# Brewless

**SER 502 Project**

Jacob Jose
Michael Kangas
Boan Li
Tahir Pervez
James Thayer

## Overview

We provides a basic framework for a simple programming language that can handle basic data types, arithmetic operations, logical operations, conditional statements, loops, and outputting. With some additional features and constructs, this language could be extended to support more complex applications.

## Language Design

Our Language describes a simple programming language that includes basic programming constructs such as declarations, assignments, conditionals, loops, and printing. The language supports three types: int, String, and boolean, and allows for arithmetic operations and logical operations between boolean expressions

The design of the programming language will be kept similar to Java/C and is supposed to be an imperative language. We will be using lex to tokenize the program. We will be producing our lexical analyzer and parse tree using DCG, and creating our semantic analyzer and runtime environment with Python.  Below are explanations for the different grammar rules.

**Program**: Baseline starting point of a program, can contain a block of code or a single command.

**Block**: Section of code within the program, contains at least one command and potentially more commands and blocks.

**Command**: Basic operation of the language, are declarations, assignments, conditionals, loops, a ternary operation, or a print statement.

**Declaration**: A type of command, is a type and an identifier used to declare a variable

**Type**: A generic term that refers to “int”, “String”, or “bool” variables.

**Assignment**: A command to assign an identifier to some expression.

**Expression**: A generic term for program statements. Contains arithmetic operations as well as identifiers and numbers.

**Multiplication**: The multiplication operation of two expressions

**Division** - Mathematical Division.

**Addition**: Addition operation for two expressions.

**Subtraction**: Subtraction operation for two expressions.

**Identifier**: Is a variable with a value that may be changed. These contain either a single character or “letter”, or potentially multiple letters. Stores data.

**Digit**: Actual digits from 0 to 9. Used to make up integer values.

**Integer**: Any whole number.

**Letter**: Equivalent to characters in other languages. All capital and lowercase letters.

**Boolean**: True or false logic, used in conditional statements.

**String**: Specific identifier not used specifically for variable name but stored as data.

**Conditional**: The actual if , else statements within the language, does not include ternary expressions. Works with Booleans and can contain new blocks of code

**Ternary**: Syntactic sugar for a conditional with a return, where left is true and right if false.

**For_loop**: Two different for loop types including traditional for loop and an in range for loop. Traditional for loop uses a loop_update value.

**While_loop**: Two forms of while loops, a traditional while loop and a do while loop. Both operate on booleans and can contain blocks of code.

**Loop_update**: incrementer for a for loop, can contain an increase or decrease increment

**Increment**: The increase increment increases a value by 1 and returns it.

**Decrement**: The decrease increment decreases a value by 1 and returns it.

**Print**: A classic print statement, prints an identifier

# Grammar

Program -> { Block }
Block -> Command; Block | Command;

Command -> Declaration | Assignment | Conditional | For_loop | While_loop  | Ternary |  Print

Declaration -> Type Identifier | Type Assignment
Type -> int | String | boolean
Assignment -> Identifier = Expression

Expression -> Expression + Term| Expression - Term| Term
Term -> Term * Paren | Term / Paren | Paren
Paren - > (Expression) | Value
Value -> Identifier | Integer | String | Boolean

Integer -> Digit{Digit}
Identifier -> Letter{Letter}
Digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Letter -> a | b | c | … | z | A | B | C | … | Z

Boolean -> true | false | Expression == Expression | not ( Boolean ) | Boolean and Boolean | Boolean or Boolean | Expression < Expression | Expression > Expression

String -> string Identifier = Identifier

Conditional -> if ( Boolean ) { Block } |  if ( Boolean ) { Block } else { Block }
Ternary -> Boolean ? Expression : Expression

For_loop -> for (Declaration; Boolean; Loop_update) { Block } | for (Identifier in Range (Integer,Integer)) { Block }

While_loop -> while (Boolean) { Block } | do { Block } while (Boolean)

Loop_update -> Assignment | Increment | Decrement
Increment -> Identifier++ | ++Identifier
Decrement ->  Identifier-- | --Identifier

Print -> print (Identifier)

