BFNP–F2016, Functional Programming   The IT University, Spring 2016

RayTracer part II

Last update 2016-03-09

This exercise contributes to the ray tracer project by implementing a module for converting arithmetic expressions into polynomials. The module is called `ExprToPoly`, and use the module `ExprParse` from the first part of the ray tracer assignment.
This exercise sheet must be handed in via LearnIt by March 22nd.
You are welcome to solve the assignments in pairs.
Your solution will consist of several files, see below. As you can only upload one file you zip all the files into one file.
Your name must be part of the filename, e.g., `BFNP-06-<name1>-<name2>.zip`, where `<nameX>` are the names of the people working together. Everyone must upload the same file to LearnIt. Failing to do so will earn you zero points. An example:
`BFNP-06-MadsAndersen-ConnieHansen.zip`.
Please consult chapter 7 about Modules in the F# book and the slides introducing the ray tracer by Jesper Bengtson.

Converting Arithmetic Expressions into Polynomials

A ray tracer works with figures of different forms and shapes in the three dimensional space. As an example, the formula, expressed in Cartesian coordinates, for a *sphere* of radius R, centered at the origin is given by $x^2+y^2+z^2 = R^2$. For simplicity we can assume that the radius R is 1 and the formula becomes $x^2+y^2+z^2-1 = 0$.
A ray tracer also has a *ray* expressed as $p + td$, where p is a point in the space, t is a scalar distance and d a normalised direction vector, written $\{d_x, d_y, d_z\}$. One task is to find the distance t where the ray hits the sphere with a given normalised vector d. To solve this we first substitute x, y and z in the formula for the sphere with the ray. That is, substitute x with $p_x + td_x$, y with $p_y + td_y$ and z with $p_z + td_z$.
This results in a new expression for the sphere:

$$(p_x + td_x)^2 + (p_y + td_y)^2 + (p_z + td_z)^2 - 1^2 = 0.$$

The interesting variable is t, that is, what is the closest non-negative distance t for which the normalised vector vector d hits the sphere? Recall that t can be negative (the intersection is behind the ray origin) and there can be more than one solution (a ray can hit the sphere two times). Maybe no solution exists and the ray will not hit the sphere at all. We want to solve the equation with respect to the variable t. In order to do this, we need to simplify the expression as much as possible and turn it into a polynomial with t as the unknown variable. We assume the other variables are known.
The first step is to simplify as much as possible. In plain English we multiply all parentheses such that we have an expression consisting of components that are added and multiplied without the need to write any parentheses. For instance, the equation above becomes:

$$p_x^2 + 2tp_xd_x + t^2d_x^2 + p_y^2 + 2tp_yd_y + t^2d_y^2 + p_z^2 + 2tp_zd_z + t^2d_z^2 - 1^2.$$

Given the simplified expression we can now collect terms into a polynomial with respect to t:

$$t^2(d_x^2 + d_y^2 + d_z^2) + t(2p_xd_x + 2p_yd_y + 2p_zd_z) + (p_x^2 + p_y^2 + p_z^2 - 1^2).$$

The purpose of this exercise is not to solve the equation but to take an arbitrary arithmetic expression, simplify it as far as it will go, and then turn it into a polynomial. The expressions that you will work with in the ray tracer project are quite complicated. They include support for rational exponents and division.
This is an introductory assignment where we only consider addition, multiplication, and integer exponents. For representing expressions we use the following datatype:

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

The example expression $x^2 + y^2 + z^2 - 1$ is expressed as

```
FAdd(FAdd(FAdd(FExponent(FVar "x",2),
               FExponent(FVar "y",2))
          FExponent(FVar "z",2)),
     FNum -1.0)
```

Using the parser `ExprParse` we can simply write `"x^2 + y^2 + z^2 - 1"`. To simplify such an expression we can make use of the below straightforward rules. A variable is denoted x, integer constants n and an arbitrary expression e.

