Functional Programming — SML

Steven Zeil

Nov. 10, 2003

Contents

Functional Programming

1. Overview
2. SML
3. Scheme

---

1 Overview

1. Imperative (reprise)
2. Functional

1.1 Imperative (reprise)

Imperative languages are characterized by

- control flow
- assignment
  - Data is *mutable*
  - Computation is accomplished via a sequence of state changes

---

- Assignments effects a state change by altering the value of one or more variables.
- Control flow sequences the assignments into the desired sequence of state changes.
  - sequencing $\Rightarrow$ time

1.2 Functional

Functional programming is characterized by

- Functions as first-class objects
- Expression evaluation (no control flow)
  - Data is *immutable*.
  - Data is constructed, not assigned

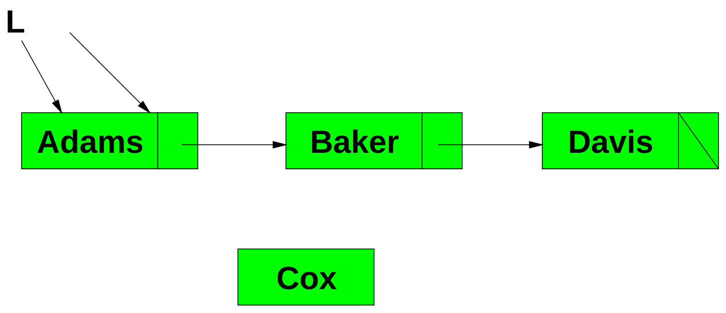
- Data values are *constructed* from simpler values.

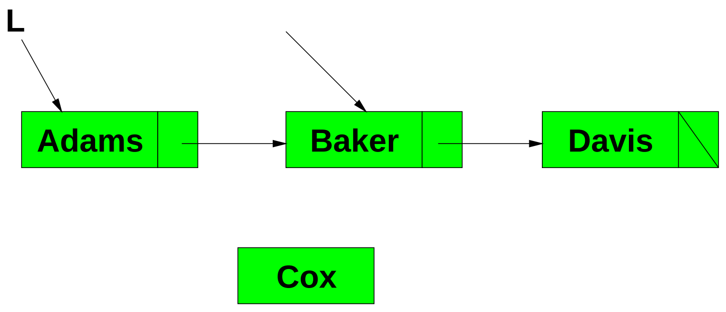
- Expression evaluation controls the flow of data.
  - Conceptually, instantaneous calculation

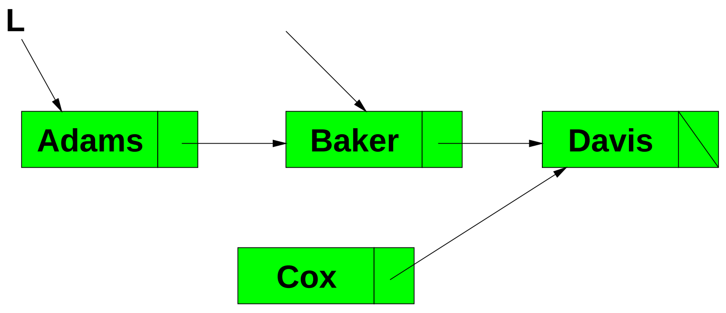
1.2.1 Constructive Data Manipulation

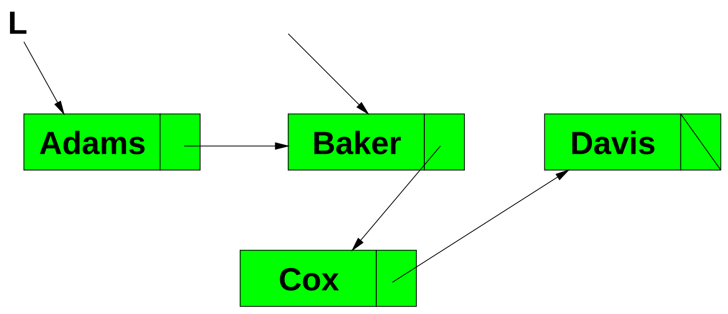
Consider the problem of adding something to the middle of a list.

