Course Outline

0) Introduction and Course Overview (week 1)
1) Scheme Crash Course (week 1, 2)
2) Anatomy of a Language Definition (week 3)
3) Environments and Scope (week 4, 5, 6)
4) Continuations and Control-flow (week 7, 8)
5) Types and Type Checking (week 9, 10)
6) Object-Oriented Programming (week 11, 12)
7) Aspect-Oriented Programming (week 13)

4) Continuations and Control-Flow

Objectives
- Understand Continuation Passing Style (CPS)
  - why convert our interpreter into CPS?
  - what are continuations? (what do they model)
  - how do we convert our interpreter into CPS
- Use Continuations to model and understand
  - procedure call and return
  - tail recursion
  - control-flow related language constructs:
    - break, return, ...
    - exception handling
    - call-with-current continuation
  - Multi-threading (we will probably skip this)
  - A debugging environment (= Homework 4)

In the next couple of lectures we will do a lot of work to rebuild the ProcLiLa interpreter:

> (run "+(1,*(3,4))")
The program returns: 13
ok
> (run "letrec fac(n) = if =(n,0) then 1 else *(n,(fac -(n,1))) in
   (fac 4)"")
The program returns: 24.
ok

Q: Why would we go through so much trouble? Rebuilding what we already have? (What’s wrong with the interpreter from section 3?)
Why convert our interpreter into CPS?

Why doesn’t Scheme need loops?

What’s the difference between these two procedures?

(define fact
  (lambda (n)
    (if (= n 0) 1
      (* n (fact (- n 1)))))

(define fact-iter
  (lambda (n acc)
    (if (= n 0) acc
      (fact-iter (- n 1) (* n acc)))))

---

Why convert our interpreter into CPS?

Q: Why can’t we answer these questions with the chapter-3 interpreter?

==⇒ Because we “cheated”:

It is because DrScheme knows how to do recursion that our ProcLiLa interpreter works.

I.e: The execution context for a procedure to “go back” to is “implicitly” maintained by the Scheme interpreter’s “recursion stack”.

==⇒ This can’t really help us understand how recursion and procedure calls work.

==⇒ We need to explicitly model “what happens next”

That’s what continuations do!

They model “what happens next” after evaluating an expression

---

Objectives

– Understand Continuation Passing Style (CPS)

✓ why convert our interpreter into CPS?

• what are continuations? (what do they model)
• how do we convert our interpreter into CPS

– Use Continuations to model and understand

• procedure call and return
• tail recursion
• control-flow related language constructs:
  – exit, return, …
  – exception handling
• Multi-threading (we will probably skip this)
• A debugging environment (= Homework 4)
The “No Unbounded Context” Rule

We will rewrite the core of the chapter 3 interpreter…

To force us into dealing with execution context explicitly, we use the following rule:

If a procedure can lead to a unbounded amount of work
(if it leads to an unbounded recursion)

then we are only allowed to “tail-call” it.

The assumption is that tail calls are “ok”, i.e. they do not cause DrScheme to build execution context.

=> If we follow this rule, then we do not implicitly depend on DrScheme recursion to model execution state.

Note: this is true in Scheme, but it may not be true in other languages (e.g. Java) because of how they are implemented.

Continuations and Cont. Passing Style

Before we do this to our interpreter

first a simpler example. Let’s try to re-implement the program below to make it follow the rule on the previous slide (so we don’t rely on DrScheme recursion support)

```
(define count-symbols
  (lambda (sexp)
    (cond ((null? sexp) 0)
          ((pair? sexp) (+ (count-symbols (car sexp))
                          (count-symbols (cdr sexp))))
          ((symbol? sexp) 1)
          (else 0)))))
```

```
(define run (lambda (sexp)
   (let ((result (count-symbols sexp)))
  (display “num atoms:”) (display result) ’ok)))
```

Q: When we call “count-symbols”, what does DrScheme remember for us?
A: DrScheme remembers what work there is still to be done when a (Scheme) procedure returns.

If we are not to rely on DrScheme for this… we must do it ourselves!

=> We must create something that represents/remembers the work remaining to be done.

=> We call this a continuation and we add it as an extra argument to our continuation passing style procedures. In CPS we must explicitly construct this continuation argument when we call a CPS procedure, rather than rely on DrScheme recursion.

