`$Id: mini-basic-language.mm,v 1.6 2021-01-17 19:38:44-08 - - $`
`PWD: /afs/cats.ucsc.edu/courses/cse112-wm/Assignments/mini-basic.d/language-mb.d`
`URL: https://www2.ucsc.edu/courses/cse112-wm/:/Assignments/mini-basic.d/language-mb.d/`

1. Introduction

BASIC (Beginner's All-purpose Symbolic Instruction Code) was designed at Dartmouth College, NH, by John Kemeny and Thomas Kurtz in 1965. A variation of that language was 32KiB ROM BASIC, distributed by IBM on their original PC in 1981. (Pricing started at $1,565 for a configuration with 16KiB RAM.) Mini Basic is a simple subset of that language. Its syntax and semantics are described here.

2. Metasyntax

Syntax is described using Backus-Naur Form (BNF). Semantics are described informally. In the metasyntax: Brackets indicate that what is enclosed is optional. Braces are used to indicate grouping. An ellipsis indicates what precedes it may be repeated zero or more times. A bar indicates alternation. Italics indicate nonterminal symbols or token classes. Quoted Courier bold indicates literal tokens.

3. Program

(a) *Program* → { [*Label* '**:**'] [*Statement*] }...

A program consists of zero or more statements, each of which might be identified by a label. The program terminates when control flows off the last statement. A statement with neither a label nor a statement is considered just a comment and not put into the statement list. A statement ends at the end of a line, or at a comment.

(b) Comments and blank lines are ignored. A comment begins with a hash (**#**) and continues up to the end of the line.

4. Statements

Statements are the only organizational structure in the language and are executed one by one in sequence, except when a control transfer occurs. There is no block structure or nesting.

(a) *Statement* → '**dim**' *Arrayref*
Arrayref → *Identifier* '**[**' *Expression* '**]**'

The **dim** statement creates an array given by the variable name and inserts it into the array table, replacing any previous array already in the array table. The dimension of the array is given by the expression. All values in the array are initialized to 0. The expression is rounded to the nearest integer before being used as the size of the array, which must be positive. A subscript i must be in the range $0 \leq i < n$, where n is the dimension.

(b) *Statement* → '**let**' *Memref* '**=**' *Expression*
Memref → *Arrayref* | *Identifier*

A **let** statement makes an assignment to a variable. The expression is first evaluated. For a variable, the value of the expression is stored in the variable table, replacing whatever was there previously. If the variable does not exist, it is entered into the variable table. For an *Arrayref*, the value is stored in the array, indexed by the subscript. It is an error if the array does not already exist, or if the subscript is out of bounds. A subscript is rounded to the nearest integer before indexing. Array names and variable names are kept in separate symbol tables.

(c) *Statement* → '`goto`' *Label*

Control transfers to the statement referred to by the *Label*. An error occurs if the *Label* is not defined. Labels are kept in a symbol table separate from the array and variable tables.

(d) *Statement* → '`if`' *Expression Relop Expression* '`goto`' *Label*
Relop → '`=`' | '`!=`' | '`<`' | '`<=`' | '`>`' | '`>=`'

The two *Expression*s are compared according to the given *Relop*, and if the comparison is true, control transfers to the statement, as is done for the **`goto`** statement.

(e) *Statement* → '`print`' [*Printable* { '`,`' *Printable* }...]
Printable → *String* | *Expression*

Each of the operands to **`print`** is printed in sequence, with a space in front of the value of each expression. A newline is printed at the end of the list. Strings may only appear in print statements.

(f) *Statement* → '`input`' [*Memref* { '`,`' *Memref* }...]

Numeric values are read in and assigned to the input variables in sequence. Arguments might be elements of an array. For each value read into an variable, the value is inserted into the symbol table under that variable's key. For arrays, the array must already exist and the subscript not be out of bounds. If an invalid word is read (anything that is not a number), the value returned is **`nan`**. At end of file, the variable **`eof`** is entered into the symbol table with the value 1, and the variable that was just read, and all further variables, are set to **`nan`**.

5. Expressions

Expressions consistitute the computational part of the language. All values dealt with at the expression level are floating point numbers. Invalid computations, such as division by zero and infinite results do not cause computation to stop. In that case, the result is infinity (**`+inf`** or **`-inf`**), or not a number (**`nan`**).

Expression → *Expression* { '`+`' | '`-`' } *Term* | *Term*
Term → *Term* { '`*`' | '`/`' } *Factor* | *Factor*
Factor → *Primary* '`^`' *Factor* | *Primary*
Primary → '`(`' *Expression* '`)`' | *Function* '`(`' *Expression* '`)`'
Primary → { '`+`' | '`-`' } *Primary* | *Number* | *Memref*

Some implementations may produce complex number results. Implementations which do not support complex numbers will likely return a **`nan`** when mathematics would expect a complex number.

6. Lexical Syntax

*Comment*s begin with a hash (`#`) and continue to the end of the line. *String*s begin and end with double quote marks (`"`). *Number*s consist of digits, an optional decimal point, and possibly an exponent. Keywords listed above in the syntax are reserved. *Label*s and *Identifier*s use the same syntax and are distinguished by context. They consist of letters, digits, and underscores, but may not begin with a digit.

7. Builtin Symbols

Uninitialized variables are automatically entered into the variable table with a value of 0. Variables and arrays are kept in separate symbol tables, so that it is possible for an identifier to be a variable, a label, and an array name in the same program. Arrays have a lower bound of 0 and an upper bound of 1 less than their size.

There is no facility for the user to add functions. In addition to the operators that are part of the language, the following functions are supported:
`abs`, **`acos`**, **`asin`**, **`atan`**, **`ceil`**, **`cos`**, **`exp`**, **`floor`**, **`log`**, **`log10`**, **`round`**, **`sin`**, **`sqrt`**, **`tan`**, **`trunc`**.

The following are part of the initial variable table:
`nan` = `0.0 / 0.0`; **`eof`** = `0.0`; **`pi`** = **`acos`** `(-1.0)`; **`e`** = **`exp`** `(1.0)`.