Functional Programming

Overview

- Philosophy
- Getting Started
  - Values
  - Types and inference
  - Tuple
  - Records
- Functions
  - Parameter passing mode
  - Currying

Philosophy

- Central Dogma
  - Everything is a function
  - The value of any expression only depends on the value of its sub expressions.
- Consequences
  - Two *styles* exist
    - Purist [also known as *Fanatics*]
    - Pragmatist [also known as *Everyone else*]
  - In its pure form
    - Functional programming has *no* assignments
  - Functions
    - Are first class citizens.
  - Memory management is automatic [GC]

Getting Started

- Values
  - All values are typed.
  - Scalar types include
    - Int/Float/Boolean/String/Char
  - Every expression denotes a value.
- Example

```
Standard ML of New Jersey v110.76 [built: Mon Aug 19 10:38:12 2013]

- 42;
val it = 42 : int
 - true;
val it = true : bool
- "Hello" ;
val it = "Hello" : string
-
```

Binding

- Values can be bound to a name
  - It replaces the default name
  - The name is an *alias* for the value it denotes
  - New bindings
    - *hide* older ones.
    - Do *not* change the older one.
  - Syntax

```
val <id> = <expression>
```

  - Scope
    - Extension in space/time of the binding itself

Binding Examples

```
Standard ML of New Jersey v110.76 [built: Mon Aug 19 10:38:12 2013]
- val x = 42;
val x = 42 : int
- x;
42 : int
- val x = “Hello”;
val x = “Hello” : string
- x;
“Hello” : string
```

This is not an assignment!

A Visual on Scope

- Where the scope
  - Starts
  - Ends

```
<some code fragment>
<some code fragment>
val x = y + z * 3;
val w = x + 3;
<some code >
<some code>
```

A Visual on Scope

- Where the scope
  - Starts: After the value declaration
  - Ends: End of the program

```
<some code fragment>
<some code fragment>
val x = y + z * 3;
val w = x + 3;
<some code >
<some code>
```

Local Binding

- Local means
  - Exist for a specific duration
  - Scope is restricted: Extend until the matching end
- Syntactic form

```
let
     <bindings>
in
     <expression>
end
```

Local Binding Example

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- val x = 42;
val x = 42 : int
- x;
val it = 42 : int
- let val x = 4102
  in   x + x
  end;
val it = 8204 : int
- x;
42 : int
```

Types

- Type Information
  - Is usually computed
    - Type inference
  - Can be specified
    - Type constraint

```
Standard ML of New Jersey v110.76 [built: Mon Aug 19 10:38:12 2013]

- val x = 42 : int;
val x = 42 : int
- x + (3 : int);
val it = 45 : int
```

Types

- Type Information
  - Is used to type check expressions

```
Standard ML of New Jersey v110.76 [built: Mon Aug 19 10:38:12 2013]
- val x = 42;
val x = 42 : int
- x + 3;
val it = 45 : int
- val y = “Hello”;
val y = “Hello” : string
- x + y;
stdIn:19.1-19.4 Error: operator and operand don't agree [tycon
mismatch]
```

Type Inference

- ML **derives** types automatically
  - Type of expression based on types of sub-expressions
  - Type systems will be studied more formally later on.
- So why do we need type constraints ?

Type inference can go
horribly wrong
….
Type Inference is DEXPTIME!

[more later]

Tuples

- Used to organize information
  - Pairs
  - Triples
  - Tuples

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- (3,4);
val it = (3,4) : int * int
- (3,4,5);
val it = (3,4,5) : int * int * int
- val x = it;
val x = (3,4,5) : int * int * int
```

Tuples

- Accessing the content of a tuple
  - Projection (field access)
  - Pattern matching

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- val  x = (3,4);
val x = (3,4) : int * int
- #1(x);
val it = 3
- val (a,b) = x;
val a = 3 : int
val b = 4 : int
```

Records

- Glorified tuples!
  - Field now have names

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- val x = { name = “Donkey”, age = 3};
val x = { age=3,name=”Donkey” } : { age : int, name : string }
- #age(x)
val it = 3 : int
- val { name=a,age=b } = x;
val a = “Donkey” : string
val b =  3 : int
```

Partial Matching

- When only part of the record is relevant
  - Match what matters.
  - Ignore the rest with an ellipsis.