Let’s do this for the count-symbols example on the laptop now.
What are Continuations… summary

A continuation models

- The work remaining to be done after a given task completes.

What can you do (operations) with a continuation

- You can request the remaining work it represents to be carried out once the task-result it is waiting for is available:

  `(apply-cont <continuation> <value>)`

the result returned by the preceding task

the pending work, to be done after some task completes

The “No Unbounded Context” Rule

Now, let’s apply this idea to our interpreter.

Alternate formulation of the rule specifically for our interpreter:

The procedure `eval-exp` and any procedure which calls it, either directly or indirectly, must only be called in tail calls.

```
(display-value (eval-program ...))
(if (true-value? (eval-exp ...))
  ...

(cond ...
  (else (eval-exp ...)
    (let ((x (eval-exp ...))
      (eval-begin ...))))
```

Continuation Passing Interpreter

No unbounded context rule

⇒ Can’t do “real” recursion in `eval-exp` anymore only tail recursion!

How? => Add a continuation argument that tells the evaluator “what to do next” with the result of `eval-exp`

```
(define eval-exp
  (lambda (exp env cont)
    ...

  (define eval-exp
    (lambda (exp env cont)
      ...

  Important implications of the no unbounded context rule:

  Eval-exp should never return a result instead, it should call “apply-cont” to pass any “returned” value on to the continuation.

  Since eval-exp does not return a result, we should never ever write expressions that use the result of eval-exp

  (define eval-exp
    (lambda (exp env cont)
      ...

Objectives

- Understand Continuation Passing Style (CPS)
  ✔ why convert our interpreter into CPS?
  ✔ what are continuations? (what do they model)
  • how do we convert our interpreter into CPS

- Use Continuations to model and understand
  • procedure call and return
  • tail recursion
  • control-flow related language constructs:
    - exit, return, …
    - exception handling
  • Multi-threading (we will probably skip this)
  • A debugging environment (= Homework 4)
**Continuation Passing Interpreter**

Converting our Chapter 3 interpreter into Continuation Passing Style (CPS).

Q: Which procedures have to be converted into CPS?

A: We can use the following rules to determine which procedures need to be converter.

1. The procedure eval-exp must be converted into CPS
2. If a procedure \( P' \) calls a procedure \( P \), and \( P \) is to be converted into CPS than \( P' \) must also be converted into CPS.

Some procedures do not have to be converted

- extend-env, apply-env, ...

**We now take a copy of the section 3 interpreter and convert it**

---

**Continuation Passing Interpreter**

How to do CPS conversion from “ordinary” recursive code?

There are several different “scenarios”:

1) A CPS procedure wants to return a result
   ⇒ call apply-cont instead

2) A CPS procedure wants to call a non CPS procedure
   ⇒ just call it, no change needed

3) A CPS procedure wants to call another CPS procedure
   a) Original code has a “tail” call
      ⇒ just pass along current cont argument
   b) Original code has “real” recursive call
      ⇒ convert into a tail call, with a new continuation

---

**Continuation Passing Interpreter**

1) A CPS procedure wants to return a result
   ⇒ call apply-cont

**Example:**

```scheme
(define eval-exp
  (lambda (exp env cont)
    (cond ((lit-num-exp? exp)
      ;org: (lit-num-exp-datum exp))
      (apply-cont cont
        (lit-num-exp-datum exp))
      (apply-cont cont
        (lit-num-exp-datum exp))
      ...
```

**Continuation Passing Interpreter**

2) A CPS procedure wants to call a non CPS procedure
   ⇒ just call it, no change needed

**Example:**

