Language Specification

Mohamed Eltuhamy James Lawson Alejandro Garcia Orchoa

October 27, 2011

1 Introduction

1.1 Project Background

This document is a specification for the *MAlice Programming Language* and is aimed at programmers who want to familiarise themselves with the syntax and semantics of the language. The MAlice programming language is part of a second year course on compilers from the Department of Computing at Imperial College London. This document has been produced by three students as part of an exercise in writing a compiler for the language.

1.2 Introduction to the Language

The MAlice programming language is a imperative programming language that uses keyphrases instead of keywords to write programs. Instead using symbols to perform the features of the language, we have phrases such as *x is a number then x became 10*. The language is a high-level programming language that is close to 4th generation programming languages such as variants of SQL and Prolog in terms of how close the language read reads to English. The language does not support commenting and function calls currently.

For this project, we have been given a group of 20 MAlice programs and have examined their source code in an attempt to product this language specification. It is based only on the source code of the given 20 programs. The following specification represents our group interpretation of the MAlice programming language.

1.3 Contents

- 1. Introduction. An introduction to the project.
- 2. BNF Grammar. A description of the syntax using Backus-Naur Form.
- 3. Semantics. A description of the meaning of keywords, expressions and features.

2 BNF Grammar

In this section, we have formally defined the syntax of the language using the Backus-Naur Form language specification tool. The non-terminal symbol `<program>` is the start symbol for every MAlice program.

Note: `ASCIICHAR` refers to *any* ASCII character.

```
<letter>          ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n  | o | p | q | r | s | t
     | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R
     | S | T | U | V | W | X | Y | Z

<digit>           ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<number>          ::= <digit> <number>

<variable>        ::= <a> <b>
<a>               ::= <letter> | _ <letter> | _ <a>
<b>               ::= <letter> | <number> | _ | <letter> <b> | <number> <b> | _ <b>

<binaryoperator>  ::= + | * | & | | | ^ | / | %
<unaryoperator>   ::= ~

<expression>      ::= <mathsExpression> | <charExpression>

<mathsExpression> ::= <number> | <variable>
<mathsExpression> ::= (<mathsExpression>)
<mathsExpression> ::= <mathsExpression> <binaryoperator> <mathsExpression>
<mathsExpression> ::= <unaryoperator> <mathsExpression>

<charExpression>  ::= 'ASCIICHAR'
<charExpression>  ::= (<charExpression>)

<type>            ::= number | letter

<declaration>     ::= <variable> was a <type>
<assignment>      ::= <variable> became <expression>

<action>          ::= <variable> ate
<action>          ::= <variable> drank

<program>         ::= <statements>
<statements>      ::= <statement> | <statement> <statements>
<statement>       ::= <sentence> <endSentence>
<sentence>        ::= <declaration> | <assignment> | <action> | Alice found  <expression>

<endSentence>     ::= and | but | then | . | , | too.
```

3 Semantics

3.1 Basic Program Structure

Every MAlice program consists of zero or more statements, separated by an ending command. Ending commands include "`and`", "`but`", "`then`", "`too.`", "`.`" and "`,`". Note how `too.` always ends with a "`.`". Whitespace, such as new lines, empty spaces " ", tabs, etc. are ignored between sentences. Below shows a example of a simple MAlice program.

```
x was a number.
x became 10.
Alice found x.
```

The MAlice programming language supports variables. A variable is a name given to a temporary place in memory. The next sections (cf. Section 3.2, Section 3.3, Section 3.4) explain the semantics of how variables can be used in MAlice. The above example has the variable x which becomes a number. This is variable assignment. x stores the value 10 in its memory location.

When Alice has found a value, the program terminates and the output is what Alice has found. So in the program above, 10 is outputted. A program that does not have `Alice has found <expression>` will not output a value, but will compile provided the rest of the program is correct.

Malice programs are case sensitive. So `x` and `X` are different variables, and `AtE` is different to the keyword `ate` (see later).

3.2 Variable Types

There are only two types in MAlice, and these are **letter** and **number**. Numbers in MAlice represent the natural numbers starting from zero. So `0`, `1`, `2`, `3`, `4`, etc. are of the `number` type.

Numbers are represented in 8 bits. There is no support for signed binary integers. There is no subtraction operation (cf. Section 3.2) and no minus operator. There is no support for floating point representations and no operations for floating point arithmetic.

The variables with the **letter** type can store 8-bit Extended ASCII characters. They can only be assigned using the explicit character in quotes e.g. `'a'`.