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- val x = { name = "Donkey", age = 3};
val x = { age=3,name="Donkey" } : { age : int, name : string }
- val { name=a, ...} = x;
val a = "Donkey" : string
```

Expressions

- What kind of expressions do we need
  - Arithmetic
  - Boolean
  - Conditional
  - Strings

Arithmetic

- The usual suspects
  - Binary
    - +,-,*,/, mod
  - Unary
    - ~ [negative]
- Work with
  - Literals, Names of the right type
  - Do not mix int and reals

Boolean

- Same old, same old
  - Conjunction
    - andalso
  - Disjunction
    - orelse
  - Negation
    - Not

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- val x = 3=1 orelse false;
val x = false : bool
```

Conditional

- One objective
  - Make a decision and branch
- A conditional is an expression
  - It has
    - A condition
    - An expression to execute if the condition is true
    - An expression to execute if the condition is false

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- val x = 4;
val x = 4 : int
- val z = if x=4 then  (“Hello”,true) else (“Bye”,false);
val z = (“Hello”,true) : string * bool
```

Strings

- Many operations are available....
- But
  - We need to first cover
    - Functions
    - Modules
- The most useful one
  - String concatenation:
    - ^

```
Standard ML of New Jersey v110.42 [FLINT v1.5], October 16, 2002
- val x = "Donkey" ^ " and Shrek";
val x = "Donkey and Shrek" : string
```

Overview

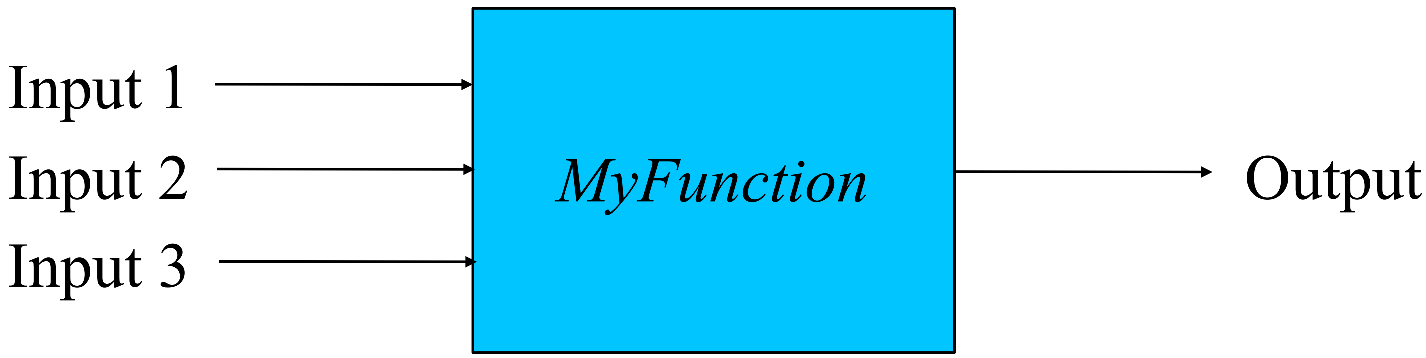
- Philosophy
- Getting Started
  - Values
  - Types and inference
  - Tuple
  - Records
- Functions
  - Parameter passing mode
  - Currying

Almost Done!

- So far, we can
  - Declare values, name (local or not)
  - Create bindings
  - Group values with
    - Tuples
    - Record
  - Type everything
    - By inference
    - By constraints
  - Compute with expressions
- What we are still missing
  - Some way to compute *interesting* stuff.

Function

- What is a function ?



- What is a good about them ?
  - Capture input-output as a black-box
  - Can be reused and composed: Lego bricks
    - Chain, stack, nest,....
  - Functions are like everything else
    - Just a value.
  - Side effect free

Function Example ?

- Is *this* a function ? [written in C]

```
int  hello(int x) {
    static int c = 0;
    int y = x * c;
    c = c + 1;
    return y;
}
```

ML Functions

- Declaration
  - Basic simplified syntax

```
fun <id> <param> = <expression>;
```

- Example

```
- fun succ x = x + 1;
val succ = fn  : int ->  int
```

- Application
  - Basic syntax

```
<id> <expression>
<id> ( <expression> )
```

- Example

```
succ 3;
val it = 4 : int
```

Parameter Passing Mode

- What options usually exist ?
  - By value
    - Example:
  - By reference
    - Example:
  - By name
    - Example:
- Which one make sense in a functional language?
  - Hint: Remember the *no side effect* rule.

Function And Value

- What happens if...

