COMS-W4115 Spring 2009

Programming Languages and Translators

Professor Stephen Edwards

Minimalistic BASIC Compiler (MBC)

Document Version 1.0

Language Reference Manual (LRM)

Joel Christner (jec2160)

via CVN

March 10th, 2009

Table of Contents

1. Introduction

This section serves as an introduction to the document.

1.1 Introduction to MBC

Minimalistic BASIC Compiler (MBC) provides a simple means of compiling BASIC programs, and supports many of the commonly-used commands and features of the BASIC language. MBC will output C-compatible code which the user can then compile with a C compiler and execute. MBC supports many of the more commonly-used BASIC statements as described throughout this document.

1.2 Styles Used in this Document

This document uses three primary styles to visualize concepts. Standard document text as shown in this paragraph is in the Cambria font, 12 point. Text that shows an example of a line of MBC code is shown in `Courier New, 9 point`. Text contained within an example of a line of MBC code that should be considered a container for user-specified code is shown italicized in *`Courier New, 9 point`*., and will typically be encapsulated in braces (such as *`[insert code here]`*).

We at the MBC compiler company are free-spirited individuals and reserve the right to place quirky, sarcastic, and generally off-beat humor throughout this document as we see fit.

1.3 No Humor Found Here

This document shall be free of quirky, sarcastic, and generally off-beat humor.

2. Lexical Conventions and Program Structure

This section defines the lexical conventions and general program structure used by the MBC language. MBC supports the ASCII character set only, and generally programs compiled with MCB are stored within a file.

2.1 General Program Requirements

This section describes the general program structure and content requirements.

2.1.1 Numerical Line Identifiers

Like BASIC, MBC expects that a program be comprised of a series of lines, each starting with a numerical line identifier, followed by one or more statements on that line. These numerical line identifiers must be in order from least to greatest when looking at the source program from beginning to end. All program statements must come after the numerical line identifier and there must be a space separating the identifier from the first letter of the first statement on that line. Numerical line identifiers can range from 1 to 65535. For example:

```
10 [statement 1]
20 [statement 2]
30 [statement 3]
```

2.1.2 Program Termination

Programs will terminate on one of two conditions. The first being that there simply are no further lines of code to process, and the second being an explicit 'END' statement within the program. For example:

```
10 [statement 1]
20 [statement 2]
```

And yet another example:

```
10 [statement 1]
20 [statement 2]
30 END
```

2.1.3 Whitespace

White space characters are excluded during tokenization. The only white space required is that which separates a numerical line identifier from the first statement on that line.

2.1 Comments

MBC supports comments through the use of 'REM' within the program. The use of 'REM' indicates that any content from that point to the end of the line will be treated as a comment and not compiled. There are no multi-line comments, however, multiple lines can each have a REM statement. For example:

```
10 REM This is a comment
15 REM This is another comment, on another line
20 [statement] REM This is a comment and statement 1 will be processed
```

2.2 Tokens

There are several classes of tokens that can be used with MBC. These include:

- Identifiers
- Keywords
- Numerical Constants
- Character Constants (including strings)
- Separators

2.3 Identifiers

Identifiers are used to declare and reference stored information. Identifiers are a series of letters only and may be a mix of upper and lower case. Identifier names are case sensitive, that is, the identifier named `Variable` is not the same identifier as the identifier named `VARIABLE`. Identifiers do not need to be declared; identifiers are implicitly declared when used for the first time. Two types of identifiers are available in MBC. The first type is a numerical identifier, which supports both integers and floating-point numbers. The second is a character identifier, which supports both characters as well as strings. Numerical identifiers are comprised of nothing more than characters (again, case-sensitive), and character identifiers must have a trailing $ at the end of the identifier name. Assignment to either type of identifier is done through the use of the assignment operator as denoted by the equals sign (=) and using statements structured as follows:

```
10 [identifier] = [value]
20 A$ = "Hello, World"
30 A = 50
```

Should an identifier be used prior to a value being assigned, an empty value is produced for character identifiers and a zero value is produced for numerical identifiers. For example:

```
10 PRINT A$
```

returns nothing, and

```
10 PRINT A
```

returns zero

Identifiers can not be coerced from numerical type to character type, nor can they be cast or promoted. However, numerical identifiers can be used interchangeably with character identifiers in statements where the value needs to be displayed. For example:

```
10 A$ = "50"
20 A = 30
30 PRINT A$
```

returns 50, and

```
40 PRINT A
```

returns 30

The assignment operator (=) is also used within iteration statements and program control statements and in such cases is not providing assignment but is providing comparison.

