Middle Endianness

From : http://en.wikipedia.org/wiki/Endianness

> ... for instance, stored some 32-bit words, counting from the most significant, as: 2nd byte first, then 1st, then 4th, and finally 3rd.
>
> - *storage of a 32-bit word on a PDP-11*
>
> *increasing addresses* →
>
> | | | | | | |
> |---|---|---|---|---|---|
> | ... | 0x0B | 0x0A | 0x0D | 0x0C | ... |
>
> Note that this can be interpreted as storing the most significant "half" (16-bits) followed by the less significant half (as if big-endian) but with each half stored in little-endian format. This ordering is known as *PDP-endianness*.

Should this format be supported ? Perl can support this thru use of Pack / Unpack . Since Bdpl tries to parse files thru defining a record structure, this can be take care of in a similar manner. Eg :

```
// define a new kind of int that takes care of endianness
struct Int
{
    byte[4]  _data satisfies {1} // satisfies {1} is always true
        then
        {
            if(endianness == 0) // little
                //make _data little endian
            else if (endianness == 1 ) // big
                // make _data big endian
            else if (endianness == 2 ) // middle
                // make data middle endian
        }
};

// use it like this :
struct some_file
{
    Int aa_from_file;
    // And define a regular int with fieldize 0 which holds a "usable" int
    // value. Note that the above Int's integer value will always be usable is
    // we first assign it to a real int.
    int aa_int_val (fieldsize 0 ) satisfies {1}
        then {aa_int_val=aa_from_file;} ;
}
```

Note that fieldsize 0 is not yet usably implemented since it (at present) means that the element reads 0 bits from input specified. Thus, when performing an operation like :

```
a=5
```

If a has fieldsize 0, then it will read 0 bits from the Rhs (the constant 5 in this case). Thus, we wont be able to initialize a to anything ever. To get around this, we can change the definition of fieldsize to be : “The size of the element when the element is being read from a file on disk “ . Thus, operations in memory will work as if fieldsize is equal to bitsize of the element, but when reading form a file, the fieldsize will be the specified size. I'm not sure if this is the right thing to do, because this has other implications.

Unix file program

The file program supports Middle endianness. file reads the magic file(s) and applies tests to the input file. Thus, we should be able to parse the magic file (which is non trivial), and apply tests to the input file (The easy way to do this is iteratively).

Reading the magic file

Since magic file is a text file and (in the loosest sense of the phrase) human readable, it is not easy to parse it in bdpl . Bdpl would have to handle regexps in the source program (i.e a “file” program written in Bdpl) and also in the data (I.e, in the data that the “file” program in Bdpl reads). Regexps / CFG support in source are required cause everything in the magic file is of variable length, integers are specified in C style (so we need an int parser), One way to do this is to add a way to make [*] arrays stop matching. Since the stop rule for every field will be “\t” (cause all values are \t delimited), everything until a \t could go into a [*] array for the field. The, we would have to write procedures to decipher the type of the field and convert the field to the type. This is not trivial in Bdpl.

A compiled version of the magic file is available which needs to be investigated. This might be easier to read in (but we will still need to read in Strings and regexps and apply regexps to data).

Assuming there is some way to ask Bdpl to stop matching the [*] array (we assume the existence of a stop keyword) , each magic rule would be something of the following form in Bdpl :

```
struct magic_rule
{
    byte[*] offset satisfies { offset[$#offset] == '\t'}
        then { stop };  // offset spec has \t as its last char
                        // we wil need to parse the text offset
                        // which would be something like 0123,
                        // 123 or 0x123

    byte[*] type satisfies { type[$#type] == '\t' }
        then { stop };  // type will be a string from a specified set
                        // we will need to convert it to an int
                        // or implement == operator for strings
                        // (byte arrays) which should not be too hard.
                        // also, assume for the moment that "type"
                        // is not a keyword in bdpl.

    byte[*] optional_rule   // this field is present if offset[0] is >
                            // we will need some way to specify optional
                            // fields. this field will also have the \t
                            // test as above
    byte[*] value satisfies { value[$#value] == '\t' }
            then { stop };  // matching in the dest file is going to be tricky
                            // each "type" will have its own stopping condition
    byte[*] string satisfies { string[$#string] == <end of line char> }
        then { stop };
}
```

Applying magic rules

It is possible that we code the magic rules into our bdpl file itself, and thus dont need the reading from a magic file step described above. Each rule jumps to a offset in the file, reads in some data , compares with the given “magic” and outputs a string. Optionally, once a rule is matched, sub-rules may need to be matched, each of which is a rue itself, but can optionally . While matching sub-rules, indirect addressing maybe used (jump to the address located at x etc).

