1

Functions and Data

1.1 INTRODUCTION

This chapter begins with an overview of the notions of function and data, followed by examples of several built-in Lisp functions. If you already have some experience programming in other languages, you can flip through this chapter in just a few minutes. You'll see arithmetic functions, followed by an introduction to symbols, one of the key datatypes of Lisp, and predicates, which answer yes-or-no questions. When you think you've grasped this material, read the summary section on page 26 to test your understanding.

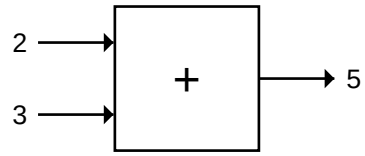
If you're new to programming, this chapter is designed specifically for you. We'll start by explaining what **functions** and **data** are.* The term data means *information*, such as numbers, words, or lists of things. You can think of a function as a box through which data flows. The function operates on the data in some way, and the **result** is what flows out.

After covering some of the built-in functions provided by Lisp, we will learn how to put existing functions together to make new ones—the essence of computer programming. Several useful techniques for creating new functions will then be presented.

*Technical terms like these, which appear in boldface in the text, are defined in the glossary at the back of the book.

1.2 FUNCTIONS ON NUMBERS

Probably the most familiar functions are the simple arithmetic functions of addition, subtraction, multiplication, and division. Here is how we represent the addition of two numbers:

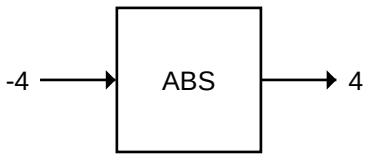


The name of the function is ‘‘+.’’ We can describe what’s going on in the figure in several ways. From the point of view of the data: The numbers 2 and 3 flow into the function, and the number 5 flows out. From the point of view of the function: The function ‘‘+’’ received the numbers 2 and 3 as inputs, and it produced 5 as its result. From the programmer’s point of view: We **called** (or **invoked**) the function ‘‘+’’ on the inputs 2 and 3, and the function **returned** 5. These different ways of talking about functions and data are equivalent; you will encounter all of them in various places in this book.

Here is a table of Lisp functions that do useful things with numbers:

+	Adds two numbers
-	Subtracts the second number from the first
*	Multiplies two numbers
/	Divides the first number by the second
ABS	Absolute value of a number
SQRT	Square root of a number

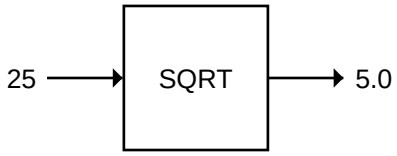
Let’s look at another example of how data flows through a function. The output of the absolute value function, ABS, is the same as its input, except that negative numbers are converted to positive ones.



The number −4 enters the ABS function, which computes the absolute value and outputs a result of 4.

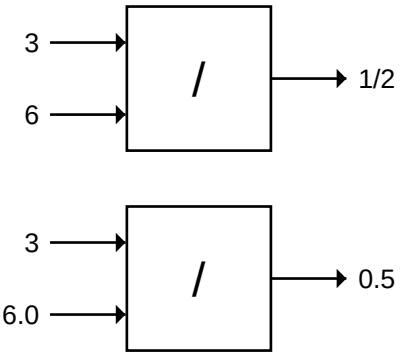
1.3 THREE KINDS OF NUMBERS

In this book we will work mostly with **integers**, which are whole numbers. Common Lisp provides many other kinds of numbers. One kind you should know about is **floating point** numbers. A floating point number is always written with a decimal point; for example, the number five would be written 5.0. The SQRT function generally returns a floating point number as its result, even when its input is an integer.



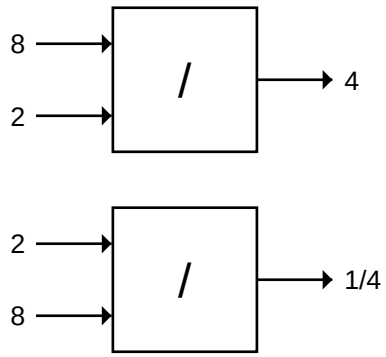
Ratios are yet another kind of number. On a pocket calculator, one-half must be written in floating point notation, as 0.5, but in Common Lisp we can also write one-half as the ratio 1/2. Common Lisp automatically simplifies ratios to use the smallest possible denominator; for example, the ratios 4/6, 6/9, and 10/15 would all be simplified to 2/3.

When we call an arithmetic function with integer inputs, Common Lisp will usually produce an integer or ratio result. If we use a mixture of integers and floating point numbers, the result will be a floating point number:

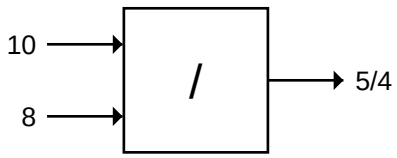


1.4 ORDER OF INPUTS IS IMPORTANT

By convention, when we refer to the ‘‘first’’ input to a function, we mean the topmost arrow entering the function box. The ‘‘second’’ input is the next highest arrow, and so on. The order in which inputs are supplied to a function is important. For example, dividing 8 by 2 is not the same as dividing 2 by 8:

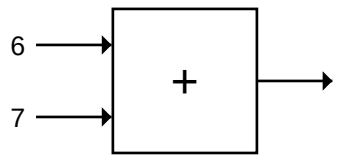


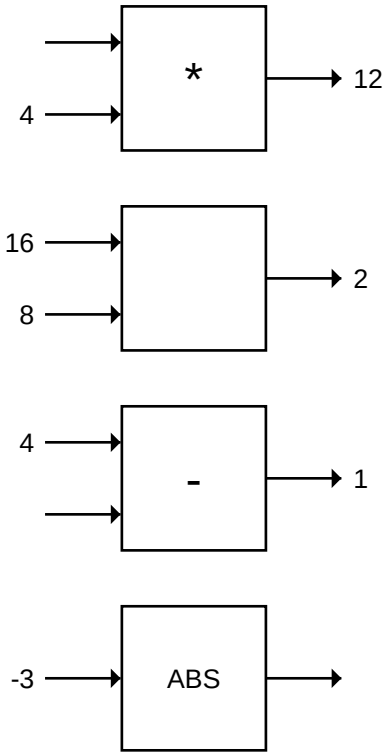
When we divide 8 by 2 we get 4. When we divide 2 by 8 we get the ratio 1/4. By the way, ratios need not always be less than 1. For example:



