Schwa User Manual
Version 1.0

Margaret M. Fleck

1 Introduction

Schwa is a hybrid between Scheme and C which gives you access to the best features of both languages, plus extensions for natural language processing. The low-level and compute-intensive parts of your code are ANSI/POSIX C, with its efficiency, succinct notation, and access to the full range of C libraries. However, they can also make use of an allocator with automatic garbage collection, and a range of scheme-like dynamic data structures. The Schwa interpreter provides a Scheme-like front-end for scripting, debugging, and building simple user interfaces.

The target audience for Schwa is researchers in natural language processing who want to build experimental tools quickly, but with reasonably good efficiency (storage space, computing time). It is particularly designed for building small packages of closely-related programs (e.g. doing several related types of analysis on several specific types of corpus data). Schwa has been deliberately kept small and general-purpose, anticipating a wide variety of possible extensions for different applications.

1.1 About this manual

This manual assumes that you have some familiarity with programming in both C and Scheme. If you've never used Scheme (or another reasonably modern dialect of LISP), it will be helpful to skim parts of an introduction to Scheme, such as Dybvig's book [3]. Be aware, however, that Schwa differs from Scheme in a number of ways, some superficial and some deeper.

The manual also assumes that you have unpacked and installed the Schwa code. This will allow you to play with the demo code and to supplement this manual with the (extensive) comments in the source files.

A companion document "Implementation Notes" contains hard-core details of the design, implementation, and advanced features.

2 Basic concepts

This section discusses some ideas central to programming in Schwa.

2.1 Programming model

Writing Schwa code requires building your own customized interpreter. **Don't panic.** All the hard work is done for you. Look at interp.c in the Schwa demo code, if you want reassurance.

A typical Schwa application consists of a customized interpreter, some files of C code, one or more files of wrapper definitions, and some Schwa scripts. An example can be found in the Schwa demo directory and the tools directory it depends on. The C code files contain your underlying library functions. The Schwa script files contain your top-level (main) programs, which will be evaluated by the interpreter. The wrapper definitions define the connection between the interpreted user interface and the C library functions.

The interpreter is fairly powerful but not especially fast. Therefore, your C code will contain all the compute-intensive functions. It is very easy to revise script files and create new ones. Therefore, high-level configuration features, e.g. the main control flow of the program, tuning constants and names of corpus directories, belong in the script files.

To get used to working with Schwa, you should first experiment with a live interpreter session. Then, learn to write scripts. Make a private copy of the demo directory and modify some of its scripts. Invoke the interpreter from the linux command line and experiment with the basic data structures and access functions. Then try to add some new functions, e.g. simple list-manipulation utilities, to the demo interpreter.

Finally, read the later sections of this manual and examine the library functions in the Schwa tools directory. You might try to build a reader for your favorite corpus file format or implement a simple algorithm from your research area. Eventually, you may wish to explore features for adding new syntax to the interpreter (special forms) and efficiently packing information for large data sets (earrays, ngram and symbol codes).

2.2 Interning

In natural language applications, it is common to build a table mapping strings (e.g. words) to integer code numbers. The code numbers can be compared much more quickly than the original strings. They can be used as keys for hash tables. Since codes are typically small integers, they are also more compact than the strings. The same process is used to implement symbols in Scheme and LISP. A symbol is the code number for a string, with a few type bits attached. Adding a new string to the symbol table is known as "interning" the string.

Schwa exploits this commonality by providing a single symbol table which provides code numbers for your natural language tokens and also for the symbols used as variables and function names in interpreted code. The codes are consecutive integers, starting near zero. For a data-intensive natural language application, only a small overhead is caused by mixing a few programming variable names into a large table of natural language tokens.

Schwa also extends the idea of interning to ngrams. An ngram is an interned list of symbols, represented by a compact (29-bit) code number. An ngram may be a sequence of 2 or 3 words in a trigram language model or it may be a sequence of phonemes representing a word pronunciation. The interpreter also uses them internally to represent the types of feature bundles. The ngram mapping table in Schwa provides a mapping from lists to approximately-consecutive integer codes (see Section 4.7 and the "Implementation notes" manual for details).

Schwa provides two facilities for associating data values (e.g. counts, probabilities, part of speech labels) with symbols or ngrams. Hash tables are used to associate values with small or sparse sets of symbols or ngrams. Packed tables (earrays) can efficiently map large blocks of the code space to values.

When you intern a symbol or list in the default code table, it stays in the table for the life of your program.

Therefore, temporary strings, e.g. intermediate stages in tokenization, should not be interned when you expect to process large volumes of input text. Schwa provides facilities for manipulating strings as well as symbols. It provides a generous range of symbol codes: essentially all of memory for free-standing symbols; 21-bit range for symbols used in ngrams. And it also allows you to build your own private code tables which can be used and then discarded midway through execution.

2.3 Bundles

Schwa introduces a new data type, called a bundle. Bundles are intended for representing

- dynamically-typed structs/records
- tree nodes, especially parse tree nodes
- feature bundles (as in linguistics)
- structured tags as in HTML or XML

Standard structs/records are compact but require all fields to be declared in advance, awkward for experimental code, scripting, and objects (e.g. HTML tags) with highly-variable features. Lists or assoc lists leave the programmer with too little type information when debugging, and they require lots of memory. Bundles are intended to be a compromise between these two extremes.

A bundle consists of a head and a set of feature/value pairs. The head and features must be symbols. The values can be any type of object. For example, the following bundle has the head fraction and contains two feature/value pairs.

```
[%fraction  numerator:2   denominator:7]
```

Features can be set to specific values or to a generic "on" value. For example, the following bundle has the flag background-noise set, with no specific value.

```
[%utterance
    words:(more ice cream please)
    intonation:declarative
    background-noise]
```

Bundles can also be created without an explicit head, to allow graceful encoding of linguistic feature bundles. For example, the phoneme b might be represented as

```
[consonantal voiced anterior stop]
```

The order of features in a bundle is not significant.

