JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 7, No. 7, 2008

Mapping and Visiting in Functional and Object-oriented Programming

Kurt Nørmark, Bent Thomsen, and Lone Leth Thomsen,
Department of Computer Science, Aalborg University, Denmark

Mapping and visiting represent different programming styles for traversals of collections of data. Mapping is rooted in the functional programming paradigm, and visiting is rooted in the object-oriented programming paradigm. This paper explores the similarities and differences between mapping and visiting, seen across the traditions in the two different programming paradigms. The paper is concluded with recommendations for mapping and visiting in programming languages that support both the functional and the object-oriented paradigms.

1 INTRODUCTION

In this paper we compare and contrast mapping, as used in many functional programming languages, with visiting, as used via the ***Visitor*** design pattern in many object-oriented languages. This includes imperative mapping in languages that support both the imperative and the functional paradigms. It also includes functional visiting in object-oriented languages, which most often ignores the functional heritage of list and tree traversal.

In our judgement, mapping, as known from functional programming, is elegant and easy to understand. In contrast, the ***Visitor*** design pattern from object-oriented programming is much more difficult to program and to understand. It is our experience that students have a hard time grasping the ***Visitor*** pattern. In Section 4 we point out "excessive scaffolding" as part of the reason for this. Even among the designers of the original collection of design patterns [4] there are diverging opinions about the ***Visitor*** pattern. Vlidssides [15] quotes Erich Gamma for saying that

> *... in my recent pattern talk I list my bottom-ten patterns, and Visitor is at the very bottom.*

Based on these observations, we find it worthwhile to seek mapping and visiting solutions that make use of inspiration from both functional and object-oriented programming.

In Section 2 we review *simple mapping* as supported by library functions of functional programming languages. We take the opportunity to generalize simple mapping to *general mapping* in Section 3. Next, in Section 4 we review visiting in object-oriented programming, and we compare it with general mapping. This includes the "natural object-oriented solution", the ***Visitor*** design pattern, a double dispatch solution, the walkabout approach, and visitor componentization in the scope of Eiffel [7]. As a particular concern, we discuss in Section 5 how rounds of visiting (and visitors) can be combined. In Section 6 we review and discuss mixed-paradigm solutions, which balance the influence of functional and object-oriented approaches. In Section 7 we conclude the paper with recommendations for the treatment of mapping and visiting in future mixed-paradigm programming languages.

Throughout the paper we illustrate technical points with program snippets written in the programming languages SML, C#, Eiffel, CLOS, or F#. Full programs are presented in Appendices A-D.

2 MAPPING IN FUNCTIONAL PROGRAMMING

Mapping is best known from traversal of linear list structures, but mapping can be applied on any data structure that represents a collection of data.

We start with the mapping problem in the functional programming paradigm. The mapping problem is the following:

- Apply a function f on each element e in the collection c and return the resulting collection of values $f(e)$.

Typically the collection c is implemented as a list of elements. Using functional programming techniques it is straightforward to write a `map` function. E.g. in SML it can be defined recursively as:

```
fun map (f,l) =
  if null l then nil
  else cons(f (hd l),map(f,(tl l)))
```

In the expression `map(f,l)` the `map` function is applied on a function `f` and a list `l`. If `f(x) = x+1` and `l = [1,2,3,4,5]` then `map(f,l)` evaluates to the list `[2,3,4,5,6]`. [1]

The `map` function is perhaps more elegantly defined recursively, using pattern matching, in its curried form, as:

```
fun map f [] = []
  | map f (h::t) = f h :: map f t
```

[1]`null`, `nil`, `hd`, `tl` are all defined in the SML standard library structure `List`. `cons` can be defined as `val cons = op::` where `::` is the SML infix concatenation operator.

There is a natural imperative variant of the mapping problem:

- Apply a procedure p on each element e in a collection c for the sake of the effects of p.

Again using SML this may be defined as:

```
fun imp_map p [] = ()
  | imp_map p (h::t) = (p h);(imp_map p t)
```

Note the only differences between `map` and `imp_map` are the sequence operator `;` instead of the infix cons operator `::` and the unit result instead of an empty list.

The map function is inherently higher-order having the type:

```
val map: ('a -> 'b) -> 'a list -> 'b list
```

and the imperative map function being a specialization for the generic map function with the type:

```
val imp_map: ('a -> unit) -> 'a list -> unit
```

Two rounds of mapping can be *combined* in two different ways. We assume that we work with a linear collection (a list) such as `[x, y, z]` and that we wish to combine the mappings of two functions `f` and `g` on this collection. In the functional programming paradigm, this can be done as follows:

```
(1)  map(g, map (f, [x; y; z]))
(2)  map(compose(g,f), [x; y; z])
```

The expression `compose(g,f)` denotes the combined function, and it corresponds to the mathematical notation `g ∘ f`. As a matter of terminology, the first form of combination will be called *pipelined combination* and the second one will be called *interleaved combination*.

In the functional paradigm, the results of (1) and (2) are equivalent, namely `[g(f(x)), g(f(y)), g(f(z))]`. Solution (2) is preferred because it involves only a single traversal, and because it avoids allocation of memory to an intermediate collection.

In the imperative programming paradigm we find that pipelined combination is the most natural combination. It can be expressed by

```
(3)  imp_map (f, [x, y, z])); imp_map (g, [x, y, z]))
```

which gives rise to the actions `f(x)`, `f(y)`, `f(z)`, `g(x)`, `g(y)`, and `g(z)` in this particular order. Thus, `f` is first applied on all elements, and next `g` is applied on all elements. If `compose(f,g)` denotes a new procedure which applies `f` and `g` sequentially (`f` before `g`), then the composition

```
(4)  imp_map(compose(f,g), [x, y, z])
```

gives rise to an interleaved combination, leading to the actions `f(x)`, `g(x)`, `f(y)`, `g(y)`, `f(z)`, and `g(z)` in this order.

Akin to mapping is filtering and folding (also known as reduction or accumulation). Filtering applies a predicate on each element of a list and returns a list of the elements satisfying the predicate:

```
fun filter p [] = []
  | filter p (h::t) = if (p h)
                      then h::(filter p t)
                      else (filter p t)
```

Folding recursively folds a list into a single result by applying a function to the head element of a list and the result of folding the tail of a list. A constant is returned for the empty list:

```
fun fold f b [] = b
  | fold f b (h::t) = f (h, fold f b t)
```

A function that sums up all the elements of a list, may easily be obtained by instantiating the generic higher-order `fold` function:

```
val sum = fold (fn x => fn y => x+y) 0
```

3 GENERAL MAPPING

We now generalize the mapping idea described above. In order to distinguish the general mapping ideas from the mapping described in Section 2, the latter is referred to as *simple mapping*.

