2
Lists

2.1 LISTS ARE THE MOST VERSATILE DATA TYPE

The name "Lisp" is an acronym for List Processor. Even though the language has matured in many ways over the years, lists remain its central data type. Lists are important because they can be made to represent practically anything: sets, tables, and graphs, and even English sentences. Functions can also be represented as lists, but we'll save that topic for the next chapter.

2.2 WHAT DO LISTS LOOK LIKE?

Every list has two forms: a printed representation and an internal one. The printed representation is most convenient for people to use, because it's compact and easy to type on a computer keyboard. The internal representation is the way the list actually exists in the computer's memory. We will use a graphical notation when we want to refer to lists in their internal form.

In its printed form, a list is a bunch of items enclosed in parentheses. These items are called the **elements** of the list. Here are some examples of lists written in parenthesis notation:

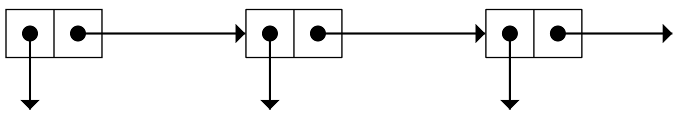
```
(RED GREEN BLUE)

(AARDVARK)
```



```
(2 3 5 7 11 13 17)
```

```
(3 FRENCH HENS 2 TURTLE DOVES 1 PARTRIDGE
 1 PEAR TREE)
```

The internal representation of lists does not involve parentheses. Inside the computer's memory, lists are organized as chains of **cons cells**, which we'll draw as boxes. The cons cells are linked together by **pointers**, which we'll draw as arrows. Each cons cell has two pointers. One of them always points to an element of the list, such as RED, while the other points to the next cons cell in the chain.[1] When we say "lists may include symbols or numbers as elements," what we are really saying is that cons cells may contain pointers to symbols or numbers, as well as pointers to other cons cells. The computer's internal representation of the list (RED GREEN BLUE) is drawn this way:[2]



Looking at the rightmost cell, you'll note that the cons cell chain ends in NIL. This is a convention in Lisp. It may be violated in some circumstances, but most of the time lists will end in NIL. When the list is written in parenthesis notation, the NIL at the end of the chain is omitted, again by convention.

EXERCISE

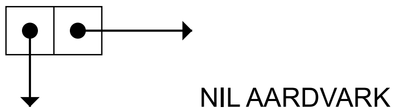
2.1. Show how the list (TO BE OR NOT TO BE) would be represented in computer memory by drawing its cons cell representation.

2.3 LISTS OF ONE ELEMENT

A symbol and a list of one element are not the same. Consider the list (AARDVARK) shown below; it is represented by a cons cell. One of the cons cell's pointers points to the symbol AARDVARK; the other points to NIL. So you see that the list (AARDVARK) and the symbol AARDVARK are different objects. The former is a cons cell that points to the latter.

[1] What each cons cell actually is, internally, is a small piece of memory, split in two, big enough to hold two addresses (pointers) to other places in memory where the actual data (like RED, or NIL, or another cons cell) is stored. On most computers pointers are four bytes long, so each cons cells is eight bytes.

[2] Note to instructors: If students are already using the computer, this would be a good time to introduce the SDRAW tool appearing in the appendix.



2.4 NESTED LISTS

A list may contain other lists as elements. Given the three lists

```
(BLUE SKY)

(GREEN GRASS)

(BROWN EARTH)
```

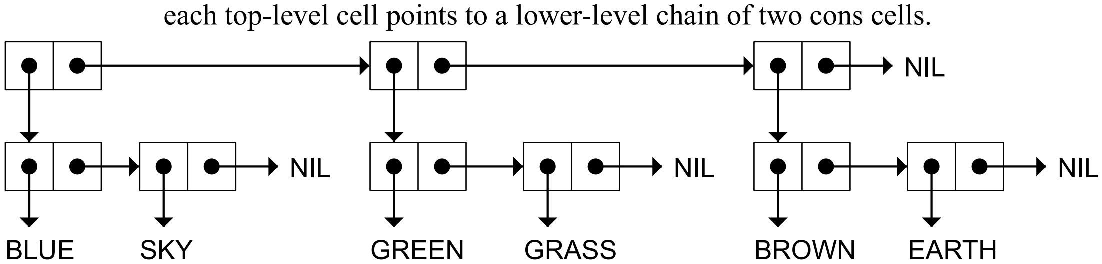
we can make a list of them by enclosing them within another pair of parentheses. The result is shown below. Note the importance of having two levels of parentheses: This is a list of *three lists*, not a list of six symbols.

```
((BLUE SKY) (GREEN GRASS) (BROWN EARTH))
```

We can display the three elements of this list vertically instead of horizontally if we choose. Spacing and indentation don't matter as long as the elements themselves and the parenthesization aren't changed. For example, the list of three lists could have been written like this:

```
((BLUE SKY)
    (GREEN GRASS)
       (BROWN EARTH))
```

The first element of this list is still (BLUE SKY). In cons cell notation, the list would be written as shown below. Since it has three elements, there are three cons cells in the top-level chain. Since each element is a list of two symbols, each top-level cell points to a lower-level chain of two cons cells.

