Syntax of 'Geko'

Darshi Doshi (21110050)
Saumya Jaiswal (21110186)
Shreya Patel (21110155)
Animesh Tumne (21110227)

General Notes:

- Our language supports static typing only!
- A semicolon `;` is the statement terminator. Every statement must end with a semicolon to allow the parser or compiler to recognize that statement as a whole.
- The language has block scoping with explicit delimiter `{ }`.
- Indentation is not taken into account.
- Only single-line comments are supported (for now), denoted by double octothorpe(hashtag) `##` marks at the beginning of the comment.

Basic data types:

Before defining different data types, there should be some ground rules to build upon. Thus, variable naming:

- Can start only using alphabets, i.e. small or capital letters.
- Cannot start with numbers or special characters.
- Cannot contain whitespace characters, newline characters, or tab characters.
- Cannot be named after reserved characters.
- Is case sensitive.

From these basic rules, we now begin defining our basic data types to be used as a part of our language.

1. Integer

- Range: -1024 to 1023 (bound to change)
- Declaration:

```
num <varname> = 5;
```

2. String

- Allowed characters: A to Z, a to z, 0 to 9, whitespaces, tabs, basic punctuation, and special characters (defined in section 12).
- Representation: Using the `~` (~Hello World~) character instead of double quotes `" "` ("Hello World")
- Declaration:

```
str <varname> = ~Hello World~;
```

- It supports related functions:
  Concatenation (joining multiple strings together)
  Using the '+' operator to join the strings(concatenation).

```
str <string1> = ~Str~;  str <string2> = ~ing~;
## '+' character used for concatenation:
str <string3> = string1 + string2;
show(string3);    ##prints "String"
```

Slicing (Extracting a portion (substring) of a screen)
Use square brackets `[ ]` with start and stop indices separated by colon ":"

```
##syntax
string[<start>:<stop>]
## example:
str name = ~SAUMYA~;
str part = name[3:5];

##indices       0    1    2    3    4    5
##characters    S    A    U    M    Y    A
##the variable "part" stores string ~MY~
```

Note: in the syntax: `str <varName> = string[<start>:<stop>]` the variable will only contain elements of the original string from element position start until element position end-1, similar to Python.

- Single-line comments:

```
## This line is a single-line comment.
##  The  compiler  ignores  lines  with  two  octothorpe  marks
(hashtags)
## at the beginning.
```

3. Boolean

- To designate true, we have "yay" in Geko
- To designate false, we have "nay" in Geko
- Declaration:

```
## for True
flag <varname> = yay;
##for False
flag <varname> = nay;
```

Compound Types:

4. Array

- Arrays are mutable as changing their values is allowed even after initialization.
- Arrays have a fixed size and the same data type for all the elements.
- Declaration:

```
## Example syntax
<type> <array_name> [<size>] = [<element1>, <element2>, ...];
## Array example:
num digits [6] = [1,2,3,4,5,7];
```

- Operations:

```
##length of array 'digits' = 6
num sizeDigits = digits.length();

##accessing elements of the array
##element indexing starts from 0
num firstDigits = digits[0];
##first element of array i.e. 1

##changing element at a specific index
digits[3] = 6;
##updates value at 3 index from 4 to 6 in the variable "digits"
```

5. Lists

- Lists are mutable as changing their values is allowed even after initialization.
- Lists have dynamic sizes and elements can be of the same or different data types.
- Declaration:

```
## Example syntax
list <list_name> = [<element1>, <element2>, …, <elementN>];
## List example:
list names = [~Shreya~,~Darshi~,~Animesh~];
```

- Operations:

```
##Length of the list 'names'
num len = length(names);

##Head (first element) of the list
str first = head(names);

##Tail(last element) of the list
list last = tail(names);

## append (adding an element to the end of the list)
list newList = append(~saumya~, names);
##string ~Saumya~ gets added to the end
show(newList);
##prints out newList, which contains all four names
```

6. Tuple

- Tuples are immutable, meaning their size and elements cannot be changed after creation. Also, due to type-inferencing, we need not explicitly define the data types of the elements.
- Declaration:

```
## Example syntax:
tup <tuple_name> = [<element1>, <element2>, …,<elementN> ];
## Tuple example:
tup memberOne = [~Shreya~, ~CSE~, 20, yay];
```

- Operations:

```
##Length of the list 'names'
num len = length(data);

##Head (first element) of the list:
str first = data[0];
```

```
## concatenate two tuples:
tup tupOne = [1,2,3];
tup tupTwo = [~a~, ~b~, ~c~];
tup tupThree = tupOne + tupTwo;
show(tupThree);
##output:[1,2,3,~a~,~b~, ~c~];
```

Keywords:

