CoBruh Report

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

1 Introduction

CoBruh was designed as a language meant for beginner programmers. As developers, we recognized the benefits of an easily readable language such as Python and wanted to design our language to be accessible. At the same time, we realized how Python's lenience in terms of variable types could lead beginner programmers to shoot themselves in the foot. Thus, we decided to take the type-safety and statically-typed system from C and combine it with the readability of Python to create a language that is powerful and fast yet also beginner-friendly.

As a language meant for beginner programmers, CoBruh shines in simpler contexts. It can be used to write simple novice programs, (basic calculator, temperature conversion, etc.), and also some of the easier, less data-structure-dependent LeetCode problems (problems 7 and 322 are perfect examples).

2 Language Tutorial

2.1 Global Declarations

CoBruh source files are conventionally denoted with a `.bruh` file extension. Source files must also abide the following structure in order to be compiled (represented as a regular expression):

```
program = (decl | fdecl)* stmts
```

In the above regular expression, `decl` refers to the declaration of a global variable, `fdecl` refers to the declaration of a function, and `stmts` is a list of statements the user wishes to execute (such as those within a `main` method in C). A global variable can be declared by the following syntax: `<name> is <datatype>`. For instance, the declaration of four global variables could be done as follows:

```
b is boolean
c is character
n is number
s is string
```

Notice the absence of statement-delimiting characters such as semicolons. This aids in code-readability by reducing the number of unnecessary symbols.

The types `boolean` (8-bit integer), `character` (8-bit integer), `number` (64-bit floating point), and `string` (array of 8-bit integers) are the four datatypes of CoBruh, and arrays can be created from each of these datatypes. The elements in an array must be be a single datatype and must match the arrays declared datatype. They can either be declared without initialization (`a is number[5]` globally or `number[5] a` locally) or with initialization, in which case type inference will infer the type (`a is [1, 2, 3]`). They can be indexed just as in C with the `arrayname[index]` syntax. Single-line comments can be achieved by an octothorp (#) followed by the comment itself.

2.2 Function Declarations

Function declarations are the done with a simple and understandable syntax:

```
define exp (number base, number exponent -> number):
  result is 1
  counter is 0
  loop counter < exponent:
    result is result * base
    counter is counter + 1
  return result
```

All function definitions must start with the `define` keyword. This is then followed by the name of the function. Next comes the parameters and return type, which are enclosed in parenthesis and followed by a colon, The parameters themselves are specified by `datatype name`, and can be delimited by a comma. The return type of the function is declared with `-> rtype`. If the function is not meant to return anything, this value can be set to `none`, and if the function does not take in any parameters, they too can be set to `none` (for example, `define say_hi (none -> none)`). These are the only two places the `none` keyword is allowed in CoBruh.

The body of the function must be indented by 2 spaces. In fact, any body of code (including function, loop, and if/else) must be indented by two additional spaces). This is how CoBruh knows which code belongs to which block, similar to Python.

Loops can be implemented in the following way:

```
loop <boolean_expr>:
  <loop body>
```

2.3 Statements

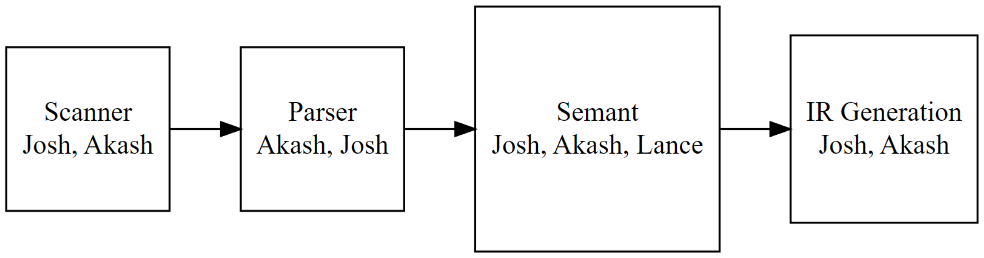
The last section of any CoBruh program is the statements. These are the lines of code that are run when the file is executed, similar to the code in the `main` method of a C program. These can be any statements, including variable declaration and assignment, if/else blocks, loops, and function calls. Below is a sample CoBruh source code file:

```
# global variable declarations
string age_string

# function declarations
define is_adult(number age -> boolean):
  if age >= 18:
    return true
  else:
    return false

# statements
# read in the user's age
say("Please input your age: ")
user_age is inputn()

# determine if they are an adult or child
if is_adult(user_age):
  age_string is "an adult"
else:
  age_string is "a child"

# break the news
say("You are ")
say(age_string)
shout(".")
```

User input can be achieved with the `inputn()` (numbers) and `inputc()` (characters) functions, Output to `stdout` is done with the `say` (output without new line) and `shout` (output with new line) commands.

3 Architectural Design

Below is a block diagram representing the pipeline of our translator. Each box represents one step in the process and says who worked on that component.



The scanner takes in the input source code and converts the text into a stream of tokens. CoBruh uses significant whitespace, so instead of simply outputting each token one by one, there are cases where we must output a list of tokens. This could include adding an `INDENT` token when entering a new code block or adding one or more `DEDENT` tokens when exiting one or multiple scopes. Thus, we had to use a queue to read in the tokens and send them to the parser one by one.

The parser takes these tokens and creates the Abstract Syntax Tree (AST) using context-free grammars (CFG) we defined. This AST is then passed into the semantics checker, which produces a semantically-checked AST that verifies scopes and types. Lastly, the SAST is passed into the IR generation phase where we generate LLVM IR.

