ΜΕΤΑΦΡΑΣΤΕΣ 2022

Ομάδα : Γεώργιος Δρογγούλας 4358

Διαμάντης Πλαχούρας 4476

Στην παρούσα εργασία έχουμε υλοποιήσει ένα μεταφραστή για την γλώσσα της Cimple σε γλώσσα Python και Assembly(risc). Η υλοποίηση έχει πέντε φάσεις τις οποίες θα αναλύσουμε στην συνέχεια. Λεκτικός Αναλυτής, Συντακτικός Αναλυτής, Ενδιάμεσος Κώδικας, Πίνακας Συμβόλων και Τελικός Κώδικας.

Πρώτο μέρος: Λεκτική Ανάλυση μεταφραστή

Ξεκινούμε με την λεκτική ανάλυση η οποία αρμόδια να διαβάσει το αρχείο της Cimple και στη συνέχεια να πράξει της απαιτούμενες λεκτικές μονάδες. Στην παρούσα φάση απλά παράγουμε τις λεκτικές μονάδες και δεν ασχολούμαστε με συντακτικά λάθη που μπορεί να κάνει ο προγραμματιστής αυτή τη δουλειά αναλαμβάνει η συντακτική ανάλυση.

Εμβαθύνοντας στην λεκτική ανάλυση σκεφτήκαμε την αντικειμενοστραφή υλοποίηση και δημιουργήσαμε την κλάση token η οποία αναφέρεται σε κάθε λεκτική μονάδα που θα αναγνωρίζει ο λεκτικός αναλυτής. Η κλάση αυτή περιέχει τα πεδία recognized_string, που αναφέρεται η λεκτική μονάδα με αλφαριθμητικό τύπο, το πεδίο family με το οποίο ξεχωρίζουμε σε ποια λεκτική οικογένεια αναφέρεται η αναγνωριζόμενη λεκτική μονάδα, το πεδίο line_number με το οποίο ξεχωρίζουμε την γραμμή στην οποία βρίσκεται η λεκτική μονάδα που αναγνωρίσαμε. Στην ουσία ο λεκτικός αναλυτής αποτελεί μια υλοποίηση ενός αυτόματου που αναγνωρίζει από την είσοδο, χαρακτήρα προς χαρακτήρα, τις λεκτικές μονάδες.

Για την υλοποίηση του αυτομάτου έχουμε τις εξής βοηθητικές συναρτήσεις: numState, loadFile, check_keywords, check_symbol η καθεμία από τις οποίες κάνει τα ακόλουθα

- numState: Η συγκεκριμένη συνάρτηση λαμβάνει έναν αριθμό (1-12) σαν όρισμα από το οποίο καταλαβαίνει ποια κατάσταση να επιστρέψει, 0:"start",1:"dig",2:"idk",3:"error", … , 12: undefined

- laodFile: Η συνάρτηση αυτή λαμβάνει όλο το αρχείο από το όρισμα του τερματικού στο οποίο θέλουμε να κάνουμε την λεκτική ανάλυσης μας.

- check_keywords : Η συνάρτησης αυτή παίρνει σαν όρισμα το ένα αλφαριθμητικό και ελέγχει από τη λίστα των keyWords αν το όρισμα που πήρε είναι μέσα σε αυτή.

- check_symbol : Η συνάρτησης αυτή παίρνει σαν όρισμα ένα αλφαριθμητικό και ελέγχει αν είναι παρένθεση , διαχωριστικός τελεστής , τελεστής πρόσθεσης η πολλαπλασιασμού.

Η βασική συνάρτηση που υλοποιεί το αυτόματο είναι η lexAnalyzer την οποία θα αναλύσουμε στην συνέχεια.

Αρχικά, σε αυτή καλούμε την loadFile για να διαβάσουμε όλο το αρχείο .ci που μας δίνεται. Με αυτήν έχουμε όλο το κείμενό μας χαρακτήρα προς χαρακτήρα σε μία λίστα. Άρα όσο διαβάζουμε κάθε χαρακτήρα κάνουμε κάποιες ενέργειες που σχηματίζουν τις λεκτικές μας μονάδες. Η συνάρτηση που αναλύουμε απαρτίζεται από μία while που διατρέχει όλη την λίστα με τους χαρακτήρες του αρχείου. Για κάθε περίπτωση που συναντούμε έχουμε και μία if για να ξεχωρίζουμε τις λεκτικές μας μονάδες.

Περιπτώσεις

Οι περιπτώσεις για τις οποίες τρέχει το αυτόματό μας είναι οι ακόλουθες:

Αναγνώριση σχολίων

Αρχικά, για κάθε κενό χαρακτήρα απλά τον αγνοούμε και δεν κάνουμε κάποια ενέργεια. Στη συνέχεια, αναγνωρίζουμε τα σχόλια που έχει βάλει ο προγραμματιστής είτε αυτά είναι μονής γραμμής (single line) είτε αυτά είναι πολλαπλής γραμμής (multi line), αυτό σημαίνει πως όταν αναγνωρίσουμε το σύμβολο # το αυτόματο μπαίνει σε κατάσταση σχόλιου και οι επόμενοι χαρακτήρες που αναγνωρίζει θα καταλάβει πως είναι σχόλια μέχρι να ξαναδεί το σύμβολο # και να επανέλθει το αυτόματο στην

αρχική του κατάσταση. Για την κατάσταση των σχολίων δεν δημιουργούμε κάποια λεκτική μονάδα και την αγνοούμε.

Αναγνώριση αριθμών

Η επόμενη κατάσταση που πρέπει να περιγράφει είναι αυτή των αριθμών στην οποία αν αναγνωρίσουμε ένα σύμβολο που είναι αριθμός τότε μεταβαίνουμε στην κατάσταση του αριθμού (diagit) και όσο στη συνέχεια διαβάζουμε αριθμό τον βάζουμε μέσα σε μια προσωρινή λίστα (elements) και συνεχίζουμε μέχρι να δούμε κάτι διαφορετικό. Μόλις αναγνωρίζουμε έναν αριθμό τότε δημιουργούμε ένα token δηλαδή μια λεκτική μονάδα για αυτόν και τον καταχωρούμε στη λίστα των λεκτικών μονάδων tokens. Όταν τελειώσουμε με την δημιουργία του token μηδενίζουμε την προσωρινή λίστα για να είναι έτοιμή για επόμενες κατάσταση του αυτομάτου που θα αναγνωρίσουμε στην συνέχεια.

Αναγνώριση id και keyword

Στη συνέχεια έχουμε θα ασχοληθούμε με την κατάσταση των χαντρών που αναγνωρίζουμε από το αρχείο. Αρχικά για κάθε χαρακτήρα που διαβάζουμε μεταβαίνουμε στην κατάσταση (idκ κατάσταση => id και keyword) και όσο θα διαβάζουμε κάποιο χαρακτήρα θα τον βάζουμε μέσα στη προσωρινή μας λίστα μέχρι να αναγνωρισουμε κάτι διαφορετικό εκτός από χαρακτήρα. Στη συνέχεια όταν φτάσουμε σε σημείο να κάνουμε τη λεκτική μας μονάδα θα πρέπει να ελέγχουμε να αυτό που αναγνωρίσαμε είναι κάποιο keyword με την βοηθητική μας συνάρτηση check_keywords , αν αυτή μας απαντήσει ότι έχουμε κάποιο keyword τότε θα κατασκευάσουμε μια λεκτική μονάδα keyword αλλιώς θα κατασκευάσουμε ένα μια λεκτική αμνάδα identifier, θα μετάβουμε στην αρχική κατάσταση του αυτομάτου και θα αδειάσουμε την προσωρινή μας λίστα.

Αναγνώριση εκχώρησης

Για να αναγνωρίσουμε μία εκχώρηση περιμένουμε να διαβάσουμε από την λίστα τον χαρακτήρα της άνω κάτω τελείας (:). Όταν τον διαβάσουμε τον εκχωρούμε στην προσωρινή λίστα και στην συνέχεια περιμένουμε να δούμε τον χαρακτήρα του (=). Όταν τον δούμε κατασκευάζουμε τη λεκτική μονάδα της εκχώρησης και αδειάζουμε την προσωρινή λίστα.

ΠΡΟΣΟΧΗ: Έχουμε έναν έλεγχο για το αν δούμε άνω κάτω τελεία στο τέλος του αρχείου όπου θέλαμε να πετάξουμε μήνυμα λάθους και να τερματίσουμε την εκτέλεση. Όμως αντί να κάνουμε την συνηθισμένη συνάρτηση exit() καλέσαμε μία συνάρτηση που δεν υπάρχει από απροσεξία. Άρα, αν τοποθετήσετε (:) στο τέλος του αρχείου τότε θα πεταχτεί μήνυμα λάθους από τον compliler της python. Line 173

Αναγνώριση ‘’>’’ και ‘’>=’’

Για την αναγνώριση του χαρακτήρα > πρέπει να τον αναγνωρίσουμε αρχικά από το αρχείο, μεταβαίνουμε στην κατάσταση για realOperator αποθηκεύουμε τον χαρακτήρα σε μία προσωρινή λίστα, στη συνέχεια διαβάζουμε τον επόμενο χαρακτήρα, για τον οποίο πρέπει να βρούμε το σύμβολο ‘’=’’. Όταν το αναγνωρίσουμε αυτό κατασκευάζουμε την λεκτική μας μονάδα και διαγράφουμε την προσωρινή μας λίστα. Στην περίπτωση που επόμενος χαρακτήρας δεν είναι ‘’=’’ κατασκευάζουμε λεκτική μονάδα για το σύμβολο ‘’>’’.

Αναγνώριση ‘’<’’ , ‘’<>’’ και ‘’<=’’

Για την αναγνώριση των παραπάνω χαρακτήρων αρχικά, αναγνωρίζουμε τον χαρακτήρα ‘’<’’ και στην συνέχεια, τον τοποθετούμε σε μία λίστα, μετά στην περίπτωση που αναγνωρίσουμε το σύμβολο ‘’>’’ δημιουργούμε λεκτική μονάδα για το τελεστή του διάφορου ,αν δούμε τον χαρακτήρα ‘’=’’ τότε δημιουργούμε λεκτική μονάδα για το χαρακτήρα ‘’<=’’ αλλιώς θα πρέπει να δημιουργήσουμε μια λεκτική μονάδα για το σύμβολο ‘’<’’. Όταν τελειώνουμε για κάθε περίπτωση καθαρίζουμε την προσωρινή λίστα .

Αναγνώριση ‘’=’’

Για την αναγνώριση του ‘’=’’ περιμένουμε να διαβάσουμε τον χαρακτήρα ‘’=’’ από το αρχείο της cimple και στη συνέχεια κάνουμε την λεκτική μονάδα μας για τον ‘’=’’.

Αναγνώριση ‘’(’’ , ‘’{’’ , ’’[’’ , ‘’]’’, ‘’}’’ , ‘’)’’ , ‘’.’’ , ‘’,’’ , ‘’;’’ , ‘’+’’ , ‘’-‘’ , ’’*’’ , ’’/’’

Για την αναγνώριση των παραπάνω χαρακτήρων ακολουθούμε την ίδια διαδικασία δηλαδή με το που τα αναγνωρισουμε από το αρχείο κατασκευάζουμε την λεκτική μας μονάδα. Η κλίση της συνάρτησης check_symbols γίνεται στην παραπάνω διαδικασία για να ξεχωρίσουμε τι λεκτική μονάδα θα παράγουμε κάθε φορά

Δεύτερο μέρος: Συντακτική Ανάλυση μεταφραστή

Η Συντακτική ανάλυση είναι ένα από τα βασικότερα τμήματα του μεταφραστή, εφοσον η λειτουργικότητα του βασίζεται στον κορμό της συντακτικής ανάλυσης. ΜΕΤΚΑ από την λεκτική ανάλυση είμαστε σε θέση να αναπτύξουμε τη συντακτική ανάλυση η οποία θα γίνει με βάση των λεκτικών μονάδων που αναγνωριστήκαν στο προηγούμενο κομμάτι της ανάλυσης. Πάνω στην παρούσα ανάλυση θα δομηθούν και όλες επόμενες φάσεις του μεταφραστή μας .

