HAMPSHIRE COLLEGE
School of Cognitive Science

Cognitive Science Technical Report Series

Push 3.0 Programming Language Descripton

Lee Spector, Chris Perry, Jon Klein, and Maarten Keijzer

Amherst, Massachusetts 01002
cogsci.hampshire.edu

TECHNICAL REPORT NO. HC-CSTR-2004-02
DATE 09.10.04

Push 3.0 Programming Language Description

Lee Spector, Chris Perry, Jon Klein, and Maarten Keijzer
School of Cognitive Science
Hampshire College

November, 2003 - September, 2004 (See the document version history at the end of this document. This document is a successor to the Push 2.0 Programming Language Description, from which it borrows large chunks of text.)

Contents

Overview

Push is a programming language intended primarily for use in evolutionary computation systems (such as genetic programming systems), as the language in which evolving programs are expressed. Push has an unusually simple syntax, which facilitates the development (or evolution) of mutation and recombination operators that generate and manipulate programs. Despite this simple syntax, Push provides more expressive power than most other program representations that are used for program evolution. Push programs can process multiple data types (without the syntax restrictions that usually accompany this capability), and they can express and make use of arbitrary control structures (e.g. recursive subroutines and macros) through the explicit manipulation of their own code (via a "CODE" data type). This allows Push to support the automatic evolution of modular program architectures in a particularly simple way, even when it Push is employed in an otherwise ordinary genetic programming system (such as PushGP, which is a "generic" GP system except that it evolves Push programs rather than Lisp-style program trees). Push can also support entirely new evolutionary computation paradigms such as "autoconstructive evolution," in which genetic operators and other components of the evolutionary system themselves evolve (as in the Pushpop and SwarmEvolve2 systems).

This document describes version 3.0 of the Push programming language (a.k.a "Push3"), which shares general features with the first and second versions of Push (a.k.a "Push1" and "Push2") although several details have changed with each new version. Although it is based on Push1, a good introduction to the basic principles of Push and its use for evolutionary computation is:

Spector, L., and A. Robinson. 2002. Genetic Programming and Autoconstructive Evolution with

the Push Programming Language. In *Genetic Programming and Evolvable Machines*, Vol. 3, No. 1, pp. 7-40. (http://hampshire.edu/lspector/pubs/push-gpem-final.pdf, pdf 216KB)

A more recent discussion, based on Push2, is in:

Spector, L. 2004. *Automatic Quantum Computer Programming: A Genetic Programming Approach*. Boston, MA: Kluwer Academic Publishers. (Note that the chapters on Push can be read independently of the chapters on quantum computing. More information on the book is available from http://hampshire.edu/lspector/aqcp/.)

The present document is a self-contained specification for Push3 that also briefly describes differences between Push1, Push2, and Push3. That is, one should be able to use Push3 or even to re-implement Push3 using this document alone. But it does not address the motivations behind the design -- it does not discuss *why one might want to* use Push in the first place. For such discussions please consult the references cited above, and/or the additional publications listed at http://hampshire.edu/lspector/push.html.

Freely available Push implementations are listed at http://hampshire.edu/lspector/push.html. A mailing list for Push-related discussions can be accessed via http://lists.hampshire.edu/mailman/listinfo/push.

Push Concepts

Push achieves its combination of syntactic simplicity and semantic power through the use of a stack-based execution architecture that includes a stack for each data type. A CODE data type, with its own stack and an associated set of code-manipulation instructions, provides many of the more interesting features of the language. Push instructions, like instructions in all stack-based languages, take any arguments that they require and leave any results that they produce on data stacks. To provide for "stack safe" execution of arbitrary code Push adopts the convention, used widely in stack-based genetic programming, that instructions requiring arguments that are not available (because the relevant stacks are empty) become NOOPs; that is, they do nothing. Because Push's stacks are typed, instructions will always receive arguments and produce results of the appropriate types (if they do anything at all), regardless of the contexts in which they occur.

The syntax of Push is simply this:

program ::= instruction | literal | (program*)

In other words:

- an instruction is a Push program
- a literal is a Push program
- a parenthesized sequence of zero or more Push programs is a Push program

Some implementations may require spaces around parentheses. Parenthesized sequences are also referred to as "lists," and Push programs can in fact be treated as list data structures. Literals are constants such as "3" (an integer constant) and "3.14" (a floating point number constant) and "TRUE" (a Boolean constant). Instruction names are not case sensitive and they can include the "." character. Instruction names generally start with the name of the type that they primarily manipulate, followed by a "."; for example, INTEGER.+ is the instruction for adding two integers, and BOOLEAN.DUP is the instruction for duplicating the value

on the top of the Boolean stack. In some cases, when an instruction interacts with data of multiple types, it is not obvious to which type the instruction "belongs"; in these cases the instruction prefix is usually taken from the type of the primary result of the instruction (if any).

Execution of a Push program involves the recursive application of the following procedure:

```
To execute program P:
    If P is a single instruction then execute it.
    Else if P is a literal then push it onto the appropriate stack.
    Else (P must be a list) sequentially execute each of the
        Push programs in P.
```

The recursive executions implied in the final line of this procedure can be implemented in at least two ways. The simplest technique, which was used in versions of Push prior to Push3, is to rely on the support for recursion in the language within which Push is implemented; that is, one simply writes the procedure as specified, with a sequence of recursive calls on the final line. One of the primary innovations in Push3 is to manage the recursion in Push itself, using a new EXEC (execution) stack (this idea was developed by Maarten Keijzer). In Push3 the procedure above is recast as follows:

```
To execute program P:
    Push P onto the EXEC stack
    LOOP until the EXEC stack is empty:
        If the first item on the EXEC stack is a single instruction
            then pop it and execute it.
        Else if the first item on the EXEC stack is a literal
            then pop it and push it onto the appropriate stack.
        Else (the first item must be a list) pop it and push all of the
            items that it contains back onto the EXEC stack individually,
            in reverse order (so that the item that was first in the list
            ends up on top).
```

In a sense this is merely an alternative implementation, with no semantic significance; it produces the same results as the simpler recursive implementation UNLESS we do new things to take advantage of the EXEC stack. One new capability provided by the EXEC stack results from the fact that the complete state of the Push interpreter now resides in the stack contents and the values of a few global variables (which store interpreter parameter and name bindings -- see below). This means that it becomes trivial to make a Push interpreter "reentrant" in the sense that it can be suspended and re-started at any time during a computation. This can be useful, for example, when a Push interpreter is embedded in an environment in which other processes (perhaps including other Push interpreters) must also be given CPU time. Such a capability could also be provided in other ways -- for example by dedicating a separate operating system process to each Push interpreter -- but the EXEC stack mechanism makes this easier and more efficient.

Other new capabilities provided by the EXEC stack result from explicit manipulation of the code that it contains. This allows for particularly efficient implementation of certain control structures including combinators. A simple example of this new functionality is given in the Simple Examples section below.

A top-level call to the Push interpreter may be provided with a second program, called the "configuration code," that can be used to preload values onto stacks and to set interpreter parameter values. In addition, the main program passed to the top-level call will itself normally be pushed onto the CODE stack before execution; this convention simplifies the expression of some recursive programs (see below for an example). If this behavior is not desired then one can turn it off by setting the TOP-LEVEL-PUSH-CODE parameter to FALSE.

The standard NAME data type provides for symbolic names that can be bound to values (thereby acting as

variables and defined instructions) using DEFINE instructions. Any identifiers that do not represent known Push instructions or literals of other type (e.g. TRUE and FALSE) are recognized as NAMEs. If a name has not previously been given a value then it is pushed onto the NAME stack when it is encountered. A subsequent call to a DEFINE instruction (such as INTEGER.DEFINE or CODE.DEFINE) will bind the name on top of the NAME stack to the top value of the designated stack. From that point forward the name will act as an instruction which, when executed, will push the bound value onto the EXEC stack. For most types this will result in the value ending up, after the next execution cycle, back on the stack from which it originally came (since the execution of a literal just pushes it onto the appropriate stack). If the bound value is code, however, this will result in the bound value being EXECUTED -- so the name will act as a newly defined instruction.

