χρησιμοποιήσαμε για την υλοποίηση του τελικού αρχείου είναι ReturnTotalNumberOfTabs, objectParam και η outputPrintf.

Επειδή, η γλώσσα simple python που έχουμε ως αρχείο εισόδου αναγνωρίζει ένα μπλοκ με τον ίδιο τρόπο όπως και η python (εσοχές) φτιάξαμε την μέθοδο ReturnTotalNumberOfTabs. Η οποία παίρνει μια παράμετρο με το όνομα line και στην συνέχεια μετράει και επιστρέφουμε τον αριθμό tabs που έχουν δοθεί σε αυτή. Η λογική της μεθόδου αυτής είναι να της δίνετε η γραμμή που θέλουμε να επιστρέψει τον αριθμό των εσοχών, να συνεχίσει το διάβασμα όσο βρίσκει κενές γραμμές για να τις προσπεράσει καθώς δεν είναι απαραίτητες για το πρόγραμμα αλλιώς -1. Τέλος, η μέθοδος αυτή ελέγχει αν βρει στην γραμμή την λέξη κλειδί class τότε επιστρέφει 0 καθώς θα πρέπει να υπάρχουν μηδέν εσοχές αφού είναι η αρχή του προγράμματος.

Η δεύτερη και σημαντικότερη μέθοδος είναι η objectParam η οποία έχει τρεις βασικές λειτουργίες εύρεση αντικειμένων, εύρεση θέσης αντικειμένου και εισαγωγή του ονόματος του στο οποίο δείχνει το αντικείμενο. Η λογική για την δημιουργία αυτής της μεθόδου ήταν ότι θα καλείται μέσα στην actualparlist για κάθε φορά που της δίνεται ένα όρισμα. Καθώς είναι η μόνη περίπτωση που μπορεί να υπάρξει γνώση για το τι τύπου θα είναι οι παράμετροι και μέσα από αυτές και οι μεταβλητές μας.

Η objectParam παίρνει ως παράμετρο ένα String paritem με το οποίο ελέγχουμε αν είναι αριθμός ή όχι, αν ναι τότε αφήνουμε το αρχείο όπως έχει και προχωράμε στο επόμενο όρισμα που μας δίνεται από την actualparitem. Εάν το string δεν είναι αριθμός τότε καταλαβαίνει ότι το token που έχει διαβάσει μάλλον είναι αντικείμενο, έτσι προχωράει στον επόμενο έλεγχο όπου βλέπει αν ανήκει σε κάποιο σύνολο μέσα στο hashmap «objectPointsClassNameMap» που περιέχει τα αντικείμενα και το όνομα της κλάσεις τους. Τότε αν όντως είναι αντικείμενο(έχουμε τελειώσει με την εύρεση αντικειμένου) προχωράει και διαβάζει το αρχικό αρχείο γραμμή γραμμή και όταν βρεθεί στο κατάλληλο struct(όπου η εύρεση της θέσης του γίνεται με την ισότητα currentLineNumber ==lineOfClassStruct.get(checkIdForParmObj).get(0)) μπαίνει μέσα και αντικαθιστά την λέξη int με το κατάλληλο όνομα κλάσεις που μας έδειχνε το αντικείμενο προηγουμένως(για να ξέρει ποιο int πρέπει να αντικαταστήσει προχωράει επαναληπτικά την γραμμή μέχρι να φτάσει το lineNumberOfstrucktParam-1). Στην συνέχεια αφού έχουν γίνει οι κατάλληλες αλλαγές μέσα στο struct ο βρόγχος προχωράει στον έλεγχο για την αλλαγή αν χρειάζεται στις παραμέτρους της κάθε συνάρτησης του συγκεκριμένου struct. Η διαδικασία αυτή γίνεται με δυο διαφορετικούς ελέγχου για την περίπτωση των init συναρτήσεων και των απλών. Επειδή διαφοροποιούνται οι συναρτήσεις στις παραμέτρους και δεν μπορεί να χρησιμοποιήσει τον μετρητή lineNumberOfstrucktParam. Η περίπτωση των διαφορετικών παραμέτρων έφερε ως θέμα την δημιουργία μιας νέας μεθόδου την placeObject, με την οποία κάνουμε την εισαγωγή του ονόματος αλλά και την δήλωση των αντικειμένων μέσα στην συνάρτηση όπως θέλει γλώσσα C.

Τέλος, η objectParam ελέγχει αν υπάρχει κληρονομικότητα για να τοποθετήσει την λέξη base στα σημεία που χρειάζεται αυτό το επιτυγχάνει μέσα από δύο βρόγχους όπου κοιτάζει αν η μεταβλητή εμπεριέχεται μέσα στο struct της συνάρτησης του και αν ναι τότε η μεταβλητή είναι τοπική αλλιώς προχωράει στο struct που του υποδεικνύει η μεταβλητη base και ελέγχει εκεί τις μεταβλητές μια μια και αν υπάρχει τότε εισάγει base.