We identify the following different aspects of the *general mapping problem*:

- **Traversal**.
  - **Element selection**. The selection of a subset of the collection, on which a function should be applied.

  - **Element order**. The order in which a function is applied on the elements of the selected subset.

- **Function selection**. The selection of a function to be applied in the calculation aspect.
- **Calculation**. The actual calculation performed by the selected function on the selected element.
- **Result**. The materialization or aggregation of the result of the mapping.

General mapping includes, of course, simple mapping, but it also embraces traversals such as filtering and folding.

In simple mapping the element selection is trivial in the sense that all elements of the collection are selected. A need for element selection can be handled by filtering before simple mapping. In addition, in simple mapping a fixed function is applied on each element. Therefore the function selection is trivial. In functional programming the element order aspect of simple mapping is of no significance, whereas in imperative programming the ordering aspect is crucial.

It is a virtue of simple mapping that the calculation aspect is strictly separated from the traversal aspect. It implies that the evaluation/action on each element of the collection is done without any knowledge about the way the elements are traversed. In simple mapping, the traversal is not in any way dependent on the outcome of the calculations that are carried out on individual elements of the input collection.

The traversal aspect and the result aspect of simple mapping are intertwined. As already mentioned, simple mapping always produces an output collection of the same form as the input collection. The resulting collection of simple mapping is formed in the “immediate slipstream” of the traversal.

In functional programming, simple mapping is used extensively on linear list structures together with filtering and folding. This leads to a programming style with multiple pipelined traversals that involves simple mapping, filtering, and possibly ends with a folding. Both filtering and folding can be understood in relation to the four aspects of general mapping introduced above. Filtering is responsible for element selection; The calculation and the result aspects of filtering are trivial. Folding is responsible for aggregating a result by pair-wise composition of elements from the list. The element selection and calculation aspects of folding are both trivial. Use of mapping, filtering, and folding separate different concerns at the expense of efficiency, due to multiple traversals.

We may indeed implement a generalized version of the `map`, `fold` and `filter` functions that combines simple mapping, filtering and folding via function parameters `f`, `c` and `b`. `f` is the function to be applied in the calculation aspects, `c` is the function that controls the result aspects, and `b` is the base value to be returned when the list is empty.

```
fun genmap f c b [] = b
  | genmap f c b (h::t) =
      c ((f h),(genmap f c b t))
```

We may obtain map, imp_map, fold and filter by suitable instantiation of genmap:

```
fun map f l =
  genmap f cons [] l

fun imp_map p l =
  genmap p (fn (x,y) => ()) () l

fun fold f b l =
  genmap (fn x => x) f b l

fun filter p l =
  genmap (fn x => x)
         (fn (x,y) => if (p x)
                         then (x::y)
                         else y)
         [] l
```

In a pure functional language, the order in which mapping, filtering or folding takes place is immaterial. However, in impure strict languages such as SML, the recursive call may need to take place in reverse order due to side effects, e.g. folding in the reverse order may be defined as:

```
fun foldl f b [] = b
  | foldl f b (h::t) = foldl f (f (h, b)) t
```

genmap may also be further generalized to take the order of traversal into account:

```
fun genmapm false f c b (h::t) =
      genmapm false f c (c(f h,b)) t
  | genmapm true f c b (h::t) =
      c ((f h),(genmapm true f c b t))
  | genmapm _ f c b [] = b
```

genmap uses a boolean flag to decide in which order the filtering/mapped function f and the folding function c are applied. When the flag is true the traversal is in reverse order of the list as the recursive call to genmap will ensure that the list is followed until the end before f and c are applied as the function returns from the recursion. When the flag is false the argument (c(f h,b)) will ensure that the result is passed to the recursive call of genmap and the functions will thus be applied in the order of the elements of the list. Clearly the order of traversal only matters if c or f have side effects.

Note that function selection in genmap is still trivial. We return to a further generalization in Section 6.

Simple mapping can also be applied on non-linear data structures, such as tree structures. Such structures are usually defined using discriminated union types, e.g.:

```
datatype 'a tree =  Leaf of 'a  | Node of 'a *
                                         'a tree * 'a tree
```

and map is straightforwardly generalized to:

```
fun treemap f (Leaf x) = Leaf (f x)
  | treemap f (Node(x,n1,n2)) =
       Node(f x,treemap f n1, treemap f n2)
```

Simple functional mapping applied on a tree produces another tree, which in an additional traversal can be folded to a simple value. Preorder, inorder, and postorder traversals represent different element orderings in imperative tree mapping. See e.g. Harrison's book on Abstract Data Types in Standard ML [5].

It is common to encode some application specific structure into the tree type, e.g. a type for abstract syntax trees of arithmetic expressions would usually be defined as:

```
datatype exp = PlusExp of exp * exp
             | MinusExp of exp * exp
             | TimesExp of exp * exp
             | DivideExp of exp * exp
             | Identifier of string
             | Literal of int
```

When encoding an application specific structure into the tree type, application specific functions for mapping, filtering and folding must be constructed. It is common to write specialized functions in such a way that they combine one or more traversals and the final folding. One reason for this is that in many practical settings multiple tree traversals and construction of intermediate tree structures are not desired. However, this blurs the mapping idea in favour of the application of a specific recursive function in which the calculation aspect and the traversal aspects are intertwined.

4 VISITING IN OBJECT-ORIENTED PROGRAMMING

The variations of mapping described in the previous section are primarily rooted in the functional programming paradigm. We now discuss mapping in the context of object-oriented programming. Due to the influence of the visitor design pattern the term "visiting" is often used instead of "mapping".

The mapping idea from above relies on function parameters, along the lines of `map(f,collection)` where `f` is a function. In many object-oriented programming languages it is not possible to pass a function/method (such as `f`) as parameter to another function/method (such as `map`). Instead, the function `f` must be embedded in some class `C`, and an object `o` of class `C` must be passed instead of `f`: `map(o,collection)`. In itself, this leads to clumsy programs. On the other hand, the class `C` behind the object `o` can be used to group several methods which are related to a particular kind of traversal, and it can be used to hold some state which is relevant for the traversal. This leads to the ***Visitor*** design pattern.

