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

CSLang SPECIFICATION

Version 1.0.2

HO CHI MINH CITY, 08/2023

CSLANG SPECIFICATION

Version 1.0.2

1 Introduction

CSlang, object-oriented programming language. Its main purpose is to serve as a tool for students practicing the implementation of a basic compiler designed for a simple object-oriented programming language.

Despite its straightforwardness, **CSlang** incorporates the core elements crucial to an object-oriented language, encompassing aspects like encapsulation, information concealment, class hierarchy, inheritance, and polymorphism.

2 Program structure

Due to its simplicity, the **CSlang** compiler lacks the ability to compile multiple files. This means that a **CSlang** program must be contained within a single file. Within this file, numerous class declarations are present (as discussed in subsection 2.1). The starting point of a **CSlang** program is a unique static method named **@main**, located within the **Program** class. This **@main** method takes no parameters and returns nothing. Consequently, only this specific method within the **Program** class serves as the entry point for the program, while the remaining methods are treated as regular ones.

2.1 Class declaration

Each class declaration starts with the keyword **class**, followed by an optional **super** part, an identifier (which is the class name), and ends with a nullable list of members enclosed within a pair of curly parentheses. The **super** part begins with an identifier (which is the super class name of the declared class, followed by **<-**. It's important to note that CSlang supports single inheritance, allowing only one super class.

Every member in a class is named using the identifier rule (subsection 3.3). It can be static if its name is an **at identifier**, otherwise, it is an instance member. A member of a class can be either an attribute or a method. There are two kinds of attributes: mutable and immutable. An immutable/mutable attribute is preceded by the keywords **const/var**.

```
class Shape {
    var @numOfShape: int = 0;
    const immutableAttribute: int = 0;
    var length, width: int;
```

```
    func @getNumOfShape():int {
        return @numOfShape;
    }
}

class Shape <- Retangle {
    func getArea():int {
        return self.length * self.width;
    }
}

class Program {
    func @main():void {
        io.@writeInt(Shape.@numOfShape);
    }
}
```

2.2 Attribute declaration

An attribute starts with the keyword **const**/**var** (to indicate mutability), followed by a non-nullable comma-separated list of attribute names. It is then followed by the mandatory type declaration, indicated by a colon (**:**) and the corresponding type name. Optionally, the next part of this declaration is value initialization, represented by an equals sign (=), followed by a non-nullable comma-separated list of expressions (matching the number of attributes). Finally, the declaration is terminated with a semicolon (**;**).

For example:

```
const My1stCons, My2ndCons: int = 1 + 5, 2;
var @x, @y : int = 0, 0;
```

2.3 Method declaration

Each method declaration has the form:

```
func <identifier> (<list of parameters>) : <return type> <block statement>
```

The `<list of parameters>` is an optional comma-separated list of parameter declarations. Each parameter declaration follows the format: `<identifier>: <type>`. When two or more consecutive named function parameters share a common type, you can omit the type from all except the last parameter in the sequence: `<list of identifiers>: <type>`.

The `<block statement>` will be described in Section 6.9.

Within a class, the name of each method is required to be unique; this means that two methods with the same name are not allowed in a class. However, there is an exceptional case for the **Constructor** method, which is described as follows:

- **Constructor** is a method whose name is `constructor` and it returns nothing in the body.

```
func constructor (<list of parameters>) <block statement>
```

- In the **constructor**, class attributes are preferred to be used on the left-hand side of an expression. This means that when there are parameters with the same name as the class attribute, the class attribute is used as expression on the left-hand side.

3 Lexical structure

3.1 Characters set

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

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

3.2 Program comment

There are two types of comment in CSlang: 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 CSlang:

- 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 Identifiers

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). CSlang is case-sensitive, therefore the following identifiers are distinct: PrintLn, println, and PRINTLN.

At identifiers are special which are only used for static members in a class. It begins with a at sign (@) and continues with a non-empty sequence of letters, underscores and digits.

3.4 Keywords

Keywords must begin with a lowercase letter (a-z). The following keywords are allowed in CSlang:

break	continue	if	else	for
true	false	int	float	bool
string	return	null	class	constructor
var	self	new	void	const
func				

3.5 Operators

The following is a list of valid operators:

+	-	*	/	\
!	&&	||	==	=
!=	<	<=	>	>=
:=	^	new	%	

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

3.6 Separators

The following characters are the separators:

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

3.7 Literals

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

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')
```

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.
```

3.7.5 Array Literals

An **array literal** is a non-nullable comma-separated list of literals enclosed within a pair of square brackets. The literals in the list can be of any type except for the array type and must all be of the same type. An array literal has a type of an array, with its element type being the type of the literals in the list.
The following are valid examples of array literals:

```
[1, 2, 3]
[2.3, 4.2, 102e3]
```

4 Type and Value

In CSlang, 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 types

4.1.1 Boolean type

The keyword `bool` denotes a boolean type. Each value of type boolean can be either `true` or `talse`.

`if` work with boolean expressions.

The operands of the following operators are in boolean type:

```
!       &&      ||      ==      !=
```

4.1.2 Integer type

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 number values:

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

4.1.3 Float type

The keyword `float` is used to represent an float type. A value of type float may be positive or negative. Only these operators can act on number values:

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

4.1.4 String type

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

An array type declaration is in the form of: `'['<size>']'<element_type>`.
Note that:

- `<element_type>` is the element type of an array. It 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 brackets. This number denotes the number (or the length) of that array. The lower bound is always 1.

