Spring 2023

SpartanScript: New Language Design for Smart Contracts

Ajinkya Lakade
San Jose State University

SpartanScript: New Language Design for Smart Contracts

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Ajinkya Lakade

May 2023

The Designated Project Committee Approves the Project Titled

SpartanScript: New Language Design for Smart Contracts

by

Ajinkya Lakade

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Prof. Thomas Austin	Department of Computer Science
Prof. Ben Reed	Department of Computer Science
Prof. Katerina Potika	Department of Computer Science

ABSTRACT

SpartanScript: New Language Design for Smart Contracts

by Ajinkya Lakade

Smart contracts have become a crucial element for developing decentralized applications on blockchain, resulting in numerous innovative projects on blockchain networks. Ethereum has played a significant role in this space by providing a high-performance Ethereum virtual machine, enabling the creation of several high-level programming languages that can run on the Ethereum blockchain. Despite its usefulness, the Ethereum Virtual Machine has been prone to security vulnerabilities that can result in developers succumbing to common pitfalls which are otherwise safeguarded by modern virtual machines used in programming languages. The project aims to introduce a new interpreted scripting programming language that closely resembles the Scheme programming language. This language is designed to run natively on Spartan Gold, which is an experimental blockchain platform that facilitates easy experimentation in a blockchain environment.

To demonstrate the usefulness of SpartanScript, the project includes the implementation of various smart contracts, such as time-sensitive smart contracts and an implementation of the ERC-20 standard. The ERC-20 standard is a crucial component of the blockchain ecosystem because it provides a standardized set of rules for creating and managing tokens on the Ethereum blockchain, enabling interoperability and easy implementation of new tokens to represent a wide range of assets, such as currencies, commodities, or even other cryptocurrencies

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Prof. Thomas Austin for his invaluable guidance, encouragement, and support throughout this project. Through working with him, I have gained valuable insights into the importance of questioning established ideas and conducting experiments to uncover the truth. He has provided me with invaluable support and guidance, enabling me to enhance my research skills, managing project while ensuring consistent progress and delivery.

I would like to extend my heartfelt thanks to the members of my defense committee, including Prof. Ben Reed and Prof. Katerina Potika, as well as all the faculty members in the Computer Science department, who have provided me with unwavering support over the course of my degree.

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

Introduction

Blockchain is becoming a prevalent underlying technology providing secure and distributed data sources using peer-to-peer network in many domain industries. Cryptocurrency is one of the most popular and earliest industries in which blockchains are being used. Cryptocurrencies are digital currencies that are used as an alternative form of payment based on blockchain technology. Due to its decentralized nature, immutability and transparency, and encryption, blockchain provide a modern way of handling transactions using digital currencies while making them more secure. Satoshi Nakamoto [1] introduced Bitcoin, the first cryptocurrency, in 2009, which has paved the way for numerous advancements in the blockchain industry.

On top of these digital currencies, blockchains can also be used to program smart contracts to store and run the business logic of certain agreements as required. These smart contracts require programming constructs that are different from traditional programming languages to be able to run on the blockchain [2]. These smart contracts require careful checks before deploying on blockchain because of their immutable nature making it challenging to verify their correctness [3]. There are many cryptocurrencies and programming languages designed to serve such smart contracts. One of the most popular cryptocurrencies that serve smart contracts is Ethereum [4].

Ethereum is an open-source, decentralized blockchain platform that was launched in 2015. It is designed to enable developers to build decentralized applications (dApps) that can execute complex code and smart contracts on a decentralized network. Ethereum is designed to run on its own Ethereum virtual machine (EVM) that provides users to develop smart contracts in a high-level programming language. The smart contracts are developed in the high-level programming language and are compiled into bytecode that can be executed on the EVM. The EVM provides a

Turing-complete virtual machine that can execute arbitrary code. Therefore, EVM has become a powerful tool to run many decentralised applications on top of the Ethereum network. However, the Ethereum virtual machine has been subjected to various vulnerabilities like the reentrancy attack, immutable bugs, and many more [5].

Using a scripting language can address some of these security concerns by simplifying the development process for smart contracts. Smart contract programming is a complex task that requires specialized knowledge of blockchain technology and cryptography. With a scripting language, developers can write smart contracts in a more familiar syntax, similar to that of traditional programming languages like Python or JavaScript, making it easier to write secure code.

Furthermore, scripting languages can reduce the computational complexity of running smart contracts on the blockchain, leading to more efficient and scalable decentralized applications. This is because scripting languages can eliminate the need for a virtual machine like the EVM, which requires a significant amount of computational resources to execute arbitrary code. By using a scripting language instead, smart contracts can be executed directly on the blockchain, reducing the computational overhead and improving the overall performance of the application.

Overall, using a scripting language can simplify the development process for smart contracts, reduce the potential for vulnerabilities and attacks, and improve the scalability and efficiency of decentralized applications. This project proposes a new programming language design that would run without the virtual machine to address some of these security concerns. This project aims to create a programming language like scheme to be able to directly run on blockchain and address these security concerns.

CHAPTER 2

Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) was first proposed by Vitalik Buterin in the Ethereum white paper published in 2013 [6]. The EVM is a runtime environment that allows smart contracts to be executed on the Ethereum blockchain. It is a key component of the Ethereum platform and is responsible for executing smart contract code. The EVM is also responsible for handling transactions and managing the state of the Ethereum network. When a user sends a transaction to the network, the EVM executes the associated smart contract code and updates the state of the network accordingly. The EVM plays a critical role in the operation of the Ethereum network, enabling developers to build decentralized applications that can execute complex logic and manage assets in a trustless and secure manner. It is also a key component of the Ethereum platform and has enabled the development of a wide range of decentralized applications, including decentralized finance (DeFi) protocols, non-fungible token (NFT) marketplaces, and more.

The EVM is a quasi-turing complete state machine based on stack architecture [7]. For simple value transfer transactions between Ethereum Externally Owned Accounts (EOAs), the EVM is not required, as these transactions do not involve state updates that require computation by the EVM. However, for any other transactions that require state updates, the EVM will be involved in computing and verifying these updates. The smart contract programming languages such as LLL, Serpent, Mutan, or Solidity [8] are compiled into the bytecode instruction set of the Ethereum Virtual Machine (EVM). The EVM then executes this bytecode instruction set to execute the smart contract on the Ethereum blockchain. This allows developers to write smart contracts in a high-level language and have it compiled into a low-level bytecode instruction set that can be executed by the EVM. This is similar to the execution of

other high-level programming languages like Java, Scala which use the JVM, or C# which uses .NET.

The instruction set of the Ethereum Virtual Machine (EVM) includes a wide range of operations that cover various aspects of smart contract execution. These operations include arithmetic and bitwise logic operations, stack, memory, and storage access, execution context inquiries, control flow operations, logging, calling, and other operators. In addition to these typical bytecode operations, the EVM also provides access to account and block information. Account information such as the address and balance of an account can be accessed and manipulated by smart contracts during execution. Block information such as the current block number and gas price, can be used to provide context and make decisions during smart contract execution. This access to the account and block information provides additional flexibility and functionality to smart contracts on the Ethereum blockchain. The entire list of such opcodes is provided by Ethereum at their official website [9].

The EVM is responsible for updating the Ethereum state by computing valid state transitions resulting from the execution of smart contract code, as outlined by the Ethereum protocol. Ethereum is regarded as a transaction-based state machine, where external actors such as account holders and miners initiate state transitions by creating, accepting, and ordering transactions. Each Ethereum address corresponds to an account that comprises an ether balance, stored as the number of wei owned by the account, a nonce that reflects the number of transactions successfully sent from an externally owned account (EOA) or the number of contracts created if it is a contract account, the account's storage that acts as a permanent data store exclusively used by smart contracts, and the account's program code that is only present if the account is a smart contract account. EOA will always have no code and empty storage. This hierarchical structure of the Ethereum state enables the EVM to facilitate the creation,

modification, and execution of smart contracts and transactions within the Ethereum ecosystem.

In Ethereum, gas is the unit of measurement for computational and storage resources required to perform actions on the blockchain. Ethereum considers every computational step performed by transactions and smart contracts. The gas serves a dual role in Ethereum: as a buffer between the volatile price of Ethereum and the reward for miners, and as a defence against denial-of-service attacks. The gas system helps to prevent accidental or malicious infinite loops or computational waste on the network. To do this, the initiator of each transaction must specify a limit to the amount of computation they are willing to pay for. This disincentivizes attackers from sending spam transactions as they must pay for the computational, bandwidth, and storage resources that they consume. Gas is a critical component of Ethereum that helps to maintain the network's security and efficiency by ensuring that users pay for the resources they consume [10].

