

## 2. Recursive types and recursive expressions

### 1 Representing propositional formulas in Haskell

You may have seen recursive definitions like (1) before.

- (1) The set  $\mathcal{F}$  of propositional formulas is defined as the smallest set such that:
- a.  $\mathbf{T} \in \mathcal{F}$
  - b.  $\mathbf{F} \in \mathcal{F}$
  - c. if  $\phi \in \mathcal{F}$ , then  $\neg\phi \in \mathcal{F}$
  - d. if  $\phi \in \mathcal{F}$  and  $\psi \in \mathcal{F}$ , then  $(\phi \wedge \psi) \in \mathcal{F}$
  - e. if  $\phi \in \mathcal{F}$  and  $\psi \in \mathcal{F}$ , then  $(\phi \vee \psi) \in \mathcal{F}$

We can define a Haskell type to represent these formulas in a way that very closely matches this definition:

- (2) `data Form = T | F | Neg Form | Cnj Form Form | Dsj Form Form deriving Show`

The five *constructors* here (`T`, `F`, `Neg`, `Cnj` and `Dsj`) correspond to the five ways to be a formula (given in (1)).

So we can use the Haskell expression `Dsj (Neg T) (Cnj F T)` to represent the formula  $(\neg\mathbf{T} \vee (\mathbf{F} \wedge \mathbf{T}))$ .

We can write a Haskell function to, for example, remove all negations from a formula, like this:

- (3) 

```
removeNegs = \form -> case form of
  T -> T
  F -> F
  Neg phi -> removeNegs phi
  Cnj phi psi -> Cnj (removeNegs phi) (removeNegs psi)
  Dsj phi psi -> Dsj (removeNegs phi) (removeNegs psi)
```

The standard denotations of these formulas is defined as follows:

- (4) a.  $\llbracket \mathbf{T} \rrbracket$  is true  
b.  $\llbracket \mathbf{F} \rrbracket$  is false  
c.  $\llbracket \neg\phi \rrbracket$  is true if  $\llbracket \phi \rrbracket$  is false; and is false otherwise  
d.  $\llbracket \phi \wedge \psi \rrbracket$  is true if both  $\llbracket \phi \rrbracket$  is true and  $\llbracket \psi \rrbracket$  is true; otherwise,  $\llbracket \phi \wedge \psi \rrbracket$  is false  
e.  $\llbracket \phi \vee \psi \rrbracket$  is true if either  $\llbracket \phi \rrbracket$  is true or  $\llbracket \psi \rrbracket$  is true; otherwise,  $\llbracket \phi \vee \psi \rrbracket$  is false

A Haskell function to calculate the denotation of a formula follows exactly the same pattern as the `removeNegs` function above. This is because pretty much *anything* you might want to “do with” a formula follows this pattern: there’s no sharp distinction to be made, computationally, between calculating the denotation of a formula and calculating the negation-free-version of a formula.

Recursion

- fundamental in languages like Haskell

- less central in using languages like C/C++/Python/Java

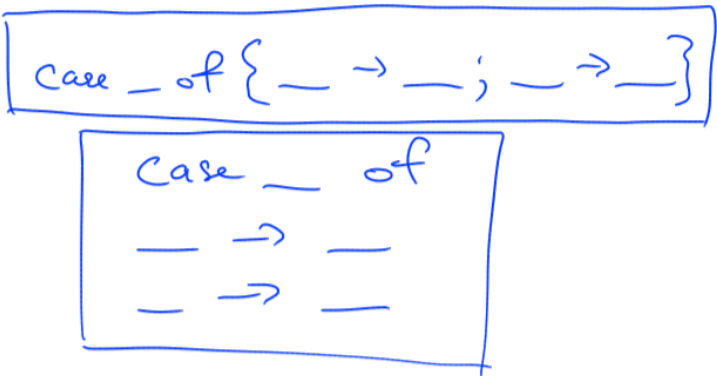
Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- fundamental to (computational) linguistics

grammars are devices which work recursively on strings/trees/...



Form  $\rightarrow \dots$

## 2 A very simple recursive type

The `Form` type that we defined above is an example of a recursive type. For a deeper understanding of exactly how these work it's useful to look at an extremely simple example of a recursive type, namely the type `Numb` which is defined like this:

```
(5) data Numb = Z | S Numb deriving Show
```

It's just like the `Result` type from last week, except that the thing “inside” it is another thing of the same type (whereas the thing “inside” a `Result` is a `Shape`).