An imperative, mutative approach would look like:

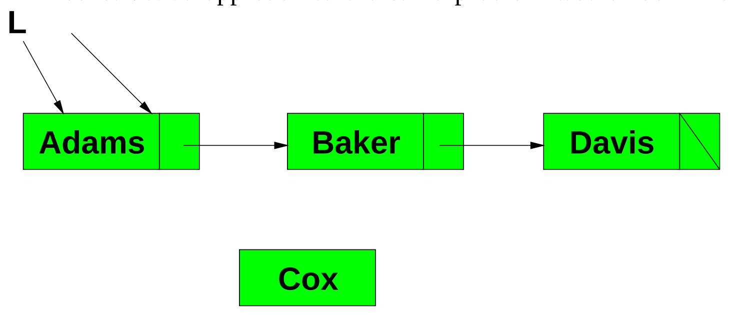


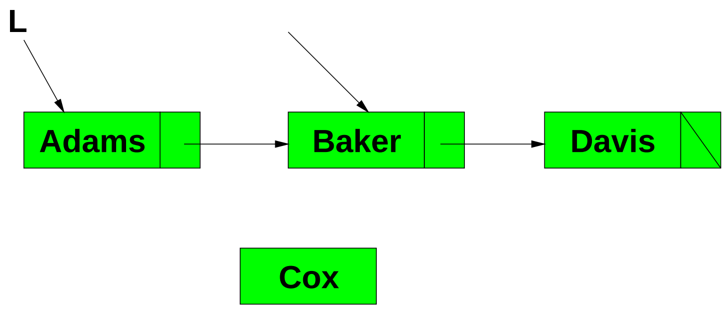


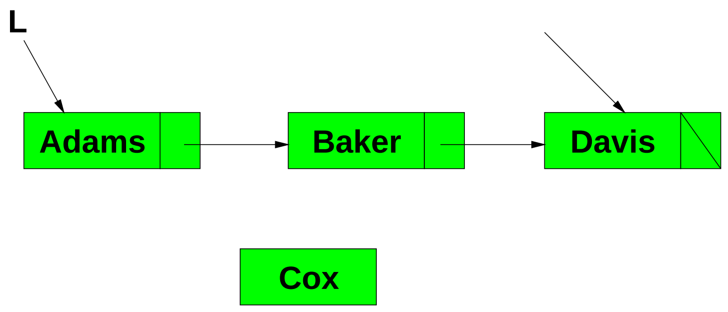


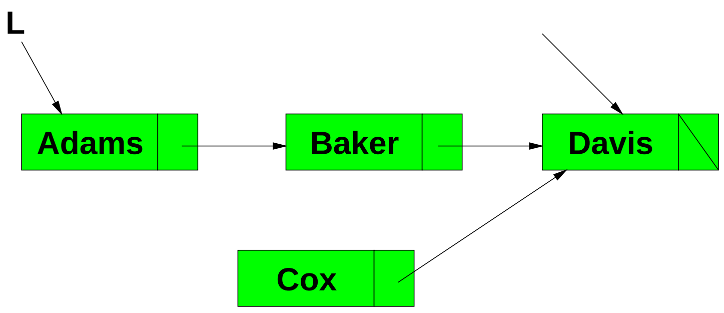


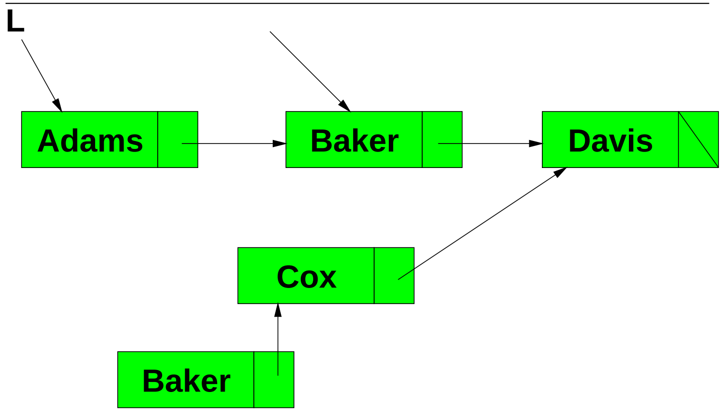
A *constructive* approach to the same problem would look like:

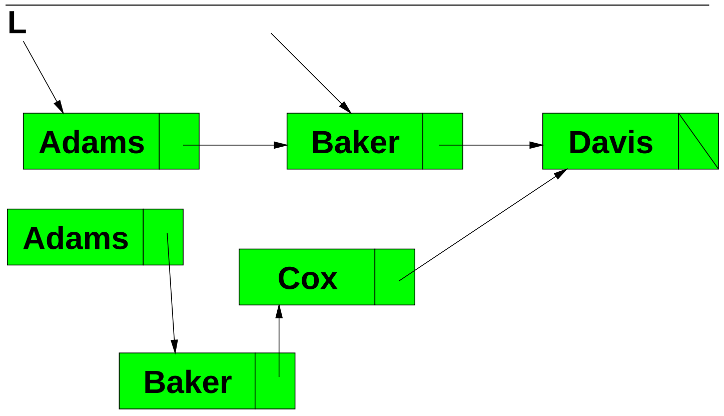












Although the constructive approach appears less efficient at first glance, this may be deceiving:

- If the data is actually pointers, shallow copy is relatively fast.
- If our application needs to retain access to both the old and the new list values, then the constructive approach copies fewer nodes.

- Constructive approach permits sharing of nodes among different lists.
- Constructive approach simplifies data management.

---

2 SML

1. Types
2. Functions
3. Expression Evaluation
4. Lexical Scope
5. SML Style
6. Type Construction

2.1 Types

1. Atomic Types
2. Constructors

2.1.1 Atomic Types

The basic types in SML are:

- Unit type: `()`, a kind of "null" type

- `bool`, with values `true` and `false`

  Main operators are

  – `andalso`, `orelse`, `not`

  – conditional expressions: `if` e_1 `then` e_2 `else` e_3

if x>0 **then** x **else** −x

  – `int`

  Negative integers are written with ˜, e.g., ˜23

- `string`: `"Hello"`
- `real`: 3.0, 0.314159E1, ˜42.1

2.1.2 Constructors

Some of the SML type constructors may remind you of our earlier discussion of types as sets.

- Tuples: t_1`*`t_2 is a cross-product.

  Values are written in parentheses, separated by commas: `(true, 1)`
- Lists: t_1 `list`
  - `nil`, also written as `[]`
  - `a::L` forms a list whose first element is `a` and whose remaining elements are in the list `L`

  Lists can be written as a series of "cons"s, `a::b::c::nil`, or in shorthand: `[a, b, c]`

- Records: {`name`$_1$=t_1`, name`$_2$=t_2`, ...`}

  Values are written similarly: {`name= "Zeil", id= 010101, isFaculty=true`}
  - Tuples are actually a special case of records, in which the "names" are 1, 2, ...
- Functions: t_1 `->` t_2 describes a function that accepts a parameter of type t_1 and returns a value of type t_2.

  `fun foo(s) = size(s)-1;`

2.2 Functions

1. Declarations
2. Application
3. 1st Class Objects

2.2.1 Declarations

A function declaration

⟨*fdecl*⟩ ::= `fun id` ⟨*formalparameter*⟩ = ⟨*exp*⟩

binds the name (`id`) to a function value.

```
fun absolute(x) =
    if x>=0.0 then x
              else 0.0-x;
fun len(x) =
    if null(x) then 0
               else 1+len(tl(x));
```

2.2.2 Application

Functions are applied to arguments by writing the call in prefix order: `abs x`

But any expression can be parenthesized, so `abs(x)` is just fine, too.

