VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PRINCIPLES OF PROGRAMMING LANGUAGES - CO3005

ANOTHERC SPECIFICATION

Version 1.0.0

Ho Chi Minh City, September 2023

Contents

ANOTHERC SPECIFICATION

Version 1.0.0

1 Introduction

This is an official specification of AnotherC, a C-like language for students to study about basic control flows, expressions, recursive functions and type checking. It is similar enough to C to feel familiar, but includes type inferring in C++ to give students some sense of alternatives.

2 Program Structure

As its simplicity, AnotherC compiler does not support to compile many files so a AnotherC program must be written in only one file. A AnotherC program consists many structure data type declarations, function declarations and variable declarations. The entry of a AnotherC program is an unique function, whose name is **main** without any parameter and return nothing (type **void**).

3 Lexical Structure

3.1 Characters set

A AnotherC 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 AnotherC. The NULL character (i.e., the ASCII NUL - '\0') is used as a default value of character literals (will be discuss later).
This definition of lines can be used to determine the line numbers produced by a AnotherC compiler.

3.2 Program comment

Both C-style and C++-style comments are valid in AnotherC.

```
/* A C-style comment */
a = 5; // A C++-style comment
```

C-style comment is considered to be non-greedy with opening and closing sign.

3.3 Identifiers

Identifiers are used to name variables, functions, parameters and structure data type. Identifiers begin with a letter (A-Z or a-z) or underscore (_) and may contain letters, digits (0-9) and underscores. AnotherC is **case sensitive**, therefore the following identifiers are distinct: abc, Abc, ABC.

3.4 Keywords

The following string in AnotherC are keywords and may not be used as identifiers:

`auto`	`false`	`int`	`float`	`double`
`char`	`string`	`void`	`bool`	`if`
`else`	`while`	`do`	`struct`	`return`
`break`	`continue`	`for`	`true`	`this`
`new`	`const`			

3.5 Operators

The following is the list of valid operators:

```
+ - * / == <= >= < > != % || && += -= *= /= %= !  ->
```

The meaning of these operators will be discuss in the following sections.

3.6 Separators

The following characters are the separators:

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

3.7 Literals

Literal is a source representation of a value of a integer, float, double, string, char, boolean and array.

3.7.1 Integer

Integer, which is a sequence of digits starting with a non-zero digit or only a zero, can be specified in decimal (base 10). The set of decimal notation is (0-9) and not precede with 0 digit.

3.7.2 Double

A **double** literal consists of two components: integer and decimal. Only integer part can be absent for this representation.

- Integer part have the same format as an integer literal.
- Decimal part starts with a floating point (.) and then an optional sequence of digits in set (0-9).

For example: `2.3, 16.108, .239, 2.` is a valid double literal representation.

3.7.3 Float

A **float** literal consists of three component: integer, decimal and exponent. Two of three components must appear in this representation (one exception is the integer part and exponent part can be absent). If the exponent part is missing, a character 'f' will replace that part to make a difference between float and double.

- Integer part and decimal part have the same format as a double literal.
- Exponent part start with a character `e` or `E` and then an optional sign (+ or -). It finalized with a sequence of digits in set (0-9).

For example: `2.3f, 16.108e3, .239e-4, 2.e10` is a valid float literal representation.

3.7.4 Character

A **character** literal contains 1 character in ASCII table (its value is in range 32 - 126) or escape sequences enclosed by single quotes (’). Single quotes are not part of character literal.
For example: ’c’, ’\0’ is a valid character literal representation.
All the supported escape sequences in AnotherC are as follows:

- \b backspace
- \f form feed
- \r carriage return
- \n newline
- \t tab
- \0 NULL
- \\ backslash
- \’ single quote

3.7.5 String

A **string** literal is a sequence of zero or more characters enclosed by double quotes (”). Use escape sequences (listed above) to represent special character within a string. Like character literal, double quotes are not part of the string. It is a compile-time error for a newline or EOF character to appear after the opening (”) and before the closing (”).
For a double quote inside a string, a backslash (\) must be written before it: \”.
For example:
”This is a string containing tab \t”
”He asked me: \”Where is John?\””

3.7.6 Boolean

A **boolean** literal is either `true` or `false` , formed from ASCII letters.