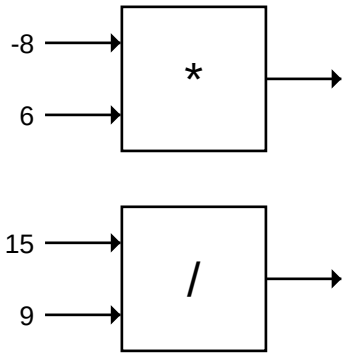
EXERCISE

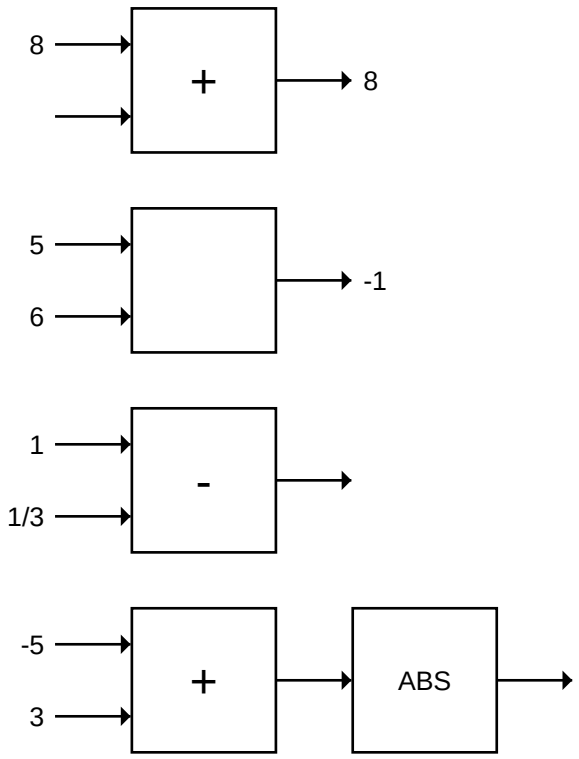
1.1. Here are some function boxes with inputs and outputs. In each case one item of information is missing. Use your knowledge of arithmetic to fill in the missing item:





Here are a few more challenging problems. I'll throw in some negative numbers and ratios just to make things interesting.





1.5 SYMBOLS

Symbols are another type of data in Lisp. Most people find them more interesting than numbers. Symbols are typically named after English words (such as TUESDAY), or phrases (e.g., BUFFALO-BREATH), or common abbreviations (like SQRT for ‘‘square root.’’) Symbol names may contain practically any combination of letters and numbers, plus some special characters such as hyphens. Here are some examples of Lisp symbols:

```
X                       ZORCH
BANANAS                 R2D2
COMPUTER                WINDOW-WASHER
LORETTA                 WARP-ENGINES
ABS                     GARBANZO-BEANS
YEAR-TO-DATE            BEEBOP
```

and even

```
ANTIDISESTABLISHMENTARIANISM
```

Notice that symbols may include digits in their names, as in ‘‘R2D2,’’ but this does not make them numbers. It is important that you be able to tell the difference between numbers—especially integers—and symbols. These definitions should help:

integer A sequence of digits ‘‘0’’ through ‘‘9,’’ optionally preceded by a plus or minus sign.

symbol Any sequence of letters, digits, and permissible special characters that is not a number.

So FOUR is a symbol, 4 is an integer, +4 is an integer, but + is a symbol. And 7-11 is also a symbol.

EXERCISE

1.2. Next to each of the following, put an ‘‘S’’ if it is a symbol, ‘‘I’’ if it is an integer, or ‘‘N’’ if it is some other kind of number. Remember: English words may sound like integers, but a true Lisp integer contains only the digits 0–9, with an optional sign.

_____ AARDVARK
_____ 87
_____ PLUMBING
_____ 1-2-3-GO
_____ 1492
_____ 3.14159265358979
_____ 22/7
_____ ZEROP
_____ ZERO
_____ 0
_____ −12
_____ SEVENTEEN

1.6 THE SPECIAL SYMBOLS T AND NIL

Two Lisp symbols have special meanings attached to them. They are:

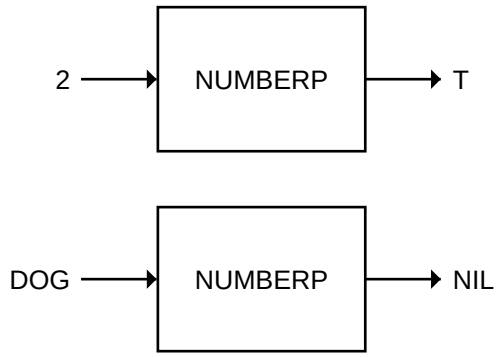
T Truth, ‘‘yes’’

NIL Falsity, emptiness, ‘‘no’’

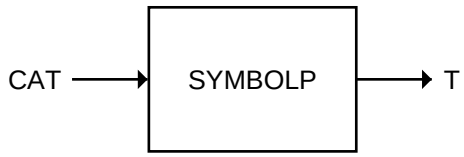
T and NIL are so basic to Lisp that if you ask a really dedicated Lisp programmer a yes-or-no question, he may answer with T or NIL instead of English. (‘‘Hey, Jack, want to go to dinner?’’ ‘‘NIL. I just ate.’’) More importantly, certain Lisp *functions* answer questions with T or NIL. Such yes-or-no functions are called **predicates**.