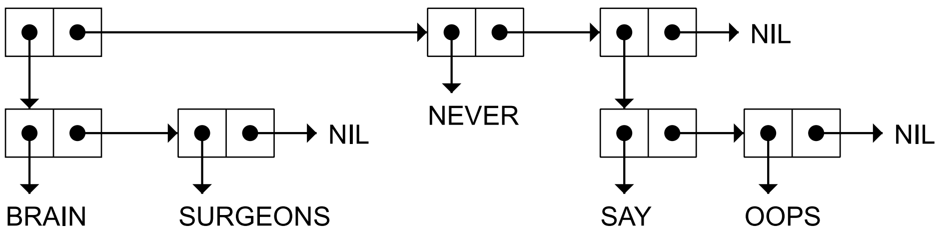


Lists that contain other lists are called **nested lists**. In parenthesis notation, a nested list has one or more sets of parentheses nested within the outermost pair. In cons cell notation, a nested list has at least one level of cons cells below the top-level chain. Lists that are not nested are called **flat lists**. A flat list has only a top-level cons cell chain.

Lists aren't always uniform in shape. Here's a nested list whose elements are a list, a symbol, and a list:

```
((BRAIN SURGEONS) NEVER (SAY OOPS))
```

You can see the pattern of parenthesization reflected in the cons cell diagram below.



Anything we write in parenthesis notation will have an equivalent description inside the computer as a cons cell structure—if the parentheses balance properly. If they don't balance, as in the malformed expression "(RED (GREEN BLUE," the computer cannot make a cons cell chain corresponding to that expression. The computer will respond with an error message if it reads an expression with unbalanced parentheses.

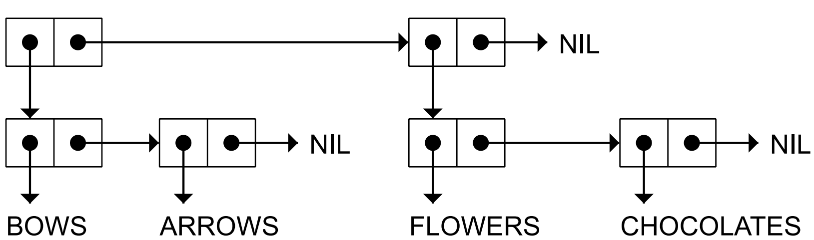
EXERCISES

2.2. Which of these are well-formed lists? That is, which ones have properly balanced parentheses?

```
(A B (C)
((A) (B))

A B )(C D)

(A (B (C))

(A (B (C))) (((A) (B)) (C))
```

2.3. Draw the cons cell representation of the list (PLEASE (BE MY) VALENTINE).

2.4. What is the parenthesis notation for this cons cell structure?

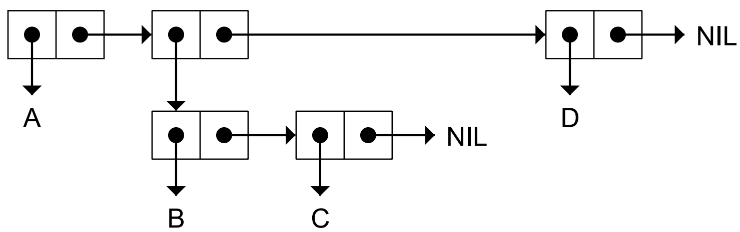


((BOWS ARROWS) (FLOWERS CHOCOLATES))

2.5 LENGTH OF LISTS

The length of a list is the number of elements it has, for example, the list (HI MOM) is of length two. But what about lists of lists? When a list is written in parenthesis notation, its elements are the things that appear inside only *one* level of parentheses. For example, the elements of the list (A (B C) D) are A, the list (B C), and D. The symbols B and C are not elements themselves, they are merely components of the element (B C).

Remember that the computer does not use parentheses internally. From the computer's point of view, the list (A (B C) D) contains three elements because its internal representation contains three top-level cons cells, like this:



So you see that the length of a list is independent of the complexity of its elements. The following lists all have exactly three elements, even though in some cases the elements are themselves lists. The three elements are underlined.

(<u>RED</u> <u>GREEN</u> <u>BLUE</u>)

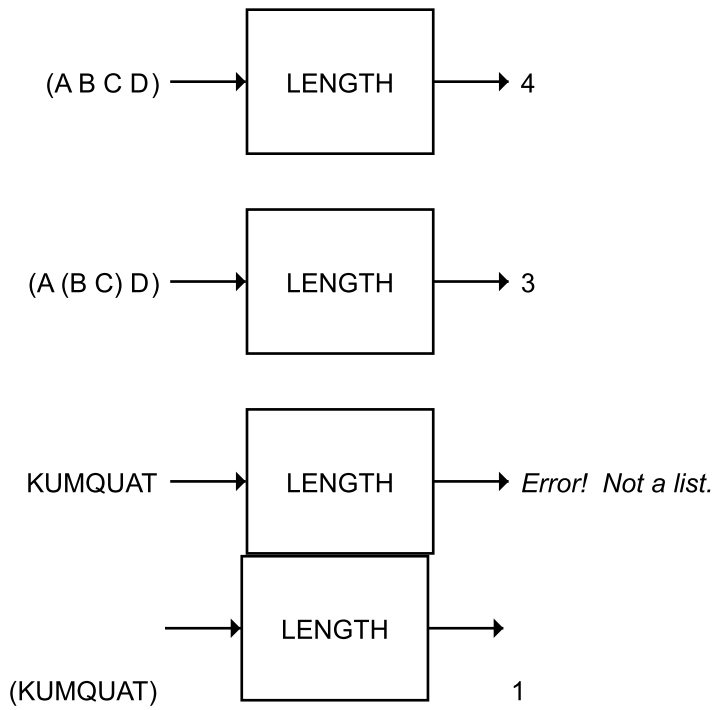
(<u>(BLUE SKY)</u> <u>(GREEN GRASS)</u> <u>(BROWN EARTH)</u>)

(<u>A</u> <u>(B X Y Z)</u> <u>C</u>)

(<u>FOO</u> <u>937</u> <u>(GLEEP GLORP)</u>)

(<u>ROY</u> <u>(TWO WHITE DUCKS)</u> <u>((MELTED) (BUTTER))</u>)

The primitive function LENGTH computes the length of a list. It is an error to give LENGTH a symbol or number as input.



