Introduction to Programming and its Mathematical Foundations

Lesson 7: Syntax and Semantics of Function Definition and Application

Summary: The many ways of defining recursive functions are exhaustively described in this Lesson.

Background: There are two fundamental operators in functional programming: function application and function abstraction (i.e., definition). In Lesson 3, we saw examples of application using built-in functions.

One important deviation from conventional practice is that we use an explicit (i.e., visible) operator for function application, the "Dijkstra dot," denoted by the symbol "`.`". Thus we write, e.g., `cos.pi` instead of the conventional `cos pi` or even `cos(pi)`. Without the dot, the names `cos` and `pi` are juxtaposed (i.e., are next to each other). This is a syntax error in our notation, but is allowed in the standard notation.

Function definitions can also be made using the application operator, see below. The abstraction operator, lambda λ (denoted as backslash "\" since our keyboard does not have Greek letters) will be discussed later.

Copied, with minor additions, from Lesson 3, "Data and Datatypes in Gofer: Compound Type Function":

A function definition consists of two parts: the type definition (or signature) and the function body. Relatively simple definitions are made in a single line (symbol "=" stands for "equality by definition.")

```
succ: Int -> Int    dbl:  Int -> Int      sq: Int -> Int
succ . n = n+1      dbl . n = n+n         sq . n = n*n
```

Repeat: Function definitions are made within a Gofer script (file extension `.gs`), and the script is saved for later use. When the functions are to be used (applied to arguments, etc.), the script is loaded into the Gofer interpreter using the `:load` or `:l` command. Once the functions are loaded into the interpreter, they can be used just like built-in functions.

Caution: When you store a script with a new or changed name, call it `newscr.gs`, take two precautions under MS-Windows: (1) put double quotes around the name, like "`newscr.gs`", otherwise `newscr.gs` will be the name and `.txt`, the file extension, so the full file name will be `newscr.gs.txt`, and (2) store the file in the same directory as the Gofer interpreter. If you don't take these precautions, you will get an error message like `ERROR "newscr.gs": Unable to open file`.

Interpreter interactions below were made after the functions `succ`, `dbl` and `sq` were loaded into it:

```
? succ.10
11 : Int
? dbl.10
20 : Int
? sq.10
100 : Int
```

Note the the function composition operator "`;`"
`(f ; g).a` means
First apply `f` to `a`, thus `f.a`
Then apply `g` to the result,
thus `(f ; g).a` equals `g.(f.a)`

```
? dbl . (sq . 10)
200 : Int
? succ. (dbl . (sq.10))
201 : Int
```

```
? (sq ; dbl).10
200 : Int
? (sq ; dbl ; succ) . 10
201 : Int
```

Terminology: We have been saying that "functions are applied to their arguments". To distinguish between function definition and function application, our usage is "*parameter* of the definition," e.g., `succ . n`, and "*argument* of the application," e.g., `succ . 10`. Thus, `n` is the parameter in the definition `succ . n = n+1` , while in the application `succ . 10` , the integer `10` is the argument.

Alternative terminology: There are other usages too, e.g., formal parameter and actual parameter (instead of parameter and argument), or confusingly, the term argument is used in both definition and application.

€: Students are to read, understand and assimilate the three steps below in the function application process.

Function definitions: Here are 3 cases: 1. one single-line definitions, and 2, 3. two multiline definitions

```
succ: Int -> Int  sumupto1: Int -> Int          sumupto2: Int -> Int
succ . n = n+1    sumupto1.0 = 0                sumupto2.0 = 0
                  sumupto1.n = sumupto1.(n-1)+n sumupto2.(n+1)=sumupto2.n +(n+1)
```

The definition of `succ` is the simplest possible: it is one line long, and it has an uncomplicated expression on the r.h.s of "=". The other two definitions involve recursive calls in the induction case.

Steps in function application for `succ`:

- (1) The first and foremost step of a function application is parameter-argument matching. If the function definition is `succ . n = n+1` and the function application to be evaluated is `succ . 10`, then parameter `n` in the definition is matched with argument `10` in the application.
- (2) The second step is the substitution of the parameter by the argument in the value of the function, i.e., in the r.h.s. of the equality `succ.n = n+1`, `n+1` becomes `10+1` after substitution.
- (3) The third and last step is the simplification of the resulting expression, thus `10+1` equals `11`.

When the function definition is on multiple lines, the first step in function application is more elaborate. Here the parameter-argument matching is attempted line by line, from top to bottom.