Difficult matching types :

1) regexps
2) strings with space compression
3) search (for strings starting at a given offset)

Notes :

1) Bdpl programs could be seen as a context sensitive grammar specification
2) Bdpl support reading well defined structures easily, but parsing (well defined) structures is still not supported.
3) Once parsing is supported, Bdpl would need the capability to match multiple rules at the same time on the input file.

Binary regular expressions in perl

Regular expressions in perl are anything but regular. Context sensitive parsing is done by using the matched tokens ($1 $2 etc.) in the regexp itself. Code can be executed while matching expressions.

Perl program to convert dos text files to unix :

```
perl -e 'while(<>){s/\x0d\x0a/\x0a/g; print;}'
```

Unpack / Pack

Unpack can be used to extract record structure from a stream of bytes. It is functionally similar to Bdpl (although it has wider variety of builtin types, and has lesser expressive power than Bdpl as it can handle only structures of builtin types and cant handle weird bitsizes at all). Unpack cannot be made context sensitive .

Pack should be looked into before we implement “write” in bdpl .

Tiff files

Here's a working bdpl program to parse tiff files headers and ifd:

```
struct image_file_header
```

```
{
      byte[2] endianness;
      byte[2] magic satisfies {magic[0]==42} else { print("bad magic::");};
      int ifd_pointer;
}header satisfies {1} then {$$=header.ifd_pointer*8;};

struct image_file_directory
{
  byte[2] numEntries;
  struct image_file_directory_entry
  {
        byte[2] tag;
        byte[2] typefield;
        byte[4] count;
        byte[4] value;
  }[numEntries[0]+numEntries[1]*8] ifdentry satisfies{1} then
{print(numEntries[0]+numEntries[1]*8);print("::");};
  int nextifd satisfies { nextifd>0 } then { print(nextifd); $
$=nextifd*8;print("::");} ;
}ifd;

file "x:/test/test.tif" inp_file;
read(inp_file,header);
read(inp_file,ifd);
print(ifd);
for(;ifd.nextifd > 0 ;)
{
      read(inp_file,ifd);
      print(ifd);
}
```

Resources

Binary parsing in lisp (including an id3 tag parser)

http://gigamonkeys.com/book/practical-parsing-binary-files.html

Patents

http://www.patentstorm.us/patents/7111286-fulltext.html

Language design

Various things to think of :

1) Error detecting / correction
2) Should Bdpl be a LL(k) error detecting, context sensitive parser ?

http://www3.interscience.wiley.com/cgi-bin/abstract/114193469/ABSTRACT might be interesting, but the content is not publicly accessible)

3) A stop keyword is required for signaling that the * array should stop reading content (refer to the tiff file implementation. The implementation has a for loop, which would not be required if we had stop, since the end of ifd linked list is indicated by the ifd pointer being null. To parse the file in 1 pass, we would need to the tell the * array that even though more content is available, we have reached a logical end of parsing our structure). This will also help writing a structure to read strings. A c-style string can be defined as :

```
struct string
{
   byte[*] data satisfies{data[$#data] != 0 } else {stop;};
}my_string;

A utf-8 character and string can be defined as :
struct utf8_char
{
   byte size (fieldsize 0);
   byte first_byte (fieldsize 0 );
   bit[8] first
      satisfies {1} then {first_byte=first;}
      satisfies {!first[0]} then {size=0;} else
      satisfies {!first[2]} then {size=1;} else
      satisfies {!first[3]} then {size=2;} else
      satisfies {!first[4]} then {size=4;};
   byte[size] data ;
}c;
```

4) Outputting data in Bdpl

5) Libraries

6) Maintaining context as a part of objects : Can we compute some context and save it as a part of the object in fields of 0 fieldsize (to indicate that these fields are not read from a file on disk) ? How can we initialize such variables ? Eg: refer to the (proposed) implementation of utf-8 char above.

7) Multiple satisfies / else statements (something like case statements).

8) How can we handle padding ? The elf file manual states that certain data type will be always begin at certain boundaries.

9)