```scheme
((var-ref-exp? exp)
  (apply-cont cont
    (deref (apply-env-ref env
      (var-ref-exp-id exp))))))
```

Procedure **apply-env-ref** and **deref** are not converted and we can simply call them like before.
Continuation Passing Interpreter

3) A CPS procedure wants to call another CPS procedure
   
a) Original code has a “tail” call
      ⇒ just pass along current cont

(define eval-exp
  (lambda (exp env cont)
    ...
    ((letrec-exp? exp)
      (let ((ext-env (extend-env-loopy (letrec-exp-ids exp)
                                    (letrec-exp-idss exp)
                                    (letrec-exp-bodies exp)
                                    env)))
         (eval-exp (letrec-exp-body exp) ext-env cont))))

Continuation Passing Interpreter

Separate into what happens before and after the recursive call.

((if-exp? exp)
  (if (true-value?
       (eval-exp (if-exp-test-exp exp) env))
       (eval-exp (if-exp-then-exp exp) env)
       (eval-exp (if-exp-else-exp exp) env)))

Turn the “after” part into a new continuation

((if-exp? exp)
  (if (true-value?
       (eval-exp (if-exp-test-exp exp) env))
       (eval-exp (if-exp-then-exp exp) env)
       (eval-exp (if-exp-else-exp exp) env)))

...must include whatever is needed to apply the continuation...))

4) Continuations and Control-Flow
Continuation Passing Interpreter

Turn the “after” part into a new continuation.

This is the “after code” (what happens with the result value):

```
(if (true-value? value)
  (eval-exp (if-exp-then-exp exp) env)
  (eval-exp (if-exp-else-exp exp) env))
```

The “after code” is converted and goes into apply-cont:

```
((test-cont? cont)
  (let ((then-exp (test-cont-then-exp cont))
      (else-exp (test-cont-else-exp cont))
      (env (test-cont-env env))
      (nested-cont (nested-cont-cont cont)))
   (if (true-value? value)
       (eval-exp then-exp env nested-cont)
       (eval-exp else-exp env nested-cont)))
```

Continuation Passing Interpreter

Turn the “after” part into a new continuation.

We also have to define a struct for representing the continuation

```
(define-struct continuation ()
  (define-struct (halt-cont continuation) ()
    (define-struct (nested-cont continuation) (cont))
    (define-struct (test-cont nested-cont) (then-exp else-exp env))

    (define-struct (prim-cont nested-cont) (prim))
```

Prim App Exps

1) In eval-exp

```
((prim-app-exp? exp)
  (eval-rands (prim-app-exp-rands exp)
   env
   (make-prim-cont cont (prim-app-exp-prim exp))))
```

2) In apply cont

```
((prim-cont? cont)
  (apply-cont (nested-cont-cont cont)
   (apply-primitive (prim-cont-prim cont) value)))
```

3) Struct to represent the cont

```
(define-struct (prim-cont nested-cont) (prim))
```

Eval-rands -> CPS

```
(define eval-rands ;; original lifted from eval-exp branch for prim-app-exp
  (lambda (rands env)
    (map (lambda (rand) (eval-exp rand env
      nested-cont)
     (else)
      (cons (eval-exp (car rands) env)
        (eval-rands (cdr rands env)))))
```

Can’t convert this… map is in the way => get rid of map first