Expression e	Reduces to
$(e_1 + e_2) * (e_3 + e_4)$	$e_1 * e_3 + e_1 * e_4 + e_2 * e_3 + e_2 * e_4$
e^n	$e * e^{n-1}$ if n >= 2
e^n	$e * e * e^{n-2}$ if n >= 2
e^1	e
e^0	1
$e + 0$	e
$0 + e$	e
$1 * e$	e
$e * 1$	e
$e * 0$	0
$x * x$	x^2
$x * x^n$	x^{n+1}

A simplified expression must additionally adhere to the following rules:

1. No additions of duplicate expressions as these should be added together. E.g., $x^2 + x^2 = 2x^2$
2. No multiplication of duplicate variables as these should be made into exponents. E.g., $x * x = x^2$.
3. No occurences of e+0, 0+e, 1*e or e*1.

Remember the commutative law, $a + b = b + a$ and $a * b = b * a$, and the associative law, $(a + b) + c = a + (b + c)$ and $(a * b) * c = a * (b * c)$. For instance $2x + 5y + 3x = 5x + 5y$ and $x * x * y * x * y = x^3 + y^2$.

Design Template

This section explains one possible design for the simplification process and turning a simplified expression into a polynomial. The design template corresponds to the template file `ExprToPoly.fs`. Other and possibly better designs exists and you are welcome to go nuts on alternative designs as long as you implement the signature `ExprToPoly.fsi` below.

```
module ExprToPoly

type expr = ExprParse.expr
val subst: expr -> (string * expr) -> expr

type simpleExpr
val ppSimpleExpr: simpleExpr -> string
val exprToSimpleExpr: expr -> simpleExpr

type poly
val ppPoly: string -> poly -> string
val simpleExprToPoly: simpleExpr -> string -> poly
```

A few comments on the module signature:

- We use the `expr` type from the `ExprParse.fs` file.

- We have included a fully defined pretty print function `ppExpr` in the template file `ExprToPoly.fs`.
- A substitution function `subst` to replace variables with arbitrary expressions. More details below.
- An intermediate representation of expressions suitable for the simplification process, `simpleExpr`. We have included a pretty print function for debugging purposes in the template file.
- A function, `exprToSimpleExpr`, to convert expressions of type `expr` into the simplified representation. More details below.
- A type to represent polynomials `poly` and a pretty printer `ppPoly`. This is fully defined in the template file.
- A function `simpleExprToPoly` that converts a simple expression into a polynomial with respect to one variable.

Substitution

Consider the example spare $x^2 + y^2 + z^2 + -R$ with the following AST:

```
let sphere = FAdd(FAdd(FAdd(FExponent(FVar "x",2),
                              FExponent(FVar "y",2)),
                         FExponent(FVar "z",2)),
                    FMult(FNum -1.0,FVar "R"))
```

As explained in the introduction, page 1, we can make the following substitutions: x with $p_x + td_x$, y with $p_y + td_y$ and z with $p_z + td_z$. The expressions to insert can be defined as

```
let ex = FAdd(FVar "px", FMult(FVar "t",FVar "dx"))
let ey = FAdd(FVar "py", FMult(FVar "t",FVar "dy"))
let ez = FAdd(FVar "pz", FMult(FVar "t",FVar "dz"))
let eR = FNum -1.0
```

Notice we are removing four variables `"x"`, `"y"`, `"z"` and `"R"` and replace with 7 new variables `"px"`, `"py"`, `"pz"`, `"dx"`, `"dy"`, `"dz"` and `"t"`.
The substitute function `subst` takes the expression to substitute in, say `sphere` and a pair with the variable to replace, say `"x"`, and the expression to substitute with, say `ex`. This fits perfectly with `List.fold` and we can do the substitution as

```
let sphereSubst = List.fold subst sphere [("x",ex);("y",ey);("z",ez);("R",eR)]
```

which gives the following result

```
val sphereSubst : expr =
  FAdd
    (FAdd
       (FAdd
          (FExponent (FAdd (FVar "px",FMult (FVar "t",FVar "dx")),2),
           FExponent (FAdd (FVar "py",FMult (FVar "t",FVar "dy")),2)),
        FExponent (FAdd (FVar "pz",FMult (FVar "t",FVar "dz")),2)),
     FMult (FNum -1.0,FNum -1.0))
```

