TERRAN BASIC REFERENCE MANUAL

For Language Version 1.0 | First Edition

Minjae Song (CuriousTorvald)
2020-12-28

O'REALLY[?] Press

Contents

Chapter 1

Introduction

Terran BASIC is a BASIC dialect and its interpreter. Terran BASIC emulates most of the common BASIC syntax while also adding more advanced and up-to-date concepts gracefully, such as user-defined function that can *actually* recurse, arbitrary list construction using CONS-operator and some of the features in the realm of functional programming from `MAP` and `FOLD` to `CURRY`-ing a function.

This is the documentation for Terran BASIC 1.0.

Henceforward this documentation will use more friendly and sometimes vulgar language because that's more fun to write!

Part I

Language

Chapter 2

Basic Concepts

> *"Caution! Under no circumstances confuse the adjective* basic *with the noun* BASIC*, except under confusing circumstances!"*
>
> — Terran BASIC Reference Manual, First Edition*

This chapter describes the basic concepts of the Terran BASIC language.

2.1 Values and Types

BASIC is a *Dynamically Typed Language*, which means variables do not know which group they should barge in; only values of the variable do. In fact, there is no type definition in the language: *we do want our variables to feel awkward.*

There are six basic types: *number*, *boolean*, *string*, *array*, *generator* and *function*.

Number represents real (double-precision floating-point or actually, *rational*) numbers. Operations on numbers follow the same rules of the underlying virtual machine†, and such machines must follow the IEEE 754 standard‡.

Boolean is type of the value that is either `TRUE` or `FALSE`. Number `0` and `FALSE` make the condition *false*. When used in numeric context, `FALSE` will be interpreted as 0 and `TRUE` as 1.

String represents immutable§ sequences of bytes.

Array represents a collection of numbers in one or more dimensions.

Generator represents a value that automatically counts up/down whenever they have been called in FOR–NEXT loop.

Functions are, well… functions¶, especially user-defined ones. Functions are *type* because some built-in functions will actually take *functions* as arguments.

---

*Original quotation: Donald R. Woods, James M. Lyon, *The INTERCAL Programming Language Reference Manual*

†if you are not a computer person, disregard

‡ditto.

§Cannot be altered directly

¶This is not a closure; there is no way you can define a local- or anonymous variable in BASIC.

2.2 Control Flow

A program is executed starting with its lowest line number. Statements on a line are executed from left to right. When all Statements are finished execution, next lowest line number will be executed. Control flow functions can modify this normal flow.

You can dive into other lines in the middle of the program with `GOTO`. The program flow will continue normally at the new line *and it will never know ya just did that*.

If you want less insane jumping, `GOSUB` is used to jump to a subroutine. Subroutine is a little section of a code that serves as a tiny program inside of a program. `GOSUB` will remember from which statement in the line you have came from, and will return your program flow to that line when `RETURN` statement is encountered. (of course, if `RETURN` is used without `GOSUB`, program will raise some error) Do note that while you can reserve some portion of a program line as a `subroutine`, Terran BASIC will not provide local variables and whatnot as all variables in BASIC are global, and you can just `GOTO` out of a subroutine to anywhere you desire and wreak havoc *if you really wanted to*.

The `ON` statement provides alternative branching construct. You can enter multiple line numbers and let your variable (or mathematical expression) to choose which index of line numbers to `GOTO`- or `GOSUB` into.

The `IF-THEN-ELSE` lets you to conditionally select which of the two branches to execute.

The `FOR-NEXT` lets you to loop a portion of a program while automatically counting your chosen variable up or down.

Chapter 3

Language Guide

> *"Begin at the beginning", the King said gravely, "and go on till you come to the end: then stop."*
>
> — Lewis Carroll, *Alice in Wonderland*

We'll begin at the beginning; how beginning? This:

```
10 PRINT 2+2
run
4
Ok
```

Oh *boy* we just did a computation! It printed out `4` which is a correct answer for $2+2$ and it didn't crash!

3.1 GOTO here and there

`GOTO` is used a lot in BASIC, and so does in Terran BASIC. `GOTO` is a simplest method of diverging a program flow: execute only the part of the program conditionally and perform a loop.

Following program attempts to calculate a square root of the input value, showing how `GOTO` can be used in such manner.

```
10 X=1337
20 Y=0.5*X
30 Z=Y
40 Y=Y-((Y^2)-X)/(2*Y)
50 IF NOT(Z==Y) THEN GOTO 30 : REM 'NOT(Z==Y)' can be rewritten to
    'Z<>Y'
100 PRINT "Square root of ";X;" is approximately ";Y
```

Here, `GOTO` in line 50 is used to perform a loop, which keeps looping until `Z` and `Y` becomes equal. This is a newtonian method of approximating a square root.

3.2 What If We Wanted to Go Back?

But `GOTO` only jumps, you can't jump *back*. Well, not with that attitude; you *can* go back with `GOSUB` and `RETURN` statement.

This program will draw a triangle, where the actual drawing part is on line 100–160, and only get jumped into it when needed.

```
10 GOTO 1000
100 REM subroutine to draw a segment. Size is stored to 'Q'
110 PRINT SPC(20-Q);
120 Q1=1 : REM loop counter for this subroutine
130 PRINT "*";
140 Q1=Q1+1
150 IF Q1<=Q*2-1 THEN GOTO 130
160 PRINT : RETURN : REM this line will take us back from the jump
1000 Q=1 : REM this is our loop counter
1010 GOSUB 100
1020 Q=Q+1
1030 IF Q<=20 THEN GOTO 1010
```

3.3 FOR ever loop NEXT

As we've just seen, you can make loops using `GOTO`s here and there, but they *totally suck*, too much spaghetti crashes your cerebral cortex faster than *Crash Bandicoot 2*. Fortunately, there's a better way to go about that: the FOR–NEXT loop!

```
10 GOTO 1000
100 REM subroutine to draw a segment. Size is stored to 'Q'
110 PRINT SPC(20-Q);
120 FOR Q1=1 TO Q*2-1
130 PRINT "*";
140 NEXT : PRINT
150 RETURN
1000 FOR Q=1 TO 20
1010 GOSUB 100
1020 NEXT
```

When executed, this program print out *exactly the same* triangle, but code is much more straightforward thanks to the `FOR` statement.

3.4 Isn't It Nice To Have a Computer That Will Question You?

What fun is the program if it won't talk with you? You can make that happen with `INPUT` statement.

```
10 PRINT "WHAT IS YOUR NAME";
20 INPUT NAME
30 PRINT "HELLO, ";NAME
```

This short program will ask your name, and then it will greet you by the name you told to the computer.

3.5 Function

Consider the following code:

```
10 DEFUN POW2(N)=2^N
20 DEFUN DCOS(N)=COS(PI*N/180)
30 FOR X=0 TO 8
40 PRINT X,POW2(X)
50 NEXT
60 PRINT "----------------"
70 FOREACH A=0!45!90!135!180!NIL
80 PRINT A,DCOS(A)
90 NEXT
```

Here, we have defined two functions to use in the program: `POW2` and `DCOS`. Also observe that functions are defined using variable `N`s, but we use them with `X` in line 40 and with `A` in line 80: yes, functions can have their local name so you don't have to carefully choose which variable name to use in your subroutine.

Except a function can't have statements that spans two- or more BASIC lines; but there are ways to get around that, including `DO` statement and *functional currying*

This sample program also shows `FOREACH` statement, which is same as `FOR` but works with arrays.

