INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING & COMPUTER SYSTEMS (IJSECS)
ISSN: 2289-8522 e-ISSN: 2180-0650
VOL. 8, ISSUE 2, 51 – 66
DOI: https://doi.org/10.15282/ijsecs.8.2.2022.6.0103

ORIGINAL ARTICLE

Towards the Design and Implementation of a Programming Language (Beex)

F. O. Chete[1*] and S. O. Ikeh[1]

[1]Department of Computer Science, University of Benin, Benin City

ABSTRACT – Software Engineers, Computer Scientists, and Software Experts alike are faced to decide which programming language is best suited for a certain purpose as the use of programming languages grows. When we consider the various types of programming languages available today, such as Domain Specific Languages (DSL), General Purpose Languages (GPL), Functional Programming Languages (FPL), Imperative Programming Languages (IPL), amongst others, this becomes complicated. In this study, we introduce BeeX, an interpreted language, with the aim of showing the process and principles involved in language design and consider various choices faced by language designers of various programming languages. BeeX was created with simplicity in mind, thus the study focused on architectural design options. We look at the implementation standpoint and try to figure out what the basic building parts of most programming languages are, such as lexical analysis, syntax analysis, and evaluation phase. To achieve this, we created an interactive command interface that evaluated various BeeX language constructs (conditional logic statements, arithmetic expressions, loop constructs etc.) which allowed students to easily experiment with the proposed language. The results of the tests showed that students and programmers alike can use the BeeX programming language to create a variety of code structures that are simple to use.

ARTICLE HISTORY
Received: 9 February 2022
Revised: 1 August 2022
Accepted: 22 August 2022

KEYWORDS
Software Engineers
Computer Scientists
Software Experts
BeeX
Programming Language

INTRODUCTION

Language translation is an essential process in programming; given that computers only execute instructions in low-level format i.e. binary sequence. Translators provide a bridge between the target language (sequence of binary digits) and source language (high-level languages e.g. C, C++, Python, etc.)

Students are faced with the need to learn multiple programming languages during the course of study in computer science and other engineering departments, this need can be justified as each programming language provides peculiar concepts or approaches to learning. But these languages usually have steep learning curves; therefore students are not able to gain enough benefits in a semester's course. In addition, learning established programming languages with many programming features can be frustrating to students trying to carry out the academic task. In this study, BeeX, an interpreted language is introduced, with minimal programming constructs, which allows students to learn these concepts quickly and easily, just enough to accomplish regular tasks and experiment with ideas from their curriculum.

BeeX is a dynamic programming language meant to be used by students to carry out programming tasks while also allowing curious students to create different versions of BeeX.

This study aims to implement a language translator, an interpreter, which directly translates and executes instructions from a source language. The interpreter implemented in this study targets the language source language BeeX - a high-level programming language designed during the course of this study. BeeX is designed with the aim of showing the process and principles involved in language design while considering various choices posed by language designers.

Based on Demaille [1] as cited in [2] reported that compiler construction is a challenging process that requires material from virtually all computer science courses in the core curriculum. Wu et al [2] posited that the idea of compilers is usually furthered and explored in detail later on in an upper-level course such as Compiler Construction, however, Xing [3] argued that the idea of interpreters rarely gets the same "treatment". Referring to [1] further submitted that there is no course targeting on interpreter constructions in most undergraduate computer science curricula at universities and colleges. In addition, [2] posited that interpreters are complex programs, and writing them successfully is hard work. To reduce this complexity and achieve the objectives stated in this study, we use the approach of exploring a concept called tokenization where the sequence of strings is grouped into units which are referred to as "lexeme". This process deals with transforming the string of characters in the source program into a stream of tokens, where the token is a string with a designated and identified meaning [4]. This phase is known as the Lexical Analysis Phase, and the output from this phase is then used in subsequent phases further down the line as the interpreter runs. Following a sequence of processes,

*CORRESPONDING AUTHOR | F. O. Chete | odichet@yahoo.com

the interpreter is able to successfully produce results from the given source inputs. Several steps are involved in the interpretation, which includes: error handling, intermediate representation, and evaluation.

The aim of this study is to design and implement a working language interpreter (BeeX), with the following objectives: (i) design a grammar rule for the proposed language.(ii) implement a tokenizer, parser, and evaluator following the given grammar rules.(iii) implement an interactive REPL (Read Evaluate Print Loop) tool to help students quickly try out language features (iv) demonstrate the impact of language design, by exploring each component in the interpreter

RELATED WORK

Interpreters translate and execute each line of source code one by one. Each program, when run, is firstly syntax-checked. If a syntax error is found, for example, a missing bracket, it is reported. Following the syntax check, each line of source code is converted (by the interpreter) into its machine code equivalent and executed. If a run-time error occurs, the program crashes.

This line-by-line approach allows the developer to test the program and quickly identify and remedy each error as it occurs, without having to go through the whole process of translating the entire program every time. Interpreters are thus very useful at the development stage of a program.

Source code that is interpreted runs more slowly than compiled code because each time the program is run, it must be translated all over again. In addition, the user must have the interpreter installed to be able to run the program. However, interpreted code can be advantageous if the programmer does not know which platform will be used to run the code. For example, JavaScript code is interpreted; the translation is handled by the browser on the user's computer.

Most interpreted languages now use byte code as an intermediate stage to speed up the translation process. The source code is compiled to produce byte code, which is interpreted (i.e. translated and executed) by a virtual machine. Python, an interpreted high-level general-purpose programming language, was mainly designed with an emphasis on code readability and its notable use of significant indentation [5]. Python was well received by the programming community due to its readability and ease of development.

Ruby, initially developed and designed by Yukihiro Matsumoto during the mid-1990s is a dynamic, reflective, general-purpose, object-oriented, interpreted, programming language that supports multiple programming paradigms, including imperative, functional, and reflective programming [6]**.** As an interpreted language, it provides an interactive interpreter for both the execution of scripts and development and testing [6]**.**

PHP, conceived in 1994 and originally the work of Rasmus Lerdorf, is a server-side scripting language designed specifically for the web [7]. The PHP code is interpreted at the web server and generates HTML [7]. It is currently one of the most popular programming languages widely used in both the open source community and in industry to build large web-focused applications and application frameworks [8]. The Zend Engine is the basic scripting engine that drives PHP and brings to PHP performance, reliability, and an easy-to-use scripting interface [9].
Several other researchers have also developed interpreters for different purposes over the last few years as listed below:

- [2] designed and implemented an interpreter for the SimpleC programming language using software engineering concepts. The work demonstrated that some of the standard software engineering concepts (such as object-oriented design, design patterns, UML diagrams, etc.), can provide a useful track of the evolution of an interpreter, as well as enhance confidence in its correctness.
- [10] designed and implemented an easy-to-use and efficient industrial robot program interpreter. The process of interpretation included lexical analysis, syntactic analysis, semantic analysis, and instruction interpretation modules. The experimental results showed that the designed interpreter had high efficiency and stability in interpreting the robot program and met the operational requirements of industrial robots.
- [11] designed and implemented a programming environment including an editor, a debugger, and an interpreter engine for Lograph, a general-purpose visual logic programming language. The engine takes full advantage of an efficient implementation of Prolog and operates on a Prolog translation of Lograph programs and queries.
- [12] designed an expert interpretation system based on matching measurement and conceptual hierarchy methods. The system increased the automation level of aerial or remotely sensed image-based measuring systems. Using the system saved time and cost and reduced human errors in spatial data extraction from images.
- [13] designed and implemented a C-like language interpreter using C++, based on the idea of modularity. The whole system was divided into three modules, namely, lexical analyzer, syntactic analyzer and evaluation of expression. Token was used during the design. The design put great emphasis towards code reusability and extensibility, which made it easy to embed the code into other C++ programs. The final testing result showed that the implemented interpreter realized almost all the interpretation functions of C-like language, including the interpretation of interface function call.

Thus, we concluded that the development of these languages and their interpreters has been pivotal to several fields in software engineering.

METHODOLOGY

This study explores a concept called tokenization where sequence of strings is grouped into units which are referred to as "lexeme". This process deals with transforming the string of characters in the source program to a stream of tokens, where the token is a string with a designated and identified meaning [4]. The output from this phase is then used in subsequent phases further down the line as the interpreter runs.

The interpreter implemented in this study follows the grammar rule developed for BeeX. The procedure involved in the implementation is given as follows:

(i) Lexings (Tokenization): This process involves breaking up sequence of characters, and organizing these sequences as units called tokens [14].

(ii) Parsing: In this phase tokens from the previous phase are organized to give semantic meaning which conforms to the grammar of the language. This process is also known as syntax analysis.

(iii) Abstract Syntax Representation: This is an intermediate code representation which is produced during the parse phase carried out by the interpreter. This form of code is also known as abstract syntax trees or simply syntax trees.

