University of Technology, Jamaica

Faculty/College: Faculty of Engineering and Computing

School of Computing and Information Technology

Academic Year: 2023/2024

Semester: 2

Module: ANALYSIS OF PROGRAMMING LANGUAGES (CIT4004)

Title: LLM Programming Language Design and Implementation

Group Members:

Hugh Scott - 1908850

Barrington Patterson - 2008034

Sharethia McCarthy - 2000191

Christina Wilson - 1903419

Tutor: Dr. David White

Due Date: April 7, 2024

TABLE OF CONTENTS

INTRODUCTION

TypeSnake

Welcome to *TypeSnake*, our proudly newly developed programming language from our brilliant team. *TypeSnake* is an innovative programming language crafted by our dedicated team of developers, designed to revolutionize the way we approach coding. With its intuitive syntax and powerful features, *TypeSnake* with its AI implementation, empowers programmers to write efficient and maintainable code with ease. Built upon a foundation of the very best education in the Caribbean and aided by The Microsoft Corporation we are proud to present *TypeSnake*.

LANGUAGE DESIGN & OVERVIEW

TypeSnake is a compiler that takes high-level language type-safe syntax and transpiles it into Python when it can be run on the Python interpreter. Together both languages form a compiler, interpreter partnership that leverages the versatile nature of Python, paired with rigid type safety to help find those pesky bugs  .

TypeSnake goes the route of the functional programmer paradigm, it truly embodies the role. It implements the evaluation of mathematical functions and avoids changing state and mutable data. It implements higher-order functions, immutability, and recursions of many forms.

Our team chose a general-purpose programming language approach to ensure versatility and applicability across various domains and applications. TypeSnake should have the potential to slither through many big and small problems. Prioritizing readability, ease of use, and abstraction in our language by abstracting low-level details for clear and concise code. Finally, TypeSnake's front-end is hosted on Vercel while the back end is on Microsoft Azure.

SAMPLE CODE

```
# Variable starts with an optional underscore, followed by a
letter, and then followed by any combination of letters, digits,
or underscores.
# “lock” and “unlock” conform to variable mutability
# int, string, float, and bool are binding to the variables
unlock int _COUNT = (0 + 2 + 3)@
lock string _Global_var = "This is a global variable."@
lock int _TEN = 10@
lock float _PI = 3.14@
unlock bool _BINARY = true@


# Delaring a variable before the assignment
lock int _Assignment@ #Declaration
_Assignment = 5+7@ #Assignment

# scribe will write to the screen
scribe("This is _Assignment", _Assignment)@
```

GRAMMAR SPECIFICATION

BNF Grammar

We decided to go with BNF as the grammar for our language because it's like the Lego instructions of syntax – straightforward and easy to follow! With BNF, we're laying down the blueprint for how our language should be structured, kind of like giving it a striking architectural design. It's our team's trusty tool for defining the rules that govern how our language's sentences and expressions come together, ensuring they fit snugly and make sense. BNF makes our language design feel like putting together a puzzle – everything clicks into place just right! Plus, it's a neat way to give our language a formal touch, adding a sprinkle of elegance to its charm. So, grab your syntax guidebook, and let's build something awesome together!

<program> ::= <statements>

<statements> ::= <statement> <statements>

| <empty>

<statement> ::= <declaration>

| <assignment>

| <abstract_call>

| <abstract_function_declaration>

| <print_statement>

| <conditionals>

| <attempt_findout_block>

<declaration> ::= <mutex> <type> <IDENTIFIER> "=" expression "@"

| <mutex> <type> <IDENTIFIER> "@"

<assignment> ::= <IDENTIFIER> "=" <expression> "@"

<abstract_call> ::= "HAIL" <FUNCTIONID> "(" <arguments> ")" "@"

<abstract_function_declaration> ::= "ABSTRACT" <FUNCTIONID> "(" <parameters> ")" "{" <statements> "}"

