CoBruh LRM

Joshua Zhou (Language Guru)
Akash Nayar (System Architect)
Lance Wong (Tester)

1 Introduction

CoBruh is an imperative language that emphasizes human readability. With a strong and static type system, CoBruh makes it easy for beginner programmers to learn the basics of coding. CoBruh draws upon Python's readability while restricting its sometimes unpredictable type system. Thus, programmers enjoy type safety alongside easily understood source code.

2 Lexical Conventions

2.1 Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and separators. Tabs and carriage returns are ignored. Spaces are ignored unless they appear at the start of a line. In this case, spaces are used to denote a new scope (if-else body, loop body, and function body). A new scope can be defined with two consecutive spaces.

```
whitespace = ['\t' '\r']
indent = "  "
```

Comments are achieved by using the # symbol. The comment ends at the end of the line.

2.2 Identifiers

An identifier is a sequence of letters, digits, and underscores where the first character is a letter. Identifiers are case sensitive.

```
letter = ['a'-'z' 'A'-'Z']
number = ['0'-'9']
id = letter ( letter | number | '_' )*
```

2.3 Keywords

```
boolean
character
continue
define
false
is
loop
none
number
return
stop
```

```
string
true
```

2.4 Literals

There are several types of literals:

A number constant consists of an integer part, a decimal point, a fraction part, an `e`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Both the integer and fraction parts must be present (i.e., `1.` and `.5` are not valid). Every number constant is taken to be 64-bit double precision.

```
digit = ['0'-'9']
exponent = ('E' | 'e') digit+
number = digit+ ('.' digit+)? exponent?
```

Boolean literals are one of two valid sequences of characters, `true` or `false`.

```
boolean = "true" | "false"
```

Character literals are single characters surrounded by single quotes (`'`).

```
ascii = [' '-'!' '#'-'[' ']'-'~']
character = ''' ascii '''
```

String constants are sequences of characters surrounded by double quotes (").

```
string = '"' ascii* '"'
```

Array literals are a sequence of non-array elements separated by commas and surrounded by square brackets. Every element in an array must be the same datatype. Array literals cannot be empty; in other words, they must contain at least one element.

```
LSQUARE expr_list RSQUARE
```

2.5 Operators

Operators are specific lexical elements reserved for use by the language.

- Arithmetic: `+ - * / // %`
- Assignment: `is`
- Equivalence: `== =/= < <= > >=`
- Logical: `not and or`

2.6 Separators

Separators are used to denote the separation between tokens. Parentheses are used to specify the order of operations. Square brackets are used to during array assignment and indexing. Colons denote the start of an if-else, loop, or function body. Commas are used to separate parameters in function definitions and calls.

```
( ) [ ] : ,
```

3 Types

3.1 Primitive Data Types

number
The `number` type stores numbers in a 64-bit double precision floating point representation.

```
number h is 6.626e-34
```

boolean
The `boolean` type stores either true or false in 8 bits.

```
boolean is_cobruh_good is true
```

character
The `character` type stores a single character in 8 bits.

```
character z is 'z'
```

3.2 Non-Primitive Data Types

string
The `string` type represents an immutable sequence of valid UTF-8 characters either as a literal constant or as some kind of variable.

```
string fact is "CoBruh is beginner friendly!"
```

array
The `array` type represents a mutable constant-size list of typed elements, defined by `<array name> is <type>[<array size>]` (global allocation), `<type>[<array size>] <array name>` (local allocation), or `<array name> is [<array contents>]` (local allocation and initialization). The size of the array need not be known at compile time. The only exception to this is global arrays: the user must input a number literal for the array's size.

The datatype of the array can be dropped if the datatype of the list elements can be inferred. Elements are 0-indexed using square brackets in the form of `<array name>[<element number>]`.

