SNUSP 1.0 Language Specification
Working Draft 1

Daniel Brockman

September 10, 2003

Contents

1 Introduction

The SNUSP language was created in September, 2003 to develop a complete and utter fucking waste of time. (The name SNUSP is a recursive acronym for "SNUSP's Not Unix, but Structured Path.") We are currently evaluating the possibilities of developing a SNUSP operating system kernel. However, variants of the SNUSP system, which use the Linux kernel, are already in use; though these systems are often referred to as "Linux," they are more accurately called "SNUSP/Linux systems."

Issue: *This is not even funny. How do you write an introduction to something like this?*

1.1 History

One rainy night in August, 2003, Francis Rogers was sitting in his apartment in [where his lives] experimenting with C. Inspired by the remarkably beautiful and symmetrical eight-instruction classic BRAINFUCK, as well as the crazy multi-dimensional stack-shuffling language BEFUNGE, he was writing an interpreter for a language he would later come to call PATH. Borrowing the basic instructions and linear memory model from BRAINFUCK, and the two-dimensional code space from BEFUNGE, he created a language both simple to understand and simple to use. Once Rogers realized that he had created something interesting—that is, once he got the interpreter to run a spectacular bell-emmitting program—he immediately posted the source code and a quick rundown on the language to the Something Awful Forums[1] for peer review.

PATH was highly appreciated as a respectable middle-ground by everyone who adored BRAINFUCK but was scared by BEFUNGE (or vice versa), and a few who seemed new to programming but decided to pick up PATH because it looked so cute. Not very surprisingly, everyone else thought the language looked horribly obfuscated, and immediately started to question the sanity of everyone involved with its development. Nevertheless, several PATH tools created by enthusiasts popped up over the course of a week: interpreters written in C, C++, PERL, and JAVA; debuggers for TK, WINDOWS, and SWING; and a simple web-based interpreter interface written in PHP.

The original PATH was not perfect, however, and as more suggestions for improving PATH were made and implemented by the interpreter writers, the language borders started to blur. Every PATH coder had his own flavour—SNUSP was one of the more well-defined ones—but noone could say what PATH really was anymore. To sort out this mess, Rogers announced that he wished to keep the name "PATH" for his original version of the language, and asked everybody who wanted changes to fork off under a new name (no pun intended). This opened the door for the SNUSP project to begin serious work on defining a completely independent new language derived from traditional PATH.

1.2 Goals

The SNUSP language, with its roots in PATH, is intended to be an æsthetically pleasing, modular language with an orthogonal instruction set and a bright future. This specification defines three increasingly sophisticated levels of the SNUSP language:

Core SNUSP is—like traditional PATH—essentially a modification of BRAINFUCK to use a two-dimensional code space;

Modular SNUSP is an extension of CORE SNUSP, adding a subroutine mechanism; finally,

Bloated SNUSP is an extension of MODULAR SNUSP, adding support for indeterminism, concurrency, and a second data memory dimension.

[1] `<http://forums.somethingawful.com/>`

The first and second levels are theoretically complete; it it unlikely that future versions of this specification will alter them. The third level, on the other hand, is specifically designated for new features—particularly ones that add bloat.

Plans exist on developing a standard library in MODULAR SNUSP, with the goal of increasing the viability of SNUSP as a development platform for mission-critical applications. It will factor out certain basic building blocks and provide subroutines for mathematical functions, string manipulation, etc.

2 Memory

There are three kinds of run-time memory in SNUSP:

code space contains run-time representations of program source;

data memory (or simply "memory") contains integers that are accessed and modified by SNUSP programs when carrying out their task; finally,

the call stack (used in MODULAR SNUSP) is, in familiar terms, a FILO queue storing the return addresses of subroutine calls, i.e., **enter** instructions.

2.1 Memory Units

Code space and data memory are both two-dimensional and made up of units called, respectively, *code cells* and *data cells*. The call stack is one-dimensional and made up of *stack frames*.

