VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PRINCIPLES OF PROGRAMMING LANGUAGES - CO3005

ZCODE SPECIFICATION

Version 1.0.0

HO CHI MINH CITY, 12/2023

ZCODE SPECIFICATION

Version 1.0.0

1 Introduction

ZCode is a general-purpose block-structured procedural programming language to serve as a tool for students practicing the implementation of a basic compiler designed for a simple programming language.

Despite its straightforwardness, **ZCode** incorporates the core elements crucial to a procedural programming language, where the program is organized into functions or procedures.

2 Program structure

Due to its simplicity, the **ZCode** compiler lacks the ability to compile multiple files. This means that a **ZCode** program must be contained within a single file.

Within this file, numerous declarations are present (as discussed in subsection 2.1). The starting point of a **ZCode** program is a unique function named **main**. This function takes no parameters and returns nothing. **All declarations and statements in this programming language must end with a newline character.**

3 Lexical structure

3.1 Characters set

A ZCode program is a sequence of characters from the ASCII characters set. Blank (' '), tab ('\t'), backspace (('\b')), form feed (i.e., the ASCII FF - '\f'), carriage return (i.e., the ASCII CR – '\r') and newline (i.e., the ASCII LF – '\n') are whitespace characters. The '\n' is used as newline character in ZCode.

This definition of lines can be used to determine the line numbers produced by a ZCode compiler.

3.2 Program comment

There is a single-line comment in ZCode starting with a "##" ignores all characters until the end of the current line, i.e, when reaching end of line or end of file. For example:

```
## This is a single comment.
```

```
a <- 5
```

ZCode does not allow the combination of comment and other declarations and statements. It appears with only a single line in source code.

For example:

```
a <- 5 ## This comment is not allowed in ZCode.
```

3.3 Identifiers

Identifiers are used to name variables, functions and parameters. Identifiers begin with a letter (A-Z or a-z) or underscore (_), and may contain letters, underscores, and digits (0-9). ZCode is case-sensitive, therefore the following identifiers are distinct: PrintLn, println, and PRINTLN.

3.4 Keywords

Keywords must begin with a lowercase letter (a-z). The following keywords are allowed in ZCode:

`true`	`false`	`number`	`bool`	`string`
`return`	`var`	`dynamic`	`func`	
`for`	`until`	`by`	`break`	`continue`
`if`	`else`	`elif`	`begin`	`end`
`not`	`and`	`or`		

3.5 Operators

The following is a list of valid operators:

`+`	`-`	`*`	`/`	`%`
`not`	`and`	`or`	`=`	`<-`
`!=`	`<`	`<=`	`>`	`>=`
`...`	`==`			

The meaning of those operators will be explained in the following sections.

3.6 Separators

The following characters are the separators:

```
( ) [ ] ,
```

3.7 Literals

A literal is a source representation of a value of a number, boolean, string.

1. **Number**: All numbers in ZCode are considered as float number in C/C++, just as real numbers. A ZCode number consists of an integer part, a decimal part and an exponent part. The integer part is a sequence of one or more digits. The decimal part is a decimal point followed optionally by some digits. The exponent part starts with the character '`e`' or '`E`' followed by an optional '`+`' or '`-`' sign, and then one or more digits. The decimal part or the exponent part can be omitted.

   For example:

   ```
   0       199     12.     12.3    12.3e3      12.3e-30
   ```

   All numbers are in decimal and not represented for other number systems.

2. **Boolean**: A **boolean** literal is either `true` or `false`, formed from ASCII letters.

3. **String**: A **string** literal includes zero or more characters enclosed by double quotes ("). Use escape sequences (listed below) to represent special characters within a string. Remember that the enclosing double quotes are not part of the string. It is a compile-time error for a new line or EOF character to appear after the opening (") and before the closing matching (").

   All the supported escape sequences are as follows:

   \b backspace
   \f form feed
   \r carriage return
   \n newline
   \t horizontal tab
   \' single quote
   \\ backslash

   For a double quote (") inside a string, a single quote (') must be written before it: '" double quote

   For example:

   "This is a string containing tab \t"

   "He asked me: '"Where is John?'""

4 Type and Value

There are three primitive (or scalar) data types in ZCode (number, boolean, string) and array type.

4.1 Boolean type

Each value of type boolean can be either `true` or `false`.

`if` and other control statements work with boolean expressions.

The operands of the following operators are in boolean type:

```
not       and       or
```

4.2 Number type

A value of number may be positive or negative. Only these operators can act on number values:

```
+         -         *         /         %
=         !=
>         >=        <         <=
```

In the case of calculating the remainder r of the division of two real numbers a and b, it is computed using the formula:

$$r = a - b * \lfloor a/b \rfloor$$

where $\lfloor a/b \rfloor$ is the integer part of the division of two real numbers a and b.

For example: $7.5\%3.5 = 0.5$, $7.8\%3.38 = 1.04$

4.3 String type

The operands of the following operators are in string type:

```
...           ==
```

4.4 Array type

ZCode supports multi-dimensional arrays.

- All elements of an array must have the same type which can be **number, string, boolean**.
- In an array declaration, a number literals must be used to represent the number (or the length) of one dimension of that array. If the array is multi-dimensional, there will be

more than one number literals. These literals will be separated by comma and enclosed by square bracket ([ ]).

For example:

```
number a[5] <- [1, 2, 3, 4, 5]
number b[2, 3] <- [[1, 2, 3], [4, 5, 6]]
```

An **array** value is a comma-separated list of literals enclosed in '[' and ']'. The literal elements are in the same type.

For example, [1, 5, 7, 12] or [[1, 2], [4, 5], [3, 5]].

- The lower bound of one dimension is always 0.

5 Expressions

Expressions are constructs which are made up of operators and operands. Expressions work with existing data and return new data.

In ZCode, there exist two types of operations: unary and binary. Unary operations work with one operand and binary operations work with two operands. The operands may be variables, data returned by another operator, or data returned by a function call. The operators can be grouped according to the types they operate on. There are five groups of operators: arithmetic, logic, string, relational, index.

5.1 Arithmetic operators

Standard arithmetic operators are listed below.

Operator	Operation	Operand's Type
-	Number sign negation	number
+	Number Addition	number
-	Number Subtraction	number
*	Number Multiplication	number
/	Number Division	number
%	Number Remainder	number

All arithmetic operations result in a number type.

5.2 Logic operators

Logic operators include logical **NOT**, logical **AND** and logical **OR**. All logic operations result in a boolean type. The operation of each is summarized below:

Operator	Operation	Operand's Type
not	Negation	boolean
and	Conjunction	boolean
or	Disjunction	boolean

5.3 String operators

Standard string operators are listed below.

Operator	Operation	Operand's Type
...	String concatenation	string

This operation result in a string type.

5.4 Relational operators

Relational operators perform arithmetic and literal comparisons. All relational operations result in a boolean type. Relational operators include:

Operator	Operation	Operand's Type
=	Equal	number
!=	Not equal	number
<	Less than	number
>	Greater than	number
<=	Less than or equal	number
>=	Greater than or equal	number
==	Compare two same strings	string

5.5 Index operators

An **index operator** is used to reference or extract selected elements of an array. It must take the following form:

```
element_expression -> expression [ index_operators ]
index_operators -> expression
                 | expression , index_operators
```

The *expression* between '[' and ']' must be of number type. The type of the expression (in the first production) must be an array type so the expression can be an identifier or a function call. The index operator returns the element of the array variable whose index is the expression. The operator has the highest precedence.

For example:

```
a[3 + foo(2)] <- a[b[2, 3]] + 4
```

5.6 Function call

The function call starts with an identifier (which is a function's name), then an opening parenthesis, a comma-separated list of arguments (this list could be empty), and a closing parenthesis. The value of a function call is the returned value of the callee function.

5.7 Operator precedence and associativity

The order of precedence for operators is listed from high to low:

Operator Type	Operator	Arity	Position	Association
Index operator	[,]	Unary	Postfix	Left
Sign	-	Unary	Prefix	Right
Logical	`not`	Unary	Prefix	Right
Multiplying	*, /, %	Binary	Infix	Left
Adding	+, -	Binary	Infix	Left
Logical	`and`, `or`	Binary	Infix	Left
Relational	=, ==, !=, <, >, <=, >=	Binary	Infix	None
String	...	Binary	Infix	None

6 Variables and Function

In a ZCode program, all variables must be declared before the first usage. A name cannot be re-declared in the same scope, but it can be visible in nested scopes. When a name is re-declared by another declaration in a nested scope, it is hidden in the nested scope.

6.1 Variables

There are three kinds of variables: **global variables, local variables and parameters of functions**. A variable declaration starts with the keyword as the name of type (`number`, `bool`,

`string`) or the implicit keyword (`var`, `dynamic`). It is followed by a single declaration, consists of an identifier and an optional value initialization.

- If the starting word is `var`, the value initialization is obligatory. It starting with an assignment sign (`<-`) and an expression. The type of expression is assigned to variable at the compile time.
- When a variable is declared with `dynamic`, The type of a variable can be known at the compile time by the type inference technique. The ability to infer types automatically makes many programming tasks easier, leaving the programmers free to omit type annotations while maintaining some level of type safety.

If a variable is declared with array type, the list of fixed emonstrated after the name and enclosed in the square bracket. Its following form is `<type> <variable-name>[size-1, size-2, size-3, ..., size-n]`. Each size should be a number literal. The implicit keyword cannot be used for array type.

1. **Global variables**: global variables are those declared outside any function in the program. Global variables are visible from the place where they are declared to the end of the program.
2. **Local variables**: Local variables are those declared inside functions (i.e., inside the body of the functions). They are visible inside the block where they are declared and all nested blocks.

   The following fragment of code is legal:

```
func foo(number a[5], string b)
begin
    var i <- 0
    for i until i >= 5 by 1
    begin
        a[i] <- i * i + 5
    end
    return -1
end
```

3. **Parameters**: In ZCode, a parameter in array type will be passed by reference while a parameter in another type will be passed by value.

   In case of passing by value, the callee function is given its value in its parameters. Thus, the callee function cannot alter the arguments in the calling function in any way. When a

function is invoked, each of its parameters is initialized with the corresponding argument's value passed from the caller function.

In case of passing by reference of array, the parameter in the callee function is given the address of its corresponding argument. Therefore, a modification in an element of the parameter really happens in the corresponding element of the argument.

Formal parameters are variables with the function scope (in section 8.2).

6.2 Function

A function is the unit to structure the program in ZCode. Every function starts with the keyword `func`, following by an identifier, a nullable list of parameters enclosed by a round bracket `()` and optionally, ends up with a return statement or a block statement.

Similar to C/C++, the language allows functions to have a declaration-only part, but afterward, there must be a complete implementation. The function's type will be inferred from the return statement; if there is no return statement, the function will not return anything.

As the first introduction, the ZCode must have the function `main` for starting entry point.

7 Statements

A statement indicates the action a program performs. There are many kinds of statements, as described as follows:

7.1 Variable declaration statement

Variable declaration statement should be the same as the variable declarations (described in the previous section).

7.2 Assignment Statement

An assignment statement assigns a value to a left hand side which can be a scalar variable, an index expression. An assignment takes the following form:

```
lhs <- expression
```

where the value returned by the `expression` is stored in the the left hand side `lhs`, which can be a local variable or an element of an array.

The type of the value returned by the expression must be compatible with the type of lhs. The following code fragment contains examples of assignment:

```
aPI <- 3.14
value <- x.foo(5)
l[3] <- value * aPi
```

7.3 If statement

The **if statement** conditionally executes one of some lists of statements based on the value of some boolean expressions. The if statement has the following forms:

```
if (expression-1) <statement-1>
[elif (expression-2) <statement-2>]?
[elif (expression-3) <statement-3>]?
[elif (expression-4) <statement-4>]?
...
[else <else-statement>]?
```

where the first `expression` evaluates to a boolean value. If the `expression` results in true then the `statement-1` will be executed.