When working in the object-oriented programming paradigm, linear structures, and in particular tree structures, are often structured according the ***Composite*** design pattern [4]. A ***Composite*** design pattern is based on a recursive data structure in which inner nodes and leaves have a common, uniform interface. A ***Composite*** design pattern may involve many different classes. The function selection aspect of general mapping can elegantly be controlled by use of virtual methods in the class hierarchy of the ***Composite*** design pattern.

In order to be concrete we study sample traversals of a composite structure. The example throughout this paper will be operations on abstract syntax trees (ASTs) of arithmetic expressions. This is a typical example, both in the literature about design patterns, but also in practical "language processing situations", such as in the domain of compiler construction. The ASTs that we work with are composed on the basis of the following BNF grammar, which corresponds to the SML `exp` datatype discussed in Section 3.

```
Exp ::= PlusExp | MinusExp | TimesExp |
        DivideExp | Identifier | Literal
PlusExp ::= Exp + Exp
MinusExp ::= Exp - Exp
TimesExp ::= Exp * Exp
DivideExp ::= Exp / Exp
```

The given BNF gives rise to a number of expression classes which together form a ***Composite*** in which `Identifier` and `Literal` are leaves. An AST may be traversed with many different purposes, such as evaluation of the value of an expression, type checking the expression, generation of lower-level code for the expressions, or source-level pretty printing of the expression. The running example illustrates an expression evaluator and a reverse polish pretty printer. These are both instances of functional mappings that involve built-in folding of the tree structure to a simple value.

The natural object-oriented solution

The natural object-oriented expression traversal solution is illustrated as C# programs in Appendix A. A given traversal is programmed by a virtual method in class `Exp`, and in the subclasses of `Exp`. (To save some space we have left out the classes for `MinusExp` and `DivideExp`). The properties of this solution are well-known:

1. Methods for different kinds of traversal (evaluation and reverse polish pretty printing) are spread throughout all the expression classes.
2. The traversal, calculation, and result aspects of general mapping are all intertwined.
3. The function selection aspect is governed by virtual functions in the `Exp` class hierarchy.
4. If an additional kind of traversal is needed it is necessary to change the source program of all involved expression classes. This is a particular problem in case the source programs of the classes are not available.

The observations in item 1 and item 4 give rise to the ***Visitor*** design pattern, which we discuss next.

The Visitor design pattern

The ***Visitor*** design pattern is typically built on top of the ***Composite*** design pattern. The ***Visitor*** pattern unites all methods of a particular traversal in a single class. Thus, the ***Visitor*** pattern is a refactoring of the natural object-oriented solution in Appendix A.

In the introduction to Section 4 we discussed the transition from `map(f,collection)` to `map(v,collection)`, where `v` is an instance of a class, say `V`. The class `V` holds `f` as an instance method. The class `V` can now be understood as a ***Visitor***, which encapsulates different functions to be applied on different types of elements. (This can be understood in light of the Function selection aspect of general mapping). Traversal with a ***Visitor*** is usually initiated by `v.map(collection)`. As a matter of terminology, the name "`map`" is substituted by "`visit`", and it leads to the activating form `v.visit(collection)`.

We now look at a version of the program in Appendix A which is programmed according to the ***Visitor*** design pattern. The ***Visitor*** pattern provides an interface from the `Exp` classes to the traversal functionality via the `Accept` method. The

Visitor pattern does not separate the traversal and the calculation aspects of general mapping, although extensions, such as the `depthfirstadaptor` in SableCC [3], provide a refinement of the pattern where functionality for pre and post depth first traversal may be specified. The ***Visitor*** design pattern should only be used for a stable `Exp` class hierarchy for which we anticipate a number of different traversals in the future. If the `Exp` class hierarchy is extended with new subclasses, each of the `Exp` Visitor classes must be modified with additional methods for these new subclasses.

The new version of the C# program from Appendix A is shown in Appendix B. In this solution the class `Exp` and its subclasses support visiting at an abstract level via the virtual `Accept` method. A given visiting need is implemented in a class of its own, which implements the `Visitor` interface. The expression classes have no concrete knowledge of any particular visitor. The `Visitor` interface is type parametrized by `T` and `D`, where `T` is the result type of the `Visit` methods, and `D` is the type of some extra information which needs to be passed around during the visiting process.

The solution shown in Appendix B requires a lot of *object-oriented scaffolding* in terms of the `Visitor` interface, the classes that implement `Visitor`, the `Visit` methods, and the `Accept` methods. The `Visit` and `Accept` methods are indirectly mutually recursive: Expression objects are passed as parameters to `Visit` methods, and visitor objects are passed as parameters to `Accept` methods. The natural object-oriented solution shown in Appendix A is simpler, due to direct recursive calls and the fact that most necessary information is present inside the expression classes. In our example, the use of the ***Visitor*** design pattern adds more than 50% extra source code on top of the natural object-oriented solution.

The reasons behind the mentioned object-oriented scaffolding can be explained in two different ways. It can be seen as the price we have to pay for generalizing, refactoring and encapsulating the methods related to a given kind of traversal. It can, alternatively, be seen as a way to simulate double dispatch on a collection type and a visiting type (see below). Either way, use of the ***Visitor*** design pattern represents an algorithmic complication because the traversal must be programmed via indirect recursion in the methods of two different classes.

A double dispatch solution

In reality, the ***Visitor*** solution discussed above relies on a double dispatch on the `Exp` type and the `Visitor` type. Because most object-oriented programming languages (such as Java and C#) only support single dispatch, the double dispatch is realized in a two step process: First, dispatch takes place on the visitor:

```
visitor.visit(expression, ...)
```

The visitor, in turn, dispatches on an expression:

```
expression.accept(visitor, ...)
```

The additional code complexity of the ***Visitor*** design pattern becomes apparent if the ***Visitor*** design pattern is viewed as a way to simulate double dispatch. The calls of both `visit` and `accept` methods in two different classes make it difficult to understand and follow the traversal and the calculations during the traversal. With this perspective, the ***Visitor*** design pattern is quite a burden, at least for less experienced programmers.

In object-oriented programming languages that support multiple dispatch the visitor pattern can be programmed more directly. In Appendix C we show a double dispatch solution in CLOS [6, 12] which relies on multi-methods. The mentioned two-step process is substituted by a single step:

```
visit(expression, visitor, ...)
```

The activation of the `visit` function dispatches on both parameters in order to locate an applicable `visit` method. The `visit` methods recurse directly instead of indirectly. It is worth noticing that the expression class hierarchy is not at all affected by the visiting needs. Thus, no `accept` methods are necessary. The visitor is programmed in a so-called generic function outside the classes. This is a direct consequence of the use of multi-methods.

