Μεταγλωττιστές

Tony

Γαλάνης Δημήτριος - 03116088 (el16088@mail.ntua.gr)
Σπυρίδων Παυλάτος - 03116113 (spyrospavlatos4@gmail.com)

The goal of Computer Science is to build something that will last at least until we've finished building it.

William C. Brown

The separation of state and church must be complemented by the separation of state and science, that most recent, most aggressive, and most dogmatic religious institution

Paul Karl Feyerabend

One of the most important lessons, perhaps, is the fact that SOFTWARE IS HARD. From now on I shall have significantly greater respect for every successful software tool that I encounter...The creation of good software demand a significantly higher standard of accuracy than those other things do, and it requires a longer attention span than other intellectual tasks

Donald Knuth

My cow is not pretty, but it is pretty to me.

David Lynch

Η παρούσα εργασία εκπονήθηκε στα πλαίσια του μαθήματος των Μεταγλωττιστών. Σκοπός είναι η κατασκευή ενός μεταγλωττιστή για μια απλή γλώσσα προγραμματισμού, την ***Tony***. Αρχικά παρουσιάζουμε μια γενική περιγραφή της αρχιτεκτονικής του μεταγλωττιστή που αναπτύξαμε. Στην συνέχεια αναλύουμε τις σχεδιαστικές επιλογές και αναλύουμε περεταίρω την λειτουργία του κάθε *module* του μεταγλωττιστή.

Περιγραφή Αρχιτεκτονικής του Μεταγλωττιστή

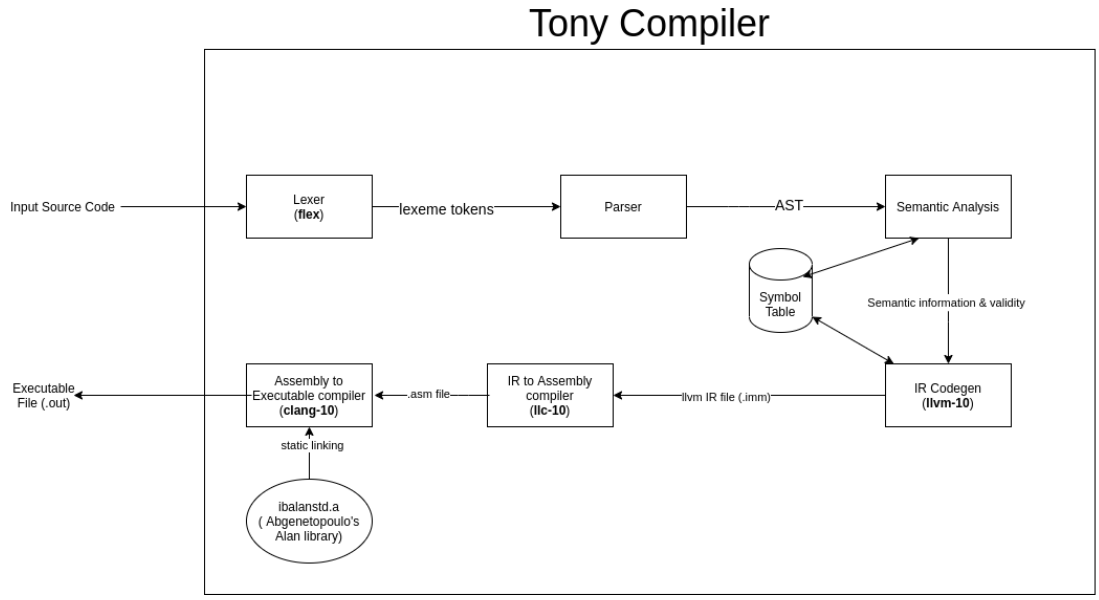
Η γενικότερη αρχιτεκτονική του μεταγλωττιστή φαίνεται στο Σχήμα 1. Το αρχείο είσόδου είναι κώδικας γραμμένος στην ***Tony***. Ο λεκτικός αναλυτής (lexer) διατρέχει το κείμενο αναγνωρίζοντας λεκτικές μονάδες (lexemes) και τις περνάει με τον αντιστοιχο κωδικο τους (token) στον συντακτικό αναλυτή (parser). Στη συνέχεια ο parser παράγει το Αφηρημένο Συντακτικό Δέντρο (Abstract Syntax Tree - AST), βάσει της συνταχτικής θέσης της κάθε λέξης. Καθώς το AST διατρέχεται bottom-up γίνεται και η σημασιολογική ανάλυση του προγράμματος και εφόσον αυτή είναι επιτυχής παρέχει τη σημασιολογική πληροφορία για την παραγωγή του ενδιάμεσου κώδικα (IR). Η παραγωγή αυτή γίνεται με χρήση του LLVM framework. Αφού έχει παραχθεί ο ενδιάμεσος κώδικας προχωράμε στην μεταγλώττιση του σε γλώσσα μηχανής (Assembly) και στην συνεχεια σε εκτελέσιμο μετά την στατική σύνδεση (static linking) με μια βασική βιβλιοθήκη.

