1413003U, Functional Programming **The IT University, Spring 2018**

RayTracer part I

Last update 2018-02-11

This exercise contributes to the ray tracer project by implementing four modules. Two modules for three dimensional *point* and *vector* arithmetics, `Point`, `Vector`, a module for handling RGB colours, `Colour`, and a module for parsing simple arithmetic expressions, `ExprParse`.
This exercise sheet must be handed in via LearnIt by March 1st. Cut off date is March 27th.
Your solution will consist of several files, see below. As you can only upload one file you zip all the files into one file. Template files are given in the zip file, also containing this document.
Your name (or ITU login) must be part of the filename, e.g., `FP-04-<name1>-<name2>.zip`, where `<nameX>` are the names (or ITU login) of the people working together, e.g.:
`FP-04-MadsAndersen-ConnieHansen.zip`.
It is important that you annotate your own code with comments. It is also important that you apply a functional style, i.e., no loops and no mutable variables.
For this hand-in you also need to consider scenarios where your solutions should return an error, i.e. an exception. The requirement is, that no matter what input you pass to your function that fulfils the function type, then the function should return the intended answer or an exception. It is up to you to define the exceptions and whether they should carry extra information, like error messages.
Please consult chapter 7 about Modules in the F# book and the note "Grammars and parsing with F#".

Vectors and Points

Ray tracers make use of 3-dimensional *points* and *vectors*. The points are used to model points in three dimensional space and the vectors are used to model directions in this space. They are normally modelled using either tuples or records. We will use tuples of floats in this exercise. As an example, you can use the below internal representation of a Vector:

```
type Vector = V of float * float * float
```

In the notation below, we write $\{v.x, v.y, v.z\}$ to denote the three values that makes up the vector v. The same notation is used for points, i.e., $\{p.x, p.y, p.z\}$ for a point p. We will also use a "dot" notation to access the vector and points components in the formulas below. For instance, $p.x$ is the x component of the point p and $v.z$ is the z component of the vector v.
First of all, there is nothing really complicated about points and vectors. Their definitions can be found in any text book on the subject, yet having operations on these abstractions defined in separate modules makes the ray tracer easier to implement later on.
There is no real need for associativity as such except for the cross product of vectors. However, in practice it is never used that way. We are often interested in cross products, but seldom as a cross product of a cross product where associativity matters.
Please consider structure and performance. For instance, the *normalize* function on vectors should of course not recompute the magnitude of the vector three times.

Three Dimensional Vectors

Exercise 5.1 Implement a module `Vector` exposing the below signature. You can use *type augmentation* and *type extension* as explained in Chapter 7 of the F# book.

The module signature `Vector.fsi` must expose the following type and functions:

Type `Vector`: An abstract type `Vector` hiding the actual representation of vectors.

`mkVector` *x y z*: A function `mkVector` of type `float -> float -> float -> Vector`. The function creates a vector that points from the origin to the point with coordinates x, y, and z.

`getX` *v*: A function `getX` of type `Vector -> float`. The function returns the x component of the vector.

`getY` *v*: A function `getY` of type `Vector -> float`. The function returns the y component of the vector.

`getZ` v: A function `getZ` of type `Vector -> float`. The function returns the z component of the vector.

`getCoord` v: A function `getCoord` of type `Vector -> float * float * float`. The function returns the components of the vector *v* as a tripple: (*v.x*, *v.y*, *v.z*).

`multScalar` s v: A function `multScalar` of type `float -> Vector -> Vector`. The function scales the vector v by the float s. The formula is $\{v.x * s, v.y * s, v.z * s\}$.

`magnitude` v: A function `magnitude` of type `Vector -> float`. The function calculates the *magnitude* of a vector (often called the length of a vector and written $|v|$). The formula is $\texttt{sqrt}(v.x^2 + v.y^2 + v.z^2)$.

`dotProduct` u v: A function `dotProduct` of type `Vector -> Vector -> float`. The function calculates the *dot product* of u and v (the cosine of the angle between u and v). The formula is $(u.x*v.x)+(u.y*v.y)+(u.z*v.z)$.

