Schwa Implementation Notes
Version 1.0

Margaret M. Fleck

This document is aimed at users with significant experience building natural language and/or programming language implementations. It addresses issues that are likely to confuse or bore the average user.

I will start off with an apology. I'm not a professional programming languages person. There are undoubtedly some stupid mistakes in the implementation which a **real** Scheme implementer would never have made. I hope they prove minor enough that they won't impact the basic usability of this release.

1 Differences from Scheme

Yes, Schwa isn't Scheme. And certain differences run quite deep. Why did I do this rather than building something compatible with the Scheme standard, with a few simplifications and extensions?

In Schwa, many functions must be used in C as well as in interpreted code. Some compromises have to be made to create functions which can fit into stylish C code as well as stylish Scheme code. For example, having a separate boolean type would cause awkwardness every time a test was used in C. Similarly, Schwa number handling mirrors that of C. Schwa seqs and file handles are designed so that C code can pass familiar array and file pointers.

A second concern was to keep the core interpreter simple and, thus, easy to implement. Certain features, such as macros and fluid-let, don't need to be in the interpreter because Schwa makes it easy to write interpreter extensions in C. Optimizing tail recursion would require analysis and transformation of the input S-expressions. Implementing call/CC is well-known to create significant complexities and overheads. Finally, some Scheme features (e.g. fractions, letrec) are of relatively limited interest.

Syntactic extensions were required to allow arbitrary strings to be symbols and to provided nice printed representations for bundles, seqs, and ngrams.

Once it became clear that Schwa would differ significantly from Scheme, I decided to fix certain aspects of Scheme that displeased me but would not have been, on their own, worth departing from the standard. Foremost among these was adding an explicit MISSING value and using it to simplify a variety of functions (e.g. assq).

Finally, I explicitly introduced some superficial differences, such as dropping the punctuation on certain basic functions (e.g. set, null), to help remind people that the code is not Scheme.

2 Storage Details

The type tagging scheme is detailed at the top of lists.c. Integers, symbols, ngrams, and strings are stored as single words. This exploits the fact that the last two bits of pointers are zero, and that few applications need the full 32-bit integer range. Other objects, including floats, are implemented as a pointer to a 2-word block of space.

Many other LISP/scheme implementations distinguish pointers to cons cells from pointers to objects, in the top-level type tagging. This speeds up list processing operations. Schwa, instead, uses this fourth type tag to make string processing more efficient, because strings are heavily used in natural language code.

Bundle signatures currently contain the features in the order they were added to the bundle. Therefore, bundles with the same set of features, but added in a different order, have different signatures. In theory, representing all the different orders for a set of features could require storing a very large number of ngrams. I don't believe this will happen in practice, because reasonable code tends to create similar bundles in similar ways and, thus, with similar feature orders. If actual problems arise, the bundle accessors could be modified to maintain the features in sorted order, with only a modest degradation in access speed.

Some extra space is required because of automatic garbage collection. In my experience with this garbage collector and in formal studies of memory allocation algorithms, this overhead is exceedingly variable and task-dependent. Boehm's garbage collector does not copy, so it does not necessary require the 100% overhead you may be used to. Actual memory for a lot of small objects often runs around twice their nominal size. However, very low overheads often occur with larger blocks of space.

All large tables in Schwa (e.g. the symbol table, hash tables) are divided into blocks of modest size. Because some of these tables can get **very** large, blocking is essential for keeping the garbage collector happy. (Even malloc won't appreciate requests to allocate very large contiguous blocks of space, especially when your application is using a large fraction of the address space.) It also, in many cases, cuts down on the total space requirements and avoids copying during resizing.

Earrays are stored as a hash table mapping initial block positions to blocks of storage. Each block contains storage for 256 positions. The block size is large enough that the top-level hash table introduces very little overhead. That is, you won't do any better (and may do worse) by using a single contiguous array or a simple realloc-and-copy extendable array.

3 Ngram storage

The main ngram table is a hash table containing 65536 "houses." Ngrams are assigned to houses using a function which is fairly random, so that houses fill up more-or-less uniformly. (Call show_ngram_occ to see occupancy statistics.) However, the function is reversible, so that the storage required for each ngram can be reduced by the 16 bits of the house ID.

Each house contains an array of ngram data, 4 bytes per ngram. (5 bits for the car of the ngram and 27 bits for the cdr.) It also contains a sorting of this array (2 bytes per ngram), to allow fast access via binary search. Making these arrays expand dynamically introduces some additional overhead. The top-level hash table adds only a very small amount of overhead for large sets of ngrams.

Thus, Schwa's ngram table isn't as compact as a standard packed-tree representation of a set of ngrams. However, bear in mind that Schwa's ngram table must serve a broader range of purposes. Specifically:

- Ngram codes are approximately consecutive.