The encapsulation and grouping of all methods that contribute a given kind of visitor is an important objective of the ***Visitor*** design pattern. In a solution based on double dispatch, where the methods are localized outside the classes, this objective must be achieved by other means. A module or package concept can be used to group related methods.

The Walkabout visitor

In the paper *The Essence of the Visitor Pattern* [11] Palsberg and Jay have described how to provide a generic visitor class which they call `Walkabout`. (Notice that the word “generic” - in this context - has nothing to do with type parametrization). The `Walkabout` class contains a single `visit` method which can be specialized to visit a particular class, for instance the `Exp` class in our example. The selection of the appropriate visit method is programmed by means of reflection. The default behaviour of the visit method, applied on some object `o`, is to traverse all objects reachable from `o`. The properties of the walkabout approach can be summarized as follows:

- All visitor needs can be programmed as visit methods in the `Walkabout` class and its subclasses.
- The visited classes (`Exp` and it's subclasses in our example) are not at all affected by the visitor. In particular, there is no need for an `accept` method in these classes.

- The `visit` method of class `Object`—which serves as the default visit method—is clumsy and slow due to the use of reflection.

Relative to general mapping the walkabout approach can be characterized as follows:

- The traversal and the calculation aspects are intertwined. The element selection is carried out by the `visit` method. The traversal is—per default—almost all-embracing because the `visit` method of class `Object` traverses all references to all reachable objects.
- The function selection is programmed explicitly.
- The result aspect is obtained by imperative means (change of the state of the `Walkabout` object, for instance). A functional `visit` method in the `Walkabout` class has not been discussed.

Visitor Componentization

In the paper *Componentization: The Visitor Example* [9] Meyer and Arnout introduce a "componentizable" visitor in Eiffel. The goal of this work is to provide for visiting via a single, reusable class in the class library of Eiffel. The `visit(c:C)` method in an instance of the `Visitor[C]` class carries out visiting of `C` objects. Relative to our example, `C` could be the class `Exp`. The `Visitor` class is implemented in Eiffel [7, 8], and it relies on programming language concepts which are specific to Eiffel. An instance of the `Visitor` class accepts a collection of procedures. In this context, a collection of procedures is dealt with by an Eiffel tuple, and procedures are represented as Eiffel agent objects. An instance of the Visitor class applies a selected procedure to elements of the collection of `C` objects during a traversal. Inside the ***Visitor***, the visitor procedures are organized such that the selection of the appropriate procedure does not require linear search. The selection of the most appropriate procedure relies on reflection [1].

5 VISITOR COMBINATION

In the paper *Visitor Combination and Traversal Control* [14], Visser describes how to combine a number of visitors to a single visitor. The combinations can be seen as aggregation of two (or more) visitors to a single visitor. The aggregated visitor itself implements the `Visitor` interface. The paper therefore, in reality, explains how to build visitors that adhere to the ***Composite*** design pattern [4].

One of the visitor combinations from [14] is called `Sequence`. Figure 1 shows the `Sequence` visitor for the expression classes. The `Sequence` visitor combines two other visitors, `first` and `second`, sequentially, as a pipelined combination (see Section 2).

Thus, the activation `someSequence.Visit(someExpression, ...)` covers a full tree traversal with the visitor `first` followed by a full tree traversal with the visitor `second`. Because of the coupling between the traversal and calculation aspects in the ***Visitor*** pattern, it is not easy to imagine an interleaved combination of two visitors.

Only imperative visitors are illustrated in [14]. Our expression `Visitor` interface, as shown in Appendix A, prescribes `Visit` operations that return some type `T`. The `Sequence` visitor, shown in Figure 1, is actually an imperative visitor. The value returned by the `first` visitor is ignored. The value of the `second` visitor is returned as the value of the combined visitor. This approach leads to an unfortunate asymmetry between call of `Accept` on `first` and call of `Accept` on `second` in Figure 1.

From a functional programming point of view, it is natural to program a more genuine functional combination of two visitors. This leads to the class `Functional-Composition`, which is shown in Figure 2. The `first` visitor produces data of type `T1` which is constrained to be of type `Exp`. This allows the second visitor to be applied on the result returned by the first visitor. The `second` visitor produces data of a non-constrained type `T2`. This combination can be used to (1) transform an expression tree followed by (2) some interpretation/rendering of the transformed tree. In Figure 2 the use of extra information relative to the combined visitor is also improved. `D1` and `D2` are the types of the extra information passed to the `first` and `second` visitors respectively. The extra information passed to a `FunctionalComposition` visitor is of type `Product<D1,D2>`, where the generic class `Product` is a simple, straightforward pairing of two values of type `T1` and `T2`.

Another combination of visitors from [14] is called `Choice` and corresponds to some extent to the filter in the functional paradigm.

Finally, [14] describes how to control the traversal. This corresponds to element selection and element order of general mapping, as addressed in Section 3.

The issue of combining traversals is important and worthwhile. However, Visser's proposed combination of **Visitors** is much more complicated to deal with than the combination of mappings described in Section 2 of this paper. The combinations shown in Figures 1 and 2 become even more complicated if it is attempted to lift them from specific expression visitors to generic visitors. Many specialized visitor classes are involved, and the derived fragmentation of control blurs the algorithmic comprehension of the traversal for most programmers.

6 MIXED-PARADIGM SOLUTIONS

Section 3 discussed generalized mapping and showed how a generalized mapping function could be defined. However, the function selection aspect of generalized mapping is trivial as lists in traditional functional programming languages, such as

```
class Sequence<T,D>: Visitor<T,D>{
  private Visitor<T,D> first, second;

  public Sequence(Visitor<T,D> first,
                  Visitor<T,D> second){
    this.first = first; this.second = second;
  }

  public T Visit(PlusExp e, D extraInfo){
    e.Accept(first, extraInfo);
    return e.Accept(second, extraInfo);
  }

  public T Visit(TimesExp e, D extraInfo){
    e.Accept(first, extraInfo);
    return e.Accept(second, extraInfo);
  }

  public T Visit(Identifier e, D extraInfo){
    e.Accept(first, extraInfo);
    return e.Accept(second, extraInfo);
  }

  public T Visit(Literal e, D extraInfo){
    e.Accept(first, extraInfo);
    return e.Accept(second, extraInfo);
  }
}
```

Figure 1: The `Sequence` visitor programmed in C#.