A NAME.QUOTE instruction is provided to move names that already have definitions back onto the NAME stack, presumably for the sake of re-definition. When NAME.QUOTE is executed a flag is set that causes the next encountered name to be pushed onto the NAME stack, whether or not it has previously been bound (and the flag is then cleared, whether or not the name was previously bound). The CODE.QUOTE instruction is similar in some respects; it causes the next encountered piece of code, whatever it is, to be pushed onto the CODE stack rather than being executed. This is useful for moving code onto the CODE stack for manipulation and/or execution by other instructions. CODE.QUOTE can be implemented with a flag, similarly to NAME.QUOTE, but the existence of the EXEC stack permits a simpler implementation: simply move the top item of the EXEC stack to the CODE stack. Push includes a full suite of list-manipulation instructions that can be used to modify code in arbitrary ways, along with execution instructions (such as CODE.DO and CODE.DO*TIMES) that can be used to execute the modified code.

A Push interpreter contains a random code generator that can be used to produce random programs or program fragments; the algorithm for this is provided in the Random Code Generation section below. The random code generator can be called from outside the interpreter (e.g. to create or mutate programs in a genetic programming system) or from a standard CODE.RAND instruction (which is analogous to RAND instructions available for other types). An "ephemeral random constant" mechanism allows randomly-generated code to include newly-generated literals of various types.

Execution safety is an essential feature of Push, in the sense that any syntactically correct program should execute without crashing or signaling an interrupt to the calling program. This is because Push is intended for use in evolutionary computing systems, which often require that bizarre programs (for example those that result from random mutations) be interpreted without interrupting the evolutionary process. The "stack safety" convention described above (that is, the convention that any instruction that finds insufficient arguments on the stacks acts as a NOOP) is one component of this feature. In addition, all instructions are written in ways that are internally safe; they have well defined behavior for all predictable inputs, and they typically NOOP in predictable "exceptional" situations (like division by zero). Additional safety concerns derive from the availability of explicit code manipulation and recursive execution instructions, which can in some cases produce exponential code growth or non-terminating programs. In response to these concerns a Push interpreter must enforce two limits:

- EVALPUSH-LIMIT: This is the maximum allowed number of "executions" in a single top-level call to the interpreter. The execution of a single Push instruction counts as one execution, as does the processing of a single literal, as does the descent into one layer of parentheses (that is, the processing of the "(" counts as one execution). When this limit is exceeded the interpreter aborts immediately, leaving its stacks in the states they were in prior to the abort (so they may still be examined by a calling program). Whether or not this counts as an "abnormal" termination is up to the calling program.
- MAX-POINTS-IN-PROGRAM: This is the maximum size of an item on the CODE stack, expressed as a number of *points*. A point is an instruction, a literal, or a pair of parentheses. Any instruction that

would cause this limit to be exceeded should instead act as a NOOP, leaving all stacks in the states that they were in before the execution of the instruction.

The convention regarding the order of arguments for instructions that are normally rendered as infix operators is that the argument on the *top* of the stack is treated as the *right-hand* argument and the argument second-from the top is treated as the *left-hand* argument. This means that the linear representation of an expression containing one of these instructions looks like the normal infix expression, except that the instruction is moved to the end. For example, we divide 3.14 by 1.23 using "(3.14 1.23 FLOAT./)" and we subtract 2 from 23 using "(23 2 -)".

While Push's stacks are generally treated as genuine stacks---that is, instructions take their arguments from the tops of the stacks and push their results onto the tops of the stacks---a few instructions (like YANK and SHOVE) do allow direct access to "deep" stack elements by means of integer indices. To this extent the stacks can be used as general, random access memory structures. This is one of the features that ensures the Turing-completeness of Push (another being the arbitrary name/value bindings supported by the NAME data type and DEFINE methods; see below).

Simple Examples

This section contains just a few simple examples, to give the reader a feel for the language and to demonstrate some of its unique features.

First, some simple arithmetic and logic:

```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
```

Execution of this code leaves the relevant stacks in the following states:

```
BOOLEAN STACK: ( TRUE )
CODE STACK: ( ( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR ) )
FLOAT STACK: ( 9.3 )
INTEGER STACK: ( 6 )
```

Next, some more "scrambled-looking" arithmetic:

```
( 5 1.23 INTEGER.+ ( 4 ) INTEGER.- 5.67 FLOAT.* )
```

Execution of this code leaves the relevant stacks in the following states:

```
CODE STACK: ( ( 5 1.23 INTEGER.+ ( 4 ) INTEGER.- 5.67 FLOAT.* ) )
FLOAT STACK: ( 6.9741 )
INTEGER STACK: ( 1 )
```

A few points to note about this example:

- Operations on integers and on floating point numbers can be interleaved; all instructions take their arguments from the appropriate stacks and push their results onto the appropriate stacks.
- The call to INTEGER.+ does nothing because there are not two integers on the INTEGER stack when it is executed.

- The call to INTEGER.- subtracts 4 (which is on top of the stack) from 5 (which is second on the stack), not the other way around.
- The parentheses in "(4)" have no effect on the results; parentheses serve mainly to group pieces of code for handling by code-manipulation instructions.

Here is a tiny program that adds an integer pre-loaded onto the stack to itself:

```
( INTEGER.DUP INTEGER.+ )
```

When run with 5 pre-loaded onto the INTEGER stack, for example, this leaves 10 on top of the stack. The following does the same thing in a slightly more complicated way, pushing code onto the CODE stack and then executing it:

```
( CODE.QUOTE ( INTEGER.DUP INTEGER.+ ) CODE.DO )
```

This can be converted into a new "DOUBLE" instruction as follows:

```
( DOUBLE CODE.QUOTE ( INTEGER.DUP INTEGER.+ ) CODE.DEFINE )
```

or equivalently:

```
( CODE.QUOTE ( INTEGER.DUP INTEGER.+ ) DOUBLE CODE.DEFINE )
```

Or even more concisely, using the EXEC stack (more explanation of which is provided below), as:

```
( DOUBLE EXEC.DEFINE ( INTEGER.DUP INTEGER.+ ) )
```

After executing any of these definitions the name DOUBLE will act as an instruction that doubles the number on top of the INTEGER stack.

The following more complicated example computes the factorial of an integer pre-loaded onto the INTEGER stack. This example makes use of the fact that top-level calls to the interpreter normally push the executed code onto the CODE stack before execution:

```
( CODE.QUOTE ( INTEGER.POP 1 )
  CODE.QUOTE ( CODE.DUP INTEGER.DUP 1 INTEGER.- CODE.DO INTEGER.* )
  INTEGER.DUP 2 INTEGER.< CODE.IF )
```

This works by first pushing two pieces of code (for the base case and recursive case of the recursive factorial algorithm, respectively) onto the CODE stack; these are pushed on top of the code for the full program, which is pre-loaded onto the CODE stack by the top-level call to the interpreter. The subsequent code compares the provided integer with 2 and, depending on the result of this, executes one of the pushed pieces of code (and discards the other; see the documentation for CODE.IF in the Type/Instruction Catalog below for details). In the base case this will produce an answer of 1, while in the recursive case it will recursively compute the factorial of one less than the provided number, and multiply that result by the provided number. When called with 5 pre-loaded on the INTEGER stack this leaves the relevant stacks in the following states:

```
CODE STACK: (( CODE.QUOTE ( INTEGER.POP 1 )
               CODE.QUOTE ( CODE.DUP INTEGER.DUP 1 INTEGER.- CODE.DO INTEGER.* )
               INTEGER.DUP 2 INTEGER.< CODE.IF ))
BOOLEAN STACK: ( )
INTEGER STACK: (120)
```

This example is interesting because it demonstrates the use of the code stack for recursion, but there are much simpler ways to calculate the factorial function in Push. For example the following code uses an iteration instruction to compute the factorial of a number pre-loaded onto the INTEGER stack:

```
( 1 INTEGER.MAX CODE.QUOTE INTEGER.* 1 CODE.DO*RANGE)
```

The initial "1 INTEGER.MAX" is necessary only for the special case of an input of zero. And an even more parsimonious factorial function can be written using an iteration instruction that operates on the EXEC stack:

```
( 1 INTEGER.MAX 1 EXEC.DO*RANGE INTEGER.* )
```

Instructions that manipulate the EXEC stack appear to take their arguments "from the right," rather than from the left as all other Push instructions. This is because execution procedure pushes all of the items in a list onto the EXEC stack before executing any of them, and because the items to the right will be further down in the stack. So given a program like:

```
( A B C )
```

C will be on top of the EXEC stack when B is being executed. So if B accesses the EXEC stack it will see C, even though C is to its right. While this may at first be confusing, it allows EXEC instructions to manipulate code without quotation, producing particularly concise code. For example, consider the following program, which is intended to be executed with at least two integers on the INTEGER stack and at least two floating point numbers on the FLOAT stack:

```
( INTEGER.= CODE.QUOTE FLOAT.* CODE.QUOTE FLOAT./ CODE.IF )
```

