CSc 520

Principles of Programming Languages

12: Haskell — Function Definitions

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg
When programming in a functional language we have basically two techniques to choose from when defining a new function:

1. Recursion
2. Composition

Recursion is often used for basic “low-level” functions, such that might be defined in a function library.

Composition (which we will cover later) is used to combine such basic functions into more powerful ones.

Recursion is closely related to proof by induction.
Defining Functions...

Here's the ubiquitous factorial function:

\[
\text{fact} :: \text{Int} \rightarrow \text{Int} \\
\text{fact} \ n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact} \ (n-1)
\]

The first part of a function definition is the \textit{type signature}, which gives the \textit{domain} and \textit{range} of the function:

\[
\text{fact} :: \text{Int} \rightarrow \text{Int}
\]

The second part of the definition is the \textit{function declaration}, the implementation of the function:

\[
\text{fact} \ n = \text{if } n == 0 \text{ then } ... 
\]
Defining Functions...

- The syntax of a type signature is
  \[
  \text{fun\_name} :: \text{argument\_types}
  \]

  \text{fact} takes one integer input argument and returns one integer result.

- The syntax of function declarations:
  \[
  \text{fun\_name param\_names} = \text{fun\_body}
  \]

  \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ is a conditional expression}
  \text{that returns the value of } e_2 \text{ if } e_1 \text{ evaluates to True. If } e_1
  \text{ evaluates to False, then the value of } e_3 \text{ is returned.}

Examples:

- \text{if False then 5 else 6} \Rightarrow 6
- \text{if 1==2 then 5 else 6} \Rightarrow 6
- \text{5 + if 1==1 then 3 else 2} \Rightarrow 8
Defining Functions...

- `fact` is defined recursively, i.e. the function body contains an application of the function itself.

- The syntax of function application is: `fun_name arg`. This syntax is known as “juxtaposition”.

- We will discuss multi-argument functions later. For now, this is what a multi-argument function application (“call”) looks like:

  ```
  fun_name arg_1 arg_2 ... arg_n
  ```

- Function application examples:

  - `fact 1`  \(\Rightarrow 1\)
  - `fact 5`  \(\Rightarrow 120\)
  - `fact (3+2)`  \(\Rightarrow 120\)`
Typically, a recursive function definition consists of a **guard** (a boolean expression), a **base case** (evaluated when the guard is True), and a **general case** (evaluated when the guard is False).

```
fact n =  
  if n == 0 then 1             ⇐ guard
  else n * fact (n-1)         ⇐ base case
```

```n-1```
Simulating Recursive Functions

- We can visualize the evaluation of \texttt{fact 3} using a tree view, box view, or reduction view.

- The tree and box views emphasize the flow-of-control from one level of recursion to the next.

- The reduction view emphasizes the substitution steps that the \texttt{hugs} interpreter goes through when evaluating a function. In our notation boxed subexpressions are substituted or evaluated in the next reduction.

- Note that the Haskell interpreter may not go through exactly the same steps as shown in our simulations. More about this later.
Tree View of \texttt{fact 3}

This is a Tree View of \texttt{fact 3}.

We keep going deeper into the recursion (evaluating the general case) until the guard is evaluated to True.
Tree View of \textbf{fact 3}

When the guard is \textbf{True} we evaluate the \textit{base case} and return back up through the layers of recursion.
Box View of fact 3

Diagram:

```
3 -> 0 == False -> if

fact 2:
2 -> 1 -> *

fact 3:

1 -> then

else
```
Box View of fact 3...

fact 3

if

then

else

fact 2

if

then

else

fact 1

if

then

else

0 == False

1

2

3

1

-1

*
Box View of fact 3...

