24. Practical: Parsing Binary Files

In this chapter I'll show you how to build a library that you can use to write code for reading and writing binary files. You'll use this library in Chapter 25 to write a parser for ID3 tags, the mechanism used to store metadata such as artist and album names in MP3 files. This library is also an example of how to use macros to extend the language with new constructs, turning it into a special-purpose language for solving a particular problem, in this case reading and writing binary data. Because you'll develop the library a bit at a time, including several partial versions, it may seem you're writing a lot of code. But when all is said and done, the whole library is fewer than 150 lines of code, and the longest macro is only 20 lines long.

Binary Files

At a sufficiently low level of abstraction, all files are "binary" in the sense that they just contain a bunch of numbers encoded in binary form. However, it's customary to distinguish between *text files*, where all the numbers can be interpreted as characters representing human-readable text, and *binary files*, which contain data that, if interpreted as characters, yields nonprintable characters.[1]

Binary file formats are usually designed to be both compact and efficient to parse--that's their main advantage over text-based formats. To meet both those criteria, they're usually composed of on-disk structures that are easily mapped to data structures that a program might use to represent the same data in memory.[2]

The library will give you an easy way to define the mapping between the on-disk structures defined by a binary file format and in-memory Lisp objects. Using the library, it should be easy to write a program that can read a binary file, translating it into Lisp objects that you can manipulate, and then write back out to another properly formatted binary file.

Binary Format Basics

The starting point for reading and writing binary files is to open the file for reading or writing individual bytes. As I discussed in Chapter 14, both **`OPEN`** and **`WITH-OPEN-FILE`** accept a keyword argument, `:element-type`, that controls the basic unit of transfer for the stream. When you're dealing with binary files, you'll specify `(unsigned-byte 8)`. An input stream opened with such an `:element-type` will return an integer between 0 and 255 each time it's

passed to **READ-BYTE**. Conversely, you can write bytes to an `(unsigned-byte 8)` output stream by passing numbers between 0 and 255 to **WRITE-BYTE**.

Above the level of individual bytes, most binary formats use a smallish number of primitive data types--numbers encoded in various ways, textual strings, bit fields, and so on--which are then composed into more complex structures. So your first task is to define a framework for writing code to read and write the primitive data types used by a given binary format.

To take a simple example, suppose you're dealing with a binary format that uses an unsigned 16-bit integer as a primitive data type. To read such an integer, you need to read the two bytes and then combine them into a single number by multiplying one byte by 256, a.k.a. 2^8, and adding it to the other byte. For instance, assuming the binary format specifies that such 16-bit quantities are stored in *big-endian*[3] form, with the most significant byte first, you can read such a number with this function:

```
(defun read-u2 (in)
  (+ (* (read-byte in) 256) (read-byte in)))
```

However, Common Lisp provides a more convenient way to perform this kind of bit twiddling. The function **LDB**, whose name stands for load byte, can be used to extract and set (with **SETF**) any number of contiguous bits from an integer.[4] The number of bits and their position within the integer is specified with a *byte specifier* created with the **BYTE** function. **BYTE** takes two arguments, the number of bits to extract (or set) and the position of the rightmost bit where the least significant bit is at position zero. **LDB** takes a byte specifier and the integer from which to extract the bits and returns the positive integer represented by the extracted bits. Thus, you can extract the least significant octet of an integer like this:

```
(ldb (byte 8 0) #xabcd) ==> 205 ; 205 is #xcd
```

To get the next octet, you'd use a byte specifier of `(byte 8 8)` like this:

```
(ldb (byte 8 8) #xabcd) ==> 171 ; 171 is #xab
```

You can use **LDB** with **SETF** to set the specified bits of an integer stored in a **SETF**able place.

```
CL-USER> (defvar *num* 0)
*NUM*
CL-USER> (setf (ldb (byte 8 0) *num*) 128)
128
CL-USER> *num*
128
CL-USER> (setf (ldb (byte 8 8) *num*) 255)
255
CL-USER> *num*
65408
```

Thus, you can also write `read-u2` like this:[5]

```
(defun read-u2 (in)
  (let ((u2 0))
    (setf (ldb (byte 8 8) u2) (read-byte in))
```

```
    (setf (ldb (byte 8 0) u2) (read-byte in))
    u2))
```

To write a number out as a 16-bit integer, you need to extract the individual 8-bit bytes and write them one at a time. To extract the individual bytes, you just need to use **`LDB`** with the same byte specifiers.

```
(defun write-u2 (out value)
  (write-byte (ldb (byte 8 8) value) out)
  (write-byte (ldb (byte 8 0) value) out))
```

Of course, you can also encode integers in many other ways--with different numbers of bytes, with different endianness, and in signed and unsigned format.

Strings in Binary Files

Textual strings are another kind of primitive data type you'll find in many binary formats. When you read files one byte at a time, you can't read and write strings directly--you need to decode and encode them one byte at a time, just as you do with binary-encoded numbers. And just as you can encode an integer in several ways, you can encode a string in many ways. To start with, the binary format must specify how individual characters are encoded.

To translate bytes to characters, you need to know both what character *code* and what character *encoding* you're using. A character code defines a mapping from positive integers to characters. Each number in the mapping is called a *code point*. For instance, ASCII is a character code that maps the numbers from 0-127 to particular characters used in the Latin alphabet. A character encoding, on the other hand, defines how the code points are represented as a sequence of bytes in a byte-oriented medium such as a file. For codes that use eight or fewer bits, such as ASCII and ISO-8859-1, the encoding is trivial--each numeric value is encoded as a single byte.

Nearly as straightforward are pure double-byte encodings, such as UCS-2, which map between 16-bit values and characters. The only reason double-byte encodings can be more complex than single-byte encodings is that you may also need to know whether the 16-bit values are supposed to be encoded in big-endian or little-endian format.