Bundles can be stored fairly compactly. Each bundle contains (a) an ngram containing the bundle's "signature" (head and list of features) and (b) a packed array of values. Recall that an ngram is a code number, which is mapped to a list of symbols stored in the ngram mapping table. Therefore, the ngram (a) occupies only a single 4-byte word in each bundle. For code that creates many bundles with a common signature (a very likely scenario), this representation requires much less space than storing a list of feature names in each bundle.

2.4 Schwa data types

The Schwa interpreter and Schwa dynamic data structures in C use dynamic typing. That is, values are stored together with bits which indicate their type. When these values are stored in C variables, the variables are declared to have the generic type Item. A suite of functions exists for converting between the raw C objects and their Item counterparts, e.g. item2long converts an integer Item to a regular C long integer.

An Item can have have one of the following types: integer, float, string, symbol, ngram, (linked) list, seq, (feature) bundle, function, closure, special.

Integers, floats, and strings are essentially like their C counterparts, except for the added type information. The integer values 0 and 1 are used as booleans, as in C.

Latin-1 and UTF-8 strings are ok, though it is your problem to keep track of which encoding is in use, make strings display ok on your terminal (if you care), and figure out how to extract individual characters. 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.

Symbols and ngrams were discussed in the previous section. Because symbols are used to represent natural language tokens, they are case-sensitive and any string is acceptable as a symbol name.

Schwa includes three special symbols: ANY, END, and MISSING. MISSING is used when no value is available, e.g. no value is associated with a hash key, a feature is set "off", or a value inside an array is not present. END means that there are no more values because we've reached the end of something, e.g. a file or an array. ANY means that a value is present but we don't want to specify what it is (e.g. used to set a bundle feature "on"). It can also represent a value that matches any input value, e.g. the default in a case statement or a wildcard in pattern matching.

Linked lists are like those in Scheme. Schwa contains a suite of basic list-manipulation functions. Notice that the empty list has the same type (list) as longer lists.

Bundles are sets of feature/value pairs, as described in the previous section. Headless bundles are created by setting the head to be MISSING. A feature can be set "on" by supply the generic value ANY.

A seq is an array of Items, terminated with END. Thus, it is like a C string but made up of Items rather than characters. When a seq is stored as an Item, it also contains a precomputed length (to speed up random access). C code can directly manipulate a seq just as it would any other array or string.

A function is a pointer to a C function, together with its arity. A closure is a function defined via a lambda expression in interpreted code (see below). These data types are primarily for internal use by the interpreter.

A special object consists of a label (a symbol) and an arbitrary C pointer. The core interpreter provides only minimal support for special objects and does not need to know what is at the end of your pointer. The standard Schwa environment defines three types of special objects: hash tables, file handles, and earrays. You can define more of your own (see below).

2.5 The interpreter

The interpreter reads Schwa code ("S-expressions") and prints out the corresponding values. It can be run as an interactive session, where you type code at a command prompt. It can also be called indirectly, to evaluate the contents of Schwa shell scripts. This should seem familiar if you've used a scripting language such as TCL, Python, or Perl, or Scheme implementations that support shell scripts (e.g. SIOD).

The C program for your customized interpreter must perform three steps:

- Get the standard Schwa environment.
- Extend the environment by linking in some of your C functions.
- Start up the interpreter.

The first and third steps are trivial. See Section 4.9 for the specific function calls. The interesting part is the second step.

A function is linked into the Schwa environment by calling defun. You supply the name of the C function, the name it should have in the interpreted code (often the same), and the number of input arguments it requires. However, this only works if the inputs and the outputs for your C function are all Items.

If some of the inputs and/or outputs of your C function are not Items, e.g. perhaps one of them is a C integer, you need to write a "wrapper" function. The wrapper function takes Item inputs, converts them (as necessary) to C data types, invokes your underlying C function, then converts its output (if necessary) to an Item. You then link the wrapper function into the environment using defun. The Schwa functions for converting from Items to C data types automatically check that their inputs have the correct dynamic type,

These basic linking methods will create a suite of interpreted functions that mirrors part of the C interface to your code. However, you are not restricted to this model. You can also create an interpreted interface very different from the underlying C one. For example:

- You can add extra utility functions.
- You can rename functions.
- You can hide certain C options from the script writer, e.g. supply default values in the wrapper.
- You can make the interface's structure more Scheme-like.

Several changes will make an interface seem more Scheme-like. A single generic interpreted function can examine the type of its input and dispatch to several different C functions. An interpreted function can take a variable number of input arguments, without the severe limitations of the C vararg mechanism. Finally, you can use the "special form" mechanism (Section 4.9) to create new looping constructs, dynamically redirect the interpreter's input and output channels, and so forth.

The wrapper functions are typically not intended to be called from C code. Therefore, they should be kept in a file separate from the underlying C library functions (which might be used without the interpreter). E.g. they might go in your interpreter's main file or in a special file of wrappers (see tools_wrappers.c in the Schwa tools directory). Wrappers should be declared static, to avoid polluting the C namespace.

3 Interpreted code

This section outlines the interpreted side of Schwa.

3.1 Basic interpreter features

An error break terminates a shell script. In an interactive session, the interpreter is restarted automatically.

To interrupt input or evaluation of a form without crashing the interpreter, type control-C. Control-D (you may need to type it twice) terminates the whole program. You can also type (exit). Top-level occurances of the symbol #END (i.e. not quoted and not inside a structure such as a list) also terminate the session or script.

The variables @, @@, and @@@ are bound to the last three values returned by the interpreter. The variable *backtrace* contains a list of all interpreted functions in progress when the last error break occurred.

This release provides only minimal command-line editing: backspace erases the last character, ctrl-C will flush the whole input line. Invoke the interpreter inside an emacs shell buffer to get a friendly command editing environment.

3.2 Script files

The first line of an executable Schwa script file must contain #! followed immediately (no spaces) by the full pathname of your interpreter program. For example:

```
#!/home/mfleck/mfleck/schwa/demo/interp
```

Your script file must have execute permission set (chmod u+x *.scm).

Script files normally run silently. The function debug_on causes forms and values to be printed to stderr. The function leave_live causes the script to terminate in an interactive session (even if it was terminated by an error break).

```
(debug_on)
(debug_off)
(leave_live)
```

Everything from a semicolon (;) to the end of the line is interpreted as a comment and, thus, ignored by the interpreter. Exception: semicolons are treated as ordinary characters in strings and special symbol representations (enclosed by double quotes and curly brackets, respectively).

