

# Indian Institute of Technology Gandhinagar



---

*Kulant: The Kul (Cool) Language Today*

---

## **CS 327 (Compilers) - Assignment 1**

### ***La Retro's Members***

*Aaryan Darad (21110001)*

*Abhay Upparwal (21110004)*

*Harshita Ramchandani (20110074)*

*Somesh Pratap Singh (19110206)*

### ***Under the guidance of***

Prof. *Abhishek Bichhawat*

# CONTENTS

---

- 1. Kulant Syntax 3**
- 2. Data Types 4**
- 3. Variables 5**
- 4. Compound Data Types 7**
- 5. Operators 10**
- 6. Branches and Loops 14**
- 7. User Built Functions 16**
- 8. Exception Handling 19**

# Kulant Syntax

---

**Kulant** is a *statically typed, compiled programming language*. This documentation contains the syntax of the language.

All the statements in *Kulant* would end with a *semicolon (;)* as it is also necessary for defining the start of a new line. Variables are defined using data type name in the start followed by name of variable.

```
int var = 8;
display(var);
word sample = "sample";
display(sample);
```

In this context, an integer variable 'var' is initialized with the value 8. The ensuing display statement provides insight into the runtime state, offering a visual representation of the variable's content. Similarly, a word variable named 'sample' encapsulates the string "sample," showcasing Kulant's versatility in managing diverse data types.

There is no separation between lines except for the semicolon, i.e. no indentations/tabs etc. You may add indentation to make the code readable for yourself. Use of appropriate spaces is recommended, as it will help with the easy compilation of the code.

For commenting in between the code, one can use a dollar (\$) symbol; all the lines after the dollar symbol will be omitted by the programming language.

# Data Types

---

## Singular Data Types

- i) **int** (*integer 4 bytes*) : all integers which fit in 4 bytes (positive and negative), eg. 15.
- ii) **bigint** (*integer 8 bytes*) : all integers which fit in 8 bytes, eg. 7205759403927935.
- iii) **char** (*1 byte*) : all characters enclosed in single quotes ('), eg. 'a', 'b', 'c', 'd'.
- iv) **double** (*decimal 8 bytes; 4 bytes before decimal place, 4 bytes after decimal*) : decimal/floating point numbers with the given criterion, eg. 8.254
- v) **bool** (*1 byte*) : Binary Values; True or False

## Compound Data Types

- i) **word** (*array of chars aka string*) : ends with '\0', eg. "king".
- ii) **list** (*variable size array*) : can contain only 1 data type at a time, eg. [2.34, 8.91, 3.01].
- iii) **set** (*tuples*) : can contain more than 1 data type at a time, eg. (1, "cat", 'k', 2.45).

# Variables

---

Since *Kulant* variables are statically typed, they can only have values of a specific type given to them.

Declaration: Variables are declared by writing the data type followed by the name of the variable. A variable can be defined with a pre-assigned value or it can be assigned the value after declaration

```
int x = 10;
```

```
dotie y;
```

We can change the value of the variable once defined by directly equating its value to the new value.

```
x = x + 1;
```

```
y = 2.54;
```

The variable can only have values of that type after it is declared with a value. This includes Booleans, Words, Lists, and Numbers.

```
int k = 2929;
```

```
k = "abhay";
```

The code shown above would generate an error pertaining to improper variable assignment.

## Naming rules for variables:

1. Function name should start with a verb.
2. Variable name can begin only with an alphabet. Digits can be used only after alphabet. Only special characters allowed for naming variables is underscore('\_').
3. Keywords cannot be used as variable names.
4. Rules for immutable variables applies as stated below.
5. Pointer variables should have an asterisk '\*' prepended.

**Immutable Variables:**

In our program, the user might want to declare values that cannot be changed during the course of the program. For that we would use the keyword `constant` before the variable type during variable declaration:

Example:

```
constant int i=5;
```

However, to declare an immutable variable the initialization of the variable value has to take place at the declaration itself.

