Our language

We attempting to implement a fully working language based off of our own designs. We will start from pretty much scratch and build a lexer, parser, and evaluater for this language. Our language implementation is a true blend of imperative and functional programming languages. All statements such as If-statements and While-loops are considered expressions, and thus these statements can be assigned to variables.

Core features:

```
 - Dynamically typed variables
- Types: ints, strings, booleans, lists
- Conditional expressions: type matching, while loops, if-statements, etc.
- Ability to call and define functions
```

Demo Code

Fibonacci Sequence:

```
 // Hi, this is a single-line comment. Fibonacci is cool!
fun fibb ($a)
   {if ($a < 1) {0}
   elif ($a <= 2) {1}
   else { @fibb($a - 1) + @fibb($a - 2)}}
$a = 13
@fibb($a)
```

This programs evaluates to:

```
233
```

which is the correct value for the 13th number in the Fibonacci sequence.

Palindrome:

```
 fun isPalindrome($s)
{$s == (~$s)}
$e = "Hello World!"
$e = $e ^ (~$e)
// ^ is the symbol for concatenation. ~ is the symbol for string reversal
$d = [@isPalindrome("noon"), @isPalindrome("hii"), @isPalindrome($e)]
```

This program evaluates to:

```
[true, false, true]
```

Syntax

Note the parenthesis is merely a substitute for the actual AST representation. It should also be noted that parser parses binary operators right associatively if order precedences are the same. Boolean operators have no order precedence. Since we wrote the parser from scratch, we can give syntax-specific errors to the user.

```
not T or F --> not (true or false) --> false
```

```
T and (1 + 3) == 3 --> true and ((1 + 3) == 3) --> false
```

```
3 * 4 + 2 - 1 + 0 --> 3 * (4 + (2 - (1 + 0))) --> 13
```

```
2 * 2 * 2 + 2 * 2 * 2 --> ((2 * (2 * 2)) + (2 * (2 * 2)) --> 16
```

A final note should be made that the syntax follows a rigid grammar, and thus separators for expressions can simply be whitespace or semicolon. For example all the lines below parse equivalently:

```
$a = 1 $b = 2
```

```
$a=1;$b=2;
```

```
 $a=1
$b=2
```

Basic Expressions on Types:

--> denotes evaluation by our parser

Booleans

```
 T --> true
F --> false
T or F --> true
T and F --> false
not F or F --> false
(not F) or F --> true
T and (not F) --> true
```

Common Errors:

```
 T or --> Parser.SyntaxError: Assigning Boolean Values failed on line 1

T or 3 --> Failed assigning booleans on logical operators (perhaps add parenthesis) on line 1
```

Integers

Our parsing for integers only goes to a maximum of 64 bit signed integer. Also, negative numbers and the minus sign differ by a space, so "-4" is negative four, while "- 4" is minus four.

```
 3023 --> 3023
-892 --> -892
3 + -2 --> 1
3 - 2 --> 1
2 + 1 * 4 + 2 --> 8
(7 - 0) / 2 --> 3
```

Common Errors:

```
 9223372036854775808 --> Syntax error at line 1: Number expected or exceeded 64 bit signed integer, got 9223372036854775808

(7 -0) --> Syntax error on consuming <number> Expected RPAREN
```

Strings

There are no escape characters in our language. The parser simply parses from the first quotation mark to the next. So a newline has to be written as a literal newline, and there is no way to write a quotation mark in our language. Note that quotation marks are omitted when printing in evaluation

```
"hi" --> hi
"hello" ^ " " ^ "world" --> hello world
~ "0114" ^ "SC" --> CS4110
(~ "0114") ^ "SC" --> SC4110

// This evaluates to an integer, not a string
len("a" ^ "b") --> 2
```

Common Errors:

```
"oh"h --> Syntax error at line 1: Unrecognized character h

"a" ^ "b" ^ 3 --> Parser.SyntaxError: Assigning String failed on line 1

len("a b" --> Parser.SyntaxError: len method needs closing parenthesis on line number 1
```

Lists

Lists are dynamically typed, so we can put anything inside, although expressions will be evaluated to a value. There are five main built-in functions on lists on top of declaration.

```
[1, "2", T, [F, F]] --> [1, "2", T, [F, F]]
[while(F){"oh no"}, if(T) {"hi"}] --> [NULL, hi]

insert(["C", "W"], "O") --> [C, O, W]
insert([3, 2], "hi", 1) --> [3, hi, 2]

remove([3, 3, 5, 6], 0) --> [3, 5, 6]
remove(["CS", "is", "not", "fun"], 2) --> [CS, is, fun]

replace([1, 2, 3, 4], 0, 1) --> [1, 0, 3, 4]
replace([0, "no"], if(T) {"yay"}, 1) --> [0, yay]
```

Common Errors:

```
["ad",,] --> Parser.SyntaxError: Parse error on token ','

insert([0,1,2],"3" , 4) --> interpreter.EvaluationError: index out of bounds

remove([1,2,3], 4) --> interpreter.EvaluationError: index out of bounds

replace([43], "A", -1) --> interpreter.EvaluationError: index out of bounds
```

List Equality

```
$x = [1,"AS",[1,2],T]
$y = [1,"AS",[1,2],T]
$z = ["no","yes",[1,2],F]
$a = $x == $y
$x != $z
```

