Principles of Programming Languages
Lecture 13

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig; PL Concepts and Constructs by Sethi; Concepts of PL by Sebesta; ML for the Working Prog. By Paulson; Prog. in Prolog by Clocksin and Mellish; PL Pragmatics by Scott

ML: pattern matching – cont'd

- **Patterns may consist of constants (integers , true, false..) , tuples and variables. Arithmetic or logical expressions are invalid.**
  - E.g fun wrong(x=y) = "…"

- **No duplicates in patterns**
  - E.g.
    ```
    fun wrong_equal (x,y)  = true
    |   wrong_equal (x,y) = false;
    ```

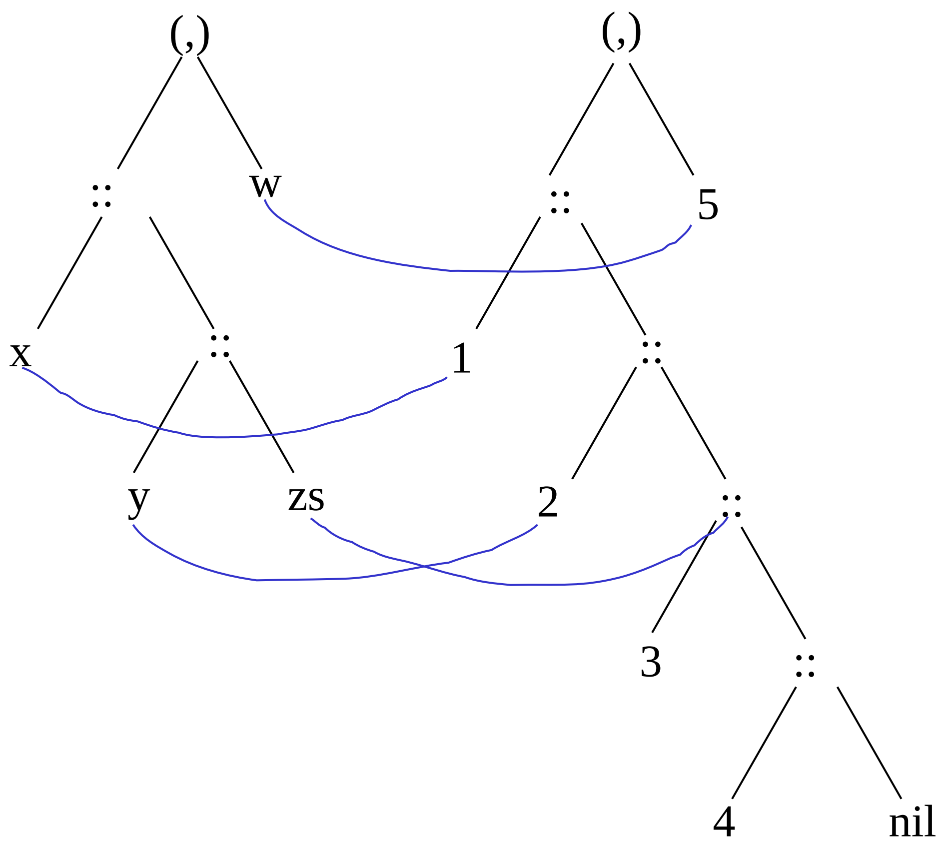
- **Pattern matching with wild cards**
  - E.g. fun first (x,_) = x;
  - Matches anything like a variable. Binds nothing.
  - Avoid need to name every pattern

- **ML does extensive pattern checking**
  - E.g.
    ```
    - fun reverse (h::t) = reverse(t) @ [h];
    > Warning: match nonexhaustive
    ```

ML: pattern matching – cont'd

- **How ML matches patterns?**

```
fun func (x::y::zs,w)  =
     |     (…)          =
     |     (…)          =

func([1,2,3,4],5)
```



ML: record type

- **We have seen list [ ] and tuple ( )…**

- **Record syntax**

  { <field$_1$>:<type$_1$>, <field$_2$>:<type$_2$>,….,<field$_n$>:<type$_n$>}

- **A record instance is defined as**

  { <label$_1$>=<value$_1$>, <label$_2$>=<value$_2$>,….,<label$_n$>=<value$_n$>}

- **A record is a structured data type in which each element is accessed by a unique name.**
  - E.g. { name: string,
    age: int,
    salary: int}
  - {name = “Dave”, age = 77, salary=99000}

  > val it = {name = “Dave”, age = 77, salary=99000} : {name:string, age:int,salary:int}

> A field called name with type string
> A field called called age with type int
> A field called salary with type int