Steps in function application for `sumupto1` and `sumupto2`:

- (1) During parameter-argument matching in the two `sumupto` examples above, if the argument is `0`, then the matching succeeds on the first line itself. If the argument is greater than `0`, then it is matched against parameter `n` in case 2, and pattern `n+1` in case 3. (Assume that the argument is `5`. Then n takes the value `5` in case 2 and `4` in case 3.)
- (2) Substitution of the parameter by the argument in `sumupto1` or `sumupto2` involves a recursive call. To demonstrate correctness of these functions, you have to depend on proof by mathematical induction. The induction hypothesis is that the recursive call of `sumupto1` or `sumupto2` returns the correct value.
- (3) The induction step is that assuming the induction hypothesis, the addition of the argument to this "correct value of the recursive call" correctly computes the value of the original call.
- (1) `n` is defined to be an `Int`. If `n` is supposed to be a natural number, `n<0` does not exist—it is an error. (Ask yourself, what does it mean to have the sum `0+1+2+...` up to `-5`?) Therefore the expressions `sumpto1.-5` or `sumpto2.-5` do not make sense. The computation does not terminate—rather, it terminates with the error message "`control stack overflow`." (The error cases are not handled in the definitions above—more elaborate definitions are given below.)

... contd. on page 3

Introduction to Programming and its Mathematical Foundations

Lesson 7, Part 2: Recursive function definitions

In the Lesson on proof by induction, we defined the sum of natural numbers up to some integer:

```
sumupto1: Int -> Int              sumupto2 Int -> Int
sumupto1. 0 = 0                   sumupto2.0 = 0
sumupto1. n = sumupto1.(n-1)+ n   sumupto2.(n+1) = sumupto2.n + (n+1)
```

If we change the operator from + to *, and the base case value from 0 to 1, we get the factorial function:

```
fact1: Int -> Int                 fact2: Int -> Int
fact1. 0 = 1                      fact2.0 = 1
fact1. n = fact1.(n-1) * n        fact2.(n+1) = fact2.n * (n+1)
```

Repeat: In the left column, the variable `n` on the left of the = is called the parameter of the function definition, while in the right column, `n+1` is called not a parameter, but the `n+1` pattern (permitted in Gofer).

Another discussion on recursion:

Function definition is one place where recursion plays an important role, especially in functional programming. Recursion was first discussed in the previous Lesson on Proof by Induction. Here is another discussion

We are using the functional programming style to learn about programming and problemsolving, and in particular, we are considering recursive function definition. In this situation,

The value of function `f` for some argument k (i.e., the value of f when applied to `k`, i.e., the value `f.k`) is written as an expression containing values of the same function f for arguments of smaller size, e.g., see above, `fact1. n = fact1.(n-1) * n`

In turn, these applications to smaller-sized arguments are defined in terms of applications with arguments of still smaller size. This cannot go on indefinitely when the initial argument n is of finite size. Soon enough we reach arguments of smallest size which have to be defined directly instead of recursively. This is the base case of the recursion.

In the general case, a definition might have several base cases, and several induction cases (see below).

Warning: When the argument of a recursive function definition is an infinite list, the "smaller-size arguments" on right hand side of the equality sign may still be infinite lists, e.g., if a function f is defined on the natural numbers, `f.[0...]` might depend on `f.[1...]` which might depend on `f.[2...]`, and so on. In that situation a smallest-size base case will never be reached. (How the situation is resolved is for another Lesson.)

End of discussion on recursion

Definition (Conditional expression): A conditional expression is one whose value depends on a Boolean-valued expression. Its syntax is as follows:

`if <expression: Bool> then <expression1> else <expression 2>`

Its value depends on the value of `<expression:Bool>`. If it is `True`, then the value of the expression is `<expression1>`, otherwise it is `<expression2>`.

The tokens `then <expression 1>` are called the *then clause* of the conditional expression, and `else <expression 2>` is the else clause.

€: Try out these examples on your interpreter:

```
? if 2+2 == 4 then 'T' else 'F'
'T' : Char
```

```
? if x<100 then 10 else 20 where x = 0
10 : Int
```

€: Verify the workings of conditional expressions by evaluating these expressions (and make your own):

```
if True then 100 else 200      if False then 100 else 200
if 1<2 then 3 else 4           if 2+2==5 then 3 else 4.
```

The factorial function may be defined in a single line with a conditional expression, with the base case and induction case forming the two clauses. See factorial function `fac5` below.

```
fac5.n = if n==0 then 1 else n*fac5.(n-1)
```

If the argument is a negative integer, then all the definitions above will go into an endless rewriting mode, without `n` taking the value `0`. We want to be able to announce an error whenever n is negative. The built-in function named error allows us to do this. See function facxx below.

Concrete example of the term-rewriting model of computation (repetition of an Example from Lesson 5):

Notation: In equations 1 to 5, `k!` means that the factorial function is being applied to its parameter `k`.