7. show

- It prints the output values on the screen, like `print()` function in Python and the `cout` function in C++.
- Print multiple items in the same line by separating them with commas.
- Declaration:

```
num age = 19;
str name = ~DARSHI~;
str city = ~Baroda~;
## to print all onto the screen, we do:
show(~Hii everyone!~);
show(name,~is ~,age,~years old and lives in ~,city, ~.~);

##output : (adds a new line b/w consecutive show())
"Hii everyone!
DARSHI is 19 years old and lives in Baroda."
##is printed on the screen.
```

8. enter

- It takes user input in the form of a string from the user and stores it and uses it further in code.
- Declaration:

```
##syntax
<dataType> <varname> = enter();
##example
str input = enter (~enter your message :~);
show(input);##prints the text input by the user
```

9. yield

- The yield keyword is used to exit a function and optionally provide a value as its result, similar to "return" in Python and C++.
- An example of how it is implemented and used can be found in the *functions* section of the document is given in section 16.

10. let, in

Introduces local variables in a specific scope. The defined variables are immutable.

Declaration:

```
##syntax
let <varname> = value;
## example 1
num x = 10;
show(x); ##output 10
```

```
{
      let x = 5;
      show(x); ##output 5
}
show (x); ##output 10
##syntax
let <variable1> = value1 in
      let <variable2> = value2 in
            let <variable3> = value3 in
                  expression_using_variables;

## example 2
num x = 5;
let result = (let x = 10 in x * 2);
show (result); ##output 20  (not 10)
```

11. NULL

'NULL' keyword represents that the variable does not refer to any valid memory location or does not have a meaningful value.

```
##syntax
str nullVar = NULL;
##Instead of garbage values, now the value of nullVar is
##initialised to NULL
```

12. fix

The fix keyword is used to define constants in Geko. The variables which have the keyword 'fix' before them will remain unchangeable and hence, throw an error on trying to change it. Equivalent to `const` keyword in C++.

```
##syntax:
fix <datatype> <varname> = <value>;
##example:
fix num intVar = -3;
intVar++;
##the above line will throw an error, defined in detail in the last
section of the document.
```

13. Special (\n, \o, \~, \\)Characters:

There are some special characters, such as newline character, tab character, etcetera; defined as follows, that are required to print in a certain way or write code:

```
## newline character:
str newlineVar = ~printed on line one. \n printed on line two.~
show(newlineVar) ##both sentences printed on different lines.

##NULL character:
str nullVar = ~\0~
show(nullVar)
##nothing is seemingly printed on the console;
##but a null character is printed, with the newline character
```

```
##(default end character for show() function)

##tilde character:
str tildeVar = ~This is a tilde character: \~ ,inside a string~
show(tildeVar)
## "This is a tilde character: ~ ,inside a string"
## is printed on the screen.

##backslash character:
str backslashVar = ~This is a backslash character: \\ ,inside a
string~
show(backslashVar)
## "This is a backslash character: \ ,inside a string"
## is printed on the screen.
```

Loops:

14. iter (for) loop:

- Using the keyword 'iter' instead of 'for' in the loop.
- Declaration:

```
##syntax
iter (<iterator>;<condition>;<increment>) {
        ...
        <do something>;
        ...
}
```

15. while loop:

- It iterates over its internal block of code as long as the specified condition (in circular brackets) is true.
- Using the keyword 'while' to specify the start of the loop
- Declaration:

```
##syntax
while (condition) {
    ...
    <do something>
    ...
}
```

16. repeat-while loop

- A repeat-while loop executes a block of code, and then continues to iterate as long as a specified condition remains true. Equivalent to do-while loop in python and C++
- Declaration:

```
repeat {
    <statement>;
} while (condition);
```

- Here the term 'stop' is employed instead of 'break' to terminate the loop prematurely, and 'skip' is utilised instead of 'continue' to bypass a specific iteration within the loop.

Conditionals:

17. given-other-otherwise

- Conditionals allow the execution of different blocks of code based on whether a specified condition evaluates to true or false. Curly braces `{ }` are used to define a block of code. No indentation is considered.
- Declaration:

```
## syntax
given(condition1){
      <given block body>;
}
other(condition2){
      <other block body>;
}
otherwise{
      <otherwise block body>;
}
```

18. functions

- Functions are reusable and self-contained block of code that performs a specific task. They can be called multiple times in a code.
- "define" keyword indicates the start of a function definition. It has its own scope enclosed by curly braces, `{ }`.
- Parameters are input values to functions. They need to be type casted when declaring the function. Example code given below.
- For naming the functions, the same rules as that of the nomenclature of the variables mentioned above are allowed.
- The function can return a value of any data type.
- Declaration:

```
##syntax
define function_name(<type> parameter1, <type> parameter2, ...){
      ...
      <do something>
      ...

      yield <result>;

      ## value to be returned by the function is returned using
      ## "yield" keyword.
}

##example function to add two integers
define giveBackSum(num a, num b) {
      yield a + b;
}
##calling the function
num first = 20;
num second = 30;
show(giveBackSum(first,second)); ##output:50
```

