**Little**

**שפת תכנות, "קומפיילר" ואינטרפרטר.**

|  |  |
| --- | --- |
| שם תלמיד: שחר מרכוס  תעודת זהות: 329545933  שם מורה: שלמה ווקנין  שמות חלופה: Multilang-Code  Minilang  תאריך הגשה: --- | תמונה שמכילה טקסט, גרפיקה, עיצוב גרפי, גופן  התיאור נוצר באופן אוטומטי |



# תוכן עניינים

[תוכן עניינים 1](#_Toc167731706)

[הערות לקורא 3](#_Toc167731707)

[מבוא 4](#_Toc167731708)

[ייזום 4](#_Toc167731709)

[תיאור ראשוני 4](#_Toc167731710)

[הגדרת הלקוח 4](#_Toc167731711)

[יעדים ומטרות 5](#_Toc167731712)

[בעיות, תועלות וחסכונות 5](#_Toc167731713)

[פתרונות קיימים 7](#_Toc167731714)

[טכנולוגיה 7](#_Toc167731715)

[הגבלות 9](#_Toc167731716)

[תחומים ותמיכה 10](#_Toc167731717)

[תיאור המערכת 11](#_Toc167731718)

[אז, מה בדיוק הפרויקט עושה? 11](#_Toc167731719)

[יכולות 19](#_Toc167731720)

[סוויטת בדיקות 20](#_Toc167731721)

[לוח זמנים 22](#_Toc167731722)

[סיכונים 23](#_Toc167731723)

[תיאור תחום הידע 25](#_Toc167731724)

[יכולות 25](#_Toc167731725)

[סוג לקוח: מפתח אפליקציות 25](#_Toc167731726)

[סוג לקוח: לומד תכנות, מתכנת **Little** 31](#_Toc167731727)

[מבנה הפרויקט 36](#_Toc167731728)

[ארכיטקטורה 36](#_Toc167731729)

[רכיבים שונים, והקשרים ביניהם 36](#_Toc167731730)

[טכנולוגיה 40](#_Toc167731731)

[שפת תכנות 40](#_Toc167731732)

[תחומי עניין 40](#_Toc167731733)

[זרימת מידע במערכת 41](#_Toc167731734)

[הקומפיילר 41](#_Toc167731735)

[האינטרפרטר 42](#_Toc167731736)

[הזיכרון 43](#_Toc167731737)

[אלגוריתמים מרכזיים 44](#_Toc167731738)

[קישור בין משתנים למקומות בזיכרון 44](#_Toc167731739)

[חישוב והערכת ביטויים 45](#_Toc167731740)

[**Hashing** 46](#_Toc167731741)

[סביבת עבודה 48](#_Toc167731742)

[פיתוח 48](#_Toc167731743)

[בדיקה 48](#_Toc167731744)

[מימוש הפרויקט 49](#_Toc167731745)

[חלקים ומחלקות 49](#_Toc167731746)

[חבילה – little.lexer 49](#_Toc167731747)

[חבילה – little.parser 50](#_Toc167731748)

[חבילה – little.interpreter 52](#_Toc167731749)

[חבילה – little.interpreter.memory 56](#_Toc167731750)

[חבילה – little.tools 65](#_Toc167731751)

[מחלקות נוספות 70](#_Toc167731752)

[קוד "יפה" ותכנות אלגוריתמים 73](#_Toc167731753)

[קריאה לפונקציות אקסטרניות 73](#_Toc167731754)

[Shunting Yard – השראה 73](#_Toc167731755)

[עיצוב ה-Referrer 81](#_Toc167731756)

[יצירת סוגים אקסטרניים 83](#_Toc167731757)

[בדיקות ותוצאות 92](#_Toc167731758)

[בדיקה ביחידות (Unit Testing) 92](#_Toc167731759)

[בדיקות ידניות 95](#_Toc167731760)

[מדריך למשתמש 97](#_Toc167731761)

[עץ קבצים 97](#_Toc167731762)

[התקנה 99](#_Toc167731763)

[למפתחי תוכנה 99](#_Toc167731764)

[מתכנתים ב-Little 101](#_Toc167731765)

[רפלקציה 103](#_Toc167731766)

[עבודה על הפרויקט 103](#_Toc167731767)

[תובנות 104](#_Toc167731768)

[הסתכלות 105](#_Toc167731769)

[אחורנית 105](#_Toc167731770)

[וקדימה 105](#_Toc167731771)

[תודות 105](#_Toc167731772)

[ביבליוגרפיה 106](#_Toc167731773)

[נספחים 107](#_Toc167731774)

[הקדמה קצרצרה 107](#_Toc167731775)

[קוד הפרויקט 108](#_Toc167731776)

# הערות לקורא

כמו בהרבה נושאים הקשורים למדעי המחשב, עברית לא תמיד מציעה את כל המילים שצריך...  
הנה מילון מושגים קצר, שאשתמש בהם במהלך הפרויקט:

* + טוקן - Token
  + לקסר - Lexer
  + פרסר - Parser
  + קומפיילר - Compiler
  + אינטרפרטר - Interpreter
  + טרנספיילר - Transpiler
  + VM – מכונה ווירטואלית, Virtual Machine.
  + עץ syntax – AST, Abstract Syntax Tree
* ישנם חלקים בפרויקט המכילים שורות ארוכות במיוחד. מומלץ לקרוא את הספר במצב עמוד אינטרנטי, באמצעות לחיצה על שנמצא בצד ימין/שמאל למטה של חלון הוורד, תלוי בשפה של וורד (בצד הימני לשפות כמו אנגלית, ובצד השמאלי לשפות כמו עברית)

# 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 מסוימים.

### 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 מסוימת, 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:

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

|  |  |
| --- | --- |
| 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” “”” |

* **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 ה-Parser של השפה) מראש. הסיבה לכך היא, שsyntax שנחשב שגוי בשפה, אך חוקי ב-macro 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, וה-macro 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 מביט, 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 גיגבייט.
* **שרת קומפליציה (completion)**  - בחלק אחד מהפרויקט, ישנו סוג שאמור להיות מתאים עם 8 סוגים אחרים. בגלל באג בשרת הקומפליציה, אי אפשר ליצור סוג אבסטרקטי עם יותר מ8 פרמטרי סוג , אם מאופשרת המרה מהם ואליהם.
* **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

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):

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

## תיאור המערכת

### אז, מה בדיוק הפרויקט עושה?

בגדול **מאוד**, הפרויקט 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 הניסיון 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:

* + [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 x = 5

define y = new ImprovedNumber(5)

y.increment(4)

print(y)

action increment(x:Number) = {

    return x + 1

}

//Instances

//File name – Improved Number

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)

}

* + [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:

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

    }

* + [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)

}

* + 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 :)

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

* + 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 להיפרד 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
    - אזכור של תכונות ייחודיות בReadme של הפרויקט
  + [ב16 במרץ 2023](https://github.com/ShaharMS/Little/commit/029cbc27218b5d3fa167e393d3e2b519e5e7478e) הוגשמה המטרה השלישית, ויצרתי חלק בקובץ הReadme שנועד לאזכור 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

* + [ב19 באפריל, 2023](https://github.com/ShaharMS/Little/commit/d8ee3e0341f8d4dcb94b08e7df063456fd20017d) נוספה לReadme התכונה השנייה, שעקרונה: Consistency Is Key. גם עליה אפשר בהמשך, על אף שהיא יחסית ברורה מדוגמת הקוד שכתבתי לReadme:

define consistent = 5

define consistent.newPropertyDeclaration = 6

action declaredJustLikeVariables(define parametersAreDefinedTheSame = 6) = {

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

}

* + [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 לזה, 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})

* + 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:

sign (left ^% right) = {

    return left + right ^ 2

}

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

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 מסוימת 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
    - **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:

define בחזקת = ^

define כפול = \*

print(5 כפול 6 בחזקת 2)

(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.
    - **גישות וקריאות:** אפשר לגשת לערכים של מזהים בעזרת השתמשות בשמם, השתמשות בפונקציה 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 ,

define x = 3

define e = read(“x”)

print({e +=7, read(Characters.fromCharCode(e + 91))})

(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:

define x = 3

after (x == 4) print(“hey”)

whenever (read(“x”) > 3) {

    print(x)

}

כש-x יוגדל מעבר ל-3, יודפס ערכו. בפעם הראשונה ש-x יהיה שווה ל-4 תודפס המילה hey.

* + - **גופי קוד:** כמו שבטח ראיתם מקודם, אפשר הרבה פעמים להשמיט את הסוגריים המסולסלות במקום שמצופה בלוק של קוד. המקרה היחיד שאי אפשר להשמיט אותן הוא כמשתמשים 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----  
    
  **‎**

|  |  |  |
| --- | --- | --- |
| 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 המילים 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" \l "keyword--standard-library-modification) יש דוגמה לשינויי מילים. אביא את הדוגמאות משם:
  + **Parser Injection:** הParser בנוי בצורה שמאפשרת "השחלה" של פונקציות parsing תוך כדי תהליך הparser הרגיל, ואף מאפשר הסרה של שלבים, על מנת לתת למפתחים שמשתמשים בפרויקט 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 בפרויקט (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 הפרויקט 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 בבניית 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)

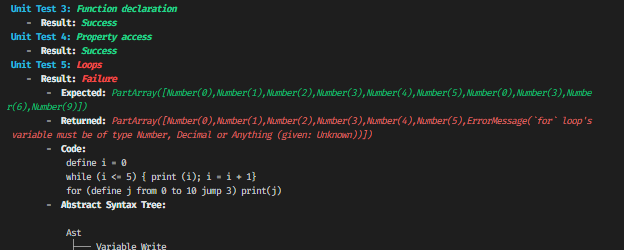
* **Pretty Printer:** באמצעות המחלקה PrettyPrinter, ניתן להדפיס עצים יפים של מערכי Enums.

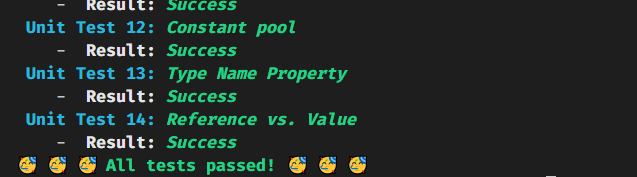
### סוויטת בדיקות

על מנת לוודא שהפרויקט 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 הבדיקות, 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 ופירמוט, 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:  
  
  
להודעה הזאת יש שתי צורות נוספות:



תמונה שמכילה טקסט, גופן, צילום מסך, גרפיקה

התיאור נוצר באופן אוטומטי

כיום, מורצות 15 בדיקות, כל אחת למטרה שונה, אך יכולה להסתמך על דברים אחרים   
שצריכים לעבוד. בדיקות מסודרות לפי שם פונקציה (test<test Number>), 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 זיכרון 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 בזיכרון, וההפך לערכים שהם pass by reference.
15. Arrays – יוצר, מאחסן ומאחזר מידע ממערכים.

### לוח זמנים

כשהתחלתי עם הפרויקט, לא בדיוק היה לי לוח זמנים, אלא רצון לסיים אותו לאחר כמות מסוימת 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 שהפרויקט הציע כשהיה צריך. אארגן לטבלה:

|  |  |  |  |
| --- | --- | --- | --- |
| **אבן דרך** | **תאריך מצופה** | **תאריך בפועל** | |
| תחילת הפרויקט | 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 הפרויקט 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)**:** פלגין לשרת במשחק Minecraft, שמטרתו להפוך אביזר "בלוק" הנקרא Monster Spawner לאביזר שאפשר להשיג בצורה אין-סופית באופן תיאורטי. כשפיתחתי את זה, עוד לא היה כלי לפיתוח פלגינים 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 זיכרון:** 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 זיכרון:** Haxe היא לא שפה שיכולה ישר לבקש ולפנות זיכרון 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 שפות תכנות 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 מהזיכרון**
  + אם מפתח צריך, הוא יכול להשתמש במגוון הפונקציות שבמחלקה Memory על מנת לקרוא ולערוך ערכים בזיכרון, בכל זמן שהוא.
  + שדות ומחלקות נחוצים:
    - Memory
      * store(token:InterpTokens):MemoryPointer
      * retrieve(token:InterpTokens):MemoryPointer
      * read(path:Rest<String>):{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}
      * write(path:Array<String>, ?value:InterpTokens, ?type:)
      * set(path:Array<String>, ?value:InterpTokens, ?type:String, ?doc:String)
      * allocate(size:Int):MemoryPointer
      * free(pointer:MemoryPointer, size:Int)
* **יכולת: גישה מהירה לזיכרון 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:
    - Storage – בשביל סוגים: (Byte, Bytes, Array, Int16, UInt16, Int32, UInt32, Float, Pointer, Function, Condition, Sign, Object, Type):
      * פונקציות store(…)
      * פונקציות set(address:MemoryPointer, …)
      * פונקציות read(address:MemoryPointer, …)
      * פונקציות free(address:MemoryPointer, …)
      * storage:ByteArray
      * reserved:ByteArray
    - Referrer
      * bytes:ByteArray
      * currentScopeStart:Int
      * currentScopeLength:Int
      * get(key:String):{address:MemoryPointer, type:String}
      * set(key:String, value:{?address:MemoryPointer, ?type:String})
      * exists(key:String):Bool
* **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 הפונקציה change של אותה מחלקה (KeywordConfig).
  + שדות ומחלקות נחוצים:
    - KeywordConfig
    - change(config:KeywordConfig)
    - (יש בין 50 ל100 מילות מפתח וסימנים שאפשר לשנות. לא אשים אותן כאן, קישור: <https://github.com/ShaharMS/Little/blob/branch/functional-programming/src/little/KeywordConfig.hx#L77>)
* **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 הטרמינל (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 הפרויקט 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.
  + printSample! – מדפיס קוד הכתוב לפי הספציפיקציה הרשמית, קיים למטרות למידה.

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.
  + "שדות ומחלקות" נחוצים:
    - Hxcpp (ספריה וכלי בנייה לC++)
    - Hashlink (ספריה, כלי בניה ומכונה וירטואלית הכתובה ובונה ל-C)
    - Haxe Interpreter (חלק מHaxeC, מריץ קוד ללא טרנספילציה)
    - HaxeC (אורז בתוכו טרנספיילר JS)
* **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.
  + שדות ומחלקות נחוצים:
    - JsExample
      * input:TextAreaElement
      * ast:TextAreaElement
      * output:TextAreaElement
      * input.addEventListener(“keyup”)
    - Little
      * run(code:String, debug:Bool)
      * reset()
* **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)
  + מחלקות ושדות נחוצים:
    - Parser
      * mergeComplexStructures(pre:Array<ParserTokens>):Array<ParserTokens>
    - Interpreter
      * declareVariable(name:InterpTokens, type:InterpTokens, doc:InterpTokens)
      * declareFunctiondeclareFunction(name:InterpTokens, params:InterpTokens, doc:InterpTokens)
* **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.
  + שדה נחוץ:
    - Interpreter
      * typecast(value:InterpTokens, type:InterpTokens):InterpTokens
* **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.
  + שדות ומחלקות נחוצים:
    - JsExample
      * ast:TextAreaElement
    - Main
      * main()
* **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)
  + שדות ומחלקות נחוצים:
    - Runtime
      * callstack:Array<{module:String, line:Int, linePart:Int, token:InterpTokens}>
      * throwError(error:InterpTokens, ?layer:Layer):InterpTokens
* **יכולת: גישה לא בטוחה לזיכרון**
  + כברירת מחדל, יש למשתמשים בשפה גישה למחלקה ששמה Memory, המספקת מספר פעולות ומשתנים העוזרים עם גישה והתעסקות "לא בטוחה" עם הזיכרון
  + שדות ומחלקות נחוצים:
    - Plugins
      * registerType(typeName:String, fields:TypeFields)
    - ExternalInterfacing.ExtTree
* **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.
  + שדה נחוץ:
    - Plugins
      * registerInstanceVariable(propertyName:String, propertyType:String, onType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:(objectValue:InterpTokens, objectAddress:MemoryPointer) -> InterpTokens)
* **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)
  + שדה נחוץ:
    - Plugins
      * registerInstanceVariable(propertyName:String, propertyType:String, onType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:(objectValue:InterpTokens, objectAddress:MemoryPointer) -> InterpTokens)
* **יכולת: שינוי מילים שמורות**
  + זמין בלקוח האינטרנטי – המשתמש יכול בעצמו לשנות את המילים השמורות לשפה אחרת, סגנון אחר, או סתם לקצר.
  + שדות נחוצים:
    - JsExample
      * keywordTable:TableElement
      * new()#update()
      * getCodeExample(keyword:String):String
* **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
  + שדות נחוצים:
    - Runtime
      * print(item:String)
      * warn(token:InterpTokens, ?layer:Layer = INTERPRETER)
      * throwError(token:InterpTokens, ?layer:Layer = INTERPRETER):InterpTokens
      * broadcast(item:String)
      * \_\_broadcast(item:String)
      * \_\_print(item:String, representativeToken:InterpTokens)
* **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".
  + שדה נחוץ:
    - Plugins
      * registerFunction(functionName:String, ?documentation:String, expectedParameters:EitherType<String, Array<InterpTokens>>, callback:Array<{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}> -> InterpTokens, returnType:String)
    - ExternalInterfacing.ExtTree
* **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)
  + שדות נחוצים:
    - Interpreter
      * declareVariable(name:InterpTokens, type:InterpTokens, doc:InterpTokens)
      * declareFunctiondeclareFunction(name:InterpTokens, params:InterpTokens, doc:InterpTokens)

# 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 הווירטואלית. בזמן שהספציפיקציה כמעט ולא נמצאת בתוך הקו, שני החלקים האחרים של הפרויקט מאוד מופרדים, ונמצאים בתיקיות 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, ואופרטור. כאן נמצא השימוש הראשון בכלים הגלובליים, על מנת לזהות מהו אופרטור 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 "לייצא" אותו, ולהשתמש בו:
    - אינטרפרטר:
  + PrepareRun – לפני הרצה, מוסיף את כל המשתנים, הפונקציות, האופרטורים, הסוגים והשדות ומייצר את הספרייה 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.convert – לוקח את העץ שקיבלנו מהקומפיילר, וממיר אותו לטוקנים הנוחים לשימוש באינטרפרטר.
  + Interpreter.run – לאחר מכן, אנו לוקחים את כל העץ, ומריצים אותו בעזרת פונקציית ההרצה העיקרית. הפונקציה משתמשת במגוון הפעולות שעל המחלקה Interpreter על מנת לממש את ההרצה, קוראת לאירועים 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 זיכרון בעזרת 5 מחלקות אחרות:
        + Storage – מערך זיכרון ענק, מספק פונקציות של כתיבה, עריכה ובקשת הרחבה של אותו מערך זיכרון, לפי מצביעים המעוצבים כדי לגשת לאותו מערך
        + Referrer – גם מערך זיכרון, 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 בזיכרון. לוקח זיכרון מ-Storage.
        + HashTables – מיצר, קורא וכותב מפות hash בעזרת האלגוריתם MurmurHash1. בעזרתו אפשר לגשת לאובייקטים.
      * Runtime – מכיל אירועים שונים, שאמורים להיקרא כל פעם שדבר מסוים 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.

להלן, הקשרים בין האלמנטים ברשימה כגרף (בתצוגה רגילה, הגרף בעמוד הבא):

תמונה שמכילה צילום מסך, טקסט, תרשים, עיצוב

התיאור נוצר באופן אוטומטי

ותרשים יותר מפורטת, מסוג גרף UML. כולל את כל הרכיבים בעל תפקיד משמעותי:



## 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 רישיון 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 ייחודיות, 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 מסוימת, לפורמט 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: זיכרון, 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

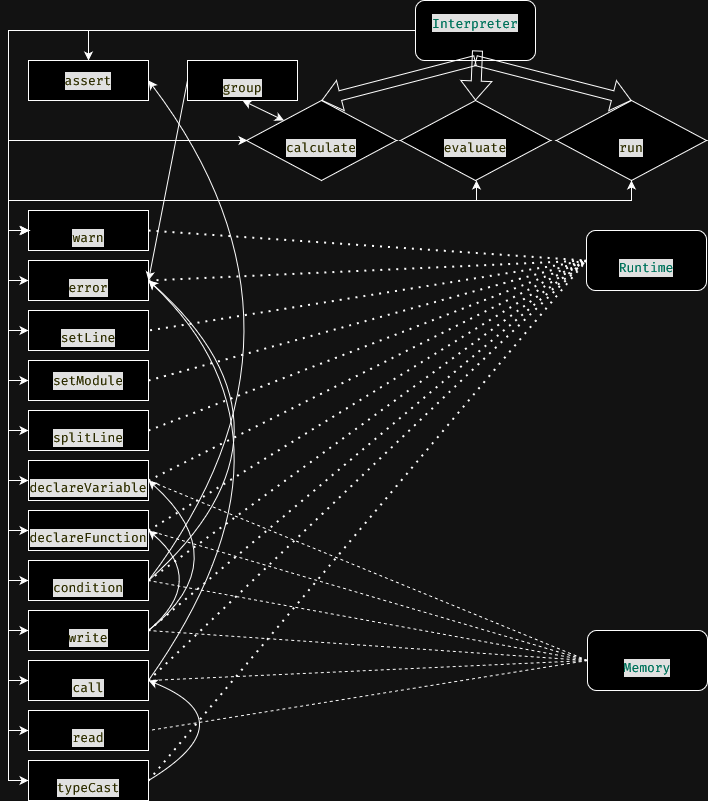
השלב הראשון ב"מימוש" הקוד. מכיל מספר יחסית קטן של רכיבים יחסית גדולים:

תמונה שמכילה צילום מסך, טקסט, תרשים, עיצוב

התיאור נוצר באופן אוטומטי

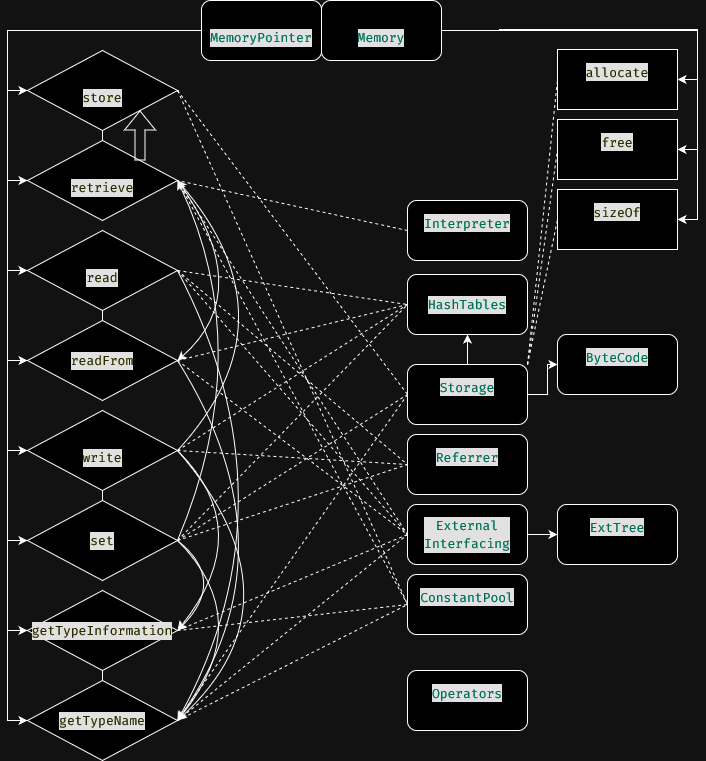
### האינטרפרטר

שלב המימוש עצמו. חצי מהמערכת ה"חיה":



### הזיכרון

אמנם נראה מבחוץ קטן יותר, אבל מכיל יותר חלקים זזים, אפילו מהפרסר והאינטרפרטר ביחד!



## 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 זיכרון.

### 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 אופרטור/פונקציה, יש כמה אופציות:
    - אם האופרטור בדרגה גבוהה יותר מהאופרטור שבראש הסטאק, הוא נוסף לסטאק
    - אם האופרטורים באותה דרגה/האופרטור החדש בדרגה נמוכה יותר, האופרטור הקיים נוסף לפלט ומורד מהסטאק, במקומו מוסיפים את האופרטור 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 ואופרטור אחד. לא נוצר סטאק לאופרטורים, במקום זה, אנו עוברים על האופרטורים בקבוצות, מהחשובים ביותר להכי פחות חשובים, וכך יוצא שטוקנים בעלי אופרטור 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, אופרטור 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 אופרטור, 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 אופרטור ו"ערך עד עכשיו" לא מוגדר, ערכו הופך לערך הזה.
      * אם אין אופרטור ויש "ערך עד עכשיו" נזרקת שגיאה ששני מספרים באו אחד אחרי השני
    - אם הוא אופרטור מציבים אותו ב"אופרטור נוכחי".
      * אם האופרטור הוא הטוקן האחרון, מחשבים את הביטוי כאשר "ערך עד עכשיו" בהתחלה, והאופרטור 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**

כשתכנתי 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 אלטרנטיבי 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 מספר 32 בתים
  + MurMur2 – הגרסה השנייה, בעל המון גרסאות אלטרנטיביות. פשוט גרסה יותר טובה של MurMur1 עם יותר אופציות
  + MurMur3 – הגרסה השלישית והעדכנית. שיפור על הגרסה השנייה, מציע גם ייצור hashים באורך 128 בתים.

מקורות:

* + [https://en.wikipedia.org/wiki/MurmurHash](https://en.wikipedia.org/wiki/SipHash)

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 עד 5.0.0 (כרגע נמצא בnightly בלבד)

vision – מגיטהאב, בטא של גרסה 2.0.0 (לפחות)

hash – גרסה 1.0.1

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

Haxelib install <library name>

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

Haxelib git <library name> <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 השנייה, 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 (הזיכרון). 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).

ספירה אבסטרקטית – הספירה הקלאסית, ללא פיצ'רים נוספים. כל אלמנט מיוצג ע"י ערך מסוג אחר. מתקמפל בסוף לסוג שהוא בנוי עליו, באותה התנהגות כמו אבסטרקט רגיל.

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/אבסטרקט/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 אחר 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 קומבינציית סימנים היא נכונה, ומתי צריך לפצל אותה. זה הכרחי, מכיוון שלא כל האופרטורים הם אות אחת, ולכן צריך לדעת מתי האופרטור הוא בודד ומתי באורך מסוים.

### 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. אותם הטוקנים 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 להיפתר למזהים (טוקנים מסוג Identifier).
  + Identifier(word:String) – מייצג את המזהה. בד"כ, כשנתקלים בו בקוד, יש למשוך את ערכו מהזיכרון 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 להימצא 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 אופרטור 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> – פונקציית 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 מבוטאות 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.
  + ConditionCode(callers:Map<Array<InterpTokens>, InterpTokens>) – מייצג את הערך של תנאי/לולאה.
  + ConditionCall(name:InterpTokens, exp:InterpTokens, body:InterpTokens) – מייצג קריאה לתנאי או לולאה מסוימת עם תנאי הרצה exp ובלוק הקוד body
  + FunctionCode(requiredParams:OrderedMap<String, InterpTokens>, body:InterpTokens) – מייצג את הערך של פונקציה (פרמטרים מחוברים עם בלוק של קוד)
  + FunctionCall(name:InterpTokens, params:InterpTokens) – מייצג קריאה לפונקציה ששמה מבוטא באמצעות name עם הפרמטרים params.
  + FunctionReturn(value:InterpTokens, type:InterpTokens) – מייצג טענת יציאה של פונקציה. הרצת בלוק של קוד מופסקת בעת התקלות בטוקן זה.
  + Write(assignees:Array<InterpTokens>, value:InterpTokens) – מייצג כתיבה של ערך value לN אלמנטים המוכלים בתוך assignees.
  + TypeCast(value:InterpTokens, type:InterpTokens) – מייצג ניסיון יציקה של הערך value אל תוך סוג המבוטא באמצעות type.
  + Expression(parts:Array<InterpTokens>, type:InterpTokens) – מייצג רצף פעולות המקובצות בתוך סוגריים רגילים
  + Block(body:Array<InterpTokens>, type:InterpTokens) – מייצג רצף פקודות, המקובצות בתוך סוגריים מסולסלים.
  + PartArray(parts:Array<InterpTokens>) – דרך לקבוץ מספר אלמנטים אל תוך טוקן יחיד בלי להוסיף metadata לגביו.
  + PropertyAccess(name:InterpTokens, property:InterpTokens) – מייצג גישה לשדה המבוטא בעזרת property שנמצא על ההורה אשר מבוטא בעזרת name.
  + Number(num:Int) – מייצג מספר שלם 32 ביטים בעל סימן.
  + Decimal(num:Float) – מייצג מספר עשרוני 64 ביטים בעל סימן.
  + Characters(string:String) – מייצג מחרוזת.
  + Documentation(doc:String) – מייצג תגובה.
  + ClassPointer(pointer:MemoryPointer) – מייצג את הערך של סוג, בעזרת מצביע אליו.
  + Sign(sign:String) – מייצג את הערך של אופרטור
  + NullValue – ערך ה-null.
  + TrueValue – ערך המייצג אמת
  + FalseValue – ערך המייצג שקר
  + Identifier(word:String) – מקבץ אותיות ללא הקשר מסוים, בד"כ מיצג גישה לאלמנט כלשהו בזיכרון, או השתמשות במילת מפתח.
  + Object(props:Map<String, {documentation:String, value:InterpTokens}>, typeName:String) – מייצג את הערך של אובייקט דינמי מסוג typeName עם השדות props.
  + HaxeExtern(func:Void -> InterpTokens) – מספק דרך להפעיל קוד הכתוב ב-Haxe תוך כדי הפעלת קוד Little.
* מחלקה – Interpreter – מכילה פעולות המסוגלות "לגשר: בין הטוקנים של InterpTokens למה שהם מייצגים ממשית.
  + convert(pre:Rest<little.parser.Tokens.ParserTokens>):Array<InterpTokens> - ממיר טוקנים מסוג ParserTokens לכאלה מסוג InterpTokens, על מנת שקורא הקוד יוכל להפעיל את אותם טוקנים.
  + error(message:String, layer:Layer = INTERPRETER):InterpTokens – זורק שגיאה ומזהה אותה עם שכבה מסוימת, ומחזיר את אותו טוקן שגיאה. ההחזרה לכאורה מיותרת , שכן קורא הקוד מפסיק לעבוד בעת שגיאה, אך היא עדיין נמצאת שם בשביל מפתחים שמעוניינים לבטל את ה"קריסה", ולעבוד עם ערך השגיאה העצמם.
  + warn(message:String, layer:Layer = INTERPRETER):InterpTokens – מעלה שגיאה בצורה של אזהרה, כך שהיא עדיין מודפסת, אך לא עוצרת את התוכנה. מחזירה את טוקן השגיאה.
  + assert(token:InterpTokens, isType:EitherType<InterpTokensSimple, Array<InterpTokensSimple>>, ?errorMessage:String = null) – מוודא שהטוקן הנתון הוא מסוג הטוקנים שיוזכרו בהמשך הפונקציה. אם הוא לא מהסוג שלהם, נזרקת שגיאה ברמה של קוד ה-Little, ומוחזר NullValue. אחרת, הטוקן שהובא מוחזר ללא שינוי.
  + setLine(l:Int) – אומר לקורא הקוד שאנחנו כרגע בשורה l, ומשגר אירועי שינוי שורה ושינוי חלק בשורה.
  + setModule(m:String) - אומר לקורא הקוד שאנחנו כרגע במודולה m, ומשגר אירועי שינוי מודולה אם אכן מודולת ההרצה השתנתה.
  + splitLine() – אומר לקורא הקוד שהחלק שאנו קוראים כרגע בשורה נגמר (מבוטא באמצעות פסיק או נקודה-פסיק). משגר אירוע פיצול שורה.
  + declareVariable(name:InterpTokens, type:InterpTokens, doc:InterpTokens) – כותב לזיכרון משתנה ששמו name, סוגו type וערכו NullValue. משגר אירועי יצירת שדה מסוג משתנה.
  + declareFunction(name:InterpTokens, params:InterpTokens, doc:InterpTokens) – כותב לזיכרון פונקציה ששמה name, וערכה קוד של פונקציה המקבל את הפרמטרים שparams מייצג, ולא עושה כלום איתם. קוד הפונקציה מקובל בעזרת פונקציית write. משגר אירועי יצירת שדה מסוג פונקציה.
  + condition(name:InterpTokens, pattern:InterpTokens, body:InterpTokens):InterpTokens – מריץ את התנאי/לולאה ששמה name, באמצעות תנאי ההרצה pattern, ומחליט כמה פעמים להריץ את בלוק הקוד body. מחזיר את הטוקן שהחזרה האחרונה נגמרה איתו. משגר אירועי קריאה לתנאי/לולאה.
  + write(assignees:Array<InterpTokens>, value:InterpTokens):InterpTokens – כותב את הערך value למשתנים ששמם נמצא בassignees:
    - למשתנים רגילים, כל ערך יעובד ויאוחסן
    - בשביל פונקציות, אם הערך הוא בלוק של קוד, הוא יערוך חלק מערך של הפונקציה, ובכך ישלים את declareFunction.

משגר אירועי כתיבה לערך אם השמות אליהם הוא כתב.

* + call(name:InterpTokens, params:InterpTokens):InterpTokens – קורא לפונקציה ששמה name עם הפרמטרים שבתוך הExpression או PartArray, params. מוסיף entry באופן זמני לLittle.runtime.callStack. משגר אירועי קריאה לפונקציה.
  + read(name:InterpTokens):InterpTokens – מחזיר את הערך של משתנה ששמו name בזיכרון.
  + typeCast(value:InterpTokens, type:InterpTokens):InterpTokens – לפי הערך והסוג שמקבלים, הפונקציה מחפשת פונקציית יציקה מתאימה. אם יש היא מפעילה אותה, אם אין נזרקת שגיאה שפונקציית יציקה לא קיימת. משגר אירועי יציקת ערך לסוג.
  + run(body:Array<InterpTokens>, propagateReturns = false):InterpTokens – מריץ בלוק של קוד בעזרת הפונקציות שהוזכרו לעיל. אם נתקל ב-FunctionReturn, וpropagateReturns הוא אמת, מפסיק הרצה מחזיר את מלוא הטוקן. אחרת, מעריך את טוקן ההחזרה ומחזיר את ההערכה. אם בכלל לא נתקלים בטוקן החזרה, הערך האחרון שעובד מוחזר.
  + evaluate(exp:InterpTokens, ?dontThrow:Bool = false):InterpTokens – מחלץ את הערך המעובד של טוקן יחיד. משתמש במגוון הפונקציות לעיל כשצריך. כש-dontThrow מופעל, שגיאות שמוערכות ע"י הפונקציה לא נזרקות החוצה, אלא רק מוחזרות.
  + calculate(p:Array<InterpTokens>):InterpTokens – מעריך את הערך של קבוצה של טוקנים בעזרת חישוב שלהם והיחס ביניהם, המבוטא בעזרת אופרטורים ומזהים למיניהם. משתמש בפונקציה group על מנת לסדר אותם עם רצף פעולות נכון.
  + group(tokens:Array<InterpTokens>):Array<InterpTokens> - לוקח רצף של טוקנים אשר ביניהם אופרטורים, ומחזיר כמין עץ, המכיל PartArrays, אחד בתוך השני, של עד ל3 אלמנטים, המבטאים את הביטוי המתמטי.
* מחלקה – ByteCode – אחראית על הפיכת קוד בצורת טוקנים, לקוד בתים קומפקטי, שאפשר לאחסן בזיכרון יחסית בקלות
  + compile(...tokens:InterpTokens):String – לוקח קוד בפורמט של עץ syntax, ומחזיר אותו בפורמט קוד בתים, כString.
  + decompile(bytecode:String):Array<InterpTokens> - לוקח קוד-בתים, והופך אותו למערך טוקנים המייצג עץ syntax, שאפשר להריץ.
* מחלקה – StdOut – מייצגת את הפלט של קורא הקוד.
  + output:String – הפלט של ההרצה, כמחרוזת
  + stdoutTokens:Array<InterpTokens> - הטוקנים האינדיבידואלים שניסו להדפיס לoutput, ללא מודפיקציה מעבר לעיבוד.
  + reset() – מאפס את השדות שנאמרו להלן.
* מחלקה – Runtime – מכיל מידע על זמן הריצה, ומידע לאחר הריצה.
  + line(default, null):Int – השורה שאותה אנו מתחילים לקרוא
  + linePart(default, null):Int – החלק בשורה אותו אנו מתחילים לקרוא. שורה מפוצלת בעזרת פסיקים או נקודה-פסיק.
  + currentToken(default, null):InterpTokens – הטוקן אותו אנו הולכים להריץ.
  + module(default, null):String – המודולה בה אנו מריצים קוד כרגע.
  + previousToken(default, null):InterpTokens – הטוקן שהרגע הורץ.
  + exitCode(default, null):Int – טענת היציאה של הקוד. יהיה שונה מ-0 אם התוכנה קרסה.
  + errorThrown(default, null):Bool – אומר האם נזרקה שגיאה או לא
  + errorToken(default, null):InterpTokens – טוקן השגיאה שנזרק.
  + onLineChanged:Array<Int -> Void> - אירוע המשוגר כאשר שורה מתחילה להיקרא
  + onModuleChanged:Array<String -> Void> - אירוע המשוגר כאשר מודולה משתנה. יכול לקרות כאשר מריצים פונקציות ממודולות אחרות.
  + onLineSplit:Array<Void -> Void> - אירוע המשוגר כאשר אנו מתחילים לקרוא חלק משורה. משוגר גם כשמחליפים שורה.
  + onTokenInterpreted:Array<InterpTokens -> Void> - אירוע המשוגר מיד לאחר שאנו מסיימים להריץ טוקן.
  + onErrorThrown:Array<(String, Int, String) -> Void> - אירוע המשוגר מיד לאחר זריקת שגיאה.
  + onWarningPrinted:Array<(String, Int, String) -> Void> - אירוע המשוגר מיד לאחר העלאת אזהרה.
  + onWriteValue:Array<Array<String> -> Void> - אירוע המשוגר מיד לאחר שנכתב ערך לאחד או יותר משתנים/פונקציות/שדות.
  + onFunctionCalled:Array<(String, Array<InterpTokens>) -> Void> - אירוע המשוגר מיד לפני הרצת פונקציה
  + onConditionCalled:Array<(String, Array<InterpTokens>, InterpTokens) -> Void> - אירוע המשוגר מיד לפני קריאה לתנאי/לולאה.
  + onFieldDeclared:Array<(String, FieldDeclarationType) -> Void> - אירוע המשוגר מיד לאחר שמוכרז משתנה או פונקציה.
  + onTypeCast:Array<(InterpTokens, String) -> Void> - אירוע המשוגר מיד לאחר ניסיון יציקת ערך לסוג שאינו שלו.
  + stdout:StdOut – הפלט הסטנדרטי.
  + callStack:Array<{module:String, line:Int, linePart:Int, token:InterpTokens}> - סטאק הקריאות, משומש כאשר נזרקת שגיאה. משתנה לפני ואחרי קריאה לפונקציה.
  + throwError(token:InterpTokens, ?layer:Layer = INTERPRETER):InterpTokens – זורק שגיאה משכבה מסוימת, ועוצר את ההרצה.
  + warn(token:InterpTokens, ?layer:Layer = INTERPRETER) – מדפיס שגיאה משכבה מסוימת, ולא עוצר את ההרצה.
  + print(item:String) – מדפיס מחרוזת כלשהי.
  + broadcast(item:String) – פונקציה שאפשר לערוך, שווה ערך פונקציונאלית לprint. נועד למפתחים.
* ספירה – FieldDeclarartionType – משומש ע"י אירוע ההכרזה.
  + VARIABLE – הכרזת משתנה. זמין באופן ישיר בפרויקט
  + FUNCTION – הכרזת פונקציה. זמין באופן ישיר בפרויקט.
  + CLASS – הכרזת סוג. לא זמין באופן ישיר בפרויקט.
  + CONDITION – הכרזת תנאי/לולאה. לא זמין בגרסה זו של הפרויקט.
  + OPERATOR – הכרזת אופרטור. לא זמין בגרסה זו של הפרויקט.

### חבילה – little.interpreter.memory

אבסטרקט – MemoryPointer – מייצג כתובת בזיכרון, פועל כמספר 32 ביטים לאחר קומפילציה.

* + rawLocation(get, set):Int – המספר שמתחת לMemoryPointer הזה.
  + toString() – ממיר את הכתובת למחרוזת
  + toArray():Array<Int> - ממיר את הכתובת למערך של ביטים, שערכם מ-128 ל127.
  + toBytes():Bytes – ממיר את הכתובת למערך ביטים אמיתי
  + toInt():Int – rawLocation רק כפונקציה.

מחלקה – Memory – נבנה כאבסטרקציה על שאר מבנה הזיכרון – מכיל פונקציות גישה, כתיבה ואחסון ערכים בצורה קלה עם דוקומנטציה ותגובות נרחבות.

* + storage:Storage – האחסון שמשתמש בו הזיכרון.
  + referrer:Referrer – הReferrer שמשתמש בו הזיכרון.
  + externs:ExternalInterfacing – interface האקסטרנים שמשתמש בו הזיכרון.
  + constants:ConstantPool – בריכת הקבועים שמשתמש בו הזיכרון.
  + operator:Operators – עוד interface לאקסטרנים, הפעם מעוצב לאופרטורים.
  + memoryChunkSize:Int = 512 – כל פעם שמבוקשת כמות מסוימת של זיכרון, משתמשים בערך הזה כדי לדעת כמה.
  + maxMemorySize:Int = 1024 \* 1024 \* 2 – כמות הזיכרון המקסימלית שהזיכרון מסוגל לבקש.
  + currentMemorySize(get, never):Int – כמות הזיכרון בשימוש כרגע, בקפיצות של memoryChunkSize.
  + reset() – מאפס או יוצר מחדש את השדות storage, referrer, externs, constants.
  + store(token:InterpTokens):MemoryPointer – מאחסן ערך פשוט בזיכרון, ומחזיר מצביע אליו. לא מסוגל לעבד מזהים, רק ערכים
  + retrieve(token:InterpTokens):MemoryPointer – כמו store, רק גם מסוגל לעקוב אחר מזהים, ועוקב אחרי חוקי pass-by-value/reference.
  + read(...path:String):{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer} – משתמש באכסוני הזיכרון הזמינים כדי להחזיר את הערך, מיקום והסוג של שדה כלשהו הנמצא בסוף path.
  + readFrom(value:{objectValue:InterpTokens, objectAddress:MemoryPointer}, ...path:String) – הרחבה של read, המסוגלת להתחיל קריאה מערך סטטי ולא רק ממזהה. שימושי במקרים כמו 6.toString(), בהם הפונקציה read תנסה לגשת למשתנה ששמו 6 ותכשל.
  + write(path:Array<String>, ?value:InterpTokens, ?type:String, ?doc:String) – יוצר ערך חדש עם סוג ודוקומנטציה מסוימת לשדה הנמצא בסוף path. אם אחד מהפרמטרים הסופיים הוא Null, נותנים ערכי "לא ידוע" לשדה (סוג Unknown/Anything, ערך NullValue)
  + set(path:Array<String>, ?value:InterpTokens, ?type:String, ?doc:String) – עורך שדה קיים הנמצא בסוף path. אם אחד מהפרמטרים הסופיים null, לא עורכים את החלק של הפרמטר הזה בשדה. אם השדה לא קיים, נזרקת שגיאה ברמת קוד Little.
  + allocate(size:Int):MemoryPointer – מבקש שלמור על כמות מסוימת של ביטים, ומחזיר את הכתובת של ההתחלה שלהם.
  + free(pointer:MemoryPointer, size:Int) – מפנה כמות מסוימת של ביטים השוכנים בכתובת מסוימת.
  + sizeOf(pointer:MemoryPointer, type:String):Null<Int> - מגלה את הגודל של ערך מסוג מסוים הנמצא במקום מסוים.
  + getTypeInformation(name:String):TypeInfo – מחלץ מידע על סוגים, בים אם הם מאוחסנים בזיכרון אקסטרני או רגיל. המידע כולל שם, שדות שונים, גודל אובייקט מסוגם, וmetadata לגבי הסוג.
  + getTypeName(pointer:MemoryPointer):String – משתמש בstorage ו-externs על מנת לחלץ שם של סוג בעזרת כתובתו בלבד.

מחלקה - Storage – אחראית על הכנסת ערכים למערך הזיכרון, ווידוא שלא נכתב שום דבר מעליהם כשלא צפוי שזה יקרה

* + storage:ByteArray – מערך הבתים שמשתמשת בו המחלקה
  + parent:Memory – backreference לMemory.
  + reserved:ByteArray – מערך בתים המייצג את המקומות הפנויים והתפוסים.
  + requestMemory() – מבקש עוד memoryChunkSize ביטים מהמערכת.
  + storeByte(b:Int):MemoryPointer – מאחסן ביט, ומחזיר את הכתובת שלו
  + setByte(address:MemoryPointer, b:Int) – מכניס ביט בכתובת מסוימת.
  + readByte(address:MemoryPointer):Int – קורא ביט מכתובת מסוימת, ומחזיר את ערכו
  + freeByte(address:MemoryPointer) – מפנה ביט הנמצא בכתובת מסוימת
  + storeBytes(size:Int, ?b:ByteArray):MemoryPointer – מאחסן מערך ביטים, ומחזיר את הכתובת שלהם
  + setBytes(address:MemoryPointer, bytes:ByteArray) – מכניס מערך בתים בכתובת מסוימת.
  + readBytes(address:MemoryPointer, size:Int):ByteArray – קורא מערך בתים מכתובת מסוימת, ומחזיר אותו
  + freeBytes(address:MemoryPointer, size:Int) – מפנה מערך בתים הנמצא בכתובת מסוימת
  + storeArray(length:Int, elementSize:Int, ?defaultElement:ByteArray):MemoryPointer – מאחסן מערך, ומחזיר את הכתובת שלו. מערך מאוחסן כאורך מערך, גודל אלמנט, ומיד לאחר מכן תוכן המערך.
  + setArray(address:MemoryPointer, length:Int, elementSize:Int, ?defaultElement:ByteArray) – מכניס מערך בכתובת מסוימת.
  + readArray(address:MemoryPointer):Array<ByteArray> – קורא מערך מכתובת מסוימת, ומחזיר אותו
  + freeArray(address:MemoryPointer) – מפנה מערך הנמצא בכתובת מסוימת
  + storeInt16(b:Int):MemoryPointer – מאחסן מספר שלם 16 בתים בעל סימן, ומחזיר את הכתובת שלו
  + setInt16(address:MemoryPointer, b:Int) – מכניס את הערך של מספר שלם 16 בתים בעל סימן בכתובת מסוימת.
  + readInt16(address:MemoryPointer):Int – קורא מספר שלם 16 בתים בעל סימן מכתובת מסוימת, ומחזיר את ערכו
  + freeInt16(address:MemoryPointer) – מפנה מספר שלם 16 בתים בעל סימן הנמצא בכתובת מסוימת
  + storeUInt16(b:Int):MemoryPointer – מאחסן מספר שלם 16 בתים ללא סימן, ומחזיר את הכתובת שלו
  + setUInt16(address:MemoryPointer, b:Int) – מכניס את הערך של מספר שלם 16 בתים ללא סימן בכתובת מסוימת.
  + readUInt16(address:MemoryPointer) – קורא מספר שלם 16 בתים ללא סימן מכתובת מסוימת, ומחזיר את ערכו
  + freeUInt16(address:MemoryPointer) – מפנה מספר שלם 16 בתים ללא סימן הנמצא בכתובת מסוימת
  + storeInt32(b:Int):MemoryPointer – מאחסן מספר שלם 32 בתים בעל סימן, ומחזיר את הכתובת שלו
  + setInt32(address:MemoryPointer, b:Int) – מכניס את הערך של מספר שלם 32 בתים בעל סימן בכתובת מסוימת.
  + readInt32(address:MemoryPointer):Int – קורא מספר שלם 32 בתים בעל סימן מכתובת מסוימת, ומחזיר את ערכו
  + freeInt32(address:MemoryPointer) – מפנה מספר שלם 32 בתים בעל סימן הנמצא בכתובת מסוימת
  + storeUInt32(b:UInt):MemoryPointer – מאחסן מספר שלם 32 בתים ללא סימן, ומחזיר את הכתובת שלו
  + setUInt32(address:MemoryPointer, b:UInt) – מכניס את הערך של מספר שלם 32 בתים ללא סימן בכתובת מסוימת.
  + readUInt32(address:MemoryPointer):UInt – קורא מספר שלם 32 בתים ללא סימן מכתובת מסוימת, ומחזיר את ערכו
  + freeUInt32(address:MemoryPointer) – מפנה מספר שלם 32 בתים ללא סימן הנמצא בכתובת מסוימת
  + storeDouble(b:Float):MemoryPointer – מאחסן מספר עשרוני 64 בתים בעל סימן, ומחזיר את הכתובת שלו
  + setDouble(address:MemoryPointer, b:Float) – מכניס את הערך של מספר עשרוני 64 בתים בעל סימן בכתובת מסוימת.
  + readDouble(address:MemoryPointer):Float – קורא מספר עשרוני 64 בתים בעל סימן מכתובת מסוימת, ומחזיר את ערכו
  + freeDouble(address:MemoryPointer) – מפנה מספר עשרוני 64 בתים בעל סימן הנמצא בכתובת מסוימת
  + storePointer(p:MemoryPointer):MemoryPointer – מאחסן כתובת בזיכרון, ומחזיר את הכתובת שלה
  + setPointer(address:MemoryPointer, p:MemoryPointer) – מכניס את הערך של כתובת בזיכרון בכתובת מסוימת.
  + readPointer(address:MemoryPointer):MemoryPointer – קורא כתובת בזיכרון מכתובת מסוימת, ומחזיר את ערכה
  + freePointer(address:MemoryPointer) – מפנה כתובת בזיכרון הנמצאת בכתובת מסוימת
  + storeString(b:String):MemoryPointer - מאחסן מחרוזת, ומחזיר את הכתובת שלה. מאוחסן כאורך ומיד לאחריו תוכן המחרוזת.
  + setString(address:MemoryPointer, b:String) – מכניס את הערך של מחרוזת בכתובת מסוימת.
  + readString(address:MemoryPointer):String – קורא מחרוזת מכתובת מסוימת, ומחזיר את ערכוה
  + freeString(address:MemoryPointer) – מפנה מחרוזת הנמצאת בכתובת מסוימת
  + storeCodeBlock(caller:InterpTokens):MemoryPointer – מאחסן ערך של פונקציה, ומחזיר את הכתובת שלו. מאוחסן כקוד-בתים.
  + setCodeBlock(address:MemoryPointer, caller:InterpTokens) – מכניס את הערך של פונקציה בכתובת מסוימת.
  + readCodeBlock(address:MemoryPointer):InterpTokens – קורא פונקציה מכתובת מסוימת, ומחזיר את ערכה
  + freeCodeBlock(address:MemoryPointer) – מפנה פונקציה הנמצאת בכתובת מסוימת
  + storeCondition(caller:InterpTokens):MemoryPointer – מאחסן תנאי/לולאה בזיכרון, ומחזיר את הכתובת שלו. מאוחסן כקוד-בתים.
  + setCondition(address:MemoryPointer, caller:InterpTokens) – מכניס את הערך של תנאי/לולאה בכתובת מסוימת.
  + readCondition(address:MemoryPointer):InterpTokens – קורא תנאי/לולאה מכתובת מסוימת, ומחזיר את ערכה
  + freeCondition(address:MemoryPointer) – מפנה תנאי/לולאה הנמצאת בכתובת מסוימת
  + storeSign(sign:String) – מאחסן אופרטור, ומחזיר את הכתובת שלו
  + setSign(address:MemoryPointer, sign:String) – מכניס אופרטור בכתובת מסוימת.
  + readSign(address:MemoryPointer):InterpTokens – קורא אופרטור מכתובת מסוימת, ומחזיר את ערכו
  + freeSign(address:MemoryPointer) – מפנה אופרטור הנמצא בכתובת מסוימת
  + storeStatic(token:InterpTokens):MemoryPointer – מאחסן אלמנט בעל גודל סטטי/מחרוזת, ומחזיר את הכתובת שבה הערך אוחסן.
  + storeObject(object:InterpTokens):MemoryPointer – מאחסן אובייקט דינמי, ומחזיר את הכתובת שלו. אובייקט כזה מאוחסן כ8 ביטים של אורך ומיקום, ומפת Hash נוספת במקום אחר.
  + setObject(address:MemoryPointer, object:InterpTokens) – מכניס אובייקט דינמי בכתובת מסוימת.
  + readObject(pointer:MemoryPointer):InterpTokens – קורא אובייקט דינמי מכתובת מסוימת, ומחזיר את ערכו
  + freeObject(pointer:MemoryPointer) – מפנה אובייקט דינמי הנמצא בכתובת מסוימת
  + storeType(name:String, statics:Map<String, {value:InterpTokens, documentation:String, type:String}>, instances:Map<String, {documentation:String, type:String}>) – מאחסן סוג, ומחזיר את הכתובת שלו. סוג מאוחסן כמו 2 אובייקטים ברצף – שני מפות hash לשדות תלויי מקרה ושדות סטטיים, שאליהם שני מצביעים ושני אורכים.
  + setType(address:MemoryPointer, name:String, statics:Map<String, {value:InterpTokens, documentation:String, type:String}>, instances:Map<String, {documentation:String, type:String}>) – מכניס סוג בכתובת מסוימת.
  + readType(pointer:MemoryPointer):TypeInfo – קורא סוג מכתובת מסוימת, ומחזיר את ערכו
  + freeType(pointer:MemoryPointer) – מפנה סוג הנמצא בכתובת מסוימת
* מחלקה – Referrer – עוקבת ומצמידה משתנים לערכם וסוגם באמצעות מבנה שילובי של סטאק ומפת hash.
  + parent:Memory – backreference ל-Memory
  + bytes:ByteArray – מערך הבתים שבו המחלקה משתמשת.
  + currentScopeStart(get, null):Int – מספר המייצג את המקום שבו המסגרת העכשווית מתחיל. לא ניתן לעריכה ישירות, יש לערוך את תחילת מערך הביטים על מנת לשנות.
  + currentScopeLength(get, null):Int – מספר המייצג את כמות המפתחות הנמצאות במסגרת העכשווית.
  + pushScope() – מוסיף header המכיל את כמות האלמנטים במסגרת הקודמת, וכמות האלמנטים במסגרת הזו (0) בתור שני מספרי 16 בתים. מעדכן את currentScopeStart.
  + popScope() – לא עורך שום מידע מחוץ לcurrentScopeStart – מחשב לפי הנתונים הזמינים את תחילת המסגרת הלפני-אחרונה, ועורך את currentScopeStart לערך הזה.
  + reference(key:String, address:MemoryPointer, type:String) – מוסיף את המפתח key ומצמיד אותו לכתובת וסוג מסוים במסגרת הנוכחית.
  + dereference(key:String) – מוחק את המפתח key מהמסגרת הנוכחית.
  + get(key:String):{address:MemoryPointer, type:String} - מאחזר את הכתובת והסוג המוצמדים למפתח מסוים. אם המפתח לא נמצא במסגרת העכשווית, צוללים למסגרת עמוקה יותר. אם המפתח לא נמצא, נזרקת שגיאה ברמת קוד הHaxe ולא Little (מפסיק הרצה של התוכנה לגמרי)
  + set(key:String, value:{?address:MemoryPointer, ?type:String}) – עורך את הכתובת או סוג של מפתח מסוים. אם המפתח לא נמצא מסגרת הנוכחית, צוללים למסגרת עמוקה יותר. אם המפתח לא נמצא, נזרקת שגיאה ברמת קוד הHaxe ולא Little (מפסיק הרצה של התוכנה לגמרי)
  + exists(key:String):Bool – על מנת למנוע את השגיאות שהוזכרו לעיל, אפשר להשתמש בפונקציה זו כדי לבדוק האם מוצמד ערך כלשהו למפתח מסוים במסגרת הנוכחית, או באחת עמוקה יותר.
  + keyValueIterator():KeyValueIterator<String, {address:MemoryPointer, type:String}> - מחזיר איטרטור העובר על כל המפתחות וערכיהן, מתחילת המסגרת המקורית לסופה, ואז למסגרת הבאה, עד שנגמרות המסגרות.
* מחלקה – HashTables – אחראי על יצירת, קריאת ועריכת מפות hash לאחסון אובייקטים בזיכרון
  + generateObjectHashTable(pairs:Array<{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer}>) – מקבל מערך של חמישיות המייצגות מפתחות וערכים במפת hash. לפי כמות המפתחות, יוצר מפה בגודל מסוים. מחזיר את המפה כמערך ביטים.
  + readObjectHashTable(bytes:ByteArray, ?storage:Storage):Array<{key:Null<String>, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer}> - מעבד מערך ביטים אל תוך מערך חמישיות המייצג את מפת ה-hash של האובייקט. כשמסופק Instance של Storage, משתמשים בו על מנת לקרוא את המפתחות המיוצגים במפה בעזרת keyPointer.
  + hashTableHasKey(hashTable:ByteArray, key:String, storage:Storage):Bool – בודק האם מפת hash מכילה מפתח מסוים. מכיוון שכל מפה מכילה את המפתח באמצעות מצביע, אנו צריכים גם את הStorage – שנותן את המסוגלות לקשר בין הכתובת למחרוזת עצמה
  + hashTableGetKey(hashTable:ByteArray, key:String, storage:Storage):{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer} – בשביל מפת hash מסוימת שהמצביעים למפתחותיה "מתייחסים" לStorage מסוים, מסוגל לאחזר מפתח מסוים, ע"י הוצאת hash שלו, וגישה למפה, או חיפוש לפי הצורך
  + objectAddKey(object:MemoryPointer, key:String, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer, storage:Storage) – מוסיף entry למפת hash מסוימת המאוחסנת בStorage מסוים, הקשורה לאובייקט מסוים. אם המפה מלאה ביותר מ70%, המפה מאוחסנת מחדש בגודל גדול יותר, והמפה הקודמת מפונה. גדול המפה החדשה יהיה פי 3 מהכמות שזקוקים אליה בשביל מפה מלאה (בשביל 10 מפתחות הלוקחים כל אחד 4 מצביעים (16 ביטים בפרויקט הזה), יבוקשו 480 ביטים)
  + objectSetKey(object:MemoryPointer, key:String, pair:{?value:MemoryPointer, ?type:MemoryPointer, ?doc:MemoryPointer}, storage:Storage) – עורך מפתח שכבר קיים במפת hash של אובייקט מסוים, ונותן לו ערך אחר, לפי הערכים של האובייקט pair, שכן ערך שבאובייקט pair שהוא null לא נערך. אם המפתח לא נמצא, נזרקת שגיאה ברמת קוד Haxe.
  + objectGetKey(object:MemoryPointer, key:String, storage:Storage):{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer} – פונקציית עזר המאחדת גישה למפת הhash של אובייקט, וקבלת מפתח באמצעות hashTableGetKey.
  + getHashTableOf(objectPointer:MemoryPointer, storage:Storage):ByteArray – פונקציית עזר המקצרת את אחזור מפת הhash של אובייקט לשורה אחת, במקום 3 שורות (לקרוא את אורך המפה ב4 ביטים ראשונים, מיקום המפה בזיכרון ב4 האחרים, וקריאת הביטים לפי האורך והמיקום.)
* מחלקה – ExternalInterfacing – מגשר בין קריאת פונקציות הכתובות ב-Haxe לערכים של Little, באמצעות "אחסון" אותם פונקציות כעצים.
  + parent:Memory – backreference ל-Memory
  + externToPointer:Map<String, MemoryPointer> - מפה המשמשת לקישור סוגים אקסטרניים למיקום בזיכרון. בד"כ משומש עם אחסון ביט יחיד: storage.storeByte()
  + pointerToExtern(get ,null):Map<MemoryPointer, String> - ההפך מהמפה לעיל. איטי יותר, כי צריך לייצר את המפה כל פעם שרוצים לגשת לזה.
  + instanceProperties:ExtTree = new ExtTree(0, null, null, 0) – עץ הפונקציות הקשורות למשתנים תלויי מקרה. (לדוגמה, “”.toLowerCase())
  + globalProperties:ExtTree = new ExtTree(0, null, null, 0) – עץ הפונקציות הקשורות ישירות לאובייקטים וסוגים. (לדוגמה, Characters.fromCharCode(72))
  + createPathFor(extType:ExtTree, ...path:String):ExtTree – יוצר את המסלול/עוקב אחריו במידת הצורך (תלוי אם המסלול קיים או לא) בעץ הנתון (instanceProperties או globalProperties), ומחזיר את האיבר שבסוף המסלול.
  + createAllPathsFor(...path:String) – יוצר במידת הצורך את המסלול בשני העצים, instanceProperties ו-globalProperties.
  + hasGlobal(...path:String):Bool – בודק האם קיים המסלול path בעץ globalProperties.
  + hasInstance(...path:String):Bool – בודק האם קיים המסלול path בעץ instanceProperties.
  + getGlobal(...path:String):{objectValue:InterpTokens, objectAddress:MemoryPointer} – על אף שמאחזר הערך שבכל איבר בעץ זקוק לשני פרמטרים, ערך אובייקט ומיקום בזיכרון, מכיוון שמדובר באובייקט סטטי, אין שימוש בשני הפרמטרים האלה – לכן, המאחזר נקרא עם null לערך האובייקט ו-1 למיקומו, והערך מוחזר.
* מחלקה – ExtTree – העץ עצמו, אחראי על המבנה של הערכים האקסטרניים.
  + getter:(objectValue:InterpTokens, objectAddress:MemoryPointer) -> {objectValue:InterpTokens, objectAddress:MemoryPointer} – מאחזר הערכים – מקבל את ערך האובייקט ההורה ואת כתובת ההורה, ולפיהם מחזיר ערך כלשהו וכתובתו בזיכרון.
  + doc:MemoryPointer – הכתובת של הדוקומנטציה של שדה אקסטרני מסוים.
  + type:MemoryPointer – הכתובת של הסוג שמחזיר המאחזר של שדה אקסטרני מסוים.
  + properties:Map<String, ExtTree> - הילדים של העץ.
* מחלקה – ConstantPool – "בריכה" של ערכים קבועים מסוימים, שלא יכולים להיות מפונים פעמיים, ולא יכולים להשתנות. קיים על מנת לשמר זיכרון.
  + capacity(default, null):Int = 24 – כמות הביטים שהבריכה לוקחת מאחסון של Storage.
  + NULL:MemoryPointer = 0 – המצביע לערך null
  + FALSE:MemoryPointer = 1 – המצביע לערך השקר
  + TRUE:MemoryPointer = 2 – המצביע לערך האמת
  + ZERO:MemoryPointer = 3 – המצביע לערך 0. לוקח 8 ביטים, על מנת שיוכל לייצג מספרים מביט 1 ועד 8 ביטים (byte -> double)
  + INT:MemoryPointer = 11 – המצביע לסוג המספר השלם
  + FLOAT:MemoryPointer = 12 – המצביע לסוג המספר העשרוני
  + BOOL:MemoryPointer = 13 – המצביע לסוג הערך הבוליאני
  + DYNAMIC:MemoryPointer = 14 – המצביע לסוג ה"דינמי" – מייצג כל סוג אחר, חוץ מ-Unknown.
  + TYPE:MemoryPointer = 15 – המצביע לסוג של מחלקות
  + UNKNOWN:MemoryPointer = 16 – המצביע לסוג שאינו ידוע. קיים בשבי בלוקים של קוד וביטויים שסוגם אינו ידוע בזמן שלפני ההרצה עצמה. גם יכול להופיע כאשר משתנה מוכרז ללא סוג או ערך.
  + ERROR:MemoryPointer = 17 – המצביע לשגיאות.
  + EXTERN:MemoryPointer = 18 – מצביע ברירת החדל לערים אקסטרניים שלא ניתן לאחסן בזיכרון רגיל. קיים בשביל פונקציות אקסטרניות, שכן לא ניתן לאחסן טוקנים מסוג HaxeExtern בזיכרון, מהגבלה של הספרייה הסטנדרטית של Haxe.
  + EMPTY\_STRING:MemoryPointer = 19 – המצביע למחרוזת הריקה
  + get(token:InterpTokens):MemoryPointer – פונקציית עזר המחליפה "קיר" של switch-case, המחלצת את המצביע של ערך מסוים המובא כטוקן. אם הטוקן לא נמצא בבריכה, נזרקת שגיאה ברמת קוד ה-Haxe.
  + getFromPointer(pointer:MemoryPointer):InterpTokens – פעולת ההפך מ-get, גם זורקת שגיאה ברמת קוד ה-Haxe.
  + hasPointer(pointer:MemoryPointer):Bool – בודקת האם הכתובת הנתונה נמצאת בבריכת הקבועים.
  + hasType(typeName:String) – בודקת האם סוג מסוים נמצא בתוך בריכת הקבועים.
  + getType(typeName:String):MemoryPointer – מאחזר את כתובת הסוג typeName הנמצא בתוך בריכת הקבועים. אם הוא לא נמצא, נזרקת שגיאה ברמת קוד Haxe.
* מחלקה – Operators – הרחבה של ExternalInterfacing המתואמת לאופרטורים במקום למזהים קלאסיים.
  + priority:Map<Int, Array<{sign:String, side:OperatorType}>> = [] – מפה המקשרת בין אופרטור מסוים, למיקום שלו בסדר הפעולות כאשר הוא פועל על צד מסוים.
  + standard:Map<String, (lhs:InterpTokens, rhs:InterpTokens) -> InterpTokens> = new Map() – מפה המקשרת בין אופרטורים הפועלים על שני הצדדים, לcallback שאמור לממש את פעולת האופרטור.
  + rhsOnly:Map<String, (InterpTokens) -> InterpTokens> = new Map() – אותו דבר כמו standard, אך בשביל אופרטורים הפועלים רק עם אופרנד בצד ימין.
  + lhsOnly:Map<String, (InterpTokens) -> InterpTokens> = new Map() – אותו דבר כמו standard, אך בשביל אופרטורים הפועלים רק עם אופרנד בצד שמאל.
  + setPriority(op:String, type:OperatorType, opPriority:String) – עורך את מפת סדר הפעולות, ומוסיף אופרטור מסוים על צד מסוים לדרגה בסדר הפעולות לפי opPriority. opPriprity יכול להיות מגוון דברים: first, last, with, before, after, between או פשוט מספר.
  + getPriority(op:String, type:OperatorType):Int – מאחזר את המיקום של אופרטור מסוים הפועל על צד מסוים בסדר הפעולות.
  + iterateByPriority():Iterator<Array<{sign:String, side:OperatorType}>> - איטרטור העובר על המפה priority לפי דרגות.
  + call(lhs:InterpTokens, op:String) – מפעיל אופרטור כלשהו עם ארגומנט צד שמאל.
  + call(op:String, rhs:InterpTokens) – מפעיל אופרטור כלשהו עם ארגומנט צד ימין.
  + call(?lhs:InterpTokens = null, op:String, ?rhs:InterpTokens = null):InterpTokens – מפעיל אופרטור כלשהו עם שני הצדדים.

### חבילה – little.tools

מחלקה – Converison – מספקת פעולות להמרה חלקה בין ערכים בHaxe לטוקנים של Little, וכן להפך.

