

Little

שפט תכנות, "קומפיילר" ואינטראפרט.



שם תלמיד: שחר מרכוס
תעודת זהות: 329545933
שם מורה: שלמה ווקנין
שמות חלופה: Multilang-Code
Minilang
תאריך הגשה: 31.5.2024



ישיבת אמית כפר גנים
מאמינים בה, מאמינים בך

תוכן עניינים

1.....	תוכן עניינים
3.....	הערות לקורא
4.....	מבוא
4.....	ייזום
4.....	תיאור ראשוני
4.....	הגדרת הלקוח
5.....	יעדים ומטרות
5.....	בעיות, תועלות וחסכנות
7.....	פתרונות קיימים
7.....	טכנולוגיה
8.....	הגבלות
10.....	תחומים ותמיכה
11.....	תיאור המערכת
11.....	از, מה בדיק הפרויקט עושה?
23.....	יכולות
24.....	סוויטת בדיקות
26.....	לוח זמנים
29.....	סיכונים
31.....	תיאור תחום הידע
31.....	יכולות
31.....	סוג לפקוח: מפתח אפליקציות
37.....	סוג לפקוח: לומד תכנות, מתכנת Little
42.....	מבנה הפרויקט
42.....	ארQUITקטורה
42.....	רכיבים שונים, והקשרים ביניהם
46.....	טכנולוגיה
46.....	שפה תכנות
46.....	תחומי עניין



47	זרימת מידע במערכת
47	הקומפיילר
48	האינטרפרטר
49	הזיכרון
50	אלגוריתמים מרכזיים
50	קישור בין משתנים למקומות בזכרון
51	חישוב והערכת ביטויים
52 Hashing
54	סביבת עבודה
54	פיתוח
54	בדיקה
55	ימוש הפרויקט
55	חלוקת ומחלקות
55	חבילה – little.lexer
56	חבילה – little.parser
58	חבילה – little.interpreter
62	חבילה – little.interpreter.memory
71	חבילה – little.tools
76	מחלקות נוספת
80	קוד "יפה" ותוכנות אלגוריתמים
80	קריאה לפונקציות אקסטרניות
80	Shunting Yard – השראה
87	עיצוב ה-Referrer
90	יצירת סוגים אקסטרניים
99	בדיקות ותוצאות
99	בדיקות ביחידות (Unit Testing)
102	בדיקות ידניות
104	מדריך למשתמש
104	עץ קבצים
106	התקנה
106	למפתחי תוכנה
108	מתכנתים ב-Little
110	רפלקציה
110	עבודה על הפרויקט

111	תובנות
112	הסתכלות
112	אחרונית
112	וקדימה
112	תודות
113	ביבליוגרפיה
114	נספחים
114	הקדמה קצרה
115	קוד הפרויקט

הערות לקורס

- כמו בהרבה נושאים הקשורים למדעי המחשב, עברית לא תמיד מציעה את כל המילים שצרים...
הנה מיליון מושגים קצר, שאשתמש בהם במהלך הפרויקט:
 - טוקן - Token
 - לקסר - Lexer
 - פרסר - Parser
 - קומפיילר - Compiler
 - אינטראפרט – Interpreter
 - טרנספайлר – Transpiler, Source-to-source compiler
 - מכונה וירטואלית – Virtual Machine VM
 - עץ syntax – Abstract Syntax Tree ,AST
 - סוג מידע אלגברי, ערךenum המכיל מידע נוסף – ADT
 - Types
 - מידע על סוגים בזמן ריצה – RTTI ,Runtime Type Information
- ישנים חלקים בפרויקט המכילים שורות ארוכות במילוי. מומלץ לקרוא את הספר במצב עמוד אינטראקטיבי, באמצעות לחיצה על שנמצא מצד ימין/שמאל למטה של חלון הוורד, תלוי בשפה של וורד (מצד הימני לשפות כמו אנגלית, ובצד השמאלי לשפות כמו עברית)
- קוד הפרויקט שבסוף הספר ארוך במילוי, וגם כאשר וורד נמצא במצב אינטראקטיבי, יכולות להיות בעיות קריאה בגל שורט עשויה word-wrapping לשורות ארוכות באופן אוטומטי. כדי להימנע מאי נוחות, אני ממליץ לקרוא את קוד בעמוד הגיטהאב שלן:

<https://github.com/ShaharMS/Little/tree/branch/functional-programming/src>

מבוא

יזום

תיאור ראשוני

הפרויקט הוא שפט תכנות חדשה שהמצאת, שמה "Little". הפרויקט כתוב בשפת התכנות Haxe, ומאפשר הריצה של קוד הכתוב ב-Little מצד המשתמש, ובמיוחד מאפשר למפתח המשמש בפרויקט לאלמנטים באפליקציה שלו לעקב בקשות אחרי כל אירוע שקרה, ולהוסיף לשפה פונקציונליות צד-שלישי, על מנת לשפר פרודוקטיביות באותה אפליקציה.

קיבלת את ההשראה להcin פרויקט בסדר גודל כזה משני אלמנטים בספריית תכנות של, Texter :

- Parser של Markdown, שאם נותן אופציית ייזואלייזציה של החווית Markdown
- אלגוריתם החלטת כיוון וסידור טקסט למחוזות המכילים טקסטים שאמורים לזרום לכיוונים שונים (לדוגמה: אנגלית ועברית, צרפתית וערבית), כמו שורות/פסקיות שונות.

שני האלמנטים עירבו יצירה והחלטה על Tokens שייצגו את המידע, ומיפויו לצורה Tokens, על מנת להציג את התוצאה הרציה (טקסט שמצוג נכון ומסמן בסטייל Markdown, מהמקרים שהוזכרו קודם). ככלמת שקומפיאיר/introprator של שפט תכנות זה, בסופו של יומם, אותו הדבר, רק בסדר גודל יותר רצני - התענינתי, והתחלתי לנסות לתכנת שפה ממשי ...

מש בהתחלה, חשבתי שזה יהיה יחסית פשוט – הנחתי שהתהליך יהיה ישיר בערך כמו הדברים שעשית בעבר, אבל תוך כדי עשייה, הבנתי שאין הולך להיתקל ביחסית הרבה מכשולים. העיקריים היו:

- זריקת שגיאות מובנות ומתאימות לחבר
- מערכת סוגים – מה עדיף – מערכת חזקה או מערכת חלה?
- ניהול זיכרון – באלו מבנים משתמש על מנת לאחסן זיכרון?
- וכמוון: איך בכלל מתמודדים עם כמות עצה גדולה של נתונים, בלי לחת הרבה זמן?

הגדרת הלקוח

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

- **סוג ראשון: לקוחות רגילים** – אנשים שרק רוצים למדוד את השפה ולתכנתה. בשbillם, ישנה חלקה באתר של המאפשרת גישה לרוב היכולות של הפרויקט – שינוי מילות מפתח, הריצה של קוד, ועודיו אפשר לראות את ה-AST של הקוד שהקלדתו בוצרה שנעימה לעיניים.
- **סוג שני: מפתחים** – מפתחים שרצו לצלול יכולות פרוגרמטיות בתוכנה שלהם. בשbill להקל עליהם, הפרויקט מעוצב לפי תבנית Facade, על מנת לפשט את האינטגרציה עם התוכנה, וגם לאפשר אותה באמצעות פחות "נפח" של קוד. מעבר לזה, הפונקציות העיקריות בפרויקט, מתייעדות בוצרה אקסטנסיבית ומסבירות בוצרה טוביה מה הן בדיקן עשוות, ועודיו כוללות דוגמאות המראות מה קורה עם קליטים מסוימים.

יעדים ומטרות

מההטבלה, החלטת שהפרויקט יכוון לשתי מטרות, יחסית ספציפיות, אך שימושיות:

- **ראשונה:** יצרת שפת תכנות שקל ללמידה וגם קל לשנות. שפה זאת יכולה בקלה להנגיש לימוד תכנות לילדים, ובמיוחד לילדים שלא בהכרח יודעים אנגלית. כאן לפרויקט של ייש יתרון מובהק על פני אופציית אחרות זמיןנות: הפרויקט עוזב תוך כדי התחשבות בריבוי שפות, כך שכל keyword או אלמנט Standard Library יכול להיות מוחלף על ידי כל מילה אחרת בכל שפה אחרת, דבר שמאפשר לתכנות לא רק בשפה האנגלית, אלא גם בשפות מדוברות אחרות, כמו עברית וערבית.
- **שנייה:** שימוש ככלי פורודוקטיביות בתוך תוכנות קיימות – כמו שהתוכנה Excel מאפשרת לשנות ערכים של משਬצות בעזרת קוד הכתוב ב-Visual Basic Analysis, מכיה תוכנות המשמשות בLittle Library יכולו לספק דרכי אינטראקטיבית עם התוכנה באמצעות קוד הכתוב בLittle. גם כאן לפרויקט יש יתרון ברור, שכן לא רק שמצד הלקווח, כתיבת קוד Little זה דבר קל בגלל syntax הפשט והעקביו של השפה, גם המפתח נהנה, מכיוון שחלקמשמעותי מהפרויקט מתרכז בהקלה על הוספת אלמנטים חיצוניים לשפה, וזה יהיה מאד קל ומהיר לשלב את הפרויקט בתוכנה של המפתח, ויחסור ממנו CAB ראש.

בעיות, תוצאות וחסכנות

כשהתחלתי עם הפרויקט לפני השנה, היה לי רק רצון אחד שהפרויקט יملא, שלא קשור בהכרח לתפקידו שלו – **שייה אפשר לתכנות בו בכל שפת אם שהיא**. הסיבה לרצון זהה, קשורה לapter-משחק שشيخנו בו בבית הספר היסודי, שקרו לו Code Monkey, שבו המטרה היא להציג קוף לעבר בננות באמצעות קוד, בכמה שפחות שורות. בעיה אחת הייתה ל' עם המשחק הזה – הקוד היה באנגלית, ואני הייתה הרבה יותר קטן, ולא דעתן אנגלית טוב – זה מאד הוביל אותי, כילקח לי מדי הרבה זמן לזכור מה כל דבר עושה, וככל שהצטברו הבעיות והתכונות שהיא אפשר להשתמש בהם, המשחק הסתבר יותר ויותר...

על מנת למש את הרצון שלי עם הפרויקט הזה, עצבתי אותו בדרך מסוימת, ונתתי גישה למגוון מערכות שונות, ולא רק שמשתי את הרצון שלי – לשמחתי, עשיתן הרבה מעבר...

מערכות:

- **אוסף מילוט מפתח** – על מנת להקל על המפתח, ניתנת גישה לכל "מילות המפתח" (Standard Library, מילים שמורות), והמפתח יכול לשנות מילים ספציפיות, או להחליף בין סיטים של מילים, כדי לשנות את ורבאליות הקוד, או השפה המדוברת בו היא כתובה. מערכת זו ממש הייתה מכוננת למימוש הרצון הראשוני שלי, של הנגשת תוכנות לילדים שלאו דוקא יודעים אנגלית. דוגמה של המערכת בפעולה:

מילות מפתח ערוכות לעברית:

```
define num = 13
print({
    num = num * 5
    Characters.fromCharCode(e)
}) """prints: "A" """

```

```
הגדיר מס = 13
הדפס ({
מס = מס * 5
אותיות.מקוד_אות(מס)
" """AMDPEIS: " { } """

```

- **מודולארי Parser** – על מנת להוסיף על יכולות המערכת הראשונה, Parser של השפה מאפשר "זרקה" ישירה של שלבי בניית AST, ואפיו נוון למפתח להציג Tokens חדשים באופן דינמי. המערכת מאפשרת את זה על מנת לאפשר תמייה בכל סוג של syntax מיוחד, דבר שאי אפשר לעשות ברוב שפות התכונות האחרות, שמרכיבות את כל AST שלו לנו פנוי מהוות אותו macros, ולכן, ניתן להוסיף לשפה רק syntax שנחשב חוקי (קריאה על ידי ה-Parser של השפה) מראש. הסיבה לכך היא, שsyntax שנחשב שגוי בשפה, אך חוקי בmacro-וינו, יזרוק שגיאה בקריאה הראשונית של הקוד אל תוך AST, ולא יגיע לmacro שלנו. דוגמה לשפה קיימת (Haxe):

גישת מערך מרובה expressions, כמו `[4 + 5, array[3, 5], ...]`, לשפה שמצויה לו identifier/expression ייחיד בגישה למערך, שיראה כך: `[2, array[6, *, ...]]`. תזרוק שגיאה בסגנון `[*, ...]`Unexpected Identifier את ה-Parser, לפניו מופיעיל את המacro שלו, שיופיע, לדוגמה, את `[5 + 4, array[3, 4, ...]]`. על מנת להימנע ממקרים כאלה, ה-Parser של השפה מאפשר הכנסת macros ממש אל תוך פונקציית parsing ולא אחרת, ובעזרה יכולת זו אפשר לדאוג שהmacro פועל לפני Parser זורק את השגיאה, והmacro יעבד כמצופה.

- **אגף להוספה Externs** – על מנת להקל על המפתח בשילוב הפרויקט בתוכנה שלו, ישנו אגף שלם הנועד רק להוספה "Plugins" שונות לשפה במגוון סוגים, ובמגוון מקדים - משתנים ופונקציות רגילות, לסוגים מיוחדים ושדות גלובליים. האגף הזה מורכב ממחלקה אחת בשם `Plugins`, שמטרתה לאפשר הוספה אלמנטים לשפה בעזרת קוד הכתוב בהaxe, ופונקציה בשם `loadModule()`, שמאפשרת להריץ קוד הכתוב בLittle, או ממש לפניו שהקוד של המשתמש רץ. דוגמה:

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

פתרונות קיימים

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

פרויקט (Little)	VBA (Visual Basic)	Hscript (Haxe)	שפה תכנות
✓	✓	✓	הכרזת משתנים/פעולות
*✗✓	✓	✓	הכרזת סוגים
✓	✓	**✓	ЛОЯЛОСТ, תנאים
✓	✗	✓	Compilation Macros
✓	✗	✗	גישה לאירועי הרצה (יצירת משתנה, משתנה נכתבה...)
✓	✗	✗	שינוי מילימ שמרוות
✓	✗	✓	Cross Platform

* לא ניתן ליצור סוגים דרך Kod Little, אך אפשר להכניס סוגים מבחן, מצד המפתח

** אין switch pattern matching ו variable capturing

טכנולוגיה

הפרויקט תוכנת בשפה ייחסית לא מוכרת, ששמה Haxe. היא שפת תכנות שייצר אחד המתכנותים של Little, ActionScript, שמטרתה לאפשר פיתוח אפליקציות למגוון מטרות ופלטפורמות, באמצעות טרנספְּלִיצָה (תרגום קוד של שפה אחת לקוד של שפה אחרת). בהתחלה, מטרת השפה הייתה לאפשר כתיבה של גם ל�� ו גם שרת באותה שפה, בעזרה השתמשות בספריות סטנדרטיות של השפות שאלייהם הקוד מתקמל. בשביל זה, נתמכה בתחילת הפיתוח רק 3 מטרות שאליין היה אפשר לקמפל:

- JavaScript, בשבייל אטרים וקוד בראשת
- ActionScript 3, בשבייל משחקים
- Neko, מכונה ווירטואלית "תוצרת בית", בשבייל תוכנות Seine.

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 נויספו, ווגנון השפה השתנה: במקום להמשיך להדමות לשפות כמו Java או ActionScript3, ובכך להקל את המעבר מהשפות האלה ל-Haxe, נויספו תוכנות נוספות ומיעדרות, על מנת להבהיר ולהקל על עיבוד מידע. בפרויקט זהה יצא לי להשתמש בשתי תוכנות עיבוד המידע העיקרי:

- Algebraic Data Types**: ב-Haxe, המונח הקלאס מורחב על מנת לתמוך

בסוג קצט יותר מתחכם של איבר, שמאפשר העברת פרמטרים שאוטו איבר מבקש. זה מאד שימושי כשרוצים לבנות עץ – בפרויקט השימושתי בתוכנה 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)
    )
);
```

- **תנאי switch הורחבו על מנת לאפשר חילוץ והתקנת **Pattern Matching****
- מידע הנמצא בתוך Algebraic Data Types, כמו מספרים, מחרוזות, מערכיים, או ADTs אחרים. ניתן לעשות זאת על ידי הצבת ערכים, העמדת תנאים מיוחדים, או הוצאהם 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`:
}
```

הgelot

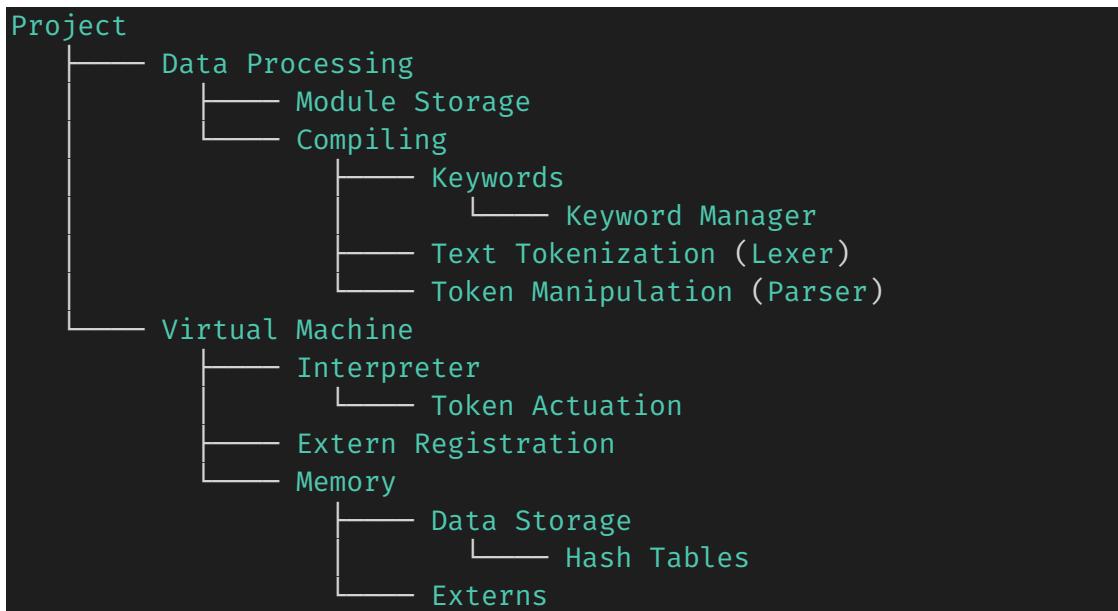
כפי שהוזכר קודם, אחת מהתכונות העיקריות של Haxe היא יכולת של השפה לעשוט טרנספְּולציה להרבה מטרות אחרות – גם שפות תכנות, וגם מכונות וירטואליות. על פניו נראה שזה יתרון ולא הגבלה, אבל זה לא בא בחינם, ויש **קאטץ'** – הספרייה הסטנדרטית קטנה ומצווצמת, וחיבת להעתאים בפונקציות המוצעות למקרה קיימכ' בכל המטרות האחרות. בפרויקט שלו, זה גורר בעיות הקשורות בכך ורק לסוגי מידע מיוחדים. אמונה מספר מקירים:

- **מערך ביתים** – אין דרך ליצור מערכים של אלמנטים שגודלם קטן מביט, מכיוון שלא כל המטרות תומכות ביצירה של מערך צזה (לדוגמה: PHP, Lua, ActionScript). נאלצתי להשתמש במערך של ביטים לזריכן (ByteArray), ולהשתמש בכל בית בתור תא יכול להכיל אלמנט אחד. בתיאוריה היחסי יכול להשתמש במערך רגיל ולאחרן את המידע בתוך הביטים של Int או Int64, אבל אז כבר אין סיבה להיות שמן עם זיכרון, שכן מערך רגיל מומר לרשימה מקוشرת או שילוב של רשימה ומערך בהרבה מהמטרות (Java: Array -> ArrayList, C++: Array -> vector, Python: Array -> list)

- **כמויות אלמנטים במערך** – לא כל המטרות שאקס מחייבת תומכות במערכות שאורכם עולה על 2.147 מיליארד אלמנטים, ומכיון שלא יכולתי להשתמש במערך שגודלו האלמנט שלו עולה על בית אחד בשביל זיכרון, הוגבלתי לזכרון בגודל מקסימלי של 2.147 גיביט.
 - **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 מ-8 פרמטרי סוג, אם אפשרותה המרה מהם ואלהם.
 - **Tables HashTables** – Haxe כן מציעה HashTable, אבל לא כל מטרה מאפשרת גישה לבתים ומקום האחסון של HashTable, ולכן הייתה צריך לתוכנת עצמי HashTable כדי שאוכל לגשת לבתים שהוא מורכב מהם, ולאחסן אותם באותו מערך זיכרון שדיברתי עליו קודם.
- מעבר להגבלות הקשורות לבניה השפה, נתקلت גם בעיות שנגרמו מחוסר הפופולריות של השפה:
- **Hashing** – כשהתחלתי תכנת את HashTable, גיליתי שאין הרבה ספירות שבכל מציאות את האלגוריתמים המתאימים (SipHash לדוגמה). לפרויקט זהה זה פחות שינה, כי לא בהכרח חיפשתי אלגוריתם Hashing שמתארתו להציגן, אך עדין יצא שהשתמשתי באלגוריתם שפותח חסין להתגשות מההויה משתמש בו בשפות אחרות (כן יש לציין שה HashTable של Haxe משתמש בsipHash, אבל זה לא בכוונה, אלא בגלל שהמטרות שאיליהן Haxe עשו טרנספילציה משתמשות בHashTable בשביל HashTable שלהן).
 - **ספירות עזר** – אין הרבה ספירות שמציאות את התכונות וסוגי המידע שאינו צריך, או לפחות הן לא עושים זאת בצורה יעללה מספיק. לכן, השתמשתי בפרויקט זהה בספריה אחרת שככetta שאינה קשורה לפרויקט בשם Vision (ספריית CV, מכילה הרבה סוגים מידע). על אף שהליך גדול מהספריה הצעת היה קיים לפני שהשתמשתי בו לפרויקט, שמתה לב שאני מדי פעם הוספתי פונקציות או תכונות שהייתי צריך בפרויקט זהה. בשפה יותר פופולרית, נראה בכלל לא הייתה נתקל בעיה דומה, ועוד יותר לא הייתה צריכה להשתמש בספריה משלוי.

תחומים ותמייה

בגدول, הפרויקט בנוי על ועוסק בעיבוד מידע באופן מסויבי. באופן יותר מדויק, הוא נכנס לענף של טוקנייזציה מידע המובא בצורה טקסט, והפעלת מניפולציות על אותם טוקנים, ובנוסף גם שימוש בהםים כאמצעי שיכל להפעיל מעין מכונה וירטואלית. אותה מכונה וירטואלית מחברת בין התוכן שאוטם טוקנים מייצגים, לפקודות שעלה המחשב לבצע. חלק מהפקודות מסופק מראש, ומה שלא מסופק מראש, מפתח שמשתמש בפרויקט יכול להויסיפ בשערת אגן מסויים בפרויקט. בשביל נוחות, אציג את הפקה אחרונה עצ, שנוצר בעזרת פונקציה שנלקחה מהפרויקט (PrettyPrintem, אחרי אדרטציה):



תיאור המערכת

از, מה בדיק הפרויקט עשה?

בגדי מאוד, הפרויקט הוא ה"קומפיילר" והמכונה ווירטואלית הרשミת של השפה שיצרתי - Little. הפרויקט גם מכיל שני אימפלמנטציות של לקוחות. אפרט, והרבה:

- Little – שפת התכנות עליה מבוסס הפרויקט. מדובר בשפה שיצרתי על מנת להציג תוכנות לילדים קטנים. השפה (יחד עם הפרויקט זהה) נוצרה לפני קצר יותר משנתים, והחליפה מספר שמות לפני שנחתה על השם Little. לשפה עוד לא היה שם.
- [ב23 באפריל, 2022](#), הפרויקט נוצר, ונקרא Multilang-Coder. לשפה עוד לא היה שם.
- [ב25 באפריל, 2022](#), הפרויקט והשפה שינו את שם ל"Minilang". באותו commit הקשור, גם כתבתי חלק מair שרציתי שהשפה תעבוד. התחלתי עם הקוד, אבל הייתה רחוק מהמטרה שהוצאה שם. באותו הזמן גם, היה הניסיון הראשון לבנייה וטכנולוגית הפרויקט, עליהם אפרט בהמשך. שם השפה עוד לא הוחלט באופן סופי ושם החיבור בה הולך הקוד נקרא פשטוט "language".
- [ב26 באפריל, 2022](#) לא היה שינוי לשם, אך הסרתי את החלק שכתבתי בו איך השפה תעבוד, והוספתי את הספציפיקציה הראשונה שהכילה נראות ומבנה הקוד בשפה. גם על זה אפרט בהמשך.
- [ב30 באפריל, 2022](#) הפרויקט והשפה שינו את שם העדכני – "Little". גם כאן הספציפיקציה השתנה קצר.

בזמן שהחליטתי השם קرتה יחסית מהר ובשלב יחסית מוקדם של הפרויקט, החלטה על ספציפיקציה לקחה הרבה יותר זמן – כמו בהחלטה על השם, היו הרבה שינויים בהתחילה, אך כאן המקרה הוא אחר – תוך כדי פיתוח, הבנתי שיש תכונות שאני רוצה להשאיר, ככל שאני רוצה להוריד, וככל 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](#), נוספה הספציפיקציה הראשונה לשפה. יש לה הרבה אלמנטים שדומים לשפה שמוצגת בפרויקט: שימוש ב-`define` וב-`action` על מנת להזכיר על משתנה ופונקציה, לדוגמה. עדין היה דמיון ממשמעותי לאקס, שכן אחת המטרות הראשונות (שנפלו בשלב מאוחר יותר) היו בניית טרנספילר לאקס. לא הוספתי אף מדריך בשלב זהה, אך כן הוספתי הסבר על חלק ממילוט המפתח שהוא 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)
}

```

- [ב3 באפריל, 2022](#) שוב השתantha הספציפיקציה, אך לאנוספו תכונות חדשות: רק הוסיף רעיון לsyntax שיטאים 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](#) נוסף הרעיון הראשוני המשמעותי שstrand (כמעט לגמרי) את מבחן הזמן, ונכנס לתוכר הסופי – לולאת ה-for. הספציפיקציה מציעה syntax-alternative, שלא משלב אופרטורים/סימנים (לדוגמא: ..., []), ומחליף אותו

במילים `from`, `to`. syntax זה עוצב גם כדי להשאיר מקום לולאת `for` שעוברת על מרכיבים, על אף שלא היה תכונן מיד לזה באותו זמן. שאר הקוד בספציפיציה זהה לה Shmuelha, אעתיק את השוני לכאן:

```
for name from 0 to 9 {
    print(i)
}
```

- שוב בג' במא, הפעם חשבתי על לכת על סטיל python כל הדרך בכתיבה 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
```

- עכשו מתחילה לkapoz בזמן: [ב91 בינואר 2023](#), קרו מספר דברים יחסית גודלים:
 - הוסרה הספציפיציה לייצור סוגים, מתוך רצון להציב מערכת עובדת לפני שמתעסקים עם ייצור סוגים.
 - האופרטור המשמש להכרזת סוג על ערך/משתנה הוחלף מאופרטור `ל-` `type`: `ל-type of`. זאת במטרה להיפרד קצת מ-Haxe, ולפשט את השפה.
 - נוסף האלמנט `every` לולאות `for`, המאפשר קפיצות בין ערכים. (לדוגמה, `from 0 to 5 every 2` יעבור על הערכים `4, 2, 0`)

הספציפיציה 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 [ה16 במרץ 2023](#), מחקתי את הספציפיקציה שנכתבה עד אז,
- [ב29 בינואר, 2023](#), מחקתי את החלק בREADME שהעד על הספציפיקציה, לעומת שלושה דברים שהציבתי לעצמי לצורך בעtid:
 - קובץ טקסט נפרד המכיל את הפירמות, הsyntax וה"זרימה" של קוד השפה
 - סוויתת בדיקות, שגם היא تستמך על תכונות השפה ותעד על המבנה שלה
 - אזכור של תכונות ייחודיות בREADME שלuproject README
- [ב16 במרץ 2023](#) הוגשמה המטרה השלישי, יצרתי חלק בקובץ README שנועד לאזכור תכונות. התוכנה הראשונה שנוספה לשם היא אחת מתכונות הבסיס של Little, וזה העיקרון **Everything can be a code block**. אסביר עליון בהמשך. הדוגמה שסופקה:

```
x = {define y = 0; y += 5; (6^2 * y)} //180
```

- [ב9 באפריל, 2023](#) נוספה לREADME התכונה השנייה, שערכונה: Consistency Is Key. גם עליה אפשר בהמשך, על אף שהיא יחסית ברורה: README מוגמת 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](#) נוספו *המן* דוגמאות ומדריכים לקובץ README – README מתכונות של השפה, לתכונות של קורא הקוד, ותיעוד של כמה לולאות, חדשות וישנות:
 - לולאת **while** – מבצעת את גוף הלולאה עד שהתנאי מוחזיר **true**
 - תנאי **if** מבצע את גוף התנאי עם התנאי הראשוני מוחזיר **true**
 - לולאת **for** שינתה את המילה **every** למילה **umpum**
 - תנאי **after** – מבצע את גוף התנאי, מיד לאחר שהמשתנה שמאזכר בתנאי הראשוני משתנה, וגורם לתנאי הראשוני להחזיר **true**
 - תנאי **whenever** – שילוב של **after**-**while** – כל פעם שאחרי עדכון המשתנה המוזכר, התנאי הראשוני מוחזיראמת, הגוף שבתוך התנאי מורץ.

