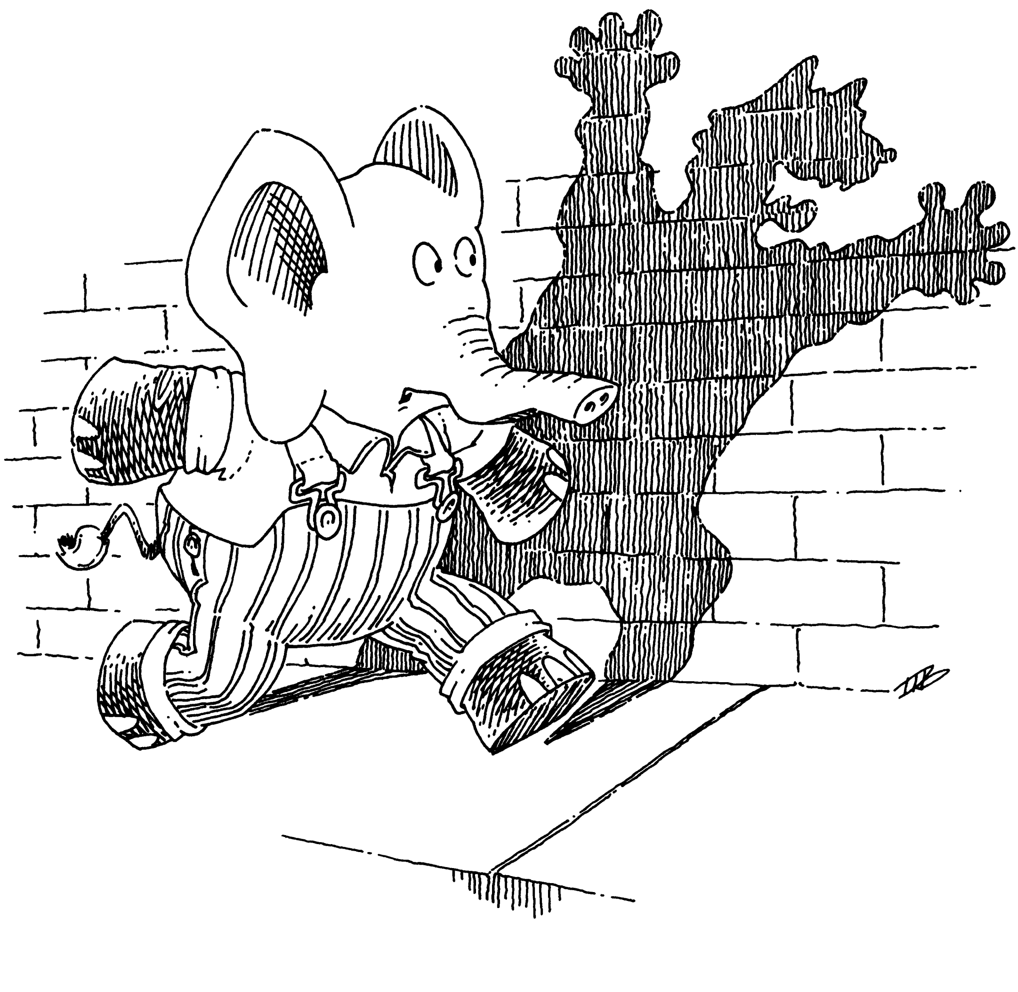
6.
Shadows



Is 1 an arithmetic expression?	Yes.
Is 3 an arithmetic expression?	Yes, of course.
Is 1 + 3 an arithmetic expression?	Yes!
Is 1 + 3 × 4 an arithmetic expression?	Definitely.
Is cookie an arithmetic expression?	Yes. Are you almost ready for one?
And, what about 3 ↑ y + 5	Yes.
What is an arithmetic expression in your words?	In ours: "For the purpose of this chapter, an arithmetic expression is either an atom (including numbers), or two arithmetic expressions combined by +, ×, or ↑."
What is (**quote** a)	a.
What is (**quote** +)	The atom +, not the operation +.
What does (**quote** ×) stand for?	The atom ×, not the operation ×.
Is (*eq?* (**quote** a) *y*) true or false where *y* is a	True.
Is (*eq?* *x* *y*) true or false where *x* is a and *y* is a	That's the same question again. And the answer is still true.