* + extractHaxeType(type:ValueType):String – לא קשור ספציפית לLittle, אך מאפשר קישור בין טוקן סוג **של Haxe** לשם של הסוג.
  + toLittleValue(val:Dynamic):InterpTokens – מנסה להמיר ערך Haxe כלשהו לטוקן Little. כאשר אי אפשר, מחזיר NullValue.
  + toHaxeValue(val:InterpTokens):Dynamic – מנסה להמיר טוקן לערך Haxe. כאשר אי אפשר, מחזיר null.
  + toLittleType(type:String) – ממיר סוגים בסיסיים בHaxe למראה שלהם בLittle (String => Characters)
* מחלקה – Extensions – מכיל פונקציות עזר המרחיבות על כאלה שכבר קיימות על סוגים כמו Array, InterpTokens, ועוד.
  + is(token:ParserTokens, ...tokens:ParserTokensSimple) – בודק האם השם כותרת של טוקן מסוים מסוג ParserTokens נמצא בתוך הפרמטרים הבאים.
  + is(token:InterpTokens, ...tokens:InterpTokensSimple) - בודק האם השם כותרת של טוקן מסוים מסוג InterpTokens נמצא בתוך הפרמטרים הבאים.
  + tokenize(code:String):Array<InterpTokens> - קיצור ל: Interpreter.convert(...Parser.parse(Lexer.lex(code)))
  + eval(code:String):InterpTokens – קיצור ל: Interpreter.run(Interpreter.convert(...Parser.parse(Lexer.lex(code))))
  + parameter(token:ParserTokens, index:Int):Dynamic – מחלץ את הפרמטר ה-index שבADT מסוג InterpTokens- token.
  + parameter(token:InterpTokens, index:Int):Dynamic – מחלץ את הפרמטר ה-index שבADT מסוג ParserTokens- token.
  + passedByValue(token:InterpTokens):Bool – בודק האם טוקן מסוים אמור להיום מעובר לפי ערך.
  + passedByReference(token:InterpTokens):Bool – בודק האם ערך מסוים אמור להיות מועבר לפי אזכור.
  + staticallyStorable(token:InterpTokens):Bool – בודק האם ניתן לקרוא ל storage.storeStatic על טוקן מסוים.
  + extractIdentifier(token:InterpTokens):String – מייצג טוקן מסוים כמחרוזת.
  + asStringPath(token:InterpTokens):Array<String> - מיצג טוקן מסוים שהוא מסלול מזהים כמערך של מזהים.
  + asJoinedStringPath(token:InterpTokens):String – לוקח את asStringPath ומצרף אותו באמצעות Little.keywords.PROPERTY\_ACCESS\_SIGN
  + type(token:InterpTokens):String – מחזיר את הסוג של טוקן מסוים, בהקשר של הריצה הנוכחית.
  + asObjectToken(o:Map<String, InterpTokens>, typeName:String):InterpTokens – לוקח מפה של מחרוזות לטוקנים, ויוצר ממנה אובייקט בעזרת שם סוג האובייקט.
  + asEmptyObject(a:Array<Dynamic>, typeName:String):InterpTokens – יוצר אובייקט ללא שדות מסוג מסוים בעזרת מערך ריק.
  + asTokenPath(string:String):InterpTokens – ההפך מasJoinedStringPath.
  + extractValue(address:MemoryPointer, type:String):InterpTokens – מנסה לקרוא ערך מהזיכרון של ההרצה הנוכחית בעזרת מיקום וסוג.
  + writeInPlace(address:MemoryPointer, value:InterpTokens) – כותב ערך מסוים למקום מסוים בזיכרון, ללא התייחסות להאם המקום פנוי או לא.
  + toIdentifierPath(propertyAccess:InterpTokens):Array<InterpTokens> - ממיר טוקן PropertyAccess לקבוצה של Identifiers. האלמנט הראשון לא חייב להיות מזהה.
  + containsAny<T>(array:Array<T>, func:T -> Bool):Bool – הרחבה למערך קלאסי, בודק האם הפונקציה הנתונה מחזירה אמת לאלמנט כלשהו במערך.
  + toArray<T>(iter:Iterator<T>):Array<T> - ממיר איטרטור (מבנה הספירה בטווח של Haxe) למערך של האלמנטים שהוא מייצג.
* ספירה – ParserTokensSimple – הכותרות של אלמנטים ב-[ParserTokens](#_חבילה_–_little.parser), בUPPER\_SNAKE case.
* ספירה – InterpTokensSimple – הכותרות של אלמנטים ב-[InterpTokens](#_חבילה_–_little.interpreter), בUPPER\_SNAKE case.
* ספירה אבסטרקטית – Layer – כאשר נזרקת שגיאה או אזהרה במצב debug, מוזכר המקום שזרק את אותה שגיאה. המקומות נלקחים מכאן
  + LEXER = "Lexer" – שלב הבניה הראשון
  + PARSER = "Parser" – שלב הבניה שני
  + PARSER\_MACRO = "Parser, Macro" – מאקרואים בשלב הבנייה השני.
  + INTERPRETER = "Interpreter" – שלב ההרצה, כללי.
  + INTERPRETER\_VALUE\_EVALUATOR = "Interpreter, Value Evaluator" – שלב ההרצה, הערכת ביטויים
  + INTERPRETER\_EXPRESSION\_EVALUATOR = "Interpreter, Expression Evaluator" – שלב ההרצה, חישוב ביטויים
  + INTERPRETER\_TOKEN\_VALUE\_STRINGIFIER = "Interpreter, Token Value Stringifier" – שלב ההרצה, הפיכת טוקן למחרוזת.
  + INTERPRETER\_TOKEN\_IDENTIFIER\_STRINGIFIER = "Interpreter, Token Identifier Stringifier" – שלב ההרצה, הפיכת מזהה למחרוזת.
  + MEMORY = "Memory" – זיכרון, כללי.
  + MEMORY\_REFERRER = "Memory, Referrer" – זיכרון, Referrer.
  + MEMORY\_STORAGE = "Memory, Storage" – זיכרון, Storage
  + MEMORY\_EXTERNAL\_INTERFACING = "Memory, External Interfacing" – זיכרון, אלמנטים אקסטרניים.
  + MEMORY\_SIZE\_EVALUATOR = "Memory, Size Evaluator" – זיכרון, חישוב גודל אלמנט.
  + getIndexOf(layer:String):Int – ממיר שכבה מסוימת למספרה לפי סדר ההכרזה בספירה האבסטרקטית. משומש כexitCode בשפה.
* מחלקה – Plugins – גישור שני, בין המפתח הרגיל לExternalInterfacing וחלק משאר הזיכרון. מאפשר הכנסה טבעית כמה שאפשר של אלמנטים מHaxe אל תוך הריצה של קוד Little.
  + registerType(typeName:String, fields:TypeFields) – מכניס סוג אקסטרני לזמן הריצה, באמצעות צורת כתיבה טבעית אך עדיין קרובה למקור ככל האפשר. ניתן להכריז על מפונקציות ומשתנים סטטיים ותלויי מקרה, וכל אחד מהם יכול להחזיר או רק ערך, או גם ערך וגם כתובת, תלוי ברצון המפתח ובמקרה. ניתן גם לאחסן אובייקטים כשדות סטטיים בדרך קלה.
  + registerVariable(variableName:String, variableType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:Void -> InterpTokens) – מכריז על משתנה גלובלי הנמצא בשורש, עם סוג, דוקומנטציה, וערך מסוים. ניתן לתת ערך סטטי, אך גם ניתן לתת ערך שיכול להשתנות, בעזרת נתינת פונקציית אחזור לפונקציה הזו.
  + registerFunction(functionName:String, ?documentation:String, expectedParameters:EitherType<String, Array<InterpTokens>>, callback:Array<{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}> -> InterpTokens, returnType:String) – מכריז על פונקציה גלובלית הנמצאת בשורש, עם דוקומנטציה, פרמטרים, וגוף במכה אחת. ניתן לספק פרמטרים גם כמחרוזת המכילה הכרזת פרמטרים בLittle, או פשוט מערך של טוקנים מסוג InterpTokens. הcallback מקבל את כל הפרמטרים שהובאו לפונקציה מהצד של Little גם כערך וגם ככתובת.
  + registerCondition(conditionName:String, ?documentation:String ,callback:(params:Array<InterpTokens>, body:Array<InterpTokens>) -> InterpTokens) – מכריז על לולאה/תנאי, עם דוקומנטציה מסוימת. המפתח יכול לקבל גישה לתנאי ההרצה של התנאי/לולאה ולגוף שלה, ולבחור כמה פעמים/איך הוא רוצה להריץ את הגוף (או, תכנית, אפילו את תנאי ההרצה, שום דבר לא מונע זאת)
  + registerInstanceVariable(propertyName:String, propertyType:String, onType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:(objectValue:InterpTokens, objectAddress:MemoryPointer) -> InterpTokens) – דומה לregisterVariable, אך בהקשר של משתנה על סוג מסוים. הcallback מקבל שדרוג, ומקבל גם מידע על ההורה של אותו שדה באותה גישה.
  + registerInstanceFunction(propertyName:String, onType:String, ?documentation:String, expectedParameters:EitherType<String, Array<InterpTokens>>, callback:(objectValue:InterpTokens, objectAddress:MemoryPointer, params:Array<{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}>) -> InterpTokens, returnType:String) – דומה לregisterFunction, אך בהקשר של פונקציה על סוג ערך מסוים. כמו מקודם, ה callbackמקבל שדרוג קל על מנת שיכול להיות תלוי בהורה שלו.
  + registerOperator(symbol:String, info:OperatorInfo) – מכריז על אופרטור, הפועל על צדדים מסוימים, עם עדיפות מסוימת ותומך בסוגים מסוימים, או, בשילוב סוגים בסדר מסוים.
* הגדרה – OperatorInfo – הגדרות לאופרטור המוכרז:
  + ?lhsAllowedTypes:Array<String> - סוגי הערכים שאפשר להעביר לצד השמאלי של האופרטור.
  + ?rhsAllowedTypes:Array<String> - סוגי הערכים שאפשר להעביר לצד הימני של האופרטור
  + ?allowedTypeCombos:Array<{lhs:String, rhs:String}> - קומבינציות סוגים מיוחדות שיש בהן תמיכה, מעבר לסוגים המוסכמים.
  + ?callback:(InterpTokens, InterpTokens) -> InterpTokens – פונקציות אחזור הערך של הפעולה עם האופרטור, כאשר האופרטור מקבל שני צדדים. אם operatorType אינו LHS\_RHS, או, גם singleSidedCallback קיים, נזרקות שגיאות הסבר מתאימות.
  + ?singleSidedOperatorCallback:InterpTokens -> InterpTokens –פונקציות אחזור הערך של הפעולה עם האופרטור, כאשר האופרטור מקבל רק צד אחד. אם operatorType אינו LHS או RHS, או, גם callback קיים, נזרקות שגיאות הסבר מתאימות.
  + ?operatorType:OperatorType – סוג האופרטור – מחליט אם הוא מקבל רק צד שמאל (LHS), רק צד ימין (RHS), או צריך את שניהם (LHS\_RHS).
  + priority:String – הדרגה של האופרטור בסדר העדיפויות. יש לשדה זה אפשרויות שאמורות להקל על הכנסת דרגה:
    - אינדקס פשוט: מספר של דרגה, ככל שיותר נמוך הדרגה יותר מוקדמת. -1 תמיד יהיה הכי גבוה.
    - first – האופרטור בעל הדרגה הנמוכה ביותר, יתייחסו אליו ראשון.
    - last – ההפך מfirst.
    - before <sign> - יצורף לדרגה אחת לפני האופרטור המוזכר. על מנת להבדיל בין צדדים, יש להשתמש ב\_: \_+ (RHS), +\_ (LHS), \_+\_ או רק + (LHS\_RHS).
    - after <sign> - ההפך מbefore, דרגה אחת אחרי
    - with <sign> - יצורף באותה דרגה כמו האופרטור המוזכר.
    - between <sign> <sign> - האופרטור יצורף בדיוק באמצע בין הדרגות של הסימנים הנתונים. ייווצרו דרגות במקרה הצורך
* .הגדרה – TypeFields – שם אחר למפה מאוד ארוכה, שאמור לבטא את אפשרויות הרכבת השדות של registerType:

typedef TypeFields = Map<String, OneOfSeven<

    // Instance fields:

    (address:MemoryPointer, value:InterpTokens) -> InterpTokens, // variable

    (address:MemoryPointer, value:InterpTokens) -> {address:MemoryPointer, value:InterpTokens}, // variable, with pointer

    (address:MemoryPointer, value:InterpTokens, givenParams:Array<InterpTokens>) -> InterpTokens, // function

        // Static fields:

    () -> InterpTokens, // variable

    () -> {address:MemoryPointer, value:InterpTokens}, // variable

    (givenParams:Array<InterpTokens>) -> InterpTokens, // function

    TypeFields // nested object

>>;

* אבסטרקט – OneOfSeven – על מנת לספק השלמת קוד למפתחים, הסוג מתקמפל לסוג דינמי, אך לפי קומפילציה מסוגל לייצג ערך שהוא אחר מ7 סוגים:

abstract OneOfSeven<T1, T2, T3, T4, T5, T6, T7>(Dynamic)

    from T1 from T2 from T3 from T4 from T5 from T6 from T7

    to T1 to T2 to T3 to T4 to T5 to T6 to T7 {}

הסיבה שיש רק 7 סוגים היא הגבלה של השפה – משום מה, שרת הקומפליצה (completion) מסרב להתייחס לאבסטרקטים כאלה של מעל ל8 סוגים בצורה צפויה. דיווחתי על הבאג למפתחי שרת הקומפליציה.

מחלקה – PrepareRun – אחראית על יצירת הספרייה הסטנדרטית

* + prepared:Bool – האם הספרייה אכן נוספה להרצה הנוכחית. נערך מבחוץ.
  + addTypes() – מוסיף סוגים מסוימים, ושדות לכל הסוגים שזקוקים להם
  + addFunctions() – מוסיף פונקציות גלובאליות, כמו הדפסה וזריקת שגיאה
  + addProps() – מוסיף שדות תלויי מקרה גלובליים, כמו .type ו.address
  + addSigns() – מוסיף את כל האופרטורים שהשפה תומכת בהם out-of-the-box
  + addConditions() – מוסיף את כל הלולאות והתנאים שהשפה תומכת בהם.
* מחלקה – PrettyPrinter – מספקת יכולות פירמוט והדפסת עץ syntax של קוד ומערכי טוקנים.
  + printParserAst(ast:Array<ParserTokens>, ?spacingBetweenNodes:Int = 6) – מחזיר מחרוזת של עץ הsyntax שמייצג מערך טוקני Parser בצורה יפה, עם ריווח מסוים בין כל קומה בעץ.
  + printInterpreterAst(ast:Array<InterpTokens>, ?spacingBetweenNodes:Int = 6) – מחזיר מחרוזת של עץ הsyntax שמייצג מערך טוקני Interpreter בצורה יפה, עם ריווח מסוים בין כל קומה בעץ.
  + stringifyParser(?code:Array<ParserTokens>, ?token:ParserTokens) – מחזיר את הקוד המובא כמערך טוקני Parser כמחרוזת קוד מפורמטת.
  + stringifyInterpreter(?code:Array<InterpTokens>, ?token:InterpTokens) - מחזיר את הקוד המובא כמערך טוקני Interpreter כמחרוזת קוד מפורמטת.
  + prettyPrintOperatorPriority(priority:Map<Int, Array<{sign:String, side:OperatorType}>>) – מחזיר מחרוזת המייצגת את מפת סדר פעולות של האופרטורים בצורה יפה.
* מחלקה - TextTools – מחלקה חיצונית שהעתקתי כי רציתי לבצע בה שינויים. מכילה מגוון פונקציות הרחבה למחרוזות. מכיוון שהמחלקה לא מהפרויקט הזה לא אפרט עליה.
* אבסטרקט – OrderedMap – מחלקה חיצונית של מפה מסודרת. אין ספריה חדישה עם אימפלמנטציה עובדת, אז הייתי צריך [להעתיק ממקור חיצוני](https://gist.github.com/crowplexus/544381cc224273d19907a454fbce25c4)

### מחלקות נוספות

מחלקה – Little – המחלקה העיקרית של הפרויקט, שנותנת גישה לכל הפונקציות "ההכרחיות" לשימוש בסיס. כל עוד מפתח לא מנסה להשתמש בשפה בצורה מתקדמת יותר, אין סיבה שישתמש בכלים שהמחלקה לא מציעה.

* + keywords:KeywordConfig – מילות המפתח שהשפה תשתמש בהם כאשר יורץ או יבנה קוד
  + runtime(default, null):Runtime – מכיל מידע על זמן הריצה, אירועים בריצה, ותוצאות אחרי הריצה, כמו טענת יציאה.
  + memory(default, null):Memory – הזיכרון שישתמש בו מריץ הקוד על מנת לאחסן ערכים.
  + plugin(default, null):Plugins – הוספת אלמנטים אקסטרניים, ממשתנים רגילים ועד סוגים ואופרטורים.
  + queue(default, null):Queue<String> - תור ההרצה, נעשה שימוש כאשר משתמשים בloadModule.
  + debug:Bool – מחליט האם הקוד מורץ במצב debug. במצב זה, כשמודפסים דברים נוספת השכבה ממנה ההדפסה נקראה.
  + version:String – הגרסה של בונה הקוד והמריץ. בפרויקט הזה, הוא יהיה 1.0.0-f, כש-f מציין שהפרויקט מאפשר "רק" Functional Programming.
  + loadModule(code:String, name:String, debug:Bool = false, runRightBeforeMain:Bool = false) – טוען קוד ומריץ אותו לפני הקוד "הראשי". אם runRightBeforeMain מאופשר, אנו מתעלמים מפרמטר הdebug ומוסיפים את הקוד לתור. אם לא, debug מופעל באופן זמני בהתאם לערך הפרמטר, והקוד מורץ תחת שם המודולה הנתונה.
  + run(code:String, ?debug:Bool) – מריץ קוד, וגם מריץ לפניו כל קוד הנמצא בתור, במצב debug, לפי ערך הפרמטר, או אם הוא Null, לפי ערך Little.debug.
  + compile(code:String):Array<InterpTokens> - רק בונה את הקוד בלי להריץ אותו, ומחזיר את עץ הsyntax.
  + format(code:String):String – בונה וממיר מחדש את הקוד למחרוזת, ובכך מפרמט את הקוד.
  + reset() – מאפס את כל האלמנטים שצריך לאפס לפני ריצה חדשה וטרייה (זיכרון, תור...)

מחלקה – KeywordConfig – כל מילות המפתח והסימנים שמשתמשת בהם השפה. קיימים בין 50 ל100, ולכן לא אכתוב הסבר על כולם, אלא על כאלה חשובים/לא בהכרח מובנים.

* + VARIABLE\_DECLARATION:String
  + FUNCTION\_DECLARATION:String
  + TYPE\_DECL\_OR\_CAST:String
  + FUNCTION\_RETURN:String
  + NULL\_VALUE:String
  + TRUE\_VALUE:String
  + FALSE\_VALUE:String
  + TYPE\_DYNAMIC:String
  + TYPE\_INT:String
  + TYPE\_FLOAT:String
  + TYPE\_BOOLEAN:String
  + TYPE\_STRING:String
  + TYPE\_OBJECT:String
  + TYPE\_MEMORY:String
  + TYPE\_ARRAY:String
  + TYPE\_FUNCTION:String
  + TYPE\_CONDITION:String
  + TYPE\_MODULE:String
  + TYPE\_SIGN:String
  + MAIN\_MODULE\_NAME:String – מודולת ברירת המחדל, ממנה מורץ הקוד של Little.run.
  + OBJECT\_TYPE\_PROPERTY\_NAME:String
  + OBJECT\_ADDRESS\_PROPERTY\_NAME:String
  + PRINT\_FUNCTION\_NAME:String
  + RAISE\_ERROR\_FUNCTION\_NAME:String
  + READ\_FUNCTION\_NAME:String
  + RUN\_CODE\_FUNCTION\_NAME:String
  + CONDITION\_PATTERN\_PARAMETER\_NAME:String – כאשר קוראים לתנאי/לולאה, תנאי ההרצה מאוחסן במשתנה בשם הזה.
  + CONDITION\_BODY\_PARAMETER\_NAME:String – כאשר קוראים לתנאי/לולאה, גוף הלולאה מאוחסן במשתנה בשם הזה.
  + CONDITION\_\_FOR\_LOOP:String
  + CONDITION\_\_WHILE\_LOOP:String
  + CONDITION\_\_IF:String
  + CONDITION\_\_ELSE:String
  + CONDITION\_\_WHENEVER:String
  + CONDITION\_\_AFTER:String
  + TYPE\_UNKNOWN:String
  + RECOGNIZED\_SIGNS:Array<String> - לא אמור להיערך, אופרטורים שנוספו מאוחסנים כאן על מנת שיוכלו להיקרא.
  + PROPERTY\_ACCESS\_SIGN:String – גישה לשדה על ערך מסוים.
  + EQUALS\_SIGN:String
  + NOT\_EQUALS\_SIGN:String
  + LARGER\_SIGN:String
  + SMALLER\_SIGN:String
  + LARGER\_EQUALS\_SIGN:String
  + SMALLER\_EQUALS\_SIGN:String
  + XOR\_SIGN:String
  + OR\_SIGN:String
  + AND\_SIGN:String
  + NOT\_SIGN:String
  + ADD\_SIGN:String
  + SUBTRACT\_SIGN:String
  + MULTIPLY\_SIGN:String
  + DIVIDE\_SIGN:String
  + MOD\_SIGN:String
  + POW\_SIGN:String
  + FACTORIAL\_SIGN:String
  + SQRT\_SIGN:String
  + NEGATE\_SIGN:String
  + POSITIVE\_SIGN:String
  + STDLIB\_\_FLOAT\_isWhole:String
  + STDLIB\_\_STRING\_length:String
  + STDLIB\_\_STRING\_toLowerCase:String
  + STDLIB\_\_STRING\_toUpperCase:String
  + STDLIB\_\_STRING\_trim:String
  + STDLIB\_\_STRING\_substring:String
  + STDLIB\_\_STRING\_charAt:String
  + STDLIB\_\_STRING\_split:String
  + STDLIB\_\_STRING\_replace:String
  + STDLIB\_\_STRING\_remove:String
  + STDLIB\_\_STRING\_contains:String
  + STDLIB\_\_STRING\_indexOf:String
  + STDLIB\_\_STRING\_lastIndexOf:String
  + STDLIB\_\_STRING\_startsWith:String
  + STDLIB\_\_STRING\_endsWith:String
  + STDLIB\_\_STRING\_fromCharCode:String
  + STDLIB\_\_ARRAY\_length:String
  + STDLIB\_\_ARRAY\_elementType:String
  + STDLIB\_\_ARRAY\_get:String
  + STDLIB\_\_ARRAY\_set:String
  + STDLIB\_\_MEMORY\_allocate:String
  + STDLIB\_\_MEMORY\_free:String
  + STDLIB\_\_MEMORY\_read:String
  + STDLIB\_\_MEMORY\_write:String
  + STDLIB\_\_MEMORY\_size:String
  + STDLIB\_\_MEMORY\_maxSize:String
  + FOR\_LOOP\_FROM:String
  + FOR\_LOOP\_TO:String
  + FOR\_LOOP\_JUMP:String
  + TYPE\_CAST\_FUNCTION\_PREFIX:String
  + INSTANTIATE\_FUNCTION\_NAME:String
  + - change(config:KeywordConfig)משנה את הקונפיגורציה הנוכחית לקונפיגורציה הנתונה בפרמטר. אם חלקים מהקונפיגורציה הנתונה הם null, מתעלמים מהם ולא עורכים את הקונפיגורציה הנוכחית באמצעותם.

## קוד "יפה" ותכנות אלגוריתמים

### קריאה לפונקציות אקסטרניות

כחלק מהפרויקט, מאופשרת קריאה לפונקציות אקסטרניות, באותה דרך ו"קשיחות" כמו פונקציה רגילה (כמות פרמטרים סטטית, סוגים קונקרטיים לכל פרמטר, סוג החזרה...)  
בזכות הטוקן HaxeExtern, שמאפשר לנו להריץ קוד Haxe תוך כדי זמן ריצה של קוד Little זה נהפך ליחסית קל, אבל הבעיות מתחילות בפרמטרים – בגלל שפרמטרים מובאים לפונקציה בתור "הדבקה" של ההכרזה שלהם לגוף הפונקציה, אי אפשר בקלות לאחזר את אותם ערכים. לכן, הייתי צריך לעשות קצת "ריקודים" בשביל להשיג את המידע הזה:

var token:InterpTokens = FunctionCode(paramMap, Block([

            FunctionReturn(HaxeExtern(() -> callback(paramMap.keys().toArray().map(key -> memory.read(key))), returnTypeToken)

        ], returnTypeToken));

(הסבר מקדים – callback היא הפונקציה האקסטרנית עצמה, paramMap מייצג את הפרמטרים שפונקציה מבקשת. FunctionCode זה ה"ערך" של פונקציה. Block זה פשוט בלוק של קוד. returnTypeToken הוא סוג הערך המוחזר ע"י הפונקציה)

קורה כאן משהו כזה – HaxeExtern מבקש פונקציה ללא פרמטרים ושמחזירה טוקן. callback מחזיר את הערך שאנחנו צריכים, אך צריך פרמטרים כמערך. אנו יודעים שכל הפרמטרים צריכים להיות מובאים, ולכן, אנו מדביקים את ערכי הפרמטרים כך:

לוקחים את השמות של הפרמטרים כIterator

ממירים למערך

לכל מפתח במערך, קוראים אותו מהזיכרון, ומחזירים את הערך, המצביע והסוג שלו.

### Shunting Yard – השראה

חישוב הביטויים של הפרויקט נעשה באמצעות אלגוריתם בהשראת Shunting Yard, לפחות בחלק החישובי שלו. האימפלמנטציה עצמה מאוד ארוכה (כ150) שורות, אבל רובה עדיין לוגית.

לפני שמתחילים, יש לדעת את ההבדל בין PartArray לExpression – על אף שהוא פשוט, הוא הכרחי להבנת האלגוריתם – Expression זה הצורה הטבעית של קיבוץ אלמנטים, בעוד שPartArray היא הגרסה המלאכותית, שהאלגוריתם מכניס.

הנה הפונקציות. נתחיל דווקא מהאחרונה, group:

public static function group(tokens:Array<InterpTokens>):Array<InterpTokens> {

    var post = tokens;

    var pre = [];

    for (operatorGroup in Little.memory.operators.iterateByPriority()) {

        pre = post.copy();

        post = [];

        // We'll group everything by only recognizing specific signs each "stage" -

        // The signs recognized first will be of the highest priority.

        // One drawback of this system is that its a little messier to detect chaining (e.g. 5!!, √√√64)

        var i = 0;

        while (i < pre.length) {

            var token = pre[i].is(IDENTIFIER, BLOCK) ? evaluate(pre[i]) : pre[i];

            switch token {

                case Sign(operatorGroup.filter(x -> x.sign == \_).length > 0 => true): {

                    // If there's an operator before this one, its RHS\_ONLY. If there's an operator after, its LHS\_ONLY

                    // If there's no operator, its LHS\_RHS

                    // First lets do a simple edge case - i = pre.length - 1 => LHS\_ONLY operator.

                    if (i == pre.length - 1) {

                        post.push(PartArray([post.pop(), token]));

                        break;

                    }

                    var lookbehind = post.length > 0 ? post[post.length - 1] /\* Post has only evaluated tokens \*/ : Sign("\_"); // Just an arbitrary "sign" to not have null here

                    var lookahead = pre[i + 1].is(IDENTIFIER, BLOCK) ? evaluate(pre[i + 1]) : pre[i + 1];

if (lookahead.is(SIGN) && operatorGroup.filter(x -> x.sign == lookahead.parameter(0)).length > 0) {

                        /\* This can be one of two cases:

                        - were working on a binary operator before a unary operator (1 or more)

                        - were working on a unary operator (1 or more) before a binary operator

                        We should naturally prioritize unary operators since they, resulting from their definition, always come first.

                        This makes the parsing very easy, since the first possible binary operator must be the binary one: (pretend +, - and ! are of the same priority)

                        5!! + ---5

                        both + and - cant be LHS\_ONLY, so grouping is:

                        (((5!)!) + (-(-(-5)))) \*/

                        if (operatorGroup.filter(x -> x.sign == token.parameter(0) && x.side == LHS\_ONLY).length > 0) {

                            post.push(PartArray([post.pop(), token]));

                        } else if (operatorGroup.filter(x -> x.sign == token.parameter(0) && x.side == LHS\_RHS).length > 0) {

                            var operand1 = post.pop();

                            var op = lookahead;

                            // We have to repeat the check in RHS\_ONLY, since RHS can also start with a sign

                            if (i + 2 >= pre.length) error("Expression ended with an operator, when an operand was expected.");

                            var lookahead2 = pre[i + 2].is(IDENTIFIER, BLOCK) ? evaluate(pre[i + 2]) : pre[i + 2];

                            if (!lookahead2.is(SIGN)) {

                                post.push(PartArray([operand1, token, PartArray([lookahead, lookahead2])]));

                                i += 2; // +2 because we consumed both lookahead and lookahead2 for the PartArray arg

                            } else {

                                var g = [];

                                while (lookahead2.is(SIGN) && operatorGroup.filter(x -> x.sign == lookahead2.parameter(0) && x.side == RHS\_ONLY).length > 0) {

                                    g.push(lookahead2);

                                    i++;

                                    if (i + 2 >= pre.length) error("Expression ended with an operator, when an operand was expected.");

                                    lookahead2 = pre[i + 2].is(IDENTIFIER, BLOCK) ? evaluate(pre[i + 2]) : pre[i + 2];

                                }

                                // Last token is an operand

                                g.push(lookahead2);

                                // And increment i since lookahead2 uses i + 1

                                i++;

                                var operand2 = g.length == 1 ? g[0] : PartArray(group(g));

                                post.push(PartArray([operand1, op, operand2]));

                            }

                        } else if (operatorGroup.filter(x -> x.sign == token.parameter(0) && x.side == RHS\_ONLY).length > 0) {

                            error("An operator that expects a right side can't be preceded by an operator that expects a left side.");

                        }

                    } else {

                        // Both sides are regular operands, so we just pop from `post` and take the lookahead

                        // And no, we shouldn't worry about order of operations here. because of this "algorithm"'s format, all

                        // operators are of the same priority, and its the user's responsibility to use parentheses when needed.

                        if (lookahead.is(SIGN)) {

                            post.push(PartArray([post.pop(), token]));

                        } else {

                            post.push(PartArray([post.pop(), token, lookahead]));

                            i++;

                        }

                    }

                }

                case Expression(parts, type): post.push(Expression(group(parts), type));

                case \_: post.push(token);

            }

            i++;

        }

    }

    return post;

}

ואז נעבור לcalculate:

public static function calculate(p:Array<InterpTokens>):InterpTokens {

    while (p.length == 1 && p[0].parameter(0) is Array && !p[0].is(BLOCK)) p = p[0].parameter(0);

    var tokens = group(p);

    var castType:InterpTokens = null;

    if (tokens.length == 1) {

        if (tokens[0].is(PART\_ARRAY)) tokens = tokens[0].parameter(0);

        else if (tokens[0].is(EXPRESSION)) {

            tokens = tokens[0].parameter(0);

            castType = tokens[0].parameter(1);

        } else if (tokens[0].is(BLOCK)) {

            tokens = [run(tokens[0].parameter(0))];

            castType = tokens[0].parameter(1);

        }

    }

    var calculated:InterpTokens = null;

    var sign:String = "";

    tokens = tokens.filter(x -> x != null); // Safety clause, for strange edge cases such as 2 + ---5.

    for (token in tokens) {

        switch token {

            case PartArray(parts): {

                if (sign != "" && calculated == null) calculated = Little.memory.operators.call(sign, calculate(parts)); // RHS operator

                else if (calculated == null) calculated = calculate(parts);

                else if (sign == "") error('Two values cannot come one after the other ($calculated, $token). At least one of them should be an operator, or, put an operator in between.');

                else {

                    calculated = Little.memory.operators.call(calculated, sign, calculate(parts)); // standard operator

                }

            }

            case Sign(s): {

                sign = s;

                if (tokens.length == 1) return token;

                if (tokens[tokens.length - 1].equals(token)) calculated = Little.memory.operators.call(calculated, sign); //LHS operator

            }

            case Expression(parts, t): {

                var val = t != null ? typeCast(calculate(parts), t) : calculate(parts);

                if (sign != "" && calculated == null) calculated = Little.memory.operators.call(sign, val); // RHS operator

                else if (calculated == null) calculated = val;

                else if (sign == "") error('Two values cannot come one after the other ($calculated, $token). At least one of them should be an operator, or, put an operator in between.');

                else {

                    calculated = Little.memory.operators.call(calculated, sign, val); // standard operator

                }

            }

            case SetModule(module): setModule(module);

            case \_: {

                if (sign != "" && calculated == null) calculated = Little.memory.operators.call(sign, token);

                else if (sign == "" && calculated != null) throw 'Unexpected token: $token After calculating $calculated';

                else if (calculated == null) calculated = token;

                else if (sign == "") error('Two values cannot come one after the other ($calculated, $token). At least one of them should be an operator, or, put an operator in between.');

                else {

                    calculated = Little.memory.operators.call(calculated, sign, token);

                }

            }

        }

    }

    if (castType != null) return typeCast(calculated, castType);

    return calculated;

}

#### group()

הפונקציה מתחילה ישר ולעניין, מכיוון שאין ערך בלבדוק למקרי קיצון, הם כולם יתנהגו נכון.  
אופרטורים בזמן הריצה מופרדים לקבוצות לפי סדר פעולות. את סדר הפעולות ואת הסימנים אנו באמת מושכים משם, בשורה:

        for (operatorGroup in Little.operators.iterateByPriority()) {

ואז מתחילים לעבור על הטוקנים של השכבה הנוכחית – כל פעם שאנו מסיימים קבוצה, הטוקנים האלו מעובדים קצת יותר:

        while (i < pre.length) {

מכיוון שאופרטור יכול להתבטא גם כמזהה או בלוק, יש לחשב אותם, ורק אז להתחיל:

            var token = pre[i].is(IDENTIFIER, BLOCK) ? evaluate(pre[i]) : pre[i];

ועכשיו, מתחיל ה"סיבוך" האמיתי – בסופו של יום יש רק 3 מקרים בהם צריך לבנות את הoutput בצורה שונה, אך אחד מהם מאוד, *מאוד* סבוך. אתחיל מהמקרים הסופיים, שהם התקלות במספר או ביטוי:

case Expression(parts, type): post.push(Expression(group(parts), type));

case \_: post.push(token);

המקרה האחר אולי נראה קצת סבוך, אבל הוא פשוט בודק האם קיבלנו אופרטור, והאם הסימן שהוא מכיל נמצא בשכבה הנוכחית:

case Sign(operatorGroup.filter(x -> x.sign == \_).length > 0 => true):

בדיקה זו מתחילה במקרה קצה פשוט, שהוא האם הסימן נמצא בסוף המערך, במקרה כזה הוא חייב להיות אופרטור של צד שמאל בלבד:

if (i == pre.length - 1) {

post.push(PartArray([post.pop(), token]));

break;

}

לאחר הבדיקה, אנו מושכים את הטוקן מלפנים ומאחורה, ובודקים שני מקרים:

**מקרה ראשון: מלפנינו יש אופרטור, והוא באותה דרגה בסדר הפעולות כמו האופרטור הזה.**

מייצג מספר מקרים:

* + **סאב-מקרה שני: אנחנו אופרטור חד צדדי של צד שמאל, ומלפנינו אופרטור דו צדדי.** במקרה זה, אנו פשוט לוקחים את הטוקן האחרון שעיבדנו, ושמים אותו איתנו כאשר הטוקן הוא אופרנד שמאלי.
  + **סאב-מקרה ראשון: אנחנו אופרטור דו-צדדי, ומלפנינו סימנים חד-צדדיים ברצף.** במקרה זה, אנו עוקבים אחר הסימנים ומוסיפים אותם למערך:

var g = [];

while (lookahead2.is(SIGN) && operatorGroup.filter(x -> x.sign == lookahead2.parameter(0) && x.side == RHS\_ONLY).length > 0) {

    g.push(lookahead2);

    i++;

    if (i + 2 >= pre.length) error("Expression ended with an operator, when an operand was expected.");

    lookahead2 = pre[i + 2].is(IDENTIFIER, BLOCK) ? evaluate(pre[i + 2]) : pre[i + 2];

}

// Last token is an operand

g.push(lookahead2);

// And increment i since lookahead2 uses i + 1

i++;

לאחר שבנינו את המערך והוספנו את האופרנד שאחרי הסימנים, הופכים אותו לאופרנד השמאלי, בעזרת קריאה רקורסיבית לgroup על המערך שבנינו (כשהאיבר האחרון במערך זה האופרנד שקבוצת האופרטורים החד צדדיים פועלים עליו):

var operand2 = g.length == 1 ? g[0] : PartArray(group(g));

עכשיו שיש לנו את הצד **הימני** של האופרטור שאנחנו עובדים עליו כרגע, וכבר מלפני זה יש לנו את האופרנד השמאלי:

var operand1 = post.pop();

לוקחים את הטוקן הקודם, ובונים PartArray על הכל:

post.push(PartArray([operand1, op, operand2]));

* **מקרה שני: מלפנינו טוקן של ערך, מאחורינו אופרטור בעל עדיפות נמוכה או ערך**

גם מייצג שני מקרים:

* + **סאב-מקרה ראשון: יש מלפנינו אופרטור כלשהו.** במקרה כזה אנחנו אופרטור חד צדדי של צד שמאל:

if (lookahead.is(SIGN)) {

post.push(PartArray([post.pop(), token]));

}

* + **סאב-מקרה שני: יש מלפנינו ערך.** אם מלפנינו ומאחורינו יש טוקן שניתן להתייחס אליו כערך, אנחנו אופרטור דו-צדדי רגיל. אנו מוסיפים לi מכיוון שצרכנו את lookahead:

else {

    post.push(PartArray([post.pop(), token, lookahead]));

    i++;

}

#### calculate()

פונקציה יותר פשוטה - הפונקציה מתחילה בדאגה למקרים פשוטים/פירוק אלמנט אחד המכיל הרבה אחרים, למערך אלמנטים נורמאלי.

אנו מכריזים על 3 משתנים: אופרטור עדכני, ערך מחושב עד עכשיו, וסוג מחושב עד עכשיו.

לאחר מכן, אנו מתחילים לעבור על הטוקנים, אחד אחרי השני. כל טוקן יכול להיות אחד מ4 מקרים.

**מקרה ראשון: קביעת מודולה.** אנו מתעלמים, שכן הטוקן הזה מופיע בתחילת כל גוף אשר מסוגל לקפוץ שורות קוד, ואין לא ערך אמיתי.

**מקרה שני: אופרטור.** האופרטור העדכני נערך לאופרטור הזה.שני מקרי קצה:

* + **ראשון-** זה הטוקן היחיד שנמצא. הוא מוחזר.
  + **שני-** זה הטוקן האחרון במערך. מחשבים קריאה לערך שחושב עד עכשיו באופרנד שמאלי לאופרטור הזה.שמים אותה בתוך הערך והסוג המחושבים.
* **מקרה שלישי: PartArray או Expression.** מדובר בסוגריים שאו הפונקציה group שמה, או שהמשתמש שם. מכאן הפעולות הבאות מותנות:
  + **יש אופרטור עדכני והמחושב עד עכשיו לא קיים.** משווים את הערך הנוכחי לקריאה לאופרטור העדכני עם אופרנד צד ימין בלבד.
  + **אין ערך מחושב עד עכשיו.** משווים אותו לחישוב של הסוגריים הנוכחיים.
  + **יש ערך מחושב, אין אופרטור.** זורקים שגיאה ברמת Little, שכן שני אופרנדים ללא הקשר לא יכולים לבוא מיד אחרי השני.
  + **אחרת:** קוראים לאופרטור הנוכחי עם הערך המחושב עד עכשיו באופרנד שמאלי, והסוגריים הנוכחיים כאופרנד ימני. משווים את התוצאה לערך העדכני.

תלוי במקרה, יש אפשרות לנסות יציקה לסוג מסוים

* **מקרה רביעי: כמו מקרה שלישי, אך עם כל טוקן ערך אחר.** התנהגות זהה למקרה 3, אך ללא אפשרות ליצוק את הערך לסוג.

### עיצוב ה-Referrer

(לא אדביק כאן קוד ספציפי, מעבר לפונקציה אחת, רק כדי להראות את ה"קטע".)

הReferrer נעשה כמין שילוב של מבנה הסטאק וה-heap – הוא בנוי כמו הסטאק בדרך שמסגרות נוספות אליו, ודומה לheap בדרך שזיכרון מנוהל בו.

כל מסגרת בנויה כמערך, שכל תא בו בגודל 16 ביטים, ומכיל 4 פיסות מידע:

4 ביטים ראשונים – ערך הhash של שם המשתנה

4 ביטים שניים – מיקום שם המשתנה במחרוזת בזיכרון

4 ביטים שלישיים – מיקום הערך בזיכרון

4 ביטים אחרונים – מיקום סוג הערך בזיכרון.

בתחילת כל מסגרת יש header של 4 ביטים, המורכב גם הוא מחלקים:

שני ביטים ראשונים – כמות האלמנטים הנמצאים במסגרת הקודמת

שני ביטים אחרונים – כמות האלמנטים שנוספו עד עכשיו למסגרת הנוכחית

מעבר לheader שיש לכל מסגרת, המערך מתחיל ב4 ביטים של UInt32 שמצביעים על מיקום המסגרת הנוכחית בזיכרון הReferrer.

מבנה זה מאפשר לבצע פעולות חשובות במהירות ובקלות, כמו לשחק עם מסגרות במהירות ובקלות, ולעבור על אלמנטים בקצת קוד, ועדיין במהירות:

/\*\*

        Creates a new scope. Fields created after this will be added to the new scope, and won't affect fields from the previous scopes.

    \*\*/

    public function pushScope() {

        var currentScopeLength = bytes.getUInt16(bytes.getInt32(0) + 2);

        var currentScopeStart = bytes.getInt32(0) + 4; // each header is 4 bytes long.

        var header = new ByteArray(4);

        header.setUInt16(0, currentScopeLength);

        header.setUInt16(2, 0); // Currently, 0 elements ahead.

        var writePosition = currentScopeStart + currentScopeLength \* KEY\_SIZE;

        if (writePosition + 2 + 2 > bytes.length) {

            requestMemory();

        }

        bytes.setBytes(writePosition, header);

        bytes.setInt32(0, writePosition); // Update the start of the scope.

    }

    /\*\*

        Removes the last scope. TODO: Garbage collection.

    \*\*/

    public function popScope() {

        var currentScopePosition = bytes.getInt32(0);

        var previousScopeLength = bytes.getUInt16(currentScopePosition);

        var currentScopeLength = bytes.getUInt16(currentScopePosition + 2);

        // Update the start of the scope. Also, -4 is for the header, since we need to point to its start.

        bytes.setInt32(0, currentScopePosition - previousScopeLength \* KEY\_SIZE - 4);

    }

/\*\*

        Returns an iterator over all key/value pairs in all scopes, starting from the current scope, and going up the parent scopes.

    \*\*/

    public function keyValueIterator():KeyValueIterator<String, {address:MemoryPointer, type:String}> {

        var map = new Map<String, {address:MemoryPointer, type:String}>();

        var checkingScope = currentScopeStart;

        var elementCount = bytes.getUInt16(currentScopeStart + 2);

        var nextScope = currentScopeStart - bytes.getUInt16(currentScopeStart) \* KEY\_SIZE - 4;

        do {

            var i = checkingScope;

            while (i < (checkingScope + elementCount \* KEY\_SIZE)) {

                var stringName = parent.storage.readString(bytes.getInt32(i + 4));

                map.set(stringName, {

                    address: MemoryPointer.fromInt(bytes.getInt32(i + 4 + POINTER\_SIZE)),

                    type: parent.storage.readString(bytes.getInt32(i + 4 + POINTER\_SIZE \* 2))

                });

                i += KEY\_SIZE;

            }

            checkingScope = nextScope;

            elementCount = bytes.getUInt16(nextScope + 2);

            nextScope = nextScope - bytes.getUInt16(nextScope) \* KEY\_SIZE - 4;

        } while (checkingScope != 0);

        return map.keyValueIterator();

    }

בעוד שהחסרונות" שבמבנה מהסוג שלו בקושי מתממשות

חסרון ראשון: גישה בסיבוכיות O(n) – לא רלוונטי מכיוון שרק לעיתים נדירות מאוד מסגרת אחת בReferrer תכיל הרבה מפתחות. במקרה כזה, אפשר אפילו להגיד שזו בעיה אצל המפתח...

חסרון שני: גודל – גם לא רלוונטי, המקרים היחידים בהם מפתח יכול למלא את הReferrer "מהר במיוחד" הם גם אותם מקרים שבהרצות רגילות הוא יקבל stack overflow, אז אולי הטרמינציה המוקדמת אפילו מועילה...

### יצירת סוגים אקסטרניים

גם כאן הקוד ארוך ואולי נראה סבוך, אך יחסית פשוט. אפצל את ההסבר והתכונות לשלבים למען הנוחות. להלן הקוד:

@:noCompletion static var \_\_noTypeCreation:Bool;

    /\*\*

        registers a class in little code, or extends the fields of an existing class. The class' fields are dictated by this function's `fields` attribute,

        which provides instance & static functions, variables, and nested objects.

        The allowed key-value types in `fields`'s key-value pairs:

        |Key Syntax                                           | Type                                                                                                                          | Application       | Description |

        |-----------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|-------------------|-------------|

        |`public <Type> <name>`                               | `(address:MemoryPointer, value:InterpTokens) -> InterpTokens`                                                                 | Instance Variable | A function that returns a value, that can be based on its parent. The returned value is stored in memory upon retrieval. |

        |`public <Type> <name>`                               | `(address:MemoryPointer, value:InterpTokens) -> {address:MemoryPointer, value:InterpTokens}`                                  | Instance Variable | A function that returns a value, that can be based on its parent. The returned value is not stored in memory, and we rely upon the given pointer to be correct. |

        |`public <Type> <name> (define <param> as <Type>)`    | `(address:MemoryPointer, value:InterpTokens, givenParams:Array<InterpTokens>) -> InterpTokens`                                | Instance Function | A function, that returns a value based on its parent & other given parameters, provided by a Little function call. The returned value is stored in memory upon retrieval. |

        |`static <Type> <name>`                               | `() -> InterpTokens`                                                                                                          | Static Variable   | A function that returns a static value. The returned value is stored in memory upon retrieval. |

        |`static <Type> <name>`                               | `() -> {address:MemoryPointer, value:InterpTokens}`                                                                           | Static Variable   | A function that returns a static value. The returned value is not stored in memory, and we rely upon the given pointer to be correct. |

        |`static <Type> <name> ()`                            | `(givenParams:Array<InterpTokens>) -> InterpTokens`                                                                           | Static Function   | A function that returns a value based on some given parameters, provided by a Little function call. The returned value is stored in memory upon retrieval. |

        |`static <Type> <name>`                               | `TypeFields`                                                                                                                  | Static Variable   | Another option for a static variable, but this time it's for a nested object. The object itself isn't allocated in Little's memory, but its decedents may be. instance objects are not available this way, since they are tied to an object, thus needing to be allocated many times. |

**\*\*Notice\*\*** - key syntax is very sensitive - must start with `public` or `static`, continue with a `little` type, then a name, and parameters if its a function. Each element separated by a single whitespace. Example:

            'public Number id'

            'static ${Conversion.toLittleType("String")} getProfile (define summed as ${Conversion.toLittleType("Bool")})'

**\*\*Notice 2\*\*** - for function parameters, syntax follows Little function parameter syntax - multiple parameter declarations, with optional type and optional default values, separated by a comma.

        @param typeName The name of the class. May be nested inside other "packages" using a `.` (for example. my\_pack.MyClass)

        @param fields A string map that has key-value pairs of certain types. Refer to the table above for more information.

    \*\*/

    public function registerType(typeName:String, fields:TypeFields) {

        var instances = memory.externs.createPathFor(memory.externs.instanceProperties, ...typeName.split("."));

        var statics = memory.externs.createPathFor(memory.externs.globalProperties, ...typeName.split("."));

        instances.type = statics.type = memory.getTypeInformation(Little.keywords.TYPE\_MODULE).pointer;

        if (\_\_noTypeCreation) \_\_noTypeCreation = false;

        else if (!memory.externs.externToPointer.exists(typeName) && !memory.constants.hasType(typeName)) {

            memory.externs.externToPointer[typeName] = memory.storage.storeByte(1);

            statics.getter = (\_, \_) -> {

                objectValue: ClassPointer(memory.externs.externToPointer[typeName]),

                objectAddress: memory.externs.externToPointer[typeName]

            }

        } else if (memory.constants.hasType(typeName) && !memory.externs.externToPointer.exists(typeName)) {

            memory.externs.externToPointer[typeName] = memory.constants.getType(typeName);

            statics.getter = (\_, \_) -> {

                objectValue: ClassPointer(memory.externs.externToPointer[typeName]),

                objectAddress: memory.externs.externToPointer[typeName]

            }

        }

        for (key => field in fields) {

            switch key.split(" ") {

                case (\_[0] == "public" && \_.length == 3) => true: {

                    var name = key.split(" ")[2];

                    var type = memory.getTypeInformation(key.split(" ")[1]).pointer;

                    instances.properties[name] = new ExtTree(type, (value, address) -> {

                        // We can't optimize for the two cases outside of the callback, since Haxe doesn't support

                        // type checking on function types.

                        try {

                            var result = untyped field(address, value);

                            if (result is InterpTokens) {

                                return {

                                    objectValue: result,

                                    objectAddress: memory.store(result)

                                };

                            }

                            return {

                                objectValue: untyped result.value,

                                objectAddress: untyped result.address

                            }

                        } catch (e) {

                            return {

                                objectValue: ErrorMessage('External Variable Error: ' + e.details()),

                                objectAddress: memory.constants.ERROR

                            }

                        }

                    });

                }

                case (\_[0] == "public") => true: {

                    var name = key.split(" ")[2];

                    var type = memory.getTypeInformation(key.split(" ")[1]);

                    var params = Interpreter.convert(...Parser.parse(Lexer.lex(key.replaceFirst('public function $name ', "").replaceFirst("(", "").replaceLast(")", ""))));

                    var paramMap = new OrderedMap<String, InterpTokens>();

                    for (entry in params) {

                        if (entry.is(SPLIT\_LINE, SET\_LINE)) continue;

                        switch entry {

                            case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, Identifier(Little.keywords.TYPE\_DYNAMIC));

                            case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, type);

                            case Write(assignees, value): {

                                switch assignees[0] {

                                    case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(value, Identifier(Little.keywords.TYPE\_DYNAMIC));

                                    case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(value, type);

                                    default:

                                }

                            }

                            default:

                        }

                    }

                    instances.properties[name] = new ExtTree(memory.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer, (value, address) -> {

                        var returnType:InterpTokens = type.typeName.asTokenPath();

                        return {

                            objectValue: FunctionCode(paramMap, Block([

                                FunctionReturn(HaxeExtern(() -> {

                                    var result = (field : (MemoryPointer, InterpTokens, Array<InterpTokens>) -> InterpTokens)(address, value, paramMap.keys().toArray().map(key -> Interpreter.evaluate(memory.read(key).objectValue)));

                                    return result;

                                }), returnType)

                            ], returnType)),

                            objectAddress: memory.constants.EXTERN

                        }

                    });

                }

                case (\_[0] == "static" && \_.length == 3) => true: {

                    var name = key.split(" ")[2];

                    var type = memory.getTypeInformation(key.split(" ")[1]).pointer;

                    if (field is StringMap) {

                        \_\_noTypeCreation = true;

                        registerType(typeName + "." + name, field);

                        continue;

                    }

                    statics.properties[name] = new ExtTree(type, (\_, \_) -> {

                        // We can't optimize for the two cases outside of the callback, since Haxe doesn't support

                        // type checking on function types.

                            try {

                                var result = untyped field();

                                if (result is InterpTokens) {

                                    return {

                                        objectValue: result,

                                        objectAddress: memory.store(result)

                                    };

                                }

                                return {

                                    objectValue: untyped result.value,

                                    objectAddress: untyped result.address

                                }

                            } catch (e) {

                                return {

                                    objectValue: ErrorMessage('External Variable Error: ' + e.details()),

                                    objectAddress: memory.constants.ERROR

                                }

                            }

                    });

                }

                case (\_[0] == "static") => true: {

                    var name = key.split(" ")[2];

                    var type = memory.getTypeInformation(key.split(" ")[1]);

                    var params = Interpreter.convert(...Parser.parse(Lexer.lex(key.replaceFirst('static function $name ', "").replaceFirst("(", "").replaceLast(")", ""))));

                    var paramMap = new OrderedMap<String, InterpTokens>();

                    for (entry in params) {

                        if (entry.is(SPLIT\_LINE, SET\_LINE)) continue;

                        switch entry {

                            case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, Identifier(Little.keywords.TYPE\_DYNAMIC));

                            case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, type);

                            case Write(assignees, value): {

                                switch assignees[0] {

                                    case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(value, Identifier(Little.keywords.TYPE\_DYNAMIC));

                                    case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(value, type);

                                    default:

                                }

                            }

                            default:

                        }

                    }

                    statics.properties[name] = new ExtTree(memory.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer, (\_, \_) -> {

                        var returnType:InterpTokens = type.typeName.asTokenPath();

                        return {

                            objectValue: FunctionCode(paramMap, Block([

                                FunctionReturn(HaxeExtern(() -> {

                                    var result = untyped field(paramMap.keys().toArray().map(key -> Interpreter.evaluate(memory.read(key).objectValue)));

                                    return result;

                                }), returnType)

                            ], returnType)),

                            objectAddress: memory.constants.EXTERN

                        }

                    });

                }

                case \_: throw 'Invalid key syntax for `$key`. Must start with either `public`/`static` continue with a type, and end with a variable name. (Example: `public Number myVar`). Each item must be separated by a single whitespace.';

            }

        }

    }

ועכשיו, ההסבר:

#### סוגי זוגות key-value

בגלל הגבלה של Haxe, המפה מספקת 7 אופציות לתכנות שדה במקום 8 (חסר מקרה ספציפי של פונקציה תלוית מקרה שמחזירה גם ערך וגם כתובת.). הפונקציה יודעת באיזה סוג משתמשים בעזרת הקלט מהמפתח. ישנם 4 סוגי קלט המוחלטים ע"י 2 זוגות של הקדמות:

public/static – אם המפתח מתחיל בpublic, השדה יהיה תלוי מקרה. static, השדה יהיה סטטי. כל דבר אחר יזרוק שגיאה.

מכיל/לא מכיל סוגריים – אם המפתח נגמר בסוגריים עם/בלי תוכן, מדובר בפונקציה. אחרת, מדובר במשתנה.

המידע הנוסף ההכרחי לא בהכרח מחליט על סוג השדה:

לאחר public או static, חייב לבוא סוג המשתנה, או סוג ההחזרה של הפונקציה

בתוך הסוגריים, חייבים להיות הכרזות פרמטרים הכתובים ספציפית בLittle. הם מחליטים על כמות וסוגם של הפרמטרים המובאים.

למפתח יש פורמט קבוע:

כל חלק במפתח מופרד ברווח אחד

סוגים ומילות מפתח הינם case sensitive

#### שדות תלויי מקרה

יוצרים עץ אקסטרנים ( ExtTree) חדש על עץ האקסטרנים של הסוג, שלקוח מinstanceProperties:

var instances = memory.externs.createPathFor(memory.externs.instanceProperties, ...typeName.split("."));

בהתאם לסוג השדה המבוקש, נותנים סוג שונה לעץ:

משתנים:

var type = memory.getTypeInformation(key.split(" ")[1]).pointer;

instances.properties[name] = new ExtTree(type, (value, address) -> {

פונקציות:

instances.properties[name] = new ExtTree(memory.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer, (value, address) -> {

מכיוון שאנו נותנים למפתח להחזיר או ערך או זוג של מצביע וערך, צריך להתמודד עם זה בקריאה:

משתנים:

try {

    var result = untyped field(address, value);

    if (result is InterpTokens) {

        return {

            objectValue: result,

            objectAddress: memory.store(result)

        };

    }

    return {

        objectValue: untyped result.value,

        objectAddress: untyped result.address

    }

} catch (e) {

    return {

        objectValue: ErrorMessage('External Variable Error: ' + e.details()),

        objectAddress: memory.constants.ERROR

    }

}

כאן, קוראים לפונקציה שניתנה לנו עם המפתח. אם התוצאה היא ערך, משלימים את המצביע. אחרת, מחזירים את הזוג. במקרה שמשהו לא עובד, זורקים שגיאה ברמת Little.

פונקציות:

var returnType:InterpTokens = type.typeName.asTokenPath();

                        return {

                            objectValue: FunctionCode(paramMap, Block([

                                FunctionReturn(HaxeExtern(() -> {

                                    var result = (field : (MemoryPointer, InterpTokens, Array<InterpTokens>) -> InterpTokens)(address, value, paramMap.keys().toArray().map(key -> Interpreter.evaluate(memory.read(key).objectValue)));

                                    return result;

                                }), returnType)

                            ], returnType)),

                            objectAddress: memory.constants.EXTERN

                        }

מכיוון שאין שני אפשרויות לערך המוחזר, זה יותר פשוט. הערך שאנו מחזירים הוא פונקציה, המקבלת כמות פרמטרים מסוימת ומכילה טוקן החזרה יחיד הקורא לקוד חיצוני להחזיר לו ערך.

#### שדות סטטיים

המקרה כאן דומה מאוד, רק שהפעם אנו עובדים עם החלק הסטטי:

var statics = memory.externs.createPathFor(memory.externs.globalProperties, ...typeName.split("."));

ויש הבדל אחד – מכיוון שבמקרה של משתנים סטטיים אפשר גם להביא פשוט עוד מפה, אנו דואגים לזה, בדרך פשוטה במיוחד:

if (field is StringMap) {

                        \_\_noTypeCreation = true;

                        registerType(typeName + "." + name, field);

                        continue;

                    }

על אף שהשדה שנוצר הוא לא סוג, אפשר פשוט ליצור פרמטר חיצוני, שיבטל יצירת סוג. הפרמטר החיצוני בא לידי ביטוי בכל קריאה לפונקציה הזאת: (קטע קוד זה לקוח מתחילת הפונקציה)

if (\_\_noTypeCreation) \_\_noTypeCreation = false;

else if (!memory.externs.externToPointer.exists(typeName) && !memory.constants.hasType(typeName)) {

    memory.externs.externToPointer[typeName] = memory.storage.storeByte(1);

    statics.getter = (\_, \_) -> {

        objectValue: ClassPointer(memory.externs.externToPointer[typeName]),

        objectAddress: memory.externs.externToPointer[typeName]

    }

} else if (memory.constants.hasType(typeName) && !memory.externs.externToPointer.exists(typeName)) {

    memory.externs.externToPointer[typeName] = memory.constants.getType(typeName);

    statics.getter = (\_, \_) -> {

        objectValue: ClassPointer(memory.externs.externToPointer[typeName]),

        objectAddress: memory.externs.externToPointer[typeName]

    }

}

פשוט מאוד – לפני שמכריזים על אותו עץ כסוג, אנחנו מכניסים תנאי קטן, שאם הוא אמת, מונע יצירת סוג, וגורם לכל מקום אחר בקוד להתייחס לסוג שאנו יוצרים פה, כמו לאובייקט רגיל...

## בדיקות ותוצאות

יפה והכל שיש לנו סוויטת בדיקות רחבה, אבל גם צריך שהיא תעבוד...  
להבא, אפרט על כל הבדיקות, מהראשונה עד לאחרונה, ואפילו קצת על בדיקות שמעבר לסוויטה...

### בדיקה ביחידות (Unit Testing)

בדיקה מס' 1: Basic Math

* + מטרה: לבדוק שחישוב הביטויים המתמטיים של Little עובד, ויודע להמיר לסוג תוצאה אחר בעת הצורך.
  + בפועל: מדפיסים ביטוי מתמטי ערוך המערב סוגריים ואת האופרטורים +, -, \*, /, ^, !.
  + תוצאה: הצלחה
  + בעיות שהתגלו: מעולם לא היו בעיות רציניות מעבר לטעויות לא משמעותיות באלגוריתם החישוב.

בדיקה מס' 2: Variable declaration

* + מטרה: לבדוק שהכרזת משתנים רגילים עובדת, ולבדוק שניתן לאחזר את הערך שלהם במקרה הצורך
  + בפועל: מכריזים על 3 משתנים, אחד עם סוג וערך, אחד רק עם סוג, ואחד בלי סוג ובלי ערך. מדפיסים את ערך המשתנה הראשון, ואת סוגם של כל המשתנים, אחד אחרי השני.
  + תוצאה: הצלחה
  + בעיות שהתגלו: כשה-Referrer היה בפיתוח, קרו שגיאות בהן הReferrer לא שמר/קרא את שמות המשתנים נכון, אז היינו יכולים להכריז על משתנה, ו"לאבד" אותו מיד אחרי.

בדיקה מס' 3: Function declaration

* + מטרה: לבדוק שהכרזת פעולות רגילות עובדת, ושניתן לקרוא להם עם פרמטרים מסוימים עם סוגים קונקרטיים קבועים ע"י המתכנת.
  + בפועל: מוכרזות 3 פונקציות המדפיסות ערכים – אחת ללא פרמטרים, אחת עם פרמטר דינמי אחד, ואחת עם פרמטר מסוג קונקרטי. קוראים לשלושת הפונקציות.
  + תוצאה: הצלחה
  + בעיות שהתגלו: אותו סוג שגיאה של הכרזת משתנים – כשה-Referrer היה בפיתוח, קרו שגיאות שמירה וקריאה.

בדיקה מס' 4: Property access

* + מטרה: לבדוק האם גישה לשדה הנמצא בתוך אובייקט דינמי יכולה לערוך ולאחזר את אותו הערך של אותו שדה
  + בפועל: מכריזים על אובייקט דינמי חדש, ובתוכו שמים שני שדות, שאחד מהם גם אובייקט חדש. גם על האובייקט השני שמים שדה. מדפיסים את כל השדות שאינם אובייקטים.
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות, אלא מודיפיקציות:
    - פעם, קריאות פונקציה היו יכולות להיחשב כשדה שניגשים אליו (PropertyAccess(f, FunctionCall(name))), דבר שהיה חסר הגיון, ולכן שונה. עכשיו זה יסרק כ: FunctionCall(PropertyAccess(f, name)).
    - לפני כן, הפונקציה שהייתה מייצרת PropertyAccessים הייתה הולכת מהסוף להתחלה, ולכן, כאשר הייתה מייצרת טוקנים כאלה שאחד בתוך השני, היה מאוד קשה לקרוא אותם: PropertyAccess(a, PropertyAccess(b, c)). כיוון הסריקה שונה, ועכשיו טוקנים כאלה נכנסים אחד בתוך השני כך: PropertyAccess(PropertyAccess(a, b), c).

בדיקה מס' 5: Loops

* + מטרה: לבדוק האם לולאות for וwhile עובדות כשורה.
  + בפועל: יוצרים משתנה, ומריצים איתו לולאת while שמדפיסה אותו עד שמגיע ל-5. מריצים לאחר מכן לולאת for המדפיסה את הערכים מ0 עד 10 בקפיצות של 3.
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות ספציפיות.

בדיקה מס' 6: Events and conditionals

* + לבדוק האם התנאי if ו"אירועי התנאי" after וwhenever עובדים כשורה.
  + בפועל: יוצרים משתנה, ומיד אחריו בודקים תנאי שחייב להיות שגוי. אחריו, מוספים את שני אירועי התנאי, ומשנים את ערך המשתנה. מצפים שהם יורצו במקרים מסוימים ובסדר מסוים.
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות ספציפיות.

בדיקה מס' 7: Code blocks

* + מטרה: לוודא שניתן לייצר ערכים בעזרת בלוק של קוד המאחזר ערך, ולא רק ערך "inline"
  + מכריזים על משתנה, שערכו הוא בלוק של קוד שאמור להחזיר את המספר 180. את הערך של המשתנה מדפיסים.
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות ספציפיות.

בדיקה מס' 8: Self assignment

* + מטרה: לוודא שניתן לשנות משתנה בעזרת הערך של עצמו. משמש לווידוא שימור קונטקסט במקרי שינוי עצמי.
  + יוצרים משתנה שערכו 1.2, ומשווים אותו לעצמו + 2 \* עצמו בסוגריים, לחלק לעצמו. מדפיסים את הערך הסופי, צריך להיות 3, מסוג מספר עשרוני.
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות ספציפיות.

בדיקה מס' 9: If-Else

* + מטרה: לבדוק ששרשראות if-else עובדות כשורה.
  + בפועל: מתחילים עם תנאי הנפתר לfalse, ועליו לא להיות מורץ. אחריו יש else-if שגם חייב לא להיות מורץ. אחריו יש else ריק, ומכיוון שלא הורץ כלום עד עכשיו, חייב להיות מורץ.
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות ספציפיות.

בדיקה מס' 10: Nested code blocks

* + מטרה: לבדוק שבלוקים של קוד הפועלים בתוך בלוקים אחרים יכולים לעשות הצללה על משתנים עם אותו השם, בבלוק חיצוני יותר (הכוונה – לשנות את הערך של משתנה באופן זמני רק לבלוק הנוכחי, בעזרת הכרזה על משתנה עם אותו שם כמו משתנה בבלוק חיצוני)
  + בפועל: מכריזים על משתנה I שערכו 3. מתחילם 3 בלוקים, אחד בתוך השני. בבלוק הכי פנימי I מוכרז שוב, עם הערך 5, ומודפס מיד לאחר מכן. אחרי הבלוקים, מודפס שוב ערכו של I, שצריך להיות הI ממקודם, שערכו 3.
  + תוצאה: הצלחה
  + בעיות שהתגלו: הייתה פעם שגיאת off-by-one שהייתה קורית כשהיית מנסה ליצור מסגרת חדשה ב-Referrer כאשר לא היו משתנים במסגרת הנוכחית. בגלל גישה שגויה למערך הבתים, הheader החדש היה יכול להיות ממוקם על מקום שכבר קים בו מידע, והיה יכול להתחשב בחלק מהמידע שם כמידע על הheader. זה היה יכול לגרום לכך שהזיכרון היה חושב שלמסגרת שלפני האחת הזאת יש כמות אלמנטים הזויה, וכך הייתה מדלגת אחורה לבתים "שליליים" וקורסת מגישה Out Of Bounds.

בדיקה מס' 11: Inline Blocks

* + מטרה: הרחבה של בדיקה 7, לוודא שניתן גם לאחזר מזהים בעזרת בלוק של קוד, ולא רק ערכים.
  + בפועל: מוכרז משתנה a שערכו בלוק של קוד, שבו מוחזר משתנה אחר שנוצר ומוכפל ב-10. אחר כך, מודפס עוד בלוק של קוד, שבו יש הוספה עצמית של 3 לערך של a, ואחרי ההוספה הוא מאוחזר.
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות ספציפיות

בדיקה מס' 12: Constant pool

* + מטרה: לבדוק שבריכת הקבועים עובדת כשורה ויכולה לאחזר ערכים מתי שצריך במקום לפנות עוד מקום.
  + בפועל: מכריזים על 4 משתנים: שני nullים, 0, ו-0.0. מדפיסים האם שני המשתנים הראשונים משתפים כתובת, והאם השניים האחרונים גם משתפים כתובת. שני הבדיקות אמורות לצאת אמת.
  + תוצאה: הצלחה
  + בעיות שהתגלו: המבנה פשוט, לא היו בעיות ספציפיות.

בדיקה מס' 13: Type Name Property

* + מטרה: לבדוק שהשדה הגלובלי type עובד כשורה, ואכן מדווח את הסוג הנכון לכל סוג ערך.
  + בפועל: מודפסים 6 סוגים, באותה מחרוזת, ללא רווח: הסוג של 5, 5.5, true, nothing (null), האופרטור + ,והסוג Number. צריך להדפיס: " NumberDecimalBooleanAnythingSignType"
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות ספציפיות.

בדיקה מס' 14: Reference vs. Value

* + מטרה: לבדוק האם כשמשחקים עם ערכים שאמורים להיות מועברים כערך או כאזכור, האם מה שמועבר זה המיקום בזיכרון (לאלמנטים מאוזכרים) או הערך המועתק (לאלמנטים המועברים כערך).
  + בפועל: יוצרים משתנה a שערכו אובייקט חדש. יוצרים משתנה חדש b ומשווים אותו ל-a. מדפיסים האם הכתובות שלהם שוות (צריך לצאת אמת). יוצרים משתנה c ומשווים אותו ל502. יוצרים עוד משתנה, d, ומשווים אותו ל-c. מדפיסים האם הכתובות שלהם שוות (צריך לצאת שקר)
  + תוצאה: הצלחה
  + בעיות שהתגלו: כשהוספתי את הבדיקה הזאת, שכחתי לגמרי מזה שצריך להתקין מערכת שתבדיל בין סוגי הערכים. בעקבות הבדיקה, שונתה קצת צורת הגישה לזיכרון, וגם עכשיו יכולה לתמוך בהחזרת מזהים במקום ערכים, והמרתם לערכים בעת הצורך (לדוגמה, אם יש משתנה x שערכו 5, עכשיו אפשר להחזיר את המזהה x, ולא את המספר 5).
* בדיקה מס' 15: Arrays
  + מטרה: לבדוק האם המערכים בספריה הסטנדרטית אכן עובדים כמצופה.
  + בפועל: יוצרים משתנה שערכו מערך. עורכים בו תא אחד ומדפיסים אותו. מדפיסים את אורך המערך, וסוג האלמנט שהוא מכיל
  + תוצאה: הצלחה
  + בעיות שהתגלו: לא היו בעיות ספציפיות.

### בדיקות ידניות

**בעבוע אחזורים מתנאים ולולאות**: כשתכנתי מחדש את הדרך שתנאים מאוחסנים בזיכרון, שמתי לב לoversight יחסית משמעותי: מכיוון שאני מריץ את גוף התנאי/הלולאה באמצעות פונקציית ההרצה הרגילה, אם תמצא בה המילה return במקום כלשהו, במקום שהיא תחזיר ערך מהפונקציה שממנה היא נקראת, היא פשוט תצא מהלולאה – כאילו היא מחזירה את הערך שמייצג בלוק הקוד של הלולאה.   
  