EXERCISE

2.5. How many elements do each of the following lists have?

```
__________        (OPEN THE POD BAY DOORS HAL)

__________    ((OPEN) (THE POD BAY DOORS) HAL)

__________      ((1 2 3) (4 5 6) (7 8 9) (10 11 12))

__________      ((ONE) FOR ALL (AND (TWO (FOR ME))))

__________    ((Q SPADES) (7
               HEARTS)
                 (6 CLUBS)
                 (5 DIAMONDS)
                 (2 DIAMONDS))

__________      ((PENNSYLVANIA (THE KEYSTONE STATE))
                 (NEW-JERSEY (THE GARDEN STATE))
                 (MASSACHUSETTS (THE BAY STATE))
                 (FLORIDA (THE SUNSHINE STATE))
                 (NEW-YORK (THE EMPIRE STATE))
                 (INDIANA (THE HOOSIER STATE)))
```

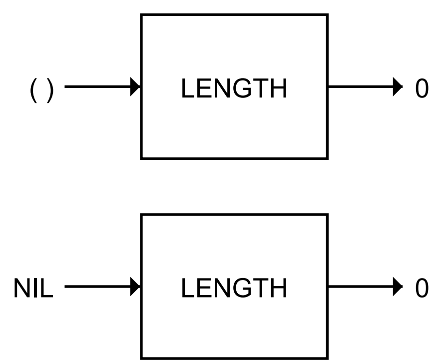
2.6 NIL: THE EMPTY LIST

A list of zero elements is called an **empty list**. It has no cons cells. It is written as an empty pair of parentheses:

```
()
```

Inside the computer the empty list is represented by the symbol NIL. This is a tricky point: the symbol NIL *is* the empty list; that's why it is used to mark the end of a cons cell chain. Since NIL and the empty list are identical, we are always free to write NIL instead of (), and vice versa. Thus (A NIL B) can also be written (A () B). It makes no difference which printed form is used; inside the computer the two are the same.

The length of the empty list is zero. Even though NIL is a symbol, it is still a valid input to LENGTH because NIL is also a list. NIL is the only thing that is both a symbol and a list.



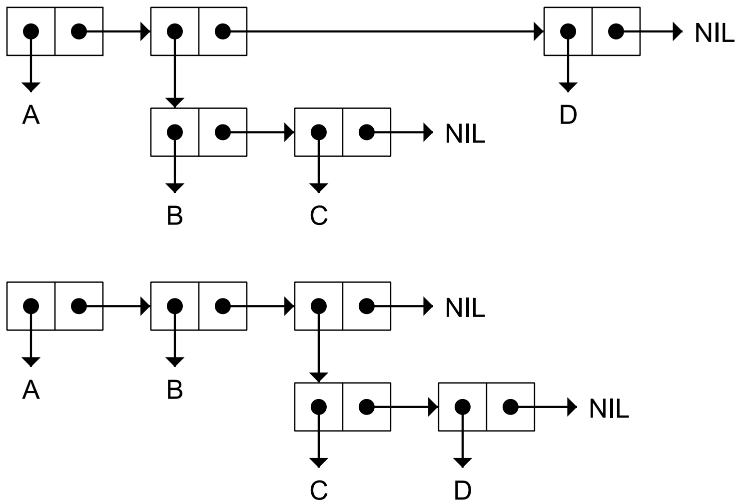
EXERCISE

2.6. Match each list on the left with a corresponding list on the right by substituting NIL for () wherever possible. Pay careful attention to levels of parenthesization.

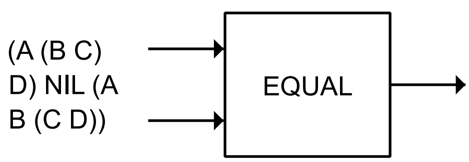
`()`	`((NIL))`
`(())`	`NIL`
`((()))`	`(NIL)`
`(() ())`	`(NIL (NIL))`
`(() (()))`	`(NIL NIL)`

2.7 EQUALITY OF LISTS

Two lists are considered EQUAL if their corresponding elements are EQUAL. Consider the lists (A (B C) D) and (A B (C D)) shown below.

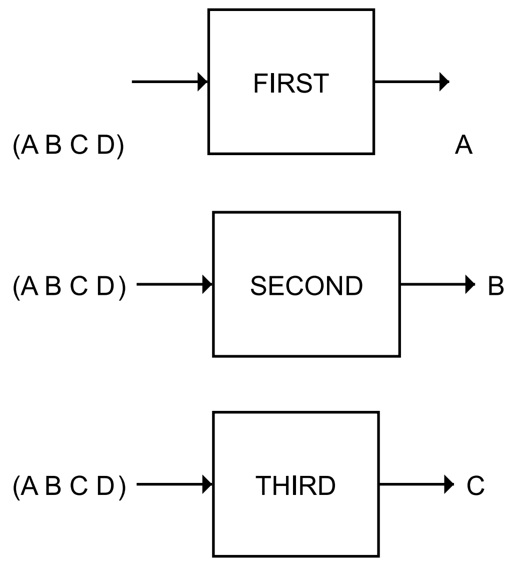


These two lists have the same number of elements (three), but they are not EQUAL. The second element of the former is (B C), while the second element of the latter is B. And neither list is equal to (A B C D), which has four elements. If two lists have different numbers of elements, they are never EQUAL.

