# 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

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.

על מנת לשרת את המטרות האלה, הפרויקט מעוצב סביב, ונותן גישה לשלושה מערכות שונות, אך קשורות:

* **אוסף מילות מפתח** – על מנת להקל על המֵפתח, ניתנת גישה לכל "מילות המַפתח" (Standard Library, מילים שמורות), והמפתח יכול לשנות מילים ספציפיות, או להחליף בין סטים שלמים של מילים, כדי לשנות את וורבאליות הקוד, או השפה המדוברת בו היא כתובה. כמו שהזכרתי במטרה הראשונה, זה שימושי מאוד כשמנסים להנגיש 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** – כ"השלמה" של 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”) |