```
(define eval-rands ;; non-CPS, but without map
  (lambda (rands env)
    (cond ((null? rands)
      ()
      (else
        (cons (eval-exp (car rands) env)
          (eval-rands (cdr rands env)))))))
```
Eval-rands -> CPS

(define eval-rands
  (lambda (rands env cont)
    (cond ((null? rands)
           (apply-cont cont '()))
          (else
           (eval-exp (car rands) env
                     (make-rest-rands-cont cont (cdr rands) env))))))

Continuation that will …
evaluate the (cdr rands) and
cons the result of evaluating (car rands) to the front

Eval-rands -> CPS

(define-struct (rest-rands-cont nested-cont) (rest-rands env))
(define-struct (cons-arg-cont nested-cont) (arg))

In apply-cont:
((rest-rands-cont? cont)
 (eval-rands (rest-rands-cont-rest-rands cont)
             (rest-rands-cont-env cont)
             (make-cons-arg-cont (nested-cont-cont cont) value)))

continuation that remembers to cons the first
argument to the front after eval of (cdr rands)
value of evaluating (car rands)

Eval of proc-app-exp -> CPS

((proc-app-exp? exp)
 ; original (non CPS code:)
 ; (apply-procedure
 ;  (eval-exp (proc-app-exp-rator exp) env)
 ;  (map (lambda (rand)
 ;         (eval-exp rand env))
 ;  (proc-app-exp-rands exp))))

In eval-exp:

Q1: Will we have to convert apply-procedure into CPS?
Q2: Why? Or why not?
Eval of proc-app-exp -> CPS

In eval-exp:

\[
\begin{aligned}
&((\text{proc-app-exp? } \text{exp}) \\
&; \text{orginal (non CPS code:)} \\
&; (\text{apply-procedure} \\
&; \quad (\text{eval-exp} (\text{proc-app-exp-rator exp}) \text{env}) \\
&; \quad (\text{eval-rands} (\text{proc-app-exp-rands exp}) \text{env})))
\end{aligned}
\]

Q1: Will we have to convert apply-procedure into CPS?
Q2: Why? Or why not?

Eval of proc-app-exp -> CPS

\[
\begin{aligned}
&\text{(define-struct (apply-procedure-cont nested-cont) (proc))}
\end{aligned}
\]

In apply-cont:

\[
\begin{aligned}
&((\text{apply-procedure-cont? cont}) \\
&\quad (\text{apply-procedure (apply-procedure-cont-proc cont}) \\
&\quad (\text{apply-procedure-cont-proc cont})) \\
&\quad \text{value} \\
&\quad (\text{nested-cont-cont cont})))
\end{aligned}
\]

apply-procedure converted to CPS

Eval of proc-app-exp -> CPS

\[
\begin{aligned}
&\text{(define apply-procedure} \\
&\quad (\lambda (\text{proc args cont}) \\
&\quad \quad \text{if (not (procedure-value? proc)} \\
&\quad \quad \quad (\text{error 'apply-procedure "Not a procedure ~s" proc}) \\
&\quad \quad \quad (\text{let ((new-env (extend-env (procedure-value-ids proc) \\
&\quad \quad \quad \quad \text{args} \\
&\quad \quad \quad \quad (\text{procedure-value-env proc}))) \\
&\quad \quad \quad (\text{eval-exp (procedure-value-body proc new-env cont})))))))
\end{aligned}
\]

This continuation remembers…
when the procedure returns…
what happens next
Now that we have done all that hard work, what’s the point?

=> We can use this new model to understand how continuations grow and shrink, how does recursion work, tail recursion etc.

Let’s have some fun and play with the finished interpreter.

It has a “tracing” feature that prints out the current continuation.

Let’s examine the output and see how we can read those “stack trace” dumps and see how these continuations determine “what to do next” at every step in the interpretation process.

How do procedures “go back”? Let’s see if we can use this new model to understand how control context really grows and shrinks.

Since we have a running model, with a lot of tracing output, we can use this as a tool to see these things in action.

So to understand this… let’s combine
a) closely examining relevant parts of the Scheme code of our model
b) examining trace output from the running model for well chosen examples.

4) Continuations and Control-Flow

Objectives

✓ Understand Continuation Passing Style (CPS)
✓ why convert our interpreter into CPS?
✓ what are continuations? (what do they model)
✓ how do we convert our interpreter into CPS
– Use Continuations to model and understand
  • procedure call and return
  • tail recursion
    • control-flow related language constructs:
      – exit, return, …
      – exception handling
    • Multi-threading (we will probably skip this)
    • A debugging environment (= Homework 4)

How do procedures “go back”? We’ll work on answering these questions together

– When a procedure is called, where/how in the interpreter does it remember where to return?
– When a procedure is called does DrScheme garbage collect the caller env?
– How does tail recursion differ from real recursion?

So let’s try to answer these questions now, playing with the interpreter on the laptop.
4) Continuations and Control-Flow

Objectives

- Understand Continuation Passing Style (CPS)
  - why convert our interpreter into CPS?
  - what are continuations? (what do they model)
  - how do we convert our interpreter into CPS
- Use Continuations to model and understand
  - procedure call and return
  - tail recursion
  - control-flow related language constructs:
    - exit, return, ...
    - exception handling
- Multi-threading (we will probably skip this)
- A debugging environment (= Homework 4)

Using the Power of Continuations

So, now we have an interpreter that has an explicit model of “what to do next” in the form of continuations.

=> Can model/implement language features and constructs that affect control flow.

Examples:

- procedure call and return
- tail recursion
- control-flow related language constructs:
  - exit, return, ...
  - exception handling

We’ll do one of these together now in lecture

Try do the other one by yourself as an exercise

One armed if as a “Parser Hack”

Q: Parser hack? What do you mean?
A: We treat a one-armed if as “syntactic sugar”. The parser replaces it with…

Q: What’s the advantage of such an implementation?

Q: Are there possible dis-advantages?
4) Continuations and Control-Flow

Return

Return ::= "return" <exp> [ return-exp (exp) ]

> (run "let compare = proc (x,y) begin
  if =(x,y) then return 0;
  if <(x,y) then return 1;
  return -(1)
end
in (compare 2 3)"

The program returns: 1.

Note: the example also uses a "one-armed" if. We’ve already implemented this as a “parser hack”.

4) Continuations and Control-Flow

Exception Handling

Objectives: to model and understand...

– Motivate = understand why add exception handling to a language?
– Model and understand:
  • How do exceptions work (by implementing in CPS interpreter)
  • How do try-catch/throw and similar constructs like proc/return, loop/exit relate to, or differ from each other?
  • Examine subtle interactions between language features
    – What happens if we have: throw | exit | return within a throw | exit | return?
    – how is tail recursion affected by use of return and try-catch?

Exception Handling: Why?

Q: Why (or why not!) would we add an exception handling mechanisms to a language?

When we add features to a language we must know why!
• What does it cost? (nothing is for free!)
• Benefits? Does it help make programs better? (Easier to write/read/understand/change/debug/…)

We have to make a judgment on whether the cost is worth the price!

This decision is subjective: trade-off something for something => a great topic for endless debates :-)"

Note: This kind of reasoning applies to any language feature, not just exception handling!
Exception Handling: Why?

example: to help us think about this...