<print_statement> ::= "SCRIBE" "(" <STRING_LITERAL> "," <IDENTIFIER> ")" "@"

| "SCRIBE" "(" <STRING_LITERAL> ")" "@"

| "SCRIBE" "(" <IDENTIFIER> ")" "@"

<conditionals> ::= <if_statement>

| <for_statement>

| <aslongas_statement>

<if_statement> ::= "IF" <expression> "{" <statements> "}"

| "IF" <expression> "{" <statements> "}" "ELSE" "{" <statements> "}"

| "IF" <expression> "{" <statements> "}" "ELIF" <expression> "{" <statements> "}"

| "IF" <expression> "{" <statements> "}" "ELIF" <expression> "{" <statements> "}" "ELSE" "{" <statements> "}"

<for_statement> ::= "FOR" <IDENTIFIER> "IN" "RANGE" "(" <arguments> ")" "{" <statements> "}"

| "FOR" <IDENTIFIER> "IN" <iterables> "{" <statements> "}"

<iterables> ::= <STRING_LITERAL>

| <IDENTIFIER>

<aslongas_statement> ::= "ASLONGAS" <expression> "{" <statements> "}"

<expression> ::= "(" <expression> ")"

| <expression> "+" <expression>

| <expression> "-" <expression>

| <expression> "*" <expression>

| <expression> "/" <expression>

| <expression> "" <expression>**

| <expression> "!=" <expression>

| <expression> "<" <expression>

| <expression> ">" <expression>

| <expression> "<=" <expression>

| <expression> ">=" <expression>

| <expression> "==" <expression>

| <expression> "&" <expression>

| <expression> "|" <expression>

| <expression> "^" <expression>

| <expression> "<<" <expression>

| <expression> ">>" <expression>

| "!" <expression>

| "+" <expression>

| "-" <expression>

| "~" <expression>

| <INTEGER>

| <FLOAT>

| <IDENTIFIER>

| <BOOLEAN>

| <STRING_LITERAL>

<attempt_findout_block> ::= <attempt_block> <findout_block>

<attempt_block> ::= "ATTEMPT" "{" <statements> "}"

<findout_block> ::= "FINDOUT" <error_type> "{" <statements> "}"

<error_type> ::= "UNBOUNDLOCALERROR"

| "TYPEERROR"

| "VALUEERROR"

| "INDEXERROR"

| "KEYERROR"

| "EXCEPTION"

| "SYNTAXERROR"

| "STOPITERATION"

| "ARITHMETICERROR"

| "FLOATINGPOINTERROR"

| "OVERFLOWERROR"

| "ZERODIVISIONERROR"

| "ASSERTIONERROR"

| "ATTRIBUTEERROR"

| "BUFFERERROR"

| "EOFERROR"

| "IMPORTERROR"

| "MODULENOTFOUNERROR"

| "LOOKUPERROR"

| "MEMORYERROR"

| "NAMEERROR"

| "CONNECTIONERROR"

| "CONNECTIONABORTEDERROR"

| "CONNECTIONREFUSEDERROR"

| "CONNECTIONRESETERROR"

| "FILEEXISTERROR"

| "FILENOTFOUNERROR"

| "PERMISSIONERROR"

| "REFERENCEERROR"

| "RUNTIMEERROR"

| "WARNING"

<parameter> ::= <type> <IDENTIFIER>

<parameters> ::= <parameter> "," <parameters>

| <parameter>

| <empty>

<arguments> ::= <argument> "," <arguments>

| <argument>

| <empty>

<argument> ::= <IDENTIFIER>

| <expression>

<mutex> ::= "UNLOCK"

| "LOCK"

<type> ::= "INT"

| "FLOAT"

| "BOOL"

| "STRING"

<empty> ::=

<FUNCTIONID> ::= <UPPERCASE_LETTER> {<UPPERCASE_LETTER> | <DIGIT> | "_"}

<BOOLEAN> ::= "true" | "false"