Is (n + 3) an arithmetic expression?	Not really, since there are parentheses around n + 3. Our definition of arithmetic expression does not mention parentheses.
Could we think of (n + 3) as an arithmetic expression?	Yes, if we keep in mind that the parentheses are not really there.
What would you call (n + 3)	We call it a representation for n + 3.
Why is (n + 3) a good representation?	Because 1. (n + 3) is an S-expression. It can therefore serve as an argument for a function. 2. It structurally resembles n + 3.
True or false: (*numbered? x*) where *x* is 1	True.
How do you represent $3 + 4 \times 5$	(3 + (4 × 5)).
True or false: (*numbered? y*) where *y* is (3 + (4 ↑ 5))	True.
True or false: (*numbered? z*) where *z* is (2 × sausage)	False, because sausage is not a number.
What is *numbered?*	It is a function that determines whether a representation of an arithmetic expression contains only numbers besides the +, ×, and ↑.

Now can you write a skeleton for *numbered?*

```
(define numbered?
  (lambda (aexp)
    (cond
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ ))))
```

is a good guess.

What is the first question?

(*atom? aexp*).

What is (*eq?* (*car* (*cdr aexp*)) (**quote** +))

It is the second question.

Can you guess the third one?

(*eq?* (*car* (*cdr aexp*)) (**quote** ×)) is perfect.

And you must know the fourth one.

(*eq?* (*car* (*cdr aexp*)) (**quote** ↑)), of course.

Should we ask another question about *aexp*

No! So we could replace the previous question by **else**.

Why do we ask four, instead of two, questions about arithmetic expressions? After all, arithmetic expressions like (1 + 3) are lats.

Because we consider (1 + 3) as a representation of an arithmetic expression in list form, not as a list itself. And, an arithmetic expression is either a number, or two arithmetic expressions combined by +, ×, or ↑.

Now you can almost write *numbered?*

Here is our proposal:

```
(define numbered?
  (lambda (aexp)
    (cond
      ((atom? aexp) (number? aexp))
      ((eq? (car (cdr aexp)) (quote +))
       ...)
      ((eq? (car (cdr aexp)) (quote ×))
       ...)
      ((eq? (car (cdr aexp)) (quote ↑))
       ...))))
```

Why do we ask (*number? aexp*) when we know that *aexp* is an atom?

Because we want to know if all arithmetic expressions that are atoms are numbers.

What do we need to know if the *aexp* consists of two arithmetic expressions combined by +

We need to find out whether the two subexpressions are numbered.

In which position is the first subexpression?

It is the *car* of *aexp*.

In which position is the second subexpression?

It is the *car* of the *cdr* of the *cdr* of *aexp*.

So what do we need to ask?

(*numbered?* (*car aexp*)) and (*numbered?* (*car* (*cdr* (*cdr aexp*)))). Both must be true.

What is the second answer?

```
(and (numbered? (car aexp))
  (numbered? (car (cdr (cdr aexp)))))
```

Try *numbered?* again.

```
(define numbered?
  (lambda (aexp)
    (cond
      ((atom? aexp) (number? aexp))
      ((eq? (car (cdr aexp)) (quote +))
       (and (numbered? (car aexp))
         (numbered?
           (car (cdr (cdr aexp))))))
      ((eq? (car (cdr aexp)) (quote ×))
       (and (numbered? (car aexp))
         (numbered?
           (car (cdr (cdr aexp))))))
      ((eq? (car (cdr aexp)) (quote ↑))
       (and (numbered? (car aexp))
         (numbered?
           (car (cdr (cdr aexp)))))))))
```

Since *aexp* was already understood to be an arithmetic expression, could we have written ***numbered?*** in a simpler way?