לא אוכל לשים כאן הכל בגל גודל הטקסט (בין 2 ל-3 עמודים), אז [הנה קישור](#)
[לקובץ בGITהאב באוטו commit](#)

- **ב6 בנובמבר, 2023**, נוספה עוד דוגמה לתוכנה בשפה – דוקומנטציה. היא קיבלה אימפלמנטציה קונקרטית באותה הזמן, בעזרה סדרת הסימנים `'''`.
- גם נוספה הרחבה לאחת הדוגמאות, שככליה יוצרת משתנה עם שם שמחולט בזמן ריצעה. מ עבר זהה, היו תיוקונים 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**, נוצרה סוויתת הבדיקה הראשונה – היא בה מספר בדיקות, והיא הדפסה לכל בדיקה האם היא עברה, ואם לא מה קרה כשהיא לא עברה. ביצירה של הבודק הוסpty 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.](#)
- **ב16 בנובמבר, 2023 וב22 בנובמבר, 2023**, נוצר קובץ הספציפיקציה "רשמי", שני חלקים – פשטוט קובץ טקסט המכיל קוד מתוכנת בשפה Little. הוא הכיל גם ספציפיקציה להכרזה של סוגים, לולאות ותנאים על מנת לנסות לקבוע סטנדרט איך השפה אמורהBehave 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})
```

- מיצרת קובץ הספציפית, לא קרו שינוי דרמטיים במבנה הקוד. השינוי הגדול האחרון קרה בתאריך 17 במרץ, 2024, שבו נוספה הספציפית להכרזה על סימנים:

```
sign (left ^% right) = {
    return left + right ^ 2
}
print(^%.priority.type) //Number
```

עם הזמן, נוספו גם בדיקות לסווית הבדיקות, אך אין צורך להזכיר אותן (לפחות כרגע), שכן הן לא מכילות קוד שאמור "להיות דוגמה", אלא רק צה שנוועד לבדוק אם דברים עובדים כמו צפופה. רוב הבדיקות בשורה אחת על עף שמצופה לשים `line` (לדוגמה, `break`).

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

- דרך כתיבה:**
 - בלוק של קוד יכול להיות כמעט הכל:** שמות משתנים, לויאות, פונקציות ואפילו סימנים אפשריים הכרזה וגישה בזמן ריצה בעזרת בלוקים של קוד – בלוק של קוד הוא כמה מוסויימת של שורות, מאוכסנות בתוך סוגרים מסולסלים. בלוק יכול להציג ערך בעזרת מילת המפתח `return`. ניתן גם להשミט אותה, וזה הערך המוחזר יהיה `token` האחרון בבלוק הקוד.
 - כתיבת קוד איחידה:** כמו שאפשר, כל הדריכים לכתוב syntax שטחוטיהן דומות (הכרזות, גישות, קרייאות) כתובות באוותה דרך, אותה צורה, או לפחות אותו זרם: תמיד משתמשים ב= על מנת להציג 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 סוג המידע יעשן זאת? לכן, אפשר גם להשתמש באופרטורים וסוגים שונים. עיקנון זה שימושי לייצרת `aliases`, וגם יכול ליצור דברים מעשניים כמו:

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

(יוצא 180)

פיצ'רים

- הכרזות:** יש תמיינה בהכרזת משתנים ופונקציות, מכל סוג שהוא, באופן סטטי ודינמי:

- **שורות:** אין צורך לשים `:` בסופי שורות, אך אפשר להשתמש ב-`-` על מנת להכנס כמה שורות קוד בשורה אחת, זהה מפצל את השורה לחלקים.
- **גישות וקריאות:** אפשר לגשת לערכים של מזהים באמצעות השתמשות בשםם, השימוש בפונקציה `read`, או בלוק של קוד שמחזיר פועלות `read`. קריאה לפונקציות מתבצעת באותה צורה, עם הוספה `()` לאחריהם. הסוגרים יכולים להכיל פרמטרים, מפוצלים בעזרה שורה חדשה או `,`

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

(`"e` י יצא

- **ולואות ותנאים:** רוב הלולאות שזרימות בשפות אחרות זמיןנות גם `for`, `while`, `if`: Little. `for`, `while`, `if` יש גם שתי ולואות/תנאים חדשים: `after` – `after` מtbody פעם אחת בדיק, מיד לאחר `whenever` שהמשתנה שמזכיר בתוך התנאי שלו משתנה וגורם לזה שאותו תנאי יחזיר `true`. `whenever` עשה אותו הדבר, אך לא מtbody אחריו 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 שבתוח ראייתם מוקדם, אפשר הרבה פעמים להשמי את הסוגרים المسؤولות במקומות שמצויה בלוק של קוד. המקלה היחיד שאינו אפשר להשמי אותן הוא שימושיים בבלוק `one-line` על מנת לייצר ערך.

עכשו, סיימנו עם השפה. הגיע הזמן לעبور ל:

- **Compiler** – הלב של הפROYיקט, ומבצע העבודה השחורה – הוא מקבל קוד הכתוב בLittle, והופך אותו ל-AST שאפשר להריץ, או לשמר bytecode. כמו לשפה עצמה, גם לו יש היסטוריה עשירה במיחוד, ובעבר שינוים אפילו יותר רבים:

- **ב-24 באפריל 2022** הוטבלה העבודה על הקומפיילר. אפשר לראות מהקיים – `commit` על התיכון הראשוני של השפה, שעליו דיברתי בהתחלה ממש – לשחק לילדים שמתרטטו לימוד קוד. במבט לאחר, מדובר ב-`commit`

יחסית מצחיק, עם תוספות שאפשר להגיד שהן, אולי אפילו מאד, questionable.

סדרת ה dzerts commits הבאים היו הרבה יותר קשורים לאינטראקטיבית מאשר לקומפילציה, דבר שמראה על רמת ההבנה שלי כשהתחלתי עם הפרויקט:

- **ב25 באפריל, 2022** נוסף אלמנט הקימפול הראשון – Regex למציאת החרת משתנים לא לדאוג, לא עוד הרבה זמן אבין שימוש בRegex זה לא הדרך...
- כשפרויקט היה בחיתולי, גם טרנספילר תוכנן. **ב26 באפריל, 2022** נוסף הטרנספילר, שהציג זיהוי משתנים ופונקציות, כתובות בשaxH, וכן, עדין באמצעות Regex.
- **ב1 במאי, 2022** לא נעשה שינוי משמעותי, אך קובצי הקומפילר שנכתבו עד עכשווי הוזזו לחיבורו ששם little, שם הפרויקט שמשמש איזוחלת. מעכשיו, לא אזכיר עוד שינויים לגבי הטרנספילר, שכן הם לא יהיו רלוונטיים יותר.
- **ב26 בדצמבר, 2022**, אחרי יחסית הרבה דם ייעז ודמעות מהתעסוקות עם המודל שלו, התחלתי את ה Refactor הראשון של הקומפילר. הרעיון החדש עכשווי הוא זהה:
 - lexer ישתמש לזיהוי אלמנטים פשוטים יחסית
 - parser ישתמש לזיהוי יותר לעומק.
 - הטוקנים בסוף יעברו להרצה.
 לצער, עדין לא למדתי את הליך שלו עם Regex.
- **ב27 בדצמבר, 2022**, ממש יומם אחריו, לצערינו, מחקתי את ה Refactor, כי לא הבנתי איך אני מתќדם מאיפה שאני נמצא. בסופו של יומם, אני די החלפתית את רעיון העיצוב של כל הפרויקט כמעט ביום אחד – היגיוני שאביה מבולבל איך להמשין. לקחתני לי הפסקה של כשבועיים, בה התרעננתי, ועבדתי על דברים אחרים.
- **ב16 בינואר, 2023** נוצר ה Lexer הראשון. הוא השתמש בטוקנים שנקבעו ComplexToken, שאמורים להכיל הרבה מידע, תחת כותרת אחת, שאליהם דואגים בהמשך. התוכנית הייתה>Create עכשווים:
 - Lexer מחלק מבנים מסוימים מהקובץ ושם אותם תחת כותרת אחת
 - Parser אוסף את הכותרות והמידע שתחתן, ומסדר אותו באופן ש יהיה קל להריץ
 - לאחר מכן, מרים את הטוקנים
- **ב21 בינואר, 2023** נוספו ל Lexer אפשרויות של זיהוי החרזות שונות, כתיבות וקריאות. הטוקנים הללו הוחלפו בкалלה שנקרא TokenLevel1, CompexToken ו Parser. יהיה בשבייל Parser.
- **ב25 בינואר, 2023** נוסף commit שאומנם לא שינה הרבה, אבל הכל נקבע את ההשראה שרדת ה כי הרבה זמן – פונקציית ההדפסה של עצי axtaxis – גם אז וגם היום, הפרויקט משתמש באותו שמו ובאותו מבנה על

מנת לייצג את העץ.

- **ב6 בפברואר 2023** התחליה העבודה על Parser.שוב, באותו מודל ועם Regex.
- **ב15 בפברואר 2023**, לאחר שהסתירה העבודה על Parser והתחלית עם Interpreter, הבנתי שאין לא יכול להמשיך במודל הנוכחי, כי הוא לא יכול לגדול בצורה טובה.
והחלק הכי חשוב – סוף סוף, הפסקתי להשתמש ב-Regex!
מאז, התוכנת הייתה בערך זהה לתוכנית של היום:
 - Lexer לוקח את הקלט, ממיר אותו לרבות טוקנים שmbatאים את תוכן הקלט
 - Parser מנסה למצוא קבצים של טוקנים, והופך אותם לפחות טוקנים יותר מפורטים
 - Interpreter קורא ומפעיל את הטוקנים האלה
 בהתום הזה גם Lexer נעשה מחדש למחרי, **ובתוום commit שמיד אחריו** התחליה העבודה על Parser.
- **ב9 במאי 2023** נוסף הפיצ'ר האחרון שתוכנן לפרסר – else. קצת אחריו החבתי שסימטי את העבודה על הפרויקט הזה שלו – לא לשכוח, כל מה שקרה עד עכשיו, היה די הרבה לפני שבכל ידעת ש5 ייחידות סייבר צריך פרויקט...
- **ב2 באוגוסט 2023** התחשק לי להוסיף תמייה חיליקת בדוקומנטציה לשפה, יותר בשבייל, ובשביל מפתחים אחרים מאשר בשבייל משתמשים.שוב, בשלב זה עדין לא ידעת אני הולך להשתמש בפרויקט הזה בשבייל סייבר.
- **ב22 בנובמבר 2023** בוצע השינוי המשמעותי האחרון לקומפיילר – האינטפרטר קיבל טוקנים משלו, ונוסף שלב המרה מטוקני פרסר לכאה של אינטפרטר.

עכשו, אחרי שעברנו גם את הקומפיילר, הגיע הזמן למן את הפיצ'רים הסופיים שלו:

define x = 3, x = x + 6 action getX() = { return x }	הגדיר ס = 3, ס = ס + 6 פעולה קבל_ס() = {} החזר ס {	السياح ع = 3, ع = ع + 6 فعل يحصل_ع() = {} استرداد ع {
---	---	--

print(getX())	הדפס(קבל ס())	מבעת(يحصل ע())
---------------	---------------	----------------

נוי מילים מפתח: אפשר לשנות את המילים השמורות בשפה לסת אחר של

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

, או לשנות מילים אינדיבידואליות בעזרת משק מיוחד. [בתוך קובץ readme](#) יש דוגמה לשינוי מילים. אביה את הדוגמאות משם:

- Parser بنוי בצורה שמאפשרת "השלה" של פונקציות parsing תוך כדי היליך parser, ואך מאפשר הסרה של שלבים, על מנת לתת למפתחים שימושיים בפרויקט לעצב את נראות השפה, להוסיף, ולהוריד לה תכונות.

- Custom Tokens: בשלב Parsing, המפתח יכול להוסיף טוקנים משלו, כדי לפחות הוספה שלבים לparser. הופך את הפיצ'ר לעיל לעוד יותר עצמאי.

- הוספת externs מצד המפתח: יש אגד שלם שמיועד אך ורק להוספה משתנים, פונקציות, שדות, סוגים ואופרטורים בדרך מהירה ולא "מנועחת" מדי. זה מאפשר למפתח להוסיף על הספרייה הסטנדרטית Little שספתק, ואףיו ליצור חלקיים שלמים למטרה של שימוש המפתח משתמש בפרויקט (לדוגמה, אם Excel היי משתמשים בפרויקט במקום VBA, הם היי משתמשים באגד הזה כדי להוסיף סוגים ופונקציות שעשוות interfacing עם Excel).

Interpreter •

שיהיה משחו שימש אותו – אנחנו הרי לא בונים טרנספילר בסוף...

בניגוד לחלקים האחרים, דזוקא הפעם לא אזכיר את ההיסטוריה של החלק הזה

בפרויקט, אלא אזכיר רק milestones עיקריים, משתי סיבות:

- האינטרפרט לא עבר שלבים של למידה כמו שאר החלקים עברו
- לא היה תכנון נרחב מאחורי שינויים בהתחלה, הם נעשו אך ורק מנוחות, ולכן קרו הרבה שינויים "桀ל"

ועכשיו, לתוכן:

חלק ראשון: הפרדה מוחלטת: **Interpreter והParser**

בהתחלת ממש, לא חשבתי שהיה צריך לفصل בין השניים – אמרתי לעצמי – למה אם כל המידע שציריך בשבייל להרץ קוד נמצא כבר בparser, צריך ליצור עוד סוג טוקנים, שבticalס מייצג אותו הדבר?

בהתחלת חשבתי שצדקתי, ואכן דברים הילכו טוב, אפילו שהשתמשתי בParserToken בתור טוקני ההרצה, ולא בטוקנים מוכונים. אבל, אולם מאוחר, בעיות התחילו לוץ, והכל התחיל בזאת commit: נראה commit תמים יחסית: סך הכל הוסיף טוקן המUID על ערך אקסטרני, אבל כאן התחילת ה"איןפלציה" בתוקני הזר – כל פעם שנוסף טוקן של parser שאינו קשור, זה מנסה יותר ויותר על הבנת הקשרי הטוקנים, ומציריך כל פונקציה שרק מנסה לדאוג לטוקנים בזמן בנייה לדאוג לעוד טוקנים לא רלוונטיים שלא אמרורים לצוץ אף פעם.

עם הזמן, המצב החמרי, ונهاה מצב שהתחלה ממש לדוחוף פשטוט ערכים לתוך הטוקנים. עם הזמן זה נהיה יותר ויותר מבולגן, כל פעם שהסתכלתי על מידע, אשרה לא הייתה מייד בטוח אם הוא עובד ע"י האינטפרטר או ישר מהפרש. אחרי עוד יותר סבל, החלטתי שאני מפסיק, **ובזאת commit**, ויצרתי סוג חדש לטוקנים של האינטפרטר – על אף שלא היה סופי, לפחות היה קיים...

חלק שני: סדר

ככל שה�택ה האינטפרטר, יותר ויותר פונקציות בו הסתמכו אחת על השנייה, אבל לא בדרך טובה – היו הרבה פונקציות מאד ארוכות אך מאוד דומות, שມטרתן קצרה שונה, ולכן הצדקה את קיומה (לדוגמא, אז היו פונקציות לייצור אובייקט זכרון או רק נסיון יצירה, או, המרת ערך למחרוזת לעומת המרת מזהה *למחרוזת*). לאט אבל בטוח, דברים יצאו משליטה, ובשלב מסוים, אמרתי מספיק, ויצרתי קובץ חדש, הנקרא **Actions**:

הקובץ לא היה אמור להיות קובץ הרצה בפני עצמו, אלא הוא הכל המון פונקציות קצרות שכל אחת משמשת למטרה אחת – שינוי שורה, כתיבה, הפעלת פונקציה... – כשפונקציות עזר אלה קיימות, אין צורך בהרי פונקציות ארוכות באינטפרטר – אפשר פשוט להשתמש בהם וזהו...
באופן שהפעיע אותי אז, גיליתי אחר זמן קצר שאפשר להשמי את קובץ האינטפרטר, ע"י הוספת פונקציית הרצה למחלקה Interpreter של דאץ. לאחר ההוספה, קרה תהילך יחסית ארוך של ניתוק המחלקה Interpreter משאר הקוד וחיבור ל**Actions**, תוך כדי ידוא ששם דבר לא נשבר. לאחר זמן מה, מחקתי את הקובץ Interpreter ואuai מצחיק – כמה שבועות לאחר מכן שיניית את שם המחלקה Actions לשם המוצלח במילוי, Interpreter (:).

חלק שלישי: זכרון דינמי וזכרון קצר פחות

מתחלת הפרויקט ועד לפני קצר יותר מחצי שנה, הנחתתי שלא אctrיך ניהול זכרון מסובך לפרויקט הזה – סך הכל מדובר בשפה קטנה – לא צריך זכרון כזה מסובך... אלה היו מילימ' אחראנות יפות במחשבה לאחר – המודל הראשוני שלי לזכרון היה דומה במקצת לבניה של זכרון האקסטרני שיש בפרויקט המקורי – בנוי עצ, שבשורשו MemoryTree שמאפשר גישה לילדוי, שמסוג MemoryObject. כל MemoryObject גם יכול להיות הורה של MemoryObject אחרים. המודל הזה הביא אליו מגוון בעיות, שאחרי כמה זמן מסויימת כבר הי' יותר מד':

- אין scopes – הכרזה על משתנה אחד יכולה להרוז את הרק של משתנה אחר בעל אותו שם מהקשר אחר
- ערכים `String` – מכיוון שכל ערך חייב להיות מוצמד לאובייקט בעל שם בעז, היה קשה מאוד להתנהל עם ערכים חסרי שם, במיוחד כשפשות ניגשים לשדות של הסוג שלהם (`String.length`, לדוגמה)
- שלושת סוגי ה"קריאות" – לא היה טוון של ערך פונקציה או תנאי, אז כל אובייקט היה כמו מולטיטאסקר עם המן שודות ואופציות, כל אחת למטרה אחרת – מאוד מבולגן...

מכיוון שתכננתי להרחיב את הפROYיקט, ויזיהיתי חלון זמן טוב ל refactor כתן, שכך בדיק סימתי עם דברים אחרים, פתחתי את ה `pull request` הבא, שמטרטטו, וגם כותרתיו, [אימפלמנטציית זכרון סטנדרטי](#):
מטרת PR זהה הייתה להמיר את המערך של אז, למערכת שאנו משתמשים בה כיום – מבוססת מערci ביטים ובקשת זכרון.
על אף שהлон הזמן היה טוב מבחינת הפיצרים שצורך להדיביך מבחינת הזכרון, PR עדין לוקח לרוב 3 שבועות של עבודה וצופה עם 64 commits, רק כדי להדיביך את הפעור ולהיות ברמה של המערכת הקודמת.

חלק רביעי: מספיק דיבורים; צריך לעשות פיצרים

אין צורך לפרט על יכולות ההרצה של האינטרפטר, שכן הוא מציע בדיק או קצת יותר ממה שפורט בחלק של השפה עצמה. כן אפרט על חלקים מיוחדים/נוחים במיוחד באינטרפטר וביציר:

- **גישה קלה לאלמנטים בזיכרון:** המחלקה `Memory` מספקת מגוון פונקציות ושדות לביצוע אינטראקציות, פשוטות ומסובכות, עם הזיכרון הנוכחי, והכל עם פונקציות בודדות, או אפילו רק פונקציה אחת – מקריאת ערך של משתנה, לקבלה מידע אקסטנסיבי על סוג, ואפילו כתיבת אלמנט מסווג ספציפי למקום ספציפי בזיכרון.
- **איורי זמן ריצה:** אחר כל פעולה שימושית שמבצע האינטרפטר, בהתאם לאיורי, הוא מפעיל איוריים מסוימים – מיפוי וירידת שורה, להכרזה על משתנה, או אפילו יציקה מוצלחת של ערך לסוג.
- **שימוש עצמאי קל:** אפילו שלא זאת המטרה, בגלל העיצוב הסופי המוצלח של המחלקה `Interpreter`, ניתן בקלות להשתמש בפונקציות שהמחלקה מציעה, על מנת להריץ קוד, לחשב ביטוי, או לבצע פעולות ספציפיות כמו לקרוא לוילאה עם תנאי הריצה מסוים וגוף מסוים.
- **Interfacing עם פונקציות של Haxe:** מסופק טוון שמטרטטו לאירוע פונקציה הכתובה בזאת Haxe בתוך ערך שמייש בשבייל קורא הקוד. זה שימושי במיוחד לפיצ'ר הבא:
- **ערכים, פונקציות, סוגים, סימנים ולולאה אקסטרניות:** בעזרת טוון הכנסת האלמנט האקסטרני, הפROYיקט מספר פונקציות המאפשרות חיבור כל ככל האפשר של אלמנטים מ `Haxe` לשדות ואלמנטים שאפשר לגשת אליהם תוך כדי זמן הריצה של Little. זה אפשרי רק בזכות הניתוב המוצלח בין חלקים בזכרון.

יכולות

לפרויקט יש שני סוגים לקוחות עיקריים:

- **מפתחי תוכנה:** אם מפתח רוצה לשלב יכולות פרוגראמיות בתוכנה שלו, כמו Excel לדוגמה, הפרויקט זהה נותן לו את האפשרות לעשות זאת, ואפילו נותן יותר אפשרויות ופיצרים מהכלים שהתחום הזה (שפות תכונות אינטגרטיביות) מציע Hscripti VBA (לדוגמה, [טבלה](#))
- **לומדי תוכנות:** מעבר לסתם כל' או ספריה, הפרויקט מספק שני סוגים של סביבות פיתוח – אחת כאתר, ואחת בשורת הפקודה. משתמשים שרוצים למדוד את השפה, או למדוד לתכננת בכלכליות, יכולים לגשת לכתובת של האתר, או, למשתמשים יותר מתקדמים שרק רוצים להתנסות בשפה, להוריד את הפרויקט ולקמפל אותו בלבד (זה [האתר המדבר](#), ניתן להריץ את הלוקו בשרת הפקודה באמצעות הרצת פקודת הבנייה `(haxe compile.hxml)`

אנו שחר מרכוס ברובו ממוקד בבניית והרצת קוד, יש לו תוכנות אחרות, שלאו דואקן קשורות לתכנות לדוגמה:

- **מתמטיקה:** אפשר להדפיס תרגילים מתמטיים בעלי פתרון צזה או אחר, אז לקחת את הפתרון שלהם באמצעות:

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

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

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

סוויתת בדיקות

על מנת לוודא שהפרויקט עובד, כשאני בונה אותו לבדיקה, אני יכול להגדיר פרמטר `Haxedef` שנראה如下 במאזעוט וויזוא שבקובץ הבונה (`compile.hxml`) נמצאות השורות:

`--define unit true` – מגדיר את `unit`: true ונונן לו בברירת מחדל את הערך

`--interp Haxe` – מגדיר את מטרת הבנייה, עשו שהקוד יורץ באמצעות interpreter של Haxe ולא יתקمل.

כש-unit מוגדר,אפשרה הפעלה השורה

```
UnitTests.run(true);
```

שומריצה את כל הבדיקות, אחת אחר השניה ללא הפסקה (הפרמטר שימושי מחייב האם להפסיק בבדיקה נכשלת וגם האם לkür את ההפסקה בין כל בדיקה מ-0.2 0.05 שניות). לנוחות וליעופי, הבדיקות מודפסות עם צבעים ופירמות, ולפי סדר. תמונה להמחשה:

```

Unit Test 3: Function declaration
- Result: Success
Unit Test 4: Property access
- Result: Success
Unit Test 5: Loops
- Result: Failure
- Expected: PartArray([Number(0),Number(1),Number(2),Number(3),Number(4),Number(5),Number(0),Number(3),Number(6),Number(9)])
- Returned: PartArray([Number(0),Number(1),Number(2),Number(3),Number(4),Number(5),ErrorMessage(`for` loop's variable must be of type Number, Decimal or Anything (given: Unknown)))])
- Code:
  define i = 0
  while (i <= 5) { print (i); i = i + 1}
  for (define j from 0 to 10 jump 3) print(j)
- Abstract Syntax Tree:

Ast
└── Variable_Write

```

```

- Stdout:

Module Main, Line 0: 0
Module Main, Line 0: 1
Module Main, Line 0: 2
Module Main, Line 0: 3
Module Main, Line 0: 4
Module Main, Line 0: 5
ERROR: Module Main, Line 0: `for` loop's variable must be of type Number, Decimal or Anything (given: Unknown)

```

כהכנה לעתיד (כשיהו יותר בדיקות), מודפסת הودעה נוספת בסוף כאשר כל הבדיקות מצליחות:

```

- Result: Success
Unit Test 12: Constant pool
- Result: Success
Unit Test 13: Type Name Property
- Result: Success
Unit Test 14: Reference vs. Value
- Result: Success
🎉 🎉 🎉 All tests passed! 🎉 🎉 🎉

```

להודעה הזאת יש שתי צורות נוספות:

```

Unit Test 15: Arrays
- Result: Success
⚠️ ⚠️ ⚠️ 7 out of 15 tests failed! ⚠️ ⚠️ ⚠️

```

