Today, we are going to talk about two of the most important ideas in functional programming: *structural recursion* and *structural induction*.

# 1 Let-binding and pattern matching

## 1.1 Let

`let` is useful not just for taking apart pairs, but in general for naming intermediate results. For example:

```ocaml
fun prism (w : int, l : int, h : int) : int * int =
  let val area : int = w * l
  val volume : int = area * h
  in
    (area, volume)
  end
```

Typing is just like `val` declarations at the top-level: the expression must have the type the variable is annotated with. So is scoping: the variables are in scope in later declarations, and in the *body*, the expression between `in` and `end`. That is, `let <decls> in <expr> end` is well-typed if the `<decls>` are, and if `<expr>` is well-typed using the variables bound in the declarations. The variables are *not* in scope outside the body of the `let` (we say `let` bindings are *lexically scoped*). This allows you to introduce local bindings.

Evaluation: To evaluate a `let`, you first evaluate the `<decls>`, substituting into subsequent declarations and `<expr>` as you go. Then, when you’re done with all of the `<decls>`, you evaluate the body `<expr>`, and its value is the value of the whole `let`.

Because you substitute the result of evaluation, you can use `let` to avoid duplicating the same work twice. E.g you only add 2+3 once in the following example, even though `x` gets mentioned twice:

```ocaml
val foo = let val x : int = 2 + 3
           val y : int = x + 1
         in x + y
         end
```

*based on materials by Brandon Bohrer, Dilsun Kaynar, Mike Erdmann and others*
1.2 Patterns and pattern matching

Recall this odd-looking syntax:

```plaintext
fun f (0 : int) : int = ...
| f (n : int) : int = ...
```

The intuition behind what’s going on here is that SML looks one by one at the different cases and sees if the argument “looks like” 0. If not, it goes on to the next case and sees if the argument “looks like” n (which will always succeed since any integer value can be bound to a variable).

In reality, 0 : int and n : int are instances of a more general construct called a pattern. A pattern can be:

- A constant (e.g. 0)
- A variable (e.g. n)
- A tuple of patterns ((p1, ..., pn))
- ... and a couple other things we’ll talk about later.

Any pattern can also be annotated with a type (e.g. n : int).

A pattern can be said to match a value:

- Constants match themselves (e.g. the pattern 0 matches only the value 0).
- Variables match any value of the correct type and bind that value to the variable.
- A tuple pattern (p1, ..., pn) matches a tuple (v1, ..., vn) and recursively matches p1 against v1, and so on. We used this to bind tuple values in lab.

Often when programming with patterns, we provide a list of patterns along with things to do in each case, separated by pipes | (like in the definition of f above). Pattern matching starts with the first pattern, attempts to match the value against it, and then proceeds to the next case if the match fails. Note that the order matters. Suppose we had swapped the cases above.

```plaintext
fun f (n : int) : int = ...
| f (0 : int) : int = ...
```

Then everything would match the n case and the 0 case would never be checked. Since this is basically never what you want, SML/NJ will actually raise a compile error if you write this.

If the last pattern in the list fails to match, an exception Match or Bind (depending on where we’re doing the pattern match) will be raised. SML/NJ can actually check if this situation is possible (i.e. if your patterns aren’t exhaustive) and will raise a compiler warning.

1.3 Using pattern matching

There are several places you can use patterns.
Val bindings (inside or outside of let). Here, we don’t get to give several cases: we just give one, along with the expression whose value the pattern should be matched against. If the pattern doesn’t match, it raises Bind. e.g.

\[
\text{val } (x : \text{int}, y : \text{int}) = (42, 15150)
\]

\[
\text{let val } (x : \text{int}, y : \text{int}) = (42, 15150)\text{ in } x + y \text{ end}
\]

Function definitions. This is how we’ve already been using pattern matching. The cases are checked when the function is applied, and the first matching case (if any) is evaluated. If no case matches, it raises Match.

Case expressions. SML provides a standalone pattern matching construct:

\[
\text{case } e \text{ of } p_1 => e_1 \mid \ldots \mid p_n => e_n
\]

Typing: if the type of \(e\) matches the type expected by all of the patterns and all of the expressions \(e_1, \ldots, e_n\) have type \(t\), then the case expression has type \(t\).