`crossProduct` u v: A function `crossProduct` of type `Vector -> Vector -> Vector`. The function calculates the cross product of u and v (the vector that is perpendicular to the plane generated by the vectors u and v). The formula is $\{u.y * v.z - u.z * v.y, u.z * v.x - u.x * v.z, u.x * v.y - u.y * v.x\}$.

`normalize` v: A function `normalize` of type `Vector -> Vector`. The function normalizes the vector v (scales it so that it has magnitude one and written $\hat{v}$). The formula is $\{v.x/|v|, v.y/|v|, v.z/|v|\}$.

Notice, that you need to be careful when you normalize vectors. Often you want normalized vectors to represent a direction. However, they are sometimes used to reason about distances and you need to keep track of what you want. For instance, say you want to find a point that is ten units from a point along a certain vector. In this case the vector must be normalized as you will otherwise multiply ten by a number not equal to one (the length of a normalized vector) when travelling from your starting point.

`round` v d: A function `round` of type `Vector -> int -> Vector`. The function rounds the vector coordiates to *d* decimals. You can use the function `System.Math.Round`.

The module `Vector.fsi` must also expose the following operations:

`-`: The unary operator $-v$ has type `Vector -> Vector`. The operator negates the vector v using the formula $\{-v.x, -v.y, -v.z\}$.

`+`: The binary operator u `+` v has type `Vector * Vector -> Vector`. The formula is $\{u.x + v.x, u.y + v.y, u.z + v.z\}$.

`-`: The binary operator u `-` v has type `Vector * Vector -> Vector`. The formula is $\{u.x - v.x, u.y - v.y, u.z - v.z\}$.

`*`: The binary operator s `*` v has type `float * Vector -> Vector`. The formula is `multScalar` v s using the function `multScalar` defined above.

`*`: The binary operator u `*` v has type `Vector * Vector -> float`. The formula is `dotProduct` u v using the function `dotProduct` defined above.

`%`: The binary operator u `%` v has type `Vector * Vector -> Vector`. The formula is `crossProduct` u v using the function `crossProduct` defined above.

`/`: The binary operator v `/` f has type `Vector * float -> Vector`. The formula is `multScalar (1.0 /` f`)` v using the function `multScalar` defined above.

You must create the file `Vector.fsi` for the signature, `Vector.fs` for the implementation. You compile the two files to generate the DLL-file `Vector.dll`[1]:

```
fsharpc -a Vector.fsi Vector.fs
```

You can use the file `VectorTest.fs` to test your implementation. You are welcome to add more test examples.

[1] All examples are run on a Mono installation on Mac. You can adjust compilation to work on Linux and Windows either from commandline or a development environment like Xamarin Studio or Visual Studio.

```
$ fsharpc -r Vector.dll VectorTest.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono VectorTest.exe
VectorTest
Test01 OK
Test02 OK
Test03 OK
Test04 OK
Test05 OK
Test06 OK
Test07 OK
Test08 OK
Test09 OK
Test10 OK
Test11 OK
Test12 OK
Test13 OK
Test14 OK
Test15 OK
$
```

Three Dimensional Points

Exercise 5.2 Implement a module `Point` exposing the below signature. You can use *type augmentation* and *type extension* as explained in Chapter 7 of the F# book.

The module signature `Point.fsi` must expose the following type and operations:

Type `Point`: An abstract type `Point` hiding the actual representation of points.

`mkPoint` x y z: A function `mkPoint` of type `float -> float -> float -> Point`. The function creates a point with the coordinates x, y and z.

`getX` p: A function `getX` of type `Point -> float`. The function returns the x component of the point.

`getY` p: A function `getY` of type `Point -> float`. The function returns the y component of the point.

`getZ` p: A function `getZ` of type `Point -> float`. The function returns the z component of the point.

`move` p v: A function `move` of type `Point -> Vector -> Point`. The function displaces the point p by the vector v using this formula $\{p.x + v.x, p.y + v.y, p.z + v.z\}$.