```

Unit Test 15: Arrays
- Result: Success
✗ ✗ ✗ 8 out of 15 tests failed! ✗ ✗ ✗

```

כיום, מורצות 51 בדיקות, כל אחת למטרה שונה, אך יכולה להסתמך על דברים אחרים

שניצבים לטעות. בדיקות מסווגות לפי שם פונקציה (`test<test Number>`), וצריכות להחזיר מספר דברים בתור אובייקט:

- שם בדיקה
- האם הבדיקה הצליחה
- תוצאת הבדיקה
- למה הבדיקה צייפה
- הקוד של הבדיקה

והבדיקות הן:

1. – מתמטיקה פשוטה, ופונקציית הדפסה Basic Math
2. – הכרחת משתנים, פונקציית הדפסה Variable Declaration
3. – הכרחת פונקציות, קרייה לפונקציות Function Declaration
4. – יצירת אובייקטים דינמיים, יצירה וגישה לשדות Property Access
5. – לולאות for while Loops
6. – EVENTS AND CONDITIONALS – תנאי after whenever
7. – שימוש בקוד סגור בסוגרים מסולסים כערך למשתנה Code Blocks
8. – שינוי ערך באמצעות הערך שלו Self Assignment
9. – If-Else – תנאי if עם else
10. – Nested Code Blocks - בלוקים של קוד אחד בתוך השני, ובדיקה לראות האם המשתנה שהוכרז בבלוק פנימי יערוך את המשתנה החיצוני (לא טוב), או שיושמד בסוף הבלוק והערך ה"ישן" "יחזר" (טוב)
11. – Code Blocks Inline Blocks – כמו Code Blocks, הפעם גם פונקציית ההדפסה מקבלת את המשתנה שעלה להדפסה בתוך בלוק של קוד
12. – בזיקה האם הכתובות של ערכיהם שאמורים להיות בconstant Pool – בזיקה האם השדה type. עובד על כל ערך, מאובייקט ועד pool אכן נמצאים שם ולא לוקחים זיכרון ממוקם אחר
13. – Type Name Property Reference vs. Value – בדיקת האם השימוש ערך שהוא pass by value הוא יוצר ערך חדש בזיכרון, וההפרש לערכים שהם pass by reference.
14. – Arrays – יוצר, מאחסן ומ使者 מידע מערכיים.

לוח זמנים

cashashatalti עם הפROYIKT, לא בדיק היה לי לוח זמנים, אלא רצון לסייע לו לאחר כמה מוסימות של זמן. לא חשבתי בכלל שאצטרך את הפROYIKT הזה לעבודה בסיביר אז, ולכן מה שקרה בפועל זה שבאמת באופן כללי סימתי מתי שרציתי, ומazel עשית תיקוני באגים קטנים או תוספות ויזואליות לאו שהפROYIKT הציע כשהיה צריך. ארגן לatable:

אבן דרך	תאריך מצופה	תאריך בפועל
תחילת הפROYIKT		23 באפריל, 2022
	פרויקט נוצר, שם – Multilang Coder	23 באפריל, 2022
תחילת עבודה	24 באפריל, 2022	
שפת תכנות עובדת	חדש אפריל, 2023	25 באפריל, 2023 – שינוי שם פרויקט - Minilang

2023 אפריל 26	ספציפיקציה ראשונה		
2022 אפריל 26	התקדמות ראשונה – משתנים ופונקציות		
2022 אפריל 30	שינוי שם פרויקט - Little		
2022 דצמבר 26	ניסיונו הראשון Refactor		
2023 ינואר 16	שינוי ספציפיקציה משמעותי ראשון		
2023 ינואר 16	Refactor ראשון		
2023 ינואר 21	הודבק הפער ממקודם		
2023 פברואר 6	רשמית, העבודה על Parser התחלתית		
2023 פברואר 13	הכנה לRefactor שני		
2023 פברואר 15	Refactor שני		
2023 פברואר 22	הודבק הפער מהוראות הראשונות		
2023 אפריל 11	תחילה עבודה על Refactor ראשוןInterpreter		
2023 אפריל 25	שפת תכנות עובדת		
2023 במאי 9	תוספה נוספת: else		
2023 במאי 9	תיקון באגים אחרונים		
2023 אוגוסט 2	דוקומנטציה ותגיות בקוד	-	פיצ'רים נוספים
13 בספטמבר, 2023			סיום העבודה על שפת התכנות, לפני סיבר
4 בנובמבר, 2023			פיתוח העבודה על הפרויקט, לסיבר
2023 נובמבר 8	יצירת סוויתת בדיקות		
2023 נובמבר 10-20	פיתוח ענף בפרויקט: זיכרון סטנדרטי		
2023 נובמבר 16	שינוי ספציפיקציה משמעותי שני		
2023 נובמבר 22	תוספה לספקטיקציה		
2023 נובמבר 22	סיום עבודה רשמי על Parser, בלבד Refactor. ראשון Interpreter. ניהול הטוקנים של Parser וה-Interpreter	חודש מאי, 2024	שפת תכנות משופרת עובדת, סיום פיצ'רים הכרחיים להגשה
2023 דצמבר 2	סיום החלק המשמעותי ביצירת heap stack וה-heap. stack אינו סופי		
2024 ינואר 25	HashTables		
2024 ינואר 29	פיתוח Pull		

		Request מהענף של זיכרון סטנדרטי		
13 בפברואר, 2024	הראשון Refactor Interpreter לשולם.			
18 בפברואר, 2024	Pull Request שולב אל תוך ענף main			
23 בפברואר, 2024	שינוי שם: Heap -> Storage Stack -> Referrer			
25 בפברואר, 2024	ניסיון לתוכנות חדש של Storage, וביטול			
25 בפברואר, 2024	ניסיון לתוכנות חדש – Referrer של המטרה מעבר למערכת המסתמכת על השאלה מקומפלים לזכרון עצמאי.			
29 בפברואר, 2024	Referrer תוכנת החדש בהצלחה			
2 במרץ, 2024	תמייה באחסון, קריית ופינוי סוגים בזיכרון			
5 במרץ, 2024	מערכת סוגים שופרה, סוגים הוכנסו לברירת קבעים, וו RTTI שופר.			
7 במרץ, 2024	Refactor שלישי: שילוב/מחיקת קבצים דומים/ללא צורך Actions (לדוגמה: Interpreter)			
8 במרץ, 2024	פיקול הפרויקט המוגש לענף שונה: מעכשי, התקדמות: להגשה היא על ענף branch/functional-programming			
17 במרץ, 2024	הסרת תמייה מתקדמת בסוגים – ניתן רק לגשת, ולא להזכיר מצד Little המתכונת בLittle			
17 במרץ, 2024	תיקון oversight משמעוני במיזח בתוך תנאים ולולאות			פיקול עבודה לענף: branch/functional-programming
18 במרץ, 2024	סיום דוקומנטציה של הפרויקט			
28 במרץ, 2024	תחילת הוספת תמייה ב-pass-by-reference			

20 באפריל, 2024	תמייה מלאה ב- pass-by-reference		
20 במאי, 2024	סגירת בעיות ישנות וחדשנות Github Issues		
24 במאי, 2024	סיום הפרויקט מבחינת פיצרים		
29 במאי, 2024	סיום הפרויקט מבחינת באגים		

סיכונים

כבר ב2022, רק כשהתחלתי, היה לי ניחוש, להפתעתו אפילו אחד מוצלח, למה הולך להפריע, ומה יכול לתקוע אותו בפיתוח, אך כשחשבתי על אותם סיכונים אז, החלדתי להמשיך הלאה בפיתוח. לדבר על סיכונים שלאו דואק קשורים לקוד עצמו, אלא לכלים שבהם השתמשתי והיו זמינים לי

- **שימוש בשפה Haxe:** אפילו שהוא השפה האהובה עלי, בכל פרויקט שאני מתחילה אני תמיד חyb' לתהות האם אני יכול להשתמש בה. לצערי, התשובה לא תמיד חיובית, ויש דוגמאות לכך:

- **Dynamically:** פרויקט אחר, שהוא אפליקציה לאירופתית בעיות מתמטיות המוצגות בעזרת גרפיקה (תרגילים בגיאומטריה, פיזיקה, מודלים חישוביים). הייתה חyb' יכולות AI לפROYיקט, וכן ספריות אלגברת ליניארית ופתרית מסוימות, דבר שאין באhxex, ולא הייתה מעוניין לפתח.
- Fact: השם של הפרויקט הזה זהה לספריה שתכוננתי לפתח בעל מטריה קשורה – פיתוח גרפיקות לצורות שאפשר לשחק איתן. הפסוקתי לפתח אותה בגלל חוסר בבקוש, מהצד שלו, ומשתמשים אחרים. **קישור**
- SpawnerTag: פלאגין לשרת במשחק Minecraft, שמטרתו להפוך אביזר "בלוק" הנקרא Monster Spawner לאביזר שאפשר להשיג בצורה אין-סופית באופן תיאורטי. כשהייתי את זה, עוד לא היה כל' לפתיחת פלאגינים לשרת Minecraft, אז הייתה חyb' להשתמש בחולין Kotlin. היום כן קיימכ' זהה, ושמו PickHaxe (משחק מילים על המילה pick, אביזר נפוץ ושימושי Minecraft)

הפעם כן יכולתי להשתמש, שכן רוב הדברים המיוחדים שהייתי צריך כבר היו קיימים (ספריית hash) ולא הייתה צריכה הרבה הרבה כילום חיצוניים חז' משש שהשפה מציעה (כן השתמשתי במחלקות ופיצרים מהספרייה Vision, אך רובם היו כל' עז, ובכל זאת אני פיתחת אתם :)

- **ניהול ذיכרון:** מהתחלתה, חשתי שהיוי לי בעיות עם אחסון מידע – בסוף, הרצת קוד זה לא רק פקודות – צריך לשמר מידע על הדריך. בהתחלה הפתרון באמצעות היסטנדרט – למדתי על heap-stack וה-stack, איך מתכניםים כל אחד ומה מיוחד לכל אחד, ועליתני על הפתרון העדכני, שמורכב מרבעה מחלקות: ConstantPool, Storage, Referrer, ExternalInterfacing. משחו חמוץ – איפה באמת heap-stack וה-stack? 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 מكمפלת את עצמה לשפות אחרות, לא לקוד מכונה. ידעתו שזה יכול להיות בעיתוי, אך הנחתתי שזאת לא בעיה ממשמעותית מסופיק, והמשכתי האלה. זאת הייתה הנחה נכונה, שכן על אף שהיא הקשתה עלי בתכנות הזיכרון, והייתי חייב ללקחת פתרון שלא היה 100% סטנדרטי, יכולתי להשתמש במגוון הפיצ'רים שנועדו לעיבוד מידע, שלא תמיד יש בשפות אחרות...
- **זמן:** ברור שלכל פרויקט יש בעית זמן צדו או אחרית, אבל אחד מהדברים שהכי חששתי מהם, זה כמה זמן זה הולך ללקחת – שפט תכנות לא בא בהקלות, וראיתי איך שפות תכנות אחרות קיימות מלא שנים, ועדין יחסית "חלשות". בגלל החשש הזה, הצבתי לי גם מטרה וגםDDL – כמו שהזכרתי בתכנון זמינים, הבוחתי לעצמי שאני מסיים את הפיתוח עד בדיקן שנה אחרי – התחלתי באפריל 2022, והבטחתי שאסיים עד סוף אפריל 2023, לא משנה אם כל הדברים שרציתי לעשות בפרויקט נמצאים בו. למזל, באמת הספקתי את הרוב המוחלט עד אז, ורק היו תיוקני באגים כאלה ואחרים מאז ועד שהפרויקט קיבל משמעות חדשה, כפרויקט סיום בסיבר.

תיאור תחום הידע

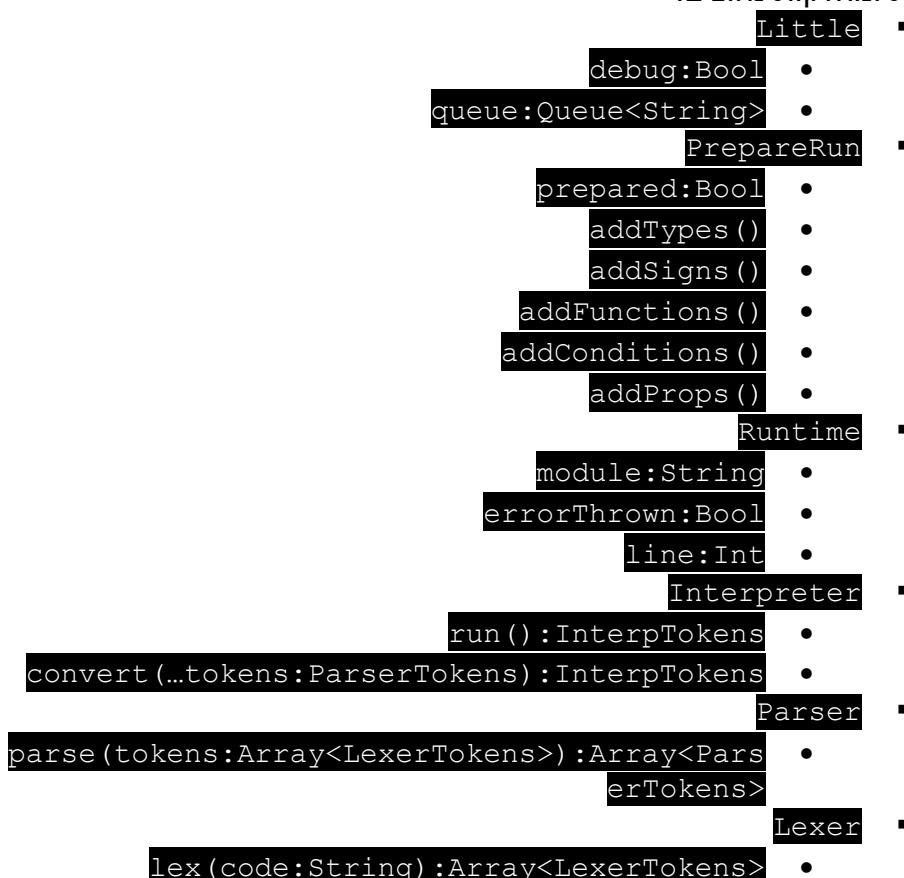
יכולות

סוג לקוחות: מפתח אפליקציות

לפתחי אפליקציות יש מגוון אפשרויות אינטראקציה עם הפרויקט, הכוללים את כל האמצעים שהפרויקט מציע, מבניה, הרצה, ואסיפת מידע דיאגנומטי. אפרט

- **יכולת: הרצת קוד הכתוב בittle בזמן ריצה**
 - מפתח יכול לחת כל קוד מכל מקום שהוא רק רוצה, ולהריץ אותו באמצעות הפונקציה `run` שבמחלקה `Little`.

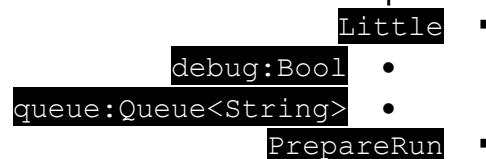
שדות ומחלקות נוחצים:



יכולת: הרצה לפני מודולות

- מפתח יכול לחת קוד, ולהריץ אותו לאחר מכן 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 בהמתנה באמצעות קריאה לפונקציה `run` שבאותה מחלקה.

שדות ומחלקות נוחצים:

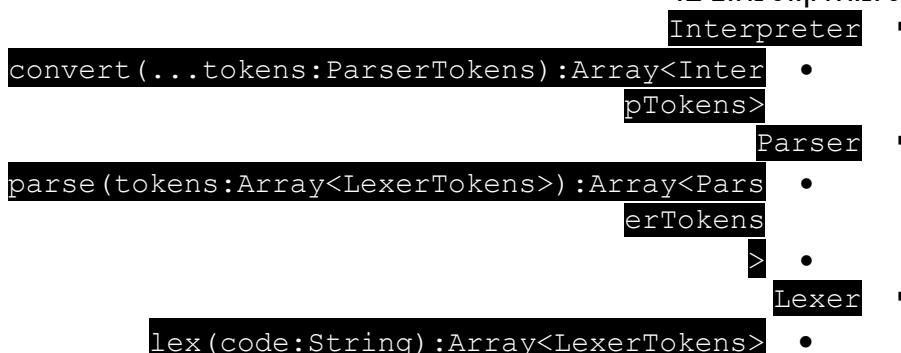




• יכולת: בניית קוד AST

- מפתח יכול לחת כל קוד מכל מקום שהוא רק רצאה, ולבנות אותו ללא הרצה באמצעות פונקציית `compile` שבמחלקה `Little`.

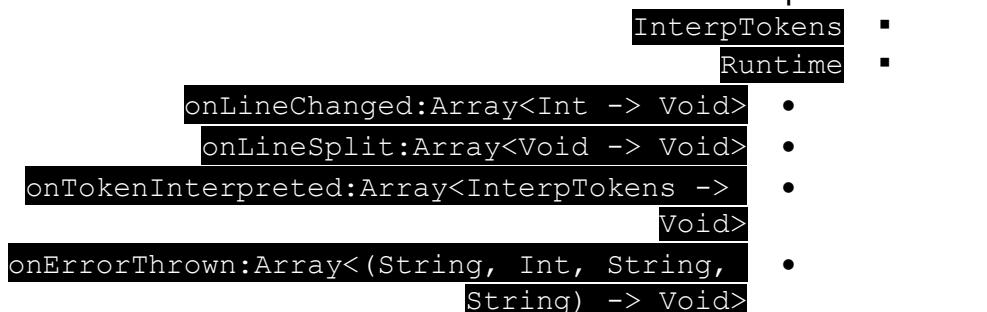
שדות ומחלקות נחוצים:



• יכולת: אסיפת מידע דיאגנומטי על אירועי ריצה

- מפתח יכול להוציא event listeners למגוון אירועים הקשורים בזמן הריצה של הקוד המובא, מאירועים בסיסיים כמו מעבר שורה עד ליותר מפורטים כמו התחלת לולאה. על מנת להוציא מזמן, יש להשתמש בשדות המתחליפים ב-`on` שבמחלקה `Runtime`. אפשר לגשת למחלקה באמצעות `Little.runtime`.

שדות ומחלקות נחוצים:



```

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):Int
                erpTokens
            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
    
```

• יכולות יצירתיות Bytecode

- מפתח יכול לשמר קוד להרצה מאוחרת בפורמט מצומצם בעזרת הפונקציה `ByteCode.compile` שבמחלקה `ByteCode`. הפונקציה הזאת מקבלת AST, שאוטו אפשר לקבל באמצעות `Little.compile`, שהוזכר קודם לכן. יצירת Bytecode מחרוזת 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
    
```

- **יכולת: הוספת אלמנטים חיצוניים**
 - ישרו אגף שלם הנועד רק כדי להקל על מפתחים להוסיף ערכיהם, 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:InterpTok
     ens, objectTypeName:String,
     objectAddress:MemoryPointer}

```

- **יכולת: קריאה וכטיבה ישירה ובתווחה מהזיכרון**
 - אם מפתח ציריך, 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) :MemoryPoint
     er
read(path:Rest<String>) :{objectValue:Int
     erTokens, 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 לדיזכרון שמפונה ע"י Interpreter**
 - אם מפתח ציריך, הוא יכול לגשת למערכי הbutים של Storage וגם של Referrer, ולהשתמש בפונקציות שבאותן מחלקות כדי לקבל מידע על אותם מערכי butים.
 - שדות ומחלקות נחוצים:

```

Storage – בשבייל סוגים: ( )
UInt16, Int32, UInt32, Float, Pointer, Function,
:(Condition, Sign, Object, Type
     store(...)) •
     set(address:MemoryPointer, ...) •
     read(address:MemoryPointer, ...) •
     free(address:MemoryPointer, ...) •

```



- יכולת: המירה חלקה בין ערכי Little לערכי Haxe

- למפתח יש גישה למחלקה Conversion, המאפשרת המרות בין ערכים בLittle, InterpTokens, לשימושים אובייקטיבים מסווגים, לערבים ורגילים, שסוגם משתנה (Int, Float או אפילו סתם אובייקט דינמי)

◦ שדות ומחלקות נחוצים:



- יכולת: החלפת מילים שמורות ומיללים בספריה הסטנדרטית

- המפתח יכול, במידה הצורך, להחליף מילים מפתח כלשה או אחרת לאיזה צירוף אותיות רציף שהוא רוצה, כל עוד הצירוף לא מכיל רווח, סימן, אינט ריק ולא מתחילה במספר. הוא עושה זאת דרך שינוי הערכים שב Little.keywords KeywordConfig או יצירת KeywordConfig אחר. השוואת השדה ל KeywordConfig change של אותה מחלוקת חדש והחלתו באמצעות הפונקציה (KeywordConfig) change(config: KeywordConfig).
- שדות ומחלקות נחוצים:



(יש בין 50 ל 100 מילים מפתח וסימנים שאפשר לשנות. לא אשום 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 כמשמעותי תכונות ופיצרים לשפה, ורוצים לנפות באגים, או לראות איך התוספת שלך מסתדרת עם שאר עץ syntax.

- שדות ומחלקות נוחицы:

```

PrettyPrinter
printParserAst(ast:Array<ParserTokens>,
    ?spacingBetweenNodes:Int = 6):String
printInterpreterAst(ast:Array<InterpTokens>
    , ?spacingBetweenNodes:Int = 6):String
getTree_PARSER(getTree_PARSER(root:ParserTo
    kens, prefix:Array<Int>, level:Int,
        last:Boolean):String
getTree_INTERP(root:InterpTokens,
    prefix:Array<Int>, level:Int,
        last:Boolean):String
stringifyParser(?code:Array<ParserTokens>,
    ?token:ParserTokens):String
stringifyInterpreter(?code:Array<ParserToke
    ns, ?token:ParserTokens)
prettyPrintOperatorPriority(priority:Map<In
    t, Array<{sign:String,
        side:OperatorType}>>):String

```

סוג לקוחות: לומד תכנות, מתכנת Little

מעבר לכלי למפתחים, גם משתמשים רגילים יכולים להשתמש בפרויקט, דרך שני הלקוחות שנותנים בו – אחד דרך הטרמינל (שורת הפקודה), ואחד [באתר אינטרנטני](#), שתוכנו נמצא בפרויקט:

- באתר האינטרנט, יש לנוUrur קוד מיוחד, בעל 3 חלקים – בחלק השמאלי של המסר, יש שלושה חלונות שנitin לכוז: הקלט, עץ syntax והפלט. בחלק הימני לעלה, יש ממשק המאפשר שינוי של המילים השמרוות, עם דוגמאות של שימוש בקוד. בחלק הימני למטה, יש מדריך QuickStart לשפה.

מראים את Urur 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, iPad) ולפעמים (Linux)).

שמראים את הפרויקט "raw", הוא ידפיס קצר מידע, ואחריו יהיה אפשר להקליד קוד. מעבר לכך, אפשר להקליד פקודות מסוימות, והן רצות כאשר הן בשורה ריקה:

- !m! – מנקה את המסר, ומתחילה להקליד קוד במצב רב-שורתי, ולא מריץ קוד מיד אחריו לחיצה על מקש enter!
- clear! – מנקה את המסר, מתחילה להקליד קוד במצב הקודם שהיינו בו
- clearLine! – מוחק את השורה האחורונה, זמין במצב רב-שורתי בלבד
- מריץ את הקוד שהוכנס עד עכשו במצב רב-שורתי – run!

- default! – מחזיר למצב שורה ייחודית ומנקה את המסר!
- ast! – מנקה את המסר, ו מעביר למצב שمدפיס את עז syntax של הקוד במקומם להריצ' אותו.
- printSample! – מדפיס קוד הכתוב לפי הספציפיקציה הרשמייה, קיימ למטרות למידה.

על אף השוני בין סוגיו ה老子, הרוב המוחלט של הפיצרים מוצעים לשנייהם, ולכנן אפרט על היכולות שלהם ביחד.

- **יכולת: סביבת ריצה cross-platform**
 - מכיוון שהפרויקט לא משתמש בשום פיצ'ר שsspציפי לפלאטפורמה מסוימת, אפשר לקמפל אותו לכל מטרה, ואףלו להריצ' אותו בעזרת interpreter של Haxe עצמו.
 - "שדות ומחלקות" נחוצים:
 - Hxcpp (ספריה וכלי בנייה ל-C++)
 - Hashlink (ספריה, כל-בניה ומוכנה וירטואלית הכתובה ובונה ל-C)
 - HaxeC Interpreter (חלק מק Haxe, מרים קוד ללא טרנספילציה)
 - HaxeC (אורץ בתוכו טרנספילר JS)

- **יכולת: הרצת קוד מהירה/אוטומטית**
 - בלקוח האינטראקטיבי, כל פעם שהמשתמש משנה את הקוד, הוא גבנה מחדש והפלט מואג. בלקוח של שורת הפקודה, תלוי במצב: במצב שורה ייחודית, הלקוח משתמש בלחץ enter בשבייל להריצ' 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:Boolean) •
              reset() •
```

- **יכולת: הכרזת משתנים, פונקציות, וכינוים**
 - שפת התכנות נותנת למשתמשים לצור פונקציות שמחזירות כל ערך, ומשתנים מכל סוג בעלי כל ערך שיש בו הגיון. כאשר נתונים למשתנה ערך שהוא לא משתנה, זה נקרא "כינוי", והמשתנה מתנהג כמו הערך שהוזמד לו (אפשר לשים אותו בין אופרנדים לדוגמה עם הערך שלו הוא אופרטור)

- מחלקות ושדות נחוצים:

```
Parser
mergeComplexStructures (pre:Array<ParserToken
  ns>) :Array<ParserTokens>
  Interpreter
    declareVariable (name:InterpTokens,
      type:InterpTokens, doc:InterpTokens
```

```
declareFunction declareFunction(name:InterpT
                               okens, params:InterpTokens,
                               doc:InterpTokens)
```

-

יכולת: המרת מהירה ופושטה בין סוגים ערכיים

- מילת המפתח המשמשת לנთינת סוג קונקרטי למשתנים גם משמשת ל"יציקה" (casting). פונקציית היציקה מוגדרות בעזרת פונקציה שנמצאת על המחלקה של האובייקט שיצקם, ושם הפונקציה הוא `<type><to>_keyword`, כאשר `to_keyword` זו מילת המפתח המוגדרת שאיתה מתחילה שמות של פונקציות המרת (ברירת מחדל: `to`), ו-`type` זה שם הסוג.

- שדה נחוץ:

```
Interpreter
typecast(value:InterpTokens,
          type:InterpTokens):InterpTokens
```

-

יכולת: ראיית עץ syntax

- בלקוח האינטראקטיבי, מעל הפלט הסטנדרטי, מופיע עץ syntax שנבונה מהקוד שלך. בלקוח של שורת הפקודה, יש להעביר למצב ראיית-h-AST בעזרת הפקודה `!ast`, ולאחר מכן הכנסת קוד כרגע.

- שדות ומחלקות נחוצים:

```
JsExample
ast:TextAreaElement
Main
main()
```

-

-

-

-

-

יכולת: שגיאות מודעות Kontext

- בכל מקום שבו נזרקת שגיאה, נעשה מאמץ לקשר את השגיאה למקום שמננו היא נזראה – השגיאה מכילה שמות משתנים, פונקציות, ואףלו קוד מהמקום בו נזרקה השגיאה, ובמקרה הצורך, אפילו חושפת מידע שהמשתמש אולי לא ידע (לדוגמא, תדועה שגיאה ספציפית לולאות `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

- כבירת מחדל, יש למשתמשים בשפה גישה למחלקה בשם `Memory`,
- המספקת מספר פעולות ומשתנים העוזרים עם גישה והתעוקות "לא בטוחה" עם הזיכרון

- שדות ומחלקות נחוצים:

Plugins

```
registerType(typeName:String,
              fields>TypeFields)
ExternalInterfacing.ExtTree
```

-

-

יכולת: קבלת כתובות של ערכים בזיכרון

- כברית מחדל, לכל ערך מאפשר 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 מאפשר השדה `type`, המחזיר את שם הסוג של ערך מסוים. הוא לא מבידיל בין סוגי של מידע – לפונקציות זה יחזיר `Function`, ולסוגים זה מחזיר `Type`. (כמובן שאפשר לשנות את שמות הסוגים, לפחות בלקוח האינטראקטיבי)
- שדה נחוץ:

Plugins

```
registerInstanceVariable(propertyName:String,
    propertyType:String, onType:String,
    ?documentation:String,
    ?staticValue:InterpTokens,
    ?valueGetter:(objectValue:InterpTokens,
        objectAddress:MemoryPointer) ->
    InterpTokens)
```

• יכולת: שינוי מיילים שמורות

- זמן בלקוח האינטראקטיבי – המשתמש יכול בעצמו לשנות את המיילים שמורות לשפה אחרת, סגנון אחר, או סתם קצר.
- שדות נחוצים:

JsExample

```
keywordTable:TableElement
    new()#update()
getCodeExample(keyword:String):String
```

• יכולת: מגוון פעולות "broadcast"

- המשתמש יכול להשתמש ב-3 פונקציות על מנת להציג דברים לקונסולה:
 - הדפסה רגילה, עם מודולה ושורה `print`
 - כmo `print`, אבל עם הרקדמה "WARNING". קוד ממשיר לרוץ `warn`
 - גם כmo `print`, עם הרקדמה "ERROR". קוד מפסיק לרוץ `error`
- 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: קרייה דינמית של משתנים**

- פעולה read יודעת להחזיר את הערך של משתנה, רק לפי שמו או מסלול המוביל אליו, כשהוא מובא כמחרוזת. זה כל' חזק מאוד, המאפשר שימוש מותנה במשתנים מסוימים, וגם תכנות עם כל מני אפשרויות "מקרו-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 בлок של קוד שמחזיר מחרוזת. יש לציין שלא כל מחרוזת מותרת: מחרוזת חייבת להיות בעל לפחות אחת, ללא רווחים, ללא אופרטורים, ולא להתחיל במספר (כਮובן, שגיאות יזרקו בהתאם, תלוי בעבר על הכללים)

- שדות נחוצים:

```
Interpreter
declareVariable(name:InterpTokens,
                (type:InterpTokens, doc:InterpTokens)
declareFunctiondeclareFunction(name:InterpT
                                okens, params:InterpTokens,
                                doc:InterpTokens)
```

מבנה הפרויקט

ארכיטקטורה

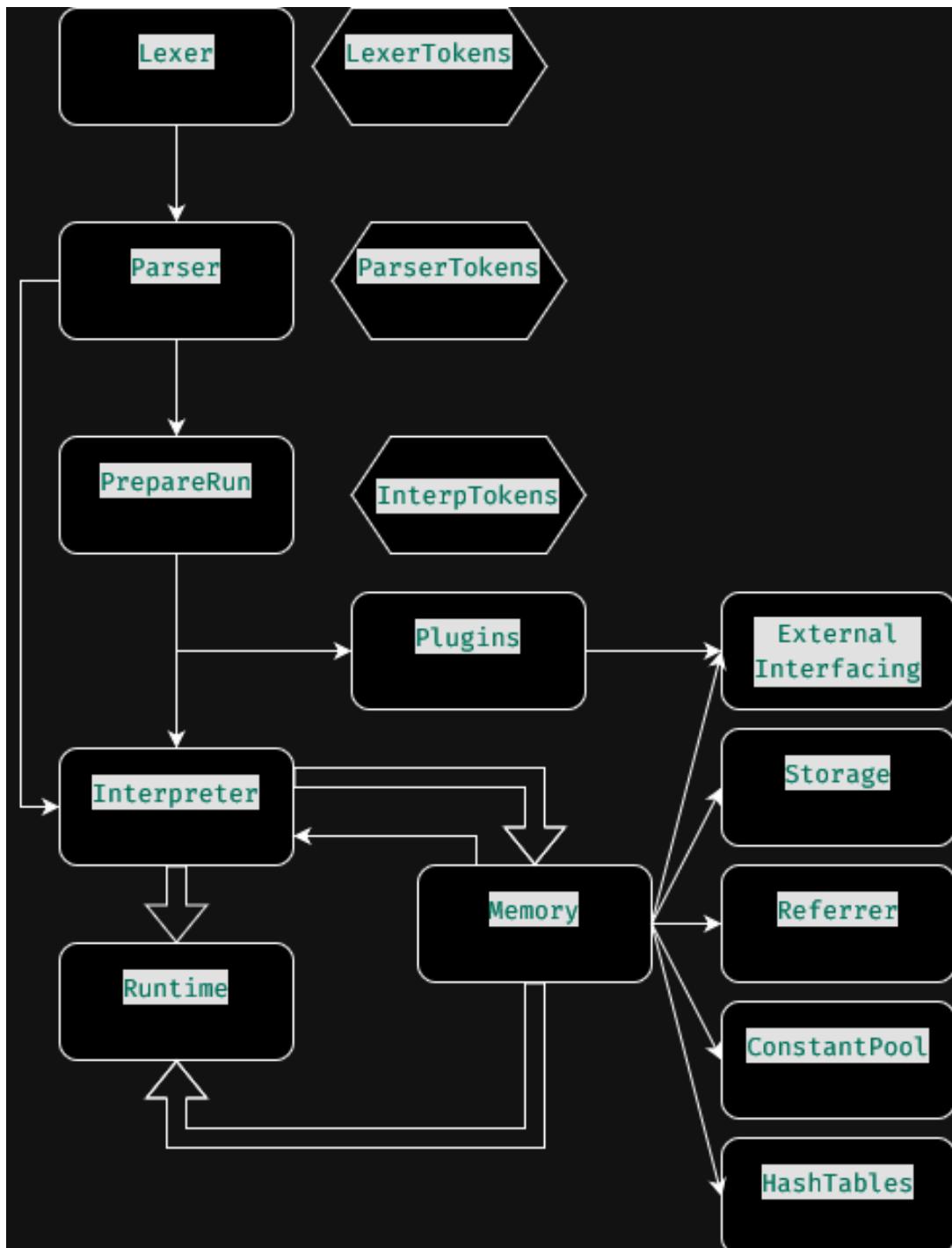
רכיבים שונים, והקשרים ביניהם

כמו שנאמר מקודם (תחת הcotרת של "מערכת"), הפרויקט עצמו מורכב מ-3 חלקים – שפה (ספצייפיציה), הקומפיילר, או, יוצר-AST, והאינטרפרטר, או, המcona הווירטואלית. בזמן שהספצייפיציה כמעט ולא נמצאת בתוך הקו, שני החלקים האחרים של הפרויקט מאוד מופרדים, ונמצאים בתיקיות שונות, ואחד לא משתמש בשני. אותן שני חלקים גם משתמשים בכלים גלובליים, היכולים להסתמך בהזירה 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, ובאי-גם שרטוט. הבה נתחילה:

- **קומפיילר:**
 - **Lexer** – לוקח את הקלט מהמשתמש, בצורת מחרוזת. המחלקה מפצלת את הטקסט לטוקנים, המופרדים לפי סוג – מזהה, מספר, אופרטור. כאן נמצא השימוש הראשון בכלים הגלובליים, על מנת לזהות מהו אופרטור בדיק. לאחר הרצאה, אנו נמצאים עם מערכת טוקנים פשוטה:
 - **Parser** – לוקח את הטוקנים מהlexer, וממיר אותם לכאליה בצורה שונה לו להתעסק איתה. לאחר מכן, מעביר את הטוקנים האלה סדרה של פונקציות, אחת אחרי השניה, ומפתח עץ syntax יותר ויותר "מוסובך" – כל שלבណה מוסיף למתונה לתכמה אחרת בשפה, וכך אפשר לראות שככל שבב עמוק יותר מקצר קצת את העץ. מדובר בקובץ מאד ארוך (כ-950 שורות), המכיל את רוב הקומפיילר, גם "פיזית" וגם לוגית. בעצם, התוצאה שקיבלנו קרובה מאד לעץ syntax הרשמי. קיבלנו את העץ הסופי, עכשו הגיע הזמן "לייצא" אותו, ולהשתמש בו:
- **אינטרפרטר:**
 - **PrepareRun** – לפני הרצאה, מוסיף את כל המשתנים, הפונקציות, האופרטורים, הסוגים והשדות ומיציר את הספרייה הסטנדרטית, באמצעות **Plugins** – מכיל מגוון פועלות להוספה אלמנטים חיצוניים לשפה, המשתנים רגילים ועד לשדות על סוגים ספציפיים ואופרטורים. משתמש במחלקה **ExternalInterfacing**, שעלה 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 ההרצאה העיקרית. הפונקציה משתמשת בוגון הפעולות של המחלקה **Interpreter** על מנת למש את הרצאה, קוראת לאירועים שבמחלקה **Runtime** שמצופה, וכשריך לשומר מידע, היא משתמשת במחלקה **Memory**. אין סדר קבוע לקרואות אלה, שכן הן תלויות בקלט מהמשתמש. אפרט עם השלשה:
 - **Interpreter** – מכיל פונקציות המפעילות את הטוקנים מעץ syntax – החרצת 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 מצבים המוצבים כדי לגשת לאותו מערך
- **Referrer** – גם מערך זיכרון, אך קטן בהרבה – מקשר ביןשמות משתנים לערכם, תלוי במקום ממנו הם מבוקשים
- **ExternalInterfacing** – המגשר הסופי בין ערכי Haxe לערכי Little, מייצג את ערכיו עצים ולא כמערך בתים
- **ConstantPool** – בריכה של ערכים נפוצים, קיים על מנת לחסוך בזיכרון. לוקץ זיכרון מ-**Storage**.
- **HashTables** – מיצר, קורא וכותב מפות hash בעזרת האלגוריתם **MurmurHash1**. בעזרתו אפשר לגשת לאובייקטים.
- **Runtime** – מכיל אירועים שונים, שאמורים להיקרא כל פעם בדבר מסוים מתבצע ע"י האינטפרטר – טוון נקרא, משתנה מוכרז... גם מכיל מידע על הריצה עצמו (כמו הפלט לדוגמה), ופונקציות ההדפסה לMINIHON (**print**, **warn**, **throwError**, **broadcast**). דומה ל-**print**, אך אפשר לעורר אותו – נועד למפתחים המשתמשים בפרויקט זהה כספריה לאפליקציות שלהם.