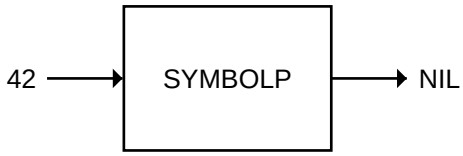
1.7 SOME SIMPLE PREDICATES

A predicate is a question-answering function. Predicates output the symbol T when they mean *yes* and the symbol NIL when they mean *no*. The first predicate we will study is the one that tests whether its input is a number or not. It is called NUMBERP (pronounced ‘‘number-pee,’’ as in ‘‘number predicate’’), and it looks like this:

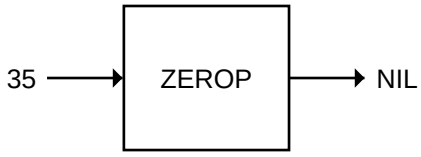


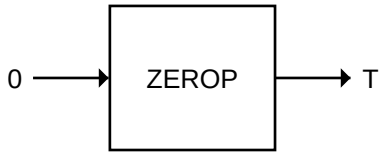
Similarly, the SYMBOLP predicate tests whether its input is a symbol. SYMBOLP returns T when given an input that is a symbol; it returns NIL for inputs that are not symbols.



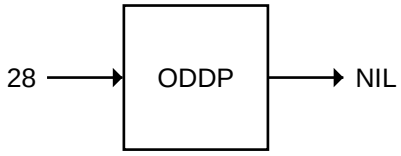


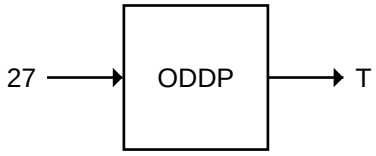
The ZEROP, EVENP, and ODDP predicates work only on numbers. ZEROP returns T if its input is zero.





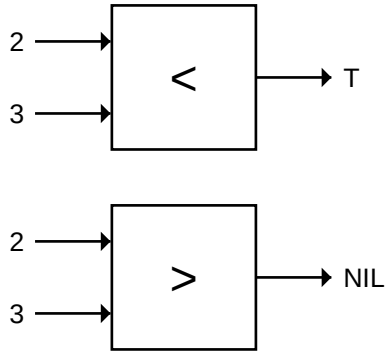
ODDP returns T if its input is odd; otherwise it returns NIL. EVENP does the reverse.





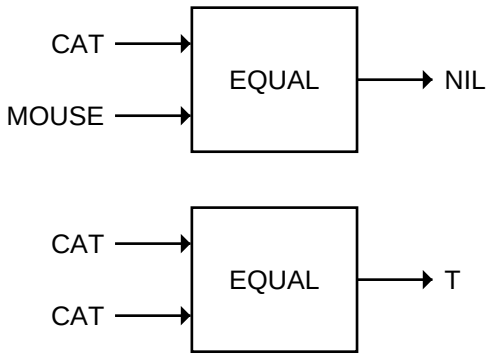
By now you've caught on to the convention of tacking a ''P'' onto a function name to show that it is a predicate. (''Hey, Jack, HUNGRYP?'' ''T, I'm starved!'') Not all Lisp predicates obey this rule, but most do.

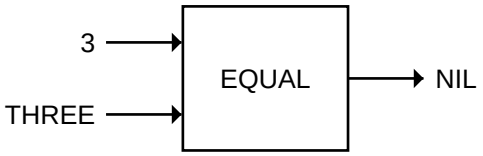
Here are two more predicates: < returns T if its first input is less than its second, while > returns T if its first input is greater than its second. (They are also our first exceptions to the convention that predicate names end with a ''P.'')



1.8 THE EQUAL PREDICATE

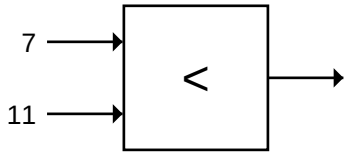
EQUAL is a predicate for comparing two things to see if they are the same. EQUAL returns T if its two inputs are equal; otherwise it returns NIL. Common Lisp also includes predicates named EQ, EQL, and EQUALP whose behavior is slightly different than EQUAL; the differences will not concern us here. For beginners, EQUAL is the right one to use.

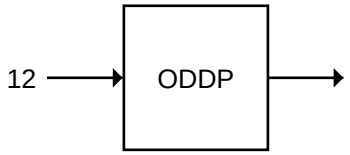


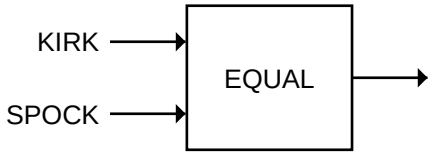


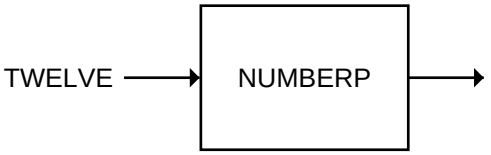
EXERCISE

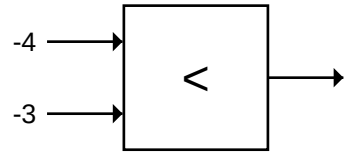
1.3. Fill in the result of each computation:

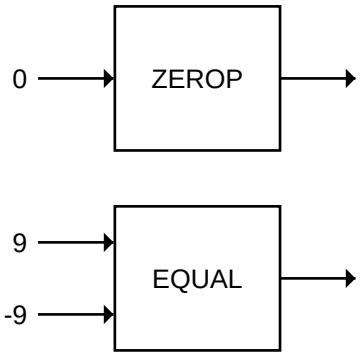










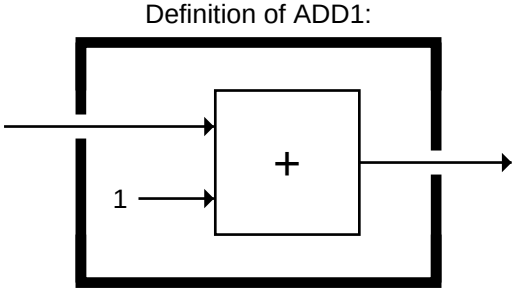


1.9 PUTTING FUNCTIONS TOGETHER

So far we've covered about a dozen of the many functions built into Common Lisp. These built-in functions are called **primitive functions**, or **primitives**. We make new functions by putting primitives together in various ways.

