10. Numbers, Characters, and Strings

While functions, variables, macros, and 25 special operators provide the basic building blocks of the language itself, the building blocks of your programs will be the data structures you use. As Fred Brooks observed in *The Mythical Man-Month*, "Representation *is* the essence of programming."[1]

Common Lisp provides built-in support for most of the data types typically found in modern languages: numbers (integer, floating point, and complex), characters, strings, arrays (including multidimensional arrays), lists, hash tables, input and output streams, and an abstraction for portably representing filenames. Functions are also a first-class data type in Lisp--they can be stored in variables, passed as arguments, returned as return values, and created at runtime.

And these built-in types are just the beginning. They're defined in the language standard so programmers can count on them being available and because they tend to be easier to implement efficiently when tightly integrated with the rest of the implementation. But, as you'll see in later chapters, Common Lisp also provides several ways for you to define new data types, define operations on them, and integrate them with the built-in data types.

For now, however, you can start with the built-in data types. Because Lisp is a high-level language, the details of exactly how different data types are implemented are largely hidden. From your point of view as a user of the language, the built-in data types are defined by the functions that operate on them. So to learn a data type, you just have to learn about the functions you can use with it. Additionally, most of the built-in data types have a special syntax that the Lisp reader understands and that the Lisp printer uses. That's why, for instance, you can write strings as `"foo"`; numbers as `123`, `1/23`, and `1.23`; and lists as `(a b c)`. I'll describe the syntax for different kinds of objects when I describe the functions for manipulating them.

In this chapter, I'll cover the built-in "scalar" data types: numbers, characters, and strings. Technically, strings aren't true scalars--a string is a sequence of characters, and you can access individual characters and manipulate strings with a function that operates on sequences. But I'll discuss strings here because most of the string-specific functions manipulate them as single values and also because of the close relation between several of the string functions and their character counterparts.

Numbers

Math, as Barbie says, is hard.[2] Common Lisp can't make the math part any easier, but it does tend to get in the way a lot less than other programming languages. That's not surprising given its mathematical heritage. Lisp was originally designed by a mathematician as a tool for studying mathematical functions. And one of the main projects of the MAC project at MIT was the Macsyma symbolic algebra system, written in Maclisp, one of Common Lisp's immediate predecessors. Additionally, Lisp has been used as a teaching language at places such as MIT where even the computer science professors cringe at the thought of telling their students that *10/4* = *2*, leading to Lisp's support for exact ratios. And at various times Lisp has been called upon to compete with FORTRAN in the high-performance numeric computing arena.

One of the reasons Lisp is a nice language for math is its numbers behave more like true mathematical numbers than the approximations of numbers that are easy to implement in finite computer hardware. For instance, integers in Common Lisp can be almost arbitrarily large rather than being limited by the size of a machine word.[3] And dividing two integers results in an exact ratio, not a truncated value. And since ratios are represented as pairs of arbitrarily sized integers, ratios can represent arbitrarily precise fractions.[4]

On the other hand, for high-performance numeric programming, you may be willing to trade the exactitude of rationals for the speed offered by using the hardware's underlying floating-point operations. So, Common Lisp also offers several types of floating-point numbers, which are mapped by the implementation to the appropriate hardware-supported floating-point representations.[5] Floats are also used to represent the results of a computation whose true mathematical value would be an irrational number.

Finally, Common Lisp supports complex numbers--the numbers that result from doing things such as taking square roots and logarithms of negative numbers. The Common Lisp standard even goes so far as to specify the principal values and branch cuts for irrational and transcendental functions on the complex domain.

Numeric Literals

You can write numeric literals in a variety of ways; you saw a few examples in Chapter 4. However, it's important to keep in mind the division of labor between the Lisp reader and the Lisp evaluator--the reader is responsible for translating text into Lisp objects, and the Lisp evaluator then deals only with those objects. For a given number of a given type, there can be many different textual representations, all of which will be translated to the same object representation by the Lisp reader. For instance, you can write the integer 10 as `10`, `20/2`, `#xA`, or any of a number of other ways, but the reader will translate all these to the same object. When numbers are printed back out--say, at the REPL--they're printed in a canonical textual syntax that may be different from the syntax used to enter the number. For example:

```
CL-USER> 10
10
CL-USER> 20/2
```

```
10
CL-USER> #xa
10
```

The syntax for integer values is an optional sign (+ or -) followed by one or more digits. Ratios are written as an optional sign and a sequence of digits, representing the numerator, a slash (/), and another sequence of digits representing the denominator. All rational numbers are "canonicalized" as they're read--that's why `10` and `20/2` are both read as the same number, as are `3/4` and `6/8`. Rationals are printed in "reduced" form--integer values are printed in integer syntax and ratios with the numerator and denominator reduced to lowest terms.

