1) Use the parse graph to the right to compute the most general type for the function

\[ f(g, h) = g(h) + 2 \]

Assume that 2 has type Integer and + has type Integer → Integer → Integer.

2) Suppose that the following Haskell definitions have been loaded:

```haskell
my_const c x = c
append [] ys = ys
append (x:xs) ys = x : append xs ys
my_map f [] = []
my_map f (x:xs) = f x : my_map f xs
```

What is the type of each of the following Haskell expressions? (Some may give an error.)

- a. \texttt{my\_const}
- b. \texttt{my\_const True}
- c. \texttt{append []}
- d. \texttt{append [True, False]}
- e. \texttt{append [3] ['a', 'b']}
- f. \texttt{append "quad" ['a', 'b']}
- g. \texttt{my\_map}
- h. \texttt{my\_map (my\_const True)}

What is the value of each of the following Haskell expressions?

- a. \texttt{my\_const 5 "octopus"}
- b. \texttt{my\_map (my\_const "squid") [1 ..]}
- c. \texttt{my\_map \textit{sqrt} [1, 2, 100]}

---

**Figure 6.10.** Parse tree for problem 6.5.

Write one or two sentences to explain succinctly and informally why \texttt{append} has the type you give. This function is intended to append one list onto another. However, it has a bug. How might knowing the type of this function help the programmer to find the bug?

6.8 Type Inference and Debugging

The \texttt{reduce} function takes a binary operation, in the form of a function \texttt{f}, and a list, and produces the result of combining all elements in the list by using the binary operation. For example:

\[ \text{reduce plus [1,2,3]} = 1 + 2 + 3 = 6 \]

if \texttt{plus} is defined by \texttt{plus(x,y::Int) = x + y}.

A friend of yours is trying to learn Haskell and tries to write a \texttt{reduce} function. Here is his incorrect definition:

\[ \text{reduce(f, x)} = x \]
\[ \text{reduce(f, (x : y)) = f(x, reduce(f, y))} \]
3) Consider the following definitions in Haskell:

\[
\begin{align*}
\text{foo } x & = x : (\text{foo } x) \\
\text{bar } x y & = \begin{cases} 
\text{if } (\text{length } x < 3) \text{ then } (\text{sum } x) \text{ else } (\text{sum } y) 
\end{cases}
\end{align*}
\]

- a. Infer the type of the definitions of functions \text{foo} and \text{bar}, including type constraints.
- b. What is the result of evaluating \text{bar} [1,2] (\text{foo} 1)?
- c. And what is the result of evaluating \text{bar} [1,2,3] (\text{foo} 1)?

4) Consider the following tail-recursive function, written in Haskell:

\[
\text{mkMin } x y = \text{if } x \leq y \text{ then } x \text{ else } \text{mkMin} (x - y) y
\]

- a. Write the type inferred for function \text{mkMin} including the type constraints.
- b. Assuming that the language also includes assignments and a \text{while} statement, transform \text{mkMin} into an equivalent non-recursive function.
- c. Assuming that the language is pure functional and includes lambda-abstraction, transform \text{mkMin} into a function in Continuation Passing Style (CPS).
- d. Infer the Haskell type of the latter function.

5) By exploiting the syntax for list comprehension of Haskell, write expressions that denote:
- a. The list of squares of even natural numbers from 0 to 100;
- b. The list of all Pythagorean triples up to \( n \), i.e., of all triples \((x, y, z)\) such that \( x, y, z \leq n \) and \( x^2 + y^2 = z^2 \).

6) Using list comprehension, we can denote the list of all even numbers from 0 to 10 as
\[[2 * x \mid x \leftarrow [0..5]]
\]
but also as
\[[2 * x \mid x \leftarrow [0..5], x < 6]
Is there any difference?

7) Infer the type of the following Haskell functions (remember that “++” is the operator of list concatenation):

\[
\begin{align*}
\text{twice } x & = [x,x] \\
\text{repl } x y & = [x..y] \\
f g h & = \{ x \to y \to (g x)++(h y) 
\end{align*}
\]

What is the result evaluating the expression \( f \text{ twice } (\text{repl} 1) 10 5 \)?

8) Consider the following function:

\[
\begin{align*}
\text{int } \text{foo} (\text{int } x) & = \\
& \begin{cases} 
\text{if } (x>100) \text{ return } x-10; \\
\text{else } \text{return } \text{foo}(\text{foo}(x+11));
\end{cases}
\end{align*}
\]

Is this tail recursive? Justify your answer.