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

CSEL SPECIFICATION

Version 1.0

HO CHI MINH CITY, 03/2021

CSEL'S SPECIFICATION

Version 1.0

1 Introduction

CSEL is a programming language in which developers optionally associate type for each variable declaration. Like other programming languages, BKIT contains a few primitive types (integer, float, boolean), array type, conditional structures (if-else), iteration structures (for, while) ...

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

2 Program structure

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

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

3 Lexical structure

3.1 Characters set

A CSEL 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 CSEL.

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

3.2 Program comment

In BKIT 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 single comment. ##
## 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 CSEL programming language, there are five kinds of tokens: *identifiers, keywords, operators, separators and literals*.

3.3.1 Identifiers

Identifiers must begin with a dollar sign ($) or a lower case letter (a-z), but the following characters may only contain letters (A-Z or a-z), underscores ('_') and digits (0-9). CSEL identifiers are **case-sensitive**, therefore the following identifiers are distinct: writeLn, writeln and wRITELN. Variable names and function names are examples of identifiers.

3.3.2 Keywords

Keywords must begin with a upper case letter (A-Z). The following keywords are allowed in CSEL:

`Break`	`Continue`	`If`	`Elif`	`Else`
`While`	`For`	`Of`	`In`	`Function`
`Let`	`True`	`False`	`Call`	`Return`
`Number`	`Boolean`	`String`	`JSON`	`Array`
`Constant`				

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 number, boolean, string, array or JSON (Javascript Object Notation) type.

1. **Number**: All numbers in CSEL are considered as float number in C/C++, just as real numbers. A CSEL number consists of an integer part, a decimal part and an exponent part. The integer part is a sequence of one or more digits with an optional '`-`' sign if negative. 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 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. **Array**: An **array** literal 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]].

5. **JSON**: A JSON literal is a comma-separeted list of key-value objects enclosed in '{' and '}'. An key-value object consists of key part and value part. A key part is an identifier which is described in 3.3.1 while value part is a literal which can be in a number, boolean, string, array and JSON type.

   For example:

```
{
    name: "Yanxi Place",
    address: "Chinese Forbidden City",
    surface: 10.2,
    people: ["Empress Xiaoxianchun", "Yongzheng Emperor"]
}
```

4 Type and Value

In CSEL, 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 BKIT (number, boolean, string) and two composite types (array and JSON).

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

CSEL supports multi-dimensional arrays.

- All elements of an array must have the same type which can be **number, string, boolean or JSON**.
- In an array declaration, an integer 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 integer literals. These literals will be separated by comma and enclosed by square bracket ([ ]).

  For example:

```
let a[5]: int = [1, 2, 3, 4, 5];
let b[2, 3] = [[1, 2, 3], [4, 5, 6]];
```

- The lower bound of one dimension is always 0.

4.5 JSON type

CSEL supports nested JSON objects.

- Any value with each key can be **number, string, boolean, array or JSON**.
- To get or set value with each key in JSON object, a string literals must be used to represent the key and enclosed by square bracket ([ ]).

  For example:

```
Let a = {
    name: "Yanxi Place",
    address: "Chinese Forbidden City",
    surface: 10.2,
    people: ["Empress Xiaoxianchun", "Yongzheng Emperor"]
};
```

```
# Set new value for name in object a
a["name"] = "Qianqing Palace";

# Get value by using key in object a
let b: String = a["name"] + a["address"];
```

5 Variables and Constants

In a CSEL program, all variables and constants must be declared before first 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**.

