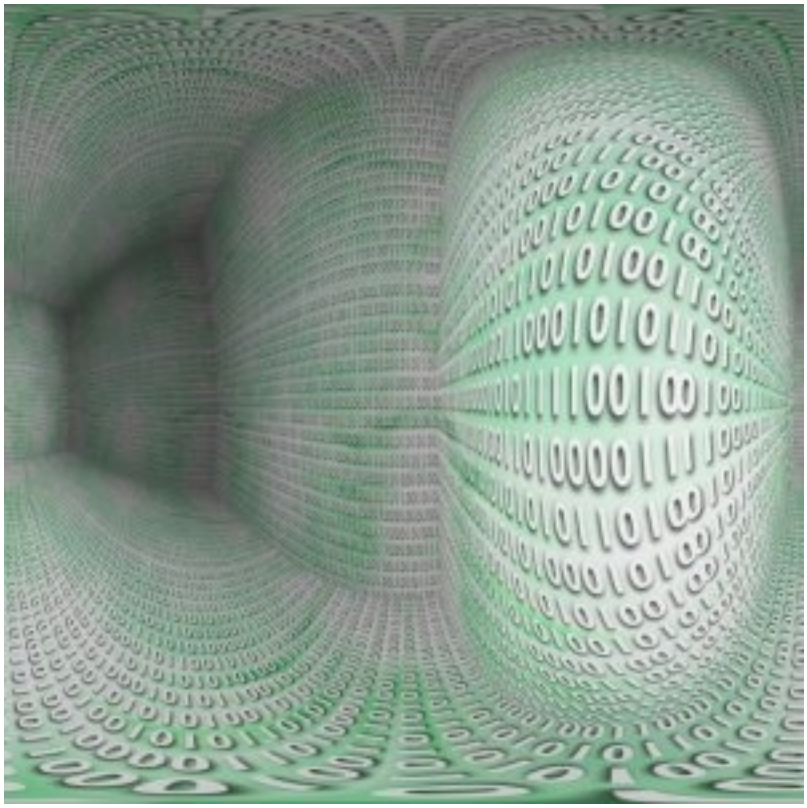
Binary Data Processing Language

Reference Manual



Aditi Rajoriya

Akshay Pundle

Bharadwaj Vellore

Preethi Narayan

Table of Contents

1 Introduction

BDPL is a programming language for processing and manipulating binary files. Specifically, the language is designed to enable a programmer to model binary file formats, read, process and write data with efficacy. In other words, the language empowers a programmer to specify a file format as one or more data structures and use these user-defined data types to parse the contents of a file.

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-gritties of

bit-level operations. The objective of BDPL is exactly this – to provide a high-level language scheme to describe the layout of, 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.

2 Lexical conventions

BDPL comprises the following types of tokens – keywords, identifiers, constants, operators and separators. The language is a free-form language; spaces, tabs and newlines only serve to separate tokens of the language. If the input stream has been parsed into tokens up to a certain character, the next token is taken to include the longest string of characters that could possible constitute a token.

2.1 Comments

Comments are always single-line only. The characters // begin a comment which terminates at the end of the line.

2.2 Identifiers

An identifier is a sequence of letters, digits and the underscore character, with the first character being either a letter or an underscore('_'). Letters are case sensitive; an upper case character counts as different from a lower case character. Only the first eight characters of identifiers are significant.

2.3 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

```
bit         byte        else        exit        fieldsize   for         if
nok         ok          on          optional    print       read        set
struct      this        type        valid       write
```

2.4 Reserved Words

The following strings are not identifiers in BDPL. However, they are reserved for future use as keywords and therefore, should not be used as identifiers in the language.

```
uimsbf      imsbf       uilsbf      ilsbf
```

2.5 Constants

There are these types of constants.

2.5.1 Decimal constants

A decimal constant is any sequence of digits.

2.5.2 Hexadecimal constants

A hexadecimal constant begins with the character sequences '0' 'x' or '0' 'X', and comprises any sequence of digits, lower case letters 'a' to 'f' and upper case letters 'A' to 'F' thereafter.

2.5.3 Binary constants

A binary constant begins with the character sequences '0' 'b' or '0' 'B', and comprises any sequence of the binary digits '1' and '0' thereafter.

2.5.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing.

2.5.5 String constants

A string is a sequence of characters surrounded by double quotes. Double quotes within strings should be preceded by an additional double quote.

3 Types

Every BDPL identifier has a 'type' which determines the meaning of the value stored in the identifier's storage.