# Compound Data Types

---

## i) Words

Words are defined as a list of characters which are exclusively expressed within double quotes (") and the array ends with `'\\0'`. Strings support indexing, concatenation and slicing operations.

### Concatenation

```
word str = "Abhay";  
Word str1 = "Aaryan";  
str = str+str1  
display(str)
```

The expected output is out `AbhayAaryan`

### Indexing

This is used to access the character at a particular index in the string. *Kulant* follows 0-indexing (the first index value is 0).

```
word x = "abhaykumar";  
display(x[6]);
```

This prints out `'u'` - the element at index 6.

### Slicing

The beginning index and the ending index are the two indices used in the slice process. The string's sliced segment is contained in the index range [start, finish).

```
word x = "abhaykumar";  
int i = 1;  
display(x[i,i+5]);
```

This displays the characters in the string from index 1 to index 5 - `"bhayk"`

## ii) Lists

Lists are declared with the datatype of the type of values inside the list followed by the name of list and square brackets `[]` equated with the values inside the list enclosed in square brackets. We can also define an empty list by equating it with anything. If required, one can always increase the size of the list by using the `push_back` or `push_front`.

```
int nums[] = [7,22,33,13,483];  
word names[] = ["abhay","somesh" , "harshita"];  
bool truths[] = [True, False, False];
```

Lists support operations - `front`, `back`, `empty`, `push_front`, `push_back`, `indexing`, `slicing`.

### front and back, empty

`front` gives the list's first element. `back` gives the last element. The function `empty` verifies if a list is empty. When it is, it returns `True`; otherwise, it returns `False`.

```
display(x.front); display(x.back); display(x.empty);
```

This displays `7, 483, False`

### push\_front, push\_back

To add an element to the front and rear of the list, use the `push_front` or `push_back` method.

```
x.push_front(60);  
x.push_back(87)  
display(x);
```

This displays `[60, 7, 22, 33, 13, 483, 87]`



## Indexing

This allows you to obtain items from a specific list index. Kulant uses 0-indexing, in which 0 is the initial index value and increases by 1.

## Slicing

The slice operation takes two indices- the start and the end index. Sliced portion of the string is in the index range `[start,end)`.

```
display(x[1,4]);
```

This prints out the characters in the list of elements from index 1 to index 3 - `[7,22,33]`.

## iii) Tuples

Tuples are declared directly by the keyword `tuple` followed by the name of tuple, equating it with the values of the tuple enclosed in brackets `()`.

```
tuple x = (1.2, 5, "abc", True);
```

Once defined, tuples can't be changed. Similar to lists, tuples also have indexing, slicing, concatenation operations.

# Operators

---

*Kulant* has several built-in operators.

## Arithmetic Operators

Arithmetic operators in *Kulant*:

The following are the binary operators in *kulant*:

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Divide
- `**`: Power (Exponent)
- `%`: Modulo

Example:

```
5 + 10 * 3 + 2 ** 3 == 43
```

The operator precedence in our language is defined by PEMDAS (left to right) which stands for Parenthesis, Exponential, Multiplication, Division, Addition, Subtraction

The `+` operator is extended to function with strings as well, specifically for concatenation purposes.

## Unary Operators

The following are the unary operators in *kulant*:

- `-`: Unary Minus
- `!`: Not
- `&`: `addressof`
- `sizeof()`: `sizeof operator`

**Examples:**

**Unary Minus**

```
int x = 2  
int y = -2  
display(y)
```

This would give a value of negative 2 to y and display it on the screen.

**Not**

This reverses the logical state of an expression.

A = True

B = False

C = A or B

D = !C

display(D)

This would print False.

**& [The addressof operator]**

This returns the address of a variable where it is stored in memory. The address returned is in the hexadecimal form. This is stored in pointer type of variables which would be represented by use of a special symbol '\*' in front of them.

```
int a = 10;  
int *ptr;  
ptr = &a;
```

```
display(ptr)
```

This would print the address 0x7ffe2910aa6c

**sizeof()**

This prints the size of the variable in bytes.

```
int a = 10;  
display(sizeof(a));
```

This would print 4, i.e. an integer is 4 bytes long. .