(iv) Object Type System: This refers to the type of system used during the evaluation phase; data types are constraints upon data objects which allow specific operations to be performed on various data.

(v) Evaluation: This involves the last translation phase, where the source program is executed statement by statement and produces the results obtained from this execution.

SYSTEMS ANALYSIS AND DESIGN

The proposed programming language

A translator, can be conceptualized as consisting of two major parts; a frontend and a backend. Though both compilers and interpreters can use the same front end, they will, however, have a different backend implementation. Figure 1 depicts the conceptual design of the BeeX interpreter.

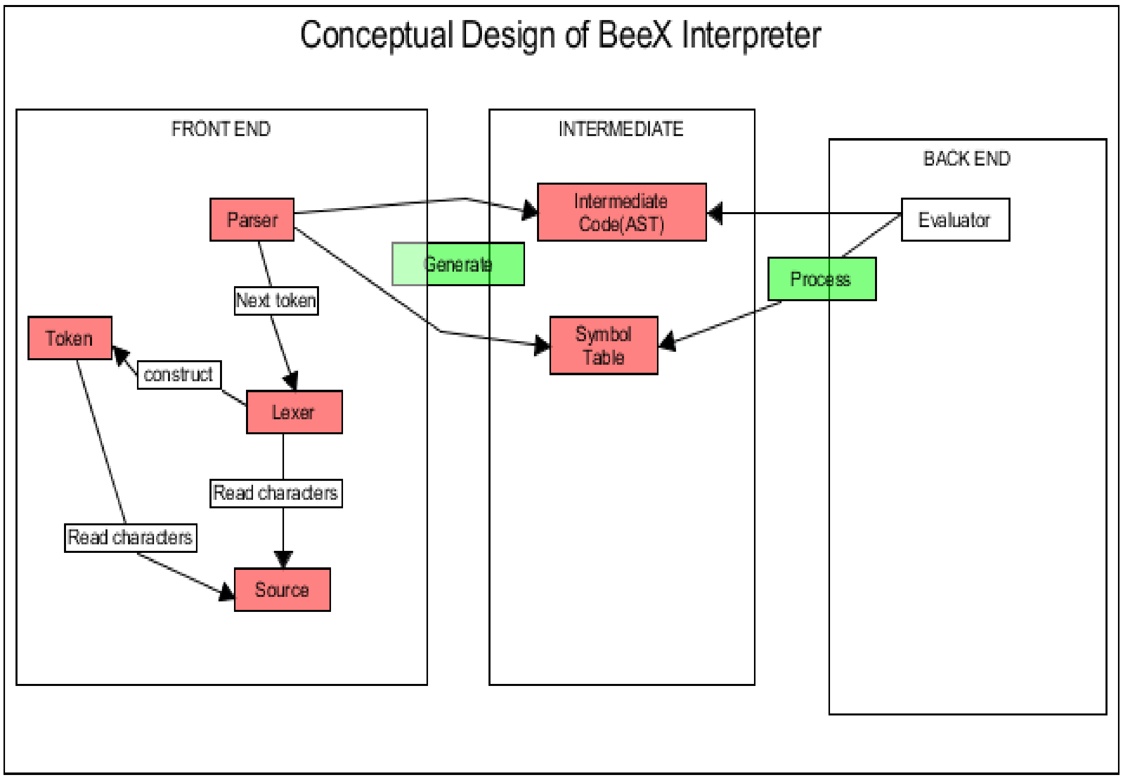


Figure 1: Conceptual Design of the BeeX Interpreter

Design components

The design depicted in Figure 1 is broken down into sub-components as follows:

- Grammar Rules
- Front End
    i. Lexing module.
    ii. Parsing module.
- Back End
    i. AST module (Abstract Syntax Tree)
    ii. Object system module.
    iii. Evaluation module.

Grammar for BeeX

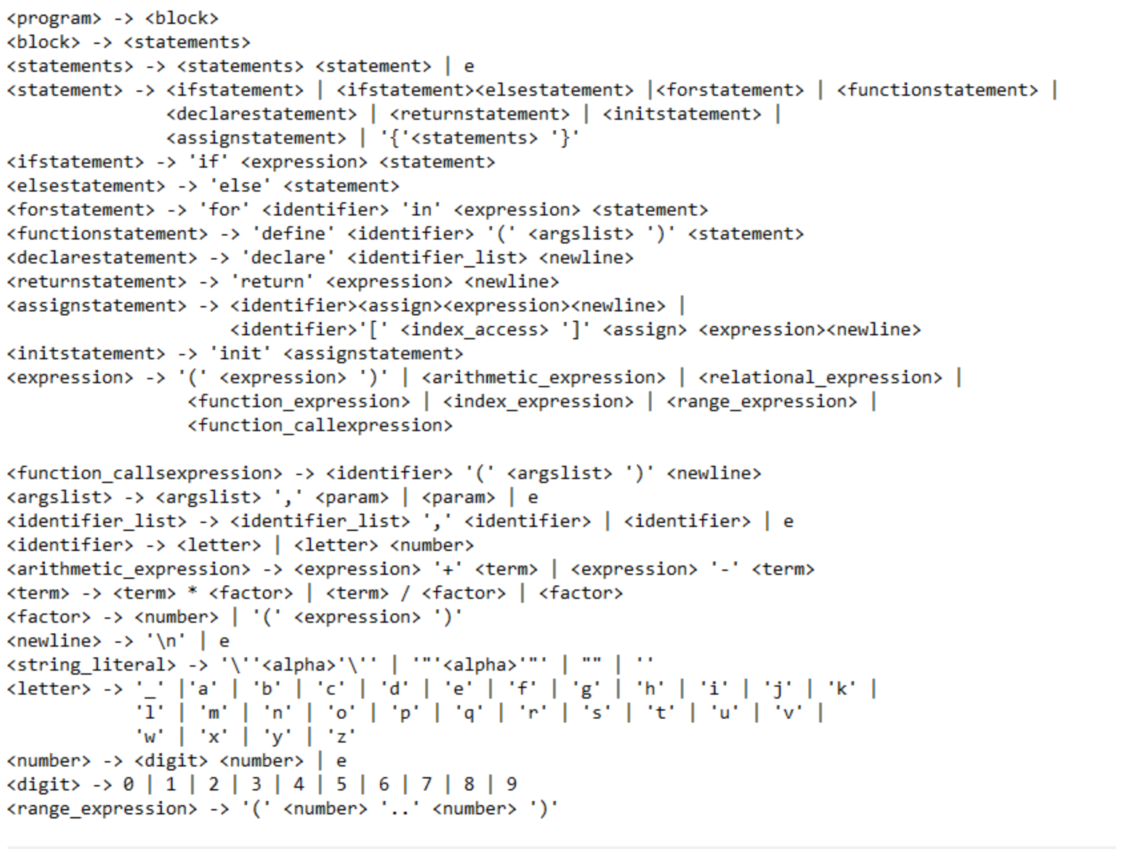
The grammar for BeeX is defined in Backus-Naur form [15], a popular notation used to describe syntax of programming languages. A section of the BeeX grammar rules is given is follows:

<program> -> *<block>*

<block> -> *<statements>*

<statements> - > *<statements>* *<statement>* | *e*

Figure 2 shows the current grammar rules of BeeX, which can be extended as the language evolves.



```
<program> -> <block>
<block> -> <statements>
<statements> -> <statements> <statement> | e
<statement> -> <ifstatement> | <ifstatement><elsestatement> |<forstatement> | <functionstatement> |
               <declarestatement> | <returnstatement> | <initstatement> |
               <assignstatement> | '{'<statements> '}'
<ifstatement> -> 'if' <expression> <statement>
<elsestatement> -> 'else' <statement>
<forstatement> -> 'for' <identifier> 'in' <expression> <statement>
<functionstatement> -> 'define' <identifier> '(' <argslist> ')' <statement>
<declarestatement> -> 'declare' <identifier_list> <newline>
<returnstatement> -> 'return' <expression> <newline>
<assignstatement> -> <identifier><assign><expression><newline> |
                     <identifier>'[' <index_access> ']' <assign> <expression><newline>
<initstatement> -> 'init' <assignstatement>
<expression> -> '(' <expression> ')' | <arithmetic_expression> | <relational_expression> |
                 <function_expression> | <index_expression> | <range_expression> |
                 <function_callexpression>

<function_callsexpression> -> <identifier> '(' <argslist> ')' <newline>
<argslist> -> <argslist> ',' <param> | <param> | e
<identifier_list> -> <identifier_list> ',' <identifier> | <identifier> | e
<identifier> -> <letter> | <letter> <number>
<arithmetic_expression> -> <expression> '+' <term> | <expression> '-' <term>
<term> -> <term> * <factor> | <term> / <factor> | <factor>
<factor> -> <number> | '(' <expression> ')'
<newline> -> '\n' | e
<string_literal> -> '\''<alpha>'\'' | '"'<alpha>'"' | "" | ''
<letter> -> '_' |'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' |
            'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' |
            'w' | 'x' | 'y' | 'z'
<number> -> <digit> <number> | e
<digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<range_expression> -> '(' <number> '..' <number> ')'
```

Figure 2: Grammar rules for BeeX

BeeX interpreter frontend

The front end consists of the lexer, parser, program source, and tokens representing code units. This section explores the functions of the front-end components and their interactions.

