Πολυτεχνική Σχολή
Τμήμα Μηχανικών Η/Υ και Πληροφορικής

ΜΕΤΑΦΡΑΣΤΕΣ

Ζουγιανός Γεώργιος 2442

Τζιόλας Γεώργιος 2591

Μάθημα: Μεταφραστές (ΜΥΥ 802)

Ιστοσελίδα Μαθήματος: http://ecourse.uoi.gr/course/view.php?id=543

Υπεύθυνος Καθηγητής: Γιώργος Μανής (www.cs.uoi.gr/~manis/)

Ακαδ. Έτος: 2019 (Εαρινό)

Πίνακας περιεχομένων

1. Εισαγωγή

Περιγραφή μαθήματος

Το μάθημα έχει ως σκοπό να δώσει στους φοιτητές τις βασικές γνώσεις της θεωρίας των μεταφραστών. Αποτελείται από το θεωρητικό μέρος, το οποίο διδάσκεται μέσα από τις διαλέξεις που γίνονται στο αμφιθέατρο, αλλά και το εργαστηριακό μέρος, κατά το οποίο οι φοιτητές καλούνται να υλοποιήσουν τον μεταφραστή μιας εκπαιδευτικής γλώσσας προγραμματισμού. Η εκπαιδευτική αυτή γλώσσα περιέχει πλούσια στοιχεία και η κατασκευή του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον, αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται και σε άλλες γλώσσες, αλλά και κάποιες πρωτότυπες. Η γλώσσα περιέχει συναρτήσεις και διαδικασίες, μετάδοση παραμέτρων με αναφορά και τιμή, αναδρομικές κλήσεις, φώλιασμα στη δήλωση συναρτήσεων και διαδικασιών κ.α.

Περιγραφή Εργασίας (πρότζεκτ)

Ο τελικός στόχος της εργασίας είναι η υλοποίηση ενός μεταφραστή μιας γλώσσας προγραμματισμού της οποίας η σύνταξη και η σημασία των εντολών δίνονται από τον καθηγητή. Οι φοιτητές καλούνται να υλοποιήσουν τον μεταφραστή στη γλώσσα **Python**. Η εργασία χωρίζεται σε 5 φάσεις με την εξής σειρά (στις παρενθέσεις αναγράφεται το βάρος στον τελικό βαθμό):

1. Λεκτικός και Συντακτικός Αναλυτής (10%)
2. Παραγωγή Ενδιάμεσου Κώδικα (30%)
3. Σημασιολογική Ανάλυση και Πίνακας Συμβόλων (10%)
4. Παραγωγή Τελικού Κώδικα (30%)
5. Παράδοση Αναφοράς (20)

2. Η Γλώσσα Προγραμματισμού Starlet

Περιγραφή

Η Startlet είναι μια μικρή γλώσσα προγραμματισμού φτιαγμένη με βάση τις ανάγκες της προγραμματιστικής άσκησης του μαθήματος. Παρόλο που οι προγραμματιστικές της ικανότητες είναι μικρές, η εκπαιδευτική αυτή γλώσσα περιέχει πλούσια στοιχεία και η κατασκευή του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον, αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται από άλλες γλώσσες, καθώς και κάποιες πρωτότυπες. Η υποστηρίζει συναρτήσεις. μετάδοση παραμέτρων με αναφορά, τιμή και αντιγραφή, αναδρομικές κλήσεις και άλλες ενδιαφέρουσες δομές. Επίσης, επιτρέπει φώλιασμα στη δήλωση συναρτήσεων κάτι που λίγες γλώσσες υποστηρίζουν (το υποστηρίζει η Pascal, δεν το υποστηρίζει η C).).

Από την άλλη όμως πλευρά, η δεν υποστηρίζει βασικά προγραμματιστικά εργαλεία όπως η δομή for, ή τύπους δεδομένων όπως οι πραγματικοί αριθμοί και οι συμβολοσειρές. Οι παραλήψεις αυτές έχουν γίνει ώστε να απλουστευτεί η διαδικασία κατασκευής του μεταγλωττιστή, μία απλούστευση όμως που έχει να κάνει μόνο με τη μείωση των γραμμών κώδικα και όχι με τη δυσκολία κατασκευής του ή την εκπαιδευτική αξία της άσκησης.

Λεκτικές μονάδες

Το αλφάβητο της αποτελείται από:

1. τα μικρά και κεφαλαία γράμματα της λατινικής αλφαβήτου («A»,...,«Z» και «a»,...,«z»),
2. τα αριθμητικά ψηφία («0»,...,«9»)
3. τα σύμβολα των αριθμητικών πράξεων («+», «-», «*», «/»)
4. τους τελεστές συσχέτισης «<», «>», «=», «<=», «>=», «<>»
5. το σύμβολο ανάθεσης «:=»
6. τους διαχωριστές («;», «,», «:»)
7. τα σύμβολα ομαδοποίησης («(»,«)»,«[»,«]»)
8. Τους χαρακτήρες διαχωρισμού σχολίων («/*»,«*/»,«//»).Δ

Τα σύμβολα «[» και «]» χρησιμοποιούνται στις λογικές παραστάσεις όπως τα σύμβολα «(» και «)» στις αριθμητικές παραστάσεις. Μερικές λέξεις είναι δεσμευμένες:

1. program, endprogram
2. declare
3. if then else endif
4. while endwhile dowhile enddowhile
5. loop, endloop, exit
6. forcase, endforcase, incase, endincase, when, default, enddefault
7. function, endfunction, return, in, inout, inandout
8. and, or, not
9. input, print

Οι λέξεις αυτές δεν μπορούν να χρησιμοποιηθούν ως μεταβλητές. Οι σταθερές της γλώσσας είναι ακέραιες σταθερές που αποτελούνται από προαιρετικό πρόσημο και από μία ακολουθία αριθμητικών ψηφίων. Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία, αρχίζοντας όμως από γράμμα. Ο μεταγλωττιστής λαμβάνει υπόψη του μόνο τα τριάντα πρώτα γράμματα. Οι λευκοί χαρακτήρες (tab, space, return)) αγνοούνται και μπορούν να χρησιμοποιηθούν με οποιονδήποτε τρόπο χωρίς να επηρεάζεται η λειτουργία του μεταγλωττιστή, αρκεί βέβαια να μην βρίσκονται μέσα σε δεσμευμένες λέξεις, αναγνωριστικά, σταθερές. Το ίδιο ισχύει και για τα σχόλια, τα οποία πρέπει να βρίσκονται μέσα στα σύμβολα /* και */ ή να βρίσκονται μετά το σύμβολο // και ως το τέλος της γραμμής. Απαγορεύεται να ανοίξουν δύο φορές σχόλια, πριν τα πρώτα κλείσουν. Δεν υποστηρίζονται εμφωλευμένα σχόλια.