3.1 Basic Types

There are these basic types in BDPL:

`bit` is a single bit of data

`byte` is a sequence of 8 bits

`int` is a 32 bit integer represented in a 2's complement form

`float` is a single precision floating quantity having a magnitude in the range approximately $10^{\pm 38}$ or 0. Their precision is 24 bits or about 7 decimal digits.

`double` is a double precision floating quantity having the same range as floats and a precision of 56 bits or about 17 decimal digits.

More types can be derived from these basic types using the following constructs:

`arrays`, which are sequences of elements of the same type

`structures`, which contain elements of various types

3.2 Arrays

An array is a sequence of elements of the same type. Arrays can be formed from basic types or from structures (defined later). Individual elements of an array can be addressed by specifying the "index" of the element. An index of an element is the position of the element from the start.

Suppose A is an array, then A[i] addresses the element at a distance of i elements from the start of the array. Thus, A[0] denotes the first element off the array (or equivalently, an element at a distance 0 from the start of the array). The nature of the elements of A determines whether each element is of the same size. This may not be the case if the array is of structures. Structures of the same type can have variable sizes (as we will see in the next section), and thus, each array element may be of a different size.

The concept of arrays in BDPL is illustrated by the following example of an array declaration:

```
byte[10] ar1,ar2;
```

This declaration defines two arrays, a1 and a2, each of which comprises 10 bytes. The expression ar1[3] refers to the fourth element of ar1 (or equivalently, an element at a distance of 3 elements form the start of ar1).

BDPL allows addressing elements at multiple offsets from the array at once. This is possible if the index element of the array is one of a comma-separated list of indices, a range of indices or a comma-separated list of ranges of indices. When the index is specified as above, the expression represents a bit-sequence which is the concatenation of all the indexed elements taken in the listed order. Essentially, this allows the application to access disjointed bit-sequences as a contiguous set of bits. The examples below illustrate this clearly.

The statement

```
b=A[3,5,7];
```

assigns to `b` the bot-sequence formed from the concatenation of the bit-sequences of array elements `A[3]`, `A[5]` and `A[7]`.

`A[1,3..5,2,9..8]` represents the bit-sequence `A[1]A[3]A[4]A[5]A[2]A[9]A[8]`.

The size of an array need not be specified at the time of declaration. BDPL provides a special notation to indicate that the number of elements in the array is not known as compile time and that it will be determined at run time. The declaration:

```
byte[*] ar1;
```

defines the array `ar1` whose size will be determined at run time.

When an array appears without an index (or a set of indices), it is assumed that the complete bit-sequence

of the array is addressed. For instance, `A = 0` sets the numerical value represented by the entire array of elements taken together to 0. Only one dimensional arrays are permitted

3.3 Structures

Structures (or structs) in BDPL are defined using the “struct “ keyword. Structs are the basic facility that BDPL provides in modeling the structure of a file. A struct represents an abstract entity that defines the layout of a new type which comprises basic types, arrays of defined types(basic types as well as structs) and structs, and models byte-by-byte the sequence of fields in the file being processed. When the contents of a file are read into a struct of a defined type, the number of bytes corresponding to the size of the struct is read from the file and interpreted as per the types of the contained fields or members. Once a struct is defined, it is reckoned a new type and can be used in operations exactly like a basic type.

The definition of a struct is done thus:

```
struct a
{  .... }x;
```

This defines a struct named “a”, which means “a” is a new type which can be used in the remainder of the BDPL program. The body of the struct, comprising valid declarations, appears between the curly braces.

The struct represents a sequence of elements of possibly heterogeneous types. The order of declarations that occur in the structure is very important. Therefore,

```
struct A
{
      int a;
      byte[10] b;
}_A;
```

is different from

```
struct A
{
      byte[10] b;
      int a;
}_A;
```

Every element of the struct is addressable by name. The name of the structure must be different from the names of its constituent elements. Elements of a structure are referenced using the '.' operator. For instance, the expression `_A.b` references the element named b in the structure named _A. The left operand of the '.' operator identifies the container, and the right operand identifies the contained. The name of the struct itself is used in conjunction with “type struct” to denote the type defined by the structure.

The bit sequence associated with the struct in memory is the concatenation of the bit-sequences of its individual elements.

3.4 Files

The principle objects of attention in BDPL programs are files, specially binary files. Files are special types in BDPL, and a file instance is created by a declaration comprising the keyword `file`. For each file that is read from or written into in a program, there may be an independent file instance. Once a file instance is created, it must be associated with content from a real file. This is done via a read statement. Also, once data corresponding to a file instance is ready, it can be committed to a real file. This is done via the write statement. Statements that operate on files are detailed later. This section deals with files and their properties.

