SER502 Project: BRO-LANG

Group 21: Rishikesh Anand, Varun Menon & Krutik Pandya April 10, 2023

Contents

1 About **1**

2 Design **2**

2.1 Program Entry point . . . 2
2.2 Declaring variables . . . 2
2.3 Assignment and expression . . . 2
2.4 Ternary operator . . . 3
2.5 If then else block . . . 3
2.6 While block . . . 3
2.7 For block . . . 3
2.7.1 Range iteration . . . 3
2.8 Print . . . 4
2.8.1 Print without new line . . . 4

3 Grammar **4**

3.1 Program entry point and higher order abstractions . . . 4
3.2 Declaration and assignments . . . 4
3.3 Control statements . . . 5
3.4 Data abstractions . . . 5
3.5 Mathematical, logical, and comparison expressions . . . 6
3.6 Expression terminals . . . 6

4 Parser and Interpreter **7**

4.1 Compiler architecture . . . 7
4.2 Execution flow . . . 7

References **8**

1 About

Bro-Lang is a toy programming language designed to be a fun-to-use, informal language with unique features inspired by bro culture and colloquialisms, inspired by BhaiLang Singh and Tripathi (2022). The characteristics of the language resemble casual speech, making it an approachable form of coding language in a conversational style. The language will use familiar

bro-related terms for its keywords, such as "my g" for defining variables/functions, "say" for print, etc.

2 Design

An interpreted strongly typed language, with support for internal blocks using curly braces and commands separated by semi-colons. The comparison operators are also valid mathematically and will produce 1 (for truthy) and 0 (for false). Any non-zero value will be considered valid.

2.1 Program Entry point

The entry point refers to the initial location or function within a program where the control is transferred from the run-time environment, marking the beginning of the program's execution. The entry point serves as a well-defined starting point for developers to structure their code.

```
dawg
// your logic
gg
```

2.2 Declaring variables

The programming language supports variables that begin with a character, followed by any combination of characters and numbers. The language allows for data types such as `doubles`, `booleans`, and `strings` to be assigned to these variables.

```
my_g a = 3
my_g b = 3.4
my_g c = "this is a string"
my_g d = true
```

2.3 Assignment and expression

The programming language provides functionality for assigning values to variables and evaluating expressions.

```
a = b + c
d = e + f
```

Asserts the operands are of the same type (is strongly typed) and supports math, comparison, and logic operators.

2.4 Ternary operator

A standard ternary operator is a concise, conditional expression used in programming languages to simplify if-else statements.

```
<bool condition> ? <block> : <block>
```

2.5 If then else block

A fundamental control structure in programming language that allows the execution of different code segments based on whether a specified condition is true or false.

```
yolo (case) pls {
    // success case logic
}
sus {
    // failing case logic
}
```

2.6 While block

A fundamental control structure in programming language that repeatedly executes a block of code as long as a specified condition remains true.

```
Let him cook(case)  {


}
```

2.7 For block

A fundamental control structure in programming language that iterates over a sequence or range of values, executing a block of code for each value in the sequence.

```
until(startwith; case; everytime) {
    // your logic
}
```

2.7.1 Range iteration

You can also use the for block to iterate over a **left inclusive, right exclusive** range.

```
until(startIdx :: Identifier :: endIdx) {
    // code
}
```

2.8 Print

Print is employed for displaying messages, debugging purposes, or logging program status. The main goal of displaying information in a human-readable format.

```
say a, b, "done";
```

2.8.1 Print without new line

```
say no more a, b, "done";
```

3 Grammar

3.1 Program entry point and higher order abstractions

$$
\begin{aligned}
\langle \text{Program} \rangle &::= \textit{dawg}\ \langle \text{Block} \rangle\ \textit{gg} \\
\langle \text{Block} \rangle &::= \langle \text{Commands} \rangle \mid \epsilon \\
\langle \text{Commands} \rangle &::= \langle \text{Command} \rangle\ \langle \text{Commands} \rangle \mid \langle \text{Command} \rangle \\
\langle \text{Command} \rangle &::= \langle \text{Declaration} \rangle \\
\langle \text{Assignment} \rangle & \\
& \quad \langle \text{If Then Else} \rangle \\
& \quad \langle \text{While} \rangle \\
& \quad \langle \text{For} \rangle \\
& \quad \langle \text{Print} \rangle
\end{aligned}
$$