3.3 Literals, types, and identifiers

Integers and floating point numbers are written in the standard way. They are limited to single floats and integers no more than 30 bits long. Integers are also used as booleans: 0 is false, all non-zero values are true.

A string is enclosed in double quotes in the usual way. Newline, double quote, and backslash are "slashified", i.e. written as

```
\n  \"   \\
```

Any sequence of non-zero bytes is an acceptable string.

A number of printing-type functions require a format string. These are similar to those used by the printf family of functions in C, except that they contain only three conversion/insertion markers. Use %s to insert an item, %x to insert a string item without its quotes (and unslashified), and %% to insert the % character.

Any symbol can be used as an identifier. The name of a symbol is case-sensitive and may be any string of characters. Symbols must be slashified and enclosed in curly brackets if their names (a) start with a digit, minus sign, or period or (b) contain whitespace or (c) contain one of the following characters:

```
\ # % $ ( ) ' { } [ ] ; " :
```

A special exception allows the function name - to be used without enclosing brackets.

The special symbols #MISSING, #END, and #ANY and the empty list #NIL are written with an initial hash in interpreted code (though not in C code). These symbols evaluate to themselves and cannot be rebound.

```
#MISSING
#NIL                    ;; you can also write '()
#ANY
#END
```

Lists, seqs, and ngrams are written as follows:

```
(a b c d)               ;; a list
#(a b c d)              ;; a seq
$(a b c d)              ;; an ngram
```

When used as input to the interpreter (e.g. typed at the prompt), these literal representations of seqs and ngrams are self-quoting. That is, nothing inside them is evaluated. By contrast, the interpreter will attempt to evaluate input lists or symbols. To stop interpretation, precede the symbol or list with a single quote or wrap the quote function around it.

```
(quote expression)
(quote a b c)
'(a b c)
(quote mysymbol)
'mysymbol
```

A bundle is enclosed in square brackets. If there is a head, the open bracket is immediately followed by a percent sign, followed by the name of the head. Features and feature/value pairs then follow. Each feature/value pair is joined together by a colon.

```
[%parsenode   a:aval    b:bval]    ;; bundle with a head and two feature/value pairs
[a:aval   b:bval]                  ;; headless bundle with two feature/value pairs
[a:aval   b]                       ;; feature b has the generic value ANY
```

These bundle literals are also self-quoting.

Functions, closures, and special objects have printed forms that start with #[and include the object's label and its internal pointer (so you can tell when two special objects are the same). These printed forms cannot be read back in.

The function item2type returns one of the following symbols: string, integer, list, float, symbol, bundle, function, closure, seq, ngram, special.

```
(item2type v1)
```

3.4 Equality, arithmetic, and conditionals

Two Items are eq if they (a) are look-alike ngrams, symbols, or integers or (b) occupy the same location in memory. Two Items are equal if (a) they are eq, (b) they are sequence objects (lists, strings, seqs) of the same type which contain equal Items in the same order, or (c) they are bundles containing the same set of feature/value pairs. It is unsafe to compare floats for equality of any sort. (Exception: use eq to compare a small discrete set of "landmark" float values.)

```
(eq exp1 exp2)
(equal exp1 exp2)
```

Functions are provided for testing whether an expression is the special object #END, or if it is present (not #END and not #MISSING).

```
(at_end expression)
(present expression)
```

A basic suite of arithmetic functions is provided. Feel free to define more of your own favorites.

```
(+ v1 v2 ...)
(* v1 v2 ...)
(- v1 v2)
(/ v1 v2)
(mod v1 v2)
(round v1)
(> v1 v2)
(>= v1 v2)
(< v1 v2)
(<= v1 v2)
(== v1 v2)          ;; integers only
```

Schwa adopts the C practice of using the integers 1 and 0 as the boolean values true and false. (All other values are treated as true.) The usual boolean connectives are provided:

```
(not exp1)
(and exp1 exp2 ...)
(or exp1 exp2 ...)
```

Schwa has the standard suite of Scheme conditionals. A when expression evaluates its body (which may be a sequence of several expressions) if its test evaluates to true. The case statement compares a key (using eq) to the list of literals at the start of each clause. The constant #ANY is used to mark a default clause.

```
(if test exp1)           ;; one-branch if
(if test exp1 exp2)      ;; two-branch if
(when test <body>)
(cond (test1 <body>)
      (test2 <body>)
      ....)
(case key
      ((l1 l2 ...) <body>)
      ((l4 l5 ...) <body>)
      ....
      (#ANY <body>))
(case (length mylist)
      ((0 1) 'short)
      ((2 3 4 5) 'medium)
      (#ANY 'long))
```

3.5 Bindings

All identifiers are defined in the top-level scope, initially to the value #MISSING. Their values can be reset using set. Local statically-scoped variables (shadowing the global ones) are defined using let.

```
(set var value)
(let ((var1 initial-value1)    ;; If the initial value is omitted, it defaults to #MISSING
      (var2 initial-value2)
      (var3 initial-value3)
      ... )
   <body>)
```

The underlying C environment contains a hook for dynamically-scoped variables. This facility is used internally to implement I/O redirection (see Section 4.9) but is not directly available to script writers.