Evaluation: Case evaluates the expression to a value and then matches the value against each pattern in order. The value of the case expression is the value of the first matching subexpression (with any substitutions).

For example, we could rewrite the above function as

\[
\text{fun } f (n : \text{int}) : \text{int} =
\text{case } n \text{ of }
\mid 0 => \ldots
\mid k => \ldots
\]

It’s generally preferable to use the more direct style without the case if you just have to do one pattern match on the argument, but sometimes you need to use nested case expressions:

\[
\text{fun } c (1 : \text{int}) : \text{int} = 1
\mid c (n : \text{int}) : \text{int} =
\text{case isEven } n \text{ of }
\mid \text{true } => c (n \text{ div } 2)
\mid \text{false } => c (n \times 3 + 1))
\]

2 Review: defining functions

Recall the factorial function from lab:
2.1 Factorial

(* fact : int -> int
* REQUIRES: n >= 0
* ENSURES: fact n ==> n! = n * (n-1) * (n-2) * ... * 1,
*)
fun fact (0 : int) : int = 1
| fact (n : int) : int = n * (fact (n - 1))

Note that it follows the same pattern as double: you give a case for zero, and a case for 1 + n, in which you make a recursive call on n − 1.

For evaluation, we can calculate as follows:

fact 2
|-> case 2 of 0 => 1 | _ => 2 * (fact (2 - 1))
|-> 2 * (fact (2 - 1))
|-> 2 * (fact 1)
|-> 2 * (case 1 of 0 => 1 | _ => 1 * (fact (1 - 1)))
|-> 2 * (1 * (fact (1 - 1)))
|-> 2 * (1 * (fact 0))
|-> 2 * (1 * (case 0 of 0 => 1 | _ => 0 * (fact (0 - 1))))
|-> 2 * (1 * 1)
|-> 2 * 1
|-> 2

2.2 Tests

Testing is checking that a program works by trying it on some examples. There are three ways you can write tests.

The first is just to open up the REPL and play around:

- fact 3;
  val it = 6 : int
- fact 5;
  val it = 120 : int

This is great for quickly convincing yourself a function works, but has two problems:

1. It’s good to save your tests for later, so that when you change your code, you can run them again. This is called regression testing.

2. For this class, we also want you to hand in your tests.

For both reasons, it’s good to write the tests in your .sml file.

One way to do this is to remember that integer constants are patterns and can be used in a val binding:

val 120 = fact 5
This is actually using an old feature—val bindings—in a new way. You can think of this as naming the result of \texttt{fact 5} as 120. Since this “name” is a value, it can only name one thing. If the expression evaluates to the same value, then everything’s okay; otherwise an exception is raised. Specifically, you will see

\begin{verbatim}
uncaught exception Bind [nonexhaustive binding failure]
   raised at: fact.sml:36.5-36.17
\end{verbatim}

This means that the constant in the \texttt{val} binding did not match the actual value, and gives you the line number of the failed test.

A downside of this style is that none of the definitions in the file are loaded into the REPL unless all of the tests pass, which means you can’t use this style to debug a function that has failing tests in the REPL. Thus, we suggest that you use the REPL directly while you are developing, and then use \texttt{val 120 = fact 5} for regression testing after the fact.

### 2.3 Five-Step Methodology

In writing this function, we followed an important methodology, which we will ask you to follow whenever you write a function:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a \texttt{REQUIRES} clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an \texttt{ENSURES} clause what the function computes (what it returns).
4. Implement the function.
5. Provide test tcases, generally in the format
   \begin{verbatim}
   val <return value> = <function> <argument value>
   \end{verbatim}

### 3 Recursion

Let’s take a closer look at what’s going on in the recursive factorial function you wrote in lab. You’ve probably seen recursion before, but this will be a slightly different take on it than you’ve seen before, which will set us up well for next lecture when we will see other kinds of recursion with which you’re probably less familiar. The main idea is that recursive functions come from recursive data.