An element that is declared to be of a file type may be perceived as a file buffer. Conceptually, this is equivalent to the following structure.

```
struct file {
      byte[*] data;
}_file;
```

File variables are declared as follows:

```
file-declaration:
      file filename-list
filename-list:
      filename
      filename-list
filename:
      identifier
```

Files, once declared, may be used in assignment statements, read, set and write statements. A file is considered to have elements that are typeless. Hence when assigning a file to a structure, the types of the structure elements dictate the interpretation of the raw data in the conceptual file buffer. Also, when assigning structures to files, the types of the structure's elements dictates the format of the bytes that are populated in the conceptual file buffer.

Properties can be attributed to any file variable that is declared. This is done using the set statement. One of the key properties of a file is its endian-ness. This may be specified as an attribute of the file once it is created.

4 Lvalues

An lvalue is an expression which refers to an object to which a value can be assigned. In other words, and lvalue is an expression that can occur on the left hand side of a BDPL assignment statement. Any identifier of a basic type, an array name, one or more indexed elements of an array, a struct type element, or a member of a struct may form an lvalue. The nature of type conversion that occurs for various lvalue expressions is detailed in the sections on assignment statements and type conversions.

5 Conversions

The operators in BDPL may, depending on the types of the operands, convert an operand from one type to another. All types conversions are implicit. The nature of type conversion or type promotion that is adopted for all operands is detailed in the table below and governed by the following rules.

1. All types other than bit[], byte[] and struct are regarded as types that possibly carry numeric values. Hence in all conversions involving these types, a best attempt is made to maintain the numeric value following conversion.
2. Bit-Order: In an array of bits, the left-most bit (index 0) is regarded as the most significant when all bits are taken together.
3. Endian-ness: In an array of bytes, the byte at index 0 is regarded as least significant when all bytes are taken together.
4. In an assignment expression, the type of the lvalue is held sacrosanct and the type of the right hand side is converted to the type of the left-hand side as per the type conversion table.
5. Float and double types are not regarded as raw bit-sequences. Hence,
    1. when converting from the floating point domain to the bit-sequence domain, the floating point number is first converted to a 32-bit integer. The rules from the conversion table are then applied to convert the integer to the desired type. The exceptions to this are conversions to bit array and byte array types.
    2. when converting from the bit-sequence domain to the floating point domain, rules from the conversion table are applied to convert the bit-sequence to a 32-bit integer. This integer is then cast to its floating point equivalent.
6. When converting a bit-sequence to an array of undefined size, with the with the whole array taken at once,
    1. the size of the array is taken to be the fieldsize if specified
    2. the size of the array is taken to be the size of the right-hand size expression rounded upwards to be a multiple of the size of the basic type from which the array is formed.
7. Expanding: When converting a bit-sequence to one of a larger size, the bit sequence is copied at the least-significant end of the target sequence, and sign-extension rules are applied if required.
8. Truncating: When converting a bit-sequence to one of a smaller size, the bit sequence is truncated to the required size by retaining the least significant end and chopping of the most significant bits that cannot be accommodated.
9. Sign extension is performed only when both the type from which conversion is being done and the type to which conversion is being done are signed.
10. <TBD> Add a clause to define when * makes an array infinite size and when it is 0 size. Example bit[*]

= bit[*]

		FROM							
		bit	byte	int	float	double	bit[]	byte[]	struct
TO	bit	Copy bit	Copy LSb	Copy LSb	float->int int->bit	double ->int int->bit	Copy LSb	Copy LSb of byte[0]	Copy next bit
	byte	Copy bit to LSb, NSE	Copy byte	Copy LSB	float->int int->byte	double -> int int -> byte	Copy 8 LSb, T, *, NSE	Copy LSB, *	Copy next byte
	int	Copy bit to LSb, NSE	Copy byte, SE	Copy bytes	C-style	C-style	Copy 32 LSb, T, *, E, NSE	Copy 4 LSB, T, *, E, NSE	Copy next 4 bytes, T, E
	float	bit->int int->float	byte->int int->float	C-style	Copy bytes	C-style	Copy 32 LSb, T, *	Copy 4 LSB, T, *, E	struct -> int int -> float
	double	bit->int int ->double	byte->int int -> double	C-style	C-style	Copy bytes	Copy 64 LSb, T, *	Copy 8 LSB, T, *, E	struct -> int int -> double
	bit[]	Copy bit to LSb, *	Copy to LSB, T, *	Copy to 32 LSb, T, *, E	Copy to 32 LSb, T, *	Copy to 64 LSb, T, *	Copy LSb to LSb, T, *, NSE	Copy LSb to LSb, T, *, NSE	Copy next bits, *
	byte[]	Copy bit to LSb of byte[0]	Copy to LSB, *	Copy to 4 LSB, T, E, *	Copy to 4 LSB, T, *	Copy to 8 LSB, T, *	Copy LSb to LSB, T, *, NSE	Copy LSB to LSB, T, *	Copy next bytes, *
	struct	Error	Error	Error	Error	Error	Copy bits, *	Copy bits, *	Type cast

