# מבוא

## ייזום:

### תיאור ראשוני

הפרויקט הוא שפת תכנות חדשה שהמצאתי, ששמה "Little". הפרויקט כתוב בשפת התכנות Haxe, ומאפשר הרצה של קוד הכתוב ב-Little מצד המשתמש, ובמיוחד מאפשר למפתח המשתמש בפרויקט לאלמנטים באפליקציה שלו לעקוב בקלות אחרי כל אירוע שקורה, ולהוסיף לשפה פונקציונליות צד-שלישי, על מנת לשפר פרודוקטיביות באותה אפליקציה.

קיבלתי את ההשראה להכין פרויקט בסדר גודל כזה משני אלמנטים בספריית תכנות שלי, Texter:

* Parser של Markdown, שגם נותן אופציית ויזואליזציה של הMarkdown
* אלגוריתם החלטת כיוון וסידור טקסט למחרוזות המכילות טקסטים שאמורים לזרום לכיוונים שונים (לדוגמה: אנגלית ועברית, צרפתית וערבית), בכמה שורות/פסקאות שונות.

שני האלמנטים עירבו יצירה והחלטה על Tokens שייצגו את המידע, ומניפולציה על אותם Tokens, על מנת להציג את התוצאה הרצויה (טקסט שמוצג נכון ומסמך בסטייל Markdown, מהמקרים שהוזכרו קודם). כשלמדתי שקומפיילר/אינטרפרטר של שפת תכנות זה, בסופו של יום, אותו הדבר, רק בסדר גודל יותר רציני - התעניינתי, והתחלתי לנסות לתכנת שפה משלי...

ממש בהתחלה, חשבתי שזה יהיה יחסית פשוט – הנחתי שהתהליך יהיה ישיר בערך כמו הדברים שעשיתי בעבר, אבל תוך כדי עשייה, הבנתי שאני הולך להיתקל ביחסית הרבה מכשולים. העיקריים היו:  
 - זריקת שגיאות מובנות ומתאימות להקשר  
 - מערכת סוגים – מה עדיף - מערכת חזקה או מערכת חלשה?  
 - ניהול זיכרון – באילו מבנים אשתמש על מנת לאחסן זיכרון?  
 - וכמובן: איך בכלל מתמודדים עם כמות כזאת גדולה של נתונים, בלי לקחת הרבה זמן?

### הגדרת הלקוח

אפשר להבין ממה שהוזכר קודם, שהפרויקט מעוצב לשני סוגי לקוחות:

* **סוג ראשון: לקוחות רגילים** – אנשים שרק רוצים ללמוד את השפה ולתכנת בה. בשבילם, ישנה חלקה באתר שלי המאפשרת גישה לרוב היכולות של הפרויקט – שינוי מילות מפתח, הרצה של קוד, ואפילו אפשר לראות את ה-AST של הקוד שהקלדת בצורה שנעימה לעיניים
* **סוג שני: מפתחים** – מפתחים שרוצים לכלול יכולות פרוגרמתיות בתוכנה שלהם. בשביל להקל עליהם, הפרויקט מעוצב לפי תבנית הFacade, על מנת לפשט את האינטגרציה עם התוכנה, וגם לאפשר אותה באמצעות פחות "נפח" של קוד. מעבר לזה, הפונקציות העיקריות בפרויקט, מתועדות בצורה אקסטנסיבית ומסבירות בצורה טובה מה הן בדיוק עושות, ואפילו כוללות דוגמאות המראות מה קורה עם קלטים מסויימים.

### יעדים ומטרות

מההתחלה, החלטתי שהפרויקט יכוונן לשתי מטרות, יחסית ספציפיות, אך שימושיות:

* **ראשונה: יצירת שפת תכנות שקל ללמד וגם קל לשנות.** שפה כזאת יכולה בקלות להנגיש לימוד תכנות לילדים, ובמיוחד לילדים שלא בהכרח יודעים אנגלית. כאן לפרויקט שלי יש יתרון מובהק על פני אופציות אחרות זמינות: הפרויקט עוצב תוך כדי התחשבות בריבוי שפות, כך שכל keyword או אלמנט בStandard Library יכול להיות מוחלף על ידי כל מילה אחרת בכל שפה אחרת, דבר שמאפשר לתכנת לא רק בשפה האנגלית, אלא גם בשפות מדוברות אחרות, כמו עברית וערבית.
* **שנייה: שימוש ככלי פרודוקטיביות בתוך תוכנות קיימות** – כמו שהתוכנה Excel מאפשרת לשנות ערכים של משבצות בעזרת קוד הכתוב ב-Visual Basic Analysis, ככה תוכנות המשתמשות בLittle כספרייה יוכלו לספק דרכי אינטראקציה עם התוכנה באמצעות קוד הכתוב בLittle. גם כאן לפרויקט יש יתרון ברור, שכן לא רק שמצד הלקוח, כתיבת קוד בLittle זה דבר קל בגלל הsyntax הפשוט והעקבי של השפה, גם המפתח נהנה, מכיוון שחלק משמעותי מהפרויקט מתרכז בהקלה על הוספת אלמנטים חיצוניים לשפה, וזה יהיה מאוד קל ומהיר לשלב את הפרויקט בתוכנה של המפתח, ויחסוך ממנו המון כאב ראש.

### בעיות, תועלות וחסכונות

כשהתחלתי עם הפרויקט לפני כשנה, היה לי רק רצון אחד שהפרויקט ימלא, שלא קשור בהכרח לתפקוד שלו – **שיהיה אפשר לתכנת בו בכל שפת אם שהיא**. הסיבה לרצון הזה, קשורה לאתר-משחק ששיחקנו בו בבית הספר היסודי, שקראו לו Code Monkey, שבו המטרה היא להזיז קוף לעבר בננות באמצעות קוד, בכמה שפחות שורות. בעיה אחת הייתה לי עם המשחק הזה – הקוד היה באנגלית, ואני הייתי הרבה יותר קטן, ולא ידעתי אנגלית טוב - זה *מאוד* הגביל אותי, כי לקח לי מדי הרבה זמן לזכור מה כל דבר עושה, וככל שהצטברו הפונקציות והתכונות שהיה אפשר להשתמש בהם, המשחק הסתבך יותר ויותר...

על מנת לממש את הרצון שלי עם הפרויקט הזה, עצבתי אותו בדרך מסויימת, ונתתי גישה למגוון מערכות שונות, ולא רק שממשתי את הרצון שלי – לשמחתי, עשיתי הרבה מעבר...

**מערכות:**

* **אוסף מילות מפתח** – על מנת להקל על המְפַתח, ניתנת גישה לכל "מילות המַפְתח" (Standard Library, מילים שמורות), והמְפַתח יכול לשנות מילים ספציפיות, או להחליף בין סטים שלמים של מילים, כדי לשנות את וורבאליות הקוד, או השפה המדוברת בו היא כתובה. מערכת זו ממש הייתה מכוננת למימוש הרצון הראשוני שלי, של הנגשת תכנות לילדים שלאו דווקא יודעים אנגלית. דוגמה של המערכת בפעולה:

|  |  |
| --- | --- |
| הגדר מס = 13  הדפס({  מס = מס \* 5  אותיות.מקוד\_אות(מס)  }) """ מדפיס: "A" """ | define num = 13  print({  num = num \* 5  Characters.fromCharCode(e)  }) “””prints: “A” “”” |

**מילות מפתח ערוכות לעברית:** **ברירת מחדל:**

* **Parser מודולארי** – על מנת להוסיף על יכולות המערכת הראשונה, הParser של השפה מאפשר "הזרקה" ישירה של שלבי בניית AST, ואפילו נותן למפתח להמציא Tokens חדשים באופן דינמי. המערכת מאפשרת את זה על מנת לאפשר תמיכה בכל סוג של syntax מיוחד, דבר שאי אפשר לעשות ברוב שפות התכנות האחרות, שמרכיבות את כל הAST שלהן לפני שהן מביאות אותו לmacros, ולכן, ניתן להוסיף לשפה רק syntax שנחשב חוקי (קריא על ידי הParser של השפה) מראש. הסיבה לכך היא, שsyntax שנחשב שגוי בשפה, אך חוקי בmacro שבנינו, יזרוק שגיאה בקריאה הראשונית של הקוד אל תוך AST, ולא יגיע לmacro שלנו. דוגמה משפה קיימת (Haxe):
  + גישת מערך מרובת expressions, כמו array[3, 5 + 4], לשפה שמצפה לidentifier/expression יחיד בגישה למערך, שיראה כך:array[2 \* 6] , תזרוק שגיאה בסגנון Unexpected Identifier `,` ויפסיק את הParser, לפני שהוא מפעיל את הmacro שלנו, שיהפוך, לדוגמה, את array[3, 4 + 5] ל array[3][4 + 5]. על מנת להימנע ממקרים כאלה, ה-Parser של השפה מאפשר הכנסת macros ממש אל תוך פונקציית הparsing ולא אחריה, ובעזרת יכולת זו אפשר לדאוג שהmacro פועל לפני שהParser זורק את השגיאה, והmacro יעבוד כמצופה.
* **אגף להוספת Externs** – על מנת להקל על המפתח בשילוב הפרויקט בתוכנה שלו, ישנו אגף שלם הנועד רק להוספת "Plugins" שונות לשפה במגוון סוגים, ובמגוון מקרים - ממשתנים ופונקציות רגילות, לסוגים מיוחדים ושדות גלובליים. האגף הזה מורכב ממחלקה אחת ששמה Plugins, שמטרתה לאפשר הוספת אלמנטים לשפה בעזרת קוד הכתוב בHaxe, ופונקציה ששמה loadModule(), שמאפשרת להריץ קוד הכתוב בLittle מראש, או ממש לפני שהקוד של המשתמש רץ. דוגמה:

|  |  |
| --- | --- |
| Plugins | Little.plugin.registerVariable(  “currentTime”, “Characters”, () -> {  return Conversion.toLittleValue(  Date.now().toString()  ); // Or alternatively, the token:  // Characters(Date.now().toString())  }  ); |
| loadModule() | Little.loadModule(“  define attachedToProgram = true  define parentProgram = “My Program”  action mySemiExtern() = {  print(“Hello World”)  }  ”, “Externs”) |

### פתרונות קיימים

ישנן כבר שפות תכנות שתומכות בשימוש לצרכים שהפרויקט שלי מיועד אליהם. להלן, השוואות בין שפות תכנות וספריות/תוכנות שמנגישות אותן, והפרויקט שלי:

|  |  |  |  |
| --- | --- | --- | --- |
| שפת תכנות | **Hscript**  **(Haxe)** | **VBA  (Visual Basic)** | **פרויקט**  **(Little)** |
| **הכרזת משתנים/פעולות** | ✔️ | ✔️ | ✔️ |
| **הכרזת סוגים** | ✔️ | ✔️ | ✔️❌\* |
| **לולאות, תנאים** | ✔️\*\* | ✔️ | ✔️ |
| **Compilation Macros** | ✔️ | ❌ | ✔️ |
| **גישה לאירועי הרצה (יצירת משתנה, משתנה נכתב...)** | ❌ | ❌ | ✔️ |
| **שינוי מילים שמורות** | ❌ | ❌ | ✔️ |
| **Cross Platform** | ✔️ | ❌ | ✔️ |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

\* לא ניתן ליצור סוגים דרך קוד Little, אך אפשר להכניס סוגים מבחוץ, מצד המפתח  
\*\* אין variable capturing ו-pattern matching בתנאי switch

### טכנולוגיה

הפרויקט תוכנת בשפה יחסית לא מוכרת, ששמה Haxe. Haxe היא שפת תכנות שייצר אחד המתכנתים של ActionScript, שמטרתה לאפשר פיתוח אפליקציות למגוון מטרות ופלטפורמות, באמצעות טרנספילציה (תרגום קוד של שפה אחת לקוד של שפה אחרת). בהתחלה, מטרת השפה הייתה לאפשר כתיבה של גם לקוח וגם שרת באותה שפה, בעזרת השתמשות בספריות סטנדרטיות של השפות שאליהם הקוד מתקמפל. בשביל זה, נתמכו בתחילת הפיתוח רק 3 מטרות שאליהן היה אפשר לקמפל:

* JavaScript, בשביל אתרים וקוד ברשת
* ActionScript 3, בשביל משחקים
* Neko, מכונה ווירטואלית "תוצרת בית", בשביל תוכנות Native.

איתן היה אפשר, לדוגמה, לכתוב לקוח בעזרת הספרייה הסטנדרטית של JavaScript, ושרת בעזרת הספרייה הסטנדרטית של Neko (או שימוש בספריות דינמיות בעזרת Externs).