Lexing:

The lexer is the component in the front end of the interpreter that performs the syntactic actions of reading the source program and breaking it into tokens.

An example of a valid statement in BeeX is given by (1)

init myvar = 5 (1)

Statement (1) represents a variable initialization statement that declares a new variable with a specified name and assigns a value to that variable.

Given a source program with the statement (1), the parser which contains (owns) a lexer, keeps requesting for tokens until no token is left. The lexer constructs the statement (1) as the following tokens, as depicted in Table 1.

Table 1: lexems for statement "init myvar = 5"

TYPE	TEXT STRING
Word (reserved word)	Init
Word(identifier)	Myvar
Token(Assign)	=
Number(integer)	5

Similar example of Table 1 was also given by [2**]**. Figure 3 depicts the UML diagram for the lexer module.

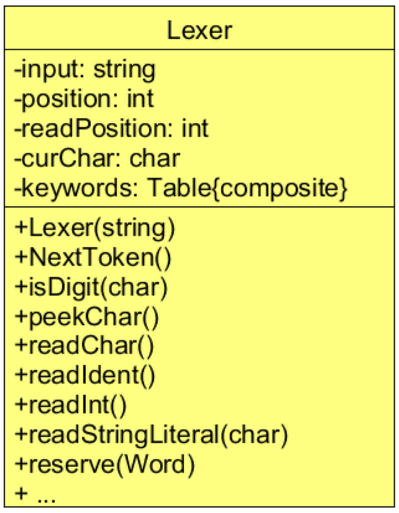


Figure 3: UML diagram for Lexer module

Parsing

Based on both the source language's syntax and semantics, a parser performs actions [2]. Syntactic actions involve: scanning the source program and extracting tokens, and looking for '=' token [2**]**. However, making entries to the symbol table (i.e. entering identifiers "a", "b" and "c" into the symbol table as variables), or looking them up on the symbol table, are semantic actions because the parser had to understand the meaning of the expression and the assignment to know that it needs to use the symbol table [2]. In addition, syntactic actions occur in the front end, while semantic actions can occur on either the front end or the back end [2]. Figure 4 depicts the UML diagram for the parser module.

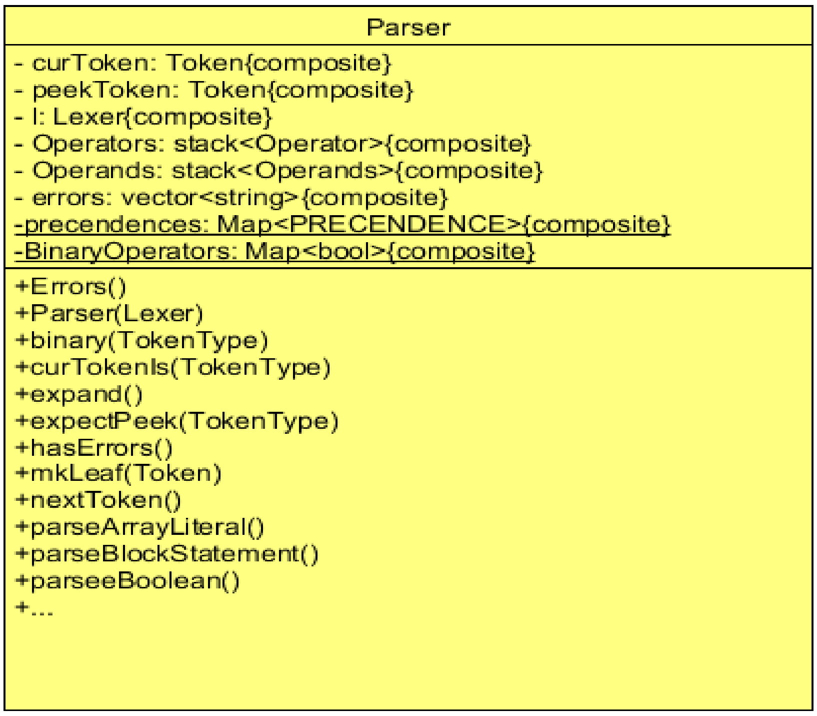

Parser
- curToken: Token{composite}
- peekToken: Token{composite}
- l: Lexer{composite}
- Operators: stack<Operator>{composite}
- Operands: stack<Operands>{composite}
- errors: vector<string>{composite}
-precendences: Map<PRECENDENCE>{composite}
-BinaryOperators: Map<bool>{composite}
+Errors()
+Parser(Lexer)
+binary(TokenType)
+curTokenIs(TokenType)
+expand()
+expectPeek(TokenType)
+hasErrors()
+mkLeaf(Token)
+nextToken()
+parseArrayLiteral()
+parseBlockStatement()
+parseeBoolean()
+...

Figure 4: UML diagram for Parser Module

Symbol Table

A key component used by different phases throughout the translation lifecycle of an interpreter or compiler is the symbol table. The parser, as part of semantic analysis, builds and maintains a symbol table throughout the process of the parse phase of interpretation. In addition, information about the source program's token, mostly the identifiers are then stored in the symbol table. The interpreter, during the translation process, creates and updates entries in the symbol table to contain information about certain token's in the source program [2]. Each entry has a name, which is the token's text string, and the entry also contains information about the identifier [2]. The interpreter, thereafter, looks up and updates the information, as the source program is translated [2].

Parsing in BeeX may require more than one symbol table (one table for each function, block statement, etc.). Since functions can be nested, the symbol tables are created to be linked to each other, starting out from the global symbol table (where global variables and functions are stored). A symbol table is created from every function, which references the immediate symbol table above it. This technique allows *scoping* (refers to the visibility of variables within a region of the program) of function variables or block level variables. Figure 5 depicts the symbol table structure for BeeX.

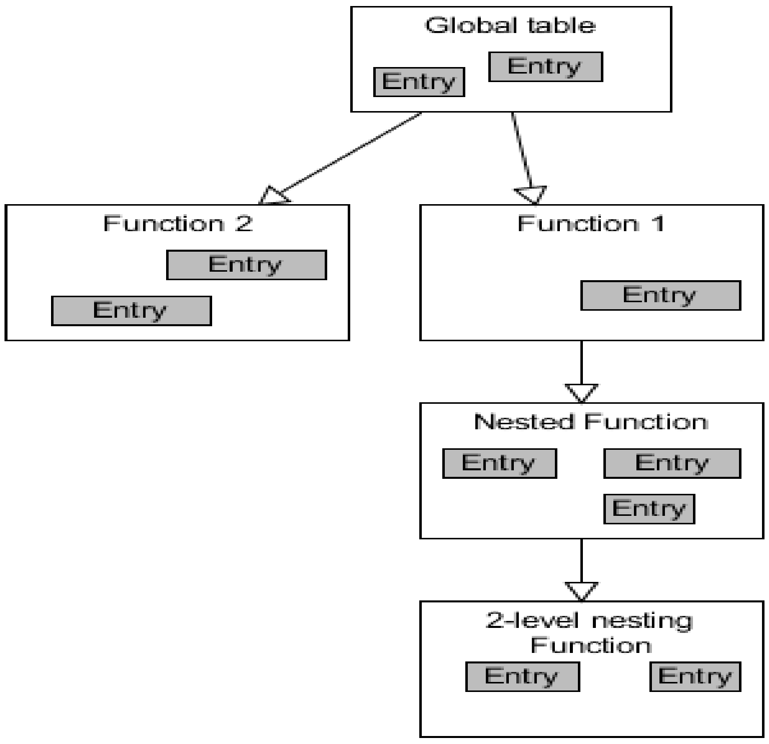


Figure 5: Diagram of Symbol Table Linked Structure for Beex

BeeX Interpreter Backend

The back end consists of the object system module, abstract syntax module and evaluation module. The parser constructs an intermediate representation of the source program known as an Abstract Syntax Tree (AST), which is an abstract representation of the source program. The AST is then passed down to the evaluator, which then evaluates these statements as instructions in the host language (BeeX) using an object system where each evaluated instruction results to an object defined by this system. Figure 6 depicts a conceptual view of the interpreter backend.