2.2.3 1st Class Objects

Functions are no different from any other type, in that we can pass them, operate on them, etc.

```
fun transform(L,f) =
   if null(L)
    then []
    else f(hd(L))::transform(tl(L),f);
```

2.3 Expression Evaluation

Although we tend to take this for granted, there are a number of possible ways to evaluate a given expression:

1. Innermost Evaluation
2. Outermost
3. Selective

Independently, we can choose

1. Left-To-Right
2. Nondeterministic

2.3.1 Innermost Evaluation

A function application ⟨*name*⟩ ⟨*param*⟩ is evaluated as follows:

1. Evaluate the expression ⟨*param*⟩
2. Substitute the result for the formal in the function body.
3. Evaluate the body expression.
4. Return its value as the answer

2.3.2 Outermost

A function application ⟨*name*⟩ ⟨*param*⟩ is evaluated as follows:

1. Substitute the actual param. for the formal in the function body.
2. Evaluate the body expression.
3. Return its value as the answer

2.3.3 Selective

In practice, we can't really go exclusively with any "pure" evaluation scheme.

What's wrong with innermost evaluation of

```
if x > 0 then transform(L, abs)
         else transform(L, negate);
```

?

- conditionals do not need both their "then" and "else" operands evaluated.
- We can short-circuit the evaluation of boolean operators like "and" and "or".

Selective evaluation is important in imperative languages too. Consider the following codes in pascal and C:

```
FUNCTION find (L: ^ListNode; X: Data): ^ListNode;
BEGIN
  WHILE (L<>NULL) and (L^.data<>X) DO
    L := L^.next;
  find := L;
END;
```

```
ListNode* find (ListNode* L, Data  X)
{
  while ((L != NULL) && (L->data != X))
    L = L->next;
  return L;
}
```

The Pascal version crashes when X is not in the list, because `AND` does not use selective (short-circuit) evaluation.

2.3.4 Left-To-Right

The simplest way to handle multiple operands is to evaluate them left-to-right.

`f(g(x), h(y))`:

1. evaluate `g(x)`
2. evaluate `h(y)`
3. evaluate `f(...)`

2.3.5 Nondeterministic

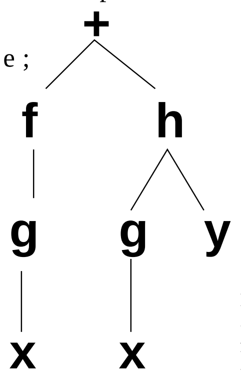
In a "pure" functional language, there is no reason why, in `f(g(x), h(y))`, we should worry whether `g(x)` is evaluated before `h(y)`.

- But if g or h has side-effects, a different matter.
  e.g., `(x + ++x)`

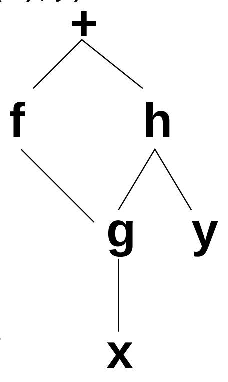
- Therefore some imperative languages specify left-to-right evaluation.
- In practice, this disallows many common compiler optimizations, so many languages (e.g., C++) deliberately do not specify expression ordering.

In functional languages, relaxing the left-to-right ordering permits capture of common subexpressions:

An expression like `f(g(x)) + h(g(x),y)` which has AST:



may instead be compiled into something like:



- Standard ML uses innermost evaluation with selective evaluation of conditionals.
- But some languages do use outermost evaluation.

2.4 Lexical Scope

Functions would quickly become unwieldy without a facility for naming recurring subexpressions.

`let` ⟨*binding*⟩ `in` ⟨*exp*⟩ `end`

binds names in ⟨*pattern*⟩ that can be used within ⟨*exp*⟩.

The scope of these bindings extends from the `in` to the closing `end`.

There are two kinds of bindings:

1. fun bindings
2. val bindings

2.4.1 fun bindings

```
let fun foo x = x+1
in foo(1)+foo(2) end
```

Bindings like this can be used to

- create "local" functions that are not of general interest
  - especially one-shot functions
- create functions whose behavior varies in different calls.

