

הנדון: דו"ח פרויקט – שפות תכנות

מחלקה: הנדסת תוכנה

פרטי המגישים:

* עידן נוישול 207813635
* ספיר גילני 322358284

קורס שפות תכנות

(10211)

תוכן עניינים

[1. הקדמה 3](#_Toc175071168)

[2. BNF – תיאור השפה והדקדוק 3](#_Toc175071169)

[3. איך להריץ – מדריך קצר 6](#_Toc175071170)

[4. תיאור חלקי המערכת 6](#_Toc175071171)

[4.1 תיאור Use Case Diagram של ה-Interpreter 7](#_Toc175071176)

[4.2 תיאור ה-Lexer 7](#_Toc175071177)

[4.3 תיאור ה-Parser 9](#_Toc175071178)

[4.4 תיאור ה-Interpreter 10](#_Toc175071179)

[5. Design Decisions 11](#_Toc175071180)

[5.1 תיאור חלקי המערכת והחלטות העיצוב 11](#_Toc175071182)

[5.2 הנחות ומגבלות של השפה 11](#_Toc175071183)

[6. Cheat Sheet 12](#_Toc175071184)

# הקדמה

המערכת מממשת interpreter עבור שפת התכנות שלנו. הקבצים הנמצאים ב-Git מחלוקים באופן הבא:

* PartA – תיקייה המכילה את קבצי ה-Interpreter.
  + קובץ GrammerText – המכיל את ה-BNF של השפה
  + קובץ ExampleTest.lambda – קובץ להפעלת הפקודות ב-File Mode של התוכנית.
  + קובץ Testing.py – יחידת בדיקות ל-Interpreter המשתמש ב-unittest.
* PartB – תיקייה המכילה את קבצי הפייתון של חלק ב' של הפרויקט.
* קובץ README.md – עבור הגיט.
* assets – תמונות עבור ה-README.md.
* קובץ pdf – מסמך תיעוד לפרויקט.

# BNF – תיאור השפה והדקדוק

תיאור מפורט של השפה – עם דוגמאות לפקודות תקינות.

הפונקציה הראשית ב-BNF המתארת את הפקודות המתאימות בשפה נקראת <language\_command>.

**<language\_command> ::= <func\_def> | <call\_func> | <comment> | <printed\_note> | <lambda> | <comp\_expression> | <TT\_EXIT>**

**<func\_def> ::= <TT\_FUNC> <func\_name> <TT\_FUNC> <args> <TT\_FUNC\_SIGN> <language\_command>**

**<call\_func> ::= <TT\_CALL\_FUNC> <func\_name> <nested\_func>+ | <TT\_CALL\_FUNC> <func\_name> <nested\_func>+**

**<arg\_value> ::= <arg\_value> <TT\_COMMA> <atom> | <atom>**

**<nested\_func> ::= <TT\_FUNC\_LBRACKET> <arg\_value> <TT\_FUNC\_RBRACKET> | <TT\_FUNC\_LBRACKET> <TT\_FUNC\_RBRACKET>**

**<atom> ::= <INT> | <TT\_STRING> | <bool>**

**<comment> ::= <TT\_COMMENT> <text>\***

**<text> ::= <TT\_STRING> | <INT> | <TT\_STRING> <text> | <INT> <text>**

**<printed\_note> ::= <TT\_PRINTED\_NOTE> <text>\***

**<lambda> ::= <TT\_LLAMBDA> <arg\_name> <TT\_LAMBDA\_SIGN> <language\_command> <TT\_RLAMBDA> <nested\_args>+**

**<nested\_args> ::= <TT\_LPAREN> <arg\_value> <TT\_RPAREN> | <TT\_LPAREN> <TT\_RPAREN>**

**<comp\_expression> ::= <TT\_NOT> <comp\_expression> | <second\_expression> <AND\_OR> <second\_expression> | <second\_expression>**

**<second\_expression> ::= <first\_expression> <BOOL\_OPS> <first\_expression> | <first\_expression> | <second\_expression> <BOOL\_OPS> <second\_expression>**

**<first\_expression> ::= <factor> <PLUS\_MINUS> <factor> | <factor> | <first\_expression> <PLUS\_MINUS> <first\_expression>**

