PROGRAMMING LANGUAGE DESIGN

Sumukh S

1 Details about the language

The language designed is a simple programming language, with similarities to Python and C++, with each statement ending with a semicolon. The CFG for the language is given at the end of this document.

1.1 Data types

The datatypes that are valid for the programming language are as follows:

- int
  Used to store integers only, default value set to 0.
  Declared like < int x; >
- unsigned int
  Used to store only whole numbers, default value set to 0.
  Declared like <unsigned int x; >
- char
  Used to store any one single character, default value set to 'null'.
  Declared like <char x;>
- bool
  Used to store only 0 or 1/True or False values, default value being 0/False.
  Declared like <bool x;>
- 1-D and 2-D arrays
  Used to store a series of values of chosen variable type, int/unsigned int/char/bool.
  Declared like < <variable type> x[<number of elements in the array>];> for 1 D array
  Eg: bool x[10];
  < <variable type> x[<number of elements in a column>][<number of elements in a row>];>
  Eg: char x[14][15];

1.2 Arithmetic Operations

This would involve addition (+), subtraction (-), multiplication (*) and division (/). We would also like to add a modulus operation (%), which gives the remainder of a number when divided by another. Am operation can involve both constants and variables
A single expression can have multiple operations involved.
Eg: x = y + 4*t - 3%2 + 8/4;

1.3 Boolean operations

This involves boolean operations like and (&&), or (||), and logical not (!). They always results in boolean values, ie 0 or 1/True or False.

1.4 Input and Output

We would like to get the values of some variables from the user during run time, and we also would like to show the value of a particular variable to the user. These are used for that:

- input
  Takes a value from the user and stores it in a variable/identifier
  Eg: <input(x);> takes the value from the user and stores it in x.
- output
  Shows user the value of an identifier by printing it to the terminal/console.
  Eg: <print(y[4]);> prints the value stored at y[4] to the console.

1.5 Control Statements

These statements allow for the execution of a particular statement block/s only if the condition/s are true.

- If then; If then else
  if(condition/s) <statementBlock1> (else <statementBlock2>)
  The statementBlock1 is executed when the condition/s is/are true. Else, the statementBlock2 is executed. Here, we have to observe that the else statement section is optional, so one can have just the 'if then' block.
- Ternary operator
  (<condition/s>)?<statement1>:<statement2>
  The statement1 is executed if the condition/s are true. Else, statement2 is executed. This can be considered a compact version of if-else block.
- For loop
  for({expression1}; expression2; {expression3} statementBlock
  Here, the statementBlock is executed repeatedly, until the expression2 has a True/non-zero value. We also have to note that the expressions 1 and 3 are not necessary.
  The loops might also contain break statements to control the flow of the program.
- While loop
  while(expression) statementBlock
  Here, the statement block is executed repeatedly until the expression is True/non-zero.
  The loops might also contain break and continue statements to control the flow of the program.
- Break statements
  break;
  Control exits the loop upon execution of this statement. Can only be used inside a loop.

1.6 Functions

Values can be passed to functions, where they execute a set of statements, and the function might or might not return a value back, ie might return a 'null' value. The variable type of the value that a function will return is decided at the time of function declaration.
A function is declared as follows:
functionDeclaration := {variabletype funcName ({variableType id}*) statementBlock {returnStatement;}?} | {void funcName ({variableType id}*) statementBlock}
'id' represents identifier.
A function with variable type other than void will always return some value, and hence the return statement at the end of the statement block. If its missing, the function will return a default value of 1 for correct execution of the function, and 0 for otherwise.

2 Semantic Analysis

- The variable has to be first declared, and then assigned a value later. It cannot be assigned a value at declaration.
  Eg: int x = 10 is invalid. int x; has to be there first, and then x=10;

- Multiple variables can not be declared in a single single line.
  Eg: int x,y,z; is an invalid statement.
- The variables can not be type casted
- The variable types have to match in the expressions, else raise a type error
  Eg: int x=7;
  char v='u';
  int i;
  i=x+v; This raises a type error.
- The input statements have to take care of any mismatch of the input and the variable type
  Eg: int x;
  input(x) when given a value of 'r' raises an error. It can only store integral values.
- The keywords mentioned in the list can not be used as identifiers.
  Eg: int input; is invalid as 'input' is a keyword

3 Grammar of the language

The following is the grammar of the language, ie CFG

3.1 Micro syntax

These are the rules that govern the lexical structure of a language, specified using regular expression and implemented using finite state machines.

letter := [A-Za-z]

digit := [0-9]

number := -?{digit}+({̇digit}+)?

characters := .

positiveInteger := {digit}+

id := {letter}{letter | digit | '_' | '$'}* - keywords

keywords := 'var' | 'if' | 'input' | 'output' | 'while' | 'break' | 'int' | 'unsigned' | 'char' | 'bool' | 'for' | 'return' | 'print'

3.2 Macro syntax

Grammar of the language written using CFGs and implemented using push down automata.

program := decl+

decl := variableDecl | functionDecl | classDecl

variableDecl := vartype id | vartype id'['positiveInteger']' | vartype id'['positiveInteger']'

vartype := 'int' | 'unsigned int' | 'char' | 'bool'

functionDecl := {vartype id ({variableType id}*) statementBlock {returnStatement;}?} | {void id ({variableType id}*) statementBlock}

stmntBlock ::= { variableDecl* stmnt* }

stmnt := expr | ifBlock | whileLoop | forLoop | breakStmnt | returnStmnt | printStmnt | inputStmnt

expr := {LValue = Expr} | {constant} | {id} | {funcCall} | {expr '+' expr} | {expr '-' expr} | {expr '*' expr} | {expr '/' expr} | {'-' expr} | {condition}+

condition := {expr '<' expr} | {expr '<=' expr} | {expr '>' expr} | {expr '>=' expr} | {expr '==' expr} | {expr '!=' expr} | {expr '&&' expr} | {expr '||' expr} | {'!' expr} | {ternCond}

constant := letter | number | True | False | null

ifBlock := if(expr) stmntBlock {else stmntBlock}?

ternCond := '('expr')' '?' stmnt : stmnt

forLoop := for({expr}?; {expr}; {expr}?) {stmntBlock {break}?}

whileLoop := while'('{expr}')' {stmntBlock {break}? }

returnStmnt := "return" expr;

break := "break";

funcCall := id '(' (id,)* ')';

printStatement := print '(' expr (',' expr)* ')';

inputStmnt := input '(' id ')'