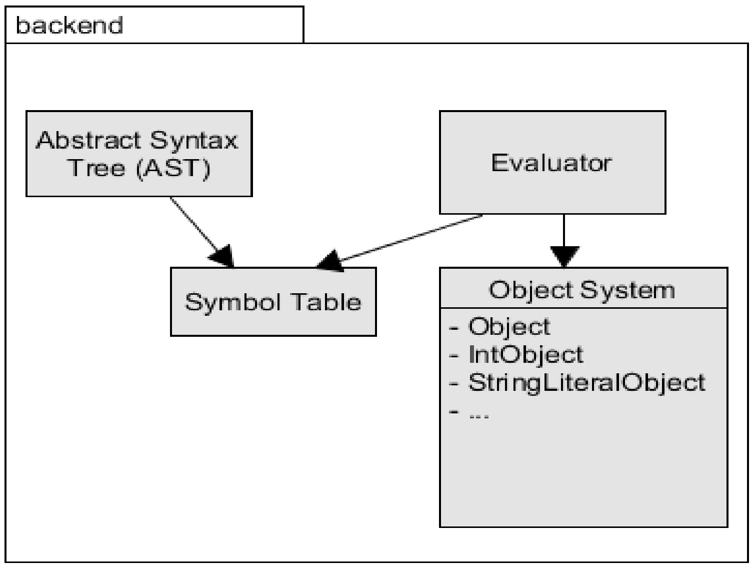


Figure 6: Backend Module

System use case

This section illustrates the system use case by describing the process involved from having a source file (e.g. **myprogram.bx**) to having the program output returned from the interpreter.

The programmer specifies a file containing the source code – a file ending with the extension "bx", for BeeX to the interpreter; this is done by passing the file name of source code to the interpreter program to be executed. An example of executing the interpreter is given as follows:

execute. /Interpreter **filename.bx**

The interpreter is executed using the specified file as the source program; the interpreter does the lexical analysis and syntax analysis to verify if the source code is syntactically correct, these are all part of the front-end system. After the front-end processes are completed, an intermediate code is passed to the back-end system which executes the instructions from the intermediate code. When the backend system processes are completed, the system returns the result of the executed program. Figure 7 depicts the system use case diagram for BeeX interpreter.

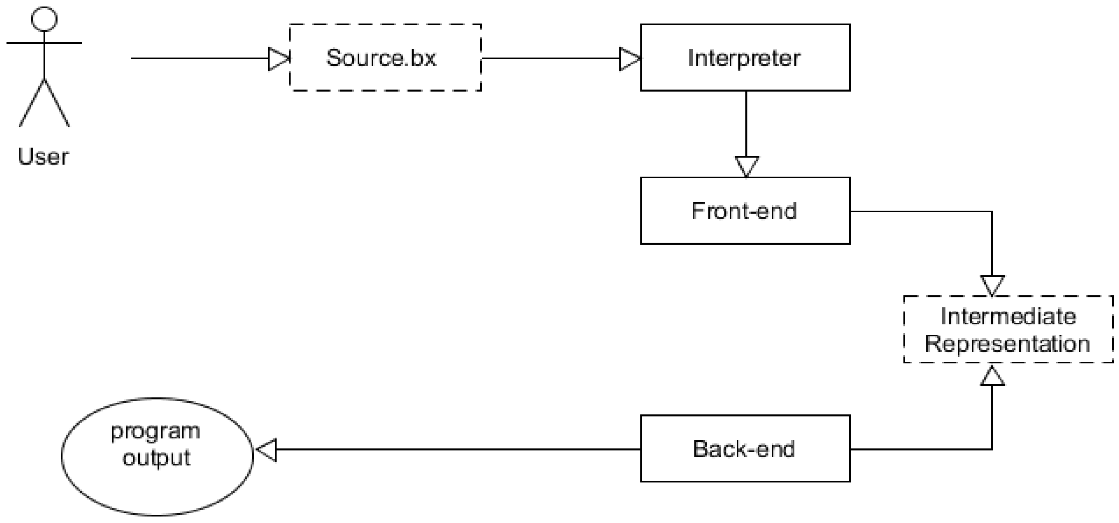


Figure 7: System use-case diagram

SYSTEM IMPLEMENTATION AND TESTING

Hardware requirements

The following are the hardware requirements used in the development of BeeX interpreter:

i. Processor: Intel® Core i5 with a clock speed of 2.60GHz

ii. Physical Memory (RAM): 8.00GB

iii. Internet devices such as modem, Wi-Fi adapter.

Choice of software tools

The following tools were used in developing the BeeX interpreter:

i. QT Creator IDE
Qt Creator is an integrated software development environment that supports both traditional C++ application development and development using the Qt project's libraries [16]. Qt is a powerful development framework that serves as a complete toolset for building cross-platform applications that help in reducing development time and improving productivity [16].

ii. CLion IDE

JetBrains CLion is a cross-platform IDE for C and C++ with support for biicode [17]. CLion can be used on Windows. However, for developers that use Microsoft's Visual Studio for C++ development, JetBrains is updating ReSharper, its VS extension, which supports C#, .NET, and web development languages to also support C++ [17].

iii. UMLet

UMLet is an open-source Java-based Unified Modeling Language tool that was designed to teach Unified Modeling Language and for rapidly making UML drawings. Diagrams are made by dragging the UML components into the view section. It is rather a drawing tool than a modeling tool because there is no fundamental vocabulary or manual of reusable design objects [18].

iv. GDB (The GNU Project Debugger)

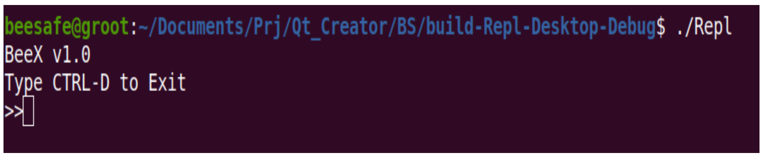
The GNU Debugger allows you to see what is going on "inside" a program while it executes, or what a program was doing at the moment it crashed [19**]**. GDB supports C, C++, Java, Fortran and Assembly among other languages [19**]**; it is also designed to work closely with the GNU Compiler Collection (GCC) [19].

Interpreter implementation

REPL (Read Evaluate Print Loop)

To aid visualization of several syntax implementations, a REPL – an interactive tool that takes user inputs, evaluates them and returns results to the user, was implemented to allow students quickly try out the language features, without having to create a new source file.

Figure 8 depicts an image of the interactive programming environment.



```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>
```

Figure 8: REPL for BeeX interpreter Evaluating expressions

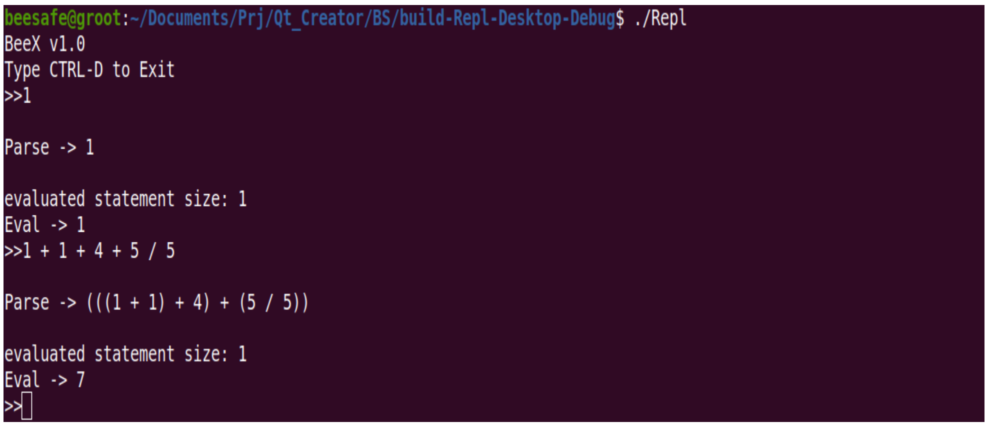
In this sub-section we demonstrated various expressions allowed in the BeeX language.

The following are expressions used in BeeX;

i. Arithmetic expressions:

Arithmetic expressions provide students the ability to perform basic and complex mathematical operations programmatically**.** These are expressions that evaluate to arithmetic values. e.g. 1 + 1, 2/2, 3 * 6 – 6, etc.

