YARTBML Language Specification

1 Introduction

YARTBML is a functional programming langauge with a set of basic primitive and complex datatypes. It is a statically typed general purpose language that runs with its own interpreter built in GO. Memory management is handled by GO, and the interpreter is a binary file which was compiled by Go to the specific machine architecture.

2 Lexical Structure

This section specifies the lexical structure of the programming language.

2.1 Keywords

The language includes reserved keywords that have special meanings and cannot be used as identifiers.

```
<keyword> ::=
      "let"
    | "fn"
    | "return"
    | "if"
    | "else"
```

2.2 Literals

2.21 Integer Literals Integers are sequences of digits

```
<integer> ::= <digit>+
```

2.22 String Literals Strings are sequences of characters enclosed in double quotes.

```
<string> ::= "\"" <char>* "\""
<char> ::= <any character except newline or double quote>
```

2.23 Boolean Literals Booleans are true/false values

```
<boolean> ::= "true" | "false"
```

2.24 Array Literals Arrays can contain sequences of either integers or strings.

```
<array_literal> ::= "[", [ <element_list> ], "]"
<element_list> ::= <element> ( "," <element> )*
<element_literals> ::= <string> | <char> | <boolean> | <array_literal>
    | <hashmap_literal>
```

2.25 Hashmap Literals Hashmaps contain a string as the key and an expression as the value

```
<record_literal> ::= "{", [ <property_list> ], "}"
<property_list> ::= <property> ( "," <property> )*
<property> ::= <element_literal> ":" <element_literal>
```

2.3 Delimiters

Delimiters separate tokens in the code.

```
<delimiter> ::=
      "("
    | ")"
    | "{"
    | "}"
    | "["
    | "]"
    | ","
    | ";"
    | ":"
```

2.4 Operators

Operators are symbols used to perform operations on values.

```
<operator> ::=
      "+"
    | "-"
    | "*"
    | "/"
    | "=="
    | "!="
    | "<"
    | ">"
    | "<="
    | ">="
```

2.5 Identifiers

Identifiers are sequences of letters, digits, and underscores that do not start with a digit.

```
<identifier> ::= <letter> (<letter>
                           | <digit>
                           | "_")*
<letter> ::=
      "a"
    | "b"
    | ...
    | "z"
    | "A"
    | "B"
    | ...
    | "Z"
<digit> ::=
      "0"
    | "1"
    | ...
    | "9"
```

2.6 White Space

Whitespace characters include spaces, tabs, and newline characters and are used to separate tokens and improve code readability.

```
<whitespace> ::=
      <space>
    | <tab>
    | <newline>
<space> ::= " "
<tab> ::= "\t"
<newline> ::= "\n"
```

3 Grammar

This section specifies the grammar of the language

3.1 Binding Values

The YARTBML REPL allows users to bind values to names using the let statement.

```
<let_statement> ::= "let" <identifier> "=" <element_literal> ";"
```

3.2 Supported Data Types

In addition to integers, booleans, and strings, YARTBML supports arrays and hashmaps.

3.5 Accessing Elements

Elements in arrays and hashmaps are accessed using index expressions.

```
Hashmaps
<index_expression> ::= <identifier> "[" <expression> "]"
<index_expression> ::= <identifier> "[" <string> "]"

Array
<index_expression> ::= <identifier> "[" <expression> "]"
<index_expression> ::= <identifier> "[" <int> "]"
```

3.6 Binding Functions

Functions can be bound to names using the `let` statement, with optional **`return`** statements.

```
<let_statement> ::=
    "let" <identifier> "=" "fn" "(" <parameters> ")" <block_statement>
<block_statement> ::= "{" <statements>* "}"
<statements> ::=
      <let_statement>
    | <expression> ";"
<expression> ::=
      <return_statement>
    | <assignment_statement>
<return_statement> ::= "return" <expression>
<assignment_statement> ::= <expression> ";"
```

3.7 Calling Functions

Functions are called by their names followed by arguments.

```
<function_call> ::= <identifier> "(" <arguments> ")"
<arguments> ::= <expression> ("," <expression>)*
```

3.8 Recursive Functions

Recursive functions are supported, enabling functions to call themselves.

3.9 Higher Order Functions

YARTBML also supports higher-order functions, which are functions that take other functions as arguments.

```
<let_statement> ::=
    "let" <identifier> "=" "fn" "(" <parameters> ")" <block_statement>
<parameters> ::= <identifier> ("," <identifier>)*
```

3.10 Selection Sequences

YARTBML supports control flow using the **`if`** keyword followed by the expression to evaluate then an optional **`else`**. If the value is **`true`** the preceding block statement is evaluated, if **`false`** the else statement is evaluated.

```
<if_statement> ::=
    "if" "(" <expression> ")" <block_statement>
    "else" <block_statement>
```

4 Scoping Rules

YARTBML has lexical scoping, meaning that the scope of a variable is determined by its location in the source code. Variables declared in outer scopes are accessible in inner scopes unless shadowed by variables with the same name. YARTBML supports block-level scoping.

5 Example Program

```
let age = 1;
let name = "YARTBML";
let result = 10 * (20 / 2);

let myArray = [1, 2, 3, 4, 5];
let john = {"name": "John", "age": 28};

myArray[0] // => 1
john["name"] // => "John"

let add = fn(a, b) { a + b; };

add(1, 2); // => 3

let fibonacci = fn(x) {
    if (x == 0) {
        0
    } else {
        if (x == 1) {
            1
        } else {
            fibonacci(x - 1) + fibonacci(x - 2);
        }
    }
};
let twice = fn(f, x) {
    return f(f(x));
};
let addTwo = fn(x) {
    return x + 2;
};
twice(addTwo, 2); // => 6
```

5 REPL (Read Eval Print Loop)

YARTBML uses a REPL to read input, send it to the interpreter for evaluation, print the result/output of the interpreter

5.1 CLI Tool

YARTBML also has a CLI Tool to help interpret whole files of YARTBML code (.ybml is the file extension).

Parsing and Interpretation order

YARTBML employs recursive descent for parsing, specifically utilizing the PRATT parsing algorithm to enhance parsing speed. A tree walker is then employed to interpret the Abstract Syntax Tree (AST) produced by the parser.

6 Appendix - Complete EBNF Form

```
<program>                   ::= <statement-list>
<statement-list>            ::= { <statement> }
<statement>                 ::= <let-statement>
                              | <return-statement>
                              | <expression-statement>
<let-statement>             ::= "let" <identifier> "=" <expression> ";"
<return-statement>          ::= "return" <expression> ";"
<expression-statement>      ::= <expression> ";"
<block-statement>           ::= "{" <statement-list> "}"

<expression>                ::= <equality-expression>
<equality-expression>       ::= <comparative-expression> {("==" | "!=")
    <comparative-expression>}
<comparative-expression>    ::= <additive-expression> {("<" | ">") <
   additive-expression>}
<additive-expression>       ::= <multiplicative-expression> {("+" | "-"
   ) <multiplicative-expression>}
<multiplicative-expression> ::= <prefix-expression> {("*" | "/") <
   prefix-expression>}
<prefix-expression>         ::= ("-" | "!") <prefix-expression>
                              | <postfix-expression>
<postfix-expression>        ::= <primary-expression> {<call-postfix> |
   <index-postfix>}
<call-postfix>              ::= "(" [<expression-list>] ")"
<index-postfix>             ::= "[" <expression> "]"
<primary-expression>        ::= <grouped-expression>
                              | <if-expression>
                              | <function>
                              | <identifier>
                              | <value>

<prefix-expression>         ::= ("-" | "!") <expression>
<grouped-expression>        ::= "(" <expression> ")"
<if-expression>             ::= "if" "(" <expression> ")" <block-
   statement> ["else" <block-statement>]
<function>                  ::= "fn" "(" [<parameter-list>] ")" <block-
   statement>
<identifier>                ::= <alpha> { <alpha> | <digit> | "_" }
<value>                     ::= <int>
```

```
                                  | <bool>
                                  | <string>
                                  | <array>
                                  | <hash>

<int>                           ::= <digit> { <digit> }
<digit>                         ::= "0..9"
<alpha>                         ::= "a..zA..Z"
<bool>                          ::= "true" | "false"
<string>                        ::= """ { <~any valid non-quotation-marks
    character> } """
<array>                         ::= "[" [<expression-list>] "]"
<hash>                          ::= "{" [<key-value-pairs>] "}"
<key-value-pairs>               ::= <expression> ":" <expression> { "," <
    expression> ":" <expression> }

<expression-list>               ::= <expression> { "," <expression> }
<parameter-list>                ::= <identifier> { "," <identifier> }
```