SML, consist of elements of the same type. In object-oriented programming it is quite common to have collections of mixed types as long as the elements share a common super type, which ultimately could be the type `Object` of all objects.

In languages combining higher-order functional programming with object-oriented programming, such as F#, a list with mixed element types, may then be defined as e.g.:

```
let mixedlist =
  [((new ListElementType1()) :> ListElements) ;
   ((new ListElementType2()) :> ListElements)];;
```

Note that in F# the elements, of type `ListElementType1`, respectively `ListElementType2`, explicitly have to cast to the common supertype `ListElements` using the `:>` operator.

Clearly `map`, `fold`, etc. may be used on such lists and the function selection may be made less trivial as we can generalize the functions to be applied to dispatch on

```
class FunctionalComposition<T1,T2,D1,D2>:  Visitor<T2,Product<D1,D2>>
  where T1: Exp {
   private Visitor<T1,D1> first;
   private Visitor<T2,D2> second;

   public FunctionalComposition(Visitor<T1,D1> first, Visitor<T2,D2> second){
     this.first = first; this.second = second;
   }

   public T2 Visit(PlusExp e, Product<D1,D2> extraInfo){
     T1 result = e.Accept(first, extraInfo.First);
     return result.Accept(second, extraInfo.Second);
   }

   public T2 Visit(TimesExp e, Product<D1,D2> extraInfo){
     T1 result = e.Accept(first, extraInfo.First);
     return result.Accept(second, extraInfo.Second);
   }

   public T2 Visit(Identifier e, Product<D1,D2> extraInfo){
     T1 result = e.Accept(first, extraInfo.First);
     return result.Accept(second, extraInfo.Second);
   }

   public T2 Visit(Literal e, Product<D1,D2> extraInfo){
     T1 result = e.Accept(first, extraInfo.First);
     return result.Accept(second, extraInfo.Second);
   }
 }
```

Figure 2: The `FunctionalComposition` visitor programmed in C#.

the type of elements in the list. In F# we can program this dispatch explicitly using pattern matching on the dynamic type, e.g.:

```
let fmixed (e:ListElements) =
  match e with
  | :? ListElementType1 as ee -> 1
  | :? ListElementType2 as ee -> 2 ;;
```

With this set-up, the result of the expression `map fmixed mixedlist` is `[1;2]`. Clearly dynamic dispatch may be used as the basis of function selection, e.g. if the super type `ListElements` defines a virtual method `f`, specialized by each subclass, we may define:

```
let fmixed args (e:ListElements) = e.f args;;
```

Paramount to the formulation of the map function based on pattern matching is the fact that the type of a list is defined as a disjoint union type, e.g. a list type in F# could be defined as:

```
type 'a mylist = Empty
               | Cons of 'a * 'a mylist
```

In fact the built-in list type in F# is defined as:

```
type 'a list = ([])
             | (::) of 'a * 'a list
```

where (::) is defined as an infix operator. Using the type mylist as defined above, map may be defined as:

```
let rec map f l =
  match l with
  | Empty -> Empty
  | Cons(h,t) -> Cons(f(h),(map f t));;
```

If we recast the definition of lists in an object-oriented setting, we typically do not use discriminated union type, but use a common (abstract) class and subtypes for each variant, following the ***Composite*** design pattern [4].

In F# an object-oriented version of list may be defined as:

```
type 'a Mylist =
  class
    new() = {}
  end;;

type 'a Mycons =
  class
    inherit 'a Mylist
    val h: 'a
    val t: 'a Mylist
    new(a,b) = {h = a; t = b}
  end;;

type 'a Myempty =
  class
    inherit 'a Mylist
    new() = {}
  end;;
```

Keeping to a functional programming style, map on the above definition of list, could be defined in F# as:

```
let rec mymap f (l:'a Mylist) =
  match l with
  | :? ('a Mycons) as e ->
      (new Mycons<'a>(f(e.h),(mymap f (e.t))))
         :> 'a Mylist
  | :? ('a Myempty) as e -> (new Myempty<'a>())
         :> 'a Mylist;;
```

mymap takes as arguments a function f and an object-oriented list l. Based on pattern matching on the run-time concrete type of the list, mymap either recurses over the list or returns an empty list. This formulation of map is very close in spirit to the functional version based on discriminated union, but has a heavy notational overhead. To reduce this overhead and make the function more "functional" we may use the notion of active patterns [13]:

```
let (|Cons|Nil|) (l:'a Mylist) =
match l with
   | :? ('a Mycons) as e ->
      Cons(e.h,e.t)
   | :? ('a Myempty) as e -> Nil;;
```

The notation (|Cons|Nil|) introduces two so-called structured names Cons and Nil which may be used in pattern matching later. We may e.g. use them in the definition of mymap:

```
let rec mymap f (l:'a Mylist) =
match l with
  | Cons(h,t) -> cons(f(h),mymap f t)
  | Nil -> nil()
```

where cons and nil are defined as:

```
let cons (h,t:'a Mylist) =
    (new Mycons<'a>(h,t)):> 'a Mylist;;

let nil () =
    (new Myempty<'a>()):> 'a Mylist;;
```

Clearly mymap may be generalized, using active patterns, with two additional functions for combining the head and tail, as well as processing the empty list as in the functional styled genmap, thus achieving a function implementing all aspects: traversal, function selection, calculation and result, of generalized mapping:

```
let rec TreeWalker f c b (ee:Exp) =
  match ee with
  | :? PlusExp as e -> (c (f (TreeWalker f c b e.e1))  (f (TreeWalker f c b e.e2)))
  | :? MinusExp as e -> (c (f (TreeWalker f c b e.e1))  (f (TreeWalker f c b e.e2)))
  | :? TimesExp as e -> (c (f (TreeWalker f c b e.e1))  (f (TreeWalker f c b e.e2)))
  | :? DivideExp as e -> (c (f (TreeWalker f c b e.e1))  (f (TreeWalker f c b e.e2)))
  | :? Identifier as e -> (b e.f1)
  | :? IntegerLiteral as e -> (b e.f1);;
```

Figure 3: A generalized tree-walker function programmed in F#.

```
let rec genmap f c b (l:'a Mylist) =
 match l with
 | Cons(h,t) ->
    (c (f h,(mymap f t)))
 | Nil -> b)
;;
```

Following the object-oriented styled definition of lists above, it is relatively straightforward to use a similar pattern to implement a generalized tree-walker function for expression trees. A generalized tree-walker function is shown in Figure 3. However, in most cases we would prefer not to use the same functions `c` and `f` as we walk a tree, but rather use specific functions depending on the node type. This may be achieved using a combination of the higher-order function technique used for the `TreeWalker` function and the `Visitor` class. An example is shown in Appendix D where the running example of operations on abstract syntax trees is presented. We bundle a set of related functionality using an instance of a concrete class (i.e. an object) implementing an abstract visitor class. This solution may especially be appropriate if the functions share some state that could be encapsulated by the object.

