The Minim Programming Language

Ian Johnson

January 30, 2014

Contents

1 Introduction

1.1 What is Minim?

At its most basic, Minim is a highly stack-based programming language, similar in principle to other such languages (ex. Forth) but providing a more minimalistic approach and a more concise (as well as obfuscated) syntax. Minim does not have any notion of a function, macro, user-defined word, subroutine, or any other such way of defining a repeatable process. This makes it easier to parse and interpret, but it also means that the language is significantly more esoteric than its traditional counterparts and thus unsuitable for any ”real” work. Thus, Minim is more of a hobby language, an effort to create something unique that’s both easy to implement and (relatively) easy to grasp at a basic level.

1.2 Inspiration

Minim was inspired in various ways by a number of languages, but the initial inspiration for the language was, like many other esoteric languages, Brainfuck. The idea of Minim came out of a desire to develop a new approach to Brainfuck, similar in its compact syntax and command set, but with more programmer-friendly features such as commonly-used arithmetic operations and the ability to represent characters or numbers directly. Inspired by Forth, I opted for a stack-based approach as opposed to the traditional Turing machine style of Brainfuck. This new approach, in my opinion, makes the language seem a little ”fresher” and presents a similar challenge to that which attracted me to Brainfuck in the first place: that of wrapping my head around a new way of thinking. Other inspirations are fairly minor, such as using the same arithmetic operators as those defined in C and the choice of ‘`$`’ as the “variable” referencing operator which I’m pretty sure is in bash and Perl and a couple other languages.

2 Language Features

The following section is not designed to explain how to use Minim in a practical sense, but to describe, in detail, how it should be implemented by an interpreter or compiler.

2.1 Syntax

With the exception of literals, every command in Minim is one character long. Therefore, spaces are completely optional and only needed for making the code look nicer and more comprehensible. This makes Minim’s syntax exceptionally simple: there is no special parsing required except for literals, which are simple to process in either an interpreted or compiled environment. The stack, registers, and allowable operators are discussed in detail in the following subsections.

2.2 The Stack

The stack functions just like the stack in Forth or any other stack-based programming language, and values on the stack are restricted to Unicode (two-byte) characters. The stack itself can extend to arbitrary length, depending on the implementation, but individual values must be constrained to the two-byte range to facilitate standard output methods.

2.3 The Registers

Unlike traditional stack-based languages, Minim supports the use of 65536 different registers (one for each possible character value), each of which can hold a two-byte value in the same manner as the stack. In practice, these allow for variables to be used without allowing them to have special names (and thus keeping the syntax minimal), which is an essential concept for performing more complicated algorithms, such as reversing the stack, which are not possible with the stack alone. Registers are manipulated using the = and `$` operators, which are described in depth in the operators section (2.4).

2.4 Operators

This is a complete list of all operators defined in Minim and their specific functions.

2.4.1 ASCII (Character) Operators

Any of the characters `a-z`, `A-Z`, and `0-9`, when encountered, push that character (specifically its Unicode value) onto the stack.

2.4.2 Arithmetic Operators

The `+`, `-`, `*`, `/`, `%`, `&`, `|`, and `^` (xor) operators work in reverse Polish notation (RPN); they will take the top two values on the stack, perform the specified operation (in reverse order that they are popped off, as in standard RPN), and push the result back onto the stack. The two original values will be popped in the process. The (not) operator is unary, so it will simply pop the top value and push its one's complement negation (bitwise not).

2.4.3 Stack Control Operators

There are three operators used specifically for manipulating the stack:

1. `_` : Pops the top element off the stack without doing anything with it.
2. `#` : Duplicates the top element on the stack; in other words, pop and then push the same value twice.
3. `@` : Swaps the positions of the two items on the top of the stack.

2.4.4 I/O Operators

The I/O operators are very simple, and are actually the same as Brainfuck with one addition:

1. `.` : Outputs the Unicode character representation of the top element, popping it in the process.

2. `,` : Gets one character of input and pushes it to the stack. If more than one character is given (if there is no way to force a one character input), then take only the first character.

3. `;` : Outputs the numerical value of the top element followed by a space, popping it in the process.