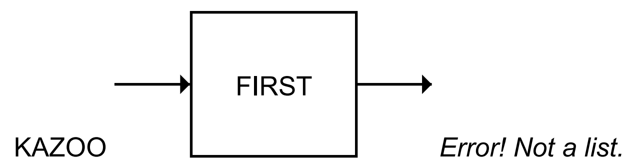


2.8 FIRST, SECOND, THIRD, AND REST

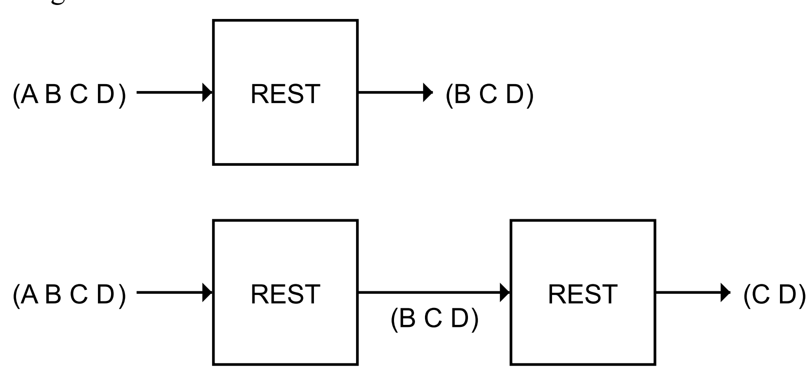
Lisp provides primitive functions for extracting elements from a list. The functions FIRST, SECOND, and THIRD return the first, second, and third element of their input, respectively.



It is an error to give these functions inputs that are not lists.

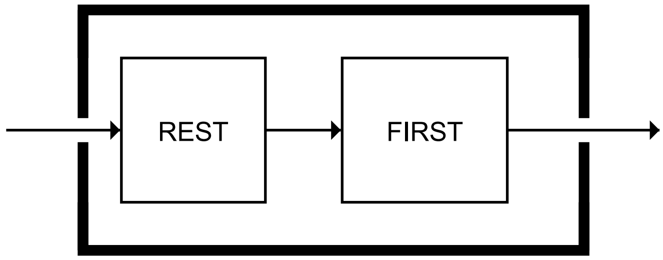


The REST function is the complement of FIRST: It returns a list containing everything *but* the first element.

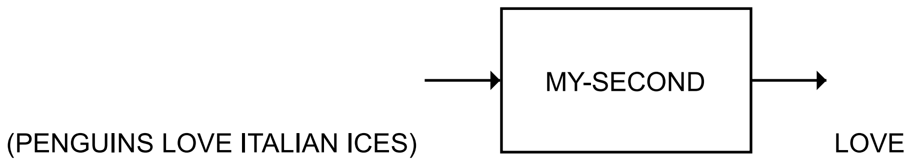


Using just FIRST and one or more RESTs, it is possible to construct our own versions of SECOND, THIRD, FOURTH, and so on. For example:

Definition of MY-SECOND:



If the input to MY-SECOND is (PENGUINS LOVE ITALIAN ICES), the REST function will output the list (LOVE ITALIAN ICES), and the FIRST element of that is LOVE.



EXERCISES

2.7. What goes on inside the MY-SECOND box when it is given the input (HONK IF YOU LIKE GEESE)?

2.8. Show how to write MY-THIRD using FIRST and two RESTs.

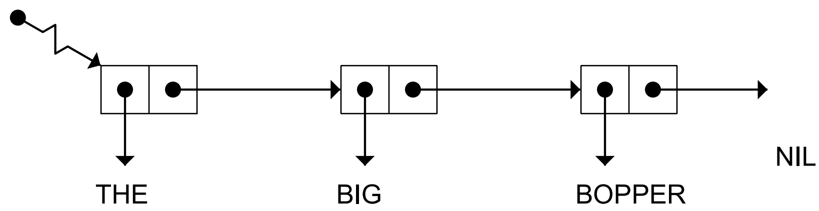
2.9. Show how to write MY-THIRD using SECOND.

2.9 FUNCTIONS OPERATE ON POINTERS

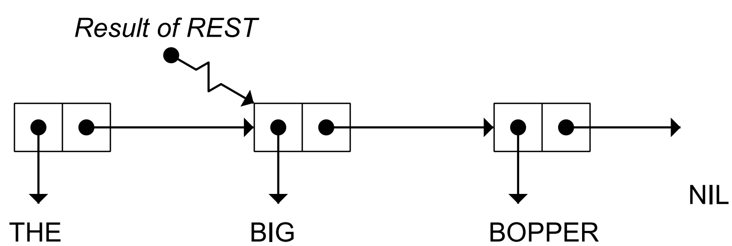
When we say that an object such as a list or symbol is an input to a function, we are speaking informally. Inside the computer, everything is done with pointers, so the real input to the function isn't the object itself, but a pointer to the object. Likewise, the result returned by a function is really a pointer.

Suppose (THE BIG BOPPER) is supplied as input to REST. What REST actually receives is a *pointer* to the first cons cell. This pointer is shown below, drawn as a wavy line. The line is wavy because the pointer's location isn't specified. In other words, it does not live inside any cons cell; it lives elsewhere in the computer. Computer scientists would say that the pointer lives "in a register" or "on the stack," but these details need not concern us.

Input to REST



The result returned by REST is a pointer to the second cons cell, which is the first cell of the list (BIG BOPPER). Where did this pointer come from? What REST did was extract the pointer from the right half of the first cons cell, and return that pointer as its result. So the result of REST is a pointer into the same cons cell chain as the input to REST. (See the figure below.) No new cons cells were created by REST when it returned (BIG BOPPER); all it did was extract and return a pointer.

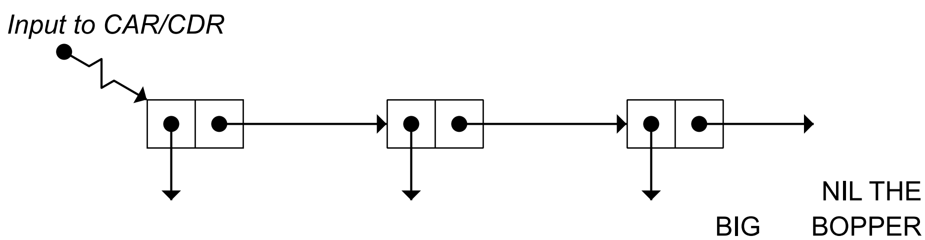


Note: We show a cons cell pointing to THE in the above figure to emphasize that the result is part of the same chain as the input to REST. But the cons cell that points to THE is not part of the result of REST. There is no way to reach this cell from the pointer returned by REST. (You can't follow pointers backward, only forward.)

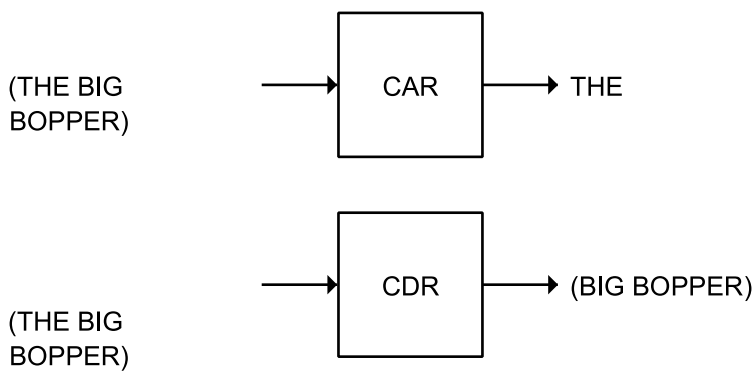
2.10 CAR AND CDR

By now you know that each half of a cons cell points to something. The two halves have obscure names. The left half is called the CAR, and the right half is called the CDR (pronounced "cou-der," rhymes with "good-er"). These names are relics from the early days of computing, when Lisp first ran on a machine called the IBM 704. The 704 was so primitive it didn't even have transistors—it used vacuum tubes. Each of its "registers" was divided into several components, two of which were the address portion and the decrement portion. Back then, the name CAR stood for Contents of Address portion of Register, and CDR stood for Contents of Decrement portion of Register. Even though these terms don't apply to modern computer hardware, Common Lisp still uses the acronyms CAR and CDR when referring to cons cells, partly for historical reasons, and partly because these names can be composed to form longer names such as CADR and CDDAR, as you will see shortly.

Besides naming the two halves of a cons cell, CAR and CDR are also the names of built-in Lisp functions that return whatever pointer is in the CAR or CDR half of the cell, respectively. Consider again the list (THE BIG BOPPER). When this list is used as input to a function such as CAR, what the function actually receives is not the list itself, but rather a pointer to the first cons cell:



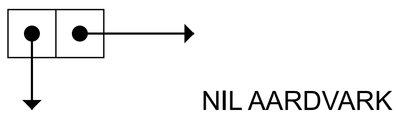
CAR follows this pointer to get to the actual cons cell and extracts the pointer sitting in the CAR half. So CAR returns as its result a pointer to the symbol THE. What does CDR return when given the same list as input?



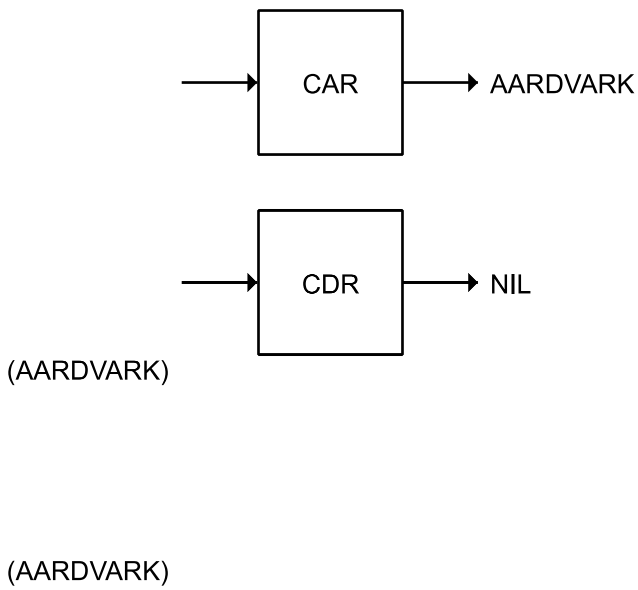
CDR follows the pointer to get to the cons cell, and extracts the pointer sitting in the CDR half, which it returns. So the result of CDR is a pointer to the list (BIG BOPPER). From this example you can see that CAR is the same as FIRST, and CDR is the same as REST. Lisp programmers usually prefer to express it the other way around: FIRST returns the CAR of a list, and REST returns the CDR.

2.10.1 The CDR of a Single-Element List

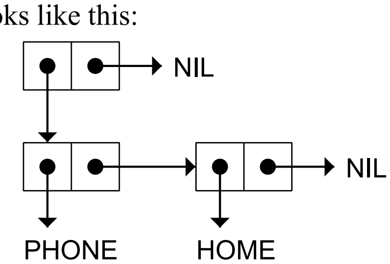
We saw previously that the list (AARDVARK) is not the same thing as the symbol AARDVARK. The list (AARDVARK) looks like this:



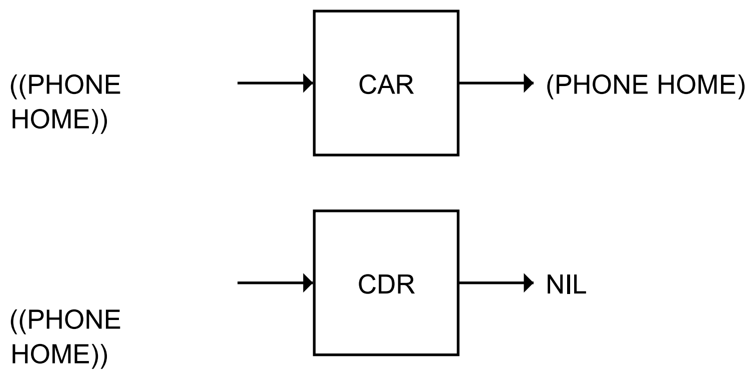
Since a list of length one is represented inside the computer as a single cons cell, the CDR of a list of length one is the list of length zero, NIL.



The list ((PHONE HOME)) has only one element. Remember that the elements of a list are the items that appear inside only one level of parentheses, in other words, the items pointed to by top-level cons cells. ((PHONE HOME)) looks like this:



Since the CAR and CDR functions extract their respective pointers from the first cons cell of a list, the CAR of ((PHONE HOME)) is (PHONE HOME), and the CDR is NIL.



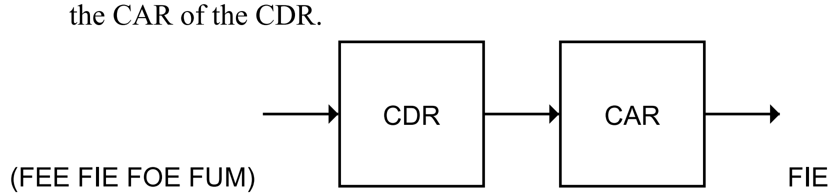
EXERCISES

2.10. Draw the cons cell representation of the list (((PHONE HOME))), which has three levels of parentheses. What is the CAR of this list? What is the CDR?

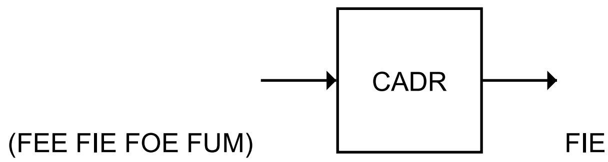
2.11. Draw the cons cell representation of the list (A (TOLL) ((CALL))).

2.10.2 Combinations of CAR and CDR

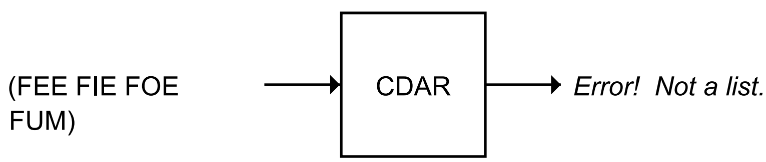
Consider the list (FEE FIE FOE FUM), the first element of which is FEE. The second element of this list is the FIRST of the REST, or, in our new terminology, the CAR of the CDR.

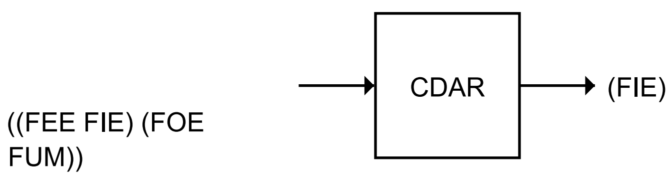


If you read the names of these function boxes from left to right, you'll read "CDR" and then "CAR." But since the input to the CAR function is the output of the CDR function, we say in English that we are computing "the CAR of the CDR" of the list, not the other way around. In Lisp, the CADR function is an abbreviation for "the CAR of the CDR." CADR is pronounced "kae-der."

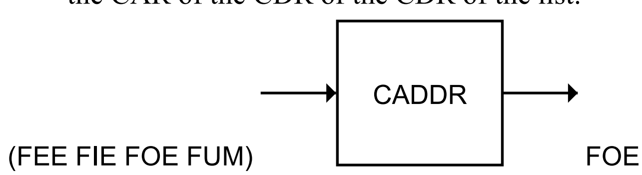


What would happen if we switched the A and the D? The CDAR ("coudar") function takes the CDR of the CAR of a list. The CAR of (FEE FIE FOE FUM) is FEE; if we try to take the CDR of that we get an error message. Obviously, CDAR doesn't work on lists of symbols. It works perfectly well on lists of lists, though.





