“Sally” Programming Language – Manual

May 23, 2017

James Durtka

**Introduction:**

Sally is a toy programming language based loosely on a somewhat modern interpretation of the Forth programming language. The primary purpose of the language was to gain practical experience writing a lexer, parser, and interpreter based on a relatively simple programming language, and also to exercise problem-solving in unique programming paradigm.

A description of the language and base dictionary reference are included below, as well as pertinent details for each part of the project, should anyone wish to duplicate it.

**Lexer:**

Sally is a very simple language, and therefore tokenizing is straightforward. This requires only one pass over the source, and breaks down into the following token types:

* Strings, delimited by either single or double quotes
* Blocks of code, which may be nested inside square brackets []
* Word definitions, which are in the form :*name code*;
* Ordinary words, which may be any alphanumeric or any non-reserved symbols (delimited by whitespace and reserved symbols)

Reserved symbols consist of the string delimiters, the square and curly brackets, the colon and semicolon, and () which are used for comments (the tokenizer just skips over these, including nested parentheses, e.g.:

(this is a comment (this is also a comment) still a comment)

(Note, however, that nesting still requires the parentheses to be matched, e.g. a smiley in the middle of a comment will cause problems!)

Strings are passed directly through, with the delimiters stripped off. Blocks of code and word definitions form the only syntactic structure possible in the language, and are thus tokenized. There is also a special reserved token for the end of file (EOF).

**Parser:**

Parsing simply means building an abstract source tree (AST) based on the special forms in the code. Ordinary words are simply chained together, like a linked list, but blocks form a separate “branch” on the tree. The body of a word definition is also considered a “block.” In general, a block is a complete section of the tree which could potentially be executed independently; however, there is special treatment for named blocks and blocks nested in square brackets.

**Interpreter:**

In this paradigm, interpreting the language is essentially equivalent to traversing the tree (in theory, the same approach could be used to compile the language, or at least a subset thereof). The interpreter is given a body of text, which comprises an execution block, which is then tokenized and parsed into a tree which may be traversed.

The essential element of a Sally program (or a Forth program) is the “word,” which is any sequence of alphanumeric (and possibly symbolic) characters. There are some built-in words, which is necessary for certain functions, and there can also be user-defined words, which are added into a dictionary.

Importantly, for a given word there is a *stack* of definitions, which allows a word to be redefined and then later reverted to the original (including, for instance, built-in words!) There is no scoping for word definitions, they are global (as is the operation of reverting them using the built-in word “forget”, which cannot be undone)

If a word cannot be found in any dictionary or in the built-in words, the *last* recourse is to evaluate it as an integer. This behavior comes directly from Forth, and is noteworthy (as it means, for instance, that it is possible to temporarily redefine the value of a number). If a word is successfully evaluated as an integer, it is pushed to the stack.

Blocks are a special feature not typically available in ordinary Forth dialects, they allow for some very neat constructions. They can be considered as first-class lambda functions, meaning they may be passed around between functions and so on.

Like integers, strings are pushed to the stack immediately when they are encountered in the evaluation chain. There are some interesting built-in features for dealing with strings, by design because this allows for all kinds of interesting extensions to the language. Crucially, there is a way to convert a string containing valid code directly into a block, which can then be executed, enabling dynamic loading of libraries and even a certain degree of self-modifying-like behavior.

**Language Description:**

In Sally, as in Forth, the operating paradigm is the stack. Programs are formed by stringing together words, which may themselves be defined as strings of words, and these words mostly exist to act on the stack.

One of the most obvious results of the heavy reliance on the stack is postfix notation. For instance, in an ordinary language one might compute 5+2 by typing something like

val = 5 + 2

In Forth, there are no named variables (except for words), and operations take place exclusively on the stack, meaning that to perform the equivalent computation we would type:

5 2 +

and subsequently find the result (7) at the top of the stack.

Sally is designed to take full advantage of this, which eliminates some of the syntactic complexities resulting from the old way of doing things. The handling of strings and blocks, both of which are placed directly on the stack, reflects this. This also makes control flow constructs much simpler, for instance we can simply type:

[“\*” print] 5 times

which results in printing the asterisk (\*) character 5 times, or:

5 [,.] forloop

which displays the integers 0-4, or:

0= [“it’s zero” print] [“it’s nonzero” print] ifelse

which, aside from the somewhat confusing infix notation, is about as readable as it gets with Forth. There are also “while” and “until” loops, and some combinators as well, enabling some very interesting approaches to problem solving.