Figure 9 depicts the screenshot of arithmetic expressions in BeeX

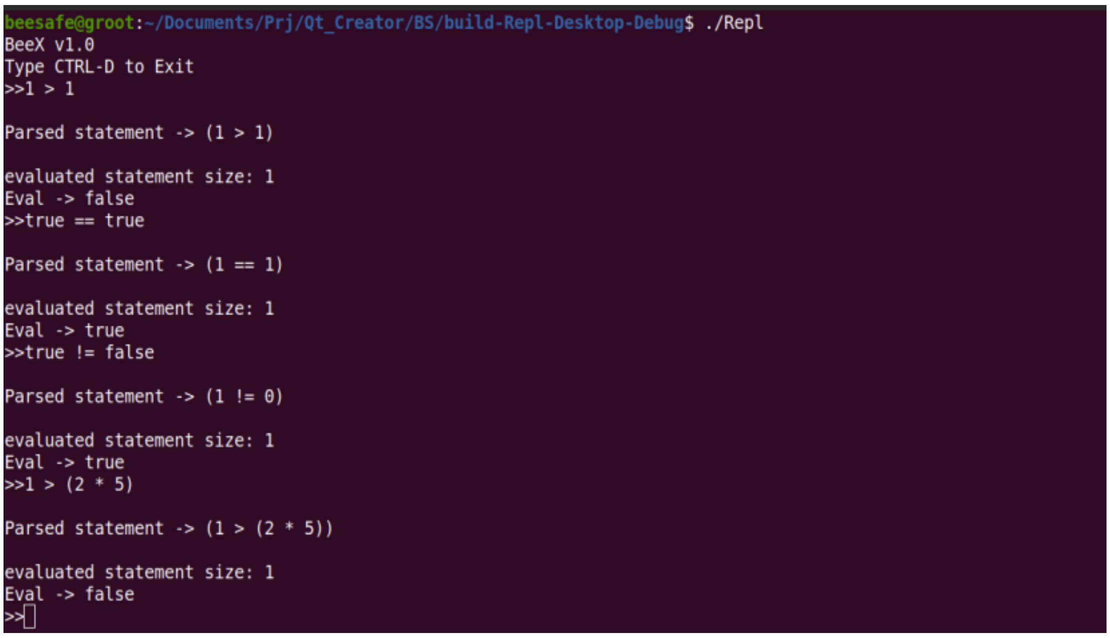


```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>1

Parse -> 1

evaluated statement size: 1
Eval -> 1
>>1 + 1 + 4 + 5 / 5

Parse -> (((1 + 1) + 4) + (5 / 5))

evaluated statement size: 1
Eval -> 7
>>
```

Figure 9: screenshot of arithmetic expressions in BeeX

ii. Logical expressions:

Logical expressions are required in programming languages to represent different control flows a program can take. BeeX logical expressions allow the programmer create control flows concisely**.** These are expressions that evaluate to logic values also known as Boolean values (true or false). e.g. 1 > 1, 2 >= 0, 1 ! = 0, etc. Figure 10 depicts the screenshot of logical expressions in BeeX

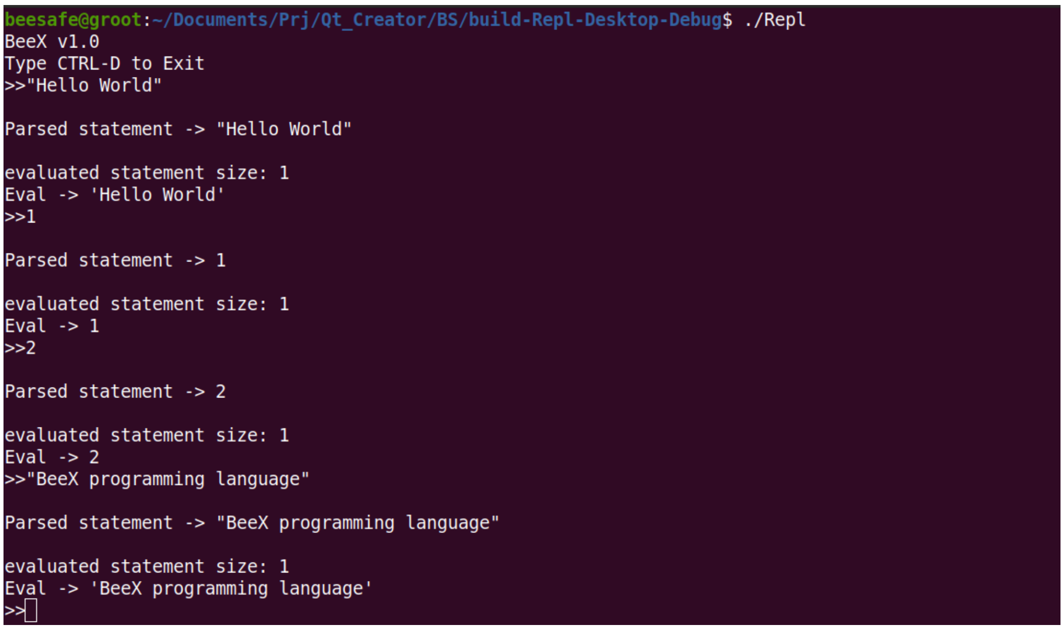


```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>1 > 1

Parsed statement -> (1 > 1)

evaluated statement size: 1
Eval -> false
>>true == true

Parsed statement -> (1 == 1)

evaluated statement size: 1
Eval -> true
>>true != false

Parsed statement -> (1 != 0)

evaluated statement size: 1
Eval -> true
>>1 > (2 * 5)

Parsed statement -> (1 > (2 * 5))

evaluated statement size: 1
Eval -> false
>>
```

Figure 10: Screenshot of logical expressions in BeeX

iii. Literal expressions:

Literals are unit structures in BeeX. They represent primitive values such as numbers, string values, and boolean values. An expression containing a literal would be evaluated with the values of the corresponding literal**.** These are expressions that contain only literal values; for example: “hello world”, 3, 4, true, false, etc. Figure 11 depicts the screenshot of literal expressions in BeeX



```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>"Hello World"

Parsed statement -> "Hello World"

evaluated statement size: 1
Eval -> 'Hello World'
>>1

Parsed statement -> 1

evaluated statement size: 1
Eval -> 1
>>2

Parsed statement -> 2

evaluated statement size: 1
Eval -> 2
>>"BeeX programming language"

Parsed statement -> "BeeX programming language"

evaluated statement size: 1
Eval -> 'BeeX programming language'
>>
```

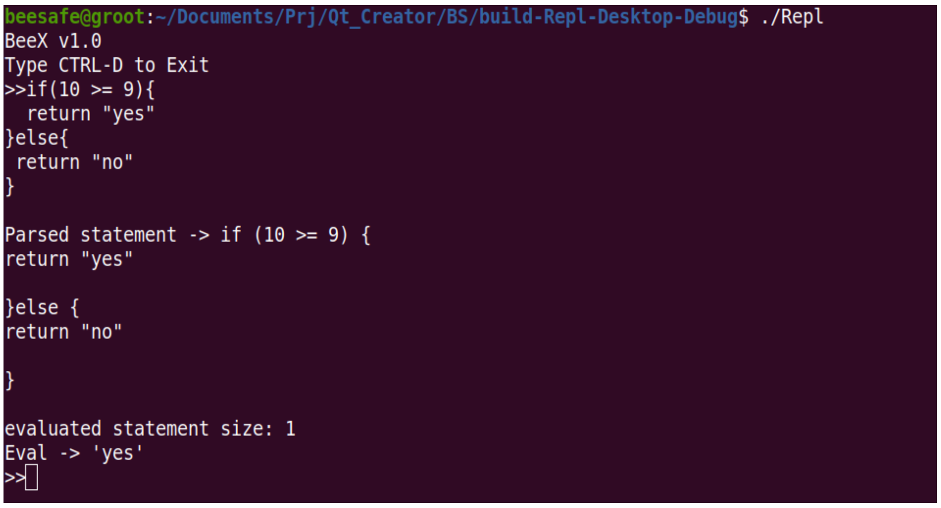
Figure 11. Screenshot of literal expressions in BeeX

Evaluating Statements

In this sub-section we demonstrated various statements allowed in the BeeX language.

i. If-Else statement

This represents the semantic conditional logic statement. Similar to logic expressions, logical statements are used to make decision of control flows on a program. BeeX supports if-else logical statement to better represent control flows.   Figure 12 depicts the screenshot of the if-else statement in BeeX.

