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

MT22 SPECIFICATION

Version 1.0.0

HO CHI MINH CITY, 01/2023

MT22 SPECIFICATION

Version 1.1

1 Introduction

This is an official specification of MT22, a C-like language for use in an undergraduate class. MT22 includes expressions, basic control flow, recursive functions, and strict type checking. It is object-code compatible with ordinary C and thus can take advantage of the standard C library, within its defined types. It is similar enough to C to feel familiar, but different enough to give you some sense of alternatives.

2 Program structure

As its simplicity, a MT22 compiler does not support to compile many files so a MT22 program is written just in one file only. A MT22 program consists of many declarations (section 5). The entry of a MT22 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 MT22 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 MT22.

This definition of lines can be used to determine the line numbers produced by a MT22 compiler.

3.2 Program comment

Both C-style and C++-style comments are valid in MT22:

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

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

3.3 Identifiers

Identifiers are used to name variables, functions and parameters. Identifiers begin with a letter (A-Z or a-z) or underscore (_), and may contain letters, underscores, and digits (0-9). MT22 is case-sensitive, therefore the following identifiers are distinct: WriteLn, writeln, and WRITELN.

3.4 Keywords

The following strings are MT22 keywords and may not be used as identifiers:

```
auto        break       boolean     do          else
false       float       for         function    if
integer     return      string      true        while
void        out         continue    of          inherit
array
```

3.5 Operators

The following is a list of valid operators:

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

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

3.6 Seperators

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 four types of array.

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 do not precede with 0 digit.

integer literals may contain underscores (_) between digits, for better readability of literals. These underscores are removed by MT22's scanner.

For example:

```
1234    123
1_72 (considered as 172 by scanner)
1_234_567 (considered as 1234567 by scanner)
```

2. **Float**: A floating number consists of three components: integer, decimal and exponent part, respectively. Exponent part can be omitted if there exists both of remaining components. Otherwise, only one of them 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).
   - Exponent part begins with a charater `e` or `E` and then an optional sign (`-`, `+`). It finalized with a sequence of digits in set (0-9).

   For instance:

   ```
   1.234   1.2e3   7E-10
   1_234.567  (considered as 1234.567)
   ```

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

4. **String**: A **string** literal includes zero or more characters enclosed by double quotes ("). Use escape sequences (listed below) to represent special characters within a string. Remember that the quotes are not part of the string. It is a compile-time error for a new line or EOF character to appear after the opening (") and before the closing matching (").

   All the supported escape sequences are as follows:

   \b backspace
   \f form feed
   \r carriage return
   \n newline
   \t horizontal tab
   \' single quote
   \\ backslash

   For a double quote (") inside a string, a backslash (\) must be written before it: \".

   For example:

"This is a string containing tab \t"

"He asked me: \"Where is John?\""

5. **Array**: An **indexed array** literal is a comma-separated list of expressions (with an array) enclosed in '{' and '}'.

   For example, `{1, 5, 7, 12}` or `{"Kangxi", "Yongzheng", "Qianlong"}`.

4 Type system and values

In MT22, types limit the values that a variable can hold (e.g., an identifier x whose type is `int` cannot hold value `int`...), 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 Atomic types

4.1.1 Boolean type

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

The operands of the following operators are in boolean type:

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

4.1.2 Integer type

The keyword `integer` is used to represent an integer type. A value of type integer may be positive or negative (begin with a minus sign). 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 operands of the following operators are in string type:

: :

4.2 Array type

MT22 also supports arrays of a row-major order, multi-dimensional and fixed size. An array type declaration is in the form of: `array [<dimensions>] of <element_type>`.

- `<element_type>` is one of four atomic types.
- `<dimensions>` is the comma-separated list of integers and each of them represents the size of the corresponding dimension. The index of the first element in each dimension is always 0.

  For example, `array [2, 3] of integer` indicates a two-dimension array whose the size of the first dimension is 2 and of the second dimension is 3. All the elements of this array can be accessed by: `a[0, 0], a[0, 1], a[0, 2], a[1, 0], a[1, 1], a[1, 2]`

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 initializer. In the case of functions, if their return type is auto then that will be evaluated by return type expression.

5 Declarations

5.1 Variable declarations

MT22 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 scopes. When a variable is re-declared by another variable in a nested scope, it is hidden in the nested scope.

5.1.1 Variables

Each variable declaration starts with the comma-separated list of identifiers, colon (`:`), the only type for all variables in list and finalized with a semicolon (`;`), called **the short form** of variable declaration:

```
<identifier-list> : <type>;
```

When the variables are declared, they have an undetermined value until they are assigned a value for the first time. But it is possible for a variable to have a specific value from the moment it is declared. This is called the **initialization** of the variable . It begins with an equal sign (`=`) and then a comma-separated list of expressions. The part will be inserted in the middle of the type and semicolon in variable declaration. Therefore, the full form of variable declaration is as follows:

```
<identifer-list>: <type> [= <expression-list>]?;
```

The scope of variable is global if declarations are put outside all functions. Otherwise, variables are considered as local in their functions and blocks.