Ο συντακτικός αναλυτής είναι υπεύθυνος για τον έλεγχο των συντακτικών σφαλμάτων που μπορεί να έχει ο χρήστης σε οποιοδήποτε σημείο του κώδικα. Για κάθε λεκτική μονάδα που θα καταναλώνεται ο αναλυτής θα πρέπει να αναγνωρίζει αν είναι σωστή η αλληλουχία με την προηγουμένη ώστε το πρόγραμμα που γράφεται από τον προγραμματιστή να έχει συντακτικό νόημα. Για παράδειγμα αν ο χρήστης της γλώσσα γράψει κάποια συνθήκη και στο τέλος δεν βάλει ερωτηματικό τότε ο αναλυτής είναι αυτός που θα πρέπει να του βγάλει το μήνυμα λάθους. Το αποτέλεσμα τις συντακτικής μας ανάλυσης περιμένουμε να παράγει ένα συντακτικό δέντρο πάνω στο οποίο θα βασίζεται η λειτουργικότητα της γλώσσας με φύλλα του να είναι οι λεκτικές μονάδες που δεν θα μπορούν να αναχθούν σε περαιτέρω κανόνες μια τέτοια μονάδα είναι τα ids .

Η γραμματική που θα εφαρμοστεί είναι μια γραμματική χωρίς συμφραζόμενα η οποία είναι κατάλληλη για την υλοποίηση της προσβλέπουσας αναδρομικής κατάβασης που θα χρησιμοποιεί στην ανάλυση μας. Η υλοποίηση της συντακτικής ανάλυσης μας λέει ότι κατά τη συντακτική ανάλυση, όταν ένας κανόνας έχει την εναλλακτική να ενεργοποιήσει περισσότερους από έναν κανόνες ή με άλλα λόγια βρίσκεται σε δίλημμα ποιον από τους διαθέσιμους κανόνες να ενεργοποιήσει, τότε η επιλογή του θα καθοριστεί από την επόμενη λεκτική μονάδα που υπάρχει διαθέσιμη στην είσοδο.

Κάθε κανόνας που πρέπει να αναγνωρίσουμε όπως ένα if ένα while αντιστοιχεί στις αντίστοιχες συναρτήσεις μέσα στο κώδικα του συντακτικού αναλυτή . Για παράδειγμα έχουμε ifStat και whileStat αντίστοιχα για τα if και while , οι συναρτήσεις αυτές καλούνται στην περίπτωση που αναγνωρίσουμε τα αντίστοιχα Keywords if και while.

Για την παρούσα ανάλυση χρησιμοποιούμε μια βοηθητική συνάρτηση get_token() με την οποία θα λαμβάνουμε την επόμενη λεκτική μονάδα από όλες τις αναγνωρισμένες λεκτικές μονάδες που μας έδωσε ο λεκτικός αναλυτής στην λίστα tokens η οποία μετρά την λεκτική ανάλυση είναι γεμάτη με όλες τις λεκτικές μονάδες του αρχείου .ci . Από τη αρχή του προγράμματος καλείται η get_token σε καταλληλά σημεία με την οποία θα καταναλώσουμε την επόμενη λεκτική μας μονάδα.

Σειρά εκτέλεσης των συναρτήσεων που οδηγούν στην συντακτική ανάλυση :

Για να ξεκινήσουμε την συντακτική ανάλυση προϋποθέτουμε ότι πρέπει να έχει γίνει η λεκτική ανάλυση διότι από αυτή θα μας δώσει την είσοδο που θα διαχειριστούμε στη συνέχεια.

Για να τη συντακτική ανάλυση καλούμε μια συνάρτηση program η οποία και θα αρχίσει να δίνει υπόσταση στο συντακτικό δέντρο που θα πρέπει να υλοποιήσει ο συντακτικός αναλυτής. Μέσα στην συνάρτηση αυτή θα κληθεί η block στη οποία είναι υπεύθυνη να δείξει πως θα πρέπει να είναι δομημένο ένα αρχείο της cimple .

Η δομή για ένα αρχείο cimple θα είναι το εξής: Αρχικά θα πρέπει κάθε πρόγραμμα να έχει ένα όνομα και πριν το όνομα θα πρέπει να γραφτεί το keyword program. Κάθε

αρχείο – κώδικας σε cimple θα πρέπει να κλείσει με την τελειά σαν τερματικό χαρακτήρα.

Program -> program ID

block

.

Όταν γραφεί και το όνομα του προγράμματος θα πρέπει ο προγραμματιστής να ανοίξει ‘’{‘’ και να γράψει τον κώδικα του , σε αυτό το σημείο έχει κληθεί η συνάρτηση block η οποία θα τερματιστεί όταν ο χρηστής κλείσει το bracket που άνοιξε στην αρχή. Μέσα στην block πρέπει να καλεστούν οι συναρτήσεις που θα δηλώνουν σε ποια σημεία θα πρέπει να γίνουν οι εκχωρήσεις , να δηλωθούν συναρτήσεις και που θα χρησιμοποιήσουμε τις δομές που παρέχει η γλώσσα μας στον προγραμματιστή όπως assignments , if,for,while statements. Από το συντακτικό της cimple έχουμε ότι στην αρχή του προγράμματος θα πρέπει να γίνεται η δήλωση (declaration) των μεταβλητών , στη συνέχεια έχουμε την δήλωση των συναρτήσεων και των διαδικασιών που θα χρειαστεί ο προγραμματιστής γράφοντας το πρόγραμμα του και στο τέλος έχουμε το main σκέλος του cimple αρχείου οπού θα χρησιμοποιούνται οι δομές της γλώσσας. Αν πάει κάτι λάθος με τη συγγραφή του αρχείου της γλώσσας μας από την μεριά του προγραμματιστή είναι υπεύθυνη η συντακτική ανάλυση να πετάξει το αντίστοιχο μήνυμα λάθους.

Η Συνάρτηση declarations() είναι αυτή η οποία θα επιτρέψει στο χρήση να κάνει την δήλωση των μεταβλητών στην αρχή του προγράμματος .Οι μεταβλητές που θα δηλώσει ο χρήστης μπορεί να περισσότερες από μια έτσι πρέπει του δίνουμε την δυνατότητα να κάνει πολλαπλή συγγραφή δηλώσεων χωρίς να ξαναγραφεί το keyword declare εφοσον κάθε δήλωση θα πρέπει να έχει και το συγκεκριμένο keyword. Αρά όταν δηλώσει μια μεταβλητή μπορεί διπλά σε αυτή διαχωριστικό το ‘’,’’ να δηλώσει και μια δεύτερη μεταβλητή χωρίς να ξαναγράψει το keyword declare. Για να γίνει αυτή τη λειτουργία καλούμε την συνάρτηση Varlist() η οποία θα διαβάσει όλα τα declaration.

Declarations ->(declare varlist ;)*

Varlist -> id

(, id)*

| ε

Στη συνέχεια έχουμε την δήλωση των συναρτήσεων στις οποίες ακολουθούν όμοιες λειτουργείες του κυρίως προγράμματος κατά συνέπεια σε κάθε μια συνάρτηση ή διαδικασία στην αρχή της θα πρέπει να έχει δηλώσεις μεταβλητών ,άλλων συναρτήσεων και στη συνέχεια δομών που είναι διαθέσιμες από τη γλώσσα. Οι συναρτήσεις και οι διαδικασίας παράλυτα πρέπει να έχουν τη δυνατότητα

περάσματος ορισμάτων για να μπορέσει ο χρήστης να χειραγωγήσει τις μεταβλητές του προγράμματος που θα τον απασχολήσουν μέσα στη συνάρτηση. Κάθε συνάρτηση έχει τη δυνατότητα να επιστρέψει τιμή ενώ η διαδικασία δεν μπορεί να το κάνει αυτό. Για να καταλάβει ο συντακτικός αναλυτής ότι στο αρχείο περιέχεται μια συνάρτησης καλείται μέσα στην block του προγράμματος η συνάρτηση subprograms() με την οποία αναγνωρίζουμε τα keywords function και procedure. Μετά από αυτό πρέπει να δοθεί ένα όνομα για την κάθε συνάρτηση και στη συνέχεια να οριστούν οι παράμετροι που θα πάρει κάθε μια συνάρτηση η διαδικασία. Για να γίνει αυτή η αναγνώριση καλείται η συνάρτηση subprogram για την οποία εφοσον αναγνωρίσουμε αν είναι function η procedure θα πρέπει να καλέσουμε την formalparameterList() για να καταλάβουμε τα ορίσματα που ενδεχόμενος θα έχει (μπορεί να μην έχει ορίσματα). Για κάθε αναγνώριση του ορίσματος γίνεται κλήση της formalparItem() με την οποία καταλαβαίνουμε καθένα ξεχωριστά τα ορίσματα. Κάθε όρισμα πρέπει να έχει δηλωμένο το τρόπο που περνάτε πχ in για τιμή και inout για αναφορά . Μετά από αυτά το πρόγραμμα περιμένει να συναντήσει το όνομα της μεταβλητής. Μετά από τα ορίσματα πρέπει να προχωρήσουμε στην κλήση της subBlock() η οποία θα αναγνώριση την ύπαρξη του block για την κάθε συνάρτησης ή διαδικασία ξεχωριστά. Το κάθε block για κάθε συνάρτηση γίνεται γνωστό με το άνοιγμα και κλείσιμο των bracket ‘’{’’ , ‘’}’’ . Μέσα στα block των συναρτήσεων μπορεί να κληθεί οποιαδήποτε δομή προσφέρει η γλώσσα , αλλά με τους αντιστοιχούν κανόνες που θα δηλωνόταν και εκτός των συναρτήσεων δηλαδή μέσα το ίδιο το πρόγραμμα . Αν κάποια από τα παραπάνω βήματα δεν ακολουθηθεί ή ακολουθηθεί με αλλαγμένη σειρά το πρόγραμμα του μεταφραστεί είναι υπεύθυνο να πετάξει ένα μήνυμα συντακτικού λάθους στην αντίστοιχη λεκτική μονάδα.

Subprograms ->(subprogram)*

Subprogram -> function ID (Formalparlist)

block

|procedure ID (Formalparlist)

Block

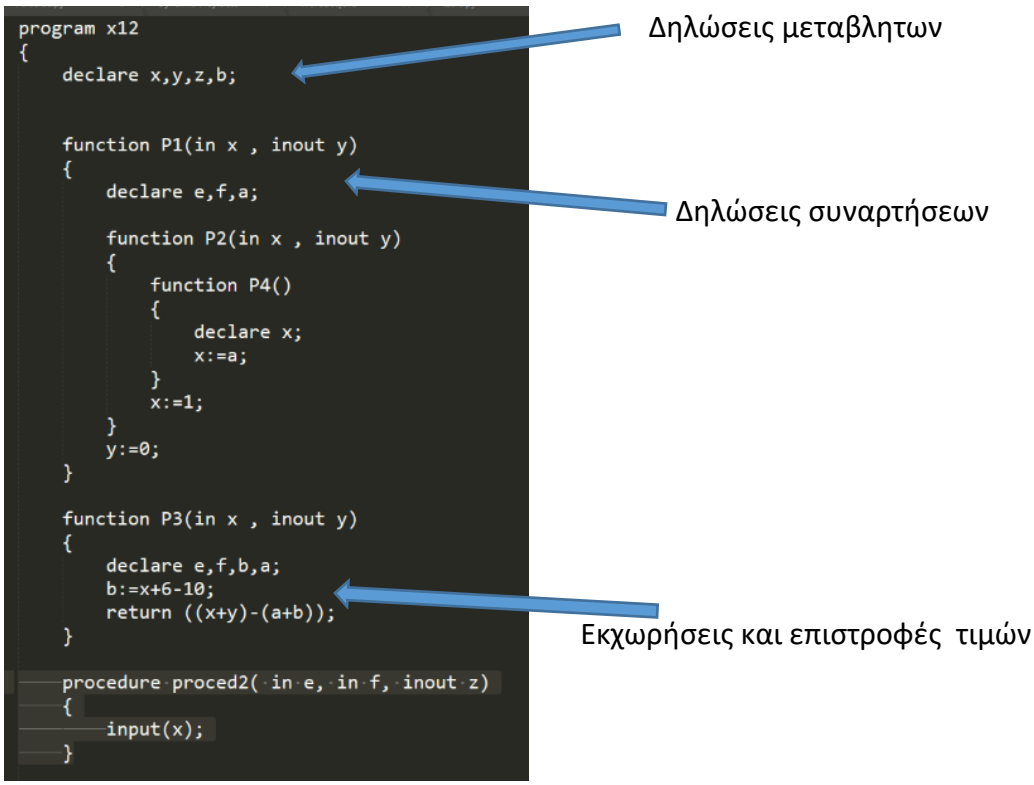
Formalparlist -> fromalparItem (, formalparitem)*

|ε

Formalparitem - > in ID

|inout ID

Ένα παράδειγμα για τη δήλωση των μεταβλητών και των συναρτήσεων θα παρουσιάσουμε παρακάτω.