Thus by mixing functional and object-oriented programming we achieve a generic program for visiting/treewalking, thus eliminating the scaffolding from the class hierarchy needed in the visitor pattern, but still keeping the visitor functions together in an implementation of the `Visitor` interface.

7 CONCLUSION

In this paper we have reviewed mapping in functional programming and visiting in object-oriented programming. We have looked at generalizations in both paradigms and seen how the concepts can be amalgamated in languages that support both paradigms. Especially by using functional programming techniques with higher-order functions, we may move the scaffolding code for tree walking, found in the visitor pattern, from the base classes to a higher-order function as demonstrated in Section 6. Combining two rounds of mapping is second nature in the functional paradigm. Combined with filtering and folding, mapping provides a powerful programming technique, which may be generalized to a single higher-order function as demonstrated in section 2. In the object-oriented paradigm this is not straightforward, but using type parametrization it is possible to transfer some of the concepts from the functional world into the object-oriented world, as illustrated in Section 5.

Both element selection and function selection are needed in the visitor pattern, thus a double dispatch solution is elegant. In Section 4 we showed how this can be done in CLOS. However, none of the current mainstream languages support double dispatch, though experimental prototypes, such as Multi-Java [2], exist. In CLOS functions are placed outside the class definitions, and CLOS offers only a rudimentary language mechanism, in the form of namespaces, for bundling together related functionality.

The F# solution, described in Section 6, bundles related functionality using an instance of a concrete class (i.e. an object) implementing an abstract visitor class. This may be an appropriate solution if the functions share some state that could be encapsulated by the object. However, in many cases this is a heavy solution as the bundle of functionality does not share state, and alternative solutions are called for. Eiffel, as described in Section 4, provides language constructs in the form of tuples and agent objects, that cater for such bundling, but the selection of the most appropriate method relies on the heavy machinery of reflection.

In most cases bundling of related functionality could be solved with a class consisting of a set of static methods. This could be an elegant solution if the language supports static overloading. One may also prefer a language mechanism, such as mixins, as found e.g. in the SCALA language [10], possibly combined with a concept like extension methods from C# 3.0, that will allow the mixin of methods at a later stage than class definition time. Alternatively type parametrization may be sufficient.

The structure of the collection to be visited has to be known to the programmer of the respective functions, be it mapping on lists or treewalkers/visitors on trees, as quite at bit of semantic knowledge is encoded in the structure, e.g. the structure of expression trees records knowledge of the type of nodes. However, if the structure of the collection is not known, we will need to be able to inspect the structure. In Section 4 we looked at one approach where reflection is used. However, there is a high performance penalty for using reflection. It would be preferable to include language constructs that directly would allow the programmer to investigate this structure dynamically, for example by allowing programmatic inspection of a class' structure.

Looking beyond mapping and visiting, other design patterns might also be abstracted using language constructs, thus eliminating the scaffolding code from the base classes. We have briefly looked at the Singleton design pattern. However, it seems that removing the scaffolding code from the base class in this case, would require language mechanisms that allow abstractions over classes, thus treating classes as first class classes. It is a matter of future research to determine if such language constructs are implementable and useful.

REFERENCES

[1] Karine Arnout. *From Patterns to Components.* PhD thesis, Swiss Federal Institute of Technology Zurich, 2004. Doctoral thesis ETH no. 15500.

[2] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3), May 2006.

[3] E. Gagnon and L. Hendren. SableCC – an object-oriented compiler framework, 1998. http://citeseer.ist.psu.edu/article/gagnon98sablecc.html.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.

[5] Rachel Harrison. *Abstract Data Types in Standard ML.* John Wiley & Sons, 1993.

[6] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley Publishing Company, 1989.

[7] Bertrand Meyer. *Eiffel the Language.* Prentice Hall, 1992.

[8] Bertrand Meyer. *Object-oriented software construction, second edition.* Prentice Hall, 1997.

[9] Bertrand Meyer and Karine Arnout. Componentization: The visitor example. *Computer*, 39(7):23–30, 2006.

[10] Martin Odersky. The scala experiment – can we provide better language support for component systems? In *Proc. ACM Symposium on Principles of Programming Languages*, pages 166–167, 2006.

[11] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society.

[12] Guy L. Steele. *Common Lisp, the language, 2nd Edition.* Digital Press, 1990.

[13] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. 2007. http://blogs.msdn.com/dsyme/attachment/2044281.ashx.

[14] Joost Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 270–282, New York, NY, USA, 2001. ACM Press.

[15] John Vlissides. Visitor in frameworks. *C++ Report*, November/December 1999.

A NATURAL OBJECT-ORIENTED VISITING

```
using System;
using System.Collections.Generic;

namespace NaturalOOSolution
{
    abstract class Exp{
      public abstract int Eval(Environment env);
      public abstract string ReversePolish();
      protected const string SPACE = " ";
    }

    class PlusExp : Exp {
        private Exp e1, e2;

        public PlusExp(Exp a, Exp b){
            e1 = a;
            e2 = b;
        }

        public override int Eval(Environment env){
          return e1.Eval(env) + e2.Eval(env);
        }

        public override string ReversePolish(){
          return e1.ReversePolish() + SPACE + e2.ReversePolish() + SPACE + "+" ;
        }
    }

    class TimesExp : Exp {
        private Exp e1, e2;

        public TimesExp(Exp a, Exp b){
            e1 = a;
            e2 = b;
        }

        public override int Eval(Environment env){
          return e1.Eval(env) * e2.Eval(env);
        }

        public override string ReversePolish(){
           return e1.ReversePolish() + SPACE + e2.ReversePolish() + SPACE + "*";
        }
    }

    class Identifier : Exp {
        private string name;

        public Identifier(string s) {name = s;}

        public override int Eval(Environment env){
            return env[name];
        }

        public override string ReversePolish(){
          return name;
        }
    }

    class Literal : Exp {
        private string litString;

        public Literal(string s) { litString = s; }

        public override int Eval(Environment env){
          return Int32.Parse(litString);
        }

        public override string ReversePolish(){
          return litString;
        }
    }

    class Environment: Dictionary<String, int>{};

    class Program {

        public static void Main() {
            Exp ast = new TimesExp(new PlusExp (new Literal("1"), new Identifier("x")),
                                   new Literal("2"));
            Environment env = new Environment(); env.Add("x",7);
            Console.WriteLine(ast.Eval(env));
            Console.WriteLine(ast.ReversePolish());
        }
    }
}
```

