

Programming Paradigms

Lecture 6

Slides are from Prof. Chin Wei-Ngan from NUS

Tupled Recursion and Exceptions

Reminder of Last

- Computing with procedures
 - lexical scoping
 - closures
 - procedures as values
 - procedure call
 - Higher-Order Programming
 - proc. abstraction
 - lazy arguments
 - genericity
 - loop abstraction
 - folding
-

Declarative Programming

Declarative Programming

- We are exploring declarative programming
 - declarative programming model
 - declarative programming techniques
- We used “declarative” variables for single-assignment variables

...what does **declarative** mean?

Declarative means...

- Programs returns
 same result
for
 same arguments
- Always, always, always...
 regardless of any other computations

Declarative Programming Properties

- Independence
 - write programs independently
 - test and debug independently
 - other components of program do not matter
- Simple reasoning
 - declarative programs only compute values
 - no hidden state, no history, ...
- This means simple development...

Is Everything Declarative?

- No, it is not...
 - ...there is no silver bullet
- Why bother then?

Be as Declarative as You Can

- Many program components can be written in a declarative style
 - use the benefits as much as possible
- For the rest, use other techniques
 - concurrency
 - state
 - objects

Significance

- Some languages are better than others at declarative programming (Oz versus C++)
- Declarative programming techniques are useful whatever language you program in
 - this course wants to sharpen your mind
 - this course uses a language that is good at declarative programming and the other techniques to come

Tupled Recursion

Functions with multiple results

Computing Average

```
fun {SumList Ls}  
  case Ls of nil then 0  
  [] X|Xs then X+{SumList Xs} end  
End
```

```
fun {Length Ls}  
  case Ls of nil then 0  
  [] X|Xs then 1+{Length Xs} end  
end
```

```
fun {Average Ls} {Sum Ls}/{Length Ls} end
```

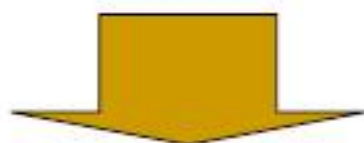
- What is the Problem?

Problem?

- Traverse the same list multiple traversals.
- Solution : compute multiple results in a single traversal!

Tupling - Computing Two Results

```
fun {CPair Ls}  
  {Sum Ls}#{Length Ls}  
end
```



```
fun {CPair Ls}  
  case Ls of nil then 0#0  
  [] X|Xs then case {CPair Xs}  
    of S#L then (X+S) # (1+L) end  
  end  
end
```

Using Tupled Recursion

```
fun {Average Ls}  
    {Sum Ls}/{Length Ls}  
end
```



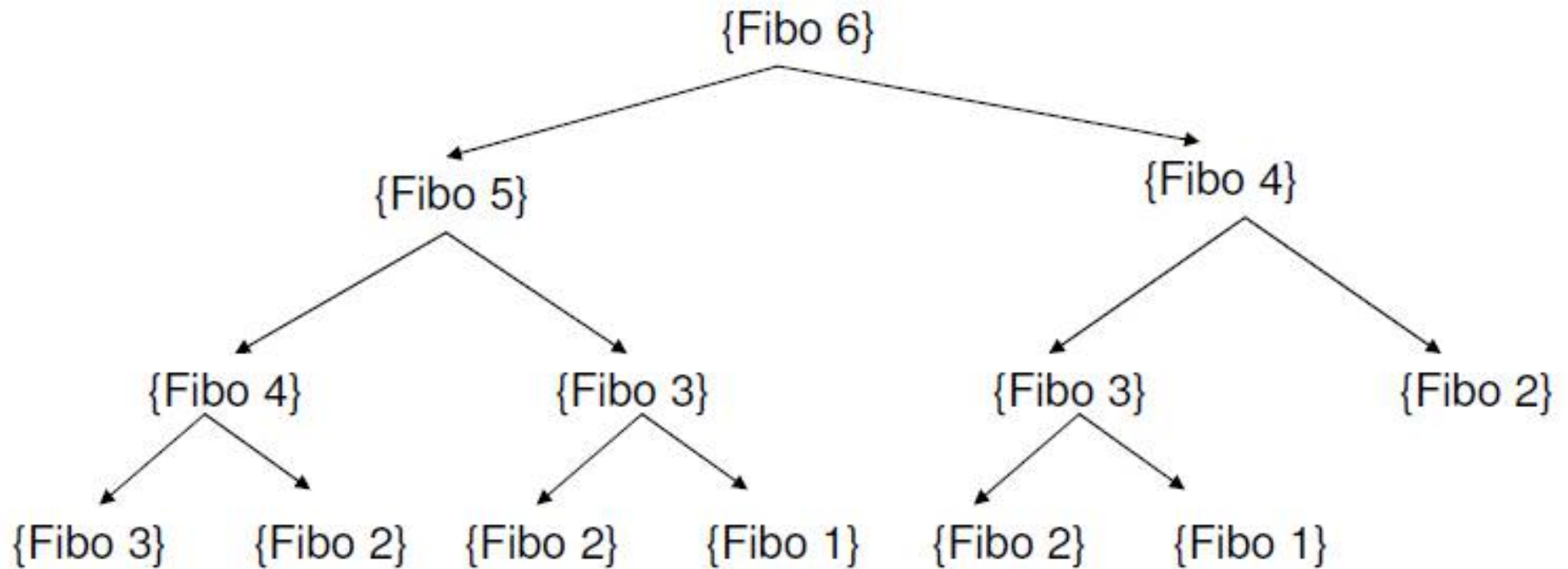
```
fun {Average Ls}  
    case {CPair Ls} of S#L then S/L end  
end
```

Inefficient Fibonacci

- Time complexity of $\{\text{Fibo } N\}$ is proportional to 2^N .

