MiniC Language Manual

Name: Swetanjal Dutta
Roll Number: 20171077

31/08/2020

1 Introduction

MiniC is a small but powerful programming language designed by taking inspiration from **C**, **C++**, **JAVA** and **Python**. This detailed manual describes the syntax and semantics of the MiniC Programming language. It provides all the necessary details for a programmer to start programming in this tiny language.

2 Data Types

MiniC offers the following data types:

1. **int**: This data type is used to store integer data in the MiniC Programming Language.
2. **uint**: This data type is used to store unsigned integer data.
3. **float**: This data type is used to store floating point numbers i.e numbers with a decimal point.
4. **char**: This data type is used to store a single alphanumeric character enclosed within single quotes.
5. **string**: This data type is used to store multiple alphanumeric characters enclosed with in double quotes.
6. **FILE**: This data type is used to store file descriptors.

MiniC allows to form compound data structures like arrays using the above datatypes.

3 Operators and Operator Precedence:

Precedence	Operator	Description	Associativity
1	* / %	Multiplication, Division, Remainder	left to right
2	+ -	Addition, Subtraction	left to right
3	< <=	Relational Operators <, ≤	left to right
4	> >=	Relational Operators >, ≥	left to right
5	== !=	Relational =, ≠	left to right
6	&&	Logical AND	left to right
7	||	Logical OR	left to right
8	?:	Ternary conditional	right to left

4 Semantics

4.1 if statement

Conditionally executes code. Used where code needs to be executed only if some condition is true.

Syntax:

```
if ( expression ) statement_true                          (1)
if ( expression ) statement_true else statement_false     (2)
```

expression must be of type **bool**. *statement_true* and *statement_false* are blocks.

4.2 for loop

Executes *init-statement* once, then executes *statement* and *iteration_assn* repeatedly, until the value of *condition* becomes false. The test takes place before each iteration.

Syntax:

```
for(init-statement; condition; iteration_assn)statement
```

init-statement must be a comma separated list of assignments. *condition* must be a boolean expression. *iteration_assn* must be a comma separated list of assignments. *statement* is a <block>.

4.3 while loop

Executes *statement* repeatedly as long as *condition* is **true**.

Syntax:

```
while(condition)statement
```

condition must be a boolean expression. *statement* is a <block>.

4.4 break

A keyword that causes enclosing loop to terminate.

Syntax:

```
break;
```

4.5 continue

Causes the remaining portion of the enclosing loop body to be skipped, and proceed to the next iteration of the loop.

Syntax:

```
continue;
```

4.6 return

Terminates the current function and returns the specified value (if any) to its caller.

Syntax:

```
return value;          (1)
return;                (2)
```

value must be of same data type as mentioned in return type of function signature. *value* should not be present and (2) should be used if function return type is **void**.

4.7 Output

The print()/println() functions are used to display an expression to the standard output stream. println() prints the provided expression and moves the cursor to a newline.

Syntax:

```
print(expr);      (1)
println(expr);    (2)
```

expr can be of data type int, uint, bool, char or string.

4.8 Input

The read_int()/read_char()/read_string()/read_float() are used to read data of the particular type from the standard input stream.

Syntax:

```
int a;
a = read_int();
```

Similar usage for read_char(), read_float(), read_string() and read_bool().

4.9 Assignment:

The = operator is used for assignment. MiniC uses value-copy semantics for assignment.

Syntax:

```
identifier = expr;
```

expr must be of the same data type as that of identifier type.

5 Macro Syntax Specification using Context Free Grammars

5.1 Meta Notation:

- <foo> means foo is a non terminal.
- **foo**(in bold font) means foo is a terminal i.e a token.
- $[x]$ means zero or one occurrence of x i.e x is optional. Note that ***brackets in quotes*** i.e '[' and ']' are terminals.
- x^* means zero or more occurrences of x.
- x^+ means one or more occurrences of x.
- x^+, means a comma separated list of one or more xs. Comma is a terminal.
- {} i.e large braces are used for grouping. Note that ***braces in quotes*** i.e '{' and '}' are terminals.
- | separates alternatives.
- Punctuation like round brackets, braces, semicolons and commas are terminals. Please note that they have not been written in bold in the CFG.

5.2 Production Rules:

1. <program> → <decl>$^+$
2. <decl> → <var_decl> | <method_decl>
3. <var_decl> → <type> <identifier>$^+$, ;
4. <method_decl> → {<type> | **VOID**} **ID** ([{<type> <identifier>}$^+$,]) <block>
5. <block> → '{' <var_decl>* <statement>* '}'
6. <type> → **INT** | **UINT** | **BOOL** | **CHAR** | **FILE** | **STRING** | **FLOAT**
7. <statement> → <assignment>$^+$, ;
   | <method_call>;
   | **IF** (<expr>) <block> [**ELSE** <block>]
   | **FOR** ([<assignment>$^+$,]; [<expr>$^+$,]; [<assignment>$^+$,]) <block>
   | **WHILE** (<expr>) <block>

| **BREAK**;
| **CONTINUE**;
| <block>
| **RETURN** [<expr>];
| **PRINT** (<expr>);
| **PRINTLN** (<expr>);

