Language Design Manual

Compilers, Monsoon, 2019

Swapnil Gupta 20161174
Vivek Kaushal 20161071

1. Introduction

This document is a manual for **Kauta,** a programming language created for the coursework of Compilers, Monsoon, 2019 under professor Suresh Purini. This document contains information regarding the syntax - context free grammar (CFG), semantics and general instructions for our programming language. We have also included 10 sample codes for Kauta for easy understanding of the reader.

Sample code in Kauta:

```
func main ()
begin
  print("Hello World")
end
```

Output for the sample code :

```
$ Hello World
```

Kauta is not a fully functional language, though it supports a fairly large selection of features that are common in modern high-level languages which would enable a programmer to create a large number of feature rich programs.

2. Specifications

The features that Kauta supports are listed as follows under relevant sub-headings:

2.1 Data Types

1. Signed and unsigned int
2. Double
3. bool
4. Char
5. String
6. 1D and 2D arrays

2.2 Operators

Arithmetic Operations Supported: add, sub, mul, div
Boolean Operations Supported: and, or, not
Control Statements: if-then, if-then-else, for loop, while loop, ternary operator
Functions: Call by value supported, recursion supported

2.3 I/O Routines

Kauta supports basic I/O routines that are limited to single value scanning and multiple variable printing in a single routine call.

3. Context Free Grammar

```
Program    → (Var | Func)*
Var        → Type ID (‘=’ Expr)? '\n'
Func       → func Type ID ‘(’ ParamList ‘)’’\n’ Block
ParamList  → (Type ID)* ‘,’ (Type ID)*
Block      → begin‘\n’ Var* Stmt* end’\n’

Stmt → Loc ‘=’ Expr '\n'
Stmt → FuncCall '\n'
Stmt → if Expr then ‘\n’ Block (else‘\n’ Block)?
Stmt → while Expr ‘\n’ Block
Stmt → for ‘\n’ Expr ’\n’ Expr ’\n’ Expr ‘\n’ Block
```

```
Stmt → return Expr? '\n'

Expr → Expr BINOP Expr | ‘(’ Expr ‘)’ | Loc | FuncCall
Expr → Expr ? Expr : Expr
Expr → BoolVal | IntVal | StringVal | CharVal | DoubleVal
Type → uint | char | int | bool | void | string | double
Loc        → ID(‘[’ Expr ‘]’)?(‘[’ Expr ‘]’)?
FuncCall   → ID ‘(’ ArgList ‘)’
ArgList    → ID* ‘,’ ID*
BoolVal    → True | False
Output     → print ’(‘ ArgList ’)’ ‘\n’
Input      → Type? ID ‘=’ scan ’(‘ StringVal? ’)’ ‘\n’
IntVal     → [0-9]+
DoubleVal  → [0-9]+([.][0-9]+)?
StringVal  → [a-zA-Z0-9]* <to be filled>
CharVal    → [a-zA-Z]
```

4. Semantics

4.1 Functions

Functions in Kauta can be declared using the keyword `func`**.** Functions need to have a return type specified, this type specification can also be `void`. Functions in Kauta allow for multiple parameters, each parameter needs to have its type-specified. Recursive function calls are also supported in Kauta.

Kauta has a special function `main` like C which doesn’t take any parameter, returns an `int` and is the starting point for execution of the program.

Semantic Check I:

```
func foo (char a)
begin
   int b = a
```

```
   return 2*b
end
```
This would result in an error as despite being CFG compliant, the function foo requires an integer as a parameter, but a character is being passed.

4.2 Variables

Kauta allows for the following data types of variables : `int`, `uint`, `bool`, `double`, `string`, `char`. Besides supporting these data types, one-dimensional arrays and two-dimensional array are also supported. 1D and 2D are accessed using the standard square bracket notation commonly used in C. Declaration of 1D and 2D arrays can be done in both list and individual value formats.

Proper scope is maintained for variables. A variable declared with a scope is accessible throughout the scope, though Kauta doesn't allow for public variable declarations. More information on the scope of variables can be found in the following sections. Variables are assigned using the assignment operator '=', as in : `int a = <Expr>`

Semantic Check II:

```
int x = "hello"
```
This would result in an error as despite being CFG compliant, integers can't store string values and are against the regex specifications of int.

4.3 Control Structures

Kauta allows for the following control structures - `if then`, `if then else`, `while loop`, `for loop`, `ternary operator`. The

construction of these control structures is identical in logic to their implementation in high-level-languages, differences are introduced in syntactic implementation.

The control structures, like codeblocks, rely on `begin` and `end` for the correct declaration of their scope and codeblocks in Kauta.

Semantic Check III:

```
char a = ‘q’
if a > 5
then
begin
print (“hello”)
end
```

This would result in an error as despite being CFG compliant, integers can’t store string values and are against the regex specifications of int.

4.4 Scope

In Kauta, scope is declared for any given code-block through two specialised keywords : `begin` and `end.` Begin indicates the start of a new scope, whereas end symbolises its termination.

Semantic Check IV

```
func foo (char a)
begin
   int b = a
   return 2*b
end

print (b)
```

This would result in an error as despite being CFG compliant, as the variable b is declared inside the scope of the foo function, hence, it can't be accessed outside its scope.

4.5 Function Call

Function calls in Kauta are similar to those of modern high-level programming languages, their syntax constitutes the function name followed by a list of arguments in parenthesis separated by a comma.

Semantic Check V

```
func foo (char a)
begin
   int b = a
   return 2*b
end

foo(a,b,c)
```

This would result in an error as despite being CFG compliant, as the function foo requires exactly one parameter, but 3 are being supplied.

PS: The sample codes are stored along with this documentation file.
