**COMS W4115**

Programming Languages and Translators

Department of Computer Science

Columbia University, New York

Concept Proposal

# Binary Data Processing Language (BDPL) [1]

Aditi Rajoriya (ar2630)

Akshay Pundle (ap2503)

Preethi Narayan (pn2156)

Bharadwaj Vellore Ramesh (vrb2102)

[1] The BDPL language is not a successor to BCPL, like it might seem from the abbreviation of the name.

So, what is BDPL?

The purpose of the Binary Data Processing Language is to provide the machinery to programmers to develop applications that parse, extracts and processes information from binary files in a facile manner. The language aims to model binary input files as a data type with a specified layout, and provide mechanisms to exploit these data types to quickly and painlessly extract various pieces of information from any binary file format.

Why BDPL?

Programming languages normally deal with file in only the ASCII or UNICODE formats. Each file is treated as a sequence of characters alpha-numeric and non-printable characters. However, binary files are not normally processed with ease by such languages. Existing languages that provide support to use system functions to read and write binary files only typically require the programmer to process raw data from memory buffers into which data is read into from the file.

Increasingly, with the proliferation of multimedia in multiple formats, the need is felt for tools that will enable format interchange and multi-format data packaging for distribution through various media. This provides an excellent case for a language that allows the representation of binary file formats in a manner that allows ease of reading and writing files without the programmer having to get into the nitty-gritty's of bit-level operations. The objective of BDPL is exactly this – to provide a high-level language scheme to access and modify binary data elements. In fact, BDPL hopes to go a step further. The BDPL programmer will have the ability to incorporate entire parsing algorithms in data types.

Why wouldn't I just use 'C' language bit-fields?

'C' language bit-fields allow several of the same features that BDPL hopes to offer to a programmer – they make variable-length bit sequence easy to read and modify, as well as allowing some basic arithmetic and logical operations on those fields. However, BDPL offers the following features which surpass the abilities of bit-fields.

- Optional fields: Many binary file formats allow certain fields to be present depending upon need. The presence of fields is usually indicated by a flag elsewhere in the file. BDPL allows specification of such conditions in files using the `OPTIONAL – ON` construct.
- Repeating fields: There are frequently contents of a binary file that are in identical formats repeated several times over. BDPL allows a programmer to model this by supporting loops inside data types.

- Run-time size determination: The sizes of chunks of data in a binary file are often variable and specified or implied elsewhere in the file. BDPL allows the programmer the flexibility to cover for these possibilities by having variable-based array-size determination.
- Exception handling: Various errors are expected when handling files – errors related to file sizes, for instance. The fundamental file operations in BDPL have in-built error reporting.

Aside from these operations, BDPL allows integer arithmetic as well as logical operations to be performed on various data elements. Floating-point operations are not supported. Character processing is supported and made simple. Character arrays can be used to process strings.

File Read and write operations allow data to be read and written as desired.

Specifically, for what on earth might I possibly use this?

- Header formation/packetization algorithms
- Packet parsers
- File format interchange programs
- CRC calculation
- String search
- String insertion
- Pattern search

How do I use the language?

Binary files have a fairly typical construction, although each binary file format is described in different ways, with these ways differing largely in notation, and to a smaller extent, in the data types that span the contents of the file. Below are three examples from varied domains that illustrate this.

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

**Figure 1 The ELF File format – Section Header Description**

```
pack() {
        pack_header()
        while (nextbits()  == packet_start_code_prefix) {
                PES_packet()
        }
}
```

```
pack_header() {
        pack_start_code                                  32    bslbf
        '01'                                              2    bslbf
        system_clock_reference_base [32..30]              3    bslbf
        marker_bit                                        1    bslbf
        system_clock_reference_base [29..15]             15    bslbf
        marker_bit                                        1    bslbf
        system_clock_reference_base [14..0]              15    bslbf
        marker_bit                                        1    bslbf
        system_clock_reference_extension                  9    uimsbf
        marker_bit                                        1    bslbf
        program_mux_rate                                 22    uimsbf
        marker_bit                                        1    bslbf
        marker_bit                                        1    bslbf
        reserved                                          5    bslbf
        pack_stuffing_length                              3    uimsbf
        for (i=0;i<pack_stuffing_length;i++) {
                stuffing_byte                             8    bslbf
        }
        if (nextbits() == system_header_start_code)  {
                system_header ()
        }
}
```

**Figure 2 Format of MPEG streams in DVD**

```
ID3v2/file identifier      "ID3"
ID3v2 version              $03 00
ID3v2 flags                %abc00000
ID3v2 size             4 * %0xxxxxxx
```

**Figure 3 Format of Tags in various audio files – These contain lyrics and album artwork**

The BDPL language allows the construction of data types that model these specifications verbatim. These data types can then be used to instantiate variables into which contents of the file can be read. There will also be some basic schemes provided to Set and Get a critical element of any binary processing function, the current ‘pos’, or index into the offset within the file

which is currently being processed. A single read operation populates the contents of the variable, and allows access to them via a familiar ‘C’-style syntax. The programmer is spared the trouble innumerable shift and mask operations that are normally required to reach parts of bit-streams. Functions are supported, and follow a syntactic style closely resembling the ‘C’ programming language.