The Ethereum Virtual Machine (EVM) is a powerful tool that enables the execution of arbitrary code on the Ethereum network. However, the EVM has been subjected to various vulnerabilities, including reentrancy attacks and immutable bugs. A reentrancy attack is when a contract calls another contract before the previous call has completed, allowing the attacker to execute malicious code and steal funds from the contract. In June 2016, a security flaw in the code of the DAO (Decentralized Autonomous Organization) was exploited to steal over 2 million ether worth around 40 million USD at the time [11]. The attackers took advantage of a reentrancy issue in the "splitDAO" function of the code. Due to insufficient attention to detail in the program's design, a call to the function that should have behaved as a regular call was manipulated into a recursive call, which allowed the attackers to make multiple unauthorized withdrawals instead of just one. Other vulnerabilities like immutable

bugs occur when a bug is discovered in a smart contract after it has been deployed to the blockchain, and the contract cannot be modified or fixed due to the immutability of the blockchain.

CHAPTER 3

The SpartanScript Language

Scheme is a functional programming language that was developed at the Massachusetts Institute of Technology (MIT) in the 1970s. It is a dialect of the Lisp programming language and is known for its minimalist syntax and powerful abstractions. Scheme was developed after various iterations and research provided in what was called the lambda papers. One of the early ideas of Scheme was to adopt lambda calculus in Lisp programming language [12].

Scheme is a dynamically typed language, which means that variable types are determined at runtime rather than explicitly declared. It also supports first-class functions, which means that functions can be passed as arguments to other functions and returned as values from functions. This allows for the creation of complex and reusable abstractions. Scheme is also known for its simplicity and elegance. The language's minimalist syntax and powerful abstractions make it a great choice for exploring fundamental programming concepts such as recursion, functional programming, and data structures.

This chapter showcases the proposed implementation of a SpartanScript interpreter for the practical application of scripting languages in the context of blockchain development. The primary aim of this interpreter is to develop self-executing and secure smart contracts. The formal syntax of the proposed language is shown in 1.

This proposed interpreter comprises three critical processes: tokenization, parsing, and evaluation. The first process involves breaking down the source code into tokens. Tokens are the smallest individual units of code that have meaning in the programming language, and they can be identifiers, operators, literals, or other elements of the language. This step is essential as it enables the interpreter to identify the distinct elements of the code and prepare them for further processing.

e ::=		*Expressions*
	x	variables/addresses
	v	values
	$(define\ x\ e)$	assignment
	$(lambda\ x\ e)$	function
	$(op\ e\ e)$	binary operations
	$(if\ e\ e\ e)$	conditional expressions
	$(require\ e\ e)$	conditional expressions with fail
	$(begin\ e)$	sequencial expressions
	$(getMap\ e\ e)$	get from map
	$(setMap\ e\ e)$	set in map
	$(hasMap\ e\ e)$	has in map
	$(deleteMap\ e\ e)$	delete from map
	$(\$balance\ e)$	balance of smart contract
	$(\$transfer\ e\ e)$	transfer to smart contract
	$(defineState\ e\ e)$	blockchain state variable
v ::=		*Values*
	n	natural numbers
	b	boolean values
	$makeMap$	new map
	$\$me$	address of smart contract
	$\$sender$	address of smart contract caller
	$\$timestamp$	time of block creation
op ::=		*Operators*
	$+ \mid - \mid * \mid / \mid \%$	arithmetic operators
	$== \mid >= \mid > \mid < \mid <=$	logical operators

Figure 1: The formal syntax for SpartanScript language

The next process involves analyzing the syntax of the tokens and constructing a parse tree that represents the structure of the program in terms of the language's grammar rules. The parse tree is used to ensure that the program follows the syntax rules of the language. This step is critical as it helps to identify any syntax errors that may cause the program to malfunction.

Finally, the interpreter evaluates and executes the program line by line. This step involves interpreting the meaning of the code and carrying out the necessary

actions to execute it correctly. The interpreter must be able to execute the program accurately, as any errors may compromise the security and functionality of the smart contract.

3.1 Tokenization

Tokenization is the process of breaking down a sequence of characters or a string of code into smaller, meaningful units called tokens. A token is a basic building block of the language and can include identifiers, keywords, operators, symbols, and literals. The proposed interpreter tokenizes proposed scheme program using the function below-

```
tokenize(contents) {
   let lines = contents.trim().split("\n");
   let tokens = [];
   lines.forEach((ln) => {
     ln = ln.replaceAll("(", " ( ").replaceAll(")", " ) ");
     ln = ln.replace(/;.*/, "");

     tokens.push(...ln.split(/\s+/).filter((s) => s.length !== 0));
   });
   return tokens;
}
```

The function first removes any leading or trailing whitespace from the input code using the `trim()` method, and then splits the code into separate lines using the newline character `\n` as a delimiter. For each of the line, the function replaces all instances of open and close parentheses with whitespace on either side, effectively splitting them into separate tokens. This ensures that parentheses are treated as separate tokens, rather than part of another token like a function name or argument. Next, the function removes any comments from the line by replacing everything from a semicolon ";" to the end of the line with an empty string. This allows the interpreter to ignore comments when tokenizing the code. Finally, the function splits the line into separate tokens using a regular expression that matches one or more whitespace

characters while also filtering the unnecessary whitespace tokens.

The function returns the tokens array, which contains all the individual tokens extracted from the input code. This tokenization process is a necessary step for interpreting or compiling the code, as it breaks down the code into individual units that can be analyzed and executed by the computer.

3.2 Parser

A parser analyzes the structure of a given input and determines its grammatical structure and meaning based on a set of predefined rules. The input is typically a sequence of symbols or tokens that are parsed and transformed into a data structure, such as a tree or a graph, that can be further analyzed or used for various purposes. A parser is used to analyze source code and convert it into an abstract syntax tree (AST) that represents the code's structure and meaning. This allows the code to be analyzed and transformed by other tools, such as compilers or interpreters, that operate on the AST. To parse the tokenized program, the interpreter uses the below function -

```
parse(tokens) {
   let ast = { children: [] };
   for (let i = 0; i < tokens.length; i++) {
     let tok = tokens[i];
     if (tok === "(") {
       let newAst = { parent: ast, type: LIST, children: [] };
       ast.children.push(newAst);
       ast = newAst;
     } else if (tok === ")") {
       ast = ast.parent;
     } else if (tok.match(/^\d+$/)) {
       ast.children.push({ type: NUM, value: parseInt(tok) });
     } else if (tok.match(/^\w+$/)) {
       ast.children.push({ type: VAR, value: tok });
     } else {
       ast.children.push({ type: OP, value: tok });
     }
   }
   return ast.children;
```

```
}
```

The `parse()` function takes an array of tokens as input and returns an AST representing the structure of the program. This function iterates over the token list supplied by tokenizer. The function checks whether it is an open or close parenthesis, a number, a variable, or an operator for each token. The function creates four types of AST using constants as shown below

```
const LIST = 1;
const OP = 2;
const NUM = 3;
const VAR = 4;
```

The AST created represents a tree-like structure in an object of either of the types given above. For a scheme code `(define getgold (lambda (amt dest) ($transfer amt dest)))`, following AST is generated.

```
{"type":1,"children":
    [
        {"type":4,"value":"define"},
        {"type":4,"value":"getgold"},
        {"type":1,"children":
            [
                {"type":4,"value":"lambda"},
                {"type":1,"children":
                    [
                        {"type":4,"value":"amt"},
                        {"type":4,"value":"dest"}]},
                {"type":1,"children":
                    [
                        {"type":4,"value":"$transfer"},
                        {"type":4,"value":"amt"},
                        {"type":4,"value":"dest"}]}]}]}
```

The contents within the parentheses that are objects categorized as type 1 represents a `LIST`. Other objects labeled as `define` and `getgold` are of the `VAR` type,

indicating that they are keywords or function names represented as strings. These objects are arranged in a tree-like structure, using their object type and the children field. Such ASTs provide a structured representation of the source code that can be manipulated programmatically and are used in interpreters to transform the source code into executable code.

