Nested patterns

- We can nest patterns as deep as we want
  - Just like we can nest expressions as deep as we want
  - Often avoids hard-to-read, wordy nested case expressions
- So the full meaning of pattern-matching is to compare a pattern against a value for the "same shape" and bind variables to the "right parts"
  - More precise recursive definition coming after examples

### Useful example: zip/unzip 3 lists

```sml
fun zip3 lists = 
case lists of
  ([],[],[]) => []
| (hd1::tl1,hd2::tl2,hd3::tl3) => (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
| _ => raise ListLengthMismatch

fun unzip3 triples = 
case triples of
  [] => ([],[],[])
| (a,b,c)::tl =>
  let val (l1, l2, l3) = unzip3 tl
  in
    (a::l1,b::l2,c::l3)
  end

More examples in .sml files
```

### Style

- Nested patterns can lead to very elegant, concise code
  - Avoid nested case expressions if nested patterns are simpler and avoid unnecessary branches or let-expressions
    - Example: `unzip3` and `nondecreasing`
  - A common idiom is matching against a tuple of datatypes to compare them
    - Examples: `zip3` and `multsign`
- Wildcards are good style: use them instead of variables when you do not need the data
  - Examples: `len` and `multsign`

### Examples

- Pattern `a::b::c::d` matches all lists with >= 3 elements
- Pattern `a::b::c::[]` matches all lists with 3 elements
- Pattern `((a,b),(c,d))::e` matches all non-empty lists of pairs of pairs

(Most of) the full definition

The semantics for pattern-matching takes a pattern `p` and a value `v` and decides (1) does it match and (2) if so, what variable bindings are introduced.

Since patterns can nest, the definition is elegantly recursive, with a separate rule for each kind of pattern. Some of the rules:

- If `p` is a variable `x`, the match succeeds and `x` is bound to `v`
- If `p` is `_`, the match succeeds and no bindings are introduced
- If `p` is `(p1,..,pn)` and `v` is `(v1,..,vn)`, the match succeeds if and only if `p1` matches `v1`,.., `pn` matches `vn`. The bindings are the union of all bindings from the submatches
- If `p` is `C(p1)`, the match succeeds if `v` is `C(v1)` (i.e., the same constructor) and `p1` matches `v1`. The bindings are the bindings from the submatch.
- ...(there are several other similar forms of patterns)
Exceptions

An exception binding introduces a new kind of exception

```ml
exception MyUndesirableCondition
exception MyOtherException of int * int
```

The `raise` primitive raises (a.k.a. throws) an exception

```ml
raise MyUndesirableException
raise (MyOtherException (7,9))
```

A handle expression can handle (a.k.a. catch) an exception

- If it doesn’t match, exception continues to propagate

```ml
e1 handle MyUndesirableException => e2
e1 handle MyOtherException(x,y) => e2
```

Actually…

Exceptions are a lot like datatype constructors...

- Declaring an exception adds a constructor for type `exn`
- Can pass values of `exn` anywhere (e.g., function arguments)
  - Not too common to do this but can be useful
- `handle` can have multiple branches with patterns for type `exn`

Recursion

Should now be comfortable with recursion:

- No harder than using a loop (whatever that is 😆)
- Often much easier than a loop
  - When processing a tree (e.g., evaluate an arithmetic expression)
  - Examples like appending lists
  - Avoids mutation even for local variables
- Now:
  - How to reason about efficiency of recursion
  - The importance of tail recursion
  - Using an accumulator to achieve tail recursion
  - [No new language features here]

Call-stacks

While a program runs, there is a call stack of function calls that have started but not yet returned

- Calling a function `f` pushes an instance of `f` on the stack
- When a call to `f` finishes, it is popped from the stack

These stack-frames store information like the value of local variables and “what is left to do” in the function

Due to recursion, multiple stack-frames may be calls to the same function

Example

```ml
fun fact n = if n=0 then 1 else n*fact(n-1)
val x = fact 3
```

```ml
fact3:3*_
fact2:2*_fact2:2*_fact1:1*_fact0: 1
```