`distance` p q: A function `distance` of type `Point -> Point -> Vector`. The function calculates the distance vector between points p and q. You will obtain a vector that points at q when originating from p. The formula is $\{q.x - p.x, q.y - p.y, q.z - p.z\}$

`direction` p q: A function `direction` of type `Point -> Point -> Vector`. The function calculates the *direction* vector (normalized distance vector) between points p and q. The formula is ˆ(`distance` p q). The ˆ (normalize) function is defined in the `Vector` module above.

`round` p d: A function `round` of type `Point -> int -> Point`. The function rounds the point coordiates to d decimals. You can use the function `System.Math.Round`.

The module `Point.fsi` must also expose the following operations:

+: The binary operator p + v has type `Point * Vector -> Point`. The formula is $\{p.x + v.x, p.y + v.y, p.z + v.z\}$.

−: The binary operator $p1$ − $p2$ has type `Point * Point -> Vector`. The formula is $\{p1.x - p2.x, p1.y - p2.y, p1.z - p2.z\}$.

You must create the file `Point.fsi` for the signature, `Point.fs` for the implementation. You compile the two files to generate the DLL-file `Point.dll`:

```
fsharpc -a -r Vector.dll Point.fsi Point.fs
```

You can use the file `PointTest.fs` to test your implementation. You are welcome to add more test examples.

```
$ fsharpc -r Point.dll PointTest.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono PointTest.exe
PointTest
Test01 OK
Test02 OK
Test03 OK
Test04 OK
Test05 OK
Test06 OK
Test07 OK
Test08 OK
Test09 OK
$
```

Colour

A RGB colour is represented by a *red*, *green* and *blue* component. Usually each component is a real number between 0 and 1, both inclusive. In the ray tracer we will modify and merge colours in ways where this invariant is broken. Hense we accept that the colour components are positive values larger than or equal to 0.

The ray tracer will build a simple bitmap with RGB colours each being an integer in the range $[0 \ldots 255]$. The task of converting colours represented as numbers ≥ 0.0 to integer values in the range $[0 \ldots 255]$ involves several steps including *tone mapping*[2] and *gamma correction*[3] which will be explained during the ray tracer project.

In the notation below, we write $\{c.r, c.g, c.b\}$ to denote the three colours red (r), green (g) and blue (b) that makes up the colour c. The components $c.r$, $c.g$ and $c.b$ must all be ≥ 0.0. In case the invariant is broken we raise the exception `ColourException`.

You can use the following internal representation of a Colour:

```
type Colour = RGB of float * float * float
```

Exercise 5.3 Implement a module `Colour` and a module signature `Colour.fsi` that expose the following type and operations:

`ColourException`: used to signal an non manageable error, e.g., a negative value or a value out of a valid range.

Type `Colour`: An abstract type `Colour` hiding the actual representation of colours.

`mkColour` *r g b*: A function `mkColour` of type `float -> float -> float -> Colour`. The function returns a `Colour` value. Raises exception `ColourException` if one of the arguments are < 0.

`getR` *c*: A function `getR` of type `Colour -> float`. The function returns the `c.r` component of the colour *c*.

`getG` *c*: A function `getG` of type `Colour -> float`. The function returns the `c.g` component of the colour *c*.

`getB` *c*: A function `getB` of type `Colour -> float`. The function returns the `c.b` component of the colour *c*.

[2] See `https://en.wikipedia.org/wiki/Tone_mapping`
[3] See `https://en.wikipedia.org/wiki/Gamma_correction`

`scale` c s: A function `scale` of type `Colour -> float -> Colour`. The function scales the colour c by the float s. The formula is $\{c.r * s, c.g * s, c.b * s\}$. Raises exception `ColourException` if $s < 0$.