עם הזמן, יותר ויותר מטרות נוספו, וסגנון השפה השתנה: במקום להמשיך להדמות לשפות כמו Java וActionScript3, ובכך להקל את המעבר מהשפות האלה ל-Haxe, נוספו תכונות נוספות ופעולות מיוחדות, על מנת להבהיר ולהקל על עיבוד מידע. בפרויקט הזה יצא לי להשתמש בשתי תכונות עיבוד המידע העיקריות:

* **Algebraic Data Types:** בHaxe, הenum הקלאסי מורחב על מנת לתמוך בסוג קצת יותר מתוחכם של איבר, שמאפשר העברת פרמטרים שאותו איבר מבקש. זה מאוד שימושי כשרוצים לבנות עץ – בפרויקט השתמשתי בתכונה זו כדי לבנות את עצי הsyntax. דוגמה לעץ:

enum Tree<T>{

BinaryTree(?left:Tree<T>, value:T, ?right:Tree<T>);

RegularTree(value:T, children:Array<Tree<T>>);

Leaf(value:T);

}

var tree:Tree<Int> = BinaryTree(  
 Leaf(2),  
 1,

BinaryTree (

//Question mark allows us to skip parameter

3,

Leaf(4)

)  
);

* **Pattern Matching:** תנאי switch הורחבו על מנת לאפשר חילוץ והתאמת מידע הנמצא בתוך Algebraic Data Types, כמו מספרים, מחרוזות, מערכים, או ADTs אחרים. ניתן לעשות זאת על ידי הצבת ערכים, העמדת תנאים מיוחדים, או הוצאתם מהADT על ידי הצבת משתנים במקומם:

switch tree {

case BinaryTree(\_, (\_ > 100000) => true, (\_ == null) => false):   
 throw “Big tree cannot contain a right child”;

case RegularTree(\_, \_) if (disableRegularTree):

throw “Regular Trees are disabled”

case RegularTree(value, arr): //Do Things  
 case BinaryTree(null, value, null) | Leaf(value): {

trace(value);

}

case BinaryTree(r, v, l): //Do Things  
 case \_: //Do nothing, can also be `default:`

}

### הגבלות

כפי שהוזכר מקודם, אחת מהתכונות העיקריות של Haxe היא היכולת של השפה לעשות טרנספילציה להרבה מטרות אחרות – גם שפות תכנות, וגם מכונות ווירטואליות. על פניו נראה שזה יתרון ולא הגבלה, אבל זה לא בא בחינם, ויש קאטצ' – הספרייה הסטנדרטית קטנה ומצומצמת, וחייבת להתאים בפונקציות המוצעות למה שכבר קיים בכל המטרות האחרות. בפרויקט שלי, זה גרר בעיות שקשורות בצורך לסוגי מידע מיוחדים. אמנה מספר מקרים:

* **מערך ביטים** – אין דרך ליצור מערכים של אלמנטים שגודלם קטן מביית, מכיוון שלא כל המטרות תומכות ביצירה של מערך כזה (לדוגמה: PHP, Lua, ActionScript). נאלצתי להשתמש במערך של בייתים לזיכרון (ByteArray), ולהשתמש בכל בייט בתור תא היכול להכיל אלמנט אחד. בתיאוריה הייתי יכול להשתמש במערך רגיל ולאחסן את המידע בתוך הביטים של Int או Int64, אבל אז כבר אין סיבה להיות שמרן עם זיכרון, שכן מערך רגיל מומר לרשימה מקושרת או שילוב של רשימה ומערך בהרבה מהמטרות (Java: Array -> ArrayList, C++: Array -> vector, Python: Array -> list)
* **כמויות אלמנטים במערך** – לא כל המטרות שHaxe מציעה תומכות במערכים שאורכם עולה על 2.147 מיליארד אלמנטים, ומכיוון שלא יכולתי להשתמש במערך שגודל האלמנט שלו עולה על בייט אחד בשביל זיכרון, הוגבלתי לזיכרון בגודל מקסימלי של 2.147 גיגהבייט.
* **HashTables** – Haxe כן מציעה HashTable, אבל לא כל מטרה מאפשרת גישה לבתים ומקום האחסון של ה- HashTable, ולכן הייתי צריך לתכנת בעצמי HashTable, כדי שאוכל לגשת לבתים שהוא מורכב מהם, ולאחסן אותם באותו מערך זיכרון שדיברתי עליו מקודם.

מעבר להגבלות שקשורות למבנה השפה, נתקלתי גם בבעיות שנגרמו מחוסר הפופולריות של השפה:

* **Hashing** – כשהתחלתי תכנת את הHashTable, גיליתי שאין הרבה ספריות שבכלל מציעות את האלגוריתמים המתאימים (SipHash לדוגמה). לפרויקט הזה זה פחות שינה, כי לא בהכרח חיפשתי אלגוריתם Hashing שמטרתו להצפין, אך עדיין יצא שהשתמשתי באלגוריתם שפחות חסין להתנגשות ממה שהייתי משתמש בו בשפות אחרות (כן יש לציין שהHashTable של Haxe משתמש בSipHash, אבל זה לא בכוונה, אלא בגלל שהמטרות שאליהן Haxe עושה טרנספילציה משתמשות בSipHash בשביל הHashTable שלהן)
* **ספריות עזר** – אין הרבה ספריות שמציעות את התכונות וסוגי המידע שאני צריך, או לפחות הן לא עושות זאת בצורה יעילה מספיק. לכן, השתמשתי בפרויקט הזה בספריה אחרת שכתבתי שאינה קשורה לפרויקט ששמה Vision (ספריית CV, מכילה הרבה סוגי מידע). על אף שחלק גדול מהספרייה הזאת היה קיים לפני שהשתמשתי בו לפרויקט, שמתי לב שאני מדי פעם הוספתי פונקציות או תכונות שהייתי צריך בפרויקט הזה. בשפה יותר פופולרית, כנראה בכלל לא הייתי נתקל בבעיה דומה, ועוד יותר לא הייתי צריך להשתמש בספרייה משלי.

### תחומים ותמיכה

Project

├──── Data Processing

│ ├──── Module Storage

│ └──── Compiling

│ ├──── Keywords

│ │ └──── Keyword Manager

│ ├──── Text Tokenization (Lexer)

│ └──── Token Manipulation (Parser)

└──── Virtual Machine

├──── Interpreter

│ └──── Token Actuation

├──── Extern Registration

└──── Memory

├──── Data Storage

│ └──── Hash Tables

└──── Externs

בגדול, הפרויקט בנוי על ועוסק בעיבוד מידע באופן מאסיבי. באופן יותר ממוקד, הוא נכנס לענף של טוקניזציית מידע המובא בצורת טקסט, והפעלת מניפולציות על אותם טוקנים, ובנוסף גם שימוש באותם טוקנים כאמצעי שיכול להפעיל מעין מכונה ווירטואלית. אותה מכונה ווירטואלית מחברת בין התוכן שאותם טוקנים מייצגים, לפקודות שעל המחשב לבצע. חלק מהפקודות מסופקות מראש, ומה שלא מסופק מראש, מפתח שמשתמש בפרויקט יכול להוסיף בעזרת אגף מסוים בפרויקט. בשביל נוחות, אציג את הפסקה אחרונה כעץ, שנוצר בעזרת פונקציה שנלקחה מהפרויקט (מPrettyPrinter, אחרי אדפטציה):

## תיאור המערכת

### אז, מה בדיוק הפרויקט עושה?

בגדול **מאוד**, הפרוייקט הוא ה"קומפיילר" והמכונה ווירטואלית הרשמית של השפה שיצרתי - Little. הפרויקט גם מכיל שני אימפלמנטציות של לקוחות. אפרט, והרבה:

* **Little** – שפת התכנות עליה מבוסס הפרויקט. מדובר בשפה שיצרתי על מנת להנגיש תכנות לילדים קטנים. השפה (יחד עם הפרויקט הזה) נוצרה לפני קצת יותר משנתיים, והחליפה מספר שמות לפני שנחתה על השם Little:
  + [ב23 באפריל, 2022](https://github.com/ShaharMS/Little/commit/0c43e3e346ed3cd2746f8fb4864f475cba7cf4ef#diff-b335630551682c19a781afebcf4d07bf978fb1f8ac04c6bf87428ed5106870f5), הפרויקט נוצר, ונקרא Multilang-Coder. לשפה עוד לא היה שם.
  + [ב25 באפריל, 2022](https://github.com/ShaharMS/Little/commit/7d11131a08a0ed1d2d5910d0ac1db51a59051850), הפרויקט והשפה שינו את שמם ל”Minilang”. באותו commit מקושר, גם כתבתי חלק מאיך שרציתי שהשפה תעבוד. התחלתי עם הקוד, אבל הייתי רחוק מהמטרה שהוצבה שם. באותו הזמן גם, היה הנסיון הראשון למבנה וטכנולוגית הפרויקט, עליהם אפרט בהמשך. שם השפה עוד לא הוחלט באופן סופי ושם החבילה בה הוכל הקוד נקרא פשוט "language".
  + [ב26 באפריל, 2022](https://github.com/ShaharMS/Little/commit/dbd33065bb5a4c23fd06f30c8c7a439e245575c9) לא היה שינוי לשם, אך הסרתי את החלק שכתבתי בו איך שהשפה תעבוד, והוספתי את הספציפיקציה הראשונה שהכילה נראות ומבנה הקוד בשפה. גם על זה אפרט בהמשך.
  + [ב30 באפריל, 2022](https://github.com/ShaharMS/Little/commit/2e8ddaa16aaa266de313f088a3070301b6888be2) הפרויקט והשפה שינו את שמם לשם העדכני – "Little". גם כאן הספציפיקציה השתנתה קצת.

בזמן שהחלטת השם קרתה יחסית מהר ובשלב יחסית מוקדם של הפרויקט, החלטה על ספציפיקציה לקחה הרבה יותר זמן – כמו בהחלטה על השם, היו הרבה שינויים בהתחלה, אך כאן המקרה הוא אחר – תוך כדי פיתוח, הבנתי שיש תכונות שאני רוצה להשאיר, כאלה שאני רוצה להוריד, וכאלה שאני רוצה להוסיף. היו יחסית הרבה שינויים בזמן כתיבת הפרויקט. אנסה, כמו קודם, לעשות בהם סדר – אבל שימו לב! זה ייקח הרבה, *הרבה* יותר זמן:

define x = 5

define y = new ImprovedNumber(5)

y.increment(4)

print(y)

action increment(x:Number) = {

return x + 1

}

//Instances

//File name – Improved Number

* + [ב26 באפריל, 2022](https://github.com/ShaharMS/Little/commit/dbd33065bb5a4c23fd06f30c8c7a439e245575c9), נוספה הספציפיקציה הראשונה לשפה. יש לה הרבה אלמנטים שדומים לשפה שמוצגת בפרויקט: שימוש בdefine- וב-action על מנת להכריז על משתנה ופונקציה, לדוגמה. עדיין היה דמיון משמעותי לHaxe, שכן אחת המטרות הראשונות (שנפלו בשלב מאוחר יותר) היו בניית טרנספיילר לHaxe. לא הוספתי אף מדריך בשלב הזה, אך כן הוספתי הסבר על חלק ממילות המפתח שהיו אז בשפה. להלן הספציפיקציה:

define baseNumber

action new(number:Number) = {

baseNumber = number

}

//write comments with a double /!

// + types for actions are automatically inferred

action increment(x:Number) = {

return baseNumber += x

}

hide action renew(number:Number) = {

return new ImprovedNumber(number)

}

define x = 5

define z:Number = 10

define y = new ImprovedNumber(5)

y.increment(4)

print(y)

//also supports classes:

className: ImprovedNumber

define baseNumber:Number

action new(number:Number) = {

baseNumber = number

}

//write comments with a double /!

// + types for actions are automatically inferred

action increment(x:Number) = {

return baseNumber += x

}

hide action dispose() = {

//nothing

}

* + [ב30 באפריל, 2022](https://github.com/ShaharMS/Little/commit/2e8ddaa16aaa266de313f088a3070301b6888be2) שוב השתנתה הספציפיקציה, אך לא נוספו תכונות חדשות: רק הוספתי רעיון לsyntax שיתאים ליצירת סוגים בתוך Little:
  + [ב3 במאי, 2022](https://github.com/ShaharMS/Little/commit/f39ec722fb56007842c4cf9aa91f753e8ffa0616) נוסף הרעיון הראשון המשמעותי ששרד (כמעט לגמרי) את מבחן הזמן, ונכנס לתוצר הסופי – לולאת ה-for. הספציפיקציה מציעה syntax אלטרנטיבי, שלא משלב אופרטורים/סימנים (לדוגמה: ..., ;), ומחליף אותם במילים from, ו-to. syntax זה עוצב גם כדי להשאיר מקום לולאת for שעוברת על מערכים, על אף שלא היה תכנון מידי לזה באותו זמן.

שאר הקוד בספציפיקציה זהה לזה שלמעלה, אעתיק את השוני לכאן:

for name from 0 to 9 {

print(i)

}

className: ImprovedNumber

define baseNumber:Number

action new(number:Number)

baseNumber = number

//write comments with a double /!