The CODE.IF instruction takes two items off of the CODE stack and executes one or the other of them depending on what's on top of the BOOLEAN stack. In this case it will execute FLOAT.*, multiplying the top two floating point numbers, if the top two integers were equal, and it will execute FLOAT./, dividing the top two floating point numbers, otherwise. With EXEC.IF, which works analogously but using the EXEC stack rather than the CODE stack, this can be expressed more concisely as:

```
( INTEGER.= EXEC.IF FLOAT.* FLOAT./)
```

The EXEC stack also permits the concise expression of recursive control structures by means of "combinators." For example, the "Y" combinator, implemented in Push as EXEC.Y, inserts, beneath the top item of the EXEC stack, a second copy of that top item wrapped in another call to EXEC.Y. The resulting recursive calling sequence might later be terminated with the EXEC.K combinator or with EXEC.POP or EXEC.FLUSH. Consider the following implementation of a "while" loop:

```
( EXEC.Y ( <BODY/CONDITION> EXEC.IF ( ) EXEC.POP ) )
```

Execution of EXEC.Y inserts a copy of the entire expression (including EXEC.Y) beneath the remainder of the expression on the EXEC stack. The <BODY/CONDITION> code is then executed. If this leaves TRUE on top of the BOOLEAN stack then EXEC.IF leaves the following empty list on top of the EXEC stack and discards the call to EXEC.POP. After the execution of the empty list (which has no effect) the full original expression will again rise to the top of the EXEC stack, starting the next recursive call. If, on the other hand, the execution of the <BODY/CONDITION> leaves FALSE on top of the BOOLEAN stack, then EXEC.IF will discard the following empty list and instead execute the EXEC.POP. The call to EXEC.POP will remove the recursive call and thereby terminate the recursion. As a concrete example, consider:

```
( ARG FLOAT.DEFINE              Define name ARG to store the top float.
```

```
EXEC.Y                        Recursively execute the following expression.
  ( ARG FLOAT.*               Push ARG and multiply it by previous top float (if any)
    1 INTEGER.-               Subtract 1 from the top integer (which acts as a counter)
    INTEGER.DUP 0 INTEGER.>   Is the top integer greater than zero?
    EXEC.IF ( ) EXEC.POP ) ) If so then recurse; otherwise terminate.
```

When run with an integer *I* pre-loaded on the INTEGER stack and floating point number *F* pre-loaded on the FLOAT stack, this will compute *F* raised to the *I* power.

Many other control structures can be implemented with combinators and other EXEC instructions, and the control structures can be added to the language as instructions using CODE.DEFINE and related instructions. **The user is warned**, however, that the use of arbitrary stack-manipulation instructions (such as ROT, YANK, and SHOVE) on the EXEC stack can create "execution spaghetti" -- that is, execution sequences that are very difficult to understand, and possibly difficult to edit/mutate without unexpected consequences. These instructions are provided for the EXEC type, as for all other types, but it might be prudent to use them sparingly. Manipulation of code on the CODE stack may also produce code that is difficult to understand, but at least it does so prior to, rather than during, its own execution. Whether evolution will make good use of EXEC stack manipulations is an open question.

Configuration

A Push interpreter is configured by setting the values of interpreter parameters, including lists of the types and instructions that can appear in randomly generated code. This information can be specified either in a configuration file, the format of which is specified below, or in code that is passed to the interpreter as "configuration code." Configuration files may not be supported by all implementations, as the configuration code mechanism is functionally equivalent but simpler to implement.

A Push3 configuration file is a plain text file. Any line beginning with "#" is a comment and is ignored. Actual configuration lines come in three forms:

- <parameter name> <value>
- type <type name>
- instruction <instruction name>

A parameter-setting line lists an implemented parameter (see below) and a value for that parameter. A type line consists of the word "type" followed by the name of an implemented type, and it has the effect of making the named type available in the interpreter; one effect of this is that the ephemeral random constant generator of the given type will be "turned on," allowing for constants of the type to be included in randomly-generated code. An instruction line consists of the word "instruction" followed by the name of an instruction; the effect of such a line is to allow the named instruction to appear in randomly-generated code.

Ideally an implementation should warn the user if the configuration "turns on" instructions that access stacks of "turned off" types, or if there are other apparent inconsistencies in the configuration. But it is not necessary that implementations do this, and there is no standardized set of inconsistencies that must be reported.

The following is a fragment of a valid configuration file:

```
## PARAMETER SETTINGS
```

```
MAX-RANDOM-FLOAT 1.0
MIN-RANDOM-FLOAT -1.0
MAX-RANDOM-INTEGER 10
MIN-RANDOM-INTEGER -10
EVALPUSH-LIMIT 1000
NEW-ERC-NAME-PROBABILITY 0.001
MAX-POINTS-IN-RANDOM-EXPRESSIONS 25
MAX-POINTS-IN-PROGRAM 100

## TYPES
type FLOAT
type NAME
type CODE
type BOOLEAN
type INTEGER

## INSTRUCTIONS
instruction INTEGER.FROMBOOLEAN
instruction INTEGER.FROMFLOAT
instruction INTEGER.>
instruction INTEGER.<
```

A Push implementation that supports configuration files should also provide a way to generate a complete configuration file, containing type and instruction lines for all implemented types and instructions. A reasonable way to configure such a Push interpreter is to start with such a complete configuration file and to comment out lines to turn off types and instructions. Note, however, that many instructions may depend on multiple types; for example, many code-manipulation and stack-manipulation instructions use integer indices and therefore require the INTEGER type, many comparison instructions require the BOOLEAN type for depositing their results, and all DEFINE instructions require the NAME type. Ideally, implementations should provide warnings when such dependencies are not satisfied by a configuration, but they need not do so; the person specifying the configuration should be sufficiently familiar with the instruction set to ensure that necessary types are "turned on."

The following parameters should be supported in configuration files:

- MIN-RANDOM-INTEGER: The minimum INTEGER that will be produced as an ephemeral random INTEGER constant or from a call to INTEGER.RAND.
- MAX-RANDOM-INTEGER: The maximum INTEGER that will be produced as an ephemeral random INTEGER constant or from a call to INTEGER.RAND.
- MIN-RANDOM-FLOAT: The minimum FLOAT that will be produced as an ephemeral random FLOAT constant or from a call to FLOAT.RAND.
- MAX-RANDOM-FLOAT: The maximum FLOAT that will be produced as an ephemeral random FLOAT constant or from a call to FLOAT.RAND.
- MAX-POINTS-IN-RANDOM-EXPRESSIONS: The maximum number of points in an expression produced by the CODE.RAND instruction.
- MAX-POINTS-IN-PROGRAM: The maximum number of points that can occur in any program on the CODE stack. Instructions that would violate this limit act as NOOPs (they do nothing).
- EVALPUSH-LIMIT: The maximum number of points that will be executed in a single top-level call to the interpreter.
- NEW-ERC-NAME-PROBABILITY: The probability that the selection of the ephemeral random NAME constant for inclusion in randomly generated code will produce a new name (rather than a name that was previously generated).
- RANDOM-SEED: A seed for the random number generator; 0 <= RANDOM-SEED <= 30081.
- TOP-LEVEL-PUSH-CODE: When TRUE (which is the default), code passed to the top level of the

interpreter will be pushed onto the CODE stack prior to execution.
- TOP-LEVEL-POP-CODE: When TRUE, the CODE stack will be popped at the end of top level calls to the interpreter. The default is FALSE.

The alternative mechanism for interpreter configuration is to pass "configuration code" as a second argument to the top level call to the interpreter. This code is simply Push code which may contain, among other things, calls to parameter-setting instructions. All of the parameters listed above have corresponding parameter-setting Push instructions, which are named with an "ENV." prefix. So, for example, the MIN-RANDOM-INTEGER parameter can be set to 100 using configuration code that includes "100 ENV.MIN-RANDOM-INTEGER". The "ENV" prefix here stands for "Environment," and it is inspired by a set of possible extensions, still under discussion, in which there would be a full-fledged environment type and an environment stack. The inclusion of parameter-setting instructions in randomly generated code is possible but probably not advisable.

The lists of "turned on" types and instructions are specified in configuration code using the ENV.TYPES and ENV.INSTRUCTIONS instructions, each of which takes an argument, which should be a list, from the CODE stack.