The CADDR ("ka-dih-der") function returns the THIRD element of a list. (If you're having trouble with these strange names, see the pronunciation guide on page 48.) Once again, the name indicates how the function works: It takes the CAR of the CDR of the CDR of the list.



To really understand how CADDR works, you have to read the As and Ds from right to left. Starting with the list (FEE FIE FOE FUM), first take the CDR, yielding (FIE FOE FUM). Then take the CDR of that, which gives (FOE FUM). Finally take the CAR, which produces FOE.

Here's another way to look at CADDR. Start with the CDDR ("cou-dihder") function, which takes the CDR of the CDR, or the REST of the REST. The CDDR of (FEE FIE FOE FUM) is (FOE FUM), and the CAR of that is FOE. The CAR of the CDDR is the CADDR!

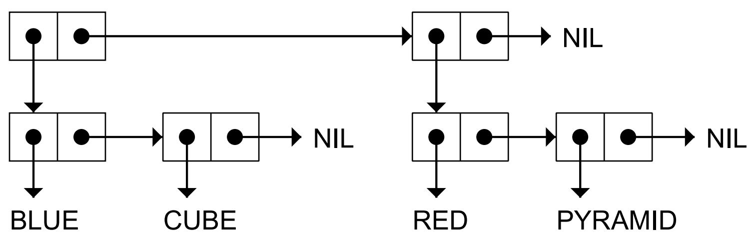
Common Lisp provides built-in definitions for all combinations of CAR and CDR up to and including four As and Ds in the function name. So CAADDR is built in, but not CAADDAR. Common Lisp also provides builtin definitions for FIRST through TENTH.

EXERCISE

2.12. What C...R name does Lisp use for the function that returns the fourth element of a list? How would you pronounce it?

2.10.3 CAR and CDR of Nested Lists

CAR and CDR can be used to take apart nested lists just as easily as flat ones. Let's see how we can get at the various components of the nested list ((BLUE CUBE) (RED PYRAMID)), which looks like this:

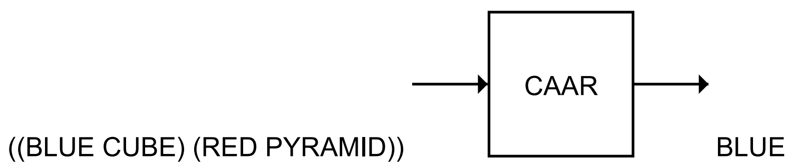


CAR/CDR Pronunciation Guide

Function	Pronunciation	Alternate Name
CAR	*kar*	FIRST
CDR	*cou-der*	REST
CAAR	*ka-ar*	
CADR	*kae-der*	SECOND
CDAR	*cou-dar*	
CDDR	*cou-dih-der*	
CAAAR	*ka-a-ar*	
CAADR	*ka-ae-der*	
CADAR	*ka-dar*	
CADDR	*ka-dih-der*	THIRD

CDAAR	*cou-da-ar*	
CDADR	*cou-dae-der*	
CDDAR	*cou-dih-dar*	
CDDDR	*cou-did-dih-der*	
CADDDR	*ka-dih-dih-der*	FOURTH
and so on		

The CAR of this list is (BLUE CUBE). To get to BLUE, we must take the CAR of the CAR. The CAAR function, pronounced ‘‘ka-ar,’’ does this.



What about getting to the symbol CUBE? Put your finger on the first cons cell of the list. Following the CAR pointer from the first cell takes us to the list (BLUE CUBE). Following the CDR pointer from that cell takes us to the list (CUBE), and following the CAR pointer from there takes us to the symbol CUBE. So CUBE is the CAR of the CDR of the CAR of the list, or, in short, the CADAR (‘‘ka-dar’’).

Here’s another way to think about it. The first element of the nested list is (BLUE CUBE), so CUBE is the SECOND of the FIRST of the list. This is the CADR of the CAR, which is precisely the CADAR.

Now let’s try to get to the symbol RED. RED is the FIRST of the SECOND of the list. You know by now that this is the CAR of the CADR. Putting the two names together yields CAADR, which is pronounced ‘‘ka-aeder.’’ Reading from right to left, put your finger on the first cons cell and follow the CDR pointer, then the CAR pointer, and then the CAR pointer again; you will end up at RED.

Let’s build a table of the steps to follow to get to PYRAMID:

Step	Result
start	`((BLUE CUBE) (RED PYRAMID))`
`C...DR`	`((RED PYRAMID))`
`C..ADR`	`(RED PYRAMID)`
`C.DADR`	`(PYRAMID) CADADR`
	`PYRAMID`

EXERCISES

2.13. Write down tables similar to the one above to illustrate how to get to each word in the list (((FUN)) (IN THE) (SUN)).

2.13

step	result
Start	(((FUN)) (IN THE) (SUN))
CAR	((FUN))
CAAR	(FUN)
CAADR	FUN
CADR	(IN THE)
CADAR	IN
CADADR	THE
CDDR	((SUN))
CADDR	(SUN)
CADDDR	SUN

2.14. What would happen if you tried to explain the operation of the CAADR function on the list ((BLUE CUBE) (RED PYRAMID) by reading the As and Ds from left to right instead of from right to left?

