

	< □ >	・ ▲ 副 ▶ ▲ 国 ▶ ▲ 国 ▶ ● ▲ ④ ▲ ④			() < ≣ < ≡ < < <
Exceptions	Tail recursion	Continuations	Exceptions	Tail recursion	Continuations
Exceptions			finally	and resource cleanup	

- In earlier lectures, we considered several approaches to error handling
- *Exceptions* are another popular approach (supported by Java, C++, Scala, ML, Python, etc.)
- The throw e statement raises an exception e
- A try/catch block runs a statement; if an exception is raised, control transfers to the corresponding handler

```
try { ... do something ... }
catch (IOException e)
```

```
{... handle exception e ...}
```

- catch (NullPointerException e)
 - {... handle another exception...}

- What if the try block allocated some resources?
- We should make sure they get deallocated!
- finally clause: gets run at the end whether or not exception is thrown

```
InputStream in = null;
try { in = new FileInputStream(fname);
      ... do something with in ... }
catch (IOException exn) {...}
finally { if(in != null)
            in.close(); }
```

• Java 7: "try-with-resources" encapsulates this pattern, for resources implementing AutoCloseable interface

Exceptions

Tail recursion

Exceptions

throws clauses

• In Java, potentially unhandled exceptions typically need to be *declared* in the types of methods

```
void writeFile(String filename)
    throws IOException {
  InputStream in = new FileInputStream(filename);
```

```
... write to file ...
in.close();
```

```
}
```

- This means programmers using such methods know that certain exceptions need to be handled
- Failure to handle or declare an exception is a type error!
 - (however, certain unchecked exceptions / errors do not need to be declared, e.g. NullPointerException)

Exceptions in Scala

• As you might expect, Scala supports a similar mechanism:

```
try { ... do something ... }
catch {
 case exn: IOException =>
    ... handle IO exception...
 case exn: NullPointerException =>
    ... handle null pointer exception...
} finally { ... cleanup ...}
```

- Main difference: The catch block is just a Scala pattern match on exceptions
 - Scala allows pattern matching on types (via isInstanceOf/asInstanceOf)
- Also: throws clauses not required

Exceptions for	or shortcutting	F	Exceptions i	n practice	
Exceptions	Tail recursion	Continuations Ex	Exceptions	Tail recursion	Continuations
	4 LL P				

```
• We can also use exceptions for "normal" computation
```

```
def product(l: List[Int]) = {
 object Zero extends Throwable
 def go(l: List[Int]): Int = 1 match {
   case Nil => 1
   case x::xs =>
     if (x == 0) {throw Zero} else {x * go(xs)}
 }
 try { go(1) }
 catch { case Zero => 0 }
}
```

potentially saving a lot of effort if the list contains 0

- Java:
 - Exceptions are subclasses of java.lang.Throwable
 - Method types must declare (most) possible exceptions in throws clause
 - compile-time error if an exception can be raised and not caught or declared
 - multiple "catch" blocks; "finally" clause to allow cleanup
- Scala:
 - doesn't require declaring thrown exceptions: this becomes especially painful in a higher-order language...
 - "catch" does pattern matching

Exceptions

Modeling exceptions

Tail recursion

• We will formalize a simple model of exceptions:

• while e_1 handle $\{x \Rightarrow e_2\}$ evaluates e_1 and, if an

• Define L_{Exn} as L_{Rec} extended with exceptions

 $e ::= \cdots | \text{raise } e | e_1 \text{ handle } \{x \Rightarrow e_2\}$

• Here, raise *e* throws an arbitrary value as an "exception"

exception is thrown during evaluation, binds the value v

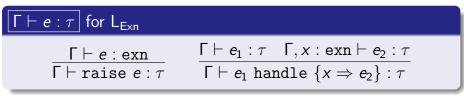
Continuations Exceptions

Exceptions and types

• Exception constructs are straightforward to typecheck:

 $\tau ::= \cdots \mid exn$

• Usually, the exn type is extensible (e.g. by subclassing)



- Note: raise *e* can have any type! (because raise *e* never returns)
- The return types of e_1 and e_2 in handler must match.