```
- fun succ x = x + 1;
val succ = fn  : int ->  int
- succ 3;
val it = 4 : int
- succ;
           ??????????????????????????????????????????
```

Function And Value

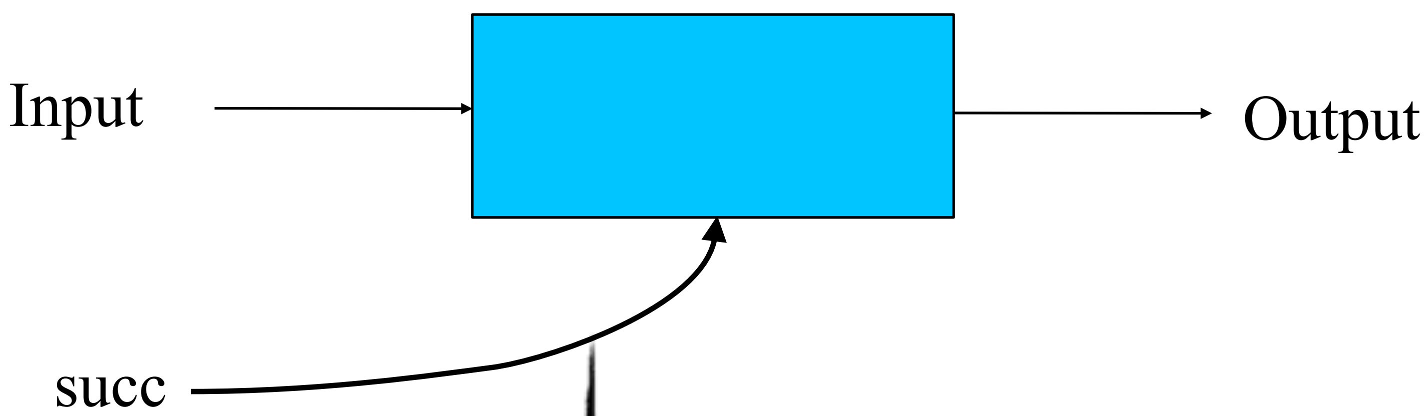
- What happens if...

```
- fun succ x = x + 1;
val succ = fn  : int ->  int
- succ 3;
val it = 4 : int
- succ;
         ?????????????????????????????????????????????????
```

```
- fun succ x = x + 1;
val succ = fn  : int ->  int
- succ 3;
val it = 4 : int
- succ;
val it = fn : int -> int
```

Function Declaration

- What actually happens
  - Two steps
    - Create a black-box to compute the function
    - Bind that black-box to a name



The binding associate a name with a value which happens to be a function

Separation of the two steps

- If a function declaration has two steps...
  - We can separate them
  - Reuse the mechanism we already have for binding
- What do we need to separate the two steps?

Anonymous Functions

- Take care of the function definition
  - Create the black box
- Syntax