Τύποι και δηλώσεις μεταβλητών

Ο μοναδικός τύπος δεδομένων που υποστηρίζει η είναι οι ακέραιοι αριθμοί. Οι ακέραιοι αριθμοί πρέπει να έχουν τιμές από –32767 έως 32767. Η δήλωση γίνεται με την εντολή **declarations**. Ακολουθούν τα

ονόματα των αναγνωριστικών χωρίς καμία άλλη δήλωση, αφού γνωρίζουμε ότι πρόκειται για ακέραιες μεταβλητές και χωρίς να είναι αναγκαίο να βρίσκονται στην ίδια γραμμή. Οι μεταβλητές χωρίζονται μεταξύ τους με κόμματα. Το τέλος της δήλωσης αναγνωρίζεται με το ελληνικό ερωτηματικό. Επιτρέπεται να έχουμε περισσότερες των μία συνεχόμενες χρήσεις της **declarations**.

Τελεστές και εκφράσεις

Η προτεραιότητα των τελεστών από τη μεγαλύτερη στη μικρότερη είναι:

1. Μοναδιαίοι λογικοί: «not»
2. Πολλαπλασιαστικοί: «*», «/»
3. Μοναδιαίοι προσθετικοί: «+», «-»
4. Δυαδικοί προσθετικοί: «+», «-»
5. Σχεσιακοί «=», «<», «>», «<>», «<=», «>=»
6. Λογικό «and»
7. Λογικό «or»

Γραμματική

<program> ::= program id <block> endprogram

<block> ::= <declarations> <subprograms> <statements>

<declarations> ::= (declare <varlist>;)*

<varlist> ::= ε | id (, id)*

<subprograms> ::= (<subprogram>)*

<subprogram> ::= function id <funcbody> endfunction

<funcbody> ::= <formalpars> <block>

<formalpars> ::= (<formalparlist>)

<formalparlist> ::= <formalparitem> (, <formalparitem>)* | ε

<formalparitem> ::= in id | inout id | inandout id

<statements> ::= <statement> (; <statement>)*

<statement> ::= ε |
<assignment-stat> |
<if-stat> |
<while-stat> |
<do-while-stat> |
<loop-stat> |

<exit-stat> |

<forcase-stat> |

<incase-stat> |

<return-stat> |

<input-stat> |

<print-stat>

<assignment-stat> ::= id := <expression>

<if-stat> ::= if (<condition>) then <statements> <elsepart> endif

<elsepart> ::= ε | else <statements>

<while-stat> ::= while (<condition>) <statements> endwhile

<do-while-stat> ::= dowhile <statements> enddowhile (<condition>)

<loop-stat> ::= loop <statements> endloop

<exit-stat> ::= exit

<forcase-stat> ::= forcase

(when (<condition>) : <statements>)*

default: <statements> enddefault

endforcase

<incase-stat> ::= incase

(when (<condition>) : <statements>)*

endincase

<return-stat> ::= return <expression>

<print-stat> ::= print <expression>

<input-stat> ::= input id

<actualpars> : := (<actualparlist>)

<actualparlist> ::= <actualparitem> (, <actualparitem>)* | ε

<actualparitem> ::= in <expression> | inout id | inandout id

<condition> ::= <boolterm> (or <boolterm>)*

<boolterm> ::= <boolfactor> (and <boolfactor>)*

<boolfactor> ::= not [<condition>] | [<condition>] |

<expression> <relational-oper> <expression>

<expression> ::= <optional-sign> <term> (<add-oper> <term>)*

<term> ::= <factor> (<mul-oper> <factor>)*

<factor> ::= constant | (<expression>) | id <idtail>

<idtail> ::= ε | <actualpars>

<relational-oper> ::= = | <= | >= | > | < | <>

<add-oper> ::= + | -

<mul-oper> ::= * | /

<optional-sign> ::= ε | <add-oper>

3. Εισαγωγή Υλοποιήσης

Ο μεταφραστής, υλοποιημένος σε γλώσσα Python είναι ένα μόνο αρχείο όπως ζητήθηκε από τον καθηγητή. Έπειτα, η μεταγλώττιση ενός προγράμματος πραγματοποιείται περνώντας το όνομα του αρχείου .slt με το πρόγραμμα Starlet ως όρισμα στο script (.py) του μεταφραστή. Το python αρχείο του μεταφραστή ενδεικτικά ονομάζεται startlet_compiler.py που ακόμα και να μετονομαστεί θα είναι ικανό να μεταφράσει ένα πρόγραμμα Starlet. Ένα παράδειγμα χρήσης του μεταφραστή και μεταγλώττισης ενός προγράμματος Starlet είναι:

python startlet_compiler.py my_startlet_program.slt

Στην παρούσα αναφορά γίνεται η περιγραφή των επιμέρους εργαλείων που χρησιμοποιούνται και απαιτούνται για τη μετάφραση ενός προγράμματος.

4. Λεκτικός Αναλυτής

Θεωρία

Καλείται ως **συνάρτηση** από το συντακτικό αναλυτή και διαβάζει **γράμμα-γράμμα** το πηγαίο πρόγραμμα που δόθηκε για μετάφραση. Κάθε φορά που καλείται επιστρέφει την επόμενη **λεκτική μονάδα.** Επιστρέφει στο συντακτικό αναλυτή έναν ακέραιο που χαρακτηρίζει τη λεκτική μονάδα και τη λεκτική μονάδα.

Υλοποίηση

Δημιουργούμε την κλάση **Lex** η οποία έχει σαν πεδία όλο το περιεχόμενο του αρχείου που θέλουμε να μεταφράσουμε, το τρέχων index και τη τρέχων γραμμή. Έπειτα δημιουργούμε τη μέθοδο get() με σκοπό να επιστρέφουμε την επόμενη λεκτική μονάδα στον συντακτικό αναλυτή. Αν ο συντακτικός αναλυτής καλέσει την get() ενώ το τρέχων index είναι μεγαλύτερο από το μέγεθος του περιεχομένου του αρχείου, σημαίνει ότι ο συντακτικός αναλυτής περιμένει λέξη, αλλά ο λεκτικός δεν βρίσκει. Συνεπώς, λαμβάνουμε ένα μήνυμα σφάλματος του τύπος «There are no more words to get.» και ο μεταφραστής τερματίζει.

