`sic` Programming Language Manual

Athreya Chandramouli, Saujas Vaduguru

August 23, 2019

Contents

1 Introduction **2**

2 Lexical Elements **2**
2.1 Identifiers 2
2.2 Keywords 2
2.3 Constants 3
2.4 Operators 4
2.5 Separators 4
2.6 Comments 4

3 Data Types **5**
3.1 Integer 5
3.2 Unsigned Integer 5
3.3 Character 5
3.4 Real 5
3.5 Boolean 6
3.6 Arrays 6
3.7 Files 6

4 Expressions and Operations **6**
4.1 Arithmetic Expression 7
4.2 Relational Expression 7
4.3 Logical Expression 7
4.4 Array Subscripting 7
4.5 Operator Precedence 7

5 Statements **8**
5.1 Declaration statements 8
5.2 Assignment statement 8
5.3 Function call 8
5.4 Break statement 9
5.5 Continue statement 9
5.6 Return statement 9

6 Compound statements **9**
6.1 `if` statements 9
6.2 `while` statements 10
6.3 `for` statements 10

7 Functions **10**

8 Top-level structure **11**

9 Importing libraries **11**

10 File handling **11**

11 I/O **12**

12 Semantic checks **12**

13 Complete syntax **12**
13.1 Microsyntax 13
13.2 Macrosyntax 14

1 Introduction

sic , short for simplified C, is a language designed for the course project of Compilers taken during Monsoon 2019

2 Lexical Elements

The lexical elements that make up a program are: keywords, identifiers, constants, operators, and separators.

2.1 Identifiers

Identifiers are used for naming variables and functions. They can include letters, decimal digits, and the underscore character '_'. Identifiers cannot begin with a digit. **sic** is case-sensitive, so `foo` and `FOO` are two different identifiers.

Grammar

```
digit := "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
alphabetic := "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
     | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
     | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
name-characters := alphabetic | digit | "char95"
space := " " | "char92t" | "char92n"
<name> := "char95"* alphabetic name-characters* space*
```

2.2 Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. These are

```
break char continue else false float for if int return true uint void while
```

Grammar

```
<return> := "return" space*
<break> := "break" space*
<continue> := "continue" space*
<if> := "if" space*
<else> := "else" space*
<for> := "for" space*
<while> := "while" space*
```

2.3 Constants

A constant is a literal numeric or character value, such as `5` or `'m'`. All constants are of a particular data type.

2.3.1 Integer Constant

It can be signed or unsigned. Signed constants are prefixed with a + of - followed by a sequence of digits.

Grammar

```
<uint> := digit+ space*
<int> := sign? <uint> space*
```

2.3.2 Character Constant

It is a single character enclosed withing single quotes. Some characters can be represented using escape sequences prefixed using \.

```
digit := "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
alphabetic := "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
     | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
     | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
special := "!" | """ | "#" | "$" | "%" | "&" | "char39" | "(" | ")" | "*" | "+" | "," | "-" | "." | "/" | ":" |
     ";" | "<" | "=" | ">" | "?" | "@" | "[" | "char92" | "]" | "^" | "char95" | "char96" | "char123" | "|" |
     "char125" | "~"
name-characters := alphabetic | digit | "char95"
space := " " | "char92t" | "char92n"
printable-character := digit | alphabetic | special | space
character := printable-character | non-printable
<char> := "char39" character "char39" space*
```

2.3.3 Real Constant

It consists of a sequence of digits which represents the integer part of the number, a decimal point, and a sequence of digits which represents the fractional part. Either the integer part or the fractional part may be omitted.

Grammar

```
<float> := sign? <uint>.<uint>? space* | sign? .<uint> space*
```

2.4 Operators

An operator performs an operation, such as addition or subtraction, on either one, two, or three operands.

The ternary operator (`<condition> ? a : b`) chooses `a` or `b` based on whether `<condition>` is true or false, respectively.

The augmented assignment operators (`+=, -=, *=, /=, %=`) perform the corresponding arithmetic operation with the first operand as the variable on the left hand side of the operator, and the second operand as the variable or value on the right hand side of the operator, and assign the result to the variable on the left hand side of the operator.