3.3 Evaluation

In an interpreter, the evaluation process typically involves parsing the source code of a program and executing it line-by-line or statement-by-statement. When a statement is encountered, the interpreter will execute the statement and then move on to the next one. If an error is encountered during the evaluation process, the interpreter will typically stop executing and report the error to the user.

```
evaluate(ast, env) {
   if (ast.type == NUM) {
     return ast.value;
   } else if (ast.value == "$me") {
     return this.$me;
   } else if (ast.value == "$sender") {
     return this.sender;
   } else if (ast.value == "$timestamp") {
     return this.contractStateVariables.get("$timestamp");
   } else if (ast.value == "makeMap") {
     return new Map();
   } else if (ast.value == "#t") {
     return ast.value;
   } else if (ast.value == "#f") {
     return ast.value;
   } else if (ast.type == VAR) {
     return this.hasVariable(env, ast.value);
   }


   let first = ast.children[0];
   let second = ast.children[1];
   let third = ast.children[2];
   let rest = ast.children.slice(2);
```

```
    ...

    switch (first.value) {
      case "+":
       return (
         rest.reduce((x, y) => x + this.evaluate(y, env), 0) +
         this.evaluate(second, env)
       );
      ...
    }
    ...
}
```

The above code provides the implementation of a part of the evaluation function which interprets and executes a certain code of the AST generated by the Parser. The `evaluate` function returns evaluation based on the type of object in AST defined by the parser. For `NUM` and `VAR` type of object in AST, the value is directly returned from the variable space set for the programming language at the time of initialization which also involves some of the blockchain primitives. For other types, all the parameters in the object are collected from the children array in the object of the AST and evaluated seperately. The above code shows one such example of evaluation rule for `+` operator.

In the following section, evaluation rules for various primitives are explained. These primitives include fundamental operations that are commonly used in other programming languages as well as primitives unique to blockchain, which facilitate effective and efficient development of smart contracts.

3.3.1 Basic Primitives

The section below describes the various primitives and their evaluation rules when encountered in the AST that are typically provided in other programming language.

3.3.1.1 Basic Operations

The following code provides the evaluation for basic arithmetic operations such as addition, subtraction, multiplication and division primitive.

```
case "+":
    return (
        rest.reduce((x, y) => x + this.evaluate(y, env), 0) +
        this.evaluate(second, env)
    );
case "-":
    return this.evaluate(third, env) - this.evaluate(second, env);
case "*":
    return (
        rest.reduce((x, y) => x + this.evaluate(y, env), 0) +
        this.evaluate(second, env)
    );
case "/":
    return this.evaluate(third, env) / this.evaluate(second, env);
```

The case for addition code returns evaluation of second expression in the list which would be first parameter after + operator and sum of all of the evaluation of each of the rest of the expressions. The case for subtraction code returns the subtraction of evaluation of second expression in the list which would be first parameter after - operator and sum of all of the evaluation of each of the rest of the expression. Similarly, the multiplication and division cases have same evaluation rule as the addition and subtraction respectively. The addition and multiplication primitives can accept any number of parameters, while the subtraction and division primitive accept only two parameters. The following script shows examples of usage of these primitives

```
(+ 4 5)
(+ (+ 4 5) 6 7)
(- 4 5)
(- (- 4 5) 6)
(* 4 5)
(* (* 4 5) 2 3)
(/ 4 20)
```

```
(/ (/ 4 20) 100)
```

In the provided code, the first line computes the value 9. Similarly, the second line computes the value 22 which is equivalent to (+ 9 6 7) after the evaluation of the first parameter. The third line computes the value 1, while the fourth line computes the value 5, which is equivalent to (- 1 6) after the evaluation of the first parameter.

3.3.1.2 Lambda

Lambda is a special form of primitive used to create anonymous functions in Scheme. It allows to define a function without giving it a name. The code below shows the evaluation containing lambda expression

```
case "lambda":
    let params =
        second.children.map(val =>
            this.evaluate(val, env));
    return
        new FunctionDef(params,
            third, new ScopingEnvironment(env));
```

The above evalution rule creates an array called `params`, which is used to store the parameter names of the lambda function by evaluating all the expression in second element of the AST. This `params` array contains evaluated parameters described for the lambda function. The code then creates a new `FunctionDef` object with the parameter list, the body of the lambda which is the third element of the parsed expression, and a new `ScopingEnvironment` object that inherits from the current environment `env`. This creates a new lambda function that is bound to the current scope.

The `FunctionDef` class is used to define a structure for a function in this proposed language. This class defines parameters that are required for the function, the AST of the function body to be evaluated and executed and the current environment of the

function defining the correct scope of the function. The `FunctionDef` class is defined as shown below

```
class FunctionDef {
  constructor(params, body, env) {
    this.params = params;
    this.body = body;
    this.env = env;
  }
}
```

The following code shows an example of lambda function. In this example, a lambda function that takes two parameter `a` and `b` and the definition of the function is a multiplication of these two parameters.

```
(lambda (a b) (* a b))
```

3.3.1.3 Define

`Define` is a special form of primitive used to bind a name to a value or expression. It is commonly used to define variables and functions. The following code shows the evaluation rules for `define` construct

```
case "define":
    env.varMap.set(second.value, this.evaluate(third, env));
    break;
```

This code assigns the evaluated expression provided in the third expression of the AST to the variable name provided in the second expression of the AST. This variable to evaluated expression mapping is stored in the `varMap` set of variables defines for the current scope of evaluation. The code below shows examples of scheme code using define primitive.

```
(define a 5)
(define add (lambda (a b) (+ a b)))
```

The line 1 in above code creates a variable named a with the value of 5. The line 2 in the above code defines a function add that takes two parameters using lambda expression a and b, evaluating to addition of these two parameters.

3.3.1.4 Operators

Apart from the basic operation defined in 3.3.1.1, there are other operators like remainder and logical operators that are useful in a programming language. The evaluation rules for these operators is given below

```
case "%":
  return this.evaluate(second, env)
      % this.evaluate(third, env) || 0;
case "<":
  return this.evaluate(second, env) < this.evaluate(third, env)
    ? "#t"
    : "#f";
case ">":
  return this.evaluate(second, env) > this.evaluate(third, env)
    ? "#t"
    : "#f";
case "==":
  return this.evaluate(second, env) == this.evaluate(third, env)
    ? "#t"
    : "#f";
case ">=":
  return this.evaluate(second, env) >= this.evaluate(third, env)
    ? "#t"
    : "#f";
case "<=":
  return this.evaluate(second, env) <= this.evaluate(third, env)
    ? "#t"
    : "#f";
```

All of these rules perform the specified operator evaluation on the evaluated code of second and third expression. The remainder operator denoted by `%` provides a extra check to avoid divide by zero error. Here are few example of usage of this operators

```
(% 6 5)
```

```
(< 4 5)
(>= 4 5)
```

The line 1 in above code evaluates to 1. The line 2 in the above code evaluates to `#t` that is define true in the proposed language and the line 3 evaluates to `#f` denoting false for the proposed language.

3.3.1.5 If

`if` is a special form of primitive that is used for conditional execution. It takes three arguments, a test expression, a consequent expression, and an alternative expression. The code below shows evaluation rules for `if`

```
case "if":
    let cond = this.evaluate(second, env);
    if (cond == "#t") {
      return this.evaluate(third, env);
    } else if (cond == "#f") {
      return this.evaluate(rest[1], env);
    } else {
      throw new
        Error("The condition does not match to true or false.");
    }
```

The code first evaluates the condition expression provide in the second variable in the given environment using the evaluate method. If the value of the condition is the Scheme boolean `#t`, it evaluates and returns the value of the evaluation of the third expression in the same environment. If the value of the condition is the Scheme boolean `#f`, it evaluates and returns the value of the evaluation of fourth expression in the same environment. If the value of the condition is neither `#t` nor `#f`, an error is thrown. The code below shows an example of SpartanScript program using if statement

```
(define x 4)
(if (== x 4) (display "x is 4") (display "x is not 4"))
```

3.3.1.6 Require

The require statement is special blockchain construct provided in programming languages like Solidity. The require statement is used to check if certain conditions are met during the execution of a function. If the condition is not met, the function execution will be immediately stopped and any changes made to the blockchain will be rolled back. The code below shows the evaluation rules for require

```
case "require":
    if (this.evaluate(second, env) == "#t") {
      break;
    } else {
      throw new Error("The require condition does not match.");
    }
```

This evaluation is similar to if construct, except the expression of false evaluation throws a error that immediately stops the current execution reverting the state of blockchain. The require statement can be used as follows

```
(require (== 5 5) doSomething)
```

The code above will return evaluation of expression `doSomething` as the second expression evaluates to true.