  For example, `var a: [5]int;` indicates a five-element array: `a[1], a[2], a[3], a[4], a[5]`.

4.3 Class type

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

5 Expressions

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

In CSlang, 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.

5.1 Arithmetic operators

Standard arithmetic operators are listed below.

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

5.2 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

5.3 String operators

Standard string operators are listed below.

Operator	Operation	Operand's Type
^	String concatenation	String

5.4 Relational operators

Relational operators perform arithmetic and literal comparisons. All relational operations result in a boolean type. Relational operators include:

Operator	Operation	Operand's Type
==	Equal	Int/Boolean
!=	Not equal	Int/Boolean
<	Less than	Int/Float
>	Greater than	Int/Float
<=	Less than or equal	Int/Float
>=	Greater than or equal	Int/Float

5.5 Index operators

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

1. 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.

2. A **static attribute access** may be in the form:

   ```
   [<identifier>.]?<at identifier>
   ```

   where the first `<identifier>` is a class name, and the second `<at identifier>` is a static attribute. The class name may be omit if the static attribute is the member of the current class. of the class.

3. 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.

4. A **static method invocation** may be in the form:

```
[<identifier>.]?<at identifier>(<list of expressions>)
```

where the first `<identifier>` is a class name and `<at identifier>` is a static method name of the class. The others are the same as those in instance invocation. The class name may be omit if the static method is the method of the current class.

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 high to low:

Operator Type	Operator	Arity	Position	Association
Object creation	`new`	Unary	Prefix	Right
Static access	.	Binary	Infix	None
Instance access	.	Binary	Infix	Left
Index operator	[]	Unary	Postfix	None
Sign	-	Unary	Prefix	Right
Logical	!	Unary	Prefix	Right
Multiplying	*, /, \, %	Binary	Infix	Left
Adding	+, -	Binary	Infix	Left
Logical	&&, ||	Binary	Infix	Left
Relational	==, !=, <, >, <=, >=	Binary	Infix	None
String	^	Binary	Infix	None

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

5.10 Type coercions

In CSlang, 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 orders

CSlang 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).

6 Statements

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

6.1 Variable and Constant Declaration Statement

Variable and constant declaration statement should be the same as the instance attribute in a class. Variable should start with the keyword `var` while constant should start with the keyword `const`.

6.2 Assignment Statement

An assignment statement assigns a value to a left hand side which can be a scalar variable, an index expression. An assignment takes the following form:

```
lhs := expression;
```

where the value returned by the `expression` is stored in the the left hand side `lhs`, 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:

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

6.3 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 form:

```
if [<pre-statement>]? <expression> <block statement>
[else <block statement>]?
```

where `<pre-statement>` is a block statement,if present, to execute before the condition, `<expression>` evaluates to a boolean value. If the `<expression>` results in true then the corresponding `<block statement>` 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 {i := 0;} j > i {j := j - 1;} else {j :=  j + 1;}
```

6.4 For statement

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

```
for <init statement> ; <condition expression> ; <post statement>
    <block statement>
```

where the `init statement` and `post statement` are written in the assignment form without the tail semicolon. The left expresssion in these assignments is a scalar variable.

```
for i := 0; i < 10; i := i + 1 {
    io.@writeInt(i);
}
```

The basic for loop has three components separated by semicolons:

- The `init statement`: executed before the first iteration
- The `condition expression`: evaluated before every iteration
- The `post statement`: executed at the end of every iteration

The loop will stop iterating once the boolean condition evaluates to false.

6.5 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;
```

6.6 Continue statement

The **continue statement** causes the program to skip the rest of the loop body in the current iteration as if the end of the statement block had been reached. 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;
```

6.7 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.

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();
```

6.9 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.

For example:

```
{
    var r, s: int;
    r := 2.0;
    var a, b: [5]int;
    s := r * r * self.myPI;
    a[0] := s;
}
```

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, CSlang provides an class, whose name is **io**, containing the following static methods:

Method	Semantic
@readInt()	Read an integer number from keyboard and return it.
@writeInt(anArg: Int)	Write an integer number to the screen.
@readFloat()	Read an float number from keyboard and return it.
@writeFloat(anArg: Float)	Write an float number to the screen.
@readBool()	Read an boolean value from keyboard and return it.
@writeBool(anArg: Boolean)	Write an boolean value to the screen.
@readStr()	Read an string from keyboard and return it.
@writeStr(anArg: String)	Write an string to the screen.

9 Change log

Version 1.0.1

- Fix the example in page 1-2

Version 1.0.2

- Fix the format of Constructor: use the keyword constructor for the name of the Constructor instead of IDENTIFIER in Section 2.3
- Add 'only' for **At identifier** in Section 3.3
- Add some keywords (constructor, const, func) to keyword list in Section 3.4
- Fix the example of array type in Section 4.2
- Change 'identifier' to 'at identifier' for static attribute access and static method invocation in Section 5.6
- Fix the operator of String, change the associativity of index operator in the table in Section 5.9
- Change 'attribute' into 'instance attribute' in Section 6.1