3.2 Declaration and assignments

$$
\begin{aligned}
\langle \text{Declaration} \rangle &::= \textit{my g}\ \langle \text{Id} \rangle\ \textit{;} \\
& \quad \textit{my g}\ \langle \text{Id} \rangle\ \texttt{=}\ \langle \text{Data} \rangle\ \textit{;} \\
\langle \text{Assignment} \rangle &::= \langle \text{Id} \rangle\ \texttt{=}\ \langle \text{Data} \rangle\ \textit{;}
\end{aligned}
$$

3.3 Control statements

⟨If Then Else⟩ ::= *yolo (* ⟨Condition⟩ *) pls {* ⟨Block⟩ *} sus {* ⟨Block⟩ *}*

⟨While⟩ ::= *let him cook (* ⟨Expr⟩ *) {* ⟨Block⟩ *}*

⟨For⟩ ::= *until (* ⟨Block⟩ *;* ⟨Expr⟩ *;* ⟨Block⟩ *) {* ⟨Block⟩ *}*

until (⟨Expr⟩ *::* ⟨Identifier⟩ *::* ⟨Expr⟩ *) {* ⟨Block⟩ *}*

⟨Print⟩ ::= *say* ⟨Vals⟩ *;*

say no more ⟨Vals⟩ *;*

3.4 Data abstractions

⟨Vals⟩ ::= ⟨Data⟩ , ⟨Vals⟩ | ⟨Data⟩

⟨Data⟩ ::= ⟨String⟩ | ⟨Expr⟩

⟨String⟩ ::= *[Valid unicode string]*

3.5 Mathematical, logical, and comparison expressions

$$
\begin{aligned}
\langle \text{Expr} \rangle &::= \langle \text{Expr} \rangle \texttt{ + } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ - } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ * } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ / } \langle \text{Expr} \rangle \\
&\quad \texttt{(} \langle \text{Expr} \rangle \texttt{)} \\
&\quad \langle \text{Logic Expr} \rangle \\
&\quad \langle \text{Comp Expr} \rangle \\
&\quad \langle \text{Ternary} \rangle \\
&\quad \langle \text{Terminal} \rangle \\
\langle \text{Logic Expr} \rangle &::= \langle \text{Expr} \rangle \texttt{ \& } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ | } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ « } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ » } \langle \text{Expr} \rangle \\
\langle \text{Comp Expr} \rangle &::= \langle \text{Expr} \rangle \texttt{ != } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ == } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ < } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ > } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ <= } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ >= } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ \&\& } \langle \text{Expr} \rangle \\
&\quad \langle \text{Expr} \rangle \texttt{ || } \langle \text{Expr} \rangle \\
\langle \text{Ternary} \rangle &::= \langle \text{Expr} \rangle \texttt{ ? } \langle \text{Expr} \rangle \texttt{ : } \langle \text{Expr} \rangle
\end{aligned}
$$

3.6 Expression terminals

The Id and Number terminals lack concrete grammar, as no separator token exists between them (usually space or a new line for everything else).

$$
\begin{aligned}
\langle \text{Terminal} \rangle &::= \langle \text{Id} \rangle \mid \langle \text{Number} \rangle \\
\langle \text{Id} \rangle &::= \textit{[Valid variable identifier]} \\
\langle \text{Number} \rangle &::= \textit{[Valid integer (can have a decimal)]}
\end{aligned}
$$

4 Parser and Interpreter

Currently, the plan is to use Antlr Parr (2022) to write a compiler cum interpreter that would both parse the language and start executing it. Initially, a **list of tokens** would be generated from the source code via a **lexical analyzer**. Then the list of tokens would be parsed to create a **Parse tree** via a **Parser**. Finally, an **execution environment** would be made available via Java, and the **runtime variables will be stored in two HashMaps**, one for doubles and one for strings.

4.1 Compiler architecture

Since our compiler would both parse the code, and execute it immediately. The compiler and the runtime are one and the same, and internally includes multiple steps as shown in 1.

