MYY802 Compilers I Assignment Report

Κωνσταντίνος Γεωργίου, 4333
Παπαποστόλου Αθανάσιος, 4147
Μάιος 28, 2022

Περιγραφή. Στην παρακάτω αναφορά θα παρουσιάσουμε την υλοποίηση μιας εκπαιδευτικής γλώσσας προγραμματισμού που υιοθετεί στοιχεία από την γλώσσα C, με όνομα $\mathscr{C}$-imple. Θα χρησιμοποιήσουμε τη γλώσσα Python για την υλοποιήση, αποφεύφοντας έτσι την διαδικασία του bootstrapping (με τη χρήση της υψηλού επιπέδου γλώσσας θα κατασκευάσουμε όλες τις λεκτικές και γραμματικές μονάδες). Θα κατασκευάσουμε δομές για λεκτική ανάλυση σε ομάδες τερματικών, και για συντακτική ανάλυση ενός αναδρομικού top-down parser. Θα παράγουμε ενδιάμεσες τετράδες ενδιάμεσου κώδικα με τη βοήθεια των οποίων θα εξάγουμε τελικό κώδικα σε MIPS assembly, ή σε C. Τέλος θα τεστάρουμε τη λειτουργία με δοκιμαστικά προγράμματα καθόλη τη διαδικασία ανάπτυξης του μεταφραστή.

1. Introduction

Η γλώσσα $\mathscr{C}$-imple ως turing complete γλώσσα περιέχει ψηφία, σύμβολα για πράξεις αριθμιτικές συσχέτισης, καθώς επίσης και από διαχωριστικά σύμβολα και παρενθέσεις. Η χρήση αλφαριθμιτικών αποτελεί μονάχα στοιχείο των identifiers, και όχι την δημιουργία string literal. Η $\mathscr{C}$-imple επομένως υποστηρίζει μόνο ακέραιους αριθμούς σαν προκαθορισμένο τύπο. Με την χρήση υπορουτινών ωστόσο μπορεί να υλοποιηθεί κάθε μορφής αναπαράσταση άλλων δομών (floating point packages, strings, complex data structures etc.). Υπάρχουν ειδικές ρουτίνες εισόδου και εξόδου (input και print) οι οποίες αντίστοιχα υλοποιούνε το διάβασμα και την εκτύπωση μεταβλητών. Αξίζει επίσης να σημειώσουμε πως στη γραμματική της γλώσσας χρησιμοποιούνται ειδικές δομές επιλογής όπως η incase και η forcase.

Η μετάφραση του πηγαίου κώδικα ακολουθεί την εξής τεχνική. Αρχικά με ένα pass ο lexical analyser διαβάζει ανά χαρακτήρα το αρχείο εισόδου, σημειώνοντας σε μια λίστα κάθε lexical token. Στη συνέχεια, ο syntax analyser χρησιμοποιεί τη λίστα με τα tokens, και τα χαρτογραφεί σύμφωνα με τους γραμματικούς κανόνες που έχουν οριστεί. Ο parser έχει τη δυνατότητα να επιστρέφει σε προηγούμενα στοιχεία του token list ανάλογα με το συντακτικό δέντρο που δημιουργείται από την είσοδο. Όταν ο parser καταλήξει σε τερματική τιμή, επιστρέφει πίσω στο stack frame του αρχικού κανόνα και εφόσον ολοκληρώσει έναν γραμματικό κανόνα παράγεται ο ανάλογος ενδιάμεσος κώδικας. Με τη χρήση ειδικών δομών υλοποιούμε scoping rules, που επιτρέπουν namespacing ανα block, και μας δίνουν τη δυνατότητα να χρησιμοποιήσουμε μεταβλητές ίδιων ονομάτων σε διαφορετικά blocks, καθώς επίσης και την δυνατότητα για χρήση αναδρομικών κλήσεων σε υπορουτίνες. Έχοντας την λίστα με τις εντολές ενδιάμεσου κώδικα μπορούμε εύκολα να εξάγουμε τον κώδικα που τελικά θα εκτελέσουμε κάνοντας transpile είτε σε MIPS assembly είτε σε C.

Για να χρησιμοποιήσουμε τον compiler εκτελούμε ως εξής:

```
Usage: python3 cimple.py <input file> [Options]

Options:
    -h, --help            show this help message and exit
    -o FILEPATH, --output=FILEPATH
                          specify compilation output name
```

Με την εκτέλεση παράγονται τα αρχεία:
<output>.c
<output>.int
<output>.asm

2. Lexer

Στο κομμάτι της λεκτικής ανάλυσης υλοποιούμε ένα αντικείμενο `Token` το οποίο αποθηκεύει για κάθε token, το string που αναγνωρίστηκε από τον lexer, την ομάδα λεκτικού στοιχείου, και την γραμμή στο αρχείο του κώδικα όπου διαβάστηκε το token. Την γραμμή αρχείου την κρατάμε για αναλυτικότερη παρουσίαση των syntax error, που θα υλοποιήσουμε στον parser.

