Induction and Recursion

Proof-Driven Functional Programming

15-150 M21

Lecture 0602
02 June 2021

- ✓ Basics of Functional Computation
- Induction and Recursion
- Polymorphism & Higher-Order Functions
- Functional Control Flow
- The SML Modules System
- Applications & Connections

0 Patterns

To write sophisticated code, we often need to *split into cases*.
This can be achieved with `if` expressions,

```
fun exp(n:int):int =
  if n=0 then 1 else 2*exp(n-1)
```

But, for more complex stuff, `if`s get clunky:

```
fun quadrant (m:int,n:int):string =
  if m=0 orelse n=0
  then "boundary"
  else if m>0
       then if n>0
            then "I"
            else "IV"
       else if n<0
            then "II"
            else "III"
```

General idea

SML provides a system of **pattern matching**, which allows us to split into cases in a elegant way.

We can pattern match against the two constructors of the `bool` type: `true` and `false`.

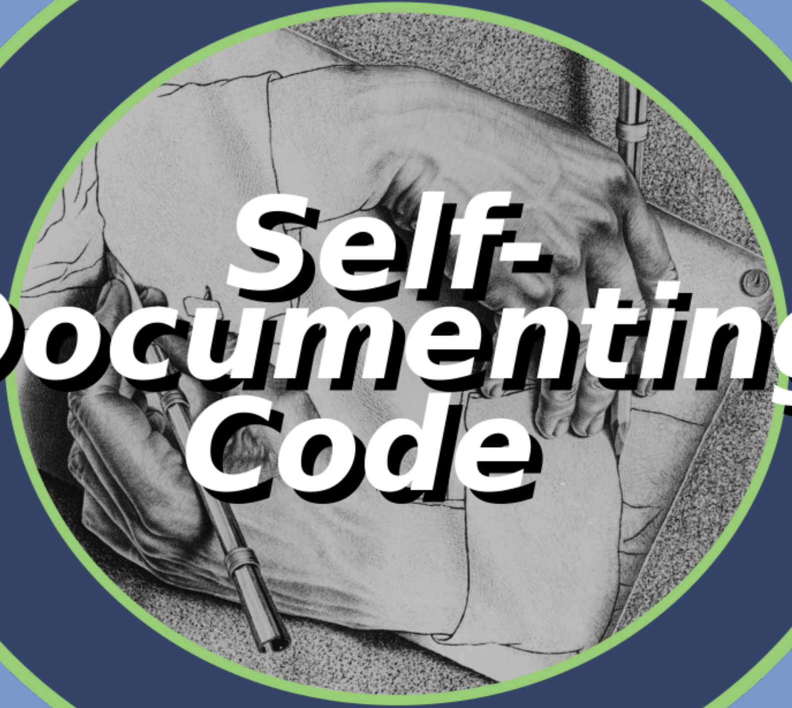
0602.0 (patterns.sml)

```
val toBit : bool -> int =
  fn true => 1 | false => 0
```

0602.1 (patterns.sml)

```
val not : bool -> bool =
  fn true => false | false => true
```

Self-Documenting Code



The if-then-else structure we've been using so far

```
if b then e1 else e2
```

is just syntactic sugar for

```
(fn true => e1 | false => e2) b
```

Check Your Understanding Verify that the typing & evaluation rules for the above expressions are indeed identical.

Can pattern match against values of type `string` (each string literal is a constructor of type `string`).

0602.2 (patterns.sml)

```
fun paren ("":string):string = ""
  | paren " " = ""
  | paren s = "(" ^ s ^ ")"
```

0602.3 (patterns.sml)

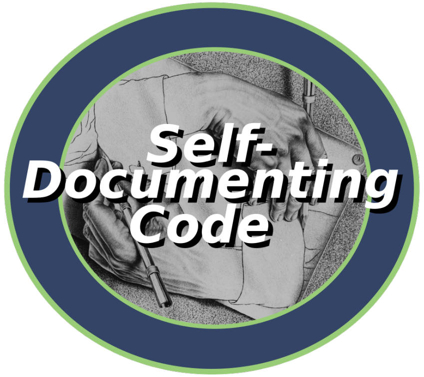
```
fun isLambda "lambda" = true
  | isLambda _ = false
```

Wildcard ignores input (use to indicate the input is irrelevant).