פתרתי את הבעיה בעזרת הוספת פרמטר לפונקציית ההרצה – propagateReturns – כאשר הפרמטר הוא true ואנחנו נתקלים בטוקן מסוג FunctionReturn, במקום להחזיר את ערכו של האחזור, הוא מחזיר ממש את טוקן האחזרה עצמו. ההחזרה מפסיקה את הלולאה, והלולאה נפתרת לטוקן FunctionReturn, אותו פונקציית ההרצה הרגילה (ללא propagateReturns) רואה, מפסיקה את הרצתה, ומחזירה את הערך המתאים.

**קריאת שדות מערכי "Inline"**: כשסיימתי עם הבסיס של מודולת הזיכרון בפרויקט, רציתי לבדוק כמה טוב היא עובדת עם גישה לשדות רגילים, שדות בתוך שדות, וכו'. אחת הבדיקות שביצעתי היא גישה לשדות גלובליים על ערכים שלא שמורים בזיכרון עדיין (לדוגמה: 5.type, 13.address), וגיליתי משהו יחסית חמור – כאשר אנו מנסים לקרוא מהזיכרון בעזרת פונקציית הקריאה הראשית, Memory.read, הפונקציה תמיד מנסה לעקוב אחרי כל מזהה שמובא לה כאילו הוא שם משתנה בפני עצמו, ולכן הפונקציה נכשלת במקרים שהמזהה הראשון הוא לא שם משתנה, אלא פשוט ערך.  
  
לאחר שהבנתי לגמרי את הבעיה, הפתרון היה יחסית פשוט – הכנתי אלטרנטיבה של הפונקציה, ששמה readFrom – הפונקציה הזאת, במקום להתחיל במשתנה, מתחילה במידע על הערך – ערכו, מיקומו בזיכרון, וסוגו. משם, הפונקציה משיגה מידע על שדות הערך, וממשיכה כמו הפונקציה read.

**השוואת משתנים לפונקציות חיצוניות**: כשבדקתי את הפרויקט לאחר כמה תוספות ושינויי, גיליתי עוד oversight משמעותי: מכיוון שהכתובת של פונקציות חיצוניות מעולם לא זמינה, הכתובת של כל פונקציה חיצונית היא כתובת הערך החיצוני הגלובלי – Memory.constants.EXTERN. לכן יוצא, שכשמשווים משתנה רגיל לפונקציה חיצונית, האינטרפרטר חושב שערכו זה קוד של פונקציה השוכן בכתובת של EXTERN. זה רע מאוד, מכיוון שבכתובת הזאת בכלל לא מאוחסנת פונקציה – אחריה עדיין נמצא חלק מבריכת הקבועים, ולכן גם לא ניתן לערוך את המשתנה, וגם אי אפשר לקרוא לפונקציה שהמשתנה מייצג.  
  
הפתרון יחסית "האקי", מכיוון שכחלק מההגבלות של השפה, לא ניתן לאחסן פונקציות "חיות" של Haxe ולהריץ אותן סתם כך, ואחסון הפונקציה מסתמך על שמירת הקוד בזיכרון בצורה כלשהי. לכן, כאשר הזיכרון מזהה ניסיון אחסון של פונקציה אקסטרנית, הוא שומר את ערך המשתנה לא כאותה פונקציה ממש, אלא כפונקציה אחרת, המקבלת את אותם פרמטרים כמו הפונקציה האקסטרנית, ומחזירה טוקן קריאה לאותה פונקציה אקסטרנית עם הפרמטרים הדרושים. זה נראה כך:

FunctionCode(params, Block([

    FunctionReturn(FunctionCall(token, PartArray(forwardedParams)), cell.objectTypeName.asTokenPath())

], cell.objectTypeName.asTokenPath()));

# מדריך למשתמש

.

├── src/

│ ├── js\_example/

│ │ └── JsExample.js.hx

│ ├── little/

│ │ ├── lexer/

│ │ │ ├── Lexer.hx

│ │ │ └── Tokens.hx

│ │ ├── parser/

│ │ │ ├── Parser.hx

│ │ │ └── Tokens.hx

│ │ ├── interpreter/

│ │ │ ├── memory/

│ │ │ │ ├── Memory.hx

│ │ │ │ ├── MemoryPointer.hx

│ │ │ │ ├── ConstantPool.hx

│ │ │ │ ├── ExternalInterfacing.hx

│ │ │ │ ├── HashTables.hx

│ │ │ │ ├── Operators.hx

│ │ │ │ ├── Referrer.hx

│ │ │ │ └── Storage.hx

│ │ │ ├── Interpreter.hx

│ │ │ ├── Runtime.hx

│ │ │ ├── ByteCode.hx

│ │ │ ├── StdOut.hx

│ │ │ └── Tokens.hx

│ │ ├── tools/

│ │ │ ├── Conversion.hx

│ │ │ ├── Extensions.hx

│ │ │ ├── Layer.hx

│ │ │ ├── Plugins.hx

│ │ │ ├── PrepareRun.hx

│ │ │ ├── PrettyPrinter.hx

│ │ │ └── TextTools.hx

│ │ ├── KeywordConfig.hx

│ │ └── Little.hx

│ ├── Main.hx

│ └── UnitTests.eval.hx

## עץ קבצים

├── clients/

│ ├── web/

│ │ ├── index.html

│ │ ├── interp.js

│ │ └── interp.map.js

│ ├── windows/

│ │ └── ---to be compiled---

│ └── linux/

│ └── ---to be compiled---

├── .gitignore

├── compile.hxml

├── demoCode.txt

├── Haxelib.json

├── index.html

├── LICSENSE

└── README.md

## התקנה

### למפתחי תוכנה

לפני שמתחילים, יש לוודא שהשפה Haxe והכלים שלה מותקנות כראוי: יש להתקין את הטרנספיילר של Haxe, ולבדוק ש:

הפקודה Haxe נמצאת בPATH, או לפחות ניתן למצוא את הקובץ ולהפעיל אותו בעזרת המסלול אליו.

הפקודה Haxelib עובדת, אמורה להדפיס מידע על שימוש במנהל הספריות.

מעבר לזה, אני ממליץ:

להוריד Visual Studio Code ולהוריד את ההרחבה Haxe Extension Pack

להוסיף בפרויקט שלהם תקייה ששמה .vscode, ולהוסיף בה את הקובץ tasks.json עם התוכן:

{

    "version": "2.0.0",

    "tasks": [

        {

            "type": "hxml",

            "file": "compile.hxml",

            "problemMatcher": [

                "$Haxe-absolute",

                "$Haxe",

                "$Haxe-error",

                "$Haxe-trace"

            ],

            "group": "build",

            "label": "Haxe: compile.hxml"

        }

    ]

}

עם הקובץ הזה, אפשר לבנות את הפרויקט שלך באמצעות לחיצה על Ctrl+Shift+B

עכשיו כשהכל מותקן כשורה, אפשר להוריד את הפרויקט כספריה באמצעות מנהל הספריות Haxelib, בעזרת הפקודה:  
Haxelib git little https://github.com/ShaharMS/Little/tree/branch/functional-programming

לכלול את הספרייה בפרויקט שלך באמצעות הוספת השורה הזאת בקובץ הקומפילציה (hxml) שלך:

--library little

ועכשיו אפשר להשתמש בקוד!

#### QuickStart

ניתן להריץ קוד בעזרת הפונקציה Little.run:

Little.run("define x = 3, print(x)").

var output = Little.runtime.stdout.output;

להוסיף "קובץ" קוד להרצה באמצעות קריאה לLittle.loadModule לפני ההרצה. כמו כן, לשני הפונקציות יש פרמטר לאפשור debugging. יש לציין שרק כאשר loadModule מורץ במקום פרמטר הdebug שלו משנה:

Little.loadModule("define x = 3", "MyModule");

Little.run("print(x)", true);

var output = Little.runtime.stdout.output;

לקמפל לקוד-בתים באמצעות הפונקציה Little.compile:

var ast = Little.compile("define x = 3, print(x + 6 ^ 3)");

File.saveContent(ByteCode.compile(...ast), "compiled.txt");

ולפרמט קוד באמצעות Little.format:

trace(Little.format("define    x=   3, print (x + 6^3)")); // define x = 3, print(x + 6 ^ 3)

הפונקציה reset משמשת למחיקת כל המידע על ההרצה האחרונה, כדי לתת לך להתכונן להרצה חדשה:

Little.run("define x = 3, print(x)"); // output: 3

Little.reset();

Little.run("print(x)"); // output: error: x is undefined

אפשר לגשת למאזיני אירועים, פונקציות הדפסה בסיסיות ומידע עדכני על ההרצה בעזרת השדה Little.runtime:

Little.runtime.onFieldDeclared.push((name, type) -> {

trace('$type declared: $name');

})

Little.run("define x = 3, print(x)"); // VARIABLE declared: x

לקרוא, לכתוב ולאחסן זיכרון בעזרת Little.memory:

Little.memory.write(["x"], Number(3), "Number");

Little.run("print(x)"); // output: 3

להוסיף אלמנטים אקסטרניים לשפה באמצעות Little.plugin:

Little.plugin.registerFunction("print3", "Idk", [], (params) ->{

    Little.runtime.print("3");

    return NullValue;

}, "Anything");

Little.run("print3()"); // output: 3

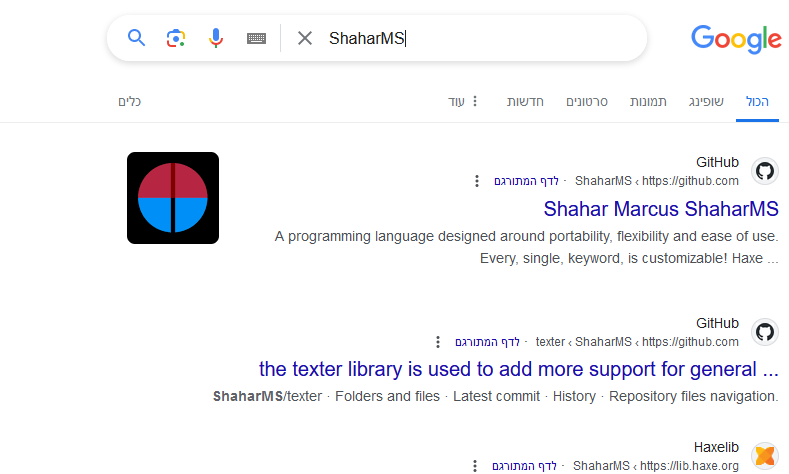
ולראות, לשנות מילים ולהחליף סטים של מילים שמורות בעזרת הפעולות שב-Little.keywords:

Little.keywords.PRINT\_FUNCTION\_NAME = "הדפס";

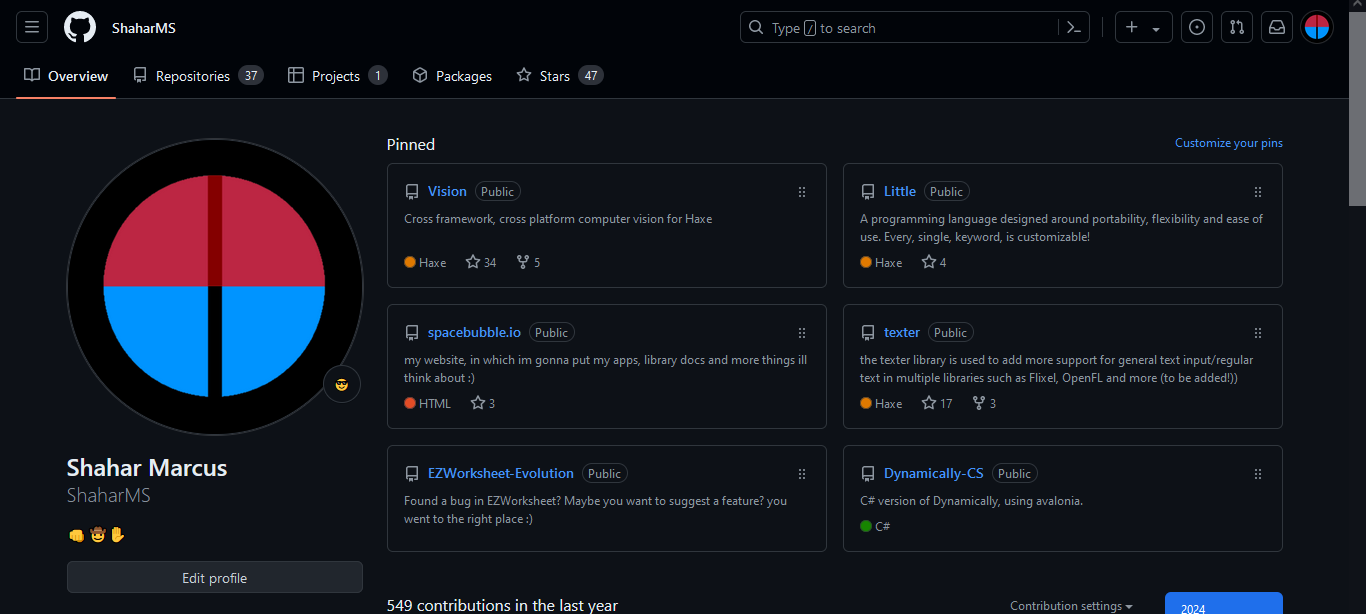
Little.run("הדפס(3)"); // output: 3

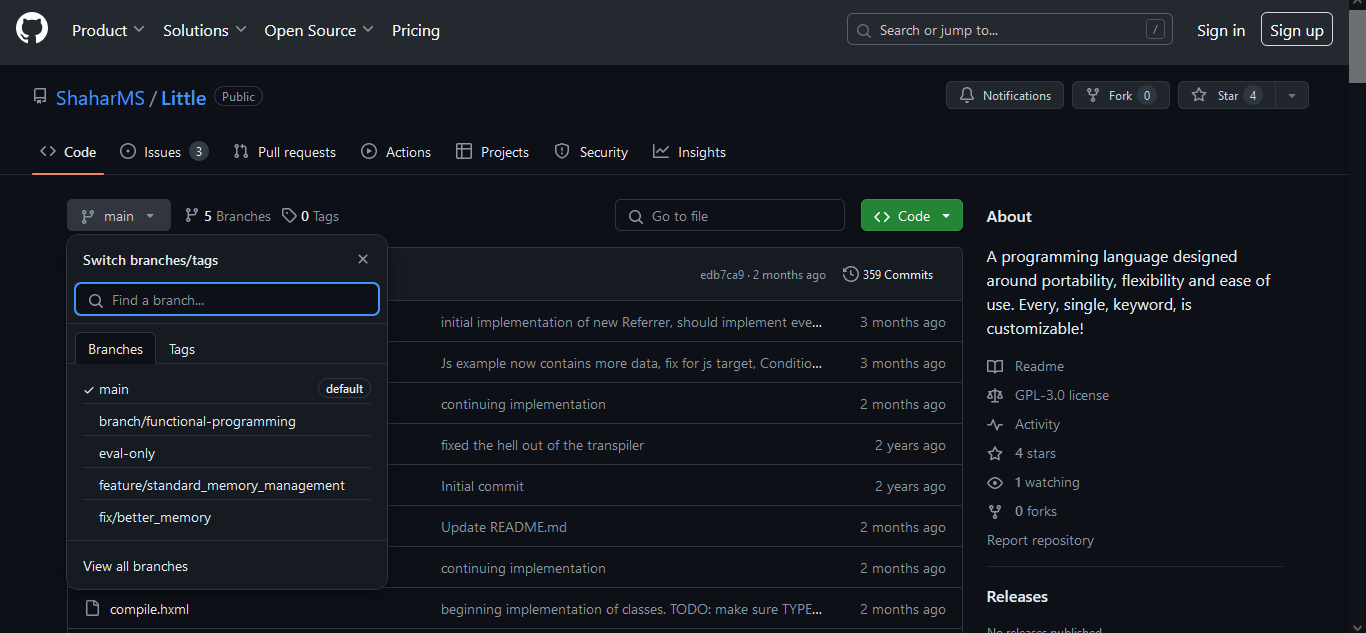
### מתכנתים ב-Little

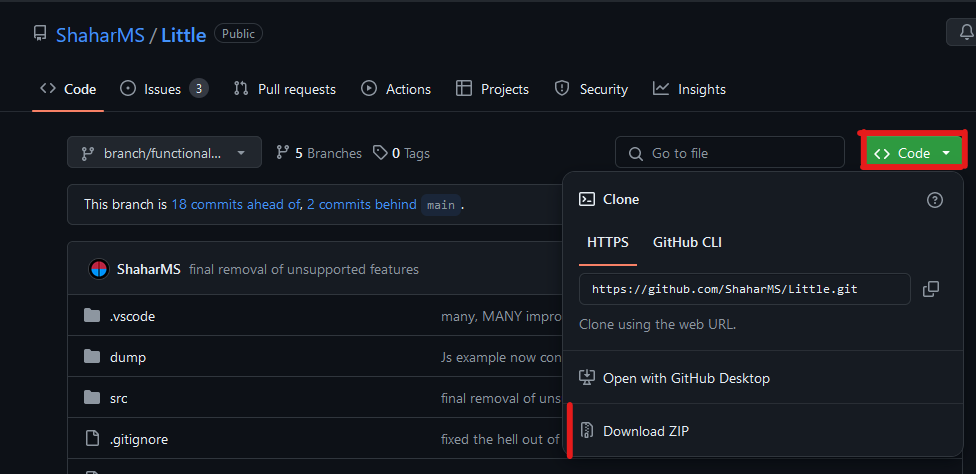
יש לחפש בדפדפן ShaharMS, ולהיכנס לתוצאה הראשונה (כנראה):



לאחר הכניסה ללינק, יוצג המסך הבא. יש ללחוץ על הפרויקט Little ב"חלון" הימני העליון:



לאחר מכן, יוצג החלון של הענף הראשי של הפרויקט. אנו לא מעוניינים בענף הזה, לכן עלינו ללחוץ על הלחצן ולבחור את האופציה :

לאחר מכן, יש ללחוץ על הלחצן ולבחור את האופציה :

לאחר הורדת קובץ הZIP, יש לחלץ אותו (כל תוכנת חילוץ טובה), לגשת אל תוך הפרויקט, ולהיכנס לתיקייה clients. בתקיה יהיו 3 סוגי לקוחות בשלוש תיקיות:

תיקייה web – הלקוח האינטרנטי, זמין גם ב <https://spacebubble.io/little/demo>. מפעילים בעזרת חיצה כפולה על הקובץ index.html, או בפתיחה שלו בעזרת דפדפן כזה או אחר.

תקיה windows – לקוח שורת הפקודה כתוכנה להרצה בwindows.

תקיה linux – לקוח שורת הפקודה כתוכנה להרצה בlinux

#### QuickStart

לקוח אינטרנטי – יש להסתכל בצד הימני של המסך – מעבר למקום להחלפת מילים שמורות, הוא גם מכיל דוגמאות קוד שאפשר ואמורים ללמוד מהם.

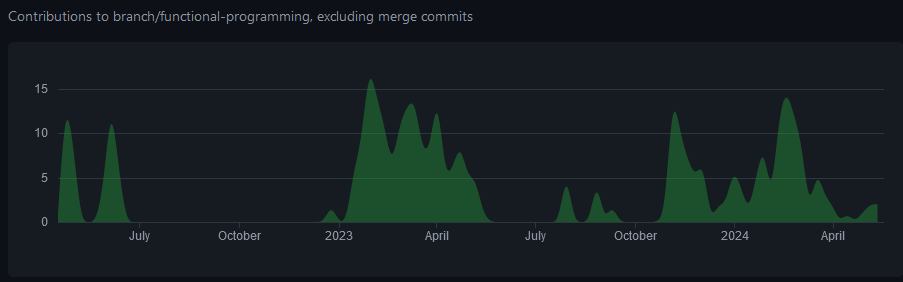
לקוח שורת פקודה – יש להשתמש בפקודה printSample! כמו שהוזכר תחת הכותרת [סוג לקוח: לומד תכנות, מתכנת Little](#_סוג_לקוח:_לומד)

# רפלקציה

## עבודה על הפרויקט

לצערי ולשמחתי, פרויקטים כאלה מעולם אינם נגמרים – אנחנו אולי יכולים להציב אבני דרך, אך אי אפשר "להשלים" פרויקטים כאלה.

אז הייתי חייב להחליט על אבן דרך "סופית". לגרסת הפרויקט שאני מגיש, הצבתי את אבן הדרך הסופית בתכנות פונקציונאלי. הצבתי את אבן הדרך הסופית שם משתי סיבות:  
הראשונה אולי ברורה, והיא שצריך לסיים עם הפרויקט מתישהו, אך מאחורי השנייה יש מחשבה – תוך כדי התכנות של הפרויקט, שמתי לב שכשאני "חופר" יותר אל תוך נושאים מסוימים, הכרזת מחלקות לדוגמה, אני מבזבז הרבה מאוד זמן על תכנות דברים שאני גם כבר יודע איך עושים, וגם כוללים הרבה עבודה שחורה.

בקלות אפשר לראות שהפרויקט הזה היה מסע, שכלל עליות ומורדות, התלהבויות, הפסקות עבודה, ולפעמים אפילו Burnout:

(בתמונה: גרף תדירות commits לפי זמן, מאפריל 2022 עד מאי 2024)

מהפרויקט הזה למדתי על הרבה דברים, אך לא ישירות, ולכן, אני לא זוכר שאי פעם הייתה לי בעיה משמעותית עם תכנות של חלק מהפרויקט – כל פעם שהגעתי למשהו קשה, כבר למדתי הרבה מאוד, והייתי יכול להשתמש בניסיון ובידע שלי כדי לפתור אותו. לכן אפשר גם לראות, שרוב האלגוריתמים שהשתמשתי בהם בפרויקט הם גרסאות "השראה" של אלגוריתמים אחרים – לא הייתי צריך להיצמד וללמוד ספציפית אך ורק ממקום אחד – כל פעם שנתקלתי בבעיה כזו או אחרת, לקחתי הרבה ידע מהרבה מקומות, ועיצבתי פתרון שבסוף עבד מצוין.

בגלל המודל שבעזרתו למדתי, יחסית קל לי להעיד על נושאים שלמדתי עליהם מהפרויקט:

Hashing

ניהול זיכרון, Stack/Heap

ByteCode וקוד מכונה

עצי syntax

שיטות עיבוד מידע בסדר גודל גדול/עצום

ומעבר לתחומים לימודיים, התחלתי להרגיש מאוד בנוח עם מגוון "דרכי תכנות", ובמיוחד רקורסיה, שכן רוב הפרויקט – מפונקציות, מחלקות ועד אובייקטים וADTים, כתוב מאילוץ בצורה רקורסיבית...

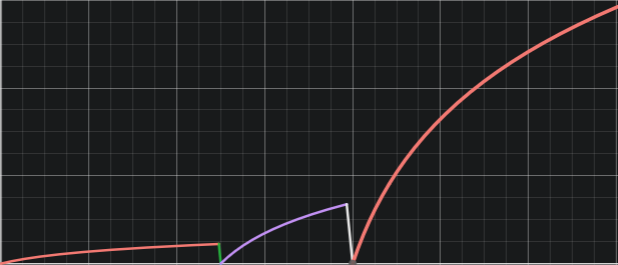
## תובנות

מתכנון ומתהליך "בניית" הפרויקט רכשתי מגוון כלים ולמדתי הרבה "לקחים" לעתיד:

**חשיבות תכנון הפרויקט**: תוך כדי בניית הפרויקט, "גיליתי" 3 סיבות ב3 מילים למה תכנון המבנה של הפרויקט זה דבר כל-כך חשוב:

* + **Scalability** – ככל שפרויקט נהיה רחב וגדול יותר, מתפתח דבר שקוראים לו "Software Rot" – יותר מדי חלקים שלא עובדים הכי טוב אחד עם השני, ולא בנויים טוב אחד על השני, מתחילים בסוף לזייף יותר ויותר, עד שאנחנו מגיעים למצב ששום דבר לא עובד כמצופה, וגם אין דרך חזור. מבנה פרויקט טוב מאט מאוד את היווצרות ה"עובש", ולכן התכנון כזה חשוב.
  + **Burnout** – כשהפרויקט בנוי כך שכל החלקי בבירור מפוצלים לחלקים שונים, שמשתמשים אך לא מסתמכים אחד על השני, אפשר לקפוץ מתכנות חלק אחד לחלק אחר בפרויקט, וההתקדמות גם נשארת קרובה לליניארית, וגם נמאס לך הרבה, *הרבה* יותר לאט...
  + **Learnability** – אחד מהדברים הכי מבאסים כשמפתח מחפש כלי להשתמש בו, או ללמוד ממנו. זה שהוא לא מצליח בכלל להבין את הכלי – איך כל דבר משתמש בכל דבר אחר, למה הרכיב הזה ספציפית חיוני, וכו'. לכן, על מנת שכולם יוכלו לפתח יותר טוב, אנחנו נרצה לפתח קוד טוב יותר, שמייצג דוגמה טובה יותר, וקל ללמוד גם ממנו, וגם לאמץ את המבנה שלו...

**אסור לפחד מנושאים גדולים**: כשנושא נראה ענק, זה בדרך כלל אשליה – בסופו של יום, כל נושא מורכב מהרבה מאוד חלקים, שחלק חיוני לדעת, וחלק פחות. דבר חשוב שלמדתי במהלך הפרויקט, ואפילו בכלליות במהלך המסע שלי כמתכנת, זה שלכל שלב נתון, צריך לדעת כמות יחסית קטנה של מידע מאותו נושא, ואולי עם הזמן צריך להרחיב אותה, אבל זה לעולם לא באופן מידי – זה לא נכון להסתכל על נושא כמונולית', אלא כערמת אבנים...

**מותר לטעות, ואפילו בגדול**: יצא לי לעשות refactoring שלוש פעמים במהלך תכנות הפרויקט, וברטרוספקט, אני שמח על כל פעם שמחקתי את כל הקוד וכתבתי אותו מחדש. באף אחת מהפעמים שעשיתי זאת לא הפסדתי התקדמות: בסוף, יוצא שאדם מתכנת משהו, במשך אפילו חצי שנה, מבין שהדבר לא טוב, מתכנת את זה טוב יותר, ומדביק את הפער ואפילו יותר תוך שבועיים. אצרף גרף לזה:  
(בגרף: כל חלק עליה זה התקדמות בפרויקט, כל ירידה היא תכנות מחדש/refactor)  


## הסתכלות

### אחורנית

הפרויקט הזה פותח במשך שנתיים, בהם למדתי המון. ברור שהיו דברים שהייתי מיישם אחרת:

אופרטורים – גם בהסתכלות אחורה, אני לא 100% בטוח למה פיצלתי בין פעולות וערכים רגילים, לאופרטורים. אין בזה שום דבר טוב. האימפלמנטציה לא מאפשרת לי דינמית להכריז על משתנים, ובמיוחד שלא לאחזר אותן לפי הצדדים שהם מבקשים. פשוט, לא משהו...

אחסון פונקציות חיצוניות – אם הייתי יודע יותר, הייתי גם משקיע יותר בפתרון לשמירת פעולות אקסטרניות בזיכרון, ולא הייתי חייב להסתבך חודש שלם על בעיות שקשורות לאימפלמנטציה של זה.

### וקדימה

אם היו לי יותר משאבים ויותר זמן, הייתי מוסיף ומשנה יחסית הרבה דברים:

מחיקה שלמה ש מערכת האופרטורים, ושילובה עם המערכת הרגילה והאקסטרנית

מוסיף הכרזה על סוגים, אופרטורים, ותנאים ולולאות

הוספת זיהוי אופרטורים רב-סימניים גם כפיצ'ר של זמן ריצה, על מנת לאפשר הכרזה דינמית לגמרי של אופרטורים

הרחבה משמעותית של הספרייה הסטנדרטית שהשפה מספקת

מוסיף קומפיילר לקוד מכונה, ולא רק לקוד-בתים ספציפי

משפר את התמיכה בדוקומנטציה ברחבי השפה, ומאפשר גישה דינמית אליהם

## תודות

קודם כל, לאח שלי גיא, שהיה מוכן לעזור לי בכל נושא, ואפילו ממש לקבוע פגישות זום כדי ללמד. תודה ענקית!

מעבר לו, לא קיבלתי עזרה מיוחדת מאדם ספציפי, אך כן מאוד נעזרתי בשלושה "מקורות":

YouTube, במיוחד ערוצים כמו Low Level Learning

Haxe Discord Server, מלא אנשים שיודעים מלא דברים, ובאופן מפתיע, מאוד ידידותיים ומוכנים תמיד לעזור

וכמובן, ויקיפדיה – באופן אירוני במיוחד, זה היה מקור המידע האולטימטיבי לאורך כל המסע הזה – ומה אני מופתע – הרבה אנשים שהרבה יותר חכמים ממני עבדו בצורה הרבה יותר מבוקרת מהרגיל על "מאמרים" שהם סוג של מחקר מדעי - הם חייבים להיות טובים...

# ביבליוגרפיה

ניהול זיכרון - כותבי ויקיפדיה, ללא תאריך. [https://en.wikipedia.org/wiki/Memory\_management#STACK](https://en.wikipedia.org/wiki/Memory_management" \l "STACK" \t "_new)

SipHash - כותבי ויקיפדיה, ללא תאריך. <https://en.wikipedia.org/wiki/SipHash>

הקצאת זיכרון מבוססת מחסנית - כותבי ויקיפדיה, <https://en.wikipedia.org/wiki/Stack-based_memory_allocation>

ערימה - כותבי ויקיפדיה, ללא תאריך. [https://en.wikipedia.org/wiki/Heap](https://en.wikipedia.org/wiki/Heap" \t "_new)

איסוף זבל (מדעי המחשב) - כותבי ויקיפדיה, ללא תאריך. [https://en.m.wikipedia.org/wiki/Garbage\_collection\_(computer\_science)](https://en.m.wikipedia.org/wiki/Garbage_collection_(computer_science)" \t "_new)

קוד בריחה של ANSI - כותבי ויקיפדיה, ללא תאריך. [https://en.wikipedia.org/wiki/ANSI\_escape\_code](https://en.wikipedia.org/wiki/ANSI_escape_code" \t "_new)

אלגוריתם Shunting Yard - כותבי ויקיפדיה, ללא תאריך. [https://en.wikipedia.org/wiki/Shunting\_yard\_algorithm](https://en.wikipedia.org/wiki/Shunting_yard_algorithm" \t "_new)

חישוב ביטויים מבוסס סמלים - כותבי ויקיפדיה, ללא תאריך. [https://en.wikipedia.org/wiki/Symbolic\_method](https://en.wikipedia.org/wiki/Symbolic_method" \t "_new)

חישוב סימבולי-נומרי - כותבי ויקיפדיה, ללא תאריך. [https://en.wikipedia.org/wiki/Symbolic-numeric\_computation](https://en.wikipedia.org/wiki/Symbolic-numeric_computation" \t "_new)

- MurmurHash כותבי ויקיפדיה, ללא תאריך. <https://en.wikipedia.org/wiki/MurmurHash>

ניהול זיכרון בשיטת Buddy-Blocks - כותבי ויקיפדיה, ללא תאריך. [https://en.wikipedia.org/wiki/Buddy\_memory\_allocation](https://en.wikipedia.org/wiki/Buddy_memory_allocation" \t "_new)

Visual Basic for Applications - כותבי ויקיפדיה, <https://en.wikipedia.org/wiki/Visual_Basic_for_Applications>

Heap דו-הורי - כותבי ויקיפדיה, ללא תאריך. <https://en.wikipedia.org/wiki/Beap>

SHA-1 - כותבי ויקיפדיה, ללא תאריך. <https://en.wikipedia.org/wiki/SHA-1>

- SHA-2 כותבי ויקיפדיה, ללא תאריך. <https://en.wikipedia.org/wiki/SHA-2>

- CityHash כותבי ויקיפדיה, ללא תאריך. <https://en.wikipedia.org/?title=CityHash&redirect=no>

F# (שפת תכנות) - כותבי ויקיפדיה, ללא תאריך. <https://en.wikipedia.org/wiki/F_Sharp_(programming_language)>

# נספחים

## הקדמה קצרצרה

לפני שנתחיל עם הקוד, בגלל האורך שלו, גם לגובה וגם לרוחב, אני ממליץ להעביר את word למצב עמוד אינטרנטי, באמצעות לחיצה על שנמצא בצד ימין/שמאל למטה של חלון הוורד, תלוי בשפה של וורד.  
  
עוד יש לציין, שבגלל אורך הקבצים תהיה התעלמות מוחלטת מהפיצול לעמודים. עדיין, למען שימור הפורמט עד עכשיו, הקוד יתחיל בעמוד הבא.

אם הפורמט בספר לא נוח, אני ממליץ להסתכל על הקוד בעמוד הגיטהאב של הפרויקט,   
<https://github.com/ShaharMS/Little/tree/branch/functional-programming>  
הוא פחות לאגי, מציג את מבנה התקייות ויודע לקשר בין מקומות בקוד.

Syntax highlighting נעשה באמצעות Visual Studio Code.

## קוד הפרויקט

package;

import little.KeywordConfig;

import little.interpreter.ByteCode;

import little.interpreter.memory.Storage;

import haxe.Resource;

import little.interpreter.memory.Memory;

import little.tools.PrepareRun;

import little.tools.PrettyOutput;

import haxe.Log;

import vision.tools.MathTools;

import haxe.io.Path;

import little.tools.Plugins;

import little.tools.Conversion;

import little.Little;

import little.interpreter.Runtime;

import little.tools.PrettyPrinter;

import little.interpreter.Interpreter;

#if js

import js.Browser;

import js\_example.JsExample;

#elseif sys

import sys.FileSystem;

import sys.io.File;

import haxe.SysTools;

#end

import little.parser.Parser;

import little.lexer.Lexer;

using StringTools;

using little.tools.TextTools;

class Main {

    /\*\*

        The main function - the entry point of the program

    \*\*/

    static function main() {

        #if (sys && special)

        var file = File.getContent("src/little/interpreter/Runtime.hx");

        var vars = ~/public(?: inline){0,1} var (.+?)(?:;$|; | =)/m;

        var matches = ~/public(?: inline){0,1} function (.+?) \{$/m;

        var fields = [];

        while (vars.match(file)) {

            fields.push(vars.matched(1));

            file = file.replaceFirst(vars.matched(0), "");

            Sys.println(vars.matched(1));

        }

        while (matches.match(file)) {

            fields.push(matches.matched(1));

            file = file.replaceFirst(matches.matched(0), "");

            Sys.println(matches.matched(1));

        }

        #elseif js

        new JsExample();

        #elseif unit

        UnitTests.run(true);

        #elseif formatting

        File.saveContent("formatted.txt", Little.format(Resource.getString("sample")));

        #elseif sys

        trace(Type.getInstanceFields(KeywordConfig).length);

        try {

            var preDefInput:String = null;

            Sys.print("Little Interpreter v"

                + Little.version

                + "\n\n"

                +

                "Type \"ml!\" for multi-line mode, \"default!\" for single-line mode, or \"ast!\" for abstract syntax tree mode.\nPress `Ctrl`+`C` to exit at any time.\n\n");

            Sys.print("If You're new to the language, type \"printSample!\" to print a file of sample code.\n\n");

            Sys.println("-------------SINGLE-LINE MODE--------------\n");

            while (true) {

                if (preDefInput == null)

                    Sys.print("  >> ");

                var input = preDefInput ?? Sys.stdin().readLine();

                if (input == "ml!") {

                    Sys.command("cls");

                    Sys.print("--------------MULTI-LINE MODE--------------\n");

                    Sys.println("Commands:\n\t- \"run!\": runs the code\n\t- \"clear!\": resets the code\n\t- \"clearLine!\": deletes the last line\n");

                    Sys.println("Command \"printSample!\" is temporarily disabled. return to single-line or ast mode to use it again\n");

package;

#if sys

import little.interpreter.Interpreter;

import little.interpreter.Tokens.InterpTokens;

import sys.io.File;

import sys.FileSystem;

import little.tools.PrettyPrinter;

import eval.luv.Stream;

import little.Little;

import little.interpreter.Runtime;

import little.lexer.Lexer;

import little.parser.Parser;

import little.tools.Layer;

using StringTools;

using little.tools.TextTools;

using little.tools.Extensions;

typedef UnitTestResult = {

    testName:String,

    success:Bool,

    returned:InterpTokens,

    expected:InterpTokens,

    code:String

}

class UnitTests {

    // ANSI colors

    static var RED = "\033[31m";

    static var GREEN = "\033[32m";

    static var YELLOW = "\033[33m";

    static var BLUE = "\033[34m";

    static var MAGENTA = "\033[35m";

    static var CYAN = "\033[36m";

    static var WHITE = "\033[37m";

    static var RESET = "\033[0m";

    static var BOLD = "\033[1m";

    static var ITALIC = "\033[3m";

    static var UNDERLINE = "\033[4m";

    public static function run(bulk:Bool = false) {

        var testFunctions = [

            test1, test2, test3, test4, test5, test6, test7, test8, test9, test10, test11, test12, test13, test14, test15

        ];

        var allSuccessful = true;

        var unsuccessful = 0;

        var i = 1;

        for (func in testFunctions) {

            var result = func();

            Sys.println('$CYAN$BOLD Unit Test $i:$RESET $BOLD$ITALIC${if (result.success) GREEN else RED}${result.testName}$RESET');

            Sys.println('    - $RESET$BOLD$WHITE Result: $ITALIC${if (result.success) GREEN else RED}${if (result.success) "Success" else "Failure"}$RESET');

            if (!result.success) {

                Sys.println('        - $RESET$BOLD$WHITE Expected:$RESET $ITALIC$GREEN${result.expected}$RESET');

                Sys.println('        - $RESET$BOLD$WHITE Returned:$RESET $ITALIC$RED${result.returned}$RESET');

                Sys.print('        - $RESET$BOLD$WHITE Code:$RESET \n            ${result.code.replace("\n", "\n            ")}$RESET\n');

                Sys.print('        - $RESET$BOLD$WHITE Abstract Syntax Tree:$RESET\n            ${PrettyPrinter.printParserAst(Parser.parse(Lexer.lex(result.code))).replace("\n", "\n            ")}$RESET\n');

                Sys.print('        - $RESET$BOLD$WHITE Stdout:$RESET\n            ${Little.runtime.stdout.output.replace("\n", "\n            ")}$RESET\n');

            }

            if (!result.success) {

                allSuccessful = false;

                unsuccessful++;

            }

            if (!result.success && !bulk)

                Sys.exit(1);

            Sys.sleep(bulk ? 0.02 : 0.2);

            Little.reset();

            i++;

        }

        if (allSuccessful) {

            Sys.println('$GREEN$BOLD🥳 🥳 🥳 All tests passed! 🥳 🥳 🥳$RESET');

        } else if (unsuccessful < testFunctions.length / 2) {

            Sys.println('$YELLOW$BOLD⚠️  ⚠️  ⚠️  $unsuccessful out of ${testFunctions.length} tests failed! ⚠️  ⚠️  ⚠️$RESET');

        } else if (unsuccessful >= testFunctions.length / 2) {

            Sys.println('$RED$BOLD❌ ❌ ❌ $unsuccessful out of ${testFunctions.length} tests failed! ❌ ❌ ❌$RESET');

        }

    }

    public static function test1():UnitTestResult {

        var code = "print((5 + (3 - 2)) \* 5^2 + 3 \* 4 / 4 + 4! + 8 + -2)";

        Little.run(code);

        var result = Little.runtime.stdout.stdoutTokens.pop();

        return {

            testName: "Basic Math",

            success: result.equals(Decimal(183)),

            returned: result,

            expected: Decimal(183),

            code: code

        }

    }

    public static function test2():UnitTestResult {

        var code = 'define x as Number = 3, define y as Decimal, define z\nprint(x + ", " + y.type.toCharacters() + ", " + z.type.toCharacters() + ", " + z)';

        Little.run(code);

        var result = Little.runtime.stdout.stdoutTokens.pop();

        return {

            testName: "Variable declaration",

            success: result.equals(Characters("3, Decimal, Anything, nothing")),

            returned: result,

            expected: Characters("3, Decimal, Anything, nothing"),

            code: code

        }

    }

    public static function test3():UnitTestResult {

        var code = "action x1() = { print(1) }\naction x2(define x as Number) = { print(x) }\naction x21(define x as Number) = { return x }\naction x3() = { print(1 + x21(5)) }\n\nx1(), x2(5), x3()";

        Little.run(code);

        var result = PartArray(Little.runtime.stdout.stdoutTokens);

        var exp = PartArray([Number(1), Number(5), Number(6)]);

        return {

            testName: "Function declaration",

            success: !Lambda.has([for (i in 0...3) Type.enumEq(result.parameter(0)[i], exp.parameter(0)[i])], false),

            returned: result,

            expected: Characters("1, 5, 6"),

            code: code

        }

    }

    public static function test4():UnitTestResult {

        var code = "define x = Object.create(), define x.value as Number = 3\ndefine x.y = Object.create(), define x.y.value = 5\ndefine x.y.z = x.y.value\nprint(x.y.z + x.y.value + x.value)";

        Little.run(code);

        var result = Little.runtime.stdout.stdoutTokens.pop();

        return {

            testName: "Property access",

            success: result.equals(Number(13)),

            returned: result,

            expected: Number(13),

            code: code

        };

    }

    public static function test5():UnitTestResult {

        var code = "define i = 0\nwhile (i <= 5) { print (i); i = i + 1}\nfor (define j from 0 to 10 jump 3) print(j)";

        Little.run(code);

        var result = PartArray(Little.runtime.stdout.stdoutTokens);

        var exp = PartArray([

            Number(0), Number(1), Number(2), Number(3), Number(4), Number(5), Number(0), Number(3), Number(6), Number(9)]);

        return {

            testName: "Loops",

            success: !Lambda.has([for (i in 0...10) Type.enumEq(exp.parameter(0)[i], result.parameter(0)[i])], false),

            returned: result,

            expected: exp,

            code: code

        };

    }

    public static function test6():UnitTestResult {

        var code = 'define i = 4, if (i != 0) print(true)\nafter (i == 6) print("i is 6"), whenever (i == i) print("i has changed")\ni = i + 1, i = i + 1';

        Little.run(code);

        var result = PartArray(Little.runtime.stdout.stdoutTokens);

        var exp = PartArray([

            TrueValue,

            Characters("i has changed"),

            Characters("i is 6"),

            Characters("i has changed")

        ]);

        return {

            testName: "Events and conditionals",

            success: !Lambda.has([for (i in 0...4) Type.enumEq(exp.parameter(0)[i], result.parameter(0)[i])], false),

            returned: result,

            expected: exp,

            code: code

        }

    }

    public static function test7():UnitTestResult {

        var code = "define x = {define y = 0; y = y + 5; (6^2 \* y)}, print(x)";

        Little.run(code);

        var result = Little.runtime.stdout.stdoutTokens.pop();

        return {

            testName: "Code blocks",

            success: result.equals(Number(180)),

            returned: result,

            expected: Number(180),

            code: code

        };

    }

    public static function test8():UnitTestResult {

        var code = '\ndefine x = 1.2\nx = (x + 2 \* x) / x\nprint(x)';

        Little.run(code);

        var result = Interpreter.evaluate(Little.runtime.stdout.stdoutTokens.pop());

        return {

            testName: "Self assignment",

            success: result.equals(Decimal(3)),

            returned: result,

            expected: Decimal(3),

            code: code

        }

    }

    public static function test9():UnitTestResult {

        var code = 'if (false) print("Wrong") else if (false && true) print("Also Wrong") else { print("Right") }';

        Little.run(code);

        var result = Little.runtime.stdout.stdoutTokens.pop();

        return {

            testName: "If-Else",

            success: result.equals(Characters("Right")),

            returned: result,

            expected: Characters("Right"),

            code: code

        }

    }

    public static function test10():UnitTestResult {

        var code = 'define i = 3\n{{{define i = 5, print(i)}}}\nprint(i)';

        Little.run(code);

        var result = PartArray(Little.runtime.stdout.stdoutTokens);

        var exp = PartArray([Number(5), Number(3)]);

        return {

            testName: "Nested code blocks",

            success: !Lambda.has([for (i in 0...2) Type.enumEq(exp.parameter(0)[i], result.parameter(0)[i])], false),

            returned: result,

            expected: exp,

            code: code

        }

    }

    public static function test11():UnitTestResult {

        var code = 'define a = {define b = 3, (b \* 10)}, print({a = a + 3, a})';

        Little.run(code);

        var result = Little.runtime.stdout.stdoutTokens.pop();

        return {

            testName: "Inline Blocks",

            success: result.equals(Number(33)),

            returned: result,

            expected: Number(33),

            code: code

        }

    }

    public static function test12():UnitTestResult {

        var code = 'define a = nothing, define b = nothing, define c = 0, define d = 0.0, print(a.address == b.address), print(c.address == d.address)';

        Little.run(code);

        var result = PartArray(Little.runtime.stdout.stdoutTokens);

        var exp = PartArray([TrueValue, TrueValue]);

        return {

            testName: "Constant pool",

            success: !Lambda.has([for (i in 0...2) Type.enumEq(exp.parameter(0)[i], result.parameter(0)[i])], false),

            returned: result,

            expected: exp,

            code: code

        }

    }

    public static function test13():UnitTestResult {

        var code = 'print(5.type.toCharacters() + 5.5.type.toCharacters() + true.type.toCharacters() + nothing.type.toCharacters() + +.type.toCharacters() + Number.type.toCharacters())';

        Little.run(code);

        var result = Little.runtime.stdout.stdoutTokens.pop();

        var exp = Characters("NumberDecimalBooleanAnythingSignType");

        return {

            testName: "Type Name Property",

            success: result.equals(exp),

            returned: result,

            expected: exp,

            code: code

        }

    }

    public static function test14():UnitTestResult {

        var code = 'define a = Object.create(), define b = a, print(a.address == b.address)\ndefine c = 502, define d = c, print(c.address == d.address)';

        Little.run(code);

        var result = PartArray(Little.runtime.stdout.stdoutTokens.slice(0, 2));

        var exp = PartArray([TrueValue, FalseValue]);

        return {

            testName: "Reference vs. Value",

            success: !Lambda.has([for (i in 0...2) Type.enumEq(exp.parameter(0)[i], result.parameter(0)[i])], false),

            returned: result,

            expected: exp,

            code: code

        }

    }

    public static function test15():UnitTestResult {

        var code = 'define a = Array.create(Number, 10), a.set(5, 172482), print(a.get(5)), print(a.elementType), print(a.length)';

        Little.run(code);

        var result = PartArray(Little.runtime.stdout.stdoutTokens);

        return {

            testName: "Arrays",

            success: !Lambda.has([for (i in 0...3) Type.enumEq(result.parameter(0)[i], result.parameter(0)[i])], false),

            returned: result,

            expected: PartArray([Number(172482), ClassPointer(Little.memory.constants.INT), Number(10)]),

            code: code

        }

    }

}

#end

                    var code = "";

                    while (true) {

                        Sys.print("  >> ");

                        var input = Sys.stdin().readLine();

                        if (input == "run!") {

                            Little.run(code, true);

                            trace(PrettyPrinter.printInterpreterAst(Interpreter.convert(...Parser.parse(Lexer.lex(code)))));

                            trace(Little.runtime.stdout.output);

                            Little.reset();

                            Sys.print(code.replaceFirst("\n", "  >> ").replace("\n", "\n  >> ") + "\n");

                        } else if (input == "default!") {

                            Sys.command("cls");

                            Sys.println("-------------SINGLE-LINE MODE--------------\n");

                            break;

                        } else if (input == "ast!") {

                            Sys.command("cls");

                            preDefInput = "ast!";

                            break;

                        } else if (input == "clear!") {

                            code = "";

                            Sys.command("cls");

                            Sys.println("--------------MULTI-LINE MODE--------------\n");

                        } else if (input == "clearLine!") {

                            Sys.command("cls");

                            Sys.println("--------------MULTI-LINE MODE--------------\n");

                            Sys.print(code.replaceFirst("\n", "  >> ").replace("\n", "\n  >> ") + "\n");

                            code = code.split("\n").slice(0, -1).join("\n");

                        } else {

                            code += "\n" + input;

                        }

                    }

                    if (preDefInput == "ast!")

                        continue; // A little hacky, i don't mind though

                } else if (input == "ast!") {

                    Sys.println('${preDefInput == "ast!" ? "" : "\n"}---------ABSTRACT SYNTAX TREE MODE---------\n');

                    while (true) {

                        Sys.print("  >> ");

                        var input = Sys.stdin().readLine();

                        if (input == "default!") {

                            Sys.println("\n-------------SINGLE-LINE MODE--------------\n");

                            break;

                        } else if (input == "printSample!") {

                            Sys.println("\n---------------SAMPLE CODE-----------------\n");

                            Sys.println(Resource.getString("sample"));

                            Sys.println("\n---------ABSTRACT SYNTAX TREE MODE---------\n");

                            continue;

                        }

                        try {

                            Sys.println(PrettyPrinter.printInterpreterAst(Interpreter.convert(...Parser.parse(Lexer.lex(input)))));

                        } catch (e) {}

                    }

                } else if (input == "printSample!") {

                    Sys.println("\n---------------SAMPLE CODE-----------------\n");

                    Sys.println(Resource.getString("sample"));

                    Sys.println("\n-------------SINGLE-LINE MODE--------------\n");

                } else {

                    Little.run(input, true);

                    // trace(PrettyPrinter.printInterpreterAst(Interpreter.convert(...Parser.parse(Lexer.lex(input)))));

                    trace(Little.runtime.stdout.output);

                    Little.reset();

                }

                preDefInput = null;

            }

        } catch (e) {

            // This jsut means Ctrl + c, or, exiting the program

            Sys.println("\n\nExiting...");

            Sys.exit(0);

        }

        #end

    }

}

package js\_example;

import js.html.Event;

import little.KeywordConfig;

import js.html.SpanElement;

import js.html.TableColElement;

import js.html.Node;

import js.Syntax;

import haxe.display.Display.Keyword;

import js.html.TableRowElement;

import js.html.TableElement;

import js.html.TextAreaElement;

import js.html.Document;

import little.interpreter.Runtime;

import little.Little;

import js.Browser;

using js\_example.JsExample;

using StringTools;

class JsExample {

    var d = Browser.document;

    public function new() {

        var input:TextAreaElement = cast d.getElementById("input");

        var ast:TextAreaElement   = cast d.getElementById("ast");

        var output:TextAreaElement = cast d.getElementById("output");

        var version:SpanElement = cast d.getElementById("version");

        // var buildDate:SpanElement = cast d.getElementById("build-date");

        // var buildNumber:SpanElement = cast d.getElementById("build-number");

        version.innerText = Little.version;

        if (Little.version.endsWith("f")) d.getElementById("casing").innerHTML += " (f - Functional programming only)";

        input.addEventListener("keyup", function(\_) {

            try {

                ast.value = little.tools.PrettyPrinter.printInterpreterAst(little.interpreter.Interpreter.convert(...little.parser.Parser.parse(little.lexer.Lexer.lex(input.value))));

            } catch (e) {}

            try {

                Little.reset();

                Little.run(input.value, true);

                output.value = Little.runtime.stdout.output;

            } catch (e) {}

        });

        input.onkeydown = function(e) {

            if (e.key == 'Tab') {

                e.preventDefault();

                var start = input.selectionStart;

                var end = input.selectionEnd;

                // set textarea value to: text before caret + tab + text after caret

                input.value = input.value.substring(0, start) + "\t" + input.value.substring(end);

                // put caret at right position again

                input.selectionStart = input.selectionEnd = start + 1;

            }

        }

        var keywordTable:TableElement = cast d.getElementById("k-table-body");

        /\*\*

            Update tables

        \*\*/

        function update() {

            var firstRow = true;

            for (row in keywordTable.rows) {

                if (firstRow) {

                    firstRow = false;

                    continue;

                }

                var p = row.getElementsByTagName("p")[0];

                var input = row.getElementsByTagName("input")[0];

                p.innerText = getCodeExample(input.id);

                p.onchange();

            }

            input.dispatchEvent(new Event("keyup"));

        }

        for (keyword in Type.getInstanceFields(KeywordConfig)) {

            if (keyword == "change")

                continue;

            if (getCodeExample(keyword) == "irrelevant") continue;

            var row = d.createTableRowElement();

            var usage = d.createTextNode(keyword.snakeToTitleCase());

            var input = d.createInputElement();

            input.id = keyword;

            input.placeholder = "single word, e.g. " + Reflect.field(Little.keywords, keyword);

            input.onchange = () -> {

                Reflect.setField(Little.keywords, keyword,

                    input.value != null ? (input.value != "" ? input.value : Reflect.field(Little.keywords,

                        keyword)) : Reflect.field(Little.keywords, keyword));

                update();

            }

            var p = d.createParagraphElement();

            var td1 = d.createTableCellElement();

            td1.appendChild(usage);

            var td2 = d.createTableCellElement();

            td2.appendChild(input);

            var td3 = d.createTableCellElement();

            td3.appendChild(p);

            row.appendChild(td1);

            row.appendChild(td2);

            row.appendChild(td3);

            keywordTable.appendChild(row);

        }

        Syntax.plainCode("Highlighter.registerOnParagraphs()");

        Syntax.plainCode('document.getElementById("input").dispatchEvent(new Event("keyup"));');

        update();

    }

    public function getCodeExample(keyword:String) {

        var ret = switch keyword {

            case "VARIABLE\_DECLARATION": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_INT} = 8';

            case "FUNCTION\_DECLARATION": '${Little.keywords.FUNCTION\_DECLARATION} y(${Little.keywords.VARIABLE\_DECLARATION} parameter, ${Little.keywords.VARIABLE\_DECLARATION} times ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_INT}) ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_STRING} =  {\n&nbsp;&nbsp;&nbsp;&nbsp;${Little.keywords.FUNCTION\_RETURN} parameter ${Little.keywords.MULTIPLY\_SIGN} times\n}\n${Little.keywords.PRINT\_FUNCTION\_NAME}(y("Hey", 3))';

            case "NULL\_VALUE": 'if (x ${Little.keywords.EQUALS\_SIGN} ${Little.keywords.NULL\_VALUE}) {}\n${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.NULL\_VALUE}';

            case "RUN\_CODE\_FUNCTION\_NAME": '${Little.keywords.RUN\_CODE\_FUNCTION\_NAME}("${Little.keywords.PRINT\_FUNCTION\_NAME}(5 ${Little.keywords.ADD\_SIGN} 3)")';

            case "RAISE\_ERROR\_FUNCTION\_NAME": '${Little.keywords.RAISE\_ERROR\_FUNCTION\_NAME}("My Own Custom Error! :D")';

            case "PRINT\_FUNCTION\_NAME": '${Little.keywords.PRINT\_FUNCTION\_NAME}("Hello World")';

            case "TYPE\_DECL\_OR\_CAST": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_STRING}\nx = ${Little.keywords.TRUE\_VALUE} ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_STRING} """ "${Little.keywords.TRUE\_VALUE}" """';

            case "TYPE\_CAST\_FUNCTION\_PREFIX": 'if (1${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_BOOLEAN}()) {\n&nbsp;&nbsp;&nbsp;&nbsp;${Little.keywords.PRINT\_FUNCTION\_NAME}("${Little.keywords.TRUE\_VALUE}"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_BOOLEAN}()${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_INT}())\n}';

            case "TYPE\_FLOAT": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_FLOAT} = 8.8';

            case "TYPE\_INT": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_INT} = 8';

            case "TYPE\_BOOLEAN": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_BOOLEAN} = ${Little.keywords.TRUE\_VALUE} || ${Little.keywords.FALSE\_VALUE}';

            case "TYPE\_STRING": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_STRING} = "Hey There!"';

            case "TYPE\_MODULE": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_MODULE} = ${Little.keywords.TYPE\_BOOLEAN}';

            case "TYPE\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_SIGN} = ${Little.keywords.ADD\_SIGN}';

            case "TYPE\_OBJECT": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_OBJECT} = ${Little.keywords.TYPE\_OBJECT}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.INSTANTIATE\_FUNCTION\_NAME}()\n${Little.keywords.VARIABLE\_DECLARATION} x${Little.keywords.PROPERTY\_ACCESS\_SIGN}y = 4\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x${Little.keywords.PROPERTY\_ACCESS\_SIGN}y) """4"""';

            case "TYPE\_MEMORY": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_allocate}(amount)\n${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_write}(x, myNumericArray)\n${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_free}(x, arrayByteAmount)\n${Little.keywords.PRINT\_FUNCTION\_NAME}(${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_size} ${Little.keywords.DIVIDE\_SIGN} ${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_maxSize})';

            case "TYPE\_UNKNOWN": '${Little.keywords.VARIABLE\_DECLARATION} x, ${Little.keywords.PRINT\_FUNCTION\_NAME}(x${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME}) """ ${Little.keywords.TYPE\_UNKNOWN} """';

            case "TYPE\_ARRAY": '${Little.keywords.VARIABLE\_DECLARATION} array ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_ARRAY} = ${Little.keywords.TYPE\_ARRAY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.INSTANTIATE\_FUNCTION\_NAME}(${Little.keywords.TYPE\_STRING}, 3)\narray${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_ARRAY\_set}(1, "Hey!")\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_ARRAY\_get}(1)) """ "Hey!" """\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_ARRAY\_length}) """ 3 """';

            case "TYPE\_FUNCTION": '${Little.keywords.FUNCTION\_DECLARATION} x() = {}\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME}) """ ${Little.keywords.TYPE\_FUNCTION} """';

            case "TYPE\_CONDITION": '${Little.keywords.PRINT\_FUNCTION\_NAME}(${Little.keywords.CONDITION\_\_IF}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME}) """ ${Little.keywords.TYPE\_CONDITION} """';

            case "FUNCTION\_RETURN": '${Little.keywords.FUNCTION\_DECLARATION} y() = {\n&nbsp;&nbsp;&nbsp;&nbsp;${Little.keywords.FUNCTION\_RETURN} 8\n}';

            case "READ\_FUNCTION\_NAME": '${Little.keywords.VARIABLE\_DECLARATION} x = 3\n${Little.keywords.READ\_FUNCTION\_NAME}("x")';

            case "TYPE\_DYNAMIC": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_DYNAMIC} = nothing';

            case "PROPERTY\_ACCESS\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} len = 8${Little.keywords.PROPERTY\_ACCESS\_SIGN}type${Little.keywords.PROPERTY\_ACCESS\_SIGN}length\n${Little.keywords.PRINT\_FUNCTION\_NAME}(len${Little.keywords.PROPERTY\_ACCESS\_SIGN}type) """ ${Little.keywords.TYPE\_INT} """';

            case "TRUE\_VALUE": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_BOOLEAN} = ${Little.keywords.TRUE\_VALUE} ${Little.keywords.OR\_SIGN} ${Little.keywords.FALSE\_VALUE}\nif (${Little.keywords.TRUE\_VALUE}) {}';

            case "FALSE\_VALUE": '${Little.keywords.VARIABLE\_DECLARATION} x ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${Little.keywords.TYPE\_BOOLEAN} = ${Little.keywords.TRUE\_VALUE} ${Little.keywords.AND\_SIGN} ${Little.keywords.FALSE\_VALUE}\nif (${Little.keywords.FALSE\_VALUE}) {}';

            case "EQUALS\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.TRUE\_VALUE} ${Little.keywords.EQUALS\_SIGN} ${Little.keywords.TRUE\_VALUE}\nif (x ${Little.keywords.EQUALS\_SIGN} ${Little.keywords.TRUE\_VALUE}) {}';

            case "NOT\_EQUALS\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.TRUE\_VALUE} ${Little.keywords.NOT\_EQUALS\_SIGN} ${Little.keywords.TRUE\_VALUE}\nif (x ${Little.keywords.NOT\_EQUALS\_SIGN} ${Little.keywords.TRUE\_VALUE}) {}';

            case "LARGER\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 5\nif (x ${Little.keywords.LARGER\_SIGN} 1) {}';

            case "SMALLER\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.NEGATE\_SIGN}5\nif (x ${Little.keywords.SMALLER\_SIGN} 1) {}';

            case "LARGER\_EQUALS\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 5\nif (x ${Little.keywords.LARGER\_EQUALS\_SIGN} 1 ${Little.keywords.ADD\_SIGN} 4) {}';

            case "SMALLER\_EQUALS\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 5\nif (x ${Little.keywords.SMALLER\_EQUALS\_SIGN} ${Little.keywords.NEGATE\_SIGN}(1 ${Little.keywords.ADD\_SIGN} 4)) {}';

            case "XOR\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.TRUE\_VALUE} ${Little.keywords.XOR\_SIGN} ${Little.keywords.TRUE\_VALUE}\nif (x ${Little.keywords.XOR\_SIGN} ${Little.keywords.TRUE\_VALUE}) {}';

            case "OR\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.TRUE\_VALUE} ${Little.keywords.OR\_SIGN} ${Little.keywords.TRUE\_VALUE}\nif (x ${Little.keywords.OR\_SIGN} ${Little.keywords.TRUE\_VALUE}) {}';

            case "AND\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.TRUE\_VALUE} ${Little.keywords.AND\_SIGN} ${Little.keywords.TRUE\_VALUE}\nif (x ${Little.keywords.AND\_SIGN} ${Little.keywords.TRUE\_VALUE}) {}';

            case "NOT\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.NOT\_SIGN}${Little.keywords.TRUE\_VALUE}\nif (${Little.keywords.NOT\_SIGN}x) {}';

            case "ADD\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 1 ${Little.keywords.ADD\_SIGN} 2\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 3 """';

            case "SUBTRACT\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 1 ${Little.keywords.SUBTRACT\_SIGN} 2\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ ${Little.keywords.NEGATE\_SIGN}1 """';

            case "MULTIPLY\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 1 ${Little.keywords.MULTIPLY\_SIGN} 2\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 2 """';

            case "DIVIDE\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 1 ${Little.keywords.DIVIDE\_SIGN} 2\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 0.5 """';

            case "MOD\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 1 ${Little.keywords.MOD\_SIGN} 2\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 1 """';

            case "POW\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 1 ${Little.keywords.POW\_SIGN} 2\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 1 """';

            case "FACTORIAL\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 1 ${Little.keywords.FACTORIAL\_SIGN}\n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 1 """';

            case "SQRT\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = 3${Little.keywords.SQRT\_SIGN}8 ${Little.keywords.ADD\_SIGN} ${Little.keywords.SQRT\_SIGN}25 \n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 7 """';

            case "NEGATE\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.NEGATE\_SIGN}1 \n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ ${Little.keywords.NEGATE\_SIGN}1 """';

            case "POSITIVE\_SIGN": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.POSITIVE\_SIGN}1 \n${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 1 """';

            case "OBJECT\_TYPE\_PROPERTY\_NAME": '${Little.keywords.PRINT\_FUNCTION\_NAME}(1${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME}) """ ${Little.keywords.TYPE\_INT} """';

            case "OBJECT\_ADDRESS\_PROPERTY\_NAME": '${Little.keywords.PRINT\_FUNCTION\_NAME}(0${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.OBJECT\_ADDRESS\_PROPERTY\_NAME}) """ ${Little.memory.constants.ZERO.rawLocation} """';

            case "CONDITION\_\_FOR\_LOOP": '${Little.keywords.CONDITION\_\_FOR\_LOOP} (${Little.keywords.FOR\_LOOP\_FROM} 0 ${Little.keywords.FOR\_LOOP\_TO} 10 ${Little.keywords.FOR\_LOOP\_JUMP} 2) { ${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 0, 2, 4, 6, 8 """ }';

            case "CONDITION\_\_WHILE\_LOOP": '${Little.keywords.CONDITION\_\_WHILE\_LOOP} (${Little.keywords.TRUE\_VALUE}) {\n&nbsp;&nbsp;&nbsp;&nbsp;${Little.keywords.PRINT\_FUNCTION\_NAME}(x), x = x ${Little.keywords.ADD\_SIGN} 1 """ 0, 1, 2, 3, 4... """\n}';

            case "CONDITION\_\_IF": '${Little.keywords.CONDITION\_\_IF} (${Little.keywords.TRUE\_VALUE}) { ${Little.keywords.PRINT\_FUNCTION\_NAME}(0) """ 0 """ }';

            case "CONDITION\_\_ELSE": '${Little.keywords.CONDITION\_\_IF} (${Little.keywords.FALSE\_VALUE}) {}\n${Little.keywords.CONDITION\_\_ELSE} ${Little.keywords.CONDITION\_\_IF} (0${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_BOOLEAN}) {}\n${Little.keywords.CONDITION\_\_ELSE} { ${Little.keywords.PRINT\_FUNCTION\_NAME}(1) """ 1 """ }';

            case "CONDITION\_\_WHENEVER": '${Little.keywords.CONDITION\_\_WHENEVER} (value ${Little.keywords.EQUALS\_SIGN} ${Little.keywords.TRUE\_VALUE}) { ${Little.keywords.PRINT\_FUNCTION\_NAME}(0) """ 0 Every time this is true. """ }';

            case "CONDITION\_\_AFTER": '${Little.keywords.CONDITION\_\_AFTER} (value ${Little.keywords.EQUALS\_SIGN} ${Little.keywords.TRUE\_VALUE}) { ${Little.keywords.PRINT\_FUNCTION\_NAME}(0) """ 0 Only once. """ }';

            case "STDLIB\_\_FLOAT\_isWhole": '5.6${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_FLOAT\_isWhole}';

            case "STDLIB\_\_STRING\_length": '"Hey There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_length}';

            case "STDLIB\_\_STRING\_toLowerCase": '"HEY THERE"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_toLowerCase}()';

            case "STDLIB\_\_STRING\_toUpperCase": '"hey there"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_toUpperCase}()';

            case "STDLIB\_\_STRING\_trim": '"  Hey There  "${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_trim}()';

            case "STDLIB\_\_STRING\_substring": '"Hey There erowih"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_substring}(0, 10)';

            case "STDLIB\_\_STRING\_charAt": '"Hey ! There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_charAt}(4)';

            case "STDLIB\_\_STRING\_split": '""${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_split}(" ")';

            case "STDLIB\_\_STRING\_replace": '"Hey There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_replace}("There", "You!")';

            case "STDLIB\_\_STRING\_remove": '"Hey, You There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_remove}(", You ")';

            case "STDLIB\_\_STRING\_contains": '"Hey There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_contains}("There")';

            case "STDLIB\_\_STRING\_indexOf": '"Hey There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_indexOf}("The")';

            case "STDLIB\_\_STRING\_lastIndexOf": '"Hey There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_lastIndexOf}("e")';

            case "STDLIB\_\_STRING\_startsWith": '"Hey There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_startsWith}("Hey")';

            case "STDLIB\_\_STRING\_endsWith": '"Hey There"${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_endsWith}("ere")';

            case "STDLIB\_\_STRING\_fromCharCode": '${Little.keywords.TYPE\_STRING}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_STRING\_fromCharCode}(72)';

            case "STDLIB\_\_ARRAY\_length": 'myArray${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_ARRAY\_length}';

            case "STDLIB\_\_ARRAY\_elementType": 'myArray${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_ARRAY\_elementType} """ An Array of Strings will return `${Little.keywords.TYPE\_STRING}`"""';

            case "STDLIB\_\_ARRAY\_get": 'myArray${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_ARRAY\_get}(0)';

            case "STDLIB\_\_ARRAY\_set": 'myArray${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_ARRAY\_set}(0, "Hey")';

            case "STDLIB\_\_MEMORY\_allocate": '${Little.keywords.VARIABLE\_DECLARATION} address = ${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_allocate}(byteAmount)';

            case "STDLIB\_\_MEMORY\_free": '${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_free}(address, byteAmount)';

            case "STDLIB\_\_MEMORY\_read": '${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_read}(address, valueType """`${Little.keywords.TYPE\_INT}`, `${Little.keywords.TYPE\_BOOLEAN}`...""")';

            case "STDLIB\_\_MEMORY\_write": '${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_write}(address, myNumericOrBooleanArray)';

            case "STDLIB\_\_MEMORY\_size": '${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_size}';

            case "STDLIB\_\_MEMORY\_maxSize": '${Little.keywords.TYPE\_MEMORY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.STDLIB\_\_MEMORY\_maxSize}';

            case "FOR\_LOOP\_FROM": '${Little.keywords.CONDITION\_\_FOR\_LOOP} (${Little.keywords.FOR\_LOOP\_FROM} 0 ${Little.keywords.FOR\_LOOP\_TO} 10 ${Little.keywords.FOR\_LOOP\_JUMP} 2) { ${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 0, 2, 4, 6, 8 """ }';

            case "FOR\_LOOP\_TO": '${Little.keywords.CONDITION\_\_FOR\_LOOP} (${Little.keywords.FOR\_LOOP\_FROM} 0 ${Little.keywords.FOR\_LOOP\_TO} 10 ${Little.keywords.FOR\_LOOP\_JUMP} 2) { ${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 0, 2, 4, 6, 8 """ }';

            case "FOR\_LOOP\_JUMP": '${Little.keywords.CONDITION\_\_FOR\_LOOP} (${Little.keywords.FOR\_LOOP\_FROM} 0 ${Little.keywords.FOR\_LOOP\_TO} 10 ${Little.keywords.FOR\_LOOP\_JUMP} 2) { ${Little.keywords.PRINT\_FUNCTION\_NAME}(x) """ 0, 2, 4, 6, 8 """ }';

            case "INSTANTIATE\_FUNCTION\_NAME": '${Little.keywords.VARIABLE\_DECLARATION} x = ${Little.keywords.TYPE\_OBJECT}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.INSTANTIATE\_FUNCTION\_NAME}()\n${Little.keywords.VARIABLE\_DECLARATION} a = ${Little.keywords.TYPE\_ARRAY}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${Little.keywords.INSTANTIATE\_FUNCTION\_NAME}(${Little.keywords.TYPE\_STRING}, 10)';

            case "RECOGNIZED\_SIGNS": "irrelevant";

            case "CONDITION\_PATTERN\_PARAMETER\_NAME": "irrelevant";

            case "CONDITION\_BODY\_PARAMETER\_NAME": "irrelevant";

            case "MAIN\_MODULE\_NAME": "irrelevant";

            case "REGISTERED\_MODULE\_NAME": "irrelevant";

            case \_: '""" No Example Yet! Stay tuned... """';

        }

        return ret == '' ? '""" No Example Yet! Stay tuned... """' : ret;

    }

    static function snakeToTitleCase(str:String):String {

        var words = str.split("\_");

        for (i in 0...words.length) {

            var word = words[i];

            if (word.length > 0) {

                var firstChar = word.charAt(0);

                var rest = word.substr(1);

                words[i] = firstChar.toUpperCase() + rest.toLowerCase();

            }

        }

        return words.join(" ");

    }

}

package little;

import little.lexer.Lexer;

import haxe.exceptions.ArgumentException;

using StringTools;

using little.tools.TextTools;

/\*\*

    Represents a set of keys to use for different keywords/features in `Little`.

