HCMC University of Technology
Faculty of Computer Science & Engineering

BKOOL

BK Object-Oriented Language

Author

Dr. Nguyen Hua Phung

August 2021

Contents

BKOOL

version 0.4

1 Introduction

BKOOL, pronounced Bi-Kool, is a mini object-oriented programming language, which is designed primarily for students practising implementing a simple compiler for a simple object-oriented language.
Despite its simplicity, BKOOL includes most important features of an object-oriented language such as encapsulation, information hiding, class hierarchy, inheritance, and polymorphism.

2 Program Structure

As its simplicity, a BKOOL compiler does not support to compile many files so a BKOOL program is written just in one file only. A BKOOL program consists of many class declarations (subsection 2.1). The entry of a BKOOL program is a static special method, whose name is **main** without any parameter and whose return type is **void**. Each such special method in any class of the BKOOL program is an entry of the program.

2.1 Class declaration

Each class declaration starts with keyword **class** and then an identifier, which is the class name, and ends with a nullable list of members enclosed by a pair of curly parentheses. Between the class name and the list of members, there may be optional keyword **extends** followed by an identifier which is the superclass name.
A *<member>* may be static, which is preceded by keyword **static**, or instance. A member of a class can be an attribute or a method. There are 2 kinds of attribute: mutable and immutable. An immutable attribute is preceded by keyword **final** while a mutable attribube is not. If there are keywords **static** and **final** in an attribute declaration, they can be in any order.
For example:

```
class Shape {
    static final int numOfShape = 0;
    final int immuAttribute = 0;
```

```
    float length,width;
    static int getNumOfShape() {
        return numOfShape;
    }
}
class Rectangle extends Shape {
    float getArea(){
        return this.length*this.width;
    }
}
```

2.2 Attribute declaration

After keywords **static** and **final** if any, an attribute declaration starts with a type, followed by a non-nullable comma-separated list of attribute names and ended with a semicolon. An attribute name in the list is an identifier optionally followed by an equal and an expression. For example:
final int My1stCons = 1 + 5,My2ndCons = 2;
static int x,y = 0;

2.3 Method declaration

After keyword **static** if any, each method declaration has the form:
<return type> <identifier> (<list of parameters>)<block statement>
The *<return type>* is a type which is described in Section 4. The *<identifier>* is the name of the method. The *<list of parameters>* is a nullable semicolon-separated list of parameter declaration. Each parameter declaration has the form:
<type> <identifier-list>
The *<block statement>* will be described in Section 6.1.
In a class, the name of a method is unique that means there are not two methods with the same name allowed in a class. A special instance method is constructor method whose name is the same as class name and it has no return type so no return statement is in the body. Its declaration has the form:
<identifier>(<list of parameters>) <block statement>

3 Lexical Structure

3.1 Character Set

The character set of BKOOL is ASCII. 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 BKOOL.
This definition of lines can be used to determine the line numbers produced by a BKOOL compiler.

3.2 Comment

There are two types of comment in BKOOL: block and line. A block comment starts with "/*" and ignores all characters (except EOF) until it reaches "*/". A line comment ignores all characters from '#' to the end of the current line,i.e, when reaching end of line or end of file.
For example:
/ This is a block comment, that*
may span in many lines/*
a := 5;#this is a line comment
The following rules are enforced in BKOOL:

- Comments are not nested
- "/*" and "*/" have no special meaning in any line comment
- '#' has no special meaning in any block comment

For example:
/ This is a block comment so # has no meaning here */*
#This is a line comment so / has no meaning here*

3.3 Identifier

Identifiers are used to name variables, constants, classes, methods and parameters.
Identifiers begin with a letter (A-Z or a-z) or underscore ('_'), and may contain letters, underscores, and digits (0-9). BKOOL is case-sensitive, therefore the following identifiers are distinct: WriteLn, writeln, and WRITELN.

3.4 Keyword

The following character sequences are reserved as keywords and cannot be used as identifiers:

boolean	**break**	**class**	**continue**	**do**	**else**
extends	**float**	**if**	**int**	**new**	**string**
then	**for**	**return**	**true**	**false**	**void**
nil	**this**	**final**	**static**	**to**	**downto**

3.5 Operator

The following is a list of **valid** operators along with their meaning:

Operator	Meaning	Operator	Meaning
+	Addition or unary plus	-	Subtraction or minus
*	Multiplication	/	Float division
\	Integer Division	%	Modulus
!=	Not equal	==	Equal
<	Less than	>	Greater than
<=	Less than or equal	>=	Greater than or equal
||	Logical OR	&&	Logical AND
!	Logical NOT	^	Concatenation
new	Object creation		

3.6 Separator

The following characters are the **separators**: left square bracket ('['), right square bracket (']'), left parenthesis ('{'), right parenthesis ('}'), left bracket ('('), right bracket (')'), semi-colone (';'), colon (':'), dot ('.') and comma (',').