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



ותרשים יותר מפורטת, מסווג גרפ' JML. כולל את כל הרכיבים בעלי תפקיד מסוות:



טכנולוגיות

שפט תכנות

הפרויקט מתוכנת בשפת תכנות שasma Haxe.

Haxe היא שפה, טרנספִּירר ו אינטראקטיבית רב-מטרת, המאפשר בניית אפליקציות למגוון שפות תכניות, קודים-בטים (bytecode), ואף יכולת הרצה על המקום באמצעות אינטראקטר מיוחד.

JavaScript, C, C++, C#, Java, מאפשרת בניית קוד למגוון שפות תכנות אחרות: Haxe ActionScript3-ו, Lua, PHP, Python

היא גם מאפשרת ייצור קוד-בתים ותמיכה מיוחדת לכמה פלטפורמות: יש תמיכה בNode.js ואפשר ליצר קובצי SWF שירות. אפשר גם לקמפל קוד בתים למכוונות הווירטואליות Neko ו-Hashlink, וכן קוד-בתים JVM.

Haxe, כמו הפרויקט הזה, הוא פרויקט open source, תחת רישיון MIT. הפיתוח של התכילת ב-2005, על ידי Nicolas Cannasse, יוצר אחד מפתחי הקומפ'ילר MTASC.

הטרנספֵילר והאינטראפטֵר של השפה כתובים ב-OCaml, שפה הידועה ביכולות עיבוד המידע שלה. וכן גם שפה טובה לכתיבת שפות תכנות.

המידע שלה, וכך גם שפה טובה לכתיבה שפות תכנות.

אגנון התכונות של השפה ותכונותיה לקחוות משפטות כמו JavaScript ו-C#, אך היא גם מציעה תכונות יחסית ייחודיות, שיעזרות בעיבוד מידע, כמו `switch` ו-`Variable Algebraic Data Types`.

תחומי עניין

באופן כללי, הפרויקט מתעסק בעיבוד, אחסון ומיניפולציה מידע בסדר גודל גדול, יותר בפקודו: קומפיילרים, אינטראקטרים, וקידוד מידע.

קומפיילר הוא תוכנה הממיר סוג תוכן מסוים, בדרך כלל קוד בשפת תכנות מסוימת, לפורמט הנitin להרצה, בדרך כלל קוד מכוונה. לרוב, קומפיילר בניו מ-3 שלבים: Lexer, Parser, ולבסוף, Generator. בפרויקט זהה, שני השלבים הראשונים קיימים באופן מלא, והשלב השלישי מתmesh ע"י יצירת קוד-בתים של עץ syntax ש"נפלט" בסוף.

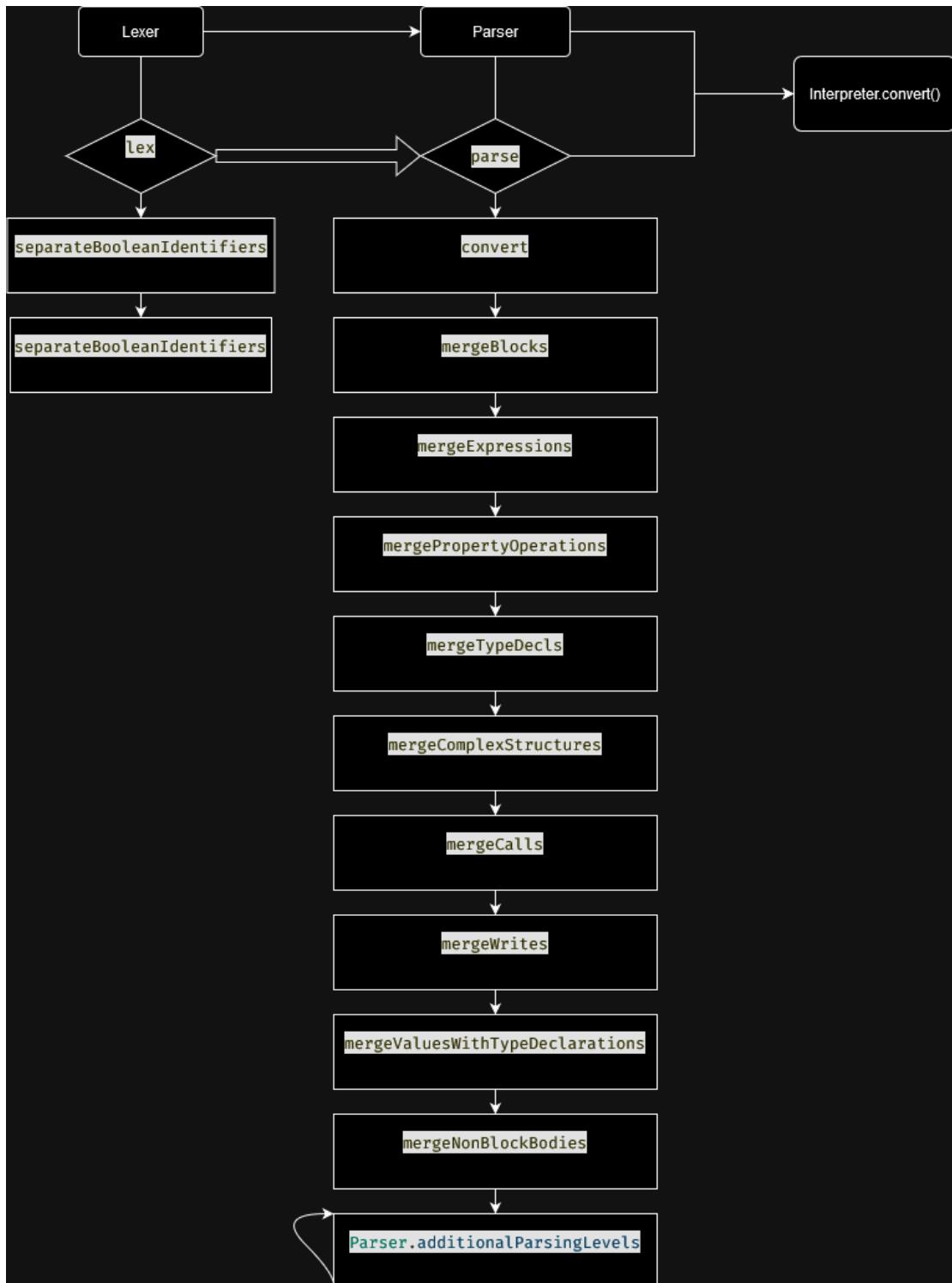
אינטראפרט היא תוכנה המקבלת מערך של פיסות מידע, ומפעילות רכיבים מסוימים בזיכרון שבד"כ מבצעת אינטראקציה עם המשתמש. האינטראפרט לרוב מורכב מכמה "מרכזי ניהול" עיקריים: זיכרון, אינטראקציה עם המערכת, וכמוון, "سورק" קוד.

קיוד מידע הוא תחום החופף הצפנת מידע – כמו בהצפנה, המידע משנה צורה, אך כאן המטרה היא, בדרך כלל, לכזב אותו או לפרמטר אותו בדרך שקללה לקרוא/לכתוב. דוגמה טובה היא אלגוריתם hash – המטרה אינה בהכרח להצפין, אלא להמיר מידע אחר באורך סטטי אך עם תוכן משתנה, בהתאם לקלט.

זרימת מידע במערכת

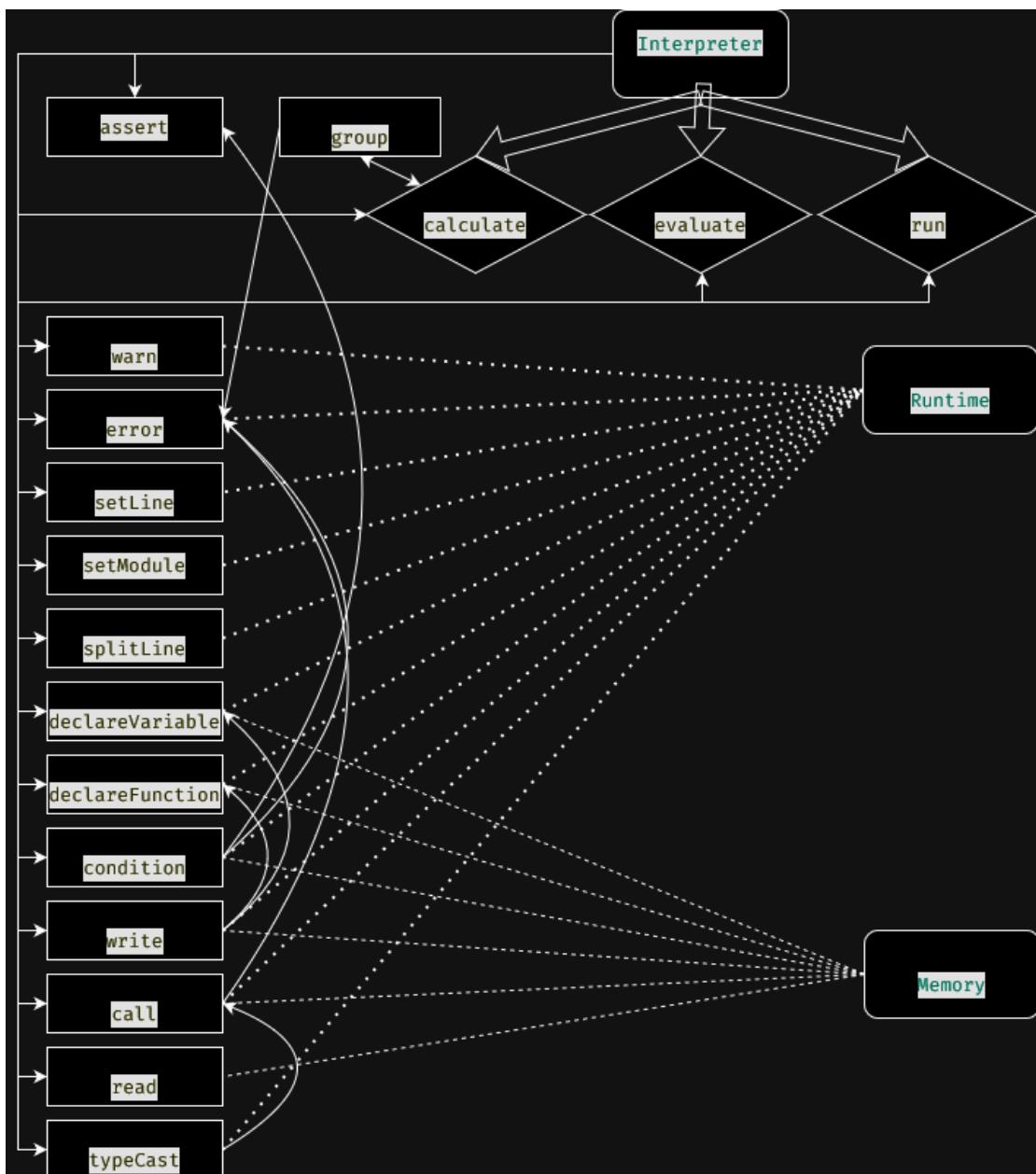
הקומפיילר

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



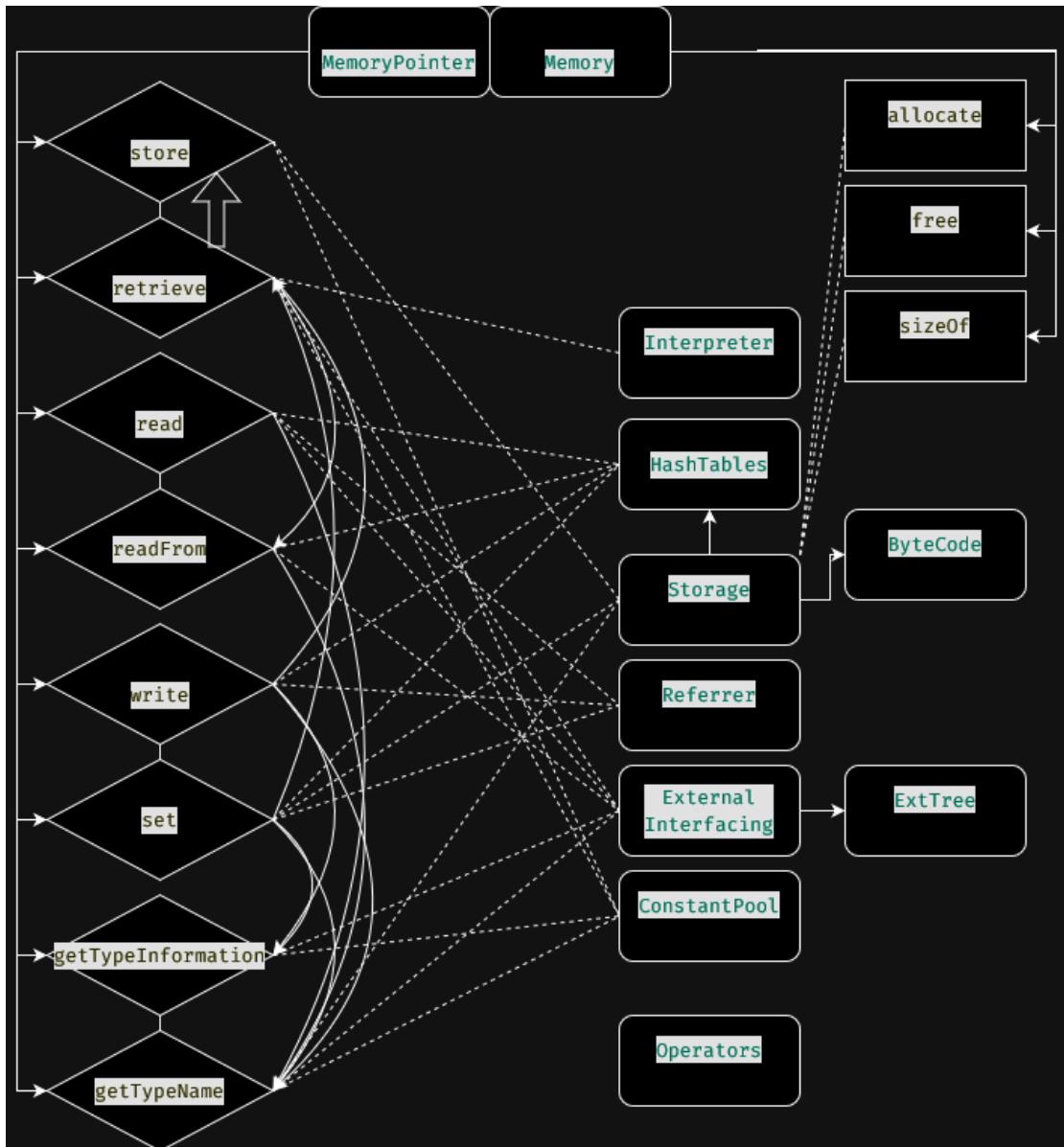
הエンטרפרט

שלב המימוש עצמו. חצי מהמערכת ה"חיה":



הזיכרון

אמנם נראה מבחוֹץ קטן יותר, אבל מכיל יותר חלקים זזים, אפילו מהפרש והאיןטרפרט ביחד!



אלגוריתמים מרכזיים

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

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 הצד כמות קטנה יחסית של מידע, ושומרת הצד השני ערכים – מצביע לתחילה stack המדבר, ומצביע שמעיד על התחלת ה"מסגרת" הנוכחית של stack. במבנה זהה אפשר לשמר מידע בגודלים שרירותיים, בצורה שהערך שנוסף אחרון נמצא גם אחרון, לפני המידע האחרון מנצל. מסגרת חדשה נוצרת כל פעם שנוסף 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, ויכולת לבקש עוד. כשمبرקים לאחסן משתנה, בהתאם לאיימפלמנטציה של ה-Heap, מפונה כמות מסוימת של מקום המתאימה לפחות המשנה, ותלויה באימפלמנטציה, יכולות לאחסן במקומות מסוימים 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 אי-יכולת הפתרונות הקיימים, הלכתי על פתרון אחר, שייתר מתאים לסגנון הפרויקט, וקצת יותר פשוט. קראתי לפתרון "Referrer".

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

ההבדל העיקרי בין הפתרון הזה לאחרים שהזכירתי קודם, שהוא גם הסיבה העיקרי לבחירה בו, היא דואקא זה שהושימוש בפתרון הנוכחי הוא ייחד – בזמן שה-Stack וה-Heap מספקים ביחיד פתרון לאחסון + מאגר אזכורים, כבר הייתה לי מערכת אחסון, ולא רציתי

لتכנת משה שאשתמש רק בחלק منهו, כי זה 1) נראה לא טוב ו2) משאיר מקום לטעות עם מפתחים שימושיים בפרויקט ספריה, ולכן לקחתי את הפתרון המקורי הזה – משומש אך ורק לזכורים, ועשה זאת בצורה מסודרת בלי לחת כמות "הזיהה" של זיכרונו.

חישוב והערכת ביטויים

חלק בלתי נפרד מתכנת זה הערכת ביטויים על מנת לייצר ערך. הבעיה היא, זה לא כל-כך פשוט – יש סדרי פעולה חשובות, סוג הביטוי יכול להשתנות באמצע, וכן הלאה. יש פתרון אחד ידוע לזה, והשתמשתי בגרסת קצרה שלו, על מנת להתחשב בפתרונות מסוימים של השפה (הגדרת אופרטורים בזמן ריצה וכינויים, כמו `+ = define add`) הפתרון קיים, והgresת 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 המסורדר, לפי פורמט שהוחלט מראש (סימן פולני, סימן פולני הפור או עצ syntax). מערך הפלט בננה כך – לכל טוקן בביטוי המקורי:
 - אם הוא מספר, מוסיפים אותו ישיר לפטט
 - אם הוא אופרטור/פונקציה, יש כמה אופציות:
 - אם האופרטור בדרגה גבוהה יותר מהאופרטור שבראש הסטאק, הוא נוסף לסטאק
 - אם האופרטור באותו דרגה/האופרטור החדש בדרגה נמוכה יותר, האופרטור הקיים נוסף לפטט ומוריד מהסטאק, במקומו מוסיפים את האופרטור החדש

מקורות:

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

gresah בפרויקט:

- Shunting Yard (Inspired) – האלגוריתם מפוץ 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, מהחובבים ביותר להכי פשוטים, וכך יצא שטוקנים בעלי אופרטור חשוב יותר, לדוגמה, (חזקה) יקובצו מוקדם יותר מכאליה ברמה נמוכה יותר, לדוגמה, $+$. בעצם, כל הפעולה נעשית קצר כמו `min place-in`. בסוף, יצא שביטוי שנכתב ככה:
 - $2 + 5! * 3 ^ 4 - 4$

יצא ככה:

$$((4 - 5) ^ 3 * (5 + 2))$$

- calculate – יוצרים 3 משתנים – ערך נכון, אופרטור נכון, ערך עד נכון. עוברים על עצם `group` מייצר, ועובד כל טוקן:
 - אם הוא מספר, מציבים אותו בערך נכון, ואם יש ביטוי, קוראים שוב לעלי `calculate`

- אם יש כבר אופרטור, זה אומר שהביטוי חייב להסתיים. מחשבים את הביטוי כאשר "ערך עד עציו" בצד שמאל ו"ערך נוכחי" בצד ימין, ומציבים את התוצאה ב"ערך עד עציו".
- אם אין אופרטור ו"ערך עד עציו" לא מוגדר, ערכו הופך לערך זהה.
- אם אין אופרטור ויש "ערך עד עציו" נזרקת שגיאה שנייה מספרים באו אחד אחריו השני.
- אם הוא אופרטור מציבים אותו ב"אופרטור נוכחי".
- אם האופרטור הוא הטוקן האחרון, מחשבים את הביטוי כאשר "ערך עד עציו" בהתחלה, והאופרטור בסוף.

ההבדל העיקרי בין האלגוריתם שלי לאחד הקלאסי הוא השני מאחסון המידע בסטאק ופלט למודיפיקציה *in-place* (גם לאחד הקלאסי יש גרסה של אחסון באמצעות עץ syntax דומה לאחד שאני מרכיב, لكن זה לא הבדל). הסיבה היחידה להבדל היא נוחות ועקביות – הרוב המוחלט של הפROYKT מתכונת בעדרת פונקציות רקורסיביות, ואני מרגיש מאד בנוח אי-יתן, אך החלטתי שגם האימפלמנטציה של האלגוריתם זהה יהיה רקורסיבי.

Hashing

כשתכננתי ניהול זיכרון, החלטתי שאובייקטים יאוחסנו כמפות hash של השדות שלהם. לכן, הייתה צריכה צריך להחילט על אלגוריתם hashing. ישנן יתרונות רבים אופציוניים, אך לקחתי את סבירה יחסית, מאילוצים:

- **SipHash** – אלגוריתם hashing המשמש בשיטת *Xor-Add-Rotate* על מנת לייצר hashים. אלגוריתם זה אינו קריפטוגרפי, או במילים אחרות, אינו ראוי להצפנה. הוא קיים על מנת להציג אלגוריתם אולטרנטיבי שטוב למלחמה במתכיפות התנגשות, בהם התוקף מנסה למצאו שני ערכים המנסים למצואו את אותו hash. אם משתמשים ב-SipHash, גם אם התוקף ידוע את המפתח ואת תוצאת האלגוריתם, או את ערך שעושים לו hashing ואת תוצאת האלגוריתם, לא יהיה ניתן לנחש את הנตอน 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 לה Tangensio, אך לא חסינה למתכיפות מכוננות. 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 אולטרנטיביות. פשוט גרסה יותר טובה של MurMur1 עם יותר אופציות
 - MurMur3 – הגרסה השלישית והעדכנית. שיפור על הגרסה השנייה, מציע גם ליצור hashים באורך 128 בתים.

מקורות:

<https://en.wikipedia.org/wiki/MurmurHash>

בחורתி באלגוריתם MurMur1 משפחחת MurMur – ספציפית בחורתி בסוג האלגוריתמים מיילוץ, ובחורתி בדור הראשון מכיוון שהוא התאים טוב למקירה שלי – רציתי להשתמש באלגוריתם כדי לייצר אינדקסים במפתח $hash$, והדור הראשון מייצר מספרים באורך 32 בתים, שלמקירה השימוש שלי מספיק.

סביבת עבודה

פיתוח

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

- **Haxe Compiler** (כרגע נמצא ב-0.0.4 עד 1. – גרסה nightly)
- **magitahab**, בטא של גרסה 2.0.0 (פחות)
- **vision** – גיטראב, גרסה 1.0.1 – גרסה hash

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

```
Haxelib install <library name>
```

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

```
Haxelib git <library name> <link to repository>
```

בדיקה

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

- **לקוח אינטרנט** – יש לשים לב שהשורות הבאות בקובץ `hxxml.compile` מבוטלות (באמצעות הוספה `#` בתחילתן):

```
#--interp
#--define unit
#--js interp.js
```

והשורה ה зат פועלת (באמצעות מחיקת `#`)

לאחר מכן, בונים את התוכנה באמצעות `Haxe compile.hxxml`, ונכנסים/מרעננים את הקובץ `index.html` הנמצא בשורש של הפרויקט.

- **לקוח שורת פקודה** – יש להחילף את `-#` בין השורה של `interp` ל `--js`, וכן להשאר את `unit` מבוטל:

```
--interp
#--js interp.js
#--define unit
```

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

- **לקוח שורת פקודה** – יש לשים לב ש `unit` מבוטל, ו `--js interp.js` לא מבוטלים:

```
--define unit
--interp
#--js interp.js
--define unit
```

לאחר מכן, בונים את הפרויקט, וכל 14 הבדיקות יורצו, אחת אחר השנייה, וידוחו עם הבדיקה הצליחה, ואם לא מה ערך הלא צפוי שהוחזר, עץ `hxax`, השם, הערך, 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. ישנו 3 חלקים "מג'וריים", בעוד אחד מהם מכיל בתוכו עוד חלק גדול יחסית (הזכרן). אתייחס לחלק זהה ומחלקה-הורה שלו בלבד.

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 קומפליציה בלבד, מתكمפל בסוף לסוג אחר שהוא בניו עליון. שדות סטטיים קיימים בסוף על אובייקט נפרד, והקוד של שדות תלויי מקרה מוצבים בשורה בה מבוקש הערך. נמצא בשימוש מטعمי מהירות.
- ספירה (enum) – הספירה הקלאסית, בתוספת התכונה המיוחדת של יצירת [algebraic data types](#).
- ספירה אבסטרקטית – הספירה הקלאסית, ללא פיצ'רים נוספים. כל אלמנט מיוצג ע"י ערך מסווג אחר. מתكمפל בסוף לסוג שהוא בניו עליון, באותה התנהגות כמו אבסטרקט רגיל.
- הגדרה (typedef) – דרך מיוחדת ליצור כינוי, או לחתם מבנה קונקרטי לאובייקט דינמי, כך שהיא חייב להכיל את השדות המזוכרים בהגדירה. ערכו יכול להיות או סוג/אבסטרקט/וכו, או "אובייקט" בעל שדות המוגדר באמצעות סוגים מסווגים.

[קישור להסביר מורחב](#)

חבילה – little.lexer

- ספירה – LexerTokens – מכיל את הטוקנים המשמשת בהם המחלקה Lexer, על מנת להפוך קוד בפורמט מחורזת, לקוד המפוץל למזהים:
 - Identifier(name:String) – מייצג מזהה שמו <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 מחורזת 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 מחורזת המכילה קוד (הכתוב בLittle) למערך מזהים מסווג LexerTokens:
 - <LexerTokens>(code:String):Array<LexerTokens> – מקבל קוד הכתוב בLittle, וממיר אותו למערך מעובד של טוקנים מסווג LexerTokens, בעזרה הפונקציות הבאות. עובר על הקוד, אותן אחר אותן.

- separateBooleanIdentifiers(tokens:Array<LexerTokens>):Array<L> – מקבל מערך של טוקנים מסווג LexerTokens, ומזהה אילו מה-(Identifier(name:String)) מכילים בתוך ה-nam שליהם ערכים בוליאניים, ומפצל אותם ל-nullValue או Boolean.
- mergeOrSplitKnownSigns(tokens:Array<LexerTokens>):Array<Le> – מקבל מערך של טוקנים מסווג LexerTokens, ומזהה מתי קומבינציית סימנים היא נכונה, ומתי צריך לפצל אותה. זה הכרחי, מכיוון שלא כל האופרטורים הם אות אחת, ולכן נדרש מתי האופרטור הוא בודד ומתי הוא מורכב מסויים.

