Patterns The Essence of Functional Programming

Up to now we have defined functions in a very traditional way:
function name + variable name parameters

Read Chap 7

In functional programming we can exploit the structure of objects during a function definition by using patterns and pattern matching.

Example: no pattern matching, factorial

```
- fun fact(x) = if x = 0 then 1 else x*fact(x-1);
```

$$x! \begin{cases} 1 \text{ if } x = 0 \\ x*(x-1)! \text{ otherwise} \end{cases}$$

Example: with pattern matching, factorial

```
- fun fact 0 = 1
    | fact n = n * fact(n-1);
```

Very simple pattern: either it is 0 or not.

Patterns

In order to use patterns we need to extend our ML syntax for function definitions:

<fun-def> ::= **fun** <fun-bodies>
<fun-bodies> ::= <fun-body>
                          | <fun-body> **|** <fun-bodies>
<fun-body> ::= <fun-name> <pattern> **=** <expression>
<pattern> ::= any function and operator free expression
                          (constructors are allowed).

Valid Patterns:
1
(a,b)
[2,3]
q::rest

Invalid Patterns:
1+a
f(q)

Patterns

Example: Pattern matching on lists. Write a function sumlist that accepts a list of integer values and returns the sum of the integers on the list.

```
 - fun sumlist ([ ]) = 0
      | sumlist(x :: xs) = x + sumlist(xs);
```

Patterns

Example: write a function that reverses a given list.

```
- fun reverse ([ ]) = [ ]
      | reverse (x :: xs) = reverse(xs) @ [x];
```

Patterns

Example: match on nested structures. Assume we have a list of persons

```
[(32,185,”married”,”pilot”),(28,160,”not-married”,”cook”),...]
```

we want to write a function that returns the age of the first person on the list.

```
- fun get1stAge ((age,weight,mstat,profession)::otherpersons) = age;
```

here we pattern match on the list as well as on the tuples that make up the list

```
- fun get1stAge (L) = #1 hd(L);
```

same function no pattern matching

Note: here we assume that the list of persons is never empty!

Anonymous Variables

Consider the following program:

```
- fun f (0) = “zero”
      | f (x) = “non-zero”;
```

The variable x is never used on the right side of the equation; bad programming practice.

We can rewrite this program using an <u>anonymous variable:</u>

```
- fun f (0) = “zero”
      | f ( _ ) = “non-zero”;
```

Here we pattern match on the structure but we don’ t exactly care what the precise values are.

Patterns

Pattern matches can also occur in other places in functional programs.

Consider,

```
- val (age,weight,mstat,profession) = (38,185,”married”,”pilot”);
```

pattern!

```
val age = 38 : int
val weight = 185 : int
val mstat = "married" : string
val profession = "pilot" : string
```

This is different from

```
- val joe = (38,185,”married”,”pilot”);
val joe = (38,185,”married”,”pilot”) : int * int * string * string
```

Local Definitions: ‘Let’ Stmt

The aim is to limit the scope of a definition.

Syntax:

<let-expr> ::= **let** <definitions> **in** <expr>
<definitions> ::= any valid variable or function definition
<expr> ::= any valid expression

Note: the value of <expr> is the return value of <let-expr>.

Pattern Matching with Let Stmt

Example: Given a list of elements, write a function that returns two lists,, each with half the elements of the original list.

```
- fun halve ([ ]) = ([ ], [ ])
     | halve ([a]) = ([a], [])
     | halve (a::b::rest) =
         let
            val (x,y) = halve(rest)
         in
            (a::x,b::y)
         end;
```

x and y are local variables.

Merge Sort

- The **`halve`** function divides a list into two nearly-equal parts
- This is the first step in a merge sort
- For practice, we will look at the rest

Example: Merge

```
fun merge ([], ys) = ys
|   merge (xs, []) = xs
|   merge (x::xs, y::ys) =
      if (x < y) then x :: merge(xs, y::ys)
      else y :: merge(x::xs, ys);
```

- Merges two sorted lists
- Note: default type for '<' is **`int`**

Example: Merge Sort

```
fun mergeSort [] = []
|   mergeSort [a] = [a]
|   mergeSort theList =
      let
        val (x,y) = halve theList
      in
        merge(mergeSort x, mergeSort y)
      end;
```

- Merge sort of a list
- Type is **`int list -> int list`**, because of type already found for **`merge`**

Merge Sort At Work

```
- fun mergeSort [] = []
= |   mergeSort [a] = [a]
= |   mergeSort theList =
=       let
=         val (x, y) = halve theList
=       in
=         merge(mergeSort x, mergeSort y)
=       end;
val mergeSort = fn : int list -> int list
- mergeSort [4,3,2,1];
val it = [1,2,3,4] : int list
- mergeSort [4,2,3,1,5,3,6];
val it = [1,2,3,3,4,5,6] : int list
```

Nested Function Definitions

- You can define local functions, just like local variables, using a **`let`**
- You should do it for helper functions that you don't think will be useful by themselves
- We can hide **`halve`** and **`merge`** from the rest of the program this way
- Another potential advantage: inner function can refer to variables from outer one (as we will see in Chapter 12)

Merge Sort

```
fun mergeSort [] = []
|   mergeSort [e] = [e]
|   mergeSort theList =
      let
        fun halve [] = ([], [])
        |   halve [a] = ([a], [])
        |   halve (a::b::cs) =
              let
                val (x, y) = halve cs
              in
                (a::x, b::y)
              end;

        fun merge ([], ys) = ys
        |   merge (xs, []) = xs
        |   merge (x::xs, y::ys) =
              if (x < y) then x :: merge(xs, y::ys)
              else y :: merge(x::xs, ys);

        val (x, y) = halve theList
      in
        merge(mergeSort x, mergeSort y)
      end;
```

Exercise

Write the function less(e,L) that returns a list of integers from the list L each of which is less than the value e.

Homework

Assignment #6 – see website – use pattern matching!

Midterm coming up end of October