3.3.1.7 Begin

A begin statement is a construct in scheme that is used to group multiple expressions into a single expression. It allows the user to evaluate several expressions in sequence and return the result of the last expression. This is a powerful construct to emulate the sequential execution of program within a function which otherwise is not common in functional programming language.

```
case "begin":
    this.evaluate(second, env);
    for (let i = 0; i < rest.length - 1; i++) {
      this.evaluate(rest[i], env);
```

```
    }
    return this.evaluate(rest[rest.length - 1], env);
```

This code evaluates all the expression after begin and returns the evaluation of the last expression in the begin statement. Here is an example of begin statement

```
(begin (define x 4) (define y 5) (+ x y))
```

The above example evaluates defines two variable x and y and the last expression evaluates to 9 using these variables..

3.3.1.8 Hash Map

A hash map is important data structure that allows you to associate a value with a particular key. To implement a hash map in the proposed language, `setMap`, `getMap`, `hasMap` and `deleteMap` primitives are provided. A hash map is defined using the `makeMap` which returns the javascript instance of hash map called `Map`. A hash map in the proposed language can be defined as `(define testMap makeMap)`. Here the `testMap` variable is intialized with a new instance of javascript `Map`. The code following shows the evaluation rules for functions `setMap`, `getMap`, `hasMap` and `deleteMap`

```
    case "getMap":
        if (second.type == VAR) {
          let res = this.hasVariable(env, second.value)
            .get(
                this.evaluate(third, env)
            );
          return res;
        } else {
          return this.evaluate(second, env)
            .get(this.evaluate(third, env));
        }
      case "setMap":
        if (second.type == VAR) {
          return this.hasVariable(env, second.value)
```

```
      .set(
          this.evaluate(third, env),
          this.evaluate(rest[1], env)
      );
  } else {
    let res = this.evaluate(second, env)
        .set(
          this.evaluate(third, env),
          this.evaluate(rest[1], env)
        );
    return res;
  }

case "hasMap":
  if (second.type == VAR) {
    return this.hasVariable(env, second.value)
      .has(
          this.evaluate(third, env)
      )
      ? "t"
      : "#f";
  } else {
    return this.evaluate(second, env)
      .has(this.evaluate(third, env))
          ? "#t"
          : "#f";
  }
case "deleteMap":
  if (second.type == VAR) {
    return this.hasVariable(env, second.value)
      .delete(
          this.evaluate(third, env)
      )
          ? "t"
          : "#f";
  } else {
    return this.evaluate(second, env)
      .delete(this.evaluate(third, env))
          ? "#t"
          : "#f";
  }
```

A check for AST type of `VAR` is added for all of these methods to incorporate support for nested hashmap. If the AST type of second expression is `VAR` then the hash map with provided variable name is checked in already defined variables. This variable would return a instance of javascript `Map` on which the respective methods are executed. However, if the AST type of second expression is not `VAR` then the evaluation of second expression returning a instance of javascript `Map` is used to perform these methods. The following code shows examples of hash map used in SpartanScript using the proposed interpreter.

```
;; Define hashmap
(define map1 makeMap)
;; Define nested hashmap
(define map2 makeMap)

;; Set hashmap
(setMap balances $me totalSupply)
;; Set nested hashmap
(setMap (getHash allowed $me) newkey newvalue)

;; Get hashmap
(getMap balances $me)
;; Get nested hashmap
(getMap ((getMap allowed $me) newkey))

;; Has hashmap
(hasMap balances $me)
;; Has nested hashmap
(hasMap ((getMap allowed $me) newkey))

;; Delete hashmap
(deleteMap balances $me)
;; Delete nested hashmap
(deleteMap ((getMap allowed $me) newkey))
```

3.3.2 Blockchain primitives

Apart from the basic primitives, there are several useful information of block and the transaction that are required for execution of smart contract, that can be used to

perform essential blockchain operations. In EVM, such information are provided in the state machine using the several opcodes. The current proposed implementation provides some of these primitives.

3.3.2.1 $me and $sender

A smart contract at the time of declaration needs to create its own account. This account is similar to client account that is used by users to perform transactions. To easily access the address of this smart contract account, the `$me` primitive returns the address of the account that deploys the smart contact. The `$me` primitive can be used directly in the SpartanScript smart contract code and it will replace this value with the address of the account of the smart contract. The value of the `$me` is set when the interpreter is initialized with the address of the smart contract.

Similar to `$me`, `$sender` is used to store the address of the client that initiates the transaction calling the smart contract. Both of these primitives are extremely useful to develop smart contracts that require to operate using the address information of both the smart contract and the client calling the contract.

3.3.2.2 $balance

This primitive provides a the smart contract to directly check balances of the account on the blockchain. Using the balances primitive can be useful that involve transfer of currency from one account to another. For example to check for sufficient balances before transfer to avoid double spending problem. The code below shows the evaluation rule for this primitive

```
case "$balance":
    return this.currBalances.get(this.evaluate(second, env));
```

In this evaluation rule, this primitive checks for the balances from the `currBalances` map, that is the latest state of balances on the blockchain for the current address provided in the first parameter. This primitive can be used as `($balance`

`$me)` which will return the balance of `$me` that is the address of the smart contract account.

3.3.2.3 $transfer

Another primitive that is essential for smart contract development is the `$transfer` primitive. This primitive provides the smart contract caller to give currency to the smart contract account. This primitive will directly deposit currency in the smart contract account and modify state after successful completion of the smart contract. The evaluation rule for this primitive is given below

```
case "$transfer":
    let destination = this.evaluate(third, env);
    let amount = this.evaluate(second, env);
    if (this.currBalances[this.$me] < parseInt(amount)) {
      throw new Error("Not enough balance.");
    }
    this.currBalances.set(
      destination,
      this.currBalances.get(destination) + amount
    );
    this.currBalances.set(
      this.$me,
      this.currBalances.get(this.$me) - amount
    );
    break;
```

In this evaluation rule, the amount to be transferred and destination of the account to transfer currency are evaluated from the second and third expression. The balances of the smart contract account is checked to avoid double spending and then the amount is transferred from address of the smart contract account to the address of the destination account. For example, `(transfer amt "addr1")` will transfer currency specified in amount from smart contract account to the address specified in the third expression that is `"addr1"`.

3.3.2.4 defineState

A smart contract state maintains its state by storing its local variable used in the smart contract program in the blockchain along with other states of blockchain. To provide easier access and modification to this state on blockchain, this proposed interpreter provides a `defineState` primitive to store variables on the blockchain. Each smart contract is provided with a variable space on the blockchain that will store a mapping of the name of the variable to the value of the variable to store the state of the smart contract on the blockchain. The evaluataion rule for `defineState` is as shown below

```
case "defineState":
    this.contractStateVariables.set(
      second.value,
      this.evaluate(third, env)
    );
    break;
```

In this evaluation rule, the interpreter changes the variable space mapping to store the variable name present in the second expression to the evaluation of third expression as value. The `contractStateVaribles` provides the variable space on the blockchain for the current smart contract being executed. Using the `defineState` is similar to setting variable using the `define` primitive except the variables are stored directly on the blockchain as shown below

```
(defineState a (+ 5 6))
```

3.3.2.5 $timestamp

Another one of the most prominent primitives required is `$timestamp` which provides the timestamp of the current block which is the time at the which the block was mined. The $timestamp is used as a local variable in the smart contract and can be used directly to replace the $timestamp with the value of the time at which

the block was mined that is usually set by a miner. The usage of these primitives is discussed in detail in 5.1.

3.4 Scoping

In Scheme programming language, scoping refers to the rules that determine the visibility and accessibility of variables and functions within different parts of a program. Scheme is a lexically scoped language, which means that the scope of a variable is determined by its position in the program's source code, or lexically rather than the scope is determined by the sequence of function calls that led to its current value [13]. When a variable is defined within a block of code, such as a function or conditional statement, it is said to be bound to that block's lexical environment. This means that the variable is only accessible within that block of code and any nested blocks. The outer blocks of code do not have access to the variable.

In Scheme, variables can be either global or local. Global variables are defined outside of any function and can be accessed from any part of the program. Local variables, on the other hand, are defined within a function and are only visible within that function, as well as any nested functions.When a function is called in Scheme, a new scope is created for it. This function contains its arguments and any local variables defined within the function. If a variable is referenced within a function, Scheme searches for its value first in the local scope, and then in any enclosing scopes, until it reaches the global scope. If the variable is not found in any of these scopes, an error is raised.

To facilitate the use of lexical scoping in the proposed SpartanScript programming language, a Linked List data structure is used where the first node describes the global scope and the subsequent nodes denoting inner scopes. The code example below describes the structure of the Linked List that is used in the proposed SpartanScript

language interpreter.

```
class ScopingEnvironment {
    constructor(parent) {
       this.varMap = new Map();
       this.parent = parent;
       this.provide = new Set();
     }


     hasVar(key) {
       if (this.varMap.has(key)) return true;
       if (this.parent !== null) return this.parent.hasVar(key);
       return false;
     }


     getVar(key) {
       // check in current scope else find in outer scope
       if (this.varMap.has(key)) return this.varMap.get(key);
       if (this.parent !== null) return this.parent.getVar(key);
       // throw new Error(‘The variable ${key} is not declared.‘);
       return null;
     }


     setVar(key, val) {
       if (this.varMap.has(key)) {
         this.varMap.set(key, val);
         return true;
       }
       if (this.parent !== null) {
         return this.parent.setVar(key, val);
       }
       throw new Error(‘The variable ${key} can not be found.‘);
     }
}
```

The `ScopingEnvironment` class defines a new scope and takes a parent environment as an optional argument. The parent environment allows access to variables defined in the outer scope. The class contains three methods:

- hasVar(key) - This method takes a variable name key as an argument and checks if the variable is defined in the current scope or any of its parent scopes. It returns true if the variable is found and false otherwise.
- getVar(key) - This method takes a variable name key as an argument and returns the value of the variable if it is defined in the current scope or any of its parent scopes. If the variable is not found in any scope, the method returns null.
- setVar(key, val) - This method takes a variable name key and a value val as arguments, and sets the value of the variable if it is defined in the current scope or any of its parent scopes. If the variable is not found in any scope, the method throws an error.

The `varMap` property is a Map object that stores the variables defined in the current scope. The `provide` property is a Set object that stores the variables that are passed from parent scopes to child scopes. This scoping environment can be used to implement lexical scoping in JavaScript or other programming languages that support closures. By keeping track of variables and their values in different scopes, the interpreter ensures that variables are accessed and modified only within their intended scope.

3.5 Usage

To utilize the SpartanScript interpreter, a miner or user must instantiate the `SpartanScript` class. Once an instance has been created, the user can initiate the evaluation process by calling the `interpret` function. This section provides examples of how to create an instance of the `SpartanScript` class, including the necessary information, and how to begin evaluating a smart contract script. The following code snippet demonstrates the constructor that is invoked when creating an instance of the `SpartanScript` class.

```
constructor(block, tx) {
```

```
    this.block = block;

    this.$me = tx.data.address;

    this.prevBalances = block.balances;
    this.currBalances = new Map(block.balances);

    this.gasLimit = tx.data.gasLimit;
    this.gasCurr = 0;

    this.prevContractStateVarible = block.contractStateVariables
        .get(
            tx.data.address
        );
    this.contractStateVariables = new Map(
      block.contractStateVariables.get(tx.data.address)
    );

    this.sender = tx.from;
    console.log(
      `The smart contract address is ${this.$me}
      and the sender address is ${this.sender}.`
    );

  }
```

To initialize the instance of `SpartanScript` class, the miner needs to provide with the current transaction that was used for contract invocation and the current block in which this transactions are stored. There are several variables set at the start of initiation like `$me` which is the address of the smart contract account. The variables `prevBalances` and `prevContractStateVariables` stores the initial state of the balances and the variable space of the smart contract. Likewise the `currBalances` and `contractStateVariables` store the intermediary state that is used to keep track of any modified state at the time of execution of the smart contract. The variable `gasLimit` describes the maximum amount of gas the client is willing to spend on the smart contract. The variable `gasCurr` maintains the gas being used at the time

of execution of the smart contract. The variable `sender` stores the address of the account that initiated the transaction for contract invocation. After creation of the instance of the `SpartanScript` class, the `interpret` function is called that performs the evaluation process along with tokenization and parsing. The interpret function is as follows

```
interpret(script, env = new ScopingEnvironment(null)) {
    this.globalEnv = env;
    let tokens = this.tokenize(script);
    let asts = this.parse(tokens);
    // console.log(this.printAST(asts));

    try {
      asts.forEach((ast) => {
        this.printAST(ast, env);
        console.log("Final value -> ",
            this.evaluate(ast, env), this.gasCurr);
      });
    } catch (error) {
      ...
    }

    ...
    return { gasUsed: this.gasCurr };
  }
```

This method takes in a smart contract script and an optional `env` parameter that represents the current environment in which the code will be evaluated. The global environment is set to the environment provided in the `env` parameter. This method then tokenizes the script using the `tokenize` method and parses the resulting tokens into an AST using the `parse` method. The `evaluate` method is called to recursively evaluate each AST node in the current environment.

3.6 Gas

In Ethereum, the gas is the unit of measurement for computational and storage resources required to perform actions on the blockchain. Ethereum considers every

computational step performed by transactions and smart contracts. The gas serves a dual role in Ethereum: as a buffer between the volatile price of Ethereum and the reward for miners, and as a defence against denial-of-service attacks. The gas system helps to prevent accidental or malicious infinite loops or computational waste on the network. To do this, the initiator of each transaction must specify a limit to the amount of computation they are willing to pay for. This disincentivizes attackers from sending spam transactions as they must pay for the computational, bandwidth, and storage resources that they consume. Gas is a critical component of Ethereum that helps to maintain the network's security and efficiency by ensuring that users pay for the resources they consume.

The gas limit is the maximum amount of gas that can be consumed by a transaction. If a transaction exceeds the gas limit, it will be rejected by the network and the gas fee will be lost. Therefore, it is important for senders to estimate the gas required for their transactions accurately and set an appropriate gas limit. Gas is a critical component of the Ethereum network that ensures the secure and efficient execution of transactions and smart contracts. It provides a mechanism for incentivizing miners to process transactions and prevents network spamming and abuse. To incorporate the usage of gas in the proposed implementation of the SpartanScript programming language, gas prices are defined similarly to the opcodes in EVM. The gas prices for the proposed implementation are defined as given below -

```
const GASPRICES = {
  ADD: 3,
  SUB: 3,
  MUL: 5,
  DIV: 5,
  MOD: 5,
  LT: 3,
  GT: 3,
  LTE: 3,
```

```
  GTE: 3,
  ET: 3,
  TRANSFER: 15,
  BALANCE: 10,
  DEFINE: 3,
  GETMAP: 5,
  SETMAP: 7,
  HASMAP: 5,
  DELETEMAP: 7,
  IF: 8,
  REQUIRE: 8,
  BEGIN: 7,
  DEFINESTATE: 5
};
```

These gas prices provides the fixed values of operation cost for a specific primitive. The total gas price of a smart contract is calculated by adding all the operation cost at the time of evaluating the primitives. For example, the following code shows adding operation cost of addition primitive

```
evaluate(ast, env) {
    ...
    if (this.gasCurr >= this.gasLimit) {
      throw new GasLimitReachedError(this.$me);
    }
    ...
    case "+":
        this.gasCurr += GASPRICES.ADD;
        return (
          rest.reduce((x, y) => x + this.evaluate(y, env), 0) +
          this.evaluate(second, env)
        );
    ...
}
```

The variable `gasCurr` is used to keep track of the total gas price for the current smart contract program. For each evaluation, if the current gas price consumed by the smart contract reached the gas limit provided by the client, a special type of

error called `GasLimitReachedError` is thrown. This error type ensures to distinguish between gas error and other errors that may arise during smart contract execution. At the end of the entire evaluation of the current program, the total gas price calculated is compared to the gas limit specified by the initiator of the transaction for this smart contract. The gas price consumed determines if the state of blockchain needs to be changed to new state created by smart contract execution. This is determined in the interpret method after completion of the evaluation in the following code.

```
interpret(script, env = new ScopingEnvironment(null)) {
    ...
    try {
      ...
    } catch (error) {
      // Catch error when gas limit is reached
      if (error instanceof GasLimitReachedError) {
        // throw error;
        return { gasUsed: this.gasLimit };
      }

      // Log other errors
      console.log(error);
    }

    for (let [key, value] of this.currBalances) {
      this.prevBalances.set(key, value);
    }
    for (let [key, value] of this.contractStateVariables) {
      this.prevContractStateVarible.set(key, value);
    }

    ...
  }
```

In the `interpret` function, if an error is thrown during the evaluation of the smart contract and the error is of type `GasLimitReachedError`, then the state of the blockchain is not changed returning the current gas used for execution. This consumed gas can be charged back to user for operational cost similar to behaviour in ethereum.

For a successful execution of the smart contract and adequete gas specifed by the client as the gas limit, the state of the blockchain is changed to new state reached by execution of the smart contact including the balances and variable space of the smart contract.

CHAPTER 4

Spartan Gold

Spartan Gold [14] is an open-source project that implements a simple cryptocurrency system, based on the blockchain technology, for educational purposes. The project consists of a set of Javascript programs that implement the core functionality of a blockchain-based cryptocurrency, including transaction verification, block creation, and mining. It uses a simple proof-of-work consensus algorithm, similar to Bitcoin, to ensure the integrity of the blockchain. The currency used in the spartan gold is referred to as gold. The project is not intended to be a fully-functional cryptocurrency system suitable for use in real-world applications. Instead, it is a simplified implementation that can be used as a starting point for further research and experimentation.

This repository defines several classes like `Transaction`, `Block`, `Blockchain`, `Miner`, `Client` and many more. The repository provides a simulation of distributed network with multiple client an miners. The clients are able to post multiple transaction that represent the change in the blockchain state. The miner works towards finding the proof for the proof-of-work consensus to validate and store this transaction on the blockchain. The type of transaction is defined in `Transaction` and the type of block is defined in `Block`. The `Blockchain` class contains information about the spartan gold blockchain like the difficulty of proof of work, the network to be used, and many more.

4.1 Support for smart contracts

To add support for smart contracts and integration of SpartanScript in spartan gold, there are various modifications implemented in spartan gold. These modifications include adding a support for two new transaction types: `ContractDeclaration` and `ContractInvocation`. These modifications also introduces new type of `Block` called `SmartBlock` exteding the existing `Block` class.

4.1.1 Transactions

The code below shows a sample transaction in current spartan gold implementation. There are various fields defined in the transaction like from that denotes the address of the payer, the fee amount the payer is willing to pay the miner, the outputs of the address where the reward gold will be transferred.

```
{
     from: '4HWTOR8cgvejeMd...',
     nonce: 0,
     pubKey: '-----BEGIN PUBLIC KEY-----\n' ...,
     sig: '83adb439...',
     fee: 1,
     outputs: [
     { amount: 25, address: 'vAy8w7bavN9...' }
     ],
     data: {}
}
```

The spartan gold supports transaction that transfer gold from one user to another user. To support smart contract in the transaction, the data field is used. Several information about the smart contract like the contract address, the contract script content, and the contract script hash are provided in the date field. To facilitate smart contract usage in spartan gold, two new types of transactions are added. The first type of transaction type is added to deploy the smart contract on blockchain for further use by clients of spartan gold. The other transaction type is the Contract invocation where the contract can be called for transactions to be performed by clients using this transaction type.

In the contract declaration transaction, the data field in the transaction is added with the `scriptHash` field that denotes the hash of the script of smart contract, the `scriptContent` that contains the entire content of the smart contract script and the address of the account for the smart contract. The transaction format for a contract

declaration transaction is as following

```
{
  from: '1yqDopA/...',
  nonce: 1,
  pubKey: '-----BEGIN PUBLIC KEY-----\n' ...,
  sig: 'a72ab0c4c...',
  fee: 0,
  outputs: [],
  data: {
    type: 'ContractDeclaration',
    scriptHash: '94f057082...',
    scriptContent: '(provide totalSupply ...,
    address: 'ZaSUQxIBr5...'
  }
}
```

In the contract invocation transaction, the data field in the transaction is added with the `scriptHash` field that denotes the hash of the script of smart contract, the `scriptCall` that contains the script for calling the smart contract, the address of the account for the smart contract and the `gasLimit` specifies the amount of maximum gas the client is willing to pay for execution of the smart contract. The transaction format for a contract invocation transaction is as following

```
 {
  from: '1yqDopA/...',
  nonce: 2,
  pubKey: '-----BEGIN PUBLIC KEY-----' ...,
  sig: '22eabddcb6...',
  fee: 0,
  outputs: [],
  data: {
    type: 'ContractInvocation',
    scriptHash: '94f057082...',
    call: '(defineState totalSupply 200) ...,
    address: 'ZaSUQxIB...',
    gasLimit: 200
  }
}
```

4.1.2 SmartBlock

To incorporate smart contracts in the spartan gold blockchain, a new block called `SmartBlock` is introduced. The new `SmartBlock` overrides the `addTransaction` function which provides support for `ContractDeclaration` and `ContractInvocation` transaction types. This new block also stores the smart contract scripts and the variable space of smart contracts in the block, to store it on the blockchain. The variable `contractMap` is used to store smart contracts mapped to the address of the smart contract account. The variable `contractStateVariables` stores the variable space of the specific smart contract. The following code shows the constructor of the `SmartBlock` where smart contract and the variable space are stored.

```
constructor(rewardAddr, prevBlock, target, coinbaseReward) {
    super(rewardAddr, prevBlock, target, coinbaseReward);

    // Storing smart contracts mapped to script hash
    this.contractMap = prevBlock
        ? new Map(prevBlock.contractMap)
        : new Map();

    //Store Smart contracts state
    this.contractStateVariables = prevBlock
      ? new Map(prevBlock.contractStateVariables)
      : new Map();
}
```

For the `ContractDeclaration` transaction type, a new variable space is created on the blockchain mapped to the address of the smart contract and the smart contract script is stored on the blockchain completing the contract deployment process. For the `ContractInvocation` transaction type, a new instance of the smart interpreter is created by passing the current block and the transaction. With the newly created interpreter instance, the interpret function is called to run the smart contract script. The interpret function returns the total gas used for the smart contract execution.

This returned gas used is then charged back to the client by updating the amount in the balances of the blockchain.

```
addTransaction(tx, client) {

    ...

    // Run smart contracts giving gold
    // to the specified output addresses
    switch (tx.data.type) {
      case "ContractDeclaration": {
        // Store smart contracts content on blockchain
        if (!this.contractStateVariables.has(tx.data.address))
          this.contractStateVariables
            .set(tx.data.address, new Map());

        this.contractStateVariables
          .get(tx.data.address)
          .set("$timestamp", Date.now());
        this.contractMap
            .set(tx.data.scriptHash, tx.data.scriptContent);
        break;
      }
      case "ContractInvocation": {
        let intrepreter = new SmartInterpreter(this, tx);

        ...

        // Call interpreter to perform
        // run smart contract operation
        let result = intrepreter.smartIntrepreter(
          this.contractMap.get(tx.data.scriptHash)
          + tx.data.call
        );

        // Taking gas from client
        let senderBalance = this.balanceOf(tx.from);
        this.balances
            .set(tx.from, senderBalance - result.gasUsed);
        let oldBalance = this.balanceOf(this.rewardAddr);
        this.balances
            .set(this.rewardAddr, oldBalance + result.gasUsed);
```

```
        break;
      }
    }

    ...
}
```

CHAPTER 5

Smart Contracts Examples

The integration of SpartanScript into the SpartanGold blockchain platform expands the range of capabilities available for developing smart contracts. Smart contracts are essential for creating innovative applications on any blockchain platform. To validate the effectiveness of SpartanScript for smart contract development, this chapter presents several examples of smart contracts that showcase different use cases. These examples are designed to run on the SpartanGold blockchain, utilizing the SpartanScript programming language.

5.1 Timestamping

In a blockchain, a timestamp is a piece of data that records the time at which a particular event occurred, such as the creation of a block or the execution of a transaction. Timestamping is an essential feature of blockchain technology, as it ensures that all nodes on the network can agree on the order in which events occurred and the state of the blockchain at any given time [1]. In most blockchain implementations, a timestamp is included in the header of each block, along with other information such as the block number, nonce, and hash value. The timestamp is typically represented as the number of seconds since a specific reference point, such as the Unix epoch time (January 1, 1970).

Timestamps serve several critical functions in blockchain, including:

1. Ordering transactions: The timestamp helps to establish the chronological order of transactions within a block and across the entire blockchain. This ensures that all nodes on the network have a consistent view of the blockchain's state and can reach a consensus on its history.

2. Preventing double-spending: The timestamp helps to prevent the double-

spending of cryptocurrency by ensuring that transactions are processed in the correct order.

3. Enabling smart contracts: Timestamps are essential for smart contract execution, as they help to establish the conditions under which contract terms are executed.

Timestamps are helpful in smart contracts because they provide a reliable way to verify when an event occurred or a specific condition was met [15]. This is critical for the execution of smart contracts, which rely on specific conditions to be met to trigger the automatic execution of code. For example, a smart contract that executes a payment after a certain condition is met, such as the delivery of goods or completion of a project, can use a timestamp to verify when the condition was fulfilled. This helps to ensure that the payment is executed only when the condition has been met and prevents fraudulent behavior. Timestamps can also be used in smart contracts to create time-based triggers or schedules. For example, a smart contract that automatically executes a payment on a specific date or after a specific amount of time can use a timestamp to trigger the payment at the appropriate time. Timestamps are also used for varied applications like time synchronization in Internet of things (IOT) where time synchronization is very important. Timestamps are utilized in diverse applications, such as time synchronization in the Internet of Things (IoT), where accurate time synchronization is critical [16].