The following is an example piece of configuration code:

```
( 150 ENV.EVALPUSH-LIMIT
  100.0 ENV.MAX-RANDOM-FLOAT
  PI 3.141592 FLOAT.DEFINE
  CODE.QUOTE ( FLOAT./ FLOAT.* FLOAT.- FLOAT.+ ) ENV.INSTRUCTIONS
  CODE.QUOTE ( FLOAT ) ENV.TYPES )
```

This configuration code sets values for two parameters (EVALPUSH-LIMIT and MAX-RANDOM-FLOAT), leaving all other parameters at their default values (some of which may be implementation specific). It then defines an instruction called PI that will push 3.141592. The final two lines specify the instructions and types that can appear in random code; the specification here is for a minimal floating-point-only arithmetic configuration.

Random Code Generation

Several algorithms for the generation of random code have been described in the genetic programming literature. Code generation is less complicated for Push programs than it is for Lisp-style code trees, since in Push one doesn't have to worry about function "arity" or about function vs. argument positions when generating code. So it is easier, for example, to generate programs with predictable size and shape distributions.

The following is the standard Push random code generation algorithm, which is used for the CODE.RAND instruction. It may also be useful for the initialization of programs in evolutionary computation systems, and it is used for this purpose in PushGP. It produces a uniform distribution of sizes and what seems to be a reasonable distribution of shapes, in a reasonable amount of time.

Note that the instruction set referred to in RANDOM-CODE-WITH-SIZE should include any NAMEs that have been bound with a DEFINE instruction.

```
Function RANDOM-CODE (input: MAX-POINTS)
  Set ACTUAL-POINTS to a number between 1 and MAX-POINTS,
     chosen randomly with a uniform distribution.
  Return the result of RANDOM-CODE-WITH-SIZE called with input
```

```
     ACTUAL-POINTS.
End

Function RANDOM-CODE-WITH-SIZE (input: POINTS)
  If POINTS is 1 then choose a random element of the instruction
     set. If this is an ephemeral random constant then return a
     randomly-chosen value of the appropriate type; otherwise
     return the chosen element.
  Otherwise set SIZES-THIS-LEVEL to the result of DECOMPOSE
      called with both inputs (POINTS - 1). Return a list
      containing the results, in random order, of
      RANDOM-CODE-WITH-SIZE called with all inputs in
      SIZES-THIS-LEVEL.
End

Function DECOMPOSE (inputs: NUMBER,  MAX-PARTS)
  If NUMBER is 1 or MAX-PARTS is 1 then return a list
      containing NUMBER
  Otherwise set THIS-PART to be a random number between 1 and
      (NUMBER - 1). Return a list containing THIS-PART and
      all of the items in the result of DECOMPOSE with inputs
      (NUMBER - THIS-PART) and (MAX-PARTS - 1)
End
```

Implementation Notes

This section describes a few of the features of some current implementations and their interfaces. They can also be interpreted as suggestions for anyone building their own Push implementations. These are not features of the language per se, but they may help one to ensure that two implementations are consistent with one another. Implementation-SPECIFIC installation and usage notes should accompany each implementation.

Test suite scheme: A more-or-less standardized "test suite" scheme is intended to help ensure that a Push implementation behaves like other implementations. The scheme is to process three files (a configuration file, a file containing a list of literals for initializing the stacks, and a file containing a program) and to produce an output file that contains a list of literals which, if read back in to the interpreter (that is, if "executed" as a program), would re-create the stacks as they were at the end of the computation. The values in the output file are listed type by type, following the order in which types are declared in the configuration file. Each implementation should also provide some mechanism for comparing its output files with those of other implementations (for example by reading the files and comparing stack states, or by comparing the files in a way that disregards insignificant white space). For systems that use configuration code, rather than configuration files, this scheme would have to be altered somewhat.

Templates: It often makes sense to create slightly different versions of the same basic instruction for multiple types. For example, many of the standard stack-manipulation instructions (e.g. DUP, POP, etc.) make sense for all types (and depending on the way that the interpreter is written it may be possible to use identical method implementations for all of them). In Push1 an instruction overloading mechanism, using a TYPE type in the language itself, allowed one to exploit these commonalities (and also affected the semantics of the language in complex ways). In Push2 and Push3 the TYPE type was dropped and all instruction names refer to single methods; instruction names often include type names but this is just a convention. To recapture the software engineering benefits of the overloading mechanism (e.g. code reuse) an implementation should provide a template mechanism for type and instruction definitions.

Libraries: The configuration mechanism is intended to simplify the integration of new types/instructions into an interpreter. Any implementation of the language should provide a clear API for including libraries of types/instructions and for building/loading configuration files and/or configuration code.

Client-provided instructions with call-backs: Push interpreters are often imbedded within "client" programs that invoke interpreters on various inputs and do various things with the outputs. The integration of the interpreter into the client environment should be as tight as possible. Minimally, the client should be able to process a configuration file and/or configuration code, push/pop onto/from stacks, and invoke the interpreter on a piece of code. Ideally, the client should also be able to install new instructions that interact both with the client's environment and the interpreter state. For example, we use our C++ interpreter as a plug-in to the BREVE simulation environment (http://www.spiderland.org/breve), and we provide a way for the BREVE user to write instructions in BREVE's scripting language (which may, among other things, manipulate the interpreter's stacks) and add them to an embedded interpreter.

Push3 vs. Push2

Push3 differs from Push2 mainly in the following ways:

"Big picture" changes:

- **EXEC stack:** The recursion implicit in the execution procedure is now handled explicitly within Push, using an EXEC type/stack and the procedure outlined in the Push Concepts section of this document. This has two significant impacts:
  1. It simplifies the construction of a "reentrant" Push interpreter that can be suspended and resumed at any point in a computation.
  2. It supports new control structures that are more concise than those that rely on the CODE stack. See in particular the definitions of EXEC.IF, EXEC.DO*RANGE, EXEC.DO*TIMES, and EXEC.DO*COUNT in the Type/Instruction Catalog below. It also supports "combinators" that simplify the construction of certain recursive control structures (see EXEC.K, EXEC.S, and EXEC.Y).
- **New NAME definition scheme:** In previous versions of Push symbolic names were always treated as literals that were pushed onto the NAME stack when they were encountered. Later calls to SET and GET instructions could then be used to store/retrieve values associated with the names. In Push3 names that already have definitions are treated as instructions rather than as literals -- that is, their values are placed on the EXEC stack when they are encountered (except in the context of NAME.QUOTE, which causes the next encountered name to be treated as a literal regardless of whether or not it has a definition). This eliminates the need for GET instructions entirely (SET is retained but given the more descriptive name DEFINE), and it also eliminates the need for calling DO or a similar instruction in order to treat a NAME bound to CODE as a defined instruction. Names bound with CODE.DEFINE and EXEC.DEFINE now act as true defined instructions or subroutines, invoked simply by including their names in code. The table below demonstrates the possibilities for a simple instruction that multiplies an integer by 2. Note that invocation of the instruction is always more parsimonious in Push3 (requiring one rather than three items), and that instruction definition can also be more parsimonious, requiring as little as two items in addition to the body of the definition. There is, however, one case in which it is less parsimonious, when the NAME has a prior binding and the body is specified on the CODE stack. A side effect of these changes is that NAMEs can no longer have multiple bindings, one per type, as was the case in Push2. A new instruction called CODE.DEFINITION allows one to retrieve the definition of a name on the NAME stack, pushing the definition code onto the CODE stack.

	Push2	Push3

Definition	`TIMES2`
	`CODE.QUOTE ( 2 INTEGER.* )`
	`CODE.SET`
		If TIMES2 is known to have no prior binding:
		
		`TIMES2`
		`CODE.QUOTE ( 2 INTEGER.* )`
		`CODE.DEFINE`
		
		or
		
		`TIMES2`
		`EXEC.DEFINE ( 2 INTEGER.* )`
		
		If TIMES2 may have a prior binding:
		
		`NAME.QUOTE TIMES2`
		`CODE.QUOTE ( 2 INTEGER.* )`
		`CODE.DEFINE`
		
		or
		
		`NAME.QUOTE TIMES2`
		`EXEC.DEFINE ( 2 INTEGER.* )`

Invocation	`TIMES2 CODE.GET CODE.DO`	`TIMES2`

A named "TIMES2" instruction in Push2 vs. Push3.