Closures (interpreted functions) are created using lambda and bound to identifiers using set. Because all identifiers are pre-defined at the top-level, functions can be defined in any order and mutual recursion works automatically.

```
(lambda (v1 v2 ....) <body>)
(set functionname (lambda (v1 v2 ....) <body>))
```

3.6 Control structure

The block special form is used to group a sequence of expressions together to form a single expression.

```
(begin <body>)
```

A very primitive dynamic exit facility is provided. The form block defines a jump point, at the end of its body. Calling return causes a jump out of the smallest enclosing block. The meaning of "enclose" is dynamic, not lexical, scope. The block/return facility is intended for a planned exit from the middle of a complex piece of code (e.g. nested loops), not for handling errors.

```
(block <body>)
(return)
```

The function error triggers an error break. Exit is intended for planned (normal) termination of the program.

```
(error formatstring arg1 arg2 ...)
(exit)
```

3.7 Strings, symbols, and ngrams

Creation and conversion functions for strings.

```
(string2symbol instring)
(symbol2string insymbol)
(stringinterned instring)      ;; is there a symbol corresponding to this string?
(string2number instring)    ;; convert string to a float or integer, as appropriate
(length string)             ;; how long is the string?
```

Formatted output into a string (useful for constructing strings).

```
(sformat formatstring v1 v2 ...)
(sformat_infull formatstring v1 v2 ...)
```

Creation and conversion functions for ngrams.

```
(ngram sym1 sym2 ....)
(ngram2list ngram)
(list2ngram list)
(listinterned list)            ;; is there an ngram corresponding to this list?
(seq2ngram seq)
(ngram2seq ngram)
```

3.8 Lists, seqs, and bundles

A modest suite of list-handling utilities. The empty list has the same type (list) as all other lists. Dotted pairs are not supported, so the second input to cons must be a list.

```
(list v1 v2 ...)
(car v1)
(cdr v1)
```

```
(cadr v1)
(cons value list)
(nth n list)
(memq key list)
(member key list)
(assq key assoclist)
(delq key list)
(null list)            ;; is list empty?
(reverse list)
(append list1 list2)
```

Basic operations for seqs.

```
(seq v1 v2 ...)        ;; create a new seq containing these values
(new_seq n)            ;; create a new seq of length n, filled with MISSING
(seqget seq position)
(seqset seq position value)
(seqfill seq value)    ;; fill all positions with this value
(sortseq seq)          ;; sort contents of seq, in place, using the
                       ;;    built-in default order
```

Lists can be converted to seqs and vice versa. The length function can be applied to a list, seq, or string.

```
(list2seq list)
(seq2list seq)
(length list)
```

Basic accessors for bundles. Delete a feature/value pair by setting the feature's value to #MISSING. A feature's value can be set to the generic value #ANY, when you want to set it "on".

```
(bundle head feat1 feat2 ...)     ;; feat can be a feature or a list (feature value)
                                  ;; supply #MISSING for the head if you want a headless bundle
(bundleget bundle feature)
(bundleset bundle feature value)  ;; value defaults to #ANY if omitted
(bundlehead bundle)
(bundlefeats bundle)
```

3.9 Hashtables and Earrays

Hash tables store (key,value) pairs. Keys are compared with eq, so they should be symbols, integers, or ngrams. (The code will accept other sorts of keys, but the result is unlikely to be what you intended.) Hash table accessors return #MISSING when no value is associated with the key.

```
(new_hash)
(hashget table key)
(hashset table key value)     ;; Use the value #MISSING to delete an entry
(hashlen table)               ;; How many elements are stored in the table?
```

The function hashkeys returns a seq containing all keys stored in the table. The functions hashincrement and hashadd simplify the very common cases where you want to (respectively) increment an integer counter stored in the table or add new Items to lists stored in the table.

```
(hashkeys table)
(hashincrement table key amount)    ;; amount should be an integer
(hashadd table key value)
```

Earrays are primarily intended for optimizing large tables and, thus, should be manipulated primarily in C code. The interpreter supports only earrays that contain Items and provides only minimal support functions.

```
(new_earray)
(earray_ref earray position)
(earray_set earray position value)
```

The print functions display only a brief label for hashtables and earrays, because they tend to contain large numbers of values. To display their contents (especially for debugging purposes), see Section 3.10.

3.10 Loops

The Schwa interpreter does not guarantee optimization of tail-recursive functions. Therefore, iterative looping should be used to traverse structures of non-trivial length. Schwa offers standard while and for loops. Notice that the for loop specifies start and end, **not** start and length.

```
(while test <body>)
(for (var startval endval) <body>)
```

Special iterators are provided to loop through the sequential datastructures. Dolist works as in Scheme or LISP, binding the variable to each element of the list in sequential order. Doseq is similar, but for seqs. Dohash binds the iteration variable to (key,value) pairs, in some unspecified order. Doearray is similar, iterating over all positions that contain non-zero values.

```
(dolist (var list) <body>)
(doseq (var seq) <body>)
(dohash (var table) <body>)
(doearray (var earray) <body>)
```

3.11 I/O

Schwa defines several functions for printing to the default output. Long or complex objects are truncated to length PRINLENGTH and depth PRINDEPTH. The "infull" variants do not truncate, at the risk of potentially looping.

```
(write expression)
(write_infull expression)
(format formatstring v1 v2 ...)
(format_infull formatstring v1 v2 ...)
```

The function read reads the next S-expression from the default input.

```
(read)
```

Open file handles are special objects with label FILE_HANDLE. The three standard C output channels are pre-defined, as are the interpreter's default input and output.

```
*stdin*
*stdout*
*stderr*
*evalin*           ;; default input
*evalout*          ;; default output
```

The following functions open and close files. They trigger an error break if the file cannot be opened.

```
(open_input_file filename)
(open_output_file filename)
(close_file filehandle)
```

The interpreter's default input and output channels can be changed locally using the functions io_from and io_to. This redirection is dynamically scoped, i.e. inherited by any other interpreted functions you may invoke inside the io_from or io_to form.

```
(io_from filehandle <body>)          ;; change input channel
(io_to filehandle <body>)            ;; change output channel
```

Finally, some useful utilities:

```
(ls specification)      ;; list files matching a pattern string (e.g. filename with
                        ;;   wildcards) by calling ``ls -d''
(read_file filename)    ;; returns a list of all S-expressions in the file
(get_open_files)        ;; list of all open file handles (except the standard I/O channels)
                        ;;   useful if code may have failed to close files
```

3.12 Advanced interpreter features

You can reset the depth and length at which the print functions will truncate expressions (if the "full" option is not set). You can also set the target length for the printer's output lines (which becomes important when printing long complex objects). Finally, you can supply a customized prompt string for the interpreter.

```
(set_printdepth n)
(set_printlength n)
(set_linewidth n)
(set_prompt newstring)
```

The function eval evaluates its input form in the current environment. Apply invokes a function or closure on a list of input arguments, again in the current environment. Load reads and evaluates the contents of a file.

```
(eval form)
(eval function arglist)
(load filename)
```

4 C code

This section outlines the C code side of Schwa.