```
letrec mul(lon) = if null?(lon) then 1
else let fst = car(lon)
  in if number?(fst)
    then *(fst, (mul cdr(lon)))
    else throw 'Bad element'
in try (mul list(1,2,4,'abc',5,6,11,23,4))
catch (e) begin
  printString("Something went wrong: ");
  print(e);
  newline()
end
```

Q: Can we write this program without using exceptions?
A: Yes of course... (Turing complete, remember :-)

Q: Which version do you like better? Why?

This could be even better:

```
letrec mul(lon) = if null?(lon) then 1
else let fst = car(lon)
  in if number?(fst)
    then *(fst, (mul cdr(lon)))
    else throw 'Bad element'
in try (mul list(1,2,4,'abc',5,6,11,23,4))
catch (e) begin
  printString("Something went wrong: ");
  print(e);
  newline()
end
```

if primitives throw exceptions when given bad arguments

Q: Why (or why not!) would we add an exception handling mechanism to a language? Why?

When we add features to a language we must know why!
• What does it cost? (nothing is for free!)
• Benefits? Does it help make programs better? (Easier to write/read/understand/change/debug/…)

We have to make a judgment on whether the cost is worth the price!

This decision is subjective: trade-off something for something => a great topic for endless debates :-)

Note: This kind of reasoning applies to any language feature, not just exception handling!

Q: Can we write this program without using exceptions?
A: Yes of course... (Turing complete, remember :-)

```
letrec mul(lon) = if null?(lon) then 1
else let fst = car(lon)
  in if number?(fst)
    then *(fst, (mul cdr(lon)))
    else throw 'Bad element'
in try (mul list(1,2,4,'abc',5,6,11,23,4))
catch (e) begin
  printString("Something went wrong: ");
  print(e);
  newline()
end
```

```
letrec mul(lon) = if null?(lon) then 1
else let fst = car(lon)
  in if number?(fst)
    then *(fst, (mul cdr(lon)))
    else throw 'Bad element'
in try (mul list(1,2,4,'abc',5,6,11,23,4))
catch (e) begin
  printString("Something went wrong: ");
  print(e);
  newline()
end
```

The program returns: true.ok
### Adding Exception support to ProcLiLa

```scheme
<exp> ::= "try" <exp> "catch" "(" <identifier> ")" <exp> 
    [ try-catch-exp (try-exp id catch-exp) ]