Λεκτική Ανάλυση

Η υλοποίηση του λεκτικού αναλυτή γίνεται μέσω του εργαλείου ανοικτού λογισμικού flex το οποίο μας δίνει την δυνατότητα αυτόματης αναγνώρισης των λεκτικών μονάδων ενός προγράμματος μέσω των κανόνων που ορίζει το documentation της γλωσσας. Η αναγνωρισθείσα λεκτική μονάδα δίνεται στον συντακτικό αναλυτή μέσω της δομής *yylval*.

Για τα δεσμευμένα αναγνωριστικά λεκτικών μονάδων (*λέξεις-κλειδιά*) φροντίζουμε οι αντίστοιχοι κανόνες που τα αναγνωρίζουν να έχουν μεγαλύτερη προτεραιότητα σε σχέση με τους κανόνες αναγνώρισης αναγνωριστικών μεταβλητών ή συναρτήσεων.

Ακόμη, στον λεκτικό αναλυτή ορίζονται κανόνες αναγνώρισης λεκτικών μονάδων που αντιστοιχούν σε δεκαεξαδικούς αριθμούς λαμβάνοντας υπόψιν την διαφοροποίηση που πρέπει να υπάρξει από την περίπτωση



Σχήμα 1: Αρχιτεκτονική του μεταγλωττιστή

όπου οι λεκτικές μονάδες αντιστοιχούν σε χαρακτήρες διαφυγής (*escape characters*) οι οποίοι επίσης ξεκινούν με τον χαρακτήρα backslash.

Τέλος, υλοποιούμε κανόνες λεκτικής αναγνώρισης single και multi-line σχολίων μέσα στα οποία δεν αναγνωρίζουμε άλλες λεκτικές μονάδες.

Συντακτική Ανάλυση

Το συντακτικό της Tony

Η υλοποίηση του συντακτικού αναλυτή γίνεται μέσω του εργαλείου ανοικτού λογισμικού *GNU-Bison*. Το εργαλείο αυτό έχοντας τους κανόνες της γραμματικής (Σχήμα 2), κατασκευάζει έναν LALR (*Look Ahead Left-Right* parser που αναγνωρίζει την γραμματική αυτή.

Φυσικά η γραμματική της Tony είναι ambiguous (*διφορούμενη*) κάτι που όμως ξεπερνούμε ορίζοντας στο bison την προσεταιριστικότητα και προτεραιότητα των τελεστών, καθώς και ορίζοντας τους μοναδιαίους τελεστές ως μη-αντιμεταθετικούς (*non-associative*).

Κατασκευή του AST

Η λειτουργία του parser εκκινά με την κλήση της συνάρτησης *yyparse*. Με την αναγνώριση μιας συντακτικής μονάδας ακολουθώνται ορισμένες λειτουργίες όπως η κατασκευή κόμβων του AST και η επεξεργασία πάνω σε ήδη υπάρχοντες. Η ιεραρχία των κλάσεων που κληρονομούν από τον θεμελιώδη κόμβο του AST δέντρου δίνονται από το Σχήμα 3.