For both array size and indexes, the numbers inputted are rounded down to the closest integer.

```
scores is number[10] # global declaration of array
number[10] temp      # local declaration of array
fibonacci is [0, 1, 1, 2, 3, 5, 8, 13] # type inference
number[5] even is [0, 2, 4, 6, 8]      # explicit definition
```

3.3 Special Data Types

none
The `none` type represents an empty value and is used only by functions that do not take in and/or return values. It cannot be used as a variable type.

```
define print_facts (none -> none):
    say("CoBruh is awesome!")
```

any
The `any` type cannot be accessed by the user and is used internally. The `say` and `shout` functions both take in `any`, allowing the user to print any type without casting. This is done for convenience.

```
shout("Hello world")
shout(1)
```

4 Operators

4.1 Arithmetic

The binary arithmetic operators are `+`, `-`, `*`, `/`, `//`, `%`. Integer division (`//`) truncates any fractional part of the result of the division. The binary + and − operators have the same precedence, which is lower than the precedence of `*`, `/`, `//`, `%` . Arithmetic operators associate left to right. They can be applied only to numbers.

```
(1 + 5) * 7 // 2 # evaluates to 21
```

4.2 Assignment

The `is` operator assigns a variable to an expression. If a datatype is not given, the type will be inferred. Whether the data type is specified leads to different behaviors in the context of scoping. If the data type is specified, then only the current scope will be checked when creating the variable. A variable with the same name in an outer scope is not affected. On the other hand, if the data type is omitted, then the current and outer scopes are checked. A variable with the same name in an outer scope will be reassigned.

```
number n is 2 # 2
x is n + 2    # 4
```

```
| dtype ID ASSIGN expr EOL { Assign ($1, $2, $4) }
| ID ASSIGN expr EOL { InferAssign (Id $1, $3) }
```

4.3 Equivalence

The relational operators are <, <=, >, >=. They all have the same precedence. Below them are == and =/=. Relational operators have lower precedence than arithmetic operators. They can also only be applied on numbers.

```
5 == 5   # true
10 =/= 5 # true
10 > 5   # true
10 >= 10 # true
```

```
| expr EQ expr { Binop ($1, Eq, $3) }
| expr NEQ expr { Binop ($1, Neq, $3) }
| expr LT expr { Binop ($1, Less, $3) }
| expr LEQ expr { Binop ($1, Leq, $3) }
| expr GT expr { Binop ($1, Greater, $3) }
| expr GEQ expr { Binop ($1, Geq, $3) }
```

4.4 Logical

The logical operators are `not`, `and`, and `or` (listed in order from highest to lowest precedence). They can be applied only on booleans.

```
if not is_cobruh_python
if is_right and is_moral
if is_red or is_green
```

```
| expr AND expr { Binop ($1, And, $3) }
| expr OR expr { Binop ($1, Or, $3) }
| NOT expr { Unop (Not, $2) }
```

4.5 Other Unary Operators

The unary negation operator (-) negates a number type. It has higher precendence than arithmetic operators. The pipe operator (two pipes `||` that surround a value) computes the absolute value of a number.

```
number a is 21
shout(-a)   # -21
shout(|-5|) # 5
```

5 Statements and Expressions

5.1 Declarations and Initialization

Variables are declared and initialized through assignment statements. Such statements consist of an optional data type, a variable name, and a value to be stored in the variable. Variables persist from the point of declaration until the end of their enclosing scope.

```
number x is 4115 # 4115
y is x - 4       # 4111
```

5.2 Literal Values

Each of the data types can be defined in a literal fashion.

```
3.14             # number literal expression
true             # boolean literal expression
'B'              # character literal expression
"CoBruh is fun!" # string literal expression
```

5.3 Array Expressions

Array expressions are written by enclosing more than zero comma-separated expressions of the same type in square brackets.

```
[2, 4, 6, 8]                     # number array
['C', 'o', 'B', 'r', 'u', 'h']   # character array
["CoBruh", "is", "awesome", "!"] # string array
```

5.4 Functions

A function defines a sequence of statements. It consists of a name, a set of parameters, and a return type. All of these fields are required to define a function, which is done with the `define` keyword. A function to compute a base raised to an exponent is shown below.

```
define power (number base, number exponent -> number):
    result is 1.
    loop i in 0 to exponent:
        result is result * base.
    return base.
```

A function that takes in no parameters has `none` in the parameters section. A function that does not return has `none` as its return type. A function with a non-none return type must return the specified type; failure to do so (returning an incorrect type or potentially never reaching a return statement) results in a compile-time error.

Built-in functions include `say` and `shout` for outputting to stdout, and `inputn` and `inputc` for reading user number and character input from stdin. The only difference between `say` and `shout` is that `shout` appends a newline character to the output. These two function can be called on any datatype except for arrays. `inputn` and `inputc` return the next integer and character found on `stdin`, respectively. Below is an example of these four functions.

```
shout(inputn()) # echo the next number
c is inputc()   # store the next character
say(c)          # print that character without a newline
```

5.5 Control Flow

Conditional logic through if statements can be used to control the flow of the program. The structure of an if statement is a conditional expression, a block of code, and an optional else statement. The conditional expression must evaluate to a boolean. If it is true, then the if code block runs. If it is false, then the else code block runs if it exists.

```
if is_valid:
    say("Valid")
else:
    say("Invalid")
```

```
| IF expr COLON EOL INDENT stmt_list DEDENT { If ($2, $6, [])
    }
| IF expr COLON EOL INDENT stmt_list DEDENT ELSE COLON EOL
INDENT stmt_list DEDENT { If ($2, $6, $12) }
```

CoBruh supports looping while a given predicate value (boolean expression) is true. These are similar to `while` loops in Python. The predicate is evaluated each time the loop is run. The `stop` and `continue` keywords can be used to break out of the loop or continue to the next iteration, respectively. Below is an example of loops in CoBruh.

```
i is 0
loop i < 10:
  i is i + 1
  if i % 2 == 0:
    continue
  shout(i)
```

```
| LOOP expr COLON EOL INDENT stmt_list DEDENT { CondLoop ($2,
    $6) }
```

6 Type Inference

When declaring a variable, if a human can infer the type, so can CoBruh. Consider the following examples:

```
# it's clear that x is of type number
x is 5

# since x is a number, so is y
y is x

# both x and y are numbers, so z is a number
z is x + y

# not only is it impossible to infer the type of this
# expression, it is also illogical and will produce a type
# mismatch error
a is z + "Hello"
```

Type inference can also be used with arrays in the following manner:

```
fibonacci is [0, 1, 1, 2, 3]
vowels is ['a', 'e', 'i', 'o', 'u']
authors is ["Akash", "Joshua", "Lance"]

first is fibonacci[0] # number
```