3.7 Literal

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

3.7.1 Integer literal

Integer literals are values that **are always expressed in decimal** (base 10). A decimal number is a string of digits (0-9) and is at least one digit long.
The following are valid integer numbers:
0 100 255 2500
Integer literals are of type **integer**.

3.7.2 Float Literal

A floating-point 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 optionally followed by some digits. The exponent part starts with 'E' or 'e', followed optionally by '+' or '-', and then a non-empty sequence of digits. The decimal part or the exponent part can be omitted, but not both.
For example: The following are valid floating literals:
9.0 12e8 1. 0.33E-3 128e+42

The following are **not** considered as floating literals:
.12 (no integer part) 143e (no digits after 'e')
Float literals are of type **float**.

3.7.3 Boolean Literal

A **boolean literal** is either *true* or *false*. Boolean literals are of type **boolean**.

3.7.4 String Literals

String literals consist zero or more characters enclosed by double quotes ("). Use escape sequences (listed below) to represent special characters within a string.
It is a compile-time error for a new line or EOF character to appear inside a string literal.
All the supported escape sequences are as follows:

\b	backspace
\f	formfeed
\r	carriage return
\n	newline
\t	horizontal tab
\"	double quote
\\	backslash

The following are valid examples of string literals:
"This is a string containing tab \t"
"He asked me: \"Where is John?\""
A string literal has a type of **string**.

3.7.5 Array Literals

An **array literal** is non-nullable comma-separated list of literals enclosed by a pair of curly parentheses. The literals in the list can be in any type except the array type and must be in the same type. An array literal has a type of array whose element type is the type of literals in the list.

The following are valid examples of array literals:
{1, 2, 3}
{2.3, 4.2, 102e3}

4 Type

Types limit the values that a variable can hold (e.g., an identifier x whose type is int cannot hold value true...), the values that an expression can produce, and the operations supported on those values (e.g., we can not apply operation + in two boolean values...).

4.1 Primitive Type

4.1.1 Integer

The keyword *int* is used to represent an integer type. A value of type integer may be positive or negative. Only these operators can act on integer values:
+ - * / < <= > >= == != \%

4.1.2 Float

The keyword *float* represents a float type. The operands of the following operators can be in float type:
+ - * / < <= > >=

4.1.3 Boolean

The keyword *boolean* denotes a boolean type. Each value of type boolean can be either *true* or *false*.
if statements work with boolean expressions.
The operands of the following operators can be in boolean type:
== != ! && ||

4.1.4 String

The keyword *string* expresses the string type. Only operator ^is used to operate on string operands.

4.1.5 Void

The keyword *void* is used to express the void type. This type is only used to a return type of a method when it has nothing to return. This type is **not** allowed to use for a variable, constant or parameter declaration.

4.2 Array Type

For simplicity reason, BKOOL supports only one-dimensional arrays. An array type declaration is in the form of:

<element type>'['<size>']'
Note that

- The element type of an array cannot be array type or, of course, void type.
- In an array declaration, it is required that there must be an integer literal between the two square bracket. This number denotes the number (or the length) of that array. The lower bound is always 0.
  For example,
  int[5] a;
  In this example, *a* is an array and has five elements: a[0],a[1],a[2],a[3],a[4].

4.3 Class Type

A class declaration defines a class type which is used as a new type in the program. *nil* is the value of an uninitialized variable in class type. An object of type *X* is created by expression *new X()*.

5 Expression

Expressions are constructs which are made up of operators and operands. They calculate on their operands and return new data. In BKOOL, there exist two types of operations, unary and binary. Unary operations work with one operand and binary operations work with two operands. There are some groups of operators: arithmetic, boolean, relational, string, index, method invocation and object creation.

5.1 Arithmetic Expression

Arithmetic expressions use the following operators:

Operator	Operation
+	Prefix unary sign identity
-	Prefix unary sign negation
+	Infix binary addition
-	Infix binary subtraction
*	Infix binary multiplication
/	Infix binary float division
\	Infix binary integer division
%	Infix binary remainder

The operands of these operators could be of **integer** or **float** type. However, the two \and % operators require all their operands must be in integer type or a type mismatch error will occur. If the operands are all in integer or float type then the operation results are in the same type of the operands. Otherwise, the result will be of float type. There is one exception in the case of operator /: the result is always in float no matter types of its operands.

5.2 Boolean Expression

Boolean expressions have logical operators, such as && (AND), || (OR), !(NOT). The operands of these operators must be in boolean type and their result type is also boolean.

