KaGa

Abhinav Gupta & Alok Kar

August 16, 2019

1 Introduction

KaGa is a very small and simple programming language, highly inspired by C and Python. This language manual describes the syntax and semantics of the language constructs in KaGa. It provides all the information that is essential for understanding the language and writing programs in it.

2 Data Types

The built-in data types in KaGa are integer (both signed and unsigned), boolean and character. Arrays (both one and two dimensional) can be a construct of any of the data types. It is essential to pre-declare the variables because the data type of the variable must be known beforehand. KaGa will not automatically assign the data-type of the variable at the time of the assignment of values. For instance,
a=5;
This statement would throw an error because the variable has not been pre-declared.
int a = 5;
Here, the variable 'a' of type integer is declared first, and then assigned the value. So this is legal.

The keywords for the data-types are as follows:

- Integer (Signed by default) - int
- Character - char
- Boolean - bool
- Unsigned integer - uint

3 Operations

We have four types of arithmetic operations in KaGa, all of which are binary operations i.e they operate upon 2 values. The 5 operations are Addition, Subtraction, Multiplication, Division and Modulus.
Staying true to the fundamentals of Boolean Algebra and for ease in digital design, KaGa also supports the boolean operations of AND, OR and NOT.
We have comparison operations as well. Those include less than, greater than, less than equal-to, greater than equal-to, equal-to, not equal-to.

4 Lexical Grammar

Here are the lexical considerations:

- KaGa is case sensitive. So 'a' and 'A' would be two different variables.
- Keywords cannot be used as variables. However, since KaGa is case sensitive, any other version of a keyword can be used as a variable name. for instance, 'For' can be used to identify a variable as the keyword is 'for'.
- Keywords in KaGa are: int, char, bool, uint, AND, OR, NOT, if, else, for, while, break, continue, return, Print, ReadInteger, ReadChar, NewArray, New2DArray
- Comments can be written with '#'. These would not be considered in the code and are used to increase the readability of the code.

5 Semantics

If-then-else

We have the basic if-then-else statement in our language.

$$if(\ expr\ )\ stmt1\ else\ stmt2;$$

This evaluates to stmt1 if expr is true, stmt2 otheriwse.

For loop

The second type of control flow we have is For-loops.

$$for(< expr1 >;\ expr2\ ;\ < expr3 >\ )\ stmt;$$

In this, 'expr1' is the initialization of the looping variable. 'expr2' is the loop condition, which when not true stops the loop. 'expr3' is the increment. Both 'expr1' and 'expr3' need not be written inside the parentheses, initialization can be done beforehand and increment can we done within the loop.

While-loop

We have while-loops in our language as well.

$$while(\ expr\ )\ stmt;$$

This loop executes the 'stmt' as long as the 'expr' evaluates to True.

Break Statement

We have break statements as well, which stop the ongoing loop.

$$break;$$

This statement only stops the closest loop. If there is a nested loop and the break statement is in the inner loop then it will only stop the inner loop, not the outer loop.

Continue Statement

This is similar to break statemenet.

$$continue;$$

It is similar to break statement but instead of stopping the loop it just initiates the next increment and conditional check in for loop, and skips to the start of the loop in a while loop.

Return Statement

This statement is the returns the value of a function.

$$return\ expr;$$

It returns the value of the expression.

Input Output

This statement outputs the value of expressions given to it. The expressions given would be separated using a comma.

$$Print(expr+,);$$

Input statements read either an integer or a character at a time and they return that value to a variable.

$$ReadInteger() \mid ReadChar()$$

Functions and Function Calls

Function declarations in this language are very similar to C.

$$Type\ \textbf{ident}\ (\ Formals\ )\ StmtBlock$$

First we define the return type of the function, then we give the functions an identity, afterwards we specify the arguments to be passed and last but not least we have a statement block which must contain a return statement except if it is a void type function. Formals in this are just one or more variables with their type.

$$\textbf{ident}\ (Actuals\ )$$

Function calls are basically the name of function followed by the list of expressions which are passed as an argument. Actuals are one or moer expressions separated by commas.

Statement and Statement Blocks

A statement block is a collection of variable Declarations and statements which starts and ends with curly braces.

$$StmtBlock\ ::=\ \{\ VaribaleDecl*\ Stmt*\ \}$$

A statement itself could be multiple things, for example, an expression, an if statement, a loop etc. A statement could be a statement block as well. This circular definition allows us to have nested blocks in our language.

Expressions

Expressions could be of many forms, they could be evaluated to some final value, or some assignment or a function call.