 ::= "throw" <exp> 
    [throw-exp (exp)]
```

We'll do this now on the laptop.

---

### Exception Handling

Objectives: to model and understand...

- Motivate = understand why add exception handling to a language?
  - How do exceptions work (by implementing in CPS interpreter)
    - How do `try-catch/throw` and similar constructs like `proc/return, loop/exit` relate to, or differ from each other?
    - Examine subtle interactions between language features
      - What happens if we have: `throw | exit | return` within a `throw | exit | return`?
      - how is tail recursion affected by use of return and try-catch?

---

Cost of Language Features

Questions to think about

- Will it make programs slower? (perhaps even if I don’t use the feature?)
- Make it harder to read or write programs?
- Make it harder to implement the language (compile or interpret).
- Make it harder to analyze programs (with meta programs)?
- Is the language already in use?
- Make it harder to read or write programs?
- How do exceptions affect other constructs, in particular loops?
- Does it make the language harder to learn?
- What happens to existing tools (debugger, code browser, code assist, …)
- How does exceptions affect code size and performance?
- How does exceptions affect implementation?
- How does exceptions affect performance?
- How does exceptions affect the ease of learning the language?
4) Continuations and Control-Flow

try-catch/throw <--?--> proc/return

These two features have a lot in common!

1. At one point a continuation gets selected/marked.
2. Sometime later “deeper in the continuation stack” an expression may just cause the interpreter skip over a bunch of continuations and go directly to the selected cont.

Q: So what’s the difference between them then?
Hint: Think of scoping mechanisms we studied in section 3

Proc/Return

Q: So what’s the difference between them then?
Hint: Think of scoping mechanisms we studied in section 3

(define apply-procedure
  (lambda (proc args cont)
    (if (not (procedure-value? proc))
      (error 'apply-procedure "Not a procedure" proc)
      (let ((new-env (extend-env (cons '%return-cont% (procedure-value-ids proc))
                                  (cons cont args)
                                  (procedure-value-env proc))))
        (eval-exp (procedure-value-body proc) new-env cont))))

In eval-exp
  ((return-exp? exp)
    (eval-exp (return-exp-exp exp) env (apply-env env '%return-cont%)))

try-catch/throw

Q: So what’s the difference between them then?
Hint: Think of scoping mechanisms we studied in section 3

In eval-exp:
  ((try-catch-exp? exp)
    (eval-exp (try-catch-exp-try-exp exp) env
      (make-catch-cont cont
        (try-catch-exp-id exp)
        (try-catch-exp-catch-exp exp) env))))

What is the “scope” of this … if this is what is used to find it when throw occurs?

(define find-catch-cont
  (lambda (cont)
    (cond ((catch-cont? cont) cont)
          ((nested-cont? cont) (find-catch-cont (nested-cont-cont cont)))
          (else (error "uncaugth exception ...)))))

Answer:

For try-catch/throw:
The catch-cont, which is “bound” by try-catch is accessible to throw-exps within the dynamic scope of the try-exp (i.e. inside the try-exp itself and inside any procedure that is (transitively) called from within the try-exp.

For proc/return:
The continuation for the procedure application (proc-app-exp) is accessible to return-exps within the body of the procedure like a statically scoped variable.

4) Continuations and Control-Flow

try-catch/throw <--?--> proc/return

Q: So what’s the difference between them then?
Hint: Think of scoping mechanisms we studied in section 3.

Answer:

For try-catch/throw:
The catch-cont, which is “bound” by try-catch is accessible to throw-exps within the dynamic scope of the try-exp (i.e. inside the try-exp itself and inside any procedure that is (transitively) called from within the try-exp.

For proc/return:
The continuation for the procedure application (proc-app-exp) is accessible to return-exps within the body of the procedure like a statically scoped variable.
**Exception Handling**

Objectives: to model and understand...

- Motivate = understand why add exception handling to a language?
- Model and understand:
  - How do exceptions work (by implementing in CPS interpreter)
  - How do try-catch/throw and similar constructs like proc/return, loop/exit relate to, or differ from each other?

  - Examine subtle interactions between language features
    - What happens if we have: throw | exit | return within a throw | exit | return?
    - how is tail recursion affected by use of return and try-catch?

**Throw | Exit | Return within a Throw | Exit | Return**

Language feature may interact (in undesirable ways!)

- How many interactions to consider? A: \(3 \times 3 = 9\)
- We can determine how these features interact by examining our model (i.e. analyze the Scheme code and/or run examples)
- We will study throw within return because there is a subtle interaction problem (in our model).

**Q1:** Can we discover this problem in the Scheme code of the model?

**Q2:** Construct an example that demonstrates it

**Q3:** How could we fix this problem?

**Exercise:** analyze (some of) the other interactions on your own.

**A throw within a return?**

**Example 1:** (as simple as possible)