\*\*/

@:structInit

class KeywordConfig {

    /\*\*

        The default keyword configuration. Here incase you want to reset keywords, or just have a reference to the original ones.

    \*\*/

    public static var defaultConfig(default, never):KeywordConfig = {};

    /\*\*

        every single character in this array will be recognized as an operator.

        If you register an operator, it should automatically exist here too.

        IMPORTANT - this is not the same as the field RECOGNIZED\_SIGNS - this is

        used strictly for lexing purposes.

    \*\*/

    public static var recognizedOperators:Array<String> = ["!", "#", "$", "%", "&", "'", "(", ")", "\*", "+", "-", ".", "/", ":", "<", "=", ">", "?", "@", "[", "\\", "]", "^", "\_", "`", "{", "|", "}", "~", "^", "√"];

    /\*\*

        Creates a new keyword config, using an existing config, made using the anonymous structure syntax.

        @param config The config to use. Not all fields have to be referenced - those that aren't referenced are set

        to their default value. For the default configuration, don't provide parameters.

        @param nullifyDefaults

    \*\*/

    public function new(?config:KeywordConfig, nullifyDefaults:Bool = true) {

        if (config == null)

            return;

        if (nullifyDefaults) {

            var fields = Type.getInstanceFields(KeywordConfig);

            fields.remove("defaultConfig");

            for (field in fields) {

                var configValue = (Reflect.field(config, field) : String);

                if (configValue.length == 0)

                    throw new ArgumentException('config.$field', "Keywords of length 0 are not allowed.");

                if (configValue.contains(" "))

                    throw new ArgumentException('config.$field', "Keywords cannot contain whitespaces.");

                if (configValue.containsAny(recognizedOperators))

                    throw new ArgumentException('config.$field', "Keywords cannot contain operators/signs.");

                if (~/[0-9]/.match(configValue.charAt(0)))

                    throw new ArgumentException('config.$field', "Keywords cannot start with numbers.");

                if (configValue == Reflect.field(defaultConfig, field))

                    Reflect.setField(this, field, null);

                else

                    Reflect.setField(this, field, configValue);

            }

        }

    }

    /\*\*

        Applies a different set of keywords onto this one. If it contains nulls, they are skipped.

        Any other value is not skipped.

        @param config the configuration to apply.

    \*\*/

    public function change(config:KeywordConfig) {

        var fields = Type.getInstanceFields(KeywordConfig);

        fields.remove("defaultConfig");

        for (field in fields) {

            var configValue = (Reflect.field(config, field) : String);

            if (configValue == null)

                continue;

            Reflect.setField(this, field, configValue);

        }

    }

    @:optional public var VARIABLE\_DECLARATION:String = "define";

    @:optional public var FUNCTION\_DECLARATION:String = "action";

    @:optional public var TYPE\_DECL\_OR\_CAST:String = "as";

    @:optional public var FUNCTION\_RETURN:String = "return";

    @:optional public var NULL\_VALUE:String = "nothing";

    @:optional public var TRUE\_VALUE:String = "true";

    @:optional public var FALSE\_VALUE:String = "false";

    @:optional public var TYPE\_DYNAMIC:String = "Anything";

    @:optional public var TYPE\_INT:String = "Number";

    @:optional public var TYPE\_FLOAT:String = "Decimal";

    @:optional public var TYPE\_BOOLEAN:String = "Boolean";

    @:optional public var TYPE\_STRING:String = "Characters";

    @:optional public var TYPE\_OBJECT:String = "Object";

    @:optional public var TYPE\_MEMORY:String = "Memory";

    @:optional public var TYPE\_ARRAY:String = "Array";

    /\*\*

        Represents the main function type.

        The underlying type is `TYPE\_STRING`.

    \*\*/

    @:optional public var TYPE\_FUNCTION:String = "Action";

    /\*\*

        Represents the general type of a condition.

        The underlying type is `TYPE\_STRING`.

    \*\*/

    @:optional public var TYPE\_CONDITION:String = "Condition";

    /\*\*

        represent the "type" type:

        for example: `5` is of type `Number`, and `Number` is of type `Type`

    \*\*/

    @:optional public var TYPE\_MODULE:String = "Type";

    /\*\*

        Represents the type of a sign (for example, +)

        Exists for fun, but still functional :)

    \*\*/

    @:optional public var TYPE\_SIGN:String = "Sign";

    @:optional public var MAIN\_MODULE\_NAME:String = "Main";

    @:optional public var OBJECT\_TYPE\_PROPERTY\_NAME:String = "type";

    @:optional public var OBJECT\_ADDRESS\_PROPERTY\_NAME:String = "address";

    @:optional public var PRINT\_FUNCTION\_NAME:String = "print";

    @:optional public var RAISE\_ERROR\_FUNCTION\_NAME:String = "error";

    @:optional public var READ\_FUNCTION\_NAME:String = "read";

    @:optional public var RUN\_CODE\_FUNCTION\_NAME:String = "run";

    @:optional public var CONDITION\_PATTERN\_PARAMETER\_NAME:String = "pattern";

    @:optional public var CONDITION\_BODY\_PARAMETER\_NAME:String = "code";

    @:optional public var CONDITION\_\_FOR\_LOOP:String = "for";

    @:optional public var CONDITION\_\_WHILE\_LOOP:String = "while";

    @:optional public var CONDITION\_\_IF:String = "if";

    @:optional public var CONDITION\_\_ELSE:String = "else";

    @:optional public var CONDITION\_\_WHENEVER:String = "whenever";

    @:optional public var CONDITION\_\_AFTER:String = "after";

    /\*\*

        No need to ever change this, this is a parser-only feature

    \*\*/

    @:optional public var TYPE\_UNKNOWN:String = "Unknown";

    public var RECOGNIZED\_SIGNS:Array<String> = [];

    /\*\*

        When changing this to a multi-char sign (such as "->"), remember to also push that sign to `RECOGNIZED\_SIGNS`, so it would be parsed correctly.

    \*\*/

    @:optional public var PROPERTY\_ACCESS\_SIGN:String = ".";

    @:optional public var EQUALS\_SIGN:String = "==";

    @:optional public var NOT\_EQUALS\_SIGN:String = "!=";

    @:optional public var LARGER\_SIGN:String = ">";

    @:optional public var SMALLER\_SIGN:String = "<";

    @:optional public var LARGER\_EQUALS\_SIGN:String = ">=";

    @:optional public var SMALLER\_EQUALS\_SIGN:String = "<=";

    @:optional public var XOR\_SIGN:String = "^^";

    @:optional public var OR\_SIGN:String = "||";

    @:optional public var AND\_SIGN:String = "&&";

    @:optional public var NOT\_SIGN:String = "!"; //on the left side

    @:optional public var ADD\_SIGN:String = "+";

    @:optional public var SUBTRACT\_SIGN:String = "-";

    @:optional public var MULTIPLY\_SIGN:String = "\*";

    @:optional public var DIVIDE\_SIGN:String = "/";

    @:optional public var MOD\_SIGN:String = "%";

    @:optional public var POW\_SIGN:String = "^";

    @:optional public var FACTORIAL\_SIGN:String = "!"; //on the right side

    @:optional public var SQRT\_SIGN:String = "√";

    @:optional public var NEGATE\_SIGN:String = "-";

    @:optional public var POSITIVE\_SIGN:String = "+";

    // Cast functions are done using the `to` keyword and type. They are not here.

    //Float: DONE

    @:optional public var STDLIB\_\_FLOAT\_isWhole:String = "isWhole";

    //String: DONE

    @:optional public var STDLIB\_\_STRING\_length:String = "length";

    @:optional public var STDLIB\_\_STRING\_toLowerCase:String = "toLowerCase";

    @:optional public var STDLIB\_\_STRING\_toUpperCase:String = "toUpperCase";

    @:optional public var STDLIB\_\_STRING\_trim:String = "trim";

    @:optional public var STDLIB\_\_STRING\_substring:String = "substring";

    @:optional public var STDLIB\_\_STRING\_charAt:String = "charAt";

    @:optional public var STDLIB\_\_STRING\_split:String = "split";

    @:optional public var STDLIB\_\_STRING\_replace:String = "replace";

    @:optional public var STDLIB\_\_STRING\_remove:String = "remove";

    @:optional public var STDLIB\_\_STRING\_contains:String = "contains";

    @:optional public var STDLIB\_\_STRING\_indexOf:String = "indexOf";

    @:optional public var STDLIB\_\_STRING\_lastIndexOf:String = "lastIndexOf";

    @:optional public var STDLIB\_\_STRING\_startsWith:String = "startsWith";

    @:optional public var STDLIB\_\_STRING\_endsWith:String = "endsWith";

    @:optional public var STDLIB\_\_STRING\_fromCharCode:String = "fromCharCode";

    //Array: DONE

    @:optional public var STDLIB\_\_ARRAY\_length:String = "length";

    @:optional public var STDLIB\_\_ARRAY\_elementType:String = "elementType";

    @:optional public var STDLIB\_\_ARRAY\_get:String = "get";

    @:optional public var STDLIB\_\_ARRAY\_set:String = "set";

    @:optional public var STDLIB\_\_MEMORY\_allocate:String = "allocate";

    @:optional public var STDLIB\_\_MEMORY\_free:String = "free";

    @:optional public var STDLIB\_\_MEMORY\_read:String = "read";

    @:optional public var STDLIB\_\_MEMORY\_write:String = "write";

    @:optional public var STDLIB\_\_MEMORY\_size:String = "size";

    @:optional public var STDLIB\_\_MEMORY\_maxSize:String = "maxSize";

    @:optional public var FOR\_LOOP\_FROM:String = "from";

    @:optional public var FOR\_LOOP\_TO:String = "to";

    @:optional public var FOR\_LOOP\_JUMP:String = "jump";

    @:optional public var TYPE\_CAST\_FUNCTION\_PREFIX:String = "to";

    @:optional public var INSTANTIATE\_FUNCTION\_NAME:String = "create";

}

package little;

import little.interpreter.Tokens.InterpTokens;

import vision.ds.Queue;

import vision.helpers.VisionThread;

import little.tools.PrettyPrinter;

import little.interpreter.memory.Memory;

import little.tools.Plugins;

import little.tools.PrepareRun;

import little.lexer.Lexer;

import little.parser.Parser;

import little.interpreter.Interpreter;

import little.interpreter.Runtime;

import little.interpreter.memory.Operators;

@:access(little.interpreter.Interpreter)

@:access(little.interpreter.Runtime)

@:expose("Little")

class Little {

    /\*\*

        A feature of the `Little` programming language is that it is possible to change keywords & other

        usually hardcoded properties.

        You can change the values here if you want to, or just

    \*\*/

    public static var keywords:KeywordConfig = {};

    /\*\*

        Used to access runtime details of the current "running instance" of `Little`.

        Contains callbacks for operations, access to the callstack, and more.

    \*\*/

    public static var runtime(default, null):Runtime = new Runtime();

    /\*\*

        The independent memory manager. Allocates and deallocates variables, using

        gradually allocated byte arrays.

    \*\*/

    public static var memory(default, null):Memory = new Memory();

    /\*\*

        A portal that allows external interfacing with little code, both during and before runtime.

        You can add classes, variables, functions, and even operators.

    \*\*/

    public static var plugin(default, null):Plugins = new Plugins(Little.memory);

    /\*\*

        Used to store code that is currently being ran/queued for running right before the main module using

        `runRightBeforeMain` in `Little.loadModule()`.

    \*\*/

    public static var queue(default, null):Queue<String> = new Queue();

    /\*\*

        When enabled:

         - `print`, `error` and `warn` calls will contain the part of the lexer/parser/interpreter hat called them (see `little.tools.Layer`)

    \*\*/

    public static var debug:Bool = false;

    /\*\*

        Indicates the version of the Little compiler & Interpreter.

        First number is the major version, second is the minor version, third is the patch.

    \*\*/

    public static var version:String = "1.0.0-f";

    /\*\*

        Loads little code, without clearing memory, stdout or the callstack. useful if you want to

        use multiple files/want to preload code for the end user to use.

        Notice - after calling this method, event listeners will dispatch (i.e. they're not exclusive to the `run()` method).

        @param code a string containing code written in Little.

        @param name a name to call the module, so it would be easily identifiable

        @param debug when runRightBeforeMain is set to false, this temporarily overrides default `Little.debug`.

        @param runRightBeforeMain When set to true, instead of parsing and running the code right after this function is called,

            we wait for `Little.run()` to get called, and then we parse and run this module right before the main module. Defaults to false.

    \*\*/

    public static function loadModule(code:String, name:String, debug:Bool = false, runRightBeforeMain:Bool = false) {

        runtime.errorThrown = false;

        runtime.line = 0;

        runtime.module = name;

        if (runRightBeforeMain) {

            Little.queue.enqueue(code);

        } else {

            final previous = Little.debug;

            #if !static if (debug != null) #end Little.debug = debug;

            if (!PrepareRun.prepared) {

                PrepareRun.addTypes();

                PrepareRun.addSigns();

                PrepareRun.addFunctions();

                PrepareRun.addConditions();

                PrepareRun.addProps();

            }

            Interpreter.run(Interpreter.convert(...Parser.parse(Lexer.lex(code))));

            #if !static if (debug != null) #end Little.debug = previous;

        }

    }

    /\*\*

        Runs a new Little program.

        If you want to preload some more code, use the `Little.loadModule()` method before calling this.

         - **\*\*pay attention - all modules & registered elements are unloaded after each run.\*\***

        If you want to use another keyword set (for example, to allow programming everything in spanish),

        make sure to set `currentKeywordSet` before calling this. If you want to use the default keywords with some changes,

        you can make some changes to the properties in `little.Keywords` instead.

        To register different types of "elements", such as definitions (variables), actions (functions), or even entire classes, you can use the various registration methods inside this class.

        If you want to add event listeners, to certain code interpretation events, check out the stats and listeners inside `Little.runtime`.

        @param code

        @param debug specifically specify whether or not to print more debugging information. Overrides default `Little.debug`.

    \*\*/

    public static function run(code:String, ?debug:Bool) {

        try {

            final previous = Little.debug;

            if (debug != null) Little.debug = debug;

            if (!PrepareRun.prepared) {

                PrepareRun.addTypes();

                PrepareRun.addSigns();

                PrepareRun.addFunctions();

                PrepareRun.addConditions();

                PrepareRun.addProps();

            }

            runtime.module = keywords.MAIN\_MODULE\_NAME;

            runtime.errorThrown = false;

            runtime.line = 0;

            Little.queue.enqueue(code);

            for (item in Little.queue) {

                Interpreter.run(Interpreter.convert(...Parser.parse(Lexer.lex(item))));

            }

            if (debug != null) Little.debug = previous;

        } catch (e) {

            // e.message == "Quitting..." ? trace(e.message) : trace(e.details());

            // Do nothing

        }

    }

    /\*\*

        Converts a string of code written in Little into an array of tokens, representing an AST.

        This array can be compiled into bytecode using `ByteCode.compile`, or run

        on the spot using `Interpreter.run`.

        This function, `ByteCode.compile` and `Interpreter.run` do no error handling on their own.

        When using this function, either verify that the coe your'e running is 100% correct, or encase

        the calls in a try-catch block.

        Aside from the difference mentioned above, code running "on the spot" behaves no

        different than code running using `Little.run`.

        @param code

    \*\*/

    public static function compile(code:String):Array<InterpTokens> {

        return Interpreter.convert(...Parser.parse(Lexer.lex(code)));

    }

    public static function format(code:String):String {

        return PrettyPrinter.stringifyParser(Parser.parse(Lexer.lex(code)));

    }

    /\*\*

        Resets all runtime details.

    \*\*/

    public static function reset() {

        runtime = new Runtime();

        Little.memory.reset();

        Little.queue = new Queue();

    }

}

package little.lexer;

enum LexerTokens {

    Identifier(name:String);

    Sign(char:String);

    Number(num:String);

    Boolean(value:String);

    Characters(string:String);

    NullValue;

    Newline;

    SplitLine;

    Documentation(content:String);

}

package little.lexer;

import little.lexer.Tokens.LexerTokens;

using StringTools;

using little.tools.TextTools;

using little.tools.Extensions;

class Lexer {

    /\*\*

        Converts a string with many items separated by word boundaries into different tokens

        of type `LexerTokens`.

    \*\*/

    public static function lex(code:String):Array<LexerTokens> {

        var tokens:Array<LexerTokens> = [];

        var i = 0;

        while (i < code.length) {

            var char = code.charAt(i);

            if (i < code.length - 2 && code.substr(i, 3).replace('"', "").length == 0) {

                var string = "";

                var queuedNewlines = 0;

                i += 3;

                while (i < code.length - 2 && code.substr(i, 3).replace('"', "").length != 0) {

                    string += code.charAt(i);

                    if (code.charAt(i) == "\n") queuedNewlines++;

                    i++;

                }

                i += 2;

                for (j in 0...queuedNewlines) tokens.push(Newline);

                tokens.push(Documentation(string.replace("<br>", "\n").replace("</br>", "\n").trim()));

            }

            else if (char == '"') {

                var string = "";

                i++;

                while (i < code.length && code.charAt(i) != '"') {

                    string += code.charAt(i);

                    i++;

                }

                tokens.push(Characters(string));

            } else if ("1234567890.".contains(char)) {

                var num = char;

                i++;

                while (i < code.length && "1234567890.".contains(code.charAt(i))) {

                    num += code.charAt(i);

                    i++;

                }

                i--;

                if (num == ".") tokens.push(Sign("."))

                else if (num.endsWith(".")) {

                    tokens.push(Number(num.replaceLast(".", "")));

                    tokens.push(Sign("."));

                }

                else tokens.push(Number(num));

            } else if (char == "\n") {

                tokens.push(Newline);

            } else if (char == ";" || char == ",") {

                tokens.push(SplitLine);

            } else if (KeywordConfig.recognizedOperators.contains(char)) {

                var sign = char;

                i++;

                while (i < code.length && KeywordConfig.recognizedOperators.contains(code.charAt(i))) {

                    sign += code.charAt(i);

                    i++;

                }

                i--;

                tokens.push(Sign(sign));

            } else if (new EReg('[^${KeywordConfig.recognizedOperators.join("")} \\t\\n\\r;,\\(\\)\\[\\]\\{\\}]', "g").match(char)) {

                var name = char;

                i++;

                while (i < code.length && new EReg('[^${KeywordConfig.recognizedOperators.join("")} \\t\\n\\r;,\\(\\)\\[\\]\\{\\}]', "g").match(code.charAt(i))) {

                    name += code.charAt(i);

                    i++;

                }

                i--;

                tokens.push(Identifier(name));

            }

            i++;

        }

        tokens = separateBooleanIdentifiers(tokens);

        tokens = mergeOrSplitKnownSigns(tokens);

        return tokens;

    }

    /\*\*

        Converts `Identifier("true"|"false"|"null")` tokens into `Boolean("true"|"false")` or `NullValue`.

    \*\*/

    public static function separateBooleanIdentifiers(tokens:Array<LexerTokens>):Array<LexerTokens> {

        return tokens.map(token -> {

            if (Type.enumEq(token, Identifier(Little.keywords.TRUE\_VALUE)) || Type.enumEq(token, Identifier(Little.keywords.FALSE\_VALUE))) {

                Boolean(token.getParameters()[0]);

            } else if (Type.enumEq(token, Identifier(Little.keywords.NULL\_VALUE))) {

                NullValue;

            } else token;

        });

    }

    /\*\*

        Some signs are more than 1 character long, so we need split/merge them when needed.

    \*\*/

    public static function mergeOrSplitKnownSigns(tokens:Array<LexerTokens>):Array<LexerTokens> {

        var post = [];

        var i = 0;

        while (i < tokens.length) {

            var token = tokens[i];

            switch token {

                case Sign(char): {

                    // First: reorder the keyword array by length

                    var recognizedSigns = TextTools.sortByLength(Little.keywords.RECOGNIZED\_SIGNS.concat([Little.keywords.PROPERTY\_ACCESS\_SIGN]));

                    recognizedSigns.reverse();

                    var shouldContinue = false;

                    while (char.length > 0) {

                        shouldContinue = false;

                        for (sign in recognizedSigns) {

                            if (char.startsWith(sign)) {

                                char = char.substring(sign.length);

                                post.push(Sign(sign));

                                shouldContinue = true;

                                break;

                            }

                        }

                        if (shouldContinue) continue;

                        post.push(Sign(char.charAt(0)));

                        char = char.substring(1);

                    }

                }

                case \_: post.push(token);

            }

            i++;

        }

        return post;

    }

}

package little.parser;

enum ParserTokens {

    SetLine(line:Int);

    SetModule(module:String);

    SplitLine;

    Variable(name:ParserTokens, type:ParserTokens, ?doc:ParserTokens);

    Function(name:ParserTokens, params:ParserTokens, type:ParserTokens, ?doc:ParserTokens);

    ConditionCall(name:ParserTokens, exp:ParserTokens, body:ParserTokens);

    Read(name:ParserTokens);

    Write(assignees:Array<ParserTokens>, value:ParserTokens);

    Identifier(word:String);

    TypeDeclaration(value:ParserTokens, type:ParserTokens);

    FunctionCall(name:ParserTokens, params:ParserTokens);

    Return(value:ParserTokens, type:ParserTokens);

    Expression(parts:Array<ParserTokens>, type:ParserTokens);

    Block(body:Array<ParserTokens>, type:ParserTokens);

    PartArray(parts:Array<ParserTokens>);

    PropertyAccess(name:ParserTokens, property:ParserTokens);

    Sign(sign:String);

    Number(num:String);

    Decimal(num:String);

    Characters(string:String);

    /\*\*

        Documentation strings

    \*\*/

    Documentation(doc:String);

    /\*\*

        Used for errors & warnings

    \*\*/

    ErrorMessage(msg:String);

    NullValue;

    TrueValue;

    FalseValue;

    /\*\*

        A custom token, if you want to implement macros with special syntax.

        You can match against your custom token using this syntax:

            switch token {

                case Custom("TokenName", [param1, param2]): {

                    // do something

                }

                case Custom("IntHaver", [num]) if (num.match(Number(\_))):

                case Custom("SimpleToken", []):

                case Custom("AnotherToken", enumParameters):

            }

    \*\*/

    Custom(name:String, params:Array<ParserTokens>);

}

package little.parser;

import little.tools.Layer;

import little.tools.PrettyPrinter;

import little.parser.Tokens.ParserTokens;

import little.lexer.Tokens.LexerTokens;

import little.interpreter.Runtime;

using StringTools;

using little.tools.TextTools;

using little.tools.Extensions;

using little.parser.Parser;

@:access(little.interpreter.Runtime)

class Parser {

    /\*\*

        An array of functions, which take in the current state of the abstract syntax tree as an array of `ParserTokens`,

        and returns a manipulated version of that abstract syntax tree as another array of `ParserTokens`.

        @see `Parser.mergeElses`

    \*\*/

    public static var additionalParsingLevels:Array<Array<ParserTokens> -> Array<ParserTokens>> = [Parser.mergeElses];

    /\*\*

        Parses the given array of `LexerTokens` into an abstract syntax tree, using tokens of type `ParserTokens`.

        To allow "macro" insertion, this function is assignable, which allows you to add parsing functions between existing ones.

        If your macros aren't parse-level sensitive, it is recommended that you use the `additionalParsingLevels`

        field instead of reassigning this function.

        @param lexerTokens The given tokens

        @return An array of tokens, representing an abstract syntax tree

    \*\*/

    public static dynamic function parse(lexerTokens:Array<LexerTokens>):Array<ParserTokens> {

        var tokens = convert(lexerTokens);

        tokens.unshift(SetModule(module));

        #if parser\_debug trace("before:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergeBlocks(tokens);

        #if parser\_debug trace("blocks:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergeExpressions(tokens);

        #if parser\_debug trace("expressions:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergePropertyOperations(tokens);

        #if parser\_debug trace("props:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergeTypeDecls(tokens);

        #if parser\_debug trace("types:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergeComplexStructures(tokens);

        #if parser\_debug trace("structures:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergeCalls(tokens);

        #if parser\_debug trace("calls:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergeWrites(tokens);

        #if parser\_debug trace("writes:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergeValuesWithTypeDecls(tokens);

        #if parser\_debug trace("casts:", PrettyPrinter.printParserAst(tokens)); #end

        tokens = mergeNonBlockBodies(tokens);

        #if parser\_debug trace("non-block bodies:", PrettyPrinter.printParserAst(tokens)); #end

        for (level in Parser.additionalParsingLevels) {

            tokens = level(tokens);

            #if parser\_debug trace('${level}:', PrettyPrinter.printParserAst(tokens)); #end

        }

        #if parser\_debug trace("macros:", PrettyPrinter.printParserAst(tokens)); #end

        return tokens;

    }

    /\*\*

        Simply converts lexer to parser tokens.

    \*\*/

    public static function convert(lexerTokens:Array<LexerTokens>):Array<ParserTokens> {

        var tokens:Array<ParserTokens> = [];

        var line = 1;

        var i = 0;

        while (i < lexerTokens.length) {

            var token = lexerTokens[i];

            switch token {

                case Identifier(name): tokens.push(Identifier(name));

                case Sign(char): tokens.push(Sign(char));

                case Number(num): {

                    if (num.countOccurrencesOf(".") == 0) tokens.push(Number(num));

                    else if (num.countOccurrencesOf(".") == 1) tokens.push(Decimal(num));

                }

                case Boolean(value): {

                    if (value == Little.keywords.FALSE\_VALUE) tokens.push(FalseValue);

                    else if (value == Little.keywords.TRUE\_VALUE) tokens.push(TrueValue);

                }

                case Characters(string): tokens.push(Characters(string));

                case NullValue: tokens.push(NullValue);

                case Newline: {

                    tokens.push(SetLine(line));

                    line++;

                }

                case SplitLine: tokens.push(SplitLine);

                case Documentation(content): tokens.push(Documentation(content));

            }

            i++;

        }

        return tokens;

    }

    /\*\*

        Merges This structure:

        ```

        { ... }

        ```

        Into a `Block()` token

    \*\*/

    public static function mergeBlocks(pre:Array<ParserTokens>):Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.push(token);}

                case SetModule(module): {Parser.module = module; post.push(token);}

                case SplitLine: {nextPart(); post.push(token);}

                case Sign("{"): {

                    var blockStartLine = line;

                    var blockBody:Array<ParserTokens> = [SetModule(module), SetLine(blockStartLine)];

                    var blockStack = 1; // Open and close the block on the correct curly bracket

                    while (i + 1 < pre.length) {

                        var lookahead = pre[i + 1];

                        if (Type.enumEq(lookahead, Sign("{"))) {

                            blockStack++;

                            blockBody.push(lookahead);

                        } else if (Type.enumEq(lookahead, Sign("}"))) {

                            blockStack--;

                            if (blockStack == 0) break;

                            blockBody.push(lookahead);

                        } else blockBody.push(lookahead);

                        i++;

                    }

                    // Throw error for unclosed blocks;

                    if (i + 1 == pre.length) {

                        Little.runtime.throwError(ErrorMessage('Unclosed code block, starting at line ' + blockStartLine));

                        return null;

                    }

                    post.push(Block(mergeBlocks(blockBody), null)); // The check performed above includes unmerged blocks inside the outer block. These unmerged blocks should be merged

                    i++;

                }

                case Expression(parts, type): post.push(Expression(mergeBlocks(parts), mergeBlocks([type])[0]));

                case Block(body, type): post.push(Block(mergeBlocks(body), mergeBlocks([type])[0]));

                case Custom(name, params): post.push(Custom(name, params.map(x -> mergeBlocks([x])[0])));

                case \_: post.push(token);

            }

            i++;

        }

        resetLines();

        return post;

    }

    /\*\*

        Merges This structure:

        ```

        (...)

        ```

        Into an `Expression()` token

    \*\*/

    public static function mergeExpressions(pre:Array<ParserTokens>):Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.push(token);}

                case SetModule(module): {Parser.module = module; post.push(token);}

                case SplitLine: {nextPart(); post.push(token);}

                case Sign("("): {

                    var expressionStartLine = line;

                    var expressionBody:Array<ParserTokens> = [];

                    var expressionStack = 1; // Open and close the block on the correct curly bracket

                    while (i + 1 < pre.length) {

                        var lookahead = pre[i + 1];

                        if (Type.enumEq(lookahead, Sign("("))) {

                            expressionStack++;

                            expressionBody.push(lookahead);

                        } else if (Type.enumEq(lookahead, Sign(")"))) {

                            expressionStack--;

                            if (expressionStack == 0) break;

                            expressionBody.push(lookahead);

                        } else expressionBody.push(lookahead);

                        i++;

                    }

                    // Throw error for unclosed expressions;

                    if (i + 1 == pre.length) {

                        Little.runtime.throwError(ErrorMessage('Unclosed expression, starting at line ' + expressionStartLine));

                        return null;

                    }

                    post.push(Expression(mergeExpressions(expressionBody), null)); // The check performed above includes unmerged blocks inside the outer block. These unmerged blocks should be merged

                    i++;

                }

                case Expression(parts, type): post.push(Expression(mergeExpressions(parts), mergeExpressions([type])[0]));

                case Block(body, type): post.push(Block(mergeExpressions(body), mergeExpressions([type])[0]));

                case Custom(name, params): post.push(Custom(name, params.map(x -> mergeExpressions([x])[0])));

                case \_: post.push(token);

            }

            i++;

        }

        resetLines();

        return post;

    }

    /\*\*

       Merges a chain of single tokens seperated by `.`s into a

       `PropertyAccess(first, second)`

       Or a nested version when there are multiple `.`s

       `PropertyAccess(PropertyAccess(first, second), third)`

    \*\*/

    public static function mergePropertyOperations(pre:Array<ParserTokens>) :Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.push(token);}

                case SetModule(module): {Parser.module = module; post.push(token);}

                case SplitLine: {nextPart(); post.push(token);}

                case Sign(\_ == Little.keywords.PROPERTY\_ACCESS\_SIGN => true): {

                    if (i + 1 >= pre.length) {

                        Little.runtime.throwError(ErrorMessage("Property access cut off by the end of file, block or expression."), Layer.PARSER);

                        return null;

                    }

                    if (post.length == 0) {

                        Little.runtime.throwError(ErrorMessage("Property access cut off by the start of file, block or expression."), Layer.PARSER);

                        return null;

                    }

                    var lookbehind = post.pop();

                    switch lookbehind {

                        case SplitLine | SetLine(\_) | SetModule(\_): {

                            Little.runtime.throwError(ErrorMessage("Property access cut off by the start of a line, or by a line split (; or ,)."), Layer.PARSER);

                            return null;

                        }

                        case Expression(\_, \_): {

                            var field = mergePropertyOperations([pre[++i]])[0];

                            // There are multiple cases, either:

                            // - ().something, in which outright parsing is valid

                            // - p().something, in which we need to generate a function call

                            // - read()().something, the latter gets a little compilcated.

                            // Also, need to handle a.b().c()().d type stuff.

                            var beforePropertyCalls:Array<ParserTokens> = [lookbehind];

                            while (post.length > 0) {

                                var last = post.pop();

                                switch last {

                                    case Identifier(\_) | PropertyAccess(\_, \_): {

                                        beforePropertyCalls.push(last);

                                        break;

                                    }

                                    case Block(body, type): {

                                        beforePropertyCalls.push(Block(mergePropertyOperations(body), mergePropertyOperations([type])[0]));

                                        break;

                                    }

                                    case Expression(parts, type): beforePropertyCalls.push(Expression(mergePropertyOperations(parts), mergePropertyOperations([type])[0]));

                                    case \_: {

                                        post.push(last);

                                        break;

                                    }

                                }

                            }

                            var parent:ParserTokens = lookbehind;

                            if (beforePropertyCalls.length > 0) {

                                parent = beforePropertyCalls.pop();

                                while (beforePropertyCalls.length > 0) {

                                    parent = FunctionCall(parent, beforePropertyCalls.pop());

                                }

                            }

                            post.push(PropertyAccess(parent, field));

                        }

                        case \_: {

                            var field = mergePropertyOperations([pre[++i]])[0];

                            post.push(PropertyAccess(lookbehind, field));

                        }

                    }

                }

                case Block(body, type): post.push(Block(mergePropertyOperations(body), mergePropertyOperations([type])[0]));

                case Expression(parts, type): post.push(Expression(mergePropertyOperations(parts), mergePropertyOperations([type])[0]));

                case Custom(name, params): post.push(Custom(name, params.map(x -> mergePropertyOperations([x])[0])));

                case \_: post.push(token);

            }

            i++;

        }

        resetLines();

        return post;

    }

    /\*\*

        Merges `as <Type>` sequences into `TypeDeclaration(null, <Type>)`

    \*\*/

    public static function mergeTypeDecls(pre:Array<ParserTokens>):Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.push(token);}

                case SetModule(module): {Parser.module = module; post.push(token);}

                case SplitLine: {nextPart(); post.push(token);}

                case Identifier(word): {

                    if (word == Little.keywords.TYPE\_DECL\_OR\_CAST && i + 1 < pre.length) {

                        var lookahead = pre[i + 1];

                        post.push(TypeDeclaration(null, mergeTypeDecls([lookahead])[0]));

                        i++;

                    } else if (word == Little.keywords.TYPE\_DECL\_OR\_CAST) {

                        // Throw error for incomplete type declarations;

                    if (i + 1 == pre.length) {

                        Little.runtime.throwError(ErrorMessage('Incomplete type declaration, make sure to input a type after the `${Little.keywords.TYPE\_DECL\_OR\_CAST}`.'));

                        return null;

                    }

                    } else {

                        post.push(token);

                    }

                }

                case Expression(parts, type): post.push(Expression(mergeTypeDecls(parts), mergeTypeDecls([type])[0]));

                case Block(body, type): post.push(Block(mergeTypeDecls(body), mergeTypeDecls([type])[0]));

                case PropertyAccess(name, property): post.push(PropertyAccess(mergeTypeDecls([name])[0], mergeTypeDecls([property])[0]));

                case Custom(name, params): post.push(Custom(name, params.map(x -> mergeTypeDecls([x])[0])));

                case \_: post.push(token);

            }

            i++;

        }

        resetLines();

        return post;

    }

    /\*\*

        Merges many complex seqences into single tokens:

         - `define <name> [as <Type>]` -> `VariableCreation()`

         - `action <name>(<params>) [as <Type>]` -> `FunctionCreation()`

         - and more...

    \*\*/

    public static function mergeComplexStructures(pre:Array<ParserTokens>):Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var currentDoc:ParserTokens = null;

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.push(token);}

                case SetModule(module): {Parser.module = module; post.push(token);}

                case SplitLine: {nextPart(); post.push(token);}

                case Documentation(doc): currentDoc = token;

                case Identifier(\_ == Little.keywords.VARIABLE\_DECLARATION => true): {

                    i++;

                    if (i >= pre.length) {

                        Little.runtime.throwError(ErrorMessage("Missing variable name, variable is cut off by the end of the file, block or expression."), Layer.PARSER);

                        return null;

                    }

                    var name:ParserTokens = null;

                    var type:ParserTokens = null;

                    while (i < pre.length) {

                        var lookahead = pre[i];

                        switch lookahead {

                            case TypeDeclaration(\_, typeToken): {

                                if (name == null) {

                                    Little.runtime.throwError(ErrorMessage("Missing variable name before type declaration."), Layer.PARSER);

                                    return null;

                                }

                                type = typeToken;

                                break;

                            }

                            case SetLine(\_) | SetModule(\_) | SplitLine | Sign("="): i--; break;

                            case Block(body, type): {

                                if (name == null) name = Block(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else if (type == null) type = Block(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else {

                                    i--;

                                    break;

                                }

                            }

                            case Expression(body, type): {

                                if (name == null) name = Expression(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else if (type == null) type = Expression(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else {

                                    i--;

                                    break;

                                }

                            }

                            case \_: {

                                if (name == null) name = lookahead;

                                else if (type == null && lookahead.is(TYPE\_DECLARATION)) type = lookahead;

                                else {

                                    i--;

                                    break;

                                }

                            }

                        }

                        i++;

                    }

                    if (name == null)

                        Little.runtime.throwError(ErrorMessage("Missing variable name, variable is cut off by the end of the file, block or expression."), Layer.PARSER);

                    post.push(Variable(name, type, currentDoc));

                    currentDoc = null;

                }

                case Identifier(\_ == Little.keywords.FUNCTION\_DECLARATION => true): {

                    i++;

                    if (i >= pre.length) {

                        Little.runtime.throwError(ErrorMessage("Missing function name, function is cut off by the end of the file, block or expression."), Layer.PARSER);

                        return null;

                    }

                    if (i + 1 >= pre.length) {

                        Little.runtime.throwError(ErrorMessage("Missing function parameter body, function is cut off by the end of the file, block or expression."), Layer.PARSER);

                        return null;

                    }

                    var name:ParserTokens = null;

                    var params:ParserTokens = null;

                    var type:ParserTokens = null;

                    while (i < pre.length) {

                        var lookahead = pre[i];

                        switch lookahead {

                            case TypeDeclaration(\_, typeToken): {

                                if (name == null) {

                                    Little.runtime.throwError(ErrorMessage("Missing function name and parameters before type declaration."), Layer.PARSER);

                                    return null;

                                }

                                else if (params == null) {

                                    Little.runtime.throwError(ErrorMessage("Missing function parameters before type declaration."), Layer.PARSER);

                                    return null;

                                }

                                type = mergeComplexStructures([typeToken])[0];

                                break;

                            }

                            case Sign("="): i--; break;

                            case Block(body, type): {

                                if (name == null) name = Block(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else if (params == null) params = Block(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else if (type == null) type = Block(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else {

                                    break;

                                }

                            }

                            case Expression(body, type): {

                                if (name == null) name = Expression(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else if (params == null) params = Expression(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else if (type == null) type = Expression(mergeComplexStructures(body), mergeComplexStructures([type])[0]);

                                else {

                                    break;

                                }

                            }

                            case \_: {

                                if (name == null) name = lookahead;

                                else if (params == null) params = lookahead;

                                else if (type == null && lookahead.getName() == "TypeDeclaration") type = mergeComplexStructures([lookahead.parameter(1)])[0];

                                else {

                                    break;

                                }

                            }

                        }

                        i++;

                    }

                    if (name == null)

                        Little.runtime.throwError(ErrorMessage("Missing function name and parameters, function is cut off by the end of the file, block or expression."), Layer.PARSER);

                    else if (params == null)

                        Little.runtime.throwError(ErrorMessage("Missing function parameters, function is cut off by the end of the file, block or expression."), Layer.PARSER);

                    post.push(Function(name, params, type, currentDoc));

                    currentDoc = null;

                }

                case Identifier(\_ == Little.keywords.FUNCTION\_RETURN => true): {

                    i++;

                    if (i >= pre.length) {

                        Little.runtime.throwError(ErrorMessage("Missing return value, value is cut off by the end of the file, block or expression."), Layer.PARSER);

                        return null;

                    }

                    var valueToReturn:Array<ParserTokens> = [];

                    while (i < pre.length) {

                        var lookahead = pre[i];

                        switch lookahead {

                            case SetLine(\_) | SetModule(\_) | SplitLine: i--; break;

                            case Block(body, type): {

                                valueToReturn.push(Block(mergeComplexStructures(body), mergeComplexStructures([type])[0]));

                            }

                            case Expression(body, type): {

                                valueToReturn.push(Expression(mergeComplexStructures(body), mergeComplexStructures([type])[0]));

                            }

                            case \_: valueToReturn.push(lookahead);

                        }

                        i++;

                    }

                    post.push(Return(if (valueToReturn.length == 1) valueToReturn[0] else Expression(valueToReturn.copy(), null), null));

                }

                case Identifier(\_): { // Conditions are definable, or at least, developers can register them dynamically, we need to look for the syntax: Identifier -> Expression -> Block.

                    i++;

                    var name:ParserTokens = Identifier(token.parameter(0));

                    var exp:ParserTokens = null;

                    var body:ParserTokens = null;

                    var fallback = i - 1; // Reason for -1 here is because of the lookahead - if this isn't a condition, i-1 is pushed and i is the next token.

                    while (true) {

                        if (body != null) break;

                        if (i >= pre.length) {

                            i = fallback;

                            break;

                        }

                        var lookahead = pre[i];

                        switch lookahead {

                            case SplitLine | SetModule(\_) | SetLine(\_): { // Encountering a hard split in any place breaks the sequence (if (), {}, if, () {})

                            if (exp != null && body != null) break;

                                i = fallback;

                                break;

                            }

                            case Block(b, type): {

                                if (exp == null) {

                                    i = fallback;

                                    break;

                                }

                                else if (body == null) body = Block(mergeComplexStructures(b), mergeComplexStructures([type])[0]);

                            }

                            case Expression(parts, type): {

                                if (exp == null) exp = PartArray(mergeComplexStructures(parts));

                                else if (body == null) {

                                    i = fallback;

                                    break;

                                }

                            }

                            case \_: {

                                if (exp == null || body == null) {

                                    i = fallback;

                                    break;

                                }

                            }

                        }

                        i++;

                    }

                    if (i == fallback) {

                        post.push(token);

                    } else {

                        i -= 1;

                        post.push(ConditionCall(name, exp, body));

                        currentDoc = null;

                    }

                }

                case Expression(parts, type): post.push(Expression(mergeComplexStructures(parts), mergeComplexStructures([type])[0]));

                case Block(body, type): post.push(Block(mergeComplexStructures(body), mergeComplexStructures([type])[0]));

                case PropertyAccess(name, property): post.push(PropertyAccess(mergeComplexStructures([name])[0], mergeComplexStructures([property])[0]));

                case Custom(name, params): post.push(Custom(name, params.map(x -> mergeComplexStructures([x])[0])));

                case \_: post.push(token);

            }

            i++;

        }

        resetLines();

        return post;

    }

    /\*\*

        Merges a token followed by an expression immediately into a `FunctionCall()`

    \*\*/

    public static function mergeCalls(pre:Array<ParserTokens>):Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.push(token);}

                case SetModule(module): {Parser.module = module; post.push(token);}

                case SplitLine: {nextPart(); post.push(token);}

                case Expression(parts, type): {

                    parts = mergeCalls(parts);

                    if (i == 0) {

                        post.push(Expression(parts, type));

                    } else {

                        var lookbehind = pre[i - 1];

                        switch lookbehind {

                            case Sign(\_) | SplitLine | SetLine(\_) | SetModule(\_): post.push(Expression(parts, type));

                            case \_: {

                                var previous = post.pop(); // When parsing a function that returns a function, this handles the "nested call" correctly

                                token = PartArray(parts);

                                post.push(FunctionCall(previous, token));

                            }

                        }

                    }

                }

                case Block(body, type): post.push(Block(mergeCalls(body), mergeCalls([type])[0]));

                case Variable(name, type, doc): post.push(Variable(mergeCalls([name])[0], mergeCalls([type])[0], mergeCalls([doc])[0]));

                case Function(name, params, type, doc): post.push(Function(mergeCalls([name])[0], mergeCalls([params])[0], mergeCalls([type])[0], mergeCalls([doc])[0]));

                case ConditionCall(name, exp, body): post.push(ConditionCall(mergeCalls([name])[0], mergeCalls([exp])[0], mergeCalls([body])[0]));

                case Return(value, type): post.push(Return(mergeCalls([value])[0], mergeCalls([type])[0]));

                case PropertyAccess(name, property): post.push(PropertyAccess(mergeCalls([name])[0], mergeCalls([property])[0]));

                case PartArray(parts): post.push(PartArray(mergeCalls(parts)));

                case Custom(name, params): post.push(Custom(name, params.map(x -> mergeCalls([x])[0])));

                case \_: post.push(token);

            }

            i++;

        }

        resetLines();

        return post;

    }

    /\*\*

        Merges a chain of single tokens separated by a `=` into a `Write([<sequence>], <end of sequence>)`

    \*\*/

    public static function mergeWrites(pre:Array<ParserTokens>):Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var potentialAssignee:ParserTokens = NullValue;

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {

                    setLine(line);

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = token;

                }

                case SetModule(module): {

                    Parser.module = module;

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = token;

                }

                case SplitLine: {

                    nextPart();

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = token;

                }

                case Sign("="): {

                    if (i + 1 >= pre.length) {

                        Little.runtime.throwError(ErrorMessage("Missing value after the `=`"), Layer.PARSER);

                        return null;

                    }

                    var currentAssignee:Array<ParserTokens> = [potentialAssignee];

                    var assignees = [currentAssignee.length == 1 ? currentAssignee[0] : Expression(currentAssignee.copy(), null)];

                    currentAssignee = [];

                    var value:ParserTokens;

                    while (i + 1 < pre.length) {

                        var lookahead = pre[i + 1];

                        switch lookahead {

                            case Sign("="): {

                                var assignee = currentAssignee.length == 1 ? currentAssignee[0] : Expression(currentAssignee.copy(), null);

                                assignees.push(assignee);

                                currentAssignee = [];

                            }

                            case SplitLine | SetLine(\_) | SetModule(\_): break;

                            case \_: currentAssignee.push(lookahead);

                        }

                        i++;

                    }

                    if (currentAssignee.length == 0) {

                        Little.runtime.throwError(ErrorMessage("Missing value after the last `=`"), Layer.PARSER);

                        return null;

                    }

                    // The last currentAssignee is the value;

                    value = if (currentAssignee.length == 1) currentAssignee[0] else Expression(currentAssignee, null);

                    var fValue = mergeWrites([value]);

                    var v = if (fValue.length == 1) fValue[0] else Expression(fValue, null);

                    post.push(Write(assignees, v));

                    potentialAssignee = null;

                }

                case Expression(parts, type): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = Expression(mergeWrites(parts), mergeWrites([type])[0]);

                }

                case PartArray(parts): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = PartArray(mergeWrites(parts));

                }

                case Block(body, type): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = Block(mergeWrites(body), mergeWrites([type])[0]);

                }

                case Variable(name, type, doc): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = Variable(mergeWrites([name])[0], mergeWrites([type])[0], mergeWrites([doc])[0]);

                }

                case Function(name, params, type, doc): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = Function(mergeWrites([name])[0], mergeWrites([params])[0], mergeWrites([type])[0], mergeWrites([doc])[0]);

                }

                case ConditionCall(name, exp, body): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = ConditionCall(mergeWrites([name])[0], mergeWrites([exp])[0], mergeWrites([body])[0]);

                }

                case Return(value, type): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = Return(mergeWrites([value])[0], mergeWrites([type])[0]);

                }

                case FunctionCall(name, params): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = FunctionCall(mergeWrites([name])[0], mergeWrites([params])[0]);

                }

                case PropertyAccess(name, property): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = PropertyAccess(mergeWrites([name])[0], mergeWrites([property])[0]);

                }

                case Custom(name, params): {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = Custom(name, params.map(x -> mergeWrites([x])[0]));

                }

                case \_: {

                    if (potentialAssignee != null) post.push(potentialAssignee);

                    potentialAssignee = token;

                }

            }

            i++;

        }

        if (potentialAssignee != null) post.push(potentialAssignee);

        post.shift();

        resetLines();

        return post;

    }

    /\*\*

        Merges `<token>, TypeDeclaration(null, <type>)` into `TypeDeclaration(<token>, <type>)`

    \*\*/

    public static function mergeValuesWithTypeDecls(pre:Array<ParserTokens>) :Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var i = pre.length - 1;

        while (i >= 0) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.unshift(token);}

                case SetModule(module): {Parser.module = module; post.unshift(token);}

                case SplitLine: {nextPart(); post.unshift(token);}

                case TypeDeclaration(null, type): {

                    if (i-- <= 0) {

                        Little.runtime.throwError(ErrorMessage("Value's type declaration cut off by the start of file, block or expression."), Layer.PARSER);

                        return null;

                    }

                    var lookbehind = pre[i];

                    switch lookbehind {

                        case SplitLine | SetLine(\_) | SetModule(\_): {

                            Little.runtime.throwError(ErrorMessage("Value's type declaration access cut off by the start of a line, or by a line split (; or ,)."), Layer.PARSER);

                            return null;

                        }

                        case \_: {

                            post.unshift(TypeDeclaration(lookbehind, type));

                        }

                    }

                }

                case Block(body, type): post.unshift(Block(mergeValuesWithTypeDecls(body), mergeValuesWithTypeDecls([type])[0]));

                case Expression(parts, type): post.unshift(Expression(mergeValuesWithTypeDecls(parts), mergeValuesWithTypeDecls([type])[0]));

                case Variable(name, type, doc): post.unshift(Variable(mergeValuesWithTypeDecls([name])[0], mergeValuesWithTypeDecls([type])[0], mergeValuesWithTypeDecls([doc])[0]));

                case Function(name, params, type, doc): post.unshift(Function(mergeValuesWithTypeDecls([name])[0], mergeValuesWithTypeDecls([params])[0], mergeValuesWithTypeDecls([type])[0], mergeValuesWithTypeDecls([doc])[0]));

                case ConditionCall(name, exp, body): post.unshift(ConditionCall(mergeValuesWithTypeDecls([name])[0], mergeValuesWithTypeDecls([exp])[0], mergeValuesWithTypeDecls([body])[0]));

                case Return(value, type): post.unshift(Return(mergeValuesWithTypeDecls([value])[0], mergeValuesWithTypeDecls([type])[0]));

                case PartArray(parts): post.unshift(PartArray(mergeValuesWithTypeDecls(parts)));

                case FunctionCall(name, params): post.unshift(FunctionCall(mergeValuesWithTypeDecls([name])[0], mergeValuesWithTypeDecls([params])[0]));

                case Write(assignees, value): post.unshift(Write(mergeValuesWithTypeDecls(assignees), mergeValuesWithTypeDecls([value])[0]));

                case PropertyAccess(name, property): post.unshift(PropertyAccess(mergeValuesWithTypeDecls([name])[0], mergeValuesWithTypeDecls([property])[0]));

                case Custom(name, params): post.unshift(Custom(name, params.map(x -> mergeValuesWithTypeDecls([x])[0])));

                case \_: post.unshift(token);

            }

            i--;

        }

        resetLines();

        return post;

    }

    /\*\*

        Merges tokens that expect a `Block` as it's last parameter with the next token, if that last parameter is `null`.

        Comes into play with condition calls:

        ```

        if (<exp>) <exp>

        ```

    \*\*/

    public static function mergeNonBlockBodies(pre:Array<ParserTokens>):Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.push(token);}

                case SetModule(module): {Parser.module = module; post.push(token);}

                case SplitLine: {nextPart(); post.push(token);}

                case FunctionCall(name, params): {

                    if (i + 1 >= pre.length) {

                        post.push(FunctionCall(mergeNonBlockBodies([name])[0], mergeNonBlockBodies([params])[0]));

                        i++;

                        continue;

                    }

                    var lookahead = pre[i + 1];

                    switch lookahead {

                        case SetLine(\_) | SplitLine | SetModule(\_) | Sign(\_): {

                            post.push(FunctionCall(mergeNonBlockBodies([name])[0], mergeNonBlockBodies([params])[0]));

                        }

                        case \_: {

                            post.push(ConditionCall(mergeNonBlockBodies([name])[0], mergeNonBlockBodies([params])[0], mergeNonBlockBodies([lookahead])[0]));

                            i++; // We consumed the lookahead, so we need to increment to its position, so that the final i++ gets to the next, correct, token.

                        }

                    }

                }

                case Block(body, type): post.push(Block(mergeNonBlockBodies(body), mergeNonBlockBodies([type])[0]));

                case Expression(parts, type): post.push(Expression(mergeNonBlockBodies(parts), mergeNonBlockBodies([type])[0]));

                case Variable(name, type, doc): post.push(Variable(mergeNonBlockBodies([name])[0], mergeNonBlockBodies([type])[0], mergeNonBlockBodies([doc])[0]));

                case Function(name, params, type, doc): post.push(Function(mergeNonBlockBodies([name])[0], mergeNonBlockBodies([params])[0], mergeNonBlockBodies([type])[0], mergeNonBlockBodies([doc])[0]));

                case ConditionCall(name, exp, body): post.push(ConditionCall(mergeNonBlockBodies([name])[0], mergeNonBlockBodies([exp])[0], mergeNonBlockBodies([body])[0]));

                case Return(value, type): post.push(Return(mergeNonBlockBodies([value])[0], mergeNonBlockBodies([type])[0]));

                case PartArray(parts): post.push(PartArray(mergeNonBlockBodies(parts)));

                case Write(assignees, value): post.push(Write(mergeNonBlockBodies(assignees), mergeNonBlockBodies([value])[0]));

                case PropertyAccess(name, property): post.push(PropertyAccess(mergeNonBlockBodies([name])[0], mergeNonBlockBodies([property])[0]));

                case Custom(name, params): post.push(Custom(name, params.map(x -> mergeNonBlockBodies([x])[0])));

                case \_: post.push(token);

            }

            i++;

        }

        resetLines();

        return post;

    }

    /\*\*

        Macro that adds support for `if`-`else` patterns.

    \*\*/

    public static function mergeElses(pre:Array<ParserTokens>):Array<ParserTokens> {

        if (pre == null) return null;

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<ParserTokens> = [];

        var i = 0;

        while (i < pre.length) {

            var token = pre[i];

            switch token {

                case SetLine(line): {setLine(line); post.push(token);}

                case SetModule(module): {Parser.module = module; post.push(token);}

                case SplitLine: {nextPart(); post.push(token);}

                case Identifier(\_ == Little.keywords.CONDITION\_\_ELSE => true): {

                    if (post.length == 0 || !post[post.length - 1].is(CONDITION\_CALL)) {

                        post.push(token);

                        i++;

                        continue;

                    }

                    if (i + 1 >= pre.length) {

                        Little.runtime.throwError(ErrorMessage('Condition has no body, body may be cut off by the end of file, block or expression.'), PARSER);

                        return null;

                    }

                    var exp:ParserTokens = post[post.length - 1].parameter(1); //Condition(name:ParserTokens, ->exp:ParserTokens<-, body:ParserTokens, type:ParserTokens)

                    exp = Expression([exp, Sign("!="), TrueValue], null);

                    i++;

                    var body:ParserTokens = pre[i];

                    switch body {

                        case SplitLine: {

                            Little.runtime.throwError(ErrorMessage('`${Little.keywords.CONDITION\_\_ELSE}` condition has no body, body cut off by a line split, or does not exist'), PARSER);

                            return null;

                        }

                        case SetLine(\_) | SetModule(\_): {

                            Little.runtime.throwError(ErrorMessage('`${Little.keywords.CONDITION\_\_ELSE}` condition has no body, body cut off by a new line, or does not exist'), PARSER);

                            return null;

                        }

                        case ConditionCall(Identifier("if"), exp2, body): post.push(ConditionCall(Identifier("if"), Expression([exp, Sign("&&"), exp2], null) , !body.is(BLOCK) ? Block([body], null) : body));

                        case \_: post.push(ConditionCall(Identifier("if"), exp, !body.is(BLOCK) ? Block([body], null) : body));

                    }

                }

                case Block(body, type): post.push(Block(mergeElses(body), mergeElses([type])[0]));

                case Expression(parts, type): post.push(Expression(mergeElses(parts), mergeElses([type])[0]));

                case Variable(name, type, doc): post.push(Variable(mergeElses([name])[0], mergeElses([type])[0], mergeElses([doc])[0]));

                case Function(name, params, type, doc): post.push(Function(mergeElses([name])[0], mergeElses([params])[0], mergeElses([type])[0], mergeElses([doc])[0]));

                case ConditionCall(name, exp, body): post.push(ConditionCall(mergeElses([name])[0], mergeElses([exp])[0], mergeElses([body])[0]));

                case Return(value, type): post.push(Return(mergeElses([value])[0], mergeElses([type])[0]));

                case PartArray(parts): post.push(PartArray(mergeElses(parts)));

                case FunctionCall(name, params): post.push(FunctionCall(mergeElses([name])[0], mergeElses([params])[0]));

                case Write(assignees, value): post.push(Write(mergeElses(assignees), mergeElses([value])[0]));

                case PropertyAccess(name, property): post.push(PropertyAccess(mergeElses([name])[0], mergeElses([property])[0]));

                case Custom(name, params): post.push(Custom(name, params.map(x -> mergeElses([x])[0])));

                case \_: post.push(token);

            }

            i++;

        }

        resetLines();

        return post;

    }

    static var line(get, set):Int;

    /\*\* Parser.line getter \*\*/ static function get\_line() return Little.runtime.line;

    /\*\* Parser.line setter \*\*/ static function set\_line(l:Int) return Little.runtime.line = l;

    static var module(get, set):String;

    /\*\* Parser.module getter \*\*/ static function get\_module() return Little.runtime.module;

    /\*\* Parser.module setter \*\*/ static function set\_module(l:String) return Little.runtime.module = l;

    static var linePart:Int = 0;

    /\*\*

        Changes the current line. Used only for error reporting.

    \*\*/

    static function setLine(l:Int) {

        line = l;

        linePart = 0;

    }

    /\*\*

        Changes the current line part. Used only for error reporting.

    \*\*/

    static function nextPart() linePart++;

    /\*\*

        Resets the line counters, used between parse stages.

    \*\*/

    static function resetLines() {

        line = 0;

        linePart = 0;

    }

}

package little.interpreter;

import little.tools.OrderedMap;

import little.interpreter.memory.MemoryPointer;

enum InterpTokens {

    SetLine(line:Int);

    SetModule(module:String);

    SplitLine;

    /\*\*

        Usage:

        @param name `Identifier`, `PropertyAccess`

        @param type `Identifier`, `PropertyAccess`

        @param doc `Characters`

    \*\*/

    VariableDeclaration(name:InterpTokens, type:InterpTokens, ?doc:InterpTokens);

    /\*\*

        Usage:

        @param name `Identifier`, `PropertyAccess`

        @param params `PartArray([\*, SplitLine, \*])`, `PartArray([\*, SetLine, \*])`, `PartArray([\*, SetLine, \*, SplitLine, \*])`, `PartArray([\*])`, `PartArray([])`

        @param type `Identifier`, `PropertyAccess`

        @param doc `Characters`

    \*\*/

    FunctionDeclaration(name:InterpTokens, params:InterpTokens, type:InterpTokens, ?doc:InterpTokens);

    /\*\*

        `callers` is a map of `InterpTokens` configs representing the structure of the condition itself, in correlation to the conditions outcome.

        Use haxe `null` to denote a wildcard - a free value decided by the user.

        for example, Little's for loop would be:

            [

                [VariableDeclaration(null, null, null), Identifier("from"), null, Identifier("to"), null, Identifier("jump"), null] => ...,

                [VariableDeclaration(null, null, null), Identifier("from"), null, Identifier("to"), null] => ...

            ]

        Ideally, to validate the "`null`" tokens (the wildcard ones) one will use the macro-ish tools the language provide (extracting type, extracting identifiers...)

**\*\*Important\*\*** - to define a "dynamic" condition (that accepts any number of parameters) you provide a `null` pattern key.

        The actual `Block` that decides how an if to run the code associated with the condition should expect two defined parameters of `InterpTokens.Characters`'s type,

        one named `Little.keywords.CONDITION\_PATTERN\_PARAMETER\_NAME` and one named `Little.keywords.CONDITION\_BODY\_PARAMETER\_NAME`.

        @param callers `Map<Array<InterpTokens.\*>, InterpTokens.Block>`

    \*\*/

    ConditionCode(callers:Map<Array<InterpTokens>, InterpTokens>);

    /\*\*

        Usage:

        @param name `Identifier`, `PropertyAccess`

        @param exp `PartArray`

        @param body `Block`

    \*\*/

    ConditionCall(name:InterpTokens, exp:InterpTokens, body:InterpTokens);

    /\*\*

        Usage:

        @param requiredParams `OrderedMap<String, InterpTokens.Identifier>`

        @param body `Block`

    \*\*/

    FunctionCode(requiredParams:OrderedMap<String, InterpTokens>, body:InterpTokens);

    /\*\*

        Usage:

        @param name `Identifier`, `PropertyAccess`

        @param params `PartArray([\*, SplitLine, \*])`, `PartArray([\*, SetLine, \*])`, `PartArray([\*, SetLine, \*, SplitLine, \*])`, `PartArray([\*])`, `PartArray([])`

    \*\*/

    FunctionCall(name:InterpTokens, params:InterpTokens);

    /\*\*

        Usage:

        @param value `Identifier`, `PropertyAccess`

        @param type `Identifier`, `PropertyAccess`

    \*\*/

    FunctionReturn(value:InterpTokens, type:InterpTokens);

    /\*\*

        Usage:

        @param assignees `Array<InterpTokens.Identifier>`

        @param value `\*`

    \*\*/

    Write(assignees:Array<InterpTokens>, value:InterpTokens);

    /\*\*

        Usage:

        @param value `\*`

        @param type `Identifier`, `PropertyAccess`

    \*\*/

    TypeCast(value:InterpTokens, type:InterpTokens);

    /\*\*

        Usage:

        @param parts `Array<InterpTokens.\*>`

        @param type `Identifier`, `PropertyAccess`

    \*\*/

    Expression(parts:Array<InterpTokens>, type:InterpTokens);

    /\*\*

        Usage:

        @param body `Array<InterpTokens.\*>`

        @param type `Identifier`, `PropertyAccess`

    \*\*/

    Block(body:Array<InterpTokens>, type:InterpTokens);

    /\*\*

        Usage:

        @param parts `Array<InterpTokens.\*>`

    \*\*/

    PartArray(parts:Array<InterpTokens>);

    /\*\*

        Usage:

        @param name `Identifier`, `PropertyAccess`

        @param property `Identifier`, `Number`, `Decimal`, `Characters`, `Sign`, `NullValue`, `TrueValue`, `FalseValue`

    \*\*/

    PropertyAccess(name:InterpTokens, property:InterpTokens);

    /\*\*Int32\*\*/

    Number(num:Int);

    /\*\*Float64\*\*/

    Decimal(num:Float);

    /\*\*String UTF8\*\*/

    Characters(string:String);

    /\*\*String UTF8\*\*/

    Documentation(doc:String);

    /\*\*32/64bit memory pointer\*\*/

    ClassPointer(pointer:MemoryPointer);

    /\*\*String UTF8\*\*/

    Sign(sign:String);

    /\*\*`null`\*\*/

    NullValue;

    /\*\*`true`\*\*/

    TrueValue;

    /\*\*`false`\*\*/

    FalseValue;

    /\*\*

        Usage:

        @param word `String`

    \*\*/

    Identifier(word:String);

    /\*\*

        - `props`' elements may either be a `Object`, a `FunctionCode`, or a **\*\*statically storable\*\*** object.

        - `typeName` must be a `String`, containing a proper, accessible type.

    \*\*/

    Object(props:Map<String, {documentation:String, value:InterpTokens}>, typeName:String);

    /\*\*

        Used for errors & warnings

    \*\*/

    ErrorMessage(msg:String);

    /\*\*

        DO NOT USE. Necessary for very specific cases (extern function calls when params are required)

    \*\*/

    HaxeExtern(func:Void -> InterpTokens);

}

package little.interpreter;

import haxe.Rest;

import little.interpreter.Tokens.InterpTokens;

import little.tools.PrettyPrinter;

import little.tools.Layer;

import little.Little.memory;

import haxe.extern.EitherType;

import little.tools.OrderedMap;

using StringTools;

using Std;

using Math;

using little.tools.TextTools;

using little.tools.Extensions;

@:access(little.interpreter.Runtime)

class Interpreter {

    public static function convert(pre:Rest<little.parser.Tokens.ParserTokens>):Array<InterpTokens> {

        if (pre.length == 1 && pre[0] == null) return [null];

        var post:Array<InterpTokens> = [];

        for (item in pre) {

            post.push(switch item {

                case SetLine(line): SetLine(line);

                case SetModule(module): SetModule(module);

                case SplitLine: SplitLine;

                case Variable(name, type, doc): VariableDeclaration(convert(name)[0], type == null ? Little.keywords.TYPE\_UNKNOWN.asTokenPath() : convert(type)[0], doc == null ? Characters("") : convert(doc)[0]);

                case Function(name, params, type, doc): FunctionDeclaration(convert(name)[0], convert(params)[0], type == null ? Little.keywords.TYPE\_UNKNOWN.asTokenPath() : convert(type)[0], doc == null ? Characters("") : convert(doc)[0]);

                case ConditionCall(name, exp, body): ConditionCall(convert(name)[0], convert(exp)[0], convert(body)[0]);

                case Read(name): null;

                case Write(assignees, value): Write(convert(...assignees), convert(value)[0]);

                case Identifier(word): Identifier(word);

                case TypeDeclaration(value, type): TypeCast(convert(value)[0], convert(type)[0]);

                case FunctionCall(name, params): FunctionCall(convert(name)[0], convert(params)[0]);

                case Return(value, type): FunctionReturn(convert(value)[0], type == null ? Little.keywords.TYPE\_UNKNOWN.asTokenPath() : convert(type)[0]);

                case Expression(parts, type): Expression(convert(...parts), type == null ? Little.keywords.TYPE\_UNKNOWN.asTokenPath() : convert(type)[0]);

                case Block(body, type): Block(convert(...body), type == null ? Little.keywords.TYPE\_UNKNOWN.asTokenPath() : convert(type)[0]);

                case PartArray(parts): PartArray(convert(...parts));

                case PropertyAccess(name, property): PropertyAccess(convert(name)[0], convert(property)[0]);

                case Sign(sign): Sign(sign);

                case Number(num): num.parseFloat().abs() > 2\_147\_483\_647 ? Decimal(num.parseFloat()) : Number(num.parseInt());

                case Decimal(num): Decimal(num.parseFloat());

                case Characters(string): Characters(string);

                case Documentation(doc): Characters('""$doc""'); // Kinda strange behavior, should maybe disable entirely/throw an error.

                case ErrorMessage(msg): ErrorMessage(msg);

                case NullValue: NullValue;

                case TrueValue: TrueValue;

                case FalseValue: FalseValue;

                case Custom(name, params): throw 'Custom tokens cannot remain when transitioning from Parser to Interpreter tokens (found $item)';

            });

        }

        return post;

    }

    /\*\*

        Raise an error in the program, with the given message.

        @param message The error message

        @param layer The layer of the error. see `little.tools.Layer`.

        @return the error token, as InterpTokens.ErrorMessage(msg:String)

    \*\*/

    public static function error(message:String, layer:Layer = INTERPRETER):InterpTokens {

        Little.runtime.throwError(ErrorMessage(message), layer);

        throw "";

        return ErrorMessage(message);

    }

    /\*\*

        Raise a warning in the program, with the given message. A warning never stops execution.

        @param message The warning message

        @param layer The layer of the warning. see `little.tools.Layer`.

        @return the warning token, as InterpTokens.ErrorMessage(msg:String)

    \*\*/

    public static function warn(message:String, layer:Layer = INTERPRETER):InterpTokens {

        Little.runtime.warn(ErrorMessage(message), layer);

        return ErrorMessage(message);

    }

    /\*\*

        If `token` is not of type `isType`, throw an error.