Το παραπάνω πρόγραμμα δεν περιγράφει κάποια συγκεκριμένη λειτουργεία απλώς παρουσιάζει τη σωστή συντακτική συγγραφή που οφείλει να έχει ο προγραμματιστής κατά τη συγγραφή του αρχείου μέχρι το σημείο δήλωσης συναρτήσεων

1)Περιγραφή για τις δομές της γλώσσας

Τελειώνοντας με τον ορισμό των συναρτήσεων στο πρόγραμμα .ci θα περάσουμε στην αναγνώριση των δομών που θα καλέσει ο προγραμματιστής της cimple.

Για να αρχίσει ο προγραμματιστής να έχει πρόσβαση στις δομές του προγράμματος στις εκχωρήσεις σε print και input και τα λοιπά, θα πρέπει πρώτα να έχει περάσει από τα declarations και τα subprograms. Συνεπώς η ροή του κώδικα θα πρέπει να περάσει μέσα στην block να μπει στα declaration να περάσει από τις συναρτήσεις ή διαδικασίες και στη συνέχεια να μπει στη blockstatements από την οποία θα αναγνωρίσουμε τις ακολουθίες δομές:

Η γλωσσα της cimple διαθετει τις εξης δομες : 1) assignments 2) if 3) switch 4) while 5) incase 6) call 7) return 8) print 9) input καθεμιά από αυτές μπορεί ο μεταφραστής να της αναγνωρίσει κατά τη διάρκεια τις μετάφρασης του αρχείο .ci .

Ο έλεγχος του κώδικα του μεταφραστή μεταβαίνει από την blockstatements στην statement η οποία αναγνωρίζει τις λεκτικές μονάδες που αντιστοιχούν στα keyword των δομών και μεταβαίνει τον έλεγχο στην αντίστοιχη δομή . Για παράδειγμα όταν

θα διαβάσει από το χρήστη το keyword while θα περάσει από την statement θα αναγνωρίσει το αντίστοιχο token και στη συνέχεια θα μεταβεί ο έλεγχος στην συνάρτηση whileStat για να αναγνωρίσει και να καταναλώσει τα tokens που θα ολοκληρώνουν μια σωστή δομή while . Για την περίπτωση που δεν αναγνωρίσει σωστά συντακτικά την δομή θα πρέπει να πετάξει το αντίστοιχο μήνυμα λάθους .

Για να γίνει κατανοητή η σύνταξη του cimple αρχείου στο κομμάτι των δομών θα δείξουμε σχηματικά το συντακτικό με τη σειρά που πρέπει να εκτελεστεί.

blockStatements -> statements(statements)*

statement-> assignStat | ifstat |whileStat | switchStat | forCaseStat| incase | return |input |print |ε

Υλοποίηση AssignStat :

assingStat ->ID := expression

Θα πρέπει κάθε φορά που αναγνωρίζουμε μια λεκτική μονάδα id και βρισκόμαστε σε φάση να αναγνωρίσουμε κάποιο statement τότε ο έλεγχος θα μεταβαίνει στη κατάσταση να αναγνωρίσει μια εκχώρηση. Όταν θα μπει σε αυτή τη διαδικασία πρέπει να γίνεται μια κλήση για να αποτιμήσουμε την έκφραση που βρίσκεται δεξιά της εκχώρησης (αυτό γίνεται με την expression που θα αναλύσουμε στη συνέχεια).

2)Υλοποίηση if,While,switchcase,incase ,forcase

Για τα statements ifstat, whileStat, switchStat, forCaseStat, incase θα χρησιμοποιήσουμε την ακόλουθη διαδικασία με μερικές διάφορες για το καθένα Παρακάτω έχουμε την ανάλυση για αυτά.

Ifstate->if(condition)statements

Elsepart

Elsepart - > statements

|ε

WhileStat - > While(condition) statements

SwitchcaseStat ->switchcase

(case (condition) statements)* default statements

forcaseStat ->forcase

(case (condition) statements)* default statements

IncaseStat -> incase

(case(condition) statements)*

3)Αναγνώριση if και while:

Κάθε φορά που στο πρόγραμμα πρέπει να αναγνωριστεί μια από τις παραπάνω δομές θα πρέπει να αναγνωρίσουμε το αντίστοιχο keyword και να μπούμε στην αντίστοιχη συνάρτηση. Όταν μπούμε στη συνάρτηση περιμένουμε να αναγνωρίσουμε το token της παρένθεσης για να λάβουμε το αντίστοιχο condition για το οποίο θα τρέξει η δομή μας (για την condition θα γίνει παρακάτω η περιγραφή). Στη συνέχεια λαμβάνουμε το επόμενο token και πρέπει να τοποθετήσουμε τη ροή του κώδικα στη statements για να οριοθετήσουμε την δομή με τα ανάλογα brackets, αλλά θα πρέπει να είμαστε σε θέση να αναγνωρίσουμε και αλλά statement που ενδεχόμενος να διαχειρίζεται η δομή που βρισκόμαστε εκείνη τη στιγμή(μέσα στα brackets). Για την περίπτωση που είμαστε στη φάση της if, έχουμε την επιλογή να μετάβουμε στο κομμάτι της else και να καλέσουμε την statements.

4)Αναγνώριση Switchcase , forcase και incase:

Για να λάβουμε την λειτουργικότητα των παραπάνω δομών πρέπει να δούμε το αντίστοιχο token και να εισέλθουμε στις αντίστοιχες συναρτήσεις. Στη συνέχεια πρέπει να είμαστε σε θέση να αναγνωρίσουμε τα cases και να διατρέξουμε τον κώδικα της condition για καθένα από αυτά. Μέσα σε κάθε case θα πρέπει να μπορούμε να έχουμε και άλλες δομές αρά καλούμε την statement. Μπορούμε να έχουμε πολλά cases και μόλις τελειώσουμε με όλα θα πρέπει να υπάρχει και η default case για την περίπτωση που κανένα από τα παραπάνω case δεν ισχύει. Δεν έχουμε την λειτουργία της default για την incase (από την γραμματική μας).

5)Η συνάρτηση statements:

Η Συνάρτηση αυτή είναι μια από τις σημαντικότερες για την λειτουργία του μεταφραστή εφοσον μέσα σε αυτή διαχειριζόμαστε το ποτέ πρέπει ο χρήσης να οριοθετήσει συντακτικά τις δομές του. Για κάθε δομεί, αφού τελειώσει με τα condition και κλείσει την παρένθεση τους θα πρέπει να μπει στη statements. Σε αυτή μεταβαίνει η ροή και περιμένει να αναγνωρίσει για token κάποια από τα σύμβολα που θα χρειαστεί συντακτικά το πρόγραμμα για να οριοθετήσουμε την κάθε δομή. Για την αρχή της δομής πρέπει να δούμε το σύμβολο ‘’{‘’ και για κάθε statements στο τέλος του πρέπει να περιμένουμε ‘’;’’. Τέλος για να βγούμε από την δομεί θα πρέπει να κλείσουμε τα brackets με ‘’}’’ , και αμέσως μετρά με ‘’;’’. Αν δεν ακολουθηθεί αυτή η ακολουθία των λεκτικών μονάδων τότε πρέπει να πεταχτούν μηνύματα λάθους και να τερματίσει η εκτέλεση.

6)Η Συνάρτηση condition :

Σε αυτή τη συνάρτηση αναγνωρίζεται η ακολουθία των λεκτικών μονάδων που συντελούν ένα condition για το οποίο θα ακολουθήσει μια δομεί. Η condition καλείτε

μόλις καταναλωθεί η λεκτική μονάδα που αφορά ένα condition, εκεί δομούμε ένα δέντρο προτεραιοτήτων με κλήση συναρτήσεων που αναγνωρίζουν την ακολουθία για το αντίστοιχο condition . Για παράδειγμα όταν μπούμε σε ένα condition πρέπει να δούμε ποιοι λογικοί τελεστές δηλώνονται μέσα στη συνθήκη και να βαρέσουμε τις αντίστοιχες συναρτήσεις . Οι συναρτήσεις μετά την condition είναι οι boolTerm και η boolFactor στην boolFactor αναγνωρίζουμε είτε τις συνθήκες για κάποια αρνητική η καταφατική έκφραση είτε παράγουμε μια λογική συνθήκη για το για το expression που αναφέρεται μέσα στην condition. Στη συνέχεια πρέπει να αναγνωρίζουμε τα orepator για τα and που μπορεί να έχει η συνθήκη μας (αυτό γίνεται με την boolterm) και μετά πρέπει να αναγνωρίσουμε τα πιθανά or που έχει η συνθήκη μας (αυτό γίνεται με την condition).Αρά όταν βγούμε από την condition έχει αποτιμηθεί η συνθήκη και είμαστε σε θέση να συνεχίσουμε την δομεί μας με τα conditions.

Αυτό το κομμάτι της συντακτικής μας ανάλυσης φαίνεται παρακάτω:

Condition - > boolterm (or boolterm)*

Boolterm -> boolFactor(and boolFactor)*

Boolfactor -> not[condition] | [condition] | expression real_operator expression

real_operator ={ <,>,<=,>=,<>,=)

7)Συνάρτηση για expressions:

Τα expressions είναι οι ακολουθίες που θα πρέπει να αναγνωρίζονται και έχουν να κάνουμε με τις πράξεις και αριθμητικά σύμβολα της γλώσσας . Για να αναγνωρίσουμε μια πρόσθεση , μια αφαίρεση ,έναν πολλαπλασιασμό και μια διαίρεση πρέπει να περάσουμε από τη συνάρτηση expression. Η συνάρτηση αυτή διατηρεί με την κλήση και άλλων συναρτήσεων τον κανόνα της προτεραιότητας που γνωρίζουμε από τα μαθηματικά. Για να το πέτυχει αυτό καλεί την συνάρτηση term που αναγνωρίζει σύμβολα πολλαπλασιασμού(*,/) αναμεσά στα οποία πρέπει να υπάρχουν αριθμοί. Όταν τελειώσει με την αποτίμηση των πολλαπλασιασμών πρέπει να γυρίσει στην expression η οποία αναγνωρίζει σύμβολα πρόσθεσης (+ , -). Συνεπώς η ακολουθία των συναρτήσεων expression και term καλύπτουν τις αριθμητικές πράξεις που πρέπει να αναγνωρίζει η γλώσσα μας.

Expression -> optionalSing term (Add_operator)*

Term -> factor(Mul_operator)*

Factor - > INTEGER |(expression) |id Tail

Idtail-> (actualparlist) | ε

optionalSing –> ADD_op |ε

Το actualparList αναφέρεται σε κλήσεις συναρτήσεων για τα πραγματικά τους ορίσματα και θα αναλυθεί στη συνέχεια.