**<factor> ::= <INT> | <bool> | <TT\_STRING> | <lambda> | <TT\_LPAREN> <comp\_expression> <TT\_RPAREN> | <factor> <MUL\_DIV\_MOD> <factor> | <call\_func> | <lambda>**

**<TT\_FUNC> ::= <whitespace>\* "$" <whitespace>\***

**<func\_name> ::= <whitespace>\* <TT\_STRING> <whitespace>\***

**<TT\_LPAREN> ::= <whitespace>\* "(" <whitespace>\***

**<TT\_RPAREN> ::= <whitespace>\* ")" <whitespace>\***

**<args> ::= <TT\_LPAREN> <arg\_name> <TT\_RPAREN> | <TT\_LPAREN> <TT\_RPAREN>**

**<arg\_name> ::= <TT\_STRING> <TT\_COMMA> <TT\_STRING> | <TT\_STRING>**

**<TT\_FUNC\_SIGN> ::= <whitespace>\* "=>" <whitespace>\***

**<TT\_LLAMBDA> ::= <whitespace>\* "[" <whitespace>\***

**<TT\_RLAMBDA> ::= <whitespace>\* "]" <whitespace>\***

**<TT\_LAMBDA\_SIGN> ::= <whitespace>\* ":" <whitespace>\***

**<TT\_CALL\_FUNC> ::= <whitespace>\* "@" <whitespace>\***

**<TT\_FUNC\_LBRACKET> ::= <whitespace>\* "{" <whitespace>\***

**<TT\_FUNC\_RBRACKET> ::= <whitespace>\* "}" <whitespace>\***

**<AND\_OR> ::= <TT\_AND> | <TT\_OR>**

**<TT\_AND> ::= <whitespace>\* "&&" <whitespace>\***

**<TT\_OR> ::= <whitespace>\* "||" <whitespace>\***

**<TT\_STRING> ::= <whitespace>\* <letters> <whitespace>\***

**<letters> ::= <small\_letter> <TT\_STRING> | <large\_letter> <TT\_STRING> | <small\_letter> | <large\_letter>**

**<small\_letter> ::= [a-z]+**

**<large\_letter> ::= [A-Z]+**

**<TRUE> ::= <whitespace>\* "True" <whitespace>\***

**<FALSE> ::= <whitespace>\* "False" <whitespace>\***

**<bool> ::= <TRUE> | <FALSE>**

**<TT\_COMMA> ::= <whitespace>\* "," <whitespace>\***