        @param token the token to check

        @param isType the type to check for

        @param errorMessage the error message to throw if `token` is not of type `isType`

    \*\*/

    public static function assert(token:InterpTokens, isType:EitherType<InterpTokensSimple, Array<InterpTokensSimple>>, ?errorMessage:String = null) {

        if ((isType is InterpTokensSimple && !token.is(isType)) || (isType is Array && !isType.containsAny(a -> token.is(a)))) {

            Little.runtime.throwError(errorMessage != null ? ErrorMessage(errorMessage) : ErrorMessage('Assertion failed, token $token is not of type $isType'), INTERPRETER);

            return NullValue;

        }

        return token;

    }

    /\*\*

        Set the current line of the program

    \*\*/

    public static function setLine(l:Int) {

        var o = Little.runtime.line;

        Little.runtime.line = l;

        Little.runtime.linePart = 0;

        for (listener in Little.runtime.onLineChanged) listener(o);

        for (listener in Little.runtime.onLineSplit) listener();

    }

    /\*\*

        Set the current module of the program

    \*\*/

    public static function setModule(m:String) {

        var o = Little.runtime.module;

        Little.runtime.module = m;

        if (o != m) for (listener in Little.runtime.onModuleChanged) listener(o);

    }

    /\*\*

        Split the current line. In other words, create a new line, but keep the old line number.

    \*\*/

    public static function splitLine() {

        Little.runtime.linePart++;

        for (listener in Little.runtime.onLineSplit) listener();

    }

    /\*\*

        Declare a new variable. That variable will be added to the current scope.

        @param name The name of the variable. Can be any token stringifiyable via `token.extractIdentifier()`.

        @param type The type of the variable. Can be any token stringifiyable via `token.extractIdentifier()`.

        @param doc The documentation of the variable. Should be a `InterpTokens.Documentation(doc:String)`

    \*\*/

    public static function declareVariable(name:InterpTokens, type:InterpTokens, doc:InterpTokens) {

        var path = name.asStringPath();

        memory.write(path, NullValue, type.extractIdentifier(), doc != null ? evaluate(doc).extractIdentifier() : "");

        for (listener in Little.runtime.onFieldDeclared)

            listener(name.asJoinedStringPath(), VARIABLE);

    }

    /\*\*

        Declare a new function. That function will be added to the current scope.

        @param name The name of the function. Can be any token stringifiyable via `token.extractIdentifier()`.

        @param params The parameters of the function. Should be a `InterpTokens.PartArray(parts:Array<InterpTokens>)`

        @param doc The documentation of the function. Should be a `InterpTokens.Documentation(doc:String)`

    \*\*/

    public static function declareFunction(name:InterpTokens, params:InterpTokens, doc:InterpTokens) {

        var path = name.asStringPath();

        var paramMap = new OrderedMap<String, InterpTokens>();

        // Because values are allowed as well as types, were gonna abuse TypeCasts:

        var array = (params.parameter(0) : Array<InterpTokens>);

        for (entry in array) {

            if (entry.is(SPLIT\_LINE, SET\_LINE)) continue;

            switch entry {

                case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, Identifier(Little.keywords.TYPE\_UNKNOWN));

                case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, type);

                case Write(assignees, value): {

                    switch assignees[0] {

                        case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(value, Identifier(Little.keywords.TYPE\_UNKNOWN));

                        case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(value, type);

                        default:

                    }

                }

                default:

            }

        }

        memory.write(path, FunctionCode(paramMap, Block([], Identifier(Little.keywords.TYPE\_UNKNOWN))), Little.keywords.TYPE\_FUNCTION, doc != null ? evaluate(doc).extractIdentifier() : "");

        for (listener in Little.runtime.onFieldDeclared)

            listener(name.asJoinedStringPath(), FUNCTION);

    }

    /\*\*

        Calls a condition. The condition's `body` is repeated `0` to `n` times, depending on the condition's `conditionParams`.

**\*\*Important\*\*** - Conditions are not functions, and thus they propagate `return`s.

        @param pattern The pattern of the condition. Should be a `InterpTokens.PartArray(parts:Array<InterpTokens>)`

        @param body The body of the condition. Should be a `InterpTokens.Block(body:Array<InterpTokens>)`

    \*\*/

    public static function condition(name:InterpTokens, pattern:InterpTokens, body:InterpTokens):InterpTokens {

        var conditionToken = memory.read(...name.asStringPath());

        trace(conditionToken, name.asStringPath(), body);

        assert(conditionToken.objectValue, CONDITION\_CODE, '${name.asStringPath()} is not a condition.');

        var patterns:Map<Array<InterpTokens>, InterpTokens> = conditionToken.objectValue.parameter(0);

        var givenPattern = pattern.parameter(0);

        function fit(given:Array<InterpTokens>, pattern:Array<InterpTokens>, currentlyFits:Bool = true):Bool {

            for (i in 0...given.length) {

                if (pattern[i] == null) continue;

                if (given[i].equals(pattern[i])) continue;

                if (given[i].getName() != pattern[i].getName()) return false;

                switch given[i] {

                    case SetLine(\_) | Number(\_) | Decimal(\_) | Characters(\_) | Documentation(\_) | Sign(\_) | Identifier(\_) | ErrorMessage(\_): if (pattern[i].parameter(0) != null) return false;

                    case VariableDeclaration(\_, \_, \_) | FunctionDeclaration(\_, \_, \_, \_): currentlyFits = currentlyFits && fit(cast given[i].getParameters(), cast pattern[i].getParameters(), currentlyFits);

                    case ConditionCode(\_): return false; // Cant be matched with, only valid in the context of a condition definition, which is not supported. Represented by other tokens in other cases

                    case FunctionCode(\_, \_): return false; // same as above

                    case ConditionCall(\_, \_, \_) | FunctionCall(\_, \_): currentlyFits = currentlyFits && fit(cast given[i].getParameters(), cast pattern[i].getParameters(), currentlyFits);

                    case FunctionReturn(\_, \_) | TypeCast(\_, \_): currentlyFits = currentlyFits && fit(cast given[i].getParameters(), cast pattern[i].getParameters(), currentlyFits);

                    case Write(assignees, value): {

                        var patternAssignees:Array<InterpTokens> = pattern[i].parameter(0);

                        if (patternAssignees != null) currentlyFits = currentlyFits && fit(assignees, patternAssignees, currentlyFits);

                        if (pattern[i].parameter(1) != null) currentlyFits = currentlyFits && fit(cast value.getParameters(), cast pattern[i].parameter(1).getParameters(), currentlyFits);

                    }

                    case Expression(parts, type) | Block(parts, type): {

                        var patternParts:Array<InterpTokens> = pattern[i].parameter(0).copy();

                        if (patternParts != null) currentlyFits = currentlyFits && fit(parts, patternParts, currentlyFits);

                        if (pattern[i].parameter(1) != null) currentlyFits = currentlyFits && fit(cast type.getParameters(), cast pattern[i].parameter(1).getParameters(), currentlyFits);

                    }

                    case PartArray(parts): {

                        var patternParts:Array<InterpTokens> = pattern[i].parameter(0);

                        if (patternParts != null) currentlyFits = currentlyFits && fit(parts, patternParts, currentlyFits);

                    }

                    case PropertyAccess(name, property): currentlyFits = currentlyFits && fit(cast given[i].getParameters(), cast pattern[i].getParameters(), currentlyFits);

                    case Object(props, typeName): return false; // Cant be matched with, only valid in the context of object instantiation. Represented by FunctionCall in most cases.

                    case \_: continue;

                }

                if (!currentlyFits) return false;

            }

            return currentlyFits;

        }

        var patternString = PrettyPrinter.stringifyInterpreter(pattern);

        // We might want to attach stuff to the body, so we need to make it so it doesn't create a new scope & strip type info from it

        var bodyString = PrettyPrinter.stringifyInterpreter(body);

        for (\_pattern => caller in patterns) {

            if (\_pattern == null || fit(givenPattern, \_pattern)) { // As per the docs, a null pattern means any pattern

                var conditionRunner = (caller.parameter(0) : Array<InterpTokens>);

                var params = [

                    Write([VariableDeclaration(Identifier(Little.keywords.CONDITION\_PATTERN\_PARAMETER\_NAME), Identifier(Little.keywords.TYPE\_STRING), null)], Characters(patternString)),

                    Write([VariableDeclaration(Identifier(Little.keywords.CONDITION\_BODY\_PARAMETER\_NAME), Identifier(Little.keywords.TYPE\_STRING), null)], Characters(bodyString)),

                ];

                for (listener in Little.runtime.onConditionCalled)

                    listener(name.asJoinedStringPath(), givenPattern, body);

                return run(params.concat(conditionRunner), true);

            }

        }

        return error('Pattern $patternString is not supported in condition ${name.asStringPath()} (patterns (`\*` means any value): \n\t(${[for (pattern in patterns.keys()) pattern].map(x -> PrettyPrinter.stringifyInterpreter(x).replace("null", "\*")).join('),\n\t(')})\n)');

    }

    /\*\*

        Assign a value to multiple variables/functions/types.

        @param assignees The variables/functions/types to assign to. Should be a `InterpTokens.VariableDeclaration(name:InterpTokens, type:InterpTokens, doc:InterpTokens)` or `ParserTokens.FunctionDeclaration(name:InterpTokens, params:ParserTokens, type:InterpTokens, doc:InterpTokens)`

        @param value The value to assign. Can be any token which has a non-void value (not `InterpTokens.SplitLine`, `InterpTokens.SetLine(line:Int)`...)

        @return The value given, evaluated using `Interpreter.evaluate(value)`

    \*\*/

    public static function write(assignees:Array<InterpTokens>, value:InterpTokens):InterpTokens {

        var vars = [], funcs = [];

        var containsFunction = false;

        var containsVariable = false;

        for (assignee in assignees) {

            switch assignee {

                case VariableDeclaration(name, type, doc): declareVariable(name, type, doc); vars.push(name); containsVariable = true;

                case FunctionDeclaration(name, params, type, doc): declareFunction(name, params, doc); funcs.push(name); containsFunction = true; //TODO: find a way to store function type

                case \_: vars.push(assignee); containsVariable = true;

            }

        }

        if (containsFunction) {

            var paths = funcs.map(x -> x.asStringPath());

            for (path in paths) {

                var func = memory.read(...path).objectValue;

                memory.set(path, FunctionCode(func.parameter(0), value), Little.keywords.TYPE\_FUNCTION, "");

            }

        }

        if (containsVariable) {

            var paths = vars.map(x -> x.asStringPath());

            // filter for identifiers/property accesses, for which Memory.retrieve does the work.

            var evaluated = evaluate(value); // No need to calculate multiple times, so we just evaluate once

            for (path in paths) {

                memory.set(path, value.is(IDENTIFIER, PROPERTY\_ACCESS) ? value : evaluated, evaluated.type(), "");

            }

        }

        for (listener in Little.runtime.onWriteValue.copy()) {

            listener(vars.map(x -> x.extractIdentifier()).concat(funcs.map(x -> x.extractIdentifier())));

        }

        return value;

    }

    /\*\*

        Calls a function and returns the result using `params`.

        @param name The name of the function. Can be any token stringifiyable via `token.value()`.

        @param params The parameters of the function. Should be a `InterpTokens.PartArray(parts:Array<InterpTokens>)`

    \*\*/

    public static function call(name:InterpTokens, params:InterpTokens):InterpTokens {

        var functionCode = evaluate(name);

        var functionName = name.asJoinedStringPath();

        var processedParams = [];

        var current = [];

        for (p in (params.parameter(0) : Array<InterpTokens>)) {

            switch p {

                case SplitLine: {

                    processedParams.push(calculate(current));

                    current = [];

                }

                case SetLine(l): setLine(l);

                case \_: current.push(p);

            }

        }

        if (current.length > 0) processedParams.push(calculate(current));

        switch functionCode {

            case FunctionCode(requiredAndOptionalParams, body): {

                var given = processedParams.length;

                var resulting:Array<InterpTokens> = [];

                var required = 0;

                var unattained:Array<String> = [];

                var attachment:Array<InterpTokens> = [];

                for (key => typeCast in requiredAndOptionalParams.keyValueIterator()) {

                    var name = key, value = null, type = Identifier(Little.keywords.TYPE\_DYNAMIC);

                    switch typeCast {

                        case TypeCast(NullValue, t): type = t;

                        case TypeCast(v, \_.parameter(0) == Little.keywords.TYPE\_UNKNOWN => true): value = v;

                        case TypeCast(v, t): type = t; value = v;

                        case \_:

                    }

                    if (value == null) required++;

                    if (processedParams.length > 0) value = processedParams.shift(); // Todo, handle mid-function optional arguments.

                    else if (value == null && processedParams.length == 0) unattained.push(name);

                    resulting.push(value);

                    attachment.push(Write([VariableDeclaration(Identifier(name), type, null)], value));

                }

                if (given > requiredAndOptionalParams.length) required = requiredAndOptionalParams.length; // Better error sensibility.

                if (required > given || given > requiredAndOptionalParams.length) {

                    return error('Incorrect number of parameters: Function `$functionName` fully requires $required parameter${required == 1 ? "" : "s"}, but${given == 0 || given > requiredAndOptionalParams.length ? "" : " only"} ${given} ${processedParams.length == 1 ? "was" : "were"} given ${required > given ? '(parameter${unattained.length == 1 ? "" : "s"} `${unattained.join(", ").replaceLast(",", " &")}` got left out).' : ""}');

                }

                for (listener in Little.runtime.onFunctionCalled) {

                    listener(functionName, resulting);

                }

                Little.runtime.callStack.push({module: Little.runtime.module, line: Little.runtime.line, linePart: Little.runtime.linePart, token: FunctionCall(name, params)});

                var t = run(attachment.concat(body.parameter(0)));

                Little.runtime.callStack.pop();

                return t;

            }

            case \_: return null;

        }

    }

    /\*\*

        Reads the value of a variable/function, When reading a function, the body is returned as a `InterpTokens.FunctionCode(requiredParams:OrderedMap<String, InterpTokens.Identifier>, body:InterpTokens)`.

        @param name The name of the variable/function. Should be one of `InterpTokens.Identifier(name:String)` or `InterpTokens.PropertyAccess(name:InterpTokens, property:InterpTokens)`

        @return The value of the variable/function

    \*\*/

    public static function read(name:InterpTokens):InterpTokens {

        return memory.read(...name.asStringPath()).objectValue;

    }

    /\*\*

        Casts a value to a type

        @param value The value to cast. Can be any token which has a non-void value (not `InterpTokens.SplitLine`, `InterpTokens.SetLine(line:Int)`...)

        @param type The type to cast to. Can be any token which resolves to a correct string path.

        @return The value given, casted to the type.

    \*\*/

    public static function typeCast(value:InterpTokens, type:InterpTokens):InterpTokens {

        var preType = evaluate(value).type().asTokenPath().asStringPath();

        var postType = type.extractIdentifier().asTokenPath().asStringPath();

        if (preType.join("") == postType.join("") || postType.join(Little.keywords.PROPERTY\_ACCESS\_SIGN) == Little.keywords.TYPE\_UNKNOWN) return value;

        preType.push(Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX + postType.join("\_"));

        value = call(preType.join(Little.keywords.PROPERTY\_ACCESS\_SIGN).asTokenPath(), PartArray([value]));

        for (listener in Little.runtime.onTypeCast) {

            listener(value, type.asJoinedStringPath());

        }

        return value;

    }

    /\*\*

        Runs the tokens and returns the result.

        Adds a new scope.

        @param body The tokens to run

        @return The result of the tokens

    \*\*/

    public static function run(body:Array<InterpTokens>, propagateReturns = false):InterpTokens {

        var returnVal:InterpTokens = null;

        memory.referrer.pushScope();

        var i = 0;

        while (i < body.length) {

            var token = body[i];

            //trace('Running: $token. $i');

            if (token == null) {i++; continue;}

            Little.runtime.currentToken = token;

            switch token {

                case SetLine(line): {

                    setLine(line);

                }

                case SetModule(module): setModule(module);

                case SplitLine: splitLine();

                case VariableDeclaration(name, type, doc): {

                    declareVariable(name.is(BLOCK) ? evaluate(name) : name, type.is(BLOCK) ? evaluate(type) : type, doc != null ? evaluate(doc) : Characters(""));

                    returnVal = NullValue;

                }

                case FunctionDeclaration(name, params, type, doc): {

                    declareFunction(name.is(BLOCK) ? evaluate(name) : name, params, doc != null ? evaluate(doc) : Characters("")); // TODO: type is not used

                    returnVal = NullValue;

                }

                case ConditionCall(name, exp, body): {

                    returnVal = condition(name, exp, body);

                    if (returnVal != null && returnVal.is(FUNCTION\_RETURN)) return evaluate(returnVal);

                }

                case Write(assignees, value): {

                    returnVal = write(assignees, value);

                }

                case FunctionCall(name, params): {

                    returnVal = call(name, params);

                }

                case FunctionReturn(value, type): {

                    if (value.is(HAXE\_EXTERN)) {

                        return value.parameter(0)();

                    }

                    // If we don't check for haxe externs, they may return a rogue value

                    // and .type() will fail.

                    var v = evaluate(value);

                    var t = v.type().asTokenPath();

                    return propagateReturns ? FunctionReturn(v, t) : v;

                }

                case Block(body, type): {

                    returnVal = run(body);

                }

                case PropertyAccess(name, property): {

                    returnVal = evaluate(token);

                }

                case Identifier(name): {

                    returnVal =  read(token);

                }

                case HaxeExtern(func): {

                    returnVal = func();

                }

                case \_: returnVal = evaluate(token);

            }

            for (listener in Little.runtime.onTokenInterpreted)

                listener(token);

            Little.runtime.previousToken = token;

            i++;

        }

        memory.referrer.popScope();

        return returnVal;

    }

    /\*\*

        A combination of `Interpreter.run` and `Interpreter.calculate`, which operates on a single `InterpTokens` token.

        @param exp the token to evaluate

        @param dontThrow if `true`, `Little.runtime.throwError` is not called when an error occurs

        @return the result of the evaluation

    \*\*/

    public static function evaluate(exp:InterpTokens, ?dontThrow:Bool = false):InterpTokens {

        switch exp {

            case Number(\_) | Decimal(\_) | Characters(\_) | TrueValue | FalseValue | NullValue | Sign(\_) | FunctionCode(\_, \_) | Object(\_, \_) | ClassPointer(\_): return exp;

            case ConditionCode(callers): return Characters("<condition>");

            case ErrorMessage(msg): {

                if (!dontThrow) Little.runtime.throwError(exp, INTERPRETER\_VALUE\_EVALUATOR);

                return exp;

            }

            case SetLine(line): {

                setLine(line);

                return NullValue;

            }

            case SetModule(module): {

                setModule(module);

                return NullValue;

            }

            case SplitLine: {

                splitLine();

                return NullValue;

            }

            case Expression(parts, t): {

                if (t.asJoinedStringPath() == Little.keywords.TYPE\_UNKNOWN) return calculate(parts);

                return typeCast(calculate(parts), t);

            }

            case Block(body, t): {

                var currentLine = Little.runtime.line;

                var returnVal = run(body);

                setLine(currentLine);

                if (t.asJoinedStringPath() == Little.keywords.TYPE\_UNKNOWN) return evaluate(returnVal, dontThrow);

                return evaluate(typeCast(returnVal, t), dontThrow);

            }

            case FunctionCall(name, params): {

                var currentLine = Little.runtime.line;

                return call(name, params);

                setLine(currentLine);

            }

            case PartArray(parts): {

                return PartArray([for (p in parts) evaluate(p, dontThrow)]);

            }

            case Identifier(word): {

                return read(exp);

            }

            case TypeCast(value, t): return typeCast(value, t);

            case Write(assignees, value): return write(assignees, value);

            case ConditionCall(name, exp, body): return condition(name, exp, body);

            case VariableDeclaration(name, type, doc): {

                declareVariable(name.is(BLOCK) ? evaluate(name) : name, type.is(BLOCK) ? evaluate(type) : type, evaluate(doc));

                return NullValue;

            }

            case FunctionDeclaration(name, params, type, doc): {

                declareFunction(name.is(BLOCK) ? evaluate(name) : name, params, evaluate(doc)); // TODO: type is not stored.

                return NullValue;

            }

            case PropertyAccess(name, property): {

                var path = exp.toIdentifierPath();

                // Two cases:

                //  - regular property access

                //  - access on inline value

                if (path.filter(p -> !p.is(IDENTIFIER)).length == 0) {

                    return read(exp);

                } else if (!path[0].is(IDENTIFIER) && path.slice(1).filter(p -> !p.is(IDENTIFIER)).length == 0) {

                    var value = evaluate(path[0]);

                    return memory.readFrom({

                        objectValue: value,

                        objectAddress: memory.store(value)

                    }, ...path.slice(1).map(ident -> ident.extractIdentifier())).objectValue; // Evaluation is never needed here, since only evaluated values can be stored.

                } else {

                    return error('Cannot access ${path.join(Little.keywords.PROPERTY\_ACCESS\_SIGN)}, path cannot contain a raw value in the middle (for property: ${PrettyPrinter.stringifyInterpreter(path.slice(1).filter(p -> !p.is(IDENTIFIER))[0])}');

                }

            }

            case FunctionReturn(value, t): return evaluate(typeCast(value, t));

            case HaxeExtern(func): return func();

            case \_: return evaluate(ErrorMessage('Unable to evaluate token `$exp`'), dontThrow);

        }

        return NullValue;

    }

    /\*\*

        Calculates the result of a given expression. An "alternative" to `Interpreter.run()`,

        but instead of having code running and memory writing capabilities, it's capable of

        calculating complex equations of different types. example:

        |          Function         |       Input       |   Result  |                           Process                              |

        |           :---:           |       :---        |     ---   |                             ---                                |

        |                           | `1 + 1`           | `1`       | picks up the last token.                                       |

        | `Interpreter.run()`       | `(2 + 2)`         | `4`       | picks up the last token. its an expression, so it's evaluated  |

        |                           | `3 + (5 \* 2)!`    | `!`       | picks up the last token.                                       |

        | ------------------------- | ----------------- | --------- | -------------------------------------------------------------- |

        |                           | `1 + 1`           | `2`       | evaluates all tokens and calculates the relations between them |

        | `Interpreter.calculate()` | `(2 + 2)`         | `4`       | evaluates all tokens and calculates the relations between them |

        |                           | `3 + (5 \* 2)!`    | `3628803` | evaluates all tokens and calculates the relations between them |

        @param parts The parts of the expression

        @return The result of the expression

    \*\*/

    public static function calculate(p:Array<InterpTokens>):InterpTokens {

        while (p.length == 1 && p[0].parameter(0) is Array && !p[0].is(BLOCK)) p = p[0].parameter(0);

        var tokens = group(p);

        var castType:InterpTokens = null;

        if (tokens.length == 1) {

            if (tokens[0].is(PART\_ARRAY)) tokens = tokens[0].parameter(0);

            else if (tokens[0].is(EXPRESSION)) {

                tokens = tokens[0].parameter(0);

                castType = tokens[0].parameter(1);

            } else if (tokens[0].is(BLOCK)) {

                tokens = [run(tokens[0].parameter(0))];

                castType = tokens[0].parameter(1);

            }

        }

        var calculated:InterpTokens = null;

        var sign:String = "";

        tokens = tokens.filter(x -> x != null); // Safety clause, for strange edge cases such as 2 + ---5.

        for (token in tokens) {

            switch token {

                case PartArray(parts): {

                    if (sign != "" && calculated == null) calculated = Little.memory.operators.call(sign, calculate(parts)); // RHS operator

                    else if (calculated == null) calculated = calculate(parts);

                    else if (sign == "") error('Two values cannot come one after the other ($calculated, $token). At least one of them should be an operator, or, put an operator in between.');

                    else {

                        calculated = Little.memory.operators.call(calculated, sign, calculate(parts)); // standard operator

                    }

                }

                case Sign(s): {

                    sign = s;

                    if (tokens.length == 1) return token;

                    if (tokens[tokens.length - 1].equals(token)) calculated = Little.memory.operators.call(calculated, sign); //LHS operator

                }

                case Expression(parts, t): {

                    var val = t != null ? typeCast(calculate(parts), t) : calculate(parts);

                    if (sign != "" && calculated == null) calculated = Little.memory.operators.call(sign, val); // RHS operator

                    else if (calculated == null) calculated = val;

                    else if (sign == "") error('Two values cannot come one after the other ($calculated, $token). At least one of them should be an operator, or, put an operator in between.');

                    else {

                        calculated = Little.memory.operators.call(calculated, sign, val); // standard operator

                    }

                }

                case SetModule(module): setModule(module);

                case \_: {

                    if (sign != "" && calculated == null) calculated = Little.memory.operators.call(sign, token);

                    else if (sign == "" && calculated != null) throw 'Unexpected token: $token After calculating $calculated';

                    else if (calculated == null) calculated = token;

                    else if (sign == "") error('Two values cannot come one after the other ($calculated, $token). At least one of them should be an operator, or, put an operator in between.');

                    else {

                        calculated = Little.memory.operators.call(calculated, sign, token);

                        trace(calculated, sign, token);

                    }

                }

            }

            trace(calculated, castType);

        }

        if (castType != null) {

            return typeCast(calculated, castType);

        }

        return calculated;

    }

    /\*\*

        Sorts out order of operations.

    \*\*/

    public static function group(tokens:Array<InterpTokens>):Array<InterpTokens> {

        var post = tokens;

        var pre = [];

        for (operatorGroup in Little.memory.operators.iterateByPriority()) {

            pre = post.copy();

            post = [];

            // We'll group everything by only recognizing specific signs each "stage" -

            // The signs recognized first will be of the highest priority.

            // One drawback of this system is that its a little messier to detect chaining (e.g. 5!!, √√√64)

            var i = 0;

            while (i < pre.length) {

                var token = pre[i].is(IDENTIFIER, BLOCK) ? evaluate(pre[i]) : pre[i];

                switch token {

                    case Sign(operatorGroup.filter(x -> x.sign == \_).length > 0 => true): {

                        // If theres an operator before this one, its RHS\_ONLY. If theres an operator after, its LHS\_ONLY

                        // If theres no operator, its LHS\_RHS

                        // First lets do a simple edge case - i = pre.length - 1 => LHS\_ONLY operator.

                        if (i == pre.length - 1) {

                            post.push(PartArray([post.pop(), token]));

                            break;

                        }

                        var lookbehind = post.length > 0 ? post[post.length - 1] /\* Post has only evaluated tokens \*/ : Sign("\_"); // Just an arbitrary "sign" to not have null here

                        var lookahead = pre[i + 1].is(IDENTIFIER, BLOCK) ? evaluate(pre[i + 1]) : pre[i + 1];

                        if (lookahead.is(SIGN) && operatorGroup.filter(x -> x.sign == lookahead.parameter(0)).length > 0) {

                            /\* This can be one of two cases:

                            - were working on a binary operator before a unary operator (1 or more)

                            - were working on a unary operator (1 or more) before a binary operator

                            We should naturally prioritize unary operators since they, resulting from their definition, always come first.

                            This makes the parsing very easy, since the first possible binary operator must be the binary one: (pretend +, - and ! are of the same priority)

                            5!! + ---5

                            both + and - cant be LHS\_ONLY, so grouping is:

                            (((5!)!) + (-(-(-5)))) \*/

                            if (operatorGroup.filter(x -> x.sign == token.parameter(0) && x.side == LHS\_ONLY).length > 0) {

                                post.push(PartArray([post.pop(), token]));

                            } else if (operatorGroup.filter(x -> x.sign == token.parameter(0) && x.side == LHS\_RHS).length > 0) {

                                var operand1 = post.pop();

                                var op = lookahead;

                                // We have to repeat the check in RHS\_ONLY, since RHS can also start with a sign

                                if (i + 2 >= pre.length) error("Expression ended with an operator, when an operand was expected.");

                                var lookahead2 = pre[i + 2].is(IDENTIFIER, BLOCK) ? evaluate(pre[i + 2]) : pre[i + 2];

                                if (!lookahead2.is(SIGN)) {

                                    post.push(PartArray([operand1, token, PartArray([lookahead, lookahead2])]));

                                    i += 2; // +2 because we consumed both lookahead and lookahead2 for the PartArray arg

                                } else {

                                    var g = [];

                                    while (lookahead2.is(SIGN) && operatorGroup.filter(x -> x.sign == lookahead2.parameter(0) && x.side == RHS\_ONLY).length > 0) {

                                        g.push(lookahead2);

                                        i++;

                                        if (i + 2 >= pre.length) error("Expression ended with an operator, when an operand was expected.");

                                        lookahead2 = pre[i + 2].is(IDENTIFIER, BLOCK) ? evaluate(pre[i + 2]) : pre[i + 2];

                                    }

                                    // Last token is an operand

                                    g.push(lookahead2);

                                    // And increment i since lookahead2 uses i + 1

                                    i++;

                                    var operand2 = g.length == 1 ? g[0] : PartArray(group(g));

                                    post.push(PartArray([operand1, op, operand2]));

                                }

                            } else if (operatorGroup.filter(x -> x.sign == token.parameter(0) && x.side == RHS\_ONLY).length > 0) {

                                error("An operator that expects a right side can't be preceded by an operator that expects a left side.");

                            }

                        } else {

                            // Both sides are regular operands, so we just pop from `post` and take the lookahead

                            // And no, we should'nt worry about order of operations here. because of this "algorithm"'s format, all

                            // operators are of the same priority, and its the user's responsibility to use parentheses when needed.

                            if (lookahead.is(SIGN)) {

                                post.push(PartArray([post.pop(), token]));

                            } else {

                                post.push(PartArray([post.pop(), token, lookahead]));

                                i++;

                            }

                        }

                    }

                    case Expression(parts, type): post.push(Expression(group(parts), type));

                    case \_: post.push(token);

                }

                i++;

            }

        }

        return post;

    }

}

package little.interpreter;

import haxe.Unserializer;

import haxe.Serializer;

import little.interpreter.Tokens.InterpTokens;

/\*\*

    A class containing functions to compile and decompile Little code into ByteCode.

\*\*/

class ByteCode {

    /\*\*

        Compiles an array of `InterpTokens` into a compact, executable string, or, ByteCode.

        @param ...tokens a Rest-array of `InterpTokens`. Use ... to pass an array of `InterpTokens`

        @return The compiled string

    \*\*/

    public static function compile(...tokens:InterpTokens):String {

        return Serializer.run(tokens); // Simple, for now.

    }

    /\*\*

        Decompiles a string representing executable bytecode into an array of `InterpTokens`.

        Warning: this will only work on strings that are generated by the `compile` function. Any other string will result in unexpected behavior.

        @param bytecode a string representing executable bytecode

        @return an array of `InterpTokens`

    \*\*/

    public static function decompile(bytecode:String):Array<InterpTokens> {

        return Unserializer.run(bytecode);

    }

}

package little.interpreter;

import little.tools.PrettyPrinter;

import little.interpreter.Tokens.InterpTokens;

import little.tools.Layer;

using StringTools;

using little.tools.TextTools;

using little.tools.Extensions;

/\*\*

    A class containing values and callbacks related to Little's runtime.

\*\*/

@:access(little.interpreter.Interpreter)

class Runtime {

    public function new() {}

    /\*\*

        The line currently interpreted.

    \*\*/

    public var line(default, null):Int = 0;

    /\*\*

        The currently interpreted part of a line, split by `,` or `;`

    \*\*/

    public var linePart(default, null):Int = 0;

    /\*\*

        The next token to be interpreted

    \*\*/

    public var currentToken(default, null):InterpTokens = null;

    /\*\*

        The module in which tokens are currently interpreted.

    \*\*/

    public var module(default, null):String;

    /\*\*

        The token that has just been interpreted

    \*\*/

    public var previousToken(default, null):InterpTokens;

    /\*\*

        | Code | Description |

        | :---:| ---         |

        | **\*\*0\*\*** | Everything went fine, aside from potential warnings. |

        | **\*\*1\*\*** | An error was thrown, and terminated the program. The error is printed to stdout, and its token is kept after the fact in `Little.runtime.errorToken`. |

    \*\*/

    public var exitCode(default, null):Int = 0;

    /\*\*

        This is set to `true` if an error was thrown. Execution should stop.

    \*\*/

    public var errorThrown(default, null):Bool = false;

    /\*\*

        The last error that was thrown. On normal settings, gets set at the same time the program terminates.

    \*\*/

    public var errorToken(default, null):InterpTokens;

    /\*\*

        Dispatches right before the interpreter starts running a line of code.

        @param line The line the interpreter just finished running.

    \*\*/

    public var onLineChanged:Array<Int -> Void> = [];

    /\*\*

        Dispatches right after the interpreter switches between running module.

        This can happen when code from another file is suddenly ran, for example,

        from a function call.

        @param previous The module the interpreter just switched from.

    \*\*/

    public var onModuleChanged:Array<String -> Void> = [];

    /\*\*

        Dispatches every time the interpreter finds a line splitter (`,` or `;`), or, right after

        a line-change event.

        @param line The line the interpreter just finished running.

    \*\*/

    public var onLineSplit:Array<Void -> Void> = [];

    /\*\*

        Dispatches after finishing interpreting a token.

        #### - What is a token?

        In order for Little to run your code from a string, it has to

         - first, extract the useful content ot of the string

         - then, extract more information from the useful content we've just extracted

         - and only then, iterate over the tokens in order to run the code

        After each iteration, this method gets called, passing the token we've just parsed as an argument.

        @param token The token that was just interpreted. Note - expressions (`()`) are passed as is, and may contain multiple tokens.

    \*\*/

    public var onTokenInterpreted:Array<InterpTokens -> Void> = [];

    /\*\*

        Dispatches right after an error is thrown, and printed to the standard output.

        @param module the module from which the error was thrown.

        @param line the line from which the error was thrown.

        @param reason The contents of the error.

    \*\*/

    public var onErrorThrown:Array<(String, Int, String) -> Void> = [];

    /\*\*

        Dispatches right after a warning is printed to the standard output

        @param module the module from which the warning was printed.

        @param line the line form which the warning was printed

        @param reason the contents of the warning

    \*\*/

    public var onWarningPrinted:Array<(String, Int, String) -> Void> = [];

    /\*\*

        Dispatches right after the program has written something to a variable/multiple variables.

        @param variables The variables that were written to. Value can be retrieved using `memory.read()`.

    \*\*/

    public var onWriteValue:Array<Array<String> -> Void> = [];

    /\*\*

        Dispatches right before a function is called, and before it is added to the callstack.

        Useful when used with the `line`, `linePart` and `module` properties.

        @param name The name of the function, may include module/object name.

        @param parameters The parameters the function was called with.

    \*\*/

    public var onFunctionCalled:Array<(String, Array<InterpTokens>) -> Void> = [];

    /\*\*

        Dispatches right before a condition is called.

        Useful when used with the `line`, `linePart` and `module` properties.

        @param name The name of the condition, may include module/object name.

        @param parameters The parameters the condition was called with.

        @param body The body of the condition. Is either a `Block` containing code, or another single token.

    \*\*/

    public var onConditionCalled:Array<(String, Array<InterpTokens>, InterpTokens) -> Void> = [];

    /\*\*

        Dispatches right after a field is declared & written to memory.

        Listeners are not triggered on external variable/function declarations.

        @param name The name of the field. Includes full path (`object.a.c`, `b`)

        @param fieldType The type of the field (variable, function, etc.)

    \*\*/

    public var onFieldDeclared:Array<(String, FieldDeclarationType) -> Void> = [];

    /\*\*

        Dispatches right after a value has successfully casted to a type.

        @param value the pre-casted value, which is castable to the type

        @param type the type the value was casted to. Given as a string, containing the path to the type.

    \*\*/

    public var onTypeCast:Array<(InterpTokens, String) -> Void> = [];

    /\*\*

        The program's standard output.

    \*\*/

    public var stdout:StdOut = new StdOut();

    /\*\*

        Contains every function call interpreted during the program's runtime.

    \*\*/

    public var callStack:Array<{module:String, line:Int, linePart:Int, token:InterpTokens}> = [];

    /\*\*

        Stops the execution of the program, and prints an error message to the console. Dispatches `onErrorThrown`.

        @param token some token which is the error, usually `ErrorMessage`

        @param layer the "stage" from which the error was called

        @return the token that caused the error (the first parameter of this function)

        \*\*/

    public function throwError(token:InterpTokens, ?layer:Layer = INTERPRETER):InterpTokens {

        var mod:String = module, reason:String;

        var content = switch token {

            case \_: {

                reason = Std.string(token).remove(token.getName()).substring(1).replaceLast(")", "");

                '${if (Little.debug) (layer : String).toUpperCase() + ": " else ""}ERROR: Module ${module}, Line $line:  ${reason}';

            }

        }

        stdout.output += '\n$content\n';

        stdout.output += callStack.map(obj -> '\tCalled from Module ${obj.module}, Line ${obj.line} (part ${obj.linePart}): ${PrettyPrinter.stringifyInterpreter(obj.token)}').join('\n');

        stdout.stdoutTokens.push(token);

        callStack.push({module: module, line: line, linePart: linePart, token: token});

        exitCode = Layer.getIndexOf(layer);

        errorToken = token;

        errorThrown = true;

        for (func in onErrorThrown) func(mod, line, reason);

        throw token; // Currently, no flag exists that disables immediate quitting, so this is fine.

        return token;

    }

    /\*\*

        Same as `throwError`, but doesn't stop execution, and has the "WARNING" prefix.

        @param token some token which is the error, usually `ErrorMessage`

        @param layer the "stage" from which the error was called

    \*\*/

    public function warn(token:InterpTokens, ?layer:Layer = INTERPRETER) {

        callStack.push({module: module, line: line, linePart: linePart, token: token});

        var reason:String;

        var content = switch token {

            case \_: {

                reason = Std.string(token).remove(token.getName()).substring(1).replaceLast(")", "");

                '${if (Little.debug) (layer : String).toUpperCase() + ": " else ""}WARNING: Module ${module}, Line $line:  ${reason}';

            }

        }

        stdout.output += '\n$content';

        stdout.stdoutTokens.push(token);

        for (func in onWarningPrinted) func(module, line, reason);

    }

    /\*\*

        Prints a Haxe string to `Little`'s standard output.

        @param item

    \*\*/

    public function print(item:String) {

        stdout.output += '\n${if (Little.debug) (INTERPRETER : String).toUpperCase() + ": " else ""}Module $module, Line $line:  $item';

        stdout.stdoutTokens.push(Characters(item));

    }

    /\*\*

        Prints a Haxe string to `Little`'s standard output, without any positional information.

        On `Little.debug` mode, the string will be prefixed with `BROADCAST: `.

    \*\*/

    public dynamic function broadcast(item:String) {

        stdout.output += '\n${if (Little.debug) "BROADCAST: " else ""}${item}';

        stdout.stdoutTokens.push(Characters(item));

    }

    /\*\*

        Quiet broadcast, without addition to stdoutTokens

    \*\*/

    dynamic function \_\_broadcast(item:String) {

        stdout.output += '\n${if (Little.debug) "BROADCAST: " else ""}${item}';

    }

    /\*\*

        A special type of `print`, which allows addition of custom tokens to stdoutTokens.

    \*\*/

    function \_\_print(item:String, representativeToken:InterpTokens) {

        stdout.output += '\n${if (Little.debug) (INTERPRETER : String).toUpperCase() + ": " else ""}Module $module, Line $line:  $item';

        stdout.stdoutTokens.push(representativeToken);

    }

}

/\*\*

    Types of field declarations.

\*\*/

enum FieldDeclarationType {

    VARIABLE;

    FUNCTION;

    CONDITION;

    CLASS;

    OPERATOR;

}

package little.interpreter;

import little.interpreter.Tokens.InterpTokens;

class StdOut {

    /\*\*

        A string, containing everything that was printed to the console during the program's runtime.

    \*\*/

    public var output:String = "";

    /\*\*

        An array of tokens consisting of all tokens that were printed-out.

    \*\*/

    public var stdoutTokens:Array<InterpTokens> = new Array<InterpTokens>();

    /\*\*

        Resets the `output` and `stdoutTokens` variables, thus resetting the console.

    \*\*/

    public function reset() {

        output = "";

        stdoutTokens = new Array<InterpTokens>();

    }

    /\*\*

        Instantiates a new `StdOut`.

    \*\*/

    public function new() {}

}

package little.interpreter.memory;

import little.tools.OrderedMap;

import little.tools.PrettyPrinter;

import little.interpreter.memory.MemoryPointer.POINTER\_SIZE;

import little.tools.TextTools;

import little.interpreter.Tokens.InterpTokens;

import vision.ds.ByteArray;

using little.tools.Extensions;

using vision.tools.MathTools;

class Memory {

    public var storage:Storage;

    public var referrer:Referrer;

    public var externs:ExternalInterfacing;

    public var constants:ConstantPool;

    public var operators:Operators;

    @:noCompletion public var memoryChunkSize:Int = 512; // 512 bytes

    /\*\*

        The maximum amount of memory that can be allocated, by bytes.

    \*\*/

    public var maxMemorySize:Int = 1024 \* 1024 \* 1024 \* 2;

    /\*\*

        The current amount of memory allocated, in bytes.

    \*\*/

    public var currentMemorySize(get, never):Int;

    /\*\* Memory.currentMemorySize getter \*\*/ @:noCompletion function get\_currentMemorySize() {

        return storage.reserved.length + referrer.bytes.length;

    }

    /\*\*

        Instantiates a new `Memory`.

    \*\*/

    public function new() {

        storage = new Storage(this);

        referrer = new Referrer(this);

        constants = new ConstantPool(this);

        externs = new ExternalInterfacing(this);

        operators = new Operators();

    }

    /\*\*

        Resets all stored values, and releases all extra allocated memory.

    \*\*/

    public function reset() {

        storage = new Storage(this);

        referrer = new Referrer(this);

        externs = new ExternalInterfacing(this);

        // Constants don't need to be reset

        operators.lhsOnly.clear();

        operators.rhsOnly.clear();

        operators.standard.clear();

        operators.priority.clear();

    }

    /\*\*

        General-purpose memory allocation for objects:

        - if `token` is `true`, `false`, `0`, or `null`, it pulls a pointer from the constant pool

        - if `token` is a string, a number, a sign or a decimal, it stores & pulls a pointer from the storage.

        - if `token` is a structure, it stores it on the storage, and returns a pointer to it.

        - if `token` is a `ClassPointer`, it returns it's address.

    \*\*/

    public function store(token:InterpTokens):MemoryPointer {

        if (token.is(TRUE\_VALUE, FALSE\_VALUE, NULL\_VALUE)) {

            return constants.get(token);

        } else if (token.staticallyStorable()) {

            return storage.storeStatic(token);

        } else if (token.is(OBJECT)) {

            return storage.storeObject(token);

        } else if (token.is(FUNCTION\_CODE, BLOCK)) {

            return storage.storeCodeBlock(token);

        } else if (token.is(CONDITION\_CODE)) {

            return storage.storeCondition(token);

        } else if (token.is(CLASS\_POINTER)) {

            return token.parameter(0);

        }

        Little.runtime.throwError(ErrorMessage('Unable to allocate memory for token `$token`.'), MEMORY\_STORAGE);

        return constants.NULL;

    }

    /\*\*

        Similar to the `store` function, but respects pass by value/reference rules.

        Whether an object of type `T` is passed by reference or not is dictated by the value of a field

        named `passedByReference`. This field cannot be changed at runtime for existing types.

        @param token A simple token (for which the returned value will be the same as a `store` call), or an identifiable

        token (evaluable by `Extensions.asJoinedStringPath`), specifically ones that return some sort of a "path" to that value.

        @return A pointer to that location/value.

    \*\*/

    public function retrieve(token:InterpTokens):MemoryPointer {

        switch token {

            case \_ if (token.is(TRUE\_VALUE, FALSE\_VALUE, NULL\_VALUE, OBJECT, FUNCTION\_CODE, BLOCK, CONDITION\_CODE, CLASS\_POINTER)

                || token.passedByValue()):

                {

                    return store(token);

                }

            case Identifier(\_) | PropertyAccess(\_, \_):

                {

                    if (token.is(PROPERTY\_ACCESS)) {

                        // Check if it doesnt start in a value

                        var temp = token;

                        while (temp.is(PROPERTY\_ACCESS)) {

                            temp = temp.parameter(0);

                        }

                        if (temp.is(BLOCK))

                            temp = Interpreter.run(temp.parameter(0));

                        if (temp.is(EXPRESSION))

                            temp = Interpreter.calculate(temp.parameter(0));

                        if (!temp.is(IDENTIFIER)) {

                            var p = store(temp);

                            // This starts with a value, jump to readFrom

                            var path = token.asStringPath();

                            path.shift();

                            var cell = readFrom({objectValue: temp, objectAddress: p}, ...path);

                            if (cell.objectAddress == constants.EXTERN) {

                                var params:OrderedMap<String, InterpTokens> = cell.objectValue.parameter(0);

                                var forwardedParams = [];

                                for (key in params.keys()) {

                                    forwardedParams.push(Identifier(key));

                                    forwardedParams.push(SplitLine);

                                }

                                forwardedParams.pop();

                                var fin = FunctionCode(params, Block([

                                    FunctionReturn(FunctionCall(token, PartArray(forwardedParams)), cell.objectTypeName.asTokenPath())

                                ], cell.objectTypeName.asTokenPath()));

                                return store(fin);

                            } else {

                                return readFrom({objectValue: temp, objectAddress: p}, ...path).objectAddress;

                            }

                        }

                    }

                    var path = token.asStringPath();

                    var cell = read(...path);

                    if (cell.objectValue.passedByValue())

                        return store(cell.objectValue);

                    if (externs.hasGlobal(...path) && cell.objectValue.is(FUNCTION\_CODE)) {

                        // Extern function "call" themselves the EXTERN pointer, but here this is not

                        // valid. A nice, nut still a little hacky solution is to create a function

                        // that references the external function, and store it instead.

                        var params:OrderedMap<String, InterpTokens> = cell.objectValue.parameter(0);

                        var forwardedParams = [];

                        for (key in params.keys()) {

                            forwardedParams.push(Identifier(key));

                            forwardedParams.push(SplitLine);

                        }

                        forwardedParams.pop();

                        var fin = FunctionCode(params, Block([

                            FunctionReturn(FunctionCall(token, PartArray(forwardedParams)), cell.objectTypeName.asTokenPath())

                        ], cell.objectTypeName.asTokenPath()));

                        return store(fin);

                    }

                    return cell.objectAddress;

                }

            case Block(\_, \_) | Expression(\_, \_):

                {

                    var result = Interpreter.evaluate(token);

                    switch result {

                        case Identifier(\_) | PropertyAccess(\_, \_): retrieve(result);

                        case \_: {

                                Little.runtime.throwError(ErrorMessage('Code block returned a value that cannot be read from (for value: ${PrettyPrinter.stringifyInterpreter(result)})'));

                                throw 'Unable to retrieve a pointer to token $result';

                            }

                    }

                }

            case \_:

        }

        Little.runtime.throwError(ErrorMessage('Unable to retrieve a pointer to token $token'));

        return constants.NULL;

    }

    function valueFromType(address:MemoryPointer, type:String, fullPath:Array<String>, ...currentPath:String) {

        return switch type {

            case(\_ == Little.keywords.TYPE\_STRING => true): Characters(storage.readString(address));

            case(\_ == Little.keywords.TYPE\_INT => true): Number(storage.readInt32(address));

            case(\_ == Little.keywords.TYPE\_FLOAT => true): Decimal(storage.readDouble(address));

            case(\_ == Little.keywords.TYPE\_BOOLEAN => true): constants.getFromPointer(address);

            case(\_ == Little.keywords.TYPE\_FUNCTION => true): storage.readCodeBlock(address);

            case(\_ == Little.keywords.TYPE\_CONDITION => true): storage.readCondition(address);

            case(\_ == Little.keywords.TYPE\_MODULE => true): ClassPointer(address);

            // Because of the way we store lone nulls (as type dynamic),

            // they might get confused with objects of type dynamic, so we need to do this:

            case((\_ == Little.keywords.TYPE\_DYNAMIC || \_ == Little.keywords.TYPE\_UNKNOWN)

                && constants.hasPointer(address)

                && constants.getFromPointer(address).equals(NullValue) => true): NullValue;

            case(\_ == Little.keywords.TYPE\_SIGN => true): storage.readSign(address);

            case(\_ == Little.keywords.TYPE\_UNKNOWN => true):

                Little.runtime.throwError(ErrorMessage('Could not get the value at ${fullPath.join(Little.keywords.PROPERTY\_ACCESS\_SIGN)} - field ${currentPath.toArray().join(Little.keywords.PROPERTY\_ACCESS\_SIGN)} was declared, but has no value/type.'),

                    MEMORY\_STORAGE);

            // Not sure how someone can even get to the error above, but it's better to be safe than sorry - maybe a developer generates an extern field of type Unknown or something...

            case \_: storage.readObject(address);

        }

    }

    /\*\*

        Reads the value at the end of the path.

        @param path The path to read. Provided as individual parameters. To use an array, use `...pathArray`.

        @return an anonymous structure: `{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}`.

    \*\*/

    public function read(...path:String):{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer} {

        // If the path is empty, we just return null

        if (path.length == 0) {

            return {

                objectValue: null,

                objectTypeName: null,

                objectAddress: -1,

            }

        }

        var current:InterpTokens = null;

        var currentAddress:MemoryPointer = -1;

        var currentType:String = null;

        var processed = path.toArray();

        var wentThroughPath = [];

        // Before anything, global external values are prioritized

        if (externs.hasGlobal(processed[0])) {

            if (externs.hasGlobal(...path)) {

                var object = externs.getGlobal(...path);

                var typeName = getTypeName(externs.createPathFor(externs.globalProperties, ...path).type);

                return {

                    objectValue: object.objectValue,

                    objectTypeName: typeName,

                    objectAddress: object.objectAddress,

                }

            } else {

                var external = [processed.shift()];

                wentThroughPath.push(external[0]);

                while (externs.hasGlobal(...external.concat([processed[0]]))) {

                    external.push(processed.shift());

                    wentThroughPath.push(external[external.length - 1]);

                }

                var object = externs.getGlobal(...external);

                current = object.objectValue;

                currentAddress = object.objectAddress;

                currentType = getTypeName(externs.createPathFor(externs.globalProperties, ...external).type);

            }

        } else {

            // If we didn't find anything on the externs, we look in the current scope.

            if (!referrer.exists(path[0])) {

                Little.runtime.throwError(ErrorMessage('Variable `${path[0]}` does not exist'), MEMORY\_REFERRER);

            }

            var data = referrer.get(path[0]);

            current = valueFromType(data.address, data.type, path, path[0]);

            currentAddress = data.address;

            currentType = data.type;

            wentThroughPath.push(processed.shift()); // We just went through with the first element.

        }

        while (processed.length > 0) {

            // Get the current field, and the type of that field as well

            var identifier = processed.shift();

            wentThroughPath.push(identifier);

            var typeName = current.type();

            // By design, the only other way properties are accessible on non-object

            // values is through externs. So, after the object checks, we only need to look there.

            // We should notice that, like before, externs are prioritized, so externs are evaluated first.

            // Property check:

            if (externs.hasInstance(...typeName.split(Little.keywords.PROPERTY\_ACCESS\_SIGN))) {

                var classProperties = externs.instanceProperties.properties.get(typeName);

                if (classProperties.properties.exists(identifier)) {

                    var newCurrent = classProperties.properties.get(identifier).getter(current, currentAddress);

                    current = newCurrent.objectValue;

                    currentAddress = newCurrent.objectAddress;

                    continue;

                }

            }

            // It is possible that the current value is also just a plain type:

            if (current.is(CLASS\_POINTER)) {

                var name = getTypeName(current.parameter(0));

                if (externs.hasGlobal(...name.split(Little.keywords.PROPERTY\_ACCESS\_SIGN))) {

                    var classProperties = externs.createPathFor(externs.globalProperties, ...name.split(Little.keywords.PROPERTY\_ACCESS\_SIGN));

                    if (classProperties.properties.exists(identifier)) {

                        var newCurrent = classProperties.properties.get(identifier).getter(current, currentAddress);

                        current = newCurrent.objectValue;

                        currentAddress = newCurrent.objectAddress;

                        continue;

                    }

                }

            }

            // If it doesn't exist on that specific type, it may exist on TYPE\_DYNAMIC:

            if (externs.hasInstance(...Little.keywords.TYPE\_DYNAMIC.split(Little.keywords.PROPERTY\_ACCESS\_SIGN))) {

                var classProperties = externs.instanceProperties.properties.get(Little.keywords.TYPE\_DYNAMIC);

                if (classProperties.properties.exists(identifier)) {

                    var newCurrent = classProperties.properties.get(identifier).getter(current, currentAddress);

                    current = newCurrent.objectValue;

                    currentAddress = newCurrent.objectAddress;

                    continue;

                }

            }

            // Then, we check the object's hash table for that field

            if (current.is(OBJECT)) {

                var objectHashTableBytes = HashTables.getHashTableOf(currentAddress, storage);

                if (HashTables.hashTableHasKey(objectHashTableBytes, identifier, storage)) {

                    var keyData = HashTables.hashTableGetKey(objectHashTableBytes, identifier, storage);

                    current = valueFromType(keyData.value, getTypeName(keyData.type), path, ...wentThroughPath);

                    currentAddress = keyData.value;

                }

            }

            // If we still don't have a value, we throw an error, cause that means that field doesn't exist.

            else {

                wentThroughPath.pop();

                var p = wentThroughPath.join(Little.keywords.PROPERTY\_ACCESS\_SIGN);

                Little.runtime.throwError(ErrorMessage('Field `$identifier` does not exist on `$p` ${current.is(NULL\_VALUE) ? '(`$p` is `${Little.keywords.NULL\_VALUE}`)' : ''}'));

                return {

                    objectValue: NullValue,

                    objectAddress: constants.NULL,

                    objectTypeName: Little.keywords.TYPE\_DYNAMIC,

                }

            }

        }

        return {

            objectValue: current,

            objectAddress: currentAddress,

            objectTypeName: current.type()

        }

    }

    /\*\*

        Starts reading from a specified value instead of preforming a lookup, and returns the value at the end of the path, when

        originating from the given value.

        @param value a value-address pair

        @param path the path to the value. Provided as individual parameters. To use an array, use `...pathArray`

    \*\*/

    public function readFrom(value:{objectValue:InterpTokens, objectAddress:MemoryPointer}, ...path:String) {

        var current = value.objectValue;

        var currentAddress = value.objectAddress;

        var processed = path.toArray();

        var wentThroughPath = [];

        while (processed.length > 0) {

            // Get the current field, and the type of that field as well

            var identifier = processed.shift();

            wentThroughPath.push(identifier);

            var typeName = current.type();

            // By design, the only other way properties are accessible on non-object

            // values is through externs. So, after the object checks, we only need to look there.

            // We should notice that, like before, externs are prioritized, so externs are evaluated first.

            // Property check:

            if (externs.hasInstance(...typeName.split(Little.keywords.PROPERTY\_ACCESS\_SIGN))) {

                var classProperties = externs.instanceProperties.properties.get(typeName);

                if (classProperties.properties.exists(identifier)) {

                    var newCurrent = classProperties.properties.get(identifier).getter(current, currentAddress);

                    current = newCurrent.objectValue;

                    currentAddress = newCurrent.objectAddress;

                    continue;

                }

            }

            // It is possible that the current value is also just a plain type:

            if (current.is(CLASS\_POINTER)) {

                var name = getTypeName(current.parameter(0));

                if (externs.hasGlobal(...name.split(Little.keywords.PROPERTY\_ACCESS\_SIGN))) {

                    var classProperties = externs.createPathFor(externs.globalProperties, ...name.split(Little.keywords.PROPERTY\_ACCESS\_SIGN));

                    if (classProperties.properties.exists(identifier)) {

                        var newCurrent = classProperties.properties.get(identifier).getter(current, currentAddress);

                        current = newCurrent.objectValue;

                        currentAddress = newCurrent.objectAddress;

                        continue;

                    }

                }

            }

            // If it doesn't exist on that specific type, it may exist on TYPE\_DYNAMIC:

            if (externs.hasInstance(...Little.keywords.TYPE\_DYNAMIC.split(Little.keywords.PROPERTY\_ACCESS\_SIGN))) {

                var classProperties = externs.instanceProperties.properties.get(Little.keywords.TYPE\_DYNAMIC);

                if (classProperties.properties.exists(identifier)) {

                    var newCurrent = classProperties.properties.get(identifier).getter(current, currentAddress);

                    current = newCurrent.objectValue;

                    currentAddress = newCurrent.objectAddress;

                    continue;

                }

            }

            // Then, we check the object's hash table for that field

            if (current.is(OBJECT)) {

                var objectHashTableBytesLength = storage.readInt32(currentAddress);

                var objectHashTableBytes = storage.readBytes(currentAddress.rawLocation + 4, objectHashTableBytesLength);

                if (HashTables.hashTableHasKey(objectHashTableBytes, identifier, storage)) {

                    var keyData = HashTables.hashTableGetKey(objectHashTableBytes, identifier, storage);

                    current = valueFromType(keyData.value, getTypeName(keyData.type), [PrettyPrinter.stringifyInterpreter(value.objectValue)].concat(path),

                        ...wentThroughPath);

                    currentAddress = keyData.value;

                }

            }

            // If we still don't have a value, we throw an error, cause that means that field doesn't exist.

            else {

                wentThroughPath.pop();

                var p = wentThroughPath.join(Little.keywords.PROPERTY\_ACCESS\_SIGN);

                Little.runtime.throwError(ErrorMessage('Field `$identifier` does not exist on `$p` ${current.is(NULL\_VALUE) ? '(`$p` is `${Little.keywords.NULL\_VALUE}`)' : ''}'),

                    MEMORY);

                return {

                    objectValue: NullValue,

                    objectAddress: constants.NULL,

                    objectTypeName: Little.keywords.TYPE\_DYNAMIC,

                }

            }

        }

        return {

            objectValue: current,

            objectAddress: currentAddress,

            objectTypeName: current.type()

        }

    }

    /\*\*

        Writes a new value to a **\*\*new\*\*** path specified by `path`.

        If a part of the path doesn't exist which is not it's end, an error will be thrown in `Little`'s runtime.

        If the path fully exists, it's value will be overwritten

        If `value` is given as an identifier/property access/block, there will be an attempt

        to retrieve the original object's address there, if it's type has `passedByReference` set to `true`. See `Memory.retrieve`

        @param path An array of strings representing the path to the value

        @param value The value to write. If `null`, the value will be set to `NullValue`

        @param type The type of the value. If `null`, the type will be set to `TYPE\_DYNAMIC`

        @param doc The documentation of the value. If `null`, the documentation will be set to `""`

    \*\*/

    public function write(path:Array<String>, ?value:InterpTokens, ?type:String, ?doc:String) {

        // A couple notices:

        /\*

            - The first n-1 elements of the path must exist beforehand, and must be objects

            - The last element will be a field of the last object

         \*/

        if (path.length == 0) {

            Little.runtime.throwError(ErrorMessage('Cannot write to an empty path'));

            // Does not make sense to have a path of length 0, but still more useful than a quiet return/crash.

        }

        if (path.length == 1) {

            referrer.reference(path[0], retrieve(value), type);

        } else {

            var pathCopy = path.slice(0, path.length - 1);

            var wentThroughPath = [path[0]];

            var current = referrer.get(pathCopy.shift());

            while (pathCopy.length > 0) {

                if (!getTypeInformation(current.type).passedByReference) {

                    Little.runtime.throwError(ErrorMessage('Cannot write to a static type. Only objects can have dynamic properties (${wentThroughPath.join(Little.keywords.PROPERTY\_ACCESS\_SIGN)} is `${current.type}`)'));

                }

                if (!HashTables.hashTableHasKey(HashTables.getHashTableOf(current.address, storage), pathCopy[0], storage)) {

                    var a = wentThroughPath.concat([pathCopy[0]]).join(Little.keywords.PROPERTY\_ACCESS\_SIGN);

                    Little.runtime.throwError(ErrorMessage('Cannot write a property to ${a}, since ${pathCopy[0]} does not exist (did you forget to define ${a}?)'));

                }

                var hashTableKey = HashTables.hashTableGetKey(HashTables.getHashTableOf(current.address, storage), pathCopy[0], storage);

                current = {

                    address: hashTableKey.value,

                    type: getTypeName(hashTableKey.type),

                }

                wentThroughPath.push(pathCopy.shift());

            }

            if (!getTypeInformation(current.type).passedByReference) {

                Little.runtime.throwError(ErrorMessage('Cannot write to a property to values of a static type. Only objects can have dynamic properties (${wentThroughPath.join(Little.keywords.PROPERTY\_ACCESS\_SIGN)} is `${current.type}`)'));

            }

            if (!HashTables.hashTableHasKey(HashTables.getHashTableOf(current.address, storage), path[path.length - 1], storage)) {

                HashTables.objectAddKey(current.address, path[path.length - 1], retrieve(value), getTypeInformation(type).pointer, storage.storeString(doc),

                    storage);

            } else if (externs.instanceProperties.properties.exists(path[path.length - 1])) {

                Little.runtime.throwError(ErrorMessage('Cannot write to an extern property (${path[path.length - 1]})'));

            } else {

                HashTables.objectSetKey(current.address, path[path.length - 1], {

                    value: value != null ? retrieve(value) : null,

                    type: type != null ? getTypeInformation(type).pointer : null,

                    doc: doc != null ? storage.storeString(doc) : null

                }, storage);

            }

        }

    }

    /\*\*

        Writes a new value to an **\*\*existing\*\*** path specified by `path`.

        If any part of the path does not exist, an error will be thrown in `Little`'s runtime.

        @param path An array of strings representing the path to the value

        @param value The value to write. If `null`, the original value will be preserved

        @param type The type of the value. If `null`, the original type will be preserved

        @param doc The documentation of the value. If `null`, the original documentation will be preserved

    \*\*/

    public function set(path:Array<String>, ?value:InterpTokens, ?type:String, ?doc:String) {

        if (path.length == 0) {

            Little.runtime.throwError(ErrorMessage('Cannot set the value of an empty path'));

            // Does not make sense to have a path of length 0, but still more useful than a quiet return/crash.

        }

        if (path.length == 1) {

            if (referrer.exists(path[0])) {

                referrer.set(path[0], {address: value != null ? retrieve(value) : null, type: type != null ? type : null});

            } else {

                Little.runtime.throwError(ErrorMessage('Variable/function ${path[0]} does not exist'));

            }

        } else {

            var pathCopy = path.slice(0, path.length - 1);

            var wentThroughPath = [path[0]];

            var current = referrer.get(pathCopy.shift());

            while (pathCopy.length > 0) {

                if (!getTypeInformation(current.type).passedByReference) {

                    Little.runtime.throwError(ErrorMessage('Cannot set properties to values of a static type. Only objects can have dynamic properties (${wentThroughPath.join(Little.keywords.PROPERTY\_ACCESS\_SIGN)} is `${current.type}`)'));

                }

                if (!HashTables.hashTableHasKey(HashTables.getHashTableOf(current.address, storage), pathCopy[0], storage)) {

                    var a = wentThroughPath.concat([pathCopy[0]]).join(Little.keywords.PROPERTY\_ACCESS\_SIGN);

                    Little.runtime.throwError(ErrorMessage('Cannot set a property of ${a}, since ${pathCopy[0]} does not exist (did you forget to define ${a}?)'));

                }

                var hashTableKey = HashTables.hashTableGetKey(HashTables.getHashTableOf(current.address, storage), pathCopy[0], storage);

                current = {

                    address: hashTableKey.value,

                    type: getTypeName(hashTableKey.type),

                }

                wentThroughPath.push(pathCopy.shift());

            }

            if (!getTypeInformation(current.type).passedByReference) {

                Little.runtime.throwError(ErrorMessage('Cannot set properties to values of a static type. Only objects can have dynamic properties (${wentThroughPath.join(Little.keywords.PROPERTY\_ACCESS\_SIGN)} is `${current.type}`)'));

            }

            if (HashTables.hashTableHasKey(HashTables.getHashTableOf(current.address, storage), path[path.length - 1], storage)) {

                HashTables.objectSetKey(current.address, path[path.length - 1], {

                    value: value != null ? retrieve(value) : null,

                    type: type != null ? getTypeInformation(type).pointer : null,

                    doc: doc != null ? storage.storeString(doc) : null

                }, storage);

            } else if (externs.instanceProperties.properties.exists(path[path.length - 1])) {

                Little.runtime.throwError(ErrorMessage('Cannot set an extern property (${path[path.length - 1]})'));

            } else {

                Little.runtime.throwError(ErrorMessage('Cannot set the value of ${path.join(Little.keywords.PROPERTY\_ACCESS\_SIGN)}, since ${path[path.length - 1]} does not exist.'));

            }

        }

    }

    /\*\*

        Allocate `size` bytes of memory.

        @param size The number of bytes to allocate

        @return A pointer to the allocated memory

    \*\*/

    public function allocate(size:Int):MemoryPointer {

        if (size <= 0)

            Little.runtime.throwError(ErrorMessage('Cannot allocate ${size} bytes'));

        return storage.storeBytes(size);

    }

    /\*\*

        Free `size` bytes of memory at `pointer`.

        @param pointer The address of the memory to free

        @param size The number of bytes to free

    \*\*/

    public function free(pointer:MemoryPointer, size:Int) {

        if (pointer.toInt() < 0)

            Little.runtime.throwError(ErrorMessage('Cannot free bytes at negative address ${pointer}'));

        if (pointer.toInt() < constants.capacity)

            Little.runtime.throwError(ErrorMessage('Cannot free bytes from the constant pool (addresses 0 to ${constants.capacity}, attempted to free address ${pointer})'));

        if (pointer.toInt() >= currentMemorySize)

            Little.runtime.throwError(ErrorMessage('Cannot free bytes at an address greater than the current memory size (${pointer} requested but ${currentMemorySize} addresses exist)'));

        if (size <= 0)

            Little.runtime.throwError(ErrorMessage('Cannot free ${size} bytes'));

        if (pointer.toInt() + size > currentMemorySize)

            Little.runtime.throwError(ErrorMessage('Cannot free bytes: The requested free overflows the current memory size (${pointer} + ${size} requested but ${currentMemorySize} addresses exist)'));

        storage.freeBytes(pointer, size);

    }

    /\*\*

        Returns the size of the object pointed to by `pointer`.

        @param pointer The pointer to the object

        @param type The type of the object

        @return The size of the object

    \*\*/

    public function sizeOf(pointer:MemoryPointer, type:String):Null<Int> {

        switch type {

            case (\_ == Little.keywords.TYPE\_INT => true): return 4;

            case (\_ == Little.keywords.TYPE\_FLOAT => true): return 8;

            case (\_ == Little.keywords.TYPE\_BOOLEAN => true): return 1;

            case (\_ == Little.keywords.TYPE\_DYNAMIC => true): return null;

            case (\_ == Little.keywords.TYPE\_UNKNOWN => true): return null;

            case (\_ == Little.keywords.TYPE\_MODULE => true): {

                if (externs.externToPointer.exists(type)) return 1;

                else return 16; // 2 \* 4 bytes for hashtable lengths, 2 \* 4 bytes for the pointers

            }

            case (\_ == Little.keywords.TYPE\_STRING => true) |

                 (\_ == Little.keywords.TYPE\_SIGN => true) |

                 (\_ == Little.keywords.TYPE\_CONDITION => true) |

                 (\_ == Little.keywords.TYPE\_FUNCTION => true): {

                var length = storage.readInt32(pointer);

                return 4 + length; // 4 bytes for the length

            }

            case (\_ == Little.keywords.TYPE\_ARRAY => true): {

                var length = storage.readInt32(pointer);

                var elementSize = storage.readInt32(pointer.rawLocation + 4);

                return 8 + length \* elementSize; // 4 bytes for the length, 4 bytes for element size.

            }

            default: return 8; // Probably an object, so 4 bytes for the pointer, 4 bytes for the length

        }

    }

    /\*\*

        Returns information about types in Little at runtime.

        @param name The name of the type. Allows property access (for example, `pack.Type`)

        @return An object containing information about the type.

