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

D95 SPECIFICATION

Version 1.2

HO CHI MINH CITY, 06/2021

D95'S SPECIFICATION

Version 1.2

1 Introduction

D95 is a programming language in which developers do not associate type for each variable declaration. Like other programming languages, D95 contains a few primitive types (integer, float, boolean), array type, conditional structures (if-else), iteration structures (for, while) ...

In D95, functions can be called recursively. Other features of D95 will be discussed in details below.

2 Program structure

D95 does not support separate compilation so all declarations (variables, constants and functions) must be resided in one single file.

Just like C, a D95 program should begin with a (global) variable/ constant declaration or an function declaration. The entry of the program is a special function call *main*.

3 Lexical structure

3.1 Characters set

A D95 program is a sequence of characters from the ASCII characters set. Blank (' '), tab ('\t'), 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 D95.

This definition of lines can be used to determine the line numbers produced by a D95 compiler.

3.2 Program comment

In D95 programming language, there is only one type of comment: block comment. All characters in a block comment will be ignored. A block comment is delimited by `/*` and `*/` like:

```
/* This is a
   multi-line
  comment.
*/
```

3.3 Tokens set

A token is a sequence of one or more characters in source code, that when grouped together, acts as a single atomic unit of the language. In the D95 programming language, there are five kinds of tokens: *identifiers, keywords, operators, separators and literals.*

3.3.1 Identifiers

There are 3 (three) types of identifiers:

- **Variable/ Parameter identifiers**: must begin with a dollar sign ($) and follow by letters (A-Z or a-z), underscores ('_') and digits (0-9). Variable names and parameter names are examples of these identifiers.
- **Constant identifiers**: must begin with a uppercase letters (A-Z) and follow by letters (A-Z or a-z), underscores ('_') and digits (0-9). Constant names are examples of these identifiers.
- **Function name identifiers**: must begin with an underscore (_) and follow by letters (A-Z or a-z), underscores ('_') and digits (0-9). Function names are examples of these identifiers.

D95 identifiers are **case-sensitive**, therefore the following identifiers are distinct: writeLn, writeln and wRITELN. Variable names and constant names are examples of identifiers.

3.3.2 Keywords

Keywords must begin with a lowercase letter (A-Z). The following keywords are allowed in D95:

`break`	`continue`	`if`	`elseif`	`else`
`while`	`foreach`	`as`	`function`	
`true`	`false`	`array`	`define`	

3.3.3 Operators

The following is a list of valid operators:

+	-	*	/	%
!	&&	||	==	=
!=	>	<=	>	>=
==.	+.			

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

3.3.4 Seperators

The following characters are the separators:

```
( ) [ ] . , ; { }
```

3.3.5 Literals

A literal is a source representation of a value of a integer, float, boolean, string, one of three types of array.

1. **Integer**: Integer can be specified in decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2) notation:

    - To use octal notation, precede the number with a 0 (zero).
    - To use hexadecimal notation precede the number with 0x or 0X.
    - To use binary notation precede the number with 0b or 0B.

    Integer literals may contain underscores (_) between digits, for better readability of literals in decimal. These underscores are removed by D95's scanner.

    For example:

    ```
    1234    0123    0x1A       0b11111111      1_234_567
    ```

2. **Float**: A floating number consists of three components: integer, decimal and exponent part, respectively. Exponent part can be omitted if there exists both of remaining components. Otherwise, only one of them can be absent for this representation.

    - Integer part is just like an integer literal but used in only demical form.
    - Decimal part starts with an point (.) following by a representation like integer part or an empty.
    - Exponent part begins with a charater `e` or `E` and then an optinal sign (`-`, `+`). It finalized with an integer-part-like form.

    For instance:

    ```
    1.234   1.2e3   7E-10   1_234.567
    ```

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

4. **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 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?’""

5. **Indexed Array**: An **indexed array** literal is a comma-separated list of literals (with an array) enclosed in ’(’ and ’)’ and started with keyword **array**. The literal elements are in the same type.

   For example, `array(1, 5, 7, 12)` or `array("Kangxi", "Yongzheng", "Qianlong")`

6. **Associative Arrays**: An **associative array** literal is a comma-separated list of key-value pairs in format `key => value`. The key can either be an int or a string. The value can be of any type.

   For example:

   ```
   array(
       1 => "abc"
       "15" => 15 + 16.1
       "place" => "Yanxi"
   )
   ```

   ,

7. **Multi-dimensional arrays** are such type of arrays which stores an another array at each index instead of single element. In other words, define multi-dimensional arrays as array of arrays.