- The ngram table is built dynamically and ngram codes cannot be changed after they are assigned. Tree representations derive much of their compactness from optimizations that only work if occupancy details are known in advance or if the contents can be freely reorganized (e.g. sorted).
- The ngram table is balanced. Standard tree representations are not, e.g. 9.86% of the 2-grams and 3-grams in Switchboard start with the words “the,” “a”, “and”, “uh”, and “of”.
- Ngram tables allow a 21-bit range of symbol IDs, whereas many tree implementations are restricted to 16-bit IDs.
- Ngram tables can store ngrams of any order, whereas many tree implementations can’t hold anything longer than 3-grams.

When you have an application where occupancy details can be computed in advance and space is very tight, nothing prevents you from implementing tree storage for ngrams, and linking these trees into your Schwa interpreter as special objects.

The code for an ngram contains its house ID plus its position in the house. Different houses fill up a different rates, but the hash function ensures that occupancy is roughly even. Thus, for a reasonably large set of ngrams, the early positions tend to be filled in almost all houses, the middle positions are often filled, and a few houses are starting to use a small set of larger positions. Because the house ID is stored in the least significant bits of the codes, this statistical occupancy pattern translates into an approximately consecutive block of allocated codes, starting at zero.

4 Speed

Schwa is designed to have a simple, reliable interpreter, low storage overhead, and quick code development. Execution speed is a secondary consideration, because natural language code tends to be memory or disk bound. Interpreted execution speed is a very low priority, because your compute-intensive code should be written in C and your interpreted scripts should be kept simple. I’ve tried to make it as fast as possible, but some overheads were necessary.

One non-trivial overhead is the processing required for automatic garbage collection. This is more than offset by better reliability and faster development time.

The basic accessor functions check the types of their inputs, as do functions converting between Items and C types. Accessors for special objects require a label match when retrieving their stored C pointers. This causes some redundant type checks, most notably in code that dispatches to several cases based on the type of a variable. However, it makes debugging much easier.

Because of such overheads, it is not very efficient to do fine-scale data manipulation with dynamically-typed objects or in interpreted code. Examples include reading files byte-by-byte, parsing strings, and smoothing arrays of signal values. Organize your code and data structures so that such operations are done in C, with raw C data types.

5 Address space size and other platforms

This code release assumes a 32-bit address space and a linux platform.

For processing very large datasets, 32-bit is a serious limitation. The way of the future is clearly 64-bit. I do not yet own a 64-bit machine and some modifications may be required to run under 64-bit. For example, bit rotation in hashing-type functions (utils.c, ngram.c) currently pushes bits off the lefthand side of the word. However, I believe the patches will be minor.

To take full advantage of 64-bit words, precision limits (e.g. for integers) should be extended and some encodings modified. For example, (single) floats could be stored directly in the item rather than indirectly via a pointer. Finding the best ways to do this is a more substantial project.

This code requires a garbage collected version of malloc, such as the Boehm garbage collector. Boehm's garbage collector runs on many platforms, including Windows. Except for the garbage collector, the Schwa implementation is ANSI C. Therefore, I believe that Schwa can be ported to non-linux platforms with modest effort.

6 Printing and reading

The reader and pretty-printing code is somewhat of a kludge. The reader is somewhat lax about error checking.

I found that naive pretty-printing algorithms don't work well on these data structures, for two reasons. First, long lists of small items occur frequently, e.g. a paragraph is a long list of word symbols. These need to be divided into multiple output lines, with a modest set of items per line (e.g. 10, 20). The two extreme options, one item per line or everything on one line, are both hard to read.

Second, moderately complex feature bundles are often hard to read if crammed onto a single line, even when list structure of apparently comparable complexity would look just fine. Therefore, printing of bundles needs to switch somewhat sooner to a multi-line display (i.e. one feature/value pair per output line).

7 Unicode

Support for non-ASCII characters is via UTF-8 and Latin-1 (or its close relatives). Schwa does not specific a particular encoding because both Latin-1 and UTF-8 are in common use. UTF-8 is standard for texts in non-European writing systems (e.g. Arabic). However, web pages in European languages often use Latin-1 or one of its close relatives.

UTF-8 is the appropriate representation of Unicode for C and Unix/Linux applications, because it allows code to use standard string-processing libraries and algorithms. When (and if) you need to examine individual non-ASCII characters, it is easy to convert a UTF-8 string to/from an array of wide characters.

Schwa does not take responsibility for display of non-ASCII strings, decoding of these strings, or classifying characters (e.g. uppercase, vowel, punctuation). Addressing such issues properly, in general, is a many-acre minefield. It seems more reasonable to hope that each user group can find a simple solution for their own, specific, needs.

Schwa's tools (e.g. the tokenizer and SGML parser) try to avoid relying on any assumptions about the fate of the non-ASCII characters. The punctuation critical for low-level parsing, e.g. whitespace, angle brackets, seems to be largely or entirely ASCII.

8 Multi-threading

The interpreter can only have a single thread of execution. Since it has only a few global state variables, it could presumably be made re-entrant. But it is unclear why this would be useful.

Your underlying C code, on the other hand, can certainly be multi-threaded. A second C thread could be very useful when doing network transactions or managing a graphical window. Boehm's garbage collector supports multi-threaded applications. Double-check that you've compiled it with threading enabled. Also check its documentation for any current bugs/limitations.