To demonstrate the use of timestamps one such example of time-based smart contracts is shown below. This code defines a procedure called `transferGold` that takes three parameters: `addr1`, `addr2`, and `amount`. The purpose of the procedure is to transfer an amount of gold from one address (`addr1`) to another address (`addr2`) based on a certain condition. The condition is based on the value of `$timestamp`, which is a built-in variable that represents the number of seconds since the Unix

epoch time (January 1, 1970) and is recorded when a new block is mined and added to the blockchain. If the value of the `$timestamp` recorded is even, then the procedure will transfer the specified amount of gold from `addr1` to the address of the account of the smart contract using the `$transfer` function. If the value of the `$timestamp` recorded is odd, then the procedure will transfer the gold from `addr2` to the address of the account of the smart contract.

```
(provide transferGold)

(define transferGold
    (lambda (addr1 addr2 amount)
        (if (== (% $timestamp 2) 0)
            ($transfer amount addr1)
            ($transfer amount addr2))))
```

Specifically, the code performs a conditional transfer of gold between two addresses based on whether the current timestamp is even or odd. This could be useful in certain situations where you want to randomise the direction of the transfer or introduce a degree of unpredictability into the transfer process. Timestamps serve as a fundamental component of blockchain technology, helping to ensure the accuracy, security, and integrity of blockchain transactions and facilitating the development of decentralized applications and smart contracts.

5.2 Token standards

Smart contract standards are a set of rules and guidelines that define how smart contracts should be designed and implemented. These standards provide a common framework for developers to create smart contracts that are interoperable and compatible with other smart contracts and blockchain platforms.

Some of the most popular smart contract standards include:

1. ERC-20 - This is the most widely used standard for creating fungible tokens on the Ethereum blockchain. It defines a set of rules for how tokens should be designed and how they should behave on the network.

2. ERC-721 - This standard defines a set of rules for creating non-fungible tokens (NFTs) on the Ethereum blockchain. NFTs are unique digital assets that represent ownership of a specific item, such as a piece of art or a collectible.

3. ERC-1155 - This standard allows for the creation of both fungible and non-fungible tokens on the Ethereum blockchain. It provides a more flexible and efficient way of creating multiple types of assets on a single smart contract.

4. NEP-5 - This is the standard used for creating tokens on the NEO blockchain. It is similar to ERC-20 in that it defines a set of rules for how tokens should be designed and how they should behave on the network.

5. BEP-20 - This is the standard used for creating tokens on the Binance Smart Chain. It is based on the ERC-20 standard and provides a similar set of rules for token design and behavior.

These smart contract standards were designed and proposed by various individuals and organizations within the blockchain community.

5.2.1 ERC-20 token standard

The ERC-20 token standard is a set of rules and requirements that govern the creation and management of tokens on the Ethereum blockchain. It was initially proposed by Fabian Vogelstellar and later finalized by the Ethereum community [17]. This standard has been developed and refined over time through a collaborative process involving the wider blockchain community. ERC-20 defines a set of functions and events that a token contract must implement in order to be considered ERC-20

compliant. These include functions for transferring tokens, checking the balance of tokens in an address, approving another address to spend tokens on your behalf, and getting the total supply of tokens in circulation. ERC-20 also includes events that are triggered when tokens are transferred or approved, allowing developers to build applications that respond to these events.

ERC-20 is a standard interface that defines six functions and two events for Ethereum-based tokens. These functions and events are as follows -

- totalSupply() - This function returns the total supply of tokens in circulation.
- balanceOf(address) - This function returns the balance of tokens held by a given address.
- transfer(address, uint256) - This function allows an address to transfer tokens to another address.
- approve(address, uint256) - This function allows an address to approve another address to spend tokens on its behalf.
- allowance(address, address) - This function returns the amount of tokens that an address is allowed to spend on behalf of another address.
- transferFrom(address, address, uint256) - This function allows an address to transfer tokens on behalf of another address.

In addition to these six functions, there are two events that are defined by the ERC-20 standard -

- Transfer - This event is triggered whenever tokens are transferred from one address to another.
- Approval - This event is triggered whenever an address approves another address to spend tokens on its behalf

There are various implementations proposed for the ERC-20. One of the most

prevalent implementations of ERC-20 standard in Solidity programming language is provided by Openzeppelin[18]. One such implementation of ERC-20 token smart contract in solidity language to be run on the Ethereum virtual machine is as below

```
pragma solidity ^0.8.0;

contract MyToken {
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 private _totalSupply;

    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256))
        private _allowances;

    constructor(string memory _name,
        string memory _symbol,
        uint8 _decimals,
        suint256 _initialSupply) {
            name = _name;
            symbol = _symbol;
            decimals = _decimals;
            _totalSupply = _initialSupply * 10**uint256(decimals);
            _balances[msg.sender] = _totalSupply;
            emit Transfer(address(0), msg.sender, _totalSupply);
    }

    event Transfer(address indexed from,
        address indexed to, uint256 value);
    event Approval(address indexed owner,
        address indexed spender, uint256 value);

    function totalSupply()
        public view returns (uint256) {
            return _totalSupply;
    }

    function balanceOf(address account)
        public view returns (uint256) {
            return _balances[account];
```

```
}

function transfer(address recipient, uint256 amount)
    public returns (bool) {
        require(recipient != address(0),
            "ERC20: transfer to the zero address");
        require(_balances[msg.sender] >= amount,
            "ERC20: transfer amount exceeds balance");

        _balances[msg.sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(msg.sender, recipient, amount);
        return true;
}

function allowance(address owner, address spender)
    public view returns (uint256) {
        return _allowances[owner][spender];
}

function approve(address spender, uint256 amount)
    public returns (bool) {
        _allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
}

function transferFrom(address sender,
    address recipient, uint256 amount) public returns (bool) {
        require(recipient != address(0),
            "ERC20: transfer to the zero address");
        require(_balances[sender] >= amount,
            "ERC20: transfer amount exceeds balance");
        require(_allowances[sender][msg.sender] >= amount,
            "ERC20: transfer amount exceeds allowance");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        _allowances[sender][msg.sender] -= amount;
        emit Transfer(sender, recipient, amount);
        return true;
}
```

```
}
```

In contrast, the following code shows an implementation of ERC-20 standard code in SpartanScript interpreted language. These program represent implementation for ERC-20 in SpartanScript to be able to deploy and execute ERC-20 tokens on SpartanGold blockchain.

```
(provide totalSupply balanceOf allowance
    transfer approve transferFrom)

(define balanceOf
    (lambda (addr)
        (getMap balances addr)))

(define transfer
    (lambda (receiver tokens)
        (begin
            (require (>= (getMap balances $sender) tokens))
            (setMap balances $sender
                (- tokens (getMap balances $sender)))
            (setMap balances receiver (+ tokens
                (if (hasMap balances receiver)
                    (getMap balances receiver)
                    0)))
            #t)))

(define allowance
    (lambda (owner addr)
        (getMap (getMap allowed owner) addr)))

(define approve
    (lambda (addr tokens)
        (begin
            (setMap
                (if (hasMap allowed $sender)
                    (getMap allowed $sender)
                    (setMap allowed $sender makeMap))
                addr tokens)
```

```
            #t)))

(define transferFrom
    (lambda (owner buyer tokens)
        (begin
            (require (<= tokens
                (getMap balances owner)))
            (require (<= tokens
                (getMap (getMap allowed owner) $sender)))
            (setMap
                balances
                owner
                (- tokens (getMap balances owner)))
            (setMap
                (getMap allowed owner)
                $sender
                (- tokens (getMap (getMap allowed owner) $sender)))
            (setMap balances buyer (+
                (if (hasMap balances buyer)
                    (getMap balances buyer)
                    0)
                tokens))
            #t)))
```

The `provide` statement at the beginning denotes that there are six public functions in this contract denoting all of the interface functions of ERC-20 standard. The `totalSupply` function returns the total number of tokens in circulation for a given token. The `totalSupply` acts as a getter property for total supply of tokens that will be stored on the blockchain. The `balanceOf` function returns the balance of tokens held by a specific address in the argument named `addr`. The `balanceOf` function access the value using the `getMap` method of Hashmap for balances defined in the contract. The `transfer` function is used to transfer tokens from one address to another. This function takes the address of the receiver and the number of tokens to be transferred as input parameters. This function initially checks for sufficient balances

for the successful execution avoiding negative balances using require statement. The rest of the function updates the balances of the sender and receiver to update the token value in the ledger.

