Binary parsing

Goals

- JPEG, TIFF, AIFF, MP3, ELF,. . .
- Research in (text) compression
- RPC, IIOP, RMI, DNS, SLP,. . .
- ASN.1 (LDAP, X.509)

Binary parsing and unparsing are transformations between primitive or composite Scheme values and their *external binary* representations.

Examples include reading and writing JPEG, TIFF, ELF file formats, communicating with DNS, Kerberos, LDAP, SLP internet services, participating in Sun RPC and CORBA/IIOP distributed systems, storing and retrieving (arrays of) floating-point numbers in a portable and efficient way. The talk will propose a set of low- and intermediate- level procedures that make binary parsing possible.

Scheme is a good language to do research in text compression. Text compression involves a great deal of building and traversing dictionaries, trees and similar data structures, where Scheme excels. Performance doesn't matter in research, but the size of compressed files does (to figure out the bpc for the common benchmarks). Variable-bit i/o is necessary.

ASN.1 corresponds to a higher-level parsing (LR parser vs. lexer). Information in LDAP responses and X.509 certificates is structural and recursive, and so lends itself to be processed in Scheme. Variable bit i/o is necessary, and so is a binary lexer for a LR parser. Parsing of ASN.1 is a highly profitable enterprise.

Primitives and streams

- read-byte (cf. read-char)
  `(eqv? (char->integer (read-char)) (read-byte))`
  $\Longrightarrow$?

- read-u8vector (cf. read-string)

- with-input-from-u8vector, with-input-from-encoded-u8vector `'base64`,...

- read-bit, read-bits
  via overlayed streams given read-byte

- mmap-u8vector, munmap-u8vector

Scheme has a character datatype, and a collection of characters: a string. There are procedures that read/write characters and strings, perform conversions between native Scheme datatypes and strings, and strings and collections of Scheme datatypes (parsing/unparsing, from trivial to advanced).

Currently all binary i/o has to make this assumption, which does not hold in general, for example, if an input file is in Unicode.

Read-byte is a fundamental primitive; it needs to be added to the standard. Most of the other functions are library procedures. They are:

read-u8vector, which is similar to common procedures read-line or read-string. Note read-u8vector can be quite useful to copy data from one port to another: it's far more efficient to copy data in large chunks without any interpretation of separate bytes (e.g., without interpreting bytes as Unicode characters).

with-input-from-u8vector, with-output-to-u8vector: building binary i/o streams from a sequence of bytes. Streams over u8vector, u16vector, etc. provide a serial access to memory.

mmap-u8vector, munmap-u8vector: This is the single pair of procedures that is OS-specific, although mapped memory has become nearly pervasive. These procedures are to deal with memory-mapped files, shared memory, PCI space, and other hardware resources. A Scheme system must verify that the mapped memory does not overlap any of the segments of the Scheme system itself (that is easy to do: just ask for a private mapping, and the OS will do copy-on-write). We can associate a binary stream with the mapped piece of memory to handle marshaling.

Conversions

- u8vector→integer u8vector endianness
  u8vector→sinteger u8vector endianness

- u8vector-reverse,. . .

- modf, frexp, ldexp

Here are proposed conversion procedures, from a sequence of bytes to an unsigned or signed integer, minding the byte order. The u8vector in question can have size 1,2,4,8, 3 etc. bytes. These two functions therefore can be used to read shorts, longs, extra longs, etc. numbers.

u8vector-reverse and other useful u8vector operations

modf, frexp, ldexp *primitives*– all can be emulated in Scheme, yet they are quite handy (for portable FP manipulation) and can be executed very efficiently by an FPU.

Higher-level parsing and combinators

- skip-bits, next-u8token,. . .
- IIOP, RPC/XDR, RMI
- binary lexer for existing LR/LL-parsers

The above were primitives and simple conversion operations. This slide introduces *combinators* that can compose primitives for more complex (possibly iterative) actions. That's where Scheme's ability to handle higher-order functions will shine. The composition of primitives and combinators will represent binary parsing language in a _full_ notation. This is similar to XPath expressions in full notation.

Later we need to find out the most-frequently used patterns of the binary parsing language and design an abbreviated notation. The latter will need a special "interpreter". The abbreviated notation may turn out to look like Olin's regular expressions.