Note: The second data memory dimension can be exploited only by programs written in BLOATED SNUSP. In lower levels of SNUSP, data memory is effectively one-dimensional, since the data pointer can only move in two opposite directions—**left** and **right**.

Note: The term "stack frame" normally refers to both the return address and the local data of a subroutine. However, in SNUSP there is no such thing as "local data," and return addresses are completely separated from data memory. As a practical convention, most subroutines guarantee the invariance of previous memory; but since the language does not actually define subroutines, there is nothing to enforce this.

2.2 Accessibility

Unlike in BEFUNGE, code space is completely inaccessible for inspection or change by SNUSP programs; it is only used internally by the interpreter. Thus, once the interpreter has loaded a program, code space does not change until another program is loaded.

Data memory, on the other hand, is completely accessible to SNUSP programs as mutable working storage—just like in BRAINFUCK.

The call stack is accessible to SNUSP programs as a side-effect of the **enter** and **leave** instructions. However, it cannot be randomly accessed.

2.3 Limitations

The following limitations apply to the three memory sections:

- Code space is bounded in all directions, and it is impossible for the instruction pointer to point outside it.
- Data memory can grow as large as physical memory restrictions allow it to. However, it is bounded in both dimensions: If at any point the number of times the data pointer has been moved to the left exceeds the number of times it has been moved to the right, the resulting behavior is undefined. The equivalent is true for the orthogonal dimension: The number of moves upwards must not exceed the number of moves downwards.

  Rationale: This does not practically impose a limit on normal SNUSP programs, but simplifies the implementation of interpreters.

  Issue: *This is the most obvious irregulatity that I know about in the SNUSP language. Should we define what happens if the data pointer falls off? We have three choices:*

  - *Leave it undefined. This leaves a hole in the language, but maybe this is the way it should be.*
  - *Define the behavior. Terminating the process seems to be the only reasonable choice here, but it is not elegant.*
  - *Remove the boundaries altogether, eliminating the issue. This seems to be the most elegant solution. Can you live with this, interpreter writers?*

- The call stack is unbounded and can grow as high as physical memory limitations allow it to.

3 Syntax

SNUSP source files are read and transplanted into code space one line at a time. A conforming SNUSP interpreter is required to recognize all of the following character sequences as end-of-line indicators:

- carriage return (13), line feed (10)
- carriage return (13)
- line feed (10)

Further, when loading a source file, conforming interpreters must behave as if all lines were padded to the right with spaces (32), so as to make all lines equally long.

3.1 Instruction Characters

When each line is read into code memory from the source file, the source characters are translated to instructions according to the following table:

ASCII	Glyph	Instruction
	BLOATED SNUSP	
37	%	**rand**
38	&	**split**
59	;	**down**
58	:	**up**
	MODULAR SNUSP	
64	@	**enter**
35	#	**leave**
	CORE SNUSP	
62	>	**right**
60	<	**left**
43	+	**incr**
45	-	**decr**
44	,	**read**
46	.	**write**
47	/	**ruld**
92	\	**lurd**
33	!	**skip**
63	?	**skipz**
32		**noop**
61	=	**noop**
124	|	**noop**

All other characters translate to **noop** instructions.

3.2 The Starting Indicator

The *starting indicator* tells the interpreter where to begin execution. If the source file contains any dollar signs (36), the first one to appear is the starting indicator; otherwise, the first character—whatever it may be—is the starting indicator.

4 Execution

A SNUSP program may be executed indirectly through an interpreter, or directly as a stand-alone process with a built-in interpreter. In any case, when a SNUSP program is invoked, there is no way to pass arguments to it; the only way to give it input it is through the standard input stream. The program, however, can give output—apart from through the standard output stream—via the process exit code.

4.1 Variables

During execution three variables are used to keep track of the program state, apart from the various kinds of memory:

the instruction pointer that points to an instruction in code space called the *current instruction*,

the data pointer that points to a cell in data memory called the *current data cell*, and

the current direction that indicates direction in which the instruction pointer is moving.