<IDENTIFIER> ::= ["_"] <LETTER> {<LETTER> | <DIGIT> | "_"}

<FLOAT> ::= <DIGIT> {<DIGIT>} "." <DIGIT> {<DIGIT>}

<INTEGER> ::= <DIGIT> {<DIGIT>}

<STRING_LITERAL> ::= '"' {<IDENTIFIER>} '"'

<UPPERCASE_LETTER> ::= "A" | "B" | "C" | ... | "Z"

<LETTER> ::= <UPPERCASE_LETTER> | <LOWERCASE_LETTER>

<LOWERCASE_LETTER> ::= "a" | "b" | "c" | ... | "z"

<DIGIT> ::= "0" | "1" | "2" | ... | "9"

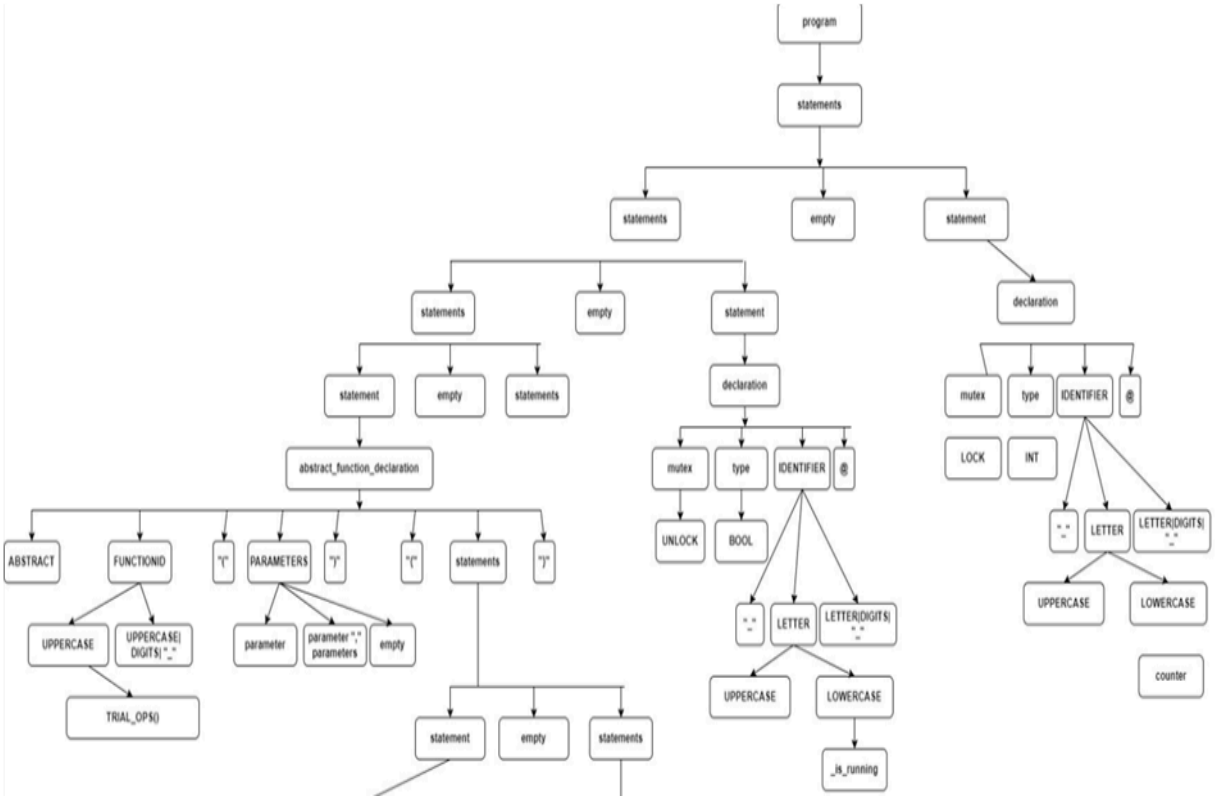
P.S. We added a lot of error types that the user could throw in preparations for more to come