*Η μέθοδος placeObject παίρνει ως παραμέτρους line, String str , String[] templine,με τις οποίες φτιάχνει και εισάγει τις δηλώσεις μέσα στις συναρτήσεις .

* currentLineNumber τοπικός μετρητής μέσα στην while

* checkIdForParmObj δημόσια μεταβλητή όπου κρατάει το όνομα του τρέχον struct

* lineOfClassStruct δημόσιο hash map το οποίο περιέχει το όνομα του στρακτ και το ζεύγος <αριθμός αρχής struct, αριθμός τέλους struct>

* lineNumberOfstrucktParam δημόσια μεταβλητή που κρατάει τον αριθμό ορισμάτων που έχει δεχτεί

Όλες οι μέθοδοι της WriteToFile βγάζουν το κατάλληλο διαγνωστικό μήνυμα σε περίπτωση exception.

Κώδικας σε actions

Συνεχίζοντας, αναλύεται η παραγωγή του τελικού κώδικα για περιπτώσεις ιδιαίτερης δυσκολίας

Tabs

Η έλεγχος της σωστής στοίχισης του κώδικα ήτανε ένα πρόβλημα εβδομάδων καθώς κάθε λύση που δοκιμάστηκε κατέληγε να αποτυγχάνει σε διαφορετικό σενάριο.

Αναφορικά, οι κύριες δυσκολίες ήτανε η αύξηση και μείωση των tabs στην είσοδο και στην έξοδο των κλάσεων, συναρτήσεων, καθώς και των if και while, η διαχείριση των tabs στα εμφωλευμένα if/while και η διατήρηση των tabs για κάθε statement μέσα σε συνάρτηση και κλάση.

Η λύση που εφαρμόστηκε είναι η χρήση ενός counter για τα tabs, ο οποίος αυξομειώνεται στα κατάλληλα σημεία για να ελέγχει την στοίχιση του αρχείου simplepp.

Το αρνητικό αυτής της υλοποίησης είναι πως τα tabs του αρχείου C δεν είναι πάντα τα πιο επιθυμητά, χωρίς φυσικά να επηρεάζουν την εκτέλεση.

Structs, Αντικειμενοστρέφεια και Προσωρινό αρχείο C

Ένα από τα μεγαλύτερα προβλήματα μεταγλώτισης κώδικα από simplepp (python) σε C οφείλεται στο ότι η Python είναι μία αντικειμενοστραφής γλώσσα προγραμματισμού, ενώ η C όχι.

Αυτό σημαίνει πως οι κλάσεις της python θα πρέπει να μεταφραστούν σε structs της C.

Επίσης, οι δηλώσεις μεταβλητών στην python δεν είναι απαραίτητες, ενώ στην C χρειάζονται μέχρι και οι τύποι των μεταβλητών.

Με άλλα λόγια, πρέπει να παραχθεί κώδικας σε C από ανύπαρκτο κώδικα python.

Συγκεκριμένα για τις κλάσεις και τα structs:

Για κάθε κλάση της simplepp χρειάζεται να γραφτεί στο αρχείο C ‘’typedef struct’’

Στην συνέχεια πρέπει να δηλωθούν μέσα στο struct όλες οι μεταβλητές της κλάσης. Εδώ χρησιμοποιείται ο πίνακας συμβόλων για να παρθούν οι παράμετροι της initFunction από το entities list του scope.

Ένα ακόμη πρόβλημα παρουσιάζεται στην περίπτωση που υπάρχει κληρονομικότητα, καθώς επηρεάζει το struct. Αυτό θα αναλυθεί ακριβώς παρακάτω.

Το κύριο πρόβλημα όμως είναι κρυμμένο πίσω από το γεγονός πως ο τύπος των μεταβλητών στο simplepp αρχείο είναι άγνωστος. Μπορεί η simplepp να δέχεται μόνο integer μεταβλητές, αυτό όμως δεν περιορίζει τις παραμέτρους των συναρτήσεων να είναι αντικείμενα άλλων κλάσεων.

Το μόνο σημείο στο οποίο γίνεται γνωστός ο τύπος της παραμέτρου, είναι στην main συνάρτηση, στο τέλος του simplepp κώδικα. Όταν όμως η γραμματική φτάσει σε αυτό το σημείο, ο κώδικας της C θα έχει ήδη παραχθεί.

Η λύση σε αυτό το πρόβλημα είναι η χρήση ενός προσωρινού αρχείου C, στο οποίο κατά το διάβασμα της main συνάρτησης του simplepp, διατρέχεται ολόκληρο και διορθώνεται με βάση τις πληροφορίες που εξάγονται από την main.