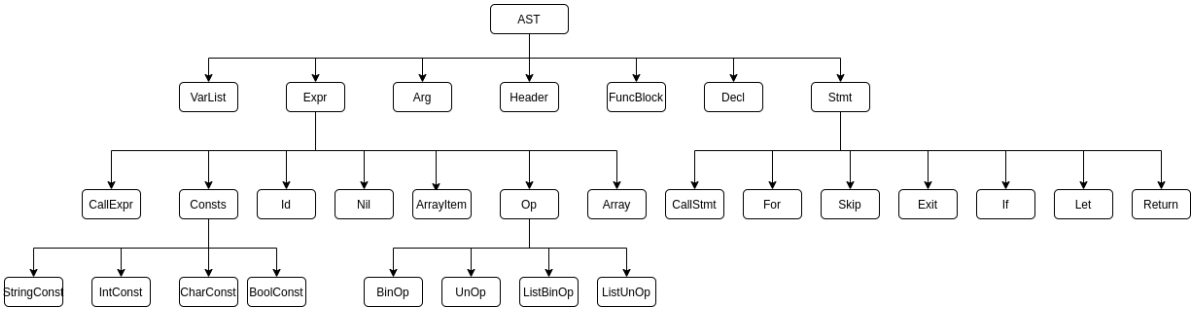
Επειδή ο parser που δημιουργείται από το *bison* είναι *Left-to-Right* παράγει πρώτα τον δεξιότερο κόμβο. Στο Σχήμα 4 παραθέτουμε ενδεικτικά ένα παράδειγμα για κατασκευή των AST nodes που αντιστοιχούν σε μια λίστα από statements της γλώσσας δημιουργείται ένα νέο αντικείμενο κάποιας υποκλάσης της κλάσης Statement. Το σύνολο των statements (μια-μια την φορά και από δεξια προς τα αριστερά) εισάγονται σε μια λίστα απο statements η οποίο χρησιμοποιείται για την δημιουργία άλλων υποκλάσεων του AST node όπως το FuncBlock.

Σημασιολογική Ανάλυση

'Εχοντας παράγει το συντακτικό δέντρο προχωράμε στη σημασιολογική ανάλυση. Κάθε κόμβος του AST, (που χρειάζεται να το κάνει) υλοποιεί την μέθοδο *sem()* η οποία κάνει την σημασιολογική ανάλυση του εκάστοτε κόμβου καλώντας την μέθοδο *sem()* των παιδιών του κόμβου. Η κλήση των *sem()* ξεκινά από τον κόμβο *FuncBlock* της main συνάρτησης του προγράμματος.

```
1  ⟨program⟩     ::= ⟨func-def⟩
2  ⟨func-def⟩    ::= "def" ⟨header⟩ ":" ( ( ⟨func-def⟩ | ⟨func-decl⟩ | ⟨var-def⟩ ) )* ( ⟨stmt⟩ )+ "end"
3  ⟨header⟩      ::= [ ⟨type⟩ ] ⟨id⟩ "(" [ ⟨formal⟩ ( ( ";" ⟨formal⟩ ) )* ] ")"
4  ⟨formal⟩      ::= ["ref"] ⟨type⟩ ⟨id⟩ ( ( "," ⟨id⟩ ) )*
5  ⟨type⟩        ::= "int" | "bool" | "char" | ⟨type⟩ "[" "]" | "list" "[" ⟨type⟩ "]"
6  ⟨func-decl⟩   ::= "decl" ⟨header⟩
7  ⟨var-def⟩     ::= ⟨type⟩ ⟨id⟩ ( ( "," ⟨id⟩ ) )*
8  ⟨stmt⟩        ::= ⟨simple⟩ | "exit" | "return" ⟨expr⟩
9                  | "if" ⟨expr⟩ ":" ( ⟨stmt⟩ )+ ( ( "elsif" ⟨expr⟩ ":" ( ⟨stmt⟩ )+ ) )*
10                     [ "else" ":" ( ⟨stmt⟩ )+ ] "end"
11                 | "for" ⟨simple-list⟩ ";" ⟨expr⟩ ";" ⟨simple-list⟩ ":" ( ⟨stmt⟩ )+ "end"
12 ⟨simple⟩      ::= "skip" | ⟨atom⟩ ":=" ⟨expr⟩ | ⟨call⟩
13 ⟨simple-list⟩ ::= ⟨simple⟩ ( ( "," ⟨simple⟩ ) )*
14 ⟨call⟩        ::= ⟨id⟩ "(" [ ⟨expr⟩ ( ( "," ⟨expr⟩ ) )* ] ")"
15 ⟨atom⟩        ::= ⟨id⟩ | ⟨string-literal⟩ | ⟨atom⟩ "[" ⟨expr⟩ "]" | ⟨call⟩
16 ⟨expr⟩        ::= ⟨atom⟩ | ⟨int-const⟩ | ⟨char-const⟩ | "(" ⟨expr⟩ ")"
17                 | ( "+" | "-" ) ⟨expr⟩ | ⟨expr⟩ ( "+" | "-" | "*" | "/" | "mod" ) ⟨expr⟩
18                 | ⟨expr⟩ ( "=" | "<>" | "<" | ">" | "<=" | ">=" ) ⟨expr⟩
19                 | "true" | "false" | "not" ⟨expr⟩ | ⟨expr⟩ ( "and" | "or" ) ⟨expr⟩
20                 | "new" ⟨type⟩ "[" ⟨expr⟩ "]" | "nil" | "nil?" "(" ⟨expr⟩ ")"
21                 | ⟨expr⟩ "#" ⟨expr⟩ | "head" "(" ⟨expr⟩ ")" | "tail" "(" ⟨expr⟩ ")"
22
```

