# 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  
    
    
    
  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