- **Configuration code:** As described in the Configuration section of this document, one can configure a Push3 interpreter by passing "configuration code" to a top-level call to the interpreter, rather than by parsing a configuration file. This simplifies the implementation of an interpreter and may also support new applications involving computed configurations.

Additional changes:

- The "constants" mechanism in Push2, which allowed the creation of constant-pushing instructions in a configuration file, has been superceded by the new NAME definition scheme, which can do the same things (and more) more easily. Everything related to the constant mechanism has been removed (including the SETCONSTANT external call).
- A new CODE.DO*RANGE instruction has been added to facilitate the implementation of CODE.DO*TIMES and CODE.DO*COUNT in the context of the new EXEC stack. These instructions are now implemented as "macros" that expand into calls to CODE.DO*RANGE. This preserves the semantics of the original instructions except that calls will now contribute more "points" to the "evalpush count," and may therefore cause one to reach the EVALPUSH-LIMIT slightly earlier. An analogous specification has been provided for EXEC.DO* instructions.
- Defined instructions (produced with <type>.DEFINE) can now appear in random code, with the same probability as built-in instructions. This is accomplished by specifying that the "instruction set" in the random code generation algorithm includes any defined instructions.
- Two new parameters have been added: TOP-LEVEL-PUSH-CODE and TOP-LEVEL-POP-CODE. These determine how top-level calls to the interpreter deal with the code stack before and after execution.
- CODE.RAND's treatment of its argument has been changed. It now uses the minimum of the absolute value of the top integer and *max-points-in-random-expressions* as the size of the generated code. Previously it used the MOD of these numbers, which produced strange results.

- ROT instructions were added for all types. "<type>.ROT" is equivalent to "2 <type>.YANK". The name comes from FORTH, which includes a similar instruction.
- FLUSH instructions were added for all types. "<type>.FLUSH" empties the stack of the specified type. Note that EXEC.FLUSH is a "halt" instruction, as it empties the execution stack. The name for FLUSH was also borrowed from FORTH.

Push2 vs. Push1

Push2 differs from Push1 mainly in the following ways:

Changes to the language per se:

- The instruction overloading scheme in Push1 was dropped. It was deemed to be overly complex and unnecessary.
- The "TYPE" data type, which was used primarily to support instruction overloading, was dropped.
- Conversions between data types, which were previously handled by a CONVERT instruction in conjunction with the TYPE type, are now handled by conversion instructions for each appropriate type/type pair.
- Type names are now integrated into instruction identifiers (like "INTEGER.POP") rather than occurring as type literals.
- A few artifacts of the Lisp derivation of Push1 have been dropped; for example the Boolean literals are now TRUE and FALSE, rather than T and NIL.
- A few names have been changed for clarity or consistency (e.g. PULL was replaced with YANK to better reflect that it is the inverse of the SHOVE, which we didn't want to name PUSH as that might imply pushing onto the top of the stack).

Refinements to our implementations and their interfaces:

- Template-like mechanisms simplify the implementation of instructions for multiple data types (now that there is no overloading).
- A configuration file system was developed to simplify the specification of a particular set of types/instructions/parameters for a particular run. This system is intended to support libraries of types and instructions, and to work for all Push2 implementations (including our current Lisp and C++ implementations).
- A test file system was also added to help ensure that the various implementations behave in the same ways.

Under Discussion

Following are a couple of items that are currently under discussion for possible inclusion in future versions of Push:

- Standardized entries in the type/instruction catalog for other types that we commonly use (point/vector, child, unitaryMatrix, etc.).
- Various ideas for dynamic generation of new types from Push code.
- EXEC.YIELD: To implement co-routines, a YIELD instruction can be defined which, when encountered on the execution stack, will immediately break out of the interpreter. Execution could be resumed at any point in time.(Suggested by Maarten Keijzer.)

- Local arguments and binding spaces, possibly implemented by treating environments as first class objects (with a stack, etc.) (Suggested by Maarten Keijzer.)
- Exceptions as first class entities (with their own stack, etc.), allowing for the explicit handling of stack underflows, divisions by zero, etc. (Suggested by Maarten Keijzer.)
- Environments (containing stack states, variable binding states, perhaps more) as first class entities, along with associated instructions that allow for the execution of code within specified environments. This might allow for the "safe" execution of arbitrary code fragments (which would be executed in a fresh environment and would not be able to affect the environment of the remainder of the code). Ideas for this range from minimal extensions (e.g. "DO&" which executes in a fresh environment, plus a mechanism to get the results back to the calling environment) to a full-fledged ENVIRONMENT type with a stack, etc. (Suggested by Maarten Keijzer.)
- Perhaps reintroduce the generic REP for all types, which was REPlace second by first -- i.e. delete second. Then EXEC.K is just EXEC.REP. In FORTH this is called NIP. Another name might be SUBPOP. Other standard FORTH instructions should also be considered (e.g. OVER).

Rejected alternative names for recent versions of Push:

- push-- ("push minus minus", on account of the removal of OOP-like features)
- p (streamlined push... "programming with p" sounds pretty funny!)
- shove (push with more oomph)
- TAP (Type Ascribed Push, "what's on tap?")
- Qush (incrementing the first letter)
- plush
- pints (Push, No Type Stack)
- Push 2: With A Vengeance
- Push II: The Empire Strikes Back
- A few more from Maarten Keijzer:
  Push.DUP (two pushes, does not necessarily mean an improvement)
  Push.INC (not in the standard function set, and afaik push isn't incorporated)
  Push.PUSH (maybe push needs to be pushed a bit more, but one can overdo it)
- Pushkin: Push's next of kin (thanks to Christophe Mckeon)

Type/Instruction Catalog

The following are descriptions of "standard" types and instructions, in the sense that any Push3 implementation that provides these types/instructions should implement them in ways that conform to these descriptions. However, some implementations may not implement all of these types or instructions, and some implementations may implement more; use your implementation's configuration mechanism (described briefly above) to configure your implementation appropriately.

Unless otherwise noted all instructions POP *all* arguments that they consult. For example, the description of INTEGER.= states that it "Pushes TRUE onto the BOOLEAN stack if the top two INTEGERs are equal, or FALSE otherwise." In this case two items (the two that are compared) are popped from the INTEGER stack by the INTEGER.= instruction. In addition, unless otherwise noted all results are pushed *after* the arguments are popped. So for example BOOLEAN.= pushes the result of the comparison onto the BOOLEAN stack *after* popping the two values to be compared from the BOOLEAN stack. If any needed argument is not available then the instruction acts as a NOOP; that is, it does nothing, and leaves all stacks in the states that they were in prior to the call.

Indexing into stacks, e.g. for SHOVE, YANK, and YANKDUP instructions, is zero-based; that is, the top of

the stack has index 0. Negative indices into stacks are interpreted as 0 (the stack top), and indices that exceed the stack depth are interpreted as the highest meaningful value (e.g. the stack bottom for YANK, or one beyond the stack bottom for SHOVE).

Type	BOOLEAN
Description	For use in comparisons, logic, etc. Literals are TRUE and FALSE. Required for use of various comparison operators (e.g. INTEGER.=) and CODE.IF.
Instructions	• BOOLEAN.=: Pushes TRUE if the top two BOOLEANs are equal, or FALSE otherwise. • BOOLEAN.AND: Pushes the logical AND of the top two BOOLEANs. • BOOLEAN.DEFINE: Defines the name on top of the NAME stack as an instruction that will push the top item of the BOOLEAN stack onto the EXEC stack. • BOOLEAN.DUP: Duplicates the top item on the BOOLEAN stack. Does not pop its argument (which, if it did, would negate the effect of the duplication!). • BOOLEAN.FLUSH: Empties the BOOLEAN stack. • BOOLEAN.FROMFLOAT: Pushes FALSE if the top FLOAT is 0.0, or TRUE otherwise. • BOOLEAN.FROMINTEGER: Pushes FALSE if the top INTEGER is 0, or TRUE otherwise. • BOOLEAN.NOT: Pushes the logical NOT of the top BOOLEAN. • BOOLEAN.OR: Pushes the logical OR of the top two BOOLEANs. • BOOLEAN.POP: Pops the BOOLEAN stack. • BOOLEAN.RAND: Pushes a random BOOLEAN. • BOOLEAN.ROT: Rotates the top three items on the BOOLEAN stack, pulling the third item out and pushing it on top. This is equivalent to "2 BOOLEAN.YANK". • BOOLEAN.SHOVE: Inserts the top BOOLEAN "deep" in the stack, at the position indexed by the top INTEGER. • BOOLEAN.STACKDEPTH: Pushes the stack depth onto the INTEGER stack. • BOOLEAN.SWAP: Swaps the top two BOOLEANs. • BOOLEAN.YANK: Removes an indexed item from "deep" in the stack and pushes it on top of the stack. The index is taken from the INTEGER stack. • BOOLEAN.YANKDUP: Pushes a copy of an indexed item "deep" in the stack onto the top of the stack, without removing the deep item. The index is taken from the INTEGER stack.