All identifiers in MBC are global; there is no concept of local identifiers in MBC.

2.4 Keywords

Keywords are those character strings that are reserved by MBC and can not be used as names of identifiers, and keywords are not case sensitive. Keywords include:

REM	PRINT	FOR	TO
STEP	IF	THEN	ELSE
GOTO	GOSUB	RETURN	PRINT
INPUT	END	LOOP	

Each of these keywords will be explained throughout the course of this document.

2.6 Numerical Constants

Numerical constants are simply numbers (both integer as well as floating-point) that are used within a program and not assigned to an identifier.

2.7 Character Constants

There are no character constants or string constants within MBC. However, character values or string values can be assigned to character identifiers, which can be used throughout the program. Without modification such identifiers provide functionality similar to that of a constant.

2.8 Separators

Each line within an MBC program will contain one or more instructions. A discrete instruction can not span one or more line, although an instruction set (for instance, a for loop) may span more than one line. The colon character should be used anywhere that multiple instructions per line is required. For example:

```
10 PRINT "Hello World" : PRINT "Yet another hello world"
```

Whitespace is ignored by MBC, although one or more spaces is required between the numerical line identifier and the first statement on that particular line.

2.9 Miscellaneous

MBC does not support special characters such as '\t', '\n', '\r\',or NULL. Print output will always automatically include a printed newline character at the end. For example:

```
10 PRINT "Hello World" : PRINT "Yet another hello world"
```

will display:

```
Hello World

Yet another hello world
```

MBC does not support pointers or structures.

3 Operations

3.1 Multiplicative Operators

The multiplicative operator, denoted by asterisk (*), provides the product of two integer constants or identifiers. This can also be used in conjunction with print statements or in assignment. The multiplicative operators use the following syntax:

```
nnn [product-identifier] = [id-or-val] * [id-or-val]
nnn [quotient-identifier] = [id-or-val] / [id-or-val]
```

For example:

```
10 A = 10
20 B = 5
30 C = A * B
```

or

```
30 C = A * B
```

C becomes 50

```
40 PRINT C
```

displays 50

```
50 PRINT "The result is: ", C
```

displays

```
The result is: 50
```

Similarly division can be performed using the forward-slash character (/). The result of this operation is the quotient. Multiplicative operators are left-associative. Multiplication holds higher precedence than division.

3.2 Additive Operators

The additive operator, denoted by the plus sign (+), provides the sum of two integer constants or identifiers. This can also be used in conjunction with print statements or in assignment. The additive operators use the following syntax:

```
nnn [sum-identifier] = [id-or-val] + [id-or-val]
nnn [difference-identifier] = [id-or-val] – [id-or-val]
```

For example:

```
10 A = 10
20 B = 5
30 C = A + B
```

or

```
10 C = 10 + 5
```

C becomes 15

```
40 PRINT C
```

displays 15

```
50 PRINT "The result is: ", C
```

displays

```
The result is: 15
```

Similarly subtraction can be performed using the minus character (-). The result of this operation is the difference between the two supplied integer constants or identifiers. Additive operators are left associative. Addition holds higher precendence than subtraction, and both are at lower precedence than division.

3.3 Relational Operators

Relational operators define the relationship of two constants or identifiers. These are employed through the use of the greater than sign (>), the less than sign (<), the equals sign (=), or a combination of these, including:

- greater than or equal to: >=
- less than or equal to: <=

Additionally, the greater than sign can be used in conjunction with the less than sign (<>) to denote inequality when used in comparing the value of two constants or identifiers. The result is a boolean value of either TRUE or FALSE, and is used to guide iteration statements and program control statements as described below. Relational operators other than equals or not equals are not applicable to character identifiers.