`merge` w $c1$ $c2$: A function `merge` of type `float -> Colour -> Colour -> Colour`. The function merges two colours *c1* and *c2* where *w* specifies how much weight to give *c1*. The number *w* must be between 0 and 1 both inclusive; otherwise raise exception `ColourException`. Assume $w' = 1.0 - w$, then the formula is as follows $\{w * c1.r + w' * c2.r, w * c1.g + w' * c2.g, w * c1.b + w' * c2.b\}$.

`toColor` c: A function `toColor` of type `Colour -> System.Drawing.Color`. The function returns a .NET `Color` value. To create the `Color` value we need to convert the color components $c.r$, $c.g$ and $c.b$ into an integer in the range $[0 \ldots 255]$. At the same time we perform a *gamma correction* to be explained later. You can use the following code:

```
let toColor (RGB(r,g,b)) =
  System.Drawing.Color.FromArgb(min (int (sqrt r*255.0)) 255,
                                min (int (sqrt g*255.0)) 255,
                                min (int (sqrt b*255.0)) 255)
```

`fromColor` c: A function `fromColor` of type `System.Drawing.Color -> Colour`. The function returns a `Colour` value. To create the `Colour` value we convert each component into the $[0 \ldots 1]$ range and reverse the *gamma correction*. You can use the following code:

```
let fromColor (c:System.Drawing.Color) =
  mkColour (System.Math.Pow (float c.R / 255.0, 2.0))
           (System.Math.Pow (float c.G / 255.0, 2.0))
           (System.Math.Pow (float c.B / 255.0, 2.0))
```

The module `Colour.fsi` must also expose the following operations:

+: The binary operator $c1 + c2$ has type `Colour * Colour -> Colour`. The formula is $\{c1.r+c2.r, c1.g+c2.g, c1.b + c2.b\}$.

*: The binary operator $c1 * c2$ has type `Colour * Colour -> Colour`. The formula is $\{c1.r * c2.r, c1.g * c2.g, c1.b * c2.b\}$.

*: The binary operator $s * c$ has type `float * Colour -> Colour`. The formula is `scale` c s using the function `scale` defined above. Exception `ColourException` is raised if $s < 0$.

You must create the file `Colour.fsi` for the signature and `Colour.fs` for the implementation. You compile the two files to generate the DLL-file `Colour.dll`:

```
fsharpc -a Colour.fsi Colour.fs
```

You can use the file `ColourTest.fs` to test your implementation. You are welcome to add more test examples:

```
$ fsharpc -r Colour.dll ColourTest.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono ColourTest.exe
ColourTest
Test01 OK
Test02 OK
Test03 OK
Test04 OK
Test05 OK
Test06 OK
Test07 OK
Test08 OK
Test09 OK
```

```
Test10 OK
Test11 OK
Test12 OK
Test13 OK
Test14 OK
Test15 OK
Test16 OK
Test17 OK
Test18 OK
Test19 OK
Test20 OK
TestExn1 OK
TestExn2 OK
TestExn3 OK
TestExn4 OK
TestExn5 OK
TestExn6 OK
TestExn7 OK
TestExn8 OK
```

Parsing simple Arithmetic Expressions

Please read sections 4, 5, 6 and 7 in the note "Grammars and parsing with F#" (GPF#) before you start this assignment.
The ray tracer will have to compute millions of polynomials. The goal is to express these polynomials as simple expressions which are later optimized and turned into optimal representations for efficient execution. The goal of this assignment is to parse a string representing a simple expression into an internal abstract syntax tree.
The grammar we use is similar to the grammar used in section 7 of the GPF# note:

```
E    = E "+" E
     | E "*" E
     | E "^" Int
     | Int
     | Float
     | Var
     | "(" E ")" .
```

where `Int` is the set of integers, `Float` is the set of floats and `Var` is the set of variables.
After fixing precedence, associativity, left recursion etc. we end with the following grammar:

```
E    = T Eopt .
Eopt = "+" T Eopt | e .
T    = F Topt .
Topt = "*" F Topt | e .
F    = P Fopt .
Fopt = "^" Int | e .
P    = Int [ Float | Var | "(" E ")" .
```

We use `e` for the empty sequence Λ. You can assume the grammar fulfills all requirements in Figure 6 in the note GPF#.

