HCMC University of Technology
Faculty of Computer Science & Engineering

BKOOL
BK Object-Oriented Language

Author
Dr. Nguyen Hua Phung

June 2015
Contents

1 Introduction 3

2 Program Structure 3
  2.1 Class declaration ........................................ 3
  2.2 Constant declaration ................................... 4
  2.3 Variable declaration .................................... 4
  2.4 Method declaration ..................................... 4

3 Lexical Structure 5
  3.1 Character Set ............................................. 5
  3.2 Comment .................................................. 5
  3.3 Identifier ............................................... 6
  3.4 Keyword .................................................. 6
  3.5 Operator .................................................. 6
  3.6 Separator .................................................. 6
  3.7 Literal .................................................... 6
    3.7.1 Integer literal ...................................... 7
    3.7.2 Float Literal ...................................... 7
    3.7.3 Boolean Literal .................................... 7
    3.7.4 String Literals ..................................... 7

4 Type 8
  4.1 Primitive Type .......................................... 8
    4.1.1 Integer ............................................ 8
    4.1.2 Float ............................................... 8
    4.1.3 Boolean ............................................ 8
    4.1.4 String .............................................. 8
    4.1.5 Void ................................................ 8
  4.2 Array Type ............................................... 9
  4.3 Class Type ............................................... 9

5 Expression 9
  5.1 Arithmetic Expression .................................... 9
  5.2 Boolean Expression ..................................... 10
  5.3 Relational Expression .................................. 10
  5.4 String Expression .................................... 10
  5.5 Index Expression ....................................... 11
  5.6 Member Access .......................................... 11
  5.7 Object Creation ......................................... 11
  5.8 Self ...................................................... 12
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 has the form:

\[
\text{class } <\text{identifier}> \begin{cases} \text{extends } <\text{identifier}> \end{cases}\} \{ <\text{list of members}> \}
\]

The first \(<\text{identifier}>\) is the class name. The second \(<\text{identifier}>\), after keyword extends, if any, is the superclass name.

A \(<\text{list of members}>\) is a list of \(<\text{member}>\) which is a \(<\text{attribute}>\), and \(<\text{method declaration}>\) (subsection 2.4). This list may be empty. A \(<\text{member}>\) may be static, which is preceded by keyword static, or instance. There are 2 kinds of attributes: mutable and immutable. A mutable attribute is declared in the form of variable declaration (subsection 2.3) while an immutable attribute is declared in the form of constant declaration (subsection 2.2).

For example:

\[
\text{class Shape} \{
\quad \text{static final integer numOfShape = 0;}
\}
\]
final integer immuAttribute = 0;
length, width: float;
integer static getNumOfShape() {
    return numOfShape;
}
}
class Rectangle extends Shape {
    float getArea(){
        return self.length*self.width;
    }
}

2.2 Constant declaration

Each constant declaration has the form:

[static] final <type> <identifier> = <expression> ;

The <expression> will be discussed later in the Expression section. Note that the expression
in a constant declaration must be evaluated statically, so it cannot include an identifier or
a method invocation.

For example:

final integer My1stCons = 1 + 5;
static final integer My2ndCons = 2;

2.3 Variable declaration

Each variable declaration has the form:

[static] <identifier-list> : <type>;

The <identifier-list> is a non-empty comma-separated list of identifiers. <type> could be
a primitive type, an array type or a class type which will be discussed later in the Type
section.

For example:

my1stVar: integer;
myArrayVar: integer[5];
static my2ndVar, my3rdVar: Shape;
static my2ndArray, my3rdArray: Shape[6];

2.4 Method declaration

Each method declaration has the form:

<return type> [static] <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:
<identifier-list>:<type>
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 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:

```plaintext
bool  break  class  continue  do  else extends  float  if  integer  new  string then  while  return  true  false  void null  self  final  static
```

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition or unary plus</td>
<td>-</td>
<td>Subtraction or minus</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>/</td>
<td>Float division</td>
</tr>
<tr>
<td>\</td>
<td>Integer Division</td>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>~</td>
<td>Concatenation</td>
</tr>
<tr>
<td>new</td>
<td>Object creation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.6 Separator
The following characters are the separators: left square bracket ('['), right square bracket (']'), left parenthesis ('{'), right parenthesis ('}'), left bracket ('('), right bracket (')'), semicolon (';'), 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 1.03e-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 bool.

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.