3.6 BRB: Bad Recursion BRB: Bad Recursion BRB: Bad Recursion BRB: Bad RecursionBRB: Bad Recursion BRBRangeError: Maximum call stack size exceeded

But don't get over-excited, as it's super-trivial to create unintentional infinite loop:

```
10 DEFUN FAC(N)=N*FAC(N-1)
20 FOR K=1 TO 6
30 PRINT FAC(K)
40 NEXT
```

(if you tried this and computer becomes unresponsive, hit Ctrl-C to terminate the execution)

This failed attempt is to create a function that calculates a factorial of `N`. It didn't work because there is no *halting condition*: didn't tell computer to when to escape from the loop.

$n \times 1$ is always n, and $0!$ is 1, so it would be nice to break out of the loop when `N` reaches 0; here is the modified program:

```
10 DEFUN FAC(N)=IF N==0 THEN 1 ELSE N*FAC(N-1)
20 FOR K=1 TO 10
30 PRINT FAC(K)
40 NEXT
```

Since `IF-THEN-ELSE` can be chained to make third or more conditions — `IF-THEN-ELSE IF-THEN` or something — we can write a recursive Fibonacci function:

```
10 DEFUN FIB(N)=IF N==0 THEN 0 ELSE IF N==1 THEN 1 ELSE
    FIB(N-1)+FIB(N-2)
20 FOR K=1 TO 10
30 PRINT FIB(K);" ";
40 NEXT
```

3.7 The Functions of the High Order

Higher-order functions are functions that either takes another function as an argument, or returns a function. This sample program shows how higher-order functions can be constructed.

```
10 DEFUN APPLY(F,X)=F(X)
```

```
20 DEFUN FUN(X)=X^2
30 K=APPLY(FUN,42)
40 PRINT K
```

Here, `APPLY` takes a function `F` and value `X`, *applies* a function `F` onto the value `X` and returns the value. Since `APPLY` takes a function, it's higher-order function.

3.8 Map

`MAP` is a higher-order function that takes a function (called *transformation*) and an array to construct a new array that contains old array transformed with given *transformation*.

Or, think about the old `FAC` program before: it merely printed out the value of $1!$, $2!$ … $10!$. What if we wanted to build an array that contains such values?

```
10 DEFUN FAC(N)=IF N==0 THEN 1 ELSE N*FAC(N-1)
20 K=MAP(FAC, 1 TO 10)
30 PRINT K
```

Here, `K` holds the values of $1!$, $2!$ … $10!$. Right now we're just printing out the array, but being an array, you can make actual use of it.

3.9 Haskell Curry Wants to Know Your Location

Two pages ago there was a mentioning about something called *functional currying*. So what the fsck is currying? Consider the following code:

```
10 DEFUN F(K,T)=ABS(T)==K
20 CF=F~<32
30 PRINT CF(24) : REM will print 'false'
40 PRINT CF(-32) : REM will print 'true'
```

Here, `CF` is a curried function of `F`; built-in operator $\prec$ applies `32` to the first parameter of the function `F`, which dynamically returns a *function* of `CF(T) = ABS(T) == 32`. The fact that Curry Operator returns a *function* opens many possibilities, for example, you can create loads of sibling functions without making loads of duplicate codes.

3.10 The Grand Unification

Using all the knowledge we have learned, it should be trivial* to write a Quicksort function in Terran BASIC, like this:

```
10 DEFUN LESS(P,X)=X<P
11 DEFUN GTEQ(P,X)=X>=P
12 DEFUN QSORT(XS)=IF LEN(XS)<1 THEN NIL ELSE
    QSORT(FILTER(LESS~<HEAD(XS),TAIL(XS))) # HEAD(XS)!NIL #
    QSORT(FILTER(GTEQ~<HEAD(XS),TAIL(XS)))
100 L=7!9!4!5!2!3!1!8!6!NIL
110 PRINT L
120 PRINT QSORT(L)
```

Line 12 implements quicksort algorithm, using `LESS` and `GTEQ` as helper functions. `LESS` is a user-function version of less-than operator, and `GTEQ` is similar. `QSORT` selects a pivot by taking the head-element of the array `XS`† with `HEAD(XS)`, then utilises curried version of `LESS` and `GTEQ` to move lesser-than-pivot values to the left and greater to the right (the head element itself does not get recursed, here `TAIL(XS)` is applied to make head-less copy of the array), and these two separated *chunks* are recursively sorted using the same `QSORT` function. Currying is exploited to give comparison functions a pivot-value to compare against, and also because `FILTER` wants a *function* and not an *expression*. `HEAD(XS)!NIL` creates a single-element array contains head-element of the `XS`.

*/s

†stands for *X's*

Chapter 4

Language Reference

This chapter describes the Terran BASIC language.

4.1 Metasyntax

In the descriptions of BASIC syntax, these conventions apply.

- `VERBATIM` — Type exactly as shown
- `IDENTIFIER` — Replace *identifier* with appropriate metavariable
- `[a]` — Words within square brackets are optional
- `{a|b}` — Choose either `a` or `b`
- `[a|b]` — Optional version of the above
- `a…` — The preceding entity can be repeated

4.2 Definitions

A *Program Line* consists of a line number followed by a *Statements*. Program Lines are terminated by a line break or by the end-of-the-file.

A *Line Number* is an integer within the range of $[0..2^{53}-1]$.

A *Statement* is special form of code which has special meaning. A program line can be composed of 1 or more statements, separated by colons. For the details of statements available in Terran BASIC, see 6.

```
STATEMENT [: STATEMENT]…
```

An *Expression* is rather normal program lines, e.g. mathematical equations and function calles. The expression takes one of the following forms. For the details of functions available in Terran BASIC, see 7.

```
VARIABLE_OR_FUNCTION
```

```
( EXPRESSION )
```

```
IF EXPRESSION THEN EXPRESSION [ELSE EXPRESSION]
```

```
FUNCTION ( [EXPRESSION {,|;} [{,|;}]] )
FUNCTION [EXPRESSION {,|;} [{,|;}]]
EXPRESSION BINARY_OPERATOR EXPRESSION
UNARY_OPERATOR EXPRESSION
```

An *Array* takes following form:

```
ARRAY_NAME ( EXPRESSION [, EXPRESSION]… )
```

4.3 Literals

4.3.1 String Literals

String literals take the following form:

```
" [CHARACTERS] "
```

where `CHARACTERS` is a 1- or more repetition of ASCII-printable letters.*

To print out graphical letters outside of ASCII-printable, use string concatenation with `CHR` function, or use `EMIT` function.

4.3.2 Numeric Literals

Numeric literals take one of the following forms:

```
[+|-][0|1|2|3|4|5|6|7|8|9]… [.][0|1|2|3|4|5|6|7|8|9]…
0{x|X}[0|1|2|3|4|5|6|7|8|9]…
0{b|B}[0|1|2|3|4|5|6|7|8|9]…
```

Hexadecimal and binary literals are always interpreted as *unsigned* integers. They must range between $[0..2^{53}-1]$.

4.3.3 Variables

Variable names must start with a letter and all characters of the name must be letters `A-Z`, figures `0-9`. Variable names must not be identical to reserved words, but may *contain* one. Variable names are case-insensitive.

*In other words, `0x20..0x7E`

Unlike conventional BASIC dialects (especially GW-BASIC), name pool of variables are shared between all the types. For example, if you have a numeric variable `A`, and define an array named `A` later in the program, the new array will overwrite your numeric `A`.

Furthermore, *sigils* are not used in the Terran BASIC and attempting to use one will raise syntax-error or undefined behaviour.