8)Συνεχίζουμε με τη δομή της call στην οποία έχουμε τις κλήσεις των συναρτήσεων.

CallStat-> call id (Actualparlist)

Για να μπούμε στην call πρέπει να διαβάσουμε το αντίστοιχο keyword και στη συνέχεια πρέπει να λάβουμε με σωστή ακολουθία το όνομα της συνάρτηση και τα ορίσματα της, ανάλογα με ποια συνάρτηση καλούμε. Για να έχουμε σωστή ακολουθία κλήσης μια συνάρτηση πρέπει αφού την καλέσουμε με το όνομα της να βάλουμε τη σωστή πρόθεση in, inout για να ξέρουμε αν τη συνάρτηση την καλούμε σωστά με τα ορίσματα που δηλώθηκε προηγούμενος , αλλά και να ξέρουμε με τι τύπο περάσματος ορισμάτων τη χρησιμοποιούμε. Για να λάβουμε τα ορίσματα καλούμε την actualParlist με την οποία καταναλώνουμε τις λεκτικές μονάδες για τα ορίσματα που γίνεται η κλήση .

9)Κλήσεις συναρτήσεων return , print , input

Return -> return(expression)

Print-> print(expression)

Input -> input(ID)

Στις παραπάνω δομές που προσφέρει η γλώσσα της cimple μπορούμε να εισαχθούμε όταν αναγνωρίσουμε από τη statement το αντίστοιχο token keyword. Στη συνέχεια καταναλώνουμε τις λεκτικές μονάδες οι οποίες και στις δυο περιπτώσεις είναι παρόμοιες. Κάθε statement από τα παραπάνω πρέπει να τελειώνει με ‘’;’’. Για την return την print πρέπει μέσα σε παρενθέσεις να καταναλώσουμε τις λεκτικές μονάδες καλώντας expression και στην input απλά διαβάζουμε τι έχει γράψει ο προγραμματιστής σαν αυτόνομο id.

10)Παρατηρήσεις

Εδώ φτάνουμε στο τέλος της φάσης του συντακτικού αναλυτή και θέλουμε να προσθέσουμε κάποιες παρατηρήσεις. Για κάθε statement ο μεταφραστής μας πρέπει να βλέπει ερωτηματικό. Στην if και while δεν μπορούμε να γράφουμε, αν έχει μια σειρά και μόνο το statement της if, χωρίς ανοικτά brackets κώδικα. Για παράδειγμα το if (condition) “\n’’ x:=1; “\n” ; δεν θα κάνει compile.

Τρίτη Φάση Ενδιάμεσου Κώδικα

Για την φάση του ενδιάμεσου κώδικα έχουμε δημιουργήσει μια αντικειμενοστραφή υλοποίηση με την οποία παράγουμε τετράδες στα καταλληλά σημεία του συντακτικού μας αναλυτή . Αυτό διότι πρέπει να κάθε φορά που αναγνωρίζουμε ορισμένες λεκτικές μονάδες να παράγουμε τον αντίστοιχο ενδιάμεσο κώδικα. Η ενδιάμεση υλοποίηση βοηθάει στην παραγωγή του τελικού κώδικα της πέμπτης φάσης του μεταφραστή.

Μια τετράδα ενδιάμεσου κώδικα είναι μια τετράδα από δεδομένα που δημιουργείται κατά την μεταγλωττιστή. Μια τέτοια τετράδα μοιάζει με την ακόλουθη : operator , operand1 ,operand2 , label . Η παρούσα τετράδα μας λέει ότι θα γίνουν κάποιες ενεργείς που ορίζει ο operator μεταξύ των 2 operand, μετρά από αυτές η εκτέλεση θα πρέπει να μετατεθεί στην τετράδα με τον χαρακτηριστικό label.

Υλοποίηση των τετράδων(quads):

Αρχικά για να γίνουν διακριτές οι τετράδες κάθε φορά που θα θέλουμε να τις δημιουργήσουμε θα δημιουργούμε στην ουσία ένα αντικείμενο (Quad), το οποίο θα έχει ορίσματα που θα αναλυθούν στην συνέχεια. Με τη δημιουργία καθενός quad θα το προσθέτουμε σε μια global λίστα για να κρατήσουμε όλα τα παραγόμενα quad στη λίστα αυτή με αποτέλεσμα αυτά να είναι προσπελάσιμα για μετέπειτα εργασίες. Κάποιες από τις τετράδες που θα παραχθούν θα είναι κάνες κατά τη διάρκεια της μετάφρασης εφοσον δεν θα γνωρίζουμε εκείνη τη στιγμή τις απαιτούμενες τελικές τους ενεργείς . Οι τετράδες αυτές θα πρέπει να είναι γεμάτες με τα απαιτούμενα στοιχεία πριν την ολοκλήρωση της μετάφρασης . Για τη δημιουργία και την διεκπεραίωση όλων των λειτουργιών που πρέπει να σχηματιστούν μετά την παραγωγή του ενδιάμεσου κώδικα. Έχουμε αρκετές βοημικές συναρτήσεις που θα καλούνται κάθε φορά για να διευκολύνουν την λειτουργικότητα του κώδικα εκεί που πρέπει να καλεστούν τις οποίες θα τις αναλύσουμε στη συνέχεια.

Ας αρχίσουμε με την δημιουργία της κλάσης quad η οποία απευθύνεται σε μια τετράδα. Κάθε ένα quad θα πρέπει να έχει τα Εξής πεδία . Ένα label με το οποίο θα αριθμούμε κάθε τετράδα που θα παράγουμε ,ξεκινώντας από το 1 έως τον αριθμό της τελευταίας τετράδας που θα παραχθεί. Στη συνέχεια έχουμε έναν operator ο οποίος θα είναι ένα αλφαριθμητικό που θα μας δίνει να καταλάβουμε τι πράξη εκτελεί η κάθε τετράδα και ποια είναι η δουλειά της. Στη συνέχεια έχουμε 2 πεδία operand1 kai operand2 τα οποία είναι αλφαριθμητικά , οπού για την ενδιάμεση αναπαράσταση θα εκτελούν την πράξη που θα δηλώνει ο operator. Τέλος έχουμε τον operand3 οπού σε κάποιες περιπτώσεις (που θα αναλύσουμε αργότερα) θα είναι ένα label μια άλλης τετράδας που θα μετατεθεί η ροή του κώδικα αν ισχύει κάποια συνθήκη , η θα είναι ένα άλλο αλφαριθμητικό το οποίο θα βοηθάει στις πράξεις που θα εκτελούν τα προηγούμενα πεδία. Αν ο operator είναι μία ενέργεια που έχει νόημα

να γίνει πάνω σε λιγότερα από τρία πεδία – τελούμενα , τότε πρέπει ένα ή δύο ή και τρία από αυτά να μείνουν κενά. Αν ο operator απαιτεί περισσότερα από τρία πεδία, τότε πρέπει να βρούμε κάποια λύση ώστε να μπορέσουμε να περιγράψουμε αυτό που θέλουμε με κάποιον άλλον τρόπο(συνήθως με προσωρινές μεταβλητές που θα δημιουργούμε σε αυτή τη φάση).

Κλάση των quad

```
class Quad:

    def __init__(self,operator,operand1,operand2,operand3):
        global label
        self.operator = operator  #einai o telesths
        self.operand1 = operand1  #einai oi teloumenoi
        self.operand2 = operand2
        self.operand3 = operand3
        self.label = label

    def write(self):
        string = str(self.label) +": ,"+ self.operator +"," +
self.operand1 +"," + self.operand2 + ","+ self.operand3 + "\n"
        return string

    def __str__(self):              #100 : begin_block , _ , _ , _
        return str(self.label) +": ,"+ self.operator +"," +
self.operand1 +"," + self.operand2 + ","+ self.operand3
```

Βοηθητικές συναρτήσεις

Για την παραγωγή του ενδιάμεσου κώδικα υπάρχουν κάποιες ενέργειες που επαναλαμβάνονται συχνά και είναι κατάλληλες για να τις υλοποιήσουμε με τη μορφή συναρτήσεων. Αυτές είναι οι ακόλουθες:

- genQuad(operator, operand1, operand2, operand3): Δημιουργεί μία νέα τετράδα, το πρώτο πεδίο της οποίας είναι το operator και τα τρία επόμενα τα operand1, operand2 και operand3. Ο αριθμός της τετράδας που δημιουργείται προκύπτει αυτόματα από τον αριθμό της τελευταίας τετράδας που δημιουργήθηκε, συν ένα.

- nextQuad(): Επιστρέφει την ετικέτα της επόμενης τετράδας που θα δημιουργηθεί, όταν κληθεί η genQuad.

- newTemp(): Επιστρέφει το όνομα της επόμενης προσωρινής μεταβλητής. Αν η τελευταία προσωρινή μεταβλητή που δημιουργήθηκε είναι η T_2, τότε θα δημιουργήσει και θα επιστρέψει την T_3,

- emptyList(): Δημιουργεί και επιστρέφει μία νέα κενή λίστα στην οποία στη συνέχεια θα τοποθετηθούν ετικέτες τετράδων

- makeList(label): Δημιουργεί και επιστρέφει μία νέα λίστα η οποία έχει σαν μοναδικό στοιχείο της την ετικέτα τετράδας label

- mergeList(list1, list2): Δημιουργεί μία λίστα και συνενώσει τις list1 και list2 σε αυτήν.

- backpatch(list,label): Διαβάζει μία μία της τετράδες που σημειώνονται στη λίστα list και για την τετράδα που αντιστοιχεί στην ετικέτα αυτή, συμπληρώνουμε το τελευταίο πεδίο της με το label. Όταν συμπληρωθούν όλες οι τετράδες που σημειώνονται στη λίστα αυτή, η λίστα δεν χρειάζεται άλλο και μπορεί να αποδεσμεύσει τη μνήμη που κατέχει

- printAllQuad() : Διαβάζει τη λίστα με όλες τις τετράδες και την τυπώνει σε ένα αρχείο .int

Στο σημείο που κατασκευάζουμε την κλάση για τα quads έχουμε και μια καλή rule την οποία τη χρησιμοποιούμε για trueList και falseList που αναφέρονται στο rule. Την υλοποίηση αυτή θα την αναλύσουμε στη συνέχεια.

```
class rule:

    def __init__(self , trueList , falseList):
        self.trueList = trueList
        self.falseList = falseList
```

Υλοποίηση του Ενδιάμεσου μέσα στον Συντακτικό μας αναλυτή:

Αρχίζουμε με τις αριθμητικές παραστάσεις. Υποστηρίζονται οι 4 βασικές αριθμητικές πράξεις, καθώς η ομαδοποίηση και η προτεραιότητα ανάμεσα σε αυτές.