4.1 Constants

New C data types

```
Item, Hashtable, Hashptr, Earray, Earrayptr, Schwa_Env,
Ngramtable, Symboltable, Closure
```

Types of objects (e.g. as returned by item2type):

```
TYPEINT, TYPESTRING, TYPESYMBOL, TYPENGRAM, TYPELIST,
TYPEFLOAT, TYPECLOSURE, TYPEFUNCTION, TYPESEQ, TYPEBUNDLE, TYPESPECIAL.
```

Constants for certain key Items. The bit-wise representation of the value MISSING is guaranteed to be all zeros.

```
NIL, MISSING, ANY, END
FALSE, TRUE (Item versions of 0 and 1)
```

Tuning constants for the interpreter's printer:

```
PRINDEPTH, PRINLENGTH, LINEWIDTH
```

4.2 Errors, types, and type checking

The function error is provided for signalling errors for which an interpreter restart might make sense. In the rare cases when the error is unquestionably fatal (e.g. garbage collector is out of memory), die should be used to terminate the program. Their inputs are just like printf.

```
void die(const char *format, ...);
void error(const char *format, ...);
```

Most type checking is handled automatically by the Schwa utilities, but you will occasionally need to invoke the function item2type or is_special_type, or one of the type-testing macros. item2type is also useful for type dispatching via a switch statement.

```
unsigned int item2type (Item input);
int is_special_type(Item object, Item label);
assert_type(myitem,mytype)          /* error if myitem isn't mytype */
assert_input_type(myitem,mytype)    /* slightly different error message
                                       for when myitem is an input arg */
assert_special_type(myitem,label)   /* error if myitem isn't a special
                                       object or has the wrong label */
assert_special_input(myitem,label)
```

The following functions test for various special constant Items.

```
int present(Item input);   /* 0 if input is MISSING or END */
int at_end(Item input);    /* 1 if input is END */
```

Two Items are eq if they (a) are look-alike ngrams, symbols, or integers or (b) occupy the same location in memory. Two Items are equal if (a) they are eq, (b) they are sequence objects (lists, strings, seqs) of the same type which contain equal Items in the same order, or (c) they are bundles containing the same set of feature/value pairs. It is unsafe to compare floats for equality of any sort. (Exception: use eq to compare a small discrete set of "landmark" float values.)

```
int eq(Item one, Item two);
int equal(Item one, Item two);
```

4.3 Converting between Items and C data types

The following functions simply convert between C data and the corresponding Items. Notice that an Item can contain only a single float, or an integer up to 30 bits long.

```
Item string2item (char *string);
char *item2string (Item input);
Item long2item (long integer);
long item2long (Item input);
Item float2item(float value);
float item2float(Item input);
Item closure2item(Closure *table);
Closure *item2closure(Item input);
Item function2item(void *function, unsigned int arity);
void *item2function(Item input);
Item seq2item(Item *data);
Item *item2seq(Item input);
```

Special items are created by supplying a C pointer and a (symbol) label. The same label must be supplied to retrieve the C pointer from the Item, to enforce type-checking. The interpreter uses the labels FILE_HANDLE, EARRAY, and HASHTABLE. The labels FUNCTION and CLOSURE should also be avoided.

```
void *item2ptr(Item input, Item label);
Item ptr2item (void *ptr, Item label);
void setspecialptr(Item special, void *newptr);
```

The following functions extract satellite information from the Item representation: the arity of a function, the label of a special object, and the stored length of a seq.

```
unsigned int item2arity(Item input);
Item item2label(Item input);
unsigned long seqlength(Item input);
```

4.4 Lists, bundles, seqs, and sorting

List manipulation functions. The empty list has the same type (list) as all other lists. Dotted pairs are not supported, so the second input to cons must be a list. The argument to the function list should be the length n of the output list followed by exactly n Items. Notice that assq returns only the matching value (not a key/value pair), or MISSING if there isn't one.

```
int null(Item input);
Item cons (Item car, Item cdr);
Item car (Item list);
Item cdr (Item list);
Item cadr (Item list);
Item reverse (Item list);
Item nreverse (Item list);              /* destructive reverse */
Item assq(Item object, Item list);
Item list(unsigned int n, ...);
Item copylist (Item list);              /* shallow copy */
void set_cdr (Item list, Item value);
void set_car (Item list, Item value);
int memq(Item object, Item list);
Item delq(Item object, Item list);
int member(Item object, Item list);
Item nconc (Item list1, Item list2);      /* destructive append */
Item append (Item list1, Item list2);
long length(Item list);
Item nth(long n, Item list);
```

Bundle manipulation functions. Bundleget returns the matching value (not a feature/value pair), or MISSING if this feature isn't present. The special symbol ANY is used to set a feature "on" without assigning it an explicit value. Bundlefeats returns a list of all features in the bundle.

```
Item new_bundle(Item head);
Item bundleset(Item bundle, Item feature, Item value);
Item bundleget(Item bundle, Item feature);
Item bundlehead(Item bundle);
Item bundlefeats(Item bundle);
int bundleisset(Item bundle, Item feature);
```

Seq manipulation functions. You must ensure that your raw C arrays of Items are terminated by END before passing them to these functions. Notice that seqlen computes the length of a seq in its raw array representation, whereas seqlength (above) accesses the precomputed length in an Item representation.

```
void seqset(Item seq, long position, Item value);
Item seqget(Item seq, long position);
Item new_seq(long length);
void seqfill(Item seq, Item value);
Item seq2list(Item *input);
Item *list2seq(Item input);
long seqlen(Item *seq);
Item *copyseq(Item *input);
Item *subseq (Item *start, Item *end);
int seqsimilar (Item *one, Item *two);
```

The functions seqcmp and itemcmp are similar to strcmp and define a default order for seqs and arbitrary Items. You can sort an array of Items into this order using sortseq.

```
int seqcmp(Item *one, Item *two);
int itemcmp(const Item one, const Item two);
Item *sortseq(Item *input);
```

4.5 Symbols and Ngrams

The following functions allow you to intern a string as a symbol, retrieve the string corresponding to a symbol, and check whether a string has been interned.

```
Item string2symbol (char *string);
int stringinterned (char *string);     ;; has this string been interned yet?
char *symbol2string (Item symbol);
```

Notice that any C string can be converted to a symbol. The conversion process does not modify the case of letters or check that the contents are ASCII. Thus, any UTF-8 or Latin-1 string can be a symbol.