**<TT\_NOT> ::= <whitespace>\* "!" <whitespace>\***

**<TT\_COMMENT> ::= <whitespace>\* "#"**

**<TT\_PRINTED\_NOTE> ::= <whitespace>\* "##"**

**<TT\_EXIT> ::= <whitespace>\* "EXIT" <whitespace>\***

**<INT> ::= "-" <INT> | <number> | <float>**

**<float> ::= <number> "." <number>**

**<number> ::= <non\_digit\_zero> <digits>\* | <zero>**

**<digits> ::= <zero> | <non\_digit\_zero>**

**<non\_digit\_zero> ::= [1-9]+**

**<zero> ::= "0"**

**<PLUS\_MINUS> ::= <TT\_PLUS> | <TT\_MINUS>**

**<TT\_PLUS> ::= <whitespace>\* "+" <whitespace>\***

**<TT\_MINUS> ::= <whitespace>\* "-" <whitespace>\***

**<MUL\_DIV\_MOD> ::= <TT\_MUL> | <TT\_DIV> | <TT\_MODULO>**

**<TT\_MUL> ::= <whitespace>\* "\*" <whitespace>\***

**<TT\_DIV> ::= <whitespace>\* "/" <whitespace>\***

**<TT\_MODULO> ::= <whitespace>\* "%" <whitespace>\***

**<BOOL\_OPS> ::= <EE> | <NE> | <GT> | <GTE> | <LT> | <LTE>**

**<EE> ::= <whitespace>\* "==" <whitespace>\***

**<NE> ::= <whitespace>\* "!=" <whitespace>\***

**<GT> ::= <whitespace>\* ">" <whitespace>\***

**<LT> ::= <whitespace>\* "<" <whitespace>\***

**<GTE> ::= <whitespace>\* ">=" <whitespace>\***

**<LTE> ::= <whitespace>\* "<=" <whitespace>\***

**<whitespace> ::= " " | "\t"**

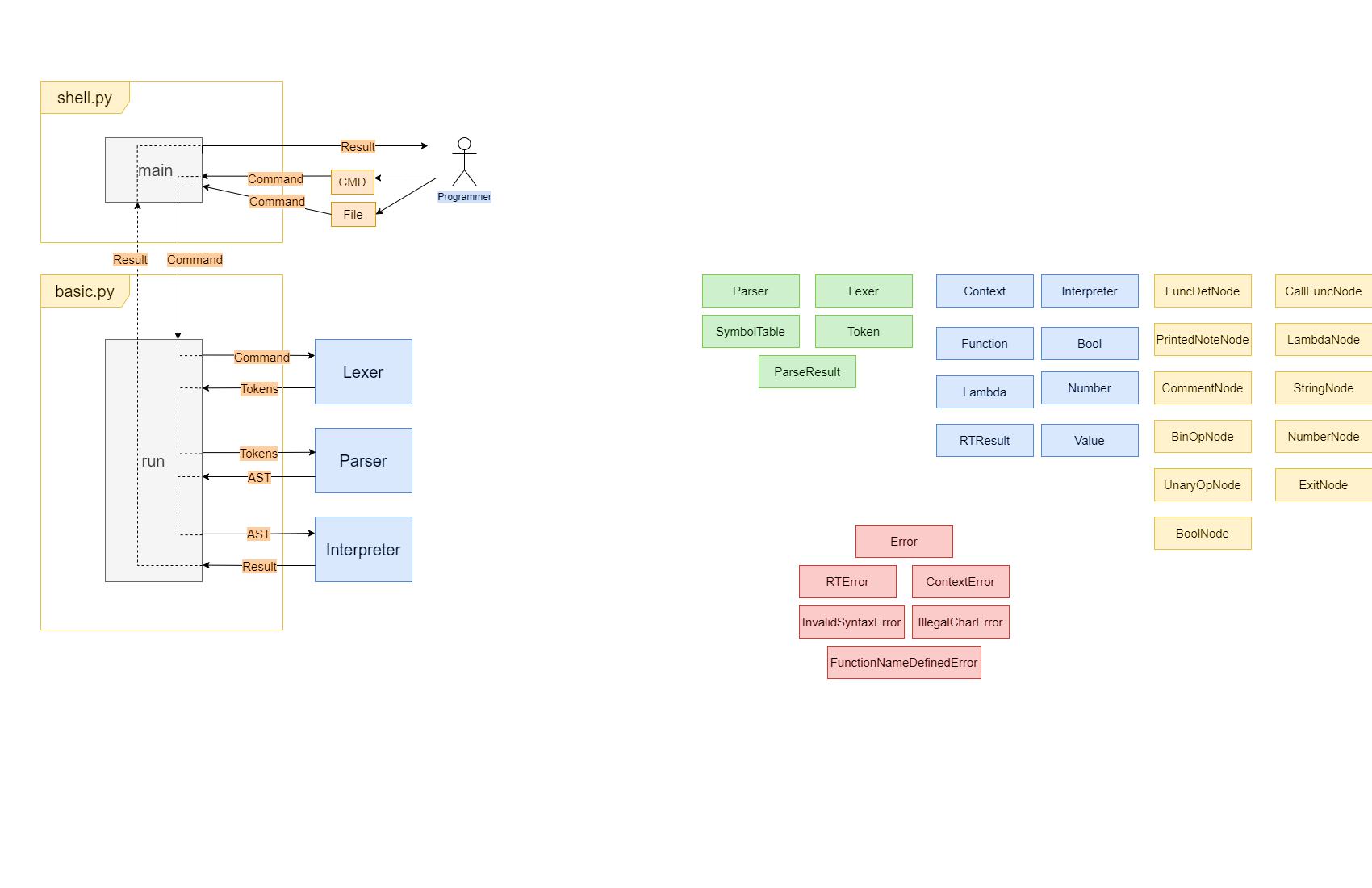
# איך להריץ – מדריך קצר

* להרצת התוכנית בטרמינל ב-Interpreter Mode:
  + python shell.py
* ליציאה ממצב Interpreter Mode:
  + EXIT
* להרצת התוכנית בטרמינל ב-File Mode:
  + python shell.py filename.lambda
* להרצת בדיקות התוכנית בטרמינל:
  + python Testing.py