We've previously defined

```
fun transform(L,f) =
  if null(L)
    then []
    else f(hd(L))::transform(tl(L),f);
```

What's the quickest way to write a function to add a given value to each member of a list?

2.4.2 val bindings

`let val pi = 3.14159 in pi*r*r end`
establishes a binding of a value (`3.14159`) to a name (`pi`) that begins at the `in` and extends to the matching `end`.

- Note that in a recursive function, value bound may be different for each activation:

  ```
  fun len(L) = if null(L) then 1 else
      let x = len(tl(L)) in x+1 end;
  ```

- but, once established, the binding never changes within the same scope and activation.

Pattern Matching

Val bindings like
`let val pi = 3.14159 in pi*r*r end`
are a special case of the more general form:

`let val` ⟨*pattern*⟩ = ⟨*exp*⟩ `in` ⟨*exp*⟩ `end`

where a pattern is an expression containing one or more variables to be bound.

- There are limits on legal patterns, the most important of which is that a variable can occur only once in a pattern.

For example, we introduced tuples and records as types, but did not give any mechanism for accessing their components. This is done easily via pattern matching:

```
type Person =
  {name:string,
  address:string,
  id:int};

fun name(p:Person) =
  let (name=n, address=a, id=z) = p
  in n end;
```

Pattern matching is a pervasive part of the SML style. For example, the code

```
fun transform(L,f) =
  if null(L)
    then []
    else f(hd(L))::transform(tl(L),f);
```

pretty much marks us as SML tyros.

A better form would be:

```
fun transform(L,f) =
  if null(L)
    then []
    else let val a::rest = L in
           f(a)::transform(rest,f)
         end;
```

The pattern `a::rest` is used to decompose the list into its first element and the list of its remaining elements.

Patterns in Function Declaration

Patterns are also used to split functions by cases. The code

```
fun len(x) =
  if null(x) then 0 else 1+len(tl(x));
```

would more typically be written as

```
fun len([]) = 0
  | len(a::rest) = 1+len(rest);
```

Sometimes, we may want to use a pattern where we don't intend to use all the matched portions. _ is an "anonymous" variable for use in such patterns.

Instead of:

```
fun len([]) = 0
  | len(a::rest) = 1+len(rest);
```

we would typically write

```
fun len([]) = 0
  | len(_::rest) = 1+len(rest);
```

There is also a shorthand for matching unwanted record fields:

```
fun name(p:Person) =
  let val {name=n, ...} = p in n end;
```

2.5 SML Style

Let's consider the problem of writing quicksort in SML.

Key elements of SML programming style:

- recursion
- constructive manipulation of data
- pattern matching
- use of functions as 1st class objects
- polymorphism
- **higher order functions**, functions that operate upon other fucntions

2.5.1 Searching a List

A recursive hunt for a given element is easy enough:

```
fun findit (x, []) = ???
  | findit (x, y::rest) = ???
```

```
fun findit (x, []) = ???
  | findit (x, y::rest) = ???
```

What should the return type of `findit` be?

```
fun findit (x, []) = false
  | findit (x, y::rest) =
      if (x = y) then true
      else ???
```

another base case

```
fun findit (x, []) = false
  | findit (x, y::rest) =
      if (x = y) then true
      else findit (x, rest);
```

the inductive case

Now, sometimes we want to search for items that satisfy a certain condition, rather than exact matches.

- Replace `x` by a predicate (boolean function) indicating if a given value from the list is what we want.

Example:

```
fun isSmall (x) = (x < 5);

findif(isSmall, [8, 4, 2, 12]);
findif(isSmall, [8, 10, 12]);
```

We modify our find function to take another function as a parameter:

```
fun findif (predicate, []) = false
  | findif (predicate, y::rest) =
      if ??? then true
      else findif (predicate, rest);
```

We modify our find function to take another function as a parameter:

```
fun findif (predicate, []) = false
  | findif (predicate, y::rest) =
      if predicate(y) then true
      else findif (predicate, rest);
```

2.5.2 Selecting from a list

Let's look at a related problem: extracting from a list all items that satisfy some condition.