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 integer 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 bool denotes a boolean type. Each value of type boolean can be either true or false. if, while, repeat and other control 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:

\[ <element\ type>[\textquoteleft<size\textquoteleft>\textquoteleft]\]

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,

\[ a:integer[5]; \]

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. \textit{null} is the value of an uninitialized variable in class type. An object of type \( X \) is created by expression \textit{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:
<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Prefix unary sign identity</td>
</tr>
<tr>
<td>-</td>
<td>Prefix unary sign negation</td>
</tr>
<tr>
<td>+</td>
<td>Infix binary addition</td>
</tr>
<tr>
<td>-</td>
<td>Infix binary subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Infix binary multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Infix binary float division</td>
</tr>
<tr>
<td>\</td>
<td>Infix binary integer division</td>
</tr>
<tr>
<td>%</td>
<td>Infix binary remainder</td>
</tr>
</tbody>
</table>

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 bool type. Relational operators include:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Applicable operand types</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal</td>
<td>Any type (except void)</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
<td>Any type (except void)</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>integer, float</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>integer, float</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td>integer, float</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>integer, float</td>
</tr>
</tbody>
</table>

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 Self

The keyword `self` expresses the current object of the enclosing class. The type of `self` 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new</code></td>
<td>right</td>
</tr>
<tr>
<td><code>.</code></td>
<td>left</td>
</tr>
<tr>
<td><code>[,]</code></td>
<td></td>
</tr>
<tr>
<td><code>+, - (unary)</code></td>
<td>right</td>
</tr>
<tr>
<td><code>!</code></td>
<td>right</td>
</tr>
<tr>
<td><code>^</code></td>
<td>left</td>
</tr>
<tr>
<td><code>*, /, \, %</code></td>
<td>left</td>
</tr>
<tr>
<td><code>+, - (binary)</code></td>
<td>left</td>
</tr>
<tr>
<td>`&amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td><code>==, &lt;&gt;</code></td>
<td>none</td>
</tr>
<tr>
<td><code>&lt;, &gt;, &lt;=, &gt;=</code></td>
<td>none</td>
</tr>
</tbody>
</table>

### 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 bool, the expression must be of the type bool. 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 list of statements preceded by a list of local variable/constant declarations. These two lists may be empty. For example:

```plaintext
{
    #start of declaration part
    r,s:float;
    a,b:integer[5];
    #list of statements
    r:=2.0;
    s:=r*r*Self.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:

\(<\text{lhs}> := <\text{expression}>;\)

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:

```plaintext
self.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:

\(\text{if} <\text{expression}> \text{ then } <\text{statement}> \text{ [else } <\text{statement}> \text{]}\)

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.

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

6.4 While statement

The while statement executes one or more statements in a loop. while statements take the following form:

\(\text{while} <\text{expression}> \text{ do } <\text{statement}>\)

where <expression> evaluates to a boolean value. If the value is true, the while loop executes repeatedly <statement> until the value of <expression> becomes false.
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 one of the followings: a for loop, a repeat loop, or a while 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 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 repeat loop, or a while 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/constants 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/constants 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:

<table>
<thead>
<tr>
<th>Method prototype</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer readInt();</td>
<td>Read an integer number from keyboard</td>
</tr>
<tr>
<td>void writeInt(anArg:integer);</td>
<td>Write an integer number to the screen</td>
</tr>
<tr>
<td>void writeIntLn(anArg:integer);</td>
<td>Write an integer number then a new line to the screen</td>
</tr>
<tr>
<td>float readFloat();</td>
<td>Read a float number from keyboard</td>
</tr>
<tr>
<td>void writeFloat(anArg:float);</td>
<td>Write a float number to the screen</td>
</tr>
<tr>
<td>void writeFloatLn(anArg:float);</td>
<td>Write a float number then a new line to the screen</td>
</tr>
<tr>
<td>bool readBool();</td>
<td>Read a boolean value from keyboard</td>
</tr>
<tr>
<td>void writeBool(anArg:bool);</td>
<td>Write a boolean value to the screen</td>
</tr>
<tr>
<td>void writeBoolLn(anArg:bool);</td>
<td>Write a boolean value then a new line to the screen</td>
</tr>
<tr>
<td>string readStr();</td>
<td>Read a string from keyboard</td>
</tr>
<tr>
<td>void writeStr(anArg:string);</td>
<td>Write a string to the screen</td>
</tr>
<tr>
<td>void writeStrLn(anArg:string);</td>
<td>Write a string then a new line to the screen</td>
</tr>
</tbody>
</table>

9 Example

9.1 Example 1

```java
class Example1 {
    integer factorial(n:integer){
```
if \( n == 0 \) then return 1; else return \( n \times \text{self.factorial}(n - 1) \);

```cpp
void main()
{
    x: integer;
    x := io.readInt();
    io.writeIntLn(self.factorial(x));
}
```

9.2 Example 2

class Shape {
    length, width: float;
    float getArea() {}
    Shape(length, width: float) {
        self.length := length;
        self.width := width;
    }
}
class Rectangle extends Shape {
    float getArea() {
        return self.length * self.width;
    }
}
class Triangle extends Shape {
    float getArea() {
        return self.length * self.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

From version 0.2

• Fix example of Section 2.1
• Remove unused keywords

From version 0.1

• Modify the description of method declaration.
• Remove := out of the table of operators.
• Remove sentence "This operator has the highest precedence" in Section 5.5.
• Revise the definition of concatenation operator in Section 5.4.