Yes:

```
(define numbered?
  (lambda (aexp)
    (cond
      ((atom? aexp) (number? aexp))
      (else
        (and (numbered? (car aexp))
          (numbered?
            (car (cdr (cdr aexp)))))))))
```

Why can we simplify?

Because we know we've got the function right.

What is (*value* *u*)
where ***u*** is 13

13.

(*value* *x*)
where
  x is (1 + 3)

4.

(*value y*) where *y* is (1 + (3 ↑ 4))	82.
(*value z*) where *z* is cookie	No answer.
(*value nexp*) returns what we think is the natural value of a numbered arithmetic expression.	We hope.
How many questions does *value* ask about *nexp*	Four.

Now, let's attempt to write *value*

```
(define value
  (lambda (nexp)
    (cond
      ((atom? nexp) ...)
      ((eq? (car (cdr nexp)) (quote +))
       ...)
      ((eq? (car (cdr nexp)) (quote ×))
       ...)
      (else ...))))
```

What is the natural value of an arithmetic expression that is a number?	It is just that number.
What is the natural value of an arithmetic expression that consists of two arithmetic expressions combined by +	If we had the natural value of the two subexpressions, we could just add up the two values.
Can you think of a way to get the value of the two subexpressions in (1 + (3 × 4))	Of course, by applying *value* to 1, and applying *value* to (3 × 4).

And in general?

By recurring with *value* on the subexpressions.

The Seventh Commandment

Recur on the *subparts* that are of the same nature:

- **On the sublists of a list.**
- **On the subexpressions of an arithmetic expression.**

Give *value* another try.

```
(define value
  (lambda (nexp)
    (cond
      ((atom? nexp) nexp)
      ((eq? (car (cdr nexp)) (quote +))
       (+ (value (car nexp))
          (value (car (cdr (cdr nexp))))))
      ((eq? (car (cdr nexp)) (quote ×))
       (× (value (car nexp))
          (value (car (cdr (cdr nexp))))))
      (else
        (↑ (value (car nexp))
           (value
             (car (cdr (cdr nexp)))))))))
```

Can you think of a different representation of arithmetic expressions?

There are several of them.

Could (3 4 +) represent 3 + 4

Yes.

Could (+ 3 4)

Yes.

Or (plus 3 4)

Yes.

Is (+ (× 3 6) (↑ 8 2)) a representation of an arithmetic expression?	Yes.

Try to write the function *value* for a new kind of arithmetic expression that is either:

- — a number
- — a list of the atom + followed by two arithmetic expressions,
- — a list of the atom × followed by two arithmetic expressions, or
- — a list of the atom ↑ followed by two arithmetic expressions.

What about

```
(define value
  (lambda (nexp)
    (cond
      ((atom? nexp) nexp)
      ((eq? (car nexp) (quote +))
       (+ (value (cdr nexp))
          (value (cdr (cdr nexp)))))
      ((eq? (car nexp) (quote ×))
       (× (value (cdr nexp))
          (value (cdr (cdr nexp)))))
      (else
        (↑ (value (cdr nexp))
           (value (cdr (cdr nexp))))))))
```

You guessed it.	It's wrong.
Let's try an example.	(+ 1 3).
(*atom?* *nexp*) where *nexp* is (+ 1 3)	No.
(*eq?* (*car* *nexp*) (**quote** +)) where *nexp* is (+ 1 3)	Yes.
And now recur.	Yes.
What is (*cdr* *nexp*) where *nexp* is (+ 1 3)	(1 3).

(1 3) is not our representation of an arithmetic expression.

No, we violated The Seventh Commandment. (1 3) is not a subpart that is a representation of an arithmetic expression! We obviously recurred on a list. But remember, not all lists are representations of arithmetic expressions. We have to recur on subexpressions.

How can we get the first subexpression of a representation of an arithmetic expression?

By taking the *car* of the *cdr*.

Is (*cdr* (*cdr nexp*)) an arithmetic expression where
nexp is (+ 1 3)

No, the *cdr* of the *cdr* is (3), and (3) is not an arithmetic expression.