Σχήμα 2: Η σύνταξη της Tony σε EBNF μορφή. Η υλοποίηση μας διαφέρει ελαφρώς από την παραπάνω αναπαράσταση της σύνταξης της Tony, αφού για λόγους δημιουργίας του AST και της σημασιολογικής ανάλυσης πάνω σε αυτό χρειάζεται αναλύσουμε περεταίρω μερικούς από τους συνταχτικούς κανόνες που φαίνονται στην εικόνα.



Σχήμα 3: Δομή ιεραρχίας υποκλάσεων του AST node

Η χρησιμότητα το σημασιολογικού αναλυτή έγκειται στην εύρεση σημασιολογικών σφαλμάτων - όπως αυτά αναφέρονται στο documentation της Tony - για τον πρόωρο τερματισμό της μεταγλώττισης, καθώς και την εύρεση πληροφοριών για τους κόμβους του AST που θα χρησιμεύσει κατά την παραγωγή του ενδιάμεσου κώδικα.
Κατά την λειτουργία του, ο σημασιολογικό αναλυτής γράφει και παίρνει πληροφορίες - για τις μεταβλητές, τις συναρτήσεις και τα αναγνωριστικά τους - από έναν πίνακα συμβόλων[1] (*SymbolTable*). Έτσι μπορούν να αναγνωρίζονται σφάλματα όπως διπλο-ορισμοί μεταβλητών, χρήση μεταβλητών που δεν έχουν οριστεί, σφάλματα τύπων, ύπαρξη ή μη εντολών επιστροφής σε συναρτήσεις που πρέπει να επιστρέφουν αποτέλεσμα κτλπ.

Παραγωγή Ενδιαμέσου Κώδικα

Η παραγωγή του ενδιάμεσου κώδικα (*Intermediate Representation - IR*) γίνεται μέσω του IRBuilder API της βιβλιοθήκης *LLVM*. Οι πληροφορίες που έχουν προκύψει για τους κόμβους του AST χρησιμοποιούνται για την παραγωγή του ενδιάμεσου κώδικα. [2]

[1] Επεκτείναμε τον κώδικα πίνακα συμβόλων που δόθηκε στα πλαίσια του μαθήματος.

[2] Για την ακρίβεια, στην υλοποίηση μας, το AST επανακατασκευάζεται από την αρχή όπως και το SymbolTable. Αυτό γίνεται διότι κατά την σημασιολογική ανάλυση γίνονται αλλαγές στο SymbolTable (καθώς αλλάζουν τα scopes στο εκάστοτε σημείο που βρισκόμαστε) και επομένως χρειάζεται το AST να γίνει parse ξανά. Οι πληροφορίες για τα AST nodes που παράγωνται κατά την σημασιολογική ανάλυση και χρειάζονται και στην παραγωγή ενδιαμέσου κώδικα επανακτόνται βάζοντας τις μεθόδους *compile()* να

```
stmt-list-plus:
    stmt { $$ = new StmtList(); $$->insert($$->begin(), $1); }
  | stmt stmt-list-plus { $2->insert($2->begin(), $1); $$ = $2; }
;

stmt:
    simple { $$ = $1;}
  | "exit" { $$ = new Exit(); }
  | "return" expr { $$ = new Return($2); }
  | if-stmt { $$ = $1;}
  | "for" simple-list ';' expr ';' simple-list ':' stmt-list-plus "end" { $$ = new For($2, $4, $6, $8);
  ;
```

Σχήμα 4: Παράδειγμα κατασκευής κόμβων του AST στο bison για μια λίστα τουλάχιστον ενός statement. Το "$$" υποδηλώνει το αριστερό μέλος του κανόνα και το "$n "το n-οστό μέλος του δεξιού μέλους του κανόνα.