3.4 Logical Operators

Logical operators can be used in between relational operator statements when AND or OR logic needs to be considered in the context of an iteration statement or program control statement. The adjacent conditions must be encapsulated in parenthesis. The result is a boolean value of either TRUE or FALSE, and is used to guide iteration statements and program control statements as described below.

3.5 Assignment Operator

As mentioned above the equals sign (=) is used as an assignment operator, and places the value found on the right of the equals sign into the container named by the name on the left of the equals sign. Assignment of numerical data to a variable requires that the variable name not have a dollars-sign ($) at the end of the name, and assignment of character or string data to a variable requires that the variable name have a dollars-sign ($) at the end of the name. The following syntax is used:

```
nnn [identifier] = [value]
```

For example:

```
10 A = 50
```

sets the value of A to 50, and

```
20 A$ = "Hello"
```

sets the value of A$ to "Hello"

This operator is also used to determine equality or inequality in conditional statements that guide program flow or impact iterations. The exception to this rule is the 'FOR' statement which increments the value of an identifier by a certain amount (defined by 'STEP') upon each iteration. For example:

```
10 A = 50
```

is an assignment

```
20 IF A = 50 THEN PRINT "Hello"
```

is a comparison, and

```
30 FOR A = 1 TO 10 STEP 1
```

will increment A by 1 through each iteration of the code that exists between the statement and the NEXT statement (FOR/NEXT loops will be discussed later in the document)

3.6 Comma Operator

The comma operator, denoted by the comma (,) is used in conjunction with input and output statements to include the contents of identifiers in the input or output. The comma operator also serves to separate the identifier from the other contents of the input or output statement. For example:

```
10 A = 42

20 PRINT "The answer is ", A
```

displays

```
The answer is 42
```

and

```
30 A$ = "So long"

40 PRINT A$, ", and thanks for all the fish"
```

displays

```
So long, and thanks for all the fish
```

and

```
50 INPUT "What is your name?  ", Name$
```

displays

```
What is your name?
```

and then allows the user to enter data, which is then assigned to the identifier 'Name$'

Use of the comma operator in an INPUT statement requires that the identifier that follows the comma exist after the body of the INPUT statement.

3.7 Statement Nesting Operator

Each line must have a numerical identifier followed by one or more statements. Any time multiple statements are contained within a line, each statement after the first must have a colon (:) to the left of that statement to separate it from the previous statement. Statements that are nested on a line are handled sequentially. For example:

```
10 PRINT "Hello World": PRINT "How are you?"
```

displays

```
Hello World

How are you?
```

A colon is not required after a statement, unless there is another statement following.

3.8 Iteration Statements

Iteration statements are used to execute statements in succession as long as the conditions associated with the iteration remain true. Iteration statements come in three forms: do/while loops, while loops, and for loops.

3.8.1 Do/While Loops

The do/while loop always executes the statements contained between the DO statement and the LOOP WHILE statement once, and will continue to iterate as long as the condition provided following the while statement is true. These statements follow this format:

```
nnn DO

nnn [statements]

nnn LOOP WHILE [condition]
```

For example:

```
10 A = 1

20 DO

30 PRINT "A is equal to ", A

40 A = A + 1

50 LOOP WHILE A < 5
```

will display

```
A is equal to 1
```

```
A is equal to 2

A is equal to 3

A is equal to 4
```

And

```
10 A = 1

20 DO

30 PRINT "A is equal to ", A

40 A = A + 1

50 LOOP WHILE A < 1
```

will display

```
A is equal to 1
```

3.8.2 While/Wend Loops

A while loop is similar to a do/while loop with the exception that the condition appears before the statements, a WEND follows the statements that are associated with the loop, and the statements are only executed when the result of the condition is true. This is in contrast to a do/while loop which always executes the statements at least once. While loops have the following structure:

```
nnn WHILE [condition]

nnn [statements]

nnn WEND
```

For example:

```
10 A = 1

20 WHILE A < 4

30 PRINT "A is equal to ", A

40 A = A + 1

50 WEND
```

will display

```
A is equal to 1

A is equal to 2

A is equal to 3

A is equal to 4
```