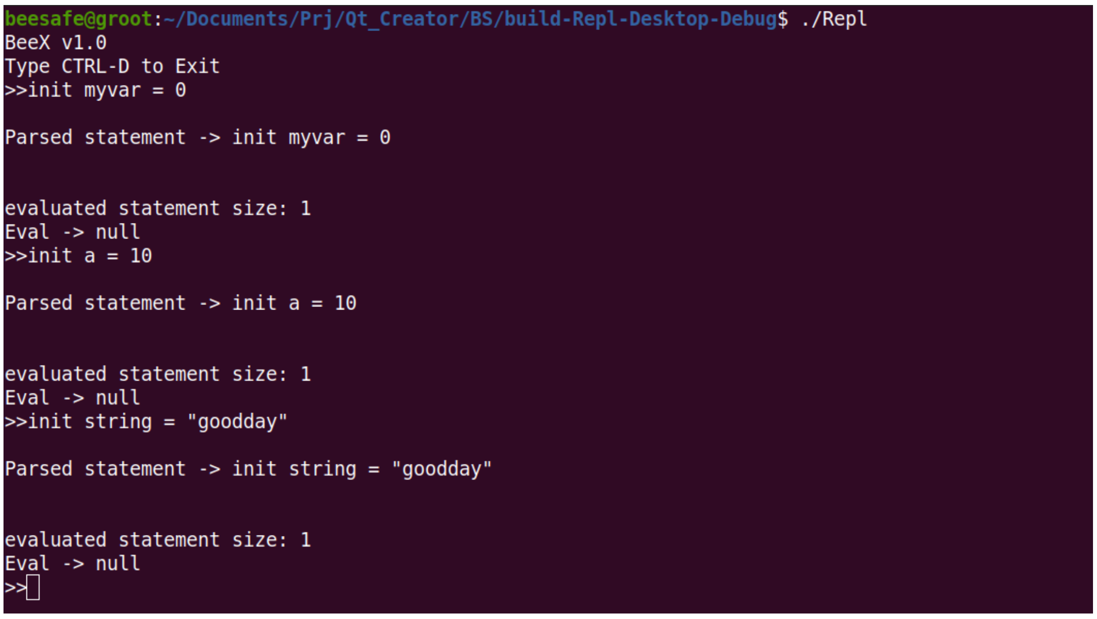


```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>if(10 >= 9){
  return "yes"
}else{
 return "no"
}

Parsed statement -> if (10 >= 9) {
return "yes"

}else {
return "no"

}

evaluated statement size: 1
Eval -> 'yes'
>>
```

Figure 12: Screenshot of If-Else statement in BeeX

ii.   Init statement

This represents the semantic action for initializing variables with values given by the programmer. Using the init keyword tells the interpreter that the variable would be provided with a value, this expressiveness enables better readability. A variable is declared and initialized with a value using the init statement depicted in Figure 13.

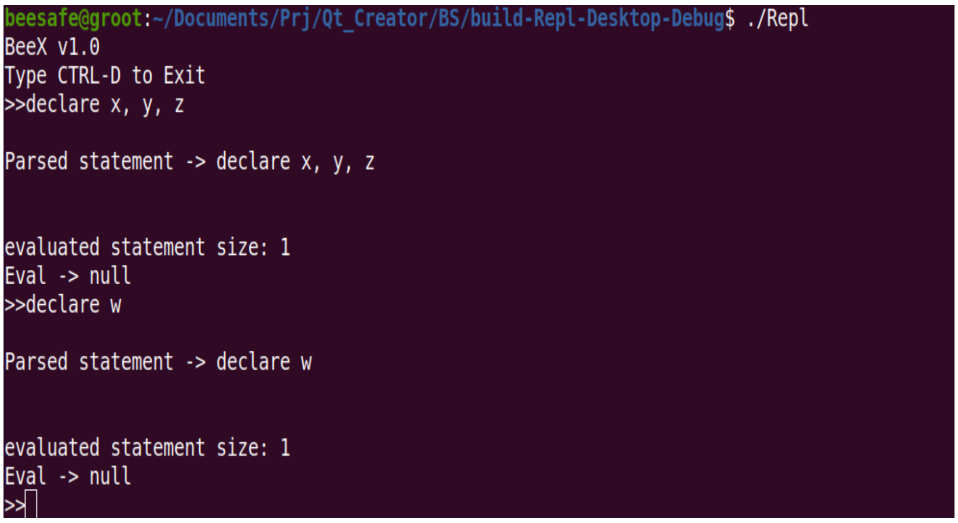


```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>init myvar = 0

Parsed statement -> init myvar = 0


evaluated statement size: 1
Eval -> null
>>init a = 10

Parsed statement -> init a = 10


evaluated statement size: 1
Eval -> null
>>init string = "goodday"

Parsed statement -> init string = "goodday"


evaluated statement size: 1
Eval -> null
>>
```

Figure 13: Screenshot of init statement in BeeX

iii. Declare statement

This represents the semantic action for declaration of variables, where variables are assigned default values. Figure 14 shows the screenshots of how variables are declared in BeeX.

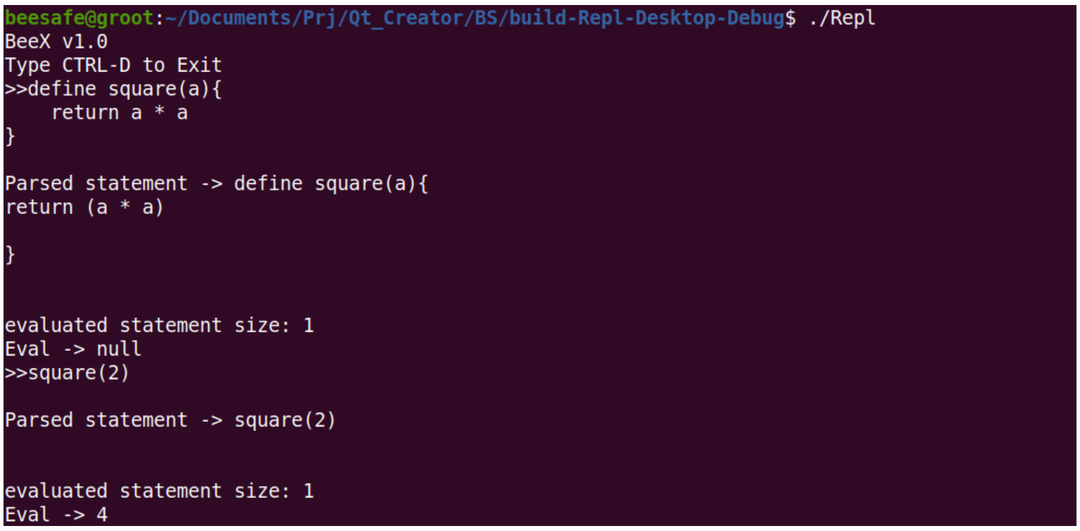


```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>declare x, y, z

Parsed statement -> declare x, y, z


evaluated statement size: 1
Eval -> null
>>declare w

Parsed statement -> declare w


evaluated statement size: 1
Eval -> null
>>
```

Figure 14: Screenshot of declare statement in BeeX

iv. Define statement

This represents the semantic action for defining a function or procedure. The define statement depicted in Figure 15 allows the programmer define custom functions in BeeX; it consists of five semantic parts namely**:** define, keyword, function name, parameters, and function body. Figure 15 shows the screenshot of how a function or procedure is defined in BeeX.

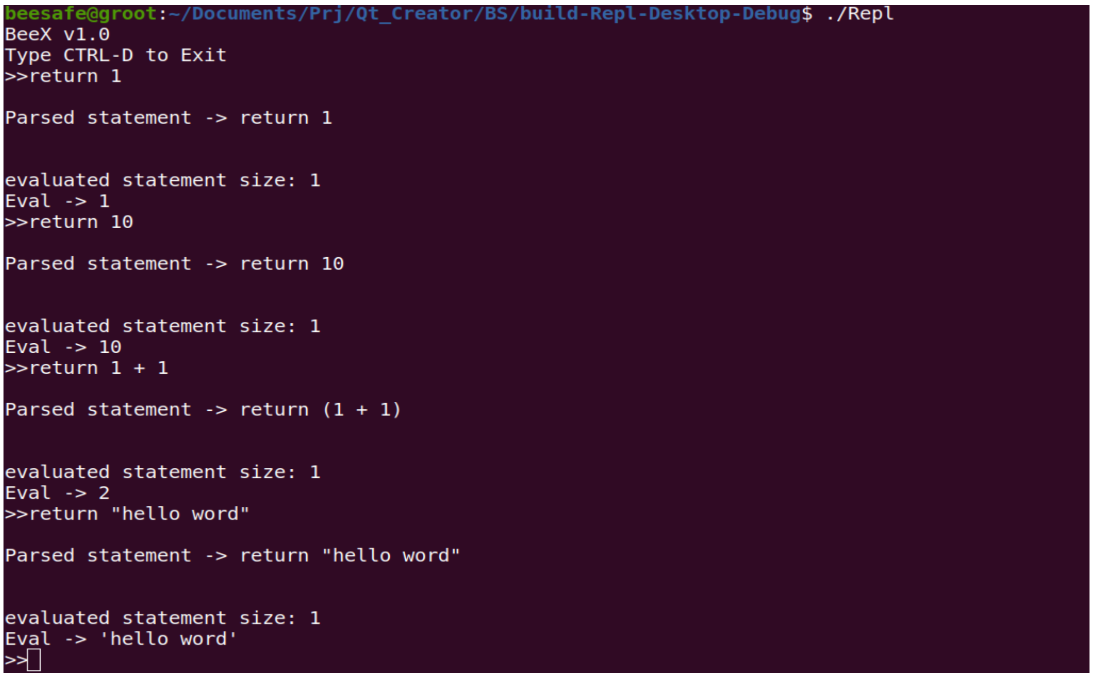


```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>define square(a){
    return a * a
}

Parsed statement -> define square(a){
return (a * a)

}


evaluated statement size: 1
Eval -> null
>>square(2)

Parsed statement -> square(2)


evaluated statement size: 1
Eval -> 4
```

Figure 15: Screenshot of define statement in BeeX

v. Return statement

This represents the semantic action for return control to the main routine. A return statement can specify a value to be returned from a function or procedure. Figure 16 shows the screenshot of a return statement in BeeX.