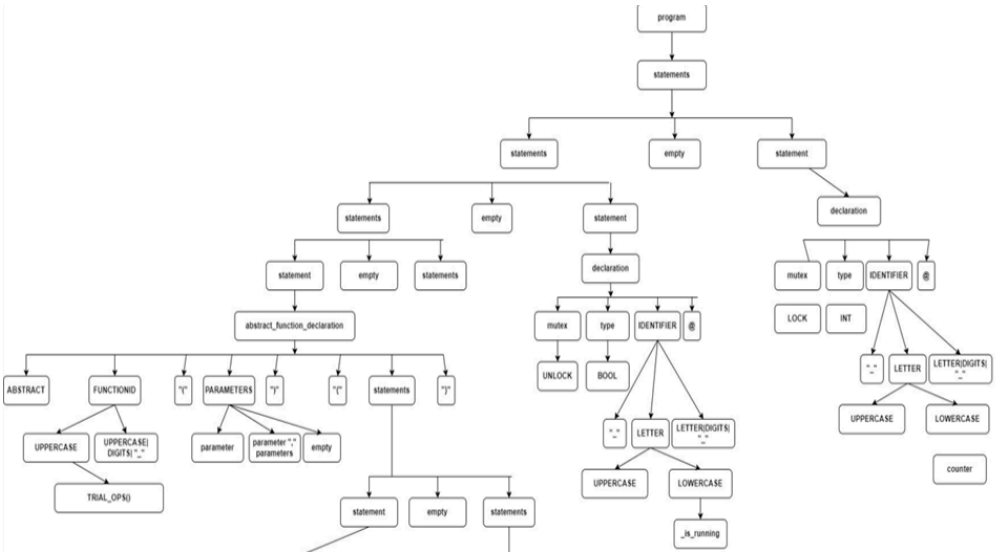
PARSE TREE

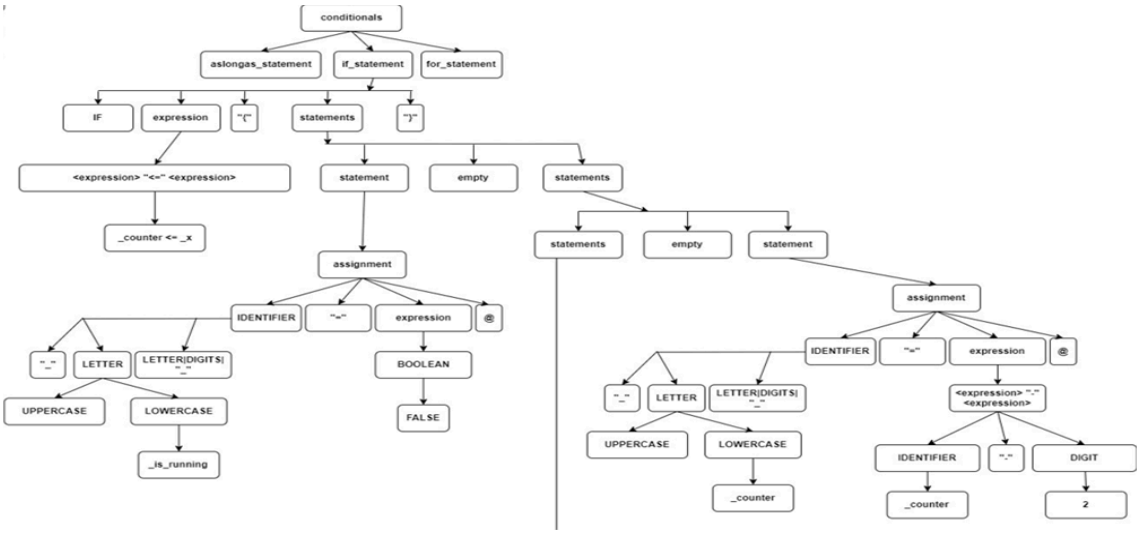
Sample program to derive the parse tree:

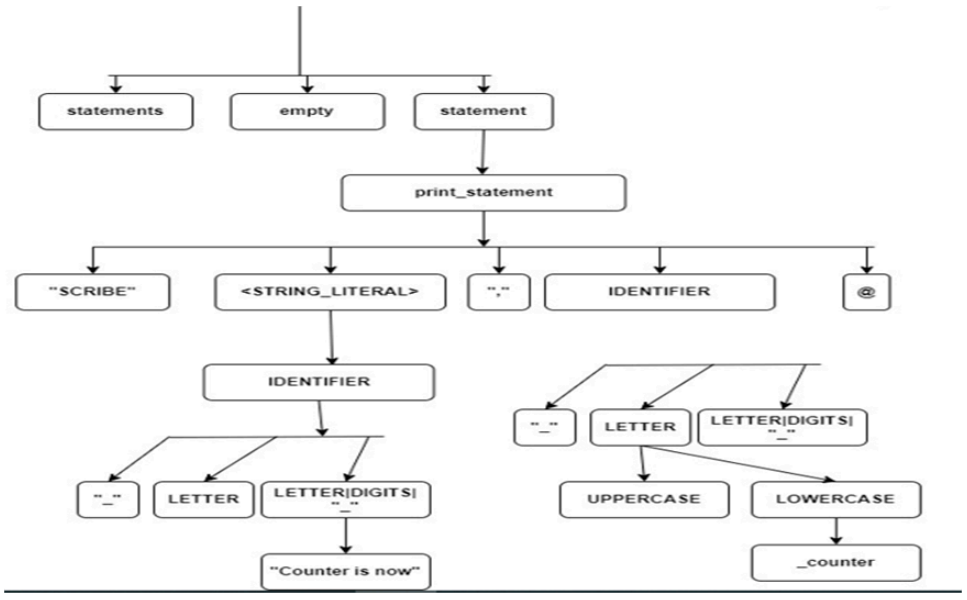
```
unlock int _Counter@
unlock bool _Is_Running = true@
abstract TRAIL_OPS(){
      _Counter = 21@
      lock int _X = 10@
      aslongas(_Is_Running){
            if(_Counter <= _X){
                  _Is_Running = false@
            }

            _Counter = _Counter - 2@
            scribe(“Counter is now”, _Counter)@
      }
}
```

The images below represent the sample code as a parse tree in snippets, however, the full parse tree can be viewed here.


program
statements
statements
empty
statement
statements
empty
statement
declaration
statement
empty
statements
declaration
mutex
type
IDENTIFIER
@
LOCK
INT
abstract_function_declaration
mutex
type
IDENTIFIER
@
"_"
LETTER
LETTER|DIGITS| "_"
ABSTRACT
FUNCTIONID
"("
PARAMETERS
")"
"{"
statements
"}"
UNLOCK
BOOL
UPPERCASE
LOWERCASE
UPPERCASE
UPPERCASE| DIGITS| "_"
parameter
parameter "," parameters
empty
"_"
LETTER
LETTER|DIGITS| "_"
counter
TRIAL_OPS()
statement
empty
statements
UPPERCASE
LOWERCASE
_is_running


program
statements
statements
empty
statement
statements
empty
statement
declaration
statement
empty
statements
declaration
mutex
type
IDENTIFIER
@
abstract_function_declaration
mutex
type
IDENTIFIER
@
LOCK
INT
ABSTRACT
FUNCTIONID
"("
PARAMETERS
")"
"{"
statements
"}"
UNLOCK
BOOL
"_"
LETTER
LETTER|DIGITS|"_"
"_"
LETTER
LETTER|DIGITS|"_"
UPPERCASE
LOWERCASE
counter
UPPERCASE
UPPERCASE|DIGITS|"_"
parameter
parameter","parameters
empty
UPPERCASE
LOWERCASE
_is_running
TRIAL_OPS()
statement
empty
statements


