BKIT Specification

Version 2.2

Compiler Subject

October 18, 2020

Contents

1 Introduction

BKIT is a language in which the developers do not need to associate type for each variable declaration. Like other programming languages, BKIT contains a few primitive types (integer, float, boolean), array type, conditional structures (if-then-else), iteration structures (for, while, do-while) ...
In BKIT, functions can be called recursively.
BKIT restricts the variable declarations to be at the beginning of program (or block).
Other features of BKIT will be discussed in details below.

2 Program Structure

BKIT does not support separate compilation so all declarations (variables and functions) must be resided in one single file.
A BKIT program should begin with an optional global variable declaration part, followed by an function declaration part. The function declaration part must contain a special function call *main*, which is the entry of the program.

2.1 Global variable declaration part

The global variable declaration part consists of several (or zero) variable declarations. Each variable declaration has the form:

```
Var: variable-list;
```

The *variable-list* is a non-empty comma-separated list of variables. Each variable, which can be scalar or composite, **may** have an initial value which is a literal. A scalar variable is just an identifier while a composite is an identifier followed by one or many dimensions. Each dimension is an integer literal enclosed in square brackets. A variable is written using the following form:

```
variable = initial-value
```

The following variable declarations are valid:

```
Var: a = 5;
Var: b[2][3] = {{2,3,4},{4,5,6}};
Var: c, d = 6, e, f;
Var: m, n[10];
```

2.2 Function declaration part

The function declaration part contains several (or zero) function declarations.
A function declaration begins with the keyword ***Function***, followed by a colon (:) and the function name. If that function has parameters, after its name is the keyword ***Parameter*** with a colon, then a parameter list. The parameter list contains one or more parameters separated by commas. A parameter can be a scalar or composite without initial value. The function declaration continues with the body which contains a nullable list of statements (described in Section 7) between **Body:** (keyword **Body** and colon) and **EndBody.** (keyword **EndBody** and dot).
Below is a sample function declarations:

```
Var: x;

Function: fact
    Parameter: n
    Body:
        If n == 0 Then
            Return 1;
        Else
            Return n * fact (n - 1);
        EndIf.
    EndBody.

Function: main
    Body:
        x = 10;
        fact (x);
    EndBody.
```

3 Lexical Structure

3.1 Characters Set

A BKIT 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 white-space characters. The '\n' is used as newline character in BKIT.
This definition of lines can be used to determine the line numbers produced by a BKIT 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:

```
** This is a single-line 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 BKIT programming language, there are five kinds of tokens: identifiers, keywords, operators, separators and literals.

3.3.1 Identifiers

Identifiers must begin with a **lower case** letter (a-z), and may contain letters (A-Z or a-z), underscores ('_') and digits (0-9). BKIT is **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 BKIT:

Body	Break	Continue	Do
Else	ElseIf	EndBody	EndIf
EndFor	EndWhile	For	Function
If	Parameter	Return	Then
Var	While	True	False
EndDo			

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 Separators

The following characters are the **separators**:

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

3.3.5 Literals

A literal is a source representation of a value of an integer, float, boolean, string or array type.

- **Integer**
  An integer literal is a sequence of one or more digits. By default, integer literals are in decimal (base 10). If the value of a decimal literal is non-zero, it **must not begin with 0**. Otherwise, the literal is 0. An integer literal may be in octal or hexadecimal base when its value is non-zero so its first digit after the prefix must be non-zero.
  The following prefixes select a different base:

Prefix	Base	Digits
	Decimal (base 10)	0-9
0x, 0X	Hexadecimal (base 16)	0-9, A-F
0o, 0O	Octal (base 8)	0-7

  The following numbers are valid integer literals:
  0 199 0xFF 0XABC 0o567 0O77

- **Float**
  A float literal 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, but not both to avoid ambiguity with integer literals. The interpretation of float literals that fall outside the range of representable floating-point values is undefined. For example, some following float literals are valid: 12.0e3 12e3 12.e5 12.0e3 12000. 120000e-1

- **Boolean**
  A boolean literal is either *True* or *False*, formed from ASCII letters.

- **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?"'"

- **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}}.

4 Type and Value

In BKIT, the programmers don't need to explicitly 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 (boolean, integer, float, string) and one composite type (array).

4.1 Boolean type

Each value of type boolean can be either *True* or *False*.
if, ***while***, ***do-while*** and other control statements work with boolean expressions.
The operands of the following operators are in boolean type:

```
!     &&     ||
```

4.2 Integer type

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

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

4.3 Float type

The operands of the following operators are in float type:

```
+.     -.     *.     \.
=/=     <.     >.     <=.     >=.
```

4.4 String type

There is no operator on string type.

4.5 Array type

BKIT supports **multi-dimensional arrays**.

- All elements of an array must have the same type which can be **string, boolean, int or float**.
- 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 the square brackets ([ ]). For example:

```
Var: a[5] = {1,4,3,2,0};
Var: b[2][3]={{1,2,3},{4,5,6}};
```

- The lower bound of one dimension is always 0.

5 Variables

In a BKIT program, all variables must be declared before use. There are three kinds of variables: global variables, local variables and parameters of functions. A variable name cannot be used for any other variable in the same scope. However, it can be reused in other scopes. When a variable is re-declared by another variable in a nested scope, it is hidden in the nested scope.

5.1 Global variables

As discussed above, global variables are those declared in the global variable declaration part (i.e., outside all functions). Global variables are visible from the place where they are declared to the end of the program.

5.2 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. BKIT requires that all local variables in a list must be declared **at the beginning of the list**.
The following fragment of code is legal:

```
Function: foo
Parameter: a[5], b
Body:
    Var: i = 0;
    While (i < 5) Do
        a[i] = b +. 1.0;
        i = i + 1;
    EndWhile.
EndBody.
```

5.3 Parameters

In BKIT, an array 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 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.

6 Expressions

Expressions are constructs which are made up of operators and operands. Expressions work with existing data and return new data.
In BKIT, 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 four groups of operators: arithmetic, boolean, relational and index. The operators are also divided into three groups: boolean operators, integer operators and float operators.

6.1 Arithmetic operators

Standard arithmetic operators are listed below.

Operator	Operation	Operand's Type
−	Integer sign negation	integer
−.	Float sign negation	float
+	Integer Addition	integer
+.	Float Addition	float
−	Integer Subtraction	integer
−.	Float Subtraction	float
∗	Integer Multiplication	integer
∗.	Float Multiplication	float
\	Integer Division	integer
\.	Float Division	float
%	Integer Remainder	integer

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 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	integer
!=	Not equal	integer
<	Less than	integer
>	Greater than	integer
<=	Less than or equal	integer
>=	Greater than or equal	integer
=/=	Not equal	float
<.	Less than	float
>.	Greater than	float
<=.	Less than or equal	float
>=.	Greater than or equal	float

6.4 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 **integer type**. The type of the expression 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;
```

6.5 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.

6.6 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	[,]	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.

6.7 Type coercions

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

- int_of_float: truncate the given floating-point number to an integer.
- float_to_int: convert an integer to floating-point.
- int_of_string: convert the given string to an integer.
- string_of_int: return the string representation of an integer, in decimal.
- float_of_string: convert the given string to a float.
- string_of_float: return the string representation of a floating-point number.

- bool_of_string: convert the given string to a boolean.
- string_of_bool: return the string representation of a boolean.

For example:

```
If bool_of_string ("True") Then
    a = int_of_string (read ());
    b = float_of_int (a) +. 2.0;
EndIf.
```

6.8 Evaluation orders

BKIT 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 Declaration Statement

A **variable declaration statement** is as a following:

```
    Var: variable-list;
```

The *variable-list* has been described in the section 2.1. For example:

```
Body:
    Var: r = 10., v;
    v = (4. \. 3.) *. 3.14 *. r *. r *. r;
EndBody.
```

In a list of statements, the variable declaration statements always appear before any other kind of statements.

7.2 Assignment Statement

An **assignment statement** assigns a value to a left hand side which can be a scalar variable or 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 Then statement-list
ElseIf expression Then statement-list
ElseIf expression Then statement-list
...
Else statement-list
EndIf.
```

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 *statement-list* after the corresponding ***Then*** 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 (i.e. **ElseIf expression Then statement-list**) while there is zero or one Else part (i.e. **Else statement-list**).

7.4 For Statement

The **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 (scalar-variable = initExpr, conditionExpr, updateExpr) Do
    statement-list
EndFor.
```

First, the *initExpr* will be evaluated and assigned to *scalar-variable* whose type is only integer. Then the *conditionExpr* will be evaluated. If the *conditionExpr* is true, the *statement-list* , and then, the *updateExpr* will be calculated and added to the current value of *scalar-variable*. The process is repeatedly executed until the *conditionExpr* returns false, and the *for statement* will be terminated (i.e., the statement next to this for loop will be executed). Note that the *conditionExpr* must be of **boolean type**.
The **statement-list** includes zero or many statements. The following are examples of for statement:

```
For (i = 0, i < 10, 2) Do
    writeln(i);
EndFor.
```

7.5 While Statement

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

```
While expression Do statement-list EndWhile.
```

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 Do-while Statement

The **do-while statement**, much like the while statement, executes the nullable **statement-list** in a loop. Unlike the while statement where the loop condition is tested prior to each iteration, the condition of do-while statement is tested after each iteration. Therefore, a do-while loop is executed at least once. A do-while statement has the following form:

```
Do statement-list While expression EndDo.
```

where the do-while loop executes repeatedly until the expression evaluates to the **boolean** value of false.

7.7 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 (i.e., in one of the followings: a for loop, a do-while loop, or a while loop). Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A break statement has the following form:

```
Break;
```

7.8 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 (i.e., in one of the followings: a for loop, a do-while loop, or a while loop). Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A continue statement has the following form:

```
Continue;
```

7.9 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 ();
```

7.10 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 semicolon.
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 5.3, the parameters are **passed by value**, except the array type parameters.
In BKIT, functions may be called recursively.

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

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

9 Change Log

- Fix the start and end symbols (just **) of a comment in Section 3.2
- Add the array type in the first line in Section 3.3.5
- Clarify the idea of the left hand side of an assignment statement in Section 7.2
- Fix the example (adding the keyword Do) in Section 5.2