3.8.3 For/Next Loops

A FOR/NEXT loop executes a series of statements and automatically increments a numerical counter identifier a specific amount (identified by the STEP keyword) after each iteration. Once the counter identifier reaches the maximum amount specified in the FOR statement, the statement block is executed for the final time. For/next loops have the following structure:

```
nnn FOR [identifier] = aaa TO bbb STEP ccc
nnn [statements]
nnn NEXT [identifier]
```

where:

- identifier is the name of the identifier used to hold the current value
- aaa is the beginning value of the counter identifier
- bbb is the ending value of the counter identifier, which determines when the final iteration of execution is
- ccc is the amount that the counter should be incremented or decremented after each iteration of the statement block

The STEP keyword and value are both optional. Should the STEP keyword and value not be present, MBC will assume that the increment applied to the counter identifier after each iteration is +1.

For example:

```
10 FOR A = 1 TO 4 STEP 1
20 PRINT "A is equal to ", A
30 NEXT A
```

will display

```
A is equal to 1
A is equal to 2
A is equal to 3
A is equal to 4
```

3.9 Program Control

MBC programs can be controlled and guided through the use of GOTO, GOSUB, IF/THEN/ELSE, and END statements.

3.9.1 GOTO

A GOTO statement jumps the program to a specific line as indicated by its numerical identifier, based on the numerical identifier specified following the GOTO statement. GOTO statements use the following syntax:

```
nnn GOTO [number]
```

For example:

```
10 PRINT "I'm on line 10"
15 GOTO 30
20 PRINT "You'll never see me!"
25 END
30 PRINT "Did I miss something?"
```

will display

```
I’m on line 10

Did I miss something?
```

3.9.2 GOSUB

A GOSUB statement guides a program to a specific line as indicated by its numerical identifier, based on the numerical identifier specified following the GOSUB statement. Unlike a GOTO statement, which has no means of returning to the original point from which the program execution was changed, the RETURN statement can be used in conjunction with a GOSUB statement to divert the program back to the original point of diversion. GOSUB/RETURN uses the following syntax:

```
nnn GOSUB [number-aaa]

…

aaa [statements]

nnn RETURN
```

For example:

```
10 PRINT “I’m on line 10”

15 GOSUB 50

20 PRINT “Where am I?”

25 END

50 PRINT “Welcome to Magrathia”

55 RETURN
```

will display

```
I’m on line 10

Welcome to Magrathia

Where am I?
```

3.9.3 IF/THEN/ELSE

The IF/THEN/ELSE statement will evaluate a condition and then execute either the statement following THEN if the condition is true, or the statement following ELSE if the condition is false. The IF/THEN/ELSE block must be terminated by the ‘END IF’ statement. These statements have the following syntax:

```
nnn IF [condition] THEN

nnn [statement-when-true]

nnn ELSE

nnn [statement-when-false]

nnn END IF
```

For example:

```
10 A = 50

20 IF A > 48 THEN

30   PRINT "A is greater than 48"

40 ELSE

50   PRINT "A is less than or equal to 48"

60 END IF
```

will display

```
A is greater than 48
```

3.9.4 END

The END statement will terminate the program when reached. Multiple END statements may exist within a program, and is expected in particular when there are many execution paths through a program. An END statement is recommended but not required, and a program will terminate normally when no additional code exists to execute. The syntax for the END statement is rather simple:

```
nnn END
```

3.10 Display Operations

The PRINT statement allows the user to display data on the standard output. When used by itself, PRINT will simply print a blank link. PRINT can be followed by a numerical or character identifier, a block of text encapsulated within quotes, or a combination of the two using comma separators. Syntax for the PRINT statement is:

```
nnn PRINT

nnn PRINT A$, "[some text goes here]"

nnn PRINT "[some text goes here]", A
```

Examples for the PRINT statement can be found in the preceding sections.

3.11 Input Operations

The INPUT statement allows the user to request keyboard input from the program user. INPUT can be used like a PRINT statement to first display a series of alphanumeric characters on the standard output prior to requesting input from the user. The INPUT statement has the following syntax:

```
nnn INPUT A

nnn INPUT A$

nnn INPUT "What is your name?  ", A$
```

4 Exceptions

MBC does not provide an exception-handling system. Hope and pray that your code works ;-)