conditionals
aslongas_statement
if_statement
for_statement
IF
expression
"{"
statements
"}"
<expression> "<=" <expression>
statement
empty
statements
_counter <= _x
assignment
IDENTIFIER
"="
expression
@
"_"
LETTER
LETTER|DIGITS|"_"
BOOLEAN
UPPERCASE
LOWERCASE
FALSE
_is_running
statements
empty
statement
assignment
IDENTIFIER
"="
expression
@
"_"
LETTER
LETTER|DIGITS|"_"
<expression> "-" <expression>
UPPERCASE
LOWERCASE
IDENTIFIER
"-"
DIGIT
_counter
_counter
2


statements
empty
statement
print_statement
"SCRIBE"
<STRING_LITERAL>
","
IDENTIFIER
@
IDENTIFIER
"_"
LETTER
LETTER|DIGITS|"_"
"_"
LETTER
LETTER|DIGITS|"_"
"Counter is now"
UPPERCASE
LOWERCASE
_counter

TOKEN LIST

Lexeme	Tokens
abstract	ABSTRACT
hail	HAIL
scribe	SCRIBE
unlock	UNLOCK
if	IF
elif	ELIF
else	ELSE
for	FOR
aslongas	ASLONGAS
public	PUBLIC
private	PRIVATE
int	INT_TYPE
float	FLOAT_TYPE
bool	BOOL_TYPE
range	RANGE
in	IN
attempt	ATTEMPT
findout	FINDOUT
10	INTEGER
23.44	FLOAT
true or false	BOOLEAN
+	PLUS
-	MINUS
*	TIMES
/	DIVIDE
=	EQUAL
==	EQUIVALENT
!=	NOTEQUAL

>	GREATERTHAN
<	LESSTHAN
>=	GREATERTHANOREQUAL
<=	LESSTHANOREQUAL
(	LPAREN
)	RPAREN
{	LBRACE
}	RBRACE
@	LINEEND
,	COMMA
:	COLON
:	SEMICOLON
!	NOT
**	POWER
&	BITWISEAND
|	BITWISEOR
^	BITWISEXOR
~	BITWISEINVERT
<<	SHIFTLEFT
>>	SHIFTRIGHT
_DogNa_me	IDENTIFIER
ADDME	FUNCTIONID
“Hello”	STRING_LITERAL

REGULAR EXPRESSIONS

TOKEN	REGEX
INTEGER	‘\d+’
FLOAT	‘\d+\.\d+’
BOOLEAN	‘true | false’
EQUAL	‘=’
PLUS	‘\+’
LINEEND	‘@’
MINUS	‘-’
TIMES	‘*’
DIVIDE	‘/’
LESSTHAN	‘<’
GREATERTHAN	‘>’
LESSTHANOREQUAL	‘<=’
GREATERTHANOREQUAL	‘>=’
NOTEQUAL	‘!=’
EQUIVALENT	‘==’
SEMICOLON	‘;’
COLON	‘:’
COMMA	‘,’
LPAREN	‘\(’
RPAREN	‘\)‘
LBRACE	‘\{’
RBRACE	‘\}’
NOT	‘!’
POWER	‘**’
BITWISEAND	‘&’
BITWISEOR	‘|’
BITWISEXOR	‘\^’
SHIFTLEFT	‘<<’
SHIFTRIGHT	‘>>’

BITWISEINVERT	‘~’
IDENTIFIER	‘_?[a-zA-Z][a-zA-Z0-9_]*’
FUNCTIONID	‘[A-Z][A-Z_0-9]*’
VISIBILITY	‘private | public’
STRING_LITERAL	‘ "[^"\\]*" ’
KEYWORDS	‘(abstract|hail|scribe|unlock|lock|if|elif|else|for|while|public|private|int|float|bool|range|in|aslongas|attempt|findout|exception|stopiteration|arithmeticerror|floatingpointerror|overflowerror|zerodivisionerror|assertionerror|attributeerror|buffererror|eoferror|importerror|modulenotfounerror|lookuperror|indexerror|keyerror|memoryerror|nameerror|unboundlocalerror|connectionerror|connectionabortederror|connectionrefusederror|connectionreseterror|r|timeouterror|referenceerror|runtimeerror|syntaxerror|systemerror|typeerror|valueerror|warning)’

