

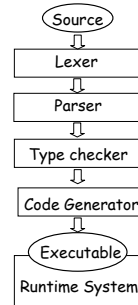
Exceptions. Language Design and Implementation Issues

Lecture 23

Prof. Necula CS 164

1

Structure of a Compiler



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

Prof. Necula CS 164

2

Lecture Summary

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

Prof. Necula CS 164

3

Exceptions. Motivation.

- "Classroom" programs are written with optimistic assumptions
- Real-world programs must consider "exceptional" situations:
 - Resource exhaustion (disk full, out of memory, ...)
 - Invalid input
 - Errors in the program (null pointer dereference)
- It is usual for solid code to contain 30-50% error handling code !

Prof. Necula CS 164

4

Exceptions. Motivation

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 !

Prof. Necula CS 164

5

Exceptions. Error Return Codes.

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

Prof. Necula CS 164

6

Error Return Codes

- It is sometimes hard to select return codes
 - What is a good error code for "double divide(...)"?
- 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

Prof. Necula CS 164

7

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 ?

Prof. Necula CS 164

8

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

Prof. Necula CS 164

9

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

Prof. Necula CS 164

10

Adding Exceptions to Cool

- We extend the language of expressions:

$$e ::= \text{throw } e \mid \text{try } e \text{ catch } x : T \Rightarrow e'$$
- (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

Prof. Necula CS 164

11

Adding Exceptions to Cool

(Informal) semantics of **try e catch x : T ⇒ e₁**

1. *e* is evaluated first
2. If evaluation of *e* terminates normally with *v* then *v* is the result of the entire expression
 - Else (evaluation of *e* terminates exceptionally)
 - If the exception parameter is of type $\leq T$ then
 - Evaluate *e₁* with *x* bound to the exception parameter
 - The (normal or exceptional) result of evaluating *e₁* becomes the result of the entire expression
 - Else
 - The entire expression terminates exceptionally (the exception is not caught here)

Prof. Necula CS 164

12

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"; }
}
```

Prof. Necula CS 164

13

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

Prof. Necula CS 164

14

Overview

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

Prof. Necula CS 164

15

Typing Exceptions

- We must extend the Cool typing rules
$$O, M, C \vdash e : T$$
 - 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

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

Prof. Necula CS 164

16

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 !

Prof. Necula CS 164

17

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

Prof. Necula CS 164

18

Overview

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

Prof. Necula CS 164

19

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
$$g ::= v \mid \text{Exc}(v)$$
- Thus given a generalized value we can:
 - Tell if it is normal or exceptional return, and
 - Extract the return value or the exception parameter

Prof. Necula CS 164

20

Operational Semantics of Exceptions (1)

- We change the evaluation judgment
- $so, E, S \vdash e : g, S'$
- Existing rules are kept (for normal values):

$$\frac{so, E, S \vdash e_1 : \text{Int}(n_1), S_1 \quad so, E, S_1 \vdash e_2 : \text{Int}(n_2), S_2}{so, E, S \vdash e_1 + e_2 : \text{Int}(n_1 + n_2), S_2}$$

$$\frac{E(\text{id}) = I_{\text{id}} \quad S(I_{\text{id}}) = v}{so, E, S \vdash \text{id} : v, S} \quad \frac{}{so, E, S \vdash \text{self} : so, S}$$

Prof. Necula CS 164

21

Operational Semantics of Exceptions (2)

- "throw e " returns exceptionally:
- $so, E, S \vdash e : v, S_1$
- $so, E, S \vdash \text{throw } e : \text{Exc}(v), S_1$
- What if the evaluation of e itself throws an exception?
 - E.g. "throw (1 + (throw 2))" is like "throw 2"
 - Formally:

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

Prof. Necula CS 164

22

Operational Semantics of Exceptions (3)

- All existing rules are changed to propagate the exception:

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

- Note: the evaluation of e_2 is aborted

$$\frac{so, E, S \vdash e_1 : \text{Int}(n_1), S_1 \quad so, E, S_1 \vdash e_2 : \text{Exc}(v), S_2}{so, E, S \vdash e_1 + e_2 : \text{Exc}(v), S_2}$$

Prof. Necula CS 164

23

Operational Semantics of Exceptions (4)

- The rules for "try" expressions:
 - Multiple rules (just like for a conditional)
- What if e terminates exceptionally?
 - We must check whether it terminates with an exception parameter of type T or not

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

Prof. Necula CS 164

24

Operational Semantics for Exceptions (5)

- If e does not throw the expected exception

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

- If e does throw the expected exception

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

Prof. Necula CS 164

25

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

Prof. Necula CS 164

26

Overview

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

Prof. Necula CS 164

27

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!
 - Optimize the common case !

Prof. Necula CS 164

28

Implementing Exceptions with Long Jumps (1)

Idea:

- "try" saves on the stack the handler context:
 - The current SP, FP and the label of the catch code
- "throw" jumps to the last saved handler label
 - Called a long jump
- We reserve the MIPS register $\$t9$ to hold the most recently saved handler context
 - We refer to $\$t9$ as $\$xp$
- We consider here exceptions without parameters

Prof. Necula CS 164

29

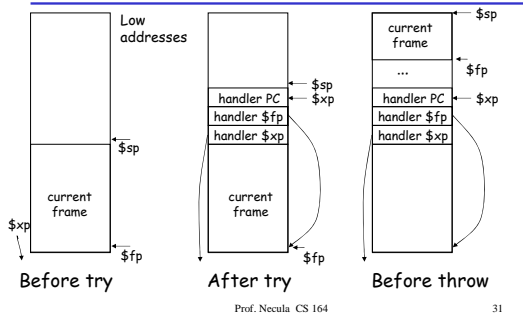
Implementing Exceptions with Long Jumps (2)