Inheritance

Η κληρονομικότητα δημιουργεί αρκετά προβλήματα στην μετάφραση από simplepp σε C. Στην simplepp, το μόνο που απαιτείται είναι η δήλωση της κλάσης που κληρονομείται, σε παρενθέσεις, μετά το όνομα την κλάσης που κληρονομεί.

Στην C τα πράγματα είναι πιο περίπλοκα. Αρχικά χρειάζεται να δηλωθεί η κλάση που κληρονομείται στο struct της κλάσης που κληρονομεί. Αν για παράδειγμα η κλάση Employee κληρονομεί την κλάση Person, τότε στο struct της Employee χρειάζεται να προστεθεί η ακόλουθη εντολή: ‘’Person base;’’.

Επίσης, η δήλωση των μεταβλητών της Person που κληρονομείται δεν χρειάζεται στο struct της Employee, αφού βρίσκεται στο struct της Person. Χρειάζεται όμως η δήλωση του constructor της Person στον constructor της Employee.

Δηλαδή, αν το simplepp αρχείο είναι το εξής:

class Employee(Person):

```
    def __init__(self, pid, born, afm, department):
    self.pid = pid
    self.born = born
    self.afm = afm
    self.department = department
```

Τότε το C αρχείο θα πρέπει να είναι το εξής:

```
typedef struct{
    Person base;
    int afm;
    int department;
}Employee;
void Employee_init (Employee *self,int pid,int born,int afm,int department) {
    Person_init((Person*)self, pid, born);
    self->afm = afm;
    self->department = department;
```

Το πιο δύσκολο κομμάτι βρίσκεται στην κλήση των μεταβλητών αντικειμένου κλάσης που περνάει ως παράμετρο στην συνάρτηση άλλης κλάσης.

Δηλαδή στο παράδειγμα η παράμετρος employee:

```
class StupidPrint:
    def __init__(self, employee):
        print(employee.pid)
```

Αυτό στην C μεταφράζεται ως:

```
typedef struct{
    Employee *employee;
}StupidPrint;
void StupidPrint_init (StupidPrint *self, Employee *employee) {
```

```
self->employee = employee;
printf ("%d \n",self->employee->base.pid);
```

Οι παράμετροι της συνάρτησης πρέπει να περαστούν με pointer στις διευθύνσεις μνήμης τους, ενώ κατά το κάλεσμά τους χρησιμοποιείται το σύμβολο ‘’->’’, για την διαχείριση των pointers.

Επίσης, επειδή η παράμετρος Employee είναι αντικείμενο άλλης κλάσης, η κλήση μίας μεταβλητής του χρειάζεται να χρησιμοποιήσει την λέξη ‘’base’’, καθώς έτσι έχει δηλωθεί στο struct.

Τέλος, όλες οι παράμετροι και οι δηλώσεις μεταβλητών στα struct δεν έχουν τον σωστό τύπο, όπως προαναφέρθηκε. Αντίθετα, ο μεταφραστής περιμένει να φτάσει η ανάγνωση του αρχικού κώδικα σε simplepp στην main, και τότε χρησιμοποιεί το προσωρινό αρχείο C, για να κάνει τις αλλαγές που απαιτούνται (πχ για φτιάξει τη παράμετρο *employee της StupidPrint_init σε τύπου Employee, καθώς κατά την ανάγνωση του .py αρχείου, η παράμετρος έχει απλά το όνομα ‘’employee’’, χωρίς να φαίνεται τι τύπου είναι.)

Main και Downcasting

Στην main, η simplepp έχει την δυνατότητα να δημιουργήσει ένα αντικείμενο μίας κλάσης και να περάσει παραμέτρους με μία εντολή.

Δηλαδή: george = Person(200223, 2002)

Αυτό στην C χρειάζεται να μεταφραστεί σε δύο εντολές, η πρώτη που δηλώνει την μεταβλητή george ως αντικείμενο τύπου Person, και η δεύτερη που καλεί τον constructor περνώντας τιμές.

```
Person  george;
Person_init(&george, 200223,2002);
```

Παρατήρηση: Το αντικείμενο george, όπως όλα τα αντικείμενα, χρειάζεται να περαστεί με αναφορά στην διεύθυνση μνήμης, για αυτό χρησιμοποιείται το σύμβολο ‘’&’’.

Το μεγαλύτερο θέμα με την main είναι η ανάγκη για downcasting.

Η simplepp έχει την δυνατότητα να καλεί άμεσα την συνάρτηση μία κλάσης που κληρονομείται, μέσω ενός αντικειμένου που την κληρονομεί.

Δηλαδή:

```
peter = Employee(200122, 2001, 990122, 1)
print(peter.getBorn())
```

Το αντικείμενο peter είναι τύπου Employee, αλλά η κλάση Employee δεν περιέχει την συνάρτηση getBorn()

Για να πραγματοποιηθεί κάτι τέτοιο στην C, χρειάζεται το αντικείμενο peter να γίνει downcasted σε Employee, μέσω της κληρονομικότητας της κλάσης Employee (Person).