Array Forming

Arrays of any type can be formed using NewArray or New2DArray functions. The former takes an expression, a type and another expression an returns a one dimensional array with initial value to be the same and equal to the value of second expression. The first expression gives the size of the array.

$$NewArray(\ expr1,\ Type,\ <epxr2>\ );$$

Similarly, for 2d Array the first two expressions give the row size and column size respectively for the array. The third argument gives the type and the last argument gives the initial values for the array elements. 1-D Char arrays can also be formed using stringConstant.

$$New2DArray(\ expr1,\ expr2,\ Type,\ <expr3>\ );$$

6 Context Free Grammar for KaGa

The reference grammar is given in a variant of extended BNF. The meta-notation used:

- **x** : (in bold) means that **x** is a terminal i.e., a token. Terminal names are also all lowercase except for those few keywords that use capitals.
- x : means x is a nonterminal. All nonterminal names are capitalized.
- <x>: means zero or one occurrence of x, i.e., x is optional
- x* : means zero or more occurrences of x
- x+ : means one or more occurrences of x
- x+, : a comma-separated list of one or more x's (commas appear only between x's)
- | : separates production alternatives
- ϵ : indicates epsilon, the absence of tokens

Macro-Syntax

Program ::= Decl+
Decl ::= VariableDecl |FunctionDecl |ClassDecl |InterfaceDecl
VariableDecl ::= Variable **;**
Variable ::= Type **ident** | Type **ident**[<Expr>]
Type ::= **int** |**bool** |**char** |**uint**
FunctionDecl ::= Type **ident** **(** Formals **)** StmtBlock | **void ident** **(** Formals **)** StmtBlock
Formals ::= Variable+**,** |ϵ
StmtBlock ::= { VariableDecl* Stmt* }
Stmt ::= <Expr>; |IfStmt |WhileStmt |ForStmt |BreakStmt |ContStmt |ReturnStmt |PrintStmt |StmtBlock

IfStmt ::= **if** (Expr) Stmt <**else** Stmt>
WhileStmt ::= **while** (Expr) Stmt
ForStmt ::= **for** (<Expr>**;** Expr **;**<Expr>) Stmt
ReturnStmt ::= **return** Expr **;**
BreakStmt ::= **break;**
ContStmt ::= **continue;**
PrintStmt ::= **Print** (Expr+**,**);
Expr ::= LValue = Expr | Constant | LValue | Call | (Expr) | Expr + Expr | Expr - Expr |
Expr * Expr | Expr / Expr | Expr % Expr | Expr < Expr | Expr <= Expr | Expr > Expr |
Expr >= Expr | Expr == Expr | Expr != Expr | Expr && Expr | Expr || Expr | !Expr |
ReadInteger() | **ReadChar()** | **NewArray** (Expr **,** Type **,** <Expr>) |
New2DArray (Expr **,** Expr **,** Type **,**<Expr>)
LValue ::= **ident** |**ident**[Expr]
Call ::= **ident** (Actuals)
Actuals ::= Expr+, |ϵ
Constant ::= **intConstant** |**boolConstant** |**charConstant** |**stringConstant**

Micro-Syntax

These are the keywords whose values are given by these regular expressions. Other keywords are just those words/characters written in bold.

$$ident -> \{a...z\ A...Z\}\{a...z\ A...Z\ 0...9\}*$$

$$stringConstant -> "Ascii\{0..255\}*"$$

$$intConstant -> \{0...9\}\{0...9\}*$$

$$boolConstant -> \{True, False\}$$

$$charConstant -> \{a...z\ A...Z\}$$

7 Semantic Checks

7.1 Types

The main semantic check would be to make sure that each assignment, or expression has the same type which is required.

7.2 Evaluation - If

The evaluations pf if-else statement would be based on the value of the expression given, as long as the expression is true the first statement would be executed, if the value is false then if the else statement is present that would be executed else it would continue to the next statement.
If the expression value is not even of Type 'bool' then there would be semantic error.

7.3 Evaluation - While

The while statement will execute until the expression inside the parentheses remains true. Similar to if statement if the expression is not of type 'bool' then there would be an error.

7.4 Evaluation - For

For statement has three expressions it depends on. The first is an initialization of loop variables, second is condition and the third is increment value. The loop runs till the condition is true. It must always have a condition expression but other two are not necessary.

7.5 Evaluation - Functions

At the time we see a function call, we should check the types of each argument passed and see if it matches the function definition. Also each function should have a return statement. The number of arguments required and passed should be same.
There will be a special function named 'main' which is where the program will start its execution from. All the other functions defined separately would be accessible by all other functions. Any variable declared inside any function will not be accessible to any other function, unless we pass its value to a function. Any variables not declared inside any of the functions would be accessible by all the functions and would be called as global variables.