There are tutorials to gain a working understanding of Forth, but for Sally the majority of the usefulness of the language lies in the defined words. Remember, there are only two syntactic constructs: word definitions (via :*name code*;) and code blocks (via [*code*])

*A Note on Convention:*

*There are times, primarily when dealing with strings, when it makes little sense for a given operation to destroy the stack. For instance, when taking the length of a string using the “len” operator it makes little sense to delete the string by default, as there is a good chance we may want to use the string. In these cases, the convention is to precede the operation with a , if it is to be non-destructive. There is an operational (but not efficiency!) equivalence between the code “dup abc” and “,abc” or between “,abc [swap\*number\_outputs] drop” and “abc” whenever some word “,abc” is included in the base dictionary. Typically one will be defined in the built-ins in sally\_base.py, and the other defined in base.sal using this type of relationship.*

*Otherwise, the following conventions are typical but not strictly followed:*

*abc, abc! - Do the action described by “abc”*

*>abc - Send or convert something* ***to*** *“abc”*

*abc> - Get or convert something* ***from*** *“abc”*

*?abc - Do “abc” only if the TOS is nonzero*

*abc? - Push zero if condition “abc” is false, else push -1*

*2abc - Extend “abc” to apply to two logical elements instead of one*

**Base Dictionary Reference:**

**Stack Operations**

*n* Push the indicated integer *n* to the top of the stack ( -- *n*)

dup Duplicate the TOS (*n* -- *n n*)

2dup Duplicate the TOS and NOS (*a b* -- *a b a b*)

?dup Duplicate TOS if and only if it’s non-zero

drop Drop the TOS (*n* -- )

2drop Drop the TOS and NOS (*a b* -- )

over Make a copy of the NOS (*a b* -- *a b a*)

swap Swap the TOS and NOS (*a b* -- *b a*)

2swap Swap the top two and bottom two items (*a b c d* -- *c d a b*)

*n* pick Make a copy of the *n*th item down, where n is zero-based

*n* roll Move the *n*th item down to the top of stack, where n is zero-based

empty? Push -1 if stack is empty, else push 0

last? Push -1 if stack contains 1 or 0 items, else push 0

empty! Empty the stack

*The following items are used to implement a secondary stack. This is based on the fact that Forth originally used a secondary stack for storing return addresses of words. It isn’t used for that in Sally, however, but provided as an auxiliary or for supporting code that might rely on it.*

>r Move the TOS to the auxiliary stack

r> Move the top of the auxiliary stack to the TOS

r@ Copy the top of the auxiliary stack to the TOS

r-empty? If the auxiliary stack is empty, push -1, else push 0

r-empty! Empty the auxiliary stack

**Integer Arithmetic**

+ Pop the TOS and NOS, add them together, push the result  
(*a b* -- *sum*)

- Pop the TOS and NOS, subtract NOS from TOS, push the result  
(*a b* -- *diff*)

\* Pop the TOS and NOS, multiply them, and push the result  
(*a b* -- *prod*)

/ Pop the TOS and NOS, integer divide NOS by TOS, and push the result

mod Perform the modulo of the TOS by the NOS

/mod First push remainder, then quotient

sq Multiply TOS by itself (*n* -- *n2)*

neg Negate TOS (*n* -- *-n*)

**Integer Comparisons**

> If NOS > TOS, push -1, else push 0 (*a b* -- *cond*)

< If NOS < TOS, push -1, else push 0 (*a b* -- *cond*)

= If NOS = TOS, push -1, else push 0 (*a b* -- *cond*)

<= If NOS <= TOS, push -1, else push 0

>= If NOS >= TOS, push -1, else push 0

0= If 0 = TOS, push -1, else push 0 (*a* -- *cond*)

1. If 0 > TOS, push -1, else push 0 (*a* -- *cond*)

0< If 0 < TOS, push -1, else push 0 (*a* -- *cond*)