3.7.7 Array

An **index array** literal is a comma-separated list of expressions (with an array), enclosed in '{' and '}'.
For example: `{1,5,7,12}, {"John", "Katy"}`.

4 Type system and values

In AnotherC, types limit the values that a variable can hold (e.g., an identifier x with type `int` cannot hold value `true` ...), the value that an expression can produce and the operations supported on those value (e.g., we cannot apply operator + in two boolean values...).

4.1 Atomic types

4.1.1 Boolean type

The keyword `bool` denotes a boolean type. Each value of type boolean can be either `true` or `false` .
The operands of these following operators are in boolean type:

`!` `&&` ||

The default value for bool type is **false**.

4.1.2 Integer Type

The keyword `int` denotes an integer type. A value of type `int` maybe positive or negative (begin with minus sign). Only these operators can act on number values:

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

The default value for int type is 0.

4.1.3 Float Type

The keyword `float` denotes a float type. A value of type `float` maybe positive or negative. Only these operators can act on number values:

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

The default value for float type is 0.0f.

4.1.4 Double Type

The keyword `double` denotes a double type. A value of type `double` maybe positive or negative. Only these operators can act on number values:

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

The default value for double type is 0.0.

4.1.5 Char Type

The keyword `char` denotes a char type. Only this operators can act on char values:

`+` (the first operator must be in `string` type if the second operand is in `char` type)

The default value for char type is '\0'.

4.1.6 String Type

The keyword `string` denotes a string type. Only this operators can act on string values:

`+`

The default value for string type is "".

4.2 Array type

AnotherC also support arrays of a row-major order, multi-dimensional and fixed size. An array type declaration is in the form:

```
<element_type>[dimension 1][dimension 2]...[dimensions n]
```

- `<element_type>` is one of 6 atomic types and structure type.
- `dimension i` (for $i = 1$ to n) is a literal integers. The index of the first element in each dimension is always 0.
  For example: `int [2][3]` indicates a two-dimension array whose the size of first dimension is 2 and of the second dimension is 3. All the elements in this array can be accessed by: `a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]` .

The default-value for array type is construct by assign default-value of type `<element_type>` to every elements in the last dimensions and all the dimensions must be fulfilled.

4.3 Void type

The `void` type, is the return type of a function that returns normally, but does not provide a result value to its caller. Usually such functions are called for their side effects, such as performing some task or writing to their output parameters.

4.4 Auto type

The `auto` type is the type of the variable that is being declared will be automatically deducted from its initial expression. In the case of functions, if their return type is `auto` then that will be evaluated by return type expression.
In the case of parameters, if parameter has type `auto` and also have default value, its type will be inferred from the default value's type. Otherwise, when we call a function for the first time (through function call or a call statement), the type of the argument passing to that parameter will be the final type of that parameter.

4.5 Structure type

A **structure** type (or **struct** type) is a composite data type (or record) declaration that defines a physically grouped list of variables under one name in a block of memory, allowing the different variables to be accessed via a single pointer or by the struct declared name which returns the same address.

Structure type starts with a keyword `struct` follow by a name (we called *struct name*).
The default value for structure type is to assign every attributes in the structure with default-value using default constructor (discuss in subsection 5.3).

5 Declarations

5.1 Variable declarations

AnotherC requires every variable to be declared with its type before its first 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 scope. When a variable is re-declared by another variable in a nested scope, it is hidden in a nested scope.

5.1.1 Variables

Each variable start with an optional keyword `const` , only one type, a comma-separated list of identifiers, each identifier may have initial expression, begins by an equal sign (=), then an expression. It finalized with a semicolon(;). So the full form of variable declaration is as follows:

`[const]? <type> list(<identifier> [= <expression>]?);` If one non-auto and non-const variable does not have initial expression, AnotherC will assign default-value based on their type to them.