Likewise with int

0602.4 (patterns.sml)

```
val isZeroOrOne : int -> bool =
   fn 0 => true | 1 => true | _ => false
```



If we REQUIRE `n>=0` and are defining a function by recursion on the natural numbers, we can have zero and successor cases

0602.5 (patterns.sml)

```
(* REQUIRES: n>=0 *)
fun exp (0:int):int = 1
  | exp n = 2 * exp(n-1)
```

Old exp traces are clunky

```
exp 4
⟹ if 4=0 then 1 else 2*exp(4-1)
⟹ 2*exp(3)
⟹ 2*(if 3=0 then 1 else 2*exp(3-1))
⟹ 2*(2*exp(2))
⟹ 2*(2*(if 2=0then 1 else 2*exp(2-1)))
⟹ 2*(2*(2*exp(1)))
⟹ 2*(2*(2*(if 1=0 then 1 else 2*exp
  (1-1))))
⟹ 2*(2*(2*(2*exp(0))))
                    0 then 1 else 2*exp
```

Much better...

```
exp 4
⟹ 2 * exp(3)
⟹ 2 * 2 * exp(2)
⟹ 2 * 2 * 2 * exp(1)
⟹ 2 * 2 * 2 * 2 * exp(0)
⟹ 2 * 2 * 2 * 2 * 1
⟹ 16
```

- Constructors

```
fn true => e1 | false => e2
```

- Variable names

```
fn (x:int) => x
```

- Wildcards

```
fn (_ : string) => 2
```

- Tuples of patterns

```
val P : int * int = ...
val (a,b) = P
```

Check Your Understanding Determine the behavior of this function

```
fun foo ((0,0),_) = "a"
  | foo ((_,0),(7,_)) = "b"
  | foo ( _, (8,8)) = "c"
  | foo _ = "d"
```

- Function applications

```
(* Doesn't work *)
val m+n = 2
val (s1 ^ s2) = "hello world"
```

- Non-match-able types

```
(* Doesn't work *)
val (fn x => e) : int -> string = f
val 2.0 = 2.0
```

- Repetitive patterns

```
(* Doesn't work *)
fun equal (m:int,m:int) = true
  | equal     = false
```

A very common expression is to apply a pattern-matching function to some expression:

```
(fn (true,_)        => e1
  | (false,true)    => e2
  | _               => e3
) (x < 7, x < 15)
```

SML provides a nicer syntax for this:

```
case (x < 7, x < 15) of
  (true,_)       => e1
| (false,true)   => e2
| _              => e3
```

The “flase” bug


```
case b of
  flase => 2
| true => 1
```

SML can tell at compile-time whether the cases you've written are exhaustive or not.

0602.6 (patterns.sml)

```
fun purple 4 = true
  | purple ~117 = false
```

It'll warn you, but allow the computation to proceed.
The exception `Match` is raised when none of the clauses match the given expression.

Defn. A value `f : t1 -> t2` is said to be **total** if, for all values `v : t1`, the expression `f(v)` is *valuable*.

Examples:

- `op+`
- `Int.toString`

Non-examples:

- `exp`
- `div`
- `purple`

If the clauses of a function are non-exhaustive, then that function cannot be total.

1 $\mathbb{N}$ Recursion/Induction

0602.7 (nat.sml)

```
(*  exp : int -> int
 *  REQUIRES: n>=0
 *  ENSURES: exp(n) == 2^n
 *)
fun exp (0:int):int = 1
  | exp n = 2 * exp(n-1)

val 1 = exp 0
val 131072 = exp 17
```

Code	**Proof**
Cases/clauses	Cases

```
fun not true = false
  | not false = true
```

Prop. `not : bool -> bool` is total.