For example, we can define the familiar natural numbers recursively (or inductively, if you prefer) as

\begin{verbatim}
A natural number is either
  0, or
  1 + n, where n is a natural number,
and that’s it!
\end{verbatim}
Note that this definition is self-referential. It is what is called an *inductive definition* of the natural numbers. Using this definition, we know that 0, 1 + 0, 1 + (1 + 0),... are natural numbers. Moreover, since every natural number follows one of these rules, if we implement a function that handles these two cases, then it handles every possible natural number.

Thinking about this avoids two ways in which a recursive function can go wrong. What if we forgot to subtract 1 \((n - 1)\) in the argument to the recursive call? Then \texttt{fact} calculates like this:

\[
\text{fact 2} \\
|\rightarrow \text{case 2 of } 0 \Rightarrow 0 \mid _\Rightarrow 2 * (\text{fact 2}) \\
|\rightarrow 2 * (\text{fact 2}) \\
|\rightarrow 2 * (\text{case 2 of } 0 \Rightarrow 0 \mid _\Rightarrow 2 * (\text{fact 2})) \\
|\rightarrow 2 * (2 * (\text{fact 2})) \\
|\rightarrow 2 * (2 * (\text{case 2 of } 0 \Rightarrow 0 \mid _\Rightarrow 2 * (\text{fact 2}))) \\
|\rightarrow 2 * (2 * (2 * (\text{fact 2}))) \\
\ldots
\]

This is an infinite loop! This is an error, and we can see where we went wrong: from the definition of a natural number we know that we met the invariants for the function—\(n\) started out as a natural number so it still is one, therefore it’s okay call \texttt{fact} with it—but it isn’t a smaller natural number. The definition defined the next natural number by making an existing natural number bigger, so our code doesn’t reflect the definition.

We if we left off the base case? Then \texttt{double} calculates like this:

\[
\text{fact 1} \\
|\rightarrow \text{case 1 of } _\Rightarrow 1 * (\text{fact (1-1)}) \\
|\rightarrow 1 * (\text{fact (1-1)}) \\
|\rightarrow 1 * (\text{fact 0}) \\
|\rightarrow 1 * (\text{case 0 of } _\Rightarrow 0 * (\text{fact (0-1)})) \\
|\rightarrow 1 * (0 * (\text{fact (~1)})) \\
\ldots
\]

This is also an infinite loop, because we ignored the other part of the definition.

To a first approximation, we can represent natural numbers as \texttt{int}s, but this is two kinds of a lie:

- There are too many \texttt{int}s (negative numbers like \texttt{-3})
- There are also too few (The largest \texttt{int} is \(2^{30} - 1\), but there are natural numbers bigger than that\(^1\)).

But let’s pretend and use \texttt{int} now; we’ll show you how to do better later in the course, when we talk about datatypes. At the moment, the syntax is a lot more light-weight if we use this slightly bad representation.

The inductive definition of natural numbers means that to define a function whose argument is a natural number, it suffices to (1) give a case for zero, and (2) give a case for any other number \(n\) in terms of the result on its predecessor \(n - 1\).

In the abstract, this looks like:

\[1\text{[citation needed]}\]
fun f (0 : int) : T = ...  
| f (n : int) : T = ... f (n - 1) ...

Note that, when we’re working with types like int that have primitive operations on them, you might not want to write all of your functions recursively (see, e.g. isEven from lab), but we will occasionally pretend that we’re working with a real type of natural numbers for which the only operation is performing structural recursion \(^2\).

4 Structural Induction and Equivalence

The next thing to cover is proving theorems about your code by induction. This is a really important skill: first, it lets you formally justify the correctness of your code. Second, it draws out the kind of reasoning you have to do when you’re writing any code, whether you do the proofs explicitly or not.

Let’s briefly review how to prove things by induction on the natural numbers.