ML: record type – cont'd

- **Operations**
  - # operator to extract a field from a record instance
  - E.g.

```
- #salary {name =“john”, age=35, salary=90};
> val it = 90 : int

- #options {startcity="toronto",endcity="boston",
                                   options=("12",10,"K")};
> val it = ("12",10,"K") : string * int * string
```

ML: record type – cont'd

- **Named Types**
  - ML provides a way to give a name to a type
    - E.g - type waitress = { name: string , wages: int, tips: int };
  - Named types can be used anywhere that ML types can be.
    - E.g.

```
- fun income (w: waitress) =
                    #wages w + #tips w;
> val income = fn : waitress -> int

- fun income (w: { name: string , wages: int, tips: int }) =
            #wages w + #tips w;
> val income = fn : {name:string, tips:int, wages:int} -> int
```

  - Named types can be used in type declaration
    - E.g.

```
- type waitresses = waitress list;
- [{name="sally", wages= 20, tips=10},
   {name="alice", wages= 15, tips=15}]
```

ML: record type – cont'd

- **Named Types – cont'd:**
  - E.g.: *finding the total income of all waitresses*

```
> type waitress = { name: string , wages: int, tips: int };
> fun income (w: waitress) =
                      #wages w + #tips w;
> type waitresses = waitress list;
> fun total (WL: waitresses) =
           if WL = [] then 0
             else income(hd WL) + total(tl WL);
```

```
- WL = [{name="sally", wages= 20, tips=10},
   {name="alice", wages= 15, tips=20},
   {name="sue", wages= 25, tips=20}]

-total(WL)
110
```

→

30 +
[{name="alice", wages= 15, tips=20}
{name="sue", wages= 25, tips=20}]
----
30 + 35 +
[{name="sue", wages= 25, tips=20}]
----
30 + 35 + 45

ML: pattern matching on records

- **Recall syntax**
```
fun <func> <pattern1>  = <expression1>
|   <func> <pattern2>  = <expression2>
.......
|   <func> <patternn>  = <expressionn>
```

- **You can use patterns to match on records**
  - E.g *finding the total income of all waitresses*

```
> type waitress = { name: string , wages: int, tips: int };
> fun income (w: waitress) =
                            #wages w + #tips w;
> type waitresses = waitress list;
> fun total ([]: waitresses) = 0
  |   total(W::WLTail)        = (income W) + (total WLTail);
```

- **You can also use wild cards …**
  - E.g.

```
> fun  costly({price:int, …}: footype) = price > 100.0;
```

ML: …

- What does ML infer about this function?

```
- fun length L =
        if (null L) then 0
              else 1 + length(tl L);
- length[1,2,3,4]
> val it = 4 : int
- length["ab","cd","xy"];
> val it = 3 : int
- length[[1,2],[3,4],[123,123,222],[1]];
> val it = 4 : int
```

  – Seems length has/accept these types
    - int list → int
    - String list → int
    - int list list → int

  – Obviously, we would like *length* to apply to any kind of list.

ML: …

- **What does ML infer about this function?**

```
- fun length L =
      if (null L) then 0
          else 1 + length(tl L);
```

  – In ML, *length* has all of these types. This is written as

    length: 'a list → int

    - 'a is a type variable. It stands for any type
    - This means that the input to length is a list of items all of type 'a where 'a can be int, string, int list, or any other type.

  – In fact, that’s what ML infers for this function

```
- fun length L = if (null L) then 0 else 1 + length(tl L);
> val length = fn : 'a list -> int
```

ML: polymorphism

- **Greek:** *poly = many , morph = form*

- **Definitions:**
  - Polymorphism:
    - dictionary.com: the capability of assuming different forms; the capability of widely varying in form. The occurrence of different forms, stages, or types
    - Software: a value/variable can belong to multiple types
  - Monomorphism:
    - Dictionary.com: having only one form, same genotype…
    - Software: every value/variable belongs to exactly one type

- **Why is useful?**
  - To avoid redundant function definitions, e.g.:

    int-length : int list → int

    real-length: real list → int

    string-length: string list → int ……….