It's also possible to write rationals in bases other than 10. If preceded by `#B` or `#b`, a rational literal is read as a binary number with `0` and `1` as the only legal digits. An `#O` or `#o` indicates an octal number (legal digits `0-7`), and `#X` or `#x` indicates hexadecimal (legal digits `0-F` or `0-f`). You can write rationals in other bases from 2 to 36 with `#nR` where *n* is the base (always written in decimal). Additional "digits" beyond 9 are taken from the letters `A-Z` or `a-z`. Note that these radix indicators apply to the whole rational--it's not possible to write a ratio with the numerator in one base and denominator in another. Also, you can write integer values, but not ratios, as decimal digits terminated with a decimal point.[6] Some examples of rationals, with their canonical, decimal representation are as follows:

```
123                            ==> 123
+123                           ==> 123
-123                           ==> -123
123.                           ==> 123
2/3                            ==> 2/3
-2/3                           ==> -2/3
4/6                            ==> 2/3
6/3                            ==> 2
#b10101                        ==> 21
#b1010/1011                    ==> 10/11
#o777                          ==> 511
#xDADA                         ==> 56026
#36rABCDEFGHIJKLMNOPQRSTUVWXYZ ==> 8337503854730415241050377135811259267835
```

You can also write floating-point numbers in a variety of ways. Unlike rational numbers, the syntax used to notate a floating-point number can affect the actual type of number read. Common Lisp defines four subtypes of floating-point number: short, single, double, and long. Each subtype can use a different number of bits in its representation, which means each subtype can represent values spanning a different range and with different precision. More bits gives a wider range and more precision.[7]

The basic format for floating-point numbers is an optional sign followed by a nonempty sequence of decimal digits possibly with an embedded decimal point. This sequence can be followed by an exponent marker for "computerized scientific notation."[8] The exponent marker consists of a single letter followed by an optional sign and a sequence of digits, which are interpreted as the power of ten by which the number before the exponent marker should be multiplied. The letter does double duty: it marks the beginning of the exponent and indicates what floating- point representation should be used for the number. The exponent markers *s*, *f*, *d*, *l* (and their uppercase equivalents) indicate short, single, double, and long floats, respectively. The letter *e* indicates that the default representation (initially single-float) should be used.

Numbers with no exponent marker are read in the default representation and must contain a decimal point followed by at least one digit to distinguish them from integers. The digits in a floating-point number are always treated as base 10 digits--the `#B`, `#X`, `#O`, and `#R` syntaxes work only with rationals. The following are some example floating-point numbers along with their canonical representation:

```
1.0       ==> 1.0
1e0       ==> 1.0
1d0       ==> 1.0d0
123.0     ==> 123.0
123e0     ==> 123.0
0.123     ==> 0.123
.123      ==> 0.123
123e-3    ==> 0.123
123E-3    ==> 0.123
0.123e20  ==> 1.23e+19
123d23    ==> 1.23d+25
```

Finally, complex numbers are written in their own syntax, namely, `#C` or `#c` followed by a list of two real numbers representing the real and imaginary part of the complex number. There are actually five kinds of complex numbers because the real and imaginary parts must either both be rational or both be the same kind of floating-point number.

But you can write them however you want--if a complex is written with one rational and one floating-point part, the rational is converted to a float of the appropriate representation. Similarly, if the real and imaginary parts are both floats of different representations, the one in the smaller representation will be *upgraded*.