```
fn  <param> => <expression>;
```

- Example

```
- fn  x => x+1;
val it = fn : int -> int
```

- Putting it all together

```
- val succ = fn  x => x+1;
val succ = fn : int -> int
```

Recursion

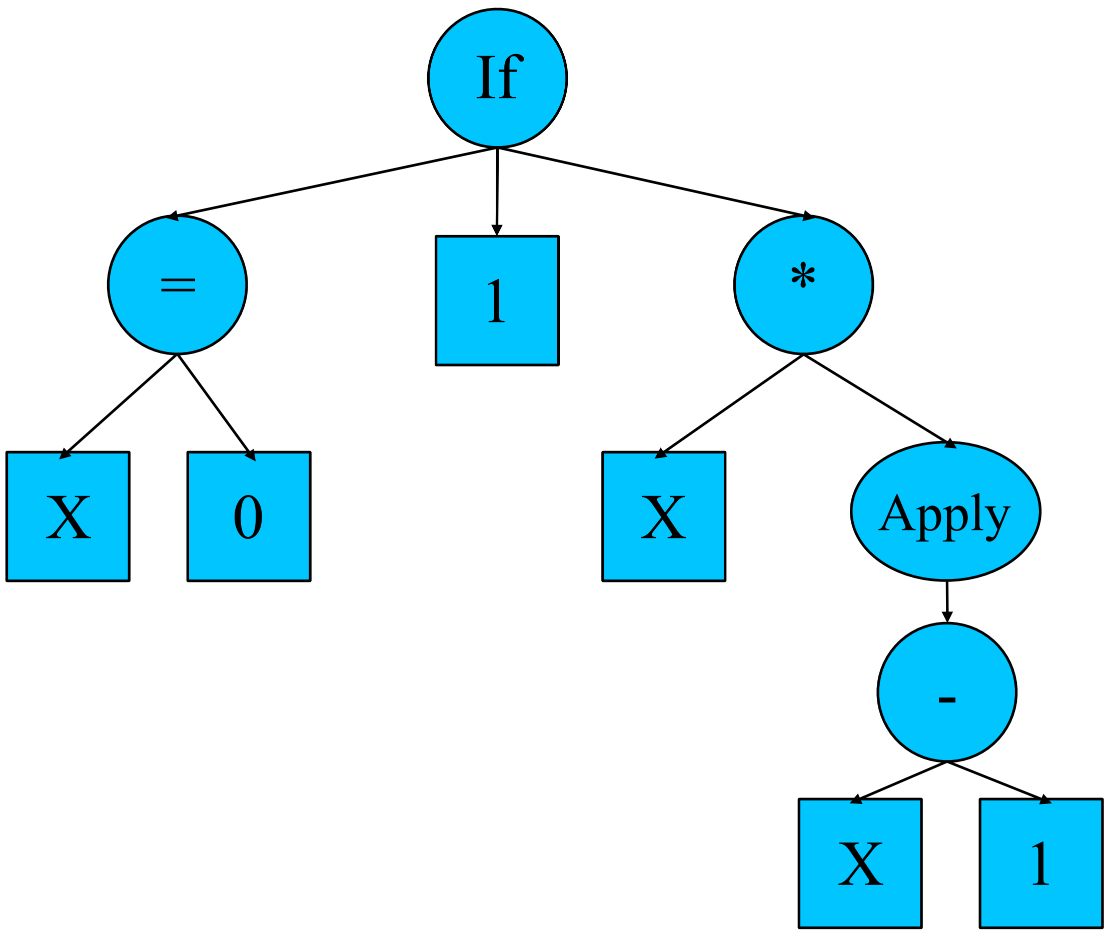
- Function can be recursive
  - Nothing special about that
- Example
  - Factorial