   For example:

```
array (
    array("Volvo", "22", "18"),
    array("a" => "BMW", "b" => 15, "c" => 13),
    array("Saab", "5", "2"),
    array("Land Rover", "17", "15")
)
```

4 Type and Value

In D95, the programmers are not required to associate each variable with a particular type. 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.

There are four primitive (or scalar) data types in D95 (int, float, boolean, string) and three composite types (indexed/ multidimensional/ associative array).

4.1 Boolean type

Each value of type boolean can be either `True` or `False`.

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

The operands of the following operators are in boolean type:

```
!       &&      ||
```

4.2 Number type

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

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

4.3 String type

The operands of the following operators are in string type:

```
+.      ==.
```

4.4 Array type

D95 supports indexed/ associative/ multidimensional arrays.

- All elements of an indexed array must have the same type which can be **int, float, string, boolean**.
- Multidimensional arrays are, in other word, arrays in indexed array.
- The lower bound of one dimension in indexed/multidimensional is always 0.

5 Variables, Constants and Function

In a D95 program, all variables and constants must be declared before usage. A name cannot be re-declared in the same scope, but it can be reused in other scopes. When a name is re-declared by another declaration in a nested scope, it is hidden in the nested scope.

5.1 Variables

There are three kinds of variables: **global variables, local variables and parameters of functions**. Variables in D95 are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive.

Variable names follow the same rules as other labels in D95 in section 3.3.1. **Expect declarations for parameter, all variable declarations always have the initalized expression with a seperation by equal sign (=) and finalized with a semicolon (;).** This means the first usage of the variable is the declaration of variable (just like Python).

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 list where they are declared and all nested lists.

   The following fragment of code is legal:

```
function _sum($x, $y) {
    $z = $x + $y;
    return $z;
}
```

3. **Parameters**: In D95, an array is always passed by reference while other arguments is 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 type, 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 local variables whose scope is the enclosing function.

5.2 Constants

A constant is an identifier (name) for a simple value. As the name suggests, that value cannot change during the execution of the script. Constants are case-sensitive. In D95, constants defined using the `define()` pseudo-function with the parameters are respectively a constant identifier and the initial value. The name of a constant follows the same rules as any label in D95. A valid constant name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. Constant names follow the same rules as other labels in D95 in section 3.3.1.

The initial value of constant must be an expression with type inferred before.

Every constant declaration is in the global scope and before all other kinds of declaration (variables and functions).

The following fragment of code is legal:

```
define(A, 10);
define(B, "Story of Yanxi Place");

function _foo($a, $b) {
    $i = 0;
    while (i < 5) {
        A[i] = ($b + 1) * $a;
        $u = i + 1;
        if (a[u] == 10) {
            return $b;
        }
        i = i + 1;
    }
    return B + ": Done";
}
```

5.3 Function

A function is the unit to structure the program in D95. Every function starts with the keyword `function`, following by an function name. A valid function name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. A following list of parameter is enclosed by a round bracket `()`. Each parameter should be a variable. Function names follow the same rules as other labels in D95 in section 3.3.1.

As the first introduction, the D95 must have the function `_main` for starting entry point.

6 Expressions

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

In D95, 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 constants, 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, boolean, relational, index and key.

6.1 Arithmetic operators

Standard arithmetic operators are listed below.

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

6.2 Boolean operators

Boolean operators include logical **NOT**, logical **AND** and logical **OR**. The operation of each is summarized below:

Operator	Operation	Operand's Type
!	Negation	boolean
&&	Conjunction	boolean
||	Disjunction	boolean
==.	Compare two same strings	string

6.3 String operators

Standard string operators are listed below.

Operator	Operation	Operand's Type
+.	String concatenation	string

6.4 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	int/float
!=	Not equal	int/float
<	Less than	int/float
>	Greater than	int/float
<=	Less than or equal	int/float
>=	Greater than or equal	int/float

6.5 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 int type (for indexed/multidimensional arrays) or string type (for associative arrays). The type of the expression (in the first production) must be an array type so the expression can be an identifier, a function call or a reference to a value in JSON object. The index operator returns the element of the array variable whose index or key is the expression. The operator has the highest precedence.