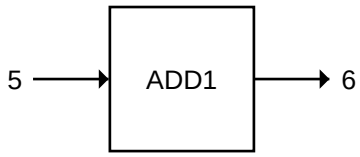
1.9.1 Defining ADD1

Let's define a function that adds one to its input.** We already have a primitive function for addition: The + function will add any two numbers it is given as input. Our ADD1 function will take a single number as input, and add one to it.

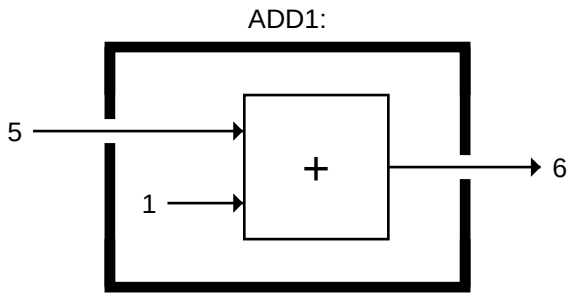


**Note to instructors: Common Lisp contains built-in functions `1+` and `1-` that add 1 to or subtract 1 from their input, respectively. But since these unusual names are almost certain to confuse beginning programmers, I will not refer to them in this book.

Now that we've defined ADD1 we can use it to add 1 to any number we like. We just draw a box with the name ADD1 and supply an input, such as 5:

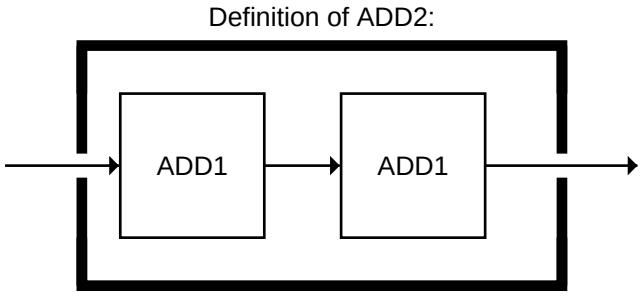


If we look inside the ADD1 box we can see how the function works:

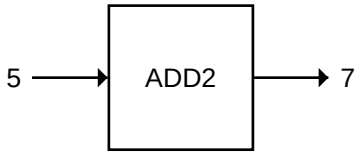


1.9.2 Defining ADD2

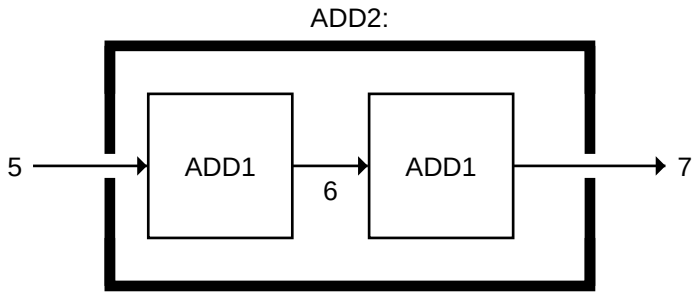
Now suppose we want a function that adds 2 to its input. We could define ADD2 the same way we defined ADD1. But in Lisp there is always more than one way to solve a problem; sometimes it is interesting to look at alternative solutions. For example, we could build ADD2 out of two ADD1 boxes:



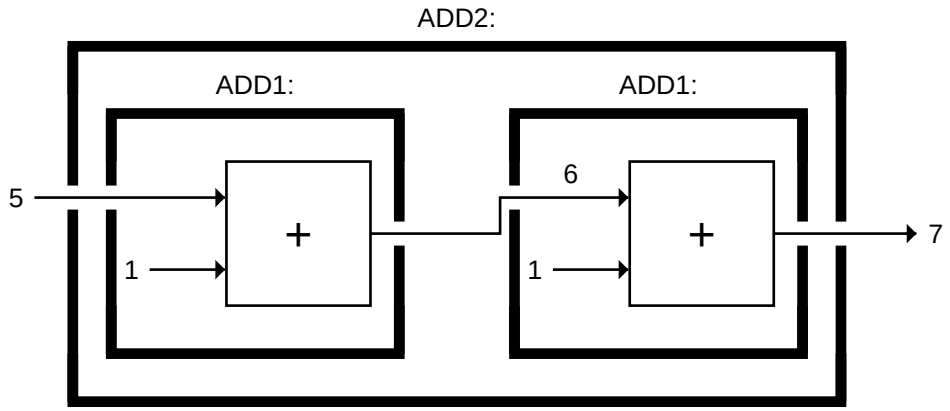
Once we've defined ADD2, we are free to use it to add 2 to any number. Looking at the ADD2 box from the outside, we have no way of knowing which solution was chosen:



But if we look inside the ADD2 box we can see exactly what's going on. The number 5 flows into the first ADD1 box, which produces 6 as its result. The 6 then flows into the second ADD1 box, and its result is 7.



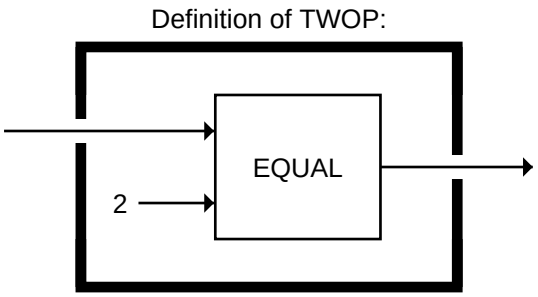
If we want to peer deeper still, we could see the + box inside each ADD1 box, like so:



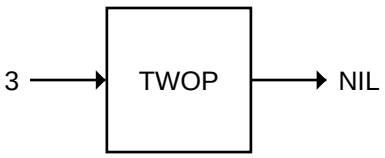
This is as deep as we can go. We can't look inside the + boxes because + is a primitive function.