However, no complex numbers have a rational real component and a zero imaginary part--since such values are, mathematically speaking, rational, they're represented by the appropriate rational value. The same mathematical argument could be made for complex numbers with floating-point components, but for those complex types a number with a zero imaginary part is always a different object than the floating-point number representing the real component. Here are some examples of numbers written the complex number syntax:

```
#c(2      1)    ==> #c(2 1)
#c(2/3  3/4)    ==> #c(2/3 3/4)
#c(2    1.0)    ==> #c(2.0 1.0)
#c(2.0  1.0d0)  ==> #c(2.0d0 1.0d0)
#c(1/2  1.0)    ==> #c(0.5 1.0)
#c(3      0)    ==> 3
#c(3.0  0.0)    ==> #c(3.0 0.0)
#c(1/2    0)    ==> 1/2
#c(-6/3   0)    ==> -2
```

Basic Math

The basic arithmetic operations--addition, subtraction, multiplication, and division--are supported for all the different kinds of Lisp numbers with the functions `+`, `-`, `*`, and `/`. Calling any of these functions with more than two arguments is equivalent to calling the same function on the first two arguments and then calling it again on the resulting value and the rest of the arguments. For example, `(+ 1 2 3)` is equivalent to `(+ (+ 1 2) 3)`. With only one argument, `+` and `*` return the value; `-` returns its negation and `/` its reciprocal.[9]

```
(+ 1 2)                ==> 3
(+ 1 2 3)              ==> 6
(+ 10.0 3.0)           ==> 13.0
(+ #c(1 2) #c(3 4))    ==> #c(4 6)
(- 5 4)                ==> 1
(- 2)                  ==> -2
(- 10 3 5)             ==> 2
(* 2 3)                ==> 6
(* 2 3 4)              ==> 24
(/ 10 5)               ==> 2
(/ 10 5 2)             ==> 1
(/ 2 3)                ==> 2/3
(/ 4)                  ==> 1/4
```

If all the arguments are the same type of number (rational, floating point, or complex), the result will be the same type except in the case where the result of an operation on complex numbers with rational components yields a number with a zero imaginary part, in which case the result will be a rational. However, floating-point and complex numbers are *contagious*--if all the arguments are reals but one or more are floating-point numbers, the other arguments are converted to the nearest floating-point value in a "largest" floating-point representation of the actual floating-point arguments. Floating-point numbers in a "smaller" representation are also converted to the larger representation. Similarly, if any of the arguments are complex, any real arguments are converted to the complex equivalents.

```
(+ 1 2.0)               ==> 3.0
(/ 2 3.0)               ==> 0.6666667
(+ #c(1 2) 3)           ==> #c(4 2)
(+ #c(1 2) 3/2)         ==> #c(5/2 2)
(+ #c(1 1) #c(2 -1))    ==> 3
```

Because `/` doesn't truncate, Common Lisp provides four flavors of truncating and rounding for converting a real number (rational or floating point) to an integer: **`FLOOR`** truncates toward negative infinity, returning the largest integer less than or equal to the argument. **`CEILING`** truncates toward positive infinity, returning the smallest integer greater than or equal to the argument. **`TRUNCATE`** truncates toward zero, making it equivalent to **`FLOOR`** for positive arguments and to **`CEILING`** for negative arguments. And **`ROUND`** rounds to the nearest integer. If the argument is exactly halfway between two integers, it rounds to the nearest even integer.

Two related functions are **`MOD`** and **`REM`**, which return the modulus and remainder of a truncating division on real numbers. These two functions are related to the **`FLOOR`** and **`TRUNCATE`** functions as follows:

```
(+ (* (floor    (/ x y)) y) (mod x y)) === x
(+ (* (truncate (/ x y)) y) (rem x y)) === x
```

Thus, for positive quotients they're equivalent, but for negative quotients they produce different results.[10]

The functions **`1+`** and **`1-`** provide a shorthand way to express adding and subtracting one from a number. Note that these are different from the macros **`INCF`** and **`DECF`**. **`1+`** and **`1-`** are just functions that return a new value, but **`INCF`** and **`DECF`** modify a place. The following equivalences show the relation between **`INCF`**/**`DECF`**, **`1+`**/**`1-`**, and **`+`**/**`-`**:

```
(incf x)    === (setf x (1+ x)) === (setf x (+ x 1))
(decf x)    === (setf x (1- x)) === (setf x (- x 1))
(incf x 10) === (setf x (+ x 10))
(decf x 10) === (setf x (- x 10))
```

Numeric Comparisons

The function **`=`** is the numeric equality predicate. It compares numbers by mathematical value, ignoring differences in type. Thus, **`=`** will consider mathematically equivalent values of different types equivalent while the generic equality predicate **`EQL`** would consider them inequivalent because of the difference in type. (The generic equality predicate **`EQUALP`**, however, uses **`=`** to compare numbers.) If it's called with more than two arguments, it returns true only if they all have the same value. Thus:

```
(= 1 1)                        ==> T
(= 10 20/2)                    ==> T
(= 1 1.0 #c(1.0 0.0) #c(1 0))  ==> T
```