7 Scanner

```
{
  open Parser

  let excess_indent_err = "too many indentations"
  let extra_space_err = "extra space"
  let mismatched_quote_err = "mismatched quotation"
  let illegal_character_err c = "illegal character " ^ Char.
   escaped c

  let is_start_of_line = ref true
  let curr_scope = ref 0 (* same as number of indents *)
  let new_spacing = ref 0 (* counts number of single spaces
   at start of line *)

  let set_new_line () = is_start_of_line := true; new_spacing
    := 0

  let get_scope () =
    if !new_spacing land 1 = 1 (* if new_spacing is odd *)
   then raise (Failure extra_space_err)
    else !new_spacing / 2

  let rec make_dedent_list num_dedents =
    if num_dedents = 0 then []
    else DEDENT::(make_dedent_list (num_dedents - 1))

  let dedent_to_zero () = make_dedent_list !curr_scope

  (* should be called on tokens that may appear at start of
   line *)
  let push token =
    if !is_start_of_line then (
      is_start_of_line := false;
      let new_scope = get_scope () in
      let scope_diff = new_scope - !curr_scope in
      curr_scope := new_scope;
      if scope_diff > 1 then raise (Failure excess_indent_err
   )
      else if scope_diff = 1 then [INDENT; token]
      else if scope_diff = 0 then [token]
      else (* scope_diff < 0 *) (make_dedent_list (-
    scope_diff)) @ [token]
    ) else [token]
}

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let ascii = [' '-'!' '#'-'[' ']'-'~']

let exponent = ('E' | 'e') digit+
let number = digit+ ('.' digit+)? exponent?
let id = letter (letter | digit | '_')*
let character = ''' ascii '''
let string = '"' ascii* '"'

rule token = parse
['\t' '\r'] { token lexbuf } (* whitespace *)
| ' '       { if !is_start_of_line then new_spacing := !
   new_spacing + 1 else (); token lexbuf }
| '\n'      { let was_start = !is_start_of_line in
   set_new_line (); if not was_start then EOL::(token lexbuf)
    else token lexbuf }
| '#'       { comment lexbuf } (* comment *)

(* Symbols *)
| '(' { push LPAREN }
| ')' { push RPAREN }
| '[' { push LSQUARE }
| ']' { push RSQUARE }
| ',' { push COMMA }
| ':' { push COLON }
| '|' { push PIPE }

(* Operators *)
| "is"  { push ASSIGN }
| '+'   { push PLUS }
| '-'   { push MINUS }
| '*'   { push TIMES }
| "//"  { push INTDIV }
| '/'   { push DIV }
| '%'   { push MOD }
| "=="  { push EQ }
| "=/=" { push NEQ }
| '<'   { push LT }
| "<="  { push LEQ }
| '>'   { push GT }
| ">="  { push GEQ }
| "and" { push AND }
| "or"  { push OR }
| "not" { push NOT }

(* Branching *)
| "if"       { push IF }
| "else"     { push ELSE }
| "loop"     { push LOOP }
| "continue" { push CONTINUE }
| "stop"     { push STOP }

(* Functions *)
| "define" { push DEFINE }
| "none"   { push NONE }
| "->"     { push GIVES }
| "return" { push RETURN }

(* Data Types *)
| "number"    { push NUMBER }
| "boolean"   { push BOOL }
| "character" { push CHAR }
| "string"    { push STRING }

(* Literals *)
| number as lex    { push (NUMBERLIT (float_of_string lex)) }
| "true"           { push (BOOLLIT true) }
| "false"          { push (BOOLLIT false) }
| character as lex { push (CHARLIT lex.[1]) }
| string as lex    { push (STRINGLIT (String.sub lex 1 (
   String.length lex - 2))) }
| id as lex        { push (ID lex) }

| eof         { if not !is_start_of_line then EOL::(
   dedent_to_zero () @ [EOF]) else dedent_to_zero () @ [EOF]
   }
| ('"' | ''') { raise (Failure(mismatched_quote_err)) }
| _ as c      { raise (Failure(illegal_character_err c)) }

and comment = parse
  '\n' { let was_start = !is_start_of_line in set_new_line ()
   ; if not was_start then EOL::(token lexbuf) else token
   lexbuf }
```

```
| eof  { if not !is_start_of_line then EOL::(dedent_to_zero
    () @ [EOF]) else dedent_to_zero () @ [EOF] }
| _    { comment lexbuf }
```