For example,

```
select(isSmall, [8, 2, 12, 4 10]);
```

should return the list:

```
[2, 4]
```

```
fun select (predicate, []) = ???
  | select (predicate, y::rest) =
      if predicate(y) then ???
      else ???
```

```
fun select (predicate, []) = []
  | select (predicate, y::rest) =
      if predicate(y) then ???
      else ???
```

```
fun select (predicate, []) = []
  | select (predicate, y::rest) =
      if predicate(y) then ???
      else select(predicate, rest);
```

```
fun select (predicate, []) = []
  | select (predicate, y::rest) =
      if predicate(y)
      then y::select(predicate, rest)
      else select(predicate, rest);
```

Note the constructive approach to building the result list.

Functions are 1st Class

To refine this further, let's allow some flexibility in what it means to be "small".

We could do

```
fun isSmall5(x) = (x < 5);
fun isSmall3(x) = (x < 3);
fun isSmall10(x) = (x < 10);
```

and use any of these with `select`.

But can we generalize this?

Yes, if we realize that functions can be maniuplated like data:

```
fun isLessThan(x, y) = (x < y);
fun isSmall5(x) = isLessThan(x, 5);
fun isSmall3(x) = isLessThan(x, 3);
fun isSmall10(x) = isLessThan(x, 10);
```

In fact, we don't need to clutter our namespace with all the "isSmall" variants:

```
fun isLessThan(x, y) = (x < y);

let fun predicate(x) = isLessThan(x,3) in
  select(predicate, [5, 1, 0, 4]);
```

SML programmers often build useful functions by having another function generate the one they want:

```
fun isSmallgenerator(value) =
  let fun isSmall(x) = x < value
  in isSmall
  end;
```

Note that each call to `isSmallGenerator` returns a "customized" function

This lets us write things like:

```
select (isSmallGenerator(2), [5, 1, 2, 4]);
val is3 = isSmallGenerator(3);
select (is3, [5, 1, 2, 4]);
```

2.6 Type Construction

SML has features for constructing new types. We've already seen

- type expressions (e.g., `int*string->bool`)
- **type abbreviation**, which gives a convenient name to a type expression:

```
type Person =
  {name:string,
  address:string,
  id:int};
```

SML also has a powerful type constructor similar to tagged variant records, the `datatype` binding.

Here is a simple example:

`datatype color = Red | Blue | Green;`

This looks like an enumerated type, and can be used as such, but it's actually more powerful.

The constants `Red`, `Blue`, and `Green` are called **constructors** of the new `color` type.

- Each introduces a distinct variant of the `color` type.

The general form of a datatype binding is

$$\langle \textit{dtbinding} \rangle ::= \texttt{datatype}\ \langle \textit{id} \rangle = \langle \textit{variant} \rangle\ [\{, \langle \textit{variant} \rangle\}]$$

$$\langle \textit{variant} \rangle ::= \langle \textit{id} \rangle\ [\texttt{of}\ \langle \textit{typeExpr} \rangle]$$

```
datatype btree =
    empty
  | leaf of int
  | node of int*btree*btree;
```

```
datatype btree =
    empty
  | leaf of int
  | node of int*btree*btree;

val tree1 =
  node(50,
       node(25,
            leaf(15),
          leaf(30)),
     node(62,
          leaf(35),
          leaf(99))
    );
```

It's also possible to do polymorphic datatypes:

```
datatype 'a btree =
   empty
 | leaf of 'a
 | node of 'a*btree*btree;

val tree1 =
  node(50,
     node(25,
          leaf(15),
          leaf(30)),
     node(62,
          leaf(35),
          leaf(99))
    );

val tree2 =
  node(baker,
     leaf(adams),
     leaf(zeil)
    );
```

```
fun treesize (empty)   = 0
  | treesize (leaf(_)) = 1
  | treesize(node(_, L, R)) =
     1 + treesize(L) + treesize(R);

fun traverse (empty)   = []
 |  traverse(leaf(x)) = [x]
 |  traverse(node(x, L, R)) =
      traverse(L) @ x::traverse(R);
```