// + types for actions are automatically inferred

action increment(x:Number)

return baseNumber += x

hide action dispose()

//nothing

* + שוב [ב3 במאי](https://github.com/ShaharMS/Little/commit/9391ce3f562129321196181aa886979fc314f914), הפעם חשבתי על ללכת על סטייל python כל הדרך בכתיבת פונקציות, רק בלי ה-:. כבר אומר מראש – זה לא שרד הרבה זמן :)
  + עכשיו מתחילים לקפוץ בזמן: [ב16 בינואר 2023](https://github.com/ShaharMS/Little/commit/029cbc27218b5d3fa167e393d3e2b519e5e7478e), קרו מספר דברים יחסית גדולים:
    - הוסרה הספציפיקציה ליצירת סוגים, מתוך רצון להציב מערכת עובדת לפני שמתעסקים עם יצירת סוגים.
    - האופרטור המשמש להכרזת סוג על ערך/משתנה הוחלף מאופרטור למילים: מ-: ל-of type. זאת במטרה להפרד קצת מ-Haxe, ולפשט את השפה.
    - נוסף האלמנט every ללולאות for, המאפשר קפיצות בין ערכים. (לדוגמה, from 0 to 5 every 2 יעבור על הערכים 0, 2, 4

הספציפיקציה המלאה:

define x = 5

define z of type Number = 10

define y = new ImprovedNumber(5)

y.increment(4)

print(y)

define fileWriter = File.write("idk.txt")

fileWriter.writeString("Yay Haxe")

fileWriter.close();

for name from 0 to 9 every 2 {

print(i)

}

* + קצת לפני ה[16 במרץ 2023](https://github.com/ShaharMS/Little/commit/029cbc27218b5d3fa167e393d3e2b519e5e7478e), מחקתי את הספציפיקציה שנכתבה עד אז,
  + [ב29 בינואר, 2023](https://github.com/ShaharMS/Little/commit/acaec403399cc437ffb97105c8d391dd90cbc50f), מחקתי את החלק בReadme שהעיד על הספציפיקציה, לעומת שלושה דברים שהצבתי לעצמי לצור בעתיד:
    - קובץ טקסט נפרד המכיל את הפירמוט, הsyntax וה"זרימה" של קוד השפה
    - סוויטת בדיקות, שגם היא תסתמך על תכונות השפה ותעיד על המבנה שלה
    - איזכור של תכונות ייחודיות בReadme של הפרויקט
  + [ב16 במרץ 2023](https://github.com/ShaharMS/Little/commit/029cbc27218b5d3fa167e393d3e2b519e5e7478e) הוגשמה המטרה השלישית, ויצרתי חלק בקובץ הReadme שנועד לאיזכור תכונות. התכונה הראשונה שנוספה לשם היא אחת מתכונות הבסיס של Little, וזה העיקרון Everything can be a code block. אסביר עליו בהמשך. הדוגמה שסופקה:  
     x = {define y = 0; y += 5; (6^2 \* y)} //180

define consistent = 5

define consistent.newPropertyDeclaration = 6

action declaredJustLikeVariables(define parametersAreDefinedTheSame = 6) = {

print("Function Bodies are also assigned using `=`")

}

* + [ב19 באפריל, 2023](https://github.com/ShaharMS/Little/commit/d8ee3e0341f8d4dcb94b08e7df063456fd20017d) נוספה לReadme התכונה השניה, שעקרונה: Consistency Is Key. גם עליה אפשר בהמשך, על אף שהיא יחסית ברורה מדוגמת הקוד שכתבתי לReadme:
  + [ב23 באפריל, 2023](https://github.com/ShaharMS/Little/commit/903167bb7495e9677386f627ceff9bbe0fa5e037) נוספו \*המון\* דוגמות ומדריכים לקובץ הReadme – מתכונות של השפה, לתכונות של קורא הקוד, ותיעוד של כמה לולאות, חדשות וישנות:
    - לולאת while – מבצעת את גוף הלולאה עד שהתנאי מחזיר true
    - תנאי if מבצע את גוף התנאי עם התנאי הראשוני מחזיר true
    - לולאת for שינתה את המילה every למילה jump
    - תנאי after – מבצע את גוף התנאי, מיד לאחר שהמשתנה שמאוזכר בתנאי הראשוני משתנה, וגורם לתנאי הראשוני להחזיר true
    - תנאי whenever- שילוב של after וwhile- – כל פעם שאחרי עדכון המשתנה המוזכר, התנאי הראשוני מחזיר אמת, הגוף שבתוך התנאי מורץ.

לא אוכל לשים כאן הכל בגלל גודל הטקסט (בין 2 ל3 עמודים), אז [הנה קישור לקובץ בגיטהאב באותו commit](https://github.com/ShaharMS/Little/tree/903167bb7495e9677386f627ceff9bbe0fa5e037?tab=readme-ov-file#language-features)

* + [ב6 בנובמבר, 2023](https://github.com/ShaharMS/Little/commit/0cfb7810685a1ce82ca17c246a0e03cb097cb0ce), נוספה עוד דוגמה לתכונה בשפה – דוקומנטציה. היא קיבלה אימפלמנטציה קונקרטית באותו הזמן, בעזרת סדרת הסימנים """.  
    גם נוספה הרחבה לאחת הדוגמאות, שכללה יצירת משתנה עם שם שמוחלט בזמן ריצה. מעבר לזהף היו תיקונים קטנים, בעיקר של שגיאות כתיב/syntax:

define {("hey" + 1)} = 3

print(hey1) //3  
  
"""

Retrieves the value of `x`

"""

define x = 3

""" Increments the value of `x""" `

action incrementX() = { x = x + 1 }

print(x) //3

print(x.documentation) //Retrieves the value of `x`

print(incrementX.documentation) //Increments the value of `x`

* + [ב8 בנובמבר, 2023](https://github.com/ShaharMS/Little/commit/12ba9f52503636d14d58c52fb97e7ee128b8bc79), נוצרה סוויטת הבדיקות לראשונה – היו בה מספר בדיקות, והיא הדפיסה לכל בדיקה האם היא עברה, ואם לא מה קרה כשהיא לא עברה. ביצירה של הבודק הוספתי 7 בדיקות. מכיוון שהן נמצאות ממש בתוך הקוד של הבודק (אפרט בהמשך על למה ואיך הוא עובד) אני לא יכול להביא אותן, אבל [הנה קישור לקובץ הבודק עצמו](https://github.com/ShaharMS/Little/blob/12ba9f52503636d14d58c52fb97e7ee128b8bc79/src/UnitTests.eval.hx).
  + [ב16 בנובמבר, 2023](https://github.com/ShaharMS/Little/commit/05980d8ad4cfeff97b379f0f354ecefd33549c23) ו[ב22 בנובמבר, 2023](https://github.com/ShaharMS/Little/commit/7372a36d82f922ca7983cd2d3b0be71302ad1282), נוצר קובץ הספציפיקציה "הרשמי", בשני חלקים – פשוט קובץ טקסט המכיל קוד מתוכנת בשפה Little. הוא הכיל גם ספציפיקציה להכרזה של סוגים, לולאות ותנאים על מנת לנסות לקבוע סטנדרט לאיך השפה אמורה בערך להראות.

"""

My Custom Class

"""

class Foo = {

define self.public = 5

define Foo.static = 5.6

"""

Constructor

"""

action self.create() = {

}

action self.somePublicAction(define someParameter, define param as Number = 2) = {

print(self + " Called for " + someParameter + " and " param)

}

action Foo.someStaticFunction() = {

print(Foo.documentation)

}

}

define someVar as Decimal = 3

action someRandomAction() = {

}

"""

doTwiceIf (3 > 5) {

print(i)

}

"""

condition doTwiceIf(define condition as Characters, define body) = {

if (run(condition)) {

runWith(body, "define i = 1")

runWith(body, "define i = 2")

}

}

condition repeat(define iterationCounter, define body as Characters) = {

define times = run(iterationCounter).toNumber()

while(times > 0) {

run(body)

times = times - 1

}

}

for (define x from 0 to -100 jump 10) print(x)

while (true != false) print({define z = 3, z})

sign (left ^% right) = {

return left + right ^ 2

}  
print(^%.priority.type) //Number

* + מיצירת קובץ הספציפיקציה, לא קרו שינויים דרמטיים במבנה הקוד. השינוי הגדול האחרון קרה בתאריך [17 במרץ, 2024](https://github.com/ShaharMS/Little/commit/4e510898acfa2edcd02b5d3ccbb42ae74bbfc34f), שבו נוספה הספציפיקציה   
    להכרזה על סימנים:  
    עם הזמן, נוספו גם בדיקות לסוויטת הבדיקות, אך אין צורך להזכיר אותן (לפחות כרגע), שכן הן לא מכילות קוד שאמור "להוות דוגמה", אלא רק כזה שנועד לבדוק אם דברים עובדים כמצופה. (רוב הבדיקות בשורה אחת על עף שמצופה לשים line breaks, לדוגמה).

עכשיו שסוף סוף, סיימנו עם ההיסטוריה של השפה, בו נתעסק בהווה: בפועל, אלה התכונות ודרכי התכנות שהשפה מאפשרת:

* + **דרך כתיבה:**
    - **בלוק של קוד יכול להיות כמעט הכל:** שמות משתנים, לולאות, פונקציות ואפילו סימנים מאפשרים הכרזה וגישה בזמן ריצה בעזרת בלוקים של קוד – בלוק של קוד הוא כמות מסויימת של שורות, מאוכסנות בתוך סוגריים מסולסלות. בלוק יכול להחזיר ערך בעזרת מילת המפתח return. ניתן גם להשמיט אותה, ואז הערך המוחזר יהיה הtoken האחרון בבלוק הקוד.
    - **כתיבת קוד אחידה:** כמה שאפשר, כל הדרכים לכתוב syntax שמטרותיהן דומות (הכרזות, גישות, קריאות) כתובות באותה דרך, אותה צורה, או לפחות אותו זרם: תמיד משתמשים ב= על מנת להצמיד ערך (אפילו לפונקציות) – יש להשתמש במילת הכרזה כאשר יוצרים property חדש על אובייקט, וכן הלאה

define בחזקת = ^  
define כפול = \*  
print(5 כפול 6 בחזקת 2)

* + - **כל דבר יכול לשמש כערך:** נובע משני העקרונות לעיל – אם למשתנה יש ערך, ופונקציה יכולה לשמש כערך, למה שלא כל סוגי המידע יעשו זאת? לכן, אפשר גם להשתמש באופרטורים וסוגים כערכים. עיקרון זה שימושי ליצירת aliases, וגם יכול ליצור דברים משעשעים כמו:  
      (יוצא 180)
  + **פיצ'רים**
    - **הכרזות:** יש תמיכה בהכרזת משתנים ופונקציות, מכל סוג שהוא, באופן סטטי ודינמי:
    - **שורות:** אין צורך לשים ; בסופי שורות, אך אפשר להשתמש ב-, על מנת להכניס כמה שורות קוד בשורה אחת, וזה מפצל את השורה לחלקים.

define x = 3  
define e = read(“x”)  
print({e +=7, read(Characters.fromCharCode(e + 91))})

* + - **גישות וקריאות:** אפשר לגשת לערכים של מזהים בעזרת השתמשות בשמם, השתמשות בפעולה read, או בלוק של קוד שמחזיר פעולת read. קריאה לפונקציות מתבצעת באותה צורה, עם הוספת () אחריהם. הסוגריים יכולים להכיל פרמטרים, מפוצלים בעזרת שורה חדשה או ,  
      (יוצא "e")
    - **לולאות ותנאים:** רוב הלולאות שזמינות בשפות אחרות זמינות גם בLittle: if, while, for. בLittle יש גם שתי לולאות/תנאים חדשים: whenever ו-after – after מתבצע פעם אחת בדיוק, מיד לאחר שהמשתנה שמוזכר בתוך התנאי שלו משתנה וגורם לזה שאותו תנאי יחזיר true. Whenever עושה אותו הדבר, אך לא מתבטל אחרי פעם אחת:כש-x יוגדל מעבר ל-3, יודפס ערכו. בפעם הראשונה ש-x יהיה שווה ל-4 תודפס המילה hey.

define x = 3  
after (x == 4) print(“hey”)  
whenever (read(“x”) > 3) {  
 print(x)  
}

* + - **גופי קוד:** כמו שבטח ראיתם מקודם, אפשר הרהב פעמים להשמיט את הסוגריים המסולסלות במקום שמצופה בלוק של קוד. המקרה היחיד שאי אפשר להשמיט אותן הוא כשמתמשים בבלוק In-line על מנת לייצר ערך.

עכשיו, סיימנו עם השפה. הגיע הזמן לעבור ל:

* **Compiler** – הלב של הפרויקט, ומבצע העבודה השחורה – הוא מקבל קוד הכתוב בLittle, והופך אותו לAST שאפשר להריץ, או לשמור כBytecode. גם  
    
  ----נמחק, להשלמה בסוף----  
    
  על אף שאין לו עבר מרשים כמו לשפה עצמה, הוא יותר ממחפה על זה בפיצ'רים שהוא מציע:  
  **‎**

|  |  |  |
| --- | --- | --- |
| define x = 3, x = x + 6 action getX() = {     return x } print(getX()) | הגדר ס = 3, ס = ס + 6 פעולה קבל\_ס() = {     החזר ס } הדפס(קבל\_ס()) | السياج ع = 3, ع = ع + 6 فعل يحصل\_ع() = {     استرداد ع } مطبعة(يحصل\_ع()) |

|  |  |
| --- | --- |
| define x = 3, x = x + 6 action getX() = {     return x } print(getX()) | var x = 3, x = x + 6 fun getX() = {     ret x } log(getX()) |

* + **שינוי מילות מפתח:** אפשר לשנות את המילות השמורות בשפה לסט אחר של מילים, או לשנות מילים אינדיבידואליות בעזרת ממשק מיוחד. [בתוך קובץ הreadme](https://github.com/ShaharMS/Little/tree/branch/functional-programming?tab=readme-ov-file#keyword--standard-library-modification) יש דוגמה לשינויי מילים. אביא את הדוגמאות משם:
  + **Parser Injections:** הParser בנוי בצורה שמאפשרת "השחלה" של פונקציות parsing תוך כדי תהליך הparser הרגיל, ואף מאפשר הסרה של שלבים, על מנת לתת למפתחים שמתמשים בפרוייקט לעצב את נראות השפה, להוסיף, ולהוריד לה תכונות.
  + **Custom Tokens:** בשלב הParsing, המפתח יכול להוסיף טוקנים משלו, כדי לפשט הוספת שלבים לParser. הופך את הפיצ'ר לעיל לעוד יותר עוצמתי.
  + **הוספת externs מצד המפתח:** יש אגף שלם שמיועד אך ורק להוספת משתנים, פונקציות, שדות, סוגים ואופרטורים בדרך מהירה ולא "מנופחת" מדי. זה מאפשר למפתח להוסיף על הספרייה הסטנדרטית שLittle מספקת, ואפילו ליצור חלקים שלמים למטרה שלשמה המפתח משתמש בפרוייקט (לדוגמה, אם Excel היו משתמשים בפרויקט במקום בVBA, הם היו משתמשים באגף הזה כדי להוסיף סוגים ופונקציות שעושות interfacing עם Excel.)
* **Interpreter** – לאחר שהקומפיילר מעבד את המידע ומייצר את הAST, צריך שיהיה   
  ---נמחק, להשלמה בסוף---