Notation	Meaning
LSb	Least Significant Bit
LSB	Least Significant Byte
NSE	No Sign Extension
SE	Sign Extension as per rules for extension
*	Follow rules for handling arrays of undefined size
T	Follow rules for truncation and expansion
E	Follow rules for endian-ness

When the operands in an expression are of different types, the following rules are applied to promote one type to the other prior to performing the operation. Note that these rules do not apply to the assignment and push operators. In the case of the assignment and push operators, the type of the lvalue expression is held sacrosanct, and type conversion is applied to the expression on the right-hand side. To convert, the rules defined in the table above are used.

1. If one of the operands is a bit and the other is either a byte, int, float or double, the bit is promoted to the other type for purposes of the operation.

2. If one of the operands is a byte and the other is either an int, float or double the byte is promoted to the other type for purposes of the operation.

3. If one of the operands is an int and the other is either a float or a double, the int is promoted to the other type for purposes of the operation.

4. If one of the operands is a float and the other, a double, the float is promoted to a double.

Bit array and byte array operands may only be used in assignment statements and shift/rotate operations. Type promotion is not relevant for the latter. In the former case, standard type conversion rules are applied.

6 Expressions

Expressions are ordered by precedence starting from the highest precedence.

6.1 Primary Expressions

6.1.1 Identifier

```
primary-expr: identifier
```

An identifier is a primary expression. Each identifier must be declared to be of a basic or defined type.

6.1.2 Numeric Constant

A numeric constant may be an decimal, binary or hexadecimal number.

6.1.3 Parenthesized Expressions

```
expr: (expr)
```

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

6.1.4 Array expression

```
expr:       primary-expr[expr]
            primary-expr[expr-list]
expr-list:  expr
                  expr,expr-list
```

An array expression is also an expression. The methods of indexing the elements of the array is already detailed in section 3.2.

6.1.5 Dot Operator

primary-expr: primary-expr.member

A primary expression followed by a dot followed by the name of the member of a structure is a primary expression. The element `member` must be defined to be a field within the structure which is being referenced.

6.2 Range Operator

range: expr..expr

The range operator taken BDPL expressions as operands, but does not return an expression. It returns a list of elements starting from the numerical value of the expression on the left, ending at the numerical value of the expression on the right and including both bounding elements.

6.3 Unary Operators

Associativity : All operators described in this section are right associative

6.3.1 Negation operators

```
expr :      ~ expr
            ! expr
```

There are two negation operators, Bitwise negation (~) and Logical negation (!). These are both unary operators. The logical negation operator logically negates expressions. Ie, converts a non-zero value to zero and a zero value to 1 . The bitwise negation operator does a bitwise flipping of bits so that ~0b1101 becomes 0b0010.

6.3.2 Offset Operator

There are 3 offset operators, Position operator ($) , Size operator ($#) and Byte-size operator (#).

Position operator

The position operator ($) can operate a struct or an array. It indicates the current position in the struct or array where next element will be written (in a push operation where the operand was a lvalue) or where the next element will be read from (in case the operand was a rvalue). This value can be modified by the programmer (i.e be used as an lvalue). Conceptually, we can view this value as being analogous to a file offset. Thus,

```
$a=10;
```

will be a valid expression.

Size Operator

The size operator ($#) returns the number of elements contained in the operand. This can operate only on

arrays. This cannot be used as an lvalue.

Byte-size Operator

The Byte-size operator returns the size of an operand in number of bytes. The value returned is the current size of the operand (0 in case of uninitialized * arrays). Note that size may change if elements are pushed to the operand. Size of basic types is fixed and will not change. The resulting value cannot be used as a lvalue.