Similar suite of functions for ngrams and ngram tables, as well as conversions to/from seqs.

```
Item ngram(unsigned int n, ...);      ;; create an ngram, first input is number of input symbols
Item list2ngram (Item list);
int listinterned (Item list);         ;; has this list been interned yet?
Item ngram2list (Item ngram);
Item seq2ngram(Item *seq);
Item *ngram2seq (Item ngram);
```

The Item representation of a symbol or ngram consists of an unsigned integer code, plus a couple type bits. Occasionally (e.g. see Section 4.7), you may need to convert between the Item representation and the raw integer codes.

```
unsigned long symbol2code (Item symbol);
Item code2symbol (unsigned long code);
Item code2ngram (unsigned long code);
unsigned long ngram2code (Item symbol);
```

You may occasionally wish to create your own private symbol or ngram tables. These are simply mappings between symbols or lists and unsigned integer codes. The numreserve input to new_symboltable allows you to reserve some codes at the start for special purposes. The autoadd parameter determines what happens if the input string is not yet in the table: does the function return -1 or does it automatically add the string to the table?

```
Symboltable *new_symboltable (unsigned int numreserve);
long intern_string (char *string, Symboltable *table, int autoadd);
char *code2string (unsigned long code, Symboltable *table);
Ngramtable *new_ngramtable ();
long intern_list (Item list, Ngramtable *table, int autoadd);
Item code2list (unsigned long code, Ngramtable *table);
```

The function symbolrange displays the min and max symbol codes currently in use. show_ngram_occ displays usage statistics for the ngram table. show_ngram_occupancy displays usage statistics for a private ngram table of your own.

```
Item symbolrange (void);
Item show_ngram_occ (void);
void show_ngram_occupancy (Ngramtable *table)
```

4.6 Hashtables

Hashtables are intended for storing key/value pairs for a sparse or not-excessively-large set of keys. Hashget returns just the stored value (not a key/value pair) or MISSING if nothing is stored under this key. Keys are compared via eq. Thus, keys should be Items that are symbols, integers, or ngrams. (The code will accept other sorts of keys, but the result is unlikely to be what you intended.)

```
Hashtable *new_hashtable (void);
void hashset(Hashtable *table, Item key, Item value);
Item hashget(Hashtable *table, Item key);
```

The function hashkeys returns a seq containing all keys stored in the table. The functions hashincrement and hashadd simplify the very common cases where you want to (respectively) increment an integer counter stored in the table or add new Items to lists stored in the table.

```
Item *hashkeys(Hashtable *table);
void hashincrement(Hashtable *table, Item key, int amount);
void hashadd(Hashtable *table, Item key, Item value);
```

An iterator is provided via hashstart and hashnext. Hashnext returns a key/value pair, or END if there are no more.

```
Hashptr *hashstart(Hashtable *table);
Item hashnext(Hashptr *ptr);
```

4.7 Earrays

Earrays are a type of self-extending array, intended to provide compact storage for situations where we expect to use long contiguous blocks of positions with only a modest number of internal gaps. This would apply, for example, to properties (e.g. frequency counts) stored for most/all symbol or ngram codes. Array positions are allocated only when accessed, but in fixed-size blocks (currently 256 positions per block) to minimize storage overhead.

Symbol codes are allocated sequentially, starting at zero. Ngram codes are allocated in an order which is somewhat randomized. However, as you allocate more and more ngrams, they gradually pack themselves into the consecutive integers starting at zero. So, at any given time, they are well approximated as a consecutive set of integers with modest gaps.

To store symbol or ngram properties in an earray, you should first convert each symbol or ngram to its unsigned integer code (Section 4.5). When retrieving information from the earray, e.g. with the iterator, you will need to convert the code back to the symbol or ngram.

The earray facility gives you a raw block of storage, somewhat in the manner of malloc. The input to new_earray should be the width of the data items you intend to store (e.g. from sizeof). The pointer returned by earray_ref must be coerced to the appropriate C data type before you attempt to get or set the value at the end of it. All storage is initially filled with zeros, which will appear as MISSING if you store Items in your earray. Once allocated, each position stays fixed in memory, so the pointers returned by earray_ref will not "go stale."

```
Earray *new_earray(unsigned int width);
void *earray_ref(Earray *table, unsigned long position);
int earray_isset(Earray *table, unsigned long position);
```

The iterator for earrays loops over all positions which don't contain all zero data. The function earraynext sets the input variable position to the next position with non-zero value and returns a pointer to the corresponding value cell. A zero pointer is returned when the iterator has finished. Positions are enumerated in storage order, which is usually **not** numerical order.

```
Earrayptr *earraystart(Earray *table);
void *earraynext(Earrayptr *ptr, unsigned long *position);
```

Hashtables and Earrays are not built-in types of Items. Rather, they are implemented as special items using the labels HASHTABLE and EARRAY.

4.8 Reading and printing

These two functions read a file containing Schwa S-expressions. Read_sexp returns END when there's nothing more in the file.

```
Item read_sexp (FILE *infile);
Item read_file (char *filename);
```

These functions create printed output. They all return strings, suitable for passing into your favorite output function. Gcprintf is just like sprintf except that it dynamically allocates the output array for you. Printitem and type2string convert a single item or a single type constant to its printed representation. Printitem truncates very long or complex inputs. The "infull" version outputs full details, at the risk of looping on circular inputs.

```
char *gcprintf(const char *fmt, ...);
char *printitem (Item input);
char *printitemfull (Item input);
char *type2string(unsigned int type);
```

Format takes a Schwa format string (Section 3.3) followed by some number of Items. Again, the "infull" version doesn't trucate and may loop.

```
char *format(Item args);
char *format_infull(Item args);
```

4.9 The interpreter

The following functions create an empty environment or the standard Schwa environment.

```
Schwa_env *null_env();
Schwa_env *standard_env();
```

