

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

[http://www.cs.uiuc.edu/class
/sp07/cs421/](http://www.cs.uiuc.edu/class/sp07/cs421/)

Based in part on slides by Mattox Beckman, as updated
by Vikram Adve and Gul Agha

Classes of Control Flow

Selection

- Structured: if/then/else, continue, switch, case
- Unstructured: GOTO, computed GOTO, labelled entry

Iteration

- Precomputed iteration space: do, foreach (range)
- Dynamic iteration space: for, while, recursion

Classes of Control Flow

Invocation

- Direct calls: function, subroutine
- Indirect calls: func. pointer, methods of a class, closure

Termination of scope

- Structured: break, break label, exceptions, CPS
- Unstructured: GOTO, setjmp/longjmp, exit

Classes of Control Flow

Concurrency

- Manual: processes, threads, futures, (coroutines)
- Automatic: doall, doacross, directives (e.g., OpenMP)
- Communication and synchronization techniques are critical.
- You should be familiar with if/then/else, switch/case, do, for, while, foreach (range), GOTO, break, exit.

What is an exception?

Exception :

- A runtime event that causes control to be transferred from the current program point to a separate code fragment that does not follow the current point in the program's control flow.

Exception handler :

- Code fragment that receives control when exception occurs.

What is an exception?

Internal exception :

- An exception generated explicitly by the program.

External exception :

- An exception generated by the runtime system.

Why do we need exceptions?

- What can cause an external exception?

- Why would you use an internal exception?

Poor Program to Multiply a List

```
let rec list_mult list =  
  match list with [ ] -> 1  
  | x::xs -> x * list_mult xs
```

Problem: Want to avoid all the
multiplications in list containing 0

Suboptimal Solution

```
let rec list_mult list =  
  match list with [ ] -> 1  
  | x::xs ->  
    if x = 0 then 0 else x * list_mult xs
```

Once it see 0, it ignores the tail of the list

Problem: Still multiplies 0 times each element in the list before it

Eg: list = [7;4;0] then still multiply 4, 7 by 0

Better Solution

```
exception Zero;;
```

```
let rec list_mult_aux list =  
  match list with [ ] -> 1  
  | x :: xs ->  
    if x = 0 then raise Zero  
    else x * list_mult_aux xs;;
```

```
let list_mult list =  
  try list_mult list with Zero -> 0;;
```

What Gets Skipped?

- Let list = [7; 4; 0; 5]
- First function:

list_mult [7; 4; 0; 5]

=> (7 * result) <- list_mult [4; 0; 5]

=> (7 * result) <- (4 * result) <- list_mult [0; 5]

=> (7 * result) <- (4 * result) <- (0 * result) <-
list_mult [5]

=> (7 * result) <- (4 * result) <- (0 * result) <-
(5 * result) <- list_mult []

What Gets Skipped?

$\Rightarrow (7 * \text{result}) \leftarrow (4 * \text{result}) \leftarrow (0 * \text{result})$
 $\leftarrow (5 * 1)$

$\Rightarrow (7 * \text{result}) \leftarrow (4 * \text{result}) \leftarrow (0 * 5)$

$\Rightarrow (7 * \text{result}) \leftarrow (4 * 0)$

$\Rightarrow (7 * 0) \Rightarrow 0$

What Gets Skipped?

- Second function:

`list_mult [7; 4; 0; 5]`

`=> (7 * result) <- list_mult [4; 0; 5]`

`=> (7 * result) <- (4 * result) <- list_mult
[0; 5]`

`=>(7 * result) <- (4 * 0)`

`=>(7 * 0)`

`=> 0`

What Gets Skipped?

- Third function:

```
list_mult [7; 4; 0; 5]
```

```
=> list_mult_aux [7; 4; 0; 5]
```

```
=> (7 * result) <- list_mult_aux [4; 0; 5]
```

```
=> (7 * result) <- (4 * result) <-  
    list_mult_aux [0; 5]
```

```
=> 0
```

- Could be big win if * were expensive

Structured Exception Propagation

On exception e :

- Unwind stack to innermost call that handles exception e
- In each stack frame, perform cleanups:
 - call local object destructors in C++
 - release locks in Java
- In stack frame with handler, execute handler
- Handler may rethrow the exception, throw a different exception, or continue

Exception Data

- Exception data (values) are needed to communicate what exception occurred, state of computation when it occurred, and perhaps where.
- OCAML, Ada: exception is a built-in type
- Java, C++: subclasses of an “exception class”
- C: int argument to longjmp()

Current Continuations

- Idea: remaining computation to be done when given result of most local expression
- Example: if (x && y) then s() else t()
- Intuitively: the current continuation of x && y is s() if x && y evaluates to true, and is t () if it evaluates to false instead

callcc

- (Will use SML/NJ version, but with Ocaml syntax)
- Built-in type of 'a cont (SMLofNJ.Cont.callcc)
- `callcc : ('a cont -> 'a) -> 'a`
if `(callcc (fun k -> not x))` then 5 else 3
- k represents the function that will either return 3 or 5
- As given, since k is not used, callcc has no effect

Throw

- The way to use a continuation “captured” by callcc is with throw:
- throw: 'a cont -> 'a -> 'b
- if (callcc (fun k -> (throw k false))) || true
then 3 else 5 = 3
- if (callcc (fun k -> (throw k false) || true))
then 3 else 5 = 5

Control Flow through callcc

- Callcc is a powerful control construct
- Can implement
 - Exceptions
 - Concurrency (done in Concurrent ML)
 - Other non-local control mechanisms

List Multiplication Example

```
let list_mult list =  
  callcc( fun zerok ->  
    let rec list_mult_aux list =  
      match list with [ ] -> 1  
      | x :: xs ->  
        if x = 0 then throw zerok 0  
        else x * list_mult_aux xs  
    in list_mult_aux list)
```