6.4 Multiplicative Operators

```
expr : expr * expr;
```

Associativity: Left to right

The '*' operator is a binary operator that indicates multiplication. The result of a multiplication expression is the product of the operands. Operands may only be of basic types. Type promotion rules are applied as detailed in the section on type conversion when the two operands are of different types.

```
expr : expr / expr;
```

Associativity: Left to right

The '/' operator is a binary operator that indicates division. The result of a division expression is the quotient. Operands may only be of basic types. Type promotion rules are applied as detailed in the section on type conversion when the two operands are of different types.

```
expr : expr % expr;
```

Associativity: Left to right

The '%' operator is a binary operator that indicates modulus. The result of a modulus expression is the remainder from dividing the left operand by the right operand. Operands may only be of basic integer types. Type promotion rules are applied as detailed in the section on type conversion when the two operands are of different types.

6.5 Additive Operators

```
expr : expr + expr;
```

Associativity: Left to right

The '+' operator is a binary operator that indicates addition. The result of an addition expression is the sum of the operands. Operands may only be of basic types. Type promotion rules are applied as detailed in the section on type conversion when the two operands are of different types.

```
expr : expr - expr;
```

Associativity: Left to right

The '-' operator is a binary operator that indicates subtraction. The result of a subtraction expression is the difference of the operands. Operands may only be of basic types. Type promotion rules are applied as detailed in the section on type conversion when the two operands are of different types.

6.6 Shift Operators

6.6.1 Bitwise shift Operators

```
expr : expr << expr;
expr : expr >> expr;
```

Associativity: Left to right

The '<<' and '>>' operators are logical shift operators. The left operand may only be of a basic integer type or an array of these types. The second operand must be a positive integer type. The result of the expression with '<<' is the bit-sequence of the first operand is rotated left as many times as indicated by the value of the right operand. Vacated bit positions are filled with 0. The result of the expression with '>>' is the bit-sequence of the first operand is rotated right as many times as indicated by the value of the right operand. Vacated bit positions are filled with 0.

6.6.2 Arithmetic Shift Operators

```
expr : expr <<< expr;
expr : expr >>> expr;
```

Associativity: Left to right

The '<<<' and '>>>' operators are arithmetic shift operators. The left operand may only be of a basic integer type or an array of these types. The second operand must be a positive integer type. The result of the expression with '<<<' is the bit-sequence of the first operand is rotated left as many times as indicated by the value of the right operand. Vacated bit positions are filled with 0. The result of the expression with '>>>' is the bit-sequence of the first operand is rotated right as many times as indicated by the value of the right operand. Vacated bit positions are filled with the most significant bit taken as sign. Essentially, the left operand is treated as a number in 2's complement form.

6.6.3 Rotation Operators

```
expr : expr <-< expr;
expr : expr >-> expr;
```

Associativity: Left to right

The '<-<' and '>->' operators are left and right rotation operators. The left operand may only be of a basic integer type or an array of these types. The second operand must be a positive integer type. The result of the expression with '<-<' is the bit-sequence of the first operand is rotated left as many times as indicated by the

value of the right operand. Vacated bit positions are filled with the bits that are rotated out of the left of the bit-sequence. The result of the expression with '>->' is the bit-sequence of the first operand is rotated right as many times as indicated by the value of the right operand. Vacated bit positions are filled with the bits that are rotated out of the right of the bit-sequence.

6.7 Comparison Operators

Associativity is not defined for comparison operators since expressions with these operators cannot be cascaded.

6.7.1 Greater than

```
expr : expr > expr;
```

6.7.2 Less than

```
expr : expr < expr;
```

6.7.3 Greater than or equal to

```
expr : expr >= expr;
```

6.7.4 Less than or equal to

```
expr : expr <= expr;
```

The result of these expressions is a 1 if the boolean result of the expression is true and 0 otherwise. The operands must be of a basic type.

6.8 Equality Operators

6.8.1 Equal to

```
expr : expr == expr;
```

6.8.2 Not Equal to

```
expr : expr != expr;
```

The result of these expressions is a 1 if the boolean result of the expression is true and 0 otherwise. The operands must be of a basic type.

6.9 Bitwise AND Operator

```
expr : expr & expr
```

Associativity: Left to right

The `&` operator performs a bitwise logical 'and' or of the operands. The operands are not required to be of the same type. However, both operands are required to be of a basic type and must have equal lengths in

bits.

6.10 Bitwise XOR Operator

expr : expr ^ expr

Associativity: Left to right

The ^ operator performs a bitwise 'exclusive or' of the operands. The operands are not required to be of the same type. However, both operands are required to be of a basic type and must have equal lengths in bits.