Example Revised

```ml
fun fact n = 
  let fun aux(n,acc) = 
    if n=0 
    then acc 
    else aux(n-1,acc*n) 
  in 
    aux(n,1) 
  end 
val x = fact 3
```

Still recursive, more complicated, but the result of recursive calls is the result for the caller (no remaining multiplication)
The call-stacks

```
fact 3
aux(3,1) aux(3,1): aux(3,1):
aux(2,3) aux(2,3): aux(2,3):
aux(1,6) aux(1,6): aux(1,6): aux(1,6):
aux(0,6) aux(0,6): aux(0,6):6
Etc...
```

An optimization

It is unnecessary to keep around a stack-frame just so it can get a callee’s result and return it without any further evaluation.

ML recognizes these tail calls in the compiler and treats them differently:
- Pop the caller before the call, allowing callee to reuse the same stack space
- (Along with other optimizations,) as efficient as a loop

Reasonable to assume all functional-language implementations do tail-call optimization.

What really happens

```
fun fact n = 
  let fun aux(n,acc) = 
    if n=0 
    then acc 
    else aux(n-1,acc*n) 
  in 
    aux(n,1) 
  end 
val x = fact 3
```

Moral of tail recursion

- Where reasonably elegant, feasible, and important, rewriting functions to be tail-recursive can be much more efficient
  - Tail-recursive: recursive calls are tail-calls
- There is a methodology that can often guide this transformation:
  - Create a helper function that takes an accumulator
  - Old base case becomes initial accumulator
  - New base case becomes final accumulator

Methodology already seen

```
fun fact n = 
  let fun aux(n,acc) = 
    if n=0 
    then acc 
    else aux(n-1,acc*n) 
  in 
    aux(n,1) 
  end 
val x = fact 3
```

Another example

```
fun sum xs = 
  case xs of 
    [] => 0 
    | x::xs' => x + sum xs'
```

```
fun sum xs = 
  let fun aux(xs,acc) = 
    case xs of 
      [] => acc 
      | x::xs' => aux(xs',x+acc) 
    in 
      aux(xs,0) 
  end
```
And another

```haskell
fun rev xs =  
  case xs of  
    [] => []  
  | x::xs' => (rev xs') @ [x]
```

```haskell
fun rev xs =  
  let fun aux(xs,acc) =  
    case xs of  
      [] => acc  
    | x::xs' => aux(xs',x::acc)  
  in  
  aux(xs,[])  
end
```

Actually much better

```haskell
fun rev xs =  
  case xs of  
    [] => []  
  | x::xs' => (rev xs') @ [x]
```

- For fact and sum, tail-recursion is faster but both ways linear time
- Non-tail recursive rev is quadratic because each recursive call uses append, which must traverse the first list
  - And 1+2+…+(length-1) is almost length*length/2
  - Moral: beware list-append, especially within outer recursion
- Cons constant-time (and fast), so accumulator version much better

Always tail-recursive?

There are certainly cases where recursive functions cannot be evaluated in a constant amount of space

Most obvious examples are functions that process trees

In these cases, the natural recursive approach is the way to go
  - You could get one recursive call to be a tail call, but rarely worth the complication

Also beware the wrath of premature optimization
  - Favor clear, concise code
  - But do use less space if inputs may be large

What is a tail-call?

The "nothing left for caller to do" intuition usually suffices
  - If the result of f x is the "immediate result" for the enclosing function body, then f x is a tail call

But we can define "tail position" recursively
  - Then a "tail call" is a function call in "tail position"

Precise definition

A tail call is a function call in tail position

- If an expression is not in tail position, then no subexpressions are
  - In fun f p = e, the body e is in tail position
  - If if e1 then e2 else e3 is in tail position, then e2 and e3 are in tail position (but e1 is not). (Similar for case-expressions)
  - If let b1 ... bn in e end is in tail position, then e is in tail position (but no binding expressions are)
  - Function-call arguments e1 e2 are not in tail position
  - ...