

MAlice Language Specification

Ethel Bardsley Joe Slade Thomas Wood

October 28, 2010

1 BNF Grammar

| | | |
|------------|---|--|
| Program | → | Statements Output |
| Statements | → | Statement Terminator Statements ϵ |
| Terminator | → | ' ,' ' .' 'and' 'but' 'then' |
| Output | → | 'Alice found' Exp '.' |
| Statement | → | <i>Id</i> 'was a' Type Too <i>Id</i> 'became' Exp <i>Id</i> 'ate' <i>Id</i> 'drank' |
| Type | → | 'number' |
| Too | → | 'too' ϵ |
| Exp | → | Exp ' ' Exp1 Exp '^' Exp1 Exp '&' Exp1 Exp1 |
| Exp1 | → | Exp1 '+' Exp2 Exp2 |
| Exp2 | → | Exp2 '*' Exp3 Exp2 '/' Exp3 Exp2 '%' Exp3 Exp3 |
| Exp3 | → | '~' Exp3 Val |
| Val | → | <i>Int</i> <i>Id</i> |

- *Int* is an integer, matching the regular expression pattern $[0-9]^+$
- *Id* is a variable identifier, matching $[a-zA-Z_]^+$

2 Semantics

2.1 Types

2.1.1 Number

Numbers are unsigned integers of length 8 bits (ie: they can hold the range 0-255). Furthermore underflow and overflow are undefined behaviours. All operators listed in the operators section can be used.

2.1.2 Letter

Although `letter` appears as a type in the given examples, there are no working examples in which its functionality is exhibited. Consequently, nothing can be inferred about this possible type, including whether it is a valid type or not!

As such, it is not included in this version of the language specification.

2.2 Statements

An Alice program is defined as a list of statements followed by the output statement.

2.2.1 Output

The `Alice found` statement is analogous to the return statement of other languages. It evaluates its parameter (an expression) and returns the value.

For example:

```
Alice found 3.  
returns the value 3.
```

2.2.2 Declaration

The `was a` statement declares the preceding identifier as a variable of the given type.

Declaring the same variable name multiple times is not permitted and will result in a compile-time error.

The keyword `too` may be placed at the end of this statement. No meaning could be inferred from the examples given, so none has been assumed at this point.

For example:

```
x was a number
```

declares a variable called x as a number

2.2.3 Assignment

The `became` statement assigns the value of an expression to the given variable.

The type of the expression must match the type of the variable, otherwise a compile-time error will result.

For example:

```
x became 5
```

assigns 5 to x .

2.2.4 Increment and Decrement

The `drank` statement decrements the given variable by 1.

The `ate` statement increments the given variable by 1.

For example:

```
x drank
```

if x is 5, x will become 4

For example:

```
x ate
```

if x is 5, x will become 6

2.3 Expressions

| Operator | Operation | Precedence |
|----------|----------------|------------|
| | Bitwise OR | 1 |
| ^ | Bitwise XOR | 1 |
| & | Bitwise AND | 1 |
| + | Addition | 2 |
| * | Multiplication | 3 |
| / | Division | 3 |
| % | Modulo | 3 |
| ~ | Bitwise NOT | 4 |

- Numerically higher precedences bind more tightly.
- All operators are mathematically associative, and implemented as left-associative.

- Division by 0 is undefined and will be handled by the operating system.
- All operators are binary, except for Bitwise NOT which is unary.
- The only precedences that were determinable from the example files given were those for the + and * operators. All other precedences have been taken from the common usage, or where none is obvious, from the C language.
- Underflow and overflow conditions are undefined.