Grammar

```
<unary-operator> := "+" space* | "-" space* | "!" space*
<add> := "+" space* | "-" space*
<mul> := "*" space* | "/" space* | "%" space*
<rel-operator> := ">" space* | "<" space* | ">=" space* | "<=" space*
<eq-operator> := "==" space* | "!=" space*
<and> := "&" space*
<or> := "|" space*
<tern-operator1> := "?" space*
<tern-operator2> := ":" space*
<assignment> := "=" space* | "+=" space* | "-=" space* | "*=" space* | "/=" space* | "%=" space*
```

2.5 Separators

A separator separates lexical elements. White space is a separator. Apart from white space, the seperators are `\t \n ( ) { } ; ,`

Grammar

```
space := " " | "char92t" | "char92n"
<list-separator> := "," space*
<statement-separator> := ";"
<open-parenthesis> := "(" space*
<close-parenthesis> := ")" space*
<open-brace> := "char123" space*
<close-brace> := "char125" space*
<open-brace> := "char123" space*
<close-brace> := "char125" space*
```

2.6 Comments

Comments are used to annotate the code with explanations and other helpful information, to make the code readable. **`sic`** comments are enclosed in double forward slashes (`// //`). Comments are ignored for the purposes of compilation and execution of the program.

Grammar

```
<comment> := "//" printable-character* "//" space*
```

3 Data Types

`sic` supports only primitive data structures. This includes integers, reals and characters. Apart from this it supports arrays of these primitive types. A void type is available only as a return type for functions.

Grammar

```
type-names := "uint" | "int" | "float" | "char" | "file"
<void> := "void"
<type> := type-names space* | array-types space*
<value> := <uint> | <int> | <float> | <char>
```

3.1 Integer

The integer (`int`) data type stores integer values that can be stored using 64 Bits. The range of an integer is from `-9223372036854775807` to `9223372036854775807`.

Integers can be operated on with the unary operators `+` and `-` to signify/change the sign of the integer, and the binary operators `+` (integer addition), `-` (integer subtraction), `*` (integer multiplication), `/` (integer division), and `%` (remainder upon integer division). These operators take two integers, or one integer and one unsigned integer (which is implicitly cast to an integer), and return an integer.

3.2 Unsigned Integer

The unsigned integer (`uint`) data type stores integer values that can be stored using 64 Bits without a sign bit. The range of an unsigned integer is from `0` to `18446744073709551615`.

Unsigned integers can be operated upon by the binary operators `+` (integer addition), `-` (integer subtraction), `*` (integer multiplication), `/` (integer division), and `%` (remainder upon integer division). These operators take two unsigned integers as operands and return an unsigned integer.

3.3 Character

The character (`char`) data type stores characters that correspond to integer values that can be stored using 8 Bits. The range of a character is from `0` to `255`. Each value corresponds to an ASCII value as defined by the ASCII standard.

Characters cannot be operated upon by any operands. To operate on the ASCII value of a character, it must first be assigned to an integer or unsigned integer variable (and implicitly type cast to an integer) and then integer operations may be used.

3.4 Real

The real (`float`) data type stores floating point values that can be stored using 32 Bit IEEE single precision format. The range of a real is from `1.175494e-38` to `3.402823e+38`.

Real numbers can be operated on with the unary operators `+` and `-` to signify/change the sign of the number, and the binary operators `+` (floating-point addition), `-` (floating-point subtraction), `*` (floating-point multiplication), and `/` (floating-point division, where the second operand must be non-zero). These operators take two real numbers, or one real number and one integer (which is implicitly cast to a real number), and return a real number.

3.5 Boolean

Boolean variables may be `true` or `false`.

They can be operated on using the unary logical NOT (`!`) operator. They can be operated on using the binary logical AND (`&`) and logical OR (`|`) operators, with `&` taking precedence.

3.6 Arrays

Arrays of an arbitrary number of dimensions can be defined, to store int, real, or char data. Arrays are initialized when declared by listing the initializing values, separated by commas, in a set of braces. The elements of an array can be accessed by specifying the array name, followed by the element index, enclosed in square brackets. Array elements are zero-indexed.

An array is declared by specifying the type, name, and size. The size of each dimension is specified by adding additional `[<size>]` units. The declaration statement is of the form
`<type> <name>[<size>]([<size>])([<size>])...`

An array literal is specified by listing the array elements in sequence between curly braces (`{}`), separated by commas (`,`). A character array literal can also be specified as a string enclosed in double quotes (`""`).