Exercise 5.4 The file `ExprParse.fs` contains a function `scan` that can scan a sequence of characters and return a list of terminals. For instance, the expression `scan "2x(2x)"` returns the following list of terminals: `[Int 2; Var "x"; Lpar; Int 2; Var "x"; Rpar]`. The scanner recognizes the two integers `2`, the two variables `x`, and the left and right paranteses. A few examples below:

```
> scan "2x(2x)";;
val it : terminal list = [Int 2; Var "x"; Lpar; Int 2; Var "x"; Rpar]
```

```
> scan "2*x*(2*x)";;
val it : terminal list =
  [Int 2; Mul; Var "x"; Mul; Lpar; Int 2; Mul; Var "x"; Rpar]
>
```

You may notice that the first string `"2x(2x)"` does not fulfil the grammar because the multiplications are implicit. The second example have all multiplications inserted and the resulting list of terminals fulfils the grammar. We would like to allow writing the expressions with implicit multiplications `"*"`. The task of this assignment is to implement a function `insertMult` *`ts`* where *`ts`* is a list of terminals. The function returns a new list of terminals where implicit multiplications are inserted explicitly.
The file `ExprParse.fs` contains the following type of terminals:

```
type terminal =
  Add | Mul | Pwr | Lpar | Rpar | Int of int | Float of float | Var of string
```

The function `insertMult` can be implemented with simple pattern matching where combinations of two terminals with a missing `Mul` terminal are indentified. You have to cover the following combinations to be complete: `Float :: Var`, `Float :: Float`, `Float :: Int`, `Var :: Float`, `Var :: Var`, `Var :: Int`, `Int :: Float`, `Int :: Var`, `Int :: Int`, `Float :: Lpar`, `Var :: Lpar` and finally `Int :: Lpar`. An additional detail is that you may have the following kind of exampels `"2 x y"` or `"2 x y z"`, where the same terminal, say `Var "x"` must have a `Mul` terminal inserted on both sides. The example below illustrates the purpose of `insertMult`:

```
scan "2 x y z";;
val it : terminal list = [Int 2; Var "x"; Var "y"; Var "z"]
> insertMult [Int 2; Var "x"; Var "y"; Var "z"];;
val it : terminal list = [Int 2; Mul; Var "x"; Mul; Var "y"; Mul; Var "z"]
>
```

The function `insertMult` is implemented in around 15 lines. You can also make use of *active patterns* and implement the function in about 9 lines. Also notice, that the file `ExprParseTest.fs` contains a number of unit tests to test your implementation of `insertMult`. You compile and execute the unit tests as follows:

```
$ fsharpc ExprParse.fs ExprParseTest.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono ExprParseTest.exe
ExprParseTest
TestScan01 OK
...
TestInsertMult01 OK
TestInsertMult02 OK
TestInsertMult03 OK
...
$
```

Exercise 5.5 In this exercise we implement the parser for the grammar above. The parser must return an abstract syntax tree using the below type:

```
type expr =
  | FNum of float
  | FVar of string
  | FAdd of expr * expr
  | FMult of expr * expr
  | FExponent of expr * int
```

A few examples are shown below:

```
>parse(insertMult(scan "2 x y z"));;
val it : expr = FMult (FMult (FMult (FNum 2.0,FVar "x"),FVar "y"),FVar "z")
> parse(insertMult(scan "2 x^2"));;
val it : expr = FMult (FNum 2.0,FExponent (FVar "x",2))
>
```

The type of `parse` is `terminal list -> expr`. The parser follows the construction found in section 6 and 7 in GPF#. A template for the function `parse` is found in file `ExprParse.fs`.
A number of unit tests exists in the file `ExprParseTest.fs` that tests the scanner and parser. You compile and execute as shown in exercise 5.4.

You can build a DLL file `ExprParse.dll` for the parser as follows:

```
$ fsharpc -a ExprParse.fsi ExprParse.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$
```