4 Test Plan

Here are some sample CoBruh programs:

Code:

```
define square(number n -> number):
  return n * n

say(square(5))
```

IR:

```
; ModuleID = 'CoBruh'
source_filename = "CoBruh"

@fmt = private unnamed_addr constant [3 x i8] c"%g\00", align
     1

declare void @printf(i8*, ...)

declare void @scanf(i8*, ...)

define double @square(double %n) {
entry:
  %n1 = alloca double, align 8
  store double %n, double* %n1, align 8
  %n2 = load double, double* %n1, align 8
  %n3 = load double, double* %n1, align 8
  %bop = fmul double %n2, %n3
  ret double %bop
}

define void @main() {
entry:
  %square_result = call double @square(double 5.000000e+00)
  call void (i8*, ...) @printf(i8* getelementptr inbounds ([3
     x i8], [3 x i8]* @fmt, i32 0, i32 0), double %
    square_result)
  ret void
}
```

Code:

```
define factorial(number n -> number):
  if n <= 1:
    return 1
  return n * factorial(n - 1) # functions can call
   themselvesz

shout(factorial(inputn()))
```

IR:

```
; ModuleID = 'CoBruh'
source_filename = "CoBruh"

@fmt = private unnamed_addr constant [4 x i8] c"%lf\00",
   align 1
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00",
   align 1

declare void @printf(i8*, ...)

declare void @scanf(i8*, ...)

define double @factorial(double %n) {
entry:
  %n1 = alloca double, align 8
  store double %n, double* %n1, align 8
  %n2 = load double, double* %n1, align 8
  %bop = fcmp ole double %n2, 1.000000e+00
  br i1 %bop, label %then, label %else

merge_if:                                         ; preds = %
   else
  %n3 = load double, double* %n1, align 8
  %n4 = load double, double* %n1, align 8
  %bop5 = fsub double %n4, 1.000000e+00
  %factorial_result = call double @factorial(double %bop5)
  %bop6 = fmul double %n3, %factorial_result
  ret double %bop6

then:                                             ; preds = %
   entry
  ret double 1.000000e+00

else:                                             ; preds = %
   entry
  br label %merge_if
}

define void @main() {
entry:
  %scanin = alloca double, align 8
  call void (i8*, ...) @scanf(i8* getelementptr inbounds ([4
   x i8], [4 x i8]* @fmt, i32 0, i32 0), double* %scanin)
  %scanin1 = load double, double* %scanin, align 8
  %factorial_result = call double @factorial(double %scanin1)
  call void (i8*, ...) @printf(i8* getelementptr inbounds ([4
    x i8], [4 x i8]* @fmt.1, i32 0, i32 0), double %
   factorial_result)
  ret void
}
```

Testing was conducted with a Python script. The script simply runs each `.bruh` file in the `tests` directory and compares the output with the expected output in the corresponding `.err` or `.out`. If any test fails, it will alert the user as to which test failed and compare the expected and observed outputs. The script can be run by executing `dune test` from the root directory or `python3 testall.py` from the `/test` directory.

5 Summary

5.1 Team Member Contributions

1. Akash: Parser core, Significant whitespace v1, If/else IRgen, Loops IRgen, Unop IRgen, User input IRgen, Stop/continue IRgen

2. Joshua: Scanner core, Semant core, Type inference, Significant whitespace v2, Assignment IRgen, Scanner upgrade, Strings/printing in IRgen, Arrays

3. Lance: Scanner test cases, Parser test cases, Semant test cases, Misc test cases, Bug hunting/fixing

5.2 Important Takeaways

1. Akash

    - When I learned about context-free grammars (CFG) in CS theory, I though that I would never have to use them again. However, they turned out to be instrumental in the definition of our language's parser, and allow us to concisely and powerfully define the syntax of CoBruh.
    - At first I was skeptical of OCaml and its apparent benefits but after writing a compiler in OCaml I can confirm that it is indeed and incredibly powerful, reliant, and fun-to-use language. I would highly recommend it to anyone writing a compiler.

2. Joshua

    - I've always wanted to explore functional programming, so learning OCaml and writing a compiler in it was awesome. My favorite feature of OCaml is the exhaustive pattern matching; it's definitely caught many forgetten cases for me.
    - I discovered how powerful semantics checking is. Figuring out how to check types and scopes was a fascinating problem that required a lot of clever problem solving. The logic was surprisingly simple to implement using recursion in OCaml.

3. Lance

    - I learned that OCaml is a very powerful language that has a steep learning curve but once learned is very useful in that code is short, concise, and super satisfying once compiled. It's really cool that once the code compiles, we can be confident that it works most of the time.
    - It was also super interesting to learn each step in how code goes from a certain human-readable language all the way to machine code which gets executed to return a result.

5.3 Advice for Future Teams

The best advice for future teams would be to be on top of things. It is super easy to fall behind with this project and it is imperative to set reachable goals and try to reach them. This will help you make sure you complete your project in time and have a finished product.

Some other helpful advice would be to not be afraid of editing language details. Sometimes you might promise to deliver some really cool features, but realize later down the line that they would be unattainable. It is encouraged to continuously recalibrate and assess what is possible and what is not. Although you may have implemented the scanner, parser, and semantics for a certain feature, IR-gen could be incredibly hard. Instead of continuing to bang your head against a wall trying to implement this feature, you could recalibrate and modify it or focus on something else.

Lastly, it is incredibly important to work as a team. As a team of three individuals, seamless teamwork and coordination were crucial to our success. Constant and effective communication allowed us to work optimally, pushing out new features like clockwork. A dysfunctional or uncoordinated team is a recipe for an unfinished or unpolished product.