2.4.5 Loop Control Operators

There are two types of loops in Minim, and their syntax is very similar to that of the loop construct in Brainfuck. The first loop, constructed using the `[` and `]` commands, will repeat the code within the loop until the value on top of the stack (checked when the `[` command is encountered) is `0` (the null character). Note that the condition is not checked at any time inside the loop, it is only evaluated at the beginning of each loop cycle with the `[` command. Similarly, there is another loop, using the { and } commands, which functions identically to the `[]` loop except that it runs until the stack is empty.

2.4.6 Register Operators

There are only two operators defined for working with registers in Minim: the `$` and = operators. The `$` operator is used to get the value of a register: it pops the top character off the stack, interpreted as a register number, and pushes the value stored in that register (without changing the contents of the register). Since there are 65536 registers, every possible character on the stack has a unique associated register that it can reference. The = operator has the opposite function: it pops two values off the stack, the top one to be used as the value to store, and the bottom one as the register number to store it in, and assigns that register's value to the specified character.

2.5 Literals

In addition to the use of standard ASCII letters and numerals to push particular values onto the stack, literals can be used for greater flexibility and ease of use in performing this otherwise tedious task for characters that do not correspond to one of these predefined values.

2.5.1 Numeric Literals

Numeric literals are enclosed in single quotes (`'`) and specify a particular numeric value (in base 10) to push to the stack. For example, an interpreter that encounters the expression `'65'` should push the character value `65` to the stack (equivalent to the `A` operator). If the number enclosed in the literal is not an allowable character value, it should be reduced modulo 65536 to a value that can be pushed to the stack.

2.5.2 String Literals

Due to Minim's stack-based nature, it would be difficult for a programmer to define a section of text in the program, even using the built-in alphanumeric character operators, because all text would have to be written backwards. Additionally, he or she would have to use numeric literals to enter characters as simple as a space or a period using numeric literals, which would prove to be a very time-consuming and frustrating process (although it can be claimed that programming in Minim itself is already that way). For this purpose, it is necessary to use string literals, enclosed in double quotes (`"`), to push textual data to the stack. When a string literal is encountered, its characters are pushed, in reverse order (right to left), to the stack. Any Unicode character (except `"`, and there are no escape codes defined for Minim string literals) can be used as a character in a string literal, making input such as `"Hello world!"` very easy to push and later output in a sensible manner.

3 Example Programs

Now that the fundamentals of Minim have been introduced, it is probably useful to walk through a couple of example programs illustrating how Minim really looks in practice.

3.1 Hello World!

The Hello World program is very easy to write in Minim, thanks to the use of string literals and loop control:

Program 1 A basic ”Hello world” program in Minim

```
"Hello world!" {.}
```

This method of printing a string works if the only thing on our stack is the string itself, but what if we have other values that we don't want to print? Taking a cue from C, we can add a null terminator to our string and use a different looping construct for a result that's just as simple:

Program 2 A different way of writing ”Hello world”

```
'0' "Hello world!" [.] _
```

The _ command at the end of our program simply makes sure that the null character is popped from the stack when the program is finished.

3.2 cat

The traditional cat program is also very easy to implement, and coincidentally is equivalent to the Brainfuck program with the same function:

Program 3 The cat program in Minim

```
,[.,]
```

This version of cat interprets the null character as the end of file, but variants could be written for any other conceivable character.

3.3 Dealing with Numbers

Now that we know how to work with strings and other character input, we should try doing some simple things with numbers. One of the simplest examples we can perform is to print out the numbers 1 through 10:

Program 4 Printing out 1 through 10 in Minim

```
'10' [#'1'-] _ {;}
```

A little explanation is necessary to fully understand what's going on here. We start with 10 on the stack, and go into a loop that repeats until the value on the top of the stack is 0. Each time the loop runs, we duplicate the number on top of the stack and subtract 1 from the duplicate, producing numbers in descending order until we get to 0, at which point the loop ends. The _ operator gets rid of the 0, and the last loop outputs the numbers from top to bottom, meaning that they get printed out in increasing order.