חvíלה – little.parser

- ספירה – ParserTokens – מכיל את הטוקנים משתמשת בהם המחלקה ParserTokens, על מנת להפוך את הקוד מקבוצת מזהים לעץ syntax שלו.
- SetLine(line:Int) – טוקן המביע צורך לשנות את מספר השורה הנוכחי לאחר מספר אחר SplitLine – מביע צורך להפסיק את קריאת השורה ברגע, להגבר את מספר החלק הנוכחי בשורה הנוכחי.
- Variable(name:ParserTokens, type:ParserTokens, ?doc:ParserTokens) – מבטא יצירת משתנה, שםנו, סוגו והדוקומנטציה שלו מבוטאות בעזרת טוקנים שונים. אולם הטוקנים ח"בם "להיפטר" למזהה/תגובה, בהתאם לפרמטר.
- Function(name:ParserTokens, params:ParserTokens, type:ParserTokens, ?doc:ParserTokens) – מבטא יצירת פונקציה בשם, סוגה, והדוקומנטציה שלה מבוטאים בעזרת טוקנים, הנפרטים למזהים/תגובות, בהתאם למקרה. params נדרש להיות טוקן מסווג Variable, PartArray, ConditionCall(name:ParserTokens, exp:ParserTokens, body:ParserTokens) – מבטא קריאה לתנאי שבו מבוטא בעזרת טוקנים הנפרטים לטוקן מסווג Identifier, Identifier "הפרמטר" של התנאי יכול להיות כל טוקן, והגוף של התנאי יהיה כל טוקן הנitin להריצה, בד"כ טוקן מסווג Block.
- Write(assigns:Array<ParserTokens>, value:ParserTokens) – טוקן המציג כתיבה של ערך, value, לקבוצה של מזהים, או לפחות טוקנים שאמורים להיפטר למזהים (טוקנים מסווג Identifier).
- Identifier(word:String) – מייצג את המזהה. בד"כ, כשתקלים בו בקוד, יש למשוך את ערכו מהזיכרון בהתאם לשם.
- TypeDeclaration(value:ParserTokens, type:ParserTokens) – מייצג ייצקת ערך מסוים לסוג שאינו בהכרח של אותו ערך. לא מבטיח התאמתו בין הערך לסוג.
- FunctionCall(name:ParserTokens, params:ParserTokens) – מבטא קריאה לפונקציה, כאשר שם מבוטא באמצעות טוקן צזה או אחר הנפטר Identifier, Identifier מובאים באמצעות טוקן מסווג PartArray.
- Return(value:ParserTokens, type:ParserTokens) – מבטא החזרת ערך בפונקציה – כמשמעותו, אמור לסיים הריצה לפני שכל הטוקנים סיימו להריץ

- Expression(parts:Array<ParserTokens>, type:ParserTokens) ○
mbatא מספר טוקנים אחד אחרי השני, המוחברים בהיוותם בתוך סוגרים רגילים.
- (– Block(body:Array<ParserTokens>, type:ParserTokens) ○
מספר טוקנים, שמטרתם להיות מורצים, אחד אחרי השני. מבוטא בעזרת קוד הסגור בסוגרים מסולסים
- (– PartArray(parts:Array<ParserTokens>) – mbatא סתם מערך של טוקנים. לא יכול להימצא בקוד רגיל, נוצר באופן מלאכותי ע"י Parsern.
- PropertyAccess(name:ParserTokens, property:ParserTokens) ○
mbatא גישה לשדה על טוקן אחר. הפעמטר name יכול להיות עצמוני, PropertyAccess, ובמקרה זה יש גישה "מושלשת" אל תוך שדה.
- <sign> – מייצג אופרטור שסימנו <num>
- Number(num:String) ○
מייצג מספר שלם שערכו <num>
- Decimal(num:String) ○
מייצג מספר עשרוני שערכו <num>
- Characters(string:String) ○
מייצג מחוזת שמכילה את האותיות <string>
- Documentation(doc:String) ○
מייצג דוקומנטציה
- <msg> – מייצג טוקן שגיאה האומר NullValue ○
מייצג את הערך null
- TrueValue ○
מייצג את הערך הבוליאני true
- FalseValue ○
מייצג את הערך הבוליאני false
- (– Custom(name:String, params:Array<ParserTokens>) ○
טוקן שימושים בשפה בתור כל יכולם לצור בעצמם, וכך לעזור להם השחלת מאקרים.
- מחלקת – Parser – מכיל פונקציות המאפשרות המירה של מערך המזהים שקיבלנו Lexern, לעץ syntax מפורט ושלם.
- >- additionalParsingLevels:Array<Array<ParserTokens>> – Array<ParserTokens> ○
מערך של שלבי עיבוד נוספים למידע שמייצרת הפונקציה parse. מרכיב מפונקציות, שיקראו מיד אחרי שהבניה ה"רגילה" הסתינימה.
- parse(lexerTokens:Array<LexerTokens>):Array<ParserTokens> ○
לקח קוד המוצג באמצעות מערך מזהים, ממיר אותו לטוקנים מסווג ParserTokens, וublisher אותו מספר שלבי בנייה, עד שנוצר עץ syntax שלם. שלבי הבניה הון הפונקציות הבאות, ואחריהם באות אלה שנמצאות additionalParsingLevels.
- convert(lexerTokens:Array<LexerTokens>):Array<ParserTokens> ○
שלב ההמרה – עובר על כל הטוקנים מסווג LexerTokens, ממיר אותם, ומחדיר מערך של טוקנים זהים המשמשות מסווג ParserTokens.
- mergeBlocks(pre:Array<ParserTokens>):Array<ParserTokens> ○
ממיר כל קבוצת טוקנים המוקפת בסוגרים מסולסים לטוקן מסווג Block.
- mergeExpressions(pre:Array<ParserTokens>):Array<ParserTokens> ○
> – ממיר כל קבוצת טוקנים המוקפת בסוגרים רגילים לטוקן מסווג Expression.
- mergePropertyOperations(pre:Array<ParserTokens>):Array<ParserTokens> ○
> – מאחד כל מקירה של טוקנים שיש ביניהם אופרטור שאמור להיעד על גישה לשדה, לטוקן מסווג PropertyAccess.

- mergeTypeDecls(pre:Array<ParserTokens>):Array<ParserTokens> ○
 > – מאחד כל מקרה שיש מזהה שימושתו "הדגשת" סוג, ואחריו טוקן TypeDeclaration
- mergeComplexStructures(pre:Array<ParserTokens>):Array<ParserTokens> ○
 > – יוצר הרכזות משתנים, פונקציות, החזרות פונקציה וקריאות לתנאים לפי רצפים מסוימים של טוקנים בקוד.
- mergeCalls(pre:Array<ParserTokens>):Array<ParserTokens> ○
 משלב טוקנים ספציפיים שאחריהם סוגרים ללא הפרדה לטוקן FunctionCall.
- mergeWrites(pre:Array<ParserTokens>):Array<ParserTokens> ○
 משלב רצפי השוואה לטוקן מסווג Write
- mergeValuesWithTypeDeclarations(pre:Array<ParserTokens>):Array<ParserTokens> ○
 – משלב טוקנים שאחריהם בא TypeDeclaration ללא ערך, לחסוך TypeDeclaration ייחיד עם ערך.
- mergeNonBlockBodies(pre:Array<ParserTokens>):Array<ParserTokens> ○
 > – מתקן מקרים של קריאות לתנאים בהם מקום לספק בלוק של קוד, המתכונת מספק שורת קוד/טוקן אחר.
- mergeElses(pre:Array<ParserTokens>):Array<ParserTokens> ○
 פונקציית הדוגמה שבתוֹךְ additionalParsingLevels, מוסיף פיצ'ר – תומכת ברכפי else אחרי תנאי if.
- line:Int – השורה הנוכחית עליה Parser נמצא ○
 על מנת למש את הטוקנים ולהפוך אותם לתוכנה עובדת.
- SetLine(line:Int) – מייצג שינוי בשורה הנוכחית בו Parser נמצא ○
 – עורף את השורה הנוכחית, מפסיק את החלק הנוכחי.
- nextPart() – מגביר את החלק הנוכחי. ○
 – מוסיף את השורה הנוכחית, מפסיק את resetLines(). ○

חביבה – little.interpreter

- ספירה – InterpTokens – מכיל את הטוקנים משתמשות בהם מחלקות ההרצה, על מנת למש את הטוקנים ולהפוך אותם לתוכנה עובדת.
- SetLine(line:Int) – מייצג שינוי בשורה הנוכחית. ○
 – מייצג מעבר לחלק הבא בשורה הנוכחית.
- SplitLine – מייצג מעבר לחלק הבא בשורה הנוכחית. ○
 VariableDeclaration(name:InterpTokens, type:InterpTokens, ?doc:InterpTokens) – מייצג הכרזת משתנה, שמו, סוג וודוקומנטציה שלו מבוטאות בעזרת טוקנים שונים. ○
 FunctionDeclaration(name:InterpTokens, params:InterpTokens, type:InterpTokens, ?doc:InterpTokens) – מייצג הכרזת פונקציה, שמה, סוג ההחזקה שלה וודוקומנטציה שלה מבוטאים בעזרת טוקנים שונים. שדה הפרמטרים חייב להיות טוקן מסווג 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 הריצה exp ובלוק הקוד body ○
 body exp 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 ששם מבוטא באמצעות name עם הפרמטרים params.
- FunctionReturn(value:InterpTokens, type:InterpTokens) – מייצג טענת יציאה של פונקציה. הרצת בלוק של קוד מופסקת בעת התיקות בטוקן זה.
- Write(assignees:Array<InterpTokens>, value:InterpTokens) – מייצג כתיבה של ערך לאlementים המוכלים בתוך assignees.
- TypeCast(value:InterpTokens, type:InterpTokens) – מייצג ניסוי יציקה של הערך value אל תוך סוג המבוטא באמצעות type.
- Expression(parts:Array<InterpTokens>, type:InterpTokens) – מייצג רצף פעולות המקבוצות בתוך סוגרים רגילים
- Block(body:Array<InterpTokens>, type:InterpTokens) – מייצג רצף פקודות, המקבוצות בתוך סוגרים מסולסים.
- PartArray(parts:Array<InterpTokens>) – דרך לקבוץ מספר אלמנטים אל תוך טוקן ייחד בלבד להסיף metadata לגביו.
- PropertyAccess(name:InterpTokens, property:InterpTokens) – מייצג גישה לשדה המבוטא בעזרת property שנמצא על ההורה אשר מבוטא באמצעות 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 מהrozenת.
- Documentation(doc:String) – מייצג תగובה.
- ClassPointer(pointer:MemoryPointer) – מייצג את הערך של סוג, באמצעות מצביע אליו.
- Sign(sign:String) – מייצג את הערך של אופרטור NullValue – ערך ה-null.
- TrueValue – ערך המציג אמת FalseValue – ערך המציג שקר
- Identifier(word:String) – מקבץ אותיות ללא הקשר מסוים, בד"כ מייצג גישה לאלמנט כלשהו בזיכרון, או השימוש במילה 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 المسؤولות "LAGSH: בין הטוקנים של InterpTokens למה שהם מייצגים ממשית.
- convert(pre:Rest<little.parser.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, ומוחזיר את אותו טוקן שגיאה. ההחזרה לכאורה מיותרת, שכן קורא

הקוד מפסיק לעבוד בעת שגיאה, אך היא עדין נמצאת שם בשביל מפתחים שמעוניינים לבטל את ה"קritisה", ולעבוד עם ערך השגיאה העצם.

- warn(message:String, layer:Layer) – מעלה שגיאה בצורה של אזהרה, אך היא עדין מודפסת, אך לא עוצרת את התוכנה. מוחזירה את טוקן השגיאה.
- assert(token:InterpTokens, isType:EitherType<InterpTokensSimple, Array<InterpTokensSimple>>, ?errorMessage:String = null) – אומר לקורא הקוד שאנו חובה בrama של קוד ה-*Little*, NullValue. מודיע שהטוקן הנתון הוא מסווג הטוקנים שיופיעו בהמשך הפונקציה. אם הוא לא מהסוג שלהם, נזרקת שגיאה בrama של קוד ה-*NullValue*. אחרת, הטוקן מוחזר ללא שינוי.
- setLine(i:Int) – אומר לקורא הקוד שאנו חובה כרגע בשורה *i*, ומ Lager אירעוני שינוי שורה ושינוי חלק בשורה.
- setModule(m:String) – אומר לקורא הקוד שאנו חובה כרגע במודולה *m*, ומ Lager אירעוני שינוי מודולה אם אכן מודולת הרצה השתנתה.
- splitLine() – אומר לקורא הקוד שהחלק שאנו קוראים כרגע בשורה נגמר (mbotא באמצעות פסיק או נקודה-פסיק). Lager אירע פיצול שורה.
- declareVariable(name:InterpTokens, type:InterpTokens, doc:InterpTokens) – כתוב לזכרון משתנה בשם *name*, סוג *type* וערך NullValue. Lager אירעוני יצירת שדה מסווג משתנה.
- declareFunction(name:InterpTokens, params:InterpTokens, doc:InterpTokens) – כתוב לזכרון פונקציה בשם *name*, וערך קוד של פונקציה המקבל את הפרמטרים *params* מייצג, ולא עושה כלום איתם. קוד הפונקציה מקובל בעדרת פונקציית write. Lager אירעוני יצירת שדה מסווג פונקציה.
- condition(name:InterpTokens, pattern:InterpTokens, body:InterpTokens) – מריץ את התנאי/*לולאה* בשם *name*, באמצעות תנאי הרצה *pattern*, ומחייבים כמה פעמים להריץ את בלוק הקוד *body*. מוחזיר את הטוקן שהחזירה האחורה נגמרה אליו. Lager אירעוני קרייה לתנאי/*לולאה*.
- write(assigees:Array<InterpTokens>, value:InterpTokens) – כתוב את הערך *value* למשתנים בשם *assigees*:
 - למשתנים רגילים, כל ערך יעובד ויוחסן
 - בשביל פונקציות, אם הערך הוא בלוק של קוד, הוא יעורר חלק מערך של הפונקציה, ובכך ישלים את declareFunction.
 Lager אירעוני כתיבה לערך אם השמות אליהם הוא כתוב.
- call(name:InterpTokens, params:InterpTokens) – קורא לפונקציה בשם *name* עם הפרמטרים שבתוך ההession או Little.runtime.callStack.params, PartArray. מוסיף entry באוף זמני ל-*callStack*. Lager אירעוני קרייה לפונקציה.
- read(name:InterpTokens) – מוחזיר את הערך של משתנה בשם *name* בזיכרון.
- typeCast(value:InterpTokens, type:InterpTokens) – מוחזשת פונקציית ייצקה מתאימה. אם לפי הערך והסוג שמקבלים, הפונקציה מוחזשת פונקציית ייצקה מתאימה. אם

יש היא מפעילה אותה, אם אין נדרקת שגיאה שפונקציית יציקה לא קיימת.
משגר אירועי יציקת ערך לסוג.

- `run(body:Array<InterpTokens>, propagateReturns false):InterpTokens` – מרץ בלוק של קוד בעזרת הפונקציות שהוזכו לעיל. אם נתקל ב-`FunctionReturn`, `propagateReturns` הואאמת, מפסיק הריצה מחזיר את מלאו הטוקן. אחרת, מעריך את טוקן החזרה ומוחזיר את ההערכה. אם בכלל לא נתקלים בטוקן החזרה, הערך האחרון שעובד מוחזר.
- `evaluate(exp:InterpTokens, ?dontThrow:Bool, ?tokens:InterpTokens false):InterpTokens` – מחלץ את הערך המעובד של טוקן יחיד. משתמש במגוון הפונקציות לעיל כנדרך. כ-`eval` מופעל, שגיאות שימושicas ע"י הפונקציה לא נדרקות 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 ביןיהם אופרטורים, ומוחזיר כמינ'ע, המכיל אחד בתוך השני, ועד עד ל-3 אלמנטים, המבטאים את הביטוי המתמטי.

- **מחלקה – ByteCode** – אחראית על הפיכת קוד בצורה טוקנים, לקוד בתים קומפקטי, שאפשר לאחסן בזיכרון יחסית בקלות
 - `compile(...tokens:InterpTokens):String` – לוקח קוד בפורמט של עץ `String`, ומוחזיר אותו בפורמט קוד בתים, `syntax`.
 - `<decompile(bytecode:String):Array<InterpTokens>` – לוקח קוד-בתים והואופך 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 האינדיידואליים שניסו להדפיס ל-`stdout`, ללא מודifikציה מעבר לעיבוד.
 - `reset()` – מאפס את השדות שנאמרו להלן.
- **מחלקה – Runtime** – מכיל מידע על זמן הריצה, ומידע לאחר הריצה.
 - `line(default, null):Int` – השורה שאotta אנו מתחילהם לקרוא.
 - `linePart(default, null):Int` – החלק בשורה אותו אנו מתחילהם לקרוא.
 - `shora מפוצלת בעזרת פסיקים או נקודת-פסיק.`
 - `currentToken(default, null):InterpTokens` – הטוקן אותו אנו הולכים להריץ.
 - `module(default, null):String` – המודוליה בה אנו מರיצים 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 נדרקה שגיאה או לא
 - `errorToken(default, null):InterpTokens` – טוקן השגיאה שנזרק.
 - `<onLineChanged:Int -> Void` – אירוע המשוגר כאשר שורה מתחילה להיקרא

- ○ אירוע המשוגר כאשר מודולה משתנה. יכול לקרות כאשר מרכיבים פונקציות מודוליות אחרות.
- ○ אירוע המשוגר כאשר אנו מתחלים לקרוא חלק מסוימת שורה.
- ○ אירוע המשוגר גם כמחליפים שורה.
- ○ אירוע המשוגר מיד לאחר שהוא מוציאים להרץ טוקן.
- ○ אירוע המשוגר מיד לאחר זריקת שגיאה.
- ○ אירוע המשוגר מיד לאחר העלאת אזהרה.
- ○ אירוע המשוגר מיד לאחר שכתב ערך לאחד או יותר משתנים/פונקציות/שדות.
- ○ אירוע המשוגר מיד לאחר הרצת פונקציה
- ○ onConditionCalled:Array<(String, Array<InterpTokens>,>
- ○ Void -> (InterpTokens) – אירוע המשוגר מיד לפניה קרייה לתנאי/לולאה.
- ○ onFieldDeclared:Array<(String, FieldDeclarationType)>
- ○ אירוע המשוגר מיד לאחר שמוכחה משתנה או פונקציה.
- ○ Void -> (InterpTokens, String) – 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 קרייה לפונקציה.
- ○ throwError(token:InterpTokens, ?layer:Layer) – זורק שגיאה משכבה מסוימת, ועוצר את ההרצה.
- ○ warn(token:InterpTokens, ?layer:Layer = INTERPRETER) – מופיע שגיאה משכבה מסוימת, ולא עוצר את ההרצה.
- ○ print(item:String) – מופיע מחרוזת כלשהי.
- ○ broadcast(item:String) – פונקציה שאפשר לעורר, שווה ערך פונקציונאלית לprint. נועד למפתחים.
- ○ ספירה – FieldDeclarartionType – משומש ע"י אירוע ההכרזה.
- ○ VARIABLE – הכרזה משתנה. זמין באופן ישיר בפרויקט.
- ○ FUNCTION – הכרזה פונקציה. זמין באופן ישיר בפרויקט.
- ○ CLASS – הכרזה סוג. לא זמין באופן ישיר בפרויקט.
- ○ CONDITION – הכרזה תנאי/לולאה. לא זמין בגרסה זו של הפרויקט.
- ○ OPERATOR – הכרזה אופרטור. לא זמין בגרסה זו של הפרויקט.

| little.interpreter.memory |

- ○ אבסטרקט – MemoryPointer – מייצג כתובות בזכרון, פועל כמספר 32 ביטים לאחר קומפיילציה.
- ○ rawLocation(get, set):Int – המספר שמתוחת לMemoryPointer זהה.
- ○ () – ממיר את הכתובת למחרוזת

- `IntArray toArray():Array<Int>` – ממיר את הכתובת לערך של ביטים, שערכם מה-
127 128.
- `Bytes toBytes():Bytes` – ממיר את הכתובת לערך ביטים אמיית'.
- `Int rawLocation – toInt():Int` – רק כפונקציה.
- **מחלקה – Memory** – נבנה אבסטרקציה על שאר מבנה הזיכרון – מכיל פונקציות
גישה, כתיבה ומחסן ערכים באורה קלה עם דוקומנטציה ותగובות נרחבות.
- `Storage storage:Storage` – האחסון שימוש בו הזיכרון.
- `Referrer referrer:Referrer` – משמש בו הזיכרון.
- `ExternalInterfacing interface – externs:Externs` – האקסטרנים משמשים בו
הזיכרון.
- `ConstantPool constants:ConstantPool` – ברירת הקבועים משמשים בו הזיכרון.
- `Operators operator:Operators` – עוד `interface` לאקסטרנים, הפעם מעוצב
לאופרטורים.
- `Int memoryChunkSize = 512` – כל פעם שמדובר בכתובת מסוימת של
זיכרון, משתמש בערך זהה כדי לדעת כמה.
- `Int maxMemorySize = 1024 * 1024 * 1024` – כמהו הזיכרון המקסימלית
שהזיכרון מסוגל לבקש.
- `Int currentMemorySize(get, never):Int` – כמהו הזיכרון בשימוש כרגע,
בקפיצות של `memoryChunkSize`.
- `void reset(storage, referrer, externs, constants)` – מאפס או יוצר מחדש את השדות,
.constants
- `MemoryPointer store(token:InterpTokens):MemoryPointer` – מאחסן ערך פשוט
בזיכרון, ומחייב מצביע אליו. לא מסוגל לעבד מזהים, רק ערכים
- `MemoryPointer retrieve(token:InterpTokens):MemoryPointer` – כמו `store`, רק גם
מסוגל לעקוב אחר מזהים, ועוקב אחרי חוקי pass-by-value/reference.
- `String read(...path:String):{objectValue:InterpTokens,
objectTypeName:String, objectAddress:MemoryPointer}` –
משתמש באחסוני הזיכרון הזמינים כדי להציג את הערך, מיקום והסוג של
שדה כלשהו הנמצא בסוף `path`.
- `void readFrom(value:{objectValue:InterpTokens,
objectAddress:MemoryPointer}, ...path:String)` – הרחבה של `read`,
המסוגלת להתחיל קריאה מערך סטטי ולא רק מזהה. שימוש במרקירים כמו
6.toString() ב�� ההפונקציה `read` תנסה לגשת למשתנה ששמו 6 ותכשל.
- `void 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, נתונים ערכי "לא ידוע"
לשדה (סוג `NullValue`, ערך `Unknown/Anything`).
- `void set(path:Array<String>, ?value:InterpTokens, ?type:String,
?doc:String)` – עורך שדה קיימ 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.
- `MemoryPointer allocate(size:Int):MemoryPointer` – מבקש שלמור על כמהו מסוימת של
ביטים, ומחייב את הכתובת של ההתחלה שלהם.
- `void free(pointer:MemoryPointer, size:Int)` – מפנה כמהו מסוימת של ביטים
השוכנים בכתובת מסוימת.

- מגלה את `sizeOf(pointer:MemoryPointer, type:String):Null<Int>`
- הגודל של ערך מסוים הנמצא במקום מסוים.
- `getTypeInfo(name:String):TypeInfo` – מחלץ מידע על סוגים,
- בים אם הם מאוחסנים בזיכרון אקסטרני או רגיל. המידע כולל שם, שדות
- שונים, גודל אובייקט מסווגם, metadata וכו' לגבי הסוג.
- `storage(storageName:pointer:MemoryPointer):String` – משתמש בזאג –
- `getStorageName(pointer:MemoryPointer)` – על מנת לחץ שם של סוג בעזרת כתובתו בלבד.
- **מחלקה - Storage** – אחראית על הכנסת נתונים לזיכרון, וידוא שלא נכתב שום דבר מעליים כשלא צפוי שזה יקרה
 - `storage:ByteArray` – מערכת הבטים המשמשת בו המחלקה
 - `backreference:parent:Memory` – מעריך בתים המציג את המקומות הפנויים.
 - `reserved:ByteArray` – ממערך בתים מכוון מכתובת מסוימת,
 - והתפקידים.
- `requestMemory(memoryChunkSize:Int):MemoryPointer` – מבקש עוד מemoryChunkSize ביטים מהמערכת.
- `storeByte(b:Int):MemoryPointer` – מאהשן ביט, ומוחזיר את הכתובת שלו.
- `setByte(address:MemoryPointer, b:Int)` – מכניס ביט בכתובת מסוימת.
- `readByte(address:MemoryPointer):Int` – קורא ביט מכתובת מסוימת, ומוחזיר את ערכו
- `freeByte(address:MemoryPointer)` – מפנה ביט הנמצא בכתובת מסוימת
- `storeBytes(size:Int, ?b:ByteArray):MemoryPointer` – מאהשן מערך ביטים, ומוחזיר את הכתובת שלהם
- `setBytes(address:MemoryPointer, bytes:ByteArray)` – מכניס מערך בתים בכתובת מסוימת.
- `readBytes(address:MemoryPointer, size:Int):ByteArray` – קורא מערך בתים מכתובת מסוימת, ומוחזיר אותו
- `freeBytes(address:MemoryPointer, size:Int)` – מפנה מערך בתים הנמצא בכתובת מסוימת
- `storeArray(length:Int, elementSize:Int, ?defaultElement:ByteArray):MemoryPointer` – מאהשן מערך, ומוחזיר את הכתובת שלו. מערך מאוחשן כארוך 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, ומוחזיר אותו
- `freeArray(address:MemoryPointer)` – מפנה מערך הנמצא בכתובת מסוימת
- `storeInt16(b:Int):MemoryPointer` – מאהשן מספר שלם 16 בתים בעל סימן, ומוחזיר 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, ומוחזיר את ערכו
- `freeInt16(address:MemoryPointer)` – מפנה מספר שלם 16 בתים בעל סימן הנמצא בכתובת מסוימת

- `storeUInt16(b:Int):MemoryPointer` – מוחסן מספר שלם 16 בתים ללא סימן, ומחזיר את הכתובת שלו
- `(b:Int) setUInt16(address:MemoryPointer)` – מכניס את הערך של מספר שלם 16 בתים ללא סימן בכתובת מסוימת.
- `(b:Int) readUInt16(address:MemoryPointer)` – קורא מספר שלם 16 בתים ללא סימן מכתובת מסוימת, ומוחזיר את ערכו
- `(b:Int) freeUInt16(address:MemoryPointer)` – מפנה מספר שלם 16 בתים ללא סימן הנמצא בכתובת מסוימת
- `storeInt32(b:Int):MemoryPointer` – מוחסן מספר שלם 32 בתים בעל סימן, ומחזיר את הכתובת שלו
- `(b:Int) setInt32(address:MemoryPointer)` – מכניס את הערך של מספר שלם 32 בתים בעל סימן בכתובת מסוימת.
- `(b:Int) readInt32(address:MemoryPointer)` – קורא מספר שלם 32 בתים בעל סימן מכתובת מסוימת, ומוחזיר את ערכו
- `(b:Int) freeInt32(address:MemoryPointer)` – מפנה מספר שלם 32 בתים בעל סימן הנמצא בכתובת מסוימת
- `storeUInt32(b:UInt):MemoryPointer` – מוחסן מספר שלם 32 בתים ללא סימן, ומחזיר את הכתובת שלו
- `(b:UInt) setUInt32(address:MemoryPointer, b:UInt)` – מכניס את הערך של מספר שלם 32 בתים ללא סימן בכתובת מסוימת.
- `(b:UInt) readUInt32(address:MemoryPointer)` – קורא מספר שלם 32 בתים ללא סימן מכתובת מסוימת, ומוחזיר את ערכו
- `(b:UInt) freeUInt32(address:MemoryPointer)` – מפנה מספר שלם 32 בתים ללא סימן הנמצא בכתובת מסוימת
- `storeDouble(b:Float):MemoryPointer` – מוחסן מספר עשרוני 64 בתים בעל סימן, ומחזיר את הכתובת שלו
- `(b:Float) setDouble(address:MemoryPointer, b:Float)` – מכניס את הערך של מספר עשרוני 64 בתים בעל סימן בכתובת מסוימת.
- `(b:Float) readDouble(address:MemoryPointer):Float` – קורא מספר עשרוני 64 בתים בעל סימן מכתובת מסוימת, ומוחזיר את ערכו
- `(b:Float) freeDouble(address:MemoryPointer)` – מפנה מספר עשרוני 64 בתים בעל סימן הנמצא בכתובת מסוימת
- `storePointer(p:MemoryPointer):MemoryPointer` – מוחסן כתובת בזיכרון, ומוחזיר את הכתובת שלה
- `(p:MemoryPointer) setPointer(address:MemoryPointer, p:MemoryPointer)` – מכניס את הערך של כתובת בזיכרון בכתובת מסוימת.
- `(p:MemoryPointer) readPointer(address:MemoryPointer):MemoryPointer` – קורא כתובת בזיכרון מכתובת מסוימת, ומוחזיר את ערכיה
- `(p:MemoryPointer) freePointer(address:MemoryPointer)` – מפנה כתובת בזיכרון הנמצאת בכתובת מסוימת
- `storeString(b:String):MemoryPointer` - מוחסן מחרוזת, ומוחזיר את הכתובת שלה. מוחסן כאורך ומיד לאחריו תוכן המחרוזת.
- `(b:String) setString(address:MemoryPointer, b:String)` – מכניס את הערך של מחרוזת בכתובת מסוימת.
- `(b:String) readString(address:MemoryPointer):String` – קורא מחרוזת מכתובת מסוימת, ומוחזיר את ערכיה

- `freeString(address:MemoryPointer)` – מפנה מחוזת הנמצאת בכתובת מסויימת.
- `storeCodeBlock(caller:InterpTokens):MemoryPointer` – מאחסן ערך של פונקציה, ומוחזיר את הכתובת שלו. מאוחסן קוד-בתים.
- `setCodeBlock(address:MemoryPointer, caller:InterpTokens)` – מכניס את הערך של פונקציה בכתובת מסויימת.
- `readCodeBlock(address:MemoryPointer):InterpTokens` – קורא פונקציה מכתובת מסויימת, ומוחזיר את ערכה.
- `freeCodeBlock(address:MemoryPointer)` – מפנה פונקציה הנמצאת בכתובת מסויימת.
- `storeCondition(caller:InterpTokens):MemoryPointer` – מאחסן תנאי/לולאה בזיכרון, ומוחזיר את הכתובת שלו. מאוחסן קוד-בתים.
- `setCondition(address:MemoryPointer, caller:InterpTokens)` – מכניס את הערך של תנאי/לולאה בכתובת מסויימת.
- `readCondition(address:MemoryPointer):InterpTokens` – קורא תנאי/לולאה מכתובת מסויימת, ומוחזיר את ערכה.
- `freeCondition(address:MemoryPointer)` – מפנה תנאי/לולאה הנמצא בכתובת מסויימת.
- `storeSign(sign:String)` – מאחסן אופרטור, ומוחזיר את הכתובת שלו.
- `setSign(address:MemoryPointer, sign:String)` – מכניס אופרטור בכתובת מסויימת.
- `readSign(address:MemoryPointer):InterpTokens` – קורא אופרטור מכתובת מסויימת, ומוחזיר את ערכו.
- `freeSign(address:MemoryPointer)` – מפנה אופרטור הנמצא בכתובת מסויימת.
- `storeStatic(token:InterpTokens):MemoryPointer` – מאחסן אלמנט בעל גודל סטטי/מחוזת, ומוחזיר את הכתובת שבה הערךAOHסן.
- `storeObject(object:InterpTokens):MemoryPointer` – מאחסן אובייקט דינמי, ומוחזיר את הכתובת שלו. אובייקט כזה מאוחסן כ8 ביטים של אורך ומיקום, ומפתח Hash נוספת במקום אחר.
- `setObject(address:MemoryPointer, object:InterpTokens)` – מכניס אובייקט דינמי בכתובת מסויימת.
- `readObject(pointer:MemoryPointer):InterpTokens` – קורא אובייקט דינמי מכתובת מסויימת, ומוחזיר את ערכו.
- `freeObject(pointer:MemoryPointer)` – מפנה אובייקט דינמי הנמצא בכתובת מסויימת.
- `storeType(name:String, statics:Map<String, {value:InterpTokens, documentation:String, type:String}>, instances:Map<String, {documentation:String, type:String}>)` – מאחסן סוג, ומוחזיר את הכתובת שלו. סוג מאוחסן כמו 2 אובייקטים ברצף – שני מפות hash לשדות תלויי מקרה ושדות סטטיים, עליהם שני מצביעים ושני ארכאים.
- `setType(address:MemoryPointer, name:String, statics:Map<String, {value:InterpTokens, documentation:String, type:String}>, instances:Map<String, {documentation:String, type:String}>, {value:InterpTokens, documentation:String, type:String}>)` – מכניס סוג בכתובת מסויימת.