By the definition of the factorial function,

```
n! = n*(n-1)*(n-2)*(n-3)* ... * 2 * 1          Eq. 1
(n-1)! = (n-1)*(n-2)* ... * 2 * 1,             Eq. 2
0! = 1                                         Eq. 3
```

First we dentify the r.h.s. of Eq. 2 (i.e., the expression `(n-1)*(n-2)* ... * 2 * 1`) in the r.h.s. of Eq. 1, and next, we subsitute the l.h.s of Eq. 2 (i.e., `n-1)!`) in the r.h.s. of Eq. 1, thus

```
n! = n*(n-1)! for n >= 1   (inductive case)    Eq. 4
0! = 1                     (base case)         Eq. 5
```

The term-rewriting model of computation proceeds through equational reasoning.

(Recall equational reasoning: If there is an equation $\alpha = \beta$, then any occurrence of α in an expression may be replaced by β, without changing the value of the expression.) In particular, whenever an expression contains a function application, during the evaluation of this expression the argument of the function application is matched against the parameter on the left hand side of the function definition, and if the match is successful, the function application may be replaced by the right hand side of the equation.

Equational reasoning is employed to evaluate factorial for a particular value of n (n=4) as follows:

```
n! = 4!           substitute argument 4 for parameter n
 = 4*3!           n! = n * (n-1)!, n = 4
 = 4*3*2!         n! = n * (n-1)!, n = 3
 = 4*3*2*1!       n! = n * (n-1)!, n = 2
 = 4*3*2*1*0!     n! = n * (n-1)!, n = 1
 = 4*3*2*1*1      0! = 1
```

The term-rewriting model applies the multiplication operator, one at a time, left to right

```
4*3*2*1*1
 = 12*2*1*1
 = 24*1*1
 = 24*1
 = 24
```

This expression is irreducible—i.e., we cannot reduce it any further. In other words, this is “the answer.”

Introduction to Programming and its Mathematical Foundations

Lesson 7, Part 3: Different kinds of syntax for recursive function definitions

Summary: In its simplest form, the recursive definition of functions has two lines: one line for the base case and another line for the induction case. The issue often reduces to whether the base case is written first, or the induction case is first. Each of these two forms have advantages and disadvantages. See below.

Sometimes, using a conditional expression, both cases can be made to fit into a single line. At other times, in somewhat more complicated definitions, there can be several base cases, or several induction cases, or both. Commonly, in order to simplify the main definitions, auxiliary "helper functions" are defined earlier. Examples for all these variations are provided below.

All the examples given below perform the same thing: computing the factorial function.

The Gofer definition of the factorial function can be written in many different ways. We use different names `fac`, `fac1`, `fac2`, `fac3`, `...` , so that these functions can be individually tested.

```
fac: Int -> Int          -- type declaration
fac.0 = 1                --  base case, first line of definition
fac.n = n* fac.(n-1)     -- induction case, second line

fac1: Int -> Int         -- type declaration
fac1.n = n* fac1.(n-1) -- induction case, first line of definition
fac1.0 = 1               -- base case, second line
```

For function `fac`, if `n>0` there are `n` failed matches in the first line of the definition, for `n`, `n-1`, `n-2`, ..., `2`, `1`, before the base case of `n=0` is reached. We might try to avoid these failures by exchanging the first and second lines, to obtain function `fac1`, where the induction case is matched first.

The difficulty is that the parameter n in line 1 of `fac1` matches both the induction case (`n > 0`) and the base case (`n = 0`). In trying to evaluate `fac.0`, there is a match between n and 0, so `0!` is rewritten `0*(-1)!` which is not only incorrect but leads to an unending computation, for `n = -1`, `-2`, `-3`, ... Try this in the interpreter. You will conclude that this arrangement of the lines leads to an incorrect definition.

Terminology: "Pattern matching" is the manner in which the Gofer interpreter employs multiline definitions like the one above. ("pattern" because, in function `fac2` below, in place of the parameter `n` we have the expression or pattern `n+1`.) We use the terms parameter and pattern interchangeably: "pattern matching" means parameter-argument matching.

Repeat: In a multiline definition, the parameter-argument matching proceeds line by line from top to bottom. The right hand side of the first line for which the argument matches the parameter pattern, replaces the function application.

```
fac1.n = n* fac1.(n-1) -- induction case, first line of definition
fac1.0 = 1               -- base case, second line
```

However, in trying to evaluate `fac.0`, there is a match between n and 0, so `0!` is rewritten `0*(-1)!` which is not only incorrect but leads to an unending computation, for `n = -1`, `-2`, `-3`, ... Try this in the interpreter. You will conclude that this new arrangement of the lines leads to an incorrect definition.