Again, we were thinking of the list (+ 1 3) instead of the representation of an arithmetic expression.

Taking the *car* of the *cdr* of the *cdr* gets us back on the right track.

What do we mean if we say the *car* of the *cdr* of *nexp*

The first subexpression of the representation of an arithmetic expression.

Let's write a function *1st-sub-exp* for arithmetic expressions.

```
(define 1st-sub-exp
  (lambda (aexp)
    (cond
      (else (car (cdr aexp))))))
```

Why do we ask **else**

Because the first question is also the last question.

Can we get by without (**cond** ...) if we don't need to ask questions?

Yes, remember one-liners from chapter 4.

```
(define 1st-sub-exp
  (lambda (aexp)
    (car (cdr aexp))))
```

Write *2nd-sub-exp* for arithmetic expressions.

```
(define 2nd-sub-exp
  (lambda (aexp)
    (car (cdr (cdr aexp)))))
```

Finally, let's replace (*car nexp*) by (*operator nexp*)

```
(define operator
  (lambda (aexp)
    (car aexp)))
```

Now write *value* again.

```
(define value
  (lambda (nexp)
    (cond
      ((atom? nexp) nexp)
      ((eq? (operator nexp) (quote +))
       (+ (value (1st-sub-exp nexp))
          (value (2nd-sub-exp nexp))))
      ((eq? (operator nexp) (quote ×))
       (× (value (1st-sub-exp nexp))
          (value (2nd-sub-exp nexp))))
      (else
        (↑ (value (1st-sub-exp nexp))
           (value (2nd-sub-exp nexp)))))))
```

Can we use this *value* function for the first representation of arithmetic expressions in this chapter?

Yes, by changing *1st-sub-exp* and *operator*.

Do it!

```
(define 1st-sub-exp
  (lambda (aexp)
    (car aexp)))
```

```
(define operator
  (lambda (aexp)
    (car (cdr aexp))))
```

Wasn't this easy?	Yes, because we used help functions to hide the representation.

The Eighth Commandment

Use help functions to abstract from representations.

Have we seen representations before?	Yes, we just did not tell you that they were representations.
For what entities have we used representations?	Truth-values! Numbers!
Numbers are representations?	Yes. For example **4** stands for the concept four. We chose that symbol because we are accustomed to arabic representations.
What else could we have used?	(() () () ()) would have served just as well. What about ((((())))))? How about (I V)?
Do you remember how many primitives we need for numbers?	Four: *number?*, *zero?*, *add1*, and *sub1*.
Let's try another representation for numbers. How shall we represent zero now?	() is our choice.
How is one represented?	(()).
How is two represented?	(() ()).

Got it? What's three?

Three is (() () ()).

Write a function to test for zero.

```
(define sero?
  (lambda (n)
    (null? n)))
```

Can you write a function that is like ***add1***

```
(define edd1
  (lambda (n)
    (cons (quote ()) n)))
```

What about ***sub1***

```
(define zub1
  (lambda (n)
    (cdr n)))
```

Is this correct?

Let's see.

What is (*zub1 n*) where *n* is ()

No answer, but that's fine.
— Recall The Law of Cdr.

Rewrite ✢ using this representation.

```
(define ✢
  (lambda (n m)
    (cond
      ((sero? m) n)
      (else (edd1 (✢ n (zub1 m)))))))
```

Has the definition of ✢ changed?

Yes and no. It changed, but only slightly.

Recall *lat?*

Easy:

```
(define lat?
  (lambda (l)
    (cond
      ((null? l) #t)
      ((atom? (car l)) (lat? (cdr l)))
      (else #f))))
```

But why did you ask?

Do you remember what the value of (*lat? ls*) is where *ls* is (1 2 3)

#t, of course.

What is (1 2 3) with our new numbers?

((()) (()()) (()()())).

What is (*lat? ls*) where *ls* is ((()) (()()) (()()()))

It is very false.

Is that bad?

You must beware of shadows.