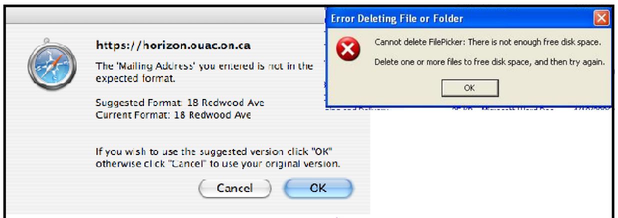


https://horizon.ouac.on.ca

The 'Mailing Address' you entered is not in the expected format.

Suggested Format: 18 Redwood Ave
Current Format: 18 Redwood Ave

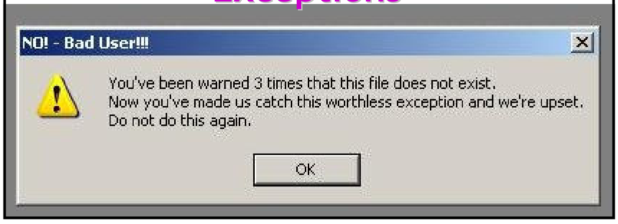
If you wish to use the suggested version click "OK" otherwise click "Cancel" to use your original version.

Error Deleting File or Folder

Cannot delete FilePicker: There is not enough free disk space.

Delete one or more files to free disk space, and then try again.

Exceptions



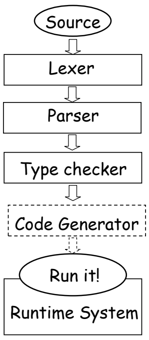
NO! - Bad User!!!

You've been warned 3 times that this file does not exist.
Now you've made us catch this worthless exception and we're upset.
Do not do this again.

One-Slide Summary

- Real-world programs must have **error-handling** code. Errors can be handled where they are detected or the error can be **propagated** to a caller.
- Passing special error return codes is itself error-prone.
- Exceptions are a formal and automated way of reporting and handling errors. Exceptions can be implemented efficiently and described formally.

Language System Structure



- We looked at each stage in turn
- A new language feature affects many stages
- We will add exceptions

Lecture Summary

- Why exceptions ?
- Syntax and informal semantics
- Semantic analysis (i.e. type checking rules)
- Operational semantics
- Code generation
- Runtime system support

Exceptional Motivation

- “Classroom” programs are written with optimistic assumptions
- Real-world programs must consider “exceptional” situations:
  - Resource exhaustion (disk full, out of memory, network packet collision, …)
  - Invalid input
  - Errors in the program (null pointer dereference)
- It is usual for code to contain 1-5% error handling code (figures for modern Java open source code)
  - With 3-46% of the program text transitively reachable

Approaches To Error Handling

Two ways of dealing with errors:

1. Handle them where you detect them
   - e.g., null pointer dereference → stop execution
2. Let the caller handle the errors:
   - The caller has more contextual information
     e.g. an error when opening a file:
     a) In the context of opening /etc/passwd
     b) In the context of opening a log file
   - But we must tell the caller about the error!

Error Return Codes

- The callee can signal the error by returning a special return value or **error code**:
  - Must not be one of the valid inputs
  - Must be agreed upon beforehand (i.e., in API)
- The caller promises to check the error return and either:
  - Correct the error, or
  - Pass it on to its own caller

Error Return Codes

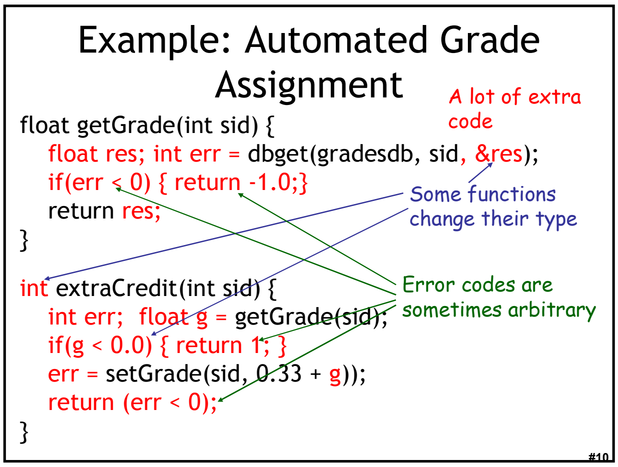
- It is sometimes hard to select return codes
  - What is a good error code for:
    - divide(num: Double, denom: Double) : Double { … }
- How many of you always check errors for:
  - malloc(int) ?
  - open(char *) ?
  - close(int) ?
  - time(struct time_t *) ?
- Easy to forget to check error return codes

