Μεταγλωττιστές

Άσκηση Μέρος 5

Κώδικας τριών διευθύνσεων
Έχοντας ολοκληρώσει την υλοποίηση του σημασιολογικό αναλυτή, δημιουργούμε μηχανισμό ο οποίος διαμορφώνει τον κώδικα προς μεταγλώττιση σε μια ενδιάμεση μορφή, συγκεκριμένα σε κώδικα τριών διευθύνσεων. Στόχος μας είναι ο νέος κώδικας να μπορεί να μετατρέπεται εύκολα σε κώδικα προς εκτέλεση στην φάση παραγωγής κώδικα.
Για να επιτευχθεί αυτό, δημιουργούμε ένα πακέτο κλάσεων με την ιδιότητα να μετατρέπει κάθε εντολή του κώδικα προς μεταγλώττιση σε μια ή παραπάνω εντολές τριών διευθύνσεων. Όλες οι κλάσεις υλοποιούν το Instruction interface. Κάθε κλάση υλοποιεί την μετατροπή των εντολών σε printable κώδικα με την μέθοδο emit().

Συντονιστής αυτής της διαδικασίας είναι η κλάση IntermediateCodeASTVisitor, η οποία κάνει την αντιστοίχηση. Το αποτέλεσμα του περάσματος είναι μία λίστα από Instructions, της οποίας η διαχείριση γίνεται μέσα σε στιγμιότυπο της κλάσης Program. Για να δημιουργηθεί το αντικείμενο Program, πρέπει να γίνεται διαχείριση αναγνωριστικών και προσωρινών μεταβλητών κατά την διάρκεια του περάσματος, οπότε χρησιμοποιούμε μια στοίβα για το λόγο αυτό. Κάθε κόμβος που χρησιμοποιεί αναγνωριστικό ή δημιουργεί προσωρινή μεταβλητή (η οποία αντιπροσωπεύει τη χρήση ενός temporary register) την προσθέτει στην στοίβα, ώστε να μπορούν να χρησιμοποιηθούν σε υψηλότερα επίπεδα του συντακτικού δέντρου, ώστε όλες εντολές να χρησιμοποιούν τα σωστά operands. Το πέρασμα αυτό δεν αφορά ελέγχους τύπων ή δηλώσεις μεταβλητών, καθώς έχουν ήδη πραγματοποιηθεί. Χρησιμοποιούμε όμως τα αποτελέσματα των προηγούμενων περασμάτων.

Κάθε κλάση αντιπροσωπεύει έναν operator της τετράδας, και τα 3 (ή λιγοτερα) υπόλοιπα operands εισάγονται με τον αντίστοιχο setter ή constructor. Έτσι, με βάση τις προδιαγραφές που έχουμε χρησιμοποιήσει μέχρι τώρα, οι εντολές διαμορφώνονται ως εξής:

Instruction types table **Color Legend: Operands, helper arguments, example values**

Constructor Call Template	Example 3AC Instruction
ArrayInitInstr(size, type, result)	*result =newArray 4 x size*
ArrIndAccessInstr(array, type, index, target)	*target = array[index]*
ArrIndAssignInstr(array, type, index, source)	*array[index] = source*
AssignInstr(source, target)	*target = source*
BinaryOpInstr(operator, arg1, arg2, result)	*result = arg1 + arg2*
CondJumpInstr(operator, arg1, arg2)	*if arg1 >= arg2 goto _*
CondJumpInstr(operator, arg1, label)	*if IS_FALSE arg1 goto label*

CondJumpInstr(operator, arg1, arg2, label)	*if arg1>= arg2 goto label*
FuncDeclInstr(function_name)	*-- function_name --*
FunctCallInstr(result ,function_name, params)	*result = call function_name, params*
FunctCallInstr(function_name, params)	*call function_name, params*
ParamInstr(function_param)	*param function_param*
GotoInstr(label)	*goto label*
LabelInstr(name)	*L1:*
GotoInstr()	*goto _*
ReturnInstr()	*return*
ReturnInstr(result)	*return result*
UnaryOpInstr(operator, source, target)	*target = ! source*

Η εκτέλεση του κώδικα δεν γίνεται πάντα σειριακά, πολλές φορές θα πρέπει να εκτελεστεί κάποιο κομμάτι κώδικα πάνω από μια φορά ή κάποιο άλλο να μην εκτελεστεί καθόλου, αυτό εξαρτάται από τις εκάστοτε συνθήκες, έτσι χρησιμοποιούμε την εντολή goto _ η οποία μεταφέρει την ροή του κώδικα σε κάποιο άλλο σημείο του αρχείου. Κατά την μεταγλώττιση μια εντολής είναι πολλές φορές αδύνατο να προσδιοριστεί η επόμενη εντολή. Για αυτό τον λόγο, χρησιμοποιούμε λίστες μοναδικές για κάθε εντολή, οι οποίες περιέχουν την τοποθεσία των εντολών εκείνων που μετά την εκτέλεση τους η ροή εκτέλεσης θα μεταπηδήσει κάπου αλλού. Η μεταγλώττιση γίνεται αναδρομικά, έτσι όταν ένας κόμβος καλέσει αναδρομικά κάποιον άλλον, ο πρώτος κόμβος θα μπορεί να προσδιορίσει τι θα πρέπει να εκτελεστεί μετά από τον κόμβο που ο ίδιος κάλεσε αναδρομικά και να ενημερώσει τις λίστες του. Αν τυχόν δεν μπορεί να το κάνει αυτό ενώνει τις λίστες του και τις λίστες των κόμβων που κάλεσε ο ίδιος και αναθέτει αυτήν την διαδικασία στον κόμβο που αρχικά τον κάλεσε. Ακολουθούμε δηλαδή την τεχνική backpatching.

Υλοποιούμε την τεχνική του Short-Circuit σε κάθε δομή ελέγχου. Αυτό σημαίνει πως δεν γίνονται όλοι οι έλεγχοι αν μπορεί να υπολογιστεί το τελικό αποτέλεσμα από τους ελέγχους που έχουν γίνει ήδη. Αυτό προσφέρει στον κώδικα προς εκτέλεση καλύτερη απόδοση. Πιο συγκεκριμένα, ελέγχουμε σε μια πράξη AND αν ο πρώτος όρος είναι false και σε μια OR αν ο πρώτος όρος είναι true. Ξέρουμε έτσι από την αρχή ποιο είναι το αποτέλεσμα της λογικής πράξης και αποφεύγουμε παραπάνω πράξεις. Τέλος δίνουμε warning άμα ανιχνεύσουμε ότι θα μπορούσαν να εξοικονομηθούν περισσότερες πράξεις, αν οι όροι απλά αντιστραφούν στην λογική πράξη.