```
- fun fact x = if x=0 then 1 else  n * fact n-1;
val fact = fn: int -> int
- fact 5;
```

Oops...

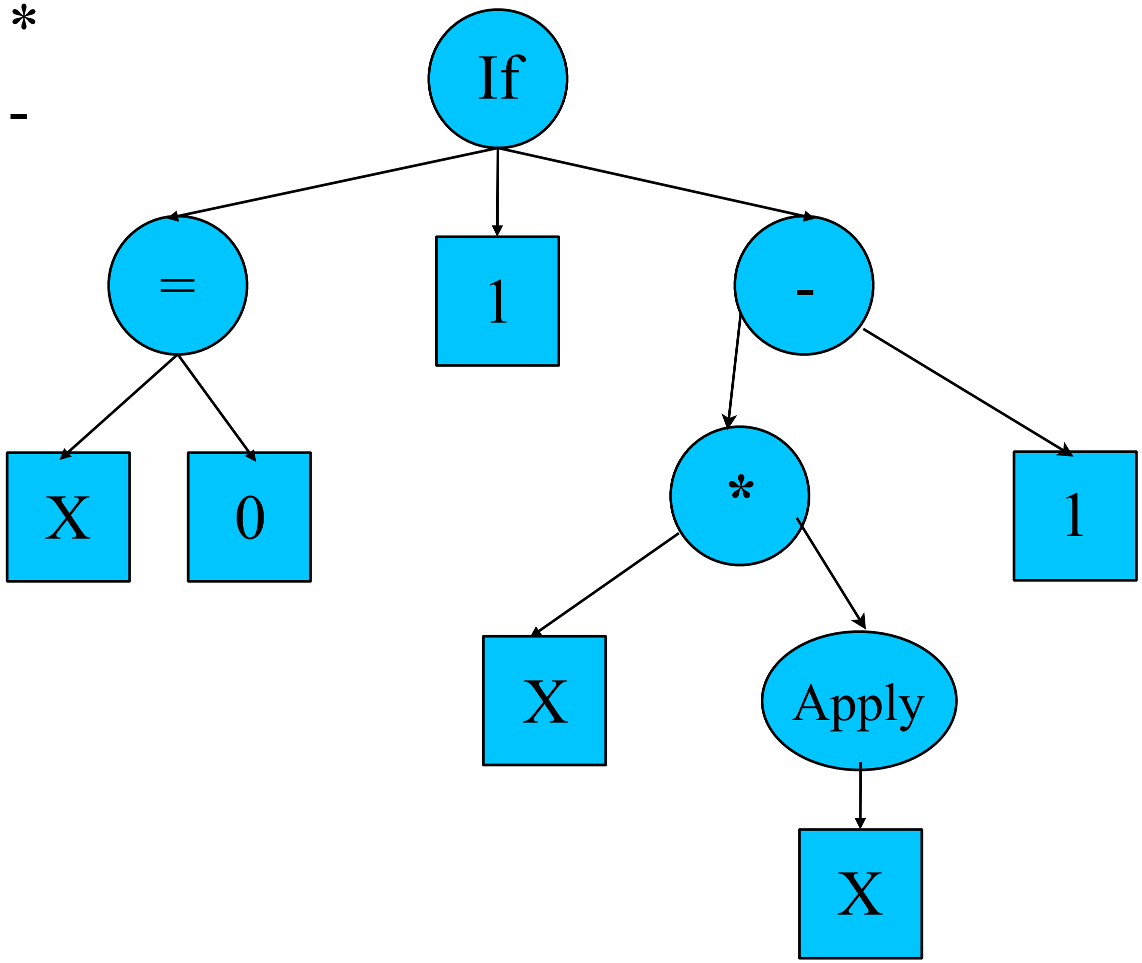
- What could possibly be wrong ?

- Remember the tree view ?



Oops...

- What could possibly be wrong ?
  - Precedence of operators
    - Apply
    - Binary *
    - Binary -



Recursion

- Function can be recursive
  - Nothing special about that
- Example
  - Factorial

```
- fun fact n = if n=0 then 1 else  n * fact (n-1);
val fact = fn: int -> int
- fact 5;
val it = 120 : int
```

Anonymous Version ?

- Exercise
  - Rewrite factorial as anonymous function

```
- val fact = fn n => if n=0 then 1 else  n * fact (n-1);

stdIn:16.42 Error: unbound variable or constructor: fact
```

- Why ?
  - Scoping rule!
- Can the language be fixed/improved ?

Anonymous Function

- Solution
  - Add a more permissive scoping rule

```
- val rec fact = fn n => if n=0 then 1 else  n * fact (n-1);
val fact = fn : int -> int
```

```
- val rec fact = fn n => if n=0 then 1 else  n * fact (n-1);
val fact = fn : int -> int
```

The keyword fun *is* ***syntactic sugar*** *for* val rec

Multiple arguments

- Objective
  - Write a function that takes two integers and returns their sum