The **`/=`** function, conversely, returns true only if *all* its arguments are different values.

```
(/= 1 1)        ==> NIL
(/= 1 2)        ==> T
(/= 1 2 3)      ==> T
(/= 1 2 3 1)    ==> NIL
(/= 1 2 3 1.0)  ==> NIL
```

The functions **`<`**, **`>`**, **`<=`**, and **`>=`** order rationals and floating-point numbers (in other words, the real numbers.) Like **`=`** and **`/=`**, these functions can be called with more than two arguments, in which case each argument is compared to the argument to its right.

```
(< 2 3)       ==> T
(> 2 3)       ==> NIL
(> 3 2)       ==> T
(< 2 3 4)     ==> T
(< 2 3 3)     ==> NIL
(<= 2 3 3)    ==> T
(<= 2 3 3 4)  ==> T
(<= 2 3 4 3)  ==> NIL
```

To pick out the smallest or largest of several numbers, you can use the function **`MIN`** or **`MAX`**, which takes any number of real number arguments and returns the minimum or maximum value.

```
(max 10 11)    ==> 11
(min -12 -10)  ==> -12
(max -1 2 -3)  ==> 2
```

Some other handy functions are **`ZEROP`**, **`MINUSP`**, and **`PLUSP`**, which test whether a single real number is equal to, less than, or greater than zero. Two other predicates, **`EVENP`** and **`ODDP`**, test whether a single integer argument is even or odd. The *P* suffix on the names of these functions is a standard naming convention for predicate functions, functions that test some condition and return a boolean.

Higher Math

The functions you've seen so far are the beginning of the built-in mathematical functions. Lisp also supports logarithms: **`LOG`**; exponentiation: **`EXP`** and **`EXPT`**; the basic trigonometric functions: **`SIN`**, **`COS`**, and **`TAN`**; their inverses: **`ASIN`**, **`ACOS`**, and **`ATAN`**; hyperbolic functions: **`SINH`**, **`COSH`**, and **`TANH`**; and their inverses: **`ASINH`**, **`ACOSH`**, and **`ATANH`**. It also provides functions to get at the individual bits of an integer and to extract the parts of a ratio or a complex number. For a complete list, see any Common Lisp reference.

Characters

Common Lisp characters are a distinct type of object from numbers. That's as it should be--characters are *not* numbers, and languages that treat them as if they are tend to run into problems when character encodings change, say, from 8-bit ASCII to 21-bit Unicode.[11] Because the Common Lisp standard didn't mandate a particular representation for characters, today several Lisp implementations use Unicode as their "native" character encoding despite Unicode being only a gleam in a standards body's eye at the time Common Lisp's own standardization was being wrapped up.

The read syntax for characters objects is simple: `#\` followed by the desired character. Thus, `#\x` is the character `x`. Any character can be used after the `#\`, including otherwise special characters such as `"`, `(`, and whitespace. However, writing whitespace characters this way isn't very (human) readable; an alternative syntax for certain characters is `#\` followed by the character's name. Exactly what names are supported depends on the character set and on the Lisp implementation, but all implementations support the names *Space* and *Newline*. Thus, you should write `#\Space` instead of `#\ `, though the latter is technically legal. Other semistandard names (that implementations must use if the character set has the appropriate characters) are *Tab*, *Page*, *Rubout*, *Linefeed*, *Return*, and *Backspace*.

Character Comparisons

The main thing you can do with characters, other than putting them into strings (which I'll get to later in this chapter), is to compare them with other characters. Since characters aren't numbers, you can't use the numeric comparison functions, such as **<** and **>**. Instead, two sets of functions provide character-specific analogs to the numeric comparators; one set is case-sensitive and the other case-insensitive.

The case-sensitive analog to the numeric **=** is the function **CHAR=**. Like **=**, **CHAR=** can take any number of arguments and returns true only if they're all the same character. The case- insensitive version is **CHAR-EQUAL**.