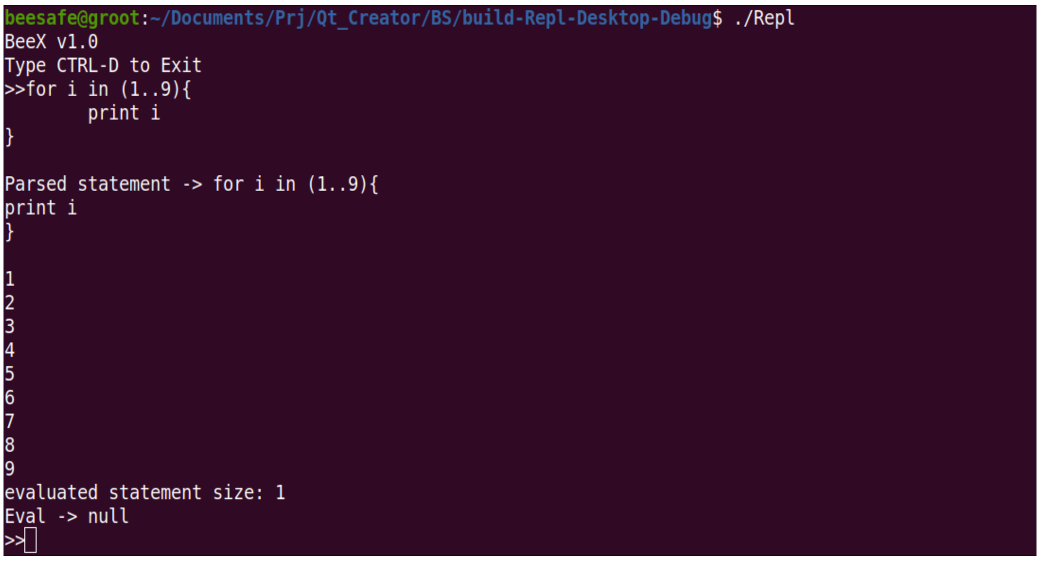


```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>return 1

Parsed statement -> return 1


evaluated statement size: 1
Eval -> 1
>>return 10

Parsed statement -> return 10


evaluated statement size: 1
Eval -> 10
>>return 1 + 1

Parsed statement -> return (1 + 1)


evaluated statement size: 1
Eval -> 2
>>return "hello word"

Parsed statement -> return "hello word"


evaluated statement size: 1
Eval -> 'hello word'
>>
```

Figure 16: Screenshot of return statement in BeeX.

vi. For-Range statement

This represents the semantic action of a loop operation using a range expression which specifies a range of integer values to loop over. Figure 17 shows the screenshot of for-range statement in BeeX.



```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Repl-Desktop-Debug$ ./Repl
BeeX v1.0
Type CTRL-D to Exit
>>for i in (1..9){
        print i
}

Parsed statement -> for i in (1..9){
print i
}

1
2
3
4
5
6
7
8
9
evaluated statement size: 1
Eval -> null
>>
```

Figure 17: Screenshot of for-range statement in BeeX

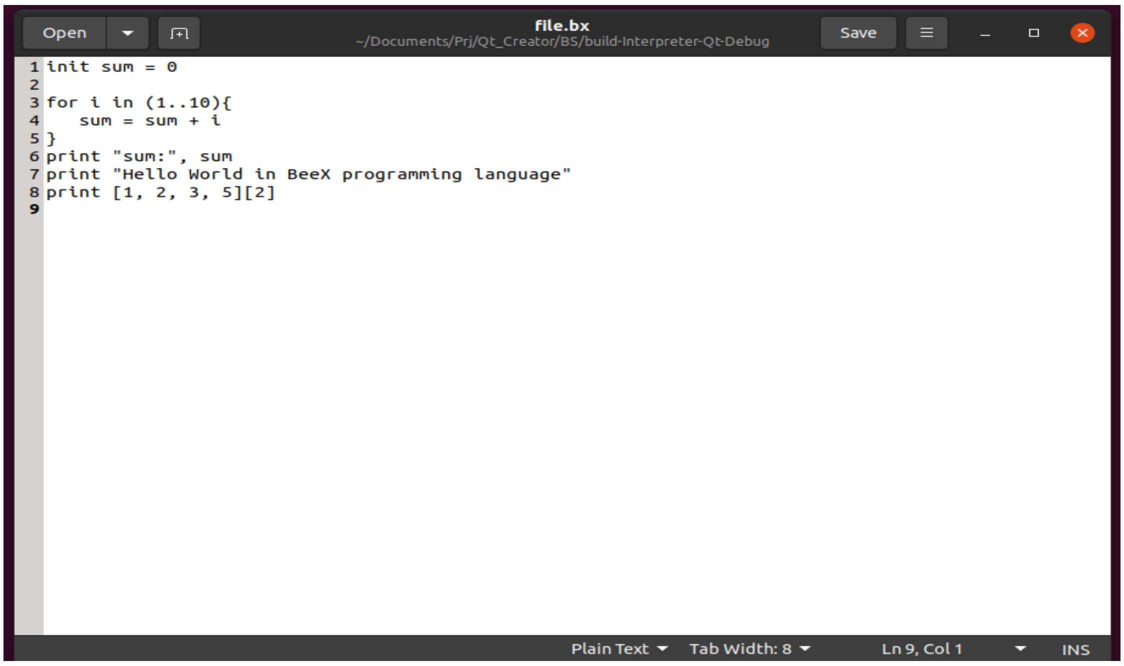
Testing

In this section we did a test run of the interpreter by using a test source code that contains correct syntax of the BeeX programming language. We also showed the results of the interpreter when given an ill-formed syntax source file.

Test run interpreter with correct syntax

A source file is specified to the interpreter program, which the interpreter executes and outputs results from the instructions given in the file.

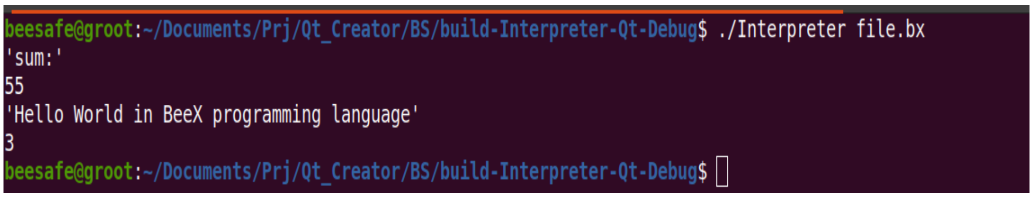
For this test run we made use of a file with the correct syntax. Figure 18 shows the screenshot of the contents of the test source file.



```
init sum = 0

for i in (1..10){
    sum = sum + i
}
print "sum:", sum
print "Hello World in BeeX programming language"
print [1, 2, 3, 5][2]
```

Figure 18: Screenshot of test source file

Figure 19 is a screenshot of the outputs of the interpreter when given a source file in Figure 18.



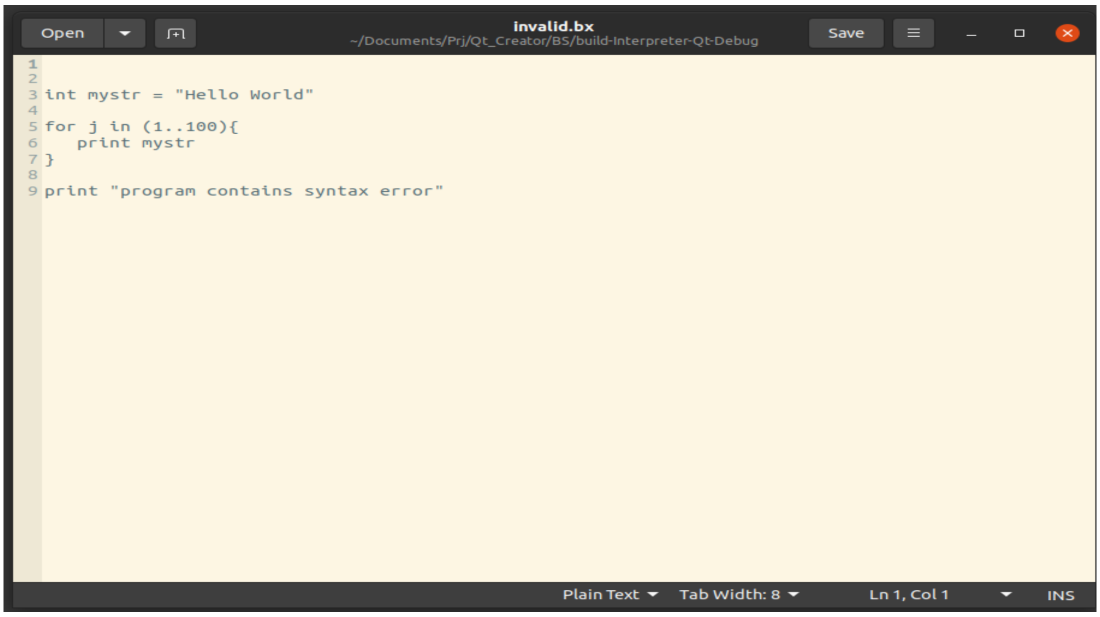
```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Interpreter-Qt-Debug$ ./Interpreter file.bx
'sum:'
55
'Hello World in BeeX programming language'
3
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Interpreter-Qt-Debug$
```