The only operation allowed on arrays is indexing. Elements of the array can be indexed as
`<array-name>[<index>]`, where the index must be an integer.

Grammar

```
<string-literal> := "char92"" character+ "char92""
<bool-literal> := "true" space* | "false" space*
<literal> := <value> | <open-brace> (<value> list-separator)* <value> <close-brace> | <string-literal>
array-types := type-names <open-square> <uint> <close-square> | array-types type-names <open-square> <uint>
     <close-square>
```

3.7 Files

To perform file operations, files can be accessed using a variable of type `file`. `file` is an internally defined structure that acts as a ‘file pointer’, and allows access to individual bytes in the file.

4 Expressions and Operations

An expression consists of at least one operand and zero or more operators. Operands are typed objects such as constants, variables, and function calls that return values.

Grammar

```
tern-exp := or-exp | <open-parenthesis> or-exp <close-parenthesis> <tern-operator1> tern-exp <tern-operator2>
     or-exp
exp := term | <unary-operator> term
term := <name> | <value> | <open-parenthesis> tern-exp <close-parenthesis> | <name> <open-square> term
     <close-square> | function-call
```

A `<name>` is an identifier, and a `<value>` is a literal.

4.1 Arithmetic Expression

An arithmetic expression is used to evaluate a series of arithmetic operations and return a numeric value representing the result.

Grammar

```
add-exp := mul-exp | add-exp <add> mul-exp
mul-exp := exp | mul-exp <mul> exp
```

4.2 Relational Expression

A relational expression is used to make comparisons between operands. Their output is either 0 or 1.

Grammar

```
rel-exp := add-exp | add-exp <rel-operator> add-exp
eq-exp := rel-exp | rel-exp <eq-operator> rel-exp
```

4.3 Logical Expression

A logical expression is used to evaluate the truth value between a pair of operands. Any non-zero value is considered true and zero is considered false.

Grammar

```
or-exp := and-exp | or-exp <or> and-exp
and-exp := eq-exp | and-exp <and> eq-exp
```

4.4 Array Subscripting

An array subscription is used to access the ith element of an array.

Grammar

```
term := <name> <open-square> term <close-square>
```

4.5 Operator Precedence

The following is a list of types of expressions, presented in order of highest precedence first. In case of equal precedence, operations are applied from left to right unless stated otherwise.

- Function calls, and array subscripting expressions.
- Unary operators, including logical negation,
- Multiplication, division, and modular division expressions.
- Addition and subtraction expressions.

- Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
- Equal-to and not-equal-to expressions.
- Logical AND expressions.
- Logical OR expressions.
- Conditional expressions (using ?:).
- All assignment expressions. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.

5 Statements

A statement is a single logical line of the program.

Grammar

```
statement := declaration | assignment-statement | function-call | <break> | <continue> | <return> | <return>
     tern-exp | if-block | while-block | for-block | <comment> statement
```

5.1 Declaration statements

Declaration statements declare a variable, specifying its type, or a function, with its return type and parameters.

Grammar

```
declaration := <type> <name> <assignment> <literal>
```

5.2 Assignment statement

An assignment statement is used to (possibly) evaluate an expression, and assign the result of the evaluation to a variable.

Grammar

```
assignment-statement := <name> <assignment> tern-exp | <name> <open-square> term <close-square> <assignment>
     tern-exp
```

5.3 Function call

A function call may be used as an expression, or even a standalone statement. A function is called by its name and a list of 0 or more arguments.

Grammar

```
function-call := <name> <open-parenthesis> arguments <close-parenthesis> | <name> <open-parenthesis>
     <close-parenthesis>
```

5.4 Break statement

Used to jump to the first statement following the enclosing iterative statement.

Grammar

```
statement := <break>
```

5.5 Continue statement

Used to return to the beginning of the enclosing iterative statement, before the evaluation of the loop condition.

Grammar

```
statement := <continue>
```

5.6 Return statement

Used to return execution to the statement from where the function was called.

Grammar

```
statement := <return> | <return> tern-exp
```

6 Compound statements

A compound statement is comprised of more than one statement, listed in a statement block, and executed sequentially starting from the first, to the last line.