5.1.2 Parameters

Function declarations require a list of parameter declarations. Each parameter declaration uses the following form:

```
<type> <identifier> [= <expression>]?
```

5.2 Function declarations

In AnotherC, a function must be declared before its first used.
There are two kinds of function declarations: forward declaration and function

definition. Only forward declaration can be absent.
AnotherC compiler will use the function definition as the final declaration of that function.

5.2.1 Forward declaration

A forward declaration tells the compiler about a function's name, return type and parameters. The form is as follows:

```
<type> <identifier> (<parameter-list>);
```

Where `<parameter-list>` is a null-able comma-separated list of parameter declarations.

5.2.2 Function definition

A function definition provides the actual body of a function. It must have the same prototype (name, return type, number of parameters, type of each parameter) with forward declaration (if exists). It uses the following form:

```
<type> <identifier> (<parameter-list>) <block-statement>
```

- `<parameter-list>` is mentioned above.
- `<block-statement>` will be discussed in subsection 7.10.

For parameter list, the following constraints must be satisfied:

1. If there is only forward declaration or function definition, parameter with default value must be at the end of the list.
2. If both forward declaration and function definition appear, parameter with default value must appear in only one place (forward declaration or function definition), and all default-value parameters must be at the end of function definition's parameter list.

5.3 Structure Declaration

Structure declaration has its form:

```
struct <identifier> <block-method>;
```

A `<block-method>` is a list of two kinds of special declarations: attribute declarations and constructor declarations, enclosed by left curly bracket('{') and right curly bracket ('}').

5.3.1 Attribute declarations

Attribute declarations have a same form as variable declarations, except:

- Only support atomic type.
- Initial expression is not allowed.

An attribute can be used before or after its declaration.

5.3.2 Constructor declarations

Constructor declarations have a same form as function definition, except:

- No return type (not `void` type), which means no return statement in the body of constructor.
- The name of constructor is the same as the name of structure and we can re-declare the constructor, but it must have different number of parameters.
- Two or more constructor declarations cannot be ambiguous when calling. For example: `f(int a, int b = 0)` and `f(int a)` are ambiguous since `f(0)` satisfied both constructors.

If no constructors provided, AnotherC will provide a default constructor, which has no parameters and assign all the attributes with the default value depends on its type.
In AnotherC, all attributes in structure data type is accessible everywhere after the structure declaration by using the member access operator '.'.

6 Expression

Expression (should be `<expression>` in forms) are constructs which made up of operators and operands, Expressions work with existing data and return new

data.

In AnotherC, there are three types of operations: unary, binary and ternary. Unary works with one operand, binary works with two operands while ternary works with three operands. The operands may be constants, variables, data returned by other operator or data returned by a function call. The operators can be grouped according to the types they operate on. They are six groups of operators: arithmetic, boolean, relational, string, index operators, increment/decrement operators.

6.1 Arithmetic operators

Standard arithmetic operators are listed below:

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

6.2 Boolean operators

Boolean operators include logical **NOT**, logical **AND**, logical **OR**. Only logical **AND** and logical **OR** have short-circuit evaluation (discuss later).

The operation of each is summarized below:

Operator	Operation	Operand's type
!	Negation	bool
&&	Conjunction	bool
||	Disjunction	bool

6.3 String operators

Standard string operators are listed below:

Operator	Operation	Operand's type
+	String concatenation	string-string or string-char

6.4 Relational operators

All relational operations result in a bool type. Relational operators include:

Operator	Operation	Operand's type
==	Equal	int/float/double
!=	Not Equal	int/float/double
<	Less than	int/float/double
<=	Less than or equal	int/float/double
>	Greater than	int/float/double
>=	Greater than or equal	int/float/double

6.5 Increment/Decrement operators

These operators will increase/decrease the variable by one. In C/C++, there are two ways to represent this operator (also have different meaning): prefix and postfix. In AnotherC, prefix and postfix representation will keep the same, and also have same meaning.

These operators include:

Operator	Operation	Operand's type
++	Increment	int/float/double
−−	Decrement	int/float/double

6.6 Index operators

An **index operator** is used to reference or extract selected elements of an array. It must take the following form:

`<identifier>[<expression 1>][<expression 2>]...[<expression n>]`

Where `<identifier>` is the name of array and `<expression i>` is the i-th expression represents the i-th indices

6.7 Member access

In order to access to structure's attributes, AnotherC has two ways to access these attributes:

```
this-><attribute>
<identifier>/<index-operator>.<attribute>
```

The first representation can only use inside structure declaration (use in constructor's body) while the second one can only use outside the structure declaration.