4.3.4 Types

Types of data recognised by Terran BASIC are distinguished by some arcane magic of Javascript auto-casing mumbo-jumbo

Type	Range	Precision
String	As many as the machine can handle	
Integer	$\pm 2^{53} - 1$	exact within the range
Float	$\pm 4.9406564584124654 \times 10^{-324}$ – $\pm 1.7976931348623157 \times 10^{308}$	about 16 significant figures

4.4 Operators

4.4.1 Order of Precedence

The order of precedence of the operators is as shown below, lower numbers means they have higher precedence (more tightly bound)

Order	Op	Associativity	Order	Op	Associativity
1	^	Right	11	BXOR	Left
2	* / \	Left	12	BOR	Left
3	MOD	Left	13	AND	Left
4	+ −	Left	14	OR	Left
5	NOT BNOT	Left	15	TO STEP	Left
6	<< >>	Left	16	!	Right
7	< > =< <= => >=	Left	17	~	Left
8	== <> ><	Left	18	#	Left
9	MIN MAX	Left	19	~<	Left
10	BAND	Left	20	=	Right

Examples

- Exponentiation is more tightly bound than negation: `-1^2 == -(1^2) == -1` but `(-1)^2 == 1`
- Exponentiation is right-associative: `4^3^2 == 4^(3^2) == 262144`. This

behaviour is *different* from GW-BASIC in which its exponentiation is left-associative.

4.4.2 Mathematical Operators

Mathematical operators operate on expressions that returns numeric value only, except for the `+` operator which will take the action of string concatenation if either of the operand is non-numeric.

Code	Operation	Result
$x = y$	Assignment	Assigns *y* into *x*
$x \wedge y$	Exponentiation	*x* raised to the *y*th power
$x * y$	Multiplication	Product of *x* and *y*
x / y	Division	Quotient of *x* and *y*
$x \backslash y$	Truncated Division	Integer quotient of *x* and *y*
x MOD *y*	Modulo	Integer remainder of *x* and *y* with sign of *x*
$x + y$	Addition	Sum of *x* and *y*
$x - y$	Subtraction	Difference of *x* and *y*
$+ x$	Unary Plus	Value of *x*
$- x$	Unary Minus	Negative value of *x*
x MIN *y*	Minimum	Lesser value of two
x MAX *y*	Maximum	Greater value of two

Notes

- Type conversion rule follows underlying Javascript implementation. In other words, *only the god knows.*

- The expression `0^0` will return `1`, even though the expression is indeterminant.

Errors

- Any expression that results `NaN` or `Infinity` in Javascript will return some kind of errors, mainly `Division by zero`.

- If `x<0` and `y` is not integer, `x^y` will raise `Illegal function call`.

4.4.3 Comparison Operators

Comparison operator can operate on numeric and string operands. String operands will be automatically converted to numeric value if they can be; if one operand is numeric and other is non-numeric string, the former will be converted to string value.

Code	Operation	Result
$x == y$	Equal	True if x equals y
$x <> y$ $x >< y$	Not equal	False if x equals y
$x < y$	Less than	True if x is less than y
$x > y$	Greater than	True if x is greater than y
$x <= y$ $x =< y$	Less than or equal	False if x is greater than y
$x >= y$ $x => y$	Greater than or equal	False if x is less than y

When comparing strings, the ordering is as follows:

- Two strings are equal only when they are of the same length and every codepoint of the first string is identical to that of the second. This includes any whitespace or unprintable characters.
- Each character position of the string is compared starting from the leftmost character. When a pair of different characters is encountered, the string with the character of lesser codepoint is less than the string with the character of greater codepoint.
- If the strings are of different length, but equal up to the length of the shorter string, then the shorter string is less than the longer string.

4.4.4 Bitwise Operators

Bitwise operators operate on unsigned integers only. Floating points are truncated[†] to integers.

Code	Operation	Result
$x << y$	Bitwise Shift Left	Shifts entire bits of x by y
$x >> y$	Bitwise Shift Right	Shift entire bits x by y, including sign bit
BNOT x	Ones' complement	$-x-1$
x BAND y	Bitwise conjunction	Bitwise AND of x and y
x BOR y	Bitwise disjunction	Bitwise OR of x and y
x BXOR y	Bitwise add-with-no-carry	Bitwise XOR of x and y

4.4.5 Boolean Operators

Boolean operators operate on boolean values. If one of the operand is not boolean, it will be cast to appropriate boolean value. See 2.1 for casting rules.

Code	Operation	Result
NOT x	Logical negation	True if x is false and vice versa
x AND y	Bitwise conjunction	True if x and y are both true
x OR y	Bitwise disjunction	True if x or y is true, or both are true

[†]truncated towards zero

4.4.6 Generator Operators

Generator operators operate on numeric values and generators to create and modify a generator.

Code	Result
x TO *y*	Creates an generator that counts from *x* to *y*
x STEP *y*	Modifies an counting stride of the generator *x* into *y*

4.4.7 Array Operators

Array operators operate on arrays and numeric values.

Code	Operation	Result
x ! *y*	Cons	Prepends a value of *x* into an array of *y*
$x \sim y$	Push	Appends a value of *y* into an array of *x*
x # *y*	Concat	Concatenates two arrays

Arbitrary arrays can be constructed using empty-array constant `NIL`.

4.4.8 Function Operators

Function operators operate on functions and some values.

Code	Operation	Result
$f \prec\!\sim x$	Curry	Apply *x* into the first parameter of the function *f*

Currying[‡] is an operation that returns new function that has given value applied to the original function's first parameter. See 3.9 for tutorials.

4.5 Constants

Some variables are pre-defined on the language itself and cannot be modified; such variables are called *constants*.

Name	Type	Value	Description
NIL	Array	Empty Array	Used to construct arbitrary array using CONS-operator
PI	Number	3.141592653589793	π
TAU	Number	6.283185307179586	2π
EULER	Number	2.718281828459045	Euler's number e

[‡] *Partial Application* should be more appropriate for this operator, but oh well: *currying* is less mouthful.