    \*\*/

    public function getTypeInformation(name:String):TypeInfo {

        // First, check for primitive types which are pre-allocated

        // in the constant pool

        var p = switch name {

            case(\_ == Little.keywords.TYPE\_INT => true): constants.INT;

            case(\_ == Little.keywords.TYPE\_FLOAT => true): constants.FLOAT;

            case(\_ == Little.keywords.TYPE\_BOOLEAN => true): constants.BOOL;

            case(\_ == Little.keywords.TYPE\_DYNAMIC => true): constants.DYNAMIC;

            case(\_ == Little.keywords.TYPE\_MODULE => true): constants.TYPE;

            case(\_ == Little.keywords.TYPE\_UNKNOWN => true): constants.UNKNOWN;

            case \_: MemoryPointer.fromInt(0);

        }

        if (p.rawLocation != 0) {

            return {

                pointer: p,

                typeName: switch p.rawLocation {

                    case 11 /\* int \*/: Little.keywords.TYPE\_INT;

                    case 12 /\* float \*/: Little.keywords.TYPE\_FLOAT;

                    case 13 /\* bool \*/: Little.keywords.TYPE\_BOOLEAN;

                    case 14 /\* dynamic \*/: Little.keywords.TYPE\_DYNAMIC;

                    case 15 /\* type \*/: Little.keywords.TYPE\_MODULE;

                    case 16 /\* unknown \*/: Little.keywords.TYPE\_UNKNOWN;

                    case \_: throw "How did we get here? 5";

                },

                passedByReference: p.rawLocation >= 14 && p.rawLocation <= 15,

                isExternal: false,

                instanceFields: [],

                staticFields: [],

                defaultInstanceSize: switch p.rawLocation {

                    case 11 /\* int \*/: 4;

                    case 12 /\* float \*/: 8;

                    case 13 /\* bool \*/: 1;

                    case 14 /\* dynamic \*/: -1;

                    case 15 /\* type \*/: -1;

                    case 16 /\* unknown \*/: -1;

                    case \_: throw "How did we get here? 51";

                }

            }

        }

        // If it's not a primitive type, the next priority is external types.

        // The easiest way to get a valid type is to check the externToPointer map

        if (externs.externToPointer.exists(name) && externs.getGlobal(name).objectValue.is(CLASS\_POINTER)) {

            var instProps = externs.createPathFor(externs.instanceProperties, ...name.split(Little.keywords.PROPERTY\_ACCESS\_SIGN));

            var statProps = externs.createPathFor(externs.globalProperties, ...name.split(Little.keywords.PROPERTY\_ACCESS\_SIGN));

            var instances = new Map<String, {type:MemoryPointer, doc:MemoryPointer}>();

            var statics = new Map<String, {type:MemoryPointer, doc:MemoryPointer}>();

            for (key => value in instProps.properties)

                instances[key] = {type: value.type, doc: value.doc};

            for (key => value in statProps.properties)

                statics[key] = {type: value.type, doc: value.doc};

            return {

                pointer: externs.externToPointer[name],

                typeName: name,

                passedByReference: true,

                isExternal: true,

                instanceFields: instances,

                staticFields: statics,

                defaultInstanceSize: 4 + POINTER\_SIZE, // Objects take 8 bytes in-place

            }

        }

        var reference = referrer.get(name);

        var typeInfo = storage.readType(reference.address);

        return typeInfo;

    }

    /\*\*

        Returns the name of the type at the given pointer.

        @param pointer The pointer to the type

        @return The name of the type

    \*\*/

    public function getTypeName(pointer:MemoryPointer):String {

        // Externs prioritized:

        var ext = externs.pointerToExtern.get(pointer);

        if (ext != null && externs.getGlobal(...ext.split(".")).objectValue.is(CLASS\_POINTER)) {

            return externs.pointerToExtern[pointer];

        }

        // Then, constants:

        if (constants.hasPointer(pointer)) {

            return constants.getFromPointer(pointer).asJoinedStringPath();

        }

        return storage.readType(pointer).typeName;

    }

}

/\*\*

    Represents runtime information about a type.

\*\*/

typedef TypeInfo = {

    pointer:MemoryPointer,

    typeName:String,

    passedByReference:Bool,

    isExternal:Bool,

    instanceFields:Map<String, {type:MemoryPointer, doc:MemoryPointer}>,

    staticFields:Map<String, {type:MemoryPointer, doc:MemoryPointer}>,

    defaultInstanceSize:Int

}

package little.interpreter.memory;

import haxe.exceptions.ArgumentException;

import little.interpreter.Tokens;

import vision.ds.ByteArray;

using little.tools.Extensions;

/\*\*

    A class allowing access to static constants in Little, without having to allocate memory for them.

\*\*/

class ConstantPool {

    /\*\*

        The amount of bytes the constant pool takes up.

    \*\*/

    public var capacity(default, null):Int = 24;

    public var NULL:MemoryPointer = 0;

    public var FALSE:MemoryPointer = 1;

    public var TRUE:MemoryPointer = 2;

    public var ZERO:MemoryPointer = 3; // size: 8 bytes.

    public var INT:MemoryPointer = 11; // Int primitive type

    public var FLOAT:MemoryPointer = 12; // Float primitive type

    public var BOOL:MemoryPointer = 13; // Bool primitive type

    public var DYNAMIC:MemoryPointer = 14; // Dynamic type

    public var TYPE:MemoryPointer = 15;

    public var UNKNOWN:MemoryPointer = 16; // Unknown type, used for in-place type inference

    public var ERROR:MemoryPointer = 17; // A thrown error has this pointer

    public var EXTERN:MemoryPointer = 18; // An extern function pointer, uses a haxeExtern token and thus cant be stored normally.

    public var EMPTY\_STRING:MemoryPointer = 19; // size: 4 bytes

    /\*\*

        Instantiates a new `ConstantPool`

    \*\*/

    public function new(memory:Memory) {

        for (i in 0...capacity) memory.storage.reserved[i] = 1; // Contains "Core" values

        memory.storage.setByte(TRUE, 1); // TRUE

    }

    /\*\*

        Converts an `InterpTokens` into a `MemoryPointer`, or throws an `ArgumentException` if it doesn't exist in the constant pool.

        @param token The token to convert

        @return The converted `MemoryPointer`

        @throws ArgumentException If the token can't be represented using the constant pool

    \*\*/

    public function get(token:InterpTokens):MemoryPointer {

        switch (token) {

            case NullValue: return NULL;

            case FalseValue: return FALSE;

            case TrueValue: return TRUE;

            case Number(0) | Decimal(0.): return ZERO;

            case Characters(""): return EMPTY\_STRING;

            case (\_.equals(Identifier(Little.keywords.TYPE\_INT)) => true): return INT;

            case (\_.equals(Identifier(Little.keywords.TYPE\_FLOAT)) => true): return FLOAT;

            case (\_.equals(Identifier(Little.keywords.TYPE\_BOOLEAN)) => true): return BOOL;

            case (\_.equals(Identifier(Little.keywords.TYPE\_DYNAMIC)) => true): return DYNAMIC;

            case (\_.equals(Identifier(Little.keywords.TYPE\_MODULE)) => true): return TYPE;

            case (\_.equals(Identifier(Little.keywords.TYPE\_UNKNOWN)) => true): return UNKNOWN;

            case ErrorMessage(\_): return ERROR;

            case FunctionCode(p, \_.parameter(0).filter(x -> x.is(HAXE\_EXTERN)) => true): return EXTERN;

            case \_: throw new ArgumentException("token", '${token} does not exist in the constant pool');

        }

    }

    /\*\*

        Converts a `MemoryPointer` into an `InterpTokens`, or throws an `ArgumentException` if it doesn't exist in the constant pool.

        @param pointer The pointer to convert

        @return The converted `InterpTokens`

        @throws ArgumentException If the pointer doesn't exist in the constant pool

    \*\*/

    public function getFromPointer(pointer:MemoryPointer):InterpTokens {

        return switch pointer.rawLocation {

            case 0x00: NullValue;

            case 0x01: FalseValue;

            case 0x02: TrueValue;

            case 0x03: Number(0);

            case 0x0B: Identifier(Little.keywords.TYPE\_INT);

            case 0x0C: Identifier(Little.keywords.TYPE\_FLOAT);

            case 0x0D: Identifier(Little.keywords.TYPE\_BOOLEAN);

            case 0x0E: Identifier(Little.keywords.TYPE\_DYNAMIC);

            case 0x0F: Identifier(Little.keywords.TYPE\_MODULE);

            case 0x10: Identifier(Little.keywords.TYPE\_UNKNOWN);

            case 0x11: ErrorMessage("Default value for error message");

            case 0x12: HaxeExtern(() -> Characters("Default value for external haxe code"));

            case 0x13: Characters("");

            case \_: throw 'pointer ${pointer} not in constant pool';

        }

    }

    /\*\*

        Checks if a `MemoryPointer` exists in the constant pool.

        @param pointer The pointer to check

        @return `true` if the pointer exists, `false` otherwise

    \*\*/

    public function hasPointer(pointer:MemoryPointer):Bool {

        return pointer.rawLocation < capacity && pointer.rawLocation >= 0;

    }

    /\*\*

        Checks if a type exists in the constant pool.

        @param typeName The type to check

    \*\*/

    public function hasType(typeName:String) {

        switch typeName {

            case (\_ == (Little.keywords.TYPE\_INT) => true): return true;

            case (\_ == (Little.keywords.TYPE\_FLOAT) => true): return true;

            case (\_ == (Little.keywords.TYPE\_BOOLEAN) => true): return true;

            case (\_ == (Little.keywords.TYPE\_DYNAMIC) => true): return true;

            case (\_ == (Little.keywords.TYPE\_MODULE) => true): return true;

            case (\_ == (Little.keywords.TYPE\_UNKNOWN) => true): return true;

            case \_: return false;

        }

    }

    public function getType(typeName:String):MemoryPointer {

        return switch typeName {

            case (\_ == (Little.keywords.TYPE\_INT) => true): INT;

            case (\_ == (Little.keywords.TYPE\_FLOAT) => true): FLOAT;

            case (\_ == (Little.keywords.TYPE\_BOOLEAN) => true): BOOL;

            case (\_ == (Little.keywords.TYPE\_DYNAMIC) => true): DYNAMIC;

            case (\_ == (Little.keywords.TYPE\_MODULE) => true): TYPE;

            case (\_ == (Little.keywords.TYPE\_UNKNOWN) => true): UNKNOWN;

            case \_: throw new ArgumentException("typeName", '${typeName} does not exist in the constant pool');

        }

    }

}

package little.interpreter.memory;

import little.interpreter.Tokens.InterpTokens;

class ExternalInterfacing {

    public var parent:Memory;

    /\*\*

        For each type registered, a pointer to the type must be provided.

    \*\*/

    public var externToPointer:Map<String, MemoryPointer>;

    /\*\*

        Inverse of `externToPointer`, not performance efficient

    \*\*/

    public var pointerToExtern(get ,null):Map<MemoryPointer, String>;

    /\*\* ExternalInterfacing.pointerToExtern getter \*\*/ @:noCompletion function get\_pointerToExtern() {

        var pointerToExtern = new Map<MemoryPointer, String>();

        for (type => pointer in externToPointer) {

            pointerToExtern[pointer] = type;

        }

        return pointerToExtern;

    }

    /\*\*

        Properties of instances of a certain type.

        for example, one may want to define a `length` property on an array

        Use `TYPE\_DYNAMIC` to have a property for every single type. PAY ATTENTION - it blocks this word from being used as a property name for an object.

    \*\*/

    public var instanceProperties:ExtTree = new ExtTree(0, null, null, 0);

    /\*\*

        Global static variables, defined using a path to the property.

    \*\*/

    public var globalProperties:ExtTree = new ExtTree(0, null, null, 0);

    /\*\*

        Instantiates a new `ExternalInterfacing`.

    \*\*/

    public function new(memory:Memory) {

        parent = memory;

        externToPointer = new Map<String, MemoryPointer>();

    }

    /\*\*

        Creates an object at the end of the path. If some of the path does not exist, it will be created.

        This function only creates paths at a specific tree - either `globalProperties` or `instanceProperties`.

        @param extType The tree to create the object in - either `globalProperties` or `instanceProperties`.

        @param path The path to the object. Provided as individual parameters. To use an array, use `...pathArray`

        @return The created object at the end of the path, of type `ExtTree` (External Tree)

    \*\*/

    public function createPathFor(extType:ExtTree, ...path:String):ExtTree {

        var identifiers = path.toArray();

        var handle = extType;

        while (identifiers.length > 0) {

            var identifier = identifiers.shift();

            if (handle.properties.exists(identifier)) {

                handle = handle.properties[identifier];

            } else {

                handle.properties[identifier] = new ExtTree();

                handle = handle.properties[identifier];

            }

        }

        return handle;

    }

    /\*\*

        Helper function that uses `createPathFor` to create all possible paths for an object,

        on both `globalProperties` and `instanceProperties`.

        @param path The path to the object. Provided as individual parameters. To use an array, use `...pathArray`

    \*\*/

    public function createAllPathsFor(...path:String) {

        for (tree in [globalProperties, instanceProperties]) {

            createPathFor(tree, ...path);

        }

    }

    /\*\*

        Checks if a static object exists at the end of the path

        @param path The path to the object. Provided as individual parameters. To use an array, use `...pathArray`

        @return `true` if the object exists, `false` otherwise

    \*\*/

    public function hasGlobal(...path:String):Bool {

        var identifiers = path.toArray();

        var handle = globalProperties;

        while (identifiers.length > 0) {

            var identifier = identifiers.shift();

            if (handle.properties.exists(identifier))

                handle = handle.properties[identifier];

            else

                return false;

        }

        return true;

    }

    /\*\*

        Checks if an instance field exists at the end of the path

        @param path The path to the object. Provided as individual parameters. To use an array, use `...pathArray`

        @return `true` if the object exists, `false` otherwise

    \*\*/

    public function hasInstance(...path:String):Bool {

        var identifiers = path.toArray();

        var handle = instanceProperties;

        while (identifiers.length > 0) {

            var identifier = identifiers.shift();

            if (handle.properties.exists(identifier))

                handle = handle.properties[identifier];

            else

                return false;

        }

        return true;

    }

    /\*\*

        Gets a static object at the end of the path.

        @param path The path to the object. Provided as individual parameters. To use an array, use `...pathArray`

        @return The object at the end of the path, as a combination of `InterpTokens` and `MemoryPointer`

    \*\*/

    public function getGlobal(...path:String):{objectValue:InterpTokens, objectAddress:MemoryPointer} {

        var identifiers = path.toArray();

        var handle = globalProperties;

        for (ident in identifiers) {

            handle = handle.properties[ident];

        }

        return handle.getter(null, -1); // Static externs are not tied to any runtime object, so this makes sense

    }

}

/\*\*

    The External Object Tree. Used to store information about an external object.

\*\*/

class ExtTree {

    /\*\*

        A getter for the extern value.

        The returned token has its parent's address in memory and value, if you want to modify it.

    \*\*/

    public var getter:(objectValue:InterpTokens, objectAddress:MemoryPointer) -> {objectValue:InterpTokens, objectAddress:MemoryPointer};

    /\*\*

        A pointer to this prop's doc. Used instead of string to avoid re-allocations.

    \*\*/

    public var doc:MemoryPointer;

    /\*\*

        A pointer to the type this prop's getter returns. Used for providing consistent behavior for runtime type info acquisition.

    \*\*/

    public var type:MemoryPointer;

    /\*\*

        This `ExtTree`'s children.

    \*\*/

    public var properties:Map<String, ExtTree>;

    /\*\*

        Instantiates a new `ExtTree`

        @param type The `Little` type this `ExtTree`'s getter should return. Used for runtime type information

        @param getter The getter for the `ExtTree`, can be used in 2 ways - in the global tree, the two arguments are `null` and `null`.

        In the instance tree, the arguments are the parent object's value and it's address in memory. In both cases, the returned value should be a value,

        and it's address in memory.

        @param properties The properties of this tree. Used to quickly populate this `ExtTree` with child trees.

        @param doc The documentation of the field this `ExtTree` represents.

    \*\*/

    public function new(?type:MemoryPointer, ?getter:(objectValue:InterpTokens, objectAddress:MemoryPointer) -> {objectValue:InterpTokens, objectAddress:MemoryPointer}, ?properties:Map<String, ExtTree>, ?doc:MemoryPointer) {

        this.getter = getter ?? (objectValue, objectAddress) -> {

            return {

                objectValue: Characters('Externally registered, attached to $objectAddress'),

                objectAddress: objectAddress,

            }

        }

        this.properties = properties ?? new Map<String, ExtTree>();

        this.doc = doc ?? Little.memory.constants.EMPTY\_STRING;

        this.type = type ?? Little.memory.constants.UNKNOWN;

    }

}

package little.interpreter.memory;

import little.interpreter.memory.MemoryPointer.POINTER\_SIZE;

import haxe.Int64;

import haxe.io.Bytes;

import haxe.hash.Murmur1;

import vision.ds.ByteArray;

class HashTables {

    public static final OBJECT\_HASH\_TABLE\_CELL\_SIZE:Int = POINTER\_SIZE \* 4;

    /\*\*

        Returns a hash table for the given key-value-type pairs.

        Each hash-table value will be 3 pointers, first to the name

        of the field, second to the value, and third to the type of the field.

        The hash-table's size is pre-estimated, and should provide a hash table

        with 0.5 size-to-store ratio.

        @param pairs an array of key-value-type triples

    \*\*/

    public static function generateObjectHashTable(pairs:Array<{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer}>) {

        var initialLength = (pairs.length > 1 ? pairs.length : 5) \* OBJECT\_HASH\_TABLE\_CELL\_SIZE \* 3;

        // a memory pointer is 8 bytes, 3 pointers is `OBJECT\_HASH\_TABLE\_CELL\_SIZE` bytes

        // We triple the memory for a nice size-to-store ratio (0.33)

        var array = new ByteArray(initialLength);

        for (pair in pairs) {

            var keyHash = Murmur1.hash(Bytes.ofString(pair.key));

            // Since the array is `OBJECT\_HASH\_TABLE\_CELL\_SIZE` bytes per entry, We need to assure that keyIndex is divisible by `OBJECT\_HASH\_TABLE\_CELL\_SIZE`

            // What the following line does is assure the value doesn't overflow and wrap around to the negative.

            // Basically, increase the ceiling, multiply by `OBJECT\_HASH\_TABLE\_CELL\_SIZE`, take the remainder, and re-reduce the ceiling.

            var khI64 = Int64.make(0, keyHash);

            var keyIndex = ((khI64 \* OBJECT\_HASH\_TABLE\_CELL\_SIZE) % array.length).low;

            if (array.getInt32(keyIndex) == 0) { // Always ok, on existing cells the first value cant be 0 because it represents `null`, and `null` fields are not creatable.

                array.setInt32(keyIndex, pair.keyPointer.rawLocation);

                array.setInt32(keyIndex + POINTER\_SIZE, pair.value.rawLocation);

                array.setInt32(keyIndex + POINTER\_SIZE \* 2, pair.type.rawLocation);

                array.setInt32(keyIndex + POINTER\_SIZE \* 3, pair.doc.rawLocation);

            } else {

                // To handle collisions, we will basically move on until we find an empty slot

                // Then, fill it with the new key-value-type triplet

                var incrementation = 0;

                var i = keyIndex;

                while (array.getInt32(i) != 0) {

                    i += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

                    incrementation += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

                    if (i >= array.length) {

                        i = 0;

                    }

                    if (incrementation >= array.length) {

                        throw 'Object hash table did not generate. This should never happen. Initial length may be incorrect.';

                    }

                }

                array.setInt32(i, pair.keyPointer.rawLocation);

                array.setInt32(i + POINTER\_SIZE, pair.value.rawLocation);

                array.setInt32(i + POINTER\_SIZE \* 2, pair.type.rawLocation);

                array.setInt32(i + POINTER\_SIZE \* 3, pair.doc.rawLocation);

            }

        }

        return array;

    }

    /\*\*

        Reads an object hash table, optionally provides key names when the storage is given.

        @param bytes the hash table's bytes

        @param storage the storage, if key names are needed. When not provided, key names are `null`.

        @return the hash table as an array.

    \*\*/

    public static function readObjectHashTable(bytes:ByteArray, ?storage:Storage):Array<{key:Null<String>, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer}> {

        var arr = [];

        var i = 0;

        while (i < bytes.length) {

            var keyPointer:MemoryPointer = bytes.getInt32(i);

            var value:MemoryPointer = bytes.getInt32(i + POINTER\_SIZE);

            var type:MemoryPointer = bytes.getInt32(i + POINTER\_SIZE \* 2);

            var doc:MemoryPointer = bytes.getInt32(i + POINTER\_SIZE \* 3);

            var key = null;

            if (keyPointer.rawLocation == 0) {

                i += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

                continue; // Nothing to do here

            }

            if (storage != null) {

                key = storage.readString(keyPointer);

            }

            arr.push({

                key: key,

                keyPointer: keyPointer,

                value: value,

                type: type,

                doc: doc

            });

            i += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

        }

        return arr;

    }

    /\*\*

        Whether a given key exists in a hash table.

        @param hashTable The bytes of the hash table, generated using the `HashTables` class

        @param key The key to check

        @param storage Must be provided in order to actually access the key values in the hash table

        @return `true` if the key exists, `false` otherwise

    \*\*/

    public static function hashTableHasKey(hashTable:ByteArray, key:String, storage:Storage):Bool {

        var keyHash = Murmur1.hash(Bytes.ofString(key));

        var khI64 = Int64.make(0, keyHash);

        var keyIndex = ((khI64 \* OBJECT\_HASH\_TABLE\_CELL\_SIZE) % hashTable.length).low;

        var incrementation = 0;

        while (true) {

            var currentKey = storage.readString(hashTable.getInt32(keyIndex));

            if (currentKey == key) {

                return true;

            }

            keyIndex += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

            incrementation++;

            if (keyIndex >= hashTable.length) {

                keyIndex = 0;

            }

            if (incrementation >= hashTable.length) {

                return false;

            }

        }

    }

    /\*\*

        Looks up a key in a hash table.

        @param hashTable The bytes of the hash table, generated using the `HashTables` class

        @param key The key to check

        @param storage Must be provided in order to actually access the keys in the hash table

    \*\*/

    public static function hashTableGetKey(hashTable:ByteArray, key:String, storage:Storage):{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer} {

        var keyHash = Murmur1.hash(Bytes.ofString(key));

        var khI64 = Int64.make(0, keyHash);

        var keyIndex = ((khI64 \* OBJECT\_HASH\_TABLE\_CELL\_SIZE) % hashTable.length).low;

        var incrementation = 0;

        while (true) {

            var currentKey = storage.readString(hashTable.getInt32(keyIndex));

            if (currentKey == key) {

                return {

                    key: key,

                    keyPointer: hashTable.getInt32(keyIndex),

                    value: hashTable.getInt32(keyIndex + POINTER\_SIZE),

                    type: hashTable.getInt32(keyIndex + POINTER\_SIZE \* 2),

                    doc: hashTable.getInt32(keyIndex + POINTER\_SIZE \* 3)

                }

            }

            keyIndex += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

            incrementation += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

            if (keyIndex >= hashTable.length) {

                keyIndex = 0;

            }

            if (incrementation >= hashTable.length) {

                throw 'Key $key not found in hash table';

            }

        }

        throw 'How did you get here? 4';

    }

    /\*\*

        Directly accesses a specific object's memory, and adds a key-value "pair" to it's hash table.

        If the hash table is too full (70% of its size is occupied), the hash table will be rehashed with the new key, and it's size will increase.

        @param object A pointer to an object

        @param key The key to add

        @param value The key's value

        @param type The key's type

        @param doc The key's documentation

        @param storage Must be provided in order to access the object.

    \*\*/

    public static function objectAddKey(object:MemoryPointer, key:String, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer, storage:Storage) {

        var hashTableBytes = storage.readBytes(storage.readPointer(object.rawLocation + POINTER\_SIZE), storage.readInt32(object.rawLocation));

        var table = HashTables.readObjectHashTable(hashTableBytes, storage);

        // Fresh objects have 0.33 size-to-fill ratio, so usually we would just need to hash and add a key.

        // In case the size-to-fill ration is grater that 0,7, we will need to rehash everything and add the key

        var tableSize = hashTableBytes.length;

        var occupied = table.length \* OBJECT\_HASH\_TABLE\_CELL\_SIZE;

        if (occupied / tableSize >= 0.7) {

            // Rehash with the the key:

            table.push({

                key: key,

                keyPointer: storage.storeString(key),

                value: value,

                type: type,

                doc: doc

            });

            var newHashTable = HashTables.generateObjectHashTable(table);

            // Free the old hash table:

            storage.freeBytes(storage.readPointer(object.rawLocation + POINTER\_SIZE), hashTableBytes.length);

            // Store the new one, retrieve the pointer to it:

            var tablePointer = storage.storeBytes(newHashTable.length, newHashTable);

            // Update the object's hash table pointer:

            storage.setPointer(object.rawLocation + 4, tablePointer);

            // Don't forget, the table length also needs to be replaced

            storage.setInt32(object.rawLocation, newHashTable.length);

            return; // The object was rehashed, the given pointer is still valid and all fields are good. Done here.

        }

        var hashTablePosition = storage.readPointer(object.rawLocation + 4);

        var keyHash = Murmur1.hash(Bytes.ofString(key));

        var khI64 = Int64.make(0, keyHash);

        var keyIndex = ((khI64 \* OBJECT\_HASH\_TABLE\_CELL\_SIZE) % hashTableBytes.length).low;

        var incrementation = 0;

        while (true) {

            if (hashTableBytes.getInt32(keyIndex) == 0) {

                storage.setPointer(hashTablePosition.rawLocation + keyIndex, storage.storeString(key).rawLocation);

                storage.setPointer(hashTablePosition.rawLocation + keyIndex + POINTER\_SIZE, value.rawLocation);

                storage.setPointer(hashTablePosition.rawLocation + keyIndex + POINTER\_SIZE \* 2, type.rawLocation);

                storage.setPointer(hashTablePosition.rawLocation + keyIndex + POINTER\_SIZE \* 3, doc.rawLocation);

                return;

            }

            keyIndex += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

            incrementation += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

            if (keyIndex >= tableSize) {

                keyIndex = 0;

            }

            if (incrementation >= tableSize) {

                throw "How did you get here? 6";

            }

        }

    }

    /\*\*

        Directly accesses a specific object's memory, and sets a key-value "pair" to it's hash table.

        @param object A pointer to an object

        @param key The key to add

        @param pair The key's value. Has 3 fields: value, type, doc. if a property is null, it will not modify that specific existing value of the key-value "pair"

        @param storage Must be provided in order to access the object.

    \*\*/

    public static function objectSetKey(object:MemoryPointer, key:String, pair:{?value:MemoryPointer, ?type:MemoryPointer, ?doc:MemoryPointer}, storage:Storage) {

        var hashTableBytesLength = storage.readInt32(object.rawLocation);

        var hashTablePosition = storage.readPointer(object.rawLocation + 4);

        var keyHash = Murmur1.hash(Bytes.ofString(key));

        var khI64 = Int64.make(0, keyHash);

        var keyIndex = ((khI64 \* OBJECT\_HASH\_TABLE\_CELL\_SIZE) % hashTableBytesLength).low;

        var incrementation = 0;

        while (true) {

            var currentKey = storage.readString(storage.readPointer(hashTablePosition.rawLocation + keyIndex));

            if (currentKey == key) {

                if (pair.value != null)

                    storage.setInt32(hashTablePosition.rawLocation + keyIndex + POINTER\_SIZE, pair.value.rawLocation);

                if (pair.type != null)

                    storage.setInt32(hashTablePosition.rawLocation + keyIndex + POINTER\_SIZE \* 2, pair.type.rawLocation);

                if (pair.doc != null)

                    storage.setInt32(hashTablePosition.rawLocation + keyIndex + POINTER\_SIZE \* 3, pair.doc.rawLocation);

                return;

            }

            keyIndex += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

            incrementation += OBJECT\_HASH\_TABLE\_CELL\_SIZE;

            if (keyIndex >= hashTableBytesLength) {

                keyIndex = 0;

            }

            if (incrementation >= hashTableBytesLength) {

                throw "Cannot set a non-existing key in the object's hash table.";

            }

        }

    }

    /\*\*

        Looks up a key in an object's hash table using only the object's pointer.

        @param object A pointer to an object

        @param key The key to look up

        @param storage Must be provided in order to access the object.

    \*\*/

    public static function objectGetKey(object:MemoryPointer, key:String, storage:Storage):{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer} {

        var hashTableBytes = storage.readBytes(storage.readPointer(object.rawLocation + POINTER\_SIZE), storage.readInt32(object.rawLocation));

        return hashTableGetKey(hashTableBytes, key, storage);

    }

    /\*\*

        Retrieves the hash table of an object, as an array of bytes.

        @param objectPointer A pointer to an object

        @param storage Must be provided in order to access the object.

    \*\*/

    public static function getHashTableOf(objectPointer:MemoryPointer, storage:Storage):ByteArray {

        var bytesLength = storage.readInt32(objectPointer.rawLocation);

        var bytesPointer = storage.readPointer(objectPointer.rawLocation + 4);

        return storage.readBytes(bytesPointer, bytesLength);

    }

}

package little.interpreter.memory;

import haxe.io.Bytes;

import haxe.Int64;

inline var POINTER\_SIZE = MemoryPointer.POINTER\_SIZE;

/\*\*

    An abstract over a 32-bit integer representing a memory address.

    Due to language limitations, we can't use an `Int64` for this. (no cross-platform support)

\*\*/

abstract MemoryPointer(Int) {

    public static inline var POINTER\_SIZE:Int = 4; //Currently, since byte array indices are 32bit.

    public var rawLocation(get, set):Int;

    /\*\* MemoryPointer.rawLocation getter \*\*/ inline function get\_rawLocation() return this;

    /\*\* MemoryPointer.rawLocation setter \*\*/ inline function set\_rawLocation(v:Int) return this = v;

    /\*\*

        Instantiates a new `MemoryPointer`.

    \*\*/

    public function new(address:Int) {

        this = address;

    }

    /\*\*

        Converts an `Int` to a `MemoryPointer`.

    \*\*/

    @:from public static function fromInt(i:Int) {

        return new MemoryPointer(i);

    }

    /\*\*

        Returns a string representation of this address.

    \*\*/

    @:to public function toString() {

        return this + "";

    }

    /\*\*

        Converts this address to an array of bytes, representing a 64-bit number.

        The last 4 bytes are filled with zeros.

        @return Array<Int>

    \*\*/

    public function toArray():Array<Int> {

        var bytes = [];

        var i = rawLocation;

        for (\_ in 0...4) {

            bytes.unshift(i & 0xFF);

            i = i >> 8;

        }

        for (\_ in 0...4) {

            bytes.unshift(0);

        }

        return bytes;

    }

    /\*\*

        Converts this address to an array of bytes, representing a 32-bit number.

    \*\*/

    public function toBytes():Bytes {

        var bytes = Bytes.alloc(4);

        for (i in 0...3) {

            bytes.set(i, (rawLocation >> (i \* 8)) & 0xFF);

        }

        return bytes;

    }

    public function toInt():Int {

        return this;

    }

}

package little.interpreter.memory;

import haxe.ds.ArraySort;

import vision.algorithms.Radix;

import little.tools.PrettyPrinter;

import haxe.extern.EitherType;

import little.interpreter.Tokens.InterpTokens;

using little.tools.TextTools;

using StringTools;

/\*\*

    An extension of `little.interpreter.memory.ExternalInterfacing`, that adds support for external operators.

\*\*/

@:access(little.lexer.Lexer)

@:allow(little.interpreter.Interpreter)

@:allow(little.tools.Plugins)

class Operators {

    /\*\*

        Instantiates the `Little.memory.operators` class.

    \*\*/

    public function new() {}

    /\*\*

        A map containing the priority of each operator, sorted by index to an array of operand-position-dependent operators.

        for example:

        | Priority | Little.memory.operators |

        | :---: | :---: |

        | 0 | `{sign: "+", side: STANDARD}`, `{sign: "-", side: STANDARD}` |

        | 1 | `{sign: "\*", side: STANDARD}`, `{sign: "/", side: STANDARD}` |

        | 2 | `{sign: "^", side: STANDARD}`, `{sign: "√", side: RHS\_ONLY}` |

    \*\*/

    public var priority:Map<Int, Array<{sign:String, side:OperatorType}>> = [];

    /\*\*

        Little.memory.operators that require two sides to work, for example:

        | Operator | Code |

        | :---: | :---: |

        | Add | `5 + 5` |

        | Subtract | `5 - 5` |

        | Exponentiation | `5^2` |

        | "Non-Standard" Square Root  | `3√5` |

    \*\*/

    public var standard:Map<String, (lhs:InterpTokens, rhs:InterpTokens) -> InterpTokens> = new Map();

    /\*\*

        Little.memory.operators that require just the right side of the equations, for example:

        | Operator | Code |

        | :---: | :---: |

        | Negate | `-5` |

        | Increment | `++5` |

        | Decrement | `--5` |

        | "Standard" Square Root  | `√5` |

    \*\*/

    public var rhsOnly:Map<String, (InterpTokens) -> InterpTokens> = new Map();

    /\*\*

        Little.memory.operators that require just the left side of the equations, for example:

        | Operator | Code |

        | :---: | :---: |

        | Post Increment | `5++` |

        | Post Decrement | `5--` |

        | Factorial | `5!` |

    \*\*/

    public var lhsOnly:Map<String, (InterpTokens) -> InterpTokens> = new Map();

    /\*\*

        Format of parameter `opPriority`:

        ### Notice

         - When using relative (`before`/`after`) positions, make sure the referenced operator exists. otherwise, it won't be inserted at all...

        | Option | Meaning | Example |

        | :--- | :--- | :---: |

        | `<index>` | Inserts the operator at the specified priority level. | `2`, `1`, `5` |

        | `first` | Inserts the operator at the first priority level (index `0`). | `first` |

        | `last` | Inserts the operator at the last priority level (index `priority.length - 1`). | `last` |

        | `with \_<sign>\_` | Inserts the operator at the same priority level as the given sign. The sign is surrounded by underscores, which means the sign is of type `LHS\_RHS`.| `with \_+\_`, `with \_\*\_`|

        | `between <sign1> <sign2>` | Inserts the operator between the two signs. the signs are **\*\*not\*\*** surrounded by **\*\*any underscores\*\***, which means these signs are of type `LHS\_RHS`.| `between ^ +\_`, ` between \_\*\_ -`|

        | `before \_<sign>` | Inserts the operator before the sign. the sign is surrounded by underscores, which means the sign is of type `LHS\_ONLY`.| `before \_!`|

        | `after <sign>\_` | Inserts the operator after the sign. the has only one underscore to the right of it, which means the sign is of type `RHS\_ONLY`.| `after -\_`, `after +\_`|

    \*\*/

    public function setPriority(op:String, type:OperatorType, opPriority:String) {

        var obj = {sign: op, side: type};

        if (opPriority == "first") {

            if (priority[-1] == null)

                priority[-1] = [];

            priority[-1].push(obj);

        } else if (opPriority == "last") {

            var i = -1;

            for (key in priority.keys())

                if (i < key)

                    i = key;

            if (priority[i + 1] == null)

                priority[i + 1] = [];

            priority[i + 1].push(obj);

        } else if (~/[0-9]+/.match(opPriority)) {

            var p = Std.parseInt(opPriority);

            if (priority[p] == null)

                priority[p] = [];

            priority[p].push(obj);

        } else if (opPriority.startsWith("before") || opPriority.startsWith("after") || opPriority.startsWith("with")) {

            var destinationOp, opSide;

            var signPos = opPriority.remove("before").remove("after").remove("with").trim();

            if (signPos.countOccurrencesOf("\_") != 1) {

                destinationOp = signPos.replace("\_", "");

                opSide = LHS\_RHS;

            } else if (signPos.startsWith("\_")) {

                destinationOp = signPos.replace("\_", "");

                opSide = LHS\_ONLY;

            } else {

                destinationOp = signPos.replace("\_", "");

                opSide = RHS\_ONLY;

            }

            for (key => value in priority) {

                if (value.filter(x -> x.side == opSide && x.sign == destinationOp).length > 0) {

                    if (opPriority.startsWith("before")) {

                        if (priority[key - 1] == null)

                            priority[key - 1] = [];

                        priority[key - 1].push(obj);

                    } else if (opPriority.startsWith("after")) {

                        if (priority[key + 1] == null)

                            priority[key + 1] = [];

                        priority[key + 1].push(obj);

                    } else {

                        // if inserted on the same priority level, and a priority level already exists since the

                        // sign was found on it, we can assume priority[key] already exists

                        priority[key].push(obj);

                    }

                    break;

                }

            }

        } else if (opPriority.startsWith("between")) {

            var signPos = opPriority.remove("between").trim();

            var signs = signPos.split(" ").map(x -> x.trim());

            var sign1Data = signPosToObject(signs[0]);

            var sign2Data = signPosToObject(signs[1]);

            var sign1Level = -1, sign2Level = -1;

            for (key => value in priority) {

                if (value.filter(x -> x.side == sign1Data.side && x.sign == sign1Data.sign).length > 0) {

                    sign1Level = key;

                }

                if (value.filter(x -> x.side == sign2Data.side && x.sign == sign2Data.sign).length > 0) {

                    sign2Level = key;

                }

            }

            if (sign1Level != -1 && sign2Level != -1 && sign1Level != sign2Level && Math.abs(sign1Level - sign2Level) <= 2) {

                if (Math.abs(sign1Level - sign2Level) == 2) {

                    var key = Std.int((sign1Level + sign2Level) / 2);

                    if (priority[key] == null)

                        priority[key] = [];

                    priority[key].push(obj);

                } else {

                    // We need to create a new level between sign1Level and sign2Level, and push everything

                    // after the sign were inserting now backwards

                    var insert = Std.int(Math.max(sign1Level, sign2Level));

                    var newMap = new Map<Int, Array<{sign:String, side:OperatorType}>>();

                    for (k => v in priority) {

                        if (k < insert) {

                            newMap[k] = v;

                        } else {

                            newMap[k + 1] = v;

                        }

                    }

                    newMap[insert] = [obj];

                    priority = newMap;

                }

            }

        }

        // We're working on a map, and negative indices can be used as keys. we need to make sure that

        // all indices are positive, while keeping the order.

        // More than that, we need the list to start at 0, so if theres no 0 in the list, we need to move everything downwards.

        var a = [for (x in priority.keys()) x];

        if (a.length == 0) return;

        ArraySort.sort(a, (x, y) -> x - y);

        var minimumKey = a[0];

        if (minimumKey != 0) {

            var diff = 0 - minimumKey;

            var priorityCopy = new Map<Int, Array<{sign:String, side:OperatorType}>>();

            for (key => value in priority) {

                priorityCopy[key + diff] = value;

            }

            priority = priorityCopy;

        }

    }

    /\*\*

        Returns the 0-based priority of the given operator.

        @param op The operator

        @param type The side of the operator - `LHS\_ONLY`, `RHS\_ONLY` or `LHS\_RHS`

        @return The operator's priority

    \*\*/

    public function getPriority(op:String, type:OperatorType):Int {

        for (index => key in priority)

            if (key.filter(x -> x.sign == op && x.side == type).length > 0) return index;

        throw 'Operator $op not found';

    }

    /\*\*

        Iterates over the operators in arrays ordered by their priority, from `0` to `n`.

    \*\*/

    public function iterateByPriority():Iterator<Array<{sign:String, side:OperatorType}>> {

        var a = [for (x in priority.keys()) x];

        ArraySort.sort(a, (x, y) -> x - y);

        var b = [for (x in a) priority[x]];

        var i = 0;

        return {

            next: () -> b[i++],

            hasNext: () -> i < b.length

        }

    }

    /\*\*

        Adds an operator to be used in the program's runtime.

        @param op the operator itself

        @param operatorType  **\*\*STANDARD\*\*** - operator that works with both sides of the equation, for example: `5 + 5` or `5 - 5`.

**\*\*LHS\_ONLY\*\*** - operator that only works with the left side of the equation, for example: `5++` or `5--`.

**\*\*RHS\_ONLY\*\*** - operator that only works with the right side of the equation, for example: `-5` or `++5`.

        @param priority a string indicating the priority of the operator using positional info/actual index. see `Little.memory.operators.setPriority`

        @param callback depending on the operatorType, either a function that takes two arguments (lhs, rhs) or a function that takes one argument (lhs) or (rhs).

    \*\*/

    public function add(op:String, operatorType:OperatorType, priority:String,

            callback:EitherType<(InterpTokens) -> InterpTokens, (InterpTokens, InterpTokens) -> InterpTokens>) {

        for (i in 0...op.length)

            if (!KeywordConfig.recognizedOperators.contains(op.charAt(i)))

                KeywordConfig.recognizedOperators.push(op.charAt(i));

        Little.keywords.RECOGNIZED\_SIGNS.push(op);

        switch operatorType {

            case LHS\_RHS:

                {

                    standard.set(op, callback);

                }

            case LHS\_ONLY:

                {

                    lhsOnly.set(op, callback);

                }

            case RHS\_ONLY:

                {

                    rhsOnly.set(op, callback);

                }

        }

        setPriority(op, operatorType, priority);

    }

    /\*\*

        Calls the operator `op` with the argument `lhs` to the left side of the equation.

    \*\*/

    overload extern inline public function call(lhs:InterpTokens, op:String) {

        if (lhsOnly.exists(op))

            return lhsOnly[op](lhs);

        else if (rhsOnly.exists(op))

            return

                ErrorMessage('Operator $op is used incorrectly - should appear after the sign ($op${PrettyPrinter.stringifyInterpreter(lhs)} instead of ${PrettyPrinter.stringifyInterpreter(lhs)}$op)');

        else if (standard.exists(op))

            return ErrorMessage('Operator $op is used incorrectly - should appear between two values (${PrettyPrinter.stringifyInterpreter(lhs)} $op <some value>)');

        else

            return ErrorMessage('Operator $op does not exist. did you make a typo?');

    }

    /\*\*

        Calls the operator `op` with the argument `rhs` to the right side of the equation.

    \*\*/

    overload extern inline public function call(op:String, rhs:InterpTokens) {

        if (rhsOnly.exists(op))

            return rhsOnly[op](rhs);

        else if (lhsOnly.exists(op))

            return

                ErrorMessage('Operator $op is used incorrectly - should appear before the sign (${PrettyPrinter.stringifyInterpreter(rhs)}$op instead of $op${PrettyPrinter.stringifyInterpreter(rhs)})');

        else if (standard.exists(op))

            return ErrorMessage('Operator $op is used incorrectly - should appear between two values (${PrettyPrinter.stringifyInterpreter(rhs)} $op <some value>)');

        else

            return ErrorMessage('Operator $op does not exist. did you make a typo?');

    }

    /\*\*

        Calls the operator `op` with the arguments `lhs` and `rhs` to the left and right side of the equation respectively.

    \*\*/

    overload extern inline public function call(?lhs:InterpTokens = null, op:String, ?rhs:InterpTokens = null):InterpTokens {

        if (standard.exists(op))

            return standard[op](lhs, rhs);

        else if (lhsOnly.exists(op))

            return

                ErrorMessage('Operator $op is used incorrectly - should not appear between two values, only to the right of one of them (${PrettyPrinter.stringifyInterpreter(rhs)}$op or ${PrettyPrinter.stringifyInterpreter(lhs)}$op)');

        else if (rhsOnly.exists(op))

            return

                ErrorMessage('Operator $op is used incorrectly - should not appear between two values, only to the left of one of them ($op${PrettyPrinter.stringifyInterpreter(rhs)} or $op${PrettyPrinter.stringifyInterpreter(lhs)})');

        else

            return ErrorMessage('Operator $op does not exist. did you make a typo?');

    }

    /\*\*

        Converts shortened `\_+\_` syntax that includes both the operator and it's side to a sign-`OperatorType` pair.

        @param signPos a string containing the operator and it's sides. see `Little.memory.operators.setPriority` for syntax

        @return a sign-`OperatorType` pair

    \*\*/

    function signPosToObject(signPos:String):{sign:String, side:OperatorType} {

        var destinationOp, opSide;

        if (signPos.countOccurrencesOf("\_") != 1) {

            destinationOp = signPos.replace("\_", "");

            opSide = LHS\_RHS;

        } else if (signPos.startsWith("\_")) {

            destinationOp = signPos.replace("\_", "");

            opSide = LHS\_ONLY;

        } else {

            destinationOp = signPos.replace("\_", "");

            opSide = RHS\_ONLY;

        }

        return {sign: destinationOp, side: opSide};

    }

}

/\*\*

    Types of operators. Rhs means the operand is on the right hand side,

    while Lhs means the operand is on the left hand side.

\*\*/

enum OperatorType {

    LHS\_RHS;

    LHS\_ONLY;

    RHS\_ONLY;

}

package little.interpreter.memory;

import haxe.Int64;

import haxe.hash.Murmur1;

import little.interpreter.memory.MemoryPointer.POINTER\_SIZE;

import vision.ds.ByteArray;

import little.tools.PrettyPrinter;

using little.tools.Extensions;

/\*\*

    A tool to map variable names to their location in memory.

    It's storage method is similar to buddy-blocks with the heap, but each block points both backwards and forwards.

\*\*/

class Referrer {

    /\*\*

        4 bytes for the hash, 1 pointer for the string, 1 pointer for the address in memory, and 1 pointer for the type.

    \*\*/

    public static var KEY\_SIZE:Int = POINTER\_SIZE \* 3 + 4;

    public var parent:Memory;

    public var bytes:ByteArray;

    public var currentScopeStart(get, null):Int;

    /\*\* Referrer.currentScopeStart getter \*\*/ @:noCompletion function get\_currentScopeStart() return bytes.getInt32(0);

    public var currentScopeLength(get, null):Int;

    /\*\* Referrer.currentScopeLength getter \*\*/ @:noCompletion function get\_currentScopeLength() return bytes.getUInt16(bytes.getInt32(0) + 2);

    /\*\*

        Instantiates a new `Referrer`.

    \*\*/

    public function new(memory:Memory) {

        parent = memory;

        bytes = new ByteArray(1024); // 1mb, no need to be too big

        bytes.setInt32(0, 4); // The current scope starts at position 4.

        bytes.setUInt16(4, 0); // always 0 elements backwards, were at the start,

        bytes.setUInt16(6, 0); // Were just starting, forward is 0 and will change.

    }

    /\*\*

        Requests 1024 bytes of extra memory for the referrer.

    \*\*/

    function requestMemory() {

        if (bytes.length > parent.maxMemorySize) {

            Little.runtime.throwError(ErrorMessage('Too many scopes have been created, referrer\'s stack has overflown (check for infinite recursion)'), MEMORY\_REFERRER);

        }

        bytes.resize(bytes.length + 1024);

    }

    /\*\*

        Creates a new scope. Fields created after this will be added to the new scope, and won't affect fields from the previous scopes.

    \*\*/

    public function pushScope() {

        var currentScopeLength = bytes.getUInt16(bytes.getInt32(0) + 2);

        var currentScopeStart = bytes.getInt32(0) + 4; // each header is 4 bytes long.

        var header = new ByteArray(4);

        header.setUInt16(0, currentScopeLength);

        header.setUInt16(2, 0); // Currently, 0 elements ahead.

        var writePosition = currentScopeStart + currentScopeLength \* KEY\_SIZE;

        if (writePosition + 2 + 2 > bytes.length) {

            requestMemory();

        }

        bytes.setBytes(writePosition, header);

        bytes.setInt32(0, writePosition); // Update the start of the scope.

    }

    /\*\*

        Removes the last scope. TODO: Garbage collection.

    \*\*/

    public function popScope() {

        var currentScopePosition = bytes.getInt32(0);

        var previousScopeLength = bytes.getUInt16(currentScopePosition);

        var currentScopeLength = bytes.getUInt16(currentScopePosition + 2);

        // Update the start of the scope. Also, -4 is for the header, since we need to point to its start.

        bytes.setInt32(0, currentScopePosition - previousScopeLength \* KEY\_SIZE - 4);

    }

    /\*\*

        References a variable in the current scope. If it already exists in the current scope, it will be overwritten.

        If it exists in parent scopes, they won't be affected.

        @param key The name of the variable

        @param address The address of the value

        @param type The type of the value

    \*\*/

    public function reference(key:String, address:MemoryPointer, type:String) {

        var keyHash = Murmur1.hash(ByteArray.from(key));

        var stringName = parent.storage.storeString(key);

        var writePosition = currentScopeStart + 4 + currentScopeLength \* KEY\_SIZE;

        if (writePosition + KEY\_SIZE > bytes.length) {

            requestMemory();

        }

        bytes.setUInt32(writePosition, keyHash);

        bytes.setInt32(writePosition + 4, stringName.rawLocation);

        bytes.setInt32(writePosition + 4 + POINTER\_SIZE, address.rawLocation);

        bytes.setInt32(writePosition + 4 + POINTER\_SIZE \* 2, parent.getTypeInformation(type).pointer.rawLocation);

        bytes.setUInt16(currentScopeStart + 2, bytes.getUInt16(currentScopeStart + 2) + 1); // Increment the length of the current scope.

    }

    /\*\*

        Removes a variable from the current scope. If it doesn't exist, throws an error.

        If it also exists in parent scopes, they won't be affected.

        @param key The name of the variable

    \*\*/

    public function dereference(key:String) {

        var keyHash = Murmur1.hash(ByteArray.from(key));

        var writePosition = currentScopeStart + 4;

        while (true) {

            var currentKeyHash = bytes.getUInt32(writePosition);

            if (currentKeyHash == keyHash) {

                var stringName = parent.storage.readString(bytes.getInt32(writePosition + 4));

                if (stringName == key) break;

            }

            writePosition += KEY\_SIZE;

            if (writePosition >= currentScopeStart + currentScopeLength \* KEY\_SIZE) throw 'Cannot dereference key that doesn\'t exist. (key: $key)';

        }

        bytes.setUInt16(bytes.getInt32(0) + 2, bytes.getUInt16(bytes.getInt32(0) + 2) - 1); // Decrement the length of the current scope.

    }

    /\*\*

        Looks up a variable in the current scope.

        If it doesn't exist in the current scope, it will look in parent scopes.

        Throws an error if it doesn't exist in any scope.

        @param key The name of the variable

        @return The address and type of the variable, in the form of an anonymous structure: `{address: MemoryPointer, type: String}`

    \*\*/

    public function get(key:String):{address:MemoryPointer, type:String} {

        // This one is a little more complicated, since it involves lookbehinds.

        var keyHash = Murmur1.hash(ByteArray.from(key));

        var checkingScope = currentScopeStart;

        var elementCount = bytes.getUInt16(currentScopeStart + 2);

        var nextScope = currentScopeStart - bytes.getUInt16(currentScopeStart) \* KEY\_SIZE - 4;

        do {

            var i = checkingScope + 4;

            while (i < (checkingScope + elementCount \* KEY\_SIZE)) {

                var testingHash = bytes.getUInt32(i);

                if (keyHash == testingHash) {

                    var stringName = parent.storage.readString(bytes.getInt32(i + 4));

                    if (stringName == key) {

                        return {

                            address: MemoryPointer.fromInt(bytes.getInt32(i + 4 + POINTER\_SIZE)),

                            type: parent.getTypeName(bytes.getInt32(i + 4 + POINTER\_SIZE \* 2))

                        }

                    }

                }

                i += KEY\_SIZE;

            }

            checkingScope = nextScope;

            elementCount = bytes.getUInt16(nextScope + 2);

            nextScope = nextScope - bytes.getUInt16(nextScope) \* KEY\_SIZE - 4;

        } while (checkingScope != 0);

        throw 'Key $key does not exist.';

    }

    /\*\*

        Sets the value and type of an existing variable in the current scope.

        If it doesn't exist in the current scope, it will look in parent scopes.

        Throws an error if it doesn't exist in any scope.

        @param key The name of the variable

        @param value The new value and type of the variable. If any of these are null, the previous value/type will remain.

    \*\*/

    public function set(key:String, value:{?address:MemoryPointer, ?type:String}) {

        var keyHash = Murmur1.hash(ByteArray.from(key));

        var checkingScope = currentScopeStart;

        var elementCount = bytes.getUInt16(currentScopeStart + 2);

        var nextScope = currentScopeStart - bytes.getUInt16(currentScopeStart) \* KEY\_SIZE - 4;

        do {

            var i = checkingScope + 4;

            while (i < (checkingScope + elementCount \* KEY\_SIZE)) {

                var testingHash = bytes.getUInt32(i);

                if (keyHash == testingHash) {

                    var stringName = parent.storage.readString(bytes.getInt32(i + 4));

                    if (stringName == key) {

                        if (value.address != null) bytes.setInt32(i + 4 + POINTER\_SIZE, value.address.rawLocation);

                        if (value.type != null) bytes.setInt32(i + 4 + POINTER\_SIZE \* 2, parent.getTypeInformation(value.type).pointer.rawLocation);

                        return;

                    }

                }

                i += KEY\_SIZE;

            }

            checkingScope = nextScope;

            elementCount = bytes.getUInt16(nextScope + 2);

            nextScope = nextScope - bytes.getUInt16(nextScope) \* KEY\_SIZE - 4;

        } while (checkingScope != 0);

        throw 'Cannot set $key -  does not exist.';

    }

    /\*\*

        Checks if a key exists in any scope.

        @param key The name of the variable

        @return true if the key exists, false otherwise.

    \*\*/

    public function exists(key:String):Bool {

        var keyHash = Murmur1.hash(ByteArray.from(key));

        var checkingScope = currentScopeStart;

        var elementCount = bytes.getUInt16(currentScopeStart + 2);

        var nextScope = currentScopeStart - bytes.getUInt16(currentScopeStart) \* KEY\_SIZE - 4;

        do {

            var i = checkingScope + 4;

            while (i < (checkingScope + elementCount \* KEY\_SIZE)) {

                var testingHash = bytes.getUInt32(i);

                if (keyHash == testingHash) {

                    var stringName = parent.storage.readString(bytes.getInt32(i + 4));

                    if (stringName == key) {

                        return true;

                    }

                }

                i += KEY\_SIZE;

            }

            checkingScope = nextScope;

            elementCount = bytes.getUInt16(nextScope + 2);

            nextScope = nextScope - bytes.getUInt16(nextScope) \* KEY\_SIZE - 4;

        } while (checkingScope != 0);

        return false;

    }

    /\*\*

        Returns an iterator over all key/value pairs in all scopes, starting from the current scope, and going up the parent scopes.

    \*\*/

    public function keyValueIterator():KeyValueIterator<String, {address:MemoryPointer, type:String}> {

        var map = new Map<String, {address:MemoryPointer, type:String}>();

        var checkingScope = currentScopeStart;

        var elementCount = bytes.getUInt16(currentScopeStart + 2);

        var nextScope = currentScopeStart - bytes.getUInt16(currentScopeStart) \* KEY\_SIZE - 4;

        do {

            var i = checkingScope;

            while (i < (checkingScope + elementCount \* KEY\_SIZE)) {

                var stringName = parent.storage.readString(bytes.getInt32(i + 4));

                map.set(stringName, {

                    address: MemoryPointer.fromInt(bytes.getInt32(i + 4 + POINTER\_SIZE)),

                    type: parent.storage.readString(bytes.getInt32(i + 4 + POINTER\_SIZE \* 2))

                });

                i += KEY\_SIZE;

            }

            checkingScope = nextScope;

            elementCount = bytes.getUInt16(nextScope + 2);

            nextScope = nextScope - bytes.getUInt16(nextScope) \* KEY\_SIZE - 4;

        } while (checkingScope != 0);

        return map.keyValueIterator();

    }

}

package little.interpreter.memory;

import little.tools.OrderedMap;

import little.interpreter.memory.Memory.TypeInfo;

import haxe.hash.Murmur1;

import vision.tools.MathTools;

import haxe.display.Display.Module;

import little.interpreter.memory.MemoryPointer;

import haxe.Int64;

import haxe.ds.Either;

import haxe.exceptions.ArgumentException;

import little.interpreter.Tokens.InterpTokens;

import haxe.xml.Parser;

import haxe.io.Bytes;

import haxe.io.UInt8Array;

import vision.ds.ByteArray;

using little.tools.Extensions;

class Storage {

    public var parent:Memory;

    public var reserved:ByteArray;

    public var storage:ByteArray;

    /\*\*

        Instantiates a new `Storage`

    \*\*/

    public function new(memory:Memory) {

        parent = memory;

        storage = new ByteArray(parent.memoryChunkSize);

        reserved = new ByteArray(parent.memoryChunkSize);

        reserved.fill(0, parent.memoryChunkSize, 0);

    }

    /\*\*

        Requests more memory if needed

    \*\*/

    function requestMemory() {

        if (storage.length > parent.maxMemorySize) {

            Little.runtime.throwError(ErrorMessage('Out of memory'), MEMORY\_STORAGE);

        }

        storage.resize(storage.length + parent.memoryChunkSize);

        reserved.resize(reserved.length + parent.memoryChunkSize);

    }

    /\*\*

        stores a byte to the storage

        @param b an 8-bit number

        @return A pointer to its address on the storage. The size of this "object" is `1`.

    \*\*/

    public function storeByte(b:Int):MemoryPointer {

        if (b == 0) return parent.constants.ZERO;

        #if !static if (b == null) return parent.constants.NULL; #end

        // Find a free spot

        var i = parent.constants.capacity;

        while (i < reserved.length && reserved[i] != 0) i++;

        if (i >= reserved.length) requestMemory();

        storage[i] = b;

        reserved[i] = 1;

        return i;

    }

    /\*\*

        stores a byte to the storage, at the given address. Can overwrite bytes at any address.

        @param address The address to store the byte

        @param b an 8-bit number

    \*\*/

    public function setByte(address:MemoryPointer, b:Int) {

        storage[address.rawLocation] = b;

        reserved[address.rawLocation] = 1;

    }

    /\*\*

        Reads a byte from the storage

        @param address The address of the byte to read

        @return The byte

    \*\*/

    public function readByte(address:MemoryPointer):Int {

        return storage[address.rawLocation];

    }

    /\*\*

        Frees a byte from the storage

        @param address The address of the byte to remove

    \*\*/

    public function freeByte(address:MemoryPointer) {

        storage[address.rawLocation] = 0;

        reserved[address.rawLocation] = 0;

    }

    /\*\*

        Stores an array of bytes.

        @param size The number of bytes to store

        @param b An optional array of bytes

        @return The address of the first byte

    \*\*/

    public function storeBytes(size:Int, ?b:ByteArray):MemoryPointer {

        var i = parent.constants.capacity;

        while (i < reserved.length - size && !reserved.getBytes(i, size).isEmpty()) i++;

        if (i >= reserved.length - size) {

            requestMemory();

            i += size; // Will leave some empty space, Todo.

            while (i + size > reserved.length) requestMemory();

        }

        for (j in 0...size) {

            storage[i + j] = j > b.length ? 0 : b[j];

            reserved[i + j] = 1;

        }

        return i;

    }

    /\*\*

        Sets an array of bytes at the given address. Can overwrite bytes at any address.

        @param address The address to store the bytes at

        @param bytes An array of bytes to write

    \*\*/

    public function setBytes(address:MemoryPointer, bytes:ByteArray) {

        for (j in 0...bytes.length) {

            storage[address.rawLocation + j] = bytes[j];

            reserved[address.rawLocation + j] = 1;

        }

    }

    /\*\*

        Reads an array of bytes from the storage.

        @param address The address of the first byte

        @param size The number of bytes to read

        @return The resulting byte array

    \*\*/

    public function readBytes(address:MemoryPointer, size:Int):ByteArray {

        if (address == parent.constants.NULL) return null;

        var bytes = new ByteArray(size);

        for (j in 0...size) {

            bytes[j] = storage[address.rawLocation + j];

        }

        return bytes;

    }

    /\*\*

        Frees an array of bytes from the storage, starting at the given address

        @param address The address of the first byte

        @param size The number of bytes to remove

    \*\*/

    public function freeBytes(address:MemoryPointer, size:Int) {

        for (j in 0...size) {

            storage[address.rawLocation + j] = 0;

            reserved[address.rawLocation + j] = 0;

        }

    }

    /\*\*

        Stores an array of elements of a specific size.

        @param length The number of elements

        @param elementSize The size of each element

        @param defaultElement An optional default element to store at each index

        @return The address of the array.

    \*\*/

    public function storeArray(length:Int, elementSize:Int, ?defaultElement:ByteArray):MemoryPointer {

        var size = elementSize \* length;

        var bytes = new ByteArray(size + 4 + 4); // First 4 bytes are the length of the array, the other 4 bytes are for element size.

        if (defaultElement != null) {

            for (i in 0...length) {

                bytes.setBytes(i + 4, defaultElement);

            }

        }

        bytes.setInt32(0, length);

        bytes.setInt32(4, elementSize);

        return storeBytes(bytes.length, bytes);

    }

    /\*\*

        Sets an array of elements of a specific size. Can overwrite bytes at any address.

        @param address The address of the array

        @param length The number of elements

        @param elementSize The size of each element

        @param defaultElement An optional default element to store at each index

    \*\*/

    public function setArray(address:MemoryPointer, length:Int, elementSize:Int, ?defaultElement:ByteArray) {

        var size = elementSize \* length;

        var bytes = new ByteArray(size + 4 + 4); // First 4 bytes are the length of the array, next 4 bytes are for element size.

        if (defaultElement != null) {

            for (i in 0...length) {

                bytes.setBytes(i + 8, defaultElement);

            }

        }

        bytes.setInt32(0, length);

        bytes.setInt32(4, elementSize);

        setBytes(address, bytes);

    }

    /\*\*

        Reads an array of elements of a specific size.

        @param address The address of the array

        @return An array of elements, each element being an array of bytes.

    \*\*/

    public function readArray(address:MemoryPointer):Array<ByteArray> {

        if (address == parent.constants.NULL) return null;

        var length = readInt32(address);

        var elementSize = readInt32(address.rawLocation + 4);

        address.rawLocation += 8;

        var array = [];

        for (i in 0...length) {

            array.push(readBytes(address, elementSize));

            address.rawLocation += elementSize;

        }

        return array;

    }

    /\*\*

        Frees an array of elements.

        @param address The address of the array

    \*\*/

    public function freeArray(address:MemoryPointer) {

        var length = readInt32(address);

        var elementSize = readInt32(address.rawLocation + 4);

        freeBytes(address, length \* elementSize + 8);

    }

    /\*\*

        Stores a 16-bit integer.

        @param b The 16-bit integer

        @return The address of the integer

    \*\*/

    public function storeInt16(b:Int):MemoryPointer {

        if (b == 0) return parent.constants.ZERO;

        #if !static if (b == null) return parent.constants.NULL; #end

        // Find a free spot

        var i = parent.constants.capacity;

        while (i < reserved.length - 1 && reserved[i] != 0 && reserved[i + 1] != 0) i++;

        if (i >= reserved.length - 1) {

            requestMemory();

            i += 2; // leaves empty byte Todo.

        }

        for (j in 0...1) {

            storage[i + j] = b & 0xFF;

            b = b >> 8;

            reserved[i + j] = 1;

        }

        return i;

    }

    /\*\*

        Sets a 16-bit integer. Can overwrite bytes at any address.

        @param address The address of the integer

        @param b The 16-bit integer

    \*\*/

    public function setInt16(address:MemoryPointer, b:Int) {

        storage[address.rawLocation] = b & 0xFF;

        storage[address.rawLocation + 1] = (b >> 8) & 0xFF;

        reserved[address.rawLocation] = 1;

        reserved[address.rawLocation + 1] = 1;

    }

    /\*\*

        Reads a 16-bit integer.

        @param address The address of the integer

        @return The 16-bit integer as a 32 bit haxe integer

    \*\*/

    public function readInt16(address:MemoryPointer):Int {

        if (address == parent.constants.NULL) return #if static 0 #else null #end;

        // Dont forget to make the number negative if needed.

        return (storage[address.rawLocation] + (storage[address.rawLocation + 1] << 8)) - 32767;

    }

    /\*\*

        Frees a 16-bit integer.

        @param address The address of the integer

    \*\*/

    public function freeInt16(address:MemoryPointer) {

        storage[address.rawLocation] = 0;

        storage[address.rawLocation + 1] = 0;

        reserved[address.rawLocation] = 0;

        reserved[address.rawLocation + 1] = 0;

    }

    /\*\*

        Stores an unsigned 16-bit integer.

        @param b  The unsigned 16-bit integer as a 32 bit haxe integer

        @return The address of the integer

    \*\*/

    public function storeUInt16(b:Int):MemoryPointer {

        return storeInt16(b < 0 ? b + 32767 : b);

    }

    /\*\*

        Sets an unsigned 16-bit integer. Can overwrite bytes at any address.

        @param address The address of the integer

        @param b The unsigned 16-bit integer

    \*\*/

    public function setUInt16(address:MemoryPointer, b:Int) {

        setInt16(address, b < 0 ? b + 32767 : b);

    }

    /\*\*

        Reads an unsigned 16-bit integer.

        @param address The address of the integer

    \*\*/

    public function readUInt16(address:MemoryPointer) {

        if (address == parent.constants.NULL) return null;

        return (storage[address.rawLocation] + (storage[address.rawLocation + 1] << 8));

    }

    /\*\*

        Frees an unsigned 16-bit integer.

        @param address The address of the integer

    \*\*/

    public function freeUInt16(address:MemoryPointer) {

        freeInt16(address);

    }

    /\*\*

        Stores an 32-bit integer.

        @param b The 32-bit integer

        @return The address of the integer

    \*\*/

    public function storeInt32(b:Int):MemoryPointer {

        if (b == 0) return parent.constants.ZERO;

        #if !static if (b == null) return parent.constants.NULL; #end

        // Find a free spot

        var i = parent.constants.capacity;

        while (i < reserved.length - 3 && reserved[i] + reserved[i + 1] + reserved[i + 2] + reserved[i + 3] != 0) i++;

        if (i >= reserved.length - 3) {

            requestMemory();

            i += 4; // leaves empty bytes Todo.

        }

        for (j in 0...4) {

            storage[i + j] = b & 0xFF;

            b = b >> 8;

            reserved[i + j] = 1;

        }

        return i;

    }

    /\*\*

        Sets an 32-bit integer. Can overwrite bytes at any address.

        @param address The address of the integer

        @param b The 32-bit integer

    \*\*/

    public function setInt32(address:MemoryPointer, b:Int) {

        storage[address.rawLocation] = b & 0xFF;

        storage[address.rawLocation + 1] = (b >> 8) & 0xFF;

        storage[address.rawLocation + 2] = (b >> 16) & 0xFF;

        storage[address.rawLocation + 3] = (b >> 24) & 0xFF;

        reserved[address.rawLocation] = 1;

        reserved[address.rawLocation + 1] = 1;

        reserved[address.rawLocation + 2] = 1;

        reserved[address.rawLocation + 3] = 1;

    }

    /\*\*

        Reads an 32-bit integer.

        @param address The address of the integer

        @return The 32-bit integer

    \*\*/

    public function readInt32(address:MemoryPointer):Int {

        if (address == parent.constants.NULL) return #if static 0 #else null #end;

        return (storage[address.rawLocation] + (storage[address.rawLocation + 1] << 8) + (storage[address.rawLocation + 2] << 16) + (storage[address.rawLocation + 3] << 24));

    }

    /\*\*

        Frees an 32-bit integer.

        @param address The address of the integer

    \*\*/

    public function freeInt32(address:MemoryPointer) {

        for (j in 0...4) {

            storage[address.rawLocation + j] = 0;

            reserved[address.rawLocation + j] = 0;

        }

    }

    /\*\*

        Stores an unsigned 32-bit integer.

        @param b The unsigned 32-bit integer

        @return The address of the integer

    \*\*/

    public function storeUInt32(b:UInt):MemoryPointer {

        return storeInt32(b);

    }

    /\*\*

        Sets an unsigned 32-bit integer. Can overwrite bytes at any address.

        @param address The address of the integer

        @param b The unsigned 32-bit integer

    \*\*/

    public function setUInt32(address:MemoryPointer, b:UInt) {

        setInt32(address, b);

    }

    /\*\*

        Reads an unsigned 32-bit integer.

        @param address The address of the integer

        @return The unsigned 32-bit integer

    \*\*/

    public function readUInt32(address:MemoryPointer):UInt {

        return readInt32(address);

    }

    /\*\*

        Frees an unsigned 32-bit integer.

        @param address The address of the integer

    \*\*/

    public function freeUInt32(address:MemoryPointer) {

        freeInt32(address);

    }

    /\*\*

        Stores a 64-bit floating point number.

        @param b The 64-bit floating point number

        @return The address of the floating point number

    \*\*/

    public function storeDouble(b:Float):MemoryPointer {

        if (b == 0) return parent.constants.ZERO;

        #if !static if (b == null) return parent.constants.NULL; #end

        var i = parent.constants.capacity;

        while (i < reserved.length - 7 &&

            reserved[i] + reserved[i + 1] + reserved[i + 2] + reserved[i + 3] +

            reserved[i + 4] + reserved[i + 5] + reserved[i + 6] + reserved[i + 7] != 0) i++;

        if (i >= reserved.length - 7) {

            requestMemory();

            i += 8; // leaves empty bytes Todo.

        }

        var bytes = Bytes.alloc(8);

        bytes.setDouble(0, b);

        for (j in 0...8) {

            storage[i + j] = bytes.get(j);

            reserved[i + j] = 1;

        }

        return i;

    }

    /\*\*

        Sets a 64-bit floating point number. Can overwrite bytes at any address.

        @param address The address of the floating point number

        @param b The 64-bit floating point number

    \*\*/

    public function setDouble(address:MemoryPointer, b:Float) {

        storage.setDouble(address.rawLocation, b);

        for (j in 0...8) {

            reserved[address.rawLocation + j] = 1;

        }

    }

    /\*\*

        Reads a 64-bit floating point number.