4.1 Template for Structural Induction on Natural Numbers

Here’s the format that any proof by induction on the natural numbers should have:

- Induction is applicable when the statement to be proved has the form “for all natural numbers \(n\), [some predicate] holds of \(n\)”.
- The proof should fill in the following skeleton:

  \[
  \text{Proof. The proof is by induction on } n. \\
  \text{Base Case: } n = 0 \\
  \text{To show: [substitute 0 into the predicate]} \\
  \text{Proof: ...} \\
  \text{Inductive Case: } n = 1 + k \\
  \text{Inductive hypothesis: [substitute } k \text{ into the predicate].} \\
  \text{To show: [substitute } 1 + k \text{ into the predicate].} \\
  \text{Proof: ... Be sure to cite when you use the inductive hypothesis. ...} \\
  \]

Note the similarities with the template for structural recursion on the natural numbers!

In order to prove things about code, we also need to talk more about equivalence.

4.2 Equivalence

Recall that, when we wrote the code for fact, we wrote REQUIRES and ENSURES clauses specifying that it only takes natural numbers, and that on \(n\), it evaluates to \(n!\). Since these are the criteria we expect to hold of a correct implementation of fact, this is exactly how we want to state our correctness theorem!

**Theorem 1.** For all natural numbers \(n\), \(\text{fact } n \cong n!\).

\(^2\)and * to implement factorial, if you look carefully, but ssshhhh... On Homework 2, we’ll make you pretend you don’t even have that.
This theorem says that the ML program \texttt{fact} \( n \) equals the numeral \( n! \). Here the \( n \) is a mathematical variable, which stands for any numeral, not an ML variable. We will allow ourselves to use math variables like \( n \), and expressions like \( n! \), in code as names for values. For example, for each numeral \( n \), \( n! \) stands for the numeral \( n \times n - 1 \times \ldots \times 1 \).

But what does \( \cong \) really mean? \textit{When are two programs equal?} Do they need to have the exact same source code? What if they have very different source code but always do the same thing, like \texttt{sum} and \texttt{sumtree} from the first lecture?

This is somewhat subtle, because programs can go into an infinite loop, or raise an exception. We will define a notion of \textit{extensional equivalence} that accounts for these possibilities. To make it clear that we’re doing something kind of subtle, we’ll use \( \equiv \) (or \( \cong \) in ASCII) to notate equivalence.

Here’s the basic idea: two programs are equivalent iff

- They both evaluate to the same value
- They both raise the same exception
- They both infinite loop

We will refine our definition of equivalence over time, but here are some rules that will let us get started:

- Equivalence is an \textit{equivalence relation}. That is to say:
  - it is reflexive: \( e \equiv e \)
  - it is symmetric: \( e_1 \equiv e_2 \) if \( e_2 \equiv e_1 \)
  - it is transitive: \( e_1 \equiv e_3 \) if \( e_1 \equiv e_2 \) and \( e_2 \equiv e_3 \)

- Equivalence is a \textit{congruence}: if two expressions are equal, then you can substitute one for the other inside any bigger program. You will also sometimes hear the \textit{congruence} rule referred to as \textit{referential transparency}. These are both just fancy ways of saying that you can always replace equal things with other equal things.

- Equivalence contains evaluation: if \( e \mapsto e' \) then \( e \equiv e' \).

### 4.3 Example: Correctness of \texttt{fact}

\textbf{Theorem 2.} For all natural numbers \( n \), \texttt{fact} \( n \equiv n! \).

There are two main ingredients in the proof:

- Doing calculations in ML and in math
- Appealing to the \textit{inductive hypothesis}: when proving a theorem about natural numbers, we get to assume the theorem is true for \( k \) while proving it for \( 1 + k \).

We assume that \(*\) correctly implements \( \times \), etc.
Proof. The proof is by induction on \( n \), using the predicate \( P(x) := \text{fact } x \cong x! \).

**Case for \( 0 \).**
To show: \( \text{fact } 0 \cong 0! \).
Proof:

\[
\begin{align*}
\text{fact } 0 & \cong 1 \quad \text{step} \\
& \cong 0! \quad \text{math}
\end{align*}
\]

On the right, we write a justification for each equivalence. *step* means “these two expressions are equivalent because one steps to the other.” *math* means “I’m using some basic mathematical fact” (we’ll tell you what you’re allowed to use). In this case the definition of factorial says that \( 0! \) is 1.