Type	CODE
Description	For explicit code manipulation and execution. May also be used as a general list data type. This type must always be present, as the top level interpreter will push any code to be executed on the CODE stack prior to execution. However, one may turn off all CODE instructions if code manipulation is not needed.
Instructions	• CODE.=: Pushes TRUE if the top two pieces of CODE are equal, or FALSE otherwise. • CODE.APPEND: Pushes the result of appending the top two pieces of code. If one of the pieces of code is a single instruction or literal (that is, something not surrounded by parentheses) then it is surrounded by parentheses first. • CODE.ATOM: Pushes TRUE onto the BOOLEAN stack if the top piece of code is a single instruction or a literal, and FALSE otherwise (that is, if it is something

surrounded by parentheses).

- CODE.CAR: Pushes the first item of the list on top of the CODE stack. For example, if the top piece of code is "(A B)" then this pushes "A" (after popping the argument). If the code on top of the stack is not a list then this has no effect. The name derives from the similar Lisp function; a more generic name would be "FIRST".
- CODE.CDR: Pushes a version of the list from the top of the CODE stack *without* its first element. For example, if the top piece of code is "(A B)" then this pushes "(B)" (after popping the argument). If the code on top of the stack is not a list then this pushes the empty list ("( )"). The name derives from the similar Lisp function; a more generic name would be "REST".
- CODE.CONS: Pushes the result of "consing" (in the Lisp sense) the second stack item onto the first stack item (which is coerced to a list if necessary). For example, if the top piece of code is "(A B)" and the second piece of code is "X" then this pushes "(X A B)" (after popping the argument).
- CODE.CONTAINER: Pushes the "container" of the second CODE stack item within the first CODE stack item onto the CODE stack. If second item contains the first anywhere (i.e. in any nested list) then the container is the smallest sub-list that contains but is not equal to the first instance. For example, if the top piece of code is "(B (C (A)) (D (A)))" and the second piece of code is "(A)" then this pushes (C (A)). Pushes an empty list if there is no such container.
- CODE.CONTAINS: Pushes TRUE on the BOOLEAN stack if the second CODE stack item contains the first CODE stack item anywhere (e.g. in a sub-list).
- CODE.DEFINE: Defines the name on top of the NAME stack as an instruction that will push the top item of the CODE stack onto the EXEC stack.
- CODE.DEFINITION: Pushes the definition associated with the top NAME on the NAME stack (if any) onto the CODE stack. This extracts the definition for inspection/manipulation, rather than for immediate execution (although it may then be executed with a call to CODE.DO or a similar instruction).
- CODE.DISCREPANCY: Pushes a measure of the discrepancy between the top two CODE stack items onto the INTEGER stack. This will be zero if the top two items are equivalent, and will be higher the 'more different' the items are from one another. The calculation is as follows:
  1. Construct a list of all of the unique items in both of the lists (where uniqueness is determined by equalp). Sub-lists and atoms all count as items.
  2. Initialize the result to zero.
  3. For each unique item increment the result by the difference between the number of occurrences of the item in the two pieces of code.
  4. Push the result.
- CODE.DO: Recursively invokes the interpreter on the program on top of the CODE stack. After evaluation the CODE stack is popped; normally this pops the program that was just executed, but if the expression itself manipulates the stack then this final pop may end up popping something else.
- CODE.DO*: Like CODE.DO but pops the stack before, rather than after, the recursive execution.
- CODE.DO*COUNT: An iteration instruction that performs a loop (the body of which is taken from the CODE stack) the number of times indicated by the INTEGER argument, pushing an index (which runs from zero to one less than the number of iterations) onto the INTEGER stack prior to each execution of the loop body. This should be implemented as a macro that expands into a call to CODE.DO*RANGE. CODE.DO*COUNT takes a single INTEGER argument (the number of times that the loop will be executed) and a single CODE argument (the body of the loop). If the

provided INTEGER argument is negative or zero then this becomes a NOOP. Otherwise it expands into:

```
( 0 <1 - IntegerArg> CODE.QUOTE <CodeArg> CODE.DO*RANGE )
```

- CODE.DO*RANGE: An iteration instruction that executes the top item on the CODE stack a number of times that depends on the top two integers, while also pushing the loop counter onto the INTEGER stack for possible access during the execution of the body of the loop. The top integer is the "destination index" and the second integer is the "current index." First the code and the integer arguments are saved locally and popped. Then the integers are compared. If the integers are equal then the current index is pushed onto the INTEGER stack and the code (which is the "body" of the loop) is pushed onto the EXEC stack for subsequent execution. If the integers are not equal then the current index will still be pushed onto the INTEGER stack but two items will be pushed onto the EXEC stack -- first a recursive call to CODE.DO*RANGE (with the same code and destination index, but with a current index that has been either incremented or decremented by 1 to be closer to the destination index) and then the body code. Note that the range is inclusive of both endpoints; a call with integer arguments 3 and 5 will cause its body to be executed 3 times, with the loop counter having the values 3, 4, and 5. Note also that one can specify a loop that "counts down" by providing a destination index that is less than the specified current index.
- CODE.DO*TIMES: Like CODE.DO*COUNT but does not push the loop counter. This should be implemented as a macro that expands into CODE.DO*RANGE, similarly to the implementation of CODE.DO*COUNT, except that a call to INTEGER.POP should be tacked on to the front of the loop body code in the call to CODE.DO*RANGE. This call to INTEGER.POP will remove the loop counter, which will have been pushed by CODE.DO*RANGE, prior to the execution of the loop body.
- CODE.DUP: Duplicates the top item on the CODE stack. Does not pop its argument (which, if it did, would negate the effect of the duplication!).
- CODE.EXTRACT: Pushes the sub-expression of the top item of the CODE stack that is indexed by the top item of the INTEGER stack. The indexing here counts "points," where each parenthesized expression and each literal/instruction is considered a point, and it proceeds in depth first order. The entire piece of code is at index 0; if it is a list then the first item in the list is at index 1, etc. The integer used as the index is taken modulo the number of points in the overall expression (and its absolute value is taken in case it is negative) to ensure that it is within the meaningful range.
- CODE.FLUSH: Empties the CODE stack.
- CODE.FROMBOOLEAN: Pops the BOOLEAN stack and pushes the popped item (TRUE or FALSE) onto the CODE stack.
- CODE.FROMFLOAT: Pops the FLOAT stack and pushes the popped item onto the CODE stack.
- CODE.FROMINTEGER: Pops the INTEGER stack and pushes the popped integer onto the CODE stack.
- CODE.FROMNAME: Pops the NAME stack and pushes the popped item onto the CODE stack.
- CODE.IF: If the top item of the BOOLEAN stack is TRUE this recursively executes the second item of the CODE stack; otherwise it recursively executes the first item of the CODE stack. Either way both elements of the CODE stack (and the BOOLEAN value upon which the decision was made) are popped.