```
cgen(throw) =  
lw $t1 0($xp) ; Load the handler address  
jr $t1       ; Jump to the handler
```

Prof. Necula CS 164

30

Long Jumps. Example.



Implementing Exceptions with Long Jumps (3)

```

cgen(try e catch e') =
  addiu $sp $sp -12 : Push 3 words
  sw $xp 12($sp) : Save old handler context
  sw $fp 8($sp) : Save FP
  sw L_catch 4($sp) : Save handler address
  addiu $xp $sp 4 : Set the new handler context
  cgen(e) : Try the body. Result in $a0
  lw $xp 12($sp) : Restore old handler context
  addiu $sp $sp 12 : Pop the context
  b_end_try
L_catch:
  addiu $sp $xp 8 : restore SP as it was before TRY
  lw $fp 4($xp)
  lw $xp 8($xp)
  cgen(e') : Catch part. Result in $a0
  end_try:
  
```

Prof. Necula CS 164 32

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
- 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

```

## 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. Example.

- Consider the expression  $e_1 + (\text{try } e_2 + (\text{try } e_3 \text{ catch } e_3') \text{ catch } e_2')$

| Regular code:              | Handlers:                 | Exception Table:                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
|----------------------------|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|----|---------|-------|--------|--------|-------|--------|----------|-------|--------|-------|----------|-------|--------|-------|-----|----------|
| $L_1$ : cgen( $e_1$ )      | $C_{23}$ : cgen( $e_2'$ ) | <table border="1"> <thead> <tr> <th>From</th> <th>To</th> <th>Handler</th> </tr> </thead> <tbody> <tr> <td><math>L_1</math></td> <td><math>L_1'</math></td> <td>caller</td> </tr> <tr> <td><math>L_2</math></td> <td><math>L_2'</math></td> <td><math>C_{23}</math></td> </tr> <tr> <td><math>L_3</math></td> <td><math>L_3'</math></td> <td><math>C_3</math></td> </tr> <tr> <td><math>C_{23}</math></td> <td><math>C_3</math></td> <td>caller</td> </tr> <tr> <td><math>C_3</math></td> <td>end</td> <td><math>C_{23}</math></td> </tr> </tbody> </table> | From     | To | Handler | $L_1$ | $L_1'$ | caller | $L_2$ | $L_2'$ | $C_{23}$ | $L_3$ | $L_3'$ | $C_3$ | $C_{23}$ | $C_3$ | caller | $C_3$ | end | $C_{23}$ |
| From                       | To                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | Handler  |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_1$                      | $L_1'$                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | caller   |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_2$                      | $L_2'$                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | $C_{23}$ |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_3$                      | $L_3'$                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | $C_3$    |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $C_{23}$                   | $C_3$                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | caller   |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $C_3$                      | end                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | $C_{23}$ |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_1'$ : push acc          | goto $L_4'$               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_2$ : cgen( $e_2$ )      | $C_3$ : cgen( $e_3'$ )    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_2'$ : push acc          | goto $L_3'$               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_3$ : cgen( $e_3$ )      |                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_3'$ : $t1 = \text{pop}$ |                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| acc ← acc + $t_1$          |                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| $L_4'$ : $t1 = \text{pop}$ |                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |
| acc ← acc + $t_1$          |                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |          |    |         |       |        |        |       |        |          |       |        |       |          |       |        |       |     |          |

There is a subtle bug here !

## 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

Prof. Necula CS 164

37

## try ... finally ...

- Another exception-related construct:
  - try  $e_1$  finally  $e_2$
  - After the evaluation of  $e_1$  terminates (either normally or 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 yards to the west", dist); ...
finally
 fclose(f)
```

Prof. Necula CS 164

38

## Code Generation for try ... finally

- Consider the expression:  $e_1 + \text{try } e_2 \text{ finally } e_2'$

Regular code:

```
L1: cgen(e1)
L1': t1 = acc
L2: cgen(e2)
L2': t2 = acc
 cgen(e2') ; Run finally
 acc ← t1 + t2
```

Handlers:

```
C2: cgen(e2')
 jr runtime_throw
```

Exception Table:

| From           | To               | Handler        |
|----------------|------------------|----------------|
| L <sub>1</sub> | L <sub>1</sub> ' | caller         |
| L <sub>2</sub> | L <sub>2</sub> ' | C <sub>2</sub> |
| C <sub>2</sub> | end              | caller         |

Code for finally clauses must be duplicated!

Prof. Necula CS 164

39

## 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

Prof. Necula CS 164

40

## JVML Subroutines Are Complicated

- Subroutines are the most difficult part of the JVM
- 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

Prof. Necula CS 164

41

## Are JVM 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 to Oak would save 13 times more space!

Prof. Necula CS 164

42

## 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)