Figure 19: Screenshot of results after interpretation

The result of the program is displayed on the console, which indicates the interpreter successfully executing the source code without any errors. We showed that BeeX can be used to perform basic programming operations that can be combined to do meaningful tasks**.**

Test run interpreter with incorrect syntax

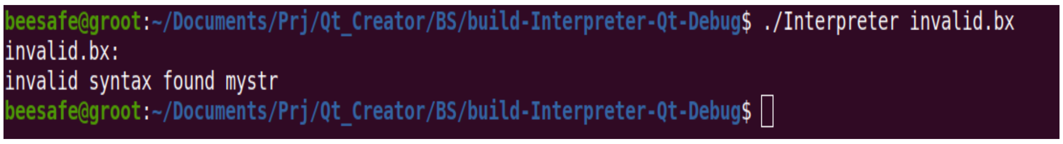
Here, we followed the steps from the first test run, but made the test source file contain invalid syntax to test how the interpreter recognizes the error. Test source file with invalid syntax is depicted in Figure 20.



```


int mystr = "Hello World"

for j in (1..100){
    print mystr
}

print "program contains syntax error"
```

Figure 20: Screen shot of test source with invalid syntax

Figure 21 depicts the screenshot of the results for the interpreter. We observed here that the interpreter encountered an error and flagged it.



```
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Interpreter-Qt-Debug$ ./Interpreter invalid.bx
invalid.bx:
invalid syntax found mystr
beesafe@groot:~/Documents/Prj/Qt_Creator/BS/build-Interpreter-Qt-Debug$ 
```

Figure 21: Screenshot of results from interpreter

With the tests given so far, we can conclude that the interpreter executes properly.

CONCLUSION

The importance of programming language design was discussed in this study by identifying the components involved in the design and defining the roles played by each component. A proposed language (BeeX) was designed, and the language's interpreter was also implemented.

Students and programmers alike can use the BeeX programming language to create a variety of code structures that are simple to use and productive. This is possible due to the language's simplicity and simple code structures. Thus, students would be more focused on the principles of computer science and engineering and spend less time attempting to master the 'nuances' of a programming language. However, these ideas can be put into practice if they are taught in a language like BeeX.

We were able to demonstrate the value of careful design choices in the creation of any language. This study only covered three interpretation phases viz: lexing, parsing, and evaluation. This is limited to basic components involved in building a language interpreter and does not cover many advanced techniques commonly used in various language interpreters which include: code optimization, byte-code representation, and other forms of abstract syntax tree construction.

This study focused on the design and implementation of a Tree-Walk interpreter, which is limited to various standard techniques adopted in more widely, used language interpreters (.e.g. Ruby interpreter, Python interpreter etc.). Given the purpose of the study, its implementation does not comply with industrial standards for language design such as handling floating point arithmetic (as stipulated by IEEE) amongst others. The language is intended for use in a learning environment, and it would be excellent for applications in that domain, such as support for mathematical modules, ease of implementing engineering procedures, and so on.

The language's first draft has a few limitations that will be addressed in later versions. The following are some of the issues/improvements that will be handled in the future:

i. Creating solid module architecture that allows users to increase the language's capabilities.

ii. Adding the ability to handle user-defined exceptions.

iii. Adding support for floating-point numbers and evaluating statements.

iv. Adding a reliable garbage collector.

REFERENCES

[1] A. Demaille, “Making Compiler Construction Projects Relevant to Core Curriculums,” In proceeding of: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2005, Caparica, Portugal, ACM SIGCSE Bulletin, 37(3):266-270, June 27-29, 2005, doi:10.1145/1151954.1067518.

[2] F. Wu, H. Narang and M. Cabral , “Design and Implementation of an Interpreter Using Software Engineering Concepts”, *International Journal of Advanced Computer Science and Applications (IJACSA),* (5) 7, 2014, doi: 10.14569/IJACSA.2014.050726.

[3] C. Xing. “How Interpreters Work: An Overlooked Topic in Undergraduate Computer Science Education,” Proc. In CCSC Southern Eastern Conference, JCSC , (25), 2. December 2009, doi:10.5555/1629036.1629062.

[4] P.T. Vaikunta , A. J. Devi and P. S. Aithal, “A Systematic Literature Review of Lexical Analyzer Implementation Techniques in Compiler Design”, *International Journal of Applied Engineering and Management Letters (IJAEML)*, 4(2), 285-301, 2020, ISSN: 2581-7000, doi:10.5281/zenodo.4454632.

[5] O. Apte, S. Agre, R. Adepu and M. Ganjapurkar, “C To Python Programming Language Translator”, *Journal of Emerging Technologies and Innovative Research (JETIR)* , 8, 5, 2021, doi:10.1729/Journal.26932.

[6] W. Müller and U. Kortenkamp , “Learning Programming With Ruby”. Conference: Proceedings of the IFIP Workshop "New developments in ICT and Informatics education” , 2010, Online [Available]: https://www.researchgate.net/publication

[7] L. Welling and L. Thomson, “PHP and MySQL Web Development. 5th Edition”, Addison - Wesley, Indianapolis, USA, 2017.

[8] M. Hills and P. Klint , "PHP AiR: Analyzing PHP Systems with Rascal," IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE) pp. 454- 457, 2014, doi:10.1109/CSMR-WCRE.2014.6747217.

[9] T. Ratschiller and T. Gerken ,“Web Application Development with PHP 4.0”, New Riders Publishing, Indianapolis, Indiana, USA, 2000.

[10] M. Xie, D. Li , M. Cheng, S. Li and Y. Luo “Design and implementation of an efficient program interpreter for industrial robot”, Journal of Physics: Conference Series, Volume 1884, 2021 International Conference on Intelligent Manufacturing and Industrial Automation (CIMIA 2021) 26-28 March 2021, Guilin, China. doi:10.1088/1742-6596/1884/1/012018.

[11] O. Banyasad and P. T. Cox , “Design and implementation of an editor/interpreter for a visual logic programming language” , *International Journal of Software Engineering and Knowledge Engineering*, (23), 6, 801-838, 2013, doi:10.1142/S0218194013500216.

[12] A. Kiani, H. Ebadi, F. F. Ahmadi and S. Masoumi , “Design and implementation of an expert interpreter system for intelligent acquisition of spatial data from aerial or remotely sensed images”, *Measurement,* (47), 676-685, January 2014, doi:10.1016/j.measurement.2013.09.038.

[13] X. Xiao and Y. Xu, "The Design and Implementation of C-like Language Interpreter," 2nd International Symposium on Intelligence Information Processing and Trusted Computing, 2011, IEEE Conference Publication. pp. 104-107, doi: 10.1109/IPTC.2011.33.

[14] A. Y. Aho, M.S. Lam, R, Sethi and J.D. Ullman , “Compilers: Principles, Techniques, and Tools”, 2ed. Addison - Wesley, Reading, MA, 2006.

[15] A. Ralston, D. D. McCracken and E. D. Reilly , “Backus-Naur form (BNF). Encyclopedia of Computer Science”, 129-131, 2003, do:10.5555/1074100.1074155.

[16] L. Z. Eng and R. Rischpater, “Application Development with Qt Creator - Third Edition”, Packt Publishing Limited, Birmingham, UK, 2020.

[17] J. Kalb and G. Ažman, “C++ Today: The Beast Is Back”. O’Reilly books and Media, Inc, Sebastopol, CA, United States of America, 2015.

[18] M. Auer, S. Biffl and T. Tschurtsche , “A Flyweight UML Modeling Tool for Software Development in Heterogeneous Environments”, Proceedings of the 29th Conference on EUROMICRO, 2003, Online: [Available]:https://www.researchgate.net/publication/4034715.

[19] R. Stallman, R. Pesch , S. Shebs, E. Suvasa and M. Lee , “Debugging with GDB: The GNU source-level debugger”, Amazon Inc, GNU Press, 2011.