The function `subst` is a recursive traversal of the `expr` type.

```
let rec subst e (x,ex) =
  match e with
  | FNum c -> FNum c
  | FVar s -> failwith "TO BE IMPLEMENTED"
```

Simple Expressions

The intermediate representation `simpleExpr` is a list of *atom groups*. An atom group is a list of *atoms*:

```
type atom = ANum of float | AExponent of string * int
type atomGroup = atom list
type simpleExpr = SE of atomGroup list
```

An atom is simply a number k, i.e., a float `ANum` k, or a variable x to some power n, i.e., `AExponent(`x`,`n`)`. A variable x is representated as `AExponent(`x`,1)` and x^2 as `AExponent(`x`,2)`. An atom group consists of atoms that implicitly are multiplied together. A simple expression is a list of atom groups which are implicitly added together. First step in converting `sphere` is

```
let sphereSE = simplify sphereSubst
```

and gives the result

```
val sphereSE : atom list list =
  [[ANum 1.0; AExponent ("px",1); AExponent ("px",1)];
   [ANum 1.0; AExponent ("px",1); AExponent ("dx",1); AExponent ("t",1)];
   [ANum 1.0; AExponent ("dx",1); AExponent ("t",1); AExponent ("px",1)];
   [ANum 1.0; AExponent ("dx",1); AExponent ("t",1); AExponent ("dx",1);
    AExponent ("t",1)]; [ANum 1.0; AExponent ("py",1); AExponent ("py",1)];
   [ANum 1.0; AExponent ("py",1); AExponent ("dy",1); AExponent ("t",1)];
   [ANum 1.0; AExponent ("dy",1); AExponent ("t",1); AExponent ("py",1)];
   [ANum 1.0; AExponent ("dy",1); AExponent ("t",1); AExponent ("dy",1);
    AExponent ("t",1)]; [ANum 1.0; AExponent ("pz",1); AExponent ("pz",1)];
   [ANum 1.0; AExponent ("pz",1); AExponent ("dz",1); AExponent ("t",1)];
   [ANum 1.0; AExponent ("dz",1); AExponent ("t",1); AExponent ("pz",1)];
   [ANum 1.0; AExponent ("dz",1); AExponent ("t",1); AExponent ("dz",1);
    AExponent ("t",1)]; [ANum -1.0; ANum -1.0]]
```

Using the pretty printer `ppSimpleExpr (SE sphereSE)` we get a better overview of the atoms and atom groups:

```
"1*px*px+1*px*dx*t+1*dx*t*px+1*dx*t*dx*t+1*py*py+1*py*dy*t+1*dy*t*py+1*dy*t*dy*t+
 1*pz*pz+1*pz*dz*t+1*dz*t*pz+1*dz*t*dz*t+-1*-1"
```

For instance, `1*px*px`, is an atom group of three atoms `[ANum 1.0;AExponent("px",1);AExponent("px",1)]`. The function `simplify` is a recursive traversal of an expression of type `expr`. The template function is given in file `ExprToPoly.fs`:

```
let rec simplify = function
  | FNum c -> [[ANum c]]
  | FVar s -> [[AExponent(s,1)]]
  | FAdd(e1,e2) -> simplify e1 @ simplify e2
  | FMult(e1,e2) -> failwith "combine ...TO BE IMPLEMENTED"
  | FExponent(e1,0) -> failwith "TO BE IMPLEMENTED"
  | FExponent(e1,1) -> failwith "TO BE IMPLEMENTED"
  | FExponent(e1,n) -> failwith "TO BE IMPLEMENTED"
```