        @param address The address of the floating point number

        @return The 64-bit floating point number

    \*\*/

    public function readDouble(address:MemoryPointer):Float {

        if (address == parent.constants.NULL) return #if static 0.0 #else null #end;

        return storage.getDouble(address.rawLocation);

    }

    /\*\*

        Frees a 64-bit floating point number.

        @param address The address of the floating point number

    \*\*/

    public function freeDouble(address:MemoryPointer) {

        for (j in 0...8) {

            storage[address.rawLocation + j] = 0;

            reserved[address.rawLocation + j] = 0;

        }

    }

    /\*\*

        Stores a memory pointer.

        @param p The memory pointer to store

        @return the address at which the pointer is stored

    \*\*/

    public function storePointer(p:MemoryPointer):MemoryPointer {

        return storeInt32(p.rawLocation); // Currently, only 32-bit pointers are supported because of the memory buffer

    }

    /\*\*

        Sets a memory pointer. Can overwrite bytes at any address.

        @param address The address of the memory pointer

        @param p The memory pointer

    \*\*/

    public function setPointer(address:MemoryPointer, p:MemoryPointer) {

        setInt32(address, p.rawLocation);

    }

    /\*\*

        Reads a memory pointer.

        @param address The address of the memory pointer

        @return The memory pointer

    \*\*/

    public function readPointer(address:MemoryPointer):MemoryPointer {

        return readInt32(address);

    }

    /\*\*

        Frees a memory pointer.

        @param address The address of the memory pointer

    \*\*/

    public function freePointer(address:MemoryPointer) {

        freeInt32(address);

    }

    /\*\*

        Stores a string.

        @param b The string

        @return The address at which the string is stored

    \*\*/

    public function storeString(b:String):MemoryPointer {

        if (b == "") return parent.constants.EMPTY\_STRING;

        if (b == null) return parent.constants.NULL;

        // Convert the string into bytes

        var stringBytes = Bytes.ofString(b, UTF8);

        // In order to accurately keep track of the string, the first 4 bytes will be used to store the length \*of the bytes\*

        var bytes = ByteArray.from(stringBytes.length).concat(stringBytes);

        // Find a free spot. Keep in mind that string's characters in this context are UTF-8 encoded, so each character is 1 byte

        var i = parent.constants.capacity;

        while (i < reserved.length - bytes.length && !reserved.getBytes(i, bytes.length).isEmpty()) i++;

        if (i >= reserved.length - bytes.length) {

            requestMemory();

            i += bytes.length; // leaves empty bytes, intentional.

            while (i + bytes.length > reserved.length) requestMemory();

        }

        // Each character in this string should be UTF-8 encoded

        storage.setBytes(i, bytes);

        reserved.setBytes(i, new ByteArray(bytes.length, 1));

        return i;

    }

    /\*\*

        Sets a string. Can overwrite bytes at any address.

        @param address The address of the string

        @param b The string

    \*\*/

    public function setString(address:MemoryPointer, b:String) {

        // Gets a bit tricky, we need to change the string length too

        var stringBytes = Bytes.ofString(b, UTF8);

        var bytes = ByteArray.from(stringBytes.length).concat(stringBytes);

        for (j in 0...bytes.length) {

            storage[address.rawLocation + j] = bytes.get(j);

            reserved[address.rawLocation + j] = 1;

        }

    }

    /\*\*

        Reads a string.

        @param address The address of the string

        @return The string

    \*\*/

    public function readString(address:MemoryPointer):String {

        if (address == parent.constants.NULL) return null;

        var length = readInt32(address.rawLocation);

        return storage.getString(address.rawLocation + 4, length, UTF8);

    }

    /\*\*

        Frees a string.

        @param address The address of the string

    \*\*/

    public function freeString(address:MemoryPointer) {

        var len = storage.getInt32(address.rawLocation) + 4;

        for (j in 0...len) {

            storage[address.rawLocation + j] = 0;

            reserved[address.rawLocation + j] = 0;

        }

    }

    /\*\*

        Stores a code block, which represents function code.

        @param caller The code block

        @return The address at which the code block is stored

    \*\*/

    public function storeCodeBlock(caller:InterpTokens):MemoryPointer {

        switch caller {

            case Block(body, \_): return storeString(ByteCode.compile(FunctionCode(new OrderedMap(), caller)));

            case FunctionCode(requiredParams, body): return storeString(ByteCode.compile(caller));

            case \_: throw new ArgumentException("caller", '${caller} must be a code block');

        }

    }

    /\*\*

        Sets a code block, which represents function code. Can overwrite bytes at any address.

        @param address The address of the code block

        @param caller The code block

    \*\*/

    public function setCodeBlock(address:MemoryPointer, caller:InterpTokens) {

        switch caller {

            case Block(body, \_):

                setString(address, ByteCode.compile(FunctionCode(new OrderedMap(), caller)));

            case FunctionCode(requiredParams, body):

                setString(address, ByteCode.compile(caller));

            case \_: throw new ArgumentException("caller", '${caller} must be a code block');

        }

    }

    /\*\*

        Reads a code block, which represents function code.

        @param address The address of the code block

        @return The code block

    \*\*/

    public function readCodeBlock(address:MemoryPointer):InterpTokens {

        return ByteCode.decompile(readString(address.rawLocation))[0];

    }

    /\*\*

        Frees a function value.

        @param address The address of the function value

    \*\*/

    public function freeCodeBlock(address:MemoryPointer) {

        freeString(address);

    }

    /\*\*

        Stores a condition code, which represents a condition.

        @param caller The condition

        @return The address at which the condition is stored

    \*\*/

    public function storeCondition(caller:InterpTokens):MemoryPointer {

        switch caller {

            case ConditionCode(\_): return storeString(ByteCode.compile(caller));

            case \_: throw new ArgumentException("caller", '${caller} must be a token of type ${ConditionCode(null).getName()}');

        }

    }

    /\*\*

        Sets a condition code, which represents a condition. Can overwrite bytes at any address.

        @param address The address of the condition

        @param caller The condition

    \*\*/

    public function setCondition(address:MemoryPointer, caller:InterpTokens) {

        switch caller {

            case ConditionCode(\_):

                setString(address, ByteCode.compile(caller));

            case \_: throw new ArgumentException("caller", '${caller} must be a token of type ${ConditionCode(null).getName()}');

        }

    }

    /\*\*

        Reads a `ConditionCode`, which represents a condition.

        @param address The address of the condition

        @return The condition

    \*\*/

    public function readCondition(address:MemoryPointer):InterpTokens {

        return ByteCode.decompile(readString(address.rawLocation))[0];

    }

    /\*\*

        Frees a condition code.

        @param address The address of the condition

    \*\*/

    public function freeCondition(address:MemoryPointer) {

        freeString(address);

    }

    /\*\*

        Stores an operator.

        @param sign The operator

    \*\*/

    public function storeSign(sign:String) {

        return storeString(sign);

    }

    /\*\*

        Sets an operator. Can overwrite bytes at any address.

        @param address The address of the operator

        @param sign The operator

    \*\*/

    public function setSign(address:MemoryPointer, sign:String) {

        setString(address, sign);

    }

    /\*\*

        Reads an operator.

        @param address The address of the operator

        @return The operator

    \*\*/

    public function readSign(address:MemoryPointer):InterpTokens {

        if (address == parent.constants.NULL) return null;

        return Sign(readString(address));

    }

    /\*\*

        Frees an operator.

        @param address The address of the operator

    \*\*/

    public function freeSign(address:MemoryPointer) {

        freeString(address);

    }

    /\*\*

        A helper function that stores tokens of a static length in memory, and strings.

        @param token The token

        @return The address at which the token is stored

    \*\*/

    public function storeStatic(token:InterpTokens):MemoryPointer {

        switch token {

            case NullValue | TrueValue | FalseValue: return parent.constants.get(token);

            case Number(num): return storeInt32(num);

            case Decimal(num): return storeDouble(num);

            case Characters(string): return storeString(string);

            case Sign(sign): return storeSign(sign);

            case ClassPointer(pointer): return pointer;

            case \_: throw new ArgumentException("token", '${token} cannot be statically stored to the storage');

        }

    }

    /\*\*

        Stores an object using 3 parts:

        - POINTER\_SIZE + 4 Bytes directly at the site of storage, including:

          - Byte 0 to 4: The length of the object's hashtable

          - Byte 4 to POINTER\_SIZE + 4: A pointer to the object's hashtable

        - The object's hashtable, at another location. The hashtable can be moved around, and when this

          is done, the data at the object's address changes.

    \*\*/

    public function storeObject(object:InterpTokens):MemoryPointer {

        if (object.is(NULL\_VALUE)) return parent.constants.NULL;

        if (!object.is(OBJECT)) throw new ArgumentException("object", '${object} is not a dynamic object');

        /\*

            We will do the same thing that python does, but simpler.

            Simply put, store everything in a hash-table that contains all object properties.

            that hash table will be stored as a pointer, for easy replacement when needed.

        \*/

        switch object {

            case Object(props, typeName): {

                var quintuples = new Array<{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer}>();

                var propsC = props.copy();

                propsC[Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME] = {

                    value: Characters(typeName),

                    documentation: 'The type of this object, as a ${Little.keywords.TYPE\_STRING}.',

                }

                for (k => v in propsC) {

                    var key = k;

                    var keyPointer = storeString(key);

                    var value = switch v.value {

                        case Object(\_, \_): storeObject(v.value);

                        case FunctionCode(\_, \_): storeCodeBlock(v.value);

                        case \_: storeStatic(v.value);

                    }

                    var type = switch v.value {

                        case Number(\_): parent.getTypeInformation(Little.keywords.TYPE\_INT).pointer;

                        case Decimal(\_): parent.getTypeInformation(Little.keywords.TYPE\_FLOAT).pointer;

                        case Characters(\_): parent.getTypeInformation(Little.keywords.TYPE\_STRING).pointer;

                        case TrueValue | FalseValue: parent.getTypeInformation(Little.keywords.TYPE\_BOOLEAN).pointer;

                        case NullValue: parent.getTypeInformation(Little.keywords.TYPE\_DYNAMIC).pointer;

                        case FunctionCode(\_, \_): parent.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer;

                        case ClassPointer(\_): parent.getTypeInformation(Little.keywords.TYPE\_MODULE).pointer;

                        case Object(\_, t): parent.getTypeInformation(t).pointer;

                        case \_: throw "Property value must be a static value, a code block or an object (given: `" + v + "`)";

                    }

                    quintuples.push({key: key, keyPointer: keyPointer, value: value, type: type, doc: storeString(v.documentation) /\*Todo, not a good solution, docs will some out of classes most of the time.\*/});

                }

                var bytes = HashTables.generateObjectHashTable(quintuples);

                var bytesLength = ByteArray.from(bytes.length);

                var bytesPointer = storeBytes(bytes.length, bytes);

                return storeBytes(4 + POINTER\_SIZE , ByteArray.from(bytes.length).concat(ByteArray.from(bytesPointer.rawLocation)));

            }

            case \_:

                throw new ArgumentException("object", '${object} must be an `Interpreter.Object`');

        }

        throw "How did you get here?";

    }

    /\*\*

        Stores an object at a specific address using 3 parts:

        - POINTER\_SIZE + 4 Bytes directly at the site of storage, including:

          - Byte 0 to 4: The length of the object's hashtable

          - Byte 4 to POINTER\_SIZE + 4: A pointer to the object's hashtable

        - The object's hashtable, at another location. The hashtable can be moved around, and when this

          is done, the data at the object's address changes.

    \*\*/

    public function setObject(address:MemoryPointer, object:InterpTokens) {

        if (object.is(NULL\_VALUE)) return parent.constants.NULL;

        if (!object.is(OBJECT)) throw new ArgumentException("object", '${object} is not a dynamic object');

        switch object {

            case Object(props, typeName): {

                var quintuples = new Array<{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer}>();

                var propsC = props.copy();

                propsC[Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME] = {

                    value: Characters(typeName),

                    documentation: 'The type of this object, as a ${Little.keywords.TYPE\_STRING}.',

                }

                for (k => v in propsC) {

                    var key = k;

                    var keyPointer = storeString(key);

                    var value = switch v.value {

                        case Object(\_, \_): storeObject(v.value);

                        case FunctionCode(\_, \_): storeCodeBlock(v.value);

                        case \_: storeStatic(v.value);

                    }

                    var type = switch v.value {

                        case Number(\_): parent.getTypeInformation(Little.keywords.TYPE\_INT).pointer;

                        case Decimal(\_): parent.getTypeInformation(Little.keywords.TYPE\_FLOAT).pointer;

                        case Characters(\_): parent.getTypeInformation(Little.keywords.TYPE\_STRING).pointer;

                        case TrueValue | FalseValue: parent.getTypeInformation(Little.keywords.TYPE\_BOOLEAN).pointer;

                        case NullValue: parent.getTypeInformation(Little.keywords.TYPE\_DYNAMIC).pointer;

                        case FunctionCode(\_, \_): parent.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer;

                        case ClassPointer(\_): parent.getTypeInformation(Little.keywords.TYPE\_MODULE).pointer;

                        case Object(\_, t): parent.getTypeInformation(t).pointer;

                        case \_: throw "Property value must be a static value, a code block or an object (given: `" + v + "`)";

                    }

                    quintuples.push({key: key, keyPointer: keyPointer, value: value, type: type, doc: storeString(v.documentation) /\*Todo, not a good solution, docs will some out of classes most of the time.\*/});

                }

                var bytes = HashTables.generateObjectHashTable(quintuples);

                var bytesLength = ByteArray.from(bytes.length);

                var bytesPointer = storeBytes(bytes.length, bytes);

                setBytes(address, ByteArray.from(bytes.length).concat(ByteArray.from(bytesPointer.rawLocation)));

            }

            case \_:

                throw new ArgumentException("object", '${object} must be an `Interpreter.Object`');

        }

        throw "How did you get here?";

    }

    /\*\*

        Reads an object.

        @param pointer The address of the object

        @return A token representing the object (of type `InterpTokens.Object`)

    \*\*/

    public function readObject(pointer:MemoryPointer):InterpTokens {

        if (pointer == parent.constants.NULL) return null;

        if (parent.constants.hasPointer(readPointer(pointer.rawLocation + 4))) throw "HashTable pointer is not valid";

        var hashTableBytes = readBytes(readPointer(pointer.rawLocation + 4), readInt32(pointer));

        var table = HashTables.readObjectHashTable(hashTableBytes, this);

        var map = new Map<String, {value:InterpTokens, documentation:String}>();

        for (entry in table) {

            map[entry.key] =

            map[entry.key] = {

                value: @:privateAccess parent.valueFromType(entry.value, parent.getTypeName(entry.type), ["<object>"]),

                documentation: readString(entry.doc)

            }

        }

        return Object(

            map,

            map[Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME].value.parameter(0) /\* This value is a `Characters`, so it first param is a `String`.\*/

        );

    }

    /\*\*

        Free an object

        @param pointer The address of the object

    \*\*/

    public function freeObject(pointer:MemoryPointer) {

        // Just free pointer size (4) + int32 size (4)

        var hashTableSize = readInt32(pointer);

        var hashTablePointer = readPointer(pointer.rawLocation + 4);

        freeBytes(hashTablePointer, hashTableSize);

        freeBytes(pointer, POINTER\_SIZE + 4);

    }

    /\*\*

        stores a type, and all its statics and instance fields.

        @param name The name of the type

        @param statics It's static fields

        @param instances It's instance fields

    \*\*/

    public function storeType(name:String, statics:Map<String, {value:InterpTokens, documentation:String, type:String}>, instances:Map<String, {documentation:String, type:String}>) {

        var bytes = ByteArray.from(storeString(name).rawLocation);

        var cellSize = POINTER\_SIZE \* 4;

        // We'll create each map with some extra space to avoid frequent collisions. more overhead? yes. faster? also probably yes.

        cellSize = POINTER\_SIZE \* 4;

        var staticsLength = statics.keys().toArray().length;

        var staticHashMap = new ByteArray(MathTools.floor(staticsLength \* cellSize /\*field name, value, type, doc\*/ \* 3 / 2));

        var instancesLength = instances.keys().toArray().length;

        var instancesHashMap = new ByteArray(MathTools.floor(instancesLength \* (cellSize - POINTER\_SIZE) /\*field name, type, doc\*/ \* 3 / 2));

        for (\_\_item in [{a: staticsLength, b: staticHashMap, c:statics }, {a: instancesLength, b: instancesHashMap, c:instances}]) {

            var elements = \_\_item.a;

            var hashTable = \_\_item.b;

            var fields:Map<String, Dynamic> = \_\_item.c;

            for (k => v in fields) {

                var keyHash = Murmur1.hash(Bytes.ofString(k));

                // Make sure divisibility by cellSize is possible. see HashTables.generateObjectHashTable for more info

                var khI64 = Int64.make(0, keyHash);

                var keyIndex = ((khI64 \* cellSize) % elements).low;

                if (hashTable.getInt32(keyIndex) == 0) {

                    var address = keyIndex;

                    hashTable.setInt32(address, storeString(k).rawLocation);

                    address += POINTER\_SIZE;

                    if (fields == statics) {

                        hashTable.setInt32(address, parent.store(v.value).rawLocation);

                        address += POINTER\_SIZE;

                    }

                    hashTable.setInt32(address, parent.getTypeInformation(v.type).pointer.rawLocation);

                    address += POINTER\_SIZE;

                    hashTable.setInt32(keyIndex, storeString(v.documentation).rawLocation);

                } else {

                    var incrementation = 0;

                    var i = keyIndex;

                    while (hashTable.getInt32(i) != 0) {

                        i += cellSize;

                        incrementation += cellSize;

                        if (i >= hashTable.length) {

                            i = 0;

                        }

                        if (incrementation >= hashTable.length) {

                            throw 'Object hash table did not generate. This should never happen. Initial length may be incorrect.';

                        }

                    }

                    var address = keyIndex;

                    hashTable.setInt32(address, storeString(k).rawLocation);

                    address += POINTER\_SIZE;

                    if (fields == statics) {

                        hashTable.setInt32(address, parent.store(v.value).rawLocation);

                        address += POINTER\_SIZE;

                    }

                    hashTable.setInt32(address, parent.getTypeInformation(v.type).pointer.rawLocation);

                    address += POINTER\_SIZE;

                    hashTable.setInt32(keyIndex, storeString(v.documentation).rawLocation);

                }

            }

            cellSize -= POINTER\_SIZE;

        }

        staticHashMap = ByteArray.from(staticHashMap.length).concat(staticHashMap);

        instancesHashMap = ByteArray.from(instancesHashMap.length).concat(instancesHashMap);

        bytes = bytes.concat(staticHashMap).concat(instancesHashMap);

        return storeBytes(bytes.length, bytes);

    }

    /\*\*

        stores a type, and all its statics and instance fields at a specific address.

        @param address The address of the type

        @param name The name of the type

        @param statics It's static fields

        @param instances It's instance fields

    \*\*/

    public function setType(address:MemoryPointer, name:String, statics:Map<String, {value:InterpTokens, documentation:String, type:String}>, instances:Map<String, {documentation:String, type:String}>) {

        var bytes = ByteArray.from(storeString(name).rawLocation);

        var cellSize = POINTER\_SIZE \* 4;

        // We'll create each map with some extra space to avoid frequent collisions. more overhead? yes. faster? also probably yes.

        cellSize = POINTER\_SIZE \* 4;

        var staticsLength = statics.keys().toArray().length;

        var staticHashMap = new ByteArray(MathTools.floor(staticsLength \* cellSize /\*field name, value, type, doc\*/ \* 3 / 2));

        var instancesLength = instances.keys().toArray().length;

        var instancesHashMap = new ByteArray(MathTools.floor(instancesLength \* (cellSize - POINTER\_SIZE) /\*field name, type, doc\*/ \* 3 / 2));

        for (\_\_item in [{a: staticsLength, b: staticHashMap, c:statics }, {a: instancesLength, b: instancesHashMap, c:instances}]) {

            var elements = \_\_item.a;

            var hashTable = \_\_item.b;

            var fields:Map<String, Dynamic> = \_\_item.c;

            for (k => v in fields) {

                var keyHash = Murmur1.hash(Bytes.ofString(k));

                // Make sure divisibility by cellSize is possible. see HashTables.generateObjectHashTable for more info

                var khI64 = Int64.make(0, keyHash);

                var keyIndex = ((khI64 \* cellSize) % elements).low;

                if (hashTable.getInt32(keyIndex) == 0) {

                    var address = keyIndex;

                    hashTable.setInt32(address, storeString(k).rawLocation);

                    address += POINTER\_SIZE;

                    if (fields == statics) {

                        hashTable.setInt32(address, parent.store(v.value).rawLocation);

                        address += POINTER\_SIZE;

                    }

                    hashTable.setInt32(address, parent.getTypeInformation(v.type).pointer.rawLocation);

                    address += POINTER\_SIZE;

                    hashTable.setInt32(keyIndex, storeString(v.documentation).rawLocation);

                } else {

                    var incrementation = 0;

                    var i = keyIndex;

                    while (hashTable.getInt32(i) != 0) {

                        i += cellSize;

                        incrementation += cellSize;

                        if (i >= hashTable.length) {

                            i = 0;

                        }

                        if (incrementation >= hashTable.length) {

                            throw 'Object hash table did not generate. This should never happen. Initial length may be incorrect.';

                        }

                    }

                    var address = keyIndex;

                    hashTable.setInt32(address, storeString(k).rawLocation);

                    address += POINTER\_SIZE;

                    if (fields == statics) {

                        hashTable.setInt32(address, parent.store(v.value).rawLocation);

                        address += POINTER\_SIZE;

                    }

                    hashTable.setInt32(address, parent.getTypeInformation(v.type).pointer.rawLocation);

                    address += POINTER\_SIZE;

                    hashTable.setInt32(keyIndex, storeString(v.documentation).rawLocation);

                }

            }

            cellSize -= POINTER\_SIZE;

        }

        staticHashMap = ByteArray.from(staticHashMap.length).concat(staticHashMap);

        instancesHashMap = ByteArray.from(instancesHashMap.length).concat(instancesHashMap);

        bytes = bytes.concat(staticHashMap).concat(instancesHashMap);

        return setBytes(address, bytes);

    }

    /\*\*

        Reads a type.

        @param pointer The address of the type

        @return A runtime type information object

    \*\*/

    public function readType(pointer:MemoryPointer):TypeInfo {

        if (pointer == parent.constants.NULL) return null;

        var className = readString(readPointer(pointer.rawLocation));

        var cellSize = POINTER\_SIZE \* 4;

        // Statics:

        var statics:Map<String, {value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer}> = [];

        var staticsLength = readInt32(pointer.rawLocation + POINTER\_SIZE);

        var i = pointer.rawLocation + POINTER\_SIZE;

        while (i < pointer.rawLocation + POINTER\_SIZE + staticsLength) {

            var keyPointer = MemoryPointer.fromInt(readInt32(i));

            var value = MemoryPointer.fromInt(readInt32(i + POINTER\_SIZE));

            var type = MemoryPointer.fromInt(readInt32(i + POINTER\_SIZE \* 2));

            var doc = MemoryPointer.fromInt(readInt32(i + POINTER\_SIZE \* 3));

            if (keyPointer.rawLocation == 0) {

                i += cellSize;

                continue; // Nothing to do here

            }

            statics[readString(keyPointer)] = {

                value: value,

                type: type,

                doc: doc

            }

            i += cellSize;

        }

        cellSize -= POINTER\_SIZE;

        // Instances:

        var instances:Map<String, {type:MemoryPointer, doc:MemoryPointer}> = [];

        var instancesLength = readInt32(i + POINTER\_SIZE);

        while (i < i + POINTER\_SIZE + instancesLength) {

            var keyPointer = readPointer(i);

            var type = readPointer(i + POINTER\_SIZE);

            var doc = readPointer(i + POINTER\_SIZE \* 2);

            if (keyPointer.rawLocation == 0) {

                i += cellSize;

                continue; // Nothing to do here

            }

            instances[readString(keyPointer)] = {

                type: type,

                doc: doc

            }

            i += cellSize;

        }

        return {

            typeName: className,

            pointer: pointer,

            passedByReference: true, // Final decision: static types cannot be created at runtime, only externally

            isExternal: false,

            instanceFields: instances,

            staticFields: statics,

            defaultInstanceSize: 4 + POINTER\_SIZE, // Objects take 8 bytes in-place

        }

    }

    /\*\*

        Frees a type.

        @param pointer The address of the type

    \*\*/

    public function freeType(pointer:MemoryPointer) {

        freeString(pointer);

        var byteCount = readInt32(pointer.rawLocation + POINTER\_SIZE);

        byteCount += readInt32(pointer.rawLocation + POINTER\_SIZE + byteCount);

        byteCount += 4 + 4;

        freeBytes(pointer, byteCount);

    }

}

package little.tools;

import little.interpreter.Tokens.InterpTokens;

import Type.ValueType;

import little.interpreter.Interpreter;

using little.tools.TextTools;

using little.tools.Extensions;

using Std;

/\*\*

    A class containing various functions to statically interface between Haxe and Little

    values/types.

\*\*/

class Conversion {

    /\*\*

        Converts a `ValueType` instance into a string, containing the type it represents.

    \*\*/

    public static function extractHaxeType(type:ValueType):String {

        return switch type {

            case TNull: "Dynamic";

            case TInt: "Int";

            case TFloat: "Float";

            case TBool: "Bool";

            case TObject: "Dynamic";

            case TFunction: "Dynamic"; // Todo: Change this?

            case TClass(c): Type.getClassName(c).split(".").pop(); // Todo: Should I remove the path or nah?

            case TEnum(e): e.getName().split(".").pop(); // Todo: Should I remove the path or nah?

            case TUnknown: "Dynamic";

        }

    }

    /\*\*

        Converts dynamic haxe values into `Little` tokens, specifically `InterpTokens`.

        Only values are supported (no functions).

        Classes and enums are yet to be implemented.

    \*\*/

    public static function toLittleValue(val:Dynamic):InterpTokens {

        if (val == null) return NullValue;

        if (val is String) return Characters(val);

        var type = Type.typeof(val);

        return switch type {

            case TNull: NullValue;

            case TInt: Number(val);

            case TFloat: Decimal(val);

            case TBool: if (val) TrueValue else FalseValue;

            case TObject if (Type.getClass(val) != null): {

                var map:Map<String, {documentation:String, value:InterpTokens}> = new Map();

                for (field in Type.getInstanceFields(Type.getClass(val))) {

                    map[field] = {

                        value: toLittleValue(Reflect.getProperty(val, field)),

                        documentation: ""

                    }

                }

                map[Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX + Little.keywords.TYPE\_STRING] = {

                    value: Block(

                        [FunctionReturn(Characters(Std.string(val)), Identifier(Little.keywords.TYPE\_STRING))], Identifier(Little.keywords.TYPE\_STRING)),

                    documentation: "The function that will be used to convert this object to a string."

                }

                map[Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME] = {

                    value: Characters(toLittleType(Type.getClassName(val))),

                    documentation: 'The type of this object, as a ${Little.keywords.TYPE\_STRING}.'

                }

                Object(map, map[Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME].value.parameter(0));

            }

            case TObject: {

                var objType = Little.keywords.TYPE\_DYNAMIC;

                var map:Map<String, {documentation:String, value:InterpTokens}> = new Map();

                for (field in Type.getInstanceFields(Type.getClass(val))) {

                    map[field] = {

                        value: toLittleValue(Reflect.getProperty(val, field)),

                        documentation: ""

                    }

                }

                Object(map, objType);

            }

            case TFunction: {

                NullValue; // Intended Behavior

            }

            case TClass(c): {

                NullValue; // Intended Behavior

            }

            case TEnum(e): {

                NullValue; // Intended Behavior

            }

            case TUnknown: NullValue;

        }

    }

    /\*\*

        Converts `InterpTokens` into a haxe value, depending on its type.

        `Little` functions are not supported.

    \*\*/

    public static function toHaxeValue(val:InterpTokens):Dynamic {

        val = Interpreter.evaluate(val);

        return switch val {

            case ErrorMessage(msg): {

                trace("WARNING: " + msg + ". Returning null");

                return null;

            }

            case TrueValue: true;

            case FalseValue: false;

            case NullValue: null;

            case Decimal(num): num;

            case Number(num): num;

            case Characters(string): string;

            case Object(props, typeName): {

                var obj:Dynamic = {};

                for (key => value in props) {

                    if (key == Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX + Little.keywords.TYPE\_STRING) continue;

                    obj.key = toHaxeValue(value.value);

                }

                obj;

            }

            case ClassPointer(pointer): {

                return Little.memory.getTypeName(pointer);

            }

            case FunctionCode(\_, \_): {

                return null;

            }

            case \_: {

                return null;

            }

        }

    }

    /\*\*

        Converts core Haxe types into `Little` types.

    \*\*/

    public static function toLittleType(type:String) {

        return switch type {

            case "Bool": Little.keywords.TYPE\_BOOLEAN;

            case "Int": Little.keywords.TYPE\_INT;

            case "Float": Little.keywords.TYPE\_FLOAT;

            case "String": Little.keywords.TYPE\_STRING;

            case "Dynamic": Little.keywords.TYPE\_DYNAMIC;

            case \_: type;

        }

    }

}

package little.tools;

import little.interpreter.memory.MemoryPointer;

import little.parser.Parser;

import little.lexer.Tokens.LexerTokens;

import little.lexer.Lexer;

import little.interpreter.Tokens.InterpTokens;

import little.interpreter.Interpreter;

import little.parser.Tokens.ParserTokens;

using little.tools.TextTools;

/\*\*

    Contains many convenience methods to help write more elegant code.

\*\*/

class Extensions {

    /\*\*

        True if `token`'s name is contained within `tokens`. False otherwise.

    \*\*/

    overload extern inline public static function is(token:ParserTokens, ...tokens:ParserTokensSimple) {

        return tokens.toArray().map(x -> x.getName().remove("\_").toLowerCase()).contains(token.getName().toLowerCase());

    }

    /\*\*

        True if `token`'s name is contained within `tokens`. False otherwise.

    \*\*/

    overload extern inline public static function is(token:InterpTokens, ...tokens:InterpTokensSimple) {

        return tokens.toArray().map(x -> x.getName().remove("\_").toLowerCase()).contains(token.getName().toLowerCase());

    }

    /\*\*

        Converts code into an array of `InterpTokens`.

    \*\*/

    public static function tokenize(code:String):Array<InterpTokens> {

        return Interpreter.convert(...Parser.parse(Lexer.lex(code)));

    }

    /\*\*

        Converts code into an array of `InterpTokens`, runs them, and returns the result.

    \*\*/

    public static function eval(code:String):InterpTokens {

        return Interpreter.run(Interpreter.convert(...Parser.parse(Lexer.lex(code))));

    }

    /\*\*

        Grabs the `index`th parameter from the enum instance `token`.

    \*\*/

    overload extern inline public static function parameter(token:ParserTokens, index:Int):Dynamic {

        return token.getParameters()[index];

    }

    /\*\*

        Grabs the `index`th parameter from the enum instance `token`.

    \*\*/

    overload extern inline public static function parameter(token:InterpTokens, index:Int):Dynamic {

        return token.getParameters()[index];

    }

    /\*\*

        Whether or not `token` is passed by value.

    \*\*/

    public static inline function passedByValue(token:InterpTokens):Bool {

        return is(token, TRUE\_VALUE, FALSE\_VALUE, NULL\_VALUE, NUMBER, DECIMAL, SIGN, CHARACTERS);

    }

    /\*\*

        Wether or not `token` is passed by reference.

    \*\*/

    public static inline function passedByReference(token:InterpTokens):Bool {

        return !passedByValue(token);

    }

    /\*\*

        Whether or not `token` can be stored statically, taking the same amount of memory each time.

        An exception is made for strings, that are stored a little differently.

    \*\*/

    public static inline function staticallyStorable(token:InterpTokens):Bool {

        return passedByValue(token) || is(token, CHARACTERS);

    }

    /\*\*

        Grabs the string from `Identifier` or `Characters` tokens.

        If `token` is not an `Identifier` or `Characters` token, it will use the result of `Interpreter.run([token])`.

    \*\*/

    public static inline function extractIdentifier(token:InterpTokens):String {

        return is(token, IDENTIFIER) ? parameter(token, 0) : parameter(Interpreter.run([token]), 0);

    }

    /\*\*

        Converts nested `PropertyAccess` and `Identifier` tokens into a string array.

    \*\*/

    public static function asStringPath(token:InterpTokens):Array<String> {

        var path = [];

        var current = token;

        while (current != null) {

            switch current {

                case PropertyAccess(source, property): {

                    path.unshift(extractIdentifier(property));

                    current = source;

                }

                case Identifier(word): {

                    path.unshift(word);

                    current = null;

                }

                case Characters(string):

                    path.unshift('"$string"');

                    current = null;

                default: {

                    path.unshift(extractIdentifier(current));

                    current = null;

                }

            }

        }

        return path;

    }

    /\*\*

        Converts nested `PropertyAccess` and `Identifier` tokens into a string.

    \*\*/

    public static function asJoinedStringPath(token:InterpTokens):String {

        return asStringPath(token).join(Little.keywords.PROPERTY\_ACCESS\_SIGN);

    }

    /\*\*

        Returns the type of `token`.

    \*\*/

    public static function type(token:InterpTokens):String {

        switch token {

            case Characters(string): return Little.keywords.TYPE\_STRING;

            case Number(number): return Little.keywords.TYPE\_INT;

            case Decimal(number): return Little.keywords.TYPE\_FLOAT;

            case TrueValue | FalseValue: return Little.keywords.TYPE\_BOOLEAN;

            case NullValue: return Little.keywords.TYPE\_DYNAMIC;

            case FunctionCode(requiredParams, body): return Little.keywords.TYPE\_FUNCTION;

            case ConditionCode(callers): return Little.keywords.TYPE\_CONDITION;

            case Sign(sign): return Little.keywords.TYPE\_SIGN;

            case Object(\_, typeName): return typeName;

            case ClassPointer(pointer): return Little.keywords.TYPE\_MODULE;

            case \_: throw '$token is not a simple token (given $token)';

        }

    }

    public static function asObjectToken(o:Map<String, InterpTokens>, typeName:String):InterpTokens {

        var map = [for (k => v in o) k => {documentation: "", value: v}];

        map[Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME] = {documentation: 'The type of this object, as a ${Little.keywords.TYPE\_STRING}.', value: InterpTokens.Characters(typeName)};

        return Object(map, typeName);

    }

    public static function asEmptyObject(a:Array<Dynamic>, typeName:String):InterpTokens {

        return Object([Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME => {value: Characters(typeName), documentation: 'The type of this object, as a ${Little.keywords.TYPE\_STRING}.'}], typeName);

    }

    /\*\*

        The reverse of `asJoinedStringPath()`.

    \*\*/

    public static function asTokenPath(string:String):InterpTokens {

        var path = string.split(Little.keywords.PROPERTY\_ACCESS\_SIGN);

        if (path.length == 1) return Identifier(path[0]);

        else return PropertyAccess(asTokenPath(path.slice(0, path.length - 1).join(Little.keywords.PROPERTY\_ACCESS\_SIGN)), Identifier(path.pop()));

    }

    public static function extractValue(address:MemoryPointer, type:String):InterpTokens {

        return switch type {

            case (\_ == Little.keywords.TYPE\_STRING => true): Characters(Little.memory.storage.readString(address));

            case (\_ == Little.keywords.TYPE\_INT => true): Number(Little.memory.storage.readInt32(address));

            case (\_ == Little.keywords.TYPE\_FLOAT => true): Decimal(Little.memory.storage.readDouble(address));

            case (\_ == Little.keywords.TYPE\_BOOLEAN => true): Little.memory.constants.getFromPointer(address);

            case (\_ == Little.keywords.TYPE\_FUNCTION => true): Little.memory.storage.readCodeBlock(address);

            case (\_ == Little.keywords.TYPE\_CONDITION => true): Little.memory.storage.readCondition(address);

            case (\_ == Little.keywords.TYPE\_MODULE => true): ClassPointer(address);

            // Because of the way we store lone nulls (as type dynamic),

            // they might get confused with objects of type dynamic, so we need to do this:

            case ((\_ == Little.keywords.TYPE\_DYNAMIC || \_ == Little.keywords.TYPE\_UNKNOWN) && Little.memory.constants.hasPointer(address) && Little.memory.constants.getFromPointer(address).equals(NullValue) => true): NullValue;

            case (\_ == Little.keywords.TYPE\_SIGN => true): Little.memory.storage.readSign(address);

            case (\_ == Little.keywords.TYPE\_UNKNOWN => true): throw 'Cannot extract value of unknown type';

                // Not sure how someone can even get to the error above, but it's better to be safe than sorry - maybe a developer generates an extern field of type Unknown or something...

            case \_: Little.memory.storage.readObject(address);

        }

    }

    public static function writeInPlace(address:MemoryPointer, value:InterpTokens) {

        var type = type(value);

        switch type {

            case (\_ == Little.keywords.TYPE\_STRING => true): Little.memory.storage.setString(address, parameter(value, 0));

            case (\_ == Little.keywords.TYPE\_INT => true): Little.memory.storage.setInt32(address, parameter(value, 0));

            case (\_ == Little.keywords.TYPE\_FLOAT => true): Little.memory.storage.setDouble(address, parameter(value, 0));

            case (\_ == Little.keywords.TYPE\_FUNCTION => true): Little.memory.storage.setCodeBlock(address, parameter(value, 0));

            case (\_ == Little.keywords.TYPE\_CONDITION => true): Little.memory.storage.setCondition(address, parameter(value, 0));

            case (\_ == Little.keywords.TYPE\_MODULE => true): Little.memory.storage.setPointer(address, parameter(value, 0));

            case (\_ == Little.keywords.TYPE\_SIGN => true): Little.memory.storage.setSign(address, parameter(value, 0));

            case (\_ == Little.keywords.TYPE\_UNKNOWN => true): throw 'Cannot extract value of unknown type';

            case \_: Little.memory.storage.setObject(address, parameter(value, 0));

        }

    }

    /\*\*

        Converts nested `PropertyAccess` and `Identifier` tokens into an array of just `Identifier` tokens.

    \*\*/

    public static function toIdentifierPath(propertyAccess:InterpTokens):Array<InterpTokens> {

        var arr = [];

        var current = propertyAccess;

        while (current != null) {

            switch current {

                case PropertyAccess(source, property): {

                    arr.unshift(property);

                    current = source;

                }

                case \_: {

                    arr.unshift(current);

                    current = null;

                }

            }

        }

        return arr;

    }

    /\*\*

        Whether or not any element of `array` matches `func`.

    \*\*/

    public static function containsAny<T>(array:Array<T>, func:T -> Bool):Bool {

        return array.filter(func).length > 0;

    }

    /\*\*

        Converts an `Iterator` to an `Array`.

    \*\*/

    public static function toArray<T>(iter:Iterator<T>):Array<T> {

        return [for (i in iter) i];

    }

}

enum ParserTokensSimple {

    SET\_LINE;

    SET\_MODULE;

    SPLIT\_LINE;

    VARIABLE;

    FUNCTION;

    CONDITION\_CALL;

    READ;

    WRITE;

    IDENTIFIER;

    TYPE\_DECLARATION;

    FUNCTION\_CALL;

    RETURN;

    EXPRESSION;

    BLOCK;

    PART\_ARRAY;

    PROPERTY\_ACCESS;

    SIGN;

    NUMBER;

    DECIMAL;

    CHARACTERS;

    DOCUMENTATION;

    MODULE;

    EXTERNAL;

    EXTERNAL\_CONDITION;

    ERROR\_MESSAGE;

    NULL\_VALUE;

    TRUE\_VALUE;

    FALSE\_VALUE;

    NOBODY;

}

enum InterpTokensSimple {

    SET\_LINE;

    SET\_MODULE;

    SPLIT\_LINE;

    VARIABLE\_DECLARATION;

    FUNCTION\_DECLARATION;

    CONDITION\_DECLARATION;

    CLASS\_DECLARATION;

    CONDITION\_CALL;

    CONDITION\_CODE;

    FUNCTION\_CALL;

    FUNCTION\_CODE;

    FUNCTION\_RETURN;

    WRITE;

    TYPE\_CAST;

    EXPRESSION;

    BLOCK;

    PART\_ARRAY;

    PROPERTY\_ACCESS;

    NUMBER;

    DECIMAL;

    CHARACTERS;

    DOCUMENTATION;

    CLASS\_POINTER;

    SIGN;

    NULL\_VALUE;

    TRUE\_VALUE;

    FALSE\_VALUE;

    IDENTIFIER;

    TYPE\_REFERENCE;

    OBJECT;

    CLASS;

    ERROR\_MESSAGE;

    HAXE\_EXTERN;

}

package little.tools;

/\*\*

    An enumeration of the different layers of the `Little` Interpreter.

\*\*/

enum abstract Layer(String) from String to String {

    var LEXER = "Lexer";

    var PARSER = "Parser";

    var PARSER\_MACRO = "Parser, Macro";

    var INTERPRETER = "Interpreter";

    var INTERPRETER\_VALUE\_EVALUATOR = "Interpreter, Value Evaluator";

    var INTERPRETER\_EXPRESSION\_EVALUATOR = "Interpreter, Expression Evaluator";

    var INTERPRETER\_TOKEN\_VALUE\_STRINGIFIER = "Interpreter, Token Value Stringifier";

    var INTERPRETER\_TOKEN\_IDENTIFIER\_STRINGIFIER = "Interpreter, Token Identifier Stringifier";

    var MEMORY = "Memory";

    var MEMORY\_REFERRER = "Memory, Referrer";

    var MEMORY\_STORAGE = "Memory, Storage";

    var MEMORY\_EXTERNAL\_INTERFACING = "Memory, External Interfacing";

    var MEMORY\_SIZE\_EVALUATOR = "Memory, Size Evaluator";

    var MEMORY\_GARBAGE\_COLLECTOR = "Memory, Garbage Collector";

    /\*\*

        Gets the 0-based index of a layer, as a string.

        @param layer An instance of the `Layer` enum, or a string representing a layer.

    \*\*/

    public static function getIndexOf(layer:String):Int {

        return switch layer {

            case LEXER: 1;

            case PARSER: 2;

            case PARSER\_MACRO: 3;

            case INTERPRETER: 4;

            case INTERPRETER\_VALUE\_EVALUATOR: 5;

            case INTERPRETER\_EXPRESSION\_EVALUATOR: 6;

            case INTERPRETER\_TOKEN\_VALUE\_STRINGIFIER: 7;

            case INTERPRETER\_TOKEN\_IDENTIFIER\_STRINGIFIER: 8;

            case MEMORY: 9;

            case MEMORY\_REFERRER: 10;

            case MEMORY\_STORAGE: 11;

            case MEMORY\_EXTERNAL\_INTERFACING: 12;

            case MEMORY\_SIZE\_EVALUATOR: 13;

            case MEMORY\_GARBAGE\_COLLECTOR: 14;

            case \_: 999999999;

        }

    }

}

package little.tools;

import haxe.ds.StringMap;

import little.interpreter.memory.MemoryPointer;

import little.interpreter.memory.ExternalInterfacing.ExtTree;

import little.interpreter.memory.Memory;

import little.interpreter.memory.Operators.OperatorType;

import haxe.exceptions.ArgumentException;

import little.interpreter.Runtime;

import haxe.extern.EitherType;

import little.lexer.Lexer;

import little.parser.Parser;

import little.interpreter.Interpreter;

import little.interpreter.Tokens.InterpTokens;

import little.Little.\*;

using little.tools.Plugins;

using little.tools.Extensions;

using little.tools.TextTools;

@:access(little.Little)

@:access(little.interpreter.Runtime)

class Plugins {

    private var memory:Memory;

    /\*\*

        Instantiates the `Plugins` class.

    \*\*/

    public function new(memory:Memory) {

        this.memory = memory;

    }

    @:noCompletion static var \_\_noTypeCreation:Bool;

    /\*\*

        registers a class in little code, or extends the fields of an existing class. The class' fields are dictated by this function's `fields` attribute,

        which provides instance & static functions, variables, and nested objects.

        The allowed key-value types in `fields`'s key-value pairs:

        |Key Syntax                                           | Type                                                                                                                          | Application       | Description |

        |-----------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|-------------------|-------------|

        |`public <Type> <name>`                               | `(address:MemoryPointer, value:InterpTokens) -> InterpTokens`                                                                 | Instance Variable | A function that returns a value, that can be based on its parent. The returned value is stored in memory upon retrieval. |

        |`public <Type> <name>`                               | `(address:MemoryPointer, value:InterpTokens) -> {address:MemoryPointer, value:InterpTokens}`                                  | Instance Variable | A function that returns a value, that can be based on its parent. The returned value is not stored in memory, and we rely upon the given pointer to be correct. |

        |`public <Type> <name> (define <param> as <Type>)`    | `(address:MemoryPointer, value:InterpTokens, givenParams:Array<InterpTokens>) -> InterpTokens`                                | Instance Function | A function, that returns a value based on its parent & other given parameters, provided by a Little function call. The returned value is stored in memory upon retrieval. |

        |`static <Type> <name>`                               | `() -> InterpTokens`                                                                                                          | Static Variable   | A function that returns a static value. The returned value is stored in memory upon retrieval. |

        |`static <Type> <name>`                               | `() -> {address:MemoryPointer, value:InterpTokens}`                                                                           | Static Variable   | A function that returns a static value. The returned value is not stored in memory, and we rely upon the given pointer to be correct. |

        |`static <Type> <name> ()`                            | `(givenParams:Array<InterpTokens>) -> InterpTokens`                                                                           | Static Function   | A function that returns a value based on some given parameters, provided by a Little function call. The returned value is stored in memory upon retrieval. |

        |`static <Type> <name>`                               | `TypeFields`                                                                                                                  | Static Variable   | Another option for a static variable, but this time it's for a nested object. The object itself isn't allocated in Little's memory, but its decedents may be. instance objects are not available this way, since they are tied to an object, thus needing to be allocated many times. |

**\*\*Notice\*\*** - key syntax is very sensitive - must start with `public` or `static`, continue with a `little` type, then a name, and parameters if its a function. Each element separated by a single whitespace. Example:

            'public Number id'

            'static ${Conversion.toLittleType("String")} getProfile (define summed as ${Conversion.toLittleType("Bool")})'

**\*\*Notice 2\*\*** - for function parameters, syntax follows Little function parameter syntax - multiple parameter declarations, with optional type and optional default values, separated by a comma.

        @param typeName The name of the class. May be nested inside other "packages" using a `.` (for example. my\_pack.MyClass)

        @param fields A string map that has key-value pairs of certain types. Refer to the table above for more information.

    \*\*/

    public function registerType(typeName:String, fields:TypeFields) {

        var instances = memory.externs.createPathFor(memory.externs.instanceProperties, ...typeName.split("."));

        var statics = memory.externs.createPathFor(memory.externs.globalProperties, ...typeName.split("."));

        instances.type = statics.type = memory.getTypeInformation(Little.keywords.TYPE\_MODULE).pointer;

        if (\_\_noTypeCreation) \_\_noTypeCreation = false;

        else if (!memory.externs.externToPointer.exists(typeName) && !memory.constants.hasType(typeName)) {

            memory.externs.externToPointer[typeName] = memory.storage.storeByte(1);

            statics.getter = (\_, \_) -> {

                objectValue: ClassPointer(memory.externs.externToPointer[typeName]),

                objectAddress: memory.externs.externToPointer[typeName]

            }

        } else if (memory.constants.hasType(typeName) && !memory.externs.externToPointer.exists(typeName)) {

            memory.externs.externToPointer[typeName] = memory.constants.getType(typeName);

            statics.getter = (\_, \_) -> {

                objectValue: ClassPointer(memory.externs.externToPointer[typeName]),

                objectAddress: memory.externs.externToPointer[typeName]

            }

        }

        for (key => field in fields) {

            switch key.split(" ") {

                case (\_[0] == "public" && \_.length == 3) => true: {

                    var name = key.split(" ")[2];

                    var type = memory.getTypeInformation(key.split(" ")[1]).pointer;

                    instances.properties[name] = new ExtTree(type, (value, address) -> {

                        // We can't optimize for the two cases outside of the callback, since haxe doesn't support

                        // type checking on function types.

                        try {

                            var result = untyped field(address, value);

                            if (result is InterpTokens) {

                                return {

                                    objectValue: result,

                                    objectAddress: memory.store(result)

                                };

                            }

                            return {

                                objectValue: untyped result.value,

                                objectAddress: untyped result.address

                            }

                        } catch (e) {

                            return {

                                objectValue: ErrorMessage('External Variable Error: ' + e.details()),

                                objectAddress: memory.constants.ERROR

                            }

                        }

                    });

                }

                case (\_[0] == "public") => true: {

                    var name = key.split(" ")[2];

                    var type = memory.getTypeInformation(key.split(" ")[1]);

                    var params = Interpreter.convert(...Parser.parse(Lexer.lex(key.replaceFirst('public function $name ', "").replaceFirst("(", "").replaceLast(")", ""))));

                    var paramMap = new OrderedMap<String, InterpTokens>();

                    for (entry in params) {

                        if (entry.is(SPLIT\_LINE, SET\_LINE)) continue;

                        switch entry {

                            case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, Identifier(Little.keywords.TYPE\_DYNAMIC));

                            case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, type);

                            case Write(assignees, value): {

                                switch assignees[0] {

                                    case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(value, Identifier(Little.keywords.TYPE\_DYNAMIC));

                                    case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(value, type);

                                    default:

                                }

                            }

                            default:

                        }

                    }

                    instances.properties[name] = new ExtTree(memory.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer, (value, address) -> {

                        var returnType:InterpTokens = type.typeName.asTokenPath();

                        return {

                            objectValue: FunctionCode(paramMap, Block([

                                FunctionReturn(HaxeExtern(() -> {

                                    var result = (field : (MemoryPointer, InterpTokens, Array<InterpTokens>) -> InterpTokens)(address, value, paramMap.keys().toArray().map(key -> Interpreter.evaluate(memory.read(key).objectValue)));

                                    return result;

                                }), returnType)

                            ], returnType)),

                            objectAddress: memory.constants.EXTERN

                        }

                    });

                }

                case (\_[0] == "static" && \_.length == 3) => true: {

                    var name = key.split(" ")[2];

                    var type = memory.getTypeInformation(key.split(" ")[1]).pointer;

                    if (field is StringMap) {

                        \_\_noTypeCreation = true;

                        registerType(typeName + "." + name, field);

                        continue;

                    }

                    statics.properties[name] = new ExtTree(type, (\_, \_) -> {

                        // We can't optimize for the two cases outside of the callback, since haxe doesn't support

                        // type checking on function types.

                            try {

                                var result = untyped field();

                                if (result is InterpTokens) {

                                    return {

                                        objectValue: result,

                                        objectAddress: memory.store(result)

                                    };

                                }

                                return {

                                    objectValue: untyped result.value,

                                    objectAddress: untyped result.address

                                }

                            } catch (e) {

                                return {

                                    objectValue: ErrorMessage('External Variable Error: ' + e.details()),

                                    objectAddress: memory.constants.ERROR

                                }

                            }

                    });

                }

                case (\_[0] == "static") => true: {

                    var name = key.split(" ")[2];

                    var type = memory.getTypeInformation(key.split(" ")[1]);

                    var params = Interpreter.convert(...Parser.parse(Lexer.lex(key.replaceFirst('static function $name ', "").replaceFirst("(", "").replaceLast(")", ""))));

                    var paramMap = new OrderedMap<String, InterpTokens>();

                    for (entry in params) {

                        if (entry.is(SPLIT\_LINE, SET\_LINE)) continue;

                        switch entry {

                            case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, Identifier(Little.keywords.TYPE\_DYNAMIC));

                            case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, type);

                            case Write(assignees, value): {

                                switch assignees[0] {

                                    case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(value, Identifier(Little.keywords.TYPE\_DYNAMIC));

                                    case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(value, type);

                                    default:

                                }

                            }

                            default:

                        }

                    }

                    statics.properties[name] = new ExtTree(memory.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer, (\_, \_) -> {

                        var returnType:InterpTokens = type.typeName.asTokenPath();

                        return {

                            objectValue: FunctionCode(paramMap, Block([

                                FunctionReturn(HaxeExtern(() -> {

                                    var result = untyped field(paramMap.keys().toArray().map(key -> Interpreter.evaluate(memory.read(key).objectValue)));

                                    return result;

                                }), returnType)

                            ], returnType)),

                            objectAddress: memory.constants.EXTERN

                        }

                    });

                }

                case \_: throw 'Invalid key syntax for `$key`. Must start with either `public`/`static` `function`/`var`, and end with a variable name. (Example: `public var myVar`). Each item must be separated by a single whitespace.';

            }

        }

    }

    /\*\*

        registers a haxe value/property inside Little code.

        @param variableName the name of the variable, for usage in Little code. If you want it nested in some kind of path, use `.` (e.g. `mother.varName`)

        @param variableType the type of the variable, in little. Use `Conversion.toLittleType` for haxe basic types if needed.

        @param documentation documentation for this variable.

        @param staticValue **\*\*Option 1\*\*** - a static value to assign to this variable

        @param valueGetter **\*\*Option 2\*\*** - a function that returns a value that this variable gives when accessed.

    \*\*/

    public function registerVariable(variableName:String, variableType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:Void -> InterpTokens) {

        var varPath = variableName.split(".");

        var object = memory.externs.createPathFor(memory.externs.globalProperties, ...varPath);

        object.type = memory.getTypeInformation(variableName).pointer;

        object.getter = (\_, \_) -> {

            return try {

                var value = staticValue == null ? valueGetter() : staticValue;

                {

                    objectValue: value,

                    objectAddress: memory.store(value),

                    // objectDoc: documentation ?? ""

                }

            } catch (e) {

                {

                    objectValue: ErrorMessage('External Variable Error: ' + e.details()),

                    objectAddress: memory.constants.ERROR,

                    // objectDoc: ""

                }

            }

        }

    }

    /\*\*

        Allows usage of a function written in haxe inside Little code.

        @param actionName The name by which to identify the function. If you want this nested in some kind of path, use `.` (e.g. `mother.funcName`)

        @param documentation documentation for this function.

        @param expectedParameters an `Array<InterpTokens>` consisting of `InterpTokens.Variable`s which contain the names & types of the parameters that should be passed on to the function. For example:

            ```

            [VariableDeclaration(Identifier(x), Identifier("Characters"))]

            ```

**\*\*alternatively\*\*** - can be normal parameter "list" written in little:

            ```

            define x as Characters, define index = 3, define y

            ```

**\*\*Important\*\*** - if variables appear in the end and have assigned values, they are optional.

        @param callback The actual function, which gets an array of the given parameters as reduced little tokens (basic types and `Object`), and returns a value based on them

        @param returnType The type of the returned value. This exists due to implementation limitations. You can use the `Conversion` class to know which types to put here.

    \*\*/

    public function registerFunction(functionName:String, ?documentation:String, expectedParameters:EitherType<String, Array<InterpTokens>>, callback:Array<{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}> -> InterpTokens, returnType:String) {

        var params = if (expectedParameters is String) {

            Interpreter.convert(...Parser.parse(Lexer.lex(expectedParameters)));

        } else (expectedParameters : Array<InterpTokens>);

        var functionPath = functionName.split(".");

        var paramMap = new OrderedMap<String, InterpTokens>();

        for (entry in params) {

            if (entry.is(SPLIT\_LINE, SET\_LINE)) continue;

            switch entry {

                case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, Identifier(Little.keywords.TYPE\_DYNAMIC));

                case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, type);

                case Write(assignees, value): {

                    switch assignees[0] {

                        case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(value, Identifier(Little.keywords.TYPE\_DYNAMIC));

                        case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(value, type);

                        default:

                    }

                }

                default:

            }

        }

        var returnTypeToken = Interpreter.convert(...Parser.parse(Lexer.lex(returnType)))[0]; // May be a PropertyAccess or an Identifier

        var token:InterpTokens = FunctionCode(paramMap, Block([

            FunctionReturn(HaxeExtern(() -> callback(paramMap.keys().toArray().map(key -> memory.read(key)))), returnTypeToken)

        ], returnTypeToken));

        var object = memory.externs.createPathFor(memory.externs.globalProperties, ...functionPath);

        object.type = memory.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer;

        object.getter = (\_, \_) -> {

            objectValue: token,

            objectAddress: memory.constants.EXTERN,

            // objectDoc: documentation ?? ""

        }

    }

    /\*\*

        Adds a condition to be used in Little.

        Conditions are can be described as a special function, that decides how many time another given function is run, and with which parameters.

        Their syntax:

            <condition\_name> (<params>) {

                <body>

            }

        @param conditionName The name by which to identify the condition. If you want this nested in some kind of path, use `.` (e.g. `mother.conditionName`)

        @param documentation documentation for this condition.

        @param callback The actual function, which gets an array of the given parameters, exactly as given (which means they might require further evaluation),

            and another array representing the block of code right after the condition, and return an outcome token of the condition.

            The outcome is usually expected to be the last value in the last iteration of the condition (for example, the same as haxe `if` statements)

    \*\*/

    public function registerCondition(conditionName:String, ?documentation:String ,callback:(params:Array<InterpTokens>, body:Array<InterpTokens>) -> InterpTokens) {

        var conditionPath = conditionName.split(".");

        var object = memory.externs.createPathFor(memory.externs.globalProperties, ...conditionPath);

        object.getter = (\_, \_) -> {

            objectValue: ConditionCode([

                null => Block([

                    HaxeExtern(() -> callback(

                        Interpreter.convert(...Parser.parse(Lexer.lex(memory.read(Little.keywords.CONDITION\_PATTERN\_PARAMETER\_NAME).objectValue.parameter(0)))).slice(1),

                        Interpreter.convert(...Parser.parse(Lexer.lex(memory.read(Little.keywords.CONDITION\_BODY\_PARAMETER\_NAME).objectValue.parameter(0))))

                        )

                    ) // No FunctionReturn here, since conditions propagate existing returns, and if we put one here, it will get propagated outside

                      // The scope of this condition, which results in unexpected behavior (program/function quits prematurely)

                ], null)

            ]),

            objectAddress: memory.constants.EXTERN,

            // objectDoc: documentation ?? ""

        }

    }

    /\*\*

        Registers a haxe-property-like variable on a little class found at `onType`.

        @param propertyName The name of the property, must not include property access sign.

        @param propertyType The type of the property. Must be a little class.

        @param onType The type of the object the property is on. Must be a little class, and if the class is nested within an object, a full path must be specified.

        @param documentation The documentation of the property

        @param staticValue **\*\*Option 1\*\***. A static value this property always returns.

        @param valueGetter **\*\*Option 2\*\***. A function that returns the value of the property. It takes in the value of the parent object, and it's address in memory.

    \*\*/

    public function registerInstanceVariable(propertyName:String, propertyType:String, onType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:(objectValue:InterpTokens, objectAddress:MemoryPointer) -> InterpTokens) {

        var classPath = onType.split(".");

        classPath.push(propertyName);

        var object = memory.externs.createPathFor(memory.externs.instanceProperties, ...classPath);

        object.type = memory.getTypeInformation(propertyType).pointer;

        object.getter = (v, a) -> {

            return try {

                var value = staticValue == null ? valueGetter(v, a) : staticValue;

                {

                    objectValue: value,

                    objectAddress: memory.store(value),

                    // objectDoc: documentation ?? ""

                }

            } catch (e) {

                {

                    objectValue: ErrorMessage('External Function Error: ' + e.details()),

                    objectAddress: memory.constants.ERROR,

                    // objectDoc: ""

                }

            }

        }

    }

    /\*\*

        Registers a method on every object of the given type, that can be called from Little.

        @param propertyName The name of the property, must not include property access sign.

        @param onType The type of the object the property is on. Must be a little class, and if the class is nested within an object, a full path must be specified.

        @param documentation The documentation of the property

        @param expectedParameters an `Array<InterpTokens>` consisting of `InterpTokens.Variable`s which contain the names & types of the parameters that should be passed on to the function. For example:

            ```

            [VariableDeclaration(Identifier(x), Identifier("Characters"))]

            ```

**\*\*alternatively\*\*** - can be normal parameter "list" written in little:

            ```

            define x as Characters, define index = 3, define y

            ```

**\*\*Important\*\*** - if variables appear in the end and have assigned values, they are optional.

        @param callback The actual function, which gets 3 parameters: the value of the object, the address of the object in memory, and an array of the given parameters, exactly as the user gave them (which means they might require further evaluation). The function should return something at the end, or a `VoidValue`.

        @param returnType The type of the returned value. This exists due to implementation limitations. You can use the `Conversion` class to know which types to put here.

    \*\*/

    public function registerInstanceFunction(propertyName:String, onType:String, ?documentation:String, expectedParameters:EitherType<String, Array<InterpTokens>>, callback:(objectValue:InterpTokens, objectAddress:MemoryPointer, params:Array<{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}>) -> InterpTokens, returnType:String) {

        var params = if (expectedParameters is String) {

            Interpreter.convert(...Parser.parse(Lexer.lex(expectedParameters)));

        } else (expectedParameters : Array<InterpTokens>);

        var paramMap = new OrderedMap<String, InterpTokens>();

        for (entry in params) {

            if (entry.is(SPLIT\_LINE, SET\_LINE)) continue;

            switch entry {

                case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, Identifier(Little.keywords.TYPE\_DYNAMIC));

                case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(NullValue, type);

                case Write(assignees, value): {

                    switch assignees[0] {

                        case VariableDeclaration(name, null, \_): paramMap[name.extractIdentifier()] = TypeCast(value, Identifier(Little.keywords.TYPE\_DYNAMIC));

                        case VariableDeclaration(name, type, \_): paramMap[name.extractIdentifier()] = TypeCast(value, type);

                        default:

                    }

                }

                default:

            }

        }

        var classPath = onType.split(".");

        classPath.push(propertyName);

        var object = memory.externs.createPathFor(memory.externs.instanceProperties, ...classPath);

        var returnTypeToken = Interpreter.convert(...Parser.parse(Lexer.lex(returnType)))[0]; // May be a PropertyAccess or an Identifier

        object.type = memory.getTypeInformation(Little.keywords.TYPE\_FUNCTION).pointer;

        object.getter = (v, a) -> {

            return try {

                {

                    objectValue: FunctionCode(paramMap, Block([

                        FunctionReturn(HaxeExtern(() -> callback(v, a, paramMap.keys().toArray().map(key -> memory.read(key)))), returnTypeToken)

                    ], returnTypeToken)),

                    objectAddress: memory.constants.EXTERN,

                    // objectDoc: documentation ?? ""

                }

            } catch (e) {

                {

                    objectValue: ErrorMessage('External Function Error: ' + e.details()),

                    objectAddress: memory.constants.ERROR,

                    // objectDoc: ""

                }

            }

        }

    }

    /\*\*

        Checks if the given `Array<{lhs:String, rhs:String}>` contains the given `lhs` and `rhs` combination.

    \*\*/

    static function combosHas(combos:Array<{lhs:String, rhs:String}>, lhs:String, rhs:String) {

        for (c in combos) if (c.rhs == rhs && c.lhs == lhs) return true;

        return false;

    }

    /\*\*

        Registers an operator, to be used in expressions in `Little`. An operator can be a single sided operator, or a double sided operator.

        An operator can also accept specific types for each side, or specific types for both sides.

        @param symbol a `String` which is the symbol of the operator. Must not contain any letters or whitespaces.

        @param info Information about the operator, including callback for when the operator is used and other properties.

    \*\*/

    public function registerOperator(symbol:String, info:OperatorInfo) {

        if (info.operatorType == null || info.operatorType == LHS\_RHS) {

            if (info.callback == null && info.singleSidedOperatorCallback != null)

                throw new ArgumentException("callback", 'Incorrect callback given for operator type ${info.operatorType ?? LHS\_RHS} - `singleSidedOperatorCallback` was given, when `callback` was expected');

            else if (info.callback == null)

                throw new ArgumentException("callback", 'No callback given for operator type ${info.operatorType ?? LHS\_RHS} (`callback` is null)');

            var callbackFunc:(InterpTokens, InterpTokens) -> InterpTokens;

            // A bunch of ifs in order to shorten the final callback function, improves performance a bit

            if (info.lhsAllowedTypes != null && info.rhsAllowedTypes == null && info.allowedTypeCombos == null) {

                callbackFunc = (lhs, rhs) -> {

                    final lType = Interpreter.evaluate(lhs).type(), rType = Interpreter.evaluate(rhs).type();

                    if (!info.lhsAllowedTypes.contains(lType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot preform $lType(${lhs.extractIdentifier()}) $symbol $rType(${rhs.extractIdentifier()}) - Left operand cannot be of type $lType (accepted types: ${info.lhsAllowedTypes})'));

                    }

                    return info.callback(lhs, rhs);

                }

            } else if (info.lhsAllowedTypes == null && info.rhsAllowedTypes != null && info.allowedTypeCombos == null) {

                callbackFunc = (lhs, rhs) -> {

                    final lType = Interpreter.evaluate(lhs).type(), rType = Interpreter.evaluate(rhs).type();

                    if (!info.rhsAllowedTypes.contains(rType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot preform $lType(${lhs.extractIdentifier()}) $symbol $rType(${rhs.extractIdentifier()}) - Right operand cannot be of type $rType (accepted types: ${info.rhsAllowedTypes})'));

                    }

                    return info.callback(lhs, rhs);

                }

            } else if (info.lhsAllowedTypes != null && info.rhsAllowedTypes != null && info.allowedTypeCombos == null) {

                callbackFunc = (lhs, rhs) -> {

                    final lType = Interpreter.evaluate(lhs).type(), rType = Interpreter.evaluate(rhs).type();

                    if (!info.rhsAllowedTypes.contains(rType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot preform $lType(${lhs.extractIdentifier()}) $symbol $rType(${rhs.extractIdentifier()}) - Right operand cannot be of type $rType (accepted types: ${info.rhsAllowedTypes})'));

                    }

                    if (!info.rhsAllowedTypes.contains(lType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot preform $lType(${lhs.extractIdentifier()}) $symbol $rType(${rhs.extractIdentifier()}) - Left operand cannot be of type $lType (accepted types: ${info.lhsAllowedTypes})'));

                    }

                    return info.callback(lhs, rhs);

                }

            } else if (info.lhsAllowedTypes != null && info.rhsAllowedTypes == null && info.allowedTypeCombos != null) {

                callbackFunc = (lhs, rhs) -> {

                    final lType = Interpreter.evaluate(lhs).type(), rType = Interpreter.evaluate(rhs).type();

                    if (!info.lhsAllowedTypes.contains(lType) && !info.allowedTypeCombos.containsCombo(lType, rType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot preform $lType(${lhs.extractIdentifier()}) $symbol $rType(${rhs.extractIdentifier()}) - Right operand cannot be of type $rType while left operand is of type $lType (accepted types for left operand: ${info.lhsAllowedTypes}, accepted type combinations: ${info.allowedTypeCombos.map(object -> '${object.rhs} $symbol ${object.lhs}')})'));

                    }

                    return info.callback(lhs, rhs);

                }

            } else if (info.lhsAllowedTypes == null && info.rhsAllowedTypes != null && info.allowedTypeCombos != null) {

                callbackFunc = (lhs, rhs) -> {

                    final lType = Interpreter.evaluate(lhs).type(), rType = Interpreter.evaluate(rhs).type();

                    if (!info.rhsAllowedTypes.contains(rType) && !info.allowedTypeCombos.containsCombo(lType, rType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot preform $lType(${lhs.extractIdentifier()}) $symbol $rType(${rhs.extractIdentifier()}) - Right operand cannot be of type $rType while left operand is of type $lType (accepted types for right operand: ${info.rhsAllowedTypes}, accepted type combinations: ${info.allowedTypeCombos.map(object -> '${object.rhs} $symbol ${object.lhs}')})'));

                    }

                    return info.callback(lhs, rhs);

                }

            } else if (info.lhsAllowedTypes != null && info.rhsAllowedTypes != null && info.allowedTypeCombos != null) {

                callbackFunc = (lhs, rhs) -> {

                    final lType = Interpreter.evaluate(lhs).type(), rType = Interpreter.evaluate(rhs).type();

                    if (!info.rhsAllowedTypes.contains(rType) && !info.allowedTypeCombos.containsCombo(lType, rType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot preform $lType(${lhs.extractIdentifier()}) $symbol ${rType}(${rhs.extractIdentifier()}) - Right operand cannot be of type $rType (accepted types: ${info.rhsAllowedTypes}, accepted type combinations: ${info.allowedTypeCombos.map(object -> '${object.rhs} $symbol ${object.lhs}')})'));

                    }

                    if (!info.rhsAllowedTypes.contains(lType) && !info.allowedTypeCombos.containsCombo(lType, rType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot preform $lType(${lhs.extractIdentifier()}) $symbol ${rType}(${rhs.extractIdentifier()}) - Left operand cannot be of type $lType (accepted types: ${info.lhsAllowedTypes}, accepted type combinations: ${info.allowedTypeCombos.map(object -> '${object.rhs} $symbol ${object.lhs}')})'));

                    }

                    return info.callback(lhs, rhs);

                }

            } else callbackFunc = info.callback;

            Little.memory.operators.add(symbol, LHS\_RHS, info.priority, callbackFunc);

        } else { // One sided operator

            if (info.singleSidedOperatorCallback == null && info.callback != null)

                throw new ArgumentException("singleSidedOperatorCallback", 'Incorrect callback given for operator type ${info.operatorType} - `callback` was given, when `singleSidedOperatorCallback` was expected');

            else if (info.singleSidedOperatorCallback == null)

                throw new ArgumentException("singleSidedOperatorCallback", 'No callback given for operator type ${info.operatorType ?? LHS\_RHS} (`singleSidedOperatorCallback` is null)');

            var callbackFunc:InterpTokens -> InterpTokens;

            if (info.operatorType == LHS\_ONLY) {

                callbackFunc = (lhs) -> {

                    var lType = Interpreter.evaluate(lhs).type();

                    if (!info.lhsAllowedTypes.contains(lType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot perform $lType(${lhs.extractIdentifier()})$symbol - Operand cannot be of type $lType (accepted types: ${info.lhsAllowedTypes})'));

                    }

                    return info.singleSidedOperatorCallback(lhs);

                }

            } else {

                callbackFunc = (rhs) -> {

                    var rType = Interpreter.evaluate(rhs).type();

                    if (!info.rhsAllowedTypes.contains(rType)) {

                        return Little.runtime.throwError(ErrorMessage('Cannot perform $symbol$rType(${rhs.extractIdentifier()}) - Operand cannot be of type $rType (accepted types: ${info.rhsAllowedTypes})'));

                    }

                    return info.singleSidedOperatorCallback(rhs);

                }

            }

            Little.memory.operators.add(symbol, info.operatorType, info.priority, callbackFunc);

        }

    }

    /\*\*

        Checks if the given array contains the given combo

    \*\*/

    static function containsCombo(array:Array<{lhs:String, rhs:String}>, lhs:String, rhs:String):Bool {

        for (a in array) {

            if (a.lhs == lhs && a.rhs == rhs) return true;

        }

        return false;

    }

}

/\*\*

    Info about an operator

\*\*/

typedef OperatorInfo = {

    ?lhsAllowedTypes:Array<String>,

    ?rhsAllowedTypes:Array<String>,

    ?allowedTypeCombos:Array<{lhs:String, rhs:String}>,

    ?callback:(InterpTokens, InterpTokens) -> InterpTokens,

    ?singleSidedOperatorCallback:InterpTokens -> InterpTokens,

    ?operatorType:OperatorType,

    /\*\*

        @see Little.memory.operators.setPriority

    \*\*/

    ?priority:String,

}

/\*\*

    Used to represent the fields of a type

\*\*/

typedef TypeFields = Map<String, OneOfSeven<

    // Instance fields:

    (address:MemoryPointer, value:InterpTokens) -> InterpTokens, // variable

    (address:MemoryPointer, value:InterpTokens) -> {address:MemoryPointer, value:InterpTokens}, // variable, with pointer

    (address:MemoryPointer, value:InterpTokens, givenParams:Array<InterpTokens>) -> InterpTokens, // function

        // Static fields:

    () -> InterpTokens, // variable

    () -> {address:MemoryPointer, value:InterpTokens}, // variable

    (givenParams:Array<InterpTokens>) -> InterpTokens, // function

    TypeFields // nested object

>>;

/\*\*

    Can be used to represent the fields of a type

\*\*/

abstract OneOfSeven<T1, T2, T3, T4, T5, T6, T7>(Dynamic)

    from T1 from T2 from T3 from T4 from T5 from T6 from T7

    to T1 to T2 to T3 to T4 to T5 to T6 to T7 {}

package little.tools;

import vision.ds.ByteArray;

import little.interpreter.memory.Memory;

import little.interpreter.memory.HashTables;

import little.interpreter.memory.MemoryPointer;

import little.interpreter.memory.MemoryPointer.POINTER\_SIZE;

import haxe.Json;

import haxe.xml.Access;

import vision.tools.MathTools;

import little.lexer.Lexer;

import little.parser.Parser;

import little.interpreter.Interpreter;

import little.interpreter.Runtime;

import little.interpreter.Tokens;

using Std;

using little.tools.TextTools;

using little.tools.Extensions;

@:access(little.interpreter.Runtime)

/\*\*

    Contains `Little`'s standard library, as a group of functions, each adding different types of features.

