The Friendly Persuasion Programming Language

By C. Dodgson

Preface

S.P., 25 A.U.

Friendly Persuasion has been a great gift to work with. When I first took on a position as a patent editor at Cepian Research Division, the ADA was a mysterious machine. Before MICS came along, it wasn't obvious how it would impact my career (or anyone else's outside of the narrow field of computational thought). In those days, I would walk past the converted conference room which housed the thing and shudder; thinking of poor souls punching in endless streams of hexadecimal numbers into their terminals.

But a new way was forming. A way which made possible not only faster methods of solving determination problems, but also the creation of tools to expedite all manner of processes. This, of course, included the creation and filing of patents; I traded in my electric typewriter for `mke`, which allowed for quick output of special characters. For the insanity this has since spared me, I am forever grateful.

In a page or so, Mr. Dodgson will begin his explanation of how you can write your very own Friendly Persuasion programs. Though I find many elements of computing and the authoring of computer programs to be well beyond my ability, I believe the way in which they are presented in Friendly Persuasion will be somewhat easily grasped by the reader. If not, please feel free to reach out to Mr. Dodgson with any questions. He's not too difficult to reach, and will be glad to explain. If the issue is pressing enough, it may be cause for a second edition.

That's enough discussion of my experience. You're here to make your own with the language, and the only way for that to happen is to learn. Have fun!

Your First Program

This text assumes that you have already installed and set up Friendly Persuasion on your system. However, it does not assume any prior programming experience, nor does it require substantial knowledge of the hardware on which Friendly Persuasion will be run. Before you begin writing Friendly Persuasion programs, it is imperative that you understand the concept of implied language. Friendly Persuasion commands (internally referred to as “opcodes”) are structured as sentences, though with some words removed for concise typing. With that, let’s look at your first program.

```
( An introductory program )
OUT “Hello, World!”
```

This program outputs “`Hello, World!`” to the console. The text within the parenthesis is called a comment, and is ignored by the compiler. You can use it to leave notes in your programs. Given that explanation, I’m sure the following implied language (found within the parentheses) will come as no surprise to you.

```
OUT(put) “Hello, World!” (to the console)
```

That’s all there is to outputting text in Friendly Persuasion. With this idea of implied language in mind, we will begin each chapter by going through the commands in the program and demonstrating how they might be read in a more human-comprehensible sense. I encourage you, when writing your programs, to think of the commands in this more conversational style. Also provided at the end of this manual will be a glossary; with each command, its implied language, and its function.

Variables and Math Operations

Essential to writing computer programs (though maybe not as fun) is the performing of math operations, and the ability to store the results of these operations in variables. Let's take a look at a program that demonstrates these ideas.

```
INT number 0
OUT number
INC number
OUT number
ADD number 6 number
OUT number
DEC number
OUT number
MUL number 2 number
OUT number
DIV number 2 number
OUT number
SUB number 3 number
OUT number
```

There's a lot happening there, but we'll break down each new command. Let's start with the first line. `INT number 0` means the following:

```
(make an) INT(eger variable named) number (with value) 0
```

There are multiple “types” that a variable can have, but we’ll get into that in a later program. For now, all you need to know is this variable’s type is `INT`. The next line is another `OUT` command, which you are now familiar with. The only difference here is that it’s being used to output a variable instead of the text within quotation marks. Let’s move on to the next unfamiliar line.

```
INC(rement) number (and store the result in number)
```

This simply adds one to `number` and stores the result within itself. The place in which the number is being stored is specified here because of the next line.

```
ADD number (and) 6 (and store the result in) number
```

In this command, the place the result of the operation is stored must be specified by the programmer. Given these two commands, you now understand the structure of all of the other commands, so I’ll just fill in what each implied part of the command itself is.

```
( Subtract one )
DEC(rement)
MUL(tiply)
DIV(ide)
SUB(tract)
```

Thus, the output of this program is the following:

```
0

1

7

6

12

6

3
```

We'll be building off of these in the next program, as well as introducing new ways for you to automate the execution of these commands.

