MAlice language specification

Jamie Ridler, Moritz Roth

November 5, 2012

Contents

1 Keywords

1.1 Reserved keywords

There are several keywords that can't be used as identifiers in MAlice programs:

```
a | Alice | and | ate | became | but | closed | drank | letter |
number | opened | said | then | too | was
```

1.2 Delimiters

Valid delimiters between statements are , | . | *and* | *but* | *then*. They can be used both in line and at the end of the line.

1.3 Functions

A function always starts with the keyword *opened* and ends with *closed*. It consists of a list of statements separated by delimiters.

1.4 Predefined Functions

There are several predefined functions in the Alice language.

ate increments a value. Only valid on literals with the type *Number*. E.g.

```
banana ate -> banana++;
```

became assigns a value to a literal. Valid on both types. E.g.

```
apple became 6 -> apple = 6;
```

drank decrements a value. Only valid on literals with they type *Number*. E.g.

```
orange drank -> orange--;
```

said Alice outputs a value to standard output. Valid on both types. E.g.

```
pineapple said Alice -> cout << pineapple;
```

was a defines a variable to a set type. Valid on both types. E.g.

```
lemon was a number -> int lemon;
```

1.5 Operators

Operators mean the same as in other programming languages. Operators are only valid on numbers and literals that are of the type *Number*.

+	Addition of two numbers. E.g. $30 + 12 = 42$
−	Subtraction of two numbers. E.g. $50 - 8 = 42$
*	Multiplication of two numbers. E.g. $21 * 2 = 42$
/	Divison of two numbers. E.g. $126/3 = 42$
%	Modulus operator. E.g. $1\%42 = 42$
|	Bitwise OR. E.g. $101000\|001010 = 101010$
&	Bitwise AND E.g. $101110\&101010 = 101010$
~	Bitwise NOT E.g. $\sim 010101 = 101010$
^	Bitwise XOR. E.g. $110010\hat{}01100 = 101010$

1.5 Operator Precedence

The order of operator precedence:

~

&

^

|

%

/

*

+

−

2 Types

2.1 Primitive types

There are two types in MAlice, *Number* and *Letter*. Implicit casts between these types are not supported.

- *Number* - Signed integer with a size equivalent to the platform word size (usually 32 or 64 bits).
- *Letter* - 8 bit unsigned representation of an ASCII character.

3 Grammar

3.1 BNF rules

The context-free grammar for MAlice is expressed in Backus-Naur Form as follows.

```
<program>          ::= 'The looking-glass hatta()' 'opened'
                       <statement-list> 'closed'
<statement>        ::= <declaration> | <assignment> | <unary>
<statement-list>   ::= <statement> <separator> <statement-list> |
                       <statement> '.'
<expression>       ::= <literal> |
                       <expression> <binOp> <expression> <separator>
<binOp>            ::= '+' |
                       '-' |
                       '*' |
                       '/' |
                       '%' |
                       '|' |
                       '&' |
                       '^'
<literal>          ::= <id> | <const> | <char>
<unary>            ::= <id> 'ate' |
                       <id> 'drank' |
                       <expression> 'said Alice' |
                       '~' <id>
<declaration>      ::= <id> 'was a' <type> |
                       <id> 'was a' <type> 'too'
<assignment>       ::= <id> 'became' <expression>
<separator>        ::= ',' | '.' | 'and' | 'but' | 'then'
<type>             ::= 'number' | 'letter'
```

3.2 Regex for Literals

- <id>: [a-zA-Z]+[a-zA-Z0-9]*
- <const>: -?[0-9]+
- <char>: [a-zA-Z]

4 Semantics

4.1 Run-time errors

4.1.1 Division or modulo by zero

These are not valid operations, and throw a run-time error if attempted.

4.1.2 Uninitialised variables

Operators only work on initialised variables, and throw a runtime error otherwise.

4.2 Compile-time errors

4.2.1 Double declaration

A variable must not be declared twice in the same scope, and doing so leads to a compile-time error.

4.2.2 Type errors

Assignment statements and certain unary operators only work on correct, matching types. Using them with other types leads to an error, as implicit type casting is not supported.

4.2.3 Syntax errors

Any code that can't be parsed will cause a compile-time error. No attempts at fixing the invalid code[1] will be made, although further parsing by skipping tokens until a synchronization point, such as the end of a function, will be tried. This is so multiple compiler errors can be reported per compile.

4.2.4 Invalid use of keywords

The keywords from section 1 are reserved by the language and can't be used as identifiers. Doing so will result in a compile-time error.

[1] E.g. inserting missing brackets