Revered words and their tokens

Reserved Word	**Token**
abstract	ABSTRACT
hail	HAIL
scribe	SCRIBE
unlock	UNLOCK
lock	LOCK
if	IF
elif	ELIF
then	THEN
else	ELSE
for	FOR
do	DO
while	WHILE
end	END

print	PRINT
contract	CONTRACT
public	PUBLIC
private	PRIVATE
internal	INTERNAL
external	EXTERNAL
return	RETURN
returns	RETURNS
emit	EMIT
event	EVENT
int	INT_TYPE
float	FLOAT_TYPE
bool	BOOL_TYPE
string	STRING_TYPE
var	VAR
range	RANGE
in	IN
aslongas	ASLONGAS
attempt	ATTEMPT
findout	FINDOUT
exception	EXCEPTION
stoptteration	STOPITERATION
arithmeticerror	ARITHMETICERROR
floatingpointerror	FLOATINGPOINTERROR
overflowerror	OVERFLOWERROR

zerodivisionerror	ZERODIVISIONERROR
assertionerror	ASSERTIONERROR
attributeerror	ATTRIBUTEERROR
buffererror	BUFFERERROR
eoferror	EOFERROR
importerror	IMPORTERROR
modulenotfounderror	MODULENOTFOUNERROR
lookuperror	LOOKUPERROR
indexerror	INDEXERROR
keyerror	KEYERROR
memoryerror	MEMORYERROR
nameerror	NAMEERROR
unboundlocalerror	UNBOUNDLOCALERROR
oserror	OSERROR
blockingioerror	BLOCKINGIOERROR
childprocesserror	CHILDPROCESSERROR
connectionerror	CONNECTIONERROR
brokenpipeerror	BROKENPIPEERROR
connectionabortederror	CONNECTIONABORTEDERROR
connectionrefusederror	CONNECTIONREFUSEDERROR
connectionreseterror	CONNECTIONRESETERROR
fileexistserror	FILEEXISTERROR
filenotfounderror	FILENOTFOUNERROR
interruptederror	INTERRUPTEDERROR
isadirectoryerror	ISADIRECTORYERROR

notadirectoryerror	NOTADIRECTORYERROR
permissionerror	PERMISSIONERROR
processlookuperror	PROCESSLOOKUPERROR
timeouterror	TIMEOUTERROR
referenceerror	REFERENCEERROR
runtimeerror	RUNTIMEERROR
syntaxerror	SYNTAXERROR
indentationerror	INDENTATIONERROR
taberror	TABERROR
systemerror	SYSTEMERROR
typeerror	TYPEERROR
valueerror	VALUEERROR
unicodeerror	UNICODEERROR
unicodeencodeerror	UNICODEENCODEERROR
unicodedecodeerror	UNICODEDECODEERROR
unicodetranslateerror	UNICODETRANSLATEERROR
warning	WARNING
userwarning	USERWARNING
deprecationwarning	DEPRECATIONWARNING
pendingdeprecationwarning	PENDINGDEPRECATIONWARNING
syntaxwarning	SYNTAXWARNING
runtimewarning	RUNTIMEWARNING
futurewarning	FUTUREWARNING
importwarning	IMPORTWARNING
unicodewarning	UNICODEWARNING

byteswarning	BYTESWARNING
resourcewarning	RESOURCEWARNING
keyboardinterrupt	KEYBOARDINTERRUPT