6.11 Bitwise OR Operator

expr : expr | expr

Associativity: Left to right

The | operator performs a bitwise 'inclusive or' of the operands. The operands are not required to be of the same type. However, both operands are required to be of a basic type and must have equal lengths in bits.

6.12 Logical AND Operator

expr : expr && expr

Associativity: Left to right

The && operator returns 1 if both of its operands have a non-zero value, and 0 otherwise. Each of the operands must be of a basic type. However, both operands are not required to be of the same type. The second operand is not evaluated is the value of the first operand is zero.

6.13 Logical OR Operator

expr : expr || expr

Associativity: Left to right

The || operator returns 1 if either of its operands has a non-zero value, and 0 otherwise. Each of the two operands must be of a basic type. However, both operands are not required to be of the same type The second operand is not evaluated if the value of the first operand is non-zero.

6.14 Assignment Operator

object = expr

The assignment operator denotes that the variable on the LHS will hold the value of the expression on the RHS. Various type conversion rules will be applied to evaluate what exactly should happen when an assignment is executed. Indeed, there are many consideration of endian-ness, loss of precession, unmatched types etc. which have been address in section 5.

6.15 Push Operator

object <- expr

The push operator takes the expression on the right and streams the value to the lvalue. It is like an assignment, but non- destructive. The current value of the object is not overwritten. Instead, the data is made available to the object for appending.

7 Declarations

Declarations are used to specify types for identifiers. Whether declarations reserve memory for identifiers is dependent upon the implementation of the compiler. However, once declared, an identifier refers to an object that is of the indicated type and which is available till the end of the program's execution. Declarations take the following form:

declaration:

type-specifier $array_{opt}$ *declarator-list* $valid\text{-}check_{opt}$ $optional\text{-}check_{opt}$*;*

Each declaration statement may comprise at most one type specifier. The type specifier field may be followed by the array operator, which makes every declared element an array of the specified type. The specification of the type is followed by a list of identifiers which are named elements of the said type. This list may be followed by two optional clauses referred to in the grammar above as valid-check and optional-check. These are detailed in later sections. Each declaration statement must be terminated with a semi-colon. A declaration statement may not appear within a statement-block.

The declarator-list is a comma separated list of declarators. The comma-separated list of n identifiers is a concise representation which is equivalent to the declaration of n identifiers one after the other via a sequence of declaration statements. This equivalence should be borne in mind when declaring a comma-separated list of identifiers within a struct type.

declarator-list:

declarator

declarator, declarator-list

declarator:

identifier $initializer_{opt}$ $fieldsize_{opt}$

(declarator)

7.1 Type Specifiers

The type specifier for a declaration may be any of the basic types available in BDPL, a structure definition, or a reference to a structure that is completely defined earlier in the BDPL source file.

type-specifiers:

int

bit

```
        byte
        float
        double
        struct-defn
        struct-type
    struct-defn:
        struct name {declarations}
    name:
        identifier
    struct-type:
        type struct name
    declarations:
        declaration
        declaration declarations
```

Each defined structure must be assigned a name, which identifies the type just created by virtue of the definition. The struct-defn must be accompanied by the declaration of an element of that struct type. The struct definition may contain any number of valid declaration statements within it, including the definition of other struct types, as detailed in 3.3. The struct-type construct must be employed when creating elements of a struct which has previously been defined. The name must be of a struct defined earlier.

7.2 Array Types

Array types may be created by following the type specification for an identifier by the array operator along with a specification of the size of the array. The size of the array may be specified in the form of any valid expression. The size of the array may also be specified as unknown, in which case, a '*' character should be used in the size field. This creates a free-size array.

```
array:
    [expr]
    [*]
```

7.3 Identifier Properties

The each identifier may be qualified by an additional specification of its environment and properties. This is in the form of an optional initializer and an optional field-size. Currently no strings are defined.

```
initializer:
    ("string" => "string")
    ("string" => "string"),initializer
```

The field-size construct may to be used to specify circumstances wherein the application would like that information be stored in a data structure of size other than in the file being processed. For an array of

unspecified size, the field-size indicates the maximum size in bits of the array element. For elements where size is not variable, the field-size indicates the size of the representation of that field in a file should the field be written to or read from a file.

```
field-size:
      fieldsize expr
```

7.4 Valid Value Checks

For each identifier which is declared, constraints may be imposed on the value, set of values or range of values that the variable may take. The variable is checked for a valid value assignments whenever an expression with the identifier as lvalue is evaluated. The valid value check takes the following form:

```
valid-check:
      valid {value-specifier} ok-block_opt nok-block_opt
ok-block:
      ok {statement-block-1}
nok-block
      nok {statement-block-2}
```

The value to be assigned to the identifier is evaluated and validated against the value-specifier. If this comparison indicates that the assigned value is valid, the statement block-1 is executed, if present. If the comparison fails, statement block-2 is executed if present. The ok and nok constructs are both optional and either or both of these may be skipped. The ok keyword and statement-block1 must either both be present, or neither. The nok keyword and statement-block2 must either both be present, or neither.

