Review

- Features learned: functions, tuples, lists, let expressions, options, records, datatypes, case expressions, type synonyms, pattern matching

- Today:
  - Exceptions
  - Deep pattern matching
  - Call stacks (intro to tail recursion)

Exceptions

An exception binding introduces a new kind of exception

```latex
exception MyFirstException
definition MySecondException of int * int
definition raise MyFirstException
raise MySecondException (7,9)
```

The `raise` primitive raises (aka, throws) an exception

```latex
hd lst handle Empty => 0
```

A `handle` expression can handle (aka, catch) an exception
- If doesn’t match, exception continues to propagate
- Not good style. better to use pattern matching on list!

Exceptions are very similar to constructors

- Declaring an exception makes a constructor for type `exn`
  - Can’t do that for datatypes! All their constructors declared in one place.
- Can pass values of type `exn` anywhere (e.g., function arguments)
  - Not too common to do this but can be useful

- Handle can have multiple branches with patterns for type `exn`

```latex
\texttt{e handle p1 => e1}
| \texttt{p2 => e2}
| \texttt{...}
| \texttt{pn => en}
```

Exception semantics

```latex
\texttt{e handle p1 => e1}
| \texttt{p2 => e2}
| \texttt{...}
| \texttt{pn => en}
```

- Type-checking: If e, e1, ... en have type t, then entire `handle` expression has type t
- Evaluation:
  - Evaluate `e` to a value v and return v
  - But if `e` raises exception that matches p1, evaluate ei to value vi and return vi
  - What if ei raises an exception? Then `handle` expression raises that exception and it’s “somebody else’s problem”
DEEP PATTERN MATCHING

Deep pattern matching
- Patterns can nest arbitrarily deep
  - (just like expressions)
  - Easy-to-read, nested patterns can replace hard-to-read, nested case expressions
- Remember, intuition is that pattern “looks like” value
- Examples:
  - Pattern \( a::b::c::d \) matches all lists with >= 3 elements
  - Pattern \( a::b::c::[] \) matches all lists with 3 elements
  - Pattern \( ((a,b),(c,d))::e \) matches all non-empty lists of pairs of pairs

Useful example: zip/unzip 3 lists

```
fun zip3 lists =
  case lists of
  ([],[],[]) => []
  | (hd1::tl1,hd2::tl2,hd3::tl3) =>
    (hd1,hd2,hd3)::zip3 tl1,tl2,tl3
  | _ => raise ListLengthMismatch

fun unzip3 triples =
  case triples of
  [] => ([],[],[])
  | (a,b,c)::tl =>
    let val (l1,l2,l3) = unzip3 tl
    in
    (a::l1,b::l2,c::l3)
    end
```

More examples in notes and code… study them!

Precise definition of pattern matching
- Evaluation rules for pattern-matching
  - Given a pattern \( p \) and a value \( v \), decide
    1. Does pattern match value?
    2. If so, what variable bindings are introduced?
- If \( p \) is a variable \( x \), the match succeeds and \( x \) is bound to \( v \)
- If \( p \) is \( _ \), the match succeeds and no bindings are introduced
- If \( p \) is \( (p1,\ldots,pn) \) and \( v \) is \( (v1,\ldots,vn) \), the match succeeds if and only if \( p1 \) matches \( v1 \), \ldots, \( pn \) matches \( vn \). The bindings are the union of all bindings from the sub-matches
- If \( p \) is \( C \ p1 \), the match succeeds if \( v \) is \( C \ v1 \) (i.e., the same constructor) and \( p1 \) matches \( v1 \). The bindings are the bindings from the sub-match.
- …more rules in notes. Study them!

Complete syntax and semantics of case expressions (for us)
- Syntax
  - `case e of p1 => e1 | p2 => e2 | ... | pn => en`
- Evaluation rules:
  1. Evaluate \( e \) to a value
  2. If \( p1 \) is the first pattern to match the value, then evaluate \( e1 \) to value \( v1 \) and return \( v1 \)
  3. Evaluation of \( e1 \) is in dynamic environment extended by the match
  4. Pattern matches value if it “looks like” the value
    - Pattern \( C (x1,\ldots,xn) \) matches value \( C1 (v1,\ldots,vn) \) and extends the environment with \( x1 \) bound to \( v1 \), \ldots, \( xn \) to \( vn \)
    - Pattern \( _ \) matches any value