\*\*/

class PrepareRun {

    /\*\*

        Whether or not the standard library has been prepared.

    \*\*/

    public static var prepared:Bool = false;

    /\*\*

        Adds Standard library types.

    \*\*/

    public static function addTypes() {

        Little.plugin.registerType(Little.keywords.TYPE\_FUNCTION, []);

        Little.plugin.registerType(Little.keywords.TYPE\_CONDITION, []);

        Little.plugin.registerType(Little.keywords.TYPE\_INT, []);

        Little.plugin.registerType(Little.keywords.TYPE\_FLOAT, []);

        Little.plugin.registerType(Little.keywords.TYPE\_STRING, []);

        Little.plugin.registerType(Little.keywords.TYPE\_SIGN, []);

        Little.plugin.registerType(Little.keywords.TYPE\_MODULE, [

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_STRING} ()' => (address, \_, \_) -> {

                return Conversion.toLittleValue(Little.memory.getTypeName(address));

            }

        ]);

        // Little.plugin.registerType("Date", [

        //  'static ${Little.keywords.TYPE\_STRING} now ()' => (\_) -> {

        //      return Conversion.toLittleValue(Date.now().toString());

        //  }

        // ]);

        Little.plugin.registerType(Little.keywords.TYPE\_INT, [

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_STRING} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue(Std.string(value.parameter(0)));

            },

            'public ${Little.keywords.TYPE\_FLOAT} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_FLOAT} ()' => (\_, value, \_) -> {

                return Decimal(value.parameter(0));

            },

            'public ${Little.keywords.TYPE\_BOOLEAN} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_BOOLEAN} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue(value.parameter(0) != 0);

            }

        ]);

        Little.plugin.registerType(Little.keywords.TYPE\_FLOAT, [

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_STRING} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue(Std.string(value.parameter(0)));

            },

            'public ${Little.keywords.TYPE\_INT} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_INT} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue(Math.floor(value.parameter(0)));

            },

            'public ${Little.keywords.TYPE\_BOOLEAN} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_BOOLEAN} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue(value.parameter(0) != 0);

            },

            'public ${Little.keywords.TYPE\_BOOLEAN} ${Little.keywords.STDLIB\_\_FLOAT\_isWhole} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue((value.parameter(0) : Float) % 1 == 0);

            }

        ]);

        Little.plugin.registerType(Little.keywords.TYPE\_STRING, [

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_STRING} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue(Std.string(value.parameter(0)));

            },

            'public ${Little.keywords.TYPE\_INT} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_INT} ()' => (\_, value, \_) -> {

                var number = Std.parseInt(value.parameter(0));

                if (number == null) {

                    Little.runtime.throwError(ErrorMessage('${Little.keywords.TYPE\_STRING} instance `"${value.parameter(0)}"` cannot be converted to ${Little.keywords.TYPE\_INT}, since it is not a number  '), INTERPRETER);

                }

                return Conversion.toLittleValue(number);

            },

            'public ${Little.keywords.TYPE\_FLOAT} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_FLOAT} ()' => (\_, value, \_) -> {

                var number = Std.parseFloat(value.parameter(0));

                if (number == Math.NaN) {

                    Little.runtime.throwError(ErrorMessage('${Little.keywords.TYPE\_STRING} instance `"${value.parameter(0)}"` cannot be converted to ${Little.keywords.TYPE\_FLOAT}, since it is not a number    '), INTERPRETER);

                }

                return Conversion.toLittleValue(number);

            },

            'public ${Little.keywords.TYPE\_SIGN} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_SIGN} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue(Sign(value.parameter(0)));

            },

            'public ${Little.keywords.TYPE\_BOOLEAN} ${Little.keywords.TYPE\_CAST\_FUNCTION\_PREFIX}${Little.keywords.TYPE\_BOOLEAN} ()' => (\_, value, \_) -> {

                return Conversion.toLittleValue(value.parameter(0) == "true" || (Std.parseFloat(value.parameter(0)) != Math.NaN && Std.parseFloat(value.parameter(0)) != 0));

            },

            'public ${Little.keywords.TYPE\_INT} ${Little.keywords.STDLIB\_\_STRING\_length}' => (\_, value) -> {

                return Conversion.toLittleValue(value.parameter(0).length);

            },

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.STDLIB\_\_STRING\_charAt} (define index as ${Little.keywords.TYPE\_INT})' => (\_, value, params) -> {

                return Conversion.toLittleValue(value.parameter(0).charAt(Conversion.toHaxeValue(params[0])));

            },

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.STDLIB\_\_STRING\_substring} (define start as ${Little.keywords.TYPE\_INT}, define end as ${Little.keywords.TYPE\_INT} = -1)' => (\_, value, params) -> {

                return Characters(value.parameter(0).substring(Conversion.toHaxeValue(params[0]), Conversion.toHaxeValue(params[1])));

            },

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.STDLIB\_\_STRING\_toLowerCase} ()' => (\_, value, \_) -> {

                return Characters(value.parameter(0).toLowerCase());

            },

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.STDLIB\_\_STRING\_toUpperCase} ()' => (\_, value, \_) -> {

                return Characters(value.parameter(0).toUpperCase());

            },

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.STDLIB\_\_STRING\_replace} (define search as ${Little.keywords.TYPE\_STRING}, define replace as ${Little.keywords.TYPE\_STRING})' => (\_, value, params) -> {

                return Characters(value.parameter(0).replace(Conversion.toHaxeValue(params[0]), Conversion.toHaxeValue(params[1])));

            },

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.STDLIB\_\_STRING\_trim} ()' => (\_, value, \_) -> {

                return Characters(value.parameter(0).trim());

            },

            'public ${Little.keywords.TYPE\_STRING} ${Little.keywords.STDLIB\_\_STRING\_remove} (define search as ${Little.keywords.TYPE\_STRING})' => (\_, value, params) -> {

                return Characters(value.parameter(0).replace(Conversion.toHaxeValue(params[0]), ""));

            },

            'public ${Little.keywords.TYPE\_BOOLEAN} ${Little.keywords.STDLIB\_\_STRING\_contains} (define search as ${Little.keywords.TYPE\_STRING})' => (\_, value, params) -> {

                return Conversion.toLittleValue(value.parameter(0).contains(Conversion.toHaxeValue(params[0])));

            },

            'public ${Little.keywords.TYPE\_INT} ${Little.keywords.STDLIB\_\_STRING\_indexOf} (define search as ${Little.keywords.TYPE\_STRING})' => (\_, value, params) -> {

                return Conversion.toLittleValue(value.parameter(0).indexOf(Conversion.toHaxeValue(params[0])));

            },

            'public ${Little.keywords.TYPE\_INT} ${Little.keywords.STDLIB\_\_STRING\_lastIndexOf} (define search as ${Little.keywords.TYPE\_STRING})' => (\_, value, params) -> {

                return Conversion.toLittleValue(value.parameter(0).lastIndexOf(Conversion.toHaxeValue(params[0])));

            },

            'public ${Little.keywords.TYPE\_BOOLEAN} ${Little.keywords.STDLIB\_\_STRING\_startsWith} (define prefix as ${Little.keywords.TYPE\_STRING})' => (\_, value, params) -> {

                return Conversion.toLittleValue(value.parameter(0).indexOf(Conversion.toHaxeValue(params[0]) == 0));

            },

            'public ${Little.keywords.TYPE\_BOOLEAN} ${Little.keywords.STDLIB\_\_STRING\_endsWith} (define postfix as ${Little.keywords.TYPE\_STRING})' => (\_, value, params) -> {

                return Conversion.toLittleValue(value.parameter(0).indexOf(Conversion.toHaxeValue(params[0])) == value.parameter(0).length - Conversion.toHaxeValue(params[0]).length);

            },

            'static ${Little.keywords.TYPE\_STRING} ${Little.keywords.STDLIB\_\_STRING\_fromCharCode} (define code as ${Little.keywords.TYPE\_INT})' => (params) -> {

                return Conversion.toLittleValue(String.fromCharCode(Conversion.toHaxeValue(params[0])));

            }

        ]);

        Little.plugin.registerType(Little.keywords.TYPE\_OBJECT, [

            'static ${Little.keywords.TYPE\_OBJECT} ${Little.keywords.INSTANTIATE\_FUNCTION\_NAME} ()' => (params) -> {

                return [].asEmptyObject(Little.keywords.TYPE\_OBJECT);

            }

        ]);

        Little.plugin.registerType(Little.keywords.TYPE\_ARRAY, [

            'static ${Little.keywords.TYPE\_ARRAY} ${Little.keywords.INSTANTIATE\_FUNCTION\_NAME} (define type as ${Little.keywords.TYPE\_MODULE}, define length as ${Little.keywords.TYPE\_INT})' => (params) -> {

                var arrayType:String = Conversion.toHaxeValue(params[0]);

                var size = Little.memory.getTypeInformation(arrayType).defaultInstanceSize;

                var length:Int = Conversion.toHaxeValue(params[1]);

                var byteArray = Little.memory.storage.storeArray(length, size);

                return ["\_\_p" => Number(byteArray.toInt()), "\_\_t" => params[0]].asObjectToken(Little.keywords.TYPE\_ARRAY);

            },

            'public ${Little.keywords.TYPE\_INT} ${Little.keywords.STDLIB\_\_ARRAY\_length}' => (address, value) -> {

                var pointer:Int = Conversion.toHaxeValue(untyped value.parameter(0).get("\_\_p").value);

                return {

                    value: Number(Little.memory.storage.readInt32(pointer)),

                    address: MemoryPointer.fromInt(pointer)

                }

            },

            'public ${Little.keywords.TYPE\_MODULE} ${Little.keywords.STDLIB\_\_ARRAY\_elementType}' => (address, value) -> {

                var typeToken:InterpTokens = untyped value.parameter(0).get("\_\_t").value;

                return {

                    value: typeToken,

                    address: typeToken.parameter(0)

                }

            },

            'public ${Little.keywords.TYPE\_DYNAMIC} ${Little.keywords.STDLIB\_\_ARRAY\_get} (define index as ${Little.keywords.TYPE\_INT})' => (address, value, params) -> {

                var pointer:Int = Conversion.toHaxeValue(untyped value.parameter(0).get("\_\_p").value);

                var elementType:String = Conversion.toHaxeValue(untyped value.parameter(0).get("\_\_t").value);

                var index = Conversion.toHaxeValue(params[0]);

                var elementSize = Little.memory.storage.readInt32(pointer + 4);

                var specificElement = pointer + 4 /\* array length\*/ + 4 /\* array element size \*/ + index \* elementSize;

                return MemoryPointer.fromInt(specificElement).extractValue(elementType);

            },

            'public ${Little.keywords.TYPE\_DYNAMIC} ${Little.keywords.STDLIB\_\_ARRAY\_set} (define index as ${Little.keywords.TYPE\_INT}, define value as ${Little.keywords.TYPE\_DYNAMIC})' => (address, value, params) -> {

                var pointer:Int = Conversion.toHaxeValue(untyped value.parameter(0).get("\_\_p").value);

                var index = Conversion.toHaxeValue(params[0]);

                var elementSize = Little.memory.storage.readInt32(pointer + 4);

                var specificElement = pointer + 4 /\* array length\*/ + 4 /\* array element size \*/ + index \* elementSize;

                MemoryPointer.fromInt(specificElement).writeInPlace(params[1]);

                return NullValue;

            }

        ]);

        Little.plugin.registerType(Little.keywords.TYPE\_MEMORY, [

            'static ${Little.keywords.TYPE\_INT} ${Little.keywords.STDLIB\_\_MEMORY\_allocate} (define amount as ${Little.keywords.TYPE\_INT})' => (params) -> {

                return Conversion.toLittleValue(Little.memory.allocate(Conversion.toHaxeValue(params[0])));

            },

            'static ${Little.keywords.TYPE\_DYNAMIC} ${Little.keywords.STDLIB\_\_MEMORY\_free} (define address as ${Little.keywords.TYPE\_INT}, define amount as ${Little.keywords.TYPE\_INT})' => (params) -> {

                Little.memory.free(Conversion.toHaxeValue(params[0]), Conversion.toHaxeValue(params[1]));

                return NullValue;

            },

            'static ${Little.keywords.TYPE\_DYNAMIC} ${Little.keywords.STDLIB\_\_MEMORY\_read} (define address as ${Little.keywords.TYPE\_INT}, define type as ${Little.keywords.TYPE\_MODULE})' => (params) -> {

                return MemoryPointer.fromInt(params[0].parameter(0)).extractValue(Conversion.toHaxeValue(params[1]));

            },

            'static ${Little.keywords.TYPE\_DYNAMIC} ${Little.keywords.STDLIB\_\_MEMORY\_write} (define address as ${Little.keywords.TYPE\_INT}, define bytes as ${Little.keywords.TYPE\_ARRAY})' => (params) -> {

                var arrayRef = Conversion.toHaxeValue(params[1]).parameter(0);

                var arrayPointer = MemoryPointer.fromInt(arrayRef.get("\_\_p").value.parameter(0)); // Number(p)

                var arrayType = arrayRef.get("\_\_t").value.parameter(0); // ClassPointer(p)

                if (arrayType != Little.memory.constants.INT || arrayType != Little.memory.constants.FLOAT || arrayType != Little.memory.constants.BOOL) {

                    Little.runtime.throwError(ErrorMessage('${Little.keywords.STDLIB\_\_MEMORY\_write} only supports  ${Little.keywords.TYPE\_INT}, ${Little.keywords.TYPE\_FLOAT} or ${Little.keywords.TYPE\_BOOLEAN} arrays, as they\'re the only ones able to meaningfully represent bytes.'));

                }

                var array = Little.memory.storage.readArray(arrayPointer);

                var byteArray = new ByteArray(array.length);

                for (i in 0...array.length) { byteArray.setUInt8(i, array[i].getUInt8(i)); }

                Little.memory.storage.setBytes(Conversion.toHaxeValue(params[0]), byteArray);

                return NullValue;

            },

            'static ${Little.keywords.TYPE\_INT} ${Little.keywords.STDLIB\_\_MEMORY\_size}' => () -> {

                return Conversion.toLittleValue(Little.memory.currentMemorySize);

            },

            'static ${Little.keywords.TYPE\_INT} ${Little.keywords.STDLIB\_\_MEMORY\_maxSize}' => () -> {

                return Conversion.toLittleValue(Little.memory.maxMemorySize);

            }

        ]);

    }

    /\*\*

        Adds standard library, top-level functions.

    \*\*/

    public static function addFunctions() {

        Little.plugin.registerFunction(Little.keywords.PRINT\_FUNCTION\_NAME, null, [VariableDeclaration(Identifier("item"), null)], (params) -> {

            var eval = params[0].objectValue;

            trace(eval, params[0]);

            Little.runtime.\_\_print(eval.is(OBJECT) ? @:privateAccess PrettyPrinter.printInterpreterAst([eval]).split("\n").slice(1).map(s -> s.substring(6)).join("\n") : PrettyPrinter.stringifyInterpreter(eval), eval);

            return NullValue;

        }, Little.keywords.TYPE\_DYNAMIC);

        Little.plugin.registerFunction(Little.keywords.RAISE\_ERROR\_FUNCTION\_NAME, null, [VariableDeclaration(Identifier("message"), null)], (params) -> {

            Little.runtime.throwError(params[0].objectValue);

            return NullValue;

        }, Little.keywords.TYPE\_DYNAMIC);

        Little.plugin.registerFunction(Little.keywords.READ\_FUNCTION\_NAME, null, [VariableDeclaration(Identifier("identifier"), Little.keywords.TYPE\_STRING.asTokenPath())], (params) -> {

            return (Conversion.toHaxeValue(params[0].objectValue) : String).asTokenPath();

        }, Little.keywords.TYPE\_DYNAMIC);

        Little.plugin.registerFunction(Little.keywords.RUN\_CODE\_FUNCTION\_NAME, null, [VariableDeclaration(Identifier("code"), Little.keywords.TYPE\_STRING.asTokenPath())], (params) -> {

            return Interpreter.run(Interpreter.convert(...Parser.parse(Lexer.lex(Conversion.toHaxeValue(params[0].objectValue)))));

        }, Little.keywords.TYPE\_DYNAMIC);

    }

    /\*\*

        Adds standard instance properties to core types.

    \*\*/

    public static function addProps() {

        Little.plugin.registerInstanceVariable(Little.keywords.OBJECT\_TYPE\_PROPERTY\_NAME, Little.keywords.TYPE\_STRING, Little.keywords.TYPE\_DYNAMIC, 'The name of this value\'s type, as a ${Little.keywords.TYPE\_STRING}',

            (value, address) -> {

                return ClassPointer(Little.memory.getTypeInformation(value.type()).pointer);

            }

        );

        Little.plugin.registerInstanceVariable(Little.keywords.OBJECT\_ADDRESS\_PROPERTY\_NAME, POINTER\_SIZE == 4 ? Little.keywords.TYPE\_INT : Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_DYNAMIC, 'The address of this value',

            (value:InterpTokens, address:MemoryPointer) -> {

                return POINTER\_SIZE == 4 ? Number(address.rawLocation) : Decimal(address.rawLocation);

            }

        );

    }

    /\*\*

        Adds standard library operators.

    \*\*/

    public static function addSigns() {

        // --------------------------------------------------

        // ------------------------RHS-----------------------

        // --------------------------------------------------

        Little.plugin.registerOperator(Little.keywords.POSITIVE\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            operatorType: RHS\_ONLY,

            priority: "last",

            singleSidedOperatorCallback: (rhs) -> {

                var r = Conversion.toHaxeValue(rhs);

                if (r is Int)

                    return Number(r);

                return Decimal(r);

            }

        });

        Little.plugin.registerOperator(Little.keywords.NEGATE\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            operatorType: RHS\_ONLY,

            priority: 'with ${Little.keywords.POSITIVE\_SIGN}\_',

            singleSidedOperatorCallback: (rhs) -> {

                var r = Conversion.toHaxeValue(rhs);

                if (r is Int)

                    return Number(-r);

                return Decimal(-r);

            }

        });

        Little.plugin.registerOperator(Little.keywords.SQRT\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            operatorType: RHS\_ONLY,

            priority: "first",

            singleSidedOperatorCallback: (rhs) -> {

                var r:Float = Conversion.toHaxeValue(rhs);

                return Decimal(Math.sqrt(r));

            }

        });

        Little.plugin.registerOperator(Little.keywords.NOT\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_BOOLEAN],

            operatorType: RHS\_ONLY,

            priority: 'with ${Little.keywords.POSITIVE\_SIGN}\_',

            singleSidedOperatorCallback: (rhs) -> {

                var r = Conversion.toHaxeValue(rhs);

                return r ? FalseValue : TrueValue;

            }

        });

        // --------------------------------------------------

        // ------------------------LHS-----------------------

        // --------------------------------------------------

        Little.plugin.registerOperator(Little.keywords.FACTORIAL\_SIGN, {

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            operatorType: LHS\_ONLY,

            priority: 'with ${Little.keywords.SQRT\_SIGN}\_',

            singleSidedOperatorCallback: (lhs) -> {

                var l = Conversion.toHaxeValue(lhs);

                var shifted = Math.pow(10, 10) \* l;

                if (shifted != Math.floor(shifted)) return Number(Math.round(MathTools.factorial(l)));

                return Decimal(MathTools.factorial(l));

            }

        });

        // --------------------------------------------------

        // ----------------------STANDARD--------------------

        // --------------------------------------------------

        Little.plugin.registerOperator(Little.keywords.ADD\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT, Little.keywords.TYPE\_STRING],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT, Little.keywords.TYPE\_STRING],

            allowedTypeCombos: [{lhs: Little.keywords.TYPE\_STRING, rhs: Little.keywords.TYPE\_DYNAMIC}, {lhs: Little.keywords.TYPE\_DYNAMIC, rhs: Little.keywords.TYPE\_STRING}],

            priority: 'with ${Little.keywords.POSITIVE\_SIGN}\_',

            callback: (lhs, rhs) -> {

                lhs = Interpreter.evaluate(lhs); rhs = Interpreter.evaluate(rhs);

                var l:Dynamic = Conversion.toHaxeValue(lhs),

                    r:Dynamic = Conversion.toHaxeValue(rhs);

                if (l is String || r is String) {

                    l ??= Little.keywords.NULL\_VALUE; r ??= Little.keywords.NULL\_VALUE;

                    return Characters("" + l + r);

                }

                if (l is Int && r is Int)

                    return Number(l + r);

                return Decimal(#if static untyped #else cast #end l + r);

            }

        });

        Little.plugin.registerOperator(Little.keywords.SUBTRACT\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            allowedTypeCombos: [{lhs: Little.keywords.TYPE\_STRING, rhs: Little.keywords.TYPE\_STRING}],

            priority: 'with ${Little.keywords.ADD\_SIGN}',

            callback: (lhs, rhs) -> {

                lhs = Interpreter.evaluate(lhs); rhs = Interpreter.evaluate(rhs);

                var l:Dynamic = Conversion.toHaxeValue(lhs),

                    r:Dynamic = Conversion.toHaxeValue(rhs);

                if (l is String) {

                    l ??= Little.keywords.NULL\_VALUE; r ??= Little.keywords.NULL\_VALUE;

                    return Characters(TextTools.subtract(l, r));

                }

                if (lhs.type() == Little.keywords.TYPE\_INT && rhs.type() == Little.keywords.TYPE\_INT)

                    return Number(untyped l - r);

                return Decimal(#if static untyped #else cast #end l - r);

            }

        });

        Little.plugin.registerOperator(Little.keywords.MULTIPLY\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            allowedTypeCombos: [{lhs: Little.keywords.TYPE\_STRING, rhs: Little.keywords.TYPE\_INT}],

            priority: 'between ${Little.keywords.ADD\_SIGN} ${Little.keywords.SQRT\_SIGN}\_',

            callback: (lhs, rhs) -> {

                lhs = Interpreter.evaluate(lhs); rhs = Interpreter.evaluate(rhs);

                var l:Dynamic = Conversion.toHaxeValue(lhs),

                    r:Dynamic = Conversion.toHaxeValue(rhs);

                if (l is String) {

                    l ??= Little.keywords.NULL\_VALUE;

                    return Characters(TextTools.multiply(l, r));

                }

                if (lhs.type() == Little.keywords.TYPE\_INT && rhs.type() == Little.keywords.TYPE\_INT)

                    return Number(untyped l \* r);

                return Decimal(#if static untyped #else cast #end l \* r);

            }

        });

        Little.plugin.registerOperator(Little.keywords.DIVIDE\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            priority: 'with ${Little.keywords.MULTIPLY\_SIGN}',

            callback: (lhs, rhs) -> {

                var l:Float = Conversion.toHaxeValue(lhs),

                    r:Float = Conversion.toHaxeValue(rhs);

                trace(l, r, Decimal(l / r));

                if (r == 0)

                    return Little.runtime.throwError(ErrorMessage('Cannot divide by 0'));

                return Decimal(#if static untyped #else cast #end l / r);

            }

        });

        Little.plugin.registerOperator(Little.keywords.POW\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            priority: 'before ${Little.keywords.MULTIPLY\_SIGN}',

            callback: (lhs, rhs) -> {

                lhs = Interpreter.evaluate(lhs); rhs = Interpreter.evaluate(rhs);

                var l:Float = Conversion.toHaxeValue(lhs),

                    r:Float = Conversion.toHaxeValue(rhs);

                if (lhs.type() == Little.keywords.TYPE\_INT && rhs.type() == Little.keywords.TYPE\_INT)

                    return Number(Math.pow(l, r).int());

                return Decimal(Math.pow(l, r));

            }

        });

        Little.plugin.registerOperator(Little.keywords.SQRT\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            priority: 'with ${Little.keywords.POW\_SIGN}',

            callback: (lhs, rhs) -> {

                var l:Float = Conversion.toHaxeValue(lhs),

                    r:Float = Conversion.toHaxeValue(rhs);

                var lPositive = l >= 0;

                var oddN = r % 2 == 1;

                if (!lPositive)

                    l = -l;

                return Decimal(Math.pow(l \* ((!lPositive && oddN) ? -1 : 1), 1 / r));

            }

        });

        // Boolean

        Little.plugin.registerOperator(Little.keywords.AND\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_BOOLEAN],

            lhsAllowedTypes: [Little.keywords.TYPE\_BOOLEAN],

            priority: "last",

            callback: (lhs, rhs) -> Conversion.toHaxeValue(lhs) && Conversion.toHaxeValue(rhs) ? TrueValue : FalseValue});

        Little.plugin.registerOperator(Little.keywords.OR\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_BOOLEAN],

            lhsAllowedTypes: [Little.keywords.TYPE\_BOOLEAN],

            priority: 'with ${Little.keywords.AND\_SIGN}',

            callback: (lhs, rhs) -> Conversion.toHaxeValue(lhs) || Conversion.toHaxeValue(rhs) ? TrueValue : FalseValue});

        Little.plugin.registerOperator(Little.keywords.EQUALS\_SIGN, {

            priority: "last",

            callback: (lhs, rhs) -> Conversion.toHaxeValue(lhs) == Conversion.toHaxeValue(rhs) ? TrueValue : FalseValue

        });

        Little.plugin.registerOperator(Little.keywords.NOT\_EQUALS\_SIGN, {

            priority: 'with ${Little.keywords.EQUALS\_SIGN}',

            callback: (lhs, rhs) -> Conversion.toHaxeValue(lhs) != Conversion.toHaxeValue(rhs) ? TrueValue : FalseValue

        });

        Little.plugin.registerOperator(Little.keywords.XOR\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_BOOLEAN],

            lhsAllowedTypes: [Little.keywords.TYPE\_BOOLEAN],

            priority: 'with ${Little.keywords.AND\_SIGN}',

            callback: (lhs, rhs) -> Conversion.toHaxeValue(lhs) != Conversion.toHaxeValue(rhs) ? TrueValue : FalseValue

        });

        Little.plugin.registerOperator(Little.keywords.LARGER\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            allowedTypeCombos: [{lhs: Little.keywords.TYPE\_STRING, rhs: Little.keywords.TYPE\_STRING}],

            priority: 'with ${Little.keywords.EQUALS\_SIGN}',

            callback: (lhs, rhs) -> {

                var l:Dynamic = Conversion.toHaxeValue(lhs),

                    r:Dynamic = Conversion.toHaxeValue(rhs);

                if (l is String)

                    return l.length > r.length ? TrueValue : FalseValue;

                return l > r ? TrueValue : FalseValue;

            }

        });

        Little.plugin.registerOperator(Little.keywords.LARGER\_EQUALS\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            allowedTypeCombos: [{lhs: Little.keywords.TYPE\_STRING, rhs: Little.keywords.TYPE\_STRING}],

            priority: 'with ${Little.keywords.EQUALS\_SIGN}',

            callback: (lhs, rhs) -> {

                var l:Dynamic = Conversion.toHaxeValue(lhs),

                    r:Dynamic = Conversion.toHaxeValue(rhs);

                if (l is String)

                    return l.length >= r.length ? TrueValue : FalseValue;

                return l >= r ? TrueValue : FalseValue;

            }

        });

        Little.plugin.registerOperator(Little.keywords.SMALLER\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            allowedTypeCombos: [{lhs: Little.keywords.TYPE\_STRING, rhs: Little.keywords.TYPE\_STRING}],

            priority: 'with ${Little.keywords.EQUALS\_SIGN}',

            callback: (lhs, rhs) -> {

                var l:Dynamic = Conversion.toHaxeValue(lhs),

                    r:Dynamic = Conversion.toHaxeValue(rhs);

                if (l is String)

                    return l.length < r.length ? TrueValue : FalseValue;

                return l < r ? TrueValue : FalseValue;

            }

        });

        Little.plugin.registerOperator(Little.keywords.SMALLER\_EQUALS\_SIGN, {

            rhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            lhsAllowedTypes: [Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_INT],

            allowedTypeCombos: [{lhs: Little.keywords.TYPE\_STRING, rhs: Little.keywords.TYPE\_STRING}],

            priority: 'with ${Little.keywords.EQUALS\_SIGN}',

            callback: (lhs, rhs) -> {

                var l:Dynamic = Conversion.toHaxeValue(lhs),

                    r:Dynamic = Conversion.toHaxeValue(rhs);

                if (l is String)

                    return l.length <= r.length ? TrueValue : FalseValue;

                return l <= r ? TrueValue : FalseValue;

            }

        });

    }

    /\*\*

        Adds standard library top-level conditions and loops.

    \*\*/

    public static function addConditions() {

        Little.plugin.registerCondition(Little.keywords.CONDITION\_\_WHILE\_LOOP, "A loop that executes code until the condition is not met", (params, body) -> {

            var val = NullValue;

            var safetyNet = 0;

            while (safetyNet < 500000) {

                var condition:Dynamic = Conversion.toHaxeValue(Interpreter.calculate(params));

                if (condition is Bool && condition) {

                    val = Interpreter.run(body);

                    safetyNet++;

                }

                else if (condition is Bool && !condition) {

                    return val;

                }

                else {

                    Little.runtime.throwError(ErrorMessage('`${Little.keywords.CONDITION\_\_WHILE\_LOOP}` condition must be a ${Little.keywords.TYPE\_BOOLEAN} or ${Little.keywords.FALSE\_VALUE}'), INTERPRETER);

                    return val;

                }

            }

            if (safetyNet >= 500000) {

                Little.runtime.throwError(ErrorMessage('Too much iteration in `${Little.keywords.CONDITION\_\_WHILE\_LOOP}` loop (is `${PrettyPrinter.stringifyInterpreter(params)}` forever `${Little.keywords.TRUE\_VALUE}`?)'), INTERPRETER);

            }

            return val;

        });

        Little.plugin.registerCondition(Little.keywords.CONDITION\_\_IF, "Executes the following block of code if the given condition is true.", (params, body) -> {

            trace(params);

            var val = NullValue;

            var cond = Conversion.toHaxeValue(Interpreter.calculate(params));

            if (cond is Bool && cond) {

                val = Interpreter.run(body);

            }

            else if (!(cond is Bool)) {

                Little.runtime.throwError(ErrorMessage('`${Little.keywords.CONDITION\_\_IF}` condition must be a ${Little.keywords.TYPE\_BOOLEAN}'), INTERPRETER);

            }

            return val;

        });

        Little.plugin.registerCondition(Little.keywords.CONDITION\_\_FOR\_LOOP, "A loop that executes code while changing a variable, until it meets a condition", (params:Array<InterpTokens>, body) -> {

            var val = NullValue;

            var fp = [];

            // Incase one does `from (4 + 2)` and it accidentally parses a function

            for (p in params) {

                switch p {

                    case FunctionCall(\_.parameter(0) == Little.keywords.FOR\_LOOP\_FROM => true, params): {

                        fp.push(Identifier(Little.keywords.FOR\_LOOP\_FROM));

                        fp.push(Expression(params.parameter(0), null));

                    }

                    case FunctionCall(\_.parameter(0) == Little.keywords.FOR\_LOOP\_TO => true, params): {

                        fp.push(Identifier(Little.keywords.FOR\_LOOP\_TO));

                        fp.push(Expression(params.parameter(0), null));

                    }

                    case FunctionCall(\_.parameter(0) == Little.keywords.FOR\_LOOP\_JUMP => true, params): {

                        fp.push(Identifier(Little.keywords.FOR\_LOOP\_JUMP));

                        fp.push(Expression(params.parameter(0), null));

                    }

                    case FunctionCall(\_.is(BLOCK) && [Little.keywords.FOR\_LOOP\_FROM, Little.keywords.FOR\_LOOP\_TO, Little.keywords.FOR\_LOOP\_JUMP].contains(Interpreter.evaluate(\_).parameter(0)) => true, params): {

                        fp.push(Identifier(Interpreter.evaluate(p.parameter(0) /\*The Block\*/).parameter(0)));

                        fp.push(Expression(params.parameter(0), null));

                    }

                    case \_: fp.push(p);

                }

            }

            params = fp;

            if (!params[0].is(VARIABLE\_DECLARATION)) {

                Little.runtime.throwError(ErrorMessage('`${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop must start with a variable to count on (expected definition/block, found: `${PrettyPrinter.stringifyInterpreter(params[0])}`)'));

                return val;

            }

            var typeName = (params[0].parameter(1) : InterpTokens).asJoinedStringPath();

            if (![Little.keywords.TYPE\_INT, Little.keywords.TYPE\_FLOAT, Little.keywords.TYPE\_DYNAMIC, Little.keywords.TYPE\_UNKNOWN].contains(typeName)) {

                Little.runtime.throwError(ErrorMessage('`${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop\'s variable must be of type ${Little.keywords.TYPE\_INT}, ${Little.keywords.TYPE\_FLOAT} or ${Little.keywords.TYPE\_DYNAMIC} (given: ${typeName})'));

            }

            var from:Null<Float> = null, to:Null<Float> = null, jump:Float = 1;

            var currentExpression = [];

            var currentlySet:Int = -1; // 0 for FROM, 1 for TO, 2 for JUMP

            for (i in 1...params.length) {

                switch params[i] {

                    case Identifier(\_ == Little.keywords.FOR\_LOOP\_FROM => true): {

                        if (currentExpression.length > 0) {

                            switch currentlySet {

                                case -1: Little.runtime.throwError(ErrorMessage('Invalid `${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop syntax: expected a `${Little.keywords.FOR\_LOOP\_TO}`, `${Little.keywords.FOR\_LOOP\_FROM}` or `${Little.keywords.FOR\_LOOP\_JUMP}` after the variable'));

                                case 0: Little.runtime.throwError(ErrorMessage('Cannot repeat `${Little.keywords.FOR\_LOOP\_FROM}` tag twice in `${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop.'));

                                case 1: to = Conversion.toHaxeValue(Interpreter.calculate(currentExpression));

                                case 2: jump = Conversion.toHaxeValue(Interpreter.calculate(currentExpression));

                            }

                        }

                        currentExpression = [];

                        currentlySet = 0;

                    }

                    case Identifier(\_ == Little.keywords.FOR\_LOOP\_TO => true): {

                        if (currentExpression.length > 0) {

                            switch currentlySet {

                                case -1: Little.runtime.throwError(ErrorMessage('Invalid `${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop syntax: expected a `${Little.keywords.FOR\_LOOP\_TO}`, `${Little.keywords.FOR\_LOOP\_FROM}` or `${Little.keywords.FOR\_LOOP\_JUMP}` after the variable'));

                                case 0: from = Conversion.toHaxeValue(Interpreter.calculate(currentExpression));

                                case 1: Little.runtime.throwError(ErrorMessage('Cannot repeat `${Little.keywords.FOR\_LOOP\_TO}` tag twice in `${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop.'));

                                case 2: jump = Conversion.toHaxeValue(Interpreter.calculate(currentExpression));

                            }

                        }

                        currentExpression = [];

                        currentlySet = 1;

                    }

                    case Identifier(\_ == Little.keywords.FOR\_LOOP\_JUMP => true): {

                        if (currentExpression.length > 0) {

                            switch currentlySet {

                                case -1: Little.runtime.throwError(ErrorMessage('Invalid `${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop syntax: expected a `${Little.keywords.FOR\_LOOP\_TO}`, `${Little.keywords.FOR\_LOOP\_FROM}` or `${Little.keywords.FOR\_LOOP\_JUMP}` after the variable'));

                                case 0: from = Conversion.toHaxeValue(Interpreter.calculate(currentExpression));

                                case 1: to = Conversion.toHaxeValue(Interpreter.calculate(currentExpression));

                                case 2: Little.runtime.throwError(ErrorMessage('Cannot repeat `${Little.keywords.FOR\_LOOP\_JUMP}` tag twice in `${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop.'));

                            }

                        }

                        currentExpression = [];

                        currentlySet = 2;

                    }

                    case \_: currentExpression.push(params[i]);

                }

            }

            switch currentlySet {

                case -1: Little.runtime.throwError(ErrorMessage('Invalid `${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop syntax: expected a `${Little.keywords.FOR\_LOOP\_TO}`, `${Little.keywords.FOR\_LOOP\_FROM}` or `${Little.keywords.FOR\_LOOP\_JUMP}` after the variable'));

                case 0: Little.runtime.throwError(ErrorMessage('Cannot repeat `${Little.keywords.FOR\_LOOP\_FROM}` tag twice in `${Little.keywords.CONDITION\_\_FOR\_LOOP}` loop.'));

                case 1: to = Conversion.toHaxeValue(Interpreter.calculate(currentExpression));

                case 2: jump = Conversion.toHaxeValue(Interpreter.calculate(currentExpression));

            }

            if (from < to) {

                while (from < to) {

                    val = Interpreter.run([

                        Write([params[0]], Conversion.toLittleValue(from))

                    ].concat(body));

                    from += jump;

                }

            } else {

                while (from > to) {

                    val = Interpreter.run([

                        Write([params[0]], Conversion.toLittleValue(from))

                    ].concat(body));

                    from -= jump;

                }

            }

            return val;

        });

        Little.plugin.registerCondition(Little.keywords.CONDITION\_\_AFTER, (params:Array<InterpTokens>, body:Array<InterpTokens>) -> {

            var val = NullValue;

            var ident:String = "";

            if (params[0].is(BLOCK)) {

                var output = Interpreter.run(params[0].parameter(0));

                Interpreter.assert(output, [IDENTIFIER, PROPERTY\_ACCESS], '`${Little.keywords.CONDITION\_\_AFTER}` condition that starts with a code block must have it\'s code block return an identifier using the `${Little.keywords.READ\_FUNCTION\_NAME}` function (returned: ${PrettyPrinter.stringifyInterpreter(output)})');

                ident = output.asJoinedStringPath();

                params[0] = output;

            } else if (params[0].is(IDENTIFIER, PROPERTY\_ACCESS)) {

                ident = params[0].extractIdentifier();

            } else {

                Little.runtime.throwError(ErrorMessage('`${Little.keywords.CONDITION\_\_AFTER}` condition must start with a variable to watch (expected definition, found: `${PrettyPrinter.stringifyInterpreter(params[0])}`)'));

                return val;

            }

            /\*\*

                Listens for when `ident` is written to.

            \*\*/

            function listener(setIdentifiers:Array<String>) {

                var cond:Bool = Conversion.toHaxeValue(Interpreter.calculate(params));

                if (setIdentifiers.contains(ident) && cond) {

                    Interpreter.run(body);

                    Little.runtime.onWriteValue.remove(listener);

                }

            }

            Little.runtime.onWriteValue.push(listener);

            return val;

        });

        Little.plugin.registerCondition(Little.keywords.CONDITION\_\_WHENEVER, (params:Array<InterpTokens>, body:Array<InterpTokens>) -> {

            var val = NullValue;

            var ident:String = "";

            if (params[0].is(BLOCK)) {

                var output = Interpreter.evaluate(params[0]);

                Interpreter.assert(output, [IDENTIFIER, PROPERTY\_ACCESS], '`${Little.keywords.CONDITION\_\_WHENEVER}` condition that starts with a code block must have it\'s code block return a `${Little.keywords.TYPE\_STRING}` (returned: ${PrettyPrinter.stringifyInterpreter(output)})');

                ident = Conversion.toHaxeValue(output);

            } else if (params[0].is(IDENTIFIER, PROPERTY\_ACCESS)) {

                ident = params[0].extractIdentifier();

            } else {

                Little.runtime.throwError(ErrorMessage('`${Little.keywords.CONDITION\_\_WHENEVER}` condition must start with a variable to watch (expected definition, found: `${PrettyPrinter.stringifyInterpreter(params[0])}`)'));

                return val;

            }

            /\*\*

                Listens for when `ident` is written to.

            \*\*/

            function listener(setIdentifiers:Array<String>) {

                var cond:Bool = Conversion.toHaxeValue(Interpreter.calculate(params));

                if (setIdentifiers.contains(ident) && cond) {

                    Interpreter.run(body);

                }

            }

            Little.runtime.onWriteValue.push(listener);

            return val;

        });

    }

}

package little.tools;

import haxe.exceptions.NotImplementedException;

import little.interpreter.Tokens.InterpTokens;

import little.interpreter.memory.Operators.OperatorType;

import haxe.ds.ArraySort;

import vision.algorithms.Radix;

import little.interpreter.Interpreter;

using StringTools;

using little.tools.TextTools;

using little.tools.Extensions;

import little.parser.Tokens;

/\*\*

    A class containing stringifiers of complex structures in this library,

    specifically `ParserTokens` and `InterpTokens`.

\*\*/

class PrettyPrinter {

    /\*\*

        Pretty-Prints an array of `ParserTokens` as a tree, with it's origin being `Ast` (Abstract Syntax Tree)

        @param ast The tokens to stringify

        @param spacingBetweenNodes The length of an indent between nested nodes.

    \*\*/

    public static function printParserAst(ast:Array<ParserTokens>, ?spacingBetweenNodes:Int = 6) {

        if (ast == null) return "null (look for errors in input)";

        s = " ".multiply(spacingBetweenNodes);

        var unfilteredResult = getTree\_PARSER(Expression(ast, null), [], 0, true);

        var filtered = "";

        for (line in unfilteredResult.split("\n")) {

            if (line == "└─── Expression")

                continue;

            filtered += line.substring(spacingBetweenNodes - 1) + "\n";

        }

        return "\nAst\n" + filtered;

    }

    /\*\*

        Pretty-Prints an array of `InterpTokens` as a tree, with it's origin being `Ast` (Abstract Syntax Tree)

        @param ast The tokens to stringify

        @param spacingBetweenNodes The length of an indent between nested nodes.

    \*\*/

    public static function printInterpreterAst(ast:Array<InterpTokens>, ?spacingBetweenNodes:Int = 6) {

        if (ast == null) return "null (look for errors in input)";

        s = " ".multiply(spacingBetweenNodes);

        var unfilteredResult = getTree\_INTERP(Expression(ast, null), [], 0, true);

        var filtered = "";

        for (line in unfilteredResult.split("\n")) {

            if (line == "└─── Expression")

                continue;

            filtered += line.substring(spacingBetweenNodes - 1) + "\n";

        }

        return "\nAst\n" + filtered;

    }

    /\*\*

        Prefix For Array

    \*\*/

    static function prefixFA(pArray:Array<Int>):String {

        var prefix = "";

        for (i in 0...l) {

            if (pArray[i] == 1) {

                prefix += "│" + s.substring(1);

            } else {

                prefix += s;

            }

        }

        return prefix;

    }

    /\*\*

        Pushes Index to Array

    \*\*/

    static function pushIndex(pArray:Array<Int>, i:Int) {

        var arr = pArray.copy();

        arr[i + 1] = 1;

        return arr;

    }

    static var s = "";

    static var l = 0;

    /\*\*

        returns string representation of tree

        @param root The token to start from

        @param prefix An array of prefixes, to determine

        @param level The depth of the tree in the current recursion

        @param last Whether or not the current node is the last

        @return The string representation

    \*\*/

    @:noCompletion static function getTree\_PARSER(root:ParserTokens, prefix:Array<Int>, level:Int, last:Bool):String {

        l = level;

        var t = if (last) "└" else "├";

        var c = "├";

        var d = "───";

        if (root == null)

            return ''; //'${prefixFA(prefix)}$t$d SetLine($line)\n'

        switch root {

            case SetLine(line): return '${prefixFA(prefix)}$t$d SetLine($line)\n';

            case SetModule(module): return '${prefixFA(prefix)}$t$d SetModule($module)\n';

            case SplitLine: return '${prefixFA(prefix)}$t$d SplitLine\n';

            case Characters(string): return '${prefixFA(prefix)}$t$d "$string"\n';

            case ErrorMessage(name): return '${prefixFA(prefix)}$t$d Error: $name\n';

            case Documentation(doc): return '${prefixFA(prefix)}$t$d Documentation: ${doc.replace("\n", "\n" + prefixFA(prefix) + '│                  ')}\n';

            case Decimal(num): return '${prefixFA(prefix)}$t$d $num\n';

            case Number(num): return '${prefixFA(prefix)}$t$d $num\n';

            case FalseValue: return '${prefixFA(prefix)}$t$d ${Little.keywords.FALSE\_VALUE}\n';

            case TrueValue: return '${prefixFA(prefix)}$t$d ${Little.keywords.TRUE\_VALUE}\n';

            case NullValue: return '${prefixFA(prefix)}$t$d ${Little.keywords.NULL\_VALUE}\n';

            case Variable(name, type, doc):

                {

                    var title = '${prefixFA(prefix)}$t$d Variable Creation\n';

                    if (doc != null) title += getTree\_PARSER(doc, prefix.copy(), level + 1, false);

                    title += getTree\_PARSER(name, prefix.copy(), level + 1, type == null);

                    if (type != null) title += getTree\_PARSER(type, prefix.copy(), level + 1, true);

                    return title;

                }

            case Function(name, params, type, doc):

                {

                    var title = '${prefixFA(prefix)}$t$d Function Creation\n';

                    if (doc != null) title += getTree\_PARSER(doc, prefix.copy(), level + 1, false);

                    title += getTree\_PARSER(name, prefix.copy(), level + 1, false);

                    title += getTree\_PARSER(params, prefix.copy(), level + 1, type == null);

                    if (type != null) title += getTree\_PARSER(type, prefix.copy(), level + 1, true);

                    return title;

                }

            case ConditionCall(name, exp, body):

                {

                    var title = '${prefixFA(prefix)}$t$d Condition\n';

                    title += getTree\_PARSER(name, prefix.copy(), level + 1, false);

                    title += getTree\_PARSER(exp, pushIndex(prefix, level), level + 1, false);

                    title += getTree\_PARSER(body, prefix.copy(), level + 1, true);

                    return title;

                }

            case Read(name):

                return '${prefixFA(prefix)}$t$d Read: $name\n';

            case Write(assignees, value):

                {

                    return'${prefixFA(prefix)}$t$d Variable Write\n${getTree\_PARSER(PartArray(assignees), pushIndex(prefix, level), level + 1, false)}${getTree\_PARSER(value, prefix.copy(), level + 1, true)}';

                }

            case Sign(value):

                {

                    return '${prefixFA(prefix)}$t$d $value\n';

                }

            case TypeDeclaration(value, type):

                {

                    return '${prefixFA(prefix)}$t$d Type Declaration\n${getTree\_PARSER(value, if (type == null) prefix.copy() else pushIndex(prefix, level), level + 1, type == null)}${getTree\_PARSER(type, prefix.copy(), level + 1, true)}';

                }

            case Identifier(value): {

                return '${prefixFA(prefix)}$t$d $value\n';

            }

            case Expression(parts, type):

                {

                    if (parts.length == 0)

                        return '${prefixFA(prefix)}$t$d <empty expression>\n';

                    var strParts = ['${prefixFA(prefix)}$t$d Expression\n${getTree\_PARSER(type, prefix.copy(), level + 1, false)}'].concat([

                        for (i in 0...parts.length - 1) getTree\_PARSER(parts[i], pushIndex(prefix, level), level + 1, false)

                    ]);

                    strParts.push(getTree\_PARSER(parts[parts.length - 1], prefix.copy(), level + 1, true));

                    return strParts.join("");

                }

            case Custom(name, parts): {

                if (parts.length == 0) return '${prefixFA(prefix)}$t$d $name\n';

                var strParts = ['${prefixFA(prefix)}$t$d $name\n'].concat([

                    for (i in 0...parts.length - 1) getTree\_PARSER(parts[i], pushIndex(prefix, level), level + 1, false)

                ]);

                strParts.push(getTree\_PARSER(parts[parts.length - 1], prefix.copy(), level + 1, true));

                return strParts.join("");

            }

            case Block(body, type): {

                if (body.length == 0)

                    return '${prefixFA(prefix)}$t$d <empty block>\n';

                var strParts = ['${prefixFA(prefix)}$t$d Block\n${getTree\_PARSER(type, prefix.copy(), level + 1, false)}'].concat([

                    for (i in 0...body.length - 1) getTree\_PARSER(body[i], pushIndex(prefix, level), level + 1, false)

                ]);

                strParts.push(getTree\_PARSER(body[body.length - 1], prefix.copy(), level + 1, true));

                return strParts.join("");

            }

            case PartArray(body): {

                if (body.length == 0)

                    return '${prefixFA(prefix)}$t$d <empty array>\n';

                var strParts = ['${prefixFA(prefix)}$t$d Part Array\n'].concat([

                    for (i in 0...body.length - 1) getTree\_PARSER(body[i], pushIndex(prefix, level), level + 1, false)

                ]);

                strParts.push(getTree\_PARSER(body[body.length - 1], prefix.copy(), level + 1, true));

                return strParts.join("");

            }

            case FunctionCall(name, params):

                {

                    var title = '${prefixFA(prefix)}$t$d Function Call\n';

                    title += getTree\_PARSER(name, pushIndex(prefix, level), level + 1, false);

                    title += getTree\_PARSER(params, prefix.copy(), level + 1, true);

                    return title;

                }

            case Return(value, type): {

                return '${prefixFA(prefix)}$t$d Return\n${getTree\_PARSER(value, prefix.copy(), level + 1, type == null)}${getTree\_PARSER(type, prefix.copy(), level + 1, true)}';

            }

            case PropertyAccess(name, property): {

                return '${prefixFA(prefix)}$t$d Property Access\n${getTree\_PARSER(name, pushIndex(prefix, level), level + 1, false)}${getTree\_PARSER(property, prefix.copy(), level + 1, true)}';

            }

        }

        return "";

    }

    /\*\*

        returns string representation of tree

        @param root The token to start from

        @param prefix An array of prefixes, to determine

        @param level The depth of the tree in the current recursion

        @param last Whether or not the current node is the last

        @return The string representation

    \*\*/

    @:noCompletion static function getTree\_INTERP(root:InterpTokens, prefix:Array<Int>, level:Int, last:Bool):String {

        l = level;

        var t = if (last) "└" else "├";

        var c = "├";

        var d = "───";

        if (root == null)

            return '';

        switch root {

            case SetLine(line): return '${prefixFA(prefix)}$t$d SetLine($line)\n';

            case SetModule(module): return '${prefixFA(prefix)}$t$d SetModule($module)\n';

            case SplitLine: return '${prefixFA(prefix)}$t$d SplitLine\n';

            case Number(num): return '${prefixFA(prefix)}$t$d ${num}\n';

            case Decimal(num): return '${prefixFA(prefix)}$t$d ${num}\n';

            case Characters(string): return '${prefixFA(prefix)}$t$d "${string}"\n';

            case Sign(sign): return '${prefixFA(prefix)}$t$d ${sign}\n';

            case NullValue: return '${prefixFA(prefix)}$t$d ${NullValue}\n';

            case TrueValue: return '${prefixFA(prefix)}$t$d ${TrueValue}\n';

            case FalseValue: return '${prefixFA(prefix)}$t$d ${FalseValue}\n';

            case Identifier(word): return '${prefixFA(prefix)}$t$d ${word}\n';

            case Documentation(doc): return '${prefixFA(prefix)}$t$d """${doc}"""\n';

            case ClassPointer(pointer): return '${prefixFA(prefix)}$t$d ClassPointer: ${pointer}\n';

            case HaxeExtern(func): return '${prefixFA(prefix)}$t$d <Haxe Extern>\n';

            case VariableDeclaration(name, type, doc):

                var title = '${prefixFA(prefix)}$t$d Variable Declaration\n';

                if (doc != null) title += getTree\_INTERP(doc, prefix.copy(), level + 1, false);

                title += getTree\_INTERP(name, prefix.copy(), level + 1, type == null);

                if (type != null) title += getTree\_INTERP(type, prefix.copy(), level + 1, true);

                return title;

            case FunctionDeclaration(name, params, type, doc):

                var title = '${prefixFA(prefix)}$t$d Function Declaration\n';

                if (doc != null) title += getTree\_INTERP(doc, prefix.copy(), level + 1, false);

                title += getTree\_INTERP(name, prefix.copy(), level + 1, false);

                title += getTree\_INTERP(params, prefix.copy(), level + 1, type == null);

                if (type != null) title += getTree\_INTERP(type, prefix.copy(), level + 1, true);

                return title;

            case ConditionCall(name, exp, body):

                var title = '${prefixFA(prefix)}$t$d Condition Call\n';

                title += getTree\_INTERP(name, prefix.copy(), level + 1, false);

                title += getTree\_INTERP(exp, pushIndex(prefix, level), level + 1, false);

                title += getTree\_INTERP(body, prefix.copy(), level + 1, true);

                return title;

            case FunctionCode(requiredParams, body):

                var title = '${prefixFA(prefix)}$t$d Function Code\n';

                title += getTree\_INTERP(Identifier(requiredParams.toString()), prefix.copy(), level + 1, false);

                title += getTree\_INTERP(body, prefix.copy(), level + 1, true);

                return title;

            case ConditionCode(callers):

                var title = '${prefixFA(prefix)}$t$d Condition Code\n';

                title += getTree\_INTERP(Characters(callers.toString()), prefix.copy(), level + 1, true);

                return title;

            case FunctionCall(name, params):

                var title = '${prefixFA(prefix)}$t$d Function Call\n';

                title += getTree\_INTERP(name, pushIndex(prefix, level), level + 1, false);

                title += getTree\_INTERP(params, prefix.copy(), level + 1, true);

                return title;

            case FunctionReturn(value, type):

                var title = '${prefixFA(prefix)}$t$d Function Return\n';

                title += getTree\_INTERP(value, prefix.copy(), level + 1, type == null);

                if (type != null) title += getTree\_INTERP(type, prefix.copy(), level + 1, true);

                return title;

            case Write(assignees, value):

                var title = '${prefixFA(prefix)}$t$d Write\n';

                title += getTree\_INTERP(PartArray(assignees), pushIndex(prefix, level), level + 1, false);

                title += getTree\_INTERP(value, prefix.copy(), level + 1, true);

                return title;

            case TypeCast(value, type):

                var title = '${prefixFA(prefix)}$t$d Type Cast\n';

                title += getTree\_INTERP(value, prefix.copy(), level + 1, false);

                title += getTree\_INTERP(type, prefix.copy(), level + 1, true);

                return title;

            case Expression(parts, type):

                if (parts.length == 0)

                    return '${prefixFA(prefix)}$t$d <empty expression>\n';

                var strParts = ['${prefixFA(prefix)}$t$d Expression\n${getTree\_INTERP(type, prefix.copy(), level + 1, false)}'].concat([

                    for (i in 0...parts.length - 1) getTree\_INTERP(parts[i], pushIndex(prefix, level), level + 1, false)

                ]);

                strParts.push(getTree\_INTERP(parts[parts.length - 1], prefix.copy(), level + 1, true));

                return strParts.join("");

            case Block(body, type):

                if (body.length == 0)

                    return '${prefixFA(prefix)}$t$d <empty block>\n';

                var strParts = ['${prefixFA(prefix)}$t$d Block\n${getTree\_INTERP(type, prefix.copy(), level + 1, false)}'].concat([

                    for (i in 0...body.length - 1) getTree\_INTERP(body[i], pushIndex(prefix, level), level + 1, false)

                ]);

                strParts.push(getTree\_INTERP(body[body.length - 1], prefix.copy(), level + 1, true));

                return strParts.join("");

            case PartArray(parts):

                var title = '${prefixFA(prefix)}$t$d Part Array\n';

                for (part in parts) {

                    title += getTree\_INTERP(part, prefix.copy(), level + 1, part == parts[parts.length - 1]);

                }

                return title;

            case PropertyAccess(name, property):

                var title = '${prefixFA(prefix)}$t$d Property Access\n';

                title += getTree\_INTERP(name, prefix.copy(), level + 1, false);

                title += getTree\_INTERP(property, prefix.copy(), level + 1, true);

                return title;

            case Object(props, \_):

                var title = '${prefixFA(prefix)}$t$d Object\n';

                var i = 0;

                for (key => value in props) {

                    i++;

                    title += getTree\_INTERP(Identifier(key), prefix.copy(), level + 1, i == [for (x in props.keys()) x].length);

                    title += getTree\_INTERP(Characters(value.documentation), i == [for (x in props.keys()) x].length ? prefix.copy() : pushIndex(prefix, level), level + 2, false);

                    title += getTree\_INTERP(value.value, i == [for (x in props.keys()) x].length ? prefix.copy() : pushIndex(prefix, level), level + 2, true);

                }

                return title;

            case ErrorMessage(msg): return '${prefixFA(prefix)}$t$d ${root}\n';

        }

    }

    static var indent = "";

    /\*\*

        Converts an array of `ParserToken`s into their code form, with standard

        indenting & formatting

        @param code An array of `ParserToken`s

        @param token If you jeu need to stringify a single token, give this instead.

    \*\*/

    public static function stringifyParser(?code:Array<ParserTokens>, ?token:ParserTokens) {

        if (token != null) code = [token];

        var s = "";

        for (token in code) {

            switch token {

                case SetLine(line):s += '\n$indent'; continue;

                case SetModule(\_): continue; // Do Nothing, this is not syntax dependent.

                case SplitLine: {

                    if (s.charAt(s.length - 1).isSpace(0)) s = s.substring(0, s.length - 1);

                    s += ",";

                }

                case Variable(name, type): s += '${Little.keywords.VARIABLE\_DECLARATION} ${stringifyParser(name)}${if (type != null  && Interpreter.convert(type)[0].asJoinedStringPath() != Little.keywords.TYPE\_UNKNOWN) ' ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${stringifyParser(type)}' else ''}';

                case Function(name, params, type): s += '${Little.keywords.FUNCTION\_DECLARATION} ${stringifyParser(name)}(${stringifyParser(params).replace(" ,", ",")})${if (type != null  && Interpreter.convert(type)[0].asJoinedStringPath() != Little.keywords.TYPE\_UNKNOWN) ' ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${stringifyParser(type)}' else ''}';

                case ConditionCall(name, exp, body): s += '${stringifyParser(name)} (${stringifyParser(exp)}) ${stringifyParser(body)}';

                case Read(name): s += stringifyParser(name);

                case Write(assignees, value): s += assignees.concat([value]).map(t -> stringifyParser(t)).join(" = ").replace("  =", " =");

                case Identifier(word): s += word;

                case TypeDeclaration(value, type): s += '${stringifyParser(value)} ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${stringifyParser(type)}';

                case FunctionCall(name, params): s += '${stringifyParser(name)}(${stringifyParser(params).replace(" ,", ",")})'.replaceIfLast(" )", ")");

                case Return(value, type): s += '${Little.keywords.FUNCTION\_RETURN} ${stringifyParser(value)}';

                case Expression(parts, type): s += stringifyParser(parts);

                case Block(body, type): {

                    indent += "\t";

                    if (body[0].is(SET\_MODULE)) body.shift();

                    if (body[0].is(SET\_LINE)) body.shift();

                    s += '{${stringifyParser(body)}} ${if (type != null && Interpreter.convert(type)[0].asJoinedStringPath() != Little.keywords.TYPE\_UNKNOWN) '${Little.keywords.TYPE\_DECL\_OR\_CAST} ${stringifyParser(type)}' else ''}';

                    s = s.replaceLast('\t} ', "}");

                    indent = indent.replaceLast("\t", "");

                }

                case PartArray(parts): s += stringifyParser(parts);

                case PropertyAccess(name, property): s += '${stringifyParser(name)}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${stringifyParser(property)}';

                case Sign(sign): s += sign;

                case Number(num): s += num;

                case Decimal(num): s += num;

                case Characters(string): s += '"' + string + '"';

                case Documentation(doc): s += '"""' + doc + '"""';

                case ErrorMessage(msg): continue;

                case NullValue: s += Little.keywords.NULL\_VALUE;

                case TrueValue: s += Little.keywords.TRUE\_VALUE;

                case FalseValue: s += Little.keywords.FALSE\_VALUE;

                case Custom(\_, \_): throw 'Custom tokens cannot be stringified, as they dont represent any output syntax (found $token)';

            }

            s += " ";

        }

        return s.ltrim().replaceLast(" ", "");

    }

    /\*\*

        Converts an array of `InterpTokens`s into their code form, with standard

        indenting & formatting

        @param code An array of `InterpToken`s

        @param token If you jeu need to stringify a single token, give this instead.

    \*\*/

    public static function stringifyInterpreter(?code:Array<InterpTokens>, ?token:InterpTokens) {

        if (token != null) code = [token];

        var s = "";

        var currentLine = -1;

        for (token in code) {

            switch token {

                case SetLine(line): {

                    s += '\n$indent';

                    continue;

                }

                case SetModule(module): continue; // Do Nothing.

                case SplitLine: {

                    if (s.charAt(s.length - 1).isSpace(0)) s = s.substring(0, s.length - 1);

                    s += ",";

                }

                case VariableDeclaration(name, type, doc): s += '${Little.keywords.VARIABLE\_DECLARATION} ${stringifyInterpreter(name)}${if (type != null  && type.asJoinedStringPath() != Little.keywords.TYPE\_UNKNOWN) ' ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${stringifyInterpreter(type)}' else ''}';

                case FunctionDeclaration(name, params, type, doc): s += '${Little.keywords.FUNCTION\_DECLARATION} ${stringifyInterpreter(name)}(${stringifyInterpreter(params).replace(' ,', ',')})${if (type != null && type.asJoinedStringPath() != Little.keywords.TYPE\_UNKNOWN) ' ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${stringifyInterpreter(type)}' else ''}';

                case Write(assignees, value): s += assignees.concat([value]).map(t -> stringifyInterpreter(t)).join(" = ").replace("  =", " =");

                case Identifier(word): s += word;

                case TypeCast(value, type): s += '${stringifyInterpreter(value)} ${Little.keywords.TYPE\_DECL\_OR\_CAST} ${stringifyInterpreter(type)}';

                case FunctionCall(name, params): s += '${stringifyInterpreter(name)}(${stringifyInterpreter(params).replace(' ,', ',')})'.replaceIfLast(" )", ")");

                case ConditionCall(name, exp, body): s += '${stringifyInterpreter(name)} (${stringifyInterpreter(exp)}) ${stringifyInterpreter(body)}';

                case FunctionReturn(value, type): s += '${Little.keywords.FUNCTION\_RETURN} ${stringifyInterpreter(value)}';

                case Expression(parts, type): s += stringifyInterpreter(parts);

                case Block(body, type):

                    indent += "\t";

                    if (body[0].is(SET\_MODULE)) body.shift();

                    if (body[0].is(SET\_LINE)) body.shift();

                    s += '{${stringifyInterpreter(body)}} ${if (type != null && type.asJoinedStringPath() != Little.keywords.TYPE\_UNKNOWN) '${Little.keywords.TYPE\_DECL\_OR\_CAST} ${stringifyInterpreter(type)}' else ''}';

                    s = s.replaceLast('\t}', "}");

                    indent = indent.replaceLast("\t", "");

                case PartArray(parts): s += stringifyInterpreter(parts);

                case PropertyAccess(name, property): s += '${stringifyInterpreter(name)}${Little.keywords.PROPERTY\_ACCESS\_SIGN}${stringifyInterpreter(property)}';

                case Sign(sign): s += sign;

                case Number(num): s += num;

                case Decimal(num): s += num;

                case Characters(string): s += '"' + string + '"';

                case Documentation(doc): s += '"""' + doc + '"""';

                case ErrorMessage(msg):

                case NullValue: s += Little.keywords.NULL\_VALUE;

                case TrueValue: s += Little.keywords.TRUE\_VALUE;

                case FalseValue: s += Little.keywords.FALSE\_VALUE;

                case ClassPointer(pointer): s += Little.memory != null ? Little.memory.getTypeName(pointer) : throw "No memory for ClassPointer token " + pointer;

                case FunctionCode(\_, body) : s += stringifyInterpreter(body);

                case \_: throw 'Stringifying token $token does not make sense, as it is represented by other tokens on parse time, and thus cannot appear in a non-manipulated InterpTokens AST';

            }

            s += " ";

        }

        return s.ltrim().replaceLast(" ", "");

    }

    /\*\*

        Pretty prints the operator priority. Little.memory.operators are registered through plugins.

        @param priority The priority map to print.

    \*\*/

    public static function prettyPrintOperatorPriority(priority:Map<Int, Array<{sign:String, side:OperatorType}>>) {

        var sortedKeys = [for (x in priority.keys()) x];

        ArraySort.sort(sortedKeys, (x, y) -> x - y);

        var string = "";

        for (key in sortedKeys) {

            string += '$key: (';

            for (obj in priority[key]) {

                if (obj.side == LHS\_RHS) string += '\_${obj.sign}\_';

                else if (obj.side == LHS\_ONLY) string += '\_${obj.sign}';

                else if (obj.side == RHS\_ONLY) string += '${obj.sign}\_';

                string += ', ';

            }

            string = string.replaceLast(', ', ')') + '\n';

        }

        return string;

    }

}