B THE VISITOR DESIGN PATTERN

```
using System;
using System.Collections.Generic;

namespace VisitorSolution
{

    interface Visitor<T,D>{
      T Visit(PlusExp e, D extraInfo);
      T Visit(TimesExp e, D extraInfo);
      T Visit(Identifier e, D extraInfo);
      T Visit(Literal e, D extraInfo);
    }
```

```
abstract class Exp{
  public abstract T Accept<T,D>(Visitor<T,D> v, D x);
}

class PlusExp : Exp {
  private Exp e1, e2;

  public PlusExp(Exp a, Exp b){
      e1 = a;
      e2 = b;
  }

  public Exp LeftOperand{get {return e1;}}
  public Exp RightOperand{get {return e2;}}

  public override T Accept<T,D>(Visitor<T,D> v, D x){
    return v.Visit(this, x);
  }
}

class TimesExp : Exp {
  private Exp e1, e2;

  public TimesExp(Exp a, Exp b){
    e1 = a;
    e2 = b;
  }

  public Exp LeftOperand{get {return e1;}}
  public Exp RightOperand{get {return e2;}}

  public override T Accept<T,D>(Visitor<T,D> v, D x){
    return v.Visit(this, x);
  }
}

class Identifier : Exp {
  private string name;

  public Identifier(string s) {name = s;}

  public string Name{get {return name;} }

  public override T Accept<T,D>(Visitor<T,D> v, D x){
    return v.Visit(this, x);
  }
}

class Literal : Exp {
  private string litString;

  public string LiteralString{get {return litString;}}

  public Literal(string s) {litString = s; }
```

```
    public override T Accept<T,D>(Visitor<T,D> v, D x){
      return v.Visit(this, x);
    }
  }

  class Environment: Dictionary<String, int>{};


  class Interpreter: Visitor<int, Environment> {
    public int Visit(PlusExp e, Environment env){
      return (e.LeftOperand.Accept(this, env) + e.RightOperand.Accept(this, env));
    }

    public int Visit(TimesExp e, Environment env){
      return (e.LeftOperand.Accept(this, env) * e.RightOperand.Accept(this, env));
    }

    public int Visit(Identifier e, Environment env){
      return env[e.Name];
    }

    public int Visit(Literal e, Environment env){
      return Int32.Parse(e.LiteralString);
    }
  }


  class ReversePolishConverter: Visitor<string, None> {

    private const string SPACE = " ";

    public string Visit(PlusExp e, None x){
      return e.LeftOperand.Accept(this, x) + SPACE + e.RightOperand.Accept(this, x) +
             SPACE + "+";
    }

    public string Visit(TimesExp e, None x){
      return e.LeftOperand.Accept(this, x) + SPACE + e.RightOperand.Accept(this, x) +
             SPACE + "*";
    }

    public string Visit(Identifier e, None x){
      return e.Name;
    }

    public string Visit(Literal e, None x){
      return e.LiteralString;
    }
  }

  class None {};

  class Program {
        public static void Main() {
            TimesExp ast = new TimesExp(new PlusExp (new Literal("1"), new Identifier("x")),
                                        new Literal("2"));  //  (1 + x) * 2
            Environment env = new Environment(); env.Add("x",9);
            Interpreter interpreter = new Interpreter();
            ReversePolishConverter reversPolishConverter = new ReversePolishConverter();
            None nothing = new None();

            Console.WriteLine(interpreter.Visit(ast, env));
            Console.WriteLine(reversPolishConverter.Visit(ast, nothing));
        }
    }
}
```

C DOUBLE DISPATCH VISITING IN CLOS

```
;;; Visitor in CLOS - Common Lisp Object System.

;; Expression Classes

(defclass Expression () ())

(defclass PlusExpression (Expression)
  ((a1 :initarg :firstAddend :accessor firstAddend)
   (a2 :initarg :secondAddend  :accessor secondAddend)))

(defclass MinusExpression (Expression)
  ((m :initarg :minuend :accessor minuend)
   (s :initarg :subtrahend :accessor subtrahend)))

(defclass TimesExpression (Expression)
  ((f1 :initarg :firstFactor :accessor firstFactor)
   (f2 :initarg :secondFactor :accessor secondFactor)))

(defclass DivideExpression (Expression)
  ((dividend :initarg :dividend :accessor dividend)
   (divisor :initarg :divisor :accessor divisor)))

(defclass Identifier (Expression)
  ((name :initarg :name :accessor name)))

(defclass IntegerLiteral (Expression)
  ((literal :initarg :literal :accessor literal)))



;; Visitor classes

(defclass Visitor () ())
```

```
(defclass Interpreter (Visitor) ())

(defclass ReversePolish (Visitor) ())



;; Visit multi methods.
;; Notice that the visit method specializes on both an Expression
;; and a (kind of) of visitor.

(defmethod visit ((e PlusExpression) (v Interpreter)  &optional extra)
  (+ (visit (firstAddend e) v extra) (visit (secondAddend e) v extra)))

(defmethod visit ((e MinusExpression) (v Interpreter) &optional extra)
  (- (visit (minuend e) v extra) (visit (subtrahend e) v extra)))

(defmethod visit ((e TimesExpression) (v Interpreter) &optional extra)
  (* (visit (firstFactor e) v extra) (visit (secondFactor e) v extra)))

(defmethod visit ((e DivideExpression) (v Interpreter) &optional extra)
  (/ (visit (dividend e) v extra) (visit (divisor e) v extra)))

(defmethod visit ((e Identifier) (v Interpreter) &optional extra)
  (lookup-identifier extra (name e)))

(defmethod visit ((e IntegerLiteral) (v Interpreter) &optional extra)
  (convert-string-to-integer (literal e)))



(defmethod visit ((e PlusExpression) (v ReversePolish)  &optional extra)
  (concatenate 'string  (visit (firstAddend e) v extra) " "
    (visit (SecondAddend e) v extra) " " "+"))

(defmethod visit ((e MinusExpression) (v ReversePolish) &optional extra)
  (concatenate 'string  (visit (minuend e) v extra) " "
    (visit (subtrahend e) v extra) " " "-"))


(defmethod visit ((e TimesExpression) (v ReversePolish) &optional extra)
  (concatenate 'string  (visit (firstFactor e) v extra) " "
    (visit (SecondFactor e) v extra) " " "*"))

(defmethod visit ((e DivideExpression) (v ReversePolish) &optional extra)
  (concatenate 'string  (visit (dividend e) v extra) " "
    (visit (divisor e) v extra) " " "/"))

(defmethod visit ((e Identifier) (v ReversePolish) &optional extra)
  (name e))

(defmethod visit ((e IntegerLiteral) (v ReversePolish) &optional extra)
  (literal e))
```

```
;; Auxiliary stuff.
(defun convert-string-to-integer (str &optional (radix 10))
  "Given a digit string and optional radix, return an integer."
  ; Details not relevant for this paper
)

(defun lookup-identifier (env name)
   (cdr (assoc name env :test (function equal))))


;; Some sample visiting:
(defun main ()
  (let* ((expr1 (make-instance 'PlusExpression
                 :firstAddend (make-instance 'IntegerLiteral :literal "3")
                 :secondAddend (make-instance 'IntegerLiteral :literal "2")))

         (expr2 (make-instance 'TimesExpression
                 :firstFactor (make-instance 'IntegerLiteral :literal "3")
                 :secondFactor (make-instance 'Identifier :name "var")))

         (expr3 (make-instance 'MinusExpression
                 :minuend expr1
                 :subtrahend expr2))

         (interpretation (make-instance 'Interpreter))
         (reverse-polish (make-instance 'ReversePolish))
         (env '(("x" . 5) ("var" . 7)))
         )
    (list (visit expr1 interpretation env)
          (visit expr2 interpretation env)
          (visit expr3 interpretation env)

          (visit expr1 reverse-polish)
          (visit expr2 reverse-polish)
          (visit expr3 reverse-polish)
          )))
```