Complete syntax and semantics of case expressions (for us)
- Syntax
  - `case e of p1 => e1 | p2 => e2 | ... | pn => en`
- Evaluation rules:
  1. Evaluate \( e \) to a value
  2. If \( p1 \) is the first pattern to match the value, then evaluate \( e1 \) to value \( v1 \) and return \( v1 \)
  3. Evaluation of \( e1 \) is in dynamic environment extended by the match bindings
- Type-checking rules:
  1. If \( e \), \( p1 \), \( \ldots \) have type \( ta \) and \( e1 \ldots en \) have type \( tb \)
     then entire case expression has type \( tb \)
  2. When type-checking \( ej \), any variables that \( pj \) binds are added to the static environment
FUNCTION PATTERNS

Last lecture...

A function that takes one triple of type int*int*int and returns an int that is their sum:

```
fun sum_triple (x, y, z) = x + y + z
```

A function that takes three int arguments and returns an int that is their sum:

```
fun sum_triple (x, y, z) = x + y + z
```

See the difference? (Me neither.)

The truth about functions

• In ML every function takes exactly one argument

• What we call multi-argument functions are just functions taking one
tuple argument, implemented with a tuple pattern in the function
binding
  – Elegant and flexible language design

• Enables cute and useful things you can’t do in Java, e.g.,

```
fun rotate_left (x, y, z) = (y, z, x)
fun rotate_right t = rotate_left(rotate_left t)
```

One-of types in function bindings

Functions can be defined “by cases”, just like in math:

```
fun f p1 = e1
  | f p2 = e2
  ...
  | f pn = en
```

Example:

```
fun eval (Constant i) = i
  | eval (Add(e1,e2)) = (eval e1) + (eval e2)
  | eval (Negate e1) = ~ (eval e1)
```

One-of types in function bindings

Functions can be defined “by cases”, just like in math:

```
fun f x =
  case x of
  p1 => e1
  | p2 => e2
  ...
  | pn => en
```

Semantics is exactly the same as:

```
fun f x =
  case x of
  p1 => e1
  | p2 => e2
  ...
  | pn => en
```

TAIL RECURSION
Recursion

Should now be comfortable with recursion:

- No harder than using a loop (whatever that is)
- For recursive datatypes, recursion is more elegant than loops:
  - When processing a tree (e.g., evaluate an arithmetic expression)
  - When appending lists
- Now:
  - How to reason about efficiency of recursion
  - The importance of tail recursion
  - Using an accumulator to achieve tail recursion
  - Functional programming idiom (no new syntax/semantics)

Call-stacks

While a program runs, there is a call stack of function calls that have started but not yet returned

- Calling a function f pushes an instance of f on the stack
- When a call to f to finishes, it is popped from the stack

These stack-frames store information like the value of local variables and "what is left to do" in the function

Because of recursion, multiple stack-frames may be calls to the same function

Example

fun fact n = if n=0 then 1 else n*fact(n-1)
val x = fact 3

fact 3
fact 3: 3*1
fact 2
fact 2: 2*1
fact 1
fact 1: 1*1
fact 0: 1

Example Revised

fun fact n = let fun aux(n,acc) = if n=0 then acc else aux(n-1,acc*n) in aux(n,1) end
val x = fact 3

fun fact n = if n=0 then 1 else n*fact(n-1)
val x = fact 3

Call-stacks

While a program runs, there is a call stack of function calls that have started but not yet returned

- Calling a function f pushes an instance of f on the stack
- When a call to f to finishes, it is popped from the stack

These stack-frames store information like the value of local variables and "what is left to do" in the function

Because of recursion, multiple stack-frames may be calls to the same function

The call-stacks

fun fact n = if n=0 then 1 else n*fact(n-1)
val x = fact 3

fun fact n = let fun aux(n,acc) = if n=0 then acc else aux(n-1,acc*n) in aux(n,1) end
val x = fact 3

Still recursive, more complicated, but the result of recursive calls is the result for the caller (no remaining multiplication)

An optimization

It is unnecessary to keep around a stack-frame just so it can get a callee’s result and return it without any further evaluation

ML recognizes these tail calls in the compiler and treats them differently:

- Pop the caller before the call, allowing callee to reuse the same stack space
- Implementation turns out to be as efficient as a loop
- Not just ML, any good implementation of functional language does tail-call optimization

Etc.