Example:
Automated Grade Assignment

```
float getGrade(int sid) {   return dbget(gradesdb, sid); }

void setGrade(int sid, float grade) {   dbset(gradesdb, sid,
   grade); }

void extraCredit(int sid) {
    setGrade(sid, 0.33 + getGrade(sid));
    }

void grade_inflator() {
    while(gpa() < 3.0) {  extraCredit(random());  }
    }
```

- What errors are we ignoring here?

Example: Automated Grade Assignment



```
float getGrade(int sid) {
   float res; int err = dbget(gradesdb, sid, &res);
   if(err < 0) { return -1.0;}
   return res;
}

int extraCredit(int sid) {
   int err;  float g = getGrade(sid);
   if(g < 0.0) { return 1; }
   err = setGrade(sid, 0.33 + g));
   return (err < 0);
}
```

A lot of extra code

Some functions change their type

Error codes are sometimes arbitrary

Exceptions

- **Exceptions** are a language mechanism designed to allow:
  - Deferral of error handling to a caller
  - Without (explicit) error codes
  - And without (explicit) error return code checking

Adding Exceptions to Cool

- We extend the language of expressions:

  e ::= throw e | try e catch x : T ⇒ e’

- (Informal) semantics of throw e
  - Signals an exception
  - Interrupts the current evaluation and searches for an exception handler up the activation tree
  - The value of e is an exception parameter and can be used to communicate details about the exception

Adding Exceptions to Cool

(Informal) semantics of try e catch x : T ⇒ e_1

1. e is evaluated first
2. If e's evaluation terminates normally with v

   then v is the result of the entire expression

   Else (e's evaluation terminates exceptionally)

   If the exception parameter is of type $\leq$ T then
   - Evaluate e_1 with x bound to the exception parameter
   - The (normal or exceptional) result of ev
   - aluating e_1 becomes the result of the entire expression

   Else
   - The entire expression terminates exceptionally

Example: Automated Grade Assignment

```
float getGrade(int sid) {   return dbget(gradesdb, sid); }

void setGrade(int sid, float grade) {
   if(grade < 0.0 || grade > 4.0) { throw (new NaG); }
  dbset(gradesdb, sid, grade); }

void extraCredit(int sid) {
    setGrade(sid, 0.33 + getGrade(sid)) }

void grade_inflator() {
    while(gpa < 3.0) {
       try extraCredit(random())
       catch x : Object ⇒ print “Nice try! Don’t give up.\n”; }
    }
```

Example Notes

- Only error handling code remains
- But no error propagation code
  - The compiler handles the error propagation
  - No way to forget about it
  - And also much more efficient (we'll see)
- Two kinds of evaluation outcomes:
  - Normal return (with a return value)
  - Exceptional "return" (with an exception parameter)
  - No way to get confused which is which

Overview

- ✓ Why exceptions ?
- ✓ Syntax and informal semantics
- Semantic analysis (i.e. type checking rules)
- Operational semantics
- Code generation
- Runtime system support

Typing Exceptions

- We must extend the Cool typing judgment

$$O, M, C \vdash e : T$$

  - Type T refers to the normal return!