Labels, Jumping, and Determination

The next program we will look at is a recreation of the Fibonacci sequence. The Fibonacci sequence, for those unfamiliar, is a sequence of numbers in which a given number is equal to the sum of the two before it. So, if we go through this sequence 9 times, it would be `0, 1, 1, 2, 3, 5, 8, 13, 21.`

```
INT i 0
INT n 9
INT p1 1
INT p2 0
INT current 0
LABEL nseries
DETERMINE i LEQ n
      DETERMINE i GREATER 2
            ADD p1 p2 current
            STORE p1 p2
            STORE current p1
            OUT current
      END
      NEXT i IS 1
            OUT p2
      END
      NEXT i IS 2
            OUT p1
      END
```

```
      INC i
      JUMP nseries
END
```

You've already seen a lot of these commands, so let's jump right into the unfamiliar.

```
(make a) LABEL (named) nseries
```

A label is simply a way to mark a place in a program. It's almost always paired with a `JUMP` command, which is what happens later on in this example. Moving on; let's look at the most important addition this program provides.

```
DETERMINE (if) i (is) LEQ (less than or equal to) n
      . . .
END (determination statement)
```

A `DETERMINE` statement is a way to ensure a section of your program only executes if a certain condition is true. In this case, the program will only execute the code within this statement while `i` is less than or equal to `n`. All `DETERMINE`s must be terminated with `END`, to indicate where the conditional part of the program ends.

```
STORE (the value in) p1 (in) p2
```

This command is a simple way to copy the value of one variable into another. Here, the value in `p1` is being stored in `p2` if `i` is greater than 2. After this `DETERMINE`'s `END` statement, there's another unfamiliar command that performs essentially the same action.

```
NEXT (determine if) i IS (equal to) 1
     . . .
END (next statement)
```

This command will only check if `i` is equal to 1 if the previous `DETERMINE` statement is false.

```
JUMP (to) nseries
```

Finally, after incrementing `i`, the program will jump back to the label `nseries`. This will lead directly into the first `DETERMINE` statement. This structure - a `LABEL` followed by a `DETERMINE` statement in which is a `JUMP` - forms the basis of all conditional loops in Friendly Persuasion. The loop itself is demonstrated in a more concise way below.

```
LABEL nseries
DETERMINE i LEQ n
     . . .
     INC i
     JUMP nseries
END
```

New Types, Arrays, and User Input

We will be looking at two programs in this section; one, a simple demonstration of an integer array and the various other variable types in Friendly Persuasion. The other is an implementation of a smaller programming language known as Albabet, which will be used to explore user input.

```
ARR INT ages 5 {19, 20, 22, 26}
INT container 0
INDEX ages 1 container
OUT container
INSERT ages 1 21
INDEX ages 1 container
OUT container
FLT x 1
DIV x 4 x
OUT x
```

Here, you see your first example of an array. An array is a way of holding data in an easily indexable form (we'll get into that, too).

```
(make an) ARR(ay) (with name) ages (of size) 5 (containing elements)
{19, 20, 22, 26}
```

So, what is indexing? This is explored in the next command. Each element in the array has what is known as an "index", a number associated with its place in the array. This counting system always starts from 0.

```
INDEX (array) ages (at index) 1 (and store result in) container
```

Indexing array `ages` at index `1` will return `20`, the second element in the array.

```
INSERT (into array) ages (at index) 1 (the value) 21
```

The INSERT command will replace any value already present in the array at the provided index with the value specified. Therefore, the output of this first section of code is as follows:

```
20
21
```

Now, after the manipulation of this array, you may notice there is a new variable created with name `x`. It's type is listed here as `FLT`, which is shorthand for a floating point number. Here, we create the float and divide it by 4, outputting the result. This outputs `0.250000`. Performing the same operation on an `INT` of the same value would output `0`, so the value of the floating point number in Friendly Persuasion is undeniable. Now, let's move on to the second program we'll be discussing in this section.

Albabet is a programming language designed by a person under the alias of "Threesodas". It's a simple language, limited to only a cell and a multiplier. It also only contains the instructions a-j which perform the following operations:

a	Increment the current value
b	Decrement the current value
c	Reset value to 0
d	Set multiplier to current value and set value to 0
e	Set multiplier to current value
f	Reset multiplier value
g	Multiply value by multiplier
h	Square the value
i	Output current value as an ASCII character
j	Add value to multiplier