### יכולות

לפרויקט יש שני סוגי לקוחות עיקריים:

* **מפתחי תוכנה:** אם מפתח רוצה לשלב יכולות פרוגרמתיות בתוכנה שלו, כמו Excel לדוגמה, הפרויקט הזה נותן לו את האפשרות לעשות זאת, ואפילו נותן יותר אפשרויות ופיצ'רים מהכלים שהתחום הזה (שפות תכנות אינטגרטיביות) מציע (VBA וHscript לדוגמה, [טבלה](#_פתרונות_קיימים))
* **לומדי תכנות:** מעבר לסתם כלי או ספריה, הפרויקט מספק שני סוגים של סביבות פיתוח – אחת כאתר, ואחת בשורת הפקודה. משתמשים שרוצים ללמוד את השפה, או ללמוד לתכנת בכלליות, יכולים לגשת לכתובת של האתר, או, למשתמשים יותר מתקדמים שרק רוצים להתנסות בשפה, להוריד את הפרוייקט ולקמפל אותו לבד (זה [האתר המדובר](https://spacebubble.io/little/demo/), ניתן להריץ את הלקוח בשרת הפקודה באמצעות הרצת פקודת הבנייה haxe compile.hxml)

אפילו שהפרויקט ברובו ממוקד בבנית והרצת קוד, יש לו תכונות אחרות, שלאו דווקא קשורות לתכנות לדוגמה:

* **מתמטיקה:** אפשר להדפיס תרגילים מתמטיים בעלי פתרון כזה או אחר, ואז לקחת את הפתרון שלהם באמצעות:

Conversion.toHaxeValue(  
 Little.runtime.stdout.stdoutTokens.pop()  
)

(ממיר לערך רגיל את הטוקן האחרון שהודפס, שהוא תוצאת המשוואה. יכול להיות מספר, בוליאני, מחרוזת או אפילו אובייקט במקרה הצורך)

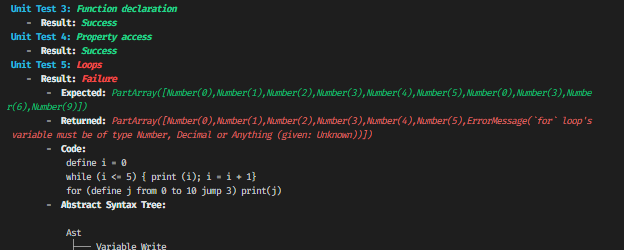
* **Pretty Printer:** באמצעות המחלקה PrettyPrinter, ניתן להדפיס עצים יפים של מערכי Enums.

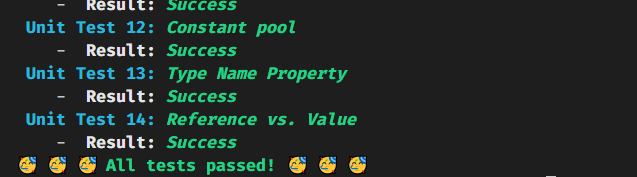
### סוויטת בדיקות

על מנת לוודא שהפרוייקט עובד, כשאני בונה אותו לבדיקה, אני יכול להגדיר פרמטר haxedef שנקרא unit באמצעות ווידוא שבקובץ הבונה (compile.hxml) נמצאות השורות:  
--define unit – מגדיר את הhaxedef: unit ונותן לו בברירת מחדל את הערך true כי לא סופק ערך אחר  
--interp – מגדיר את מטרת הבנייה, עושה שהקוד יורץ באמצעות הinterpreter של haxe ולא יתקמפל.

כש-unit מוגדר, מאופשרת השורה

UnitTests.run(true);

שמריצה את כל הבדיקת, אחת אחר השנייה ללא הפסקה (הפרמטר שמועבר מחליט האם להפסיק בבדיקה נכשלת וגם האם לקצר את ההפסקה בין כל בדיקה מ0.2 שניות ל0.05 שניות). לנוחות וליופי, הבדיקות מודפסות עם צבעים ופרמוט, ולפי סדר. תמונה להמחשה:  
  
  
...  
A black screen with white text

Description automatically generated  
  
כהכנה לעתיד (כשיהיו יותר בדיקות), מודפסת הודעה נוספת בסוף כאשר כל הבדיקות מצליחות:  
  
  
  
כיום, מורצות 14 בדיקות, כל אחת למטרה שונה, אך יכולה להסתמך על דברים אחרים   
שצריכים לעבוד. בדיקות מסודרות לפי שם פונקציה (test<testNumber>), וצריכות להחזיר מספר דברים בתור אובייקט:

* + - * שם בדיקה
      * האם הבדיקה הצליחה
      * תוצאת הבדיקה
      * למה הבדיקה ציפתה
      * הקוד של הבדיקה

והבדיקות הן:

1. Basic Math – מתמטיקה פשוטה, ופונקציית הדפסה
2. Variable Declaration – הכרזת משתנים, פונקציית הדפסה
3. Function Declaration – הכרזת פונקציות, קריאה לפונקציות
4. Property Access – יצירת אובייקטים דינמיים, יצירה וגישה לשדות
5. Loops – לולאות while וfor
6. Events and Conditionals – תנאי after וwhenever
7. Code Blocks – שימוש בקוד סגור בסוגריים מסולסלים כערך למשתנה
8. Self Assignment – שינוי ערך באמצעות הערך שלו
9. If-Else – תנאי if עם else
10. Nested Code Blocks- בלוקים של קוד אחד בתוך השני, ובדיקה לראות האם משתנה שהוכרז בבלוק פנימי יערוך את המשתנה החיצוני (לא טוב), או שיושמד בסוף הבלוק והערך ה"ישן" "יחזור" (טוב)
11. Inline Blocks – כמו Code Blocks, הפעם גם פונקציית ההדפסה מקבלת את המשתנה שעליה להדפיס בתור בלוק של קוד
12. Constant Pool – בזיקה האם הכתובות של ערכים שאמורים להיות בconstant pool אכן נמצאים שם ולא לוקחים זכרון ממקום אחר
13. Type Name Property- בדיקה האם השדה .type עובד על כל ערך, מאובייקט ועד לאופרטור
14. Reference vs. Value – בודק האם כשמשווים ערך שהוא pass by value הוא יוצר ערך חדש בזכרון, וההפך לערכים שהם pass by reference.

### לוח זמנים

כשהתחלתי עם הפרויקט, לא בדיוק היה לי לוח זמנים, אלא רצון לסיים איתו לאחר כמות מסויימת של זמן. לא חשבתי בכלל שאצטרך את הפרויקט הזה לעבודה בסייבר אז, ולכן מה שקרה בפועל זה שבאמת באופן כללי סיימתי מתי שרציתי, ומאז עשיתי תיקוני באגים קטנים או תוספות ויזואליות לUI שהפרוייקט הציע כשהיה צריך.אארגן לטבלה:

|  |  |  |  |
| --- | --- | --- | --- |
| **אבן דרך** | **תאריך מצופה** | **תאריך בפועל** | |
| תחילת הפרוייקט | 23 באפריל, 2022 | | |
| שפת תכנות עובדת | חודש אפריל, 2023 |  |  |
|  |  |
|  |  |
| שפת תכנות עובדת | 25 באפריל, 2023 |
| סיום העבודה על שפת התכנות, לפני סייבר | 13 בספטמבר, 2023 | | |
| פתיחת עבודה על הפרויקט, לסייבר | 4 בנובמבר, 2023 | | |
|  |  | | |
|  |  | | |
|  |  | | |
|  |  | | |
|  |  | | |
|  |  | | |

### סיכונים

כבר ב2022, רק כשהתחלתי, היה לי ניחוש, להפתעתי אפילו אחד מוצלח, למה הולך להפריע, ומה יכול לתקוע אותי בפיתוח, אך כשחשבתי על אותם סיכונים אז, החלטתי להמשיך הלאה בפיתוח. אדבר על סיכונים שלאו דווקא קשורים לקוד עצמו, אלא לכלים שבהם השתמשתי והיו זמינים לי

* **שימוש בשפה Haxe:** אפילו שהיא השפה האהובה עלי, בכל פרויקט שאני מתחיל אני תמיד חייב לתהות האם אני יכול להשתמש בה. לצערי, התשובה לא תמיד חיובית, ויש דוגמאות לזה:
  + [**Dynamically**](https://github.com/ShaharMS/Dynamically-CS)**:** פרויקט אחר, שהוא אפליקציה לאיור ופתירת בעיות מתמטיות המוצגות בעזרת גרפיקה (תרגילים בגיאומטריה, פיזיקה, מודלים חישוביים). הייתי חייב יכולות AI לפרויקט, וכן ספריות אלגברה ליניארית ופתירת משוואות, דבר שאין בHaxe, ולא הייתי מעוניין לפתח.  
    Fact Fun: השם של הפרוייקט הזה זהה לספריה שתכננתי לפתח בעל מטרה קשורה – פיתוח גרפיקות לצורות שאפשר לשחק איתן. הפסקתי לפתח אותה בגלל חוסר בביקוש, מהצד שלי ומשתמשים אחרים. [קישור](https://github.com/ShaharMS/Dynamically)
  + [**SpawnerTag**](https://github.com/ShaharMS/SpawnerTag)**:** פלאגין לשרת במשחק Minecraft, שמטרתו להפוך אביזר "בלוק" הנקרא Monster Spawner לאביזר שאפשר להשיג בצורה איןזופית באופן תיאורתי. כשפיתחתי את זה, עוד לא היה כלי לפיתוח פלאגינים לשרתי Minecraft, אז הייתי חייב להשתמש בKotlin. היום כן קיים כלי כזה, ושמו [PickHaxe](https://github.com/EliteMasterEric/PickHaxe) (משחק מילים על המילה pickaxe, אביזר נפוץ ושימושי בMinecraft)

הפעם כן יכולתי להשתמש, שכן רוב הדברים המיוחדים שהייתי צריך כבר היו קיימים (ספריית hash) ולא הייתי צריך הרבה כלים חיצוניים חוץ ממש שהשפה מציעה (כן השתמשתי במחלקות ופיצ'רים מהספרייה [Vision](https://github.com/ShaharMS/Vision), אך רובם היו כלי עזר, ובכל זאת אני פיתחתי אותם :) )

* **ניהול זכרון:** מההתחלה, חששתי שיהיו לי בעיות עם אחסון מידע – בסוף, הרצת קוד זה לא רק פקודות – צריך לשמור מידע על הדרך. בהתחלה הפתרון באמת היה לא טוב (כמו שהוזכר מקודם), ולכן הבנתי שהפתרון הטוב ביותר הוא כנראה האחד הסטנדרטי – למדתי על הstack- וה-heap, איך מתכנתים כל אחד ומה מיוחד לכל אחד, ועליתי על הפתרון העדכני, שמורכב מארבעה מחלקות: ConstantPool, Storage, Referrer, ExternalInterfacing.. אבל, גם בפתרון העדכני אפשר לראות משהו חמור – איפה באמת ה-stack וה-heap?  
  זה אכן היה סיכון שני, שאפצל לנקודה שנייה:
* **בקשה ופינוי של זכרון:** Haxe היא לא שפה שיכולה ישר לבקש ולפנות זכרון למערכת, בגלל אופי הפעולה שלה: היא מקמפלת את עצמה לשפות אחרות, לא לקוד מכונה. ידעתי שזה יכול להיות בעייתי, אך הנחתי שזאת לא בעיה משמעותית מספיק, והמשכתי האלה. זאת הייתה הנחה נכונה, שכן על אף שהיא הקשתה עלי בתכנות הזיכרון, והייתי חייב לקחת פתרון שלא היה 100% סטנדרטי, יכולתי להשתמש במגוון הפיצ'רים שנועדו לעיבוד מידע, שלא תמיד יש בשפות אחרות...
* **זמן:** ברור שלכל פרויקט יש בעיית זמן כזו או אחרת, אבל אחד מהדברים שהכי חששתי מהם, זה כמה זמן זה הולך לקחת – שפת תכנות לא באה בקלות, וראיתי איך שפו תתכנות אחרות קיימות מלא שנים, ועדיין יחסית "חלשות". בגלל החשש הזה, הצבתי לי גם מטרה וגם דדליין – כמו שהזכרתי בתכנון זמנים, הבטחתי לעצמי שאני מסיים את הפיתוח עד בדיוק שנה אחרי – התחלתי באפריל 2022, והבטחתי שאסיים עד סוף אפריל 2023, לא משנה אם כל הדברים שרציתי לעשות בפרויקט נמצאים בו. למזלי, באמת הספקתי את הרוב המוחלט עד אז, ורק היו תיקוני באגים כאלה ואחרים מאז ועד שהפרויקט קיבל משמעות חדשה, כפרויקט סיום בסייבר.