		▲□▶ ▲□▶ ▲≣▶ ▲≣▶ ■ うみの		・ロ・・西・・川・・田・ 日 うくぐ
Exceptions	Tail recursion	Continuations Exceptions	Tail recursion	Continuations
1.	 	C ···	· · · · ·	

Interpreting exceptions

to x and evaluates e.

• We can extend our Scala interpreter for L_{Rec} to manage exceptions as follows:

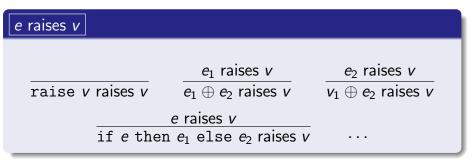
```
case class ExceptionV(v: Value) extends Throwable
def eval(e: Expr): Value = e match {
    ...
    case Raise(e: Expr) => throw (ExceptionV(eval(e)))
    case Handle(e1: Expr, x: Variable, e2:Expr) =>
    try {
        eval(e1)
        } catch (ExceptionV(v)) {
        eval(subst(e2,v,x))
        }
```

• This might seem a little circular!

・ロト ・日本 ・ 日本 ・ 日本 ・ 日本 ・ の へ (や)

Semantics of exceptions

- To formalize the semantics of exceptions, we need an auxiliary judgment *e* raises *v*
- Intuitively: this says that expression *e* does not finish normally but instead raises exception value *v*



• The most interesting rule is the first one; the rest are "administrative"

Tail recursion

Semantics of exceptions

• We can now define the small-step semantics of handle using the following additional rules:

$e\mapsto e'$	
$e_1\mapsto e_1'$	
$\overline{e_1 ext{ handle } \{x \Rightarrow e_2\} \mapsto e_1' ext{ handle } \{x \Rightarrow e_2\}}$	
$\overline{v_1 \; ext{handle} \; \{x \Rightarrow e_2\} \mapsto v_1}$	
e_1 raises v	
$e_1 ext{ handle } \{x \Rightarrow e_2\} \mapsto e_2[v/x]$	

- If e_1 steps normally to e'_1 , take that step
- If e_1 raises an exception v, substitute it in for x and evaluate e₂

Tail recursion

- A function call is a *tail call* if it is the last action of the calling function. If every recursive call is a tail call, we say f is tail recursive.
- For example, this version of fact is not tail recursive:

```
def fact1(n: Int): Int =
 if (n == 0) {1} else {n * (fact1(n-1))}
```

• But this one is:

```
def fact2(n: Int) = {
 def go(n: Int, r: Int): Int =
   if (n == 0) \{r\} else \{go(n-1,n*r)\}
 go(n,1)
}
```

◆□▶ ◆□▶ ◆三▶ ◆三▶ ● 三 のへの - 32 Tail recursion Exceptions Tail recursion Continuations Tail recursion and efficiency

- Tail recursive functions can be compiled more efficiently
- because there is no more "work" to do after the recursive call
- In Scala, there is a (checked) annotation @tailrec to mark tail-recursive functions for optimization

```
def fact2(n: Int) = {
  @tailrec
 def go(n: Int, r: Int): Int =
   if (n == 0) \{r\} else \{go(n-1,n*r)\}
 go(n,1)
}
```

Continuations [non-examinable]

- Conditionals, while-loops, exceptions, "goto" are all form of control abstraction
- Continuations are a highly general notion of control abstraction, which can be used to implement exceptions (and much else).
- Material covered from here on is non-examinable.
 - just for fun!
 - (Depends on your definition of fun, I suppose)