Όταν καλείται η get(), ο λεκτικός αναλυτής ξεκινάει να διαβάζει από το τρέχων index. Όσο βρίσκει χαρακτήρες τους οποίους πρέπει να αγνοεί (χαρακτήρες κενού RETURN, SPACE, TAB), το τρέχων index αυξάνεται χωρίς να γίνει επιστροφή τιμής και ελέγχει τον επόμενο χαρακτήρα. Σε περίπτωση που βρει μη κενό χαρακτήρα αναγνωρίζει τι τύπου είναι η λεκτική μονάδα, και τη διαβάζει. Πιο συγκεκριμένα, υπάρχουν οι εξής περιπτώσεις που γίνονται αποδεκτές:

- Αν ο χαρακτήρας είναι αλφαβητικό, το κρατάει και στη συνέχεια κρατάει και διαβάζει τα επόμενα αλφαβητικά ή αριθμούς μέχρι να βρει οτιδήποτε άλλο. Τέλος επιστρέφει το σύνολο (τη λέξη που έφτιαξε από τα αλφαβητικά ή αριθμούς που διάβασε).
- Αν ο χαρακτήρας είναι αριθμός, τον κρατάει και στη συνέχεια κρατάει και διαβάζει τους επόμενους αριθμούς μέχρι να βρει οτιδήποτε άλλο. Τέλος πριν επιστρέφει τον αριθμό ως λέξη (String), τον μετατρέπει σε αριθμό (integer) και ελέγχει αν η τιμή του είναι μεταξύ -32767 και 32767. Αν δεν είναι,

λαμβάνουμε μήνυμα σφάλματος του τύπου «Arithmetic constants must be between -32767 and 32767.»
- Αν ο χαρακτήρας είναι κάποιος τελεστής αριθμητικής πράξης αλλά όχι ο τελεστής διαίρεσης (**/**), το επιστρέφει.
- Αν ο χαρακτήρας είναι ο «<», διαβάζει τον επόμενο και αν είναι «=», θα επιστρέψει «<=». Ειδάλλως θα μειώσει το τρέχων index κατά 1 (για να μην χάσουμε το χαρακτήρα που έπρεπε να ελέγξουμε αν είναι «=») και θα επιστρέψει «<».
- Το ίδιο, αν ο χαρακτήρας είναι ο «>» ή ο «:» (πιθανή εκχώρηση τιμής)
- Αν ο χαρακτήρας είναι ο χαρακτήρας διαίρεσης «/», θα ελέγξει αν ο επόμενος είναι «*» ή ένα δεύτερο σύμβολο διαίρεσης «/». Αν ισχύει οτιδήποτε από αυτά σημαίνει ότι διαβάζει σχόλια. Στην περίπτωση που ο δεύτερος χαρακτήρας είναι «*», προσπερνάει όλους τους χαρακτήρες μέχρι να συναντήσει τους χαρακτήρες «*» και «/» διαδοχικά, οπού σημαίνει τέλος σχολίου. Στην περίπτωση που ο δεύτερος χαρακτήρας είναι «/», προσπερνάει όλους τους χαρακτήρες μέχρι να βρει τον χαρακτήρα νέας γραμμής «\n». Αν δεν ισχύει τίποτα από αυτά, επιστρέφει τον χαρακτήρα διαίρεσης.
- Αν ο χαρακτήρας είναι παρένθεση, αγκύλη, ερωτηματικό (semicolon) η κόμμα τον επιστρέφει.
- Τέλος, αν ο χαρακτήρας δεν είναι τίποτα από όσα αναφέρθηκαν παραπάνω λαμβάνουμε μήνυμα λάθους του τύπου «Unknown symbol: <current_character>»

Για την υλοποίηση της get() χρησιμοποιούμε για κάθε κατάσταση μια διαφορετική μέθοδο. Για παράδειγμα, αν είμαστε στη περίπτωση που ο χαρακτήρας είναι αλφαβητικός, καλείται η μέθοδος word() του λεκτικού αναλυτή, για να διαβάσει τη λέξη και να την επιστρέψει.

5. Συντακτικός Αναλυτής

Θεωρία

Ο λειτουργία της συντακτικής ανάλυσης λαμβάνει χώρα για να διαπιστώσουμε εάν το πηγαίο πρόγραμμα ανήκει η όχι στη γλώσσα. Δημιουργεί το κατάλληλο «περιβάλλον» μέσα από το οποίο αργότερα θα κληθούν οι σημαντικές ρουτίνες. Υπάρχουν πολλοί τρόποι για να κατασκευαστεί ένας συντακτικός αναλυτής.

Υλοποίηση

Για την υλοποίηση της συντακτικής ανάλυσης – συντακτικού αναλυτή θα προτιμήσουμε τη συντακτική ανάλυση με αναδρομική κατάβαση η οποία βασίζεται σε γραμματική LL(1). Η γραμματική LL(1) αναγνωρίζει από αριστερά στα δεξιά, την αριστερότερη δυνατή παραγωγή και όταν βρίσκεται σε δίλλημα ποιον κανόνα να ακολουθήσει της αρκεί να κοιτάξει το αμέσως επόμενο σύμβολο στην συμβολοσειρά εισόδου.

Αρχικά, δημιουργούμε ένα αντικείμενο της κλάσης λεκτικού αναλυτή (class Lex) ώστε να είμαστε ικανοί να ελέγχουμε τη σειρά των λεκτικών μονάδων και το κρατάμε σε ένα global πεδίο. Έπειτα υλοποιούμε την μέθοδο lex() η οποία απλά επιστρέφει την τιμή lex.get(). Όταν ξεκινήσει η μετάφραση, θα κληθεί η μέθοδος analyze() η οποία είναι η κύρια μέθοδος που θα εκκινήσει την συντακτική ανάλυση. Κατά την συντακτική ανάλυση, αν ο αναλυτής συναντήσει μια ανεπιθύμητη (σε λάθος σημείο) εμφανίζει το κατάλληλο μήνυμα λάθους που περιέχει ποια λεκτική μονάδα θα έπρεπε να διαβαστεί, το τρέχων index και τη γραμμή του λεκτικού, και τέλος η μετάφραση διακόπτεται και το πρόγραμμα τερματίζει.