3.3 Variable Identification

In MAlice, a variable must have a unique identifier that is chosen when it is first introduced in program execution. The identifier must be a string of alphanumerical characters that can contain zero or more underscores anywhere in the string.

There cannot be identifiers that have only digits. There cannot be identifiers that have only underscores but no other punctuation marks. If a number is used, it must be preceded by at least one letter. Variable names must not contain spaces.

The language has some reserved keywords. These are `number`, `letter`, `was`, `a`, `became`, `ate`, `drank`, `and`, `but`, `then`, `too`, `Alice`, `found`. As previously mentioned, however, we can have variable names which are similar to keywords, but with different case, e.g. `NUMBER` is a valid variable name.

Examples of **valid** identifiers: `x`, `y`, `xy`, `this`, `a_3`, `a3`, `a4b5`, `a__t`, `aat_3`, `a4_t1`, `alice`.

Examples of **invalid** identifiers: `_a`, `a`, `some space`, `hello!`, `0a`, `___`, `_5c`, `Alice`.

3.4 Variable Declaration and Assignment

Example	Explanation
`x was a number. y was a letter`	Declares `x` as a `number` and `y` as a `letter`. The variables can now be used in the program.
`x became 2` `y became 'a'`	Assigns the value `2` to the variable `x` Assigns the value `'a'` to `y`.
`something became 4 + x + 7`	Computes the value of `4+x+7` and assigns that value to the variable The value of `x` is evaluated and used in the expression.

3.5 Operators and Expressions

The MAlice programming language provides 7 binary operators and 1 unary operator to be performed on the integer variables of the language. Two are arithmetic operations and the rest are bitwise operations that operate on the binary representation of the operand(s). These operators only work on `number` types and are undefined on character types.

Name	Symbol	Example	Result
Addition	`+`	`a + b`	Performs integer addition on `a` and `b`
Multiplication	`*`	`a * b`	Performs integer multiplication on `a` and `b`
Division	`/`	`a / b`	Result is the quotient of `a` and `b`.
Modulo	`%`	`a % b`	Result is the remainder of `a` divided by `b`
Bitwise AND	`&`	`a & b`	Bits that are set in both `a` and `b` are set in the result
Bitwise OR	`|`	`a | b`	Bits that are set in either `a` or `b` are set in the result
Bitwise XOR	`^`	`a ^ b`	Bits that are set in `a` or `b` but not both are set in the result
Bitwise NOT	`~`	`~a`	Bits that are set in `a` are not set in the result, and vice versa

The operands of these 8 operators may be either variables of type **number** or a literal (immediate) value. The immediate values must be within the range 0-255 (ie they must be able to be represented by 8 bits). These operators will not work with operands with type **letter**. These expressions allow bracketing to enforce precedence (the order in which the operations are evaluated). As stated earlier, the operators above only work on `number` types, however, bracketing *also* works on `letter`.

We also require that the variable has already been declared in program execution if we use a variable as an operand. We also specify that integer division with a divisor of 0 is forbidden and any results have no significant meaning. We may have an overflow with the addition and multiplication operators. This is a runtime error that our compiler may not be able to see in advance.

The precedence of these operators (and also bracketing), from highest to lowest are:

(1) brackets **(2)** Bitwise NOT `~` **(3)** Multiplication *, Division `/` and Modulo `%` **(4)** Addition `+` **(5)** Bitwise AND `&` **(6)** Bitwise XOR `^` **(7)** Bitwise OR `|`

These operators discussed previously are key for computation to give values for Alice to find. By combining valid operands and zero or more operators we can form expressions that are evaluated at runtime.

Variables of type **number** can be assigned to expressions that use the operators of the previous section, with operands that are numerical constants or other number variables. Variables of type **letter** can only be assigned to a constant ASCII character, but note they (constant ASCII characters) are *also* expressions. So for example, `(((’a’)))` is a valid expression.

3.6 Actions

So far we have looked at statements that involve variable declaration and expression assignment. We will now cover additional statements for manipulating numbers. A statement can contain the past participle of two verbs, to drink and to eat that represent a number being decreased and increased by one respectively.

The `ate` action increments the number, while the `drank` action decrements it. When we decrement zero, the number wraps back round to the largest number that we can represent (255). Similarly, when we increment 255 using `ate`, the number loops back round to zero.

Example	Explanation

`x was a number.`
`x became 2.`
`x ate.`
`Alice found x.`
	Declares x as a number and increments it.
	So the program will output 3.

`x was a number.`
`x became 6.`
`x drank.`
`Alice found x.`
	Declares x as a number and increments it.
	So the program will output 5.