8. <assignment> → <identifier> **ASSIGN** <expr>
9. <method_call> → **ID** ([<expr>$^{+}$,])
10. <expr> → <expr8>
11. <expr8> → <expr8> **THEN** <expr8> **OTHERWISE** <expr8>
    | <expr7>
12. <expr7> → <expr7> **OR** <expr6> | <expr6>
13. <expr6> → <expr6> **AND** <expr5> | <expr5>
14. <expr5> → <expr5> **EQ** <expr4> | <expr5> **NE** <expr4> | <expr4>
15. <expr4> → <expr4> **GT** <expr3> | <expr4> **GE** <expr3> | <expr3>
16. <expr3> → <expr3> **LT** <expr2> | <expr3> **LE** <expr2> | <expr2>
17. <expr2> → <expr2> **ADD** <expr1> | <expr2> **SUB** <expr1> | <expr1>
18. <expr1> → <expr1> **MUL** <expr0>
    | <expr1> **DIV** <expr0>
    | <expr1> **MOD** <expr0>
    | <expr0>
19. <expr0> → <identifier>
    | <literal>
    | <method_call>
    | **NOT** <expr>
    | **SUB** <expr>
    | (<expr>)
    | **READ_INT**()
    | **READ_CHAR**()
    | **READ_BOOL**()
    | **READ_FLOAT**()
    | **READ_STRING**()
20. <identifier> → **ID** | **ID**{'[' <expr> ']'}*
21. <literal> → **INT_LIT** | **FLOAT_LIT** | **CHAR_LIT** | **STRING_LIT** | <bool_lit>
22. <bool_lit> → **TRUE** | **FALSE**
23. <arithmetic_op> → **ADD** | **SUB** | **MUL** | **DIV** | **MOD**
24. <relational_op> → **LT** | **GT** | **LE** | **GE**
25. <conditional_op> → **AND** | **OR**
26. <equality_op> → **EQ** | **NE**

5.3 Start Symbol:

- program

6 Micro Syntax Specification using Regular Expressions

6.1 Meta Notation:

- Token Type → Lexeme
- [x] matches exactly one occurrence of regular expression(x).

6.2 Rules:

1. FALSE → false
2. TRUE → true
3. NOT → !
4. NEGATE → ˜
5. VOID → void
6. INT → int
7. FILE → FILE
8. FLOAT → float
9. STRING → string
10. UNINT → uint
11. CHAR → char
12. BOOL → bool
13. THEN → ?
14. OTHERWISE → :
15. FOR → for
16. WHILE → while
17. IF → if
18. ELSE → else
19. BREAK → break
20. CONTINUE → continue
21. RETURN → return
22. ADD → +
23. SUB → -
24. MUL → *
25. DIV → /
26. MOD → %
27. LT → <
28. GT → >
29. LE → <=
30. GE → >=

31. AND → &&
32. OR → ||
33. EQ → ==
34. NE → !=
35. ASSIGN → =
36. PRINT → print
37. PRINTLN → println
38. READ_INT → read_int
39. READ_CHAR → read_char
40. READ_BOOL → read_bool
41. READ_FLOAT → read_float
42. READ_STRING → read_string
43. , → ,
44. ; → ;
45. (→ (
46.) →)
47. { → {
48. } → }
49. INT_LIT → [0-9][0-9]*
50. FLOAT_LIT → [0-9][0-9]*(.[0-9][0-9]*)?
51. CHAR_LIT → '[a-zA-Z0-9 _.,;]' | '\[nt]'
52. ID → [a-zA-Z_][a-zA-Z0-9_]*
53. STRING_LIT → "[a-zA-Z0-9 _.,;\]*"

7 Lexical Considerations

1. All keywords and identifiers are case sensitive.
2. Keywords are reserved words. Identifiers cannot have the same name as any of the keywords.
3. White space may appear between lexical tokens.
4. Keywords and identifiers must be separated by white spaces.
5. The longest sequence of matching characters forms a token. For example, **intlr** is considered to be an identifier and not parsed as the keyword **int** followed by identifier **lr**.

8 Semantic Checks

1. Type Checking: The MiniC Language supports many data types that can appear within expressions. It is essential to check the compatibility of applying various operators on different types of data. For example, we cannot apply addition operator on bool data types. Like wise an expression returning a bool data type can be assigned to variables with type bool only. It cannot be assigned to say a variable with type int. Further more at certain places, an expression of a particular datatype is expected. For example, the <expr> within **if** expects a boolean. Another example being that of array indexes, which expect expressions to have a value of int datatype which is positive or zero. All these checks are essential during compilation so that program can be executed properly.

2. No identifier can be used before it is declared.

3. The program must contain a **main()** method from where execution of the program begins. This function should have no parameters.

4. The expression in a return statement must have the same type as the declared result type of the enclosing method definition.

5. All **break** and **continue** statements must be contained within the body of for/while loops.

6. The number and types of arguments in a method call must be the same as the number and types of the formals in the function signature.

7. If a method call is used as an expression, the method must return a result.