The function invoke_interpreter starts up the interpreter.

```
void invoke_interpreter(int argc, char *argv[], Schwa_env *environment);
```

The functions setsymbol and defun modify the input environment.by setting the value of a symbol or defining a new function.

```
void setsymbol(Item symbol, Item value, Schwa_env *environment);
void defun(char *name, void *function, int arity, Schwa_env *environment);
```

The variants defun_vararg and defun_sf allow you to define, respectively, functions that take a variable number of inputs and "special forms". A vararg function receives a single input, which is a list of all the inputs supplied by the caller. A special form receives a list of its inputs **unevaluated**, together with the current Schwa environment, and must call eval itself. Special forms are used to create new syntactic constructs (e.g. new looping commands), redirect input and output channels, and the like.

```
void defun_vararg(char *name, void *function, Schwa_env *environment);
void defun_sf(char *name, void *function, Schwa_env *environment);
```

Special forms can use the following functions to evaluate Schwa expressions. The function evalfile reads and evaluates the contents of a file. Evalsequence evaluates a list of expressions, returning the value of the last one. Evalargs evaluates a list of expressions, returning a list of the corresponding values.

```
Item evalsymbol(Item symbol, Schwa_env *environment);
Item eval(Item expression, Schwa_env *environment);
Item evalsequence(Item exprs, Schwa_env *environment);
Item evalargs(Item exprs, Schwa_env *environment);
void evalfile(char *filename, Schwa_env *environment, int interactive);
Item apply(Item head, Item args, Schwa_env *environment);
```

You may wish to write new special forms which create local variable bindings. These bindings are stored in your Schwa environment, in the fields localbindings (statically-scoped bindings) and dynamic bindings (dynamically-scoped). Redirecting the interpreter's input and output usually requires dynamic scoping. Otherwise, your local bindings should typically be created as statically scoped.

A binding list is simply a list of pairs (symbol, value). Your special form should save the old bindings in one of its C variables, add new pairs to the front of the appropriate binding list, do its work, then restore the old binding list. If your special form is a loop, add a new binding for the loop variable **only once** and reset its value for each iteration using setsymbol.

4.10 Utilities

The following miscellaneous utilities were required to implement Schwa. They are made externally visible because they would seem likely to be useful in building your own code.

Interface to Boehm's garbage collector. These functions trigger a fatal error if the garbage collector can't allocate the requested space.

```
void *safe_malloc(size_t nbytes);
void *safe_realloc(void *old, size_t newsize);
```

There are three important things you should know about these functions. First, safe_malloc and safe_realloc fill the newly-allocated space with zero bytes. Second, bad memory problems will occur if the only pointer to a garbage-collected object is stored in space created by (regular C) malloc. Third, the garbage collector will treat an object as "live" if your code is using a pointer to anywhere inside that object. You don't need to maintain a pointer to the beginning of it.

When allocating objects that cannot contain pointers, such as arrays of data values (e.g. digitized images, speech waveforms), replace safe_malloc with safe_malloc_atomic. This is important for good garbage collector performance, especially when you are using large amounts of memory. The allocator does not fill atomic objects with zeros. However, atomic objects can be resized with safe_realloc.

```
void *safe_malloc_atomic(size_t nbytes);
```

String and file input utilities. These differ from familiar C functions in that they use the garbage collector to dynamically allocate the required space. The functions read_token and sread_token read the next whitespace-delimited token from, respectively, a file or a string.

```
char *copystring (char *input);
char *substring (char *start, char *end);
char *getline(FILE *infile);
char *read_token (FILE *infile);
char *sread_token(char **string);
```

Convert a string (e.g. as read from an input source) into an integer or float Item, as appropriate.

```
Item string2number(char *input);
```

Basic utilities for constructing hash tables.

```
unsigned long hashstring(const char* key, unsigned long tablesize);
unsigned long hashint (unsigned long input, unsigned int shift);
unsigned long findhashprime (unsigned long target);
```

5 Debugging FAQ

This section describes some helpful debugging suggestions and some common symptoms whose cause might be difficult to recognize.

If your script refuses to run, with “Command not found” or a similar error, check the following. Does the first line of the script file contain the correct pathname for the interpreter? Does the interpreter executable exist (e.g. did you do “make clean” but never re-made your code)? Does the script file have execute permission set?

When your script is producing an error, first try setting debug_on and leave_live. Examine the variable *backtrace* after the error break. This will tend to catch errors in the interpreted part of your code.

To trace errors in the compiled part of your code, try running gdb. Run gdb on your interpreter, do “break error” to set a breakpoint in the error function, then do “run foo.scm” where foo.scm is the name of your script. The command “bt” will then give you a backtrace, showing the compiled function calls.

Making some temporary test scripts, e.g. modified versions of the problem one, is often very useful. Editing throw-away scripts is much faster than typing test commands by hand or recompiling your C code.

The garbage collector and dynamic type checking prevent many segmentation faults. If you get one, you should obviously look for the standard C sources of seg faults, e.g. array overruns. The most common Schwa-specific cause of seg faults is giving the wrong number or type inputs to a printing or format function. For example, handing an Item directly to printf without calling printitem first.

You can also get seg faults if the number of inputs to the functions list or ngram doesn’t match the number specified by the first argument. Similarly, problems happen if the arity specified in a call to defun doesn’t match the arity of the C function.

“My script seems to hang when it has finished. I don’t see either the Schwa prompt or shell prompt.” It may be that you’ve set leave_live but redirected the output of the script into a file. The interpreter is still reading what you type, but its prompt and evaluation outputs are going to the file rather than the terminal.

“Symbol XXX has too high an ID to use in an ngram.” You’ve interned excessively many symbols. Although symbols have a 29-bit range, symbols used in ngrams are restricted to 21 bits. A likely explanation is that your tokenization routines are interning numbers. Textbooks in Information Retrieval suggest subdividing long numbers into sequences of no more than 4 digits. Or try mapping all numbers to a generic token such as <NUMBER>. Similar problems can occur with large sets of symbolic unique IDs (e.g. story IDs in newswire data).