- CODE.INSERT: Pushes the result of inserting the second item of the CODE stack into the first item, at the position indexed by the top item of the INTEGER stack (and replacing whatever was there formerly). The indexing is computed as in CODE.EXTRACT.
- CODE.INSTRUCTIONS: Pushes a list of all active instructions in the interpreter's current configuration.
- CODE.LENGTH: Pushes the length of the top item on the CODE stack onto the INTEGER stack. If the top item is not a list then this pushes a 1. If the top item *is* a list then this pushes the number of items in the *top level* of the list; that is, nested lists contribute only 1 to this count, no matter what they contain.
- CODE.LIST: Pushes a list of the top two items of the CODE stack onto the CODE stack.
- CODE.MEMBER: Pushes TRUE onto the BOOLEAN stack if the second item of the CODE stack is a member of the first item (which is coerced to a list if necessary). Pushes FALSE onto the BOOLEAN stack otherwise.
- CODE.NOOP: Does nothing.
- CODE.NTH: Pushes the nth element of the expression on top of the CODE stack (which is coerced to a list first if necessary). If the expression is an empty list then the result is an empty list. N is taken from the INTEGER stack and is taken modulo the length of the expression into which it is indexing.
- CODE.NTHCDR: Pushes the nth "CDR" (in the Lisp sense) of the expression on top of the CODE stack (which is coerced to a list first if necessary). If the expression is an empty list then the result is an empty list. N is taken from the INTEGER stack and is taken modulo the length of the expression into which it is indexing. A "CDR" of a list is the list without its first element.
- CODE.NULL: Pushes TRUE onto the BOOLEAN stack if the top item of the CODE stack is an empty list, or FALSE otherwise.
- CODE.POP: Pops the CODE stack.
- CODE.POSITION: Pushes onto the INTEGER stack the position of the second item on the CODE stack within the first item (which is coerced to a list if necessary). Pushes -1 if no
  match is found.
- CODE.QUOTE: Specifies that the next expression submitted for execution will instead be pushed literally onto the CODE stack. This can be implemented by moving the top item on the EXEC stack onto the CODE stack.
- CODE.RAND: Pushes a newly-generated random program onto the CODE stack. The limit for the size of the expression is taken from the INTEGER stack; to ensure that it is in the appropriate range this is taken modulo the value of the MAX-POINTS-IN-RANDOM-EXPRESSIONS parameter and the absolute value of the result is used.
- CODE.ROT: Rotates the top three items on the CODE stack, pulling the third item out and pushing it on top. This is equivalent to "2 CODE.YANK".
- CODE.SHOVE: Inserts the top piece of CODE "deep" in the stack, at the position indexed by the top INTEGER.
- CODE.SIZE: Pushes the number of "points" in the top piece of CODE onto the INTEGER stack. Each instruction, literal, and pair of parentheses counts as a point.
- CODE.STACKDEPTH: Pushes the stack depth onto the INTEGER stack.
- CODE.SUBST: Pushes the result of substituting the third item on the code stack for the second item in the first item. As of this writing this is implemented only in the Lisp implementation, within which it relies on the Lisp "subst" function. As such, there are several problematic possibilities; for example "dotted-lists" can result in certain cases with empty-list arguments. If any of these problematic possibilities occurs the

	stack is left unchanged. • CODE.SWAP: Swaps the top two pieces of CODE. • CODE.YANK: Removes an indexed item from "deep" in the stack and pushes it on top of the stack. The index is taken from the INTEGER stack. • CODE.YANKDUP: Pushes a copy of an indexed item "deep" in the stack onto the top of the stack, without removing the deep item. The index is taken from the INTEGER stack.

Type	EXEC
Description	Code queued for execution. The EXEC stack maintains the execution state of the Push interpreter. Instructions that specifically manipulate the EXEC stack can be used to implement various kinds of control structures. The CODE stack can also be used in this way, but manipulations to the EXEC stack are "live" in the sense that they are manipulating the actual execution state of the interpreter, not just code that might later be executed.
Instructions	• EXEC.=: Pushes TRUE if the top two items on the EXEC stack are equal, or FALSE otherwise. • EXEC.DEFINE: Defines the name on top of the NAME stack as an instruction that will push the top item of the EXEC stack back onto the EXEC stack. • EXEC.DO*COUNT: An iteration instruction that performs a loop (the body of which is taken from the EXEC stack) the number of times indicated by the INTEGER argument, pushing an index (which runs from zero to one less than the number of iterations) onto the INTEGER stack prior to each execution of the loop body. This is similar to CODE.DO*COUNT except that it takes its code argument from the EXEC stack. This should be implemented as a macro that expands into a call to EXEC.DO*RANGE. EXEC.DO*COUNT takes a single INTEGER argument (the number of times that the loop will be executed) and a single EXEC argument (the body of the loop). If the provided INTEGER argument is negative or zero then this becomes a NOOP. Otherwise it expands into: `( 0 <1 - IntegerArg> EXEC.DO*RANGE <ExecArg> )` • EXEC.DO*RANGE: An iteration instruction that executes the top item on the EXEC stack a number of times that depends on the top two integers, while also pushing the loop counter onto the INTEGER stack for possible access during the execution of the body of the loop. This is similar to CODE.DO*COUNT except that it takes its code argument from the EXEC stack. The top integer is the "destination index" and the second integer is the "current index." First the code and the integer arguments are saved locally and popped. Then the integers are compared. If the integers are equal then the current index is pushed onto the INTEGER stack and the code (which is the "body" of the loop) is pushed onto the EXEC stack for subsequent execution. If the integers are not equal then the current index will still be pushed onto the INTEGER stack but two items will be pushed onto the EXEC stack -- first a recursive call to EXEC.DO*RANGE (with the same code and destination index, but with a current index that has been either incremented or decremented by 1 to be closer to the destination index) and then the body code. Note that the range is inclusive of both endpoints; a call with integer arguments 3 and 5 will cause its body to be executed 3 times, with the loop counter having the values 3, 4, and 5. Note also that one can specify a loop that "counts down" by providing a destination index that is less than the specified current index.

	• EXEC.DO*TIMES: Like EXEC.DO*COUNT but does not push the loop counter. This should be implemented as a macro that expands into EXEC.DO*RANGE, similarly to the implementation of EXEC.DO*COUNT, except that a call to INTEGER.POP should be tacked on to the front of the loop body code in the call to EXEC.DO*RANGE. This call to INTEGER.POP will remove the loop counter, which will have been pushed by EXEC.DO*RANGE, prior to the execution of the loop body. • EXEC.DUP: Duplicates the top item on the EXEC stack. Does not pop its argument (which, if it did, would negate the effect of the duplication!). This may be thought of as a "DO TWICE" instruction. • EXEC.FLUSH: Empties the EXEC stack. This may be thought of as a "HALT" instruction. • EXEC.IF: If the top item of the BOOLEAN stack is TRUE then this removes the second item on the EXEC stack, leaving the first item to be executed. If it is false then it removes the first item, leaving the second to be executed. This is similar to CODE.IF except that it operates on the EXEC stack. This acts as a NOOP unless there are at least two items on the EXEC stack and one item on the BOOLEAN stack. • EXEC.K: The Push implementation of the "K combinator". Removes the second item on the EXEC stack. • EXEC.POP: Pops the EXEC stack. This may be thought of as a "DONT" instruction. • EXEC.ROT: Rotates the top three items on the EXEC stack, pulling the third item out and pushing it on top. This is equivalent to "2 EXEC.YANK". • EXEC.S: The Push implementation of the "S combinator". Pops 3 items from the EXEC stack, which we will call A, B, and C (with A being the first one popped). Then pushes a list containing B and C back onto the EXEC stack, followed by another instance of C, followed by another instance of A. • EXEC.SHOVE: Inserts the top EXEC item "deep" in the stack, at the position indexed by the top INTEGER. This may be thought of as a "DO LATER" instruction. • EXEC.STACKDEPTH: Pushes the stack depth onto the INTEGER stack. • EXEC.SWAP: Swaps the top two items on the EXEC stack. • EXEC.Y: The Push implementation of the "Y combinator". Inserts beneath the top item of the EXEC stack a new item of the form "(EXEC.Y <TopItem>)". • EXEC.YANK: Removes an indexed item from "deep" in the stack and pushes it on top of the stack. The index is taken from the INTEGER stack. This may be thought of as a "DO SOONER" instruction. • EXEC.YANKDUP: Pushes a copy of an indexed item "deep" in the stack onto the top of the stack, without removing the deep item. The index is taken from the INTEGER stack.

Type	FLOAT
Description	Floating-point numbers (that is, numbers with decimal points).
Instructions	• FLOAT.%: Pushes the second stack item modulo the top stack item. If the top item is zero this acts as a NOOP. The modulus is computed as the remainder of the quotient, where the quotient has first been truncated toward negative infinity. (This is taken from the definition for the generic MOD function in Common Lisp, which is described for example at http://www.lispworks.com/reference/HyperSpec/Body/f_mod_r.htm.) • FLOAT.*: Pushes the product of the top two items. • FLOAT.+: Pushes the sum of the top two items. • FLOAT.-: Pushes the difference of the top two items; that is, the second item minus the top item.

	• FLOAT./: Pushes the quotient of the top two items; that is, the second item divided by the top item. If the top item is zero this acts as a NOOP. • FLOAT.<: Pushes TRUE onto the BOOLEAN stack if the second item is less than the top item, or FALSE otherwise. • FLOAT.=: Pushes TRUE onto the BOOLEAN stack if the top two items are equal, or FALSE otherwise. • FLOAT.>: Pushes TRUE onto the BOOLEAN stack if the second item is greater than the top item, or FALSE otherwise. • FLOAT.COS: Pushes the cosine of the top item. • FLOAT.DEFINE: Defines the name on top of the NAME stack as an instruction that will push the top item of the FLOAT stack onto the EXEC stack. • FLOAT.DUP: Duplicates the top item on the FLOAT stack. Does not pop its argument (which, if it did, would negate the effect of the duplication!). • FLOAT.FLUSH: Empties the FLOAT stack. • FLOAT.FROMBOOLEAN: Pushes 1.0 if the top BOOLEAN is TRUE, or 0.0 if the top BOOLEAN is FALSE. • FLOAT.FROMINTEGER: Pushes a floating point version of the top INTEGER. • FLOAT.MAX: Pushes the maximum of the top two items. • FLOAT.MIN: Pushes the minimum of the top two items. • FLOAT.POP: Pops the FLOAT stack. • FLOAT.RAND: Pushes a newly generated random FLOAT that is greater than or equal to MIN-RANDOM-FLOAT and less than or equal to MAX-RANDOM-FLOAT. • FLOAT.ROT: Rotates the top three items on the FLOAT stack, pulling the third item out and pushing it on top. This is equivalent to "2 FLOAT.YANK". • FLOAT.SHOVE: Inserts the top FLOAT "deep" in the stack, at the position indexed by the top INTEGER. • FLOAT.SIN: Pushes the sine of the top item. • FLOAT.STACKDEPTH: Pushes the stack depth onto the INTEGER stack. • FLOAT.SWAP: Swaps the top two BOOLEANs. • FLOAT.TAN: Pushes the tangent of the top item. • FLOAT.YANK: Removes an indexed item from "deep" in the stack and pushes it on top of the stack. The index is taken from the INTEGER stack. • FLOAT.YANKDUP: Pushes a copy of an indexed item "deep" in the stack onto the top of the stack, without removing the deep item. The index is taken from the INTEGER stack.

Type	INTEGER
Description	Integer numbers (that is, numbers without decimal points).
Instructions	• INTEGER.%: Pushes the second stack item modulo the top stack item. If the top item is zero this acts as a NOOP. The modulus is computed as the remainder of the quotient, where the quotient has first been truncated toward negative infinity. (This is taken from the definition for the generic MOD function in Common Lisp, which is described for example at http://www.lispworks.com/reference/HyperSpec/Body/f_mod_r.htm.) • INTEGER.*: Pushes the product of the top two items. • INTEGER.+: Pushes the sum of the top two items. • INTEGER.-: Pushes the difference of the top two items; that is, the second item minus

	the top item. • INTEGER./: Pushes the quotient of the top two items; that is, the second item divided by the top item. If the top item is zero this acts as a NOOP. • INTEGER.<: Pushes TRUE onto the BOOLEAN stack if the second item is less than the top item, or FALSE otherwise. • INTEGER.=: Pushes TRUE onto the BOOLEAN stack if the top two items are equal, or FALSE otherwise. • INTEGER.>: Pushes TRUE onto the BOOLEAN stack if the second item is greater than the top item, or FALSE otherwise. • INTEGER.DEFINE: Defines the name on top of the NAME stack as an instruction that will push the top item of the INTEGER stack onto the EXEC stack. • INTEGER.DUP: Duplicates the top item on the INTEGER stack. Does not pop its argument (which, if it did, would negate the effect of the duplication!). • INTEGER.FLUSH: Empties the INTEGER stack. • INTEGER.FROMBOOLEAN: Pushes 1 if the top BOOLEAN is TRUE, or 0 if the top BOOLEAN is FALSE. • INTEGER.FROMFLOAT: Pushes the result of truncating the top FLOAT. • INTEGER.MAX: Pushes the maximum of the top two items. • INTEGER.MIN: Pushes the minimum of the top two items. • INTEGER.POP: Pops the INTEGER stack. • INTEGER.RAND: Pushes a newly generated random INTEGER that is greater than or equal to MIN-RANDOM-INTEGER and less than or equal to MAX-RANDOM-INTEGER. • INTEGER.ROT: Rotates the top three items on the INTEGER stack, pulling the third item out and pushing it on top. This is equivalent to "2 INTEGER.YANK". • INTEGER.SHOVE: Inserts the second INTEGER "deep" in the stack, at the position indexed by the top INTEGER. The index position is calculated after the index is removed. • INTEGER.STACKDEPTH: Pushes the stack depth onto the INTEGER stack (thereby increasing it!). • INTEGER.SWAP: Swaps the top two INTEGERs. • INTEGER.YANK: Removes an indexed item from "deep" in the stack and pushes it on top of the stack. The index is taken from the INTEGER stack, and the indexing is done after the index is removed. • INTEGER.YANKDUP: Pushes a copy of an indexed item "deep" in the stack onto the top of the stack, without removing the deep item. The index is taken from the INTEGER stack, and the indexing is done after the index is removed.

Type	NAME
Description	For creating bindings between symbolic identifiers and values of various types; that is, for implementing (global) variables and defined instructions. Bindings are created with DEFINE instructions. Any identifier that is not a known Push instruction or a known literal of any other type is considered a NAME and will be pushed onto the NAME stack when encountered, unless it has a definition (in which case its associated value will be pushed on the EXEC stack when it is encountered. The NAME.QUOTE instruction can be used to get a name that already has a definition onto the NAME stack.
Instructions	• NAME.=: Pushes TRUE if the top two NAMEs are equal, or FALSE otherwise. • NAME.DUP: Duplicates the top item on the NAME stack. Does not pop its argument (which, if it did, would negate the effect of the duplication!).

	• NAME.FLUSH: Empties the NAME stack. • NAME.POP: Pops the NAME stack. • NAME.QUOTE: Sets a flag indicating that the next name encountered will be pushed onto the NAME stack (and not have its associated value pushed onto the EXEC stack), regardless of whether or not it has a definition. Upon encountering such a name and pushing it onto the NAME stack the flag will be cleared (whether or not the pushed name had a definition). • NAME.RAND: Pushes a newly generated random NAME. • NAME.RANDBOUNDNAME: Pushes a randomly selected NAME that already has a definition. • NAME.ROT: Rotates the top three items on the NAME stack, pulling the third item out and pushing it on top. This is equivalent to "2 NAME.YANK". • NAME.SHOVE: Inserts the top NAME "deep" in the stack, at the position indexed by the top INTEGER. • NAME.STACKDEPTH: Pushes the stack depth onto the INTEGER stack. • NAME.SWAP: Swaps the top two NAMEs. • NAME.YANK: Removes an indexed item from "deep" in the stack and pushes it on top of the stack. The index is taken from the INTEGER stack. • NAME.YANKDUP: Pushes a copy of an indexed item "deep" in the stack onto the top of the stack, without removing the deep item. The index is taken from the INTEGER stack.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0308540 and Grant No. 0216344, and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30502-00-2-0611.

Document Version History

The first version of this document was created in November, 2003.

```
YYYYMMDD
20031229: - Created this version history.
          - Addition re: use of names for code variables.
          - Additions related to instruction constants.
          - Addition of RANDOM-SEED parameter.
20040109: - Tweaked descriptions of *.GET instructions (thanks Maarten Keijzer)
20040112: - Added Random Code Generation section.
          - Changed specified behavior of division/modulus by zero to
            be NOOP rather than pushing zero. (& related text changes).
          - Change specified behavior of *.GET with unbound NAME to be NOOP.
          - Added "Under Discussion" section.
20040113: - More rejected names for Push2.
20040412: - Added CODE.FROMBOOLEAN, CODE.FROMINTEGER, CODE.FROMFLOAT
            CODE.FROMNAME, CODE.DO*COUNT, CODE.DO*TIMES.
20040730: - Major revisions for Push2->Push3 transition.
          - Maarten Keijzer added as co-author.
20040731: - Fixed typos, added "Pushkin," other minor fixes.
          - Fixed factorial examples for input of zero.
```

```
20040802: - Added correct attribution to Christophe Mckeon.
20040829: - Fixed bug in EXEC.IF example, thanks to Christophe Mckeon.
20040901: - Removed duplicate entry for BOOLEAN.POP, thanks to Christophe Mckeon.
20040910: - Cosmetic changes for publication as a Hampshire College Cognitive
            Science Technical Report.
```

[end]