Ξεκινώντας λοιπόν τη συντακτική ανάλυση, αρχικοποιούμε το global πεδίο με όνομα token, με τη πρώτη λεκτική μονάδα που επέστρεψε η μέθοδος lex(). Αναμένουμε αυτή η μονάδα να ισούται με τη λέξη «program». Η επόμενη λεκτική μονάδα αναμένουμε να είναι μια λέξη (μη δεσμευμένη από τη γλώσσα) και τη κρατάμε (σε global μεταβλητή), καθώς είναι το όνομα του προγράμματος.

Στη συνέχεια, περνάμε στο κομμάτι των declarations (και μέθοδος στο πρόγραμμα), όπου αν διαβάσουμε τη λέξη **declare**, αναμένουμε να διαβαστούν λέξεις, τα οποία είναι ονόματα μεταβλητών και χωρίζονται με το χαρακτήρα κόμμα («**,**»). Στη τελευταία μεταβλητή που δηλώνεται, απαιτούμε να διαβαστεί ερωτηματικό «**;**», καθώς είναι μέρος της γραμματικής της γλώσσας Starlet. Μετά από αυτό το ερωτηματικό, αναμένουμε το επόμενο **declare** και μεταβλητές οσεσδήποτε φορές. Ένα κομμάτι ψευδοκώδικα που περιγράφει τη λειτουργία ανάλυσης της λεκτικής μονάδας είναι το εξής:

```
Declarations():
 while (token == “declare”)
     token = lex()
     if (isVariable(token))
             token = lex()
             variableName= token
             while (token == “,”)
                     token = lex()
                     if (isVariable(token)
                             variableName= token
                             token = lex()
                     else error
             if (token == “;”)
                     token = lex()
             else error
     else error
```

Μετά το προαιρετικό κομμάτι των declarations αναμένουμε οσεσδήποτε δηλώσεις υποπρογραμμάτων. Η μέθοδος που τις αναλύει, ονομάζεται **subprograms().** Όσο ο λεκτικός δίνει στο συντακτικό τη λέξη **function,** ο συντακτικός «τρώει» την επόμενη λέξη η οποία πρέπει να είναι όνομα μεταβλητής (λέξη) που αντιστοιχεί στην ονομασία της μεθόδου και στη συνέχεια καλείται η **formalpars()** η οποία είναι υπεύθυνη για τη σωστή σύνταξη των παραμέτρων της συνάρτησης. Έπειτα καλείται η **declarations()** καθώς τα υποπρογράμματα δικαιούνται να έχουν τις δικές τους global μεταβλητές. Μετά τη κλήση της, ο αναλυτής επαναλαμβάνει τη διαδικασία με τη κλήση της **subprograms()** διότι η συναρτήσεις είναι ικανές να έχουν τις δικές τους υποσυναρτήσεις. Ένα αντίστοιχο κομμάτι ψευδοκώδικα για την ανάλυση υποπρογραμμάτων είναι:

```
Subprograms():
while (token == “function”)
     functionName = token
     token = lex()
     formalpars()
     declarations()
```

```
    subprograms()
```

όπου:

```
formalpars():
if token == “in” or token == ”inout” or token == “inandout”
    par_type = token
    token = lex
    if (is_variable(token)
        par_name = token
        token = lex
        while (token == “,”)
            token = lex
            if token == “in” or token == ”inout” or token == “inandout”
                …
            else error
    else error
else error
```

Έπειτα της συντακτικής ανάλυσης υποπρογραμμάτων, ο αναλυτής ξεκινάει να αναλύσει τη ροή ενός προγράμματος. Η υπεύθυνη για αυτή τη δουλειά μέθοδος ονομάζεται **sequence().** Η **sequence()** ανάλογα το τρέχων token που επέστρεψε ο λεκτικός αναλυτής, καλεί και την αντίστοιχη μέθοδο. Πιο συγκεκριμένα:

- Αν το token είναι **μεταβλητή**, καλείται η μέθοδος **assignment()** η οποία είναι υπεύθυνη για την ανάλυση εκχώρησης μεταβλητής. Αναμένει το επόμενο token να είναι το σύμβολο εκχώρησης **:=** και έπειτα είτε να υπάρχει κάποια αριθμητική σταθερά η κάποια έκφραση, η οποία αναλύεται από τη μέθοδο **expression().**

- Αν το token είναι **if**, καλείται η μέθοδος **if_statement()** η οποία αναλύει την εντολή if. Αρχικά «τρώει» το token **if** και στη συνέχεια αναμένει μια κατάσταση περιφραγμένη από παρένθεση, της οποίας η ανάλυση γίνεται με τη μέθοδο **parenthesis_condition_parenthesis().** Έπειτα περιμένει τη δεσμευμένη λέξη **then** και καλεί την **sequence().** Όταν τελειώσει το εμφωλευμένο κάλεσμα της **sequence()** αν διαβάσει τη λέξη **else** ξανακαλεί την **sequence().** Τέλος, αναμένει τη λέξη **endif.**

- Αν το token είναι **dowhile**, καλείται η μέθοδος **do_statement()** και αναλύεται η εντολή **dowhile**. Καλεί τον **lex** για να πάρει το επόμενο token και καλεί την **sequence()**. Τέλος περιμένει το token να είναι το **enddowhile**.

- Αν το token είναι **while,** καλείται η μέθοδος **while_statement()** η οποία αναλύει συντακτικά την εντολή while και έχει όμοια μορφή με τη ροή της **do_statement()** με τη μόνη διαφορά ότι στο τέλος περιμένει τη λέξη **endwhile**.

- Αν το token ισούται με **return,** γίνεται η κλήση της **return_statement()** η οποία αφού καλέσει τον lex για να παρθεί το επόμενο token, αναμένει μια έκφραση (καλεί την **expression()).**

- Αν το token είναι **forcase**, καλείται η **forcase_statement()** προκειμένου να γίνει η ανάλυση της εντολής **forcase**. Στη συνέχεια όσες φορές διαβαστεί η λέξη **when** καλείται η **parenthesis_condition_parenthesis().** Μετά αναμένεται το διάβασμα του συμβόλου «**:**» και καλείται η **sequence().** Μετά περιμένει τη λέξη **default** καθώς είναι αναγκαίο η εντολή **forcase** να περιέχει ένα default block το οποίο έχει μέσα εμφωλευμένη **sequence().** Τέλος αναμένει να διαβάσει **endforcase**.