“cdr of list has ngram code too high for forming new ngram.” “Overflow in array for house XXX.” The current packed 32-bit representation limits ngram IDs to 29-bits. Moreover, the cdr of an ngram must have an ID no longer than 27 bits. So both of these errors mean that you've reached the size limits of the ngram table. These limits will loosen as we get more memory and move to 64-bit.

"My code is very slow or consumes much more space than I expect." You may be doing too much work in the interpreter. Move compute-intensive functions to compiled code. Looping through very large sets of values may also be unwise in interpreted code.

6 Comparison to other tools

There is a long tradition of systems which have attempted to mate compiled low-level code with an interpreted front-end. One approach (e.g. Lush, BIGLOO [11, 17]) is to provide a compiler that converts LISP/Scheme into C, coupled with an interpreter and provision for linking in fragments of C code. Other systems (e.g. TCL and Python [12, 13, 14]) reduce the role of the high-level language to scripting and "glue" code, so that it does not need to be efficient and can be entirely interpreted. Interfaces between the low-level and scripting languages can be generated automatically [1, 8]. Both types of systems focus on ease of programming in the high-level language, with very little support for writing companion C code. Therefore, there is a natural tendency to import many C functions into the high-level language, causing it to become heavyweight.

Schwa, by contrast, concentrates on making it easy to handle Scheme-like dynamic data structures in C, with an easy interface for linking this C code into the interpreter. A similar approach was used in the Elk extension kit [7]. The C side of Elk, however, includes a much smaller set of functional primitives. Garbage collection is neither automatic nor fully reliable. And the implementation is complicated by a dump-based model of constructing applications and fragile dynamic loading code.

The Festival and Flite speech synthesis systems (based on the SIOD implementation of Scheme) [2, 9, 10] seem to have a similar design philosophy. In particular, they include basic dynamic data structures for supporting linguistic algorithms [19]. However, these systems are primarily speech toolkits. The Scheme-like parts of the package are hard to extract from the speech code and not general-purpose. The Scheme-like parts of their implementations are incomplete and apparently a bit inefficient, and the performance of the garbage collector is uncertain.

Schwa is very lightweight. It is built on top of a well-debugged general-purpose garbage collector for C [4]. It uses only standard C tools for compilation and builds runnable Scheme-like programs as shell scripts (rather than via dump). It includes only a minimal core of features because it's easy to link new functions into the interpreter, add new control structures, and embed arbitrary C objects into the dynamic data structures.

Schwa also provides the natural language programmer with features specifically designed for text-handling. Users have direct access to mappings from strings and ngrams to packed ID numbers, a normally hidden component of symbol, tuple, and language model implementations. Hash tables and some basic operations on garbage-collected strings (e.g. printf, substring) are built in. A new "bundle" datatype can represent both feature bundles and dynamic structures, useful for implementing a wide range of language processing constructs (e.g. [19]).

Many features of Schwa were inspired by years of experience with previous LISP and Scheme implementations, too numerous to acknowledge individually. I would particularly like to acknowledge Kelsey and Rees's Scheme48 [6] implementation, George Carrette's SIOD implementation [2], Manuel Serrano's thesis [16], and Christian Queinnec's book [15]. The design was also strongly influenced by my joint work with Daniel Stevenson [18] on Scheme extensions for computer vision, and my experiments with using garbage-collected C to process web pages at HP Labs. Thanks also to Geoff Kuenning for improving my understanding of hash functions.

References

[1] David M. Beazley (196) “Swig: An Easty to Use Tool for Integrating Scripting Languages with C and C++”, 4th Annual Tcl/Tk Workshop.

[2] Carrette, George (1996) “SIOD: Scheme in One Defun,” http://www.cs.indiana.edu/scheme-repository/imp/siod.html.

[3] Dybvig, R. Kent (2003) “The Scheme Programming Language”, 3rd edition, MIT Press, Cambridge, MA.

[4] Boehm, H., and M. Weiser, ”Garbage Collection in an Uncooperative Environment”, Software Practice & Experience, September 1988, pp. 807-820.

[5] Harms, Daryl and Kenneth McDonald (1999) *The Quick Python Book*, Manning Publications, Greenwich, CT.

[6] Kelsey, Richard and Jonathan Rees (1994) “A Tractable Scheme Implementation”, *Lisp and Symbolic Computation 7(4)*, pp. 315–335.

[7] Laumann, Oliver and Carsten Bormann (1994) “Elk: the Extension Language Kit,” *USENIX Computing Systems* 7/4, pp. 419-449.

[8] Ewing, Greg, “ Pyrex - a Language for Writing Python Extension Modules,” http://www.cosc.canterbury.ac.nz/ greg/python/Pyrex/.

[9] “The Festival Speech Synthesis System,” http://www.cstr.ed.ac.uk/projects/festival/

[10] Black, A. and Lenzo, K. (2001) “Flite: a small fast run-time synthesis engine”, ISCA, 4th Speech Synthesis Workshop, Scotland, pp. 157–162.

[11] http://lush.sourceforge.net

[12] J. Ousterhout (1994) *Tcl and the Tk Toolkit*, Addison-Wesley.

[13] J. Ousterhout (1998) “Scripting: Higher Level Programming for the 21st Century,” *IEEE Computer* 31/3, pp. 23–30.

[14] http://www.python.org

[15] Christian Queinnec (1996) “Lisp in Small Pieces,” Cambridge University Press.

[16] Serrano, Manuel (1994) “Vers une compilation portable et performante des langages fonctionnels,” Thèse de doctorat d’universite, Universite, Pierre et Marie Curie (Paris 6), Paris.

[17] Serrano, M. and Weis, P. (1995) “Bigloo: a portable and optimizing compiler for strict functional languages,” 2nd Static Analysis Symposium (SAS), Lecture Notes on Computer Science, pp. 366–381.

[18] Stevenson, Daniel E. and Margaret M. Fleck (1997) “Programming Language Support for Digitized Images or, The Monsters in the Closet,” 1997 Usenix conference on Domain-Specific Languages, pp. 271–284.

[19] Taylor, P., Black, A., and Caley, R. (2001) “Hetrogeneous Relation Graphs as a Mechanism for Representing Linguistic Information,” Speech Communications 33, pp 153-174