# תיאור חלקי המערכת



## תיאור Use Case Diagram של ה-Interpreter

שרטוט המתאר את ה-Interpreter של שפת התכנות שלנו:

המתכנת המשתמש בשפה שלנו ומריץ פקודות ב-Interpreter יכול להריך בשני אופנים: פקודה-פקודה ב-Command Line או כתוכנית שלמה בקובץ נפרד המסתיים ב-\*.lambda המכיל את הפקודות הדרושות.

קובץ ה-shell.py הוא התוכנית המפעילה את ה-Interpreter ומעבירה את הפקודות המגיעות מהמתכנת לפונקציית run() שבקובץ ה-basic.py. הפקודה עוברת את שלבי ה-Lexer, ה-Parser, ולבסוף ה-Interpreter שיוסברו בסעיפים הבאים, עד אשר היא מופעלת ומוחזר התוצאה חזרה למשתמש.

## תיאור ה-Lexer

השלב הראשון בתהליך, בו מבוצע ניתוח מילולי ורצף התווים שהוכנסו על ידי המתכנת מתורגמים ל-token-ים המוכרים על ידי השפה. הסיווג בשפה שלנו וה-token-ים בהם השפה תומכת הם:

|  |  |  |  |
| --- | --- | --- | --- |
| **ביטויים אריתמטיים** | **פעולות בוליאניות** | **פעולות השוואה** | **הגדרת פונקציות** |
| INT  FLOAT (.)  PLUS (+)  MINUS (-)  MUL (\*)  DIV (/)  MODULO (%)  LPAREN ‘(‘  RPAREN ‘)’ | AND (&&)  OR (||)  NOT (!)  BOOL (‘True’, ‘False’) | EQ (=)  EE (==)  NE (!=)  LT (<)  GT (>)  LTE (<=)  GTE (>=) | FUNC ($)  FUNC\_NAME  FUNC\_ARGS  FUNC\_SIGN (=>)  FUNC\_BODY |
|  |  |
|  |  |  |
|  |  |  |
| **קריאה לפונקציות** | **Lambda** | **אחר** |  |
| CALL\_FUNC (@)  FUNC\_LBRACKET ‘{‘  FUNC\_RBRACKET ‘}‘ | LLAMBDA ‘[‘  RLAMBDA ‘]’  LAMBDA\_SIGN (:) | STRING  COMMA (,)  COMMENT (#)  PRINTED\_NOTE (##)  EOF  EXIT (‘EXIT’) |  |
|  |
|  |
|  |  |
|  |  |  |

**מספר הערות:**

* השפה תומכת בסימנים (המתוארים מעלה), באותיות a-z,A-Z, במספרים חיובים ושליליים.
* ה-token הנקרא STRING הוא עבור כל רצף אותיות עוקבות. מכיוון שהשפה לא תומכת במחרוזות, המשמעות כאן היא עבור שמות של פונקציות, שמות של משתנים מקומיים בפונקצייה או ב-Lambda, הערות או הערות להדפסה, ושמות שמורים כמו True, False, EXIT.
* הפקודה EXIT נועדה לשימוש ב-Interpreter mode להרצת פקודות אחת אחרי השנייה ב-command line ועבור יציאה מהתוכנית ניתן לכתוב EXIT.
* ה-token הנקרא EOF נועד עבור סימון של סוף הפקודה.