D VISITING IN F#

```
type Exp =
  class
    new() = {}
  end;;

type PlusExp =
  class
    inherit Exp
        val e1:Exp
        val e2:Exp
        new(a,b) = {e1 = a; e2 = b}
  end;;

type MinusExp =
  class
    inherit Exp
        val e1:Exp
        val e2:Exp
        new(a,b) = {e1 = a; e2 = b}
  end;;

type TimesExp =
  class
    inherit Exp
        val e1:Exp
        val e2:Exp
        new(a,b) = {e1 = a; e2 = b}
  end;;

type DivideExp =
  class
    inherit Exp
        val e1:Exp
        val e2:Exp
        new(a,b) = {e1 = a; e2 = b}
  end;;

type Identifier =
  class
    inherit Exp
        val f1:string
        new(s) = {f1 = s}
  end;;

type IntegerLiteral =
  class
    inherit Exp
        val f1:string
        new(s) = {f1 = s}
  end;;

type Environment =
  class
    inherit System.Collections.Generic.Dictionary<string,int>
        new() = {}
end;;

let env = new Environment();;

let Lookup (s:string) =
  match env.TryGetValue(s) with
   | (_,x) -> x
;;
```

```
type 'a Visitor =
  class
    abstract member visitPlusExp: 'a * 'a -> 'a
    abstract member visitMinusExp: 'a * 'a -> 'a
    abstract member visitTimesExp: 'a * 'a -> 'a
    abstract member visitDivideExp: 'a * 'a -> 'a
    abstract member visitIdentifier: string -> 'a
    abstract member visitIntegerLiteral: string -> 'a
    new() = {}
end;;

let rec TreeWalker (c:'a Visitor) (ee:Exp) =
  match ee with
  | :? PlusExp as e -> (c.visitPlusExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? MinusExp as e -> (c.visitMinusExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? TimesExp as e -> (c.visitTimesExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? DivideExp as e -> (c.visitDivideExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? Identifier as e -> (c.visitIdentifier e.f1)
  | :? IntegerLiteral as e -> (c.visitIntegerLiteral e.f1);;

type Interpreter =
  class
    inherit int Visitor
    override x.visitPlusExp (x,y) = x + y
    override x.visitMinusExp (x,y) = x - y
    override x.visitTimesExp (x,y) = x * y
    override x.visitDivideExp (x,y) = x / y
    override x.visitIdentifier s = Lookup s
    override x.visitIntegerLiteral s = System.Int32.Parse s
    new() = {}
end;;

let inter (ee:Exp) = TreeWalker (new Interpreter() :> int Visitor) ee;;

let SPACE = " ";;

type ReversePolish =
  class
    inherit string Visitor
    override x.visitPlusExp (x,y) = x + SPACE + y + SPACE + "+"
    override x.visitMinusExp (x,y) = x + SPACE + y + SPACE + "-"
    override x.visitTimesExp (x,y) = x + SPACE + y + SPACE + "*"
    override x.visitDivideExp (x,y) = x + SPACE + y + SPACE + "/"
    override x.visitIdentifier s = (Lookup s).ToString()
    override x.visitIntegerLiteral s = s
    new() = {}
end;;

let reversepolish (ee:Exp) = TreeWalker (new ReversePolish() :> string Visitor) ee;;

let ast = new TimesExp(new PlusExp (new IntegerLiteral("1"), new Identifier("x")),
                                    new IntegerLiteral("2")) :> Exp;;

do env.Add("x",9);;
do System.Console.WriteLine(inter ast);;
do System.Console.WriteLine(reversepolish ast);;
```

ABOUT THE AUTHORS

Kurt Nørmark is an associate professor at the department of computer science at Aalborg University, Denmark. His research interests include programming languages and tools, including program documentation tools and the relationships between XML technology and programming technology. Contact him at normark@cs.aau.dk.

Dr. Bent Thomsen is an associate professor at the department of computer science at Aalborg University, Denmark. His research interests include object oriented, functional and concurrent programming, real-time programming, future programming languages and programming technology. Contact him at bt@cs.aau.dk.

Dr. Lone Leth Thomsen is an associate professor at the department of computer science at Aalborg University, Denmark. Her research interests include object oriented, functional and concurrent programming, web and BPM programming, future programming languages and programming technology. Contact her at lone@cs.aau.dk.