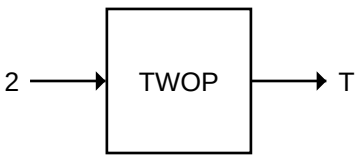
1.9.3 Defining TWOP

We can use our new knowledge to make our own predicates too, since predicates are just a special type of function. Predicates are functions that return a result of T or NIL. The TWOP predicate defined below returns T if its input is equal to 2.



Some examples of the use of TWOP:





EXERCISES

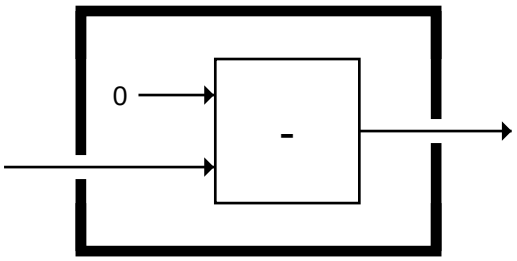
1.4. Define a SUB2 function that subtracts two from its input.

1.5. Show how to write TWOP in terms of ZEROP and SUB2.

1.6. The HALF function returns a number that is one-half of its input. Show how to define HALF two different ways.

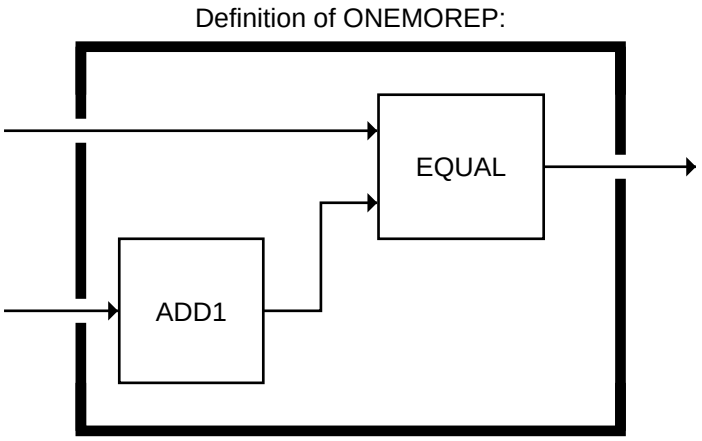
1.7. Write a MULTI-DIGIT-P predicate that returns true if its input is greater than 9.

1.8. What does this function do to a number?



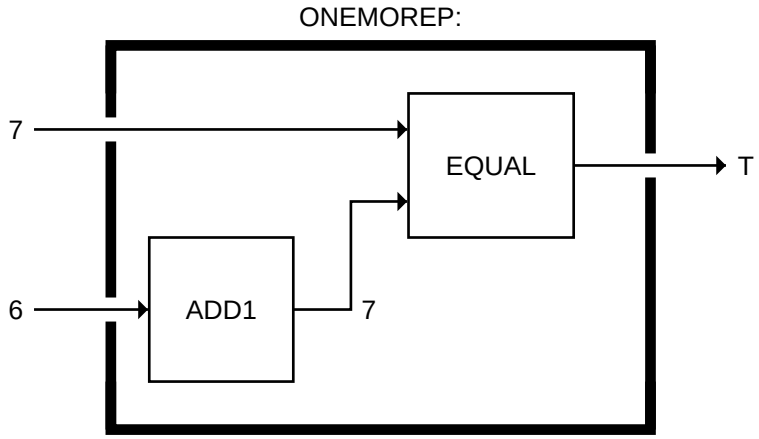
1.9.4 Defining ONEMOREP

Let's try defining a function of two inputs. Here is the ONEMOREP predicate, which tests whether its first input is exactly one greater than its second input.

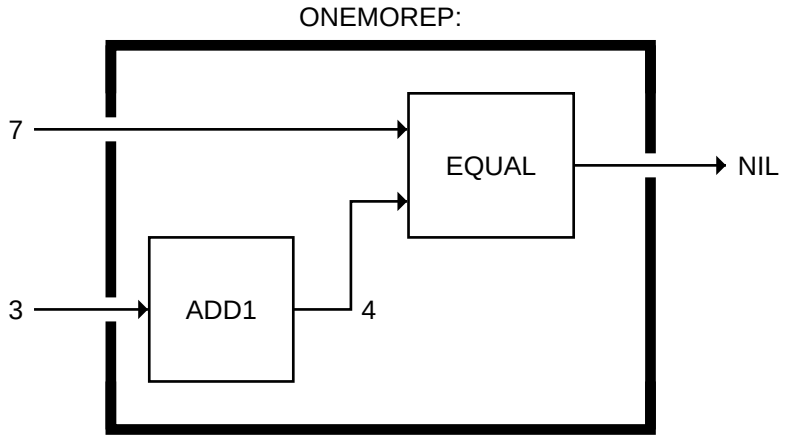


Do you see how ONEMOREP works? If the first input is one greater than the second input, adding 1 to the second input should make the two equal. In this case, the EQUAL predicate will return T. On the other hand, if the first

input to ONEMOREP isn't one greater than the second input, the inputs to EQUAL won't be equal, so it will return NIL. Example:



In your mind (or out loud if you prefer), trace the flow of data through ONEMOREP for the preceding example. You should say something like this: ''The first input is a 7. The second input, a 6, enters ADD1, which outputs a 7. The two 7's enter the EQUAL function, and since they *are* equal, it outputs a T. T is the result of ONEMOREP.'' Here is another example to trace:



For this second example you should say: ''The first input is a 7. The second input, a 3, enters ADD1, which outputs a 4. The 7 and the 4 enter the

EQUAL function, and since they *are not* equal, it outputs a NIL. NIL is the result of ONEMOREP.''