Ξεκινάμε με την συνάρτηση expression() στην οποία αναγνωρίζονται οι πράξεις της πρόσθεσης και της αφαίρεσης. Πριν αναγνωρίσουμε κάποια από αυτές τις πράξεις η ροή του κώδικα μεταφέρεται στην term() στην οποία αναγνωρίζονται οι πράξεις πολλαπλασιασμού και διαίρεσης. Πριν αναγνωρίσουμε και αυτές τις πράξεις μεταφέρουμε τη ροή του κώδικα στη factor() η οποία αναγνωρίζει τις τερματικές λεκτικές μονάδες (τους αριθμούς). Γυρνώντας από την factor() αν έχουμε πολλαπλασιασμό ή διαίρεση, τότε δημιουργούμε την τετράδα με operator που συμβολίζει την αντίστοιχη πράξη και operand1 και operand2 να έχουν τα expression() που γύρισαν οι δύο factor(). Για το operand3 έχουμε μία καινούρια προσωρινή μεταβλητή. Βγαίνοντας από την term πιθανό να έχουμε πρόσθεση ή αφαίρεση, άρα δημιουργούμε την αντίστοιχη τετράδα με operator την πράξη της πρόσθεσης ή της αφαίρεσης, με τα δύο επόμενα πεδία να παίρνουν τις τιμές των term() που κλήθηκαν στην expression(). Σχηματικά έχουμε τα παρακάτω:

Για την expression(): E → T (1) (+ T (2) {p1}) ∗ {p2}

{p1} : w = newTemp()

genQuad('+', T (1).place, T (2).place, w)

T (1).place = w

{p2} : E.place = T (1).place

Για την term(): T → F (1) (* F (2) {p1}) ∗ {p2}

{p1} : w = newTemp()

genQuad('+', F (1).place, F (2).place, w)

F (1).place = w

{p2} : T.place = F (1).place

Για την factor(): F → (E) {p1}

{p1} : F.place = E.place

Για την factor(): F → ID {p1}

{p1}: F.place = ID.place

Υλοποίηση για λογικές για λογικές συνθήκες:

Κάθε λογική συνθήκη αναγνωρίζεται με την επιστροφή του κώδικα από τη condition(). Με την εισαγωγή στην condition() πριν την αναγνώριση του or ο κώδικας μεταφέρεται στην boolterm() οπου πριν την αναγνώριση του end ο κώδικας μεταφέρεται στην boolfactor(). Στην boolfactor() αναγνωρίζουμε τις περιπτώσεις της άρνησης ενός condition οπου και κατασκευάζουμε τις δύο λίστες του κανόνα trueList και falseList ανάποδα από τον κανόνα που επιστρέφει το condition για να έχουμε άρνηση, τις περιπτώσεις της κατάφασης ενός condition οπου κατασκευάζουμε τις δύο λίστες του κανόνα κανονικά όπως επιστρέφονται από ένα condition. Αν δεν έχουμε αγκύλες για κατάφαση ή άρνηση τότε μεταβαίνουμε στο κομμάτι του κώδικα που συγκρίνει δύο αριθμητικά expression με έναν operator. Για την παραγωγή του quad που θα δημιουργηθεί σε αυτή την περίπτωση σημειώνουμε μία λίστα με το label της τετράδας που θα δημιουργηθεί, με την βοηθητική συνάρτηση makeList(). Η λίστα αυτή γίνενται return σαν την καταφατική λίστα του κανόνα. Σε αυτό το σημείο δημιουργούμε την τετράδα με το operator που συγκρίθηκαν τα δύο αριθμητικά expression. Μετά δημιουργούμε με τον ίδιο τρόπο όπως και στην καταφατική λίστα, την αρνητική λίστα του κανόνα σημειώνοντας την επόμενη τετράδα. Τέλος, πριν βγούμε από την boolfactor() δημιουργούμε μία τετράδα κενού jump στην οποία θα κάνουμε backpatch στην συνέχεια. Οι κανόνες που δημιουργήθηκαν γίνονται return για την boolterm(). Βγαίνοντας από την boolfactor() και έχοντας τους κανόνες που αυτή γυρίζει μπαίνουμε στην boolterm(). Στην περίπτωση που έχουμε end δημιουργούμε τις λίστες του κανόνα και όταν διαβάσουμε end κάνουμε backpatch με την καταφατική λίστα την τετράδα που αφήσαμε ασυμπλήρωτη πριν

η οποία περιείχε τον λογικό operator (η τετράδα με το jump γίνεται backpatch σε μία από τις δομές για την οποία θα κληθεί το αντίστοιχο condition). Στην συνέχεια κατασκευάζουμε την trueList του κανόνα της booterm() και με mergeList() κατασκευάζουμε την λίστα του κανόνα falseList. Στο τέλος της συνάρτησης κάνουμε return τον κανόνα στην condition() με τις λίστες που δημιουργήσαμε. Γυρνώντας από την boolterm() κατασκευάζουμε λίστες από τον κανόνα που γύρισε η boolterm() και αν συναντήσουμε or κάνουμε backpatch με το falseList με την επόμενη τετράδα που θα δημιουργηθεί. Με την επόμενη κλήση της boolterm() έχουμε τον επόμενο κανόνα από τον οποίοι κατασκευάζουμε τις δύο λίστες που θα επιστρέψει η condition(). Η καταφατική λίστα που γυρνάει η condition() γίνεται με την κλήση της mergeList με ορίσματα την λίστα του κανόνα που είχαμε στην αρχή του condition και σαν δεύτερο όρισμα την λίστα που μόλις μας γύρισε η boolterm().

Σχηματικά έχουμε τα παρακάτω:

Για την condition(): B → Q (1) {p1} (or {p2} Q (2) {p3}) ∗

{p1}: B.true = Q (1).true

B.false = Q (1).false

{p2}: backpatch(B.false, nextquad())

{p3}: B.true = mergeList(B.true,Q (2).true)

B.false = Q (2).false

Για την boolterm(): Q → R (1) {p1} (or {p2} R (2) {p3}) ∗

{p1} : Q.true = R (1).true

Q.false = R (1).false

{p2} : backpatch(Q.true, nextquad())

{p3} : Q.false = mergeList(Q.false,R (2).false)

Q.true = R (2).true

Για την boolfactor(): R → E (1) rel_op E(2) {p1}

{p1} : R.true = makeList(nextQuad())

genQuad(rel_op, E (1).place, E (2).place}, '_')

R.false = makeList(nextQuad())

genQuad('jump', '_', '_', '_') R → [B]

R → [B] {p1}

{p1} : R.true = B.true

R.false = B.false

R → not [B] {p1}

{p1} : R.true = B.false

R.false = B.true

Υλοποίηση για το Τέλος και την Αρχή του προγράμματος:

Για την οριστικοποίηση της αρχή και του τέλος ενός block αλλά και για την οριοθέτησης του ιδίου του προγράμματος έχουμε την δημιουργία των begin και end quad το οποία έχουν την παρακάτω δομεί begin_block, name,_,_ και end_block,name,_,_. Για κάθε συνάρτηση που θα δημιουργήσει ο προγραμματιστής θα πρέπει να έχουμε και την αντίστοιχη begin και end quad. Για τις συναρτήσει τα quad εμφανίζονται ψηλότερα από τα αυτά τα begin και end quad του κυρίως προγράμματος. Τα σημεία που έχουμε δημιουργήσεις τα begin kai end block για της συναρτήσεις είναι μέσα στην SubBlock () και για το πρόγραμμα έχουμε την begin στην αρχική συνάρτηση block μετρά τα subprograms και την end Block με το που τελειώσει το πρόγραμμα.

Υλοποίηση για την είσοδο και έξοδο των δεδομένων

Σε αυτή την κατηγορία συναντούμε τις συναρτήσεις return print και input .Σε κάθε μια από αυτές τις συναρτήσεις δεν κάνουμε τίποτε παραπάνω από το να δημιουργήσουμε την αντίστοιχη τετράδα για την κάθε περίπτωση. Για το return έχουμε το quad (‘’ret”,expression,”_”,”_”). Για την περίπτωση του print έχουμε το quad (‘’out”,expression,”_”,”_”). Τέλος για την περίπτωση του input έχω quad (‘’in”,InputString,”_”,”_”).

Υλοποίηση για το τερματισμό του Προγράμματος

Δημιουργούμε μια τετράδα quad (‘’Hult”,’’_’’,”_”,”_”) . Η halt βοηθάει και στο να είμαστε βέβαιοι ότι όταν παράγουμε κώδικα για μία δομή και κάνουμε backpatch() έξω από αυτήν, θα υπάρξει μία εντολή η οποία θα δημιουργηθεί, ακόμα κι αν φαινομενικά η δομή που μεταφράζουμε είναι η τελευταία του προγράμματος.

Υλοποίηση Δομών Του κυρίως προγράμματος

Υλοποίηση if – else

Με την εισαγωγή του κώδικα στην if και μετά την αποτίμηση του condition μας επιστρέφεται ένας κανόνας ifRule στον οποίο κάνουμε backpatch με την trueList του κανόνα στο nextQuad που θα δημιουργηθεί. Στη συνέχεια, δημιουργούμε μία λίστα με makeList() με το nextrQuad το οποίο και θα δημιουργήσουμε αμέσως μετά το οποίο θα είναι ένα κενό jump. Στην συνέχεια κάνουμε backpatch για το falseList του κανόνα της condition() με το nextQuad. Με την εισαγωγή του κώδικα στην else δεν παράγουμε κάποια τετράδα και μετά από αυτή κάνουμε backpatch στο κενό jump που δημιουργήσαμε πριν με το nextQuad.

Σχηματικα εχουμε για την if-else :ifStat → if (condition) {p1} statements(1) {p2}

elsePart {p3}

elsePart → else statements(2)

| ε

{p1}: backpatch(condition.true,nextquad())

{p2}: ifList=makeList(nextQuad())

genQuad('jump','_','_','_')

backpatch(condition.false,nextquad())

{p3}: backpatch(ifList,nextquad())

Υλοποίηση while

Με το που μπούμε στην συνάρτηση της while σημειώνουμε σε μία μεταβλητή με τη συνάρτηση nextQuad() την επόμενη τετράδα. Με την condition() μετά την λεκτική μονάδα της while λαμβάνουμε ένα κανόνα τον οποίο θα χρησιμοποιήσουμε στη συνέχεια. Μετά την condition() κάνουμε backpatch με το trueList στο nextQuad. Μετά από τα statement που μπορεί να έχει το while δημιουργούμε μία τετράδα quad (‘’jump”,’’_’’,”_”,”το label της τετράδας που σημειώσαμε στην αρχή”). Μετά από αυτό κάνουμε backpatch με το falseList, που μας είχε γυρίσει η condition(), στο nextQuad

Σχηματικά για while έχουμε: whileStat → while {p0} (condition) {p1} statements {p2}

{p0} : condQuad = nextQuad()

{p1} : backpatch(condition.true,nextQuad())

{p2} : genQuad('jump','_','_',condQuad)

backpatch(condition.false,nextQuad())

Υλοποίηση forCase()

Με το που μπούμε στον κώδικα της forstat() σημειώνουμε την επόμενη τετράδα σε μία μεταβλητή firstQuad και περιμένουμε να διαβάσουμε τα cases στα οποία καλούμε την condition η οπία και επιστρέφει έναν κανόνα μετά από κάθε condition ενός case κάνουμε backpatch με την trueList στο nextQuad. Μετά από τα statement δημιουργούμε μία τετράδα με jump και τελευταίο operand το firstQuad. Στη συνέχει κάνουμε backpatch με την falseList του rules condition στο nextQuad.

Σχηματικά έχουμε για την forstat(): forcaseStat → forcase {p1} (case (condition) {p2}

statements(1) {p3}) ∗ default statements(2)

{p1} : firstCondQuad = nextQuad()

{p2} : backpatch(condition.true,nextQuad())

{p3} : genQuad('jump','_','_','_')

backpatch(condition.false,nextQuad())

Υλοποίηση switchstat

Με το που μπούμε στην Switch θα πρέπει να δημιουργήσουμε μια κενή λίστα με την συνάρτηση exitList και να την αποθηκεύσουμε σε μια προσωρινή λίστα. Όσο βλέπουμε ότι έχουμε cases καλούμε την condition η οποία επιστρέφει ένα rule. Μετά την condition θα έχουμε για κάθε case μια backpatch με τη trueList ,που μας γύρισε η condition, στο nextQuad. Στη συνέχεια μετά τα statements θα κατασκευάσουμε μια λίστα με το nextQuad με την χρήση της makeList() και αμέσως μετά θα δημιουργήσουμε ένα κενό jump. Μετά θα γεμίσουμε το exitList με τη χρήση της mergeList η οποία θα πάρει για ορίσματα το exitList και την λίστα που κατασκευάσαμε πριν το jump. Αμέσως μετά θα γίνει backpatch με το falseList, του κανόνα της condition της case, στο nextQuad. Μετά τα cases όταν έρθει η ώρα για το default θα πρέπει στο τέλος να γίνει backpatch με το exitList ,που δημιουργήθηκε πριν, στο nextQuad().