```
> (run "let foo = proc ()
    try return throw 'bad'
    catch (e) e
    in (foo)")
ERROR: find-catch-cont: An uncaught exception occurred
```

In this example, because of how we implemented return the throw gets executed as if it is outside of the procedure (and therefore outside of the try-catch!)

**Note:** the remaining slides represent some fairly advanced and tricky analysis of interactions between language features like return, try-catch/throw and tail recursion.

They were not covered in depth in lecture. You are not expected to understand them in depth.

You can work through these slides as an exercise to test how well you really understand the materials in this section of the course.

Remember from these slides at least the following:

- tricky interactions between features can and do occur!
- => this can result in subtle bugs in language design and implementation
A throw within a return?

Example 2: (similar but more elaborate)

```lisp
> (run "letrec mul(lon) = 
  if null?(lon) 
  then 1 
  else let fst = car(lon) 
  in if number?(fst) 
  then *(fst, (mul cdr(lon))) 
  else throw 'Bad element' 
foo() = try return (mul list(1,2,4,'abc',5,6,11,23,4)) 
catch (e) 0 
in (foo")
```

ERROR: find-catch-cont: An uncaught exception occurred

A throw within a return?

Q: Can you fix the problem?

A: Yes, of course. Two possible ways to fix (or more perhaps :-)

1. Make return work more like throw: wait to switch to the
   %return-cont% until after return-exp-exp is evaluated.
   or

2. Make the try-catch-exp patch the %return-exp% (if it is
   defined) by adding a catch-cont to it as well. Thus it is
   ensured that the catch block will not be lost by return.

This is left as an exercise for the smart and eager!

Note: Both solutions will work, but one of them is nicer with
respect to keeping tail-recursion. (see also next slide!)

A throw within a return?

Q: So what's the problem here? Let's examine the trace output!

```lisp
> (run "let foo = proc () try return throw 'bad' catch (e) e in (foo")
```

ERROR: find-catch-cont: An uncaught exception occurred

Exception Handling

Objectives: to model and understand...

- Motivate = understand why add exception handling to a
  language?
  - Model and understand:
    - How do exceptions work (by implementing in CPS interpreter)
    - How do try-catch/throw and similar constructs like
      proc/return, loop/exit relate to, or differ from each other?
      - Examine subtle interactions between language features
      - What happens if we have:
        throw | exit | return within a throw | exit | return?
        - how is tail recursion affected by use of return and try-catch?
**4) Continuations and Control-Flow**

---

**return and tail recursion**

These two programs are equivalent. Therefore, it is reasonable to expect that they both behave as tail recursive processes.

**Exercise:** Verify this by analyzing:

- a) trace output
- b) the interpreter code.

**note:** how you did the exercise on slide 66 affects this!

---

**try-catch and tail recursion**

Q: After introducing a try-catch-block above, does the program still behave as a tail recursive process?

**Exercise:**

a) Answer the question above by thinking about and analyzing the implementation of try-catch.

b) verify your answer by relating it to the trace output.

---

**Exception Handling**

**Objectives:** to model and understand...

- ✔ Motivate = understand why add exception handling to a language?
- ✔ Model and understand:
  - ✔ How do exceptions work (by implementing in CPS interpreter)
  - ✔ How do try-catch/throw and similar constructs like proc/return, loop/exit relate to, or differ from each other?
  - ✔ Examine subtle interactions between language features
    - ✔ What happens if we have: throw | exit | return within a throw | exit | return?
    - ✔ how is tail recursion affected by use of return and try-catch?

---

**4) Continuations and Control-Flow**

---

**Objectives**

- ✔ Understand Continuation Passing Style (CPS)
  - ✔ why convert our interpreter into CPS?
  - ✔ what are continuations? (what do they model)
  - ✔ how do we convert our interpreter into CPS
- Use Continuations to model and understand
  - ✔ procedure call and return
  - ✔ tail recursion
  - ✔ control-flow related language constructs:
    - ✔ exit, return, ...
    - ✔ exception handling
- ✗ Multi-threading (we will probably skip this) **skipped**

- ✔ A debugging environment (= Homework 4)

   **This is now up to you!**