MAlice Language Specification

Magdalena Gocek Varun Verma Harshwardhan Ostwal

December 13, 2012

1 Introduction

This specification defines the language constructs of MAlice, focusing on the syntax and the semantics of the language. The syntax will be defined using the Extended Backus-Naur Form (EBNF) grammar and Regular Expressions form.

2 Syntax Definition

Number → [0-9]+
Letter → '[a-zA-Z]'
Identifier → [a-zA-Z][a-z]∗
FunctionName → Identifier
Data → Number | Identifier
Expr → MONOOP Expr | Data (ϵ | BINOP Expr)
BINOP → '+' | '%' | '/' | '^' | '&' | '|' | '∗'
MONOOP → '∼'
DataType → *'number'* | *'letter'*
Statement → Identifier (*'was a'* DataType [*'too'*]
| *'became'* (Letter | Expr)
| (FunctionName | *'ate'* | *'drank'*)
)
| Expr *'said' 'Alice'*
StatementConjunctions → *'.'* | *'and'* | *'but'* | *','* | *'then'*
StatementList → ϵ | Statement {StatementConjunctions Statement} '.'
Function → *'The looking-glass'* FunctionName *'()' 'opened'* StatementList '*closed*'
Program → {Function}

3 Semantics

A MAlice program makes use of variables and functions which are building blocks for most of the widely used programming languages. It consists of the following components which have been described further in respective sub-sections together with their limitations:

- Data Types
- Variables
- Mathematical Operations
- Functions
- Statements

3.1 Data Types

MAlice supports two basic data types, *number* and *letter*. The letter data type is represented by a single ASCII character enclosed in single quotes (' ') ranging from 'a' to 'z' and 'A' to 'Z'. The number data type is represented by a 16-bit signed integer. MAlice uses the Two's Copmlement representation for the number data type which defines the most significant bit to be the complement. This constraints the data type number to have a specified range of *-65536 to 65535* and introduces the following errors:

- **InvalidAssignment**: This error would be encountered if the value assigned to a variable is out of the range of data type number.
- **IntegerOverflow**: This error would be encountered if the accumulating result or the final result of an expression is out of the range of the integer.

3.2 Variables

Variables in MAlice have a String identifier which consists of letters only. The identifier can start with upper or lower case letter and all the following letters must be lower case. For example, "TExt" and "MyVar" are not valid identifiers whereas "Variable" and "variable" are valid. Variables can hold a value of a type *number* or *letter*. The possible errors encountered when using variables include:

- **InvalidAssignment**: A variable of type *number* cannot be assigned to a variable of a type *letter* and vice versa.
- **InvalidInitialisation**: An identifier name for a variable which has already been initialised cannot be used again. All identifier names must be unique.

3.3 Mathematical Operations

MAlice supports various mathematical operations, the functionality of which is described below with their precedence shown in brackets:

- **'∼'(2)**: Bitwise NOT. The return value is obtained by carrying out logical NOT operation on individual bits of the given argument.
- **'∗'(3)**: Mathematical multiply - returns the product of the given arguments.
- **'/'(3)**: Division operator. A runtime error would be encountered when trying to divide by 0.
- **'%'(3)**: Modulus operator - returns the reminder after division of the given arguments.
- **'+'(4)**: Mathematical add - returns the sum of the given arguments.
- **'&'(8)**: Bitwise AND - the return value is obtained by carrying out logical AND operation on individual bits of the given arguments.
- **'^'(9)**: Bitwise XOR - the return value is obtained by carrying out logical XOR operation on individual bits of the given arguments.
- **'|'(10)**: Bitwise OR - the return value is obtained by carrying out logical OR operation on the individual bits of the given arguments.

3.4 Functions

MAlice has two built-in functions, *'ate'* and *'drank'*. These functions respectively increment and decrement the value of the given argument by 1 and can only be applied to variables of a type number. MAlice also supports user-defined functions which can be used in any other functions within the program. Each user-defined function must start with the keywords *'The looking-glass'* followed by the name of the function with empty brackets. The user can specify the parameters in the brackets. The function body is enclosed within the keywords *'opened'* and *'closed'*. User defined functions in MAlice can accept either 1 or no arguments which are passed by value. The entry point of a MAlice program is the function *'hatta'*. An example of a user-defined function is shown below:

```
The looking-glass Functionname(parameter)
opened
  {body}
closed
```

3.5 Statements

A MAlice program consists of a sequence of various statements. This can be of one of the following forms:

- **Initialisation Statement**: The initialisation in MAlice is done by choosing an identifier for the variable and then defining its data type by using the keyword *'was a'*, for example:

  ```
  Seven was a number. A was a letter.
  ```

  Here 'Seven' and 'A' are the identifieres for two variables of type *letter* and *number* respectively. The identifiers will be used to access these variables in the function they are defined in.

- **Assignment Statement**: The assignment can be done by stating the variable identifier and using the keyword *'became'* followed by a letter, variable, number or mathematical expression, for example:

  ```
  Seven became 2. A became 'e'. Seven became Seven*4.
  ```

- **Print Statement**: The print statement is used to output to the standard output. The print statement is invoked by stating the variable identifier, letter, number or a mathematical expresion followed by the key phrase *'said Alice'*. For example, the following statement would print 24 to standard output.

  ```
  6*4 said Alice.
  ```

- **Function Statement**: The function statement is only used when there is a need to apply one of the MAlice's predefined functions or the functions defined in the program. To do this, we state a number or a variable's identifier followed by the name of the function, for example:

  ```
  Variable drank. Seven ate. Variable arbitaryFunction.
  ```

The statements in a program are separated by the following keywords and symbols: { *'.'* , *','* , *'and'* , *'then'* , *'but'*}. Each statement is executed in sequential order. Any statement which does not comply with the rules of BNF would cause a *SyntaxError*.