			Tail recursion	Continuat			
ontinuations		How doe	How does this work?				
 A continuation is a function representing "the rest of the computation" Any function can be put in "continuation-passing form" for example def fact3[A] (n: Int, k: Int => A): A = if (n == 0) {k(1)} else {fact3(n-1, {m => k (n * m)})} This says: if n is 0, pass 1 to k otherwise, recursively call with parameters n − 1 and λr.k(n × r) "when done, multiply the result by n and pass to k" 		$\begin{array}{c} \text{if (r} \\ \text{if (r} \\ \\ \text{ing form''} \\ \end{array} \\ \begin{array}{c} f \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ $	$ \begin{array}{l} \hline \text{def fact3[A] (n: Int, k: Int => A): A =} \\ \text{if (n == 0) {k(1)} else {fact3(n-1, {r => k (n * r)})} \\ \hline \\ & fact3(3, \lambda x.x) \\ \mapsto & fact3(2, \lambda r_1.(\lambda x.x) (3 \times r_1)) \\ \mapsto & fact3(1, \lambda r_2.(\lambda r.(\lambda x.x) (3 \times r)) (2 \times r_2)) \\ \mapsto & fact3(0, \lambda r_3.(\lambda r_2.(\lambda r_1.(\lambda x.x) (3 \times r_1)) (2 \times r_2)) (1 \times r_3)) \\ \mapsto & (\lambda r_3.(\lambda r_2.(\lambda r_1.(\lambda x.x) (3 \times r_1)) (2 \times r_2)) (1 \times r_3)) 1 \\ \mapsto & (\lambda r_2.(\lambda r_1.(\lambda x.x) (3 \times r_1)) (2 \times r_2)) (1 \times 1) \\ \mapsto & (\lambda r_1.(\lambda x.x) (3 \times r_1)) (2 \times 1) \\ \mapsto & (\lambda x.x) (3 \times 2) \end{array} $				
		\mapsto (
ptions •	Tail recursion	< 문 → < 문 > 문 ∽) < (~ Continuations Exceptions	Tail recursion	▶ < @ > < 볼 > < 볼 > = Continua			
		< 문 → < 문 > 문 ∽) < (~ Continuations Exceptions	< 🗆 >				
terpreting def eval[A] // Arithm	<pre>Tail recursion LArith using continuations (e: Expr, k: Value => A): A = e m</pre>	Interpret Interpret Interpret Interpret Interpret Interpret	Tail recursion	Continua			
terpreting def eval[A] // Arithm case Num(case Plus eval(e1	<pre>Tail recursion LArith using continuations (e: Expr, k: Value => A): A = e m netic</pre>	Continuations Exceptions Interpret match { // Bo case case ())})})	Tail recursion Tail recursion al [A] (e: Expr, k: Value => A): A poleans	Continua = e match {			

・ロン・4回シ・ミン・4目シー目・4回シ

Exceptions	Tail recursion	Continuations	Exceptions	Tail recursion	Continuations
Interpretin	g L_{Let} using continuations		Interpreting	L _{Rec} using continuations	
 // Let- case Le eval(e	<pre>[A] (e: Expr, k: Value => A): A = e match -binding et(e1,x,e2) => e1, {v => l(subst(e2,v,x),k)})</pre>		<pre>// Function case Lambd case Rec(f case Apply eval(e1, eval(e case case eva }}))) }</pre>	<pre>ha(x,ty,e) => k(LambdaV(x,ty,e) f,x,ty1,ty2,e) => k(RecV(f,x,ty n(e1,e2) =></pre>) 1,ty2,e))
Exceptions	Tail recursion	Continuations	Exceptions	Tail recursion	Continuations
Interpretin	g L _{Exn} using continuations		Summary		

Interpreting L_{Exn} using continuations

To deal with exceptions, we add a second continuation h for handling exceptions. (Cases seen so far just pass h along.)

When raising an exception, we forget ${\tt k}$ and pass to ${\tt h}.$ When handling, we install new handler using e2

- Today we completed our tour of
 - Type soundness
 - References and resource management
 - Evaluation order
 - Exceptions and control abstractions (today)
- which can interact with each other and other language features in subtle ways
- Next time:
 - review lecture
 - information about exam, reading