**Boolean Logic**

true Push -1 ( -- -1)

false Push 0 ( -- 0)

or If either TOS or NOS are nonzero, push -1, else push 0 (*a b* -- *cond*)

and If both TOS and NOS are nonzero, push -1, else push 0  
(*a b* -- *cond*)

xor If either TOS or NOS are nonzero, but not both, push -1  
(*a b* -- *cond*)

not If TOS is zero, push -1, else push 0 (*a* -- *cond*)

**Bitwise Logic**

*Note: The bitwise logic functions are very ad-hoc, and mainly exist to allow the Sierpinski gasket program to function. They work, but they directly use Python’s bitwise operations on Python’s integer type and thus may not always behave as expected!*

|| Bitwise OR on the TOS and NOS (*a b* -- *a|b*)

&& Bitwise AND on the TOS and NOS (*a b* -- *a&b*)

^^ Bitwise XOR on the TOS and NOS (*a b* -- *a^b*)

~ Bitwise NOT on the TOS (*a* -- *~a*)

**Block Combinators**

eval Pop a string from the stack, parse it, and push the resulting block of code

exec Pop a block of code from the stack and execute it

forget Pop a string from the stack and forget the most recent definition of the word named in the string

times Pop a number and a block from the stack and run the block that many times

forloop Pop a block and a number from the stack and run the block that many times, each time pushing a zero-based index to the stack (and popping it off afterward)

ifelse Pop two blocks elsecase and ifcase and a condition. The elsecase block runs if the condition is zero, the ifcase runs if the condition is nonzero.

while Pop a block. The top of the stack will be a condition, which is checked before entering the loop and each time after. The loop runs as long as the condition is nonzero.

until Same as while, but the loop runs as long as the condition is nonzero.

dip Pop a block followed by any item off the top of the stack. The item is saved until the block completes, then returned to the top of the stack.

keep Pop a block and save a copy of the item on top of the stack (without popping it). After the block completes, the item is pushed to the stack.

bi Pop two blocks q followed by p from the top of the stack. Run them in reverse order (i.e. p, then q)

2forloop Pop three blocks out2, in1, out1 and two numbers n2, n1. Nest two forloops with zero-based indices, n1 outer and n2 inner. Out1 and out2 run before and after the inner loop, respectively.

**String Functions**

asc Pop a single character off the stack and push its ASCII as integer.

chr Pop an integer off the stack and push its ASCII character.

lcase Pop a string off the stack, convert it to lowercase and push.

ucase Pop a string off the stack, convert it to uppercase and push.

str Pop anything off the stack and convert it to a string.

unescape Convert escape sequences in a string into their character representations, e.g. the string ‘\”’ gets converted to ‘”’

int Pop anything off the stack and attempt to convert it to an integer.

cat Pop two strings off the stack, concatenate, push.

cut Pop a number followed by a string, cut the string at that number.

charat Pop a number followed by a string, extract the character at that (0-based) index and push.

,len Push the length of the string formed from the top stack element.

explode Pop a string and, in reverse order, push each individual character.

unexplode Pop a number, then pop that many items and concatenate them as strings.

reverse Pop something off the stack, treat it as a string, reverse the string and push.

**Numeric Base Operations**

>hex Pop an integer, push its hexadecimal representation as a string

hex> Pop a string containing hexadecimal, convert to integer and push

>oct Pop an integer, push its octal representation as a string

oct> Pop a string containing octal, convert to integer and push

>bin Pop an integer, push its binary representation as a string

bin> Pop a string containing binary, convert to integer and push

**I/O and System Functions**

. Pop the TOS and display it

,. Display the TOS without destroying it

emit Pop an integer, convert to an ASCII character and display

cr Display a newline

**,**print Display the stack top as a string

readline Wait for the user to enter a line, push it to the stack as a string

load Pop a string with a filename off the stack, open the file, load its contents into a string and push the string to the stack

store Pop a string with a filename off the stack, pop a string off the stack, and write the string to the specified filename. (Note: if writing binary data to a file, it may be necessary to unescape the data string, otherwise e.g. the output will be the literal string “\000” rather than the desired NUL)

bye Exit the interpreter

**Debugging Functions**

.s Display the contents of the stack without removing them

.r Display the contents of the auxiliary stack without removing them

.dict Display the contents of the dictionary (mostly useful to know if a word is defined, or if it has been defined multiple times or reverted).