EXERCISES

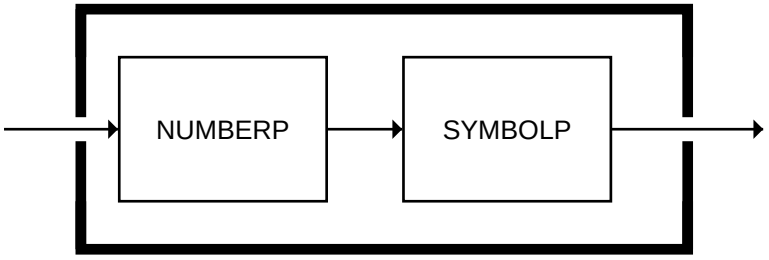
1.9. Write a predicate TWOMOREP that returns T if its first input is exactly two more than its second input. Use the ADD2 function in your definition of TWOMOREP.

1.10. Find a way to write the TWOMOREP predicate using SUB2 instead of ADD2.

1.11. The average of two numbers is half their sum. Write the AVERAGE function.

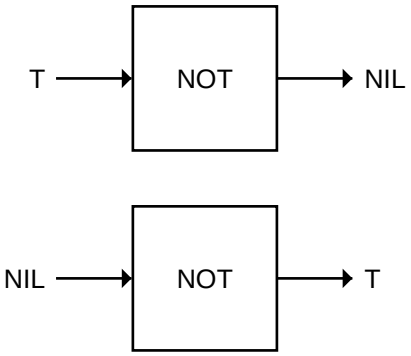
1.12. Write a MORE-THAN-HALF-P predicate that returns T if its first input is more than half of its second input.

1.13. The following function returns the same result no matter what its input. What result does it return?

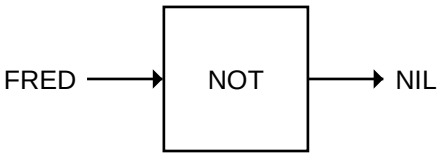


1.10 THE NOT PREDICATE

NOT is the ''opposite'' predicate: It turns *yes* into *no*, and *no* into *yes*. In Lisp terminology, given an input of T, NOT returns NIL. Given an input of NIL, NOT returns T. The neat thing about NOT is that it can be attached to any other predicate to derive its opposite; for example, we can make a ''not equal'' predicate from NOT and EQUAL, or a ''nonzero'' predicate from NOT and ZEROP. We'll see how this is done in the next section. First, some examples of NOT:



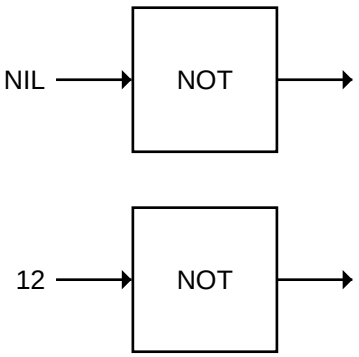
By convention, NIL is the only way to say *no* in Lisp. Everything else is treated as *yes*. So NOT returns NIL for every input except NIL.

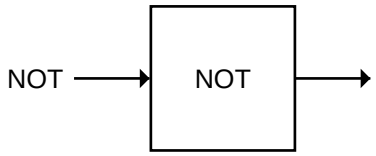


This is not just an arbitrary convention. It turns out to be extremely useful to treat NIL as the only ‘‘false’’ object. You’ll see why in later chapters.

EXERCISE

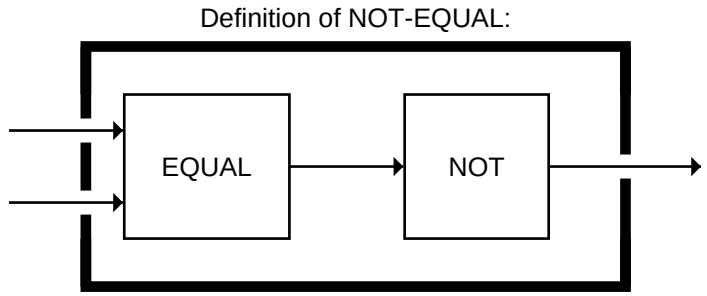
1.14. Fill in the results of the following computations:



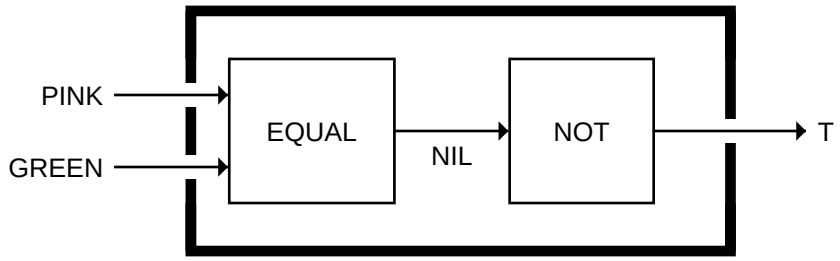


1.11 NEGATING A PREDICATE

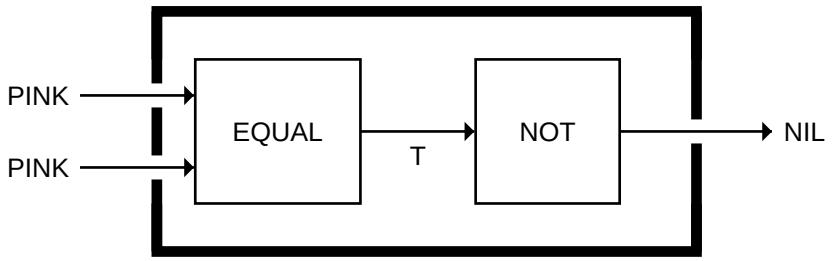
Suppose we want to make a predicate that tests whether two things are not equal—the opposite of the EQUAL predicate. We can build it by starting with EQUAL and running its output through NOT to get the opposite result:



Because of the NOT function, whenever EQUAL would say ‘‘T,’’ NOT-EQUAL will say ‘‘NIL,’’ and whenever EQUAL would say ‘‘NIL,’’ NOT-EQUAL will say ‘‘T.’’ Here are some examples of NOT-EQUAL. In the first one, the symbols PINK and GREEN are different, so EQUAL outputs a NIL and NOT changes it to a T.