**דוגמה להמרת פקודה לרשימת token-ים:**

* הפקודה: (3-1)\*5
* [INT:5, MUL, LPAREN, INT:3, MINUS, INT:1, RPAREN]

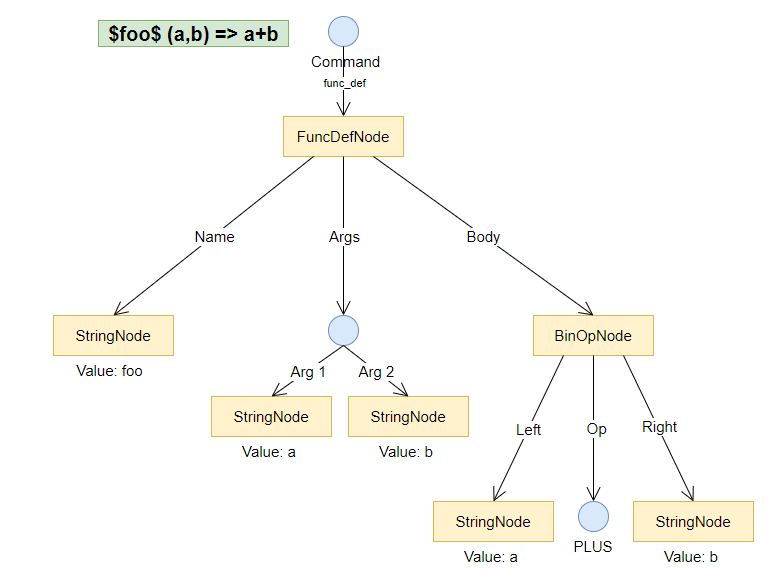
## תיאור ה-Parser

בשלב השני בתהליך, מבוצע ניתוח מחרוזות של הסמלים וה-token-ים וסיווגם לפי אובייקטים ומרכיבים המתאימים לקשר ביניהם. הפקודות בשפה שלנו מפורשים למספר סוגי פקודות והם:

* הגדרה של פונקציה
* קריאה לפונקציה
* הערות בקוד
* הערות מודפסות
* למבדות
* יציאה מה-interpreter
* ביטויים בוליאנים/אריתמטיים ושילובם

אובייקטים אותם ה-parser בונה הם:

|  |  |
| --- | --- |
| * FuncDefNode * CallFuncNode * LambdaNode * CommentNode * PrintedNoteNode * StringNode | * NumberNode * BinOpNode * UnaryOpNode * BoolNode * ExitNode |

דוגמה לבניית AST (Abstract Syntax Tree) על ידי ה-Parser עבור הגדרת פונקציה חדשה:

לאחר בניית ה-AST, ה-parser מחזיר אובייקט ParseResult שבמידה ופעולת הפרסור הצליחה חוזר האוביקט המתאים עם הערך המפורסר, אחרת, חוזרת שגיאה.