We can write some simple functions that work with this type.

```
isZero = \n -> case n of {Z -> True; S n' -> False}

isOne = \n -> case n of
  Z -> False
  S n' -> case n' of {Z -> True; S n'' -> False}

lessThanTwo = \n -> case n of
  Z -> True
  S n' -> case n' of {Z -> True; S n'' -> False}
```

Peano arithmetic

data Result = Draw | Win Shape  
data Form = F | Neg Form

zero Z  
one S Z  
two S (S Z)  
three S (S (S Z))  
four S (S (S (S Z)))

The evaluation rules for `Numb` follow the pattern we saw with the `Result` type.

```
(1) case Z of {Z -> e1; S v -> e2} ==> e1
case (S e) of {Z -> e1; S v -> e2} ==> [e/v]e2
```

Here's how evaluation proceeds if we use the `lessThanTwo` function above on `S Z` (i.e. on “the number one”).

```
lessThanTwo (S Z)
==> (\n -> case n of {Z -> True; S n' -> case n' of {Z -> True; S n'' -> False}}) (S Z)
==> case (S Z) of {Z -> True; S n' -> case n' of {Z -> True; S n'' -> False}}
==> [Z/n']case n' of {Z -> True; S n'' -> False} = case Z of {Z -> True; S n'' -> False}
==> True
```

### Recursive expressions

Each of the functions for working with `Numbs` above only looks a *fixed depth* into the structure of its argument. For this reason, each of these functions is insensitive to distinctions between the various `Numbs` that differ in ways that one can only see by looking beyond that fixed depth. To write functions that are sensitive to all of the distinctions between the various `Numbs`, we need a way to express the idea of “keeping going as far as necessary”.

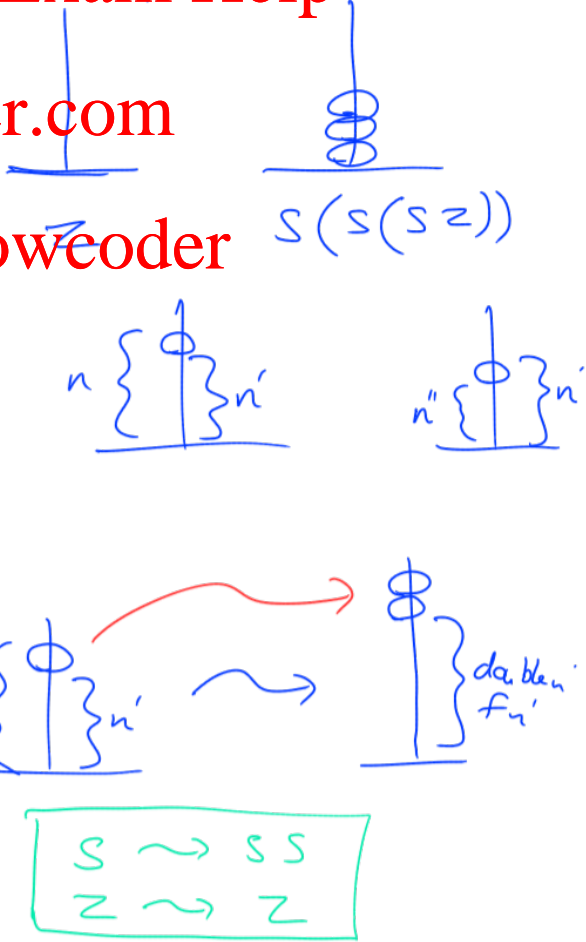
This requires something new: recursively defined *expressions*.

We've already seen expressions of the form `let v = e1 in e2`, where the variable `v` is bound within `e2`. What we left aside last week is that the variable `v` is actually bound inside `e1` too.<sup>1</sup>

For example, we can define and use a function for doubling `Numbs` like this:

```
(2) let f = \n -> case n of {Z -> Z; S n' -> S (S (f n'))} in f (S (S (S Z)))
```

<sup>1</sup>In this respect, Haskell is different from OCaml: in OCaml an expression of the form `let v = e1 in e2` only binds `v` within `e2`, and if you want it bound in `e1` as well you need to make this explicit by using `let rec`. In Haskell, every `let` works like OCaml's `let rec`.



Notice that `f` appears in between the equals sign and the `in`.