We make use of the table showing how expressions can be reduced on page 2. The case for `FAdd` simplifies each expression `e1` and `e2` each giving a list of atom groups. They are simply concatenated as atom groups are implicitly added together. The interesting case is `FMult` where we have to implement the reduction rule for `e1 * e2`. The expressions `e1` and `e2` may be arbitrary, which includes something like $(e_{11} + e_{12}) * (e_{21} + e_{22})$ or $(e_{11} + e_{12} + e_{13}) * (e_{21} + e_{22})$ etc. The purpose of the function `combine` is to multiply all components and eliminate the paranteses. Notice, that each e_i is an atom group.

Simplifying an Atom Group

The function `simplifyAtomGroup` makes two simplifications on each atom group:

- The atoms x^n are collected such that each variable x are only represented once. For instance, the atom group `[AExponent ("px",1); AExponent ("px",2)]` is reduced to `[AExponent ("px",3)]` because $x^1 * x^2$ is equal to x^3. This can be implemented using a map mapping variables x to their exponent n. For `[AExponent ("px",1)]` the map is `"px"` $\rightarrow 1$. For the next atom `[AExponent ("px",2)]` then map gets updated to `"px"` $\rightarrow 3$. The map is created folding over the atoms. The map can easily be unfolded to a new atom group with each variable occuring only once.
- Secondly we need to sum all constants k into one sum. E.g., the atom group `[ANum -2.0; ANum -2.0]` becomes `[ANum 4.0]`. Remember to consider the cases where the sum is 0.0 and 1.0.[1]

The call

```
simplifyAtomGroup [AExponent ("px",1); AExponent ("px",2); ANum -2.0; ANum -2.0]
```

returns

```
> val it : atom list = [ANum 4.0; AExponent ("px",3)]
```

The template function is given in file `ExprToPoly.fs`

```
let simplifyAtomGroup ag = failwith "TO BE IMPLEMENTED"
```

Simplify Simple Expressions

The function `simplifySimpleExpr` takes an arbitrary simple expression and simplify it as far as it will go using the following steps:

1. Simplifying each atom group using `List.map` with the function `simplifyAtomGroup`.
2. Adding all atom groups with only constants together. For instance, the following atom groups `[[ANum 3.0];[ANum 4.0]]` is reduced to `[ANum 7.0]`. This is done folding over the atom groups identifying the atom groups with only one atom being of the form `ANum` k and accumulating the sum.
3. Last we group similar atom groups into one group. For instance, the atom groups `[[AExponent("x",2); AExponent("y",3)];[AExponent("x",2); AExponent("y",3)]]` is reduced to `[[ANum 2.0; AExponent("x",2); AExponent("y",3)]]`. This is done using a map mapping atom groups to an integer being the number of times the atom group has been seen so far. Notice the convenience that we can use a structured type like `atomGroup` as the type of keys in a map.

The call

```
let _ = simplifySimpleExpr
          (SE [[ANum 3.0];[ANum 4.0];
               [AExponent("x",2); AExponent("y",3)];
               [AExponent("x",2); AExponent("y",3)]])
```

returns

```
> val it : simpleExpr =
  SE [[ANum 7.0]; [ANum 2.0; AExponent ("x",2); AExponent ("y",3)]]
>
```

The template function is given in file `ExprToPoly.fs`

```
let simplifySimpleExpr (SE ags) =
  let ags' = List.map simplifyAtomGroup ags
  // Add atom groups with only constants together.
  failwith "TO BE IMPLEMENTED"
  // Last task is to group similar atomGroups into one group.
  failwith "TO BE IMPLEMENTED"
```

[1] It is normal practice not to pattern match on float values due to the binary representation. This is however accepted in this assignment.

