MiniC Language Manual

Name: Vaibhav Garg
Roll number: 20171005

Introduction

The MiniC language is inspired from C and has its basic functionality. This manual contains all the information to write a program in MiniC and serves as a reference for the syntax and language semantics. Some basic points to consider:

1. It is case-sensitive, which means variable '*X*' is different from variable '*x'.*
2. KeyWords like '*for','while'* cannot be used to name variables.
3. Comments can be written using *'#'.* The will be ignored while compiling the code.
4. The words *'variable'* and *'identifier'* have been used interchangeably in the manual.
5. The file extension for source text should be *.mc*

Data Types

The language comes equipped with the following data-type primitives:

- 64-bit signed Integer: Used to store an integer and is declared using the '*int'* keyword.

```
int a = -4;
```

- 64-bit unsigned integer: Used to store a '+ve' integer and is declared using the '*uint'* keyword.

```
uint a = 4;
```

- Character: Used to store a character and is declared using the '*char'* keyword.

```
char c = 'x';
```

- Boolean: Used to store a boolean variable and is declared using the '*bool'* keyword.

```
bool a = true;
```

These basic primitive data-types can further be grouped using 1D arrays and 2D arrays.

```
int a[3] = {4, 5, -13};
char b[6] = {'H', 'e', 'l', 'l', 'o', ‘\0’};
```

***Note:** Assigning a value to a variable before declaring its type will lead to an error.*

Operations

1. MiniC provides functionality for 5 binary operations: *Add*, *Subtract*, *Multiply*, *Divide* and *Modulus*.
2. It comes with 3 boolean operators as well: *And, Or, Not.*
3. To make looping easier, it also comes with increment(++) and decrement(--) unary operators.
4. All the basic comparison operations(>=, ==, <=, >, <, !=) are also available.

Scope Rules

Variables declared globally can be used anywhere in the program, however variables declared inside a function cannot be referred inside other functions unless sent as parameters, similar to what happens in C/Python. Functions have their own scope in which new variables can be declared.

Statements

- Assignment statement: Stores the value of expression to the variable.

  ```
  var = expr;
  var++; // Equivalent to 'var = var + 1;'
  ```

- If-then branching: Depending on the value of boolean expressions, executes the statements present in the block.

  ```
  if(expr) statementBlock
  ```

- If-then-else branching: If the boolean expression is True, executes statementBlock1, otherwise executes statementBlock2.

  ```
  if(expr) statementBlock1 else statementBlock2
  ```

- Branched assignment: Depending on the boolean expression, variable is assigned val1 if true, otherwise val2.

  ```
  var = (expr) ? val1 : val2;
  ```

- For loop: The init_assgn statement(optional) is executed just before the loop begins, the boolean expression is used to terminate the loop. The step_assgn statement(optional) is executed after every iteration of the loop

  ```
  for(<init_assgn>; expr; <step_assgn>) statementBlock
  ```

- While loop: Keeps on executing the statements until the boolean expression is false.
  ```
  while(expr) statementBlock
  ```
- Break statement: Used only inside a loop to forcefully break out of the loop.
  ```
  break;
  ```
- Continue statement: Used only inside a loop to skip the following statements in the current iteration.
  ```
  continue;
  ```
- Return statement: Exits from a function by giving the expression value to the calling function.
  ```
  return expr;
  ```

Functions

Subroutines which have a unique signature composed of the return type, function name, parameter list. They are helpful in recursion.

```
<Data-type> functionName(<Data-type> p1, <Data-type> p2, ...){
    statement*
    return expr;
}
```

Input/Output

Next integer can be read from STDIN and stored using the function ***readInt()***, similarly ***readChar()*** can be used to read the next character from STDIN and ***readString(ident)*** can be used to read next line from STDIN.Similarly, printing to STDOUT can be done using the ***print(..., end='\n')*** function, which takes comma separated values(the ones that need to be printed) as parameters.

Semantic Checks

1. *Function Return value Check:* A function should always return the same data-type as defined in the function signature.
2. *Type Check:* During assignment statement, the data-type of both the sides should be the same.
3. *Loop-condition Check:* The terminating expression used in While/For loop should always return a boolean data-type.
4. *Name Check:* Function and variables shouldn't be declared/defined more than once.
5. *If-condition Check:* The branching condition expression should always return a boolean data-type.

Micro-Syntax

Micro-Syntax helps the lexical analyzer to break the given source text into relevant tokens. These token-types are dependent on the macro-syntax and can be specified using Regular Expressions.

Token	**Regex**
Integer	int
UnsignedInt	uint
Character	char
Boolean	bool
For	for
While	while
If	if
Else	else
Break	break
Continue	continue
Return	return
BoolConstant	true | false
IntConstant	[-]?[0-9][0-9]*
CharConstant	'*ASCII*{0..127}'
Identifier	[a-zA-Z][a-zA-Z0-9]*

Assign	=
IncrementOp	++
DecrementOp	--
AddOp	+
SubtractOp	-
MultiplyOp	*
DivideOp	/
ModOp	%
NotOp	!
AndOp	&&
OrOp	||
GTEOp	>=
LTEOp	<=
EqualOp	==
GTOp	>
LTOp	<
NotEqualOp	!=
LPar	(
RPar	)
LCurly	{
RCurly	}
Semicolon	;
LSquare	[
RSquare	]
Comment	#

Two important rules are followed while tokenizing to avoid ambiguities:

1. Always prefer a longer prefix to a shorter prefix.

2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the specification.

Macro-Syntax

Context Free Grammars are a very efficient and interpretable way to specify the macro-syntax. To verify the validity, all we need to do is to check whether or not the program belongs to language described by the CFG.

Meta notation used:

x (in bold) means that x is a terminal, they will be lowercase
x means that x is a non-terminal, they will be capitalized
<x> means x is optional, zero or one occurrence
x* means 0 or more occurrences of x
x+ mean 1 or more occurrences of x
x+, comma separated list of one or more x's
| separates production alternatives
ϵ Absence of tokens

CFG

Program: Decl+
Decl: VariableDecl **|** FunctionDecl
VariableDecl: Variable **;**
Variable: Type **identifier**<**[**Expr**]**><**[**Expr**]**>
Type: **int | uint | char | bool**
FunctionDecl: Type **identifier (**Params**)** StatementBlock
Params: Variable+, | ϵ
StatementBlock: Statement | **{** VariableDecl* Statement* **}**
Statement: Assign | For | While | If | If-Else | Break | Continue | Return
Assign: **ident**<**[**Expr**]**><**[**Expr**]**> **=** Expr **;** | **identifier**<**[**Expr**]**><**[**Expr**]**>UnaryOp **;**
For: **for(**<Assign>**;** Expr**;** <Assign>**)** StatementBlock
While: **while(**Expr**)** StatementBlock
If: **if(**Expr**)** StatementBlock
If-Else: **if(**Expr**)** StatementBlock **else(**Expr**)** StatementBlock
Break: **break;**
Continue: **continue;**
Return: **return** Expr**;**
Expr: Expr BinaryOp Expr | **(**Expr**)** | **-**Expr | **!**Expr | Call | Constant | **identifier**<**[**Expr**]**><**[**Expr**]**>
Call: **identifier(**CallParams**)**
CallParams: Expr+, | ϵ
BinaryOp: **+ | - | * | / | % | && | || | <= | == | >= | > | < | !=**
UnaryOp: **++ | --**
Constant: **IntConstant | UnsignedIntConstant | CharConstant | BoolConstant**

Note: *For readability, the operators are represented by the lexeme itself and not the Token, for example, instead of 'GTOp', '>' is being used.*