How does an expression like this get evaluated? It actually requires a couple of special tricks<sup>2</sup> under the hood, but to a good first approximation (good enough for us throughout this course) we can think of it like this:

```
(3) let f = \n -> case n of {Z -> Z; S n' -> S (S (f n'))} in f (S (S (S Z)))
=> (\n -> case n of {Z -> Z; S n' -> S (S (f n'))}) (S (S (S Z)))
=> case (S (S (S Z))) of {Z -> Z; S n' -> S (S (f n'))}
=> [S (S Z)/n,] S (S (f n')) = S (S (f (S (S Z))))
~ S (S ((\n -> case n of {Z -> Z; S n' -> S (S (f n'))}) (S (S Z))))
=> S (S (case (S (S Z)) of {Z -> Z; S n' -> S (S (f n'))}))
=> S (S (S (S (f (S Z)))))
~ S (S (S (S ((\n -> case n of {Z -> Z; S n' -> S (S (f n'))}) (S Z)))))
=> S (S (S (S (case (S Z) of {Z -> Z; S n' -> S (S (f n'))}))))
=> S (S (S (S (S (S (f Z)))))
~ S (S (S (S (S (S ((\n -> case n of {Z -> Z; S n' -> S (S (f n'))}) Z)))))
=> S (S (S (S (S (S (case Z of {Z -> Z; S n' -> S (S (f n'))}))))))
=> S (S (S (S (S (S Z)))))
```

It's a good exercise to think about *why* this sequence of evaluation steps doesn't follow from the evaluation rules for `let`-expressions that we introduced last week (i.e. what's being glossed over by the `~` symbol above).

3 Another recursive type: lists/strings

We can represent lists of, say, integers, using a very similar structure to what we used for `Numb`.

```
(6) data IntList = Empty | NonEmpty Int IntList deriving Show
```

For example, the list containing 5 followed by 7 followed by 2 (and nothing else) would be represented as:

```
(7) NonEmpty 5 (NonEmpty 7 (NonEmpty 2 Empty))
```

Using this type we could write a function to calculate, say, the sum of such a list of integers like this:

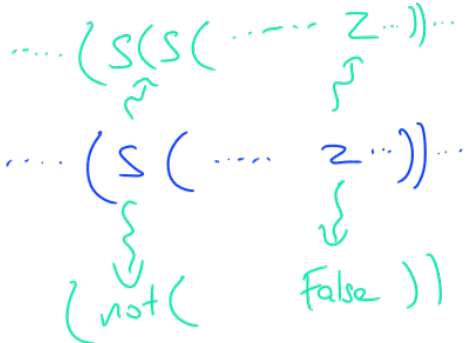
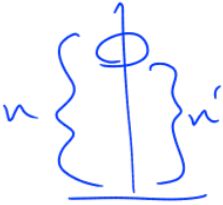
```
(8) total = \l -> case l of
    Empty -> 0
    NonEmpty x rest -> x + total rest
```

Haskell has a built-in type to represent lists, which uses some compact syntax. The compact syntax is convenient, but it can obscure the fact that this built-in type actually has exactly the same kind of recursive structure as this `IntList` type. Using this built-in type, we write the list containing 5 then 7 then 2 as in (9) instead of (7); and we write the function for summing a list as in (10) instead of (8).

```
(9) 5 : (7 : (2 : []))
```

```
(10) total = \l -> case l of
    [] -> 0
    x : rest -> x + total rest
```

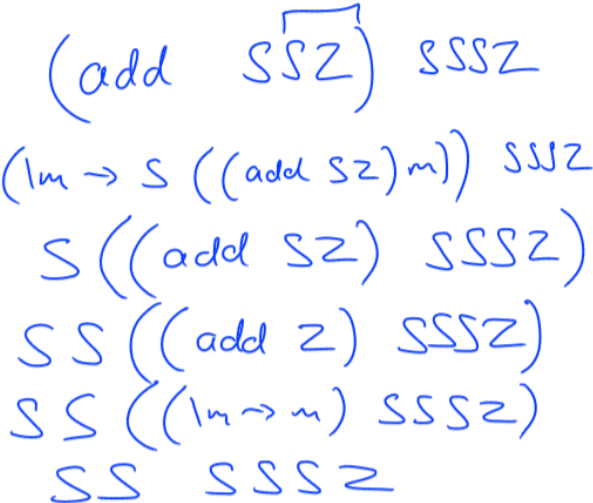
<sup>2</sup>If you're curious, look up recursion and *fixed point* operators. It's closely related to the idea of "boosting" a partially-working function, which is discussed in the appendix.



Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder



[] Empty  
: NonEmpty

The differences are just that:

- the built-in type uses `[]` instead of `Empty`;
- the built-in type uses the colon (“cons”) instead of `NonEmpty`; and
- the colon is written *between* its two arguments, unlike `NonEmpty` and other constructors we’ve seen.