Another function in ERC-20 involves the use of the allowance to monitor the permission for other users for spending on behalf of the token owner. These permissions are tracked in a nested Hashmap structure named allowance that stores a map of all the approved spender and their corresponding amount for each token owner. The `approve` function is used to grant permission to another address to spend tokens on behalf of the sender. This function takes the address of the spender and the number of tokens to be approved as input parameters. These values are added or updated in the allowance Hashmap using the if statement while checking for empty values for the respective token owners. The `allowance` function returns the number of tokens that a spender is allowed to transfer from the owner's account. This function is used by the spender to determine how many tokens they are allowed to spend on behalf of the owner. The `transferFrom` function is used by the spender to transfer tokens from the owner's account to another account. This function takes the address of the owner, the address of the buyer, and the number of tokens to be transferred as input parameters. This function also forces the owner to have sufficient balance and the buyer to have an allowance for transfer from the sender. The balances and allowances of the owner and buyer are updated after successfully checking for balances and allowances in this function.

To deploy this smart contract, the statements below are used to initialize the amount of the total supply of tokens, balances and allowances. The `defineState` statement ensures the storage of these variables on the blockchain. The entire `totalsupply` is assigned to the contract deployer when the contract is deployed and stored on the blockchain for the first time.

```
(defineState totalSupply 2000)
(defineState balances makeMap)
(defineState allowed makeMap)

(setMap balances $sender totalSupply)
```

The ERC-20 token standard was created to provide a common set of rules and functions that Ethereum-based tokens can follow. It allows for a standardized way of creating, issuing, and managing tokens on the Ethereum blockchain. With the ERC-20 standard, tokens can be created that are interoperable with other tokens and applications on the Ethereum blockchain. This means that ERC-20 tokens can be traded on decentralized exchanges, used as collateral for decentralized finance (DeFi) applications, and integrated with other blockchain-based applications. The standard provided by ERC-20 is a set of guidelines and functions that developers can follow to create tokens on the Ethereum blockchain. This makes it easier and more efficient for developers to create new tokens since they can follow a standard set of rules and functions instead of starting from scratch. The ERC-20 standard includes several security features, such as preventing double spending and ensuring that tokens can only be transferred by the token holder. These features help to prevent fraud and make ERC-20 tokens more secure.

The ERC-20 standard allows for the tokenization of real-world assets, such as commodities, real estate, and securities. This makes it easier to trade these assets on the blockchain and opens up new possibilities for asset ownership and investment. ERC-20 tokens are commonly used for Initial Coin Offerings (ICOs), enabling companies to raise funds by issuing tokens to investors [19]. One of the most popular appications of an ERC-20 token is Tether [20], which is a stablecoin that is pegged to the US dollar. Tether is designed to maintain a 1:1 ratio with the US dollar, providing stability to

users who want to transact in cryptocurrencies without being exposed to the volatility of the crypto markets. Tether is widely used in cryptocurrency trading, with many exchanges allowing users to trade USDT pairs for other cryptocurrencies.

They are also used in decentralized finance (DeFi) applications, allowing users to access various financial services on the blockchain, such as lending, borrowing, and trading. Additionally, ERC-20 tokens can be used for reward programs, gaming, and identity management systems. The demonstrated ability of SpartanScript to interface with the ERC-20 standard on the SpartanGold blockchain provides evidence that utilizing an interpreted scripting language in place of a more complex virtual machine is a feasible approach.

CHAPTER 6

Conclusion

This research project has proposed a new scripting language SpartanScript for smart contracts that aims to address some of the current challenges faced in developing smart contracts on blockchain platforms. The proposed language has been designed to run natively on the blockchain, without the need for a virtual machine, providing greater security and efficiency. The language also features constructs specifically tailored for smart contracts, enabling developers to write more reliable and robust contracts. By integrating SpartanScript into the experimental blockchain SpartanGold, it will become simpler to execute smart contracts on SpartanGold, while also facilitating innovation and the introduction of new concepts to the platform.

For example, it can prevent reentrancy attacks, which occur when a contract is called recursively before the previous call has completed. Additionally, it can prevent certain immutable bugs that occur due to the inherent complexity of smart contract programming. Using a scripting language can also simplify the development process for smart contracts, making it easier for developers to write and deploy secure code on the blockchain. Finally, using a scripting language can reduce the computational complexity of running smart contracts on the blockchain, leading to more efficient and scalable decentralized applications.

Overall, the development of a new scripting language for smart contracts has the potential to drive innovation in the blockchain industry, enabling the creation of more sophisticated and powerful decentralized applications. The proposed language provides a promising solution to some of the security challenges faced by existing virtual machines, while also offering a more accessible and streamlined development process for smart contracts. This research project provides a valuable contribution to the ongoing development of blockchain technology, and opens up new possibilities for

the widespread adoption of decentralized applications and smart contracts.

LIST OF REFERENCES

[1] S. Nakamoto, ''Bitcoin: A peer-to-peer electronic cash system,'' *Decentralized business review*, p. 21260, 2008.

[2] B. K. Mohanta, S. S. Panda, and D. Jena, ''An overview of smart contract and use cases in blockchain technology,'' in *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*. IEEE, 2018, pp. 1--4.

[3] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, ''An overview on smart contracts: Challenges, advances and platforms,'' *Future Generation Computer Systems*, vol. 105, pp. 475--491, 2020.

[4] V. Buterin *et al.*, ''A next-generation smart contract and decentralized application platform,'' *white paper*, vol. 3, no. 37, pp. 2--1, 2014.

[5] N. Atzei, M. Bartoletti, and T. Cimoli, ''A survey of attacks on ethereum smart contracts (sok),'' in *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer, 2017, pp. 164--186.

[6] G. Wood *et al.*, ''Ethereum: A secure decentralised generalised transaction ledger,'' *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1--32, 2014.

[7] Ethereumbook, ''Ethereumbook/13evm.asciidoc at develop · ethereumbook/ethereumbook,'' Jan 2021. [Online]. Available: https://github.com/ethereumbook/ethereumbook/blob/develop/13evm.asciidoc

[8] ''Solidity.'' [Online]. Available: https://docs.soliditylang.org/en/develop/#

[9] ''Opcodes for the evm.'' [Online]. Available: https://ethereum.org/en/developers/docs/evm/opcodes/

[10] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, ''An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks,'' in *Information Security Practice and Experience: 13th International Conference, ISPEC 2017, Melbourne, VIC, Australia, December 13--15, 2017, Proceedings 13*. Springer, 2017, pp. 3--24.

[11] E. G. Sirer, ''Thoughts on the dao hack.'' [Online]. Available: http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/

[12] G. J. Sussman and G. L. Steele Jr, ‘‘Scheme: A interpreter for extended lambda calculus,’’ *Higher-Order and Symbolic Computation*, vol. 11, no. 4, pp. 405--439, 1998.

[13] G. L. Steele Jr and G. J. Sussman, ‘‘The art of the interpreter or the modularity complex (parts zero, one, and two).’’ MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, Tech. Rep., 1978.

[14] T. Austin, ‘‘Taustin/spartan-gold: A simplified blockchain-based cryptocurrency for experimentation.’’ [Online]. Available: https://github.com/taustin/spartan-gold

[15] G. Estevam, L. M. Palma, L. R. Silva, J. E. Martina, and M. Vigil, ‘‘Accurate and decentralized timestamping using smart contracts on the ethereum blockchain,’’ *Information Processing & Management*, vol. 58, no. 3, p. 102471, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306457320309602

[16] K. Fan, S. Sun, Z. Yan, Q. Pan, H. Li, and Y. Yang, ‘‘A blockchain-based clock synchronization scheme in iot,’’ *Future Generation Computer Systems*, vol. 101, pp. 524--533, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18326657

[17] F. Vogelsteller and V. Buterin, ‘‘Eip 20: Erc-20 token standard,’’ *Ethereum Improvement Proposals*, vol. 20, 2015.

[18] ‘‘Erc 20.’’ [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/token/erc20

[19] P. Cuffe, ‘‘The role of the erc-20 token standard in a financial revolution: the case of initial coin offerings,’’ in *IEC-IEEE-KATS Academic Challenge, Busan, Korea, 22-23 October 2018*. IEC-IEEE-KATS, 2018.

[20] [Online]. Available: https://assets.ctfassets.net/vyse88cgwfbl/5UWgHMvz071t2Cq5yTw5vi/c9798ea8db99311bf90ebe0810938b01/TetherWhitePaper.pdf