Grammar

```
statement-block := <open-brace> statement-list <close-brace>
statement-list := statement <statement-separator> | statement-list statement <statement-separator>
```

6.1 `if` statements

`if` blocks are used for conditional execution. The `statement-block` to be executed is selected by evaluating the expression (`tern-exp`). If the expression evaluates to true, the block is executed. If the statement evaluates to false, and an `else` block is present, that block is executed.

Grammar

```
if-block := <if> <open-parenthesis> tern-exp <close-parenthesis> statement-block |
          <if> <open-parenthesis> tern-exp <close-parenthesis> statement-block <else> statement-block
```

6.2 `while` statements

`while` statements are used for repeated or iterative execution. At the beginning of each iteration, the condition expression (`tern-exp`) is evaluated. If it is true, the `statement-block` is executed. If it is false, execution moves to the first statement after the statement block.

Grammar

```
<while> <open-parenthesis> tern-exp <close-parenthesis> statement-block
```

6.3 `for` statements

`for` statements are also used for repeated execution. The execution of the block is controlled by 2 statements and one expression, which may or may not be specified in the `for` statement. The first expression is called the initialisation, the second the condition, and the third the increment. The initialisation is a statement that initialises variables that are accessible in the scope of the loop statement, that is, they are considered declared for statements in the loop's statement block. The condition is an expression which may evaluate to true or false. If absent, it is always assumed to evaluate to true. At the beginning of each iteration, the condition is evaluated. If it is true, the `statement-block` is executed. If it is false, execution moves to the first statement after the statement block. The increment statement, if present, is executed at the end of each iteration.

Grammar

```
<for> <open-parenthesis> <statement-separator> <statement-separator> <close-parenthesis> statement-block |
          <for> <open-parenthesis> <statement-separator> <statement-separator> statement <close-parenthesis>
               statement-block |
          <for> <open-parenthesis> <statement-separator> tern-exp <statement-separator> <close-parenthesis>
               statement-block |
          <for> <open-parenthesis> <statement-separator> tern-exp <statement-separator> statement
               <close-parenthesis> statement-block |
          <for> <open-parenthesis> statement <statement-separator> <statement-separator> <close-parenthesis>
               statement-block |
          <for> <open-parenthesis> statement <statement-separator> <statement-separator> statement
               <close-parenthesis> statement-block |
          <for> <open-parenthesis> statement <statement-separator> tern-exp <statement-separator>
               <close-parenthesis> statement-block |
          <for> <open-parenthesis> statement <statement-separator> tern-exp <statement-separator> statement
               <close-parenthesis> statement-block
tern-exp := or-exp | <open-parenthesis> or-exp <close-parenthesis> <tern-operator1> tern-exp <tern-operator2>
     or-exp
```

7 Functions

A function is defined by a name, a return type, and a list of 0 or more parameters. If a function has a `void` return type, it may not have any return statements. If not, then it must have a return statement returning a value of the specified type.

Arguments of type `int`, `float`, `char`, and `bool` are passed by value, while arrays and `file` type variables are passed by reference.

Grammar

```
function-definition := <type> <name> <open-parenthesis> parameters <close-parenthesis> statement-block |
                      <type> <name> <open-parenthesis> <close-parenthesis> statement-block |
                      <void> <name> <open-parenthesis> parameters <close-parenthesis> statement-block |
                      <void> <name> <open-parenthesis> <close-parenthesis> statement-block
parameters := <type> <name> | parameters <list-separator> <type> <name>
```

A function is called by its name, and a list of 0 or more arguments.

Grammar

```
function-call := <name> <open-parenthesis> arguments <close-parenthesis> | <name> <open-parenthesis>
     <close-parenthesis>
arguments := tern-exp | arguments <list-separator> tern-exp
```

8 Top-level structure

At the highest level, a program is a series of variable declarations and function definitions. A valid program must contain a function called `main`, that returns an integer. All `import` statements must be at the top, i.e., must precede any declarations or definitions.

Grammar

```
start := imports program | program
imports := <import> <string-literal> | imports <import> <string-literal>
program := declaration | function-definition | declaration program | function-definition program | <comment>
     program
```

9 Importing libraries

Functions and variables in other files (libraries) can be imported using the `import` statement, where `import` is followed by a file name as a string literal. The file has to be specified in the current directory, or in the shell resource file (like `.bashrc`). Once imported, all functions and global variables (variables declared outside any function definition) can be used by simply calling the function with its name and arguments.