**Case for \( 1 + k \).**
Inductive hypothesis: \( \text{fact } k \cong k! \)
To show: \( \text{fact } (1 + k) \cong (1 + k)! \).
Proof:

\[
\begin{align*}
\text{fact}(1 + k) & \cong (1 + k) * (\text{fact } ((1 + k) - 1)) \quad \text{step} \\
& \cong (1 + k) * \text{fact } k \quad \text{math} \\
& \cong (1 + k) * k! \quad \text{IH, cong.} \\
& \cong (1+k)! \quad \text{math}
\end{align*}
\]

In the second step, we take the non-zero branch—this correct since \( 1 + k \neq 0 \), and \( k \) is a value. In the second-to-last step, we use the IH to replace \( \text{fact } k \) with \( k! \) inside the bigger expression \( n * \text{fact } k \), exploiting the congruence property of equivalence. In the final step, we use a mathematical fact about the factorial function.

Note that the structure of the proof mirrors the structure of the code and the structure of the definition we gave for natural numbers: in the code, we have a zero case and an \( n + 1 \) case; same in the proof. In the code, we have a recursive call on \( n - 1 \); in the proof, we have an inductive hypothesis for \( n - 1 \).

### 5 Additional Recursion Patterns

Sometimes, the natural way to write a function is not to recur on \( n - 1 \). Another way to write a function on the natural numbers is to give (1) a case for 0, (2) a case for 1, and (3) a case for \( n + 2 \) in terms of a recursive call on \( n \). This recursion principle can be formally justified using induction, but it should be intuitively clear: every natural number is either 0, or it’s 1 or it’s 2 + \( n \), where \( n \) is a natural number. We can use this to define \texttt{isEvenSlow}.

```ocaml
fun isEvenSlow (0 : int) : bool = true
  | isEvenSlow (1 : int) : bool = false
  | isEvenSlow (n : int) : bool = isEvenSlow (n - 2)
```

This function is written mirroring a different, but equivalent definition of the natural numbers.
Specifically, we could have defined
A natural number is either
0, or
1, or
2 + n, where n is a natural number,

and that’s it!

and ended up with a provably identical structure of natural numbers.

It’s not immediately obvious that this really is defined over the naturals that we know and love, but it’s not hard to show that the definitions are the same. We would use this same structure for the inductive proof that, say, \( \text{isEvenSlow} \cong \text{isEven} \).

What if we want to prove the following program correct?

```haskell
fun power (_ : int, 0 : int) : int = 1
| power (n : int, k : int) : int =
  if isEven k then (power (n, k div 2)) * (power (n, k div 2))
  else n * power (n, k - 1)
```

Here, depending on the input, we might make a recursive call on \( k - 1 \) or \( k/2 \). Because we’re doing a recursive call using division, we can no longer pretend that we’re using nice natural numbers with a clean inductive definition. But this function is still perfectly well-defined on integers and we might want to say something about it. In this case, we could use strong induction:

Formula. The proof is by strong induction on \( n \).

**Base Case:** \( n = 0 \)
To show: [substitute 0 into the predicate]
Proof: …

**Inductive Case:** \( n > 0 \)
Inductive hypothesis: for all \( 0 \leq k < n \), [substitute \( k \) into the predicate].
To show: [substitute \( n \) into the predicate].
Proof: … Be sure to cite when you use the inductive hypothesis. …

5.1 Non-examples

Consider the function \( c \) from earlier in the lecture. It looks like it only ever returns 1, so we might wish to prove that \( c \ n \cong 1 \) for all natural numbers \( n \). However, this may not be true, since \( c \) could fail to terminate (in fact, whether the function \( c \) always terminates is the subject of an open problem known as the Collatz Conjecture).

We already proved that \( \text{fact} \) always terminates (with the value \( n! \)) and we already showed that recursive calls with arguments like \( n-2 \) are tenable using strong induction. So why can’t we prove the above equivalence and solve an open problem? It’s the other recursive call that creates the issue: \( 3n + 1 \) is a larger natural number than \( n \) and so there would be no appropriate inductive hypothesis to use to show this case.