7.5 Optional Field Checks

When an element is being declared, an additional condition may be specified to allow the presence or absence of a field based on a condition. This should be done using the optional-on construct, which requires the specification of an expression. The element is considered present if the expression evaluates to a non-zero value, and absent otherwise. Should an element be marked absent, its field-size is no more relevant.

```
optional-check:
      optional on (expr)
```

8 Statements

Statements are executed in the order in which they are written in the program, except when the order is explicitly changed by the program itself, using the statement constructs indicated below.

8.1 Expressions

Construction:

```
expression;
```

An expression terminated by a semi-colon constitutes a basic statement.

8.2 Statement Blocks

Construction:

```
{statement-block}
      statement-block:
            statement
            statement statement-block
```

A statement block is a collection of statements that can be used together in lieu of a single statement.

8.3 Conditional Statements

Construction:

```
if (expression) statement else statement
if (expression) statement
```

The conditional statement can occur either with or without an else section. If the expression evaluates to a non-zero value, the first statement is executed and if the expression evaluates to zero, the second statement is executed, if present. The else is connected with the last encountered else-less if.

8.4 For Loop

Construction:

$$\texttt{for(expression-1}_{opt}\texttt{;expression-2}_{opt}\texttt{;expression-3}_{opt}\texttt{) statement}$$

The loop statement is used to execute a statement several times depending upon a condition. In the said syntax, expression-1 specifies an initialization for the loop, expression-2 specifies a condition which is evaluated prior to each iteration and expression-3 specifies an incrementing action that is performed after each iteration and before the evaluation of expression-2. Any or all of the expressions may be omitted.

8.5 Continue

Construction:

```
continue
```

This statement must be used within the for loop. It causes control to skip over the remaining statements in the enclosing loop statement block and pass to the end of the loop.

8.6 Break

Construction:

```
break
```

This statement must be used within the for loop. It causes control to break out of the loop and pass to the statement immediately following the loop statement.

8.7 Set

Construction:

```
set(string => string,identifier)
```

This statement allows attributes to be set for files. The first element should be a pair of strings separated by the => symbol. The first of these strings identifies the attribute to be set, and the second, the value for that attribute. The only pre-defined attribute string for files is “endian”, which defines the endian-ness of the file. Its value may either be “big” or “little”. The default value is “little”. The second element with braces must be the name of the file element for which the attribute is being set. This file element must already have been declared through a file declaration statement.

8.8 Read

Construction:

```
read(string,identifier)
```

This statement allows the contents of a file to be read into a conceptual file buffer. The path to the file whose contents have to be read is supplied as a string, and forms the first element of the read statement within braces. The file type variable which represents the buffer into which the file must be read is the second element. This file element must already have been declared through a file declaration statement.

8.9 Print

Construction:

```
print(string)
```

This statement allows the programmer to write message to the console from within the BDPL program. The message to be written must be a string enclosed within braces that follow the print keyword.

8.10 Exit

Construction:

```
exit(int)
```

The execution of this statement stops execution of the current program and returns. The argument (integer) is returned by exit to the caller.

8.11 Write

Construction:

`write(string,identifier)`

This statement allows the contents of a conceptual file buffer to be written to a file. The path to the file whose contents have to be created is supplied as a string, and forms the first element of the write statement within braces. The file type variable which represents the buffer from which the file must be written is the second element. This file element must already have been declared through a file declaration statement.

9 Scope and lifetime

Every element is visible in the portion of the BDPL source file following its declaration. Every element has the lifetime of the program.

10 Examples

The following examples demonstrate some real applications of the language.