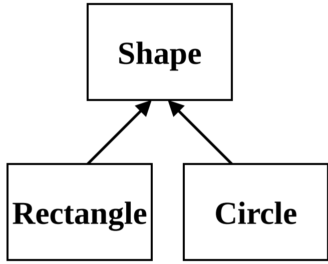
    code for each of these functions would be virtually identical!
  - Polymorphism adds flexibility & great convenience…. but…

ML: polymorphism types

- **Ad-hoc polymorphism:**
  - Different operations on different types known by the same name *(also called overloading)*
  - E.g. 3 + 4 ***vs.*** 3.1 + 4 *compiler/interpreter must change 4 to 4.0 first*
    vs. "ab" + "cd"
- **Inheritance polymorphism:**
  - Use sub-classing to define new versions of existing functions *(OO)*
  - E.g.:

```
public class Shape{

    public void draw( int x, int y){
// do nothing
    }
  }

public class Rectangle extends Shape{
    public void draw( int x, int y){
            // draws a rectangle
    }
}
```

```
public class Circle extends Shape{

    public void draw( int x, int y){
    }
}
....
Shape myShape;
myShape = new Rectangle( );

..... // some other piece of code
myShape.draw( );
```



ML: polymorphism types – cont'd

- **Parametric Polymorphism(ML):**
  - Allows types to be parameters to functions and other types.
  - Basic idea is to have a type variable…
  - Type of function depend on type of parameter
  - Implementation (ML):
    - One copy of code is generated
    - Polymorphic parameters must internally be implemented as pointers

ML: polymorphism – cont’d

- **Polymorphic functions are very common in ML:**

```
- fun id X = X;
> val id = fn : 'a -> 'a
```

```
- id 7;
> val it = 7 : int
- id "abc";
> val it = "abc" : string
```

```
- fun listify X = [X];
> val listify = fn : 'a -> 'a list
```

```
- listify 3;
> val it = [3] : int list
- listify 7.3;
> val it = [7.3] : real list
```

```
- fun double X = (X,X);
> val double = fn : 'a -> 'a * 'a
```

```
- double “xy”;
> val it = ("xy","xy") : string * string
- double [1,2,3];
> val it = ([1,2,3],[1,2,3]) : int list * int list
```

```
- fun inc(N,X) = (N+1,X);
> val inc = fn : int * 'a -> int * 'a
```

```
- inc (4,(34,5));
val it = (5,(34,5)) : int * (int * int)
```

ML: polymorphism – cont'd

- **Polymorphic functions are very common in ML:**

```
- fun swap(X,Y) = (Y,X);
> val swap = fn : 'a * 'b -> 'b * 'a
```

```
- swap ("abc",7);
> val it = (7,"abc") : int * string
- swap (13.4,[12,3,3]);
val it = ([12,3,3],13.4) : int list * real
```

```
- fun pair2list(X,Y) = [X,Y];
> val pair2list = fn : 'a * 'a -> 'a list
```

```
- pair2list(1,2);
> val it = [1,2] : int list
- pair2list(1,"cd");
?
```

```
- fun apply(Func,X) = Func X;
> val apply = fn : ('a -> 'b) * 'a -> 'b
```

```
- apply (hd, [1,2,3]);
> val it = 1 : int
- apply (length, [23,100]);
> val it = 2 : int
```

```
- fun applytwice(Func,X) = Func(Func X);
> val applytwice = fn : ('a -> 'a) * 'a -> 'a
```

```
- applytwice (square,3);
> val it = 81 : int
- applytwice (tl, [1,2,3,4]);
- ?
```

ML: polymorphism – cont'd

- **Operators that restrict polymorphism**
  - Arithmetic operators: + , -, * and ~
  - Division-related operations such as / , div and mod
  - Inequality comparison operators: < , <=, >=, and >
  - Boolean connectives: andalso, orelse and not
  - String concatenation operator: ^
  - Type conversion operators
    - E.g. ord, chr, real, str, floor, ceiling, round, truncate,…

- **Operators that allow polymorphism**
  - Tuple operators
  - List operators
  - Equality operators = and <>