# תיאור תחום הידע

## יכולות

### סוג לקוח: מפתח אפליקציות

למפתחי אפליקציות יש מגוון אפשרויות אינטראקציה עם הפרויקט, הכוללים את כל האמצעים שהפרויקט מציע, מבנייה, הרצה, ואסיפת מידע דיאגנוסטי. אפרט

* **יכולת: הרצת קוד הכתוב בLittle בזמן ריצה**
  + מפתח יכול לקחת כל קוד מכל מקום שהוא רק רוצה, ולהריץ אותו באמצעות הפונקציה run שבמחלקה Little.
  + שדות ומחלקות נחוצים:
    - Little
      * debug:Bool
      * queue:Queue<String>
    - PrepareRun
      * prepared:Bool
      * addTypes()
      * addSigns()
      * addFunctions()
      * addConditions()
      * addProps()
    - Runtime
      * module:String
      * errorThrown:Bool
      * line:Int
    - Interpreter
      * run():InterpTokens
      * convert(…tokens:ParserTokens):InterpTokens
    - Parser
      * parse(tokens:Array<LexerTokens>):Array<ParserTokens>
    - Lexer
      * lex(code:String):Array<LexerTokens>
* **יכולת: הרצה לפני מודולות**
  + מפתח יכול לקחת קוד, ולהריץ אותו לפני קוד אחר, או לשים אותו בהמתנה על מנת להריץ אותו מיד לפני שמריצים את המודולה העיקרית. ניתן לעשות צורה זאת של preloading באמצעות הפונקציה loadModule שבמחלקה Little, ולהריץ את המודולות שבהמתנה באמצעות קריאה לפונקציה run שבאותה מחלקה.
  + שדות ומחלקות נחוצים:
    - Little
      * debug:Bool
      * queue:Queue<String>
    - PrepareRun
      * prepared:Bool
      * addTypes()
      * addSigns()
      * addFunctions()
      * addConditions()
      * addProps()
    - Runtime
      * module:String
      * errorThrown:Bool
      * line:Int
    - Interpreter
      * run():InterpTokens
      * convert(…tokens:ParserTokens):InterpTokens
    - Parser
      * parse(tokens:Array<LexerTokens>):Array<ParserTokens>
    - Lexer
      * lex(code:String):Array<LexerTokens>
* **יכולת: בניית קוד לAST**
  + מפתח יכול לקחת כל קוד מכל מקום שהוא רק רוצה, ולבנות אותו ללא הרצה באמצעות פונקציית compile שבמחלקה Little.
  + שדות ומחלקות נחוצים:
    - Interpreter
      * convert(...tokens:ParserTokens):Array<InterpTokens>
    - Parser
      * parse(tokens:Array<LexerTokens>):Array<ParserTokens
      * >
    - Lexer
      * lex(code:String):Array<LexerTokens>
* **יכולת: אסיפת מידע דיאגנוסטי על אירועי ריצה**
  + מפתח יכול להוסיף event listeners למגוון אירועים שקורים בזמן הריצה של הקוד המובא, מאירועים בסיסים כמו מעבר שורה עד ליותר מפורטים כמו התחלת לולאה. על מנת להוסיף מאזין, יש להשתמש בשדות המתחילים ב-on שבמחלקה Runtime. אפשר לגשת למחלקה באמצעות Little.runtime.
  + שדות ומחלקות נחוצים :
    - InterpTokens
    - Runtime
      * onLineChanged:Array<Int -> Void>
      * onLineSplit:Array<Void -> Void>
      * onTokenInterpreted:Array<InterpTokens -> Void>
      * onErrorThrown:Array<(String, Int, String, String) -> Void>
      * onWriteValue:Array<Array<String> -> Void>
      * onFunctionCall:Array<(String, Array<InterpTokens>)- > Void>
      * onConditionCall:Array<(String, Array<InterpTokens>, InterpTokens) -> Void>
      * onFieldDeclared:Array<(String, FieldDeclarationType) -> Void>
      * onTypeCast:Array<(InterpTokens, String) -> Void>
      * throwError(token:InterpTokens, ?layer:Layer):InterpTokens
      * warn(token:InterpTokens, ?layer:Layer)
    - Interpreter
      * setline(l:Int)
      * splitLine()
      * declareVariable(name:InterpTokens, type:InterpTokens, doc:InterpTokens)
      * declareFunction(name:InterpTokens, params:InterpTokens, doc:InterpTokens)
      * condition(name:InterpTokens, pattern:InterpTokens,body:InterpTokens):InterpTokens
      * write(assignees:Array<InterpTokens>, value:InterpTokens):InterpTokens
      * call(name:InterpTokens, params:InterpTokens):InterpTokens
      * read(name:InterpTokens):InterpTokens
      * typeCast(value:InterpTokens, type:InterpTokens):InterpTokens
      * run(body:Array<InterpTokens>, propagateReturns:Bool):InterpTokens
* **יכולת: יצירת Bytecode**
  + מפתח יכול לשמור קוד להרצה מאוחרת בפורמט מצומצם בעזרת הפונקציה compile שבמחלקה ByteCode. הפונקציה הזאת מקבלת AST, שאותו אפשר לקבל באמצעות Little.compile, שהוזכר קודם לכן. יצירת Bytecode ממחרוזת נראית כך: ByteCode.compile(Little.compile(“..code”)).
  + שדות ומחלקות נחוצים:
    - InterpTokens
    - Interpreter
      * convert(...tokens:ParserTokens):Array<InterpTokens>
    - Parser
      * parse(tokens:Array<LexerTokens>):Array<ParserTokens
      * >
    - Lexer
      * lex(code:String):Array<LexerTokens>
* **יכולת: הוספת אלמנטים חיצוניים**
  + ישנו אגף שלם הנועד רק כדי להקל על מפתחים להוסיף ערכים, פונקציות, מחלקות ואופרטורים חיצוניים. ניתן לגשת לאגף באמצעות Little.plugin.
  + שדות ומחלקות נחוצים:
    - Plugins
      * registerType(typeName:String, fields:TypeFields)
      * registerVariable(variableName:String, variableType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:Void -> InterpTokens)
      * registerFunction(functionName:String, ?documentation:String, expectedParameters:EitherType<String, Array<InterpTokens>>, callback:Array<InterpTokens> -> InterpTokens, returnType:String)
      * registerCondition(conditionName:String, ?documentation:String ,callback:(params:Array<InterpTokens>, body:Array<InterpTokens>) -> InterpTokens
      * registerInstanceVariable(propertyName:String, propertyType:String, onType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:(objectValue:InterpTokens, objectAddress:MemoryPointer) -> InterpTokens)
      * registerInstanceFunction(propertyName:String, onType:String, ?documentation:String, expectedParameters:EitherType<String, Array<InterpTokens>>, callback:(objectValue:InterpTokens, objectAddress:MemoryPointer, params:Array<InterpTokens>) -> InterpTokens, returnType:String)
      * registerOperator(symbol:String, info:OperatorInfo)
      * combosHas(combos:Array<{lhs:String, rhs:String}>, lhs:String, rhs:String)
    - InterpTokens
    - Conversion
      * toHaxeValue(token:InterpTokens):Dynamic
      * toLittleValue(value:Dynamic):InterpTokens
    - ExternalInterfacing
      * createPathFor(extType:ExtTree, ...path:String):ExtTree
      * createAllPathsFor(...path:String)
    - ExtTree
    - Operators
      * add(op:String, operatorType:OperatorType, priority:String, callback:EitherType<(InterpTokens) -> InterpTokens, (InterpTokens, InterpTokens) -> InterpTokens>)
    - ConstantPool
      * ERROR:MemoryPointer
      * EXTERN:MemoryPointer
    - Storage
      * storeByte(b:Int):MemoryPointer
    - Memory
      * store(token:InterpTokens):MemoryPointer
      * read(...path:String):{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}­­
* **יכולת: קריאה וכתיבה ישירה ובטוחה מהזכרון**
  + אם מפתח צריך, הוא יכול להשתמש במגוון הפונקציות שבמחלקה Memory על מנת לקרוא ולערוך ערכים בזכרון, בכל זמן שהוא.
  + שדות ומחלקות נחוצים:
    - **--- להשלים במועד מאוחר יותר---**
* **יכולת: גישה מהירה לזכרון שמפונה ע"י הInterpreter**
  + אם מפתח צריך, הוא יכול לגשת למערכי הבתים של Storage וגם של Referrer, ולהשתמש בפונקציות שבאותן מחלקות כדי לקבל מידע על אותם מערכי בתים.
  + שדות ומחלקות נחוצים:
    - ---להשלים במועד מאוחר יותר
* **יכולת: העברה בין low level ל-high level**
  + המפתח יכול לבחור אם הוא רוצה לתת למשתמשים בשפה גישה לפעולות, שדות ואופרטורים מסוימים בשביל לתת למשתמשים חווית תכנות ב-low level, או להפעיל את אוסף האשפה מלכתחילה כדי לתת חווית high level
  + שדות ומחלקות נחוצים:
    - ---להשלים במועד מאוחר יותר---
* **יכולת: המרה חלקה בין ערכי Little לערכי Haxe**
  + למפתח יש גישה למחלקה Conversion, המאפשרת המרות בין ערכים בLittle, שמיוצגים בעזרת אובייקטים מסוג InterpTokens, לערכים רגילים, שסוגם משתנה (Int, Float או אפילו סתם אובייקט דינמי)
  + שדות ומחלקות נחוצים:
    - InterpTokens
    - Type
      * getClass<T>(o):Class<T>
      * getClassName(c:Class<Dynamic>):String
      * getInstanceFields(c:Class<Dynamic>):Array<String>
      * getProperty(o:Dynamic, field:String):Dynamic
    - Reflect
      * getProperty(o:Dynamic, field:String):Dynamic
    - Interpreter
      * evaluate(exp:InterpTokens):InterpTokens
* **יכולת: החלפת מילים שמורות ומילים בספרייה הסטנדרטית**
  + המפתח יכול, במידת הצורך, להחליף מילות מפתח כאלה או אחרות לאיזה צירוף אותיות רציף שהוא רוצה, כל עוד הצירוף לא מכיל רווח, סימן, אינו ריק ולא מתחיל במספר. הוא עושה זאת דרך שינוי הערכים שב-Little.keywords, השוואת אותו השדה ל-KeywordConfig אחר, או יצירת KeywordConfig חדש והחלתו באמצעות הפעולה change של אותה מחלקה (KeywordConfig).
  + שדות ומחלקות נחוצים:
    - ---
* **יכולת: הדפסה יפה של אלמנטים חשובים**
  + המפתח יכול להשתמש במחלקה PrettyPrinter על מנת לקבל הדפסה יפה של קוד, או של עצי syntax. זה שימושי כשמנסים להוסיף תכונות ופיצ'רים לשפה, ורוצים לנפות באגים, או לראות איך התוספת שלך מסתדרת עם שאר עץ הsyntax.
  + שדות ומחלקות נחוצים:
    - PrettyPrinter
      * printParserAst(ast:Array<ParserTokens>, ?spacingBetweenNodes:Int = 6):String
      * printInterpreterAst(ast:Array<InterpTokens>, ?spacingBetweenNodes:Int = 6):String
      * getTree\_PARSER(getTree\_PARSER(root:ParserTokens, prefix:Array<Int>, level:Int, last:Bool):String
      * getTree\_INTERP(root:InterpTokens, prefix:Array<Int>, level:Int, last:Bool):String
      * stringifyParser(?code:Array<ParserTokens>, ?token:ParserTokens):String
      * stringifyInterpreter(?code:Array<ParserTokens>, ?token:ParserTokens)
      * prettyPrintOperatorPriority(priority:Map<Int, Array<{sign:String, side:OperatorType}>>):String

### סוג לקוח: לומד תכנות, מתכנת **Little**

מעבר לכלי למפתחים, גם משתמשים רגילים יכולים להשתמש בפרויקט, דרך שני הלקוחות שנתונים בו – אחד דרך הטרמניל (שורת הפקודה), ואחד [באתר אינטרנטי](https://spacebubble.io/little/demo/), שתוכנו נמצא בפרויקט:

* באתר האינטרנט, יש כמו עורך קוד מיוחד, בעל 3 חלקים – בחלק השמאלי של המסך, יש שלושה חלונות שניתן לכווץ: הקלט, עץ הsyntax והפלט. בחלק הימני למעלה, יש ממשק המאפשר שינוי של המילים השמורות, עם דוגמאות של מימוש בקוד. בחלק הימני למטה, יש מדריך Quickstart לשפה.
* מריצים את עורך הקוד בשורת הפקודה בעזרת בניית הפרוייקט באמצעות haxe compile.hxml, או באמצעות שימוש באפליקציות למגוון הפלטפורמות שנמצאות בפרויקט עצמו (הערה חשובה – אין אפליקציות לכל הפלטפורמות, מכיוון שצריך את המכשיר מהסוג הספציפי על מנת לבנות אליו לפעמים (iphone, mac, ולפעמים linux)).  
  כשמריצים את הפרויקט "raw", הוא ידפיס קצת מידע, ואחריו יהיה אפשר להקליד קוד. מעבר לקוד, אפשר להקליד פקודות מסוימות, והן רצות כאשר הן בשורה ריקה:
  + ml! – מנקה את המסך, ומתחיל להקליד קוד במצב רב-שורתי, ולא מריץ קוד מיד אחרי לחיצה על מקש הenter
  + clear! – מנקה את המסך, מתחילים להקליד קוד במצב הקודם ששהינו בו
  + clearLine! – מוחק את השורה האחרונה, זמין במצב רב-שורתי בלבד
  + run! – מריץ את הקוד שהוכנס עד עכשיו במצב רב-שורתי
  + default! – מחזיר למצב שורה יחידה ומנקה את המסך
  + ast! – מנקה את המסך, ומעביר למצב שמדפיס את עץ הsyntax של הקוד במקום להריץ אותו.