10.1 MPEG Files

The following program parses Mpeg files :

```
struct trp_file
{
    struct transport_packet
    {
       byte magic
         valid {0x47} nok {print("Sync Byte Error");};
       bit[1] error;
       bit[1] priority;
       bit[1] something;
       bit[13] pid;
       bit[2] adaptation_field_control;
       bit[2] transport_scrambling_control;
       bit[4] continuity_counter
         valid {((tp[$#tp-1].continuity_counter + 1)%16)}
         nok
         {
           print("Continuity counter error ");
         };
       byte[184] payload;
    }[*] tp;
}trp_f;
```

```
file f1;
read("hello-world.mpeg",f1);
trp_f = f1;
print("mpeg file parsing complete");
```

The program starts out by defining the layout of the mpeg packet. The main program merely opens the file and populate elements into the struct "`trp_file`". How does this work ? Mpeg files comprise of packets. We have defined the struct "transport_packet" to mimic this. Then we define the structure `trp_file` which contains an array of transport_packets . The size of this array is undefined initially. Depending on the amount of data available, the size of the transport_packet array (tp) will increase. So when we execute the statement

```
trp_f = f1;
```

What actually happens is that bytes are read from the file and populated into the struct `trp_file`. This in turn streams the bytes to the next element of the array tp (because thats all the `trp_file` contains. Thus, packets get read incrementally form the file. Notice the line

```
valid {(tp[$#tp-1].continuity_ counter + 1)%16)}
```

tp refers to the array of packets. `$#tp` means the number of elements in tp . Thus, `$#tp -1` is the last element that was read. So , the code actually means that the continuity counter of the current packet is dependent on the continuity counter of the last packet. If the value of the continuity counter does not match the expected result, then an error will be flagged.

10.2 TIFF files

Here's a program to model the structure of a tiff file. Tiff files comprise of a header and a linked list of image file directories. The position of where the next image file directory starts is given in the current directory. The operation of reading the directory involves jumping to the new location and reading the directory from there. We will demonstrate how this can be easily achieved in BDPL.

```
struct image_file_directory
{
  byte[2] numEntries;
  struct image_file_directory_entry
  {
      byte[2] tag;
      byte[2] typefield;
      byte[4] count;
```

```
      byte[4] value;
  }ifdentry;
  byte[4] nextifdpointer;
}[*] ifd;

read("hello-world.tiff",f1);
tiff.header <- f1;
$f1 = tiff.header.ifd_pointer;

type struct image_file_directory temp_idf;
for(;1;)
{
  temp_idf <- f1 ;
  idf <- temp_idf;
  if(temp_idf.nextifdpointer > 0 && temp_idf.nextifdpointer < $#f1 )
  { $f1 = temp_idf.nextifdpointer; }
  else { break; }
}
print("tiff file parsing complete")
```

The program defines 2 structures image_file_header and image_file_directory which describes the layout of the 2 entities. The main part of the program reads the header from the file and gets the value of the initial offset where the first image_file_directory resides. The offset of the file is set to this value with the statement :

```
$(f1) = ifh.ifd_pointer;
```

$ is a unary operator which returns the position inside the current structure. This value can be modified to jump to a new position.

Notice that the we have declared an array of image_file_directory structures. We intend to read structures one by one from the file, push it into the array and jump to the new position where the next directory structure will be found. The Algorithm is modelled exactly this way.

Lines :

```
  temp_idf <- f1 ;
  idf <- temp_idf;
```

get the next image_file_directory entry and push it into the array. Note here the "push" operator <- . This operators reads a bitstream from the rhs and pushes it onto the LHS. This operator is non destructive.

The next lines :

```
  if(temp_idf.nextifdpointer > 0 && temp_idf.nextifdpointer < $#(f1) )
```

```
{ $(f1) = temp_idf.nextifdpointer; }
else { break; }
```

Check if the end of the linked list has been reached and break accordingly.

11 Appendix

11.1 Appendix A – Operators, precedence

Operator	Name	Precedence	Associativity	Sample usage	Notes
[]	Aray index	1			
,	List separator	1			
$	Position	1			
$#	Size	1			
#	Byte size	1			
()		1			
..	Range	1			
~	Bitwise Negation	2			
!	Logical Negation	2			
*	Multiplication	3			
/	Division	3			
%	Modulo	3			
+	Addition	4			
_	Substraction	4			
<<	Logical Left shift	5			
<<<	Arithmetic left shift	5			
>>	Logical Right Shift	5			
>>>	Arithmetic right shift	5			

>->	Right rotate	5			
<-<	Left Rotate	5			
<	Less than	6			
>	Greater than	6			
<=	Less than equals	6			
>=	Greater than equals	6			
==	Equals	7			
!=	Not equals	7			
&	Bitwise And	8			
^	Bitwise Xor	8			
|	Bitwise Or	8			
&&	Logical And	9			
||	Logical Or	9			
? :	Conditional	10			
=	Assignment	11			
<-	Push	11			
=>		11			