## Comparison Operators

Comparison operators in *Kulant*:

- `==`: Equal to
- `!=`: Not equal to
- `<`: Less than
- `>`: Greater than
- `<=`: Less than or equal to
- `>=`: Greater than or equal to

Example:

```
25 > 5
```

returns `True`

These give a Boolean value.

## Logical Operators

*Kulant* has the following logical operators:

```
and(&&), not(!), or(||)
```

Here is an example:

```
(10 > 5) && (2 < 1) evaluates to False.
```

## Other In-built Operators

An operation to print values to the screen looks like this in our language:

1. If you want to print the exact phrase, statement, command, put the exact expression inside double quotes.

```
a.display("contents to be shown") ;
```

2. If you want to print the result of an arithmetic/logical expression, put the expression inside `display()`.

```
a.display(10+20) ;
```

3. If you want to print some directional phrases and also the result of an operation, this can be done in the following way.

```
a=10;
```

```
b=20;
```

```
display("the sum of a and b is: ", a+b);
```

An operation to take input from users looks like this in *Kulant*:

```
int a;
```

```
input(a);
```

# Branches and loops

---

## 1. Branching

### a. if

One of the branching and conditional statements is 'if'. Similar to 'if' in most of the programming languages, this specifies the condition on which you have to branch or not. An if statement in *Kulant* looks like

```
if (condition){  
    $ do something  
}
```

The program flow would go inside the branch only when the condition is true.

### b. otif (otherwise-if)

If the condition in 'if' is not true the flow goes inside the otherwise-if (otif) branch. The syntax for otif in *Kulant* looks like this:

```
otif (condition){  
    $do something  
}
```

### c. otw (otherwise)

When condition in none of the if and otif are true the flow finally goes to otherwise, which is similar to else in other programming languages. The syntax for otw looks like this:

```
otw {  
    $do something  
}
```

Eg:

```
if( a < 10){  
    x = x + 1;  
}  
otif ( 10 < a < 15){  
    x = x + 2;  
}  
otw {  
    x = x + 3;  
}
```

## 2. Loops

### a. for

The syntax of 'for' loop in *Kulant* looks like this

```
for (define/specify iterator; increment/decrement
    iterator; ending criteria){
    $do something
}
for ( i = 1; i = i+1; i <= 10){
    display("inside for")
}
```

### b. while

The syntax of 'while' loop in *Kulant* looks like this

```
while (specify stopping criteria){
    $do something
}
while (i != 10){
    print("inside while");
}
```

## 3. Getting out of the loop when a criteria is met, or don't want to execute the condition in the loop for every time the iterator increments/decrements.

### a. get\_out

This could be used to get out of the loop when a specific condition is met. Example of the same is illustrated below:

```
for ( i = 1; i = i+1; i <= 10){
    display("inside for");
    if (i == 7){
        get_out;
    }
}
```

This would take the control flow outside the for loop.

### b. go\_on

This could be used when you want to skip the execution of condition inside the loop and hop over to the next loop iteration. A use case of this functionality is illustrated below:

```
for ( i = 1; i = i+1; i <= 10){
    if (i == 7){
        go_on;
    }
    display("inside for and we'll not print 7", i);
}
```

# Functions

---

'Start' function serves as the entry point of the program. The program execution starts from the 'start' function.

```
int start(){  
    $ program code  
  
    return 0;  
  
}
```

Let typedef be data type then *Kulant* will follow the syntax-

```
typedef fun(typedef x, typedef y) {  
  
    x = x + y;  
  
    return x;  
  
}
```

Example:

Let int be datatype -

```
int fun(int x, int y) {  
  
    x = x + y;  
  
    return x;  
  
}
```

We need to define the return datatype of the function behind the function name.

'Void' keyword is used to indicate that a function does not return a value.

```
Void fun(){  
  
    display("This function doesn't return anything");  
  
}
```



## Closures

In *Kulant*, closures refer to the combination of a function and the lexical environment within which that function was declared. The lexical environment includes all the variables that were in scope at the time of the closure's creation. Closures in *Kulant* enable a function to access and manipulate variables from its outer (enclosing) scope even after that scope has concluded.

