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

MT232 SPECIFICATION

Version 1.0.0

Ho Chi Minh City, December 2023

Contents

MT232 SPECIFICATION

Version 1.0.0

1 Introduction

This is an official specification of MT232, 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, MT232 compiler does not support to compile many files so a MT232 program must be written in only one file. A MT232 program consists many structure data type declarations, function declarations and variable declarations. The entry of a MT232 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 MT232 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 MT232. 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 MT232 compiler.

3.2 Program comment

Both C-style and C++-style comments are valid in MT232.

```
/* 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. MT232 is **case sensitive**, therefore the following identifiers are distinct: abc, Abc, ABC.

3.4 Keywords

The following string in MT232 are keywords and may not be used as identifiers:

auto	false	int	float	string
void	bool	if	else	while
do	out	return	break	continue
for	true	array	of	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, string, 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 Float

A **float** literal consists of three component: integer, decimal and exponent. At least 2 components must appear in the float 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 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: `2.3, 16.108e3, .239e-4, 2.e10` is a valid float literal representation.

3.7.3 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?\""

All the supported escape sequences in MT232 are as follows:

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

3.7.4 Boolean

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

3.7.5 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 MT232, 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:

```
! && ||
```

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:

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

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:

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

4.1.4 String Type

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

```
+
```

4.2 Array type

MT232 also support arrays of a row-major order, multi-dimensional and fixed size. An array type declaration is in the form:

```
array [dimensions] of <type>
```

Where:

- `<type>` is one of 4 atomic types.

- `dimensions` is a comma - separated list of literal integers. The index of the first element in each dimension is always 0.
  For example: `array [2, 3] of int` indicates a two-dimension array of type `int` 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]` .

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` 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 type of that parameter.

5 Declarations

5.1 Variable declarations

MT232 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> (<identifier> [= <expression>]?)+;
```

5.1.2 Parameters

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

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

If a parameter has a keyword `out`, that parameter is considered to be a pass by value - result parameter.

5.2 Function declarations

In MT232, a function can be declared before or after its first used. It uses the following form:

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

Where:

- `<parameter-list>` is a nullable comma - separated list of parameter declarations.
- `<block-statement>` will be discussed in subsection 7.10.

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 MT232, 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 MT232, 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>[<expr_list>]
```

Where `<identifier>` is the name of array and `<expr_list>` is the comma - separated list of expression, each expression represents the indices of corresponding dimension.

6.7 Function call

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

6.8 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.

6.9 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	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

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 (no `const` keyword).
- 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 MT232, 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, MT232 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 MT232, 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 Change log