4.6 Syntax In EBNF

If you're *that* into the language theory of computer science, texts above are just waste of bytes/inks/pixel-spaces/whatever; this little section should be more than enough!

```
(* quick reference to EBNF *)
(* { word } = word is repeated 0 or more times *)
(* [ word ] = word is optional (repeated 0 or 1 times) *)

line =
      linenumber , stmt , {":" , stmt}
    | linenumber , "REM" , ? basically anything ? ;
linenumber = digits ;

stmt =
      "REM" , ? anything ?
    | "IF" , expr_sans_asgn , "THEN" , stmt , ["ELSE" , stmt]
    | "DEFUN" , [ident] , "(" , [ident , {" , " , ident}] , ")" , "=" , expr
    | "ON" , expr_sans_asgn , ("GOTO" | "GOSUB") , expr_sans_asgn , {"," , expr_sans_asgn}
    | "(" , stmt , ")"
    | expr ; (* if the statement is 'lit' and contains only one word, treat it as function_call
                e.g. NEXT for FOR loop *)

expr = (* this basically blocks some funny attemps such as using DEFUN as anon function
          because everything is global in BASIC *)
      lit
    | "(" , expr , ")"
    | "IF" , expr_sans_asgn , "THEN" , expr , ["ELSE" , expr]
    | kywd , expr - "(" (* also deals with FOR statement *)
    (* at this point, if OP is found in paren-level 0, skip function_call *)
    | function_call
    | expr , op , expr
    | op_uni , expr ;

expr_sans_asgn = ? identical to expr except errors out whenever "=" is found ? ;

function_call =
      ident , "(" , [expr , {argsep , expr} , [argsep]] , ")"
    | ident , expr , {argsep , expr} , [argsep] ;
kywd = ? words that exists on the list of predefined function that are not operators ? ;

(* don't bother looking at these, because you already know the stuff *)

argsep = "," | ";" ;
ident = alph , [digits] ; (* variable and function names *)
lit = alph , [digits] | num | string ; (* ident + numbers and string literals *)
op = "^" | "*" | "/" | "MOD" | "+" | "-" | "<<" | ">>" | "<" | ">" | "<="
    | "=<" | ">=" | "=>" | "==" | "<>" | "><" | "BAND" | "BXOR" | "BOR"
    | "AND" | "OR" | "TO" | "STEP" | "!" | "~" | "#" | "=" ;
op_uni = "-" | "+" ;

alph = letter | letter , alph ;
digits = digit | digit , digits ;
hexdigits = hexdigit | hexdigit , hexdigits ;
bindigits = bindigit | bindigit , bindigits ;
num = digits | digits , "." , [digits] | "." , digits
    | ("0x"|"0X") , hexdigits
    | ("0b"|"0B") , bindigits ; (* sorry, no e-notation! *)
visible = ? ASCII 0x20 to 0x7E ? ;
string = '"' , (visible | visible , stringlit) , '"' ;
```

```
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" ;
hexdigit = "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f" | "0" | "1"
    | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
bindigit = "0" | "1" ;

(* all possible token states: lit num op bool qot paren sep *)
(* below are schematic of trees generated after parsing the statements *)

IF (type: function, value: IF)
1. cond
2. true
[3. false]

FOR (type: function, value: FOR)
1. expr (normally (=) but not necessarily)

DEFUN (type: function, value: DEFUN)
1. funcname
    1. arg0
    [2. arg1]
    [3. argN...]
2. stmt

ON (type: function, value: ON)
1. testvalue
2. functionname (type: lit)
3. arg0
[4. arg1]
[5. argN...]

FUNCTION_CALL (type: function, value: PRINT or something)
1. arg0
2. arg1
[3. argN...]
```

Chapter 5

Commands

This chapter describes commands accepted by the Terran BASIC editor.

5.1 The Editor

When you first launch the Terran BASIC, all you can see is some generic welcome text and two letters: `Ok`. Sure, you can just start type away your programs and type `run` to execute them, there's more things you can do with.

5.1.1 DELETE

```
DELETE LINE
```

```
DELETE LINE_START LINE_END
```

Deletes a given line. If two arguments were given, deletes any lines between them, start- and end-inclusive.

5.1.2 LOAD

```
LOAD FILENAME
```

Loads BASIC program by the file name. Default working directory for Terran BASIC is `/home/basic`.*

5.1.3 LIST

```
LIST [LINE_NUMBER]
```

```
LIST [LINE_FROM LINE_TO]
```

Displays BASIC program that currently has been typed. When no arguments were given, shows entire program; when single line number was given, displays that line; when range of line numbers were given, displays those lines.

*This is a directory within the emulated disk. On the host machine, this directory is typically `PWD/assets/diskN/home/basic`, where `PWD` is working directory for the TSVM, `diskN` is a number of the disk.

5.1.4 NEW

```
NEW
```

Immediately deletes the program that currently has been typed and resets the environment: wipes out `DATA`, variables and line labels, and resets `DATA` cursor and `OPTIONBASE` to zero

5.1.5 RENUM

```
RENUM
```

Re-numbers program line starting from 10 and incrementing by 10s. Jump targets will be re-numbered accordingly. Nonexisting jump targets will be replaced with `undefined`.†

5.1.6 RUN

```
RUN
```

Executes BASIC program that currently has been typed. Execution can be arbitrarily terminated with Ctrl-C key combination (except in `INPUT` mode).

5.1.7 SAVE

```
SAVE FILENAME
```

Saves BASIC program that currently has been typed. Existing files are overwritten *silently*.

5.1.8 SYSTEM

```
SYSTEM
```

Exits Terran BASIC.

†This behaviour is simply Javascript's null-value leaking into the BASIC. This is nonstandard behaviour and other Terran BASIC implementations may act differently.

Chapter 6

Statements

A Program line is composed of a line number and one or more statements. Multiple statements are separated by colons `:`.

6.1 IF

```
IF TRUTH_VALUE THEN TRUE_EXPRESSION [ELSE FALSE_EXPERSSION]
```

If `TRUTH_VALUE` is truthy, executes `TRUE_EXPRESSION`. If `TRUTH_VALUE` is falsy and `FALSE_EXPERSSION` is specified, executes that expression; otherwise next line or next statement will be executed.

Notes

- **`IF`** is both statement and expression. You can use IF-clause after **`ELSE`**, or within functions as well, for example.
- **`THEN`** is *not* optional, this behaviour is different from most of the BASIC dialects.
- Also unlike the most dialects, **`GOTO`** cannot be omitted; doing so will make the number be returned to its parent expression.

6.2 ON

```
ON INDEX_EXPRESSION {GOTO|GOSUB} LINE0 [, LINE1]…
```

Jumps to the line number returned by the `INDEX_EXPRESSION`. If the result is outside of range of the arguments, no jump will be performed.

Parameters

- `LINEn` can be a number, numeric expression (aka equations) or a line label.
- When `OPTIONBASE 1` is used within the program, `LINEn` starts from 1 instead of 0.

6.3 DEFUN

There it is, the DEFUN. *All those new-fangled parser*[*] *and paradigms*[†] *are tied to this very statement on Terran BASIC, and only Wally knows its secrets…*

```
DEFUN NAME ( [ARGS0 [, ARGS1]…] ) = EXPRESSION
```

With the aid of other statements[‡] and functions, DEFUN will allow you to ascend from traditional BASIC and do godly things such as *recursion*[§] and *functional programming*.

Oh, and you can define your own function, in traditional `DEF FN` sense.

Parameters

- `NAME` must be a valid variable name.
- `ARGSn` must be valid variable names, but can be a name of variables already used within the BASIC program; their value will not be affected nor be used.

[*] a computer program that translates program code entered by you into some data bits that only it can understand

[†] a guidance to in which way you must think to assimilate your brain into the computer-overlord

[‡] Actually, only the IF is useful, unless you want to *transcend* from the *dung* of mortality by using DEFUN within DEFUN (a little modification of the source code is required)

[§] see recursion

Chapter 7

Functions

Functions are a form of expression that may taks input arguments surrounded by parentheses. Most of the traditional BASIC *statements* that does not return a value are *functions* in Terran BASIC, and like those, while Terran BASIC functions can be called without parentheses, it is highly *discouraged* because of the ambiguities in syntax. **Always use parentheses on function call!**

7.1 Mathematical

7.1.1 ABS

```
Y = ABS(X)
```

Returns absolute value of `X`.

7.1.2 ACO

```
Y = ACO(X)
```

Returns inverse cosine of `X`.

7.1.3 ASN

```
Y = ASN(X)
```

Returns inverse sine of `X`.

7.1.4 ATN

```
Y = ATN(X)
```

Returns inverse tangent of `X`.

7.1.5 CBR

```
Y = CBR(X)
```

Returns cubic root of `X`.

7.1.6 CEIL

```
Y = CEIL(X)
```

Returns integer value of `X`, truncated towards positive infinity.

7.1.7 COS

```
Y = COS(X)
```

Returns cosine of `X`.

7.1.8 COSH

```
Y = COSH(X)
```

Returns hyperbolic cosine of `X`.