- Αν το token είναι incase, ο αναλυτής ξεκινάει την ανάλυση της εντολής incase. Η ανάλυση της incase γίνεται αναμένοντας οσεσδήποτε καταστάσεις τύπου **when <block>** όπου το κομμάτι **<block>** καλεί την **sequence().** Τέλος είναι αναγκαίο να διαβαστεί η λέξη **endincase**.

- Αν το token είναι **loop**, αναλύεται η εντολή **loop** με τη μέθοδο **loop**_statement**()** η οποία είναι όμοια της **while** και **dowhile**.

- Αν το token είναι **exit**, καλείται η **exit_statement()** που απλά το διαβάζει.

- Αν το token είναι **return**, καλείται η **return_statement()** και γίνεται η ανάλυση της έκφρασης μετά τη λέξη **return.** Για την ανάλυση της έκφρασης, καλείται η **expression().**

- Ομοίως αν το token είναι **print** ή **input**, με τη μέθοδο **print_statement()** και **input_statement()** αντίστοιχα.

- Τέλος, αν το token είναι το σύμβολο semicolon «;», ξαναγίνεται κλήση της sequence για να γίνει συντακτική της επόμενης εντολής.

Άλλες μέθοδοι που καλούνται κατά τη συντακτική ανάλυση ενός προγράμματος και αντιστοιχούν στα αντίστοιχα σύμβολα της γραμματικής της γλώσσας Starlet είναι:

- **optional_sign()**
- **term()**
- **factor()**
- **boolFactor()**
- **boolTerm()**

Προγράμματα ελέγχου (tests)

Βρίσκονται στο φάκελου syntax_tests του παραδοτέου αρχείο και δημιουργήθηκαν για τον έλεγχο του συντακτικού αναλυτή. Με άλλα λόγια, θέλουμε να ελέγξουμε αν ο συντακτικός αναλυτής λειτουργεί επιθυμητά, και στη περίπτωση συντακτικών λαθών είναι ικανός να τα εντοπίσει. Το κυριότερο από αυτά είναι στο αρχείο example1.slt και ελέγχει το μεγαλύτερο μέρος της συντακτικής ανάλυσης.

```
program example1

    declare d,i,g,f;
```

```
function two (in g)
        function three (in g, inout x, inandout m)
            declare k, j;
            j:=g;
            dowhile
                if (k>i) then
                    k:=k-1
                endif;
                j:=j*k;
                k:=k+g
            enddowhile (k<1);
                m:=j;
                return m+1;
                x:=7
        endfunction
        i:=three (in i+2, inout d, inandout f)
    endfunction

    i:=5;
    g:=1;
    g:=one(in g) //δεν έχει οριστεί αλλά η σύνταξη είναι αποδεκτή
endprogram
```

6. Παραγωγή Ενδιάμεσου Κώδικα

Υλοποίηση

Για την υλοποίηση του επιμέρους κομματιού για την παραγωγή ενδιάμεσου κώδικα, αρχικά υλοποιούμε τις βοηθητικές μεθόδους και κλάσεις, οι οποίες μας βοηθάνε σημαντικά.

- Κλάση Quad: Αναπαριστά και κρατάει τα απαραίτητα δεδομένα για μία τετράδα ενδιάμεσου κώδικα. Πιο συγκεκριμένα περιέχει τα εξής πεδία:
    - Label: το «id» της τετράδας – αριθμός
    - Operator: Ο τελεστής της τετράδας («+», «-», «jump»,…)
    - Argument #1: Το πρώτο όρισμα της τετράδας
    - Argument #2: Το δεύτερο όρισμα της τετράδας

- Resource: Ο πόρος της τετράδας.

- Μέθοδος next_quad: Επιστρέφει το label της επόμενης τετράδας

- Μέθοδος gen_quad: Δέχεται 4 ορίσματα, operator, argument1, argument2, και resource και δημιουργεί τη τετράδα με τις αντίστοιχες τιμές. Έπειτα βάζει τη τετράδα που παράχθηκε στη λίστα (global μεταβλητή) με τις τετράδες

- Μέθοδος new_temp: Δημιουργεί την επόμενη προσωρινή μεταβλητή με όνομα T_1, T_2, T_3 ..

- Μέθοδος empty_list(): Δημιουργεί και επιστρέφει μια άδεια λίστα

- Μέθοδος make_list(): Δέχεται ως όρισμα ένα αριθμό (label) και δημιουργεί μια λίστα με αυτό τον αριθμό

- Μέθοδος merge(): Δέχεται ως όρισμα 2 λίστες και τις συνενώνει. (Η δεύτερη στο τέλος της πρώτης)

- Μέθοδος backpatch(): Δέχεται ως όρισμα μια λίστα από τετράδες, και έναν αριθμό. Για όλες τις τετράδες τις λίστας, αλλάζει το resource της τετράδας στον αριθμό αυτό.

Οι τετράδες παράγονται καλώντας τις βοηθητικές μεθόδους κατά την συντακτική ανάλυση. Πιο συγκεκριμένα, κατά την συντακτική ανάλυση του κεντρικού προγράμματος, αλλά και των συναρτήσεων παράγονται τετράδες της μορφής:

begin_block, _ , _, <όνομα συνάρτησης> //Παράγεται στην αρχή της συντακτικής ανάλυσης μιας συνάρτησης

end_block, _ , _, <όνομα συνάρτησης> //Παράγεται στο τέλος της συντακτικής ανάλυσης μιας συνάρτησης

Κατά την ανάλυση της εντολής **input** παράγεται η τετράδα της μορφής:

in, _, _, id

Κατά την ανάλυση της εντολής **print** παράγεται η τετράδα της μορφής:

out, _, _, expression.place

Κατά την ανάλυση της εντολής **incase** ο ενδιάμεσος κώδικας παράγεται ως εξής:

incase {p1} (when cond {p2} : sequence {p3})* {p4}

Όπου {p1}:

flag = new_temp()

s_quad = next_quad()

gen_quad(:=, _ , _, flag)

{p2}:

ok = next_quad(0

backpatch(cond.true, ok)

gen_quad(:=, 1, _, flag)

{p3}:

next_when = next_quad()

backpatch(c.false, next_whten)

{p4}:

gen_quad(= , 1 , flag, s_quad)

Κατά την ανάλυση της εντολής **forcase** η παραγωγή ενδιάμεσου κώδικα γίνεται ως εξής:

forcase {p1} (when : {p2} sequence {p3})* (default {p4})+ {p5}

Όπου {p1}:

s_quad = next_quad()

exit_list = make_list()

{p2}:

backpatch(c.true, next_quad())