- קורא סוג מ כתובה – `readType(pointer:MemoryPointer):TypeInfo`
- מסויימת, ומחייב את ערכו – `freeType(pointer:MemoryPointer)`
- מחלקת – `Referrer` – עוקבת ומחייבת משתנים לערךם וסוגם באמצעות מסויימת
 - שילובי של סטאק ומפתח `hash`.
 - `Memory backreference – parent:Memory`
 - `bytes:ByteArray` – מערך הבטים שבו המחלקת משתמשת.
 - `currentScopeStart(get, null):Int` – מספר המיצג את המיקום שבו המסגרת העכשוית מתחליל. לא ניתן לערכיה ישירות, יש לעורר את תחילת מערך הביטים על מנת לשנות.
 - `currentScopeLength(get, null):Int` – מספר המיצג את כמות המפתחות הנמצאות במסגרת העכשוית.
 - `header(pushScope())` – מוסיף המכיל את כמות האלמנטים במסגרת הקודמת, וכמות האלמנטים במסגרת זו (0) בתור שני מספרי 16 ביטים. מעדכן את `currentScopeStart`.
 - `currentScopeStart(popScope())` – לא עורך שום מידע מחוץ ל `currentScopeStart` – מחשב לפי הנתונים הזמינים את תחילת המסגרת הלפני-אחרונה, ועורך את `currentScopeStart` לערך זהה.
 - `reference(key:String, address:MemoryPointer, type:String)` – מוסיף את המפתח `key` ומחייב אותו לכתובת וסוג מסוים במסגרת הנוכחית.
 - `dereference(key:String)` – מוחק את המפתח `key` מהמסגרת הנוכחית.
 - `get(key:String):{address:MemoryPointer, type:String}` – ממחזר את הכתובת והסוג המוצמדים למפתח מסוים. אם המפתח לא נמצא במסגרת העכשוית, צוללים למסגרתعمוקה יותר. אם המפתח לא נמצא, נזרקת שגיאה ברמת קוד הhexax 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 במסגרת הנוכחית, צוללים למסגרתעמוקה יותר. אם המפתח לא נמצא, נזרקת שגיאה ברמת קוד הhexax 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 עמקה יותר.
 - `keyValueIterator():KeyValueIterator<String, {address:MemoryPointer, type:String}>` – מחייב איטרטור העובר על כל המפתחות וערךיהן, מתחילה המסגרת המקורית לסוף, ואז למסגרת הבאה, עד שנגמרות המסגרות.
- מחלקת – `HashTables` – אחראי על יצירתי, קריית ועריכת מפות `hash` לאחסון אובייקטים בזיכרון
 - `generateObjectHashTable(pairs:Array<{key:String, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer}>)` – מקבל מערך של חמישיות המיצגות מפתחות וערכים במפתח `hash`. לפי כמות המפתחות, יוצר מפה בגודל מסוים. מחייב את המפה כמערך ביטים.
 - `readObjectHashTable(bytes:ByteArray, ?storage:Storage):Array<{key:Null<String>, keyPointer:MemoryPointer, value:MemoryPointer, type:MemoryPointer}>`

{type:MemoryPointer, doc:MemoryPointer} - מעבד מערך ביטים אל תוך מערך חמישיות המציג את מפת ה-hash של האובייקט. כמשמעות Instance של Storage, משתמשים בו על מנת לקרוא את המפתחות המופיעים במאיצים במאפה בעזרת keyPointer.

- **hashTableHasKey(hashTable:ByteArray, key:String, storage:Storage):Bool** – בודק האם מפת hash מכילה מפתח מסוים. מכיוון שככל מפה מכילה את המפתח באמצעות מצביע, אנו צריכים גם את השגונ – שנותן את המוסgalות לקשר בין הכתובת למחרצת עצמה – בשביל מפתח hash מסויימת שהמצביעים למפתחותיה "מתיחסים" לשני Storage מסוימים, מסוגל לאחר מכן למצוא מפתח מסוים, ע"י הוצאת hash שלו, וגיישה למפה, או חיפוש לפי הצורך.
- **objectAddKey(object:MemoryPointer, key:String, value:MemoryPointer, type:MemoryPointer, doc:MemoryPointer)** – מוסיף entry למפתח hash מסויימת המאוחסנת/storage – הקשורה לאובייקט מסוימים. אם המפה מלאה ביותר בעוגן מסוימים, הקשורה לאובייקט מסוימים. אם המפה מלאה ביותר מ-70%, המפה מאוחסנת מחדש בגודל גדול יותר, והמפה הקודמת מפונה. גודל המפה החדשה יהיה פי 3 מהכמות שזקוקים אליה בשביל מפה מלאה (בשביל 10 מפתחות הולוקחים כל אחד 4 מצביעים (16 ביטים בפרק זה), יבקשו 480 ביטים)

objectSetKey(object:MemoryPointer, key:String, pair:{?value:MemoryPointer, ?type:MemoryPointer, ?doc:MemoryPointer}, storage:Storage) – עורך מפתחות שכבר קיימים במאוף hash של אובייקט מסוים, ונותן לו ערך אחר, לפי הערכים של האובייקט זpair, שכן ערך שבאובייקט זpair שווהillo 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 למפתח hash של אובייקט, וקבלת מפתח באמצעות .hashTableGetKey()

getHashTableOf(objectPointer:MemoryPointer, storage:Storage):ByteArray – פונקציית עזר המקצרת את אחזור מפתח hash של אובייקט לשורה אחת, במקום 3 שורות (לקראת אורך המפה ב-4 ביטים הראשונים, מקום המפה בזיכרון ב-4 האחרים, וקריאה הביטים לפני האורח והמיוקם).

- **מחלקה – ExternallInterfacing** – מגשר בין קריאת פונקציות הכתובות ב-Haxe לערכים של Little, באמצעות "אחסון" אותם פונקציות עצים.
- **Memory backreference – parent:Memory**
- **externToPointer:Map<String, MemoryPointer>** – מפה המשמשת לקישור סוגים אקסטרניים למקומות בזיכרון. בד"כ משומש עם אחסון בית יחיד: storage.storeByte()

- ○ מהמפה לעיל. איטי יותר, כי צריך לייצר את המפה כל פעם שורוצים לגשת להז.
- ○ הפונקציות הקשורות לשנתנים תלויי מקרה. (לדוגמה, “.toLowerCase()”)
 - ○ הפונקציות הקשורות לשירות לאובייקטים וסוגים. (לדוגמה, “Characters.fromCharCode(72)”)
- ○ המסלול/עקב אחריו במידת הצורך (תליי אם המסלול קיים או לא) בעץ הנתון (globalProperties instanceProperties או globalProperties instanceProperties) יוצר את האיבר שבסוף המסלול.
- ○ יוצר במידת הצורך את המסלול בשני העצים, globalProperties instanceProperties.
- ○ בודק האם קיים המסלול path בעץ globalProperties.hasGlobal(...path:String):Bool – בודק האם קיים המסלול path בעץ instanceProperties.hasInstance(...path:String):Bool – בודק האם קיים המסלול path בעץ instanceProperties.getGlobal(...path:{objectValue:InterpTokens, objectAddress:MemoryPointer}) – על אף שהאזור הערך שבכל איבר בעץ זוקק לשני פרמטרים, ערך אובייקט ומיקום בזיכרון, מכיוון שמדובר באובייקט סטטי, אין שימוש בשני הפרמטרים האלה – לכן, המאוחר נקרא עם alluch לערך האובייקט ו-1 – למיקומו, והערך מוחזר.
- ○ מחלוקת – העץ עצמו, אחראי על המבנה של הערכים האקסטרניים. getter:(objectValue:InterpTokens, objectAddress:MemoryPointer) –-> {objectValue:InterpTokens, objectAddress:MemoryPointer} מאוחר הערכים – מקבל את ערך האובייקט ההורה ואת כתובות ההורה, ולפיהם מחזיר ערך כלשהו וכותבו בזיכרון.
- ○ doc:MemoryPointer – הכתובת של הדוקומנטציה של שדה אקסטרני מסויים.
- ○ type:MemoryPointer – הכתובת של הסוג שמחזיר המאוחר של שדה אקסטרני מסויים.
- ○ properties:Map<String, ExtTree> – הילדים של העץ.
- ○ מחלוקת – ConstantPool – "בריכה" של ערכים קבועים מסויימים, שלא יכולים להיות מפונים פעמים, ולא יכולים להשתנות. קיים על מנת לשמור זיכרון.
 - ○ capacity(default, null):Int = 24 – כמות הביטים שהבריכה לוקחת מהסון 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 הבולי אני.

- **DYNAMIC:MemoryPointer = 14** – המצביע לסוג ה"динמי" – מייצג כל סוג אחר, חוץ מ-Unknown.
- **TYPE:MemoryPointer = 15** – המצביע לשוג של מחלקות UNKNOWN:MemoryPointer = 16 – המצביע לשוג שאינו ידוע. קיימ בשבי בлокים של קוד וביטויים שסוגם אינם ידוע בזמן שלפני הרצה עצמה. גם יכול להופיע כאשר משתנה מוכרך ללא סוג או ערך.
- **ERROR:MemoryPointer = 17** – המצביע לשגיאות.
- **EXTERN:MemoryPointer = 18** – מצביע בירית החדל לערים אקסטרניטים שלא ניתן לאחסן בזכרון רגיל. קיים בשביל פונקציות אקסטרניט, שכך לא ניתן לאחסן טוקנים מסווג HaxeExtern בזכרון, מהגבלה של הספרייה הסטנדרטית של Haxe.
- **EMPTY_STRING:MemoryPointer = 19** – המצביע למחוזת הריקה get(token:InterpTokens):MemoryPointer – פונקציית עזר המחליפה "קיר" של switch-case, המחליצה את המצביע של ערך מסוים המובא כטוקן. אם הטוקן לא נמצא בבריכה, נזרקת שגיאה ברמת קוד ה-Haxe.
- **getFromPointer(pointer:MemoryPointer):InterpTokens** – פעולה הפך מ-get, גם זורקת שגיאה ברמת קוד ה-Haxe. hasPointer(pointer:MemoryPointer):Bool – בודקת האם הכתובת הנთונה נמצאת בבריכת הקבועים.
- **hasType(typeName:String)** – בודקת האם סוג מסוים נמצא בתוך ברייכת הקבועים.
- **getPriority(typeName:String):MemoryPointer** – מieżהרא את כתובות הסוג typeName הנמצא בתוך ברייכת הקבועים. אם הוא לא נמצא, נזרקת שגיאה ברמת קוד Haxe.
- **מחלקה – Operators** – הרחבה של ExternallInterfacing המתואמת לאופרטורים במקומם למזהים קלאסיים.
 - **{ priority:Map<Int, Array<{sign:String, side:OperatorType}> } = []** – מפה המחברת בין אופרטור מסוים, למיקום שלו בסדר הפעולות כאשר הוא פועל על צד מסוים.
 - **standard:Map<String, (lhs:InterpTokens, rhs:InterpTokens) -> ()** – מפה המחברת בין אופרטורים הפעילים על שני הצדדים, לcallback שאמור למש את פעולה האופרטור.
 - **rhsOnly:Map<String, (InterpTokens) -> Map<String>** – אותו דבר כמו standard, אך בשביל אופרטורים הפעילים רק עם אופrnd בצד ימין.
 - **lhsOnly:Map<String, (InterpTokens) -> InterpTokens** – אותו דבר כמו standard, אך בשביל אופרטורים הפעילים רק עם אופrnd 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 יכול להיות מגוון דברים: first, last, with, before, after, between
 - **getPriority(op:String, type:OperatorType):Int** – מieżהרא את המיקום של אופרטור מסוים הפעול על צד מסוים בסדר הפעולות.
 - **iterateByPriority():Iterator<Array<{sign:String, side:OperatorType}>}** – איטרטטור העובר על המפה priority לפי דרגות.

- call(lhs:IntpTokens, op:String) – פעולה אופרטור כלשהו עם ארגומנט צד שמאל.
- call(op:String, rhs:IntpTokens) – פעולה אופרטור כלשהו עם ארגומנט צד ימין.
- call(?lhs:IntpTokens = null, op:String, ?rhs:IntpTokens = null) – פעולה אופרטור כלשהו עם שני הצדדים null: IntpTokens.

| little.tools – חבילה |

- מחלקה – Conversion – מספקת פועלות להמרה חלקה בין ערכים בזאת Haxe לトーונים של Little, וכן להפך.
- extractHaxeType(type:ValueType):String – לא קשור ספציפית לlittle, אך אפשר קישור בין טוקן סוג **של Haxe** לשם של הסוג.
- toLittleValue(val:Dynamic):IntpTokens – מנסה להמיר ערך Haxe כלשהו לטוקן Little. כאשר אי אפשר, מחזיר NullValue.
- toHaxeValue(val:IntpTokens):Dynamic – מנסה להמיר טוקן לערך Haxe. כאשר אי אפשר, מחזיר null.
- toLittleType(type:String) – ממיר סוגים בסיסיים בזאת Little Type (String => Characters).
- מחלקה – Extensions – מכיל פונקציות עזר המרחיבות על כליה שכבר קיימות על סוגים כמו `tokens:ParserTokens`, `tokens:ParserTokensSimple` – ועוד.
 - `is(token:ParserTokens, ...tokens:ParserTokensSimple)` – בודק האם השם כותרת של טוקן מסוים מסווג ParserTokens נמצא בתוך הפרמטרים הבאים.
 - `is(token:InterpTokens, ...tokens:InterpTokensSimple)` – בודק האם השם כותרת של טוקן מסוים מסווג 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 הערך(index) שבTOKEN מוגדר token.
- parameter(token:InterpTokens, index:Int):Dynamic – מחלץ את הערך(index) שבTOKEN מוגדר token.
- passedByValue(token:IntpTokens):Bool – בודק האם טוקן מסוים אמרור להיום מעובר לפי ערך.
- passedByReference(token:IntpTokens):Bool – בודק האם אמרור להיות מעובר לפי אזכור.
- staticallyStorable(token:IntpTokens):Bool – בודק האם ניתן לקרוא storage.storeStatic על טוקן מסוים.
- extractIdentifier(token:IntpTokens):String – מייצג טוקן מסוים כמחרוזת.
- asStringPath(token:IntpTokens):Array<String> – מייצג טוקן מסוים שהוא מסלול מזהים כמערך של מזהים.

- asJoinedStringPath(token:InterpTokens):String – לijkח את asStringPath ומיצרף אותו באמצעות Little.keywords.PROPERTY_ACCESS_SIGN
- type(token:InterpTokens):String – מחזיר את הסוג של טוקן מסוים, בהקשר של הרצאה הנוכחיית.
- asObjectToken(o:Map<String, InterpTokens>, typeName:String):InterpTokens – לijkח מפה של מחוזות לטוקנים, יוצר ממנה אובייקט עזרת שם סוג האובייקט.
- asEmptyObject(a:Array<Dynamic>, typeName:String):InterpTokens – יוצר אובייקט ללא שדות מסוג מסוים בעזרת מערך ריק.
- asTokenPath(string:String):InterpTokens – ההפך asJoinedStringPath.
- extractValue(address:MemoryPointer, type:String):InterpTokens – מנסה לקרוא ערך מהזיכרון של הרצאה הנוכחיית בעזרת מיקום וסוג.
- writelnPlace(address:MemoryPointer, value:InterpTokens) – כותב ערך מסוים למקום מסוים בזיכרון, ללא התיחסות להאם המקום פנוי או לא.
- toIdentifierPath(propertyAccess:InterpTokens):Array<InterpToken<Identifiers>> – ממיר טוקן propertyAccess לקבוצה של Identifiers. האלמנט הראשון לא חייב להיות מזהה.
- containsAny<T>(array:Array<T>, func:Boolean):Bool – הרחבה למערך כללי, בודק האם הפונקציה הנתונה מחזירה אמת לאלמנטים כלשהו במערך.
- T<T>:Array<T>(iter:Iterator<T>) – ממיר איטרטור (מבנה הספירה בטוויה של Haxe) למערך של אלמנטים שהוא מייצג.
- ספירה – ParserTokensSimple – הכותרות של אלמנטים ב-ParserTokens
- .case UPPER_SNAKE
- ספירה – InterpTokensSimple – הכותרות של אלמנטים ב-InterpTokens
- .case UPPER_SNAKE
- ספירה אבסטרקטית – Layer – כאשר נדרקת שגיאה או אזהרה במצב debug, מזכיר המיקום שزرק את אותה שגיאה. המיקומות נלקחים מכאן LEXER = "Lexer" – שלב הבניה הראשי.
- PARSER = "Parser" – שלב הבניה השני.
- PARSER_MACRO = "Parser, Macro" – מקורותים בשלב הבניה השני.
- INTERPRETER = "Interpreter" – שלב ההרצה, כללי.
- INTERPRETER_VALUE_EVALUATOR = "Interpreter, Value" – 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 למחוזות.
- INTERPRETER_TOKEN_IDENTIFIER_STRINGIFIER = "Interpreter, Token Identifier Stringifier" – שלב ההרצה, הפיכת מזהה למחוזות.
- MEMORY = "Memory" – זיכרון, כללי.

- .Referrer "MEMORY_REFERRER = "Memory, Referrer
 - Storage "MEMORY_STORAGE = "Memory, Storage
 - MEMORY_EXTERNAL_INTERFACING = "Memory, External Interfacing – זיכרון, אלמנטים אקסטרנליים.
 - – "MEMORY_SIZE_EVALUATOR = "Memory, Size Evaluator
 - זיכרון, חישוב גודל אלמנט.
- getIndexOf(layer:String):Int – ממיר שכבה מסוימת למספרה לפי סדר ההכרזה בספירה האבסטרקטית. משומש exitCode בשפה.
- מחלוקת – Plugins – גישור שני, בין המפתח הרגיל `ExternalInterfacing` וחלק משאר הזיכרון. מאפשר הכנסה טבעית כמה שאפשר של אלמנטים מhex H אל תוך הריצה של קוד Little.
 - registerType(typeName:String, fields>TypeFields) – מכנייס סוג אקסטרני בזמן הריצה, באמצעות צורת כתיבה טבעית אך עדין קרובה למקור כל האפשר. ניתן להזכיר על מפונקציות ומשתנים סטטיים ותלו依 מקרה, וכל אחד מהם יכול להחזיר או רק ערך, או גם ערך וגם כתובות, תלוי ברצון המפתח ובמקרה. ניתן גם לאחסן אובייקטים חדשות סטטיים בדרך קלה.
 - registerVariable(variableName:String, variableType:String, ?documentation:String, ?staticValue:InterpTokens, ?valueGetter:Void -> InterpTokens) – מכרייז על משתנה גלובלי הנמצא בשורש, עם סוג, דוקומנטציה, וערך מסוים. ניתן לתת ערך סטטי, אך גם ניתן לתת ערך שיוכל לשנתנות, בעזרת נתינת פונקציה אחזר לפונקציה זו.
 - registerFunction(functionName:String, ?documentation:String, expectedParameters:EitherType<String, Array<InterpTokens>>, callback:Array<{objectValue:InterpTokens, objectTypeName:String, objectAddress:MemoryPointer}> -> InterpTokens, returnType:String) – מכרייז 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 callback InterpTokens מקבל את כל הפרמטרים שהובאו לפונקציה מהצד של Little גם ערך וגם כתובות.
 - registerCondition(conditionName:String, ?documentation:String, callback:(params:Array<InterpTokens>, body:Array<InterpTokens> -> (InterpTokens, returnType:String)) – מכרייז על תנאי/תנאי, עם דוקומנטציה מסוימת. המפתח יכול לקבל גישה לתנאי ההרצה של התנאי/ולולאה ולגוף שלו, 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 callback, registerVariable, אך בהקשר של משתנה על סוג מסוים. מקבל שדרוג, ומתקבל גם מידע על הוראה 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}>) ->
 registerFunction – InterpTokens, returnType:String)
 אך – דומה ל `registerFunction`, בהקשר של פונקציה על סוג ערך מסוים. כמו מקודם, ה `callback` מקבל שדרוג קל על מנת שיוכן להיות תלוי בהורה שלו.
- (`registerOperator(symbol:String, info:OperatorInfo)` – מכריז על אופרטור, הפעול על צדדים מסוימים, עם עדיפות מסוימת ותומך בסוגים מסוימים, או, בשילוב סוגים בסדר מסוים.
 - הגדרה – OperatorInfo – הגדרות לאופרטור המוכר:
- `lhsAllowedTypes:Array<String?>` - סוגים הערכיים שאפשר להעביר לצד השמאלי של האופרטור.
 - `rhsAllowedTypes:Array<String?>` - סוגים הערכיים שאפשר להעביר לצד הימני של האופרטור.
 - `allowedTypeCombos:Array<{lhs:String, rhs:String?}>` - קומבינציות סוגים מיוחדות שיש בהן תמיכה, מעבר לסוגים המוסכמים.
 - `InterpTokens? -> InterpTokens` – פונקציות אחזור הערך של הפעולה עם האופרטור, כאשר האופרטור מקבל שני צדדים. אם `operatorType` אינו `LHS`, או, גם `singleSidedCallback` קיימ, נזקקות שגיאות הסבר מתאימות.
 - – `singleSidedOperatorCallback:InterpTokens? -> InterpTokens`? – פונקציות אחזור הערך של הפעולה עם האופרטור, כאשר האופרטור מקבל רק צד אחד. אם `operatorType` אינו `RHS`, או, גם `callback` קיימ, נזקקות שגיאות הסבר מתאימות.
 - `operatorType:OperatorType? => operatorType:OperatorType` – סוג האופרטור – מחייב אם הוא מקבל רק צד שמאל (`LHS`), רק צד ימני (`RHS`), או צריך את שניהם (`LHS_RHS`). רק צד שמאל (`LHS`), רק צד ימני (`RHS`), או צריך את שניהם (`LHS_RHS`). רק צד שמאל (`LHS`), רק צד ימני (`RHS`), או צריך את שניהם (`LHS_RHS`).
 - `priority:String` – הדרגה של האופרטור בסדר העדיפויות. יש לשדה זה אפשרות שאמורויות להקל על הכנסת דרגה:
 - אינדקס פשוט: מספר של דרגה, ככל שייתר נמוך הדרגה יותר מוקדמת. 1 - תמיד יהיה הכי גבוהה.
 - `first` – האופרטור בעל הדרגה הנמוכה ביותר, יתייחסו אליו ראשון. `last` – ההפוך מ `first`.
 - `<sign> before <sign>` - יוצרף לדרגה אחת לפני האופרטור המוזכר. על מנת להבדיל בין צדדים, יש להשתמש ב _+ (_+ (`RHS`), _+ (`LHS`), _+ (_+ (`LHS_RHS`)).
 - `before - after <sign>` – ההפך מ `before`, `before - after <sign>` – דרגה אחת אחריו.
 - `- with <sign>` - יוצרף באוטה דרגה כמו האופרטור המוזכר.
 - `<sign> between <sign> <sign>` – האופרטור יוצרף בדיק באמצע בין הדרגות של הסימנים הנתונים. יוצראו דרגות במקורה הצורך.
 - הגדרה – `TypeFields` – שם אחר למפה מאד אורך, שאמור לבטא את אפשרויות `:registerType` הרכבות 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 מתפרק לשוג דינמי, אך לפי קומפליציה מסווגלייז עריך שהוא אחר מ-7 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 הקומפליציה מסרב להתייחס לאbstракטים כלשהם מלל מעלה 8 סוגים בקרה צפוייה. דיווחתי על הבאג למפתחי שרת הקומפליציה.

- מחלקה – PrepareRun – אחראית על ייצרת הספרייה הסטנדרטית
 - prepared:Bool
 - מוסיף סוגים מסוימים, ושודות לכל הסוגים שזקוקים להם
 - מוסיף פונקציות גלובאליות, כמו הדפסה וזריקת שגיאה
 - מוסיף שדות תלויי מקרה גלובליים, כמו type. addressi.
 - מוסיף את כל האופרטורים שהשפה תומכת בהם -of-out-the-box
 - מוסיף את כל הולאות והתנאים שהשפה תומכת בהם.
- מחלקה – PrettyPrinter – מספקת יכולות פירמות והדפסת עץ syntax של קוד ומארבי טוקנים.
 - printParserAst(ast:Array<ParserTokens>, ?spacingBetweenNodes:Int = 6) – מחזיר מחורצת של עץ syntax שמייצג מערך טוקני Parser בצורה יפה, עם רישום מסוים בין כל קומה בעץ.
 - printInterpreterAst(ast:Array<InterpTokens>, ?spacingBetweenNodes:Int = 6) – מחזיר מחורצת של עץ syntax שמייצג מערך טוקני Interpreter בצורה יפה, עם רישום מסוים בין כל קומה בעץ.
 - stringifyParser(?code:Array<ParserTokens>, ?token:ParserTokens) – מחזיר את הקוד המובא כמחרוזת קוד מפורמת.

- `stringifyInterpreter(?code:Array<InterpTokens>, ?token:InterpTokens)` – מחזיר את הקוד המובא כמערך טוקני Interpreter כמחרוזת קוד מפורמת.
- `prettyPrintOperatorPriority(priority:Map<Int, Array<{sign:String, side:OperatorType}>)` –מחזיר מחרוזת המציגת את מפת סדר פעולות של האופרטורים בצורה יפה.
- **מחלקה - TextTools** – מחלקת חיצונית שהעתיקת כי רציתי לבצע בה שינויים. מכילה מגוון פונקציות הרחבה למחרוזות. מכיוון שהמחלקה לא מהפרויקט זהה לא פרט אליה.
- **אבסטרקט – OrderedMap** – מחלקת חיצונית של מפה מסודרת. אין ספריה חדשה עם אימפלמנטציה עובדת, אז [היה צריך ללהתיק מקור חיצוני](#)

מחלקות נוספות

- **מחלקה – Little** – המחלקה העיקרית של הפרויקט, שנوتנת גישה לכל הפונקציות "הכרחיות" לשימוש בסיס. כל עוד מפתח לא מנסה להשתמש בשפה בצורה מתקדמת יותר, אין סיבה שישתמש בכלים שהמחלקה לא מציעה.
- `keywords:KeywordConfig` – מילוט המפתח שהשפה משתמש בהם כאשר יורץ או יבנה קוד
- `runtime(default, null):Runtime` – מכיל מידע על זמן הריצה, אירועי בריצה, ותוצאות אחרי הריצה, כמו טענת יציאה.
- `memory(default, null):Memory` – הזיכרון שישתמש בו מריץ הקוד על מנת לאחסן ערכים.
- `plugins(default, null):Plugins` – הוסף אלמנטים אקסטרניים, ממשתנים רגילים ועוד סוגים ואופרטורים.
- `queue<String>(default, null):Queue<String>` – תור ההרצה, נעשה שימוש כאשר משתמש בloadModule.
- `debug:Bool` – מחייב האם הקוד מורץ במצב debug. במצב זה, כשמודפסים דברים נוספים השכבה ממנה ההדפסה נקראה.
- `version:String` – הגרסתה של בניית הקוד והמריץ. בפרויקט זהה, הוא יהיה `1.0.0-f`, כ-`f` מציין שהפרויקט מאפשר "רק" Functional Programming.
- `loadModule(code:String, name:String, debug:Bool = false, runRightBeforeMain:Bool = false)` – טוען קוד ומריץ אותו לפני בניית הקוד "הרاسي". אם `runRightBeforeMain` מאפשר, אנו מתעלמים מפרמטר `debug` ומושפעים את התור. אם לא, `debug` מופעל באופן זמני בהתאם לערך הפרמטר, והקוד מורץ תחת שם המודולה הנוכחי.
- `(code:String, ?debug:Bool) run(code:String, ?debug:Bool)` – מריץ קוד, וגם מריץ לפני 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, ומוחזיר את עץ ה-`syntaxTree`.
- `format(code:String):String` – בונה וממיר מחדש את הקוד למחרוזת, ובכך מפרמט את הקוד.
- `(reset())` – מפסס את כל האלמנטים שצריך לאפס לפני ריצה חדשה וטרייה (זיכרון, תור...).

- מחלקת – KeywordConfig – כל מילוט המפתח והסימנים שמשתמשת בהם השפה. קיימים בין 50 ל-100, ولكن לא נכתב הסבר על כולם, אלא על אלה 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	○
.Little.run	הקוד של

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	○
CONDITION_BODY_PARAMETER_NAME:String	○
CONDITION_PARAMETER_NAME:String	○
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 אמרו להעיר, אופרטורים শনօփօ մաօխսնիմ կան ուլ մնտ շիօկլօ լիկրա.
PROPERTY_ACCESS_SIGN:String	○
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 הנוכחיית לkonfigורציה הנתונה בפרמטר. אם חלקיים מהקונפיגורציה הנתונה הם null, מתעלמים מהם ולא עורכים את הקונפיגורציה הנוכחיית באמצעותם.