- We'll start with the rule for try:
  - Parameter "x" is bound in the catch expression
  - try is like a conditional

$$\frac{O, M, C \vdash e : T_1 \qquad O[T/x], M, C \vdash e' : T_2}{O, M, C \vdash \text{try } e \text{ catch } x : T \Rightarrow e' : T_1 \sqcup T_2}$$

Typing Exceptions

- What is the type of "throw e" ?
- The type of an expression:
  - Is a description of the possible return values, and
  - Is used to decide in what contexts we can use the expression
- "throw" does not return to its immediate context but directly to the exception handler!
- The same "throw e" is valid in any context:

  if throw e then (throw e) + 1 else (throw e).foo()

- As if "throw e" has *any type*!

Typing Exceptions

$$\frac{O, M, C \vdash e : T_1}{O, M, C \vdash \text{throw } e : T_2}$$

- As long as "e" is well typed, "throw e" is well typed with *any type needed* in the context
- This is convenient because we want to be able to signal errors from any context

Overview

✓ Why exceptions ?

✓ Syntax and informal semantics

✓ Semantic analysis (i.e. type checking rules)

- Operational semantics
- Code generation
- Runtime system support

Operational Semantics of Exceptions

- Several ways to model the behavior of exceptions
- A **generalized value** is
  - Either a normal termination value, or
  - An exception with a parameter value

$$g ::= \text{Norm}(v) \mid \text{Exc}(v)$$

- Thus given a generalized value we can:
  - Tell if it is normal or exceptional return, and
  - Extract the return value or the exception parameter

Operational Semantics of Exceptions (1)

- The existing rules are modified to use Norm(v) :

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : Norm(Int(n_1)), S_1 \\ so, E, S_1 \vdash e_2 : Norm(Int(n_2)), S_2 \end{array}}{so, E, S \vdash e_1 + e_2 : Norm(Int(n_1 + n_2)), S_2}$$

$$\frac{\begin{array}{c} E(id) = l_{id} \\ S(l_{id}) = v \end{array}}{so, E, S \vdash id : Norm(v), S}$$

$$\frac{}{so, E, S \vdash self : Norm(so), S}$$

Operational Semantics of Exceptions (2)

- “throw” returns exceptionally:

$$\frac{so, E , S \vdash e : v, S_1}{so, E, S \vdash throw\ e : Exc(v), S_1}$$

- The rule above is *not well formed*! Why?

Operational Semantics of Exceptions (3)

- “throw e” returns exceptionally:

$$\frac{so, E , S \vdash e : Norm(v), S_1}{so, E, S \vdash throw\ e : Exc(v), S_1}$$

- What if the evaluation of e itself throws an exception?
  - E.g. “throw (1 + (throw 2))” is like “throw 2”
  - Formally:

$$\frac{so, E , S \vdash e : Exc(v), S_1}{so, E, S \vdash throw\ e : Exc(v), S_1}$$

Operational Semantics of Exceptions (4)

- All existing rules are changed to propagate the exception:

$$\frac{\text{so}, E, S \vdash e_1 : \text{Exc}(v), S_1}{\text{so}, E, S \vdash e_1 + e_2 : \text{Exc}(v), S_1}$$

  - Note: the evaluation of e_2 is aborted

$$\frac{\begin{array}{c}\text{so}, E, S \vdash e_1 : \text{Norm}(\text{Int}(n_1)), S_1 \\ \text{so}, E, S_1 \vdash e_2 : \text{Exc}(v), S_2\end{array}}{\text{so}, E, S \vdash e_1 + e_2 : \text{Exc}(v), S_2}$$

Operational Semantics of Exceptions (5)

- The rules for “try” expressions:
  - Multiple rules (just like for a conditional)

$$\frac{\text{so}, E, S \vdash e : \text{Norm}(v), S_1}{\text{so}, E, S \vdash \text{try } e \text{ catch } x : T \Rightarrow e' : \text{Norm}(v), S_1}$$

  - What if e terminates exceptionally?
    - We must check whether it terminates with an exception parameter of type T or not

Operational Semantics for Exceptions (6)