{p3}:

temp_list = make_list(next_quad)

gen_quad(jump, _, _, _)

exit_list = merge (exit_list, temp_list)

backpatch (c.false, next_quad())

{p4}:

genquad(jump, _, _, s_quad)

{p5}:

backpatch(exit_list, next_quad())

Κατά την ανάλυση της εντολής **while** η παραγωγή ενδιάμεσου κώδικα γίνεται ως εξής:

while {p1} B do {p2} S1 {p3}

Όπου:

{p1}:

Bquad:=nextquad()

{p2}:

backpatch(B.true,nextquad())

{p3}:

genquad(“jump”,” _ ”,” _ ”,Bquad)

backpatch(B.false,nextquad())

Προγράμματα Ελέγχου (Τεστς)

Βρίσκονται στο φάκελο intermediate_tests και δημιουργήθηκαν για την επιβεβαίωση της σωστής παραγωγής ενδιάμεσου κώδικα. Για κάθε εντολή της γλώσσας υπάρχει και το αντίστοιχο τεστ. Για παράδειγμα το αρχείο ελέγχου της εντολής if-else:

```
program ifelse
    declare a,b,c,d;
    a:=5;
    b:= 6;
    c := 7;
    d := 9;
    if (a>b) then
        c:= 5;
        print(c)
    else
        d := 12
    endif;
    d := 10
endprogram
```

Παράγει τον εξής τελικό κώδικα:

```
0: begin_block, _, _, ifelse
1: :=, 5, _, a
2: :=, 6, _, b
3: :=, 7, _, c
4: :=, 9, _, d
5: >, a, b, 7
6: jump, _, _, 10
7: :=, 5, _, c
8: out, _, _, c
9: jump, _, _, 11
10: :=, 12, _, d
11: :=, 10, _, d
12: halt, _, _, _
13: end_block, _, _, ifelse
```

7. Πίνακας Συμβόλων

Υλοποίηση

Για την υλοποίηση του επιμέρους κομματιού δημιουργούμε τις εξής κλάσεις και μεθόδους:

- Κλάση Scope: Αναπαριστά ένα Scope
- Κλάση Argument: Αναπαριστά ένα όρισμα σε μια μέθοδο
- Κλάση Entity: Αναπαριστά μια οντότητα
- Κλάση Variable: Αναπαριστά μια μεταβλητή (επεκτείνει τη κλάση Entity)
- Κλάση Function: Αναπαριστά μια συνάρτηση (επεκτείνει τη κλάση Entity)
- Κλάση Parameter: Αναπαριστά μια παράμετρο (επεκτείνει τη κλάση Entity)
- Κλάση TempVariable: Αναπαριστά μια προσωρινή (επεκτείνει τη κλάση Entity)

- Μέθοδος unique_entity: Δέχεται ως όρισμα μια οντότητα, ελέγχει τη λίστα με τα Scopes (global μεταβλητή) και επιστρέφει True η False αν η οντότητα αυτή είναι μοναδική.

- Μέθοδος search_entity_by_name: Δέχεται ένα όρισμα τύπου String και επιστρέφει την οντότητα και το level του scope που βρίσκεται αυτή. Αν δεν υπάρχει η οντότητα αυτή στον πίνακα συμβόλων, επιστρέφει τιμή null

- Μέθοδος add_function_argument: Βρίσκει την οντότητα συνάρτησης με το δοθέν όνομα, και προσθέτει μια οντότητα τύπου Argument στην οντότητα της συνάρτησης αυτής.

- Μέθοδος add_var_entity: Προσθέτει μια οντότητα τύπου Variable στο τρέχων Scope με το δοθέν όνομα.

- Μέθοδος add_parameter_entity: Όμοια της add_var_entity μόνο που αντί Variable προσθέτει οντότητα τύπου Parameter.

- Μέθοδος update_function_entity_framelen: Δέχεται ως όρισμα ένα όνομα και έναν ακέραιο και ενημερώνει το frame length της συνάρτησης που έχει όνομα ίσο με το δοθέν ακέραιο.

- Μέθοδος update_function_entity_quad: Δέχεται ως όρισμα ένα όνομα συνάρτησης, βρίσκει τη συνάρτηση αυτή και αλλάζει την αρχική τετράδα της συνάρτησης αυτής με τη τιμή που θα επιστρέψει η μέθοδος **next_quad().**

- Μέθοδος add_function_entity: Προσθέτει μια συνάρτηση με το δοθέν όνομα στο τρέχων Scope.

- Μέθοδος add_new_scope: Προσθέτει ένα νέο Scope στη λίστα με τα Scopes.

Όταν ο συντακτικός αναλυτής, κάνει ανάλυση στο κεντρικό πρόγραμμα ή στα υποπρογράμματα προσθέτουμε νέο Scope και κρατάμε τη τετράδα που άρχισε το μπλοκ αυτό. Όταν τελειώσει η ανάλυση του μπλοκ και είναι συνάρτηση, ενημερώνουμε το frame length της συνάρτησης καλώντας τη μέθοδο update_function_entity_framelen(). (Παράγουμε τον τελικό κώδικα του μπλοκ). Έπειτα, το μπλοκ αυτό έχει αναλυθεί, επομένως τη βγάζουμε από τη λίστα με τα scopes, αφού δε το χρειαζόμαστε πλεον.

Κατά τη διάρκεια που ο συντακτικος ελέγχει τη σύνταξη των παραμέτρων μιας συνάρτησης, προσθέτουμε αυτές τις παραμέτρους στην οντότητα συνάρτησης που την αντικατοπτρίζει. Επίσης, όπως είναι λογικό, κρατάμε και τον τύπο της παραμέτρου, καθώς θα τον χρειαστούμε στη παραγωγή τελικού κώδικα.

Επιπρόσθετα όταν κατά τη παραγωγή του ενδιάμεσου κώδικα, καλείται η μέθοδος new_temp(), πριν επιστρέψουμε την προσωρινή μεταβλητή, τη προσθέτουμε στο τρέχων scope.