The Store evaluates to

```
a : true
x : [1, AS, [1, 2], true]
y : [1, AS, [1, 2], true]
z : [1, AS, [1, 2], false]
```

The Program evaluates to

```
true
```

Variable Assignment:

```
$a = T
$b = if ($a) {3}
var c = "ASDF"
var d = 3 + 2
$e = null
$f = $g = 0
```

The Store evaluates to

```
a : true
b : 3
c : ASDF
d : 5
e : NULL
f : 0
g : 0
```

The Program evaluates to

```
0
```

If statements:

Example 1

```
$c = if (T or F) {$a = 3 $b = 5}
```

The Store evaluates to

```
a : 3
b : 5
c : 5
```

The Program evaluates to

```
5
```

Example 2

```
$a = 42
if (T and F) {$a = 3 $b = 5}
elif (F) {$d = "hi"}
elif (T) {"I am here"}
else {$a}
```

The Store evaluates to

```
a : 42
```

The Program evaluates to

```
I am here
```

While loops:

Example 1

```
$a = "hi"
$c = while ($a == "b") {$a = 3 $b = 5}
```

The Store evaluates to

```
a : hi
c : NULL
```

The Program evaluates to

```
NULL
```

Example 2

```
$a = 0
$c = "hi"
while ($a < 3) {
$c = $c ^ $c
$a = $a + 1
// ^ is string concatenation
}
$c
```

The Store evaluates to

```
a : 3
c : hihihihihihihihi
```

The Program evaluates to

```
hihihihihihihihi
```

Functions and Calling Functions:

Example 1

```
fun double ($a) {$a = $a * 2}
fun mul ($a, $b) {$a = $a * $b}
$a = 2
$b = @double($a)
$c = @mul($a,$b)
@double(@mul($a, $b))
```

The Store evaluates to

```
a : 2
b : 4
c : 8
```

The Program evaluates to

```
16
```

Example 2

```
fun doubleList ($list) {
    $a = 0
    while ($a < size($list)) {
        $b = 2*get($list,$a)
        replace($list, $b, $a)
        $a = $a + 1
        $list
    }
}
@doubleList ([1,2,3,4,5,6])
```

The Store is empty.

The Program evaluates to

```
[2, 4, 6, 8, 10, 12]
```

Global Variables:

To declare a global variable 'x'. We can do as shown below (both are syntactically equivalent), we can have a global variable even in functions:

```
$x.
```

or

```
var x.
```

Example 1

```
fun add($b) {$x = $x + $b}
var x.
$x = 0
@add(3)
$x
```

The (Global) Store evaluates to

```
x : 3
```

The Program evaluates to

```
3
```

Example 2

```
fun doubleList ($list) {
    $a = 0
    while ($a < size($list)) {
        $b = 2*get($list,$a)
        replace($list, $b, $a)
        $a = $a + 1
        $list
    }
}
@doubleList ([1,2,3,4,5,6])
```

The Store is empty.

The Program evaluates to

```
[2, 4, 6, 8, 10, 12]
```

Example 3

Our language does not support function arguments sharing names with global variables.

```
fun add($x) {$x = $x + $b}
var x.
$x = 0
@add(3)
$x
```

This will throw the error:

```
interpreter.EvaluationError: The function's parameters overlap with existing global variables
```

Similarly:

```
fun add($x) {$x. $x = $x + $b}
$x = 0
@add(3)
$x
```

Throws the same error:

```
interpreter.EvaluationError: The function's parameters overlap with existing global variables
```

Type Matching:

Example 1

```
$x = [0,1,2]
match $x :
int : $x + 1
string : $x ^ $x
list : insert($x,3)
bool : if ($x) {not $x}
null : "null"
size($x)
```

The Store evaluates to:

```
x : [0, 1, 2, 3]
```

The Program evaluates to:

```
4
```

Example 2

We don't necessarily need to match to every type, so the example below is valid:

```
$x = "no"
match $x :
int : $x + 1
string : $x ^ (~$x)
```

The Store evaluates to:

```
x : no
```

The Program evaluates to:

```
noon
```

Example 3

If a type is not matched, then it will simply return null.

```
 $x = "no"
match $x :
int : $x + 1
bool : if ($x) {not $x}
null : 1
```

The Store evaluates to:

```
x : no
```

The Program evaluates to:

```
NULL
```

Example 4

Since only variables are dynamically typed, it only makes sense that match can only be performed on variables.

```
 $x = 1
match 3 :
int : $x + 1
```

Throws the error:

```
Parser.SyntaxError: A variable should be followed after keyword match
```

Other Errors (Syntax on If/While):

The below demonstrates our user-friendly parser, which can explicitly tell users where something has gone wrong. If statements all need parenthesis for the guard and brackets for the body:

```
 $a = 30
if (T) {$a
```

Throws the error

```
Parser.SyntaxError: If statement body needs a closing bracket on line number 2
```

and the program

```
if (T or F {"hi"}
```

Throws the error

```
Parser.SyntaxError: If statement guards need closing parenthesis on line number 1
```

Similarly for while loops:

```
 $a = 30
while ($a < 0)
{$a = $a - 1
```

Throws the error

```
Parser.SyntaxError: While statement body needs a closing bracket on line number 3
```

and

```
 $a = 30
while F) {$a}
```

Throws the error

```
Parser.SyntaxError: While statement guard needs parenthesis on line number 2
```