In the second example, PINK and PINK are the same, so EQUAL outputs a T. NOT changes this to NIL.



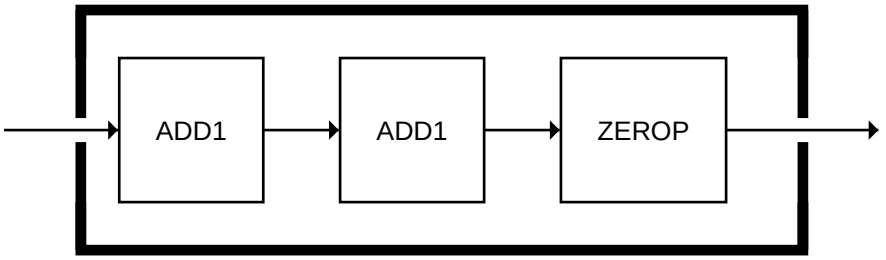
EXERCISES

1.15. Write a predicate NOT-ONEP that returns T if its input is anything other than one.

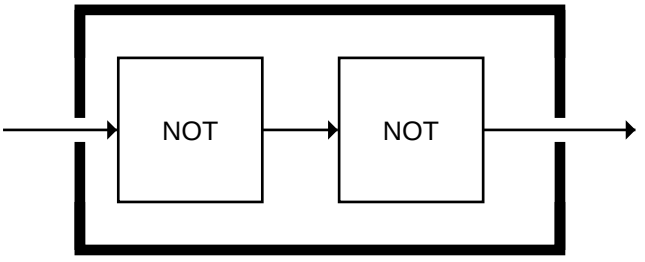
1.16. Write the predicate NOT-PLUSP that returns T if its input is not greater than zero.

1.17. Some earlier Lisp dialects did not have the EVENP primitive; they only had ODDP. Show how to define EVENP in terms of ODDP.

1.18. Under what condition does this predicate function return T?



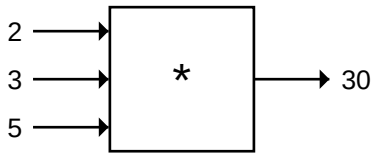
1.19. What result does the function below produce when given the input NIL? What about the input T? Will all data flow through this function unchanged? What result is produced for the input RUTABAGA?



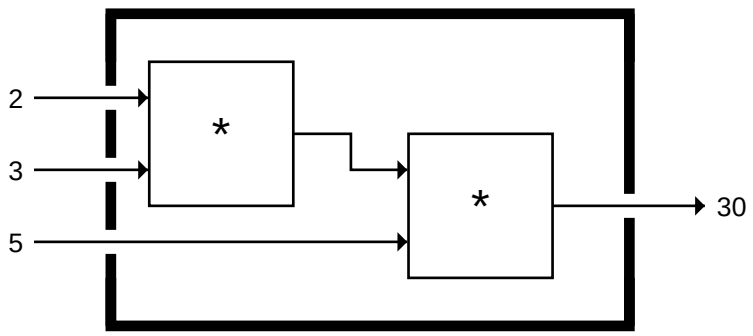
1.20. A **truth function** is a function whose inputs and output are truth values, that is, *true* or *false*. NOT is a truth function. (Even though NOT accepts other inputs besides T or NIL, it only cares if its input is true or not.) Write XOR, the exclusive-or truth function, which returns T when one of its inputs is NIL and the other is T, but returns NIL when both are NIL or both are T. (*Hint:* This is easier than it sounds.)

1.12 NUMBER OF INPUTS TO A FUNCTION

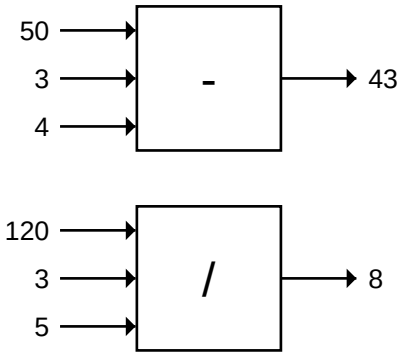
Some functions require a fixed number of inputs, such as ODDP, which accepts exactly one input, and EQUAL, which takes exactly two. But many functions accept a variable number of inputs. For example, the arithmetic functions `+`, `-`, `*`, and `/` will accept any number of inputs.



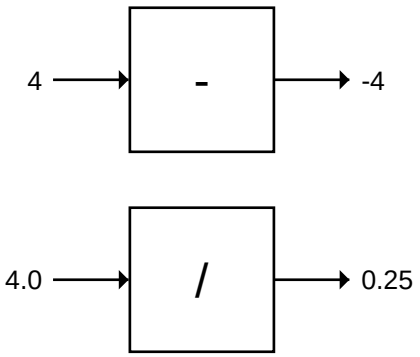
To multiply three numbers, the `*` function multiplies the first two, then multiplies the result by the third, like so:



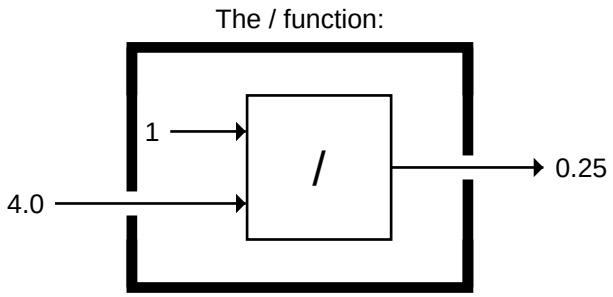
When `-` or `/` is given more than two inputs, the result is the first input diminished (or divided, respectively) by the remaining inputs.