Todo: *Maybe add a section about threads here.*

4.2 Ticks and Turns

At the start of execution, a thread is created, its instruction pointer is set to point to the cell that contains the starting indicator, and its current direction is set to **right**. Its call stack starts out empty and the data memory originally contains nothing but zeroes.

Execution of a SNUSP program is then carried out in small steps called *ticks*. Each thread gets one *turn* per tick, but the order in which the turns are taken is undefined. The thread that is currently taking its turn is called the *active thread*. A turn proceeds as follows:

1. The current instruction is carried out.
2. The instruction pointer is moved one step in the current direction unless this would cause the instruction pointer to point outside code space, in which case the active thread is *stopped*.

When a thread is stopped, all its resources are released and it ceases taking turns. When all threads are stopped, the process terminates with the exit code set to the value of the current memory cell of the last thread to take a turn.

5 Instructions

All instructions in SNUSP are atomic, in the sense that there are no real syntactic or semantic restrictions on how they are to be combined. Some instructions access and/or mutate the current memory cell, but no other parts of data memory are ever touched.

The **noop** instruction is special, as it actually denotes *lack* of any instruction at all:

noop (, |, =) Do nothing.

5.1 Core SNUSP

The first six instructions in this set—**left**, **right**, **incr**, **decr**, **read**, and **write**—are identical to their BRAINFUCK counterparts. The remaining four—**ruld**, **lurd**, **skip**, and **skipz**—replace the pair of looping instructions found in BRAINFUCK—[and]—as general-purpose flow control instructions that can be combined to create loops and similar code structures.

left (>) Move the data pointer one cell to the left.

right (<) Move the data pointer one cell to the right.

incr (+) If the value of the current data cell is less than the maximum allowed value, increment it; otherwise, set it to zero.

decr (-) If the value of the current data cell is greater than zero, decrement it; otherwise, set it to the maximum allowed value.

read (,) Read a byte from standard input and put it in the current data cell. If the input stream is exhausted, block until more data becomes available.

write (.) If the value of the current data cell is representable by a single byte, write this byte to standard output. Otherwise, the behavior is implementation-defined.

> ***Issue:*** *Ruling run-time errors out, there are a number of different methods for squeezing a 32-bit value into a byte:*
>
> - *doing it modulo the maximum value,*
> - *outputting zero, and*
> - *outputting the maximum value.*
>
> *Should we choose one of these?*

ruld (\) If the current direction is

- **left**, change it to **up**
- **right**, change it to **down**,

and **u** [illegible]

The following example demonstrates how to implement a subroutine called `ECHO`, using the **enter** and **leave** instructions, and how to call it twice from the main program execution path:

```
       /==!/======ECHO==,==.==#
       |    |
$==>==@/==@/==<==#
```

5.3 Bloated SNUSP

This level adds four new instructions, for a grand total of sixteen SNUSP instructions. The first two simply add ways of moving through the second data memory dimension; this is particularly useful in the context of concurrency, which is provided by another instruction for starting new threads. The last instruction provides a way to obtain random numbers in arbitrary ranges.

up (`:`) Move the data pointer one cell upwards.

down (`;`) Move the data pointer one cell downwards.

split (`&`) Create a new thread, and move the instruction pointer of the old thread one step in the current direction.

rand (`%`) Set the value of the current data cell to a random number between zero and the current value of the cell, inclusive.

All threads share a single code space and a single data memory; however, each thread has its own instruction pointer, direction, memory pointer, and call stack. Upon thread creation, the instruction pointer, direction, and memory pointer is copied from the creating thread; the call stack, on the other hand, is created empty.

Todo: *Some or all of the above should be moved.*

The following example demonstrates how to print "`!`" until a key is pressed, using two concurrent threads:

```
                    /==.==<==\
                    |        |
     /++++++++++==&\==>===?!/==<<==#
     \++++++++++\  |
$==>==++++++++++/  \==>==,==#
```