We can also print out the numbers in reverse order:

Program 5 Printing out 1 through 10 in reverse order

```
'10' [# '11'@- @'1'-] _ {;}
```

As with the normal version, we start with 10 on the stack and work our way from there. In this case, however, after duplicating our number, we use the command string `'11'@-` to subtract that number from 11. With our initial number as 10, this means that the first number pushed to the stack will be 11-10, or 1. The last part of the loop swaps the two items on top of the stack so that our original counter is back on top, and decrements it by 1 to continue the loop.

As a significantly more complex final example, let's write a program that will allow the user to input the digits of a base-10 number in increasing order (low to high place value) and then output that number as a whole using `;` (meaning the number has actually been stored in its entirety on the stack):

Program 6 Reading in a number in Minim

```
'0''10000'[#'10'/]_ ,0- *@ [ ,0- *+@ ] _ ;
```

We start this program by pushing 0 to the stack followed by 10000, looping in order to push the remaining powers of 10 down to 1. These numbers will be the multipliers for each digit; since the digits will be entered in increasing place value, the lower powers should be at the top of the stack. The command string `,0-` is important, because it converts the Unicode value of the digit into its numerical value, which is then multiplied by the corresponding place value. The loop repeats this process, each time adding the result of the multiplication to the previous result, which acts as the accumulator for the calculation. Once the 0 is reached, we simply pop it and output the number.

3.4 Using the registers

The registers, while often not essential for basic programs, are very useful in performing actions without interfering with existing data on the stack. For example, consider the problem of repeating a certain block of code a set number of times. If we restricted ourselves to the stack alone, the loop counter variable could get mixed up with other stack data and it would take some clever improvisation to ensure that it functions correctly. With registers, however, the process is easy. Consider the following program, which outputs the letter 'a' 10 times:

Program 7 Printing a letter 10 times

```
'10' ['1'- i@= a. i$]
```

While in this case it would have been just as simple to use another element on the stack to keep track of the counting, the approach presented in this example is significantly more general and functions as a template: the `a.` command string can be replaced with any other program segment and the loop will still function.

Another very useful application of registers is allowing for more flexible stack manipulations. While the # operator can copy the top value on the stack, what if we want to copy the top two elements? Registers make this otherwise impossible task possible by allowing us to use temporary storage to hold the values on the stack, then pushing the values off the registers in the desired order. For example, consider the following program that prints the first 12 Fibonacci numbers:

Program 8 Printing the first 12 Fibonacci numbers

```
i'12''2'-= '0''1' i$['1'- i@= b@= a@= a$b$ a$b$+ i$] _ {;}
```

Registers are used in this program both to store the loop counter and to allow the previous two Fibonacci numbers to be copied so their sum can be computed. Without registers, it would be significantly harder, or even impossible, to write such a program, and certainly less general: the 12 at the beginning of the program can be replaced with any desired number of terms greater than or equal to 2.

In the above example, we were able to push the first 12 Fibonacci numbers onto the stack and output them, but due to the nature of the stack they were printed in reverse order. In this particular case, some clever tricks could be employed using the properties of the Fibonacci sequence to reverse this order, but in many cases this will not be possible without some other way of reversing the elements on the stack. As a final example of the power of the register system, and of a longer, more complicated Minim program than those encountered before, consider the following program segment to reverse the order of all items on the stack:

Program 9 Reversing the order of the stack

```
'0''2'= {'0'$@= '0'$'1'+ '0'@=} '0'$'1'- '1'@=
'0''2'= '1'$['1'- '1'@= '0'$$ '0'$'1'+ '0'@= '1'$] _
```

The program segment is split into two parts: the first part stores every item on the stack in consecutive registers, and the second part loops back through those registers and pushes them back onto the stack in the same order they were processed, resulting in the order being reversed. This method is not without its limitations: the stack cannot have more than 65534 elements because of the limited number of registers, and register contents will be lost after the operation has completed, but in general this represents a very good use of registers to do something that would normally be impossible. Note the critical application of the `$` operator in the second line: by applying it once to register `'0'`, we obtain the counter, the current register being accessed, and by applying it a second time we are actually able to push the value stored in that register.