6.8 New expression

To create a structure, AnotherC has a `new` keyword and it uses this form:

```
new <identifier>(<expression-list>)
```

- `<identifier>` is the name of structure data type
- `<expression-list>` is a null-able comma-separated list of expression.

6.9 Function call

The function call starts with an identifier (which is also a function's name), then an opening parenthesis, then a null-able comma-separated list of expressions, and a closing parenthesis. The value of a function call is the returned value of the callee function.

6.10 Ternary expression

The ternary expression takes the following form:

```
<condition-expr>? <true-expr>: <false-expr>
```

First, the `<condition-expr>` will be evaluated. If the `<condition-expr>` returns true, the `<true-expr>` is evaluated, otherwise the `<false-expr>` is evaluated. Both `<true-expr>` and `<false-expr>` must be of the same type.
In C/C++, ternary operator is right associative but in AnotherC, it is non-associative.

6.11 Operator precedence and associativity

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

Operator Type	Operator	Arity	Position	Association
Member access	. ->	Binary	Infix	Left
Index operator	[]	Unary	Infix	Left
Increment/Decrement	++ −−	Unary	Prefix/Postfix	Right/Left
Unary sign	+ -	Unary	Prefix	Right
Logical	!	Unary	Prefix	Right
Multiplying	* / %	Binary	Infix	Left
Adding	+ -	Binary	Infix	Left
Relational	< > <= >=	Binary	Infix	Left
Equality	== !=	Binary	Infix	Left
Logical AND	&&	Binary	Infix	Left
Logical OR	||	Binary	Infix	Left
Conditional	? :	Ternary	Infix	None
Concatenate	+	Binary	Infix	Left
Create structure	new	Unary	Prefix	Right

Operators in left bracket ('(') and right bracket (')') have highest precedence.

7 Statements

A statement (should be `<statement>` in form) indicates the action a program performs. There are many kinds of statements, as described as follows:

7.1 Assignment statement

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

```
<lhs> <assignment-operator> <expression>;
```

Where `<assignment-operator>` is one of the following operator:

- =: assignment operator
- +=: addition assignment operator
- -=: subtraction assignment operator
- *=: multiplication assignment operator
- /=: division assignment operator
- %=: remainder assignment operator

The first operator assign value returned by `<expression>` to the left hand side `<lhs>` .

The assignment statement with the last 5 operators can be rewritten as:

```
<lhs> = <lhs> <op> <expression>;
```

Where `<op>` is one of 5 operators + - * / %.

7.2 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 <expression>:  <true-statement>
[else if <expression 1>:  <statement 1>]?
[else if <expression 2>:  <statement 2>]?
...
[else if <expression n>:  <statement n>]?
[else:  <false-statement>]?
```

Where all the expressions evaluate to a boolean value.

If the first `<expression>` result in true value, the corresponding `<true-statement>` is executed. Otherwise, the first else if statement is executed, then the second else if statement (if the expression of the first one is false),...up to the n-th else if statement. If all expressions of all else if statements is false, the `<false-statement>`

is executed.

If all else if statements and else statement are not given and the expression of the if statement is false, the if statement is passed over.

7.3 For statement

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

```
for (<init-var-declaration>, <condition-expr>, <update-expr>)
  <statement>
for (<assignment-statement>, <condition-expr>, <update-expr>)
  <statement>
```

The `<init-var-declaration>` is similar to a variable declaration, except:

- Only support `int` type.
- Mutable.
- Only have one identifier and must have initial expression.

The `<assignment-statement>` is the same as an assignment statement mentioned in subsection 7.1, except:

- `<lhs>` is only scalar-variable.
- `<lhs>` and `<expression>` must have `int` type.

First, the `<init-var-declaration>` (or `<assignment-statement>`) is executed. Then the `<condition-expr>` will be evaluated. If the `<condition-expr>` is true, the `<statement>` , and then, the `<update-expr>` will be calculated and added to the current value of initial variable or scalar variable. The process is repeatedly executed until the `<condition-expr>` returns false, and the for statement will be terminated (i.e., the statement next to this for loop will be executed). Note that the `<condition-expr>` must be of boolean type.

In AnotherC, the `<condition-expr>` and the `<update-expr>` is not protect.

7.4 While statement

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