You could then extract the symbol table from an elf file to walk through its contents, insert lyrics and album artwork into audio files, rip DVDs and build a variety of other tools.

Enough of the marketing. Show me some real stuff.

The following snippet of annotated BDPL source code attempts to illustrate several of the syntactic and semantic elements of the language.

Every program must comprise two sections:

1. A data definition section which defines a template for the file.
2. A method section which is procedural in nature and comprises blocks of instructions which operates on the data extracted into variables of the above data types.

Primitive Data Types

BITFIELD -> an array of bits of non-zero length

CHAR -> 8-bit character type

INT -> Integer

FLOAT -> Float

User Defined Types

ARRAYS

STRUCT -> Ordered aggregation of primitive types, STRUCTS, and arrays of these types.

The following is an instance.

```
STRUCT Data
{
        BITFIELD[8] Signature VALID {0xFF}
        BITFIELD[8] Length;
        CHAR[Length] Data;
};

ENUMERATION encryption_scheme {none,aes,des,tdes,};
```

```
STRUCT File
{
        STRUCT Header
        {
                BITFIELD[8*4] Signature VALID {0xDEADBEEF};
                BITFIELD[2] encryption_scheme VALID {1,2,4};
                BITFIELD[13] Length range 0b0000000000000 0b1000000000000 ;
        };
        encryption_scheme es;  // 2 bits
        Data[*] data_array; // Read the rest of the (valid) input into data array
        CHAR[*] Extra; // If anything is left, read it into extra
};
```

Method Section

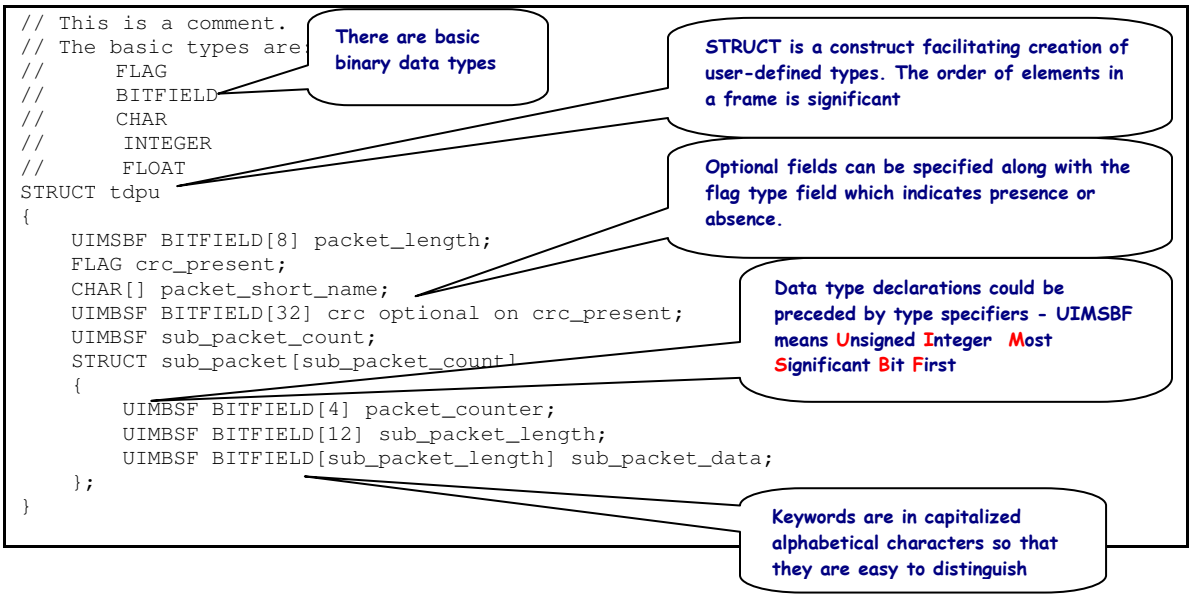
The method section is the algorithmic part of the language. This section comprises of a series of c-style statements (arithmetic, bitwise operation, while loops, conditionals).

One of the main operations provided by this language is the automatic mapping of a file to the defined data type. The method section can invoke a command "READ" to start reading a file (or portions of a file) into a defined data type. This causes a validation scheme to be run to check for compliance with the expected format. If the file is not in expected format, an error will be generated. Methods for error recovery may be written to be executed whenever non-compliances are detected at parsing. These methods will be associated with data fields or STRUCTs .

For any data type, MIN and MAX value ranges may be specified. Alternately, permitted bit-values as well as set of these values may be specified to check that the file is constructed as per the expected standard. This might be useful when validating the headers of packets in a file or in the file itself.

```
//start of the method section

File input_file;
READ ("/home/aa.jpg",input_file);
PRINT (input_file.Header.Signature);
```



```
File source;
File dest;

STRUCT tdpu packet = READ("/home/music.mp3,source);
FOR(i=0;i<packet->sub_packet_count;i++)
{
    IF(packet->encryption_status = aes_encrypted)
    {
        Decrypt(packet->sub_packet[i]);
        Dest_packet_buffer = WRITE packet->sub_packet[i];
    }
}

IF(packet->crc_present)
    check_crc();
```