על אף השוני בין סוגי הUI, הרוב המוחלט של הפיצ'רים מוצעים לשניהם, ולכן אפרט על היכולות שלהם ביחד

* **יכולת: סביבת הרצה cross-platform**
  + מכיוון שהפרויקט לא משתמש בשום פיצ'ר שספציפי לפלטפורמה מסוימת, אפשר לקמפל אותו לכל מטרה, ואפילו להריץ אותו בעזרת הinterpreter של haxe עצמו.
* **יכולת: הרצת קוד מהירה/אוטומטית**
  + בלקוח האינטרנטי, כל פעם שהמשתמש משנה את הקוד, הוא נבנה מחדש והפלט מוצג. בלקוח של שורת הפקודה, תלוי במצב: במצב שורה יחידה, הלקוח משתמש בלחצן הenter בשביל להריץ את הקוד. במצב מולטי-שורה, הלקוח מקליד run! בשורה ריקה.
* **יכולת: הכרזת משתנים, פונקציות, וכינויים**
  + שפת התכנות נותנת למשתמשים לצור פונקציות שמחזירות כל ערך, ומשתנים מכל סוג בעלי כל ערך שיש בו הגיון. כאשר נותנים למשתנה ערך שהוא לא משתנה, זה נקרא "כינוי", והמשתנה מתנהג כמו הערך שהוצמד לו (אפשר לשים אותו בין אופרנדים לדוגמה עם הערך שלו הוא אופרטור)
* **יכולת: המרה מהירה ופשוטה בין סוגי ערכים**
  + מילת המפתח המשמשת לנתינת סוג קונקרטי למשתנים גם משמשת ל"יציקה" (casting). פונקציות היציקה מוגדרות בעזרת פונקציה שנמצאת על המחלקה של האובייקט שיוצקים, ושם הפונקציה הוא <to\_keyword><type>, כאשר to\_keyword זו מילת המפתח המוגדרת שאיתה מתחילים שמות של פונקציית המרה (ברירת מחדל : to), ו-type זה שם הסוג.
* **יכולת: ראיית עץ הsyntax**
  + בלקוח האינטרנטי, מעל הפלט הסטנדרטי, מופיע עץ הsyntax שנבנה מהקוד שלך. בלקוח של שורת הפקודה, יש להעביר למצב ראיית ה-AST בעזרת הפקודה ast!, ולאחר מכן הכנסת קוד כרגיל.
* **יכולת: שגיאות מודעות קונטקסט**
  + בכל מקום שבו נזרקת שגיאה, נעשה מאמץ לקשור את השגיאה למקום שממנו היא נקראה – השגיאה מכילה שמות משתנים, פונקציות, ואפילו קוד מהמקום בו נזרקה השגיאה, ובמקרה הצורך, אפילו חושפת מידע שהמשתמש אולי לא ידע (לדוגמה, תדווח שגיאה ספציפית ללולאות for המשתמשות בבלוק של קוד כדי לאחזר את המשתנה, אך הבלוק לא החזיר אחד)
* **יכולת: גישה לא בטוחה לזכרון**
  + כברירת מחדל, יש למשתמשים בשפה גישה למחלקה ששמה Memory, המספקת מספר פעולות ומשתנים העוזרים עם גישה והתעסקות "לא בטוחה" עם הזיכרון
* **יכולת: קבלת כתובות של ערכים בזיכרון**
  + כברירת מחדל, לכל ערך מאופשר השדה address, המחזיר את המקום בזיכרון של אותו ערך, בין אם סתם ערך, משתנה, או פונקציה.
* **יכולת: אחזור סוג של ערך בזמן ריצה**
  + לכל ערך מאופשר השדה type, המחזיר את שם הסוג של ערך מסוים. הוא לא מבדיל בין סוגים של מידע – לפונקציות זה יחזיר Function, ולסוגים זה מחזיר Type. (כמובן שאפשר לשנות את שמות הסוגים, לפחות בלקוח האינטרנטי)
* **יכולת: שינוי מילים שמורות**
  + זמין בלקוח האינטרנטי – המשתמש יכול בעצמו לשנות את המילים השמורות לשפה אחרת, סגנון אחר, או סתם לקצר.
* **יכולת: מגוון פעולות "broadcast"**
  + המשתמש יכול להשתמש ב3 פונקציות על מנת להדפיס דברים לקונסולה:  
    print – הדפסה רגילה, עם מודולה ושורה  
    warn – כמו print, אבל עם ההקדמה "WARNING". קוד ממשיך לרוץ  
    error – גם כמו print, עם ההקדמה "ERROR". קוד מפסיק לרוץ
* **יכולת: קריאה דינמית של משתנים**
  + פעולת read יודעת להחזיר את הערך של משתנה, רק לפי שמו או מסלול המוביל אליו, כשהוא מובא כמחרוזת. זה כלי חזק מאוד, המאפשר שימוש מותנה במשתנים מסוימים, וגם תכנות עם כל מני אפשרויות "מאקרו-איות".
* **יכולת: הכרזה דינמית של משתנים**
  + כאשר מכריזים על משתנה/פונקציה, שם השדה יכול להיות בלוק של קוד שמחזיר מחרוזת. יש לציין שלא כל מחרוזת מותרת: מחרוזת חייבת להיות בעל לפחות אות אחת, ללא רווחים, ללא אופרטורים, ולא להתחיל במספר (כמובן, שגיאות ייזרקו בהתאם, תלוי במעבר על הכללים)

# מבנה הפרויקט

## ארכיטקטורה

### רכיבים שונים, והקשרים ביניהם

כמו שנאמר מקודם (תחת הכותרת של "מערכת"), הפרויקט עצמו מורכב מ3 חלקים – שפה (ספציפיקציה), הקומפיילר, או, יוצר ה-AST, והאינטרפרטר, או, המכונה הוירטואלית. בזמן שהספציפיקציה כמעט ולא נמצאת בתוך הקו, שני החלקים האחרים של הפרויקט מאוד מופרדים, ונמצאים בתקיות שונות, ואחד לא משתמש בשני. אותם שני חלקים גם משתמשים בכלים גלובליים, היכולים להסתמך בחזרה על אותן מערכות במקרי קצה יחסית ספציפיים (לדוגמה, קבלת סוג של ערך, כאשר הערך מובא בתור מזהה). אערוך את הזרימה בין המערכות ובין המחלקות בתור רשימה, ואביא גם שרטוט. הבה נתחיל:

* + - קומפיילר:
  + Lexer – לוקח את הקלט מהמשתמש, בצורת מחרוזת. המחלקה מפצלת את הטקסט לטוקנים, המופרדים לפי סוגם – מזהה, מספר, וסימן. כאן נמצא השימוש הראשון בכלים הגלובליים, על מנת לזהות מהו סימן בדיוק. לאחר ההרצה, אנו נמצאים עם מערך טוקנים פשוט:
  + Parser – לוקח את הטוקנים מהLexer, וממיר אותם לכאלה בצורה שנוח לו להתעסק איתה. לאחר מכן, מעביר את הטוקנים האלה סדרה של פונקציות, אחת אחרי השנייה, ומפתח עץ syntax יותר ויותר "מסובך" – כל שלב נועד לתת מענה לתכונה אחרת בשפה, ולכן אפשר לראות שכל שלב מעמיק וגם מקצר קצת את העץ. מדובר בקובץ מאוד ארוך (כ950 שורות), המכיל את רוב הקומפיילר, גם פיזית וגם לוגית. עכשיו, התוצאה שקיבלנו קרובה מאוד לעץ הsyntax הרשמי. קיבלנו את העץ הסופי, עכשיו הגיע הזמן ל"ייצא" אותו, ולהשתמש בו:
    - אינטרפרטר:
  + Interpreter.convert – לוקח את העץ שקיבלנו מהקומפיילר, וממיר אותו לטוקנים הנוחים לשימוש באינטרפרטר.
  + PrepareRun – לפני הרצה, מוסיף את כל המשתנים, הפונקציות, האופרטורים, הסוגים והשדות ומייצר את הספריה הסטנדרטית, באמצעות:
    - * Plugins – מכיל מגוון פעולות להוספת אלמנטים חיצוניים לשפה, ממשתנים רגילים ועד לשדות על סוגים ספציפיים ואופרטורים. משתמש במחלקה ExternalInterfacing, שעליה אפרט בהמשך.
  + Interpreter.run – לאחר מכן, אנו לוקחים את כל העץ, ומריצים אותו בעזרת פונקצית ההרצה העיקרית. הפונקציה משתמשת במגוון הפעולות שעל המחלקה Interpreter על מנת לממש את ההרצה, קוראת לאיוונטים שבמחלקה Runtime כשמצופה, וכשצריך לשמור מידע, היא משתמשת במחלקה Memory. אין סדר קבוע לקריאות האלה, שכן הן תלויות בקלט מהמשתמש. אפרט עם השלושה:
    - * Interpreter – מכיל פונקציות המפעילות את הטוקנים מעץ הsyntax – הכרזת משתנה, חישוב ביטוי, קריאה לתנאי...
      * Memory – מכיל פונקציות המאחסנות, קוראות וכותבות זכרון בעזרת 5 מחלקות אחרות:
        + Storage – מערך זכרון ענק, מספק פונקציות של כתיבה, עריכה ובקשת הרחבה של אותו מערך זכרון, לפי מצביעים המעוצבים כדי לגשת לאותו מערך
        + Referrer – גם מערך זכרון, אך קטן בהרבה – מקשר בין שמות משתנים לערכם, תלוי במקום ממנו הם מבוקשים
        + ExternalInterfacing – המגשר הסופי בין ערכי haxe לערכי Little, מייצג את ערכיו כעצים ולא כמערך בתים
        + ConstantPool – בריכה של ערכים נפוצים, קיים על מנת לחסוך בזכרון. לוקח זכרון מ-Storage.
        + HashTables – מיצר, קורא וכותב מפות hash בעזרת האלגוריתם MurmurHash1. בעזרתו אפשר לגשת לאובייקטים.
      * Runtime – מכיל איוונטים שונים, שאמורים להקרא כל פעם שדבר מסויים מתבצע ע"י האינטרפרטר – טוקן נקרא, משתנה מוכרז...  
        גם מכיל מידע על הריצה עצמו (כמו הפלט לדוגמה), ופונקציות ההדפסה למיניהן (print, warn, throwError, broadcast). broadcast דומה ל-print, אך אפשר לערוך אותו – נועד למפתחים המשתמשים בפרויקט הזה כספריה לאפליקציות שלהם.

## טכנולוגיה

### שפת תכנות

הפרויקט מתוכנת בשפת תכנות ששמה Haxe.

Haxe היא שפה, טרנספיילר ואינטרפרטר רב-מטרתי, המאפשר בניית אפליקציות למגוון שפות תכנות, קודי-בתים (bytecode), ואפילו הרצה על המקום בעזרת אינטרפרטר מיוחד.

Haxe מאפשרת בניית קוד למגוון שפות תכנות אחרות: JavaScript, C, C++, C#, Java, Lua, PHP, Python ו-ActionScript3.  
היא גם מאפשרת ייצור קוד-בתים ותמיכה מיוחדת לכמה פלטפורמות: יש תמיכה בNodeJS ואפשר לצור קובצי SWF ישירות. אפשר גם לקמפל קוד בתים למכונות הווירטואליות Neko ו-Hashlink, וכן קוד-בתים JVM.

Haxe, כמו הפרויקט הזה, הוא פרויקט open source, תחת רשיון MIT. הפיתוח שלו התחיל ב2005, ע"י Nicolas Cannasse, יוצר ואחד ממפתחי הקומפיילר [MTASC](https://github.com/ncannasse/mtasc).  
 הטרנספיילר והאינטרפרטר של השפה כתובים ב-OCaml, שפה הידועה ביכולות עיבוד המידע שלה, ולכן גם שפה טובה לכתיבת שפות תכנות.

סגנון התכנות של השפה ותכונותיה לקוחות משפות כמו JavaScript וC#, אך היא גם מציעה תכנות יחסית יחודיות, שעוזרות בעיבוד מידע, כמו Algebraic Data Types וVariable Capturing בתנאי switch.

### תחומי עניין

באופן כללי, הפרויקט מתעסק בעיבוד, אחסון ומניפולציית מידע בסדר גודל גדול, ויותר בפוקוס: קומפיילרים, אינטרפרטרים, וקידוד מידע.

קומפיילר הוא תוכנה הממירה סוג תוכן מסוים, בדרך כלל קוד בשפת תכנות מסויימת, מילפורמט הניתן להרצה, בדרך כלל קוד מכונה. לרוב, קומפיילר בנוי מ-3 שלבים: Lexer, Parser, ולבסוף, Generator. בפרויקט הזה, שני השלבים הראשונים קיימים באופן מלא, והשלב השלישי מתממש ע"י יצירת קוד-בתים של עץ הsyntax ש"נפלט" בסוף.

אינטרפרטר היא תוכנה המקבלת מערך של פיסות מידע, ומפעילות רכיבים מסוימים בצורה שבד"כ מבצעת אינטראקציה עם המשתמש. האינטרפרטר לרוב מורכב מכמה "מרכזי ניהול" עיקריים: זכרון, אינטראקציה עם המערכת, וכמובן, "סורק" קוד.

קידוד מידע הוא תחום החופף הצפנת מידע – כמו בהצפנה, המידע משנה צורה, אך כאן המטרה היא, בדרך כלל, לכווץ אותו או לפרמט אותו בדרך שקלה לקרוא/לכתוב. דוגמה טובה היא אלגוריתמי hash – המטרה אינה בהכרח להצפין, אלא להמיר מידע למידע אחר באורך סטטי אך עם תוכן משתנה, בהתאם לקלט.

## זרימת מידע במערכת

### הקומפיילר

השלב הראשון ב"מימוש" הקוד. מכיל מספר יחסית קטן של רכיבים יחסית גדולים:

**---לעתיד---**

## אלגוריתמים מרכזיים

בכותרות הבאות, אפרט על בעיות מסוימות, שהשתמשתי/פיתחתי אלגוריתמים כדי לפתור אותם.