Όπως και στην σημασιολογική ανάλυση, η παραγωγή του ενδιάμεσου κώδικα ξεκινά από το AST node του *FuncBlock* της main συνάρτησης μέσω της κλήσης της μεθόδου *compile()* η οποία θα καλέσει με την σειρά της τις μεθόδους *compile()* των παιδιών του AST node της main.

Η λειτουργία παραγωγής ενδιάμεσου llvm κώδικα είναι αρκετά ετερογενής για τις διάφορες υποκλάσεις της AST node και επομένως δεν θα περιγράψουμε αναλυτικά την παραγωγή ενδιαμέσου κώδικα για κάθε μια από αυτές. Επιγραμματικά αναφέρουμε τις παρακάτω λειτουργίες:

- Μετατροπή των τύπων της Tony στους αντίστοιχους τύπους της llvm και των σταθερών στις αντίστοιχες σταθερές της llvm.
- Ορισμός των llvm συναρτήσεων με τα κατάλληλα ορίσματα. Στα ορίσματα πέραν αυτών που υπάρχουν στον ορισμό της συνάρτησης, εισάγουμε και μεταβλητές εξωτερικής εμβέλειας οι οποίες χρησιμοποιούνται εσωτερικά.
- Υλοποίηση custom συναρτήσεων (με χρήση Phi nodes), που πραγματοποιούν short-circuited εκδόσεις των λογικών πράξεων *or* και *and*.

Βελτιστοποίηση Ενδιαμέσου Κώδικα

Ο μεταγλωττιστής που αναπτύξαμε δίνει την δυνατότητα βελτιστοποίησης του παραγόμενου ενδιαμέσου κώδικα, άμα το επιλέξει ο χρήστης. Ορίζουμε μέσω της βιβλιοθήκης του *llvm* εναν *function pass manager* στον οποίο προσθέτουμε έτοιμα optimization passes όπως *global value numbering, instruction combining*, κτλπ.

Παραγωγή Εκτελέσιμου Αρχείου

Assembly Code Generation

Μέσω του μεταγλωττιστή llc, ο *LLVM* ενδιάμεσος κώδικας που έχει παραχθεί μέχρι τώρα μετατρέπεται σε assembly κώδικα. Επιλέγουμε σαν target αρχιτεκτονική την x86_64-pc-linux-gnu. Ακόμη, ο μεταγλωττιστής της Tony που αναπτύξαμε, δίνει την δυνατότητα να χρησιμοποιηθούν οι δυνατότητες του llc για παραγωγή βελτιστοποιημένου assembly κώδικα, εφόσον αυτό ζητηθεί μέσω του κατάλληλου flag κατά την μεταγλώττιση.

Garbage Collector (GC)

Χρησιμοποιούμε τον Garbage Collector του Hans Boehm ο οποίος λειτουργεί με την μέθοδο *mark and sweep* κατά την οποία ακολουθώνται όλοι οι δείκτες που βρίσκονται στο stack με σκοπό να βρεθούν όλες οι προσβάσιμες θέσεις μνήμης που βρίσκονται στο heap. Στην συνέχεια οι μη-προσβάσιμες μπαίνουν σε μια λίστα για την επαναχατανομή τους.

υλοποιούν μέρος ή ολόκληρες τις αντίστοιχες μεθόδους *sem()*. Μια εναλλακτική σχεδιαστική επιλογή θα ήταν η συνένωση των δύο διαδικασιών σε μία, κάτι που εμείς απορρίψαμε θέλοντας να διατηρίσουμε τις δύο διαδικασίες ξεχωριστές.

External Linking & Executable Generation

Κατά την παραγωγή του ενδιάμεσου κώδικα, για την κατανομή μνήμης (*memory allocation*) μέσω του GC και την αρχικοποίησή του είχαμε χρησιμοποιήσει τις συναρτήσεις *GC_malloc* και *GC_init* τις οποίες είχαμε δηλώσει στο llvm πως θα τις συνδέσουμε εξωτερικά. Ακόμη, το ίδιο είχαμε κάνει για τις συναρτήσεις της βασικής βιβλιοθήκης της Tony. Η σύνδεση του assembly αρχείου που έχει παραχθεί μέχρι τώρα με την assembly βασική βιβλιοθήκη του Α.Βενετόπουλου και τον garbage collector του Boehm γίνεται (στατικά) μέσω του *clang* μεταγλωττιστή (βλ. Σχήμα 1) ο οποίος παράγει και το τελικό τοπικά εκτελέσιμο αρχείο.