Variable-width encodings use different numbers of octets for different numeric values, making them more complex but allowing them to be more compact in many cases. For instance, UTF-8, an encoding designed for use with the Unicode character code, uses a single octet to encode the values 0-127 while using up to four octets to encode values up to 1,114,111.[6]

Since the code points from 0-127 map to the same characters in Unicode as they do in ASCII, a UTF-8 encoding of text consisting only of characters also in ASCII is the same as the ASCII encoding. On the other hand, texts consisting mostly of characters requiring four bytes in UTF-8 could be more compactly encoded in a straight double-byte encoding.

Common Lisp provides two functions for translating between numeric character codes and character objects: **CODE-CHAR**, which takes an numeric code and returns as a character, and **CHAR-CODE**, which takes a character and returns its numeric code. The language standard doesn't specify what character encoding an implementation must use, so there's no guarantee you can represent every character that can possibly be encoded in a given file format as a Lisp character. However, almost all contemporary Common Lisp implementations use ASCII, ISO-8859-1, or Unicode as their native character code. Because Unicode is a superset ofISO-8859-1, which is in turn a superset of ASCII, if you're using a Unicode Lisp, **CODE-CHAR** and **CHAR-CODE** can be used directly for translating any of those three character codes.[7]

In addition to specifying a character encoding, a string encoding must also specify how to encode the length of the string. Three techniques are typically used in binary file formats.

The simplest is to not encode it but to let it be implicit in the position of the string in some larger structure: a particular element of a file may always be a string of a certain length, or a string may be the last element of a variable-length data structure whose overall size determines how many bytes are left to read as string data. Both these techniques are used in ID3 tags, as you'll see in the next chapter.

The other two techniques can be used to encode variable-length strings without relying on context. One is to encode the length of the string followed by the character data--the parser reads an integer value (in some specified integer format) and then reads that number of characters. Another is to write the character data followed by a delimiter that can't appear in the string such as a null character.

The different representations have different advantages and disadvantages, but when you're dealing with already specified binary formats, you won't have any control over which encoding is used. However, none of the encodings is particularly more difficult to read and write than any other. Here, as an example, is a function that reads a null-terminated ASCII string, assuming your Lisp implementation uses ASCII or one of its supersets such as ISO-8859-1 or full Unicode as its native character encoding:

```
(defconstant +null+ (code-char 0))

(defun read-null-terminated-ascii (in)
  (with-output-to-string (s)
    (loop for char = (code-char (read-byte in))
          until (char= char +null+) do (write-char char s))))
```

The **WITH-OUTPUT-TO-STRING** macro, which I mentioned in Chapter 14, is an easy way to build up a string when you don't know how long it'll be. It creates a **STRING-STREAM** and binds it to the variable name specified, `s` in this case. All characters written to the stream are collected into a string, which is then returned as the value of the **WITH-OUTPUT-TO-STRING** form.

To write a string back out, you just need to translate the characters back to numeric values that can be written with **`WRITE-BYTE`** and then write the null terminator after the string contents.

```
(defun write-null-terminated-ascii (string out)
  (loop for char across string
        do (write-byte (char-code char) out))
  (write-byte (char-code +null+) out))
```

As these examples show, the main intellectual challenge--such as it is--of reading and writing primitive elements of binary files is understanding how exactly to interpret the bytes that appear in a file and to map them to Lisp data types. If a binary file format is well specified, this should be a straightforward proposition. Actually writing functions to read and write a particular encoding is, as they say, a simple matter of programming.

Now you can turn to the issue of reading and writing more complex on-disk structures and how to map them to Lisp objects.

Composite Structures

Since binary formats are usually used to represent data in a way that makes it easy to map to in-memory data structures, it should come as no surprise that composite on-disk structures are usually defined in ways similar to the way programming languages define in-memory structures. Usually a composite on-disk structure will consist of a number of named parts, each of which is itself either a primitive type such as a number or a string, another composite structure, or possibly a collection of such values.

For instance, an ID3 tag defined in the 2.2 version of the specification consists of a header made up of a three-character ISO-8859-1 string, which is always "ID3"; two one-byte unsigned integers that specify the major version and revision of the specification; eight bits worth of boolean flags; and four bytes that encode the size of the tag in an encoding particular to the ID3 specification. Following the header is a list of *frames*, each of which has its own internal structure. After the frames are as many null bytes as are necessary to pad the tag out to the size specified in the header.

If you look at the world through the lens of object orientation, composite structures look a lot like classes. For instance, you could write a class to represent an ID3 tag.

```
(defclass id3-tag ()
  ((identifier    :initarg :identifier    :accessor identifier)
   (major-version :initarg :major-version :accessor major-version)
   (revision      :initarg :revision      :accessor revision)
   (flags         :initarg :flags         :accessor flags)
   (size          :initarg :size          :accessor size)
   (frames        :initarg :frames        :accessor frames)))
```

An instance of this class would make a perfect repository to hold the data needed to represent an ID3 tag. You could then write functions to read and write instances of this class. For example,

assuming the existence of certain other functions for reading the appropriate primitive data types, a `read-id3-tag` function might look like this:

```
(defun read-id3-tag (in)
  (let ((tag (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) tag
      (setf identifier    (read-iso-8859-1-string in :length 3))
      (setf major-version (read-u1 in))
      (setf revision      (read-u1 in))
      (setf flags         (read-u1 in))
      (setf size          (read-id3-encoded-size in))
      (setf frames        (read-id3-frames in :tag-size size)))
    tag))
```

The `write-id3-tag` function would be structured similarly--you'd use the appropriate `write-*` functions to write out the values stored in the slots of the `id3-tag` object.

It's not hard to see how you could write the appropriate classes to represent all the composite data structures in a specification along with `read-foo` and `write-foo` functions for each class and for necessary primitive types. But it's also easy to tell that all the reading and writing functions are going to be pretty similar, differing only in the specifics of what types they read and the names of the slots they store them in. It's particularly irksome when you consider that in the ID3 specification it takes about four lines of text to specify the structure of an ID3 tag, while you've already written eighteen lines of code and haven't even written `write-id3-tag` yet.

What you'd really like is a way to describe the structure of something like an ID3 tag in a form that's as compressed as the specification's pseudocode yet that can also be expanded into code that defines the `id3-tag` class *and* the functions that translate between bytes on disk and instances of the class. Sounds like a job for a macro.

Designing the Macros

Since you already have a rough idea what code your macros will need to generate, the next step, according to the process for writing a macro I outlined in Chapter 8, is to switch perspectives and think about what a call to the macro should look like. Since the goal is to be able to write something as compressed as the pseudocode in the ID3 specification, you can start there. The header of an ID3 tag is specified like this:

```
ID3/file identifier      "ID3"
ID3 version              $02 00
ID3 flags                %xx000000
ID3 size             4 * %0xxxxxxx
```

In the notation of the specification, this means the "file identifier" slot of an ID3 tag is the string "ID3" in ISO-8859-1 encoding. The version consists of two bytes, the first of which--for this version of the specification--has the value 2 and the second of which--again for this version of the specification--is 0. The flags slot is eight bits, of which all but the first two are 0, and the size consists of four bytes, each of which has a 0 in the most significant bit.

Some information isn't captured by this pseudocode. For instance, exactly how the four bytes that encode the size are to be interpreted is described in a few lines of prose. Likewise, the spec describes in prose how the frame and subsequent padding is stored after this header. But most of what you need to know to be able to write code to read and write an ID3 tag is specified by this pseudocode. Thus, you ought to be able to write an s-expression version of this pseudocode and have it expanded into the class and function definitions you'd otherwise have to write by hand--something, perhaps, like this:

```
(define-binary-class id3-tag
  ((file-identifier (iso-8859-1-string :length 3))
   (major-version   u1)
   (revision        u1)
   (flags           u1)
   (size            id3-tag-size)
   (frames          (id3-frames :tag-size size))))
```

The basic idea is that this form defines a class `id3-tag` similar to the way you could with **`DEFCLASS`**, but instead of specifying things such as `:initarg` and `:accessors`, each slot specification consists of the name of the slot--`file-identifier`, `major-version`, and so on--and information about how that slot is represented on disk. Since this is just a bit of fantasizing, you don't have to worry about exactly how the macro `define-binary-class` will know what to do with expressions such as `(iso-8859-1-string :length 3)`, `u1`, `id3-tag-size`, and `(id3-frames :tag-size size)`; as long as each expression contains the information necessary to know how to read and write a particular data encoding, you should be okay.

Making the Dream a Reality

Okay, enough fantasizing about good-looking code; now you need to get to work writing `define-binary-class`--writing the code that will turn that concise expression of what an ID3 tag looks like into code that can represent one in memory, read one off disk, and write it back out.

To start with, you should define a package for this library. Here's the package file that comes with the version you can download from the book's Web site:

```
(in-package :cl-user)

(defpackage :com.gigamonkeys.binary-data
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :define-binary-class
           :define-tagged-binary-class
           :define-binary-type
           :read-value
           :write-value
           :*in-progress-objects*
           :parent-of-type
           :current-binary-object
           :+null+))
```

The COM.GIGAMONKEYS.MACRO-UTILITIES package contains the with-gensyms and once-only macros from Chapter 8.

Since you already have a handwritten version of the code you want to generate, it shouldn't be too hard to write such a macro. Just take it in small pieces, starting with a version of define-binary-class that generates just the **DEFCLASS** form.

If you look back at the define-binary-class form, you'll see that it takes two arguments, the name id3-tag and a list of slot specifiers, each of which is itself a two-item list. From those pieces you need to build the appropriate **DEFCLASS** form. Clearly, the biggest difference between the define-binary-class form and a proper **DEFCLASS** form is in the slot specifiers. A single slot specifier from define-binary-class looks something like this:

```
(major-version u1)
```

But that's not a legal slot specifier for a **DEFCLASS**. Instead, you need something like this:

```
(major-version :initarg :major-version :accessor major-version)
```

Easy enough. First define a simple function to translate a symbol to the corresponding keyword symbol.

```
(defun as-keyword (sym) (intern (string sym) :keyword))
```

Now define a function that takes a define-binary-class slot specifier and returns a **DEFCLASS** slot specifier.

```
(defun slot->defclass-slot (spec)
  (let ((name (first spec)))
    `(,name :initarg ,(as-keyword name) :accessor ,name)))
```

You can test this function at the REPL after switching to your new package with a call to **IN-PACKAGE**.

```
BINARY-DATA> (slot->defclass-slot '(major-version u1))
(MAJOR-VERSION :INITARG :MAJOR-VERSION :ACCESSOR MAJOR-VERSION)
```

Looks good. Now the first version of define-binary-class is trivial.

```
(defmacro define-binary-class (name slots)
  `(defclass ,name ()
     ,(mapcar #'slot->defclass-slot slots)))
```

This is simple template-style macro--define-binary-class generates a **DEFCLASS** form by interpolating the name of the class and a list of slot specifiers constructed by applying slot->defclass-slot to each element of the list of slots specifiers from the define-binary-class form.

To see exactly what code this macro generates, you can evaluate this expression at the REPL.

```
(macroexpand-1 '(define-binary-class id3-tag
  ((identifier      (iso-8859-1-string :length 3))
   (major-version   u1)
   (revision        u1)
   (flags           u1)
   (size            id3-tag-size)
   (frames          (id3-frames :tag-size size)))))
```

The result, slightly reformatted here for better readability, should look familiar since it's exactly the class definition you wrote by hand earlier:

```
(defclass id3-tag ()
  ((identifier      :initarg :identifier    :accessor identifier)
   (major-version   :initarg :major-version :accessor major-version)
   (revision        :initarg :revision      :accessor revision)
   (flags           :initarg :flags         :accessor flags)
   (size            :initarg :size          :accessor size)
   (frames          :initarg :frames        :accessor frames)))
```

Reading Binary Objects

Next you need to make `define-binary-class` also generate a function that can read an instance of the new class. Looking back at the `read-id3-tag` function you wrote before, this seems a bit trickier, as the `read-id3-tag` wasn't quite so regular--to read each slot's value, you had to call a different function. Not to mention, the name of the function, `read-id3-tag`, while derived from the name of the class you're defining, isn't one of the arguments to `define-binary-class` and thus isn't available to be interpolated into a template the way the class name was.

You could deal with both of those problems by devising and following a naming convention so the macro can figure out the name of the function to call based on the name of the type in the slot specifier. However, this would require `define-binary-class` to generate the name `read-id3-tag`, which is possible but a bad idea. Macros that create global definitions should generally use only names passed to them by their callers; macros that generate names under the covers can cause hard-to-predict--and hard-to-debug--name conflicts when the generated names happen to be the same as names used elsewhere.[8]

You can avoid both these inconveniences by noticing that all the functions that read a particular type of value have the same fundamental purpose, to read a value of a specific type from a stream. Speaking colloquially, you might say they're all instances of a single generic operation. And the colloquial use of the word *generic* should lead you directly to the solution to your problem: instead of defining a bunch of independent functions, all with different names, you can define a single generic function, `read-value`, with methods specialized to read different types of values.

That is, instead of defining functions `read-iso-8859-1-string` and `read-u1`, you can define `read-value` as a generic function taking two required arguments, a type and a stream, and possibly some keyword arguments.

```
(defgeneric read-value (type stream &key)
  (:documentation "Read a value of the given type from the stream."))
```

By specifying **&key** without any actual keyword parameters, you allow different methods to define their own **&key** parameters without requiring them to do so. This does mean every method specialized on `read-value` will have to include either **&key** or an **&rest** parameter in its parameter list to be compatible with the generic function.

Then you'll define methods that use **EQL** specializers to specialize the type argument on the name of the type you want to read.

```
(defmethod read-value ((type (eql 'iso-8859-1-string)) in &key length) ...)

(defmethod read-value ((type (eql 'u1)) in &key) ...)
```

Then you can make `define-binary-class` generate a `read-value` method specialized on the type name `id3-tag`, and that method can be implemented in terms of calls to `read-value` with the appropriate slot types as the first argument. The code you want to generate is going to look like this:

```
(defmethod read-value ((type (eql 'id3-tag)) in &key)
  (let ((object (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) object
      (setf identifier    (read-value 'iso-8859-1-string in :length 3))
      (setf major-version (read-value 'u1 in))
      (setf revision      (read-value 'u1 in))
      (setf flags         (read-value 'u1 in))
      (setf size          (read-value 'id3-encoded-size in))
      (setf frames        (read-value 'id3-frames in :tag-size size)))
    object))
```

So, just as you needed a function to translate a `define-binary-class` slot specifier to a **DEFCLASS** slot specifier in order to generate the **DEFCLASS** form, now you need a function that takes a `define-binary-class` slot specifier and generates the appropriate **SETF** form, that is, something that takes this:

```
(identifier (iso-8859-1-string :length 3))
```

and returns this:

```
(setf identifier (read-value 'iso-8859-1-string in :length 3))
```

However, there's a difference between this code and the **DEFCLASS** slot specifier: it includes a reference to a variable `in`--the method parameter from the `read-value` method--that wasn't derived from the slot specifier. It doesn't have to be called `in`, but whatever name you use has to be the same as the one used in the method's parameter list and in the other calls to `read-value`. For now you can dodge the issue of where that name comes from by defining `slot->read-value` to take a second argument of the name of the stream variable.

```
(defun slot->read-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(setf ,name (read-value ',type ,stream ,@args))))
```

The function `normalize-slot-spec` normalizes the second element of the slot specifier, converting a symbol like `u1` to the list `(u1)` so the **`DESTRUCTURING-BIND`** can parse it. It looks like this:

```
(defun normalize-slot-spec (spec)
  (list (first spec) (mklist (second spec))))

(defun mklist (x) (if (listp x) x (list x)))
```

You can test `slot->read-value` with each type of slot specifier.

```
BINARY-DATA> (slot->read-value '(major-version u1) 'stream)
(SETF MAJOR-VERSION (READ-VALUE 'U1 STREAM))
BINARY-DATA> (slot->read-value '(identifier (iso-8859-1-string :length 3)) 'stream)
(SETF IDENTIFIER (READ-VALUE 'ISO-8859-1-STRING STREAM :LENGTH 3))
```

With these functions you're ready to add `read-value` to `define-binary-class`. If you take the handwritten `read-value` method and strip out anything that's tied to a particular class, you're left with this skeleton:

```
(defmethod read-value ((type (eql ...)) stream &key)
  (let ((object (make-instance ...)))
    (with-slots (...) object
      ...
    object)))
```

All you need to do is add this skeleton to the `define-binary-class` template, replacing ellipses with code that fills in the skeleton with the appropriate names and code. You'll also want to replace the variables `type`, `stream`, and `object` with gensymed names to avoid potential conflicts with slot names,[9] which you can do with the `with-gensyms` macro from Chapter 8.

Also, because a macro must expand into a single form, you need to wrap some form around the **`DEFCLASS`** and **`DEFMETHOD`**. **`PROGN`** is the customary form to use for macros that expand into multiple definitions because of the special treatment it gets from the file compiler when appearing at the top level of a file, as I discussed in Chapter 20.

So, you can change `define-binary-class` as follows:

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
       (defclass ,name ()
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
         (let ((,objectvar (make-instance ',name)))
           (with-slots ,(mapcar #'first slots) ,objectvar
             ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))
           ,objectvar)))))
```

Writing Binary Objects

Generating code to write out an instance of a binary class will proceed similarly. First you can define a `write-value` generic function.

```
(defgeneric write-value (type stream value &key)
  (:documentation "Write a value as the given type to the stream."))
```

Then you define a helper function that translates a `define-binary-class` slot specifier into code that writes out the slot using `write-value`. As with the `slot->read-value` function, this helper function needs to take the name of the stream variable as an argument.

```
(defun slot->write-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(write-value ',type ,stream ,name ,@args)))
```

Now you can add a `write-value` template to the `define-binary-class` macro.

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
       (defclass ,name ()
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
         (let ((,objectvar (make-instance ',name)))
           (with-slots ,(mapcar #'first slots) ,objectvar
             ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))
           ,objectvar))

       (defmethod write-value ((,typevar (eql ',name)) ,streamvar ,objectvar &key)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Adding Inheritance and Tagged Structures

While this version of `define-binary-class` will handle stand-alone structures, binary file formats often define on-disk structures that would be natural to model with subclasses and superclasses. So you might want to extend `define-binary-class` to support inheritance.

A related technique used in many binary formats is to have several on-disk structures whose exact type can be determined only by reading some data that indicates how to parse the following bytes. For instance, the frames that make up the bulk of an ID3 tag all share a common header structure consisting of a string identifier and a length. To read a frame, you need to read the identifier and use its value to determine what kind of frame you're looking at and thus how to parse the body of the frame.

The current `define-binary-class` macro has no way to handle this kind of reading--you could use `define-binary-class` to define a class to represent each kind of frame, but you'd have no way to know what type of frame to read without reading at least the identifier. And if other code reads the identifier in order to determine what type to pass to `read-value`, then that will break `read-value` since it's expecting to read all the data that makes up the instance of the class it instantiates.

You can solve this problem by adding inheritance to `define-binary-class` and then writing another macro, `define-tagged-binary-class`, for defining "abstract" classes

that aren't instantiated directly but that can be specialized on by `read-value` methods that know how to read enough data to determine what kind of class to create.

The first step to adding inheritance to `define-binary-class` is to add a parameter to the macro to accept a list of superclasses.

```
(defmacro define-binary-class (name (&rest superclasses) slots) ...
```

Then, in the **`DEFCLASS`** template, interpolate that value instead of the empty list.

```
(defclass ,name ,superclasses
  ...)
```

However, there's a bit more to it than that. You also need to change the `read-value` and `write-value` methods so the methods generated when defining a superclass can be used by the methods generated as part of a subclass to read and write inherited slots.

The current way `read-value` works is particularly problematic since it instantiates the object before filling it in--obviously, you can't have the method responsible for reading the superclass's fields instantiate one object while the subclass's method instantiates and fills in a different object.

You can fix that problem by splitting `read-value` into two parts--one responsible for instantiating the correct kind of object and another responsible for filling slots in an existing object. On the writing side it's a bit simpler, but you can use the same technique.

So you'll define two new generic functions, `read-object` and `write-object`, that will both take an existing object and a stream. Methods on these generic functions will be responsible for reading or writing the slots specific to the class of the object on which they're specialized.

```
(defgeneric read-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Fill in the slots of object from stream."))

(defgeneric write-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Write out the slots of object to the stream."))
```

Defining these generic functions to use the **`PROGN`** method combination with the option `:most-specific-last` allows you to define methods that specialize `object` on each binary class and have them deal only with the slots actually defined in that class; the **`PROGN`** method combination will combine all the applicable methods so the method specialized on the least specific class in the hierarchy runs first, reading or writing the slots defined in that class, then the method specialized on next least specific subclass, and so on. And since all the heavy lifting for a specific class is now going to be done by `read-object` and `write-object`, you don't even need to define specialized `read-value` and `write-value` methods; you can define default methods that assume the type argument is the name of a binary class.

```
(defmethod read-value ((type symbol) stream &key)
  (let ((object (make-instance type)))
    (read-object object stream)
    object))

(defmethod write-value ((type symbol) stream value &key)
  (assert (typep value type))
  (write-object value stream))
```

Note how you can use **`MAKE-INSTANCE`** as a generic object factory--while you normally call **`MAKE-INSTANCE`** with a quoted symbol as the first argument because you normally know exactly what class you want to instantiate, you can use any expression that evaluates to a class name such as, in this case, the `type` parameter in the `read-value` method.

The actual changes to `define-binary-class` to define methods on `read-object` and `write-object` rather than `read-value` and `write-value` are fairly minor.

```
(defmacro define-binary-class (name superclasses slots)
  (with-gensyms (objectvar streamvar)
    `(progn
       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Keeping Track of Inherited Slots

This definition will work for many purposes. However, it doesn't handle one fairly common situation, namely, when you have a subclass that needs to refer to inherited slots in its own slot specifications. For instance, with the current definition of `define-binary-class`, you can define a single class like this:

```
(define-binary-class generic-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)
   (data (raw-bytes :bytes size))))
```

The reference to `size` in the specification of `data` works the way you'd expect because the expressions that read and write the `data` slot are wrapped in a **`WITH-SLOTS`** that lists all the object's slots. However, if you try to split that class into two classes like this:

```
(define-binary-class frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)))

(define-binary-class generic-frame (frame)
  ((data (raw-bytes :bytes size))))
```

you'll get a compile-time warning when you compile the `generic-frame` definition and a runtime error when you try to use it because there will be no lexically apparent variable `size` in

the `read-object` and `write-object` methods specialized on `generic-frame`.

What you need to do is keep track of the slots defined by each binary class and then include inherited slots in the **`WITH-SLOTS`** forms in the `read-object` and `write-object` methods.

The easiest way to keep track of information like this is to hang it off the symbol that names the class. As I discussed in Chapter 21, every symbol object has an associated property list, which can be accessed via the functions **`SYMBOL-PLIST`** and **`GET`**. You can associate arbitrary key/value pairs with a symbol by adding them to its property list with **`SETF`** of **`GET`**. For instance, if the binary class `foo` defines three slots--`x`, `y`, and `z`--you can keep track of that fact by adding a `slots` key to the symbol `foo`'s property list with the value `(x y z)` with this expression:

```
(setf (get 'foo 'slots) '(x y z))
```

You want this bookkeeping to happen as part of evaluating the `define-binary-class` of `foo`. However, it's not clear where to put the expression. If you evaluate it when you compute the macro's expansion, it'll get evaluated when you compile the `define-binary-class` form but not if you later load a file that contains the resulting compiled code. On the other hand, if you include the expression in the expansion, then it *won't* be evaluated during compilation, which means if you compile a file with several `define-binary-class` forms, none of the information about what classes define what slots will be available until the whole file is loaded, which is too late.

This is what the special operator **`EVAL-WHEN`** I discussed in Chapter 20 is for. By wrapping a form in an **`EVAL-WHEN`**, you can control whether it's evaluated at compile time, when the compiled code is loaded, or both. For cases like this where you want to squirrel away some information during the compilation of a macro form that you also want to be available after the compiled form is loaded, you should wrap it in an **`EVAL-WHEN`** like this:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get 'foo 'slots) '(x y z)))
```

and include the **`EVAL-WHEN`** in the expansion generated by the macro. Thus, you can save both the slots and the direct superclasses of a binary class by adding this form to the expansion generated by `define-binary-class`:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ',name 'slots) ',(mapcar #'first slots))
  (setf (get ',name 'superclasses) ',superclasses))
```

Now you can define three helper functions for accessing this information. The first simply returns the slots directly defined by a binary class. It's a good idea to return a copy of the list since you don't want other code to modify the list of slots after the binary class has been defined.

```
(defun direct-slots (name)
  (copy-list (get name 'slots)))
```

The next function returns the slots inherited from other binary classes.

```
(defun inherited-slots (name)
  (loop for super in (get name 'superclasses)
        nconc (direct-slots super)
        nconc (inherited-slots super)))
```

Finally, you can define a function that returns a list containing the names of all directly defined and inherited slots.

```
(defun all-slots (name)
  (nconc (direct-slots name) (inherited-slots name)))
```

When you're computing the expansion of a `define-generic-binary-class` form, you want to generate a **`WITH-SLOTS`** form that contains the names of all the slots defined in the new class and all its superclasses. However, you can't use `all-slots` while you're generating the expansion since the information won't be available until after the expansion is compiled. Instead, you should use the following function, which takes the list of slot specifiers and superclasses passed to `define-generic-binary-class` and uses them to compute the list of all the new class's slots:

```
(defun new-class-all-slots (slots superclasses)
  (nconc (mapcan #'all-slots superclasses) (mapcar #'first slots)))
```

With these functions defined, you can change `define-binary-class` to store the information about the class currently being defined and to use the already stored information about the superclasses' slots to generate the **`WITH-SLOTS`** forms you want like this:

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(progn
       (eval-when (:compile-toplevel :load-toplevel :execute)
         (setf (get ',name 'slots) ',(mapcar #'first slots))
         (setf (get ',name 'superclasses) ',superclasses))

       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Tagged Structures

With the ability to define binary classes that extend other binary classes, you're ready to define a new macro for defining classes to represent "tagged" structures. The strategy for reading tagged structures will be to define a specialized `read-value` method that knows how to read the values that make up the start of the structure and then use those values to determine what

subclass to instantiate. It'll then make an instance of that class with **`MAKE-INSTANCE`**, passing the already read values as initargs, and pass the object to `read-object`, allowing the actual class of the object to determine how the rest of the structure is read.

The new macro, `define-tagged-binary-class`, will look like `define-binary-class` with the addition of a `:dispatch` option used to specify a form that should evaluate to the name of a binary class. The `:dispatch` form will be evaluated in a context where the names of the slots defined by the tagged class are bound to variables that hold the values read from the file. The class whose name it returns must accept initargs corresponding to the slot names defined by the tagged class. This is easily ensured if the `:dispatch` form always evaluates to the name of a class that subclasses the tagged class.

For instance, supposing you have a function, `find-frame-class`, that will map a string identifier to a binary class representing a particular kind of ID3 frame, you might define a tagged binary class, `id3-frame`, like this:

```
(define-tagged-binary-class id3-frame ()
  ((id   (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

The expansion of a `define-tagged-binary-class` will contain a **`DEFCLASS`** and a `write-object` method just like the expansion of `define-binary-class`, but instead of a `read-object` method it'll contain a `read-value` method that looks like this:

```
(defmethod read-value ((type (eql 'id3-frame)) stream &key)
  (let ((id (read-value 'iso-8859-1-string stream :length 3))
        (size (read-value 'u3 stream)))
    (let ((object (make-instance (find-frame-class id) :id id :size size)))
      (read-object object stream)
      object)))
```

Since the expansions of `define-tagged-binary-class` and `define-binary-class` are going to be identical except for the read method, you can factor out the common bits into a helper macro, `define-generic-binary-class`, that accepts the read method as a parameter and interpolates it.

```
(defmacro define-generic-binary-class (name (&rest superclasses) slots read-method)
  (with-gensyms (objectvar streamvar)
    `(progn
       (eval-when (:compile-toplevel :load-toplevel :execute)
         (setf (get ',name 'slots) ',(mapcar #'first slots))
         (setf (get ',name 'superclasses) ',superclasses))

       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       ,read-method

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (declare (ignorable ,streamvar))
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Now you can define both `define-binary-class` and `define-tagged-binary-class` to expand into a call to `define-generic-binary-class`. Here's a new version of `define-binary-class` that generates the same code as the earlier version when it's fully expanded:

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (declare (ignorable ,streamvar))
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))))))
```

And here's `define-tagged-binary-class` along with two new helper functions it uses:

```
(defmacro define-tagged-binary-class (name (&rest superclasses) slots &rest options)
  (with-gensyms (typevar objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let* ,(mapcar #'(lambda (x) (slot->binding x streamvar)) slots)
          (let ((,objectvar
                 (make-instance
                  ,@(or (cdr (assoc :dispatch options))
                        (error "Must supply :dispatch form."))
                  ,@(mapcan #'slot->keyword-arg slots))))
            (read-object ,objectvar ,streamvar)
            ,objectvar))))))

(defun slot->binding (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(,name (read-value ',type ,stream ,@args))))

(defun slot->keyword-arg (spec)
  (let ((name (first spec)))
    `(,(as-keyword name) ,name)))
```

Primitive Binary Types

While `define-binary-class` and `define-tagged-binary-class` make it easy to define composite structures, you still have to write `read-value` and `write-value` methods for primitive data types by hand. You could decide to live with that, specifying that users of the library need to write appropriate methods on `read-value` and `write-value` to support the primitive types used by their binary classes.

However, rather than having to document how to write a suitable `read-value`/`write-value` pair, you can provide a macro to do it automatically. This also has the advantage of making the abstraction created by `define-binary-class` less leaky. Currently, `define-binary-class` depends on having methods on `read-value` and `write-value` defined in a particular way, but that's really just an implementation detail. By defining a macro that generates the `read-value` and `write-value` methods for primitive types, you hide those details behind an abstraction you control. If you decide later to change the implementation of `define-binary-class`, you can change your primitive-type-defining

macro to meet the new requirements without requiring any changes to code that uses the binary data library.

So you should define one last macro, `define-binary-type`, that will generate `read-value` and `write-value` methods for reading values represented by instances of existing classes, rather than by classes defined with `define-binary-class`.

For a concrete example, consider a type used in the `id3-tag` class, a fixed-length string encoded in ISO-8859-1 characters. I'll assume, as I did earlier, that the native character encoding of your Lisp is ISO-8859-1 or a superset, so you can use **`CODE-CHAR`** and **`CHAR-CODE`** to translate bytes to characters and back.

As always, your goal is to write a macro that allows you to express only the essential information needed to generate the required code. In this case, there are four pieces of essential information: the name of the type, `iso-8859-1-string`; the **`&key`** parameters that should be accepted by the `read-value` and `write-value` methods, `length` in this case; the code for reading from a stream; and the code for writing to a stream. Here's an expression that contains those four pieces of information:

```
(define-binary-type iso-8859-1-string (length)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in))))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out))))
```

Now you just need a macro that can take apart this form and put it back together in the form of two **`DEFMETHOD`**s wrapped in a **`PROGN`**. If you define the parameter list to `define-binary-type` like this:

```
(defmacro define-binary-type (name (&rest args) &body spec) ...
```

then within the macro the parameter `spec` will be a list containing the reader and writer definitions. You can then use **`ASSOC`** to extract the elements of `spec` using the tags `:reader` and `:writer` and then use **`DESTRUCTURING-BIND`** to take apart the **`REST`** of each element.[10]

From there it's just a matter of interpolating the extracted values into the backquoted templates of the `read-value` and `write-value` methods.

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (with-gensyms (type)
    `(progn
      ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
        `(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
          ,@body))
      ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
```

```
      `(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
        ,@body)))))
```

Note how the backquoted templates are nested: the outermost template starts with the backquoted **`PROGN`** form. That template consists of the symbol **`PROGN`** and two comma-unquoted **`DESTRUCTURING-BIND`** expressions. Thus, the outer template is filled in by evaluating the **`DESTRUCTURING-BIND`** expressions and interpolating their values. Each **`DESTRUCTURING-BIND`** expression in turn contains another backquoted template, which is used to generate one of the method definitions to be interpolated in the outer template.

With this macro defined, the `define-binary-type` form given previously expands to this code:

```
(progn
  (defmethod read-value ((#:g1618 (eql 'iso-8859-1-string)) in &key length)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in))))
      string))
  (defmethod write-value ((#:g1618 (eql 'iso-8859-1-string)) out string &key length)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out))))
```

Of course, now that you've got this nice macro for defining binary types, it's tempting to make it do a bit more work. For now you should just make one small enhancement that will turn out to be pretty handy when you start using this library to deal with actual formats such as ID3 tags.

ID3 tags, like many other binary formats, use lots of primitive types that are minor variations on a theme, such as unsigned integers in one-, two-, three-, and four-byte varieties. You could certainly define each of those types with `define-binary-type` as it stands. Or you could factor out the common algorithm for reading and writing *n*-byte unsigned integers into helper functions.

But suppose you had already defined a binary type, `unsigned-integer`, that accepts a `:bytes` parameter to specify how many bytes to read and write. Using that type, you could specify a slot representing a one-byte unsigned integer with a type specifier of `(unsigned-integer :bytes 1)`. But if a particular binary format specifies lots of slots of that type, it'd be nice to be able to easily define a new type--say, `u1`--that means the same thing. As it turns out, it's easy to change `define-binary-type` to support two forms, a long form consisting of a `:reader` and `:writer` pair and a short form that defines a new binary type in terms of an existing type. Using a short form `define-binary-type`, you can define `u1` like this:

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
```

which will expand to this:

```
(progn
  (defmethod read-value ((#:g161887 (eql 'u1)) #:g161888 &key)
    (read-value 'unsigned-integer #:g161888 :bytes 1))
```

```
  (defmethod write-value ((#:g161887 (eql 'u1)) #:g161888 #:g161889 &key)
    (write-value 'unsigned-integer #:g161888 #:g161889 :bytes 1)))
```

To support both long- and short-form `define-binary-type` calls, you need to differentiate based on the value of the `spec` argument. If `spec` is two items long, it represents a long-form call, and the two items should be the `:reader` and `:writer` specifications, which you extract as before. On the other hand, if it's only one item long, the one item should be a type specifier, which needs to be parsed differently. You can use **`ECASE`** to switch on the **`LENGTH`** of `spec` and then parse `spec` and generate an appropriate expansion for either the long form or the short form.

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (ecase (length spec)
    (1
     (with-gensyms (type stream value)
       (destructuring-bind (derived-from &rest derived-args) (mklist (first spec))
         `(progn
            (defmethod read-value ((,type (eql ',name)) ,stream &key ,@args)
              (read-value ',derived-from ,stream ,@derived-args))
            (defmethod write-value ((,type (eql ',name)) ,stream ,value &key ,@args)
              (write-value ',derived-from ,stream ,value ,@derived-args))))))
    (2
     (with-gensyms (type)
       `(progn
          ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
             `(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
                ,@body))
          ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
             `(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
                ,@body)))))))
```

The Current Object Stack

One last bit of functionality you'll need in the next chapter is a way to get at the binary object being read or written while reading and writing. More generally, when reading or writing nested composite objects, it's useful to be able to get at any of the objects currently being read or written. Thanks to dynamic variables and `:around` methods, you can add this enhancement with about a dozen lines of code. To start, you should define a dynamic variable that will hold a stack of objects currently being read or written.

```
(defvar *in-progress-objects* nil)
```

Then you can define `:around` methods on `read-object` and `write-object` that push the object being read or written onto this variable before invoking **`CALL-NEXT-METHOD`**.

```
(defmethod read-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))

(defmethod write-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))
```

Note how you rebind `*in-progress-objects*` to a list with a new item on the front rather than assigning it a new value. This way, at the end of the **`LET`**, after **`CALL-NEXT-METHOD`** returns, the old value of `*in-progress-objects*` will be restored, effectively popping the object of the stack.

With those two methods defined, you can provide two convenience functions for getting at specific objects in the in-progress stack. The function `current-binary-object` will return the head of the stack, the object whose `read-object` or `write-object` method was invoked most recently. The other, `parent-of-type`, takes an argument that should be the name of a binary object class and returns the most recently pushed object of that type, using the **`TYPEP`** function that tests whether a given object is an instance of a particular type.

```
(defun current-binary-object () (first *in-progress-objects*))

(defun parent-of-type (type)
  (find-if #'(lambda (x) (typep x type)) *in-progress-objects*))
```

These two functions can be used in any code that will be called within the dynamic extent of a `read-object` or `write-object` call. You'll see one example of how `current-binary-object` can be used in the next chapter.[11]

Now you have all the tools you need to tackle an ID3 parsing library, so you're ready to move onto the next chapter where you'll do just that.

---

[1]In ASCII, the first 32 characters are nonprinting *control characters* originally used to control the behavior of a Teletype machine, causing it to do such things as sound the bell, back up one character, move to a new line, and move the carriage to the beginning of the line. Of these 32 control characters, only three, the newline, carriage return, and horizontal tab, are typically found in text files.

[2]Some binary file formats *are* in-memory data structures--on many operating systems it's possible to map a file into memory, and low-level languages such as C can then treat the region of memory containing the contents of the file just like any other memory; data written to that area of memory is saved to the underlying file when it's unmapped. However, these formats are platform-dependent since the in-memory representation of even such simple data types as integers depends on the hardware on which the program is running. Thus, any file format that's intended to be portable must define a canonical representation for all the data types it uses that can be mapped to the actual in-memory data representation on a particular kind of machine or in a particular language.

[3]The term *big-endian* and its opposite, *little-endian*, borrowed from Jonathan Swift's *Gulliver's Travels*, refer to the way a multibyte number is represented in an ordered sequence of bytes such as in memory or in a file. For instance, the number 43981, or `abcd` in hex, represented as a 16-bit quantity, consists of two bytes, `ab` and `cd`. It doesn't matter to a computer in what order these two bytes are stored as long as everybody agrees. Of course, whenever there's an arbitrary choice to be made between two equally good options, the one thing you can be sure of is that everybody is not going to agree. For more than you ever wanted to know about it, and to see where the terms *big-endian* and *little-endian* were first applied in this fashion, read "On Holy Wars and a Plea for Peace" by Danny Cohen, available at `http://khavrinen.lcs.mit.edu/wollman/ien-137.txt`.

[4]**`LDB`** and **`DPB`**, a related function, were named after the DEC PDP-10 assembly functions that did essentially the same thing. Both functions operate on integers as if they were represented using twos-complement format, regardless of the internal representation used by a particular Common Lisp implementation.

[5]Common Lisp also provides functions for shifting and masking the bits of integers in a way that may be more familiar to C and Java programmers. For instance, you could write `read-u2` yet a third way, using those functions, like this:

```
(defun read-u2 (in)
  (logior (ash (read-byte in) 8) (read-byte in)))
```

which would be roughly equivalent to this Java method:

```
public int readU2 (InputStream in) throws IOException {
  return (in.read() << 8) | (in.read());
}
```

The names **LOGIOR** and **ASH** are short for *LOGical Inclusive OR* and *Arithmetic SHift*. **ASH** shifts an integer a given number of bits to the left when its second argument is positive or to the right if the second argument is negative. **LOGIOR** combines integers by logically *or*ing each bit. Another function, **LOGAND**, performs a bitwise *and*, which can be used to mask off certain bits. However, for the kinds of bit twiddling you'll need to do in this chapter and the next, **LDB** and **BYTE** will be both more convenient and more idiomatic Common Lisp style.

[6]Originally, UTF-8 was designed to represent a 31-bit character code and used up to six bytes per code point. However, the maximum Unicode code point is `#x10ffff`, so a UTF-8 encoding of Unicode requires at most four bytes per code point.

[7]If you need to parse a file format that uses other character codes, or if you need to parse files containing arbitrary Unicode strings using a non-Unicode-Common-Lisp implementation, you can always represent such strings in memory as vectors of integer code points. They won't be Lisp strings, so you won't be able to manipulate or compare them with the string functions, but you'll still be able to do anything with them that you can with arbitrary vectors.

[8]Unfortunately, the language itself doesn't always provide a good model in this respect: the macro **DEFSTRUCT**, which I don't discuss since it has largely been superseded by **DEFCLASS**, generates functions with names that it generates based on the name of the structure it's given. **DEFSTRUCT**'s bad example leads many new macro writers astray.

[9]Technically there's no possibility of `type` or `object` conflicting with slot names--at worst they'd be shadowed within the **WITH-SLOTS** form. But it doesn't hurt anything to simply **GENSYM** all local variable names used within a macro template.

[10]Using **ASSOC** to extract the `:reader` and `:writer` elements of `spec` allows users of `define-binary-type` to include the elements in either order; if you required the `:reader` element to be always be first, you could then have used `(rest (first spec))` to extract the reader and `(rest (second spec))` to extract the writer. However, as long as you require the `:reader` and `:writer` keywords to improve the readability of `define-binary-type` forms, you might as well use them to extract the correct data.

[11]The ID3 format doesn't require the `parent-of-type` function since it's a relatively flat structure. This function comes into its own when you need to parse a format made up of many deeply nested structures whose parsing depends on information stored in higher-level structures. For example, in the Java class file format, the top-level class file structure contains a *constant pool* that maps numeric values used in other substructures within the class file to constant values that are needed while parsing those substructures. If you were writing a class file parser, you could use `parent-of-type` in the code that reads and writes those substructures to get at the top-level class file object and from there to the constant pool.