Έχουμε τις εξής ομάδες λεκτικών στοιχείων:

- number - στοιχεία που ικανοποιούν το [0-9]+
- addOperator - στοιχεία που ικανοποιούν το ['+', '-']
- mulOperator - στοιχεία που ικανοποιούν το ['*', '/']
- groupSymbol - στοιχεία που ικανοποιούν το ['(', ')', '[', ']', '{', '}']
- delimiter - στοιχεία που ικανοποιούν το [',', ';', '.']
- assignment - στοιχεία που ικανοποιούν το [':=']
- relOperator - στοιχεία που ικανοποιούν το ['=', '<=', '>=', '<', '>', '<>']
- keyword - στοιχεία που ικανοποιούν το **keyword list**
- identifier - στοιχεία που ικανοποιούν το [a-zA-Z][a-zA-Z0-9]*

keyword list = ['program', 'declare', 'if', 'else', 'while', 'switchcase', 'not', 'function', 'input', 'forcase', 'incase', 'case', 'default', 'and', 'or', 'procedure', 'call', 'return', 'in', 'inout', 'print']

Τα keywords αποτελούν υποσύνολο των identifiers: όταν κάνουμε lex, ενα identifier token, πριν την ανάθεση ομάδας ελέγχουμε αν το identifier βρίσκεται στο keyword list.

Για την ανάλυση, υλοποιούμε ένα αντικείμενο `Lexer`, το οποίο με το μήνυμα lex εκτελεί τη μέθοδο της λεκτικής ανάλυσης. Αποθηκεύει το αποτέλεσμα στο πεδίο `token_table` που περιέχει `Token` αντικείμενα. Με τη λίστα που δημιουργούμε μπορεί ο syntax analyser να αναλύσει με βάση τους γραμματικούς κανόνες.

3. Parser

Στο κομμάτι της συντακτικής ανάλυσης υλοποιούμε ένα αντικείμενο Parser το οποίο με το μήνυμα analyze_syntax εκτελεί τις μεθόδοθς για την συντακτική ανάλυση. Πιο συγκεκριμένα, στέλνει το self.program() και ξεκικά την αναδρομική ανάλυση. Για κάθε μη τερματικό συντακτικό κανόνα εφαρμόζουμε μια βοηθητική μέθοδο, με όνομα το όνομα του κάθε κανόνα από τη γραμματική. Σε κάθε κανόνα διαβάζουμε το επόμενο token στη λίστα, και εξετάζουμε εαν περιέχεται στον γραμματικό κανόνα. Σε οποιαδήποτε περίπτωση που κάποιο token βρίσκεται λανθασμένα στην επόμενη θέση στη λίστα, σταματάμε αμέσως την μετάφραση και αναφέρουμε το συντακτικό λάθος στο output. Εφόσον τα tokens που αναλύουμε βρίσκονται στη σωστή σειρά (αυτή που

προϋποθέτει ο parser ή στο πιθανό AST που μπορούμε να εξάγουμε), τότε ο parser θα καταλήξει οπωσδήποτε σε κάποιο τερματικό κανόνα. Οι περιπτώσεις αυτές ελέγχονται με την βοηθητική υπορουτίνα `terminal_match`. Σε αυτή την περίπτωση έχουμε φτάσει στα φύλλα του συντακτικού δέντρου για τον γραμματικό κανόνα και επιστρέφουμε πίσω στο stack frame του κάθε ενδιάμεσου κανόνα συμπληρώνοντας τα ανάλογα strings ενδιάμεσου κώδικα.

Όταν εφαρμόζουμε κανόνες που περιγράφουν λίστες όπως για παράδειγμα λίστες από παραμέτρους συναντήσεων χωρισμένες με κόμμα, ή λίστες από declarations χωρισμένα με `;` τότε καλούμε τον ίδιο κανόνα αναδρομικά.

Πχ:

```
    def blockstatements(self):
        self.statement();
        if self.get_token().recognized_string == ";":
            self.blockstatements();
        elif self.token.recognized_string == "}":
            self.token = self.undo_token();
        else:
            self.error(f"expected `;` after blockstatement instead
got `{self.token.recognized_string}`");
```

Ο κανόνας `blockstatements` περιγράφεται στη γραμματική με BNF form ως εξής:

blockstatements : statement ‘;’
| blockstatements ‘;’ statement

Η πρώτη κλήση βρίσκεται στο self.statement(), δηλαδή εκτελούμε τον κανόνα statement, ο οποίος καταναλώνει τα κομμάτια του token list που του αντιστοιχούν. Εφόσον το επόμενο token είναι το ‘;’ τοτε μπορούμε να ξαναστείλουμε το blockstatements ολοκληρώνοντας το δεύτερο κομμάτι του κανόνα. Σε περίπτωση που βρούμε την αγκύλη τότε σημαίνει πως έχουμε τελειώσει καθώς αυτό είναι το αναμενόμενο επόμενο σύμβολο στον ακριβώς προηγούμενο κανόνα. Για διατήρηση του format στην ανάλυση στέλουμε undo στο token_list το οποίο διορθλωνει τον δείκτη στο πρόσφατο token, μια θέση πίσω. Σε οποιαδήποτε άλλη περίπτωση έχουμε συντακτικό λάθος και αναφέρουμε ένα syntax error στη γραμμή που εμφανίζεται το λάθος και τερματίζουμε πρόωρα.

Με παρόμοιο τρόπο υλοποιούνται όλοι οι κανόνες της γραμματικής εφαρμόζοντας στενά τη μορφή της γραμματικής. Αξίζει να σημειώσουμε μια ιδιαιτερότητα στον κανόνα `statement` όπου ο parser κοιτά ένα token πιο μπροστά, κάνοντας αμέσως undo, με σκοπό να καθορίσει ποιόν υποκανόνα statement να εκτελέσει.

4. Intermediate Code

Generation inside parser code blocks, Quad data object, Int block examples (showcase quad examples) write saved variables list, simple conversion. write to output

5. Symbol Table

-

6. Final Code Generation

Print quads, convert to C code. Turn quads into sufficient asm code. Use stack when calling subroutines.

7. Conclusion

Καταλήγοντας