```
fact 3

0 == False

3

fact 2

1

fact 1

1

if then

else

1

*
Reduction View of $\text{fact } 3$

\[
\text{fact } 3 \Rightarrow \\
\text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact}(3-1) \Rightarrow \\
\text{if False then } 1 \text{ else } 3 \times \text{fact}(3-1) \Rightarrow \\
3 \times \text{fact}(3-1) \Rightarrow \\
3 \times \text{fact } 2 \Rightarrow \\
3 \times \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact}(2-1) \Rightarrow \\
3 \times \text{if False then } 1 \text{ else } 2 \times \text{fact}(2-1) \Rightarrow \\
3 \times (2 \times \text{fact}(2-1)) \Rightarrow \\
3 \times (2 \times \text{fact } 1) \Rightarrow \\
3 \times (2 \times \text{if } 1 == 0 \text{ then } 1 \text{ else } 1 \times \text{fact}(1-1)) \\
\Rightarrow \ldots
\]
Reduction View of $\text{fact~}3$...

\[
3 \times (2 \times \text{if } 1 == 0 \text{ then } 1 \text{ else } 1 \times \text{fact } (1-1)) \Rightarrow \\
3 \times (2 \times \text{if } \text{False} \text{ then } 1 \text{ else } 1 \times \text{fact } (1-1)) \Rightarrow \\
3 \times (2 \times (1 \times \text{fact } (1-1))) \Rightarrow \\
3 \times (2 \times (1 \times \text{fact } 0)) \Rightarrow \\
3 \times (2 \times (1 \times \text{if } 0 == 0 \text{ then } 1 \text{ else } 0 \times \text{fact } (0-1))) \Rightarrow \\
3 \times (2 \times (1 \times \text{if } \text{True} \text{ then } 1 \text{ else } 0 \times \text{fact } (0-1))) \Rightarrow \\
3 \times (2 \times (1 \times 1)) \Rightarrow \\
3 \times (2 \times 1) \Rightarrow \\
3 \times 2 \Rightarrow \\
6
\]
Recursion Over Lists

In the fact function the guard was \( n == 0 \), and the recursive step was \( \text{fact}(n-1) \). I.e. we subtracted 1 from fact’s argument to make a simpler (smaller) recursive case.

We can do something similar to recurse over a list:
1. The guard will often be \( n == [\ ] \) (other tests are of course possible).
2. To get a smaller list to recurse over, we often split the list into its head and tail, \( \text{head:tail} \).
3. The recursive function application will often be on the tail, \( f \text{ tail} \).
The **length** Function

**In English:**
The length of the empty list \([\ ]\) is zero. The length of a non-empty list \(S\) is one plus the length of the tail of \(S\).

**In Haskell:**

```haskell
len :: [Int] -> Int
len s = if s == [ ] then
    0
else
    1 + len (tail s)
```

We first check if we’ve reached the end of the list \(s == [\ ]\). Otherwise we compute the length of the tail of \(s\), and add one to get the length of \(s\) itself.
Reduction View of \( \text{len \ [5,6]} \)

\[
\text{len \ s} = \text{if \ s == \ [ \ ] \ then \ 0 \ else \ 1 + \ \text{len \ (tail \ s)}}
\]

\[
\text{len \ [5,6]} \Rightarrow \\
\text{if \ [5,6]==[ \ ] \ then \ 0 \ else \ 1 + \ \text{len \ (tail \ [5,6])}} \Rightarrow \\
1 + \ \text{len \ (tail \ [5,6])} \Rightarrow \\
1 + \ \text{len \ [6]} \Rightarrow \\
1 + (\text{if \ [6]==[ \ ] \ then \ 0 \ else \ 1 + \ \text{len \ (tail \ [6])}) \Rightarrow \\
1 + (1 + \ \text{len \ (tail \ [6])}) \Rightarrow \\
1 + (1 + \ \text{len \ [ ]}) \Rightarrow \\
1 + (1 + (\text{if \ [ \ ]==[ \ ] \ then \ 0 \ else \ 1+\text{len \ (tail \ [ \ ])})) \Rightarrow \\
1 + (1 + 0) \Rightarrow 1 + 1 \Rightarrow 2
\]
len :: [Int] -> Int
len s = if s==[] then 0
    else 1+len(tail s)

Tree View of len
[5,6,7]