10 File handling

Functions to open and close files, and perform other file operations are in the `"files.sic"` library.

The `open(<filename>, <mode>)` function takes the file name as a string and mode of operation (read, write, or append) as a character and returns a `file` type variable.

The `close(<filevariable>)` function closes the file.

The `seek(<filevariable>, <nbytes>, <reference>)` moves the file pointer corresponding to `<filevariable>` to a location `<nbytes>` bytes away from the reference, which may be 0, 1, or 2 for beginning of file, correct location, and end of file respectively. `seek` changes the value of the same `<filevariable>`, since it is passed by reference.

11 I/O

The functions `read` and `write` are in the `"io.sic"` library and can be used to read and write from the standard input and output devices.

The `read()` function reads a single character (byte) from the standard input stream. The `write(<char[]>)` function writes a string to screen.

`"io.sic"` also has `fileread(<filevariable>)` which reads a single character from the file, and `filewrite(<filevariable>, <char[]>)` which writes a string to the file.

The library `"stdlib.sic"` provides functions `toint(<char[]>)`, `tofloat(<char[]>)`, and `tobool(<char[]>)` to convert from `char[]` to `int`, `float`, and `bool` respectively. These functions may fail if input is improperly specified (rules for specification are same as rules for literals).

12 Semantic checks

We propose to perform the following semantic checks on an input program:

- The program must contain a function called `main` that returns an integer.
- A variable must be declared before definition or access. A variable is declared with a declaration statement, and defined by assigning a value to it.
- A variable must defined with the correct type. This condition holds when both operands of any assignment operator are of the same type.
- All operands of an operator must be of the same type.
- All literals are within specified bounds for their respective data type:
  - Integers: 64-bit, from `-9223372036854775807` to `9223372036854775807`
  - Unsigned integers: 64-bit, from `0` to `18446744073709551615`
  - Real numbers: From `1.175494e-38` to `3.402823e+38`
  - Characters: ASCII characters only
- A function, if not of `void` return type, must have at least one return statement, and all return statements must return values of the specified return type.
- All elements of an array literal are of the same type.
- An array can be indexed only by a variable of type unsigned integer, or integer, or an integer literal that is non-negative and less than the size of the array as specified during declaration.
- The arguments passed to a function must match the type of the corresponding parameter.
- A variable that is accessed with an index must be an array.

13 Complete syntax

The syntax of the language is divided into two parts – the microsyntax (for tokenisation), and the macrosyntax (for parsing).

13.1 Microsyntax

Here is the complete microsyntax of the language, defined in terms of regular expressions:

```
digit := "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
alphabetic := "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
     | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
     | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
special := "!" | """ | "#" | "$" | "%" | "&" | "char39" | "(" | ")" | "*" | "+" | "," | "-" | "." | "/" | ":" |
     ";" | "<" | "=" | ">" | "?" | "@" | "[" | "char92" | "]" | "^" | "char95" | "char96" | "char123" | "|" |
     "char125" | "~"
name-characters := alphabetic | digit | "char95"
space := " " | "char92t" | "char92n"
printable-character := digit | alphabetic | special | space
character := printable-character | non-printable
<list-separator> := "," space*
<statement-separator> := ";"
sign := "+" | "-"
<unary-operator> := "+" space* | "-" space* | "!" space*
<add> := "+" space* | "-" space*
<mul> := "*" space* | "/" space* | "%" space*
<rel-operator> := ">" space* | "<" space* | ">=" space* | "<=" space*
<eq-operator> := "==" space* | "!=" space*
<and> := "&" space*
<or> := "|" space*
<uint> := digit+ space*
<int> := sign? <uint> space*
<float> := sign? <uint>.<uint>? space* | sign? .<uint> space*
<char> := "char92"" character "char92"" space*
<name> := "char95"* alphabetic name-characters* space*
<assignment> := "=" space* | "+=" space* | "-=" space* | "*=" space* | "/=" space* | "%=" space*
<open-parenthesis> := "(" space*
<close-parenthesis> := ")" space*
<open-brace> := "char123" space*
<close-brace> := "char125" space*
<open-brace> := "char123" space*
<close-brace> := "char125" space*
<open-square> := "[" space*
<close-square> := "]" space*
type-names := "uint" | "int" | "float" | "char" | "file"
<void> := "void"
array-types := type-names <open-square> <uint> <close-square> | array-types type-names <open-square> <uint>
     <close-square>
<type> := type-names space* | array-types space*
<value> := <uint> | <int> | <float> | <char>
<string-literal> := """ character+ """
<bool-literal> := "true" space* | "false" space*
<literal> := <value> | <open-brace> (<value> list-separator)* <value> <close-brace> | <string-literal>
<comment> := "//" printable-character* "//" space*
<return> := "return" space*
<break> := "break" space*
<continue> := "continue" space*
<if> := "if" space*
<else> := "else" space*
<for> := "for" space*
<while> := "while" space*
<import> :+ "import" space*
```