Σχηματικά για την switchcase έχουμε: switchcaseStat → switchcase {p0}

(case (condition) {p1} statements(1) {p2}))∗
default statements(2) {p3}

{p0}: exitList = emptyList()

{p1}: backpatch(condition.true,nextQuad())

{p2}: t = makeList(nextQuad)

genQuad('jump','_','_','_')

exitList = mergeList(exitList,t)

backpatch(condition.false,nextQuad())

{p3}: backpatch(exitList,nextQuad())

Υλοποίηση incase

Με την εισαγωγή του κώδικα στη συνάρτηση incase δημιουργούμε μια προσωρινή μεταβλητή(newTemp()) και την αποθηκεύουμε σε μια μεταβλητή flag, επίσης αποθηκεύουμε την ετικέτα της επομένης τετράδας σε μια μεταβλητή firstQuad και δημιουργούμε μια τετράδα εκχώρησης genQuad(':=','0','_','flag'). Μετα για κάθε case έχουμε και ένα condition που μας επιστρέφει κάποιο κανόνα. Στη συνέχεια για κάθε case κάνουμε backpatch με το truelist του rule της condition στο nextQuad.Μετά από τα statement του case δημιουργούμε μια τετράδα genQuad(':=','1','_','flag') και κάνουμε backpatch με το falseList ,του κανόνα της condition του case, στο nextQuad. Μετα από τα cases δεν έχουμε default συνθήκη και δημιουργούμε τα quad genQuad('=','1','flag','firstQuad').

Σχηματικά για την incase έχουμε: incaseStat -> incase {p1} (case (condition) {p2} statements(1) {p3}) ∗

{p1}: flag = newTemp()

firstQuad = nextQuad()

genQuad(“:=”,”0”,”_”,flag)

{p2}: backpatch(incaseRule.trueList,nextQuad)

{p3}: genQuad(“:=”,”1”,”_”,flag)

Backpatch(incaseRule.flaseList,nextQuad)

{p4}: genQuad(“=”,”1”,”flag”,firstQuad)

ΓΙΑ ΤΗΝ ΥΛΟΠΟΙΗΣΗ ΤΗΣ INCASE ΔΕΝ ΕΙΜΑΣΤΕ ΣΙΓΟΥΡΟΙ ΠΩΣ ΕΚΤΕΛΕΙΤΑΙ ΣΩΣΤΑ ΓΙΑ ΤΟΝ ΕΝΔΙΑΜΕΣΟ ΑΡΑ ΤΟ ΛΑΘΟΣ ΜΕΤΑΦΕΡΕΤΑΙ ΚΑΙ ΣΤΙΣ ΕΠΟΜΕΝΕΣ ΦΑΣΕΙΣ

Τέταρτη Φάση Πίνακας Συμβολών

πίνακας συμβόλων είναι δυναμική δομή στην οποία αποθηκεύεται πληροφορία σχετιζόμενη με τα συμβολικά ονόματα που χρησιμοποιούνται στο υπό μεταγλώττιση πρόγραμμα. Η δομή αυτή ακολουθεί τη μεταγλώττιση και μεταβάλλεται δυναμικά, με την προσθήκη ή αφαίρεση πληροφορίας σε και από αυτήν, ώστε σε κάθε σημείο της διαδικασίας της μεταγλώττισης να περιέχει ακριβώς την πληροφορία που εκείνη τη στιγμή πρέπει να έχει. Σε ένα πίνακα συμβόλων διατηρούμε πληροφορία για τα συμβολικά ονόματα που εμφανίζονται στο πρόγραμμα. Έτσι, σε έναν πίνακα συμβόλων αποθηκεύεται πληροφορία που σχετίζεται με τις μεταβλητές του προγράμματος, τις διαδικασίες και τις συναρτήσεις, τις παραμέτρους και με τα ονόματα των σταθερών. Για κάθε ένα από αυτά υπάρχει διαφορετική εγγραφή στον πίνακα και στην εγγραφή αυτή αποθηκεύεται διαφορετική πληροφορία, ανάλογα με το είδος του συμβολικού ονόματος. Η πληροφορία αυτή είναι χρήσιμη για έλεγχο σφαλμάτων, αλλά είναι διαθέσιμη να ανακτηθεί κατά τη φάση της παραγωγής του τελικού κώδικα. Για υλοποίηση του πίνακα συμβολών έχουμε χρησιμοποιήσει αντικειμενοστραφή υλοποίηση την οποία θα περιγράψουμε στη συνέχεια.

Αντικειμενοστραφή Υλοποίηση.

Σε αυτό το κομμάτι της ανάλυσης θα περιγράψουμε το ιεραρχικό μοντέλο που θα χρησιμοποιήσουμε για την υλοποίηση. Ας αρχίσουμε με τον πίνακα ο οποίος θα διαθέτει όλες της πληροφορίες του μέσα σε μια λίστα. Συνεπώς όλη τη υπόσταση του table θα είναι αποθηκευμένη σε μια globa λίστα. Ο Πίνακας διαθέτει επίπεδα ανάλογα με την υλοποίηση του. Κάθε επίπεδο είναι ένα scope το οποίο θα δημιουργείται όταν ξεκινάμε τη μετάφραση μιας νέας συνάρτησης. Ένα scope διαγράφεται όταν τελειώνουμε τη μετάφραση μιας συνάρτησης. Με τη διαγραφή διαγράφουμε την εγγραφή (record) του Scope και όλες τις λίστες με τα Entity και τα Argument που εξαρτώνται από αυτήν. Κάθε ένα scope διαθέτει πολλά entities τα οποία είναι αντικείμενα διαφορετικών τύπων που θα αναλύσουμε στην συνέχεια. Η προσθήκη ενός entity σε κάθε scope γίνεται όταν συναντάμε δήλωση μεταβλητής, όταν δημιουργείται νέα προσωρινή μεταβλητή, όταν συναντάμε δήλωση νέας συνάρτησης, όταν συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης. Στη συνέχεια της ανάλυσης θα περιγράψουμε και τις βοηθητικές συναρτήσεις που θα χρειαστούν για την παραγωγή του πίνακα.

Αρχικα εχουμε την κλαση scope η οποια εχει σαν πεδιο της μια λιστα από entities.

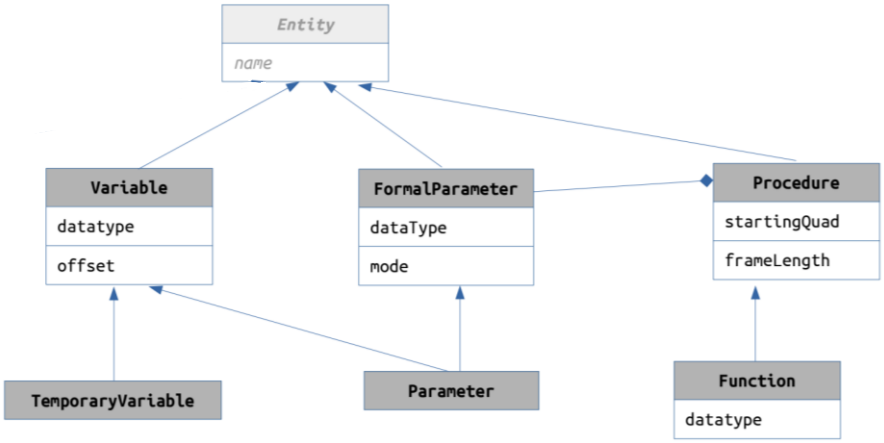
```
class scope():
    def __init__(self):
        self.entitylist = []
```

Στη συνεχεια εχουμε την κλαση entity η ποια περιγραφει την κάθε οντοτητα κάθε μεταβλητης ή συναρτησης ξεχωριστα. Αρα θα εχουμε ένα πεδιο name για καθε entity.

```
class Entity():

    def __init__(self,name):
        global scopeNum
        self.name = name
```

Στη συνεχεια αρχιζει η ιεραρχικη υλοποιηση η οποια φαινεται στο παρακατω σχημα.



Ας αρχίσουμε με την κλάση Variable την οποία υλοποιήσαμε σαν υπόκλιση του entity. Η κλάση αυτή συμπεριλαμβάνει όλες τις μεταβλητές που μπορούμε να συναντήσουμε μέσα στο πρόγραμμα οι οποίες δηλώνονται σαν declares στην αρχή κάθε συνάρτησης ή του κυρίως προγράμματος. Η κλάση variable διαθέτει τα πεδία name , το οποίο το κληρονομεί από το entity , το πεδίο datatype το οποίο είναι πάντα integer , το πεδίο offset το οποίο μας λέει σε ποιο σημείο του εγγραφήματος δραστηριοποίησης για κάθε scope θα βρίσκεται η μεταβλητή και τέλος έχει ένα πεδίο scopelevel για να γνωρίζουμε σε ποιο scope βρίσκεται η κάθε μεταβλητή εκείνη την ώρα που την βλέπουμε. Επίσης έχουμε ένα πεδίο isglobal για κάθε variable το οποίο είναι true αν το scopelevel είναι μηδέν και είναι false αν το scopelevel είναι μεγαλύτερο.

```
class Variable(Entity): #child of entity

    def __init__(self,name,datatype,offset,scopeLevel):
        Entity.__init__(self, name)
        self.datatype =datatype
        self.offset = offset
        self.scopeLevel = scopeLevel
```

```
        if (self.scopeLevel == 0): # the variable is global
            self.isglobal = True
        else:
            self.isglobal = False
```

Στη συνέχεια έχουμε την κλάση formlaParameter. Ένα αντικείμενο formalParameter το κατασκευάζουμε όταν διαβάζουμε τις λεκτικές μονάδες για τα τυπικά ορίσματα στη δήλωση μιας συνάρτησης. Συνεπώς θα έχουμε μια κλάση για αυτά τα αντικείμενα η οποία θα έχουν τα εξής πεδία : Ένα name το οποίο το κληρονομεί από το entity ένα datatype το οποίο θα είναι integer, ένα offset για να ξέρουμε σε ποιο σημείο του εγγραφήματος δραστηριοποίησης του scope θα βρίσκεται η παραμετρος , ένα πεδίο mode για να γνωρίζουμε τον τρόπο περάσματος της παραμέτρου και ένα scopelevel για να ξεχωρίζουμε σε ποιο dcope ανήκει.

```
class FormalParameter(Entity):

    def __init__(self,name,datatype,offset,mode,scopeLevel):
        Entity.__init__(self,name)
        self.datatype = datatype
        self.offset = offset
        self.mode = mode
        self.scopeLevel = scopeLevel
        self.isglobal = False
```

Στη συνέχεια έχουμε μια κλάση procedure η οποία αναφέρεται στις διαδικασίες του αρχείου που μεταφράζουμε. Το αντικείμενο αυτό το δημιουργούμε όταν συναντήσουμε μια διαδικασία ,τότε δημιουργούμε και ένα νέο scope για το εγγράφημα δραστηριοποίησης της διαδικασίας .Στη συνέχεια το τοποθετούμε στη λίστα με τα entities για να μπει στο αντίστοιχο scope. Το αντικείμενο αυτό έχει τα εξής πεδία : Ένα πεδίο startingQuad για να ξέρουμε την τετράδα με την οποία ξεκινάει μια διαδικασία , ένα πεδίο framelength για να γνωρίζουμε το μέγεθος του εγγραφήματος δραστηριοποίησης της διαδικασίας, εάν πεδίο offset για το οποίο ξέρουμε που σε ποιο σημείο θα αρχίσει η διαδικασία για το προηγούμενο scope και ένα scopeLevel για να γνωρίζουμε σε ποιο επίπεδο που θα έχουμε την αντίστοιχη διαδικασία.

```
class Procedure(Entity):

    def
__init__(self,name,startingQuad,framelength,formalParameter,offset,scop
eLevel):
        Entity.__init__(self,name)
        self.startingQuad = startingQuad
        self.framelength = framelength
        self.offset=offset
        self.scopeLevel = scopeLevel
```

Η επόμενη κλάση είναι ακριβώς αντίστοιχη με την προηγούμενη, μόνο που τώρα αναφερόμαστε σε συναρτήσεις και την ονομάζουμε function η οποία έχει ακριβώς τα ιδιά πεδία και ένα ακόμα το datatype. Η function είναι κλάση παιδί της procedure και κληρονομεί όλα της τα πεδία. Ένα αντικείμενο function το δημιουργούμε όταν αναγνωρίσουμε μια συνάρτηση και τότε δημιουργούμε και ένα νέο scope για το εγγράφηκα δραστηριοποίησης της συνάρτησης. Στη συνέχεια το τοποθετούμε στη λίστα με τα entities για να μπει στο αντίστοιχο scope.

```
class function(Procedure):

    def__init__(self,name,startingQuad,framelength,formalParameter,data
type,offset,scopeLevel):
        Procedure.__init__(self,name,startingQuad,framelength,formalPar
ameter,offset,scopeLevel)
        self.datatype = datatype
```

Τέλος έχουμε μια κλάση TemporaryVariable αντικείμενα της οποία δημιουργούμε όταν χρειάζεται να κατασκευάσουμε μια νέα προσωρινή μεταβλητή. Η συγκεκριμένη κλάση είναι παιδί της κλάσης variable αρά κληρονομεί όλα της τα πεδία. Κάθε φορά που δημιουργούμε μια newTemp από τον ενδιάμεσο τότε τοποθετούμε αυτή τη μεταβλητή και στον πίνακα συμβολών ,στο αντίστοιχο scope.