Polynomials

We represent polynomials with the type `poly`:

```
type poly = P of Map<int,simpleExpr>
```

The type represents polynomials of the form $x_1^{n_1} * se_1 + \cdots + x_m^{n_m} * se_m$, where x_i are variables and se_i are simple expressions, $0 <= i <= m$. For instance, the polynomial

$$t^2(d_x^2 + d_y^2 + d_z^2) + t(2p_x d_x + 2p_y d_y + 2p_z d_z) + (p_x^2 + p_y^2 + p_z^2 - 1^2).$$

is represented by the following map

$0 \rightarrow (p_x^2 + p_y^2 + p_z^2 - 1^2)$
$1 \rightarrow (2p_x d_x + 2p_y d_y + 2p_z d_z)$
$2 \rightarrow (d_x^2 + d_y^2 + d_z^2).$

where the isolated variable t is implicit. The task is to split all atoms into groups where each atom, being member of the same group, has the variable t as part of the atom with the same power n. Consider the simple expression

$$p_x^2 + 2tp_x d_x + t^2 d_x^2 + p_y^2 + 2tp_y d_y + t^2 d_y^2 + p_z^2 + 2tp_z d_z + t^2 d_z^2 - 1^2.$$

being the simple expression before converting to a polynomial. Given t we can simply go through each atom group, say $2tp_y d_y$ and see if t to some power is part of the group. We create a function `splitAG` *v m ag* that

1. iterates over an atom group *ag*.
2. try find the pattern `AExponent(`*v*`,`*n*`)` as part of the atom group *ag* for some power n. If such exists then update the map *m* with $n \rightarrow$ *ag'* where *ag'* is the atom group already in the map for the key *n* extended with the atom group *ag* with the atom t^n removed.

For instance

```
splitAG "t" Map.empty [ANum 1.0; AExponent ("pz",1); AExponent ("dz",1); AExponent ("t",2)]
```

returns the map

```
val m : Map<int,simpleExpr> =
  map [(2, SE [[ANum 1.0; AExponent ("pz",1); AExponent ("dz",1)]])]
```

with and entry for `2` mapped to the atom group without the atom `AExponent ("t",2)`. If we then apply `splitAG` again on

```
splitAG "t" m [ANum 1.0; AExponent ("pz",1); AExponent("dz",1); AExponent ("t",2)]
```

we extend the map for the same key `2`

```
val m : Map<int,simpleExpr> =
  map
    [(2,
      SE
        [[ANum 1.0; AExponent ("pz",1); AExponent ("dz",1)];
         [ANum 1.0; AExponent ("pz",1); AExponent ("dz",1)]])]
```

Given `splitAG` you can fold over the atom groups in the simple expression and build the polynomial. Template code is found in `ExprToPoly.fs`.

Build

A library for `ExprToPoly.fs` is build as follows:

```
$ fsharpc -a ExprParse.fs ExprToPoly.fsi ExprToPoly.fs
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ ll ExprToPoly.dll
-rw-r--r-- 1 nh staff 51712 Mar  7 08:56 ExprToPoly.dll
```

Testing

A number of unit tests are included in the file `ExprToPolyTest.fs`. The tests are relevant if you use the above design template. First compile the tests including the parser `ExprParse.fs`.

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

and running the tests

```
$ mono ExprToPolyTest.exe
ExprToPoly Test
TestPoly01 OK
TestPoly02 OK
TestPoly03 OK
TestPoly04 OK
TestPoly05 OK
TestPoly06 OK
TestSimplify01 OK
TestSimplify02 OK
TestSimplify03 OK
TestSimplify04 OK
TestSimplify05 OK
TestSimplify01Parse OK
TestSimplify02Parse OK
TestSimplify03Parse OK
TestSimplify04Parse OK
TestSimplify05Parse OK
TestSimplify06 OK
TestSimplify07 OK
TestSimplify08 OK
TestSimplify09 OK
TestSimplify10 OK
TestSimplify06Parse OK
TestSimplify07Parse OK
TestSimplify08Parse OK
TestSimplify09Parse OK
TestSimplify10Parse OK
TestSphere01 OK
TestSphere02 OK
TestSE01 OK
TestSE02 OK
$
```