5.3 Relational Expression

Relational operators perform comparisons on their operands. The operands of == and != must be in the same type while the operands of the other relational operator can be in different types. All relational operations result in a boolean type. Relational operators include:

Operator	Meaning	Applicable operand types
==	Equal	integer, boolean
!=	Not equal	integer,boolean
>	Greater than	integer, float
<	Less than	integer, float
>=	Greater than or equal	integer, float
<=	Less than or equal	integer, float

5.4 String Expression

The ^ is used to concat two strings and return a new string that is formed by appending the second string to the first string.

5.5 Index Expression

An **index operator** is used to reference or extract selected elements of an array. It must take the following form:
<expression> '[' expression ']'
The type of the first *<expression>* must be an array type. The second expression, i.e. the one between '[' and ']', must be of integer type. The index operator returns the corresponding element of the array.
For example,
a[3+x.foo(2)] := a[b[2]] +3;

The above assignment is valid if the return type of foo is integer and the element type of b is integer.
x.b[2] := x.m()[3];
The above assignment is valid if the type of attribute b and the return type of method m are the same type that is an array type.

5.6 Member Access

An **instance attribute access** may be in the form:
<expression>.<identifier>
where *<expression>* is an expression that returns an object of a class and *<identifier>* is an attribute of the class.

A **static attribute access** may be in the form:
<identifier>.<identifier>
where the first *<identifier>* is a class name, and the second *<identifier>* is a static attribute of the class.

An **instance method invocation** may be in the form:
<expression>.<identifier>(<list of expressions>)
where *<expression>* is an expression that returns an object of a class and *<identifier>* is a method name. The type of the first *<expression>* must be a class type. The *<list of expressions>* is the comma-separated list of arguments, which are expressions. The type of the invocation is the return type of the invoked method.

A **static method invocation** may be in the form:
<identifier>.<identifier>(<list of expressions>)
where the first *<identifier>* is a class name and *<identifier>* is a static method name of the class. The others are the same as those in instance invocation.

5.7 Object Creation

An object of a class type is only created by expression:
new <identifier>(<list of expressions>)
The *<identifier>* must be in a class type. The *<list of expressions>* is the comma-separated list of arguments. The list may be empty when the constructor of the class has no parameter.

5.8 This

The keyword **this** expresses the current object of the enclosing class. The type of **this** is the class type of the enclosing class.

5.9 Operator Precedence and Associativity

The order of precedence for operators is listed from highest to lowest:

Operator	Associativity
new	right
.	left
[,]	
+, - (unary)	right
!	right
^	left
*, /, \, %	left
+, - (binary)	left
&&, ||	left
==, !=	none
<, >, <=, >=	none

5.10 Type coercion

In BKOOL, mixed-mode expressions are permitted. Mixed-mode expressions are those whose operands have different types.
The operands of the following operators:
+ - * / < <= > >=
can have either type integer or float. If one operand is float, the compiler will implicitly convert the other to float. Therefore, if at least one of the operands of the above binary operators is of type float, then the operation is a floating-point operation. If both operands are of type integer, then the operation is an integral operation.
Assignment coercions occur when the type of a variable (the left side) differs from that of the expression assigned to it (the right side). The type of the right side will be converted to the type of the left side.
The following coercions are permitted:

- If the type of the variable is integer, the expression must be of the type integer.
- If the type of the variable is float, the expression must have either the type integer or float.
- If the type of the variable is boolean, the expression must be of the type boolean. Since an argument of a method call is an expression, type coercions also take place when arguments are passed to methods.

Note that, as other object-oriented languages, an expression in a subtype can be assigned to a variable in a superclass type without type coercion.

5.11 Evaluation order

BKOOL 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 method invocation, the actual parameters must be evaluated from left to right.
Every operand 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).

6 Statement

A statement, which does not return anything, indicates the action a program performs. There are many kinds of statements, as describe as follows

6.1 Block statement

A block statement begins by the left parenthesis '{' and ends up with the right parenthesis '}'. Between the two parentheses, there may be a nullable list of statements preceded by a nullable list of mutable/immutable variable declarations which are written like mutable/immutable attribute without keyword **static**.
For example:

```
{
    #start of declaration part
    float r,s;
    int[5] a,b;
    #list of statements
    r:=2.0;
    s:=r*r*this.myPI;
    a[0]:= s;
}
```

6.2 Assignment statement

An **assignment statement** assigns a value to a local variable, a mutable attribute or an element of an array. An assignment takes the following form:
<lhs> := <expression>;

where the value returned by the *<expression>* is stored in the *<lhs>*, which can be a local variable, a mutable attribute 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:

```
this.aPI := 3.14;
value := x.foo(5);
l[3] := value * 2;
```

