Overview

- We have been considering several high-level aspects of language design:
  - Type soundness
  - References
  - Evaluation order
- Today we complete this tour and examine:
  - Exceptions
  - Tail recursion
  - Other control abstractions
Exceptions

- 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*
  
  ```java
  try { ... do something ... }
  catch (IOException e)
  { ... handle exception e ... }
  catch (NullPointerException e)
  { ... handle another exception... }
  ```
finally and resource cleanup

- 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

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

```java
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:

```scala
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 shortcutting

- We can also use exceptions for “normal” computation

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

- potentially saving a lot of effort if the list contains 0
Exceptions in practice

- **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
Modeling exceptions

- We will formalize a simple model of exceptions:

  \[ e ::= \cdots \mid \text{raise } e \mid e_1 \text{ handle } \{ x \Rightarrow e_2 \} \]

- Here, \text{raise } e throws an arbitrary value as an “exception”

- while \( e_1 \text{ handle } \{ x \Rightarrow e_2 \} \) evaluates \( e_1 \) and, if an exception is thrown during evaluation, binds the value \( \nu \) to \( x \) and evaluates \( e \).

- Define \( L_{\text{Exn}} \) as \( L_{\text{Rec}} \) extended with exceptions
Exceptions and types

- Exception constructs are straightforward to typecheck:
  \[ \tau ::= \cdots \mid \text{exn} \]

- 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.
Interpreting exceptions

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

```scala
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!
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 $v$

\[
\begin{align*}
e &\quad \text{raises} \quad v \\
\text{raise} \quad v &\quad \text{raises} \quad v \\
\text{if} \quad e \quad \text{then} \quad e_1 \quad \text{else} \quad e_2 &\quad \text{raises} \quad v \\
\end{align*}
\]

$e_1$ raises $v$

$e_2$ raises $v$

$v_1$ raises $v$

$e_1 \oplus e_2$ raises $v$

$v_1 \oplus e_2$ raises $v$

$v_1 \oplus e_2$ raises $v$

$\therefore$ The most interesting rule is the first one; the rest are “administrative”
Semantics of exceptions

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

\[
\begin{align*}
& e \mapsto e' \\
& e_1 \mapsto e_1' \\
& e_1 \text{ handle } \{ x \Rightarrow e_2 \} \mapsto e_1' \text{ handle } \{ x \Rightarrow e_2 \} \\
& v_1 \text{ handle } \{ x \Rightarrow e_2 \} \mapsto v_1 \\
& e_1 \text{ raises } v \\
& e_1 \text{ handle } \{ x \Rightarrow e_2 \} \mapsto e_2[v/x]
\end{align*}
\]

- 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 \( e_2 \)
### 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:

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

- But this one is:

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

```scala
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)
Continuations

- A continuation is a function representing “the rest of the computation”
- Any function can be put in “continuation-passing form”
- for example

```scala
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 $\lambda r. k(n \times r)$
- “when done, multiply the result by $n$ and pass to $k$”
How does this work?

```
def fact3[A](n: Int, k: Int => A): A =
  if (n == 0) {k(1)} else {fact3(n-1, {r => k (n * r)})}
```

```
\[
\text{fact3}(3, \lambda x.x)
\]
\[\mapsto \text{fact3}(2, \lambda r_1. (\lambda x.x) (3 \times r_1))\]
\[\mapsto \text{fact3}(1, \lambda r_2. (\lambda r. (\lambda x.x) (3 \times r)) (2 \times r_2))\]
\[\mapsto \text{fact3}(0, \lambda r_3. (\lambda r_2. (\lambda r_1. (\lambda x.x) (3 \times r_1)) (2 \times r_2)) (1 \times r_3))\]
\[\mapsto (\lambda r_3. (\lambda r_2. (\lambda r_1. (\lambda x.x) (3 \times r_1)) (2 \times r_2)) (1 \times r_3)) 1\]
\[\mapsto (\lambda r_2. (\lambda r_1. (\lambda x.x) (3 \times r_1)) (2 \times r_2)) (1 \times 1)\]
\[\mapsto (\lambda r_1. (\lambda x.x) (3 \times r_1)) (2 \times 1)\]
\[\mapsto (\lambda x.x) (3 \times 2)\]
\[\mapsto 6\]
```
def eval[A](e: Expr, k: Value => A): A = e match {
  // Arithmetic
  case Num(n) => k(NumV(n))

  case Plus(e1,e2) =>
    eval(e1, {case NumV(v1) =>
      eval(e2, {case NumV(v2) => k(NumV(v1+v2))})})

  case Times(e1,e2) =>
    eval(e1, {case NumV(v1) =>
      eval(e2, {case NumV(v2) => k(NumV(v1*v2))})})

  ...
}
Interpreting $L_{if}$ using continuations

```scala
def eval[A](e: Expr, k: Value => A): A = e match {
  ...
  // Booleans
  case Bool(n) => k(BoolV(n))

  case Eq(e1,e2) =>
    eval(e1,{v1 =>
      eval(e2,{v2 => k(BoolV(v1 == v2))})})

  case IfThenElse(e,e1,e2) =>
    eval(e,{case BoolV(v) =>
      if(v) { eval(e1,k) } else { eval(e2,k) } })
  ...
}
```
def eval[A](e: Expr, k: Value => A): A = e match {
  ...
  // Let-binding
  case Let(e1,x,e2) => eval(e1,{v =>
      eval(subst(e2,v,x),k)})
  ...
}
def eval[A](e: Expr, k: Value => A): A = e match {
    ...
    // Functions
    case Lambda(x,ty,e) => k(LambdaV(x,ty,e))
    case Rec(f,x,ty1,ty2,e) => k(RecV(f,x,ty1,ty2,e))
    case Apply(e1,e2) =>
        eval(e1, {v1 =>
            eval(e2, {v2 => v1 match {
                case LambdaV(x,ty,e) => eval(subst(e,v2,x), k)
                case RecV(f,x,ty1,ty2,e) =>
                    eval(subst(subst(e,v2,x),v1,f),k)
            }}))
        ...
    }
}
Interpreting $L_{\text{Exn}}$ using continuations

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

```scala
def eval[A](e: Expr, h: Value => A, k: Value => A): A = e match {
  ...
  // Exceptions
  case Raise(e0) => eval(e0,h,h)
  case Handle(e1,x,e2) =>
    eval(e1,{v => eval(subst(e2,v,x),h,k)},k)
}
```

When raising an exception, we forget $k$ and pass to $h$. 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