To avoid this kind of situation, Gofer provides the `n+1` pattern, a special case of the `n+k` pattern. The `n+1` pattern matches arguments `1` and above; thus the first line of function `fac2` below does not match the base case, but the second line does, with the result that `fac2` correctly computes the factorial.

```
fac2.(n+1) = (n+1)* fac2.n      -- induction case, first line of definition
fac2.0 = 1                      -- base case, second line
```

The n+k pattern: An argument matches the n+k pattern whenever the argument is at least `k` (i.e., whenever n does not become negative). Thus, for an argument `c`>0, the `n+3 pattern` can match arguments `c`, `c-1`, `c-2`, . . ., `4`, `3` but it cannot match `2`, `1`, or `0`. Verify the correctness of function `fc` below, also.

```
fac3 : Int -> Int
fac3.(n+3) = (n+3) * fac3.(n+2)
fac3.2 = 2
fac3.1 = 1
fac3.0 = 1
```

In functions `fac4` and `fac5` below, conditional expressions are used to calculate factorials

```
fac4, fac5 : Int -> Int

fac4. n = if n==0 then 1 else n * fac4. (n-1)

fac5.n = if n==0 then 1
   else if n>0 then n*fac5.(n-1)
   else error. "Invalid argument--negative integer "
```

If a negative argument is given to `fac4`, the `else` clause is executed, leading to infinite depth of recursion. Function `fac5` computes the correct answer when the argument is `0` or positive, and sends an error message when the argument is negative.

~~R~~: A long statement like the one for `fac5` can be spread over 2 or more lines, as long as the second and subsequent lines are indented, all by the same amount.

~~R~~: The interpreter, which works in interactive mode can take a single line of arbitrary length, but it cannot handle a multiline statement. You will have to enter a multiline in a script and then load the script before executing the multiline statement. This is what is happening with function `fac5`.

~~R~~: In the case of a nested conditional expression, as with function `fac5` (i.e., when a clause of a conditional statement is itself a conditional statement) each `else` clause matches the nearest `then` clause, e.g., the else clause with the error message matches not the leftmost "`then`" (…`then 1 else` …) but the other "`then`" (…`then n*fac3.(n-1)` …).

For function definitions with multiple induction cases, there is another facility called statement guards. Each line of the function definition is guarded by a Boolean valued expression Immediately to the right of the guard is the equality sign followed by an expression. During pattern matching the guards are evaluated, top to bottom. The expression to the right of the first guard that evaluates to True is the value of the function. As before, the second and subsequent lines have to be indented, all by the same amount. See example below:

```
fct.n
   | n>0 = n*fac4.(n-1)
   | n==0  = 1
   | n<0 = error."Negative argument"
```

You don't have to explicitly state that the final guard is `n<0`. There is a keyword called `otherwise`, which always evaluates to `True`.

€: Enter `otherwise` in the interpreter to determine its value.

It is important to use `otherwise` as the last guard in a multiline guarded statement, because otherwise, control will never go past the line with the `otherwise`.

```
fct1.n
   | n>0 = n* fct1.(n-1)
   | n==0  = 1
   | otherwise = error."Negative argument"
```

Function `fct2` is identical to function `fct1` above, but the form (appearance) is reader-friendly. While `fct1` uses the conventional notation "`|`" for guards, `fct2` uses commas and "`if`" for the same purpose. It appears that most students prefer to use the notation below, because it is more clear for them.

```
fct2 . n
   = error . "Negative number", if n<0
   = 1, if n == 0
   = n * fct2 . (n-1), otherwise
```

Function `fct3` uses guards below, but there is no explicit case (either `otherwise`, or `n<0`) for negative arguments. Function `fct3` will evaluate correctly for zero or positive values of `n`, but there is no match for negative values of `n`. This is a program error. See what happens when `fct3. (-1)` is evaluated.

```
fct3.n
   = n* fct3.(n-1), if n>0
   = 1, n==0
```

♋ ♦ ♋

Problem set 7, Syntax and Semantics of Recursive Functions

Background: In Lesson 6, Proof by Induction, we extensively considered recursive solutions of different kinds for a model problem: finding the sum of all natural numbers up to the parameter n. Specifically, we employed parameter n, the (n+1) pattern, basis case first, induction case first, and so on. In Lesson 7, Syntax and Semantics of Recursive Functions, we found that we could get the factorial function by employing the multiplication operator instead of addition, and the basis case value of 1 instead of 0.

The problem set below uses the same syntactic structures as the solutions for finding sum of all the numbers till a given number and finding the factorial of a given number.

1. Count the number of digits in the given number.

2. Add the digits of a given number

3. Check if a given number is prime.

4. Reverse sequence of digits in a given integer

5. Find out if a given number is a perfect number. (Definition: A natural number is a perfect number if the sum of its factors adds up to the number, for example, the perfect numbers `6 = 1+2+3`, or `28 = 1+2+4+7+14`. Perfect numbers are rare: there are only `4` perfect numbers among the first `1000` natural numbers, and `5` perfect numbers among the next `10,000` numbers.)