Στην C, ο κώδικας θα μοιάζει ως εξής:

```
printf("%d\n", Person_getBorn((Person *)&peter));
```

Με το downcasting να γίνεται ως (Person *) &peter

Όλη αυτή η διαδικασία πραγματοποιείται στον κανόνα *callStat*, με την χρήση μίας Boolean μεταβλητής classNameForDowncasting, η οποία αποθηκεύει το όνομα της κλάσης στην οποία θα γίνει το downcasting.

Print και %d

Η εντολή print της python δουλεύει χωρίς την ανάγκη δήλωσης του τύπου των μεταβλητών. Η C από την άλλη, χρειάζεται το πρόθεμα ‘’%d’’, για να αναγνωρίσει πως οι μεταβλητές που θα τυπώσει είναι integers.

Αυτή η διαδικασία πραγματοποιείται στον κανόνα ID, του κανόνα factor, που καλείται από τον term, από τον expression και τέλος από τον κανόνα printStat.

IMPLEMENTATION DIFFICULTIES WITH ANTLR ACTIONS.

Σημαντική αναφορά στον τρόπο υλοποίησης του μεταφραστή. Στις διαλέξεις του μαθήματος έγινε αναφορά στα actions της ANTLR και πως μέσα από αυτά δίνεται η δυνατότητα εκτέλεσης κώδικα (στην παρούσα φάση java).

Λόγο αυτού, πάρθηκε η απόφαση η υλοποίηση του μεταφραστή να γίνει σε ένα αρχείο γραμματικής g4 του ANTLR, με χρήση μόνο actions.

Αυτό δημιουργεί μεγάλα προβλήματα στην συγγραφή του κώδικα, ειδικά στο κομμάτι της εμβέλειας των μεταβλητών. Συγκεκριμένα, στην ANTLR, η εμβέλεια μία μεταβλητής που δηλώνεται σε ένα action, περιορίζεται μέσα στον κανόνα της γραμματικής που το περιέχει (το action).

Για παράδειγμα, η λύση του προβλήματος των παραμέτρων με το ίδιο όνομα θα ήταν πολύ απλή. Στον κανόνα *formalparlist* θα μπορούσε να αρχικοποιηθεί μία λίστα η οποία στον κανόνα *formalparitem* θα έλεγχε εάν η παράμετρος υπάρχει ήδη.

Εάν όχι, θα πρόσθετε την παράμετρο στην λίστα και θα συνέχιζε στην επόμενη.

Δυστυχώς όμως, η λίστα αρχικοποιείται στον κανόνα *formalparlist* και η εμβέλεια του δεν φτάνει μέχρι τον κανόνα *formaparitem,* παρόλο που ο δεύτερος καλείται μέσα στον πρώτο. Άρα αυτή η λύση δεν δουλεύει.

Ούτε η χρήση μίας global λίστας με τις παραμέτρους θα δούλευε καθώς δύο συναρτήσεις μπορούν να χρησιμοποιούν παραμέτρους με το ίδιο όνομα, σύμφωνα με την simplepp.

Ευτυχώς, στον πίνακα συμβόλων αποθηκεύονται οι συναρτήσεις και οι παραμέτρους τους, οπότε ο έλεγχος μπορεί να πραγματοποιηθεί από εκεί.

Αυτό το μικρό παράδειγμα δείχνει τις μεγάλες δυσκολίες που δημιουργήθηκαν σε κάθε βήμα της συγγραφής του μεταφραστή, για κάθε πρόβλημα και κάθε λειτουργία.

Επίσης, δεν γίνεται χρήση των listeners και γενικότερα του abstract syntax tree (AST), τα οποία φαίνεται να είναι και τα πιο ενδιαφέρονται εργαλεία του ANTLR.

Λόγο όλων αυτών, ο κώδικας του μεταφραστή κατέληξε να είναι δυσνόητος, δύσχρηστος και σχεδόν αδύνατον να τροποποιηθεί, αφού δεν ακολουθεί σωστές τεχνικές αντικειμενοστραφή προγραμματισμού.

Έτσι, την τελευταία εβδομάδα ξεκίνησε η εκ νέου συγγραφή του κώδικα, αυτήν την φορά με καλύτερη γνώση του ANTLR, και με χρήση listeners και αντικειμενοστρέφειας. Προφανώς, αυτός ο κώδικας δεν θα μπορέσει να παραδοθεί μέχρι το deadline, αλλά είναι ένα todo project.

Ακολουθούν μερικές εικόνες για το πως θα μοιάζει η δεύτερη έκδοση του μεταφραστή:

Main.java | CFileWriter.java | ParserRules.g4

Project > v4 - Better Implementation > src > ParserRules.g4 > ...

```
parser grammar ParserRules;

import LexerRules;

classes
    :class+ main
;

class
    : CLASS ID COLON initFunction functions
    | CLASS ID LPARA ID RPARA COLON initFunction functions
;

initFunction
    :DEF INIT LPARA formalparlist RPARA COLON
    statements
;

functions
    :function (function)*
    |
;

function
    :DEF ID LPARA formalparlist RPARA COLON
    statements
;

main
    :IF NAME REL_OP MAIN COLON
    statements
;

statements
    :statement+
;
```

Language.g4

Project > v4 - Better Implementation > src > Language.g4 > ...

```
grammar Language;

import ParserRules, LexerRules;

prog
    :classes
;
```

LexerRules.g4 | Main.java | CFileWriter.java

Project > v4 - Better Implementation > src > LexerRules.g4 > ...

```
lexer grammar LexerRules;

// Binary operations
ADD_OP            : '+'|'-';
MUL_OP            : '*'|'/';

// Boolean operations
REL_OP            : '=='|'<='|'>='|'!='|'<'|'>';

// Keywords
ELSE              : 'else';
IF                : 'if';
WHILE             : 'while';
RETURN            : 'return';
CLASS             : 'class';
DEF               : 'def';
INIT              : '__init__';
PASS              : 'pass';
PRINT             : 'print';
AND               : 'and';
NOT               : 'not';
OR                : 'or';
NAME              : '__name__';
MAIN              : '\'__main__\'';

// Flow Controls
LPARA             : '(';
RPARA             : ')';
LBRAC             : '{';
RBRAC             : '}';
COMMA             : ',';
COLON             : ':';
POINT             : '.';

// Assignment operation
AssignOP          : '=' ;
```

Main.java | MyListener.java | ProgListener.java

Project > v4 - Better Implementation > src > Main.java > Main > main(String[])

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.IOException;

public class Main {
    Run | Debug
    public static void main(String[] args) throws IOException {
        // Open and create the generated C file
        String generatedCFile = "generatedC.c";
        CFileWriter.getInstance().openFile(generatedCFile);

        // Input the Python file
        String filePath = "pythonExample1.py";
        CharStream input = CharStreams.fromFileName(filePath);

        // Instantiate the lexer
        LanguageLexer lexer = new LanguageLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        // Instantiate the parser
        LanguageParser parser = new LanguageParser(tokens);

        ParseTree tree = parser.prog(); // 'prog' is the start ru

        // Create a custom listener
        ParseTreeWalker walker = new ParseTreeWalker();
        MyListener myListener = new MyListener();

        // Walk the parse tree with the listener
        walker.walk(myListener, tree);

        // Close the file
        CFileWriter.getInstance().closeFile();
    }
}
```

Main.java | CFileWriter.java

Project > v4 - Better Implementation > src > CFileWriter.java > CFileWriter

```java
import java.io.FileWriter;
import java.io.IOException;

public class CFileWriter {
    private FileWriter fileWriter;
    private static CFileWriter instance;

    private CFileWriter() {}

    public static CFileWriter getInstance() {
        if (instance == null) {
            instance = new CFileWriter();
        }
        return instance;
    }

    public void openFile(String filePath) throws IOException {
        fileWriter = new FileWriter(filePath);
        System.out.println("\nOpening file"+filePath);
    }

    public void writeToFile(String content) throws IOException {
        fileWriter.write(content);
        System.out.println("Writing to file: "+content);
    }

    public void closeFile() throws IOException {
        if (fileWriter != null) {
            fileWriter.close();
            System.out.println(x:"Closing file");
        }
    }
}

```

MyListener.java

Project > v4 - Better Implementation > src > MyListener.java > ...

```java
public class MyListener extends LanguageBaseListener {
    private ProgListener progListener;

    public MyListener(){
        progListener = new ProgListener();
    }

    @Override
    public void enterProg(LanguageParser.ProgContext ctx) {
        progListener.enterProg(ctx);
    }
}

```

MyListener.java | ProgListener.java

Project > v4 - Better Implementation > src > ProgListener.java > ProgListener

```java

public class ProgListener extends LanguageBaseListener {
    private ClassesListener classesListener;

    public ProgListener(){
        classesListener = new ClassesListener();
    }

    @Override
    public void enterProg(LanguageParser.ProgContext ctx) {
        System.out.println(x:"Entered rule prog");
    }

    @Override
    public void enterClasses(LanguageParser.ClassesContext ctx) {
        classesListener.enterClasses(ctx);
    }
}
```

Χωρισμός των λεκτικών κανόνων από των συντακτικών. Τα g4 αρχεία περιέχουν μόνο τους κανόνες της γλώσσας και όχι κώδικα java.

Χρήση μίας main κλάσης η οποία παίρνει το αρχείο σε simplepp, το περνάει από τον lexer για να παράγει τα tokens, τα οποία τα παίρνει ο parser και φτιάχνει το δέντρο με τα nodes.

Ο walker περπατάει το δέντρο και χρησιμοποιεί τους listeners όταν εισέρχεται στον κατάλληλο κανόνα.

Οι listeners είναι συναρτήσεις που παράγει το ANTLR, οι οποίες ενεργοποιούνται όταν ο walker μπαίνει και βγαίνει από κάθε κανόνα.

Επίσης, γίνεται χρήση της κλάσης CFileWriter.java σε μορφή singleton για την εγγραφή στο τελικό αρχείο C.

SYMBOL TABLE

Έγινε μία προσπάθεια χρήσης της λογικής του πίνακα συμβόλων, όπως είχε γίνει και στο μάθημα Μεταφραστές 1. Η χρήση του κατέληξε να είναι ιδιαίτερα περιορισμένη.

Αρχικά, τι είναι ο πίνακας συμβόλων. Η ονομασία του μπορεί να προϊδεάζει την τυπική εικόνα ενός πίνακα με γραμμές και στήλες, αλλά στην παρούσα χρήση λειτουργεί περισσότερο ως stack, με nesting levels. Ο λόγος αυτής της υλοποίησης είναι για να υποστηρίξει την δυνατότητα των συναρτήσεων και κλάσεων να μπαίνουν σε nesting levels, δηλαδή η μία να καλεί την άλλη, διατηρώντας την σωστή εμβέλεια μεταβλητών.

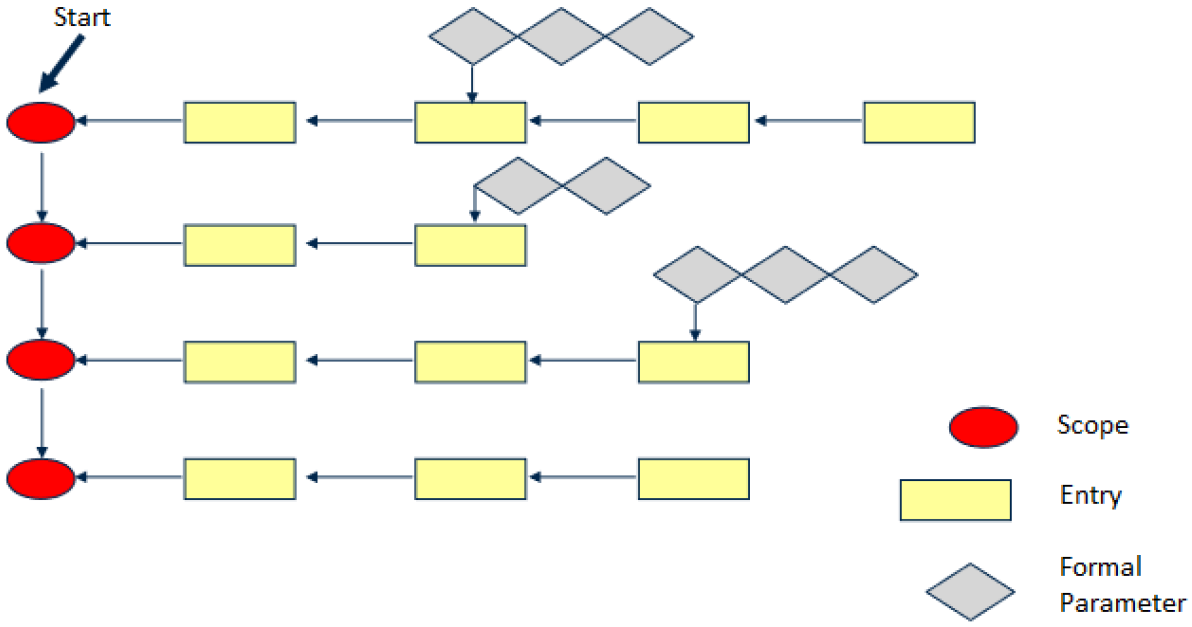
Δομή Πίνακα Συμβόλων

Ο πίνακας συμβόλων αποτελείται από scopes. Κάθε scope αντιπροσωπεύει μία κλάση ή μία συνάρτηση. Στην αρχή μίας κλάσης ή συνάρτησης, ένα καινούργιο scope προστίθεται στο stack, και όταν τελειώνουν, αφαιρείται.

Για παράδειγμα, δύο συναρτήσεις, f1 και f2, με την f2 να εκτελείται μέσα στην f1. Πρώτα το scope της f1 δημιουργείται και μέσα σε αυτό, το scope της f2.

Με αυτήν την υλοποίηση του πίνακα συμβόλων είναι εγγυημένη η σωστή εμβέλεια των μεταβλητών, σύμφωνα με τους κανόνες της simplepp.

Σχηματικό παράδειγμα:



Αδυναμίες του πίνακα συμβόλων στον παρόν μεταφραστή

Ένα αρχικό πρόβλημα που παρατηρήθηκε στην εφαρμογή του πίνακα συμβόλων ήταν τι δεδομένα θα αποθηκεύονται σε αυτό.

Η simplepp, κατ'επέκταση και η python, δεν περιέχουν δήλωση μεταβλητών όπως για παράδειγμα η C.

Δημιουργήθηκαν όμως οι εξής κλάσης οι οποίες αντιπροσωπεύουν δεδομένα

public class Scope: αντιπροσωπεύει ένα scope και περιέχει έναν integer για το nesting level και μια λίστα με τα αντικείμενα του, entities_list.

public class Entity: αντιπροσωπεύει ένα entity, δηλαδή μία μεταβλητή, μία κλάση, μία παράμετρο ή μία συνάρτηση. Περιέχει μόνο το όνομα του entity, καθώς γίνεται extend από τις προαναφερόμενες κλάσεις.

public class Variable: αντιπροσωπεύει μία μεταβλητή και περιέχει μόνο το όνομα, καθώς στην simplepp δεν δηλώνεται τύπος μεταβλητής.

public class Class: αντιπροσωπεύει μία κλάση.

public class FormalParmeter: αντιπροσωπεύει μία παράμετρο.

public class Function: αντιπροσωπεύει μία συνάρτηση και περιέχει μία λίστα από τις παραμέτρους της (formal_par_list).

Χρησιμοποιούνται επίσης και κάποιες συναρτήσεις για την διαχείριση του πίνακα συμβόλων.

Public class AddScope: περιέχει τις συναρτήσεις

add_new_scope: προσθέτει ένα scope στις κατάλληλες θέσεις, όπως προαναφέρθηκε.

remove_scope: αφαιρεί το scope όταν χρειάζεται

public class AddEntity: περιέχει τις συναρτήσεις

add_new_function: προσθέτει μία καινούργια συνάρτηση στο scope

add_parameter: προσθέτει μία παράμετρο στην λίστα με τα entities του τελευταίου scope.

add_variable: προσθέτει μία μεταβλητή στην λίστα με τα entities, του τελευταίου scope, αφού ελέγξει πρώτα εάν υπάρχει ήδη.

checkUniqueParameter: καλείται στην **add_parameter** και ελέγχει εάν η παράμετρος υπάρχει ήδη για την συγκεκριμένη συνάρτηση.

TESTING

Μερικά από τα τεστ που λειτουργούν:

Η πιο απλή μορφή δημιουργία δύο κλάσεων με απλές συναρτήσεις όπου χρησιμοποιούν τα statements print, return, assign και με μία απλή δημιουργία ενός αντικειμένου καλεί τις συναρτήσεις

Α)

```
class Person:
    def __init__(self, pid, born):
        self.pid = pid
        self.born = born
    def getPid(self):
        return self.pid
    def getBorn(self):
        return self.born
class StupidPrint:
    def __init__(self,employee):
        print(employee.pid)
        print(employee.born)
        print(employee.afm)
        print(employee.department)
if __name__ == '__main__':
    george = Person(200223, 2002)
    john = Person(200055, 2000)
    print(george.getBorn())
    print(john.getBorn())
```

Με έξοδο

```
#include <stdio.h>

typedef struct{
        int pid;
        int born;
```

```
}Person;
void Person_init (Person *self,int pid,int born) {
        self->pid = pid;
        self->born = born;
}
int  Person_getPid (Person *self) {
        return self->pid;
}
int  Person_getBorn (Person *self) {
        return self->born;
}
typedef struct{
        int employee;
}StupidPrint;
void StupidPrint_init (StupidPrint *self,int employee) {
        printf ("%d \n",self->employee->pid);
        printf ("%d \n",self->employee->born);
        printf ("%d \n",self->employee->afm);
        printf ("%d \n",self->employee->department);
}
int main(){
        Person  george;
        Person_init(&george, 200223,2002);
        Person  john;
        Person_init(&john, 200055,2000);
        printf ("%d \n",Person_getBorn(&george  ));
        printf ("%d \n",Person_getBorn(&john  ));
}
```

Β) Μια μορφή της simple python που εξετάζουμε την κληρονομικότητα

```
class Person:
   def __init__(self, pid, born):
      self.pid = pid
```

```
        self.born = born
    def getPid(self):
        return self.pid
    def getBorn(self):
        return self.born
class Employee(Person):
    def __init__(self, pid, born, afm, department):
        self.pid = pid
        self.born = born
        self.afm = afm
        self.department = department
    def getDepartment(self):
        return self.department
    def setDepartment(self, department):
        self.department = department
    def getPid(self):
        return self.afm
if __name__ == '__main__':
    george = Person(200223, 2002)
    john = Person(200055, 2000)
    peter = Employee(200122, 2001, 990122, 1)
    print(george.getBorn())
    print(john.getBorn())
    print(peter.getBorn())
    print(peter.millenium())
    print(peter.getDepartment())
    peter.setDepartment(2)
    print(peter.getDepartment())
    print(peter.getPid())
```

Γ) Εξετάζει μέσα στην main αν μπορεί να καλέσει συναρτήσει με το ίδιο όνομα που ανήκουν σε διαφορετικές κλάσεις

```
class Person:
```

```
    def __init__(self, pid, born):
        self.pid = pid
        self.born = born
    def getPid(self):
        return self.pid
    def getBorn(self):
        return self.born

class Employee(Person):
    def __init__(self, pid, born, afm, department):
        self.pid = pid
        self.born = born
        self.afm = afm
        self.department = department
    def getDepartment(self):
        return self.department
    def setDepartment(self, department):
        self.department = department
    def getPid(self):
        return self.afm
if __name__ == '__main__':
    george = Person(200223, 2002)
    peter = Employee(200122, 2001, 990122, 1)
    print(george.getPid())
    print(peter.getPid())
    peter.setDepartment(2)
```

Η έξοδός του

```
#include <stdio.h>
typedef struct{
        int pid;
        int born;
}Person;
```

```
void Person_init (Person *self,int pid,int born) {
	self->pid = pid;
	self->born = born;

}
int  Person_getPid (Person *self) {
	return self->pid;
}
int  Person_getBorn (Person *self) {
	return self->born;
}
typedef struct{
	Person base;
	int afm;
	int department;
}Employee;
void Employee_init (Employee *self,int pid,int born,int afm,int department) {
	Person_init((Person*)self, pid, born);
	self->afm = afm;
	self->department = department;
}
int  Employee_getDepartment (Employee *self) {
	return self->department;
}
void  Employee_setDepartment (Employee *self,int department) {
	self->department = department;
}
int  Employee_getPid (Employee *self) {
	return self->afm;
}
int main(){
	Person  george;
	Person_init(&george, 200223,2002);
```

```
        Employee        peter;
        Employee_init(&peter, 200122,2001,990122,1);
        printf ("%d \n",Person_getPid(&george  ));
        printf ("%d \n",Employee_getPid(&peter  ));
Employee_setDepartment(&peter, 2);
}
```

Δ) Εξέταση πολλαπλών ορισμάτων στη συνάρτηση print

```
class Person:
   def __init__(self, pid, born):
      self.pid = pid
      self.born = born
   def getPid(self):
      return self.pid
   def getBorn(self):
      return self.born



if __name__ == '__main__':
   george = Person(200223, 2002)
   print(george.getPid(),5,george.Born())
```

η έξοδος του Δ κώδικα

```
#include <stdio.h>

typedef struct{
        int pid;
        int born;
}Person;
void Person_init (Person *self,int pid,int born) {
        self->pid = pid;
        self->born = born;
}
int  Person_getPid (Person *self) {
        return self->pid;
```

```
}
int  Person_getBorn (Person *self) {
        return self->born;
}
int main(){
        Person  george;
        Person_init(&george, 200223,2002);
        printf ("%d, %d, %d \n",Person_getPid(&george  ),5,Person_getBorn(&george  ));
}
```

UNRESOLVED ISSUES AND BUGS

Α) Στην περίπτωση όπου η συνάρτηση δεν είναι void αν εισαχθεί if statement και έχει την περίπτωση του else part δεν θα εμφανιστεί στο τέλος της συνάρτησης return αν δεν υπάρχει μέσα στο else

Β) Σε οποιαδήποτε περίπτωση έχουμε if statement τα statements που θα βρεθούν από κάτω του θα εισαχθούν ως εμφωλευμένα

Παράδειγμα:

```
class Person:
        def __init__(self, pid, born):
                self.pid = pid
                self.born = born
        def getPid(self):
                 return self.pid
        def getBorn(self):
                 return self.born
        def millenium(self):
```

```
            if self.born < 2000:
                print(self.pid)
                return self.pid
            if(self.born<2004):
                return self.born
    if __name__ == '__main__':
            george = Person(200223, 2002)
            john = Person(200055, 2000)
            print(george.getBorn())
            print(john.getBorn())
```

Αποτέλεσμα:

```
#include <stdio.h>
typedef struct{
    int pid;
    int born;
}Person;
void Person_init (Person *self,int pid,int born) {
    self->pid = pid;
    self->born = born;
}
int  Person_getPid (Person *self) {
    return self->pid;
}
int  Person_getBorn (Person *self) {
    return self->born;
}
int  Person_millenium (Person *self) {
    if (self->born<2000){
```

```
                printf ("%d \n",self->pid);
                return self->pid;
                if ((self->born<2004)){
                        return self->born;
                }
        }
        return 0;
}
int main(){
        Person        george;
        Person_init(&george, 200223,2002);
        Person        john;
        Person_init(&john, 200055,2000);
        printf ("%d \n",Person_getBorn(&george  ));
        printf ("%d \n",Person_getBorn(&john  ));
}
```