The - and / functions behave differently when given only one input. What - does is negate its input, in other words, it changes the sign from positive to negative or vice versa by subtracting it from zero. When the / function is given a single input, it divides one by that input, which gives the **reciprocal**.

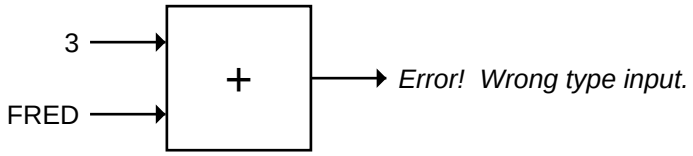


The two-input case is clearly the defining case for the basic arithmetic functions. While they can accept more or fewer than two inputs, they convert those cases to instances of the two-input case. For example, the above computation of the reciprocal of 4.0 is really just a division:

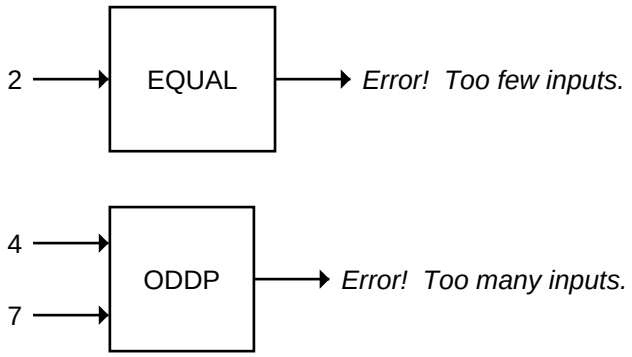


1.13 ERRORS

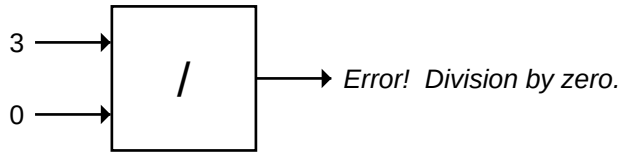
Even though our system of functions is a very simple one, we can already make several types of errors in it. One error is to give a function the wrong type of data. For example, the + function can add only numbers; it cannot add symbols:



Another error is to give a function too few or too many inputs:



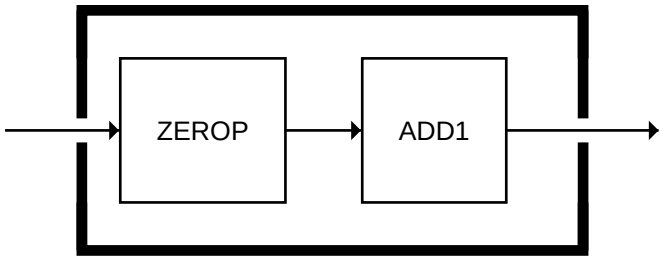
Finally, an error may occur because a function cannot do what is requested of it. This is what happens when we try to divide a number by zero:

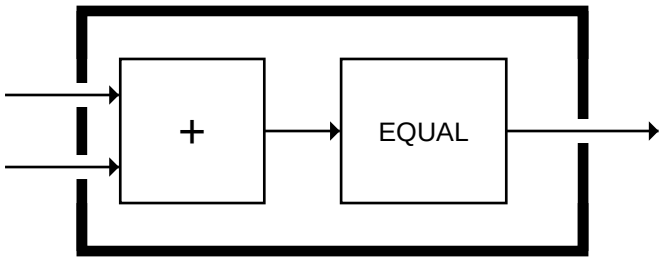


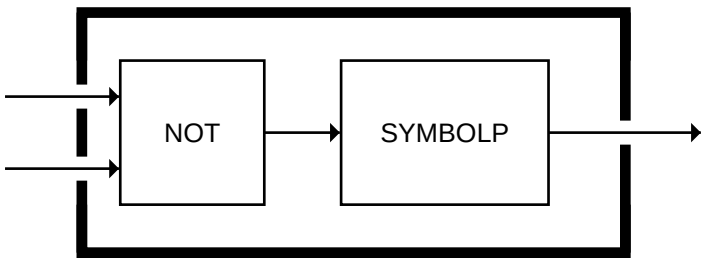
Learning to recognize errors is an important part of programming. You will undoubtedly get lots of practice in this art, since few computer programs are ever written correctly the first time.

EXERCISE

1.21. What is wrong with each of these functions?







SUMMARY

In this chapter we covered two types of data: numbers and symbols. We also learned several built-in functions that operate on them.

Predicates are a special class of functions that use T and NIL to answer questions about their inputs. The symbol NIL means *false*, and the symbol T means *true*. Actually, anything other than NIL is treated as *true* in Lisp.

A function must have a definition before we can use it. We can make new functions by putting old ones together in various ways. A particularly useful combination, used quite often in programming, is to feed the output of a predicate through the NOT function to derive its opposite, as the NOT-EQUAL predicate was derived from EQUAL.

REVIEW EXERCISES

1.22. Are all predicates functions? Are all functions predicates?

1.23. Which built-in predicates introduced in this chapter have names that do not end in ‘‘P’’?

1.24. Is NUMBER a number? Is SYMBOL a symbol?

1.25. Why is FALSE true in Lisp?

1.26. True or false: (a) All predicates accept T or NIL as input; (b) all predicates produce T or NIL as output.

1.27. Give an example of the use of EVENP that would cause a wrong-type-input error. Give an example that would cause a wrong-number-of-inputs error.

FUNCTIONS COVERED IN THIS CHAPTER

Arithmetic functions: `+`, `-`, `*`, `/`, ABS, SQRT.

Predicates: NUMBERP, SYMBOLP, ZEROP, ODDP, EVENP, `<`, `>`, EQUAL, NOT.