Proof. Want to show: `not v` valuable for all values $\mathtt{v} : \mathtt{bool}$.

- $\mathtt{v} = \mathtt{true}$

$$\mathtt{not\ true} \Longrightarrow \mathtt{false} \qquad \text{(First clause of } \mathtt{not}\text{)}$$

- $\mathtt{v} = \mathtt{false}$

$$\mathtt{not\ false} \Longrightarrow \mathtt{true} \qquad \text{(Second clause of } \mathtt{not}\text{)}$$

Check Your Understanding

Prove `paren` or `isLambda` total

It's simple...

We want to prove facts about the behavior of `exp(n)` for all nonnegative integers `n`, i.e. for all natural numbers `n`. How do we prove something about all natural numbers? **Induction**!

Principle The principle of **weak** or **simple** induction says that if

- $P(0)$ holds
- For each $n \in \mathbb{N}$, $P(n)$ implies $P(n+1)$

then $P(n)$ holds for all $n \in \mathbb{N}$.

0602.7 (nat.sml)

```
fun exp (0:int):int = 1
  | exp n = 2 * exp(n-1)
```

Code	**Proof**
Cases/clauses	Cases
Recursion	Induction
Simple recursion (`n` calls `n-1`)	Weak Induction (assume n, prove $n+1$)

Prop. For all values `n : int` with $n \geq 0$, `exp(n)` is valuable.

Proof by weak induction on `n`.

BC: `n=0`.

$$\texttt{exp 0} \implies \texttt{1.} \qquad \text{(first clause, } \texttt{exp}\text{)}$$

IH: Suppose for some `n>=0`, `exp(n)` is valuable.

WTS: `exp(n+1)` is valuable.

$$\begin{aligned} \texttt{exp(n+1)} &\implies \texttt{2 * exp(n)} && \text{(second clause, } \texttt{exp}\text{)} \\ &\implies \texttt{2 * v} && \text{(for some value } \texttt{v}\text{, by IH)} \\ &\implies \texttt{v'} && \text{(for some value } \texttt{v'}\text{, by totality of } \texttt{op*}\text{)} \end{aligned}$$

Check Your Understanding

Prove:

Prop. For all values `n : int` $n \geq 0$,

$$\texttt{exp(n)} \quad \cong \quad 2^{\texttt{n}}$$

(using the mathematical facts that $2^0 = 1$ and $2^{n+1} = 2 \cdot 2^n$ for all $n \in \mathbb{N}$)

Key Skill:

Implementing the spec

The mathematical notion of the *factorial* can be given as:

$$0! = 1$$
$$(n+1)! = (n+1) \cdot n!$$

```
fact : int -> int
```
REQUIRES: `n>=0`
ENSURES: `fact(n)` $\cong$ `n!`

0602.8 (nat.sml)

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

val 1 = fact 0
val 720 = fact 6
```

Today's slogan:

Assume smaller, make bigger (and don't forget the base case)

Prop. For all values `n : int` with $n \geq 0$, `fact(n)` $\cong$ `n`!.

Proof by weak induction on `n`.

BC: `n=0`.

$$\texttt{fact 0} \cong 1 \qquad \text{(Defn. of } \texttt{fact}\text{)}$$
$$= 0! \qquad \text{(math)}$$

IH: Suppose for some `n>=0`, `fact(n)` $\cong$ `n`!.

WTS: `fact(n+1)` $\cong$ `(n+1)`!.

$$\texttt{fact(n+1)} \cong \texttt{(n+1) * fact(n)} \qquad \text{(Defn. of } \texttt{fact}\text{)}$$
$$\cong \texttt{(n+1) * n}! \qquad \text{IH}$$
$$= \texttt{(n+1)}! \qquad \text{(math)}$$

Code	**Proof**
Cases/clauses	Cases
Recursion	Induction
Simple induction (`n` calls `n-1`)	Weak Induction
Recursive call	Inductive hypothesis

Any inductive proof you turn in for 150 should include:

- A statement of what kind of induction you're using
- A statement of which variable you're inducting on
- Explicit code stepping/equivalences
- Citations for each step (even math)
- An explicit inductive hypothesis

5-minute break

2 Strong Induction

If n is even, then

$$2^n = \left(2^{n/2}\right)^2$$

If n is odd, then

$$2^n = 2 \cdot \left(2^{\lfloor n/2 \rfloor}\right)^2$$

0602.9 (pow.sml)

```
fun square (x:int):int = x * x
```

0602.10 (pow.sml)

```
fun even (x:int):bool = (x mod 2)=0
```

Lemma 1 `square` is total

Lemma 2 `even` is total

Lemma 3 `square(x)` $\cong$ $\mathtt{x}^2$ for all `x : int`

Lemma 4 `even(n)` $\cong$ `true` iff `n` is even

Code	**Proof**
Cases/clauses	Cases
Recursion	Induction
Simple recursion	Weak Induction
Recursive call	Inductive hypothesis
Helper function	Lemma

0602.11 (pow.sml)

```
fun pow (0:int):int = 1
  | pow n =
      case (even n) of
        true => square(pow(n div 2))
      | false => 2 * square(pow(n div 2))
```

Thm. For all values `n : int` where `n>=0`,

$$\texttt{exp(n)} \quad \cong \quad \texttt{pow(n)}.$$

Proof next time

Code	Proof
Cases/clauses	Cases
Recursion	Induction
Simple recursion	Weak Induction
Recursive call	Inductive hypothesis
Helper function	Lemma
Non-simple recursion	Strong Induction

3 An Intro to Lists

Base types:

- `int`
- `bool`
- `string`
- `real`
- `char`

Type constructions

- `*`
- `->`

- For each type `t`, there is a type

  `t list`

  of *lists of elements of* `t`
- There are two constructors of type `t list`:
  - `[] : t list`
  - If `x : t` and `xs : t list`, then

    `(x::xs) : t list`
- The values of type `t list` are lists `[x1, x2, ..., xn]`, including `[]`. This is just syntactic sugar for `[]` and `::`, however:
  - `[1] : int list` is `1::[]`
  - `["functions","are","values"] : string list` is just `"functions"::"are"::"values"::[]`

0602.12 (lists.sml)

```
val null : string list -> bool =
  fn [] => true | _ => false
```

```
len : int list -> int
```
REQUIRES: true

ENSURES: `len L` evaluates to the length of `L`

0602.13 (lists.sml)

```
fun len ([] : int list):int = 0
  | len (x::xs) = 1 + len xs

val 5 = len [1,2,3,4,5]
val 2 = len [~5000,19]
val 0 = len []
```

```
(op @) : int list * int list -> int list
```

REQUIRES: true

ENSURES: If `L1` is a list of length m and `L2` is a lsit of length n, then `L1@L2` evaluates to a list of length $m + n$ whose first m elements are the elements of `L1` (in the same order they appear in `L1`) and whose last n elements are the elements of `L2` (in the same order they appear in `L2`)

0602.14 (lists.sml)

```
infix @
fun ([]:int list) @ L = L
  | (x::xs) @ (L:int list) =
        x::(xs @ L)
```

- Pattern-matching facilitates concise, elegant function declarations
- Well-written functional code corresponds to its own correctness proof
- Most interesting functions are recursive, and have inductive correctness proofs
- Lists are data structures in SML defined by the `[]` and `::` constructors

Next Time

- `pow` proof
- More about lists
- Tail recursion
- Recurrences & sequential runtime analysis

Thank you!