Symbols which are referenced in the macrosyntax are marked with surrounding angular brackets (<>).

13.2 Macrosyntax

Here is the macrosyntax of the language, written as a context-free grammar:

```
start := imports program | program
imports := <import> <string-literal> <statement-separator> | imports <import> <string-literal>
     <statement-separator>
program := declaration | function-definition | declaration program | function-definition program | <comment>
     program
declaration := <type> <name> <assignment> <literal>
function-definition := <type> <name> <open-parenthesis> parameters <close-parenthesis> statement-block |
                      <type> <name> <open-parenthesis> <close-parenthesis> statement-block |
                      <void> <name> <open-parenthesis> parameters <close-parenthesis> statement-block |
                      <void> <name> <open-parenthesis> <close-parenthesis> statement-block
parameters := <type> <name> | parameters <list-separator> <type> <name>
statement-block := <open-brace> statement-list <close-brace>
statement-list := statement <statement-separator> | statement-list statement <statement-separator>
statement := declaration | assignment-statement | function-call | <break> | <continue> | <return> | <return>
     tern-exp | if-block | while-block | for-block | <comment> statement
assignment-statement := <name> <assignment> tern-exp | <name> <open-square> term <close-square> <assignment>
     tern-exp
function-call := <name> <open-parenthesis> arguments <close-parenthesis> | <name> <open-parenthesis>
     <close-parenthesis>
arguments := tern-exp | arguments <list-separator> tern-exp
if-block := <if> <open-parenthesis> tern-exp <close-parenthesis> statement-block |
           <if> <open-parenthesis> tern-exp <close-parenthesis> statement-block <else> statement-block
while-block := <while> <open-parenthesis> tern-exp <close-parenthesis> statement-block
for-block := <for> <open-parenthesis> <statement-separator> <statement-separator> <close-parenthesis>
     statement-block |
           <for> <open-parenthesis> <statement-separator> <statement-separator> statement <close-parenthesis>
                statement-block |
           <for> <open-parenthesis> <statement-separator> tern-exp <statement-separator> <close-parenthesis>
                statement-block |
           <for> <open-parenthesis> <statement-separator> tern-exp <statement-separator> statement
                <close-parenthesis> statement-block |
           <for> <open-parenthesis> statement <statement-separator> <statement-separator> <close-parenthesis>
                statement-block |
           <for> <open-parenthesis> statement <statement-separator> <statement-separator> statement
                <close-parenthesis> statement-block |
           <for> <open-parenthesis> statement <statement-separator> tern-exp <statement-separator>
                <close-parenthesis> statement-block |
           <for> <open-parenthesis> statement <statement-separator> tern-exp <statement-separator> statement
                <close-parenthesis> statement-block
tern-exp := or-exp | <open-parenthesis> or-exp <close-parenthesis> <tern-operator1> tern-exp <tern-operator2>
     or-exp
or-exp := and-exp | or-exp <or> and-exp
and-exp := eq-exp | and-exp <and> eq-exp
eq-exp := rel-exp | rel-exp <eq-operator> rel-exp
rel-exp := add-exp | add-exp <rel-operator> add-exp
add-exp := mul-exp | add-exp <add> mul-exp
mul-exp := exp | mul-exp <mul> exp
exp := term | <unary-operator> term
term := <name> | <value> | <open-parenthesis> tern-exp <close-parenthesis> | <name> <open-square> term
     <close-square> | function-call
```

Symbols marked in angular brackets (<>) are terminal symbols, and the other symbols are non-terminals.