- If e does not throw the expected exception

$$\frac{\begin{array}{l}\text{so}, E, S \vdash e : \text{Exc}(v), S_1 \\ v = X(\ldots) \\ \text{not } (X \leq T)\end{array}}{\text{so}, E, S \vdash \text{try } e \text{ catch } x : T \Rightarrow e' : \text{Exc}(v), S_1}$$

- If e does throw the expected exception

$$\frac{\begin{array}{l}\text{so}, E, S \vdash e : \text{Exc}(v), S_1 \\ v = X(\ldots) \\ X \leq T \\ l_{new} = \text{newloc}(S_1) \\ \text{so}, E[l_{new}/x]\ , S_1[v/l_{new}] \vdash e' : g, S_2\end{array}}{\text{so}, E, S \vdash \text{try } e \text{ catch } x : T \Rightarrow e' : g, S_2}$$

Operational Semantics of Exceptions. Notes

- Our semantics is precise
- But is not very clean
  - It has two or more versions of each original rule
- It is not a good recipe for implementation
  - It models exceptions as "compiler-inserted propagation of error return codes"
  - There are much better ways of implementing exceptions
- There are other semantics that are cleaner and model better implementations

Overview

✓Why exceptions ?

✓Syntax and informal semantics

✓Semantic analysis (i.e. type checking rules)

✓Operational semantics

- Code generation

- Runtime system support

Code Generation for Exceptions

- One method is suggested by the operational semantics
- Simple to implement
- But not very good
  - We pay a cost at each call/return (i.e. often)
  - Even though exceptions are rare (i.e. exceptional)
- A good engineering principle:
  - Don't pay often for something that you use rarely!
    - *What is Amdahl's Law?*
  - Optimize the common case!

Long Jumps

- A long jump is a non-local goto:
  - In one shot you can jump back to a function in the caller chain (bypassing many intermediate frames)
  - A long jump can "return" from many frames at once
- Long jumps are a commonly used implementation scheme for exceptions
  - Take a compilers class for details
- Disadvantage:
  - (Minor) performance penalty at each try

Implementing Exceptions with Tables (1)

- We do not want to pay for exceptions when executing a "try"
  - Only when executing a "throw"

```
cgen(try e catch e') =
    cgen(e)                  ; Code for the try block
    goto end_try
L_catch:
    cgen(e')                 ; Code for the catch block
end_try:
     …
cgen(throw) =
    jr runtime_throw         ; <- this is the trick!
```

Implementing Exceptions with Tables (2)

- The normal execution proceeds at full speed
- When a throw is executed we use a runtime function that finds the right catch block
- For this to be possible the compiler produces a table saying for each catch block to which instructions it corresponds

Implementing Exceptions with Tables. Notes

- runtime_throw looks at the table and figures which catch handler to invoke

- Advantage:
  - No cost, except if an exception is thrown
- Disadvantage:
  - Tables take space (even 30% of binary size)
  - But at least they can be placed out of the way

- Java Virtual Machine uses this scheme

---

try … finally …

- Another exception-related construct:

  try e_1 finally e_2

  - After the evaluation of e_1 terminates (either normally <u>or</u> exceptionally) it evaluates e_2
  - The whole expression then terminates like e_1
- Used for cleanup code:

```
try
    f = fopen("treasure.directions", "w");
    … compute … fprintf(f, "Go %d paces to the west", paces); …
finally
    fclose(f)
```

Try-Finally Semantics

- Typing rule:

$$\frac{O, M, C \vdash e_1 : T_1 \qquad O, M, C \vdash e_2 : T_2}{O, M, C \vdash \text{try } e_1 \text{ finally } e_2 : T_2}$$

- Operational semantics:

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : \text{Norm}(v), S_1 \\ so, E, S_1 \vdash e_2 : \mathbf{g}, S_2 \end{array}}{so, E, S \vdash \text{try } e_1 \text{ finally } e_2 : \mathbf{g}, S_2}$$