```
while <expression>:
  <statement>
```

Where the `<expression>` evaluates to a boolean value. If the value is true, the while loop repeatedly the `<statement>` until the expression becomes false.

7.5 Do-while statement

The **do-while statement**, much like the **while statement**, executes the `<block-statement>` in the loop (`<block-statement>` must be in the form of block statement in subsection 7.10). 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
  <block-statement>
while (<expression>);
```

Where the do-while loop executes repeatedly until the `<expression>` evaluates to the boolean value of false.

7.6 Break statement

Using the **break statement**, we can leave the loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. It must reside in a loop. Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A break statement has the following form:

```
  break;
```

7.7 Continue statement

The **continue statement** causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. It must reside in a loop. Otherwise, an error will be generated (This will be discussed in Semantic Analysis phase). A continue statement has the following form:

```
continue;
```

7.8 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 the keyword `return` which is optionally followed by an expression and ends with a semi colon.
A return statement must appear within a function.

7.9 Call statement

A **call statement** is like a function call except that it doesn't join to any expression and is always terminated by a semi colon.
For example:

```
foo(2 + x, 4.0/y);
goo();
```

7.10 Block statement

A **block statement** begins by the left curly bracket ('{') and ends up with a right curly bracket ('}'). Between the two brackets, there are a null-able list of statements or variable declarations.

8 Special functions

To perform input and output operations, AnotherC provides some special functions as follows:

Function	Semantic
readInteger()	Read an integer from keyboard and return it.
printInteger(anArg: int)	Write an integer number to the screen.
readFloat()	Read a float from keyboard and return it.
printFloat(anArg: float)	Write a float number to the screen.
readDouble()	Read a double from keyboard and return it.
printDouble(anArg: double)	Write a double number to the screen.
readString()	Read a string from keyboard and return it.
printString(anArg: string)	Write a string to the screen.
readChar()	Read a char from keyboard and return it.
printChar(anArg: char)	Write a char to the screen.
readBoolean()	Read a boolean value from keyboard and return it.
printBoolean(anArg: bool)	Write a boolean value to the screen.

All the print functions will print the newline character ('\n') at the end of the output.

9 Short-circuit evaluation

In AnotherC, there are two operators which have **short-circuit evaluation**: && and ||. The associativity of these operators are left-to-right, but if one expression returns the value satisfied the truth and falsehood (**true** for ||, **false** for &&), the evaluation will stop.

10 Semantic

10.1 Type coercion between `double` , `float` and `int`

In AnotherC, using float and int for double alternatives, using int for float alternatives is allowed but the opposite side is not.
Every arithmetic operators (subsection 6.1) which works with three types: double, float and int has to follow the rules:

- If at least one operands has the type `double` , the result type will be in `double.`
- If at least one operand has type `float` and the type of the other operand is not `double` , the result type will be in `float.`
- Otherwise, the result type will be in `int.`

10.2 Declarations with `auto` type

A **variable** of type `auto` indicates an automatic type which is to be inferred by the value given on the right hand side. So an initial expression must be given in the right hand side if a variable declares with type **auto**. In the case of infer type for function and parameter, see subsection 4.4

10.3 Declarations with `const` keyword

A **variable** with `const` keyword must have initial expression in the right hand side of variable declarations.
If a constant declarations with `auto` type has no initial expression, the exception for not initialize an `auto` type variable is raised.

10.4 Calling a function with default-value argument

Since we can assign a default value to a parameter in AnotherC, we can call a function with the number of arguments is less than the number of parameters (the default value will be use to that corresponding missing argument). If a parameter corresponding to the missing argument has no default value, an error will be generated.

11 Change log