# Documentation

Μπάρμπα Παναγιώτα Νικολέττα - el18604

Ευαγγελάτος Ανδρέας - el18069

Περιεχόμενα :

# Features

1. 10 byte real variables
2. Type casting
3. Optimization with LLVM
4. Function Overloading

# Περιγραφή αρχείων του compiler

- `main.cpp`

  Ορισμός της main, συνάρτηση εισόδου και σκελετός του μεταγλωττιστή.

- `inc/tree.h`, `src/tree.c`

  Ορίζεται class για το τύπωμα δέντρων. Οι κόμβοι του AST την κληρονομούν. Χρησιμοποιείται για την οπτικοποίηση του συντακτικού δέντρου.

- `ast.hpp`

  Ορισμός των κόμβων του AST.

- `ast.cpp`

  Οι destructors των κλάσεων που ορίστηκαν στο `ast.hpp`

- `tojsonstring.cpp`

  Οι υλοποιήσεις των συναρτήσεων toJSONString για τους κόμβους του AST. Αυτή η συνάρτηση χρησιμοποιείται μόνο κατά το testing του parser για την σύγκριση συντακτικών δέντρων.

- `lexer.l`

  Ο lexer. Meta-program του flex.

- `lexer.hpp`

  Ορισμός των συναρτήσεων του lexer για χειρισμό από άλλα μέρη.

- `parser.y`

  Ο Parser. Meta-program του bison.

- `llvmhead.hpp`

  #include τα απαραίτητα για το llvm.

- `error.hpp`, `error.cpp`

  Συναρτήσεις για τον χειρισμό σφαλμάτων και ErrorInfo Class για τους κόμβους του AST.

- `general.hpp`, `general.cpp`

  Ορισμός καθολικών μεταβλητών του state και υλοποιήσεις συναρτήσεων χειρισμού ορισμάτων (flags) και εξόδου (print to stdout, asm, imm).

- `symbol.hpp`, `symbol.cpp`

Ορισμός και υλοποίηση symbol table και τύπων.

- `semantical.cpp`

  Ορισμός της sem (σημασιολογικής ανάλυσης) για τους κόμβους του AST.

- `codegen.cpp`

  Ορισμός της codegen (παραγωγή κώδικα και βοηθητικές) για τους κόμβους του AST.

Για τα υπόλοιπα αρχεία, δείτε το Testing

# Syntax

```
<program> ::= (<declaration>)+
<declaration> ::= <variable-declaration> | <function-declaration>
| <function-definition>
<variable-declaration> ::= <type> <declarator> ( “,” <declarator>)* “;”
<type> ::= <basic-type> ( “*” )*
<basic-type> ::= “int” | “char” | “bool” | “double”
<declarator> ::= <I> [ “[” <constant-expression> “]” ]
<function-declaration> ::= <result-type> <I> “(” [<parameter-List>]
“)” “;”
<result-type> ::= <type> | “void”
<parameter-List> ::= <parameter> ( “,” <parameter>)*
<parameter> ::= [ “byref” ] <type> <I>
<function-defintion> ::= <result-type> <I> “(” [<parameter-List>] “)”
“{” (<declaration>)* (<statement>)* “}”

<statement> ::= “;” | <expression> “;” | “{” (<statement>)* “}”
| “if” “(” <expression> “)” <statement> [ “else” <statement>]
| [<I> “:” ] “for” “(” [<expression>] “;” [<expression>] “;”
[<expression>] “)” <statement>
| “continue” [<I>] “;” | “break” [<I>] “;” | “return” [<expr>] “;”

<expression> ::= <I> | “(” <expression> “)” | “true” | “false” | “NULL”
| <int-const> | <char-const> | <double-const> | <string-literal>
| <I> “(” [<expression-list>] “)” | <expression> “[” <expression> “]”
| <unary-operator> <expression>
| <expression> <binary-operator> <expression>
| <unary-assignment> <expression>
| <expression> <unary-assignment>
| <expression> <binary-assignment> <expression>
| “(” <type> “)” <expression>
| <expression> “?” <expression> “:” <expression>
| “new” <type> [ “[” <expression> “]” ] | “delete” <expression>
<expression-list> ::= <expression> ( “,” <expression>)*

<constant-expression> ::= <expression>
<unary-operator> ::= “&” | “*” | “+” | “” | “!”
<binary-operator> ::= “*” | “/” | “%” | “+” | “” | “<” | “>” | “<=”
| “>=” | “==” | “!=” | “&&” | “||” | “,”
<unary-assignment> ::= “++” | “–”
<binary-assignment> ::= “=” | “*=” | “/=” | “%=” | “+=” | “=”
```

# Types

## Basic Types

- int (2 bytes)
- char (1 byte)
- double (10 bytes)
- bool (1 byte)

## Extra Types For Function Results

- void

## Type Constructors

- Pointer t*

# Semantics

- Κάθε πρόγραμμα πρέπει να ορίσει με συνάρτηση της μορφής *void main ()*
- Σε pointer variables που ορίζονται σαν arrays δεν μπορούν να γίνουν αναθέσεις.

```
int a[100], b[200];
a = b; //Wrong
```

- Pascal's scopes
- Κάθε έκφραση έχει μοναδικό τύπο (! εκτός του NULL που έχει τύπο t* ( $\forall \tau, \text{type}(\tau) \Rightarrow \text{NULL} : \tau*$), και κάθε έκφραση μπορεί να αποτιμηθεί σε ένα στοιχείο του τύπου της.
    - Αποτελέσματα της αποτίμησης μπορεί να είναι l-values ή r-values
- Δεν επιτρέπεται *(NULL)
- Τα ορίσματα για binary operations αποτιμώνται από τα δεξιά στα αριστερά.
- Μόνο οι τελεστές [] και * μπορούν να δώσουν l-values όπως περιγράφεται παρακάτω.

LVAL και RVAL είναι σύνολα από expressions (predicates)

## L-value Rules

$\Gamma$ context. $\text{LVAL}$ predicate.

1. $\Gamma \vdash x : \tau \wedge (\text{VARIABLE}(x) \vee \text{PARAMETER}(x)) \Rightarrow \text{LVAL}(x, \tau)$
    1. $\Gamma \vdash x : \tau \wedge \text{VARIABLE}(x) \Rightarrow \text{LVAL}(x, \tau)$
    2. $\Gamma \vdash x : \tau \wedge \text{PARAMETER}(x) \Rightarrow \text{LVAL}(x, \tau)$
2. $\Gamma \vdash x : \text{Pointer}(t) \Rightarrow *x : t \wedge \text{LVAL}(*x, t)$
3. $\Gamma \vdash x : \text{Pointer}(t) \wedge \Gamma \vdash e : \text{int} \Rightarrow x[e] : t \wedge \text{LVAL}(x[e], t)$

# R-Values Rules

RVAL predicate

1. $\Gamma \vdash \text{IntConst} : \text{int}$
2. $\Gamma \vdash \text{true} : \text{bool}$
3. $\Gamma \vdash \text{false} : \text{bool}$
4. $\Gamma \vdash \text{DoubleConst} : \text{double}$
5. $\Gamma \vdash \text{CharConst} : \text{char}$
6. $\forall \tau, \Gamma \vdash \text{NULL} : \text{Pointer}(\tau)$
7. $\Gamma \vdash \text{StringConst} : \text{Pointer(char)}$
8. $\Gamma \vdash x : \text{DefinedAsArray}(t) \Rightarrow \Gamma \vdash \text{RVAL}(x, Pointer)$
9. $\Gamma \vdash \text{LVALUE}(l, t) \Rightarrow \Gamma \vdash \&l : \text{Pointer}(t) \wedge \text{RVAL}(\&l, \text{Pointer}(t))$
10. $\Gamma \vdash x : \text{int} \Rightarrow \Gamma \vdash +x : \text{int}$
11. $\Gamma \vdash x : \text{int} \Rightarrow \Gamma \vdash -x : \text{int}$
12. $\Gamma \vdash x : \text{double} \Rightarrow \Gamma \vdash +x : \text{double}$
13. $\Gamma \vdash x : \text{double} \Rightarrow \Gamma \vdash -x : \text{double}$
14. $\Gamma \vdash x : \text{bool} \Rightarrow \Gamma \vdash !x : \text{bool}$
15. $\forall \circ \in \{+, -, *, /, \%\}, \Gamma \vdash x : \text{int} \wedge \Gamma \vdash y : \text{int} \Rightarrow \Gamma \vdash x \circ y : \text{int}$
16. $\forall \circ \in \{+, -, *, /\}, \Gamma \vdash x : \text{double} \wedge \Gamma \vdash y : \text{double} \Rightarrow \Gamma \vdash x \circ y : \text{double}$
17. $\forall t \forall \circ \in \{+, -\}, \Gamma \vdash x : \text{Pointer}(t) \wedge \Gamma \vdash y : \text{int} \Rightarrow \Gamma \vdash x \circ y : \text{Pointer}(t)$
18. $\forall t \forall \circ \in \{==, !-, >, <, >=, <=\}, \Gamma \vdash x : t \wedge \Gamma \vdash y : t \Rightarrow \Gamma \vdash x \circ y : \text{bool}$
19. $\forall \circ \in \{||, \&\&\}, \Gamma \vdash x : \text{bool} \wedge \Gamma \vdash y : \text{bool} \Rightarrow \Gamma \vdash x \circ y : \text{bool}$
20. $\forall p, \forall q, \Gamma \vdash x : p \wedge \Gamma \vdash y : q \Rightarrow \Gamma \vdash x, y : q$
21. $\forall t, \Gamma \vdash e : \text{bool} \wedge \Gamma \vdash x : t \wedge \Gamma \vdash y : t \Rightarrow \Gamma \vdash e ? x : y : t$
22. $\forall t, \Gamma \vdash \text{LVAL}(x, t) \wedge \Gamma \vdash y : t \Rightarrow \Gamma \vdash x = y : t$
23. $\forall op \in \{+, -, *, /, \%\}, \Gamma \vdash \text{LVAL}(x, int) \wedge \Gamma \vdash y : int \Rightarrow \Gamma \vdash x\ op = y : int$
24. $\forall op \in \{+, -, *, /\}, \Gamma \vdash \text{LVAL}(x, \text{double}) \wedge \Gamma \vdash y : \text{double} \Rightarrow \Gamma \vdash x\ op = y : \text{double}$
25. $\forall op \in \{+, -\} \forall t, \Gamma \vdash \text{LVAL}(x, \text{Pointer}(t)) \wedge \Gamma \vdash y : \text{int} \Rightarrow \Gamma \vdash x\ op = y : \text{Pointer}(t)$
26. $\forall op \in \{++, --\}, \Gamma \vdash \text{LVAL}(x, \text{int}) \Rightarrow \Gamma \vdash x\ op : \text{int} \wedge \Gamma \vdash op\ x : \text{int}$
27. $\forall op \in \{++, --\}, \Gamma \vdash \text{LVAL}(x, \text{double}) \Rightarrow \Gamma \vdash x\ op : \text{double} \wedge \Gamma \vdash x\ op : \text{double}$
28. $\forall op \in \{++, --\} \forall t, \Gamma \vdash \text{LVAL}(x, \text{Pointer}(t)) \Rightarrow \Gamma \vdash x\ op : \text{Pointer}(t) \wedge \Gamma \vdash op\ x : \text{Pointer}(t)$
29. $\forall n, \forall t_n, \forall a_n, \forall q, (\Gamma \vdash f : (a_n\ t_n \rightarrow q), (a_i = \text{byref} \Rightarrow \Gamma \vdash \text{LVAL}(\text{x}_\text{i}, \text{t}_\text{i})), (a_i = \text{bycall} \Rightarrow \Gamma \vdash x_i : t_i)) \Rightarrow \Gamma \vdash f(x_1, x_2, ..., x_n) : q$
30. $\forall t, \Gamma \vdash e : \text{int} \Rightarrow \Gamma \vdash \text{new } t[e] : \text{Pointer}(t)$

31. $\forall t, \Gamma \vdash \text{new } t : \text{Pointer}(t)$
32. $\forall t, \Gamma \vdash e : \text{Pointer}(t) \Rightarrow \Gamma \vdash \text{delete } e : \text{Pointer}(t)$

## Statement Semantics

1. $\Gamma \vdash e : \text{bool} \Rightarrow \Gamma \vdash \text{if } e\ s_1 \text{else } s_2$
2. $\Gamma \vdash e_2 : \text{bool} \Rightarrow [\text{label} :] \text{for}(e_1; e_2, e_3)\ s$

   Αν η $e_2$ δεν δίνεται είναι true.

3. Το $\text{label}$ θα χρησιμοποιηθεί σαν τύπος στο Symbol Table.
    1. Στα forloops κατά την σημασιολογική ανάλυση μπαίνει στο symbol table ως active label. Γίνεται η σημασιολογική ανάλυση του εσωτερικού της loop. Όταν τελειώσει η σημασιολογική ανάλυση του σώματος γίνεται inactive στο symbol table η label. (Αυτό γίνεται διότι πρέπει να είναι μοναδικό το όνομα του label στο σώμα μίας συνάρτησης αλλά μπορεί να χρησιμοποιηθεί μόνο εντός της loop στην οποία δηλώθηκε)
    2. Στα break && continue που έχουν label πρέπει να ελεγχθεί αν είναι active το label για να είναι valid. Αλλιώς ορίστηκε σε άλλη loop και είναι σημασιολογικό λάθος.
4. $\text{return } e;$ Πρέπει να είναι εντός συνάρτησης. Αν ο τύπος επιστροφής της συνάρτησης είναι void πρέπει η $e$ να παραλείπεται. Αν δεν είναι void πρέπει $e : \text{return\_type}$

# Translating to LLVM

## Types to LLVM Types

| Edsger | LLVM |
|---|---|
| int | i16 |
| char | i8 |
| bool | i8 |
| double | x86_fp80 |
| t* | t* |
| t* (array) | [size * element type] |

## Global Variables

Τα global variables μπορεί να είναι όλοι οι τύποι (εκτός από void) γίνονται define αυτούσιοι. Θεωρούνται αυτόματα pointers στις παρακάτω δηλώσεις

```
@var = global type
```

## Local Variables

```
%var = alloca type ; this puts it in stack
```

Άμεσο για int, bool, char, double, t*.

Για array :

```
%ptrtoArray = alloca insideType, insideType size
```

Τελικά, γίνεται ένα alloca στην είσοδο κάθε συνάρτησης όπως περιγράφεται στην Nested Functions

## Constants

Αυτά μπορούν να εμφανιστούν μόνο σε expressions. Όλα είναι άμεσες σταθερές που περνάνε ως έχουν στο evaluation των παρακάτω. Μόνο τα strings έχουν διαφορά. Γίνονται initialize σε global array στην τιμή που χρειάζεται και επιστρέφεται ο δείκτης στο πρώτο στοιχείο τους (char*). Ορίζονται σαν global variables me private linkage.

Παράδειγμα

```
@.str = private global [13 x i8] c"Hello World!\00", align 1
```

Το 0 μπαίνει στο τέλος.

## Casts

Participating to a cast : [int, char, double, bool, pointer]

| From | To | Type of Cast |
|---|---|---|
| int | bool | truncating downcast |
| int | char | truncating downcast |
| int | double | sitofp instruction in llvm |
| int | pointer | inttoptr instruction in llvm |
| bool | int | unsigned upcast (zero-extend) |
| bool | char | bitcast |
| bool | double | uitofp instruction in llvm |
| bool | pointer | inttoptr instruction in llvm |
| char | int | signed upcast (sign-extend) |
| char | bool | bitcast |
| char | double | sitofp instruction llvm |
| char | pointer | inttoptr instruction in llvm |
| double | int | fptosi .. to llvm instruction |
| double | bool | fptoui .. to llvm instruction |
| double | char | fptosi .. to llvm instruction |
| double | pointer | fptoui & inttoptr |
| pointer | int | ptrtoint instruction in llvm |
| pointer | bool | ptrtoint instruction in llvm |
| pointer | char | ptrtoint instruction in llvm |
| pointer | double | ptrtoint & uitofp |
| pointer | pointer | bitcast (to different type*) |

## Function Overloading

Τα ονόματα των συναρτήσεων επεκτείνονται κατά τον εσωτερικό χειρισμό από τον compiler με πληροφορίες για τους τύπους των ορισμάτων. Έτσι για παράδειγμα οι

```
bool prime(int n);
bool prime(char c);
```

εσωτερικά παίρνουν τα ονόματα *prime_int* και *prime_char*.

Για αυτό το λόγο οι εξωτερικές συναρτήσεις πρέπει να χρησιμοποιούν το ίδιο naming convention. Η βιβλιοθήκη της Edsger έχει γραφεί σε C ακολουθώντας την σύμβαση (στον φάκελο `lib`)

## Nested Functions

Οι συναρτήσεις βρίσκονται στο Global Scope ορίζονται (σύμφωνα με την σύμβαση ονομάτων) όπως στον πηγαίο κώδικα ως προς τα ορίσματα τους. Οι ένθετες συναρτήσεις παίρνουν ένα επιπλέον όρισμα στον κώδικα του llvm (το prevEnv) που είναι ο σύνδεσμος για τις μεταβλητές της συνάρτησης που βρίσκεται ένα παραπάνω στατικό επίπεδο φωλιάσματος. Κάθε συνάρτηση έχει μία μεταβλητή (env) και αν αυτή είναι ένθετη αναθέτει χώρο για την αποθήκευση του (prevEnv) με το οποίο κλήθηκε.

Για το env ισχύει η ακόλουθη σύμβαση :

```
env -> previous static env # memory grows down here
       local variable 0
       local variable 1
       local variable 2
       ...
```

Άρα για να πάρει μία συνάρτηση την τιμή μίας μεταβλητή που βρίσκεται δύο στατικά επίπεδα πάνω αρκεί να ακολουθήσει τα παρακάτω βήματα.

```
prevEnv  = env[0];
prevEnv2 = prevEnv[0];
varPtr   = prevEnv2 + variableOffset
varVal   = *varPtr;
```

Κατά την κλήση μίας ένθετης συνάρτησης η καλούσα συνάρτηση δίνει το σωστό prevEnv στην συνάρτηση που καλείται.

Παράδειγμα

```
void main(){
   void x(int n){
      ...
   }
   ...
   x(10)
   ...
}
```

Γίνεται

```
define void @x_int(i8* %prevEnv, i16 %n) {
entry:
  ; Allocate space for current env
  %env = alloca i8, i64 10
  ; save prevEnv in env+0
  %env.0 = getelementptr i8, i8* %env, i64 0
  %env.0.casted = bitcast i8* %env.0 to i8**
  store i8* %prevEnv, i8** %env.0.casted
  ; save n in env+sizeof(env)
  %env.01 = getelementptr i8, i8* %env, i64 8
  %env.0.casted2 = bitcast i8* %env.01 to i16*
  store i16 %n, i16* %env.0.casted2
  ...
}

define void @_main(){
entry:
  env = alloca i8, i64 n
  ...
  call void @x_int(i8* %env, i16 10)
  ...
}
```

### Exiting to System

Κάθε πρόγραμμα πρέπει να ορίσει μία συνάρτηση *void main()*. Ο μεταγλωττιστής παράγει τον κώδικα για αυτή την συνάρτηση με όνομα *_main* και την καλεί από μία άλλη συνάρτηση με όνομα *main* η οποία επιστρέφει *int 0*.

## Invariants

### Semantic Invariants

- Η Function Stack βρίσκεται σε global vector.
  - Όταν καλείται newFunction το entry για την συνάρτηση γίνεται push στην στοίβα
  - Όταν καλείται closeScope() το entry για την συνάρτηση γίνεται pop από την στοίβα (αν υπήρχε τέτοιο γιατί η close scope καλείται και στο τέλος του global scope).
- Όλες οι For Loops έχουν ένα μοναδικό label (μοναδικό όνομα) με το οποίο σχετίζονται μετά την σημασιολογική ανάλυση.
- Όλα τα Continues, Breaks έχουν ένα non NULL label στο οποίο δείχνουν μετά την σημασιολογική ανάλυση.
- Το scope nesting level για το σώμα μίας συνάρτησης είναι πάντα > 1. Το Global scope έχει nesting level 1.
- Non Nested Function Declarations can be defined externally by using the same name convention

## Codegen Invariants

- Τα Basic Blocks for Statements and Expressions όπου πρέπει να ξεκινήσουν να γράφουν κώδικα είναι ήδη ορισμένα πριν από την κλήση της codegen σε αυτά.

## Calling And Env

- Σε κατωτερες nested συναρτησεις δίνεται πάντα το local env της καλούσας συνάρτησης.
- Πιθανά offsets κατά την κλήση συναρτήσεων
    1. +1 local nested function $\rightarrow$ Κλήση με local env
    2. 0 Αναδρομή ή nested function στο ίδιο επίπεδο $\rightarrow$ Κλήση με το env που κλήθηκε η τωρινή συνάρτηση. |nestingLevelOfTheCalle - youNestingLevel| + 1.
    3. <0 Συναρτήσεις στις οποίες η τωρινή συνάρτηση είναι ένθετη $\rightarrow$ Κλήση με το env ανεβαίνοντας κατά |nestingLevelOftheCalle - yourNestingLevel| + 1.
- Στο symbol table για κάθε συνάρτηση υπάρχει το context της.

# Testing

Για κάθε φάση του compiler γίνεται ξεχωριστό testing από διαφορετικές `main.cpp` που βρίσκονται στους φακέλους `tests/<phase>`

## Lexer Testing

Για το testing του Lexer χρησιμοποιήθηκαν μόνο απλά γραμμένα tests που είτε πρέπει να πετύχουν είτε πρέπει να αποτύχουν, ελέγχοντας σε αυτά που πρέπει να πετύχουν πως παράγονται τα σωστά tokens.

Ο φάκελος `tests/lexer/` περιλαμβάνει

- `tests/lexer/main.cpp` διαβάζει το αρχείο και τυπώνει τα token και τα lexemes. Το εκτελέσιμο για αυτήν την main λέγεται *lexertest* και πρέπει να φτιαχτεί από την make test στο αρχικό directory του ggec.
- hard coded tests στον φάκελο `tests/lexer/programs`
- Τα επιθυμητά αποτελέσματα με ίδια ονόματα και άλλη κατάληξη στον `tests/lexer/results`
- `tests/lexer/runner.sh` (σε python) τρέχει τα tests καλώντας το *lexertest* και συγκρίνει τα αποτελέσματα με αυτά στον φάκελο των αποτελεσμάτων. Πρέπει να κληθεί από την make test.

## Parser Testing

Για το testing του parser χρησιμοποιήθηκαν έτοιμα γραμμένα προγράμματα και το συντακτικό δέντρο που παραγόταν από αυτά ελεγχόταν ως προς την ισότητα με το αναμενόμενα. Επιπλέον, χρησιμοποιήθηκε ο syntax_gen που βρίσκεται στο `examples/syntax_gen`, που λήφθηκε από το (https://github.com/kostis/ntua_compilers), για έλεγχο σταθερότητας και ορθότητας του parser. Ο φάκελος αυτός έχει διαφοροποιηθεί. Έχει αλλάξει το Makefile και έχει προστεθεί `tests/examples/gen.sh` που χρησιμοποιείται για την αυτοματοποιημένη παραγωγή πολλών συντακτικά ορθών προγραμμάτων.

Ο φάκελος `tests/parser/` περιλαμβάνει

- `tests/parser/main.cpp` διαβάζει το αρχείο και τυπώνει το συντακτικό δέντρο σε JSON (χρησιμοποιεί την συνάρτηση toJSONString). Το εκτελέσιμο για αυτήν την main λέγεται *parsertest* και πρέπει να φτιαχτεί από την make test στο αρχικό directory του ggec.
- `customjsonbuilder.py` είναι ένα βοηθητικό πρόγραμμα που χρησιμοποιήθηκε για την γρήγορη γραφή των αναμενόμενων αποτελεσμάτων για τα προγράμματα εισόδου.
- hard coded tests στον φάκελο `tests/parser/programs`
- Τα επιθυμητά αποτελέσματα με ίδια ονόματα και άλλη κατάληξη στον `tests/parser/results`
- `tests/parser/runner.sh` (σε python) τρέχει τα tests καλώντας το *parsertest* και συγκρίνει τα αποτελέσματα με αυτά στον φάκελο των αποτελεσμάτων. Πρέπει να κληθεί από την make test.
- `tests/parser/runner_gen.sh` (σε python) τρέχει τα tests καλώντας το *parsertest* με είσοδο τα παραγόμενα προγράμματα από το *syntax_gen* και συγκρίνει ότι τερματίζει ορθά ο parser. Πρέπει να κληθεί από την make test.

## Semantics Testing

Για το testing της σημασιολογικής ανάλυσης, όπως και για του parser, χρησιμοποιήθηκαν έτοιμα γραμμένα σημασιολογικά ορθά ή μη προγράμματα και ελέγχθηκε ότι ο σημασιολογικός αναλυτής έχει την αναμενόμενη συμπεριφορά, και επιπλέον ένα πρόγραμμα που παράγει σημασιολογικά ορθά προγράμματα και βρίσκεται στον φάκελο `EdsgerProgramGenerator`. Είναι γραμμένο σε Haskell και γι' αυτό χρειάζεται η haskell stack κατά την εκτέλεση των tests.

Ο φάκελος `tests/semantics/` περιλαμβάνει

- `tests/semantics/main.cpp` διαβάζει το αρχείο κάνει την σημασιολογική ανάλυση (και την παραγωγή κώδικα στην περίπτωση που αυτή πετύχει, αν και αυτή αγνοείται). Το εκτελέσιμο για αυτήν την main λέγεται *semanticstest* και πρέπει να φτιαχτεί από την make test στο αρχικό directory του ggec.
- hard coded tests στον φάκελο `tests/semantics/programs`
- Τα επιθυμητά αποτελέσματα (επιτυχία ή αποτυχία) με ίδια ονόματα και άλλη κατάληξη στον `tests/semantics/results`
- `tests/semantics/runner.sh` (σε python) τρέχει τα tests καλώντας το *semanticstest* και συγκρίνει ανάλογα με αυτά στον φάκελο των αποτελεσμάτων αν η σημασιολογική ανάλυση έπρεπε να πετύχει ή να αποτύχει. Πρέπει να κληθεί από την make test.
- `tests/semantics/runner_gen.sh` (σε python) τρέχει τα tests καλώντας το *semanticstest* με είσοδο τα παραγόμενα προγράμματα από το `EdsgerProgramGenerator` και συγκρίνει ότι τερματίζει ορθά ο σημασιολογικός αναλυτής. Πρέπει να κληθεί από την make test.

## Codegen Testing

Για το testing της παραγωγής κώδικα, για κάθε test παράγεται το εκτελέσιμο αφού γίνει link με την βιβλιοθήκη της Edsger και της απαραίτητες της C που αυτή χρειάζεται. Το ίδιο test έχει γραφεί στην C-gnu11 (γι αυτό χρειάζεται και ο gcc για το testing) που υποστηρίζει nested functions. Τα προγράμματα γραμμένα στην C γίνονται και αυτά τελικά link με την libEdsger. Εκτελούνται τα δύο εκτελέσιμα παράλληλα και ελέγχεται ότι η έξοδός τους είναι η ίδια.

Ο φάκελος `tests/codegen/` περιλαμβάνει

- `tests/codegen/main.cpp` έχει ίδια λειτουργία με τον `ggec/main`. Το εκτελέσιμο για αυτήν την main λέγεται *codegentest* και πρέπει να φτιαχτεί από την make test στο αρχικό directory του ggec.
- hard coded tests στον φάκελο `tests/codegen/programs`
- Τα ίδια tests γραμμένα σε C στον φάκελο `tests/codegen/inC` χρησιμοποιώντας αντίστοιχους τύπους όπως long doubles για πραγματικούς και int16_t για ακεραίους και άλλες αντίστοιχες μετατροπές όπου αυτές χρειάζονται.
- Οι είσοδοι των εκτελέσιμων στον φάκελο `tests/codegen/inputs`.
- Το `tests/codegen/compile.sh` έχει παρόμοια λειτουργία με αυτή του `ggec/compile.sh` και παράγει εκτελέσιμο αν του δωθεί
- `tests/codegen/runner.sh` (σε python) τρέχει τα tests καλώντας το `tests/codegen/compile.sh` για την παραγωγή του εκτελέσιμου. Στην συνέχεια τρέχει το εκτελέσιμο και το αντίστοιχο που παράγεται από την C και ελέγχει ότι η έξοδος είναι ίδια. Πρέπει να κληθεί από την make test.