7.1.9 EXP

```
Y = EXP(X)
```

Returns exponential of `X`, i.e. e^X.

7.1.10 FIX

```
Y = FIX(X)
```

Returns integer value of `X`, truncated towards zero.

7.1.11 FLOOR, INT

```
Y = FLOOR(X)
```

```
Y = INT(X)
```

Returns integer value of `X`, truncated towards negative infinity.

7.1.12 LEN

```
Y = LEN(X)
```

Returns length of `X`. `X` can be either a string or an array.

7.1.13 LOG

```
Y = LOG(X)
```

Returns natural logarithm of `X`.

7.1.14 ROUND

```
Y = ROUND(X)
```

Returns closest integer value of `X`, rounding towards positive infinity.

7.1.15 RND

```
Y = RND(X)
```

Returns a random number within the range of $[0..1)$. If `X` is zero, previous random number will be returned; otherwise new random number will be returned.

7.1.16 SIN

```
Y = SIN(X)
```

Returns sine of `X`.

7.1.17 SINH

```
Y = SINH(X)
```

Returns hyperbolic sine of `X`.

7.1.18 SGN

```
Y = SGN(X)
```

Returns sign of `X`: 1 for positive, -1 for negative, 0 otherwise.

7.1.19 SQR

```
Y = SQR(X)
```

Returns square root of `X`.

7.1.20 TAN

```
Y = TAN(X)
```

Returns tangent of `X`.

7.1.21 TANH

```
Y = TANH(X)
```

Returns hyperbolic tangent of `X`.

7.2 Input

7.2.1 CIN

```
S = CIN()
```

Waits for the user input and returns it.

7.2.2 DATA

```
DATA CONST0 [, CONST1]…
```

Adds data that can be read by `READ` function. `DATA` declarations need not be reacheable in the program flow.

7.2.3 DGET

```
S = DGET()
```

Fetches a data declared from `DATA` statements and returns it, incrementing the `DATA` position.

7.2.4 DIM

```
Y = DIM(X)
```

Returns array with size of `X`, all filled with zero.

7.2.5 GETKEYSDOWN

```
K = GETKEYSDOWN()
```

Stores array that contains keycode of keys held down into the given variable.

Actual keycode and the array length depends on the machine: in TSVM, array length will be fixed to 8. For the list of available keycodes, see 9.

7.2.6 INPUT

```
INPUT VARIABLE
```

Prints out `?` to the console and waits for user input. Input can be any length and terminated with return key. The input will be stored to given variable.

This behaviour is to keep the compatibility with the traditional BASIC. For function-like usage, use `CIN` instead.

7.2.7 READ

```
READ VARIABLE
```

Assigns data declared from `DATA` statements to given variable. Reading starts at the current `DATA` position, and the data position will be incremented by one. The position is reset to the zero by the `RUN` command.

This behaviour is to keep the compatibility with the traditional BASIC. For function-like usage, use `DGET` instead.

7.3 Output

7.3.1 EMIT

```
EMIT( EXPR [{,|;} EXPR]… )
```

Prints out characters corresponding to given number on the code page being used.

`EXPR` is numeric expression.

7.3.2 PRINT

```
PRINT( EXPR [{,|;} EXPR]… )
```

Prints out given string expressions.

`EXPR` is a string, numeric expression, or array.

`PRINT` is one of the few function that differentiates two style of argument separator:

`;` will simply concatenate two expressions (unlike traditional BASIC, numbers will not have surrounding spaces), `,` tabulates the expressions.

7.4 Program Manipulation

7.4.1 CLEAR

```
CLEAR
```

Clears all declared variables.

7.4.2 END

```
END
```

Stops program execution and returns control to the user.

7.4.3 FOR

```
FOR LOOPVAR = START TO STOP [STEP STEP]
```

```
FOR LOOPVAR = GENERATOR
```

Starts a FOR–NEXT loop.

Initially, `LOOPVAR` is set to `START` then statements between the `FOR` statement and corresponding `NEXT` statements are executed and `LOOPVAR` is incremented by `STEP`, or by 1 if `STEP` is not specified. The program flow will continue to loop around until `LOOPVAR` is outside the range of `START`–`STOP`. The value of the `LOOPVAR` is equal to `STOP` + `STEP` when the looping finishes.

7.4.4 FOREACH

```
FOREACH LOOPVAR IN ARRAY
```

Same as `FOR` but fetches `LOOPVAR` from given `ARRAY`.

7.4.5 GOSUB

```
GOSUB LINENUM
```

Jumps to a subroutine at `LINENUM`. The next `RETURN` statements makes program flow to jump back to the statement after the `GOSUB`.

`LINENUM` can be either a numeric expression or a Label.

7.4.6 GOTO

```
GOTO LINENUM
```

Jumps to `LINENUM`.

`LINENUM` can be either a numeric expression or a Label.

7.4.7 LABEL

```
LABEL NAME
```

Puts a name onto the line numeber the statement is located. Jumping to the `NAME`

Notes

- `NAME` must be a valid variable name.

7.4.8 NEXT

```
NEXT
```

Iterates FOR–NEXT loop and increments the loop variable from the most recent `FOR` statement and jumps to that statement.

7.4.9 RESTORE

```
RESTORE
```

Resets the `DATA` pointer.

7.4.10 RETURN

```
RETURN
```

Returns from the `GOSUB` statement.

7.5 String Manipulation

7.5.1 CHR

```
CHAR = CHR(X)
```

Returns the character with code point of `X`. Code point is a numeric expression in the range of $[0 - 255]$.

7.5.2 LEFT

```
SUBSTR = LEFT( STR , NUM_CHARS )
```

Returns the leftmost `NUM_CHARS` characters of `STR`.

7.5.3 MID

```
SUBSTR = MID( STR , POSITION , LENGTH )
```

Returns a substring of `STR` starting at `POSITION` with specified `LENGTH`.

When `OPTIONBASE 1` is specified, the position starts from 1; otherwise it will start from 0.

7.5.4 RIGHT

```
SUBSTR = RIGHT( STR , NUM_CHARS )
```

Returns the rightmost `NUM_CHARS` characters of `STR`.

7.5.5 SPC

```
STR = SPC( STR , NUM_CHARS )
```

Returns a string of `NUM_CHARS` spaces.

7.6 Array Manipulation

7.6.1 HEAD

```
K = HEAD(X)
```

Returns the head element of the array `X`.

7.6.2 INIT

```
K = INIT(X)
```

Constructs the new array from array `X` that has its last element removed.

7.6.3 LAST

```
K = LAST(X)
```

Returns the last element of the array `X`.

7.6.4 TAIL

```
K = TAIL(X)
```

Constructs the new array from array `X` that has its head element removed.

7.7 Graphics

7.7.1 PLOT

```
PLOT( X_POS , Y_POS , COLOUR )
```

Plots a pixel to the framebuffer of the display, at XY-position of `X_POS` and `Y_POS`, with colour of `COLOUR`.

Top-left corner of the pixel will be 1 if `OPTIONBASE 1` is specified; otherwise it will be 0.

7.8 Meta

7.8.1 OPTIONBASE

```
OPTIONBASE {0|1}
```

Specifies at which number the array/string/pixel indices begin.

7.8.2 OPTIONDEBUG

```
OPTIONDEBUG {0|1}
```

Specifies whether or not the debugging messages should be printed out. The messages will be printed out to the *serial debugging console*, or to the stdout.

7.8.3 OPTIONTRACE

```
OPTIONTRACE {0|1}
```

Specifies whether or not the line numbers should be printed out. The messages will be printed out to the *serial debugging console*, or to the stdout.

