Faculty of Engineering and Computing

School of Computing and Information Technology

University of Technology, Jamaica

Demali Gregg, Nave Bent, Javel Morgan

CIT4004: Analysis of Programming Languages

Tutor: David White

Table of Contents

Background and Description of Language

ChatCode

ChatCode embraces multiple paradigms, including object-oriented, procedural, and functional programming. It incorporates object-oriented principles through its smart contract feature, procedural aspects by utilizing functions and arithmetic within its contracts, and also includes support for functional programming.

Domain-Specific vs. General-Purpose:

ChatCode is **domain-specific**. While it provides general programming constructs, its unique feature of smart contract creation and interaction makes it particularly tailored for blockchain-related tasks.

ChatCode is designed to be simple and intuitive, but depending on the user, it can become more complex. Features like easy arithmetic expressions, control structures, and smart contract creation make it user-friendly and more in line with **high-level** languages.

Grammar

```
<program> ::= { <statement> | <contract_declaration> }

<statement> ::= <assignment> | <if_statement> | <loop> | <display> | <return_statement>

|<event_emit> | function_call “;” | <deploy_statement>

<assignment> ::= ["var"] <identifier> "=" <expression> ";"

<if_statement> ::= "if" "(" <condition> ")" "then" <block> ["else" <block>] "end"

<loop> ::= <for_loop> | <while_loop>

<for_loop> ::= "for" "(" <assignment> <condition> ";" <assignment> ")" "do" <block> "end"

<while_loop> ::= "while" "(" <condition> ")" "do" <block> "end"

<block> ::= “{“ {statement} “}”

<display> ::= "print" "(" <expression> ")" ";"

<condition> ::= <expression> <comparator> <expression>

<expression> ::= <term> { <operator> <term>} | <function_call> | <int>| <string> |

<bool>|<identifier>

<address_literal> ::= "0x" <hex_digit> {40} |  <hex_digit> {64}

<hexadecimal_digit> ::= "0" | "1" | ... | "9" | "a" | "b" | ... | "f" | "A" | "B" | ... | "F"

<term> ::=  <factor> {<operator> <factor>}

<factor> ::= <number> | "(" <expression> ")" | <identifier> | <function_call

<return_statement> ::= "return" <expression> ";"

<identifier> ::= <letter> {<letter> | <digit>}

<comparator> ::= "<" | ">" | "==" | "!=" | "<=" | ">="

<number> ::= <digit> {<digit>}
```

```
<digit> ::= "0" | "1" | ... | "9"

<letter> ::= "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"

<character> ::= <letter> | <digit> | "$" | "!" | ... | "~"

<contract_declaration> ::= "contract" <identifier> "{" { <contract_body> } "}"

<contract_body> ::= {<statement> | <contract_statement>}

<contract_statement> ::= <state_declaration> | <function_declaration> | <event_declaration>|

<event_emit> | initialize_function

<state_declaration> ::= "var" <identifier> <type_specifier> ["=" <expression>] ";"

<function_declaration> ::= <visibility> "function" <identifier> "(" <params> ")" ["returns" "("

<type_specifier> ")"] "{" { <function_body> } "}"

<initialize_function>::= “public” “function” “initailize” "(" <params> ")" ["returns" "("

<type_specifier> ")"] "{" { <function_body> } "}"

<function_call>::= <qualified_identifier> “(” <expression> [{“,” <expression>}] “;”

qualified_identifier::= <identifier>{“.” <identifier>}

<function_body> ::= {<statement> | <event_emit>}

<params> ::= <identifier> <type_specifier> { "," <identifier> <type_specifier> }

<visibility> ::= "public" | "private"

<deploy_statement>::= “deploy” <function_call> “;”

<event_declaration> ::= "event" <identifier> "(" { params> } ")" ";"

<event_emit> ::= "emit" <identifier> "(" [<expression> { "," <expression> }] ")" ";"

<type_specifier> ::= "int" | "string" | "bool" | "address"
```

Sample Code

```
// This is how you comment in ChatCode
# This is another way to comment
x = 5;
y = 10;
sum = x + y;
if (sum > 10) then {
   print("Sum is greater than 10");
} else {
   print("Sum is not greater than 10");
}end
for (i = 0; i < 5; i = i + 1;) do {
   print(i);
}end
contract MathOperations {
   var result int;
  public function add(int a, int b) returns (int) {
      result = a + b;
      return result;
   }
   MathOperation.add(3,6);
```