For example:

```
$a[3 + _foo(2)] = $a[b[2]["place"]] + 4;
```

6.6 Function call

The function call starts with the functional name, an opening parenthesis ('('), the nullable comma-separated parameters value list anda closing parenthesis (')').The value of a function call is the returned value of the callee function.

For example:

```
$a[2] = _foo(2) + _foo(bar(2, 3));
```

6.7 Operator precedence and associativity

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

Operator Type	Operator	Arity	Position	Association
Function call	`Function name`	Unary	Prefix	None
Index operator	[,]	Unary	Postfix	Left
Sign	-	Unary	Prefix	Right
Logical	!	Unary	Prefix	Right
Multiplying	*, /, %	Binary	Infix	Left
Adding	+, -	Binary	Infix	Left
Logical	&&, ||	Binary	Infix	Left
Relational	==, !=, <, >, <=, >=	Binary	Infix	None
String	+., ==.	Binary	Infix	None
Assocative	=>	Binary	Infix	None

The expression in parentheses has highest precedence so the parentheses are used to change the precedence of operators.

6.8 Type coercions

In D95, the type coercions must be explicitly used to satisfy the type constraint of operators. The following functions can be used for type conversion:

- str2int: convert the given string to an integer.
- int2str: return the string representation of an integer.
- str2float: convert the given string to a float.
- float2str: return the string representation of a float.
- str2bool: convert the given string to a bool.
- bool2str: return the string representation of a boolean.

6.9 Evaluation orders

D95 requires the left-hand operand of a binary operator must be evaluated first before any part of the right-hand operand is evaluated. Similarly, in a function call, the actual parameters must be evaluated from left to right.

Every operands of an operator must be evaluated before any part of the operation itself is performed. The two exceptions are the logical operators && and ||, which are still evaluated from left to right, but it is guaranteed that evaluation will stop as soon as the truth or falsehood is known. This is known as the **short-circuit evaluation**. We will discuss this later in detail (code generation step).

7 Statements

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

7.1 Variable and Constant Declaration Statement

Variable and constant declaration statement should be grammatically described in section 5. In D95, every variable and constant will be created singly in a command. Each object must be associated with an initialized expression/ value at the time of creation.

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`.

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) { statement-list }
elseif (expression) { statement-list }
elseif (expression) { statement-list }
elseif (expression) { statement-list }
...
else { statement-list }
```

where the first `expression` evaluates to a boolean value. If the `expression` results in true then the statement-list following the reserved word Then is executed.

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

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

The statement-list includes zero or many statements.

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

7.4 Foreach statement

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

```
foreach (array as key => value) { statement-list }
```

`array` is a variable/constant (form in section 5) to iterate over the `array`. In Foreach statement, `array` must be an array type. `key` and `value` will be corresponding for each pair in associate arrays with the key should be the index in indexed/ multidimensional arrays. The value of `key` and `value` can be changed but it do not effect on the element of array (like pass-by-value mechanism).

```
$a = array(1, 2, 3);
foreach ($a as $key => $value) {
    _echo((("Element " +. int2str($key)) +. ": ") +. int2str($value));
}
```

This prints out:

```
Element 0: 1
Element 1: 2
Element 2: 3
```

7.5 While statement

The **while statement** executes repeatedly nullable statement-list in a loop. While statements take the following form:

```
while (expression) { statement-list }
```

where the `expression` evaluates to a boolean value. If the value is true, the while loop executes repeatedly the list of statement until the expression becomes false.

7.6 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;
```

7.7 Continue statement

The **continue statement** causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. 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;
```

7.8 Call statement

A **call statement** is like a function call except that it does not join to any expression and is always terminated by a semicolon.

For example:

```
_foo(2 + $x, 4 / $y);
_goo();
```

7.9 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.

A return statement must appear within a function.

8 Functions

Functions are a sequence of instructions that are separate from the main code block. A function may return a value or not.

Functions may be called from one or more places throughout your program. Functions make source code more readable and reduce the size of the executable code because repetitive blocks of code are replaced with calls to the corresponding function. The parameters of function allow the calling routine to communicate with the function. As discussed in section 5, the parameters are passed by value, except the array type parameters.

For convenience, D95 provides the following built-in functions:

- `_echo(arg)`: print string `arg` to the screen.
- `_read()`: read a string from the input.

9 Change log

- Change all rules for identifiers to be clear in 3.3.1.
- Change some examples to be followed by the identifier rule in 3.3.1.
- Initial value of constant.
- Clear the single line comment.
- Modify the rule for variable declaration.
- Clarify the rule for constant declaration.