#flo #inclass #intersession

---

1 | Functional! SML!

^ = append "hello world" is hello world in SML. ## but let's get to the functions define functions like in mathematical notation: `fn x = x*3`

functions are valid variables. inp func leads to oup func, but u can call like `(func)(param)` `fn 0 => true | x => false`

absolute value func: `fn x => if x` > 0 then x else ~1*x= (remember, negative sign is ~) or, we can say `val abs = {above func}`

we can also have tuple inputs, like `fn (x, y) => x`. all functions have one input type and one output type

this is, a statically typed language!

renaming variables is shadowing! the old items still use the old value.

or 0 function

```
val f = fn (0, x) => true | (x, 0) => true | _ => false;
f(1, 1);
```

we can allow recursive with val r factorial function:

```
val rec fact = fn 1 => 1 | x => x * fact(x-1);
fact(1);
```

it's written in LATEX?? > average exacting spec in LaTeX fan vs average UB enjoyer - @david

proven to be entirely type safe

1.1 | types!

all of our declarations are: - expression - val x = e - val rec x = e

but now we get… - type t = types - datatype t =..

```
datatype intoption =
    SOME of int
| NONE

(*SOME 3 : intoption val double : *)

fn SOME x => x;
| NONE => 0;
```

pattern matching to extract data

omg..

1.1.1 | **recursive data types**

```
datatype ilist =
    Empty
| Front of int*ilist;
```

`of` is the name of the constructor from the data that it carries

product of a list

```
val rec prod = fn Empty => 1
    | Front (x, xs) => x * prod(xs)
```

1.1.2 | **lists! the builtin kind**

```
nil x::xs
(*  list of type of x, [x, ...xs] *)
(* or we can just use *) [1, 2, 3]
```

1.2 | **curryin'**

```
In mathematics and computer science, currying is the **technique of converting a function that takes mul
```

but really, it let's us define functions with other functions instead of tuples, we have **iterated function calling**

```
fn (x,y) => f(x, y)
```

```
val uncurry = fn f => fn (x, y) => f (x,y)
(* of type (int -> (int -> int)) -> (int * int) -> int) *)
val curry = fn f => fn x => fn y => f (x, y)
```

val rec filter : ('a -> bool) -> 'a list -> 'a list = fn

```
(* direct pattern matching!*)
case x of
0 => "zero"
| 1 => "one"
| _ => "more than one"
```

if statements are really

```
if a then b else c
(fn true => b | false => c ) a
```

`()` is the unit: the zero size of tuples

one thing in the type unit, as opposed to bool which has 2. this means we already know what it will be! void really returns `()`, the unit

1.3 | **Types!**

what is a type?

first, **groups**. - a group has: - elements : types - an operator : composition? forming tuples - identity : unit - what about inverse…? and closed? - assosiative KBxGroupAndMatricesIntro mathematics/linear$_{\text{algebra}}$/index

- two groups are equivalent
- KBrefIsomorphicVectorSpace we can have isomorphic types!
- prove with pair of invertible functions
- **magnitude**: the size of the unit type
- note: * is composed, and x is ’times.’

type	size
unit	1
bool	2
a * b	
a	x
b	
unit * a	
a	

1.3.1 | **Rings**

like groups, but extra. two operations instead of one? we also need assosiativity

note: ’a means alpha option = ’a, unit

$$\text{'a + option} \quad = 1 + \quad a \quad = \quad \text{'a} \quad + \quad 1 \quad ?$$

the + op forms a datatype? + works on datatypes

adding in + to `x` means we get a **semi-ring**

we cannot have something of (type, value)? `void` `void option` ~= to `()` note: ~= is isomorphic

```
fn NONE => 0
fn () => NONE
```

$|a \to b|$ is exponentiation, $|b|^{|a|}$

1.3.2 | **the weirder stuff**

two things are isomprhic if you create a pair of functions that are inverses

for example, `curry` and `uncurry`! forms an isomorphism between two types we have proves that $(a^b)^c = a^{(b \cdot c)}$ with the functions current and uncurry?? ????

this is a type theoretical proof.

> ”proofs and programs are actully one and the same” ”types and propositions are also one and the same”

the definition of a **list**: $L(\alpha) = 1 + \alpha \times L(\alpha)$ $L(\alpha) = \alpha \times L(\alpha) = 1$ $L(\alpha) = \frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 \dots$

wait wait wait WHAT? we just got the TAYLOR SERIES?? KBhMATH401TaylorSeriesApprox

$\frac{d}{da}L(a) = \frac{d}{da}\frac{1}{1-a}$

we can do DERIVATIVES? KBhMATH401Derivatives the derivative of a list is.. two lists? a pair of lists?

1.3.3 | the structure of data

we can delete things. a^2 , a data structure that holds two `a=s, we have two ways to delete (rm either =a`) same is true for $a + a$

for a^3, we have three things to choose from

the question is, what is left behind after something get's punched out? $a + a - a$ needs to maintain the same structure so a^2 punched out `a` gives us $a +$ bool so we know where the rmed `a` was. so really, we have `a*option, a*option`

wait, hole punching is a derivative?

$a^2 => 2a$ essentially, just put a hole where the thing was removed

hence why, if we punch something out a linked list, we get two lists! everything prior to the hole, and everything after the hole. and, since we can get the element at the beginning and end of a list, we can get the things around the hole instantly

we can move the hole, which means we can see in a vicinity and traverse, like a directory structure!

we can define a tree as $T(a)^2 \times L(a \times 2 \times T(a))$. this fully defines a tree!

taking the second derivative means, we get two holes instead of one what does this mean? i don't know!

done with type algebra, time for ## lambda calculus *istfg.* not actual calculus. it just means, ways for operations to interact

church goes, how about, no, to turing. instead, let's use functions!

three constructs: - functions: fn x => exp - exp are exp(exp), x, fn - $\lambda x \cdot exp$

$\lambda x \cdot x$ is a simplex example function?

1. booleans true is $\lambda x \cdot \lambda y \cdot x$ false is $\lambda x \cdot \lambda y \cdot y$

   if a then b else c = abc which means, true bc => b and false bc => c

2. numbers the encoding of 1 =

   $\bar{0} = \lambda f.\lambda x.x$
   $\bar{1} = \lambda f.\lambda x.fx$
   $\bar{2} = \lambda f.\lambda x.f(fx)$

   now we define, incrementing! and, other stuff

   succ $= \lambda n.\lambda f.\lambda x.f(n(fx))$ add $= \lambda a.\lambda b.\lambda f.\lambda x.bf(afx)$ mult $= \lambda a.\lambda b.\lambda f.\lambda x.a(bf)x)$

   this is really, programming language theory. past function programming

   the Y combinator actually comes from lambda calculus Y $= \lambda f.\ (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$

1.4 | abstractions

opaue types: types that we don't know the definition of we only know the functions that can interact w/ them

exit functions can return void, because they exit anyways! we don't care. -> technicaly, they are polymorphic

nvm, actully we are doing

1.5 | finish up sml!

```
(* what we have been doing *)
val rec name = fn pat => exp
(*instead, we can do *)
fun name pat = exp
(*eg.*)
fun fact 0 = 1
    | fact n = n * fact(n-1)
(* this is pattern mathching *)
```

can define using `deftype` and `type`

we can represent using expression trees

```
fact 4 ->
    / * \
    4   / app \(the apply function)
      fact     n-1
```

and then we do this recursively until we get to our final result this is evaluation! **memory** is linearly consumed with this unraveling instead, if we pass down an accumulator, we can make this better

recursive call (ie. fact 3 4 -> fact (3-1) (4 * 3)) -> result -> fills hole from above in accumulator then you pass the final value all the way up

this actully **doesnt help**, but we can add an optimization which makes it help

1.5.1 | tailcall optimization

> Tail-call optimization is where you are able to avoid allocating a new stack frame for a function because the calling function will simply return the value that it gets from the called function. -stackoverflow (or not, ig, cus tailcall means we don't have it)

```
    / app \
   /   |   \
fact   3    1

->

    / app \
   /   |   \
fact   -    .
      / \  / \
      3 1  3  4
```

this doesnt take up any extra space, because we dont need the previous steps!

```
fun fact 0 k = k1
    | fact n k =
        fact (n-1) (fn r => k (r * n))
```

this means that k acts as out accumulator and, this has no backtracking, as it is fully tailcall-optimized

this is dont by the compiler automatically to everything, which is why we don't need a stack

1.5.2 | an example

cps func find1 : ('a -> bool) -> 'a shrub -> ('a -> 'z) -> (unit -> 'z') -> 'z

```
fun find1 p T s f = case T of (* s = succes *) (* also, this automatically currys it! *)
    Leaf x => if p x then s x else f () (*if we are in a tree with only one thing, then return succes x
    | Branch(l,r) => find1 p l s (fn () =>
                    find1 p r s f)
```

this is almost, imperative! once you are done being called, you are a whole. the stack -> heap of clojures we get, one active line then heap memory

when everything is tailcalls, that means we can get rid of functions and instead just have **continuations** once you call something, you do not exists. `oh_no()`

1.5.3 | continuations

we can define `letcc l => k` then, when we define k, we can quit out?

we can jump through time, with the old values? what??

let's "call for help" whenever we see a neg number

```
fn sumpos k nil = 0
    | sum pos k (x::xs) =
        if x >= 0
            then x + sumpos k xs
        else (letcc j => then k (j, +)+sumpos
```

this let's us call k for help whenever we want we can invoke an entirely different CONTROL FLOW, and then that returns the right thing in the right type!

the call stack is still in mem, but when j becomes inaccessible, we don't need need it anymore and it gets garbage collected!

proofs are programs and from programs you can extract a