### קישור בין משתנים למקומות בזיכרון

כשרוצים לאחסן ערך מסוים, לא מספיק רק לשמור אותו בזיכרון – צריך גם לדעת איפה הוא נמצא, ומה הגודל שלו.

לבעיה יש מגוון פתרונות קיימים:

יצירת Stack – תוכנה מפנה לה בצד כמות קטנה יחסית של מידע, ושומרת בצד שני ערכים – מצביע לתחילת הstack המדובר, ומצביע שמעיד על התחלת ה"מסגרת" הנוכחית של הstack. במבנה הזה אפשר לשמור מידע בגדלים שרירותיים, בצורה שהערך שנוסף אחרון נמצא גם אחרון, לפני המידע הלא מנוצל. מסגרת חדשה נוצרת כל פעם שנוספת קריאה למקום מסוים, כמו בלוק של קוד או פונקציה, וכל המסגרת נמחקת בסוף ההרצה של אותו מקום. מקורות:

* + <https://en.wikipedia.org/wiki/Stack-based_memory_allocation>
  + <https://www.geeksforgeeks.org/stack-implementation-in-operating-system-uses-by-processor/>

יצירת Heap – תוכנה מקבלת בעת התחלתה כמות מסוימת של מידע, ויכולה לבקש עוד. כשמבקשים לאחסן משתנה, בהתאם לאימפלמנטציה של ה-Heap, מפונה כמות מסוימת של מקום המתאימה לפחות לגודל המשתנה, ותלוי באימפלמנטציה, יכולים לאחסן במקום מסוים בתוך ה-Heap מידע מסוים לגבי בקשות אחסון, כמו המקום שלהם וגודלם לדוגמה. מקורות:

* + <https://en.wikipedia.org/wiki/Memory_management#HEAP>
  + <https://discourse.julialang.org/t/a-nice-explanation-of-memory-stack-vs-heap/53915>

בסוף, על אף איכות הפתרונות הקיימים, הלכתי על פתרון אחר, שיותר מתאים לסגנון הפרויקט, וקצת יותר פשוט. קראתי לפתרון "Referrer".

הReferrer מורכב ממספר מפות של כתובות הנמצאות אחד אחרי השני, ומופרדות באמצעות "כותרות", המכילות את מיקום הבלוק ככתובת, ואורך המפה הקודמת והנוכחית כמספר בעל 16 ביטים (ערך מקסימלי: 65535). בצד, בדומה ל-Stack, אנו שומרים את מיקום הכותרת האחרונה, אך בשונה, גם את האורך **הנוכחי** שלה. כל פעם שמוכרז קשר כלשהו בין מזהה לערך, נוספת למפה האחרונה זוג מפתח-ערך, כאשר ה-hash של המזהה הוא המפתח, והכתובת + סוג הערך המוצמד למזהה מהווים את הערך של המפתח במפה. כל פעם שנפתחת מסגרת חדשה (התחלת בלוק של קוד: קריאה לפונקציה, תנאי...) נוספת כותרת חדשה עם מפה חדשה, וההפך קורה כשנגמר בלוק הקוד. זה מאפשר variable shadowing – הגדרת משתנה לאותו שם פעמיים, כשאחד מההכרזות נמצאות במסגרת "עמוקה יותר".

ההבדל העיקרי בין הפתרון הזה לאחרים שהזכרתי מקודם, שהוא גם הסיבה העיקרית לבחירה בו, היא דווקא זה שהשימוש בפתרון הנוכחי הוא יחיד – בזמן שה-Stack וה-Heap מספקים ביחד פתרון לאחסון + מאגר אזכורים, כבר הייתה לי מערכת אחסון, ולא רציתי לתכנת משהו שאשתמש רק בחלק ממנו, כי זה 1) נראה לא טוב ו2) משאיר מקום לטעות עם מפתחים שמשתמשים בפרויקט כספריה, ולכן לקחתי את הפתרון המקורי הזה – משומש אך ורק לאזכורים, ועושה זאת בצורה מסודרת בלי לקחת כמות "הזויה" של זכרון.

### חישוב והערכת ביטויים

חלק בלתי נפרד מתכנות זה הערכת ביטויים על מנת לייצר ערך. הבעיה היא, זה לא כל-כך פשוט – יש סדרי פעולות חשבוניות, סוג הביטוי יכול להשתנות באמצע, וכן הלאה. יש פתרון אחד ידוע לזה, והשתמשתי בגרסה קצת שונה שלו, על מנת להתחשב בפיצ'רים מסוימים של השפה (הגדרת אופרטורים בזמן ריצה וכינויים, כמו define add = +)

הפתרון קיים, והגרסת ה"השראה" שלו:

Shunting Yard –לוקחים את הביטוי המתמטי כמערך של מזהים, ויוצרים סטאק ריק שאמור להכיל את האופרטורים, ומערך פלט שאמור להכיל את הביטוי המסודר, לפי פורמט שהוחלט מראש (סימון פולני, סימון פולני הפוך או עץ syntax). מערך הפלט נבנה כך – לכל טוקן בביטוי המקורי:

* + אם הוא מספר, מוסיפים אותו ישר לפלט
  + אם הוא סימן/פונקציה, יש כמה אופציות:
    - אם הסימן בדרגה גבוהה יותר מהסימן שבראש הסטאק, הוא נוסף לסטאק
    - אם הסימנים באותה דרגה/הסימן החדש בדרגה נמוכה יותר, הסימן הקיים נוסף לפלט ומורד מהסטאק, במקומו מוסיפים את הסימן החדש

מקורות:

* + <https://en.wikipedia.org/wiki/Shunting_yard_algorithm>
  + <https://brilliant.org/wiki/shunting-yard-algorithm/>

גרסה בפרויקט:

Shunting Yard (Inspired) – האלגוריתם מפוצל לשני שלבים:

* + group – לפי האופרטורים המוגדרים ולפי ערכי המזהים בקלט, מרכיב עץ syntax המורכב מביטויים קטנים יותר באורך של עד 3 טוקנים, המכילים עד 2 ערכים וסימן אחד. לא נוצר סטאק לאופרטורים, במקום זה, אנו עוברים על האופרטורים בקבוצות, מהחשובים ביותר להכי פחות חשובים, וכך יוצא שטוקנים בעלי סימן חשוב יותר, לדוגמה, ^ (חזקה) יקובצו מוקדם יותר מכאלה ברמה נמוכה יותר, לדוגמה, +. בעצם, כל הפעולה נעשית קצת כמו מיון מערך In-place. בסוף, יוצא שביטוי שנכנס ככה:
    - 2 + 5! \* 3 ^ -4

יצא ככה:

* + - (2 + ((5 !) \* (3 ^ (- 4))))
  + calculate – יוצרים 3 משתנים – ערך נוכחי, סימן נוכחי, ערך עד עכשיו. עוברים על עץ הsyntax שgroup מייצר, ועבור כל טוקן:
    - אם הוא מספר, מציבים אותו ב"ערך נוכחי", ואם יש ביטוי, קוראים שוב לcalculate עליו
      * אם יש כבר סימן, זה אומר שהביטוי חייב להסתיים. מחשבים את הביטוי כאשר "ערך עד עכשיו" בצד שמאל ו"ערך נוכחי" בצד ימין, ומציבים את התוצאה ב"ערך עד עכשיו".
      * אם אין סימן ו"ערך עד עכשיו" לא מוגדר, ערכו הופך לערך הזה.
      * אם אין סימן ויש "ערך עד עכשיו" נזרקת שגיאה ששני מספרים באו אחד אחרי השני
    - אם הוא סימן מציבים אותו ב"סימן נוכחי".
      * אם הסימן הוא הטוקן האחרון, מחשבים את הביטוי כאשר "ערך עד עכשיו" בהתחלה, והסימן בסוף.

ההבדל העיקרי בין האלגוריתם שלי לאחד הקלאסי הוא השינוי מאחסון המידע בסטאק ופלט למודיפיקציה In-place (גם לאחד הקלאסי יש גרסה של אחסון באמצעות עץ syntax דומה לאחד שאני מרכיב, לכן זה לא הבדל). הסיבה היחידה להבדל היא נוחות ועקביות – הרוב המוחלט של הפרויקט מתכונת בעזרת פונקציות רקורסיביות, ואני מרגיש מאוד בנוח איתן, אז החלטתי שגם האימפלמנטציה של האלגוריתם הזה יהיה רקורסיבי.

### **Hashing**

כשתכנתתי ניהול זיכרון, החלטתי שאובייקטים יאוחסנו כמפות hash של השדות שלהם. לכן, הייתי צריך להחליט על אלגוריתם hashing. ישנן יחסית הרבה אופציות זמינות, אך לקחתי אחת סבירה יחסית, מאילוצים:

SipHash – אלגוריתם hashing המשתמש בשיטת Add -> Rotate -> Xor על מנת לייצר hashים. אלגוריתם זה אינו קריפטוגרפי, או במילים אחרות, אינו נועד להצפנה. הוא קיים על מנת להציב אלגוריתם אלטרנטבי שטוב למלחמה במתקפות התנגשות, בהם התוקף מנסה למצוא שני ערכים המנסים למצוא את אותו הhash. אם משתמשים בSipHash, גם אם התוקף יודע את המפתח ואת תוצאת האלגוריתם, או את ערך שעושים לו hashing ואת תוצאת האלגוריתם, לא יהיה ניתן לנחש את הנתון החסר. מקורות:

* + <https://en.wikipedia.org/wiki/SipHash>

MurmurHash – אלגוריתם hashing המבוסס על אותה שיטה (ARX). גם הוא לא קריפטוגרפי, ומטרתו להיות פונקציית hashing מהירה. יש לה חסינות סבירה להתנגשויות, אך לא חסינה למתקפות מכוונות. MurMur היא משפחה של פונקציות hashing בעלת 3 דורות:

* + MurMur1 – הגרסה הראשונה, ניסתה להיות שיטה מהירה יותר של Lookup3, והצליחה. מאפשרת hashing רק אל תוך ערכים בעלי 32 ביטים
  + MurMur2 – הגרסה השנייה, בעל המון גרסאות אלטרנטיביות. פשוט גרסה יותר טובה של MurMur1 עם יותר אופציות
  + MurMur3 – הגרסה השלישית והעדכנית. שיפור על הגרסה השניה, מציע גם ייצור hashים באורך 128 בתים.

בחרתי באלגוריתם MurMur1 ממשפחת MurMur – ספציפית בחרתי בסוג האלגוריתמים מאילוץ, ובחרתי בדור הראשון מכיוון שהוא התאים טוב למקרה שלי – רציתי להשתמש באלגוריתם כדי לייצר אינדקסים במפת hash, והדור הראשון מייצר מספרים באורך 32 בתים, שלמקרה השימוש שלי מספיק.

## סביבת עבודה

### פיתוח

על מנת לתכנת ולהריץ את הקוד, יש להוריד את הקומפיילר של Haxe, ומספר ספריות:

Haxe Compiler – גרסה .14.3 לפחות

vision – מגיטהאב, בטא של גרסה 2.0.0 (לפחות)

hash – גרסה 1.0.1

hxOrderedMaps – לא מפורסם רשמית, מגיטהאב.

את הספריות ניתן להתקין באמצעות haxelib, המותקן באופן אוטומטי כשמתקינים את הקומפיילר של Haxe. את הספריות הרשמיות ניתן להתקין באמצעות:

haxelib install <libname>

או, עם ספריות מגיטהאב/גיטלאב/כל שירות אחר, באמצעות:

haxelib git <libname> <link\_to\_repository>

### בדיקה

לפני שבודקים את הקוד, יש צורך להוריד את התוכנות והספריות שהוזכרו בפיתוח. לאחר מכן, ניתן לבדוק בשלושה דרכים:

לקוח אינטרנטי – יש לשים לב שהשורות הבאות בקובץ compile.hxml מבוטלות (באמצעות הוספת # בתחילתן):  
--interp#  
#--define unit  
והשורה הזאת פועלת (באמצעות מחיקת ה#)  
--js interp.js  
לאחר מכן, בונים את התוכנה באמצעות haxe compile.hxml, ונכנסים/מרעננים את הקובץ index.html הנמצא בשורש של הפרויקט.

לקוח שורת פקודה – יש להחליף את ה-# בין השורה של --interp ו --js interp.js, וכן להשאיר את unit מבוטל:  
--interp  
#--js interp.js  
#--define unit

לאחר מכן, כמו קודם, בונים את הפרויקט, ומקבלים קצת מידע ושני "משולשים" בשורת הפקודה, ומשם ניתן לבדוק את הפרויקט. יש לבנות מחדש כדי לשנות את הפרויקט

unit testing – יש לשים לב ש --js interp.js מבוטל, ו --interp,  
 --define unit לא מבוטלים:  
--interp  
#--js interp.js  
--define unit

לאחר מכן, בונים את הפרויקט, וכל 14 הבדיקות יורצו, אחת אחר השניה, וידווחו עם הבדיקה הצליחה, ואם לא מה הערך הלא צפוי שהוחזר, עץ הsyntax, הקוד, והפלט.

# מימוש הפרויקט

## חלקים ומחלקות

כמו שהוזכר קודם לכן, הפרויקט מורכב ממספר חלקים העובדים ברצף, אך אינם מסתמכים אחד על השני, וחלק מיוחד הנקרא tools, המספק כלים לשאר החלקים. ישנם 3 חלקים "מג'וריים", בעוד שאחד מהם מכיל בתוכו עוד חלק גדול יחסית (הזכרון). אתייחס לחלק הזה ולחלק-הורה שלו בנפרד.