Βοηθητικές Συναρτήσεις.

addEntity() : προσθήκη ενός νέου entity στον πίνακα συμβολών σε συγκεκριμένο scope που λαμβάνει σαν όρισμα . Καλείται κάθε φορά που δημιουργούμε μια μεταβλητή προσωρινή η κανονική ,αλλά και όταν θέλουμε να εισάγουμε μια συνάρτηση

addScope(): Προσθήκη νέου scope, δημιουργούμε ένα κενό εγγράφημα δραστηριοποίησης για το οποίο θα προσθέτουμε entities μέσα σε αυτό. Αυξάνει τον αριθμό του scope για να γνωρίζουμε που σε ποιο scope βρισκόμαστε. Η συνάρτηση αυτή καλείται κάθε φορά που ξεκινάει η μεταγραφή μια συνάρτησης η διαδικασίας και πρέπει να δημιουργήσουμε χώρο στη μνήμη για αυτές.

deleteScope() : Αν υπάρχει διαθέσιμο scope και είμαστε στο τέλος της μετάφρασης μια συνάρτησης ή διαδικασία τότε πρέπει να διαγράψουμε το scope της συνάρτησης που ολοκληρώσαμε την μετάφραση και να επιστρέψουμε στο προηγούμενο εγγράφημα δραστηριοποίησης. Με το που κάνουμε διαγραφή κάποιου scope διαγράφουμε και τα entities που έχει μέσα αυτό εφοσον τα δεν θέλουμε πλέον μετά την μετάφραση τους. Για παράδειγμα αν έχω ένα scope για μια συνάρτηση περιμένω να γίνει μετάφραση και διαγραφή των scope όλων των υποσυναρτήσεων της συνάρτησης και στο τέλος διαγραφώ το scope της.

getScopeNum(): συνάρτηση για να μου δώσει τον αριθμό του τρέχον scope εκείνη τη στιγμή της μετάφρασης που το χρειάζομαι.

udateFields(): Για την συμπλήρωση των πεδίων κατά τη δημιουργία ενός αντικειμένου function η procedure ,εφοσον όταν ξεκινούμε την μετάφραση μιας διαδικασίας η συνάρτησης δεν ξέρουμε το χώρο μνήμης που θα καταναλώσει για να βάλει στο πίνακα τα entities ,αρά θέλουμε αυτή τη συνάρτηση να την καλέσουμε λίγο πριν τελειώσουμε με το scope της οπού εκείνη τη στιγμή θα γνωρίζουμε το χώρο της μνήμης που χρησιμοποίησε. Αρά σκοπός της συνάρτησης είναι να ενημερώνει τα πεδία του entity που παίρνει σαν όρισμα, το οποίο είναι ένα function ή procedure.

SearchEntity() : Αυτή η συνάρτησης χρησιμοποιείται για να διατρέξει τον πίνακα συμβολών και να βρει τα entity μέσα σε αυτόν. Έχει σαν όρισμα το όνομα του entity που ψάχνει.

SearchEntityScope():Αυτή η συνάρτησης χρησιμοποιείται για να διατρέξει τον πίνακα συμβολών και να βρει το scope των entity μέσα σε αυτόν. Έχει σαν όρισμα το όνομα του entity που ψάχνει. Την χρησιμοποιούμε όταν θέλουμε το επίπεδο κάποιου entity που γνωρίζουμε ότι υπάρχει.

addVariable(): Η συνάρτηση αυτή χρησιμοποιείται για να δημιουργήσει μια μεταβλητή και να την εισάγει μέσα στο πίνακα συμβολών. Καλείται μόνο στα σημεία που θέλουμε να κατασκευάσουμε μεταβλητές. Στο κομμάτι του parser για τα declarations.Τέλος τοποθετεί τη μεταβλητή στο scope με κλήση της addEntity.

addTemporaryVariable() : Η συνάρτηση αυτή χρησιμοποιείται για να δημιουργήσει μια προσωρινή μεταβλητή. Καλείται μόνο εκεί που κατασκευάζονται προσωρινές μεταβλητές μέσα στο κομμάτι του συντακτικού κώδικα. Τέλος τοποθετεί τη μεταβλητή στο scope με κλήση της addEntity.

Πέμπτη Φάση Τελικός Κώδικας

Η τελευταία φάση της παραγωγής κώδικα είναι η παραγωγή του τελικού κώδικα. Ο τελικός κώδικας προκύπτει από τον ενδιάμεσο κώδικα με τη βοήθεια του πίνακα συμβόλων. Συγκεκριμένα, από κάθε εντολή ενδιάμεσου κώδικα προκύπτει μία σειρά εντολών τελικού κώδικα, η οποία για να παραχθεί ανακτά πληροφορίες από τον πίνακα συμβόλων. Για την τελική γλώσσα θα παράξουμε κώδικα σε assembly του επεξεργαστή RISK-V.

Βοηθητικές Συναρτήσεις

• gnlvcode(): δημιουργεί τελικό κώδικα για την προσπέλαση πληροφορίας που βρίσκεται αποθηκευμένη στο εγγράφημα δραστηριοποίησης κάποιου προγόνου της συνάρτησης ή της διαδικασίας που αυτή τη στιγμή μεταφράζεται

Ειδικότερα αυτή η βοηθητική συνάρτηση παράγει τελικό κώδικα για την προσπέλαση μεταβλητών ή διευθύνσεων που είναι αποθηκευμένες σε κάποιο εγγράφημα δραστηριοποίησης διαφορετικό από της συνάρτησης που αυτή τη στιγμή μεταφράζεται.

Η συνάρτηση αυτή παίρνει δύο ορίσματα στην δικιά μας υλοποίηση με το πρώτο να είναι το όνομα της μεταβλητής που ψάχνουμε σε κάποιο διαφορετικό εγγράφημα δραστηριοποίησης. Ψάχνουμε με το όνομα που περάσαμε σαν όρισμα το entity μέσα στον πίνακα συμβόλων και αν δεν το βρούμε σταματάμε την εκτέλεση του μεταφραστή. Στην περίπτωση που το βρούμε, αποθηκεύουμε σε μία μεταβλητή το offset του. Στην συνέχεια με την searchEntityScope() παίρνουμε το αντίστοιχο scope που έχει η μεταβλητή που ψάξαμε πριν και αν αυτή δεν είναι global (Αν είναι global υπάρχει λάθος διότι αυτή η συνάρτηση καλείται μόνο όταν θέλουμε να προσπελάσουμε μεταβλητές σε μικρότερο scope από αυτό που βρισκόμαστε), παράγουμε τον τελικό κώδικα για να ανεβούμε όσα scope χρειαστεί για να βρούμε την μεταβλητή που ψάχνουμε. Όταν φτάσουμε στο scope που θέλουμε προσθέτουμε στον καταχωρητή t0 το offset της μεταβλητής για την οποία καλέσαμε την συνάρτηση. Ο κώδικας φαίνεται παρακάτω:

```
def gnlvcode(name,quad):
    #1) search the verible in table
    x = searchEntity(name)
    #if cant find the veriable then x =none
    if(x== None):
        print("Error that veriable "+ name +" counld't find in symbol
table")
        exit()

    offsetV = 0 - x.offset

    #2) find the level that the veriable has been found
    level = searchEntityScope(name) #level of entity scope tha prepei
na anebei mexri to epipedo level
    if (level >= 1 ): # den einai global auth pou psaxnw
        produce(quad.label,"lw","t0","-4(sp)","")

        scopeNum = getScopeNum() # se poio scope eimai auto einai to
trexon scope
        hops = scopeNum - level # hops is hops that i must do to reach
the variable
        if(scopeNum > level):

            for i in range(hops-1):
                produce(quad.label,"lw","t0","-4(t0)","")
        #3) prosethese to offset gia na breis thn metablhth pou 8es
        produce(quad.label,"addi","t0","t0",str(offsetV))
```

• loadvr(): παράγει τελικό κώδικα ο οποίος διαβάζει μία μεταβλητή που είναι αποθηκευμένη στη μνήμη και την μεταφέρει σε έναν καταχωρητή.

Η loadvr() είναι η συνάρτηση η οποία παράγει τον κώδικα για να διαβαστεί η τιμή μιας μεταβλητής από τη μνήμη, δηλαδή από μία θέση στη στοίβα, και να μεταφερθεί σε έναν καταχωρητή. Η συγκεκριμένη συνάρτηση παίρνει τρία ορίσματα, το πρώτο είναι το name της μεταβλητής που θέλουμε να κάνουμε load, το δεύτερο είναι το όνομα του καταχωρητή στο οποίο θα φορτώσουμε την μεταβλητή και το τρίτο όρισμα είναι το quad το οποίο σχετίζεται με την μεταβλητή που περάσαμε. Εδώ διακρίνουμε κάποιες περιπτώσεις για το τι είδος μεταβλητή είναι αυτή που θα ασχοληθεί η load. Αρχικά, αν η μεταβλητή μας είναι σταθερός αριθμός και θέλουμε να την φορτώσουμε στον καταχωρητή που έχουμε σαν όρισμα παράγουμε το παρακάτω κώδικα:

```
produce(quad.label,"li" , regName , vName ,"")
```

Στην περίπτωση που η μεταβλητή είναι global τότε παράγουμε κώδικα που θα φορτώνουμε στον καταχωρητή που περάσαμε σαν όρισμα τη διεύθυνση του global καταχωρητή που αναφέρεται στην αντίστοιχη μεταβλητή

```
produce(quad.label,"lw", regName, str(offsetV)+"(gp)", "")
```

Για την περίπτωση που η μεταβλητή θα είναι local δηλαδή θα είμαστε στο τρέχον scope ή μεταβλητή μας είναι προσωρινή θα παράγουμε τελικό κώδικα που θα φορτώνει στον καταχωρητή t0 (συνήθως) τον stackPointer με αρνητικό offset για να βρούμε την διεύθυνση της ζητούμενης μεταβλητής

```
produce(quad.label,"lw", regName, str(offsetV)+"(sp)", "")
```

Η περίπτωση όπου η μεταβλητή που θέλουμε να φορτώσουμε είναι παράμετρος και έχει περαστεί με αναφορά στη συνάρτηση που είμαστε τότε παράγουμε για τελικό κώδικα ακριβώς το ίδιο με την προηγούμενη περίπτωση.