קוד "יפה" ותכנות אלגוריתמיים

קריאה לפונקציות אקסטרניות

בחלק מהפרויקט, מאפשרת קריאה לפונקציות אקסטרניות, באותה דרך ו"קשיות" כמו פונקציה רגילה (כמויות פרמטרים סטטיות, סוגים קונקרטיים לכל פרמטר, סוג החזרה...). בזאת הטוקן HaxeExtern, שמאפשר לנו לה裏ץ קוד Haxe תוך כדי ריצה של קוד Little HaxeExtern. זה נחפר ליחסית קל, אבל הבעיה מתחילה בפרמטרים – בגלל שפרמטרים מובאים לפונקציה בתור "הדבקה" של ההכרזה שלהם לגוף הפונקציה, אי אפשר בקבלה לאחזר את אותן ערכים. לכן, היתי צריך לעשות קצת "ריידיים" בשביב להציג את המידע הזה:

```
var token:InterpTokens = FunctionCode(paramMap, Block([
    FunctionReturn(HaxeExtern(() -> callback(paramMap.keys().toArray().map(key -> memory.read(key))), returnTypeToken)
], returnTypeToken));
```

(הסביר 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 צזה – HaxeExtern מבקש פונקציה ללא פרמטרים ושמחזירה טוקן. callback מחזיר את הערך שאנו צריכים, אך צריך פרמטרים כמעריך. אנו יודעים שככל הפרמטרים צריכים להיות מובאים, ולכן, אנו מבדיקים את ערכי הפרמטרים כך:

- לוקחים את השמות של הפרמטרים כזוררים
- ממירים למערך
- לכל מפתח במערך, קוראים אותו מהזיכרון, ומוחזירים את הערך, המצביע והסוג שלו.

השראה – Shunting Yard

חישוב הביטויים של הפרויקט נעשה באמצעות אלגוריתם בהשראת Shunting Yard, לפחות בחלק החישובי שלו. האימפלמנטציה עצמה מאוד ארוכה (כ-150) שורות, אבל רובה עדין לוגית.

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

הנה הפונקציות. נתחיל דוקא 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 == RHS_ONLY).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 ל calculate

```

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()

הfonקציה מתחילה ישר ולענין, מכיוון שאין ערך בלבדוק למקרי קיצון, הם יכולים יתרהגו נכוון. אופרטורים בזמן הריצה מופרדים לקבוצות לפי סדר פעולות. את סדר הפעולות ואת הסימנים אלו באמת מושכים משם, בשורה:

```

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

```

ואז מתחילה לעבור על הטוקנים של השכבה הנוכחית – כל פעם שהוא מס'םים קבוצה, הטוקנים האלו מעובדים קצת יותר:

```

while (i < pre.length) {

```

מכיוון שאופרטור יכול להתבטא גם כמזהה או בлок, יש ליחס אותם, ורק אז להתחיל:

```

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

```

וכשיו, מתחיל ה"סיבוך" האמתי – בסופו של יומם יש רק 3 מקרים בהם צריך לבנות את התוצאות בצורה שונה, אך אחד מהם מאוד, מאוד סבוך. את החל מהמרקמים הסופיים, שהם התקלות במספר או ביטוי:

```

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

```

המקרה האחרון 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 זה 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 ומאחרה, ובודקים שני מקרים:

- **מקרה ראשון: מלפניו יש אופרטור, והוא באוֹתָה דְּרֶגָּה בְּסִדר הַפְּעֻולּוֹת כְּמוֹ האופרטור זהה.**

מייצג מספר מקרים:

- **סָאֵב-מָקָרָה שְׁנִי:** אנחנו אופרטור חד צדי של צד שמאל, ומפניו אופרטור דו צדי. במקרה זה, אנו פשוט לפקחים את הטוקן האחרון שעיבדנו, ושים אותו איתנו כאשר הטוקן הוא אופrnd שמאל.
- **סָאֵב-מָקָרָה רְאִישׁוֹן:** אנחנו אופרטור דו-צדדי, ומפניו סימנים חד-צדדיים בראף. במקרה זה, אנו עוקבים אחר הסימנים ומוציאים 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 שבנו את המערך והוספנו את האופrnd שאחרי הסימנים, הופכים אותו לאופrnd השמאלי, בעזרה קרייה רקורסיבית לgroup על המערך שבינו (כשהיאבר האחרון במערך זה האופrnd שקובוצת האופרטורים חד צדיים פועלם עליו):

```

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

```

עכשו שיש לנו את הצד הימני של האופרטור שאנו עובדים עליו כרגע, וכבר לפני זה יש לנו את האופrnd השמאלי:

```

var operand1 = post.pop();

```

לקחhim את הטוקן הקודם, ובונים PartArray על הכל:

```

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

```

- **מקרה שני: מלפניו טוקן של ערך, מאחורינו אופרטור בעל עדיפות נמוכה או ערך גם מייצג שני מקרים:**

- **סָאֵב-מָקָרָה רְאִישׁוֹן: יש מלפניו אופרטור כלשהו.** במקרה זה אנחנו אופרטור חד צדי של צד שמאל:

```

if (lookahead.is(SIGN)) {
    post.push(PartArray([post.pop(), token]));
}

```

{}

- **סאב-מקרה שני:** יש מילפנינו ערך. אם מילפנינו ומאחרינו יש טוקן שניין להתייחס אליו כערך, אנחנו אופרטור דו-צדדי רגיל. אנו מוסיףם לו מכיוון שצרכנו את lookahead:

```
else {
    post.push(PartArray([post.pop(), token, lookahead]));
    i++;
}
```

calculate()

פונקציה נוספת - הפונקציה מתחילה בדאגה למקרים פשוטים/פירוק אלמנט אחד המכיל הרבה אחרים, למערך אלמנטים נורמלי.

אנו מכrazים על 3 משתנים: אופרטור עדכני, ערך מחושב עד עכשו, וסוג מחושב עד עכשו. לאחר מכן, אנו מתחילים לעבור על הטוקנים, אחד אחרי השני. כל טוקן יכול להיות אחד מ-4 מקרים.

- **מקרה ראשון: קבועות מודולוה.** אנו מתעלמים, שכן הטוקן זהה מופיע בתחילת כל גוף אשר מסוגל לקפוץ שורות קוד, ואין לא ערך אמיתי.
- **מקרה שני: אופרטור.** האופרטור העדכני נערך לאופרטור זהה. שני מקרי קצה:
 - **ראשון**- זה הטוקן היחיד שנמצא. הוא מוחזר.
 - **שני**- זה הטוקן האחרון במערך. מחשבים קריאה לערך שחווב עד עכשו באופrnd שמאלי לאופרטור זהה. שמיים אותה בתוך הערך והסוג המחשבים.
- **מקרה שלישי: Expression או PartArray.** מדובר בסוגרים שאו הפונקציה group שמה, או שהמשמש שם. מכאן הפעולות הבאות מותנות:
 - **יש אופרטור עדכני ומהוועב עד עכשו לא קיימ.** משווים את הערך הנוכחי לкриאה לאופרטור העדכני עם אופrnd צד ימין בלבד.
 - **אין ערך מהוועב עד עכשו.** משווים אותו לחישוב של הסוגרים הנוכחיים.
 - **יש ערך מהוועב, אין אופרטור.** זורקים שגיאה ברמת Little, שכן שני אופrndים ללא הקשר לא יכולים לבוא מיד אחרי השני.
 - **אחרת:** קוראים לאופרטור הנוכחי עם הערך מהוועב עד עכשו באופrnd שמאל, והסוגרים הנוכחיים כאופrnd ימני. משווים את התוצאה לערך העדכני.
 תלוי במקרה, יש אפשרות לנסוט 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 אדיביק כאן קוד ספציפי, מעבר לפונקציה אחת, רק כדי להראות את ה"קטע".)

העזה נעשה כמו שילוב של מבנה הסטאק וה-heap – הוא בניי כמו הסטاك בדרך שמסגרות נוספות אליו, ודומה לheap בדרך שזיכרון מנוהל בו.

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

- 4 ביטים ראשונים – ערך heap של שם המשתנה
- 4 ביטים שניים – מקום שם המשתנה במחוזת בזכרון

- 4 ביטים שלישים – מיקום הערך בזיכרון.
- 4 ביטים אחרונים – מיקום סוג הערך בזיכרון.

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

- שני ביטים ראשוניים – כמות האלמנטים הנמצאים במסגרת הקודמת
- שני ביטים אחרונים – כמות האלמנטים שנוספו עד עכשיו למסגרת הנוכחיות

מעבר לheader שיש לכל מסגרת, המערך מתחילה ב4 ביטים של `UInt32` שמצויבים על מיקום המסגרת הנוכחיות בזיכרון `Referrer`.

מבנה זה מאפשר לבצע פעולות חשובות במהירות ובקלות, כמו לשחק עם מסגרות במהירות ובקלות, ולעבור על אלמנטים בקטת קוד, ועדין 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.getInt16(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.getInt16(currentScopePosition);
    var currentScopeLength = bytes.getInt16(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.getInt16(currentScopeStart + 2);
    var nextScope = currentScopeStart -
bytes.getInt16(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.getInt16(nextScope + 2);
        nextScope = nextScope - bytes.getInt16(nextScope) *
KEY_SIZE - 4;
    } while (checkingScope != 0);

    return map.keyValueIterator();
}
```

בעוד שהחזרנות" שבמבנה מהסוג שלו בקושי מתממשות

- חסרון ראשון: גישה בסיבוכיות (ח)O – לא רלוונטי מכיוון שרק לעתים נדירות מאוד מסגרת אחת בטל Referrer תכיל הרבה מפתחות. במקרה זה, אפשר אפילו להגיד שזו בעיה אצל המפתח... .
- חסרון שני: גודל – גם לא רלוונטי, המקרים היחידים בהם מפתח יכול למלא את stack "מהר במירוח" הם גם אוטם מקרים שבהרצות רגילות הוא יקבל overflow, אז אולי הטרמינציה המקדמת אפילו מועילה...

צירת סוגים אקסטרניים

גם כאן הקוד ארוך ואולי נראה סבור, אך יחסית פשוט. אפצל את ההסביר והתקנות 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 |
     | Description | Application |
     |-----|-----|
     |-----|-----|
     |-----|-----|
     |`public <Type> <name>` | Instance |
     `^(address:MemoryPointer, value:InterpTokens) -> InterpTokens` |
     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]).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  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        var name = key.split(" ")[2];
        var type = memory.getTypeInformation(key.split(
    ")[1]);
        var params =
Interpreter.convert(...Parser.parse(Lexer.lex(key.replaceFirst('publi
c 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).poin
ter, (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 תלויות מקרה שמחזירה גם ערך וגם כתובת). הפונקציה יודעת באיזה סוג משתמשים בעזרת הקלט מהמפתח. ישנו 4 סוג קלט המוחלטים ע"י 2 זוגות של הקדמות:

- public/static – אם המפתח מתחילה בpublic, השדה יהיה תלוי מקרה. static, השדה יהיה סטטי. כל דבר אחר יזרוק שגיאה.
- מכיל/לא מכיל סוגרים – אם המפתח נגמר בסוגרים עם/בלי תוכן, מדובר בפונקציה אחרת, מדובר במשתנה.

המידע הנוסף הכרחי לא בהכרח מחייב על סוג השדה:

- לאחר public או static, חyb לבוא סוג המשתנה, או סוג ההחזרה של הפונקציה
- בתוך הסוגרים, חybם להיות הרכזות פורמטרים כתובים ספציפית בLittle. הם מחייבים על כמות וסוגם של הפורמטרים המובאים.

למפתח יש פורמט קבוע:

- כל חלק במפתח מופרד ברווח אחד
- סוגים ומילוט מפתח הינם case sensitive.

שדות תלויי מקרה

יצרים עץ אקסטרנים (ExtTree) חדש על עצם האקסטרנים של הסוג, שלקוט :instanceProperties

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

בתואם לסוג השדה המבוקש, נתונים 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 שאנחנו נתונים 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 שכינתה לנו עם המפתח. אם התוצאה היא ערך, משלימים את המצביע. אחרת, מחזירים את הזוג. במקרה שימושו לא עובד, זורקים שגיאה בرمת 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, מקבלת כמות פרמטרים מסוימת ומכליה TOKEN החרזה ייחיד הקורא לקוד חיצוני להחזיר לו ערך.

שדות סטטיים

הmarker-caן דומה מאד, רק שהפעם אנו עוסקים עם החלק הסטטי:

```

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

```

יש הבדל אחד – מכיוון שבmarker 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 הצתת: (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 מאד – לפני שמכריזים על אותו עץ כסוג, אנחנו מכניסים תנאי קטן, שאם הואאמת, מונע יצירת סוג, וגורם לכל מקום אחר בקוד להתייחס לסוג שהוא יוצרים פה, כמו לאובייקט רגיל...

בדיקות ותוצאות

יפה והכל שיש לנו סווית בדיקות רחבה, אבל גם צריך שהוא בדיקות רחבה, מהראשונה עד לאחרונה, ואפילו קצת על בדיקות שמעבר לסתויטה...

בדיקה ביחידות (Unit Testing)

- בדיקה מס' 1: Basic Math
 - מטרה: לבדוק שחישוב הביטויים המתמטיים של Little עובד, יודע להמיר לסוג תוצאה אחר בעת הצורך.
 - בפועל: מדפסים ביטוי מתמטי עורך המערב סוגרים ואת האופרטורים +, -, *, /, ^, !.
 - תוצאה: הצלחה
 - בעיות שהתגלו: מעולם לא היו בעיות רציניות מעבר לטעויות לא משמעותיות באלאגוריתם החישוב.
- בדיקה מס' 2: Variable declaration
 - מטרה: לבדוק שהכרזת משתנים רגילים עובדת, ולבדוק שניתן לאחזר את הערך שלהם במקורה הצורך.
 - בפועל: מכrazים על 3 משתנים, אחד עם סוג וערך, אחד רק עם סוג, ואחד בלי סוג ובלי ערך. מדפסים את ערך המשתנה הראשון, ואת סוגם של כל המשתנים, אחד אחרי השני.
 - תוצאה: הצלחה
 - בעיות שהתגלו: כשה-Referrer היה בפיתוח, קרו שגיאות בהן ה-Referrer לא שמר/קרא את שמות המשתנים נכון, אז הינו יכולים להזכיר על משתנה, "לאבד" אותו מיד אחר.
- בדיקה מס' 3: Function declaration
 - מטרה: לבדוק שהכרזת פועלות רגילות עובדת, שניתן לקרוא להם עם פרמטרים מסוימים עם סוגים קונקרטיים קבועים ע"י המתכנת.
 - בפועל: מוכrazות 3 פונקציות המדפיסות ערכים – אחת ללא פרמטרים, אחת עם פרמטר דינמי אחד, ואחת עם פרמטר מסווג קונקרטי. קוראים לשילוש 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 ניתן לשדה הנמצא בתוך אובייקט דינמי יכולת לעורך ולהציג את אותו הערך של אותו שדה
 - בפועל: מכrazים על אובייקט דינמי חדש, ובתוכו שמיים שני שדות, אחד מהם גם אובייקט חדש. גם על האובייקט השני שמיים שדה. מדפסים 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))), FunctionCall(PropertyAccess(f, name)))
 - הגיון, ولكن שונה. עכשו זה יפרק כ: .FunctionCall(PropertyAccess(f, name))

- לפני כן, הפונקציה שהייתה מייצרת `PropertyAccess`ים הייתה הולכת מהסוף להתחלה, ולכן, כאשר הייתה מייצרת טוקנים כאלה שאחד בתוך השני, היה מאד קשה לקרוא אותן: `PropertyAccess(a, PropertyAccess(b, c))`. כיוון הסקירה שונה, ועכשו טוקנים כאלה נכונים אחד בתוך השני כך: `PropertyAccess(PropertyAccess(a, b), c)`.
- בדיקה מס' 5: Loops
 - מטרה: לבדוק האם לולאות `for` ו-`while` עובדות כשרה.
 - בפועל: יוצרים משתנה, ומricsים אליו לולאת `while` שמדפסה אותו עד שmagiu ל-5. מricsים לאחר מכן לולאת `for` המדפסה 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"` `whenever` ו-`after` עובדים כשרה.
 - בפועל: יוצרים משתנה, ומיד אחריו בודקים תנאי שחייב להיות נכון. אחריו, מוסיפים את שני אירועי התנאי, ומשנים את ערך המשתנה. מצפים שהם יורצו במקיריים מסוימים ובסדר מסוים.
 - תוצאה: הצלחה
 - בעיות שהתגלו: לא היו בעיות ספציפיות.
- בדיקה מס' 7: Code blocks
 - מטרה: לוודא שניתן ליצור ערכים בעזרה בлок של קוד המ אחזר ערך, ולא רק ערך `"inline"`
 - מricsים על משתנה, שערך הוא בлок 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 קונטיקסט במקרי שינוי עצמי.
 - יוצרים משתנה שערך 1.2, ומשווים אותו לעצמו + 2 * עצמו. בוטוגרים, לחلك 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 שרשות `else-if` עובדות כשרה.
 - בפועל: מתחילה עם תנאי הנפטר `false`, ועלוי לא להיות מושך. אחריו יש `if-else` שאם חייב לא להיות מושך. אחריו יש `else` ריק, ומכיון שלא הורץ כלום עד עכשו, חייב להיות מושך.
 - תוצאה: הצלחה
 - בעיות שהתגלו: לא היו בעיות ספציפיות.
- בדיקה מס' 10: Nested code blocks
 - מטרה: לבדוק שבЛОוקים של קוד הפעלים 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)

- בעועל: מכירזים על משתנה `a` שערכו 3. מתחילה 3 בלוקים, אחד בתוך השני.
בבלוק הכי פנימי `|` מוכרך שוב, עם הערך 5, ומודפס מיד לאחר מכן. אחריו
הבלוקים, מודפס שוב ערכו של `|`, שצירר להיות ה- `o` מקודם, שערכו 3.
תוצאה: הצלחה
- בעיות שהתגלו: הייתה פעם שגיאת `one-off` שהייתה קורית כשהיא
מנסה ליצור מסגרת חדשה ב-`referrer` כאשר לא היו משתנים במסגרת
הקיימת. בגלל גישה שגoya למערך הבטים, ה-`header` החדש היה יכול להיות
ממוקם על מקום שכבר קיים בו מידע, והיה יכול להתחשב במקרה ש
כמידע על ה-`header`. זה היה יכול לגרום לכך שהזיכרון היה חושב של מסגרת
שלפנוי האחת הצעית `IS` כמוות אלמנטים חזיה, וכך הייתה מדלגת אחרת
`Out Of Bounds` וקורסוט מגישה `LB`.
- בדיקה מס' 11: **Inline Blocks**
 - מטרה: הרחבה של בדיקה 7, לוודא שניתן גם לאחזר מזהים בעזרת בלוק של
קוד, ולא רק ערכים.
 - בעועל: מוכרך 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 הי בעיות ספציפיות
- בדיקה מס' 12: **pool**
 - מטרה: לבדוק שברירת הקבועים עובדת כשרה ויכולת לאחזר ערכים מתי
שצריך במקום לפנות עוד מקום.
 - בעועל: מכירזים על 4 משתנים: שני `nums`, 0, ו-0.0. מדפיסים האם שני
המשתנים הראשונים מעתיפים כתובת, והאם השניים האחרונים גם מעתיפים
כתובת. שני הבדיקות אמורים לצאת אמת.
תוצאה: הצלחה
 - בעיות שהתגלו: המבנה פשוט, לא הי בעיות ספציפיות.
- בדיקה מס' 13: **Type Name Property**
 - מטרה: לבדוק שהשדה הגלובלי `type` עובד כשרה, וכן מודיע את הסוג
הנכון לכל סוג ערך.
 - בעועל: מודפסים 6 סוגי, באותה מחרוזת, ללא רווח: הסוג של 5, 5.5, `true`,
`null` (`nothing`, האופרטור `+` והסוג `Number`). צירר להדפס:


```
"NumberDecimalBooleanAnythingSignType"
```

 תוצאה: הצלחה
 - בעיות שהתגלו: לא הי בעיות ספציפיות.
- בדיקה מס' 14: **Reference vs. Value**
 - מטרה: לבדוק האם שימושים עם ערכים שאמורים להיות מועברים כערך
או כזכור, האם מה שMOVBR זיה המיקום בזיכרון (אלמנטים מאזכרים) או
הערך המועתק (אלמנטים המועברים כערך).
 - בעועל: יוצרים משתנה `a` שערכו אובייקט חדש. יוצרים משתנה חדש `b`
ומשוים אותו ל-`a`. מדפיסים האם הכתובות שליהם שוות (צריך לצאת אמת).
יוצרים משתנה `c` ומשווים אותו ל-502. יוצרים עוד משתנה, `d`, ומשווים אותו
ל-`c`. מדפיסים האם הכתובות שליהם שוות (צריך לצאת שקר)
תוצאה: הצלחה
 - בעיות שהתגלו: כשהוופטי את הבדיקה הצעית, שכחתי לגמרי מזה שצריך
להתקן מערכת שתבדיל בין סוגים הערכים. בעקבות הבדיקה, שונתה קצת

צורת הגישה לזיכרון, וגם עכשו יכולה לתמוך בהחזרת מזהים במקום ערכים, והמרתם לערכים בעת הצורך (לדוגמה, אם יש משתנה `x` שערכו 5, עכשו אפשר להחזיר את המזהה `x`, ולא את המספר 5).

- **בדיקה מס' 15: Arrays**

- מטרה: לבדוק האם המרכיבים בספריה הסטנדרטית אכן עובדים כמצופה.
- בפועל: יוצרים משתנה שערך מערך. ערכים בו תא אחד ומדפים אותו. מדפים את אורך המערך, וכך האלמנט שהוא מכיל תוצאה: הצלחה
- בעיות שהתגלו: לא היו בעיות ספציפיות.

בדיקות ידניות

• **בעבוק אחזרים מתנאים ולולאות:** כשתכנתי מחדש את הדרך שתנאים מאוחסנים בזיכרון, שמנת ללב `oversight` יחסית משמעותי: מכיוון שאינו מרים את גוף התנאי/הלוואה באמצעות פונקציית הרצתה הרגילה, אם נמצא בה המילה `return` במקום כלשהו, במקומות שבהיא תחזיר ערך מהפונקציה שממנה היא נקראת, היא פשוט תצא מהלוואה – אולי היא מחזירה את הערך שמייצג בлок הקוד של הלואה.

פתרתי את הבעיה בעדרת הוספה פרמטר לפונקציית הרצתה – `propagateReturns` – כאשר הפרמטר הוא `true` ואנו חנו נתקלים בטוקן `FunctionReturn` בסוג `FunctionReturn`, במקום להחזיר את ערכו של האחזור, הוא מחזיר ממש את טוקן האחזרה עצמו. ההחזרה מפסיקת הלוואה, והלוואה נפתרת לטוקן `FunctionReturn`, אותו פונקציית הרצתה הרגילה (`propagateReturns` ללא `type`) רואה, מפסיקת הרצתה, ומחייבת את הערך המתאים.

• **קריאה שדות מערכי "Inline":** כשהיימתי עם הבסיס של מודול הזיכרון בפרויקט, רציתי לבדוק כמה טוב היא עובדת עם גישה לשדות רגילים, שדות בתוך שדות, וכו'. אחת הבדיקות שבייצעת היא גישה לשדות גלובליים על ערכים שלא שמורים בזכרון עדין (לדוגמה: `13.address`, `5.type`),gilithי משווה יחסית חמור – כאשר אנו מנסים לקרוא מהזיכרון בעדרת פונקציית הקרייה הראשית, `Memory.read`, הפונקציה תמיד מנסה לעקוב אחרי כל מזהה שמובא לה אולי הוא שם משתנה בפנוי עצמו, ולכן הפונקציה נכשלת במקרים שהמזהה הראשון הוא לא שם משתנה, אלא פשוט ערך.

לאחר שהבנתי לגמרי את הבעיה, הפתרון היה יחסית פשוט – הכתני אלטרנטיבת ההפונקציה, שהיא `readFrom` – הפונקציה הזאת, במקום להתחילה במשתנה, מתחילה במידע על הערך – ערכו, מקום בזכרון, ועוד. ממש, הפונקציה מושגת מיד על שדות הערך, וממשיכה כמו הפונקציה `read`.

• **השווות משתנים לפונקציות חייזניות:** כשבדקתי את הפרויקט לאחר כמה תוספות ושינויי, גיליתי עוד `oversight` משמעותי: מכיוון שהכתבות של פונקציות חייזניות מעולים לא זמינה, הכתבות של כל פונקציה חייזנית היא כתובת הערך החיזוני הגלובלי – `Memory.constants.EXTERN`. לכן יצא, שכשמשווים משתנה רגיל לפונקציה חייזנית, האינטפרטר חושב שערך זה קוד של פונקציה השוכן בכתובת של

EXTERN. זה רע מאד, מכיוון שכנתובת הזאת בכלל לא מאוחסנת פונקציה – אחרתה עדין נמצא חלק מברירת הקבועים, ולכן גם לא ניתן לעורר את המשתנה, וגם אי אפשר לקרוא לפונקציה שהמשתנה מייצג.

הפתרון יחסית "האקי", מכיוון שחלק מההגבלות של השפה, לא ניתן לאחסן פונקציות "חיות" של Haxe ולהריץ אותן סתם כך, ואחסן הפונקציה מסתמך על שמירת הקוד בזיכרון בצורה כלשהי. לכן, כאשר הזיכרון מזהה ניסיון אחסון של פונקציה אקסטרנית, הוא שומר את ערך המשתנה לא אותה פונקציה ממש, אלא כפונקציה אחרת, המקבלת את אותם פרמטרים כמו הפונקציה האקסטרנית, ומחזירה טוקן קרייה 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

עץ קבצים

```
C:.
├── .gitattributes
├── .gitignore
├── book.docx
├── compile.hxml
├── demoCode.ltl
├── formatted.ltl
├── haxelib.json
├── LICENSE
├── Little.desktop
├── Logo-small.png
├── Logo.ico
├── Logo.png
├── rcedit-x64.exe
├── README.md
└── tree.txt

├── .vscode
│   ├── launch.json
│   ├── settings.json
│   └── tasks.json

├── compilers
│   ├── linux.hxml
│   └── windows.hxml

└── src
    ├── Main.hx
    ├── UnitTests.hx
    └── js_example
        └── JsExample.js.hx

    └── little
        ├── KeywordConfig.hx
        └── Little.hx

        └── interpreter
            ├── ByteCode.hx
            ├── Interpreter.hx
            ├── Runtime.hx
            ├── StdOut.hx
            └── Tokens.hx

            └── memory
```

```
ConstantPool.hx
ExternalInterfacing.hx
HashTables.hx
Memory.hx
MemoryPointer.hx
Operators.hx
Referrer.hx
Storage.hx

lexer
Lexer.hx
Tokens.hx

parser
Parser.hx
Tokens.hx

tools
Conversion.hx
Extensions.hx
Layer.hx
OrderedMap.hx
Plugins.hx
PrepareRun.hx
PrettyPrinter.hx
TextTools.hx
```

התקנה

למפתחי תוכנה

לפניהם שמתחילה, יש לוודא שהשפה Haxe והכליים שלה מותקנות כראוי: יש להתקין את הטרנספִילר של Haxe, ולבדק ש:

- הפקודה Haxe נמצאת בPATH, או לפחות ניתן למצוא את הקובץ ולהפעיל אותו באמצעות המסלול אליו.
- הפקודה Haxelibעובדת, אמורה להדפיס מידע על שימוש במנהל הספריות. מעבר לזה, אני ממליץ:

- להוריד Visual Studio Code ולהוריד את הרחבה Haxe Extension Pack
- להוסיף בפרויקט שלהם תקיה בשם vscode.tasks.json, 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 זהה, אפשר לבנות את הפרויקט שלו באמצעות לחיצה על

Ctrl+Shift+B

עכשו כהכל מותקן כשרה, אפשר להוריד את הפרויקט כספירה באמצעות מנהל הספריות, באמצעות הפקודה:

Haxelib git little
<https://github.com/ShaharMS/Little/tree/branch/functional-programming>

לכלול את הספרייה בפרויקט שלו באמצעות הוסף השורה הזאת בקובץ הקומpileציה (hxml) שלו:

--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;
```

הויסיף "קובץ" קוד להרצה באמצעות קריאה `loadModule` Little. loadModule לפני ההרצה. כמו כן, לשני הפונקציות יש פרמטר לאפשר `debugging`. יש לציין שהק `loadModule` מורץ במקום פרמטר `debug` שלו משנה:

```
Little.loadModule("define x = 3", "MyModule");
Little.run("print(x)", true);
var output = Little.runtime.stdout.output;
```

לכמפל לקוד-בתים באמצעות הפונקציה `:Little.compile`

```
var ast = Little.compile("define x = 3, print(x + 6 ^ 3)");
File.saveContent(ByteArray.compile(...ast), "compiled.txt");
:Little.format
```

ולפרמטר קוד באמצעות הפונקציה `: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 האחרונות, כדי לחת לתוכן 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 אירועי, 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
```

לקראא, לכתוב ולאחסן זיכרון בעזרת `:Little.memory`

```
Little.memory.write(["x"], Number(3), "Number");
Little.run("print(x)"); // output: 3
```

הויסיף 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 ולהחליפ סטימ של מילים שמורות בעזרת הפעולות שב-
:Little.keywords

```
Little.keywords.PRINT_FUNCTION_NAME = "הדף";
Little.run("הדף(3)"); // output: 3
```

מתכנתים ב-Little

יש לחפש בדף ShaharMS, ולהיכנס לתוכה הראשונה (כנראה):

GitHub · ShaharMS · לדף המתורגם · https://github.com · Shahar Marcus ShaharMS

A programming language designed around portability, flexibility and ease of use.
Every, single, keyword, is customizable! Haxe ...

GitHub · ShaharMS · לדף המתורגם · https://github.com · ShaharMS/texter · Folders and files · Latest commit · History · Repository files navigation.