7.8.4 TYPEOF

```
X = TYPEOF( VALUE )
```

Returns a type of given value.

BASIC Type	Returned Value	BASIC Type	Returned Value
Number	`num`	Array	`array`
Boolean	`bool`	Generator	`generator`
String	`str`	User Function	`usrdefun`

7.9 System

7.9.1 PEEK

```
BYTE = PEEK( MEM_ADDR )
```

Returns whatever the value stored in the `MEM_ADDR` of the Scratchpad Memory.

Address mirroring, illegal access, etc. are entirely up to the virtual machine which the BASIC interpreter is running on.

7.9.2 POKE

```
POKE( MEM_ADDR , BYTE )
```

Puts a `BYTE` into the `MEM_ADDR` of the Scratchpad Memory.

7.10 Higher-order Function

7.10.1 DO

```
DO( EXPR0 [; EXPR1]… )
```

Executes `EXPRn`s sequentially.

7.10.2 FILTER

```
NEWLIST = FILTER( FUNCTION , ITERABLE )
```

Returns an array of values from the `ITERABLE` that passes the given function. i.e. values that makes `FUNCTION(VALUE_FROM_ITERABLE)` true.

Parameters

- `FUNCTION` is a user-defined function with single parameter.
- `ITERABLE` is either an array or a generator.

7.10.3 FOLD

```
NEWVALUE = FOLD( FUNCTION , INIT_VALUE , ITERABLE )
```

Iteratively applies given function with accumulator and the value from the `ITERABLE`, returning the final accumulator. Accumulator will be set to `INIT_VALUE` before iterating over the iterable. In the first execution, the accumulator will be set to `ACC=FUNCTION(ACC,ITERABLE(0))`, and the execution will continue to remaining values within the iterable until all values are consumed. The `ITERABLE` will not be modified after the execution.

Parameters

- `FUNCTION` is a user-defined function with two parameters: first parameter being accumulator and second being a value.

7.10.4 MAP

```
NEWLIST = MAP( FUNCTION , ITERABLE )
```

Applies given function onto the every element in the iterable, and returns an array that contains such items. i.e. returns tranformation of `ITERABLE` of which the transformation is `FUNCTION`. The `ITERABLE` will not be modified after the execution.

Parameters

- `FUNCTION` is a user-defined function with single parameter.

Part II

Implementation

Chapter 8

Technical Reference

8.1 Resolving Variables

This section describes all use cases of `BasicVar`.

When a variable is `resolve`d, an object with instance of `BasicVar` is returned. A `bvType` of Javascript value is determined using `JStoBASICtype`.

Typical User Input	TYPEOF(**Q**)	Instanceof
`Q=42.195`	`num`	*primitive*
`Q=42>21`	`boolean`	*primitive*
`Q="BASIC!"`	`string`	*primitive*
`Q=DIM(12)`	`array`	Array (JS)
`Q=1 TO 9 STEP 2`	`generator`	ForGen
`DEFUN Q(X)=X+3`	`usrdefun`	BasicAST

Notes

- `TYPEOF(Q)` is identical to the variable's bvType; the function simply returns `BasicVar.bvType`.

- Do note that all resolved variables have `troType` of `Lit`, see next section for more information.

8.2 Unresolved Values

Unresolved variables has JS-object of `troType`, with *instanceof* `SyntaxTreeReturnObj`. Its properties are defined as follows:

Properties	Description
`troType`	Type of the TRO (Tree Return Object)
`troValue`	Value of the TRO
`troNextLine`	Pointer to next instruction, array of: [#line, #statement]

Following table shows which BASIC object can have which `troType`:

BASIC Type	troType
Any Variable	`lit`
Boolean	`bool`
Generator	`generator`
Array	`array`
Number	`num`
String	`string`
DEFUN'd Function	`internal_lambda`
Array Indexing	`internal_arrindexing_lazy`
Assignment	`internal_assignment_object`

Notes

- All type that is not `lit` only appear when the statement returns such values, e.g. `internal_lambda` only get returned by DEFUN statements as the statement itself returns defined function as well as assign them to given BASIC variable.
- As all variables will have `troType` of `lit` when they are not resolved, the property must not be used to determine the type of the variable; you must `resolve` it first.
- The type string `function` should not appear outside of TRO and `astType`; if you do see them in the wild, please check your JS code because you probably meant `usrdefun`.

Chapter 9

Implementation Details

This chapter explains implementation details of Terran BASIC running on TSVM.

9.1 Keycodes

This is a table of keycodes recognised by the LibGDX, a framework that TSVM runs on.

Key	Code	Key	Code	Key	Code	Key	Code
1	8	+	70	V	50	F2	245
2	9	A	29	W	51	F3	246
3	10	B	30	X	52	F4	247
4	11	C	31	Y	53	F5	248
5	12	D	32	Z	54	F6	249
6	13	E	33	LCtrl	57	F7	250
7	14	F	34	RCtrl	58	F8	251
8	15	G	35	LShift	59	F9	252
9	16	H	36	RShift	60	F10	253
0	17	I	37	LAlt	129	F11	254
↩	66	J	38	RAlt	130	Num 0	144
BkSp	67	K	39	↑	19	Num 1	145
Tab	61	L	40	↓	20	Num 2	146
`	68	M	41	←	21	Num 3	147
'	75	N	42	→	22	Num 4	148
;	43	O	43	Ins	133	Num 5	149
,	55	P	44	Del	112	Num 6	150
.	56	Q	45	PgUp	92	Num 7	151
/	76	R	46	PgDn	93	Num 8	152
[	71	S	47	Home	3	Num 9	153
]	72	T	48	End	132	NumLk	78
-	69	U	49	F1	244	*	17

Keys not listed on the table may not be available depending on the system, for example, F12 may not be recognised.

Part III

More Goodies

Chapter 10

99 Bottles of Beer

This is a sample program that prints out the infamous *99 Bottles of Beer*.

```
10 FOR I = 99 TO 1 STEP -1
20 MODE = 1
30 GOSUB 12
40 PRINT I;" bottle";BOTTLES;" of beer on the wall, ";i;"
    bottle";BOTTLES;" of beer."
50 MODE = 2
60 GOSUB 12
70 PRINT "Take one down and pass it around, ";(I-1);"
    bottle";BOTTLES;" of beer on the wall."
80 NEXT
90 PRINT "No more bottles of beer on the wall, no more bottles of
    beer."
100 PRINT "Go to the store and buy some more. 99 bottles of beer on
    the wall."
110 END
120 IF I == MODE THEN BOTTLES = "" ELSE BOTTLES = "s"
130 RETURN
```