Όταν η παράμετρος έχει περαστει με αναφορά φορτώνουμε στο t0 την διεύθυνση της ζητούμενης μεταβλητής από τον stackPointer και στην συνέχεια φορτώνουμε τον καταχωρητή που περάσαμε σαν όρισμα την τιμή του t0.

```
produce(quad.label,"lw","t0", str(offsetV)+"(sp)", "")
produce(quad.label,"lw",regName,"(t0)", "")
```

Μία άλλη περίπτωση είναι η μεταβλητή που θέλουμε να φορτώσουμε να έχει περαστεί με τιμή σε πρόγονο συνάρτηση ή να είναι τοπική μεταβλητή σε πρόγονο συνάρτηση. Τότε θα καλέσουμε την gnlvcode() για να βρούμε την ζητούμενη μεταβλητή της προγόνου συνάρτησης και στην συνέχεια εφόσον έχουμε στον t0 την

τιμή – διεύθυνσή της στον t0, θα φορτώσουμε στο καταχωρητή που έχουμε περάσει σαν όρισμα το περιεχόμενο του t0.

```
gnlvcode(v.name,quad)
produce(quad.label,"lw",regName,"(t0)", "")
```

Όταν η μεταβλητή που θέλουμε να φορτώσουμε έχει περαστεί σε πρόγονο με αναφορά τότε βρίσκουμε από την gnlvcode() τη διεύθυνσή της, την φορτώνουμε στον t0, στην συνέχεια απανωγράφουμε με την διευθυνσή της τον t0 και τέλος φορτώνουμε στον καταχωρητή που περάσαμε σαν όρισμα το περιεχόμενο του t0.

```
gnlvcode(v.name,quad)
produce(quad.label,"lw","t0","(t0)","")
produce(quad.label,"lw",regName,"(t0)","")
```

• storerv(): κάνει την αντίστροφη διαδικασία από το loadvr, παράγει τελικό κώδικα ο οποίος αποθηκεύει στη μνήμη την τιμή μιας μεταβλητής η οποία βρίσκεται σε έναν καταχωρητή. Αρά για τις περισσότερες περιπτώσεις δεν αλλάζουμε τίποτα για τον τελικό πέρα από την γραμμή του τελικού οπού θέλουμε να αποθηκεύσουμε αντί να φορτώσουμε τις μεταβλητές μας.

Για την περίπτωση που θέλουμε να αποθηκεύσουμε κάποια σταθερά σε έναν καταχωρητή καλούμε την loadrv() με το όνομα της σταθερά , τον t0 και το αντίστοιχο quad και στη επόμενη εντολή έχουμε την storerv() με την οποία αποθηκεύουμε την αντίστοιχη τιμή του t0 στον καταχωρητή που έχουμε σαν όρισμα.

```
loadvr(vName,"t0",quad)
storerv("t0",regName,quad)
```

Για την περίπτωση που θέλουμε να αποθηκεύσουμε μια global μεταβλητή, να αποθηκεύσουμε μια local μεταβλητή της συνάρτησης που είμαστε ,ή να κανουμε store μεταβλητές που έχουν περαστεί με αναφορά ή με τιμή, παράγουμε τον τελικό κώδικα ίδιο με της αντίστοιχες περιπτώσεις που είχαμε στη loadrv με την μονή διαφορά ότι εδώ αντί για lw στην τελική εντολή έχουμε sw.

1)

```
produce(quad.label,"sw", regName, str(offsetV)+"(gp)", "")
```

2),3)

```
produce(quad.label,"sw", regName, str(offsetV)+"(sp)", "")
```

4)

```
produce(quad.label,"lw","t0", str(offsetV)+"(sp)", "")
produce(quad.label,"sw",regName,"(t0)","")
```

Όταν έχουμε να ασχοληθούμε με μεταβλητές που έχουν περαστεί με τιμή σε έναν προγονό τότε με την gnlvcode() ψάχνουμε να βρούμε την θέση τους την οποία και αποθηκεύουμε στο t0 και στην συνέχεια κάνουμε store τον t0 στο καταχωρητή που έχουμε σαν όρισμα.

```
gnlvcode(v.name,quad)
produce("sw",regName,"(t0)","","")
```

Όταν έχουμε να ασχοληθούμε με μεταβλητές που έχουν περαστεί με αναφορά σε έναν προγονό τότε πρέπει με την κλήσης της gnlvcode() να βρούμε την διεύθυνση της μεταβλητής που θέλουμε να αποθηκεύσουμε και να τη βάλουμε στον t0,στη συνέχεια να απανωγράψουμε τον t0 με τη διεύθυνση του t0 που έδωσε σαν αποτέλεσμα η gnlvcode και τέλος πρέπει να αποθηκεύσουμε τον t0 στον καταχωρητή που έχει περαστεί σαν όρισμα στη συνάρτηση storevr().

Αντικειμενοστραφή υλοποίηση για τον final:

Για να παράγουμε τελικό κώδικα καλούμε μια συνάρτηση produce μέσα στον συντακτικό αναλυτή σε σημεία που παράγεται ενδιάμεσος κώδικας και άλλες φορές σε σημεία που έχουμε σημειώσει τα quads από το quadlist με όλες τις εντολές του ενδιάμεσου . Τα σημεία που καλούμε την produce θα τα εξηγήσουμε στη συνέχεια. Η produce είναι μια βοηθητική συνάρτηση που καλείτε για να πράξει τις τελικές μονάδες του κώδικα . Αρά έχουμε δημιουργήσει μια κλάση Finalobj με την οποία αποθηκεύουμε κάθε εντολή τελικού κώδικα σε μια δομή. Τα αντικείμενα finalobj δημιουργούνται μόνο στο produce και σε κανένα άλλο σημείο του κώδικα μέσα στην συντακτική ανάλυση. Επίσης έχουμε μια λίστα με όλες τις γραμμές του τελικού κώδικα (η λίστα περιέχει όλα τα Finalobj) για να μας βοηθήσει στη συνέχεια να παράγουμε το αρχείο που ζητείται για τον τελικό. Η κλάση που κατασκευάζει τα αντικείμενα είναι η ακόλουθη και ο σκοπός της δημιουργίας της είναι να έχουμε συμπυκνωμένη πληροφορία για τις τελικές γραμμές του τελικού. Για το τελικό οπού θέλουμε να αρχίσουμε την main μετρά το j main θα χρησιμοποιήσουμε την produceMain η οποία κάνει την ιδιά δουλειά με την produce μόνο που τώρα έχουμε σημειωμένα τα object του τελικού με το mainstatus πεδίο (ότι δηλαδή ‘’ναι’’ είναι τμήματα της main).

Άλλες βοηθητικές συναρτήσεις είναι:

Τρεις συναρτήσει backpatch τις οποίες χρησιμοποιούμε σε αντίστοιχα σημεία ,όπου και στον ενδιάμεσο κάνουμε backpatch, διότι με την δική μας υλοποίηση όταν προσπαθούσαμε να πράξουμε τελικό εκεί που δημιουργούσαμε quads, αρά δεν

είχαμε συμπληρωμένες εκείνη τη στιγμή τις τετράδες μας, κάναμε παραγωγή του τελικού μας και στη συνέχεια γεμίζαμε τα πεδία των finalObj με τις αντίστοιχες backpatch . Για παράδειγμα όταν πρέπει να κάνουμε backpatch στη συνάρτηση της if τότε ακριβώς από κάτω κάνουμε backpatch και τα finalobjs μας που έχουν δημιουργηθεί όταν συγκρίναμε τα expressions από την boolFactor(). Τα backpatch γίνονται στην ιδιά λίστα με τα κανονικά backpatch του ενδιάμεσου κώδικα για να έχουμε σωστά ολοκληρωμένα και τα quad και τα finalObj

Ένας άλλος τρόπος να γίνει αυτό ήταν να μαρκάρουμε από το γενικό QuadList τις τετράδες που ασχολούνται με τα quad που μας αφορούσαν κάθε φορά και να παράγουμε τελικό κάθε φορά που τελειώνουμε ένα scope.Όταν τελειώναμε το scope έχουμε συμπληρώσει ήδη τις τετράδες αρά δεν χρειάζεται κάποιο backpatch (αυτό ο τρόπος προτείνατε στο μάθημα αλλά δεν τον ακολουθήσαμε σε όλο το κομμάτι του τελικού)

Οι Τρεις παραπάνω backpatch είναι οι εξης:

backpatchTrueTeliko(trueList,nextQuad) : κάνει backpatch με το trueList σε finalObj που είναι κενά με quad label το αντίστοιχο που πρέπει να συμπληρωθούν.

backpatchfalseTeliko(falseList,nextQuad) : κάνει backpatch με το flaseList σε finalObj που είναι κενά με quad label το αντίστοιχο που πρέπει να συμπληρωθούν.

backpatchMain(framelength) : συμπληρώνει στο τέλος της main το μέγεθος που έχει η main σε μνήμη. (Για να κάνουμε την addi sp sp framelength)

Μια ακόμη βοηθητική συνάρτηση είναι η productFinal() η οποία καλείται σε καταλληλά σημεία στον συντακτικό αναλυτή εκεί που παράγονται οι τετράδες για να δημιουργηθούν ο τελικός κώδικας. Σε αυτό το σημείο συναντήσαμε τη δυσκολία με τις κάνες εντολές του ενδιάμεσου και χρειάστηκαν οι συναρτήσεις backpatch που προηγούμενος αναλύσαμε.

Ακόμη έχουμε μια productfinalForCalls():

Η συνάρτηση αυτή δουλεύει με τον άλλο τρόπο που εξηγήσαμε προηγουμένως. Δηλαδή μαρκάρουμε τα quad που δεσμεύονται να πράξουν ενδιάμεσο για την κλήση συναρτήσεων τα οποία θα χρειαζόντουσαν packpatch για να συμπληρωθούν στο τέλος του call. Στα ανάλογα σημεία καλούμε την produce για να παράγουμε τον τελικό. Και εδώ έχουμε περιπτώσεις για τα περάσματα μεταβλητών με τιμή , με αναφορά , αν είναι global ,αν είναι με τιμή από συνάρτηση προγονό ή αν είναι με αναφορά από συνάρτηση προγονό.

Έχουμε μια συνάρτηση : produceFinalForSubBlocks(list):

Η συνάρτηση αυτή έχει σαν όρισμα μια λίστα την οποία την δημιούργει ο parser στον οποίο την καλούμε. Αυτή η συνάρτηση παράγει τον κώδικα με τον οποίο πρέπει να φορτώσουμε στον ra τον sp για την έξοδο από τη συνάρτηση και να παράγουμε την

τελική γραμμή jr ra για να βγούμε εκτός της συνάρτησης. Η περιγράφουσα συνάρτηση καλείται όταν έχουμε τελειώσει με την μετάφραση της συνάρτησης.

Συνάρτηση productFinalBegin():

Τέλος έχουμε μια ακόμη συνάρτηση την productFinalBegin με την οποια δημιουργούμε τον τελικό για την αρχή κάθε συνάρτησης ή διαδικασίας, η οποία παράγει το sw ra (sp) για να κάνουμε αποθήκευση του ra στον sp και να μπούμε στη συνάρτηση. Η συνάρτηση αυτή καλείται όταν ξεκινάμε ένα block συνάρτησης.

Τέλος παράγουμε ένα αρχείο που μας ζητείται με κατάλληλο format για να τρέξει στον rars την τελείται συνάρτηση printList(): Η οποία διατρέχει όλη τη λίστα με τα παραγόμενα τελικά αντικείμενα. Παρακάτω φαίνεται το υλοποιούμενο format

```
L0:     j main
L1:     sw ra (sp)
L2:     lw t0 -4(sp)
        lw t0 -4(t0)
        addi t0 t0 -28
        lw t0 (t0)
        sw t0 -12(sp)
L3:     lw ra (sp)
        jr ra
L4:     sw ra (sp)
L5:     li t0 1
        sw t0 -12(sp)
L6:     lw ra (sp)
        jr ra
....
```