Haxelib · ShaharMS · https://lib.haxe.org · GitHub · ShaharMS · https://lib.haxe.org · ShaharMS

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

Overview · Repositories 37 · Projects 1 · Packages · Stars 47

Pinned

Little · Public

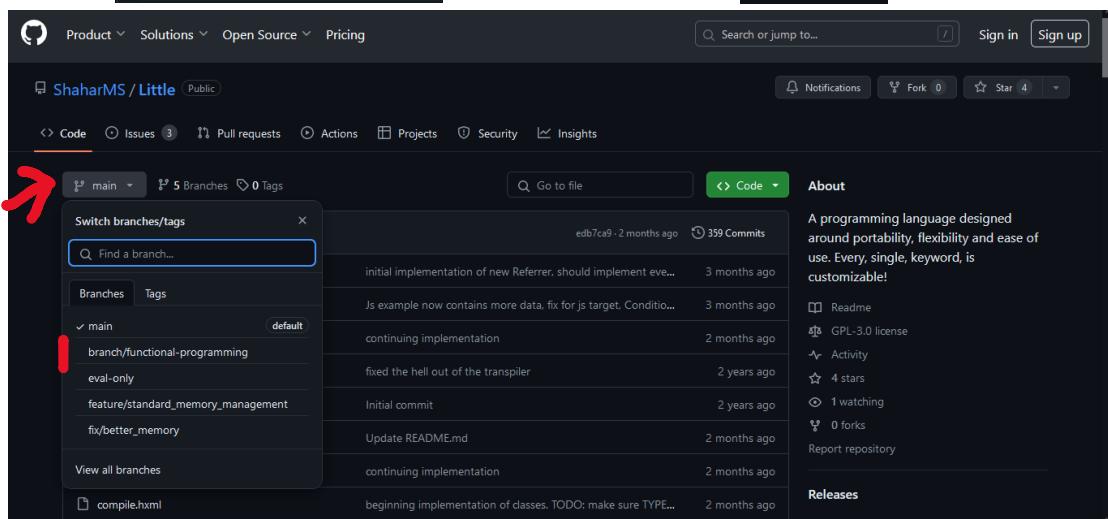
A programming language designed around portability, flexibility and ease of use.
Every, single, keyword, is customizable!

Shahar Marcus
ShaharMS

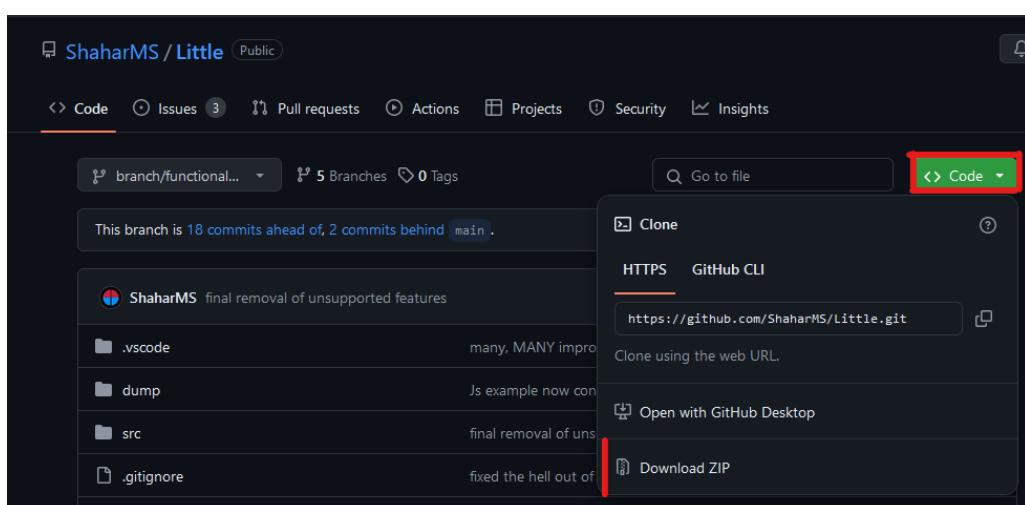
549 contributions in the last year · Contribution settings · 2024

לאחר מכן, יציג החלון של הענף הראשי שלuproject. אנו לא מעוניינים בענף זהה, לכן עליינו:

- **ללחוץ על הלחצן `branch/functional-programming`** ולבחור את האופציה `main`



- **ולבחור את האופציה `<> Code`**



לאחר הורדת הקובץ ZIP, יש ללחוץ אותו (כל תוכנת חילוץ טובה), לגשת אל תוךuproject, ולהיכנס לתיקייה `clients`. בתקיה יהיו 3 סוגים ל��וחות בשלוש תיקיות:

- **תיקייה web** – הלקוח האינטרנט, זמין גם ב <https://spacebubble.io/little/demo> מפעליים בעזרת לחיצה כפולה על הקובץ `index.html`, או בפתחה שלו בעזרת דפדפן כזה או אחר.
- **תיקייה המתחילה ב-windows** – לkopio שורת הפקודה כתוכנה להריצה בwindows.
- **תיקייה המתחילה ב-axunlin** – לkopio שורת הפקודה כתוכנה להריצה באסchnlin

QuickStart

- **לקוקו אינטרנט** – יש להסתכל מצד הימני של המסך – מעבר למקום להחלה מילימ'
- שמורות, הוא גם מכיל דוגמאות קוד שאפשר ואמורים ללמידה מהם.
- **לקוקו שורת פקודה** – יש להשתמש בפקודה `printSample!` כמו שהוזכר תחת **הគטורת סוג ל��וח: לומד תכנות, מתכנת Little**

רפלקסיה

עבודה על הפרויקט

לצערי ולשחתתי, פרויקטים כאלה מעולם אינם נגמרים – אנחנו אול' יכולים להציב אבני דרך, אך אי אפשר "להשלים" פרויקטים כאלה.

از היתי חייב להחליט על אבן דרך "סופית". לgresת הפרויקט שאני מגיש, הצבתי את אבן הדרך הסופית בתכנות פונקציונלי. הצבתי את אבן הדרך הסופית שם משתי סיבות: הראונה אולי ברורה, והוא צריך לסייע עם הפרויקט מתישתו, אך אחורי השניה יש מחשבה – תוך כדי התכנות של הפרויקט, שמתי לב שכשאני "חוֹפֵר" יותר אל תוך נושאים מסוימים, הכרחתי מחלקות לדוגמה, אני מבacz הרבה מאוד זמן על תכנות דברים שאני גם כבר יודע איך עושים, וגם כוללים הרבה עבודה שחורה.

בקלות אפשר לראות שהפרויקט הזה היה מסע, שככל UILות ומורדות, התלבויות, הפסיקות עבודה, ולפעמים אפילו Burnout:



(בתמונה: גרפ' תדרות commits לפי זמן, ממרץ 2022 עד מאי 2024)

הפרויקט 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 מאד, והייתי יכול להשתמש בניסיון ובידע שלי כדי לפתור אותו. لكن אפשר גם לראות, שרוב האלגוריתמים שהשתמשו בהם בפרויקט הם גרסאות "השרהה" של אלגוריתמים אחרים – לא הייתה צורך להיצמד וללמוד ספציפית אך ורק מקום אחד – כל פעם שנטקתי בעיה ציו או אחרת,לקחתי הרבה ידע מהרבה מקומות, ויצבתי פתרון בסופו עבדמצו.

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

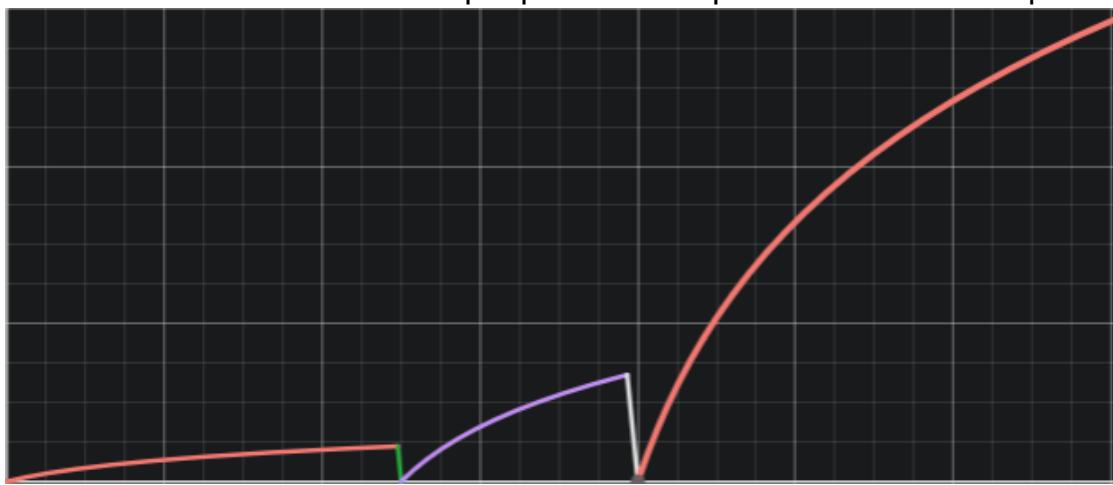
- Hashing
- ניהול זיכרון, Stack/Heap
- ByteCode וקוד מכונה
- עצי syntax
- שיטות עיבוד מידע בסדר גודל גודל/עוצום

ומעבר לתחומים למודים, התחרתי להרגיש מאוד בנווח עם מגוון "דרכי תכנות", ובמיוחד רקורסיה, שכנן רוב הפרויקט – מפונקציות, מחלקות ועד אובייקטים וADTs, כתוב מאילץ בצורה רקורסיבית...

תובנות

מתכנן ומתהיליך "בנייה" הפרויקט רכשתי מגוון כלים ולמדתי הרבה "לקיים" לעתיד:

- **חשיבות תוכנון הפרויקט:** תוך כדי בניית הפרויקט, "גיליתי" 3 סיבות ב-3 מילימ' למה תוכנון 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 היטב הכל'ם עם השני, ולא בנויים טוב אחד על השני, מתחילה בסוף לחיף יותר וייתר, עד שאנוanno מגיעים למצב שיום דבר לא עובד כמעט, וגם אין דרך חזרה. מבנה פרויקט טוב מאט מאד את הייצורות ה"עובד", וכן התוכנון הזה חשוב.
 - **Burnout** – כשהפרויקט בניין כרך שכל החלקי בבירור מפוצלים לחלקים שונים, שימושים אך לא מסתמכים אחד על השני, אפשר לקפוץ מתוכנות חלק אחד לחלק אחר בפרויקט, והתקדמות גם נשארת קרובה לリンיארית, וגם נמאס לר' הרבה, הרבה יותר לאט...
 - **Learnability** – אחד מהדברים הכל'ם מבאים כשמפתח מփש כל' להשתמש בו, או ללמד ממנה. זה שהוא לא מצליח בכלל להבין את הכל' – איך כל דבר משתמש בכל דבר אחר, למה הרכיב הזה ספציפית חיוני, וכו'. לכן, על מנת שcoli יוכלו לפתח יותר טוב, אנחנו נרצה לפתוח קוד טוב יותר, שמייצג דוגמה טובה יותר, וכל ללמידה גם ממנה, וגם לאמץ את המבנה שלו....
- **אסור לפחד מנושאים גדולים:** כשנושא נראה ענק, זה בדרך כלל אשליה – בסופו של יום, כל נושא מורכב מרבה מאוד חלקים, שחלק חיוני לדעת, וחלק פחות. דבר חשוב שלמדתי במהלך הפרויקט, ואפילו בנסיבות במהלך המסע שלי כמתכננת, זה שלכל שלב נתון, צריך לדעת כמה יחסית קטנה של מידע מסוינו נושא, ואולי עם הזמן צריך להרחב אותה, אבל זה לעולם לא באופן מיידי – זה לא נכון להסתכל על נושא כמוניית', אלא כurmaת אבניים...
- **מותר לטעות, ואףילו בגודל:** יצא לי לעשות refactoring שלוש פעמים במהלך תכנונות הפרויקט, וברטורופקט, אני שמח על כל פעם שמחקתי את כל הקוד וכתבתו אותו מחדש. באף אחת מהפעמים שעשית זאת לא הפסדתי התקדמות: בסוף, יצא שאדם מתכונת משחו, במשך אףילו חצי שנה, מבין שהדבר לא טוב, מתכונת את זה טוב יותר, וմבדיק את הפער ואףילו יותר תוך שבועיים. אציג גרפ' לזה:



(בגרף: כל חלק עליה זה התקדמות בפרויקט, כל ירידה היא תכונות חדש/refactor)

הסתכלות

אחרוניית

הפרויקט הזה פותח במשך שניםים, בהם למדתי המונ. ברור שהיו דברים שהייתי מישם אחרת:

- אופרטורים – גם בהסתכלות אחרת, אני לא 100% בטוח למה פיצלתי בין פעולות וערכים רגילים, לאופרטורים. אין בזה שום דבר טוב. האימפלמנטציה לא מאפשרת לדינמית להכריז על משתנים, ובמיוחד שלא לאחר מכן לפי הצדדים שהם מבקשים. פשוט, לא ממשו...
- אחסון פונקציות חיצונית – אם הייתה יודעת יותר, הייתה גם משקיע יותר בפתרון לשימרת פעולות אקסטרניות בזיכרון, ולא הייתה חייב להסתבר מחדש שלם על בעיות הקשורות לאימפלמנטציה של זה.

וקדימה

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

- מחיקה שלמה של מערכת האופרטורים, ושיולה עם המערכת הרגילה והאקסטרנית
- מוסיף הכרזה על סוגי אופרטורים, ותנאים ולולאות
- הוסיף זיהוי אופרטורים רב-סימניים גם כפיצר של זמן ריצה, על מנת לאפשר הכרזה דינמית לגמרי של אופרטורים
- הרחבה משמעותית של הספרייה הסטנדרטית שהשפה מספקת
- מוסיף קומפ'ילר לקוד מכונה, ולא רק לקוד-בתים ספציפי
- מוסיף את התמיכה בדוקומנטציה ברוחבי השפה, ומאפשר גישה דינמית אליהם

תודות

קדום כל, לאח שלי גיא, שהוא 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
- וcomma, וקייפידה – באופן אירוני במיוחד, זה היה מקור המידע האולטימטיבי לאורך כל המסע הזה – ומה אני מופתע – הרבה אנשים שהרבבה יותר חכמים ממוני עבדו בצורה הרבה יותר מבודקת מהרגיל על "מאמרם" שהם סוג של מחקר מדעי - הם ח"בים להיות טוביים...

ביבליוגרפיה

- אנליזה לקסיקלית, כותבי ויקיפדיה, ללא תאריך.
https://en.wikipedia.org/wiki/Lexical_analysis
- ניהול זיכרון - כותבי ויקיפדיה, ללא תאריך.
https://en.wikipedia.org/wiki/Memory_management#STACK
- 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>
- 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))
- קוד בריחה של ANSI - כותבי ויקיפדיה, ללא תאריך.
https://en.wikipedia.org/wiki/ANSI_escape_code
- אלגוריתם Shunting Yard - כותבי ויקיפדיה, ללא תאריך.
https://en.wikipedia.org/wiki/Shunting_yard_algorithm
- חישוב ביטויים מבוסס סמלים - כותבי ויקיפדיה, ללא תאריך.
https://en.wikipedia.org/wiki/Symbolic_method
- חישוב סימבולי-נומירי - כותבי ויקיפדיה, ללא תאריך.
https://en.wikipedia.org/wiki/Symbolic-numeric_computation
- 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
- Visual Basic for Applications - כותבי ויקיפדיה,
https://en.wikipedia.org/wiki/Visual_Basic_for_Applications
- Heap זו-הורן - 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>
- (שפת תכנות) - כותבי ויקיפדיה, ללא תאריך.
[https://en.wikipedia.org/wiki/F_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/F_Sharp_(programming_language))

נופחים

הקדמה קצרה

- לפני שנתחיל עם הקוד, בגלל האורך שלו, גם לגובה וגם לרוחב, אני ממליץ להעיבר את word למצב עמוד אינטרנטי, באמצעות  לחיצה על שנמצא בצד ימין/שמאל למטה של חלון הורד, תלוי בשפה של ווד.

עוד יש לציין, שבגלל אורך הקבצים תהיה התעלמות מוחלטת מהפיצול לעמודים. עדין, למען שימור הפורמט עד עכšíו, הקוד יתחל 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),
<https://github.com/ShaharMS/Little/tree/branch/functional-programming> הוא פחות ל.cgi, מציג את מבנה התיקיות ויודע לקשר בין מקומות בקוד.

- Visual Studio Code נעשה באמצעות Syntax highlighting •

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 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;
#endif
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();
    #elseif formatting
    File.saveContent("formatted.txt",
Little.format(Resource.getString("sample")));
    #elseif sys
    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");
                var code = "";
                while (true) {
                    Sys.print(" >> ");
                    var input = Sys.stdin().readLine();
                    if (input == "run!") {
                        Little.run(code, true);
                        Sys.println("Output:");
                        Sys.println(Little.runtime.stdout.output
+ "\n");
                        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("Output:");

```

```
Sys.println(PrettyPrinter.printInterpreterAst(Interpreter.convert(...  
Parser.parse(Lexer.lex(input)))) + "\n");  
        } 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);  
    Sys.println("Output:");  
    Sys.println(Little.runtime.stdout.output + "\n");  
    Little.reset();  
}  
preDefInput = null;  
}  
}  
} catch (e) {  
    // This jsut means Ctrl + c, or, exiting the program  
    Sys.println("\n\nExiting...");  
    Sys.exit(0);  
}  
#end  
}  
}  
  
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');
            }
        }
    }
}
```



```
}

public static function test3():UnitTestFixture {
    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():UnitTestFixture {
    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():UnitTestFixture {
    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])]),
        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():UnitTestFixture {
    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():UnitTestFixture {
    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():UnitTestFixture {
    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():UnitTestFixture {
    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():UnitTestFixture {
        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():UnitTestFixture {
        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():UnitTestFixture {
        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
    }
}
}

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.getElementsByName("p")[0];
        var input = row.getElementsByName("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.createElement();
    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} ${Little.keywords.TYPE_STRING} =  
{\n&ampnbsp&ampnbsp&ampnbsp&ampnbsp${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_F  
UNCTION_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}${Little.keywords.TYPE_BOOLEAN}())  
\n&ampnbsp&ampnbsp&ampnbsp${Little.keywords.PRINT_FUNCTION_NAME}("${${Little.keywords.  
TRUE_VALUE}"${Little.keywords.PROPERTY_ACCESS_SIGN}${Little.keywords.  
TYPE_CAST_FUNCTION_PREFIX}${Little.keywords.TYPE_BOOL  
EAN}()${Little.keywords.PROPERTY_ACCESS_SIGN}${Little.keywords.TYPE_C  
AST_FUNCTION_PREFIX}${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.VAR  
IABLE_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.PROP  
ERTY_ACCESS_SIGN}${Little.keywords.STDLIB__MEMORY_free}(x,  
arrayByteAmount)\n${Little.keywords.PRINT_FUNCTION_NAME}(${Little.key  
words.TYPE_MEMORY}${Little.keywords.PROPERTY_ACCESS_SIGN}${Little.key  
words.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_AC  
CESS_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_ST  
RING},  
3)\narray${Little.keywords.PROPERTY_ACCESS_SIGN}${Little.keywords.STD  
LIB__ARRAY_set}(1,  
"Hey!")\n${Little.keywords.PRINT_FUNCTION_NAME}(x${Little.keywords.P  
ROPERTY_ACCESS_SIGN}${Little.keywords.STDLIB__ARRAY_get}(1)) """  
"Hey!"  
"""\\n${Little.keywords.PRINT_FUNCTION_NAME}(x${Little.keywords.PROPER  
TY_ACCESS_SIGN}${Little.keywords.STDLIB__ARRAY_length}) """ 3 """;  
        case "TYPE_FUNCTION":  
'${Little.keywords.FUNCTION_DECLARATION} x() =  
{}\\n${Little.keywords.PRINT_FUNCTION_NAME}(x${Little.keywords.PROPERT
```

```
Y_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_TY  
PE_PROPERTY_NAME}) """ ${Little.keywords.TYPE_CONDITION} """;  
    case "FUNCTION_RETURN":  
'${Little.keywords.FUNCTION_DECLARATION} y() =  
{\n &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.PROPERT  
Y_ACCESS_SIGN}length\n${Little.keywords.PRINT_FUNCTION_NAME}(len${Lit  
tle.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_N
AME}(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_A  
CCESS_SIGN}${Little.keywords.OBJECT_TYPE_PROPERTY_NAME}) """  
${Little.keywords.TYPE_INT} """;  
        case "OBJECT_ADDRESS_PROPERTY_NAME":  
'${Little.keywords.PRINT_FUNCTION_NAME}(0${Little.keywords.PROPERTY_A  
CCESS_SIGN}${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.STDL  
IB__ARRAY_length}';  
            case "STDLIB__ARRAY_elementType":  
'myArray${Little.keywords.PROPERTY_ACCESS_SIGN}${Little.keywords.STDL  
IB__ARRAY_elementType} """ An Array of Strings will return  
` ${Little.keywords.TYPE_STRING}` """;  
            case "STDLIB__ARRAY_get":  
'myArray${Little.keywords.PROPERTY_ACCESS_SIGN}${Little.keywords.STDL  
IB__ARRAY_get}(0)';  
            case "STDLIB__ARRAY_set":  
'myArray${Little.keywords.PROPERTY_ACCESS_SIGN}${Little.keywords.STDL  
IB__ARRAY_set}(0, "Hey")';  
            case "STDLIB__MEMORY_allocate":  
'${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.VAR  
IABLE_DECLARATION} a =  
${Little.keywords.TYPE_ARRAY}${Little.keywords.PROPERTY_ACCESS_SIGN}$  
${Little.keywords.INSTANTIATE_FUNCTION_NAME}(${${Little.keywords.TYPE_ST  
RING}, 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 = "\u221a";  
@: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 that called them (see  
 * `little.tools.Layer`)  
 */  
public static var debug:Boolean = 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:Boolean = false, runRightBeforeMain:Boolean = 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 code you're
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").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<LexerToke
ns> {
    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([Littl
e.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 sequences 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 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(`='): {
        if (post.length == 0) {
            Little.runtime.callStack.unshift({module:
Parser.module, line: line, linePart: linePart, token: Write(null,
null)}));
        Little.runtime.throwError(ErrorMessage("Missing assignee before
`='"), Layer.PARSER);
        return null;
    }
    var assignee = post.pop();
    if (assignee.is(SET_MODULE, SPLIT_LINE,
SET_LINE)) {
        Little.runtime.callStack.unshift({module:
Parser.module, line: line, linePart: linePart, token:
Write(Interpreter.convert(assignee), null)});
        Little.runtime.throwError(ErrorMessage("Missing assignee before `=`,
assigning operation is cut off by a line split, a new file or a
line."), Layer.PARSER);
    }
    if (i + 1 >= pre.length) {
        Little.runtime.callStack.unshift({module:
Parser.module, line: line, linePart: linePart, token:
Write(Interpreter.convert(assignee), null)});
    }
    Little.runtime.throwError(ErrorMessage("Missing value after the last
`='"), Layer.PARSER);
    return null;
}
var lookahead = mergeWrites([pre[i + 1]])[0];
if (assignee.is(WRITE)) {
    var previousAssignees =
assignee.parameter(0);
    previousAssignees.push(assignee.parameter(1));
    post.push(Write(previousAssignees,
lookahead));
}
else post.push(Write([assignee], lookahead));
```

```
        }
        case Variable(name, type, doc):
post.push(variable(mergeWrites([name])[0], mergeWrites([type])[0],
mergeWrites([doc])[0]));
        case Function(name, params, type, doc):
post.push(function(mergeWrites([name])[0], mergeWrites([params])[0],
mergeWrites([type])[0], mergeWrites([doc])[0]));
        case ConditionCall(name, exp, body):
post.push(conditionCall(mergeWrites([name])[0],
mergeWrites([exp])[0], mergeWrites([body])[0]));
        case Read(name):
post.push(read(mergeWrites([name])[0]));
        case TypeDeclaration(value, type):
post.push(typeDeclaration(mergeWrites([value])[0],
mergeWrites([type])[0]));
        case FunctionCall(name, params):
post.push(functionCall(mergeWrites([name])[0],
mergeWrites([params])[0]));
        case Return(value, type):
post.push(return(mergeWrites([value])[0], mergeWrites([type])[0]));
        case Expression(parts, type):
post.push(expression(mergeWrites(parts), mergeWrites([type])[0]));
        case Block(body, type):
post.push(block(mergeWrites(body), mergeWrites([type])[0]));
        case PartArray(parts):
post.push(partArray(mergeWrites(parts)));
        case PropertyAccess(name, property):
post.push(propertyAccess(mergeWrites([name])[0],
mergeWrites([property])[0]));
        case Write(assignees, value):
post.push(write(mergeWrites(assignees), mergeWrites([value])[0]));
        case Custom(name, params): post.push(custom(name,
params.map(x -> mergeWrites([x])[0])));
        case _: post.push(token);
    }

    i++;
}

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<InterpToke
ns> {
    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(assignedees, value):
Write(convert(...assignedees), 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:Int, 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
stringifiable via `token.extractIdentifier()`.

    @param type The type of the variable. Can be any token
stringifiable 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
stringifiable 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:Boolean = true):Boolean {
        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; // Can't 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; //
Can't 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(assigees:Array<InterpTokens>,
value:InterpTokens):InterpTokens {

    var vars = [], funcs = [];
    var containsFunction = false;
    var containsVariable = false;
    for (assignee in assigees) {
        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
stringifiable 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(assignedees, value): {
        returnVal = write(assignedees, 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:Boolean = 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 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;
}
}
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 its 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 doesn't 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, but 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_ACSES
S_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,
get TypeName(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 performing 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_ACCE
S_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_A
CESS_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.addres
s, storage), pathCopy[0], storage)) {
        var a =
wentThroughPath.concat([pathCopy[0]]).join(Little.keywords.PROPERTY_A
CESS_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 than 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 functionToInt():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:Intertokens,
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:Intertokens) {
        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.

 * Its 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 overflowed (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.getInt16(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.getInt16(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.setInt32(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.setInt16(currentScopeStart + 2,
bytes.getInt16(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.getInt32(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.setInt16(bytes.getInt32(0) + 2,
bytes.getInt16(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.getInt16(currentScopeStart + 2);
    var nextScope = currentScopeStart -
bytes.getInt16(currentScopeStart) * KEY_SIZE - 4;

    do {
        var i = checkingScope + 4;
        while (i < (checkingScope + elementCount * KEY_SIZE)) {
            var testingHash = bytes.getInt32(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.getInt32(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.getInt16(nextScope + 2);
    nextScope = nextScope - bytes.getInt16(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.getInt16(currentScopeStart + 2);
    var nextScope = currentScopeStart -
bytes.getInt16(currentScopeStart) * KEY_SIZE - 4;

    do {
        var i = checkingScope + 4;
        while (i < (checkingScope + elementCount * KEY_SIZE)) {
            var testingHash = bytes.getInt32(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.getInt16(nextScope + 2);
    nextScope = nextScope - bytes.getInt16(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.getInt16(currentScopeStart + 2);
    var nextScope = currentScopeStart -
bytes.getInt16(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.getInt16(nextScope + 2);
        nextScope = nextScope - bytes.getInt16(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.rawLo
cation)));
}
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.rawLo
cation)));
}
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 its 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:Intertokens, 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.getClassName(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);
}

/***
     Whether 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(assigns, value): {
            switch assigns[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:
            }
        }
    }

    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.keyw
ords.CONDITION_PATTERN_PARAMETER_NAME).objectValue.parameter(0))).sl
ice(1),

Interpreter.convert(...Parser.parse(Lexer.lex(memory.read(Little.keyw
ords.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 =  

```

```
@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_ST
RING} ()' => (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_ST
RING} ()' => (_, value, _) -> {
            return
Conversion.toLittleValue(Std.string(value.parameter(0)));
        },
    'public ${Little.keywords.TYPE_FLOAT}
${Little.keywords.TYPE_CAST_FUNCTION_PREFIX}${Little.keywords.TYPE_FL
OAT} ()' => (_, value, _) -> {
            return Decimal(value.parameter(0));
        },
    'public ${Little.keywords.TYPE_BOOLEAN}
${Little.keywords.TYPE_CAST_FUNCTION_PREFIX}${Little.keywords.TYPE_BO
OLEAN} ()' => (_, 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_ST
RING} ()' => (_, value, _) -> {
            return
Conversion.toLittleValue(Std.string(value.parameter(0)));
        },
        'public ${Little.keywords.TYPE_INT}'
${Little.keywords.TYPE_CAST_FUNCTION_PREFIX}${Little.keywords.TYPE_IN
T} ()' => (_, value, _) -> {
            return
Conversion.toLittleValue(Math.floor(value.parameter(0)));
        },
        'public ${Little.keywords.TYPE_BOOLEAN}'
${Little.keywords.TYPE_CAST_FUNCTION_PREFIX}${Little.keywords.TYPE_BO
OLEAN} ()' => (_, 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_ST
RING} ()' => (_, value, _) -> {
            return
Conversion.toLittleValue(Std.string(value.parameter(0)));
        },
        'public ${Little.keywords.TYPE_INT}'
${Little.keywords.TYPE_CAST_FUNCTION_PREFIX}${Little.keywords.TYPE_IN
T} ()' => (_, 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.toHaxe
Value(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.toHaxe
Value(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(p
arams[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).defaultInstanceState;
        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.toInt(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.toInt(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.toInt(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_N
AME, 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(Convers
ion.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-----
        // -----dashed-----

        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'));
    }
});
```

```
    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('Condition must be a Boolean value'), INTERPRETER);  
        }  
    });  
}
```

```
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:Boolean):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$d SplitLine\n';
        case Characters(string): return '${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: ${doc.replace("\n", "\n" + prefixFA(prefix) + '|')}';
        case Decimal(num): return '${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$d\n${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:IntpTokens,
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 just 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)}${Little.keywords.PROPERTY_ACCESS_SIGN}${str
ingifyParser(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 just 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;  
}  
}
```