8 Grammar

```
%{
  open Ast
%}

%token INDENT DEDENT EOL
%token LPAREN RPAREN LSQUARE RSQUARE COMMA COLON PIPE
%token ASSIGN PLUS MINUS TIMES INTDIV DIV MOD EQ NEQ LT LEQ
   GT GEQ AND OR NOT
%token IF ELSE LOOP CONTINUE STOP
%token DEFINE NONE GIVES RETURN
%token NUMBER BOOL CHAR STRING
%token USE
%token TAB
%token EOF
%token <float> NUMBERLIT
%token <bool> BOOLLIT
%token <char> CHARLIT
%token <string> STRINGLIT
%token <string> ID

%start program
%type <Ast.program> program

%nonassoc ASSIGN
%left OR
%left AND
%right NOT
%nonassoc EQ NEQ
%nonassoc LT LEQ GT GEQ
%left PLUS MINUS
%left TIMES INTDIV DIV MOD
%nonassoc UMINUS
%nonassoc LSQUARE RSQUARE

%%

program:
  decls stmt_list EOF { (List.rev (fst $1), List.rev (snd $1)
   , $2) }

decls:
    /* nothing */ { ([], [])                }
  | decls vdecl { (($2 :: fst $1), snd $1) }
  | decls fdecl { (fst $1, ($2 :: snd $1)) }

vdecl:
  ID ASSIGN global_dtype EOL { ($3, $1) }

bind:
  dtype ID { ($1, $2) }

params_list:
    bind                    { [$1] }
  | bind COMMA params_list { $1::$3 }

opt_params_list:
    NONE        { [] }
  | params_list { $1 }

func_rtype:
    dtype { $1 }
  | NONE  { None }

fdecl:
  DEFINE ID LPAREN opt_params_list GIVES func_rtype RPAREN
   COLON EOL INDENT stmt_list DEDENT
  {
    {
      fname=$2;
      params=$4;
      rtype=$6;
      body=$11
    }
  }

fcall:
  ID LPAREN expr_list_opt RPAREN { ($1, $3) }

expr:
    atom                       { $1 }
  | LSQUARE expr_list RSQUARE { ArrayLit $2 }
  | ID                         { Id $1 }
  | expr LSQUARE expr RSQUARE { Elem ($1, $3) }
  | expr PLUS expr             { Binop ($1, Plus, $3) }
  | expr MINUS expr            { Binop ($1, Minus, $3) }
  | expr TIMES expr           { Binop ($1, Times, $3) }
  | expr INTDIV expr          { ECall ("intdiv", [$1; $3]) }
  | expr DIV expr             { Binop ($1, Div, $3) }
  | expr MOD expr             { Binop ($1, Mod, $3) }
  | MINUS expr %prec UMINUS   { Unop (Neg, $2) }
  | PIPE expr PIPE            { ECall ("abs", [$2]) }
  | expr EQ expr              { Binop ($1, Eq, $3) }
  | expr NEQ expr             { Binop ($1, Neq, $3) }
  | expr LT expr              { Binop ($1, Less, $3) }
  | expr LEQ expr             { Binop ($1, Leq, $3) }
  | expr GT expr              { Binop ($1, Greater, $3) }
  | expr GEQ expr             { Binop ($1, Geq, $3) }
  | expr AND expr             { Binop ($1, And, $3) }
  | expr OR expr              { Binop ($1, Or, $3) }
  | NOT expr                  { Unop (Not, $2) }
  | LPAREN expr RPAREN        { $2 }
  | fcall                     { ECall (fst $1, snd $1) }

atom:
    NUMBERLIT { NumberLit $1 }
  | BOOLLIT   { BoolLit $1 }
  | CHARLIT   { CharLit $1 }
  | STRINGLIT { StringLit $1 }

expr_list_opt:
    /* nothing */ { [] }
  | expr_list     { $1 }

expr_list:
    expr                 { [$1] }
  | expr COMMA expr_list { $1::$3 }

stmt_list:
    /* nothing */  { [] }
  | stmt stmt_list { $1::$2 }

stmt:
    dtype ID ASSIGN expr EOL
                           { Assign ($1, $2, $4) }
  | ID ASSIGN expr EOL
                           { InferAssign (Id $1, $3) }
  | ID LSQUARE expr RSQUARE ASSIGN expr EOL
                           { InferAssign (Elem (Id $1, $3),
   $6) }
  | atomic_dtype LSQUARE expr RSQUARE ID EOL
                            { Alloc ($1, $3, $5) }
  | IF expr COLON EOL INDENT stmt_list DEDENT
                            { If ($2, $6, []) }
  | IF expr COLON EOL INDENT stmt_list DEDENT ELSE COLON EOL
   INDENT stmt_list DEDENT { If ($2, $6, $12) }
  | LOOP expr COLON EOL INDENT stmt_list DEDENT
                            { CondLoop ($2, $6) }
  | RETURN expr EOL
                            { Return $2 }
  | CONTINUE EOL
                            { Continue }
  | STOP EOL
                            { Stop }
  | fcall EOL
                            { SCall (fst $1, snd $1) }

atomic_dtype:
    NUMBER { Number }
  | BOOL   { Bool }
  | CHAR   { Char }
  | STRING { String }

dtype:
    atomic_dtype                 { $1 }
  | atomic_dtype LSQUARE RSQUARE { Array $1 }

global_dtype:
    atomic_dtype                           { $1 }
  | atomic_dtype LSQUARE NUMBERLIT RSQUARE { FixedArray ($1,
   $3) }
```