# SWI DCG Parsing

```
program(P) --> block(P).

block([C|B]) --> command(C), ";", block(B).
block([C]) --> command(C).

command(C) --> declaration(C) | assignment(C) | conditional(C) | for_loop(C) | while_loop(C) | ternary(C) | print(C).

declaration(declare(T, I)) --> type(T), identifier(I).
declaration(declare(T, I, E)) --> type(T), identifier(I), "=", expression(E).
type(int) --> "int".
type(string) --> "String".
type(boolean) --> "boolean".

assignment(assign(I, E)) --> identifier(I), "=", expression(E).

expression(E) --> arith_expression(E).
expression(E) --> ["("], arith_expression(E), [")"].
expression(E) --> boolean_expression(E).

arith_expression(E) --> simple_expression(E).
arith_expression(add(E1, E2)) --> simple_expression(E1), "+", expression(E2).
arith_expression(sub(E1, E2)) --> simple_expression(E1), "-", expression(E2).
arith_expression(mul(E1, E2)) --> simple_expression(E1), "*", expression(E2).
arith_expression(div(E1, E2)) --> simple_expression(E1), "/", expression(E2).
```

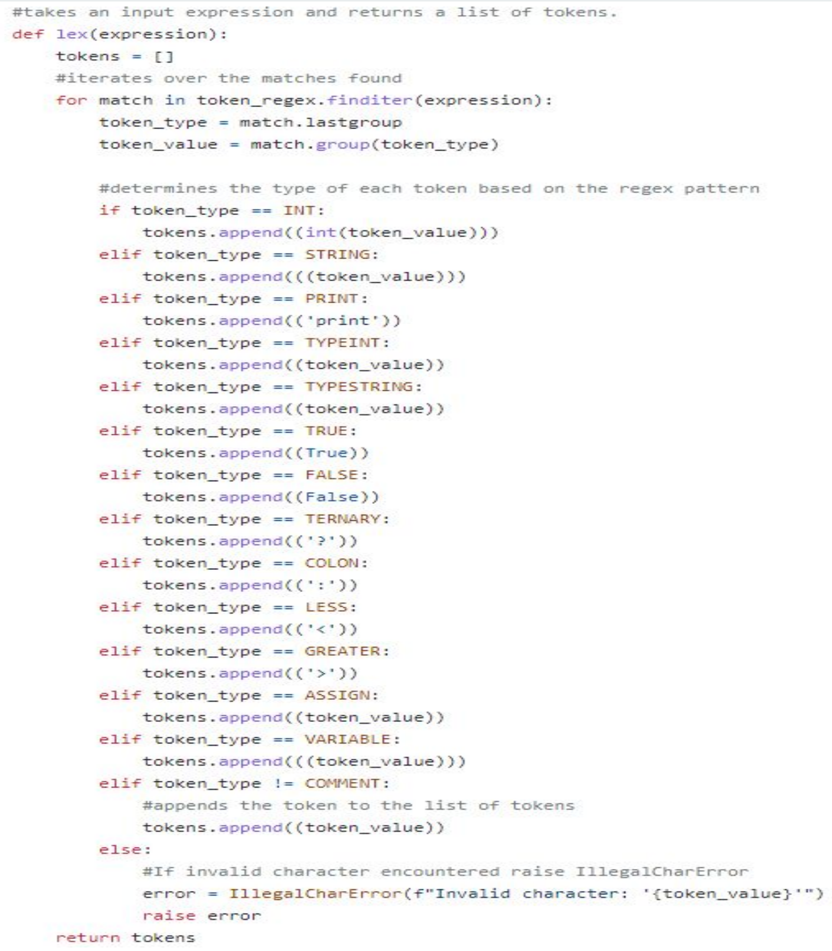
```
simple_expression(E) --> identifier(E) | integer(E) | string(E) | boolean(E).

integer(I) --> digit(D), integer(I2), { atom_concat(D, I2, I) }.
integer(I) --> digit(I).
identifier(I) --> letter(L), identifier(I2), { atom_concat(L, I2, I) }.
identifier(I) --> letter(I).
digit(D) --> [D], { between(0, 9, D) }.
letter(L) --> [L], { char_type(L, alpha) }.

boolean(true) --> "true".
boolean(false) --> "false".

boolean_expression(E) --> simple_boolean(E).
boolean_expression(eq(E1, E2)) --> arith_expression(E1), "==", arith_expression(E2).
boolean_expression(not(B)) --> "not", "(", simple_boolean(B), ")".
boolean_expression(and(B1, B2)) --> simple_boolean(B1), "and", simple_boolean(B2).
boolean_expression(or(B1, B2)) --> simple_boolean(B1), "or", simple_boolean(B2)
```

```
simple_boolean(B) --> boolean(B).

string(string(S)) --> "\"", identifier(S), "\"".

conditional(if(B, P)) --> "if", "(", boolean_expression(B), ")", "{", block(P), "}".
conditional(if_else(B, P1, P2)) --> "if", "(", boolean_expression(B), ")", "{", block(P1),
"}", "else", "{", block(P2), "}".

ternary(ternary(B, E1, E2)) --> boolean_expression(B), "?", expression(E1), ":",
expression(E2).

for_loop(for(A, B, LU, P)) --> "for", "(", assignment(A), ";", boolean_expression(B), ";",
loop_update(LU), ")", "{", block(P), "}".
for_loop(for_in(I, R1, R2, P)) --> "for", "(", identifier(I), "in", "Range", "(", integer(R1),
",", integer(R2), ")", ")", "{", block(P), "}".
```

```
while_loop(while(B, P)) --> "while", "(", boolean_expression(B), ")", "{", block(P), "}".
while_loop(do_while(P, B)) --> "do", "{", block(P), "}", "while", "(", boolean_expression(B), ")".

loop_update(LU) --> assignment(LU) | increment(LU) | decrement(LU).
increment(inc_pre(I)) --> "++", identifier(I).
increment(inc_post(I)) --> identifier(I), "++".
decrement(dec_pre(I)) --> "--", identifier(I).
decrement(dec_post(I)) --> identifier(I), "--".

print(print(I)) --> "print", "(", identifier(I), ")".
```

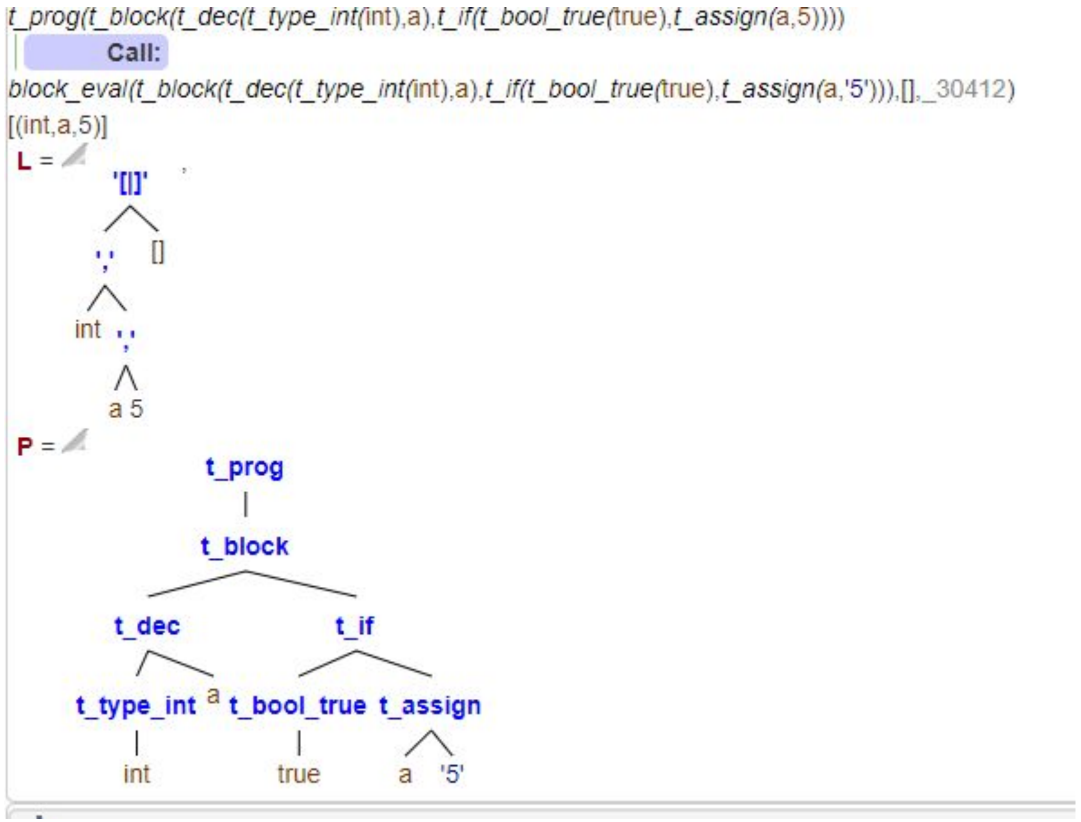
## Compiler

```
class Error:
   def __init__(self, error_name, details):
       self.error_name = error_name
       self.details = details


   def as_string(self):
       result = f'{self.error_name}: {self.details}'
       return result


#Inherits error handler, handles illegal chars
class IllegalCharError(Error):
   def __init__(self, details):
       super().__init__('Illegal Character', details)
```



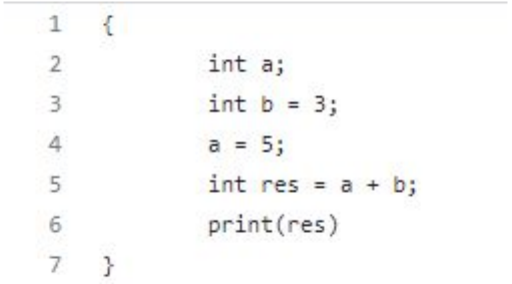
```
#takes an input expression and returns a list of tokens.
def lex(expression):
    tokens = []
    #iterates over the matches found
    for match in token_regex.finditer(expression):
        token_type = match.lastgroup
        token_value = match.group(token_type)

        #determines the type of each token based on the regex pattern
        if token_type == INT:
            tokens.append((int(token_value)))
        elif token_type == STRING:
            tokens.append(((token_value)))
        elif token_type == PRINT:
            tokens.append(('print'))
        elif token_type == TYPEINT:
            tokens.append((token_value))
        elif token_type == TYPESTRING:
            tokens.append((token_value))
        elif token_type == TRUE:
            tokens.append((True))
        elif token_type == FALSE:
            tokens.append((False))
        elif token_type == TERNARY:
            tokens.append(('?'))
        elif token_type == COLON:
            tokens.append((':'))
        elif token_type == LESS:
            tokens.append(('<'))
        elif token_type == GREATER:
            tokens.append(('>'))
        elif token_type == ASSIGN:
            tokens.append((token_value))
        elif token_type == VARIABLE:
            tokens.append(((token_value)))
        elif token_type != COMMENT:
            #appends the token to the list of tokens
            tokens.append((token_value))
        else:
            #If invalid character encountered raise IllegalCharError
            error = IllegalCharError(f"Invalid character: '{token_value}'")
            raise error
    return tokens
```

## Snapshots of the demonstration

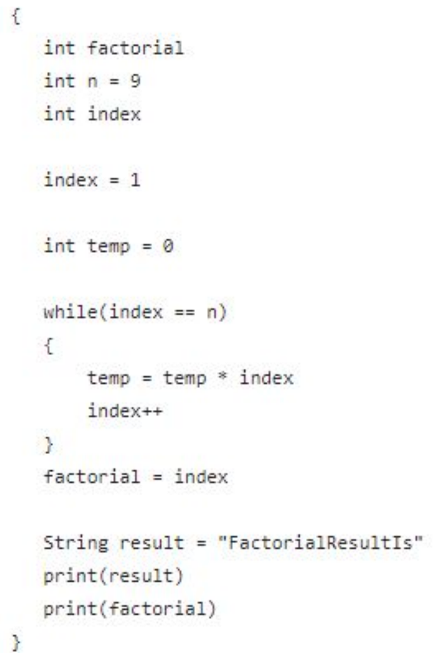


```
t_prog(t_block(t_dec(t_type_int(int),a),t_if(t_bool_true(true),t_assign(a,5))))
Call:
block_eval(t_block(t_dec(t_type_int(int),a),t_if(t_bool_true(true),t_assign(a,'5'))),[],_30412)
[(int,a,5)]
L =
P =
```

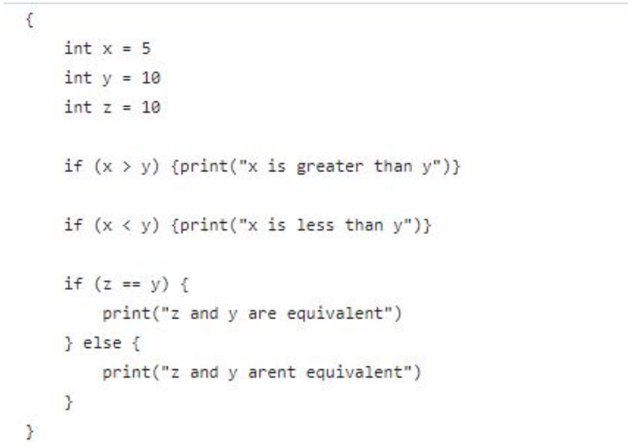
## Test Sample



```
{
        int a;
        int b = 3;
        a = 5;
        int res = a + b;
        print(res)
}
```



```
{
   int factorial
   int n = 9
   int index

   index = 1

   int temp = 0

   while(index == n)
   {
       temp = temp * index
       index++
   }
   factorial = index

   String result = "FactorialResultIs"
   print(result)
   print(factorial)
}
```



```
{
    int x = 5
    int y = 10
    int z = 10

    if (x > y) {print("x is greater than y")}

    if (x < y) {print("x is less than y")}

    if (z == y) {
        print("z and y are equivalent")
    } else {
        print("z and y arent equivalent")
    }
}
```

## Thank you!