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : \mathbf{Exc(v_1)}, S_1 \\ so, E, S_1 \vdash e_2 : \text{Norm}(v_2), S_2 \end{array}}{so, E, S \vdash \text{try } e_1 \text{ finally } e_2 : \mathbf{Exc(v_1)}, S_2}$$

Psycho Corner Case

- Operational Semantics

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : \mathbf{Exc(v_1)}, S_1 \\ so, E, S_1 \vdash e_2 : \mathbf{Exc(v_2)}, S_2 \end{array}}{so, E, S \vdash \text{try } e_1 \text{ finally } e_2 : \mathbf{???}, S_2}$$

- Difficulty in understanding try-finally is one reason why Java programmers tend to make at least 200 exception handling mistakes per million lines of code

14.20.2 Execution of try-catch-finally

- A try statement with a finally block is executed by first executing the try block. Then there is a choice:
- If execution of the try block completes normally, then the finally block is executed, and then there is a choice:
  - If the finally block completes normally, then the try statement completes normally.
  - If the finally block completes abruptly for reason *S*, then the try statement completes abruptly for reason *S*.
- If execution of the try block completes abruptly because of a throw of a value *V*, then there is a choice:
  - If the run-time type of *V* is assignable to the parameter of any catch clause of the try statement, then the first (leftmost) such catch clause is selected. The value *V* is assigned to the parameter of the selected catch clause, and the *Block* of that catch clause is executed. Then there is a choice:
    - If the catch block completes normally, then the finally block is executed. Then there is a choice:
      - If the finally block completes normally, then the try statement completes normally.
      - If the finally block completes abruptly for any reason, then the try statement completes abruptly for the same reason.
    - If the catch block completes abruptly for reason *R*, then the finally block is executed. Then there is a choice:
      - If the finally block completes normally, then the try statement completes abruptly for reason *R*.
      - If the finally block completes abruptly for reason *S*, then the try statement completes abruptly for reason *S* (and reason *R* is discarded).
  - If the run-time type of *V* is not assignable to the parameter of any catch clause of the try statement, then the finally block is executed. Then there is a choice:
    - If the finally block completes normally, then the try statement completes abruptly because of a throw of the value *V*.
    - If the finally block completes abruptly for reason *S*, then the try statement completes abruptly for reason *S* (and the throw of value *V* is discarded and forgotten).
- If execution of the try block completes abruptly for any other reason *R*, then the finally block is executed. Then there is a choice:
  - If the finally block completes normally, then the try statement completes abruptly for reason *R*.
  - If the finally block completes abruptly for reason *S*, then the try statement completes abruptly for reason *S* (and reason *R* is discarded).

Avoiding Code Duplication for try … finally

- The Java Virtual Machine designers wanted to avoid this code duplication

- So they invented a *new* notion of *subroutine*
  - Executes within the stack frame of a method
  - Has access to and can modify local variables
  - One of the few true innovations in the JVM

JVML Subroutines Are Complicated

- Subroutines are the most difficult part of the JVML
- And account for the several bugs and inconsistencies in the bytecode verifier
- Complicate the formal proof of correctness:
  - 14 or 26 proof invariants due to subroutines
  - 50 of 120 lemmas due to subroutines
  - 70 of 150 pages of proof due to subroutines

Are JVML Subroutines Worth the Trouble ?

- Subroutines save space?
  - About 200 subroutines in 650,000 lines of Java (mostly in JDK)
  - No subroutines calling other subroutines
  - Subroutines save 2427 bytes of 8.7 Mbytes (0.02%) !
- Changing the name of the language from Java back to Oak would save 13 times more space !

Exceptions. Conclusion

- Exceptions are a very useful construct
- A good **programming language solution** to an important **software engineering problem**
- But exceptions are complicated:
  - Hard to implement
  - Complicate the optimizer
  - Very hard to debug the implementation (exceptions are exceptionally rare in code)

Homework

- WA7 due today
- For Tuesday – Read Graham paper on gprof
- Midterm 2 – Thursday April 12 (7 days)
  - Covers Lectures 12 – 21 and all reading, WA's and PA's done during that time