SCOPE & BINDING

Scope

In our TypeSnake language, we've adopted both Global and Block scoping for efficient variable management. We've chosen these scopes to mitigate the risk of name collisions, ensuring that variables declared within specific blocks remain isolated from each other. This means that a variable can be declared multiple times within different blocks without causing any interference. Global scope facilitates seamless data sharing across the entire program, simplifying communication between different parts of the codebase. Below is a code snippet showcasing the usage of both global and local scopes within our TypeSnake language.

```
lock string Global_var = "This is a global variable. "@
attempt {
      scribe ("Trying to access _Global_var in global its scope:",
_Global_var) @
      lock bool Inside Attempt = true@
      scribe ("This is Inside_Attempt", Inside_Attempt) @
}
findout unboundlocalerror{
      scribe("UnboundLocalError _Global_var is defined in this
scope. ")@
scribe ("This is _Inside_Attempt", _Inside_Attempt) @
```

Binding

TypeSnake employs static binding, enabling the compiler to resolve variable references during compilation. This results in the determination of the variable to be accessed based on their declared types at compile time, rather than at runtime. Within TypeSnake, all data types are statically bound to their variables, and all variables set as “lock” are statically bound to their values. The below code snippet illustrates such:

```
# int, string, float, and bool are binding to the variables
unlock int _COUNT = (0 + 2 + 3)@
lock string _Global_var = "This is a global variable."@
lock int _TEN = 10@
lock float _PI = 3.14@
unlock bool _BINARY = true@
```

CHARACTERISTICS OF GOOD PROGRAMMING LANGUAGE

TypeSnake, our modern programming language, embodies simplicity, clarity in syntax, robust data types, comprehensive exception handling, and effective type-checking capabilities. These attributes are meticulously designed to elevate readability, writability, and reliability, fostering a seamless programming experience for developers.

Simplicity:

A hallmark of a good programming language is simplicity. TypeSnake achieves this by employing a concise set of well-defined rules and constructs, making it easy to learn, write, read, and maintain code. For instance, the "lock" keyword signifies a variable declaration, emphasizing immutability, and enhancing code clarity.

Syntax Design:

TypeSnake boasts clear and consistent syntax, ensuring readability, writability, and reliability. The use of the "@" symbol to denote statement termination aids readability by delineating the boundaries of each statement. Additionally, meaningful keywords such as "attempt," "findout," and "unboundlocalerror" contribute to self-explanatory code.

Data Types:

Our language offers a diverse range of data types, explicitly defined by the "p_type" rule, enhancing readability and minimizing typographical errors in variable declarations. The extensibility of the "p_type" rule facilitates easy integration of new data types, further bolstering writability.

Exception Handling:

TypeSnake excels in comprehensive exception handling, addressing both syntax and semantic errors. Through functions like "p_error" for syntax error management and

"type_checking" for semantic type verification, developers benefit from clear error messages, aiding in quick issue detection and resolution.

Type Checking:

TypeSnake demonstrates robust type-checking capabilities through the "type_checking" semantic rule. Verifying variable types against expected values ensures correctness and prevents unintended type mismatches, thereby enhancing program reliability.

Overall, TypeSnake's emphasis on simplicity, clear syntax design, robust data types, comprehensive exception handling, and effective type checking collectively contribute to a superior programming experience, empowering developers to write clean, reliable code with confidence.

PROJECT REPORT

Member	Contribution
Sharethia McCarthy 2000191	● BNF Grammar ● Regular Expressions for tokens ● List of tokens ● Characteristics ● Programming language used ● Lexer and parser
Hugh Scott 1908850	● BNF Grammar ● Regular Expressions for tokens ● List of tokens ● Semantics & Code Gen ● Parser
Barrington Patterson 2008034	● Classification of language (procedural, general purpose, and high level) ● Characteristics ● List of tokens ● Front-End
Christina Wilson- 1903419	● List of tokens ● Parse tree ● Characteristics vs Criteria ● Scope and Binding ● UI

USER MANUAL

For direct access to the TypeSnake platform, click https://apl-cit-4004-web-ui.vercel.app/.

Click User Manual