6.3 If statement

The **if statement** conditionally executes one of two statements based on the value of an expression. The form of an if statement is:
if <expression> then <statement> [else <statement>]
where *<expression>* evaluates to a boolean value. If the expression results in *true* then the *<statement>* following the reserved word *then* is executed. If *<expression>* evaluates to *false* and an else clause is specified then the *<statement>* following *else* is executed. If no else clause exists and expression is false then the if statement is passed over. The following is an example of an if statement.

```
if flag then
    io.writeStrLn("Expression is true");
else
    io.writeStrLn ("Expression is false");
```

6.4 For statement

The **for** statement allows repetitive execution of one or more statements. For statement executes a loop for a predetermined number of iterations. For statements take the following form:
for <scalar variable> := <expression1> (to|downto) <expression2> do <statement>
First, *<expression1>* will be evaluated and assigned to <scalar variable>. Then BKOOL calculates *<expression2>* . In case of *to* clause being used, if the value of *<expression1>* (i.e., the current value of *<scalar variable>*) is less or equal to the value of *<expression2>*, *<statement>* will be executed. After that, *<scalar variable>* will be incremented by 1. The process continues until the <scalar variable> hits the value of *<expression2>*. If *<scalar variable>* is greater than *<expression2>*, the *<statement>* will be skipped (i.e., the statement next to this for loop will be executed).
If *downto* clause is used, the iterative process is the same except that the *<statement>* will be executed if *<scalar variable>* are greater than or equal to *<expression2>* and the

<scalar variable> will be decremented by 1 after each iteration.
Note that *<scalar variable>*, *<expression1>*, *<expression2>* must be of integer type.
For example,

```
for i := 1 to 100 do {
    io.writeIntLn(i);
    Intarray[i] := i + 1;
}
for x := 5 downto 2 do
    io.writeIntLn(x);
```

6.5 Break statement

Using **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 a for loop). Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A break statement is written as follows.

```
break;
```

6.6 Continue statement

The **continue** statement 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 a for loop). Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A continue statement is written as follows.

```
continue;
```

6.7 Return statement

A **return** statement aims at transferring control to the caller of the method that contains it. It must be of the form:

```
return <expression> ;
```

6.8 Method Invocation statement

A **method invocation** statement is an instance/static method invocation, that was described in subsection 5.6, with a semicolon at the end.
For example,

```
Shape.getNumOfShape();
```

7 Scope

There are 4 levels of scope: global, class, method and block.

7.1 Global scope

All class names, static attributes and method names have global scope. A class name or a static attribute is visible everywhere and a method can be invoked everywhere, too.

7.2 Class scope

All instance attributes of a class have class scope, i.e., they are visible in the code of all methods of the class and its subclasses.

7.3 Method scope

All parameters/variables declared in the body block have the method scope. They are visible from the places where they are declared to the end of the enclosing method.

7.4 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.

8 IO

To perform input and output operations, BKOOL provides an class, whose name is **io**, containing the following static methods:

Method prototype	Semantic
int readInt();	Read an integer number from keyboard
void writeInt(anArg:int);	Write an integer number to the screen
void writeIntLn(anArg:int);	Write an integer number then a new line to the screen
float readFloat();	Read an float number from keyboard
void writeFloat(anArg:float);	Write a float number to the screen
void writeFloatLn(anArg:float);	Write a float number then a new line to the screen
boolean readBool();	Read a boolean value from keyboard
void writeBool(anArg:boolean);	Write a boolean value to the screen
void writeBoolLn(anArg:boolean);	Write a boolean value then a new line to the screen
string readStr();	Read a string from keyboard
void writeStr(anArg:string);	Write a string to the screen
void writeStrLn(anArg:string);	Write a string then a new line to the screen

9 Example

9.1 Example 1

```
class Example1 {
    int factorial(int n){
        if n == 0 then return 1; else return n * this.factorial(n - 1);
    }

     void main(){
        int x;
        x := io.readInt();
        io.writeIntLn(this.factorial(x));
    }
}
```

9.2 Example 2

```
class Shape {
    float length,width;
    float getArea() {}
    Shape(float length,width){
        this.length := length;
        this.width := width;
    }
```

```
}
class Rectangle extends Shape {
    float getArea(){
        return this.length*this.width;
    }
}
class Triangle extends Shape {
    float getArea(){
        return this.length*this.width / 2;
    }
}
class Example2 {
    void main(){
        s:Shape;
        s := new Rectangle(3,4);
        io.writeFloatLn(s.getArea());
        s := new Triangle(3,4);
        io.writeFloatLn(s.getArea());
    }
}
```

10 Change Log