Parse Tree for Sample Code

```
<program>
|
|-- <statement> (Assignment: x = 5)
|
|-- <statement> (Assignment: y = 10)
|
|-- <statement> (Assignment: sum = x + y)
|
|-- <statement> (If-Else statement)
|   |
|   |-- <condition> (sum > 10)
|   |
|   |-- <block> (True block)
|   |   |
|   |   |-- <statement> (Display: print("Sum is greater than 10"))
|   |
|   |-- <block> (False block)
|       |
|       |-- <statement> (Display: print("Sum is not greater than 10"))
|
|-- <statement> (For loop)
|   |
|   |-- <condition> (i < 5)
|   |
|   |-- <block> (Loop body)
|       |
|       |-- <statement> (Display: print(i))
|
|-- <contract_declaration> (Contract: MathOperations)
    |
    |-- <contract_statement> (State declaration: var result)
    |
    |-- <contract_statement> (Function declaration: add)
        |
        |-- <params> (int a, int b)
        |
        |-- <block> (Function body)
            |
            |-- <statement> (Assignment: result = a + b)
            |
            |-- <statement> (Return: return result)
```

Full list of Tokens in ChatCode

Reserved Keywords

Control structures: if, then, else, for, do, while, print, end

Contract-related: contract, function, public, private, return, returns, emit, event, deploy

Data types: int, string, bool, address

Keywords

Not reserved, but has a special meaning: var

Operators

Arithmetic: =, +, -, *, /

Relational: <, >, <=, >=, !=, ==

Delimiters

Semicolon and comma: ; ,

Parentheses: (,)

Curly braces: {, }

Literals

Integer literals: Examples - 123, 5, 0

String literals: Examples - "Hello", "ChatCode"

Boolean literals: true, false

Identifiers

Names used for variables, functions, contracts, etc. They follow a specific pattern.

Comments Indicators: //, #

Special Characters Miscellaneous symbols: !, @, $

Regular Expressions used to recognise the Tokens in ChatCode

Reserved Keywords: `^(if|then|else|for|do|while| end |contract|function|deploy|public| private|return|returns|emit|event|int| string|bool|address|print)$`

Keyword: `^var$`

Arithmetic: `^(\+|-|\*|/|==|!=|<|>|<=|>=)$`

Delimiters: `^(;|,|\(|\)|\{|\}|\[|\])$`

Integer literals: `^\d+$`

String literals: `^"([^"\\]|\\.)*"$`

Boolean literals: `^(true|false)$`

Identifiers: Names used for variables, functions, contracts, etc.: `^[A-Za-z]\w*$`

Comment Indicators: `^(//|#).*`

Special Characters: `(!|@|\$)$`

Scope and Binding in ChatCode

```
contract MyContract {

  var x = 0;
  var y = 0;

  public function setValues() {

    if(x < 10) then {
      var x = 5;
      var z = 10;
    }end

    while(y < 5)do {
      var w = y + 1;
      y = w;
    }end

    print(x); // 0
    print(y); // 5
    print(z); // Error, z not in scope

  }

  public function initialize() {
    setValues();
  }

}
```

Details on the programming language used to create Compiler

The language used in the development of Chatcode was Python, which is an interpreted high-level general-purpose programming language that supports features such as object-oriented programming, dynamic binding, and a large number of internal and external libraries and packages. Python was selected to carry out the development of ChatCode due to its simplicity, writability, readability, and overall ease of use.

Characteristics of ChatCode

When evaluating the qualities of a programming language, consider readability, writability, and reliability. Readability refers to how easy code is to understand, writability to how simple writing code is, and reliability to minimizing unexpected behavior. In regard to readability, ChatCode provides intuitive syntax, appropriate use of control structures, and strong type declarations to facilitate understandability. To promote writability, ChatCode builds on common programming languages, ensuring familiarity while adding domain-specific syntax tailored for blockchain development. Simpler code authoring translates to faster feature development. Finally, for reliability, ChatCode employs strict type checking, encapsulation mechanisms, and visibility modifiers, limiting unanticipated issues. These provide guardrails, preventing classes of errors upfront. Moreover, the compiler integrates directly with a language model to further verify logic. By focusing on readability, writability, and reliability in its design, ChatCode exemplifies multiple hallmarks of good programming languages critical for productive software development.