And, as an added convenient-but-potentially-misleading bonus, we can also write `[5,7,2]` in place of `5 : (7 : (2 : []))`.

total [5,7,2]  
total (5:(7:(2:[])))  
5 + total (7:(2:[]))  
5 + 7 + total (2:[])  
5 + 7 + 2 + total []  
5 + 7 + 2 + 0

## 4 Regular expressions and stringsets

Now we can put some of these ideas together to start talking about something that looks (a bit) like linguistics.

Some standard stage-setting concepts:

- (11) For any set  $\Sigma$ , we define  $\Sigma^*$  as the smallest set such that:
- $\epsilon \in \Sigma^*$ , and
  - if  $x \in \Sigma$  and  $u \in \Sigma^*$  then  $(x:u) \in \Sigma^*$ .
- (12) For any two strings  $u \in \Sigma^*$  and  $v \in \Sigma^*$ , we define  $u \mathbin{++} v$  as follows:
- $\epsilon \mathbin{++} v = v$
  - $(x:w) \mathbin{++} v = x:(w \mathbin{++} v)$

First we’ll define what regular expressions *are*, i.e. what counts as a regular expression. That’s all we’re saying in (13). It’s analogous to defining what counts as a propositional formula in (4).

- (13) Given an alphabet  $\Sigma$ , we define  $\text{RE}(\Sigma)$ , the set of regular expressions over  $\Sigma$ , as follows:
- if  $x \in \Sigma$ , then  $\underline{x} \in \text{RE}(\Sigma)$
  - if  $r_1 \in \text{RE}(\Sigma)$  and  $r_2 \in \text{RE}(\Sigma)$ , then  $(r_1 \mid r_2) \in \text{RE}(\Sigma)$
  - if  $r_1 \in \text{RE}(\Sigma)$  and  $r_2 \in \text{RE}(\Sigma)$ , then  $(r_1 \cdot r_2) \in \text{RE}(\Sigma)$
  - if  $r \in \text{RE}(\Sigma)$ , then  $r^* \in \text{RE}(\Sigma)$
  - $\mathbf{0} \in \text{RE}(\Sigma)$
  - $\mathbf{1} \in \text{RE}(\Sigma)$

So if we have the alphabet  $\Sigma = \{a, b, c\}$ , then here are some elements of  $\text{RE}(\Sigma)$ :

- (14) a.  $(\underline{a} \mid \underline{b})$   
b.  $((\underline{a} \mid \underline{b}) \cdot \underline{c})$   
c.  $((\underline{a} \mid \underline{b}) \cdot \underline{c})^*$

Now, any regular expression  $r \in \text{RE}(\Sigma)$  denotes a particular subset of  $\Sigma^*$ , i.e. denotes a stringset. We’ll write  $\llbracket r \rrbracket$  for the stringset denoted by  $r$ . (The following definition is analogous to the definition of the denotations of propositional formulas in (4).)

- (15) Given a regular expression  $r \in \text{RE}(\Sigma)$ , we define the set  $\llbracket r \rrbracket \subseteq \Sigma^*$  as follows:
- $\llbracket x \rrbracket = \{x\}$
  - $\llbracket (r_1 \mid r_2) \rrbracket = \llbracket r_1 \rrbracket \cup \llbracket r_2 \rrbracket$
  - $\llbracket (r_1 \cdot r_2) \rrbracket = \{u \mathbin{++} v \mid u \in \llbracket r_1 \rrbracket, v \in \llbracket r_2 \rrbracket\}$
  - $\llbracket r^* \rrbracket$  is the smallest set such that:
    - $\epsilon \in \llbracket r^* \rrbracket$
    - if  $u \in \llbracket r \rrbracket$  and  $v \in \llbracket r^* \rrbracket$ , then  $u \mathbin{++} v \in \llbracket r^* \rrbracket$

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

We'll come back to regular expressions in a couple of weeks.

e.  $\llbracket \mathbf{0} \rrbracket = \emptyset = \{\}$

f.  $\llbracket \mathbf{1} \rrbracket = \{\epsilon\}$

The tricky part here is the  $r^*$  case. It says roughly that  $\llbracket r^* \rrbracket$  is the set comprising all strings that we can get by concatenating *zero* or more strings from the set  $\llbracket r \rrbracket$ . Concatenating *zero* such strings produces  $\epsilon$ , so  $\epsilon \in \llbracket r^* \rrbracket$ . Concatenating *n* such strings, where *n* is *non-zero*, really means concatenating some string *u*, which is in  $\llbracket r \rrbracket$ , with some string *v*, which is the result of concatenating some  $n - 1$  such strings.