Thankfully, this is not an Albabet manual. However, the simplicity of the language makes it easy to implement. Below is a Friendly Persuasion interpreter for the language, which makes use of some new features yet unseen.

```
INT i 0

INT cell 0

INT mul 0

STR code 1000 "x"

CHR c 'a'

CHR output 'a'

OUTWN ">"

IN code

LABEL loop

INDEX code i c

DETERMINE c IS '\0'
```

```
     JUMP exit
END
DETERMINE c IS 'a'
     INC cell
END
NEXT c IS 'b'
     DEC cell
END
NEXT c IS 'c'
     STORE 0 cell
END
NEXT c IS 'd'
     STORE cell mul
     STORE 0 cell
END
NEXT c IS 'e'
     STORE cell mul
END
NEXT c IS 'f'
     STORE 0 mul
END
NEXT c IS 'g'
     MUL cell mul cell
END
NEXT c IS 'h'
     MUL cell cell cell
```

```
END
NEXT c IS 'i'
      CAST cell CHR output
      OUTWN output
END
NEXT c IS 'j'
      ADD cell mul cell
END
INC i
JUMP loop
LABEL exit
```

The first new feature present in this program is the variable type `STR`, shorthand for string. A string is simply what we would think of as text. But take a look at the syntax for declaring a string variable:

```
STR(ing with name) code (of size) 1000 (with text) "x"
```

A string really behaves as an array! This means a programmer can index into a string. An individual element of a string is referred to as a character; of which there are two in this program: `c`, and `output`. Directly following the declaration of these two characters is where the user input occurs.

```
OUTWN (output without newline) ">" (to the console)
(read) IN (user input into variable) code
```

`OUTWN` ensures the cursor stays directly after the printed text, and following with the `IN` command means the program will wait for user input before proceeding. This allows the user to type an Albabet program into the space after the "`>`".

This loop, which just reads through the inputted text character-by-character, behaves slightly differently from the one seen before. Instead of comparing the incrementor value `i` to another value, it checks if the current indexed character is equal to `\0`; the special character which indicates an empty array element. If it detects this, it will jump to the label exit, which sits outside of the main loop.

```
DETERMINE c IS '\0'
     JUMP exit
END
```

The only other new command present in this program is `CAST`. `CAST` converts a variable from one type to another. Let's take a look at the syntax (recall that `cell` was originally of type `INT`).

```
CAST (variable) cell (to type) CHR (and store result in) output
```

You now have access to an Albabet interpreter. I won't be discussing how to write an introductory program in this language here, but I'm sure you can imagine that it's rather strenuous.

Concluding Thoughts

There are a number of features which were not discussed in this tutorial-style section of the manual, given they will be available and explained in a more concise way in the glossary. The glossary is grouped by function and type; and within those groups by alphabetical order. I hope that you have found this to be a good introduction to writing programs with the Friendly Persuasion programming language, and that you might write some programs of your own using the commands which were taught here.

Feel free to reach out with any questions regarding the language, or suggested revisions to the manual itself. I'm eager for a reason to write a second edition, so the issue may not have to be as pressing as you might think.

An aside: A Sir Weekend created the amazing logo on the cover of this manual. To him I show extreme gratitude.

Have fun with computers always,

C. Dodgson

Glossary

Input and Output

Command	Implied Language	Function
`IN <variable>`	`(read) IN (user input into) <variable>`	Waits for user input and reads it into the provided variable.
`OUT <element>`	`OUT(put) <element> (to the console)`	Outputs provided text or variable to the console, with a newline character.
`OUTWN <element>`	`OUTWN (output without newline) <element> (to the console)`	Outputs provided text or variable to the console without a newline character.

Variables and Casting

`CAST <variable> <type> <variable>`	`CAST (variable) <variable> (to type) <type> (and store result in) <variable>`	Converts a given variable of one type to the specified type.
`CHR <name> <value>`	`(make a) CHR (character variable named) <name> (with value) <value>`	Creates a character variable of specified name and value.
`FLT <name> <value>`	`(make a) FLT (float variable named) <name> (with value) <value>`	Creates a float variable of specified name and value.
`INT <name> <value>`	`(make an) INT(eger variable named) <name> (with value) <value>`	Creates an integer variable of specified name and value.
`STORE <value> <variable>`	`STORE (the) <value> (in) <variable>`	Stores the provided value (or value within variable) in the provided variable.
`STR <name> <size>`	`STR(ing with name)`	Creates a string variable of

`<text>`	`<name> (of size) <size> (with text) <text>`	specified name, size, and text.

Arrays

`ARR <type> <name> <size> {<elements>}`	`ARR(ay of) <type> (with) <name> (of) <size> (containing) {<elements>}`	Creates an array of specified type, name, and size with specified elements.
`INDEX <array> <index> <variable>`	`INDEX (array) <array> (at index) <index> (and store result in) <variable>`	Indexes into the specified array at specified index and stores the result in the provided variable.
`INSERT <array> <index> <value>`	`INSERT (into array) <array> (at index) <index>  (the value) <value>`	Inserts provided value into specified array at provided index.

Determination Statements

`DETERMINE <operation>`	`DETERMINE (if) <operation>`	Determines if a given operation is true or false; only if it is true, it executes the code within.
`END`	`END (determination, otherwise or next statement)`	Marks the end of a segment of code to be executed following determine, otherwise, or next.
`NEXT <operation>`	`NEXT (determine if) <operation>`	Determines if a given operation is true or false; only if the previous determine statement is false.
`OTHERWISE`	`OTHERWISE`	Executes code following the statement; only if the previous determine statement(s) is/are false.

Determination Operators

<value> GEQ <value>	<value> GEQ (greater than or equal to) <value>	Compares two values, returning true if the first is greater than or equal to the second.
<value> GREATER <value>	<value> GREATER (than) <value>	Compares two values, returning true if the first value is greater than the second.
<value> IS <value>	<value> IS (equal to) <value>	Compares two values, returning true if the first value is equal to the second.
<value> ISN'T <value>	<value> ISN'T (equal to) <value>	Compares two values, returning true if the first value is not equal to the second.
<value> LESS <value>	<value> LESS (than) <value>	Compares two values, returning true if the first value is less than the second.
<value> LEQ <value>	<value> LEQ (less than or equal to) <value>	Compares two values, returning true if the first value is less than or equal to the second.
<string> SAYS <string>		Compares two strings, returning true if the first string is equal to the second.

Logical Operators

<value> AND <value> <variable>	(evaluate) <value> AND <value> (and store result in) <variable>	Evaluates two booleans with operator and, and stores result in <variable> (true only if both are true).
<value> BAND <value> <variable>	(evaluate) <value> BAND (bitwise and) <value> (and store result in) <variable>	Evaluates two numbers with operator bitwise and, and stores result in <variable> (returns value of corresponding bits).
<value> BOR <value> <variable>	(evaluate) <value> BOR (bitwise or) <value> (and store result in) <variable>	Evaluates two numbers with operator bitwise or, and stores result in <variable> (returns value of bits which are either

		true or false).
`<value> NAND <value> <variable>`	`(evaluate) <value> NAND (not and) <value> (and store result in) <variable>`	Evaluates two booleans with operator nand, and stores result in <variable> (true only if at least one is false).
`<value> NOR <value> <variable>`	`(evaluate) <value> NOR (not or) <value> (and store result in) <variable>`	Evaluates two booleans with operator nand, and stores result in <variable> (true only if neither are true).
`NOT <value> <variable>`	`NOT (negate) <value> (and store result in) <variable>`	Evaluates the opposite of a boolean and stores result in <variable>.
`<value> OR <value> <variable>`	`(evaluate) <value> OR <value> (and store result in) <variable>`	Evaluates two booleans with operator and, and stores result in <variable> (true only if one is true).

Labels

`JUMP <label>`	`JUMP (to) <label>`	Jumps to the specified label.
`LABEL <name>`	`(create a) LABEL (with name) <name>`	Creates a label (a marker for a place in a program) of specified name.

Math Operations

`ADD <value> <value> <variable>`	`ADD <value> (to) <value> (and store the result in) <variable>`	Adds two values together, storing the result in the specified variable.
`DEC <variable>`	`DEC(rement) <variable> (and store result in variable)`	Subtracts one to the specified variable and stores the result within itself.
`DIV <value> <value> <variable>`	`DIV(ide) <value> (by) <value> (and store the result in) <variable>`	Divides two values, storing the result in the specified variable.
`INC <variable>`	`INC(rement) <variable> (and store result in variable)`	Adds one to the specified variable and stores the result within itself.

`MUL <value> <value> <variable>`	`MUL(tiply) <value> (and) <value> (and store the result in) <variable>`	Multiplies two values, storing the result in the specified variable.
`SUB <value> <value> <variable>`	`SUB(tract) <value> (and) <value> (and store the result in) <variable>`	Subtracts the second value from the first value, storing the result in the specified variable.