קיימים ארבעה סוגים עיקריים של מבני מידע בפרויקט:

מחלקה (class) – הסוג הקלאסי, יכול לייצג אובייקט ואת עצמו. מכיל שדות סטטיים ותלויי מקרה

אבסטרקט (abstract) – סוג של זמן קומפילציה בלבד, מתקמפל בסוף לסוג אחר שהוא בנוי עליו. שדות סטטיים קיימים בסוף על אובייקט נפרד, והקוד של שדות תלויי מקרה מוצבים בשורה בה מבוקש הערך. נמצא בשימוש מטעמי מהירות.

ספירה (enum) – הספירה הקלאסית, בתוספת התכונה המיוחדת של יצירת [algebraic data types](https://code.haxe.org/category/beginner/enum-adt.html).

הגדרה (typedef) – דרך מיוחדת ליצור כינוי, או לתת מבנה קונקרטי לאובייקט דינמי, כך שיהיה חייב להכיל את השדות המוזכרים בהגדרה. ערכו יכול להיות או סוג/אבסרקט/וכו', או "אובייקט" בעל שדות המוגדר באמצעות סוגריים מסולסלים. [קישור להסבר מורחב](https://haxe.org/manual/type-system-typedef.html)

### חבילה – little.lexer

ספירה – LexerTokens – מכיל את הטוקנים שמשתמשת בהם המחלקה Lexer, על מנת להפוך קוד בפורמט מחרוזת, לקוד המפוצל למזהים:

* + Identifier(name:String) – מייצג מזהה ששמו <name>
  + Sign(char:String) – מייצג אופרטור שסימנו <char>
  + Number(num:String) – מייצג מספר כלשהו שערכו <num>
  + Boolean(value:String) – מייצג ערך בוליאני שערכו true/false
  + Characters(string:String) – מייצג מחרוזת שתוכנה <string>
  + NullValue – מייצג את המזהה null
  + Newline – מייצג שורה חדשה
  + SplitLine – מייצג פיצול בשורה
  + Documentation(content:String) – מייצג דוקומנטציה/תגובה
* מחלקה – Lexer – מכיל פונקציות שביכולתן להפוך מחרוזת המכילה קוד (הכתוב בLittle) למערך מזהים מסוג LexerTokens:
  + lex(code:String):Array<LexerTokens> – מקבל קוד הכתוב בLittle, וממיר אותו למערך מעובד של טוקנים מסוג LexerTokens, בעזרת הפונקציות הבאות. עובר על הקוד אות אות.
  + separateBooleanIdentifiers(tokens:Array<LexerTokens>):Array<LexerTokens> – מקבל מערך של טוקנים מסוג LexerTokens, ומזהה אילו מה-Identifier(name:String) מכילים בתוך הname שלהם ערכים בוליאניים, ומפצל אותם לBoolean או NullValue.
  + mergeOrSplitKnownSigns(tokens:Array<LexerTokens>):Array<LexerTokens> – מקבך מערך של טוקנים מסוג LexerTokens, ומזהה מתי קומבינצית סימנים היא נכונה, ומתי צריך לפצל אותה. זה הכרחי, מכיוון שלא כל האופרטורים הם אות אחת, ולכן צריך לדעת מתי הסימן הוא בודד ומתי באורך מסויים.

### חבילה – little.parser

ספירה – ParserTokens – מכיל את הטוקנים שמשתמשת בהם המחלקה Parser, על מנת להפוך את הקוד מקבוצת מזהים לעץ syntax שלם.

* + SetLine(line:Int) – טוקן המביע צורך לשנות את מספר השורה הנוכחית למספר אחר
  + SplitLine – מביע צורך להפסיק את קריאת השורה ברצף, להגביר את מספר החלק הנוכחי בשורה הנוכחית.
  + Variable(name:ParserTokens, type:ParserTokens, ?doc:ParserTokens) – מבטא יצירת משתנה, ששמו, סוגו והדוקומנטציה שלו מבוטאות בעזרת טוקנים שונים. אתם הטורנים חייבים "להיפתר" למזהה/תגובה, בהתאם לפרמטר.
  + Function(name:ParserTokens, params:ParserTokens, type:ParserTokens, ?doc:ParserTokens) – מבטא יצירת פונקציה ששמה, סוגה, והדוקומנטציה שלה מבוטאים בעזרת טוקנים, הנפתרים למזהים/תגובות, בהתאם למקרה. params צריך להיות טוקן מסוג PartArray, המכיל הכרזות משתנה מסוג Variable.
  + ConditionCall(name:ParserTokens, exp:ParserTokens, body:ParserTokens) – מבטא קריאה לתנאי ששמו מבוטא בעזרת טוקנים הנפתרים לטוקן מסוג Identifier, "הפרמטר" של התנאי יכול להיות כל טוקן, והגוף של התנאי יהיה כל טוקן הניתן להרצה, בד"כ טוקן מסוג Block.
  + Write(assignees:Array<ParserTokens>, value:ParserTokens) – טוקן המייצג כתיבה של ערך, value, לקבוצה של מזהים, או לפחות טוקנים שאמורים להפתר למזהים (טוקנים מסוג Identifier).
  + Identifier(word:String) – מייצג את המזהה. בד"כ, כשנתקלים בו בקוד, יש למשוך את ערכו מהזכרון בהתאם לשמו.
  + TypeDeclaration(value:ParserTokens, type:ParserTokens) – מיצג יציקת ערך מסוים לסוג שאינו בהכרח של אותו ערך. לא מבטיח התאמה בין הערך לסוג.
  + FunctionCall(name:ParserTokens, params:ParserTokens) – מבטא קריאה לפונקציה, כאשר שמה מבוטא באמצעות טוקן כזה או אחר הנפתר לIdentifier, והפרמטרים מובאים באמצעות טוקן מסוג PartArray
  + Return(value:ParserTokens, type:ParserTokens) – מבטא החזרת ערך בפונקציה – כשמורץ, אמור לסיים הרצה לפני שכל הטוקנים סיימו להריץ
  + Expression(parts:Array<ParserTokens>, type:ParserTokens) – מבטא מספר טוקנים אחד אחרי השני, המחוברים בהיותם בתוך סוגריים רגילים.
  + Block(body:Array<ParserTokens>, type:ParserTokens) – מבטא מספר טוקנים, שמטרתם להיות מורצים, אחד אחרי השני. מבוטא בעזרת קוד הסגור בסוגריים מסולסלים
  + PartArray(parts:Array<ParserTokens>) – מבטא סתם מערך של טוקנים. לא יכול להמצא בקוד רגיל, נוצר באופן מלאכותי ע"י הParser.
  + PropertyAccess(name:ParserTokens, property:ParserTokens) – מבטא גישה לשדה על טוקן אחר. הפרמטר name יכול להיות בעצמו PropertyAccess, ובמקרה זה יש גישה "משולשת" אל תוך שדה.
  + Sign(sign:String) – מייצג אופרטור שסימנו <sign>
  + Number(num:String) – מייצג מספר שלם שערכו <num>
  + Decimal(num:String) – מייצג מספר עשרוני שערכו <num>
  + Characters(string:String) – מייצג מחרוזת שמכילה את האותיות <string>
  + Documentation(doc:String) – מייצג דוקומנטציה
  + ErrorMessage(msg:String) – מייצג טוקן שגיאה האומר <msg>
  + NullValue – מייצג את הערך null
  + TrueValue – מייצג את הערך הבוליאני true
  + FalseValue – מייצג את הערך הבוליאני false
  + Custom(name:String, params:Array<ParserTokens>) – טוקן שמשתמשים בשפה בתור כלי יכולים לצור בעצמם, וכך לעזור להם השחלת מאקרואים.
* מחלקה – Parser – מכיל פונקציות המאפשרות המרה של מערך המזהים שקיבלנו מLexer, לעץ syntax מפורט ושלם.
  + additionalParsingLevels:Array<Array<ParserTokens> -> Array<ParserTokens>> - מערך של שלבי עיבוד נוספים למידע שמייצרת הפונקציה parse. מורכב מפונקציות, שיקראו מיד אחרי שהבניה ה"רגילה" הסתיימה.
  + parse(lexerTokens:Array<LexerTokens>):Array<ParserTokens> – לוקח קוד המיוצג באמצעות מערך מזהים, ממיר אותו לטוקנים מסוג ParserTokens, ומעביר אותם מספר שלבי בניה, עד שנוצר עץ syntax שלם. שלבי הבניה הן הפונקציות הבאות, ואחריהן באות אלה שנמצאות בadditionalParsingLevels.
  + convert(lexerTokens:Array<LexerTokens>):Array<ParserTokens> – שלב ההמרה – עובר על כל הטוקנים מסוג LexerTokens, ממיר אותם, ומחזיר מערך של טוקנים זהים במשמעותם מסוג ParserTokens.
  + mergeBlocks(pre:Array<ParserTokens>):Array<ParserTokens> – ממיר כל קבוצת טוקנים המוקפת בסוגריים מסולסלים לטוקן מסוג Block.
  + mergeExpressions(pre:Array<ParserTokens>):Array<ParserTokens> – ממיר כל קבוצת טוקנים המוקפת בסוגריים רגילים לטוקן מסוג Expression.
  + mergePropertyOperations(pre:Array<ParserTokens>):Array<ParserTokens> – מאחד כל מקרה של טוקנים שיש ביניהם סימן שאמור להעיד על גישה לשדה, לטוקן מסוג PropertyAccess.
  + mergeTypeDecls(pre:Array<ParserTokens>):Array<ParserTokens> – מאחד כל מקרה שיש מזהה שמשמעותו "הדגשת" סוג, ואחריו טוקן אחר לTypeDeclaration
  + mergeComplexStructures(pre:Array<ParserTokens>):Array<ParserTokens> – יוצר הכרזות משתנים, פונקציות, החזרות פונקציה וקריאות לתנאים לפי רצפים מסוימים של טוקנים בקוד.
  + mergeCalls(pre:Array<ParserTokens>):Array<ParserTokens> – משלב טוקנים ספציפיים שאחריהם סוגריים ללא הפרדה לטוקן FunctionCall.
  + mergeWrites(pre:Array<ParserTokens>):Array<ParserTokens> – משלב רצפי השוואה לטוקן מסוג Write
  + mergeValuesWithTypeDeclarations(pre:Array<ParserTokens>):Array<ParserTokens> – משלב טוקנים שאחריהם בא TypeDeclaration ללא ערך, לTypeDeclaration יחיד עם ערך.
  + mergeNonBlockBodies(pre:Array<ParserTokens>):Array<ParserTokens> – מתקן מקרים של קריאות לתנאים בהם במקום לספק בלוק של קוד, המתכנת מספק שורת קוד/טוקן אחר.
  + mergeElses(pre:Array<ParserTokens>):Array<ParserTokens> – פונקצית הדוגמה שבתוך additionalParsingLevels, מוסיפה פיצ'ר – תומכת ברצפי else אחרי תנאי if.
  + line:Int – השורה הנוכחית עליה הParser נמצא
  + linePart:Int – החלק הנוכחי בשורה הנוכחית בו הParser נמצא
  + setline(l:Int) – עורך את השורה הנוכחית, מאפס את החלק הנוכחי
  + nextPart() – מגביר את החלק הנוכחי.
  + resetLines() – מאפס את השורה הנוכחית, מאפס את החלק הנוכחי.

### חבילה – little.interpreter

ספירה – InterpTokens – מכיל את הטוקנים שמשתמשות בהם מחלקות ההרצה, על מנת לממש את הטוקנים ולהפוך אותם לתוכנה עובדת.

* + SetLine(line:Int) – מייצג שינוי בשורה הנוכחית.
  + SplitLine – מייצג מעבר לחלק הבא בשורה הנוכחית.
  + VariableDeclaration(name:InterpTokens, type:InterpTokens, ?doc:InterpTokens) – מייצג הכרזת משתנה, ששמו, סוגו והדוקומנטציה שלו מובטאות בעזרת טוקנים שונים.
  + FunctionDeclaration(name:InterpTokens, params:InterpTokens, type:InterpTokens, ?doc:InterpTokens) – מייצג הכרזת פונקציה, ששמה, סוג ההחזרה שלה והדוקומנטציה שלה מבוטאים בעזרת טוקנים שונים. שדה הפרמטרים חייב להיות טוקן מסוג PartArray.
  + ConditionDeclaration(name:InterpTokens, conditionType:InterpTokens, ?doc:InterpTokens) – מייצג
  + ClassDeclaration(name:InterpTokens, ?doc:InterpTokens)
  + ConditionCode(callers:Map<Array<InterpTokens>, InterpTokens>)
  + ConditionCall(name:InterpTokens, exp:InterpTokens, body:InterpTokens)
  + FunctionCode(requiredParams:OrderedMap<String, InterpTokens>, body:InterpTokens)
  + FunctionCall(name:InterpTokens, params:InterpTokens)
  + FunctionReturn(value:InterpTokens, type:InterpTokens)
  + Write(assignees:Array<InterpTokens>, value:InterpTokens)
  + TypeCast(value:InterpTokens, type:InterpTokens)
  + Expression(parts:Array<InterpTokens>, type:InterpTokens)
  + Block(body:Array<InterpTokens>, type:InterpTokens)
  + PartArray(parts:Array<InterpTokens>)
  + PropertyAccess(name:InterpTokens, property:InterpTokens)
  + Number(num:Int)
  + Decimal(num:Float)
  + Characters(string:String)
  + Documentation(doc:String)
  + ClassPointer(pointer:MemoryPointer)
  + Sign(sign:String)
  + NullValue
  + TrueValue
  + FalseValue
  + Identifier(word:String)
  + Object(props:Map<String, {documentation:String, value:InterpTokens}>, typeName:String)
  + HaxeExtern(func:Void -> InterpTokens)