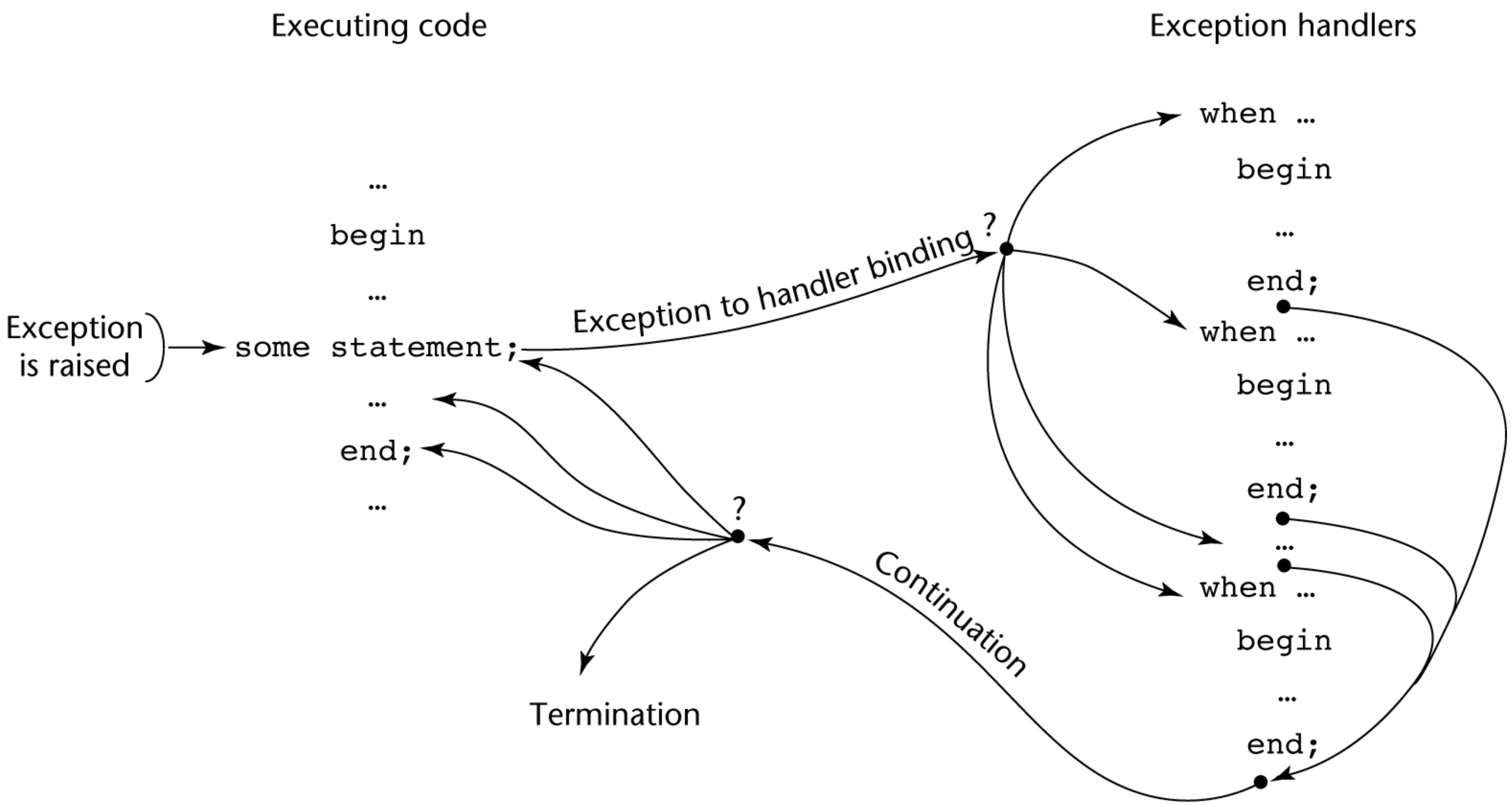
Exceptions: introduction

- **An exception is any unusual event, erroneous or not, that is detectable either by hardware or software and that may require special processing.**

- **The special processing that may be required by the detection of an exception is called exception handling. This processing is done by a code unit called the exception handler.**

- **Why do we need exceptions if the language is strongly typed?**
  - In a language without exception handling: when an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated.
  - In a with exception handling: programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing.

Exceptions: execution flow



Exceptions: why?

- **How was error handling done in early programming languages?**
  - Send an auxiliary parameter or use the return value to indicate the return status of a subprogram (e.g. C standard library functions)

```
nError = mult(matrix1,matrix2,product);
if( nError == -1){ // error

}
else{  // no error, continue normally

}
```

  - Pass a label parameter to the subprogram. If an error occurs, use the label to jump to another location in the program (e.g. FORTRAN)

```
mult(matrix1,matrix2,product,label)
          ….
          if error
            goto label
```

  - Pass an error-handler subprogram to the called subprogram.

```
mult(matrix1,matrix2,product,error_func)

          if error
            error_func(…)
```

ML: exceptions

- **Syntax** **exception <exception-name> of <type-expression>**

- **Example:**

```
- exception NegArg of int;
> exception NegArg of int

- fun fact N =   if N = 0 then 1
                  else if N > 0 then N * fact(N-1)
                     else raise NegArg(N);
> val fact = fn : int -> int

-  fact(5);
> val it = 120 : int

- fact(~5);
> uncaught exception NegArg raised at: …
```