Chapter 11

Amazing

This is a sample program that draw a randomised maze. The original program was on *BASIC Computer Games: Microcomputer Edition* and was translated into Terran BASIC.

```
1 OPTIONBASE 1
10 PRINT SPC(28);"AMAZING PROGRAM"
20 PRINT SPC(15);"CREATIVE COMPUTING MORRISTOWN, NEW JERSEY"
30 PRINT:PRINT:PRINT
100 PRINT "WHAT ARE YOUR WIDTH";:INPUT H
102 PRINT "WHAT ARE YOUR LENGTH";:INPUT V
105 IF H<>1 AND V<>1 THEN GOTO 110
106 PRINT "MEANINGLESS DIMENSIONS. TRY AGAIN.":GOTO 100
110 WS=DIM(H,V):VS=DIM(H,V)
120 PRINT:PRINT:PRINT:PRINT
160 Q=0:Z=0:X=INT(RND(1)*H+1)
165 FOR I=1 TO H
170  IF I==X THEN GOTO 173
171  PRINT ".--";
172  GOTO 175
173  PRINT ". ";
180 NEXT
190 PRINT "."
195 C=1:WS(X,1)=C:C=C+1
200 R=X:S=1:GOTO 260
210 IF R<>H THEN GOTO 240
215 IF S<>V THEN GOTO 230
220 R=1:S=1
222 GOTO 250
230 R=1:S=S+1:GOTO 250
240 R=R+1
250 IF WS(R,S)==0 THEN GOTO 210
260 IF R-1==0 THEN GOTO 530
265 IF WS(R-1,S)<>0 THEN GOTO 530
270 IF S-1==0 THEN GOTO 390
280 IF WS(R,S-1)<>0 THEN GOTO 390
290 IF R==H THEN GOTO 330
300 IF WS(R+1,S)<>0 THEN GOTO 330
310 X=INT(RND(1)*3+1)
320 ON X GOTO 790,820,860
```

```
334 IF Z==1 THEN GOTO 370
338 Q=1:GOTO 350
340 IF WS(R,S+1)<>0 THEN GOTO 370
350 X=INT(RND(1)*3+1)
360 ON X GOTO 790,820,910
370 X=INT(RND(1)*2+1)
380 ON X GOTO 790,820
390 IF R==H THEN GOTO 470
400 IF WS(R+1,S)<>0 THEN GOTO 470
405 IF S<>V THEN GOTO 420
410 IF Z==1 THEN GOTO 450
415 Q=1:GOTO 430
420 IF WS(R,S+1)<>0 THEN GOTO 450
430 X=INT(RND(1)*3+1)
440 ON X GOTO 790,860,910
450 X=INT(RND(1)*2+1)
460 ON X GOTO 790,860
470 IF S<>V THEN GOTO 490
480 IF Z==1 THEN GOTO 520
485 Q=1:GOTO 500
490 IF WS(R,S+1)<>0 THEN GOTO 520
500 X=INT(RND(1)*2+1)
510 ON X GOTO 790,910
520 GOTO 790
530 IF S-1==0 THEN GOTO 670
540 IF WS(R,S-1)<>0 THEN GOTO 670
545 IF R==H THEN GOTO 610
547 IF WS(R+1,S)<>0 THEN GOTO 610
550 IF S<>V THEN GOTO 560
552 IF Z==1 THEN GOTO 590
554 Q=1:GOTO 570
560 IF WS(R,S+1)<>0 THEN GOTO 590
570 X=INT(RND(1)*3+1)
580 ON X GOTO 820,860,910
590 X=INT(RND(1)*2+1)
600 ON X GOTO 820,860
610 IF S<>V THEN GOTO 630
620 IF Z==1 THEN GOTO 660
625 Q=1:GOTO 640
630 IF WS(R,S+1)<>0 THEN GOTO 660
640 X=INT(RND(1)*2+1)
650 ON X GOTO 820,910
660 GOTO 820
670 IF R==H THEN GOTO 740
680 IF WS(R+1,S)<>0 THEN GOTO 740
```

```
685 IF S<>V THEN GOTO 700
690 IF Z==1 THEN GOTO 730
695 Q=1:GOTO 830
700 IF WS(R,S+1)<>0 THEN GOTO 730
710 X=INT(RND(1)*2+1)
720 ON X GOTO 860,910
730 GOTO 860
740 IF S<>V THEN GOTO 760
750 IF Z==1 THEN GOTO 780
755 Q=1:GOTO 770
760 IF WS(R,S+1)<>0 THEN GOTO 780
770 GOTO 910
780 GOTO 1000
790 WS(R-1,S)=C
800 C=C+1:VS(R-1,S)=2:R=R-1
810 IF C==H*V+1 THEN GOTO 1010
815 Q=0:GOTO 260
820 WS(R,S-1)=C
830 C=C+1
840 VS(R,S-1)=1:S=S-1:IF C==H*V+1 THEN GOTO 1010
850 Q=0:GOTO 260
860 WS(R+1,S)=C
870 C=C+1:IF VS(R,S)==0 THEN GOTO 880
875 VS(R,S)=3:GOTO 890
880 VS(R,S)=2
890 R=R+1
900 IF C==H*V+1 THEN GOTO 1010
905 GOTO 530
910 IF Q==1 THEN GOTO 960
920 WS(R,S+1)=C:C=C+1:IF VS(R,S)==0 THEN GOTO 940
930 VS(R,S)=3:GOTO 950
940 VS(R,S)=1
950 S=S+1:IF C==H*V+1 THEN GOTO 1010
955 GOTO 260
960 Z=1
970 IF VS(R,S)==0 THEN GOTO 980
975 VS(R,S)=3:Q=0:GOTO 1000
980 VS(R,S)=1:Q=0:R=1:S=1:GOTO 250
1000 GOTO 210
1010 FOR J=1 TO V
1011  PRINT "|";
1012  FOR I=1 TO H
1013    IF VS(I,J)<2 THEN GOTO 1030
1020    PRINT "  ";
1021    GOTO 1040
```

```
1030    PRINT " |";
1040   NEXT
1041   PRINT
1043   FOR I=1 TO H
1045     IF VS(I,J)==0 THEN GOTO 1060
1050     IF VS(I,J)==2 THEN GOTO 1060
1051     PRINT ": ";
1052     GOTO 1070
1060     PRINT ":--";
1070   NEXT
1071   PRINT "."
1072 NEXT
1073 END
```