We can use this definition to work out the stringsets denoted by the regular expressions in (14).

(16) a.  $\llbracket (\underline{a} \mid \underline{b}) \rrbracket = \llbracket \underline{a} \rrbracket \cup \llbracket \underline{b} \rrbracket$

$$= \{a\} \cup \{b\}$$

$$= \{a, b\}$$

b.  $\llbracket ((\underline{a} \mid \underline{b}) \cdot \underline{c}) \rrbracket = \{u \mathbin{++} v \mid u \in \llbracket (\underline{a} \mid \underline{b}) \rrbracket, v \in \llbracket \underline{c} \rrbracket\}$

$$= \{u \mathbin{++} v \mid u \in \{a, b\}, v \in \{c\}\}$$

$$= \{a \mathbin{++} c, b \mathbin{++} c\}$$

$$= \{ac, bc\}$$

c.  $\llbracket ((\underline{a} \mid \underline{b}) \cdot \underline{c})^* \rrbracket = \{\epsilon, ac, bc, acac, acbc, bcac, bcbc, acacac, acacbc, \dots\}$

# Assignment Project Exam Help

---

*(appears on the following pages)*  
<https://powcoder.com>

## Add WeChat powcoder

## A Appendix: The logic behind recursive functions

Learning to write recursive functions can be tricky.<sup>3</sup> Let’s take a close look at the logic of “discovering” the definition of the `double` function.

Suppose as a starting point we define `double1`, which gives the right answer only for zero and one, and `double2`, which gives the right answer only for zero, one and two, as follows:

```
(17) double1 = \n -> case n of
      Z -> Z
      S n' -> S (S Z)

(18) double2 = \n -> case n of
      Z -> Z
      S n' -> case n' of {Z -> S (S Z); S n'' -> S (S (S (S Z)))}
```

Neither of these is recursive, but working out the *relationship between* these two functions is the key to working out how to write our fully-fledged `double` function.

Notice that, if `double2` is given any non-zero number (i.e. anything of the form `(S ...)`), the result will never be less than two, i.e. the result will be of the form `(S (S ...))`. So, pulling those two layers out to the front, we can rewrite `double2` like this:

```
(19) double2 = \n -> case n of
      Z -> Z
      S n' -> S (S (case n' of {Z -> Z; S n'' -> S (S Z)}))
```

Now here comes the crucial step. The thing that’s inside `(S (S ...))` now is exactly equivalent to `double1 n'`. So we can rewrite `double2` again like this:

```
(20) double2 = \n -> case n of
      Z -> Z
      S n' -> S (S (double1 n'))
```

And importantly, we can generalize: *the logic here has nothing in particular to do with one and two, it applies to any number and its successor*. For example if we suppose, just for the sake of argument, that someone had already written a function called `double73` which — somehow, we don’t really care how — correctly doubles any number up to and including 73. Then we could write `double74`, which goes one better, like this:

```
(21) double74 = \n -> case n of
      Z -> Z
      S n' -> S (S (double73 n'))
```

Notice that each of these partially-working doubling functions has type `Numb -> Numb`. We can encapsulate that relationship that holds between `double1` and `double2`, and holds between `double73` and `double74`, and so on, into a function with type `(Numb -> Numb) -> (Numb -> Numb)`. Given any partially-working doubling function, the following function `doubleBooster` will produce a new partially-working function that goes one better:

```
(22) doubleBooster = \f -> \n -> case n of
      Z -> Z
      S n' -> S (S (f n'))
```

---

<sup>3</sup>Graham Hutton puts it nicely in his textbook, *Programming in Haskell* (p.66): “Defining recursive functions is like riding a bicycle: it looks easy when someone else is doing it, may seem impossible when you first try to do it yourself, but becomes simple and natural with practice.”

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

This function will turn `double1` into `double2`, will turn `double73` into `double74`, etc.<sup>4</sup> And notice that `doubleBooster` is a closed term: it does not contain any free variables, nor any recursion.

The important idea to take away is this: when we use the name of the function-being-defined inside that function, we’re using it in the way that `doubleBooster` uses its argument `f`. Your task in writing a recursive function is just to say how we “boost” a partially-working function into a function that goes one better.