## תיאור ה-Interpreter

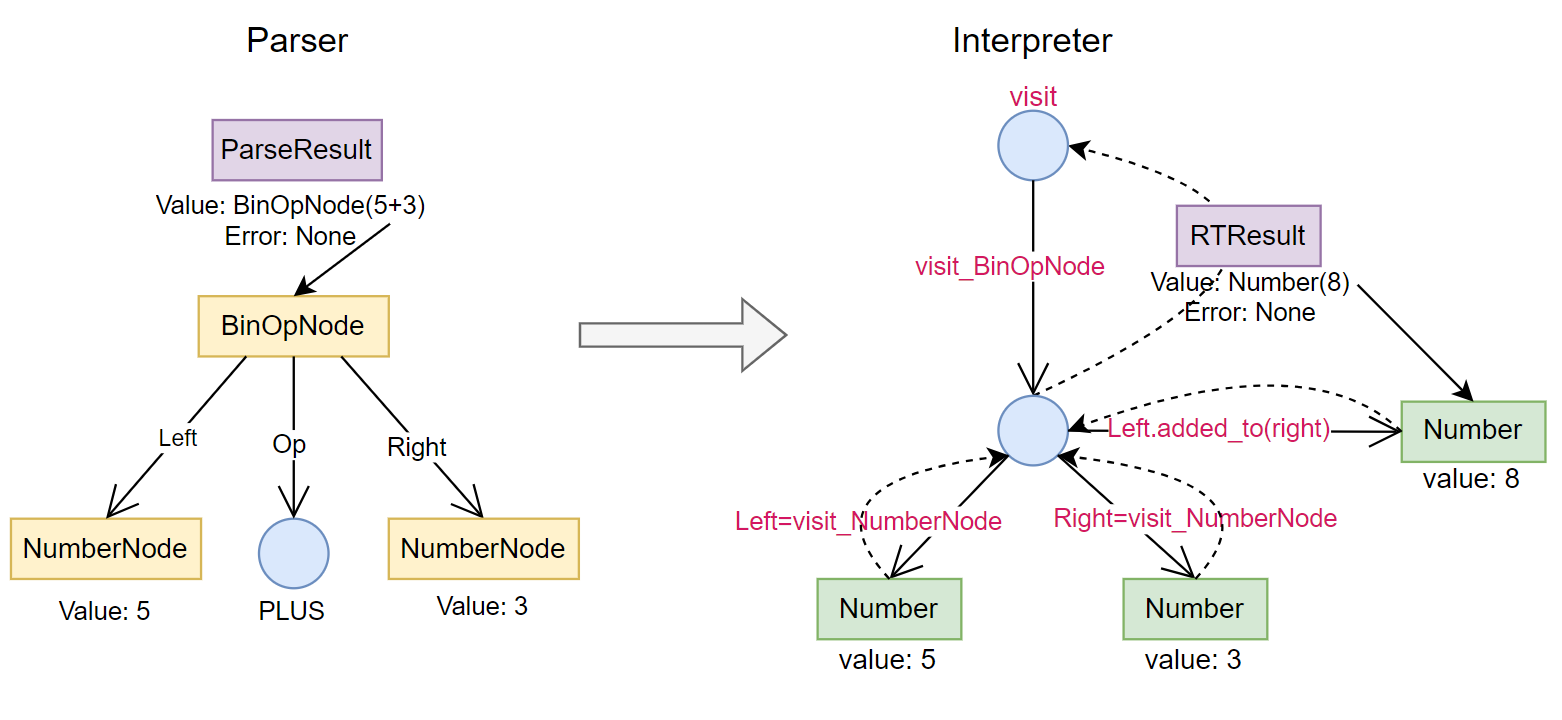
בשלב האחרון של התהליך, האוביקט שחזר מתוצאת הפרסור מגיע ל-Interpreter, לפונקציה הראשית הנקראת visit. פונקציה זו אחראית על הפעלת המתודות המקמפלות את האוביקט המתאים. לכל סוג של אוביקט קיימת פונקציה התואמת את שמה (visit\_{ObjectName}). כך למשל, עבור האוביקט FuncDefNode, קיימת הפונקציה visit\_FuncDefNode האחראית על קמפול הפקודה. בהתאם לכל אוביקט וכל סוג פקודה, ייבנה לבסוף האוביקט שיכיל את הערך המתאים ואת התוצאה הסופית.

האוביקטים שבהם ה-interpreter תומך הם:

* Number
* Bool
* Function
* Lambda

ה-interpreter מחזיר אוביקט RTResult שבמידה ופעולת הקמפול הצליחה, חוזר האוביקט המתאים עם תוצאת הקמפול, אחרת, חוזרת שגיאה.

דוגמה עבור הפקודה 5+3:

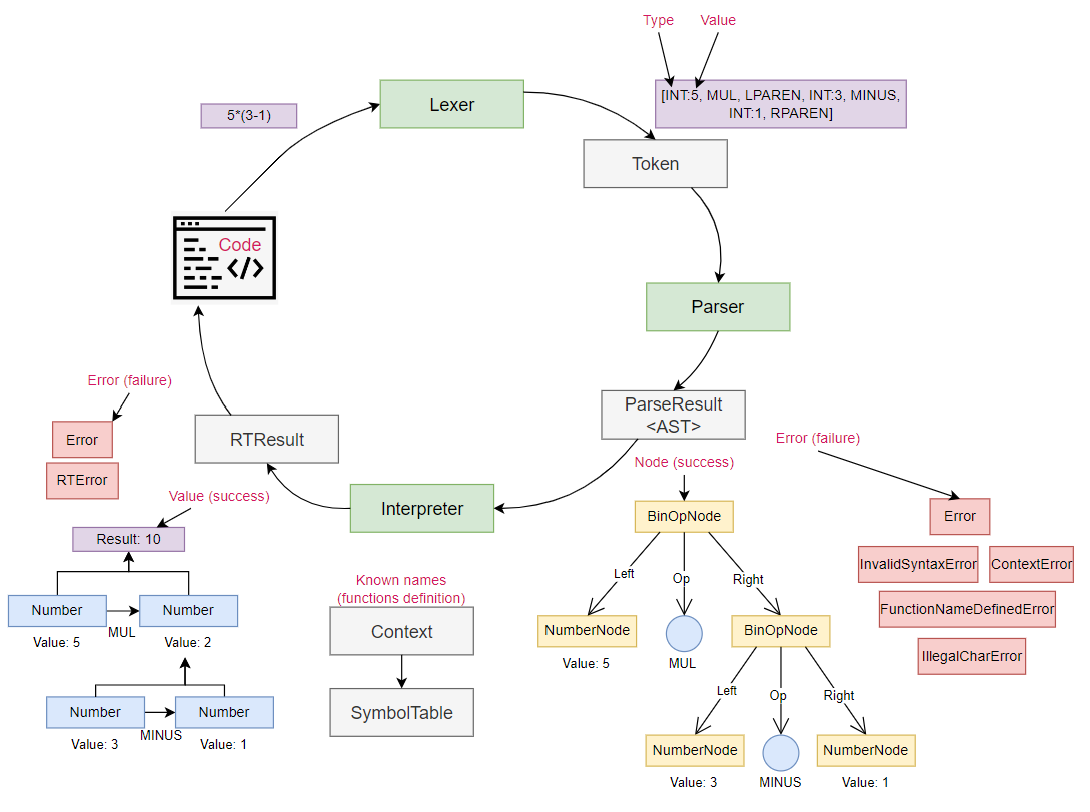


צד שמאל, זה תוצאת ה-parser המועברת לעיבוד אצל הן-interpreter בצד ימין. תוצאה המוחזרת זה אוביקט מסוג Number שערכו 8 בערך ה-value של ה-RTResult כאשר אין אף שגיאה. במידה וקורית שגיאה, מוחזר אוביקט Error וה-value הוא ריק.

# Design Decisions



## תיאור חלקי המערכת והחלטות העיצוב

שרטוט חלקי המערכת והמעברים:

## הנחות ומגבלות של השפה

* להפעלת פונקציה בתוך פעולה בינארית יש להוסיף סוגריים עגולים
  + למשל: (@a{4}) + 3
* להפעלת למבדה פנימית בתוך למבדה יש להוסיף סוגריים עגולים,
  + למשל: [x:([y: y+4](2)) \* x](3)
* שמות של פונקציות ושל local arguments הם באותיות גדולות או קטנות ללא סימנים נוספים.

# Cheat Sheet

|  |  |
| --- | --- |
| **Comparison** | **Data Types** |
| |  |  |  |  | | --- | --- | --- | --- | | *Op* | *meaning* | *True* | *False* | | == | equal | 2==2 | 2==3 | | != | not equal | 3 != 2 | 2 != 2 | | < | less than | 2 < 13 | 2 < 2 | | <= | less than or equal | 2 <= 2 | 3 <= 2 | | > | greater than | 13 > 2 | 2 > 13 | | >= | greater than or equal | 3 >= 2 | 2 >= 3 | | |  |  |  |  | | --- | --- | --- | --- | | *type* | *Set of values* | *operators* | *sample* | | Int | Integers | + - \* / % | 1 -2 33 | | Float | Floating point numbers | + - \* / % | 3.14 -2.5 | | Bool | Boolean values | && || ! | True False | | String | Function names and local arguments |  | foo a b | |
| **Functions Declare** | **Lambda** |
| Function body  Function sign  Local arguments variable  Function name  Function declaration  $foo$ (a,b) => a+b | Local arguments values  Lambda body  Local arguments variable  Lambda declaration  [x,y:x\*5+y-2](10,12) |
| **Functions Call** | **Comments & Printed Notes** |
| Function Call  Function name  Local arguments values  @foo{3,4} | ##for a printed note  #for a comment |
| **How To Run** |
| |  |  | | --- | --- | | **File mode** | Shell.py ExampleTest.lambda | | **Interpreter mode** | Shell.py | |
| **Higher Order** |
| Arguments for aa  Arguments for bb  $bb$ () => $aa$ (x) => 1+x @bb{}{2} | **Exit Interpreter Mode** |
| EXIT |