2.15. Using the list `((A B) (C D) (E F))`, fill in the missing parts of this table.

Function	Result
CAR	(A B)
CDDR	~~((E F))~~
CADR	(C D)
CDAR	B
CDAR	B
CDDAR	D
CAAR	A
CDADDR	F
CADADDR	F

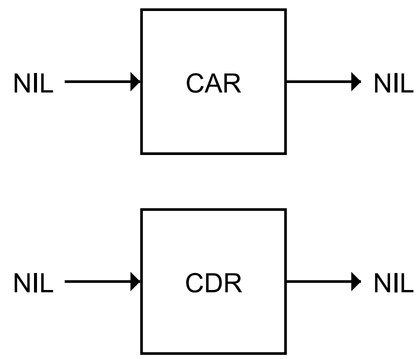
2.16. What does CAAR do when given the input (FRED NIL)?

CAR extracts the first element of the list (FRED NIL), which is FRED

CAAR attempts to apply CAR again to the result. Since FRED is not a list, the result is an error

2.10.4 CAR and CDR of NIL

Here is another interesting fact about NIL: The CAR and CDR of NIL are defined to be NIL. At this point it's probably not obvious why this should be so. In some earlier Lisp dialects it was actually an error to try to take the CAR or CDR of NIL. But experience shows that defining the CAR and CDR of NIL to be NIL has useful consequences in certain programming situations. You'll see some examples in later chapters.


NIL
CAR
NIL
NIL
CDR
NIL

6. In what language is most of UNIX written?

Most of UNIX is written in C

7. What is the disadvantage of having too many features in a language?

Increased complexity, making the language harder to learn and use.

Reduced readability, as multiple ways to perform the same task may confuse programmers.

Potential for misuse of features, leading to harder-to-maintain code.

8. How can user-defined operator overloading harm the readability of a program?

Operators might perform unexpected or non-intuitive actions based on the programmer's implementation.

The same operator might have different meanings in different contexts, requiring readers to inspect the implementation for clarity.

It can obscure the intent of the code, especially if overloading is done inconsistently.

9. What is one example of a lack of orthogonality in the design of C?

Arrays and pointers: While arrays can be treated like pointers in some contexts such as when being passed to functions, they are not fully interchangeable. For instance, you cannot assign arrays like pointers.

Mixing data types: Some operations that work with certain types do not work with others such as structs cannot be directly compared using ==

10. What language used orthogonality as a primary design criterion?

The programming language ALGOL 68 used orthogonality as a primary design criterion. Orthogonality in ALGOL 68 ensures that language features can be combined freely without unexpected restrictions or behaviors

11. What primitive control statement is used to build more complicated control statements in languages that lack them?

The goto statement is often used as a primitive control statement to build more complex control structures like loops or conditionals

12. What does it mean for a program to be reliable?

Proper handling of errors and exceptions.

Adherence to specifications.

Avoidance of undefined or unpredictable behavior.

13. Why is type checking the parameters of a subprogram important?

Correctness: It prevents passing incompatible data types to a function or procedure, reducing runtime errors.

Readability: It helps clarify the expected input, making the code easier to understand.

Security: It minimizes the risk of vulnerabilities caused by unintended operations on incorrect data types.

14. What is aliasing?

Aliasing occurs when two or more variables reference the same memory location. Changes to one variable affect the others, potentially causing unintended side effects and making debugging more difficult

15. What is exception handling?

Aliasing occurs when two or more variables reference the same memory location. Changes to one variable affect the others, potentially causing unintended side effects and making debugging and reasoning about the code more difficult.

16. Why is readability important to writability?

Code that is easy to read and understand allows developers to build upon it more effectively.

It reduces the likelihood of introducing errors when modifying or extending the code.

Clear and readable code makes it easier for teams to collaborate and maintain consistency in programming practices.

20. What two programming language deficiencies were discovered as a result of the research in software development in the 1970s?

Inadequate support for data abstraction: Early languages lacked mechanisms for creating abstract data types to encapsulate and manipulate data effectively

Poor control structures: Many languages relied on primitive constructs like goto, making it harder to write structured and maintainable programs

21. What are the three fundamental features of an object-oriented programming language?

Encapsulation: Bundling data and methods together in objects and restricting direct access to some components

Inheritance: Enabling new classes to inherit properties and behaviors from existing classes

Polymorphism: Allowing objects of different types to be treated uniformly through a common interface, typically achieved via method overriding or overloading

22. What language was the first to support the three fundamental features of object-oriented programming?

The first language to support all three fundamental features of object-oriented programming was Smalltalk

23. What is an example of two language design criteria that are in direct conflict with each other?

Reliability requires extensive error checking, type safety, and runtime validations, which can add overhead

Efficiency aims for faster execution and minimal resource usage, often at the cost of reduced safety checks

24. What are the three general methods of implementing a programming language?

Compilation: Translating the source code into machine code before execution

Pure Interpretation: Executing the source code directly by an interpreter without prior translation

Hybrid Implementation: Combining compilation and interpretation, typically by compiling to intermediate code, then interpreting it

25. Which produces faster program execution, a compiler or a pure interpreter?

A compiler produces faster program execution because the code is translated into machine code ahead of time. In contrast, a pure interpreter incurs overhead during execution as it processes and executes the source code line by line

29. What are the advantages in implementing a language with a pure interpreter?

Improved Debugging Capabilities

Platform Independence

Interactive Execution

Flexibility

Dynamic Features