## B Appendix: Recursion and induction

You’ve probably seen *proofs by induction* before. This is a technique for proving that some property holds of all natural numbers. To construct such a proof, you proceed in two steps. First, you show that the property holds of zero; this is known as the “base case”. Second, you show that if the property holds of some number  $k$  then it also holds of  $k + 1$ ; this is known as the “inductive step”.

Here’s an example:

- (23) Prove that, for all natural numbers  $n$ , the sum of all natural numbers less than or equal to  $n$  is  $\frac{n \times (n+1)}{2}$ .
- a. *Base case:* We need to show that zero has the relevant property, i.e. that the sum of all natural numbers less than or equal to zero is  $\frac{0 \times (0+1)}{2}$ . Well, the sum of all natural numbers less than or equal to zero is zero, and  $\frac{0 \times (0+1)}{2} = 0$ , so this part is done.
  - b. *Inductive step:* We need to show that if  $k$  has the relevant property then  $k + 1$  does too. In other words, we need to show that the sum of all natural numbers less than or equal to  $k + 1$  is  $\frac{(k+1) \times ((k+1)+1)}{2}$ , and we get to assume, along the way, that the sum of all natural numbers less than or equal to  $k$  is  $\frac{k \times (k+1)}{2}$ . Well, the sum of all natural numbers up to  $k + 1$  is

$$(0 + 1 + 2 + \cdots + k) + (k + 1)$$

and by the assumption that we get to make about  $k$  this is equal to

$$\frac{k \times (k + 1)}{2} + (k + 1)$$

which we can reshuffle to the desired result as follows:

$$\begin{aligned} \frac{k \times (k + 1)}{2} + (k + 1) &= \frac{k \times (k + 1)}{2} + \frac{2 \times (k + 1)}{2} \\ &= \frac{k^2 + 3 \times k + 2}{2} \\ &= \frac{(k + 1) \times (k + 2)}{2} \\ &= \frac{(k + 1) \times ((k + 1) + 1)}{2} \end{aligned}$$

Since this property holds of zero, and it holds of  $k + 1$  whenever it holds of  $k$ , we can conclude that it holds for all natural numbers.

A similar logic underlies the recursive functions we can write on the type `Numb`. In particular, the way we *assume* that the function we’re writing will work on smaller arguments corresponds precisely to the way we assume that  $k$  has the relevant property when we’re trying to show that  $k + 1$  has it. For example, when we write the recursive call to `f` in

```
f = \n -> case n of {Z -> Z; S n' -> S (S (f n'))}
```

<sup>4</sup>And actually, if we define `double0` as `\n-> Z` (i.e. the version of the function that works correctly only for zero), then it will also turn `double0` into `double1`. But it’s more intuitive maybe to start with `double1`.

Assignment Project Exam Help  
<https://powcoder.com>  
Add WeChat powcoder

we assume that the function works correctly on the argument  $n'$ , as part of our getting it to work correctly on the argument  $S\ n'$ , i.e. on the argument  $n$ .

To bring out the connection, we can write out a proof by induction that this function  $f$  really does double its argument. We need a bit of notation to make this precise: I'll write  $S^n\ Z$  for the expression which is  $Z$  with  $n$ -many  $S$ s “on top of it”, so  $S^0\ Z$  is  $Z$ ,  $S^1\ Z$  is  $S\ Z$ ,  $S^2\ Z$  is  $S\ (S\ Z)$ , etc.

- (24) Prove that, for all natural numbers  $n$ ,  $f\ (S^n\ Z)$  will evaluate to  $S^{2n}\ Z$ .
- a. *Base case:* The term whose evaluation we are interested in is  $f\ (S^0\ Z)$ , i.e.  $f\ Z$ . This will evaluate to  $Z$  via the first branch in the case statement, which is  $S^{2 \times 0}\ Z$  as required.
  - b. *Inductive step:* We need to show that  $f\ (S^{k+1}\ Z)$  evaluates to  $S^{2(k+1)}\ Z$ , and we get to assume, along the way, that  $f\ (S^k\ Z)$  evaluates to  $S^{2k}\ Z$ . To begin, notice that  $f\ (S^{k+1}\ Z)$  is  $f\ (S\ (S^k\ Z))$ , so by the second branch of the case statement this will evaluate to  $S\ (S\ (f\ (S^k\ Z)))$ . By the inductive assumption, this evaluates to  $S\ (S\ (S^{2k}\ Z))$ , which is  $S^{2k+2}\ Z = S^{2(k+1)}\ Z$ , as required.

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder