Exercise

State the type of the following expression (without using your computer!)

```
fun f(a:int) = 2;
val t = (true,f,f,1);
```

Solution

```
bool * (int -> int) * (int -> int) * int
```

Exercise

Write a function that returns the value 2 for arguments 0, 1, and 2, and returns 3 for all other (integer) arguments. Write two solutions (a) using pattern matching (b) using if-then-else

```
fun example(0)=2
   | example (1)=2
   | example (2)=2
   | example (_)=3;

fun example(x) = if (x>2) then 3 else if (x<0) then 3 else 2;
```

Exercise

Write a function that returns -1 for negative numbers, 0 for 0, and +1 for positive arguments. Can you use pattern matching?

Solution

```
fun example(x) = if (x=0) then 0 else if (x<0) then ~1 else 1;
```

Exercise

Write a function that returns the smaller of its two arguments (a) using an argument pattern, (b) using a local declaration

Solution

```
fun min(a,b) = if (a<b) then a else b;

fun min(t:int*int) = let val (a,b)=t in if (a<b) then a else b end;
```

Exercise

Write a funtion that calculates x^9, with as few (explicit) uses of multiplication as possible

Solution

```
fun power(x,n) = if (n=0) then 1 else power(x,n-1)*x;

power(8,9);
```

Exercise

Calculae $n * z$ without using multiplication, where n is a positive integer and z is an integer

Solution

```
fun mul(n,z) = if (n=1) then z else z+mul(n-1,z);
```

Towers of Hanoi: Recursive algorithm in C

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
void move disk(const char *from , const char *to)
{
  printf (Move disk from %s a %s\n , from , to );
}

void move(unsigned int n, const char *from, const char *to, const char *via)
{
  if (n == 1) {
  move disk(from , to );
  return ;
}
move(n-1, from, via, to);
move disk(from , to );
move(n-1, via, to, from);
}
```

Continued

```
int main(int argc, char *argv[])
{
  unsigned int height = 0;
  if (argc > 1) {
  height = atoi(argv[1]);
}
if (height <= 0) {
  height = 8;
}
move(height , Left, Right, Center);
return 0 ;
}
```

Alternative version

```
void move(unsigned int n, const char *from, const char *to, const char via)
{
   if (n == 0) { return ;
   }
move(n-1, from, via, to);
move disk(from , to );
move(n-1, via, to, from);
}
```

Towers of Hanoi

- Not functional
  - `move()` and `move_disk()` do not have results
  - `printf` has side effects
- Purely functional
  - Add results to these functions
  - Instead of printing, build up (`concat`) a string with a description of the moves

Functional version

```
const char *concat(const char *a, const char *b)
{
  char *res;
  res = malloc(strlen(a) + strlen(b) + 1);
  memcpy(res , a, strlen(a));
  memcpy(res + strlen(a), b, strlen(b));
  res[strlen(a) + strlen(b)] = 0;
  return res ;
}
const char *move disk(const char *from , const char *to)
{
  return concat(concat(concat(concat("Move disk from ", from), " to "),
          to), "\n");
}
const char *move(int n, const char *from, const char *to, const char *via)
{
   return (n == 1) ?
       move disk ( from , to ):
         concat(concat(move(n - 1, from, via , to),
          move disk(from, to)), move(n - 1, via, to, from));

}
```

Functional version

- Use of `concat()` reduces readability (infix operators are more readable than prefix)
- `move()` and `move_disk()` are now pure functions, without side effects
- All side effects are in `main()` that handles I/O (some side effects are needed...)
- Program allocates memory, but never frees it, so there is an implicit assumption of a garabage collection
- Some of the problems are due to the syntax of C. The next slide shows an implementation in C++, and we shall later see one in ML

Version in C++

```
#include <cstdlib>
#include <iostream>
#include <string>

std::string move disk(std::string from , std::string to)
{
 return "Move_disk_from " + from + " to " + to + "\n";
}

std::string move(int n, std::string from , std::string to , std::string via)
{
       return (n == 1) ?
          move_disk ( from , to )
          :
          move(n - 1, from, via, to) +
           move_disk(from, to) + move(n - 1, via, to, from);
}
```

Continued

```
int main(int argc, char *argv[])
{
   int height = 0;
   std::string res;
   if (argc > 1)
   {
         height = atoi(argv[1]);
   }
   if (height <= 0)
   {
    height = 8;
   }
   res = move(height , "Left", "Right", "Center");
   std::cout << res ;
   return 0 ;
}
```

History

- Work on decidability
  - Before computers, and even before Turing
  - Church, Kleene: Lambda calculus
- First functional language: John McCarthy - LISP
- Algol-60 defined by translating it to the lamba-calculus
  - Inspired by APL, imperative language with functional aspects
- Other languages: OCaml, Haskell, ML
  - Javascript, XQuery/XSLT have aspects of functional languages

Data types

- We have seen the basic types
- Complex types: Simplest technique is tuples and functions

```
val sum_squares =   fn (a,b) => a*a+b*b;
```

- Pattern matching: for tuples

```
val pair = ("greek_pi",3.14);
val (greek_pi,pi) = (pair);
```

Synonyms

- Define a synonym for a type

```
type time_t = int;
```

- Assign value

```
val a:time_t = 5;
```

- But not

```
val a:time_t = 3.4;
```

- What is the type of

```
a+3;
```

Synonyms

- Further examples

```
type integer_pair = int * int;
type first_last_name = string * string;
type real_pair = real*real;
```

- Use in function definition

```
val fr = fn(x,y):real_pair => x * y;
```

- Compare with

```
val fs = fn(x,y) => x * y;
```

Another example

- Function `c2t` takes pair `(i1,i2)` and parameter `i3` and returns triple `(i1,i2,i3)`

```
val c2t = fn (x,y) => fn z => (x,y,z);
```

- Notice the type!

```
c2t ("hello," world") "!";
```

- To restrict to a function on integers

```
val c2t = fn (x:int, y:int) => fn z:int => (x,y,z);
```

- Or:

```
val c2t = fn (x,y): integer_pair => fn z:int => (x,y,z);
c2t (2,3) 4;
```

Types

- `type` does not declare a new type, merely a synonym
- Pattern matching: Suppose we want to match only a finite number of values
- Example: Currency converion
  - Convert a pair of type `(real,string)` where the string is one of `eur`, `usd`, and `ounce_gold`
  - Result should be of the same form

Example

```
type currency = string;
type money = real * currency;

fun convert (amount, to) =
    let val toeur = fn
       (x,"eur") => x
     | (x,"usd") => x / 1.05
     | (x,"ounce_gold") => x * 1113.0
   in
     ( case to of
        "eur" => toeur amount
      | "usd" => toeur amount * 1.05
      | "ounce_gold" => toeur amount / 1113.0
      , to)
  end;


 convert ((1.2,"usd"),"eur");
```

Exhaustive list

- Values do not cover all strings
- Causes warning message
- Simple typos are not caught

```
convert ((1.2,"eur"),"usb");
```

Wildcards

- Use wildcard _ to catch exceptions
- Report error message ~1

Modified example

```
type currency = string;
type money = real * currency;

fun convert (amount, to) =
    let val toeur = fn
       (x,"eur") => x
     | (x,"usd") => x / 1.05
     | (x,"ounce_gold") => x * 1113.0
     | (_,_) => ~1.0
   in
     ( case to of
        "eur" => toeur amount
      | "usd" => toeur amount * 1.05
      | "ounce_gold" => toeur amount / 1113.0
      | _ => ~1.0
      , to)
  end;
```

Datatype

- Better to avoid runtime errors and get compilation/static checking errors instead
- Keyword `datatype` defines new types
- `currency` type: exhaustive list

```
datatype currency = eur | usd | ounce_gold;
```

- Try

```
val c = eur;
```

- The values are actually *value constructors* and could have arguments

Using type

- Define `money` datatype ("eur" etc. already used above)

```
datatype money = Eur of real | Usd of real | Ounce_gold of real;
```

- What is `Eur`? Try

```
Eur;
Eur 0.5;
```