Προγράμματα Ελέγχου (Τεστς)

Για τον έλεγχο σωστής λειτουργίας και αποθήκευσης του πίνακα συμβόλων, ο μεταγλωττιστής παράγει (στον ίδιο φάκελο που βρίσκεται το πρόγραμμα που μεταφράζει) ένα αρχείου κειμένου και κάνει κάποιου είδος pretty print του main scope. Έπειτα στον φάκελο symbol_tests υπάρχουν προγράμματα σε γλώσσα starlet τα οποία μεταφράζουμε για να ελέγξουμε τη λειτουργία του πίνακα συμβολών. Περιέχονται επίσης προγράμματα που δε θα έπρεπε να μεταφράζει ο μεταφραστής. Ένα παράδειγμα ελέγχου, το οποίο δε θα έπρεπε να μεταφράζετε είναι:

```
program unknownvar
    declare a,b;
    a := 12;
    b := 25;
    c := 1000 //Δεν έχει δηλωθεί η μεταβλητή
endprogram
```

Επίσης ένα παράδειγμα παραγωγής αρχείου κειμένου με σκοπό τη προβολή του main scope είναι:

```
----->VARIABLE: x, offset: 12
----->VARIABLE: y, offset: 16
----->VARIABLE: z, offset: 20
----->FUNCTION: p1, start_quad: 12, frame_length: 40
----------->x: (CV, next = z)
----------->z: (REF, next = v)
----------->v: (REF, next = None)
----->TEMP_VARIABLE: T_7, offset: 24
----->TEMP_VARIABLE: T_8, offset: 28
```

8. Παραγωγή Τελικού Κώδικα

Υλοποίηση

Ο τελικός κώδικας παράγεται σε Assembly και η κατάληξη των αρχείων που παράγονται είναι .asm. Για την παραγωγή του τελικού κώδικα δημιουργούμε τις εξείς βοηθητικές μεθόδους:

- gnvlcode: μεταφέρει στον $t0 την διεύθυνση μιας μη τοπικής μεταβλητής και από τον πίνακα συμβόλων βρίσκει πόσα επίπεδα επάνω βρίσκεται η μη τοπική μεταβλητή και μέσα από τον σύνδεσμο προσπέλασης την εντοπίζει
- loadvr: Μεταφορά δεδομένων στον καταχωρητή r η οποία μεταφορά μπορεί να γίνει από τη μνήμη (στοιβα) ή να εκχωρηθεί στο r μια σταθερά. Έπειτα διαχωρίζουμε περιπτώσεις για τη μεταβλητή v
- storerv: Μεταφέρει δεδομένα από τον καταχωρητή r στη μνήμη (μεταβλητή v). Η σύνταξή της είναι store(r,v). Όπως και στη loadvr διακρίνουμε περιπτώσεις για τη μεταβλητή v.

Στη συνέχεια δημιουργούμε τη μέθοδο block_to_assembly η οποία διαβάζει τις τετράδες ενός μπλοκ και παράγει το τελικό κώδικα του. Ανάλογα το τύπο της τετράδας παράγεται και ο αντίστοιχος τελικός κώδικας. Για να γράψουμε στο αρχείο του τελικού κώδικα χρησιμοποιούμε τη μέθοδο write_assembly() η οποία προσθέτει τις δοθέν assembly εντολές σε μια global μεταβλητή τύπου String, το οποίο αξιοποιούμε στο τέλος για να γράψουμε τον κώδικα στο αρχείο.

Προγράμματα Ελέγχου (Τεστς)

Για τον έλεγχο σωστής λειτουργίας δημιουργούμε τα κατάλληλα tests στο φάκελο final_code_tests του παραδοτέου. Ένα παράδειγμα προγράμματος ελέγχου παραγωγής τελικού κώδικα μοιάζει ώς εξής:

Πηγαίο πρόγραμμα:

```
program example2
    declare z,x;

    function p1(in x)
        return x * 3
    endfunction
    z := 10;
    z:= p1(in x)
endprogram
```

Τελικός κώδικας που παράγεται:

```
j L_example2

L_0:            # begin_block, _, _, p1
sw $ra, 0($sp)

L_1:            # *, x, 3, T_1
lw $t1, -12($sp)    # loadvr()
li $t2, 3           # loadvr()
mul $t1, $t1, $t2
sw $t1, -16($sp)    # storerv()

L_2:            # retv, _, _, T_1
lw $t1, -16($sp)    # loadvr()
lw $t0, -8($sp)
sw $t1, 0($t0)
```

```
L_3:                # end_block, _, _, p1
lw $ra,0($sp)
jr $ra

L_example2:         #main label

L_4:                # begin_block, _, _, example2
sw $ra, 0($sp)

L_5:                # :=, 10, _, z
li $t1, 10          # loadvr()
sw $t1, -12($s0)    # storerv()

L_6:                # par, x, CV, _
addi $fp, $sp, -24
lw $t0, -16($s0)    # loadvr()
sw $t0, -12($fp)

L_7:                # par, T_2, RET, _
addi $t0, $sp, -20
sw $t0, -8($fp)

L_8:                # call, _, _, p1
lw $t0, -4($sp)
sw $t0, -4($fp)
addi $sp, $sp, 24
jal L_0
addi $sp, $sp, -24

L_9:                # :=, T_2, _, z
lw $t1, -20($sp)    # loadvr()
sw $t1, -12($s0)    # storerv()

L_10:               #: halt, _, _, _
li $v0, 10          #exit
```

```
syscall

L_11:           #: end_block, _, _, example2
j L_10
```

Ολοκληρωμένο παράδειγμα μετάφρασης

Πηγαίος Κώδικας

```
program example2
    declare x,y,z;

    function p1(in x, inout z, inout v)
        declare w;

        function p2(inout z)
            declare q;
            q:=y+w;
            z:=q*x;
            v:= p2(inout q);
            return 0 //added
        endfunction
        if (x<y) then
            w:=x+y
        else
            w:=x*y
        endif;
        z:= p2(inout x);
        return 0
    endfunction
    x:=1;
    y:=2;
    z:= p1(in x+y, inout z,inout z)
endprogram
```

Ενδιάμεσος Κώδικας

0: begin_block, _, _, p2

1: +, y, w, T_1

2: :=, T_1, _, q

3: *, q, x, T_2

4: :=, T_2, _, z

5: par, q, REF, _

6: par, T_3, RET, _

7: call, _, _, p2

8: :=, T_3, _, v

9: retv, _, _, 0

10: end_block, _, _, p2

11: begin_block, _, _, p1

12: <, x, y, 14

13: jump, _, _, 17

14: +, x, y, T_4

15: :=, T_4, _, w

16: jump, _, _, 19

17: *, x, y, T_5

18: :=, T_5, _, w

19: par, x, REF, _

20: par, T_6, RET, _

21: call, _, _, p2

22: :=, T_6, _, z

23: retv, _, _, 0

24: end_block, _, _, p1

25: begin_block, _, _, example2

26: :=, 1, _, x

27: :=, 2, _, y

28: +, x, y, T_7

29: par, T_7, CV, _

30: par, z, REF, _

31: par, z, REF, _

32: par, T_8, RET, _

33: call, _, _, p1

34: :=, T_8, _, z

35: halt, _, _, _

36: end_block, _, _, example2

Εμφάνιση του main scope:

----->VARIABLE: x, offset: 12

----->VARIABLE: y, offset: 16

----->VARIABLE: z, offset: 20

----->FUNCTION: p1, start_quad: 12, frame_length: 40

---------->x: (CV, next = z)

---------->z: (REF, next = v)

---------->v: (REF, next = None)

----->TEMP_VARIABLE: T_7, offset: 24

----->TEMP_VARIABLE: T_8, offset: 28

Τελικός Κώδικας

```
j L_example2

L_0:                # begin_block, _, _, p2
sw $ra, 0($sp)

L_1:                # +, y, w, T_1
lw $t1, -16($s0)      # loadvr()
lw $t0, -4($sp)
addi $t0, $t0, -24    # gnvlcode()
lw $t2, 0($t0)        # loadvr()
add $t1, $t1, $t2
sw $t1, -20($sp)      # storerv()

L_2:                # :=, T_1, _, q
lw $t1, -20($sp)      # loadvr()
sw $t1, -16($sp)      # storerv()

L_3:                # *, q, x, T_2
lw $t1, -16($sp)      # loadvr()
```

```
lw $t0, -4($sp)
addi $t0, $t0, -12     # gnvlcode()
lw $t2, 0($t0)         # loadvr()
mul $t1, $t1, $t2
sw $t1, -24($sp)       # storerv()

L_4:                   # :=, T_2, _, z
lw $t1, -24($sp)       # loadvr()
lw $t0,-12($sp)        # storerv()
sw $t1, 0($t0)         # storerv()

L_5:                   # par, q, REF, _
addi $fp, $sp, -32
lw $t0, -4($sp)
addi $t0, $t0, -16     # gnvlcode()
sw $t0, -12($fp)

L_6:                   # par, T_3, RET, _
addi $t0, $sp, -28
sw $t0, -8($fp)

L_7:                   # call, _, _, p2
lw $t0, -4($sp)
sw $t0, -4($fp)
addi $sp, $sp, 32
jal L_0
addi $sp, $sp, -32

L_8:                   # :=, T_3, _, v
lw $t1, -28($sp)       # loadvr()
lw $t0, -4($sp)
addi $t0, $t0, -20     # gnvlcode()
lw $t0, 0($t0)         # storerv()
sw $t1, 0($t0)         # storerv()
```

```
L_9:                # retv, _, _, 0
li $t1, 0           # loadvr()
lw $t0, -8($sp)
sw $t1, 0($t0)

L_10:               #: end_block, _, _, p2
lw $ra,0($sp)
jr $ra

L_11:               #: begin_block, _, _, p1
sw $ra, 0($sp)

L_12:               #: <, x, y, 14
lw $t1, -12($sp)      # loadvr()
lw $t2, -16($s0)      # loadvr()
blt $t1, $t2, L_14

L_13:               #: jump, _, _, 17
j L_17

L_14:               #: +, x, y, T_4
lw $t1, -12($sp)      # loadvr()
lw $t2, -16($s0)      # loadvr()
add $t1, $t1, $t2
sw $t1, -28($sp)      # storerv()

L_15:               #: :=, T_4, _, w
lw $t1, -28($sp)      # loadvr()
sw $t1, -24($sp)      # storerv()

L_16:               #: jump, _, _, 19
j L_19

L_17:               #: *, x, y, T_5
lw $t1, -12($sp)      # loadvr()
```

```
lw $t2, -16($s0)       # loadvr()
mul $t1, $t1, $t2
sw $t1, -32($sp)       # storerv()

L_18:                  #: :=, T_5, _, w
lw $t1, -32($sp)       # loadvr()
sw $t1, -24($sp)       # storerv()

L_19:                  #: par, x, REF, _
addi $fp, $sp, -40
lw $t0, -4($sp)
addi $t0, $t0, -12     # gnvlcode()
sw $t0, -12($fp)

L_20:                  #: par, T_6, RET, _
addi $t0, $sp, -36
sw $t0, -8($fp)

L_21:                  #: call, _, _, p2
sw $sp,-4($fp)
addi $sp, $sp, 40
jal L_0
addi $sp, $sp, -40

L_22:                  #: :=, T_6, _, z
lw $t1, -36($sp)       # loadvr()
lw $t0,-16($sp)        # storerv()
sw $t1, 0($t0)         # storerv()

L_23:                  #: retv, _, _, 0
li $t1, 0              # loadvr()
lw $t0, -8($sp)
sw $t1, 0($t0)

L_24:                  #: end_block, _, _, p1
```

```
lw $ra,0($sp)
jr $ra

L_example2:          #main label

L_25:                #: begin_block, _, _, example2
sw $ra, 0($sp)

L_26:                #: :=, 1, _, x
li $t1, 1            # loadvr()
sw $t1, -12($s0)     # storerv()

L_27:                #: :=, 2, _, y
li $t1, 2            # loadvr()
sw $t1, -16($s0)     # storerv()

L_28:                #: +, x, y, T_7
lw $t1, -12($s0)     # loadvr()
lw $t2, -16($s0)     # loadvr()
add $t1, $t1, $t2
sw $t1, -24($sp)     # storerv()

L_29:                #: par, T_7, CV, _
addi $fp, $sp, -32
lw $t0, -24($sp)     # loadvr()
sw $t0, -12($fp)

L_30:                #: par, z, REF, _
addi $t0, $sp, -20
sw $t0, -16($fp)

L_31:                #: par, z, REF, _
addi $t0, $sp, -20
sw $t0, -20($fp)
```

```
L_32:               #: par, T_8, RET, _
addi $t0, $sp, -28
sw $t0, -8($fp)

L_33:               #: call, _, _, p1
lw $t0, -4($sp)
sw $t0, -4($fp)
addi $sp, $sp, 32
jal L_11
addi $sp, $sp, -32

L_34:               #: :=, T_8, _, z
lw $t1, -28($sp)      # loadvr()
sw $t1, -20($s0)      # storerv()

L_35:               #: halt, _, _, _
li $v0, 10           #exit
syscall

L_36:               #: end_block, _, _, example2
j L_35
```