Similar to other languages, in *Kulant*, when a function is defined within another function, it retains access to the variables of the outer function. This is achieved by creating a closure, bundling the function code along with the environment in which it was created.

```
typedef outerFunction(x) {  
    function innerFunction(y) {  
        return x + y;  
    }  
    return innerFunction;  
}  
  
typedef closureInstance = outerFunction(10);  
typedef result = closureInstance(5);  
display(result); $ Output: 15
```

In this *Kulant* example, `outerFunction` takes a parameter `x` and defines `innerFunction` inside it. The `innerFunction` has access to the `x` parameter even after `outerFunction` has completed execution. The returned `closureInstance` is now a function that can be invoked with its own arguments (`y`), and it remembers the value of `x` from the outer scope.

### Characteristics of Closures in Kulant

**Access to Outer Variables:** Closures in *Kulant* allow functions to access variables from their containing scope even when the outer function has finished executing.

**Preservation of State:** Closures can be used in *Kulant* to preserve the state of a function. The inner function retains access to the variables from the outer function, allowing it to remember previous states.

Encapsulation: Closures provide a form of encapsulation in Kulant by limiting the scope of certain variables to the functions that need them, avoiding global scope pollution.

Dynamic Behavior: Closures enable dynamic behavior in Kulant as the inner function can be executed with different arguments while still maintaining access to the outer variables.

### Scope and Lifetime of Variables in Kulant

Variables defined in the outer function, which are not parameters or returned values, are enclosed within the closure and have no meaning outside the function. They are accessible only within the closure, ensuring a level of encapsulation.

```
typedef outerFunction() {  
    x = 10;  
  
    function innerFunction(y) {  
        return x + y;  
    }  
  
    return innerFunction;  
}  
  
typedef closureInstance = outerFunction();  
result = closureInstance(5);  
$ print(x);  $ Error: x is not defined in this scope
```

In this Kulant example, the variable `x` is not accessible outside the closure. It is encapsulated within the closure created by `outerFunction`.

Closures in Kulant serve as a powerful feature, providing a way to implement functional programming concepts and enhancing code readability, maintainability, and expressiveness. They contribute to the flexibility and richness of the Kulant programming language.

# Error and Exception Handling

---

While writing the code, multiple errors might occur as a result of user carelessness and logic incompatibility, these errors can occur at runtime or compile time. The exceptions are raised during execution of a program.

The runtime errors would raise an exception known as:

**runtime\_error** : It would occur in situations such as: invalid input, infinite loops, division by zero, when trying to open a non-existing file to read and accessing memory that has not been allocated.

For *Kulant*, the exception handling constructs use three blocks to handle errors and exceptions gracefully.

```
try {  
    Code with possible error  
} catch (error) {  
    Error to throw / display to show when an error is  
    encountered.  
    Ex: throw Error("Your_error");  
} finally {  
    Code to be executed even if the exception was encountered.  
}
```

Example:

```
try {  
    declare numerator = 10;  
    declare denominator = 0;  
    declare result = numerator / denominator;  
} catch (error) {  
    throw new Error("Division by zero is not allowed");  
} finally {  
    display("This code will be executed regardless of whether  
    an exception occurred or not.");  
}
```

Compile time errors are encountered during the compilation of a program.

**syntax\_error** : This error would come into play whenever the user written code does not abide by the syntax rules of the programming language.

Examples of such cases would be, misspelled function names, mismatched parenthesis, missing semicolons and incorrect variable names.

**linker\_error** : This error would occur when combining separately compiled or assembled program files into a single file. The situation in which this exception could occur would be multiple definitions and mismatched data type of a function or variable in two separate files.

**logical\_error** : This error does not crash the program, and as a result it is hard to detect. This error is caused by misunderstanding the requirements, calculation errors, giving faulty conditions, incorrect algorithms and faulty data.