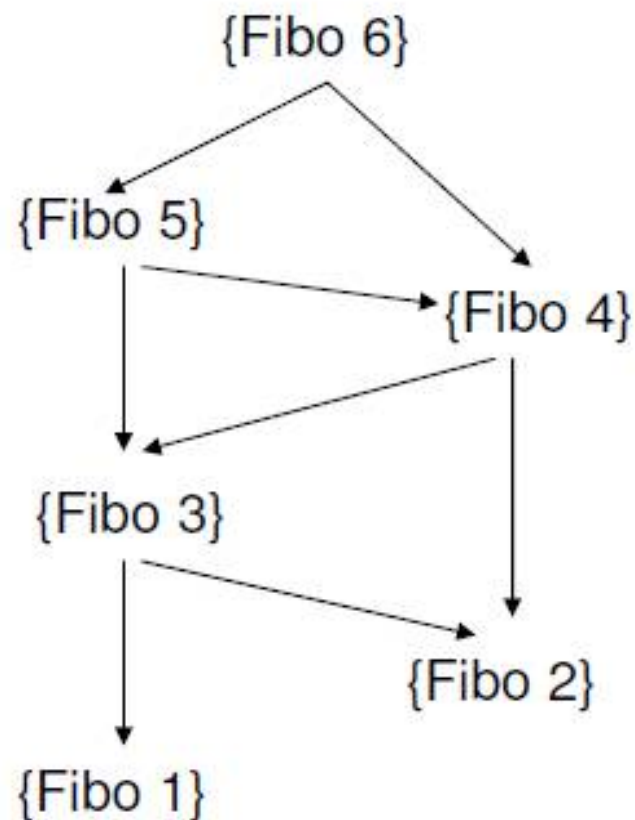
```
fun {Fibo N}
  case N of
    1 then 1
  [] 2 then 1
  [] M then {Fibo (M-1)} + {Fibo (M-2)}
  end
end
```


A Call Tree of Fibo



Many repeated calls!

A Call Graph of Fibo

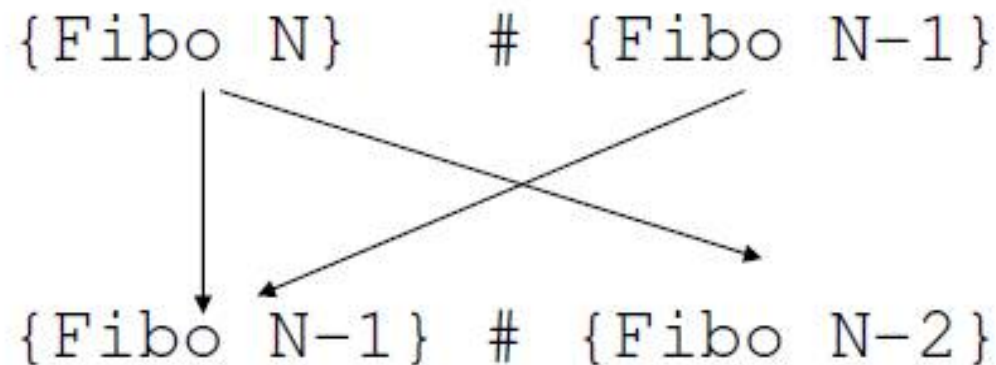


No repeated call through reuse of identical calls

Tupling - Computing Two Results

```
fun {FPair N}  
  {Fibo N}#{Fibo N-1}  
end
```

Compute two calls from next two:



Tupling - Computing Two Results

```
fun {FPair N}  
    {Fibo N}#{Fibo N-1}  
end
```



```
fun {FPair N}  
    case N of  
        2 then 1#1  
        [] M then case {FPair M-1}  
                    of S#L then (S+L)#S end  
    end  
end
```

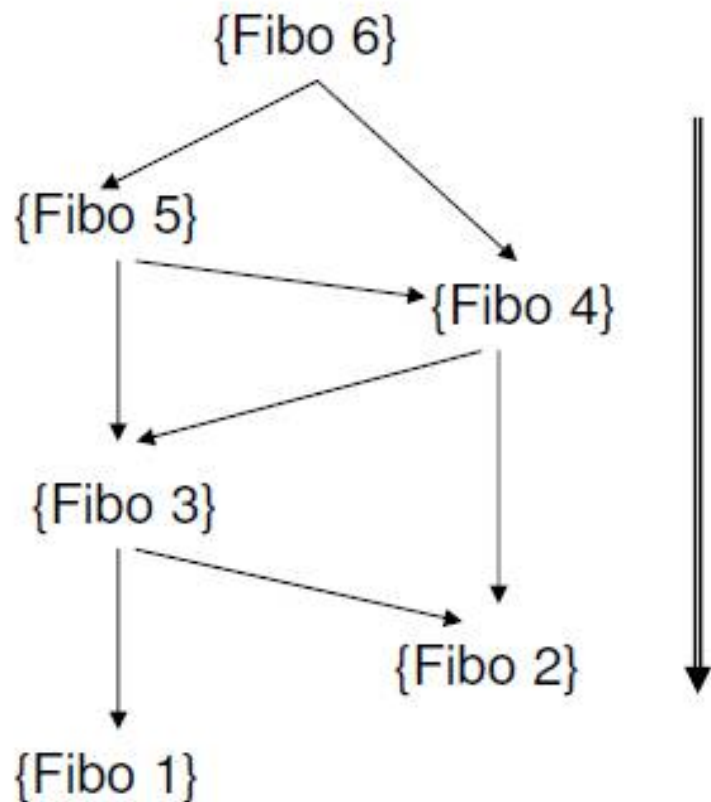
Using the Tupled Recursion

```
fun {Fibo N}  
  case {Fibo N+1}#{Fibo N} of  
    A#B then B end  
end
```