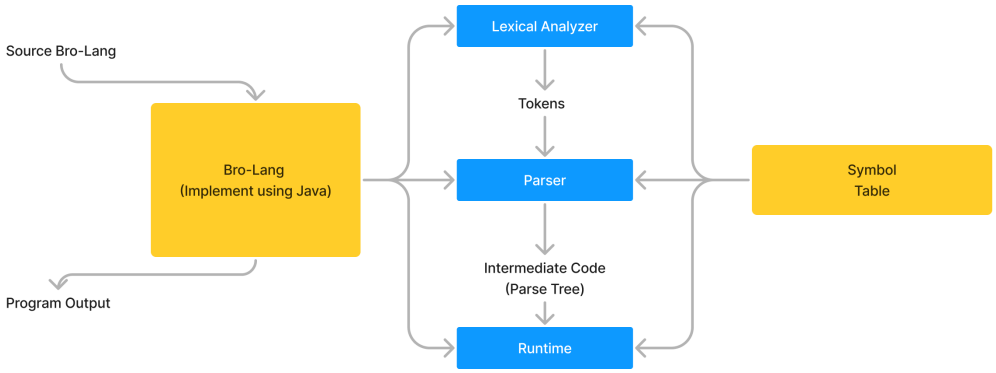

Figure 1: Compiler architecture

4.2 Execution flow

The flow diagram in 2 shows the planned internal flow of both compilation and execution.

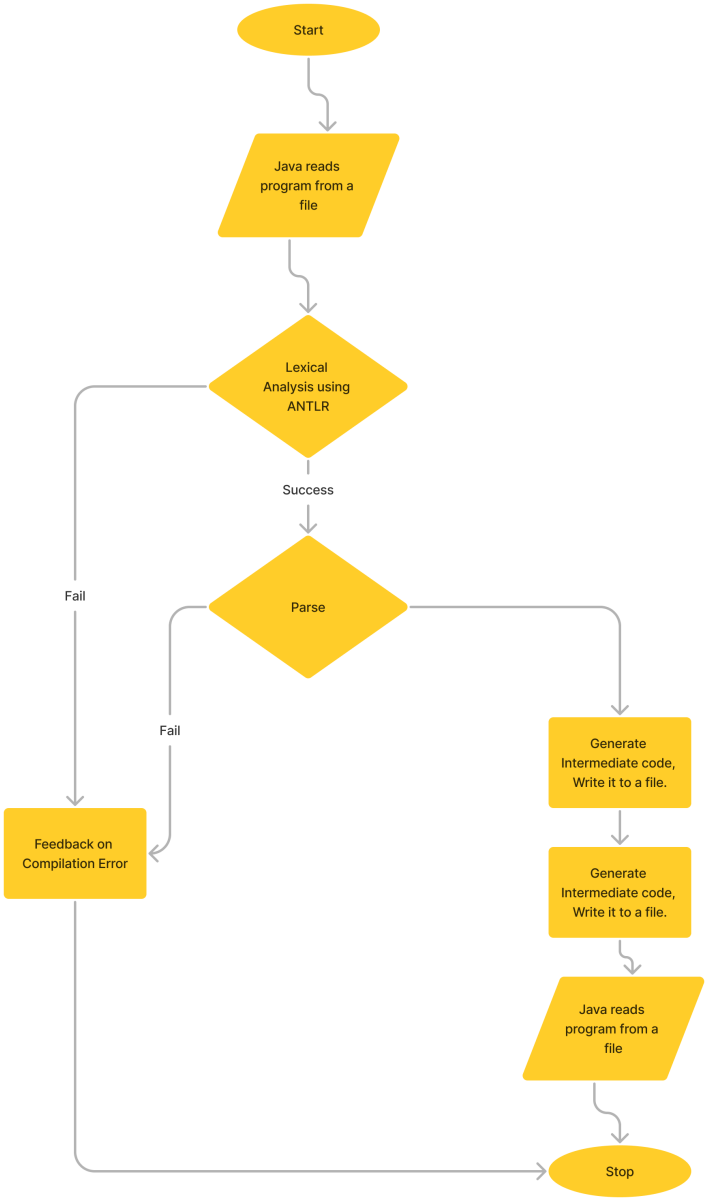

Figure 2: Execution lifecycle

References

Parr, T. (2022), 'Another tool for language recognition'.
URL: *https://www.antlr.org/*

Singh, A. and Tripathi, R. (2022), 'Bhailang - a toy programming language written in typescript'.
URL: *https://bhailang.js.org/*