Chapter 12

Hamurabi

This is a sample program that is the *Hamurabi* game. The original program was on *BASIC Computer Games: Microcomputer Edition* and was translated into Terran BASIC. This game is considered as the grand ancestor of the strategy, simulation and city-building games; in fact it's so great it has got its own Wikipedia article.

```
10 PRINT SPC(32);"HAMURABI"
20 PRINT SPC(15);"CREATIVE COMPUTING MORRISTOWN, NEW JERSEY"
30 PRINT:PRINT:PRINT
80 PRINT "TRY YOUR HAND AT GOVERNING ANCIENT SUMERIA"
90 PRINT "FOR A TEN-YEAR TERM OF OFFICE.":PRINT
95 D1=0:P1=0
100 Z=0:P=95:S=2800:H=3000:E=H-S
110 Y=3:A=H/Y:I=5:Q=1
210 D=0
215 PRINT:PRINT:PRINT "HAMURABI: I BEG TO REPORT TO YOU,":Z=Z+1
217 PRINT "IN YEAR ";Z;", ";D;" PEOPLE STARVED, ";I;" CAME TO THE
    CITY,"
220 P=P+I
227 IF Q>0 THEN GOTO 230
228 P=INT(P/2)
229 PRINT "A HORRIBLE PLAGUE STRUCK! HALF THE PEOPLE DIED."
230 PRINT "POPULATION IS NOW ";P
232 PRINT "THE CITY NOW OWNS ";A;" ACRES."
235 PRINT "YOU HARVESTED ";Y;" BUSHELS PER ACRE."
250 PRINT "THE RATS ATE ";E;" BUSHELS."
260 PRINT "YOU NOW HAVE ";S;" BUSHELS IN STORE."
261 PRINT
270 IF Z==11 THEN GOTO 860
310 C=INT(10*RND(1))
311 Y=C+17
312 PRINT "LAND IS TRADING AT ";Y;" BUSHELS PER ACRE."
320 PRINT "HOW MANY ACRES DO YOU WISH TO BUY";
321 INPUT Q
322 IF Q<0 THEN GOTO 850
323 IF Y*Q<=S THEN GOTO 330
324 GOSUB 710
325 GOTO 320
330 IF Q==0 THEN GOTO 340
331 A=A+Q:S=S-Y*Q:C=0
```

```
334 GOTO 400
340 PRINT "HOW MANY ACRES DO YOU WISH TO SELL";
341 INPUT Q
342 IF Q<0 THEN GOTO 850
343 IF Q<A THEN GOTO 350
344 GOSUB 720
345 GOTO 340
350 A=A-Q:S=S+Y*Q:C=0
400 PRINT
410 PRINT "HOW MANY BUSHELS DO YOU WISH TO FEED YOUR PEOPLE";
411 INPUT Q
412 IF Q<0 THEN GOTO 850
418 REM *** TRYING TO USE MORE GRAIN THAN IS IN SILOS?
420 IF Q<=S THEN GOTO 430
421 GOSUB 710
422 GOTO 410
430 S=S-Q:C=1:PRINT
440 PRINT "HOW MANY ACRES DO YOU WISH TO PLANT WITH SEED";
441 INPUT D
442 IF D==0 THEN GOTO 511
443 IF D<0 THEN GOTO 850
444 REM *** TRYING TO PLANT MORE ACRES THAN YOU OWN?
445 IF D<=A THEN GOTO 450
446 GOSUB 720
447 GOTO 440
449 REM *** ENOUGH GRAIN FOR SEED?
450 IF INT(D/2)<=S THEN GOTO 455
452 GOSUB 710
453 GOTO 440
454 REM *** ENOUGH PEOPLE TO TEND THE CROPS?
455 IF D<10*P THEN GOTO 510
460 PRINT "BUT YOU HAVE ONLY ";P;" PEOPLE TO TEND THE FIELDS! NOW
    THEN,"
470 GOTO 440
510 S=S-INT(D/2)
511 GOSUB 800
512 REM *** A BOUNTIFUL HARVEST!
515 Y=C:H=D*Y:E=0
521 GOSUB 800
522 IF INT(C/2)<>C/2 THEN GOTO 530
523 REM *** RATS ARE RUNNING WILD!!
525 E=INT(S/C)
530 S=S-E+H
531 GOSUB 800
532 REM *** LET'S HAVE SOME BABIES
```

```
533 I=INT(C*(20*A+S)/P/100+1)
539 REM *** HOW MANY PEOPLE HAD FULL TUMMIES?
540 C=INT(Q/20)
541 REM *** HORROS, A 15% CHANCE OF PLAGUE
542 Q=INT(10*(2*RND(1)-0.3))
550 IF P<C THEN GOTO 210
551 REM *** STARVE ENOUGH FOR IMPEACHMENT?
552 D=P-C
553 IF D>0.45*P THEN GOTO 560
554 P1=((Z-1)*P1+D*100/P)/Z
555 P=C
556 D1=D1+D
557 GOTO 215
560 PRINT
561 PRINT "YOU STARVED ";D;" PEOPLE IN ONE YEAR!!!"
565 PRINT "DUE TO THIS EXTREME MISMANAGEMENT YOU HAVE NOT ONLY"
566 PRINT "BEEN IMPEACHED AND THROWN OUT OF OFFICE BUT YOU HAVE"
567 PRINT "ALSO BEEN DECLARED NATIONAL FINK!!!!"
568 GOTO 990
710 PRINT "HAMURABI: THINK AGAIN. YOU HAVE ONLY"
711 PRINT S;" BUSHELS OF GRAIN. NOW THEN,"
712 RETURN
720 PRINT "HAMURABI: THINK AGAIN. YOU OWN ONLY ";A;" ACRES. NOW THEN,"
730 RETURN
800 C=INT(RND(1)*5)+1
801 RETURN
850 PRINT
851 PRINT "HAMURABI: I CANNOT DO WHAT YOU WISH."
855 PRINT "GET YOURSELF ANOTHER STEWARD!!!!!"
857 GOTO 990
860 PRINT "IN YOUR 10-YEAR TERM OF OFFICE, ";P1;" PERCENT OF THE"
862 PRINT "POPULATION STARVED PER YEAR ON THE AVERAGE, I.E. A TOTAL
    OF"
865 PRINT D1;"PEOPLE DIED!!"
866 L=A/P
870 PRINT "YOU STARTED WITH 10 ACRES PER PERSON AND ENDED WITH"
875 PRINT L;"ACRES PER PERSON."
876 PRINT
880 IF P1>33 THEN GOTO 565
885 IF L<7 THEN GOTO 565
890 IF P1>10 THEN GOTO 940
892 IF L<9 THEN GOTO 940
895 IF P1>3 THEN GOTO 960
896 IF L<10 THEN GOTO 960
900 PRINT "A FANTASTIC PERFORMANCE!!! CHARLEMANGE, DISRAELI, AND"
```

```
905 PRINT "JEFFERSON COMBINED COULD NOT HAVE DONE BETTER!"
906 GOTO 990
940 PRINT "YOUR HEAVY-HANDED PERFORMANCE SMACKS OF NERO AND IVAN IV."
945 PRINT "THE PEOPLE (REMIANING) FIND YOU AN UNPLEASANT RULER, AND,"
950 PRINT "FRANKLY, HATE YOUR GUTS!!"
951 GOTO 990
960 PRINT "YOUR PERFORMANCE COULD HAVE BEEN SOMEWHAT BETTER, BUT"
965 PRINT "REALLY WASN'T TOO BAD AT ALL. ";INT(P*0.8*RND(1));" PEOPLE"
970 PRINT "WOULD DEARLY LIKE TO SEE YOU ASSASSINATED BUT WE ALL HAVE
    OUR"
975 PRINT "TRIVIAL PROBLEMS."
990 PRINT
991 FOR N=1 TO 10
992 PRINT EMIT(7;)
993 NEXT
995 PRINT "SO LONG FOR NOW."
996 PRINT
999 END
```

Chapter 13

Plotter

This is a plotter that draws a graph of a function.

`ZEROLINE` specifies which column of the text screen is $y = 0$, and `AMP` specifies the Y-zoom of the plotter, line 100 controls the plotting range of x.

This example program uses sinc function as an example, specified in line 10. You can re-define the function with whatever you want, then modify line 110 to call your function i.e. `PRINT PLOTLINE(YOUR_FUNCTION_HERE,I)`.

```
1 ZEROLINE=10
2 AMP=20
10 DEFUN SINC(P)=IF P==0 THEN 1.0 ELSE SIN(P)/P
20 DEFUN TOCHAR(P,X)=IF (X==ROUND(ZEROLINE+P*AMP)) THEN "@" ELSE IF
    (X==ZEROLINE) THEN "|" ELSE CHR(250)
30 DEFUN SCONCAT(ACC,S)=ACC+S
40 DEFUN PLOTLINE(F,X)=FOLD(SCONCAT,"",MAP(TOCHAR~<F(X),1 TO
    ZEROLINE+AMP))
100 FOR I=-40 TO 40
110 PRINT PLOTLINE(SINC,I)
120 NEXT
```

Chapter 14

Bibliography

- Hagemans, Rob. 2020. “PC-BASIC Documentation.” Updated 2020-09-26 19:20:45. `https://robhagemans.github.io/pcbasic/doc/2.0/`.
- Ahl, David H and North, Steve. 1978. *BASIC Computer Games*. Microcomputer Edition. New York: Workman Pub.

Disclaimers

O'REALLY? Press is entirely fictional publishing entity; **O'REALLY?** Press has no affiliation whatsoever with any of the real-world publishers.