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 foo(a[5], b) {
    Let i = 0;
    While (i < 5) {
        a[i] = b + 1;
        let u: Integer = i + 1;
        If (a[u] == 10) {
            Return a[i];
        }
    }
    Return -1;
}
```

3. **Parameters**: In CSEL, an array or JSON variable 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 or JSON 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

There are three kinds of constants: **global constants and local constants**. The name associated with a constant is only utilized to read the value so any declaration has the initial value.

1. **Global constants**: global constants are those declared outside any function in the program. Global constants are visible from the place where they are declared to the end of the program.

2. **Local constants**: Local constants 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:

```
Constant $a = 10;
Function foo(a[5], b) {
    Constant $b: String = "Story of Yanxi Place";
    Let i = 0;
    While (i < 5) {
        a[i] = (b + 1) * $a;
        let u: Integer = i + 1;
        If (a[u] == 10) {
            Return $b;
        }
        i = i + 1;
    }
    Return $b + ": Done";
}
```

6 Expressions

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

In CSEL, 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	number
+	Number Addition	number
-	Number Subtraction	number
*	Number Multiplication	number
%	Number Remainder	number

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

6.3 String operators

Standard string operators are listed below.

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

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 number type, recommended as integer and implicitly converted to integer if real number. 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 is the expression. The operator has the highest precedence.

For example:

```
a[3 + Call(foo, [2])] = a[b[2, 3]] + 4;
```

6.6 Key operators

Like index operator, a **key operator** is used to reference or extract the value with the key in a JSON object. It must take the following form:

```
element_expression -> expression key_operators
key_operators -> [ expression ]
                 | [ expression ] key_operators
```

The *expression* between '[' and ']' must be of string type. The type of the expression (in the first production) must be an JSON type so the expression can be an identifier, a function call, an array or a reference to a value in JSON object. The key operator returns the value in JSON whose key is the expression. The operator has the highest precedence with the index operator.

For example:

```
a[foo(2)] = a[b["name"]["first"]] + 4;
```

6.7 Function call

The function call starts with the key word `Call` (just like call the function `Call` in C/C++ or other programming languages), then an opening parenthesis ('('), the function name, the nullable comma-separated parameters value list enclosed in [] and a closing parenthesis (')'). The value of a function call is the returned value of the callee function.

For example:

```
a[2] = Call(foo, [2]) + Call(foo, [Call(bar, [2, 3])];
```

6.8 Operator precedence and associativity

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

Operator Type	Operator	Arity	Position	Association
Function call	`Call`	Unary	Prefix	None
Index, Key	[,]	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

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

6.9 Type coercions

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

- str2num: convert the given string to a number.
- num2str: return the string representation of an integer, in decimal.
- str2bool: convert the given string to a bool.
- bool2str: return the string representation of a boolean.

6.10 Evaluation orders

CSEL 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

This statement starts with the key word `Let` (with variable) and `Constant` (with constant) followed by a comma-separated list of identifiers described in the section 5.

For example:

```
let r = 10, v;
v = 23;
```

We optionally use the type's name to clarify to type of each variable or constant. This part is followed by identifier, starting with a colon (:) and the name of type. After that, the initial value part can be declared as example.

7.2 Assignment Statement

An assignment statement assigns a value to a left hand side which can be a scalar variable, an index expression or a key 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 }
Elif (expression) { statement-list }
Elif (expression) { statement-list }
Elif (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 `Elif`, 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 `Elif` parts while there is zero or one `Else` part.

7.4 For-In/ For-Of statement

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

```
For variable In array
    { statement-list }
For variable Of json
    { statement-list }
```

`variable` is an identifier to iterate over the `array` or **json**. In For/In statement, `array` must be an array type while the json type in For/Of statment. In For/Of, it iterates over the all key in order of declarations of a JSON object.

The **statement-list** includes zero or many statements. The following are example of for/in or for/of statements:

```
For i In [1, 2, 3] {
    Call(printLn, [i])
}
```

This prints out three lines:

```
1
2
3
```

Another example:

```
Let a = {
        name: "Yanxi Place",
        address: "Chinese Forbidden City",
    };
For key In a {
    Call(printLn, ["Value of " + key + ": " + a[key]])
}
```

This prints out two lines:

```
Value of name: Yanxi Place
Value of address: Chinese Forbidden City
```

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:

```
Call(foo, [2 + x, 4 / y]);
Call(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 and JSON type parameters.

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

- `print(arg)`: print string `arg` to the screen.
- `printLn(arg)`: print string `arg` and a new line to the screen.
- `read()`: read a string from the input.

9 Changelog