When a declaration is written in the full format, the size of the identifier list must be equal to that of the expression list and this constraint must be checked in the syntax analysis phase.

5.1.2 Parameters

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

```
[inherit]? [out]? <identifier>: <type>
```

in which, `out` is the optional keywords for passing by value-result; otherwise, parameter will be passed by value. The semantic of keyword `inherit` will be discussed later.

5.2 Function declarations

A function declaration consists of two parts: function prototype and function body. Function prototype must starts with an identifier as the function name, a colon (`:`) followed by the keyword `function`, the return type, the comma-separated list of parameter declarations enclosed by round brackets. It optionally finalizes by the inheritance subpart, that starts with the keyword `inherit` and an identifier as the function name (also, parent function). The semantic of inheritance relationship will be discussed later. The form is as follows:

`<identifier>: function <return-type> (<paramter-list>) [inherit <function-name>]?`

Function body is a block statement (in subsection 7.10). The meaning of inherits will be discussed later.

Consider the following example:

```
1  x: integer = 65;
2  fact: function integer (n: integer) {
3      if (n == 0) return 1;
4      else return n * fact(n - 1);
5  }
6  inc: function void (out n: integer, delta: integer) {
7      n = n + delta;
8  }
9  main: function void () {
10     delta: integer = fact(3);
11     inc(x, delta);
12     printInteger(x);
13 }
```

- There are three functions (fact, inc, main) in the above program.
- x is a global variable, delta declared in function main is a local variable that effects in the scope of this function.
- The parameter n in inc will be passed by value-result while other parameters (n in fact, delta in inc) will be passed by value.

6 Expressions

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

In MT22, 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, string, index operators.

6.1 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	int/float
%	Number Remainder	int

6.2 Boolean operators

Boolean operators include logical **NOT**, logical **AND** and logical **OR**. The operation of each is summarized below:

Operator	Operation	Operand's Type
!	Negation	boolean
&&	Conjunction	boolean
||	Disjunction	boolean

6.3 String operators

Standard string operators are listed below.

Operator	Operation	Operand's Type
::	string concatenation	string

6.4 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

6.5 Index operators

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

```
<identifier> [<expression-list>]
```

in which, `<identifier>` is the name of array and `<expression-list>` is comma-separated and denotes the list of indices.

6.6 Function call

The function call starts with an identifier (which is a function's name), then an opening parenthesis, a comma-separated list of arguments (this list could be empty), and a closing parenthesis. The value of a function call is the returned value of the callee function.

6.7 Operator precedence and associativity

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

Operator Type	Operator	Arity	Position	Association
Index operator	[,]	Unary	Postfix	Left
Sign	-	Unary	Prefix	Right
Logical	!	Unary	Prefix	Right
Multiplying	*, /, %	Binary	Infix	Left
Adding	+, -	Binary	Infix	Left
Logical	&&, ||	Binary	Infix	Left
Relational	==, !=, <, >, <=, >=	Binary	Infix	None
String	::	Binary	Infix	None

7 Statements

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

7.1 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`.

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 forms:

```
if (<expression>) <true-statement>
[else <false-statement>]?
```

where the first `<expression>` evaluates to a boolean value. If the `<expression>` results in true then the corresponding `<true-statement>` is executed.

If `<expression>` evaluates to false, the statement following `else`, if any, is executed. If no `else` clause exists and expression is false then 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 (scalar-variable = <init-expr>, <condition-expr>, <update-expr>)
    <statement>
```

First, the `<init-expr>` will be evaluated and assigned to scalar-variable whose type is only `int`. 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 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.

The following are examples of for statement:

```
for (i = 1, i < 10, i + 1) {
    writeInt(i);
}
```

7.4 While statement

The **while statement** executes repeatedly nullable 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 executes 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 a loop (`<block-statement>` must be in the form of block statement in 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 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;
```

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 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 does not join to any expression and is always terminated by a semicolon.

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 the right curly bracket `}`. Between the two brackets, there may be a nullable list of statements or variable declarations.

For example:

```
{
    r, s: int;
    r = 2.0;
    a, b: array [5] of int;
    s = r * r * myPI;
    a[0] = s;
}
```

8 Special functions

To perform special features, i.e, input and output operations, MT22 provides some special functions as follows:

Function	Semantic
readInteger()	Read an integer number from keyboard and return it.
printInteger(anArg: integer)	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.
readBoolean()	Read an boolean value from keyboard and return it.
printBoolean(anArg: boolean)	Write an boolean value to the screen.
readString()	Read an string from keyboard and return it.
printString(anArg: string)	Write an string to the screen.
super(<expr-list>)	Call the parent function with the list of expression as paramaters.
preventDefault()	Prevent default parent function to be called.

9 Change log

9.1 Version 1.0.1 (18/01/2023)

- Add keyword `array` in subsection 3.4
- Change keyword `int` to `integer`
- Change some special function names: `readInt` to `readInteger`, `printInt` to `printInteger`

9.2 Version 1.1 (19/01/2023)

- Add `inherit` feature to parameter declarations.