```
- fun add (x,y) = x + y;
val add = fn : int * int -> int
- add(3,5);
val it = 8 : int
```

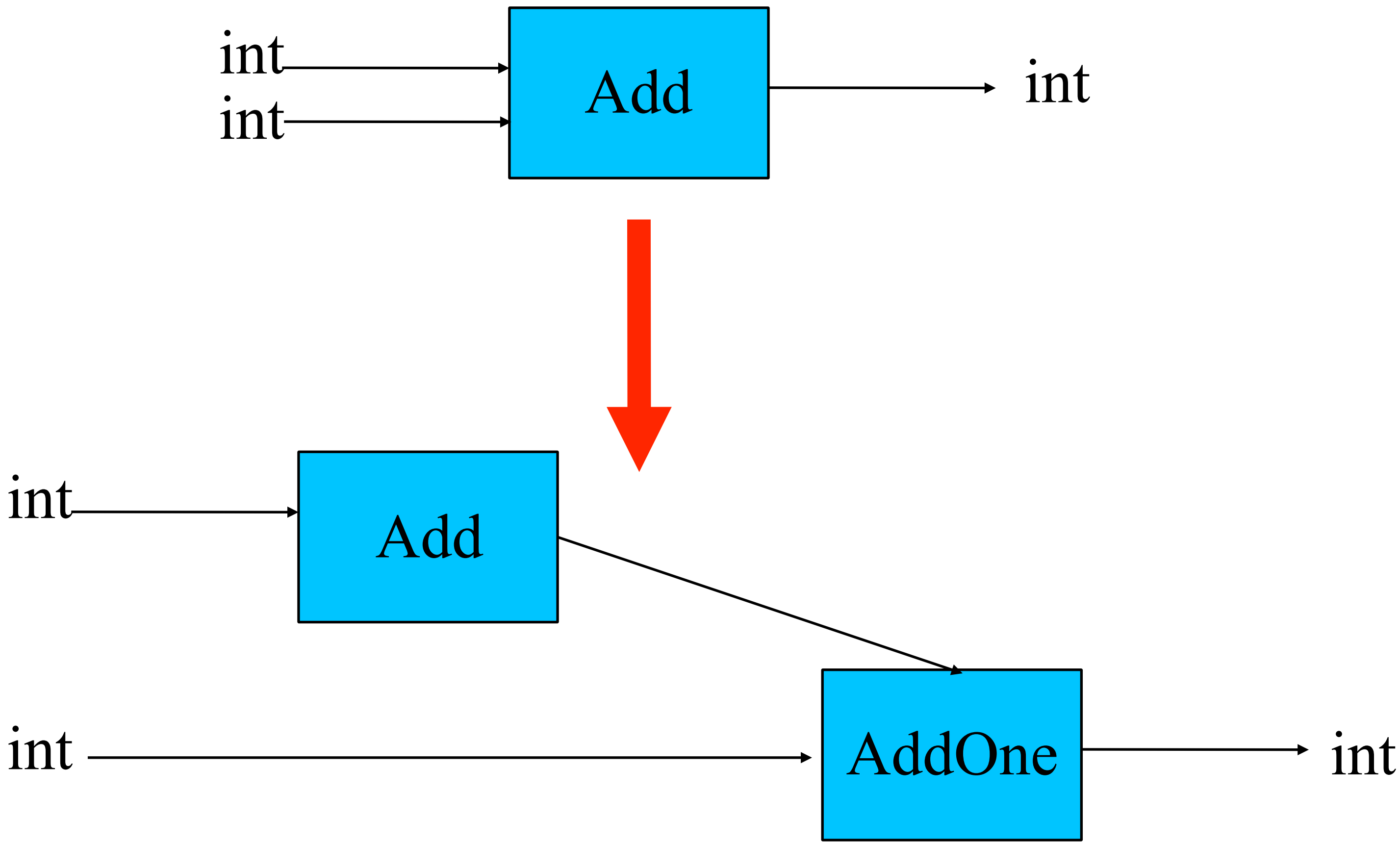
- Does this look familiar ?

One Argument

- ML function take only one argument
- The argument can be a tuple
  - Useful to pack several arguments.
- Question
  - Do we really need tuples ?
  - Can you write add without a tuple ?

Curry

- Or how to transform a function of two arguments into a function of one argument



Curry

- The addition example
  - The purpose of the Add box
    - Take an input argument x
    - Return a function that
      - Take one input argument y
      - Add y to x.
  - The return value of the Add box is
    - A specialized function that add its input to a fixed value
    - This is known as partial evaluation
    - Why is this good ?

Example