If `expression-1` evaluates to false, the `expression-2`, if any, will be calculated. The corresponding statement is executed if the value of the expression is true.

If all previous expressions return false then the `else-statement` following `else`, if any, is executed. If no `else` clause exists and expression is false then the if statement is passed over.

There are zero or many `elif` parts while there is zero or one `else` part.

7.4 For statement

In general, **for statement** allows repetitive execution of <**statement**>. For statement executes a loop for a predetermined number of iterations. For statements take the following form:

```
for <number-variable> until <condition expression> by <update-expression>
    <statement>
```

`<number-variable>` must be a scalar variable of data type number. The loop will iterate the variable `<number-variable>`, check the `<condition expression>`, and if it evaluates to true, it will terminate; otherwise, it will execute the `<statement>`. After each iteration, the value of `<number-variable>` will be incremented by `<update-expression>`. Upon completion or abrupt termination of the loop using break, the value of `<number-variable>` will retain its value as it was before the loop.

```
var i <- 0
for i until i >= 10 by 1
    print i
```

7.5 Break statement

Using the **break statement**, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. It must reside in a loop. Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A break statement has the following form:

```
break
```

7.6 Continue statement

The **continue statement** causes the program to skip the rest of the loop body in the current iteration as if the end of the statement block had been reached. It must reside in a loop . Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A continue statement has the following form:

```
continue
```

7.7 Return statement

A **return statement** aims at transferring control and data to the caller of the function that contains it. The return statement starts with keyword `return` which is optionally followed by an expression and ends with a semi-colon.

7.8 Function call statement

The function call statement starts with an identifier (which is a function's name), then an opening parenthesis, a comma-separated list of arguments (this list could be empty), and a closing parenthesis. The value of a function call is the returned value of the callee function.

7.9 Block statement

A **block statement** begins by the keyword `begin` and ends up with the keyword `end`. In the body of block statement, there may be a nullable list of statements.

For example:

```
begin
    number r
    number s
    r <- 2.0
```

```
    number a[5]
    number b[5]
    s <- r * r * 3.14
    a[0] <- s
end
```

8 Scope

There are 3 levels of scope: global, function and block.

8.1 Global scope

All function names and variable names outside the block of functions have global scope.

8.2 Function scope

All parameters declared in the function have the function scope. They are visible from the places where they are declared to the end of the enclosing function unless another variables is declared with the same name in body block statement and it nested blocks.

8.3 Block scope

All variables declared in a block have the block scope, i.e., they are visible from the place they are declared to the end of the block.

9 IO

To perform input and output operations, For convenience, CSEL provides the following built-in functions:

Function	Semantic
readNumber	Read an number from keyboard and return it.
writeNumber(anArg)	Write an number to the screen.
readBool()	Read an boolean value from keyboard and return it.
write(anArg)	Write an boolean value to the screen.
readString()	Read an string from keyboard and return it.
writeString(anArg)	Write an string to the screen.

These function names are in global scope.

10 Examples of Correct Source Code in ZCode

10.1 Source code 1

```
func areDivisors(number num1, number num2)
    return (num1 % num2 = 0 or num2 % num1 = 0)

func main()
    begin
        var num1 <- readNumber()
        var num2 <- readNumber()

        if areDivisors(num1, num2) printString("Yes")
        else printString("No")
    end
```

10.2 Source code 2

```
func isPrime(number x)

func main()
    begin
        number x <- readNumber()
        if isPrime(x) printString("Yes")
        else printString("No")
    end

func isPrime(number x)
    begin
        if x <= 1 return false
        var i <- 2
        for i until i > x / 2 by 1
        begin
            if x % i = 0 return false
        end
        return true
    end
```

11 Changelog