19. Scope of Objects and Closure

- The scope of an object written inside curly brackets is within the brackets only.
- The scope of an object written outside is everywhere in the code, i.e. global scope.

```
define fOne() {
      num x = 4;
      define fTwo() {
            num y = 7;
            show(y,x); ##output:7 4
      }
      show(y,x); ##output: scopeError
}
```

20.Numeric Arithmetic operations

Assuming a and b are two integral variables, the following numeric (integer) Binary arithmetic operations are possible in our language with decreasing precedence:

Operation	Output
`a ** b`	a raised to power of b
`a * b`	Product of a and b
`a / b`	a divided by b (integer division)
`a + b`	Sum of a and b
`a - b`	Difference of a and b
`a % b`	Remainder of a / b (modulo)

21. Unary Operations

Operators operating on a single operand. Assume a is an integral variable.

Operation	Output
`+a`	doesn't change the sign of operand
`-a`	changes the sign of operand
`!a`	negates the value of a boolean expression
`++a`	increases value by 1 and stores a+1
`--a`	decreases value by 1 and stores a-1
`a++`	increases value by 1 and stores a
`a--`	decreases value by 1 and stores a

22.Comparison Operators (returns boolean value)

Operators	Meaning
`a < b`	True if a strictly less than b
`a > b`	True if a strictly greater than b

`a <= b`	True if a less than or equal to b
`a >= b`	True if a greater than or equal to b
`a == b`	True if a if equal to b
`a != b`	True if a if not equal to b

23. Logical Operators (operates on boolean operands)

Operation	Syntax	Output
Logical AND	`a && b`	true if both true else false
Logical OR	`a || b`	true if atleast one is true
Bitwise AND	`a & b`	Bitwise Calculation on Integers
Bitwise OR	`a | b`	Bitwise Calculation on Integers
NOT	`!a`	negates the value
XOR	`a ^ b`	true if both operands are different

Exception Handling:

Based on C++, we have try, throw, and catch for exception handling where it will print something instead of crashing the program.

- try → test
- throw → pop
- catch → arrest

```
test {
      ## Part of Code where there is a chance of error
      pop <Statement>;
} arrest (<error condition>) {
      ## Handling the exception
      show(~Caught an Exception!~);
}
```

Error Messages:

When there are bugs in the code, the program may encounter runtime errors that cause the program to stop running. These errors can be accompanied by an error message that provides information about the cause and location of the error.

The following is the error message a user can get:

```
Oops! :3 Error on line <x>! Go correct it! (error statement)
```

The user can have three types of errors with this language:

1) **Syntax Error:** This means the rules of the programming language are violated in the program. This kind of error includes missing semicolons, brackets, etc. The following error message is shown when there is a syntax error:

```
Oops! :3 Error on line <1>! Go correct it!
show(~Hello world!);
SyntaxError: unterminated string literal
```

OR

```
Oops! :3 Error on line <1>! Go correct it!
show(~Hello world!~);
SyntaxError: expected ';' at the end of the statement
```

2) **Semantic Error:** This means the code is correct in terms of syntax but does not produce the expected output due to a mistake in the meaning of the code. They do not produce error messages that indicate the location of the error.

3) **Runtime Error:** This means that the code stops running and displays an error message related to the problem like division by zero, list index out of range, no such file or directory, etc. The following error message is shown when there is a syntax error:

```
Oops! :3
Error on line <1>! Go correct it!
num result = num1 + num2;
NameError: name 'num1' is not defined
```

OR

```
##For the code below:
str x = ~Hello, ~;
num y = 42;
str z = x + y;  ## This line will cause a compilation error

show(result);
```

The error message will be:

```
Oops! :3
Error on line <3>! Go correct it!
Z = x+y;
error: invalid operands of types 'str' and 'num' to
binary'operator+'
```

OR

```
##when incrementing fix num x = 2, the following error
##message will show up
Oops! :3
Error on line <2>! Go correct it!
x++;
Error: increment of read-only variable x
```

OR

```
IndexError: list index out of range
```

OR

```
ZeroDivisionError: division by zero
```

For multiple errors in the code, error messages for all errors are shown:

```
show(~Hello~);
num a = 10;
show (a);

Oops! :3
Error on line <1>! Go correct it!
SyntaxError: expected ';' at the end of the statement
Error on line <3>! Go correct it!
SyntaxError: expected ';' at the end of the statement
```