```
- fun add x = fn y => x+y;
val add = fn : int -> int -> int
- val z = add 3;
val z = fn : int -> int
- z 5;
val it = 8 : int
- z 7;
val it = 10 : int
- ((add 3) 5);
val it = 8 : int
- add 3 5;
val it = 8 : int
```

*Function application is **left** associative*

Syntactic Sugar

- Curried style is verbose
  - ML provides an abbreviation

```
- fun add x = fn y => x+y;
val add = fn : int -> int -> int
- fun sub x y = x - y;
val sub = fn : int -> int -> int
```

Notation is similar to standard form. Simply drop the parenthesis. Added benefit of automatic partial evaluation

Curry*ing*

- Curried and standard form are distinct
- Can we go from one to the other automatically ?
  - Problem 1
    - From a binary function
    - Produce a curried version
  - Problem 2
    - From a curried function
    - Produce a binary function

Currying

- Problem 1

```
- fun curry f = fn x => fn y => f(x,y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

- fun add(x,y) = x+y;
val add = fn : int * int -> int;

- curry add;
val it = fn : int -> int -> int
```

- Notice that
  - The function returned by curry *captures* f
  - The function returned by the adder *captures* x
  - This can be generalized to any number of arguments.
  - Place holders are used for the argument types.

Un-currying

- Problem 2

```
- fun uncurry f = fn (x,y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c )-> 'a * 'b -> 'c

- fun cadd x y = x+y;
val add = fn : int -> int -> int;

- val add = uncurry cadd;
val add = fn : int * int -> int
```

Food for Thought

- We have seen that
  - ML only offers functions of 1 argument
  - More arguments turn into tuples
  - We can go back and forth between the two notations
- Questions
  - Can we write functions of zero arguments in ML ?
  - Does it make sense ?
  - How can it be done ?

λ Type Inference is DEXPTIME

- Consider the SML Fragment

```
fun pair x y = fn z => z x y;

let val x1=fn y => pair y y in
   let val x2=fn y => x1(x1(y)) in
      let val x3=fn y => x2(x2(y)) in
         let val x4=fn y => x3(x3(y)) in
            x4(fn z => z)
         end
      end
   end
end;
```

Look at the Output….

```
-> (((((((((((?.X1 -> ?.X1) -> (?.X1 -> ?.X1) -> ?.X2) -> ?.X2)
                        -> (((?.X1 -> ?.X1) -> (?.X1 -> ?.X1) -> ?.X2)
                            -> ?.X2)
                           -> ?.X3)
                       -> ?.X3)
                      -> (((((?.X1 -> ?.X1) -> (?.X1 -> ?.X1) -> ?.X2)
                            -> ?.X2)
                           -> (((?.X1 -> ?.X1) -> (?.X1 -> ?.X1) -> ?.X2)
                               -> ?.X2)
                              -> ?.X3)
                          -> ?.X3)
                         -> ?.X4)
                     -> ?.X4)
                    -> (((((((?.X1 -> ?.X1) -> (?.X1 -> ?.X1) -> ?.X2)
                            -> ?.X2)
                           -> (((?.X1 -> ?.X1) -> (?.X1 -> ?.X1) -> ?.X2)
                               -> ?.X2)
                              -> ?.X3)
                          -> ?.X3)
                         -> (((((?.X1 -> ?.X1) -> (?.X1 -> ?.X1) -> ?.X2)
                               -> ?.X2)
                              -> (((?.X1 -> ?.X1) -> (?.X1 -> ?.X1) -> ?.X2)
                                  -> ?.X2)
                                 -> ?.X3)
                             -> ?.X3)
                            -> ?.X4)
                        -> ?.X4)
                       -> ?.X5)
                   -> ?.X5)
```

Type Inference Today?

- Yes!
- “auto”

```
int x = 10;
```

→

```
auto x = 10;
```

```
std::vector<int> container = …;
for(std::vector<int>::iterator i = container.begin();
    i != i.end();
    i++) …
```

```
std::vector<int> container = …;
for(auto i = container.begin();
    i != i.end();
    i++) …
```