1 Fruitful Functions

Many of the functions we’ve encountered so far can be understood as commands; however there is another important class of functions that yield results. We call this class fruitful functions. For example, turtle.forward doesn’t yield a result; it moves the turtle. But the function math.sqrt does yield a result.

```python
>>> turtle.forward(100)
>>> math.sqrt(2)
1.4142135623730951
```

We can also write fruitful functions. To do so we need to know how to tell Python that our function should yield a result. The special word to use is return, which causes the function to exit yielding the value specified.

Example

```python
def addOne(x):
    """addOne: Number -> Number""
    return x+1

>>> addOne(2)
3
```

Sometimes, it may be advantageous to have a fruitful function that always yields the same value. This technique allows us to define a constant name for a value that would be used repeatedly and improves maintainability. Rather than having to update that value everywhere in the code, changes only need to be made in one location.

Example

```python
def PI():
    return 3.14

def circleArea(r):
    return r * r * PI()

def sphereSurfaceArea(r):
    return r * r * 4 * PI()
```
If we were to decide that we needed more precision for PI, we need only change the return value of PI(); we wouldn’t have to modify either circleArea() or sphereSurfaceArea() individually.

## Two Classic Fruitful Functions

There are two mathematical functions that computer scientists like to use as examples: the factorial function and the Fibonacci function. The factorial of a natural number \( n \) is written as \( n! \), the \( n \)th Fibonacci number is written \( F_n \). The mathematical definitions are below.

\[
\begin{align*}
0! &= 1 \\
n! &= n \times (n-1)!
\end{align*}
\]

\[
\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{align*}
\]

The translations of these functions to code now follows.

```python
def fact(n):
    """fact : NatNum -> NatNum""
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)

def fib(n):
    """fib : NatNum -> NatNum""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

## Substitution Trace

The technique of constructing execution diagrams is important and useful for understanding and debugging of recursive functions. However, it involves a lot of details that can be omitted for a certain classes of functions. Another form of tracing called substitution tracing can eliminate a lot of that detail when tracing fruitful functions.

Substitution tracing is a generalization of arithmetic expression evaluation. It involves writing a function call with its arguments, an equal sign, and then the expression it evaluates to. This process is repeated until the result is computed.
3.1 Example: Factorial

\[
\text{fact}(3) = 3 \times \text{fact}(2)
\]
\[
= 3 \times (2 \times \text{fact}(1))
\]
\[
= 3 \times (2 \times (1 \times \text{fact}(0)))
\]
\[
= 3 \times (2 \times (1 \times 1))
\]
\[
= 3 \times (2 \times 1)
\]
\[
= 3 \times 2
\]
\[
= 6
\]

3.2 Example: Fibonacci

\[
\text{fib}(3) = \text{fib}(2) + \text{fib}(1)
\]
\[
= (\text{fib}(1) + \text{fib}(0)) + \text{fib}(1)
\]
\[
= (1 + \text{fib}(0)) + \text{fib}(1)
\]
\[
= (1 + 0) + \text{fib}(1)
\]
\[
= 1 + \text{fib}(1)
\]
\[
= 1 + 1
\]
\[
= 2
\]

4 Tail-Recursive Fruitful Functions

4.1 Tail-Recursive Formulation

We can also characterize fruitful functions as being tail-recursive or not. The examples above are not tail-recursive. For the \textbf{fact} function, we see in the substitution trace in section 3.1 that there is work to do after the recursive call \textbf{fact}(2): we need to multiply by three. For the \textbf{fib} function, we see in the substitution trace in section 3.2 that there is work to do after the recursive calls \textbf{fib}(1) and \textbf{fib}(2): we need to add those results together.

It turns out that there is a tail-recursive way of writing each of those functions. Doing so involves introducing a new accumulation parameter. Coming up with the accumulative formulation of a fruitful function is often tricky. The tail-recursive code is below.

```python
def factAccum(n, a):
    """factAccum: NatNum * NatNum -> NatNum""
    if n == 0:
```
return a
else:
    return factAccum(n - 1, n * a)

def fact(n):
    """fact: NatNum -> NatNum"""
    return factAccum(n, 1)

How did we determine this definition for factAccum? We know that we need to multiply from the definition of factorial; however, if we want a tail-recursive definition, we can't multiply on the outside. Thus we need one more parameter. Hence we are led to a generalization of factorial: factAccum(n, a) = n! × a. From this definition, we can infer that factAccum(0, a) = a, and factAccum(n, a) = n! × a = n × (n - 1)! × a = (n - 1)! × (n × a) = factAccum(n - 1, n × a).

def fibAccum(n, a, b):
    """fibAccum: NatNum * NatNum * NatNum -> NatNum"""
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        return fibAccum(n - 1, b, a + b)

def fib(n):
    """fib: NatNum -> NatNum"""
    return fibAccum(n, 0, 1)

How did we determine this definition for fibAccum? We know that we need to add from the definition of the Fibonacci function; however, if we want a tail-recursive definition, we can't add on the outside. Further, we suspect that since the previous two values are involved, that we need two more parameters. At this point, we will resort to intuition and note that the two additional parameters can be viewed as a window into the sequence. As the counter n decreases, the window moves forward in the sequence.

4.2 Example: Substitution Trace of Tail-Recursive Factorial

\[
\begin{align*}
fact(3) &= factAccum(3, 1) \\
       &= factAccum(2, 3) \\
       &= factAccum(1, 6) \\
       &= factAccum(0, 6) \\
       &= 6
\end{align*}
\]
4.3 Example: Substitution Trace of Tail-Recursive Fibonacci

\[
\begin{align*}
\text{fib}(3) &= \text{fibAccum}(3, 0, 1) \\
&= \text{fibAccum}(2, 1, 1) \\
&= \text{fibAccum}(1, 1, 2) \\
&= 2
\end{align*}
\]