The rest of the character comparators follow this same naming scheme: the case-sensitive comparators are named by prepending the analogous numeric comparator with `CHAR`; the case-insensitive versions spell out the comparator name, separated from the `CHAR` with a hyphen. Note, however, that <= and >= are "spelled out" with the logical equivalents **NOT-GREATERP** and **NOT-LESSP** rather than the more verbose `LESSP-OR-EQUALP` and `GREATERP-OR-EQUALP`. Like their numeric counterparts, all these functions can take one or more arguments. Table 10-1 summarizes the relation between the numeric and character comparison functions.

Table 10-1. Character Comparison Functions

Numeric Analog	Case-Sensitive	Case-Insensitive
=	CHAR=	CHAR-EQUAL
/=	CHAR/=	CHAR-NOT-EQUAL
<	CHAR<	CHAR-LESSP
>	CHAR>	CHAR-GREATERP
<=	CHAR<=	CHAR-NOT-GREATERP
>=	CHAR>=	CHAR-NOT-LESSP

Other functions that deal with characters provide functions for, among other things, testing whether a given character is alphabetic or a digit character, testing the case of a character, obtaining a corresponding character in a different case, and translating between numeric values representing character codes and actual character objects. Again, for complete details, see your favorite Common Lisp reference.

Strings

As mentioned earlier, strings in Common Lisp are really a composite data type, namely, a one-dimensional array of characters. Consequently, I'll cover many of the things you can do with strings in the next chapter when I discuss the many functions for manipulating sequences, of which strings are just one type. But strings also have their own literal syntax and a library of functions for performing string-specific operations. I'll discuss these aspects of strings in this chapter and leave the others for Chapter 11.

As you've seen, literal strings are written enclosed in double quotes. You can include any character supported by the character set in a literal string except double quote (`"`) and backslash (\). And you can include these two as well if you escape them with a backslash. In fact, backslash always escapes the next character, whatever it is, though this isn't necessary for any character except for `"` and itself. Table 10-2 shows how various literal strings will be read by the Lisp reader.

Table 10-2. Literal Strings

Literal	Contents	Comment
`"foobar"`	foobar	Plain string.
`"foo\"bar"`	foo"bar	The backslash escapes quote.
`"foo\\bar"`	foo\bar	The first backslash escapes second backslash.
`"\"foobar\""`	"foobar"	The backslashes escape quotes.
`"foo\bar"`	foobar	The backslash "escapes" *b*

Note that the REPL will ordinarily print strings in readable form, adding the enclosing quotation marks and any necessary escaping backslashes, so if you want to see the actual contents of a string, you need to use function such as **`FORMAT`** designed to print human-readable output. For example, here's what you see if you type a string containing an embedded quotation mark at the REPL:

```
CL-USER> "foo\"bar"
"foo\"bar"
```

`FORMAT`, on the other hand, will show you the actual string contents:[12]

```
CL-USER> (format t "foo\"bar")
foo"bar
NIL
```

String Comparisons

You can compare strings using a set of functions that follow the same naming convention as the character comparison functions except with `STRING` as the prefix rather than `CHAR` (see Table 10-3).

Table 10-3. String Comparison Functions

Numeric Analog	Case-Sensitive	Case-Insensitive
=	**STRING=**	**STRING-EQUAL**
/=	**STRING/=**	**STRING-NOT-EQUAL**
<	**STRING<**	**STRING-LESSP**
>	**STRING>**	**STRING-GREATERP**
<=	**STRING<=**	**STRING-NOT-GREATERP**
>=	**STRING>=**	**STRING-NOT-LESSP**

However, unlike the character and number comparators, the string comparators can compare only two strings. That's because they also take keyword arguments that allow you to restrict the comparison to a substring of either or both strings. The arguments--`:start1`, `:end1`, `:start2`, and `:end2`--specify the starting (inclusive) and ending (exclusive) indices of substrings in the first and second string arguments. Thus, the following:

```
(string= "foobarbaz" "quuxbarfoo" :start1 3 :end1 6 :start2 4 :end2 7)
```

compares the substring "bar" in the two arguments and returns true. The `:end1` and `:end2` arguments can be **`NIL`** (or the keyword argument omitted altogether) to indicate that the corresponding substring extends to the end of the string.

The comparators that return true when their arguments differ--that is, all of them except **STRING=** and **STRING-EQUAL**--return the index in the first string where the mismatch was detected.

```
(string/= "lisp" "lissome") ==> 3
```

If the first string is a prefix of the second, the return value will be the length of the first string, that is, one greater than the largest valid index into the string.

```
(string< "lisp" "lisper") ==> 4
```

When comparing substrings, the resulting value is still an index into the string as a whole. For instance, the following compares the substrings "bar" and "baz" but returns 5 because that's the index of the *r* in the first string:

```
(string< "foobar" "abaz" :start1 3 :start2 1) ==> 5   ; N.B. not 2
```

Other string functions allow you to convert the case of strings and trim characters from one or both ends of a string. And, as I mentioned previously, since strings are really a kind of sequence, all the sequence functions I'll discuss in the next chapter can be used with strings. For instance, you can discover the length of a string with the **LENGTH** function and can get and set individual characters of a string with the generic sequence element accessor function, **ELT**, or the generic array element accessor function, **AREF**. Or you can use the string-specific accessor, **CHAR**. But those functions, and others, are the topic of the next chapter, so let's move on.

---

[1]Fred Brooks, *The Mythical Man-Month*, 20th Anniversary Edition (Boston: Addison-Wesley, 1995), p. 103. Emphasis in original.

[2]Mattel's Teen Talk Barbie

[3]Obviously, the size of a number that can be represented on a computer with finite memory is still limited in practice; furthermore, the actual representation of *bignums* used in a particular Common Lisp implementation may place other limits on the size of number that can be represented. But these limits are going to be well beyond "astronomically" large numbers. For instance, the number of atoms in the universe is estimated to be less than 2^269; current Common Lisp implementations can easily handle numbers up to and beyond 2^262144.

[4]Folks interested in using Common Lisp for intensive numeric computation should note that a naive comparison of the performance of numeric code in Common Lisp and languages such as C or FORTRAN will probably show Common Lisp to be much slower. This is because something as simple as `(+ a b)` in Common Lisp is doing a lot more than the seemingly equivalent `a + b` in one of those languages. Because of Lisp's dynamic typing and support for things such as arbitrary precision rationals and complex numbers, a seemingly simple addition is doing a lot more than an addition of two numbers that are known to be represented by machine words. However, you can use declarations to give Common Lisp information about the types of numbers you're using that will enable it to generate code that does only as much work as the code that would be generated by a C or FORTRAN compiler. Tuning numeric code for this kind of performance is beyond the scope of this book, but it's certainly possible.

[5]While the standard doesn't require it, many Common Lisp implementations support the IEEE standard for floating-point arithmetic, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/ IEEE Std 754-1985* (Institute of Electrical and Electronics Engineers, 1985).

[6]It's also possible to change the default base the reader uses for numbers without a specific radix marker by changing the value of the global variable ***READ-BASE***. However, it's not clear that's the path to anything other than complete insanity.

[7]Since the purpose of floating-point numbers is to make efficient use of floating-point hardware, each Lisp implementation is allowed to map these four subtypes onto the native floating-point types as appropriate. If the hardware supports fewer than four distinct representations, one or more of the types may be equivalent.

[8]"Computerized scientific notation" is in scare quotes because, while commonly used in computer languages since the days of FORTRAN, it's actually quite different from real scientific notation. In particular, something like `1.0e4` means `10000.0`, but in true scientific notation that would be written as 1.0 x 10^4. And to further confuse matters, in true scientific notation the letter *e* stands for the base of the natural logarithm, so something like 1.0 x *e*^4, while superficially similar to `1.0e4`, is a completely different value, approximately 54.6.

[9]For mathematical consistency, **`+`** and **`*`** can also be called with no arguments, in which case they return the appropriate identity: 0 for **`+`** and 1 for **`*`**.

[10]Roughly speaking, **`MOD`** is equivalent to the `%` operator in Perl and Python, and **`REM`** is equivalent to the % in C and Java. (Technically, the exact behavior of % in C wasn't specified until the C99 standard.)

[11]Even Java, which was designed from the beginning to use Unicode characters on the theory that Unicode was the going to be *the* character encoding of the future, has run into trouble since Java characters are defined to be a 16-bit quantity and the Unicode 3.1 standard extended the range of the Unicode character set to require a 21-bit representation. Ooops.

[12]Note, however, that not all literal strings can be printed by passing them as the second argument to **`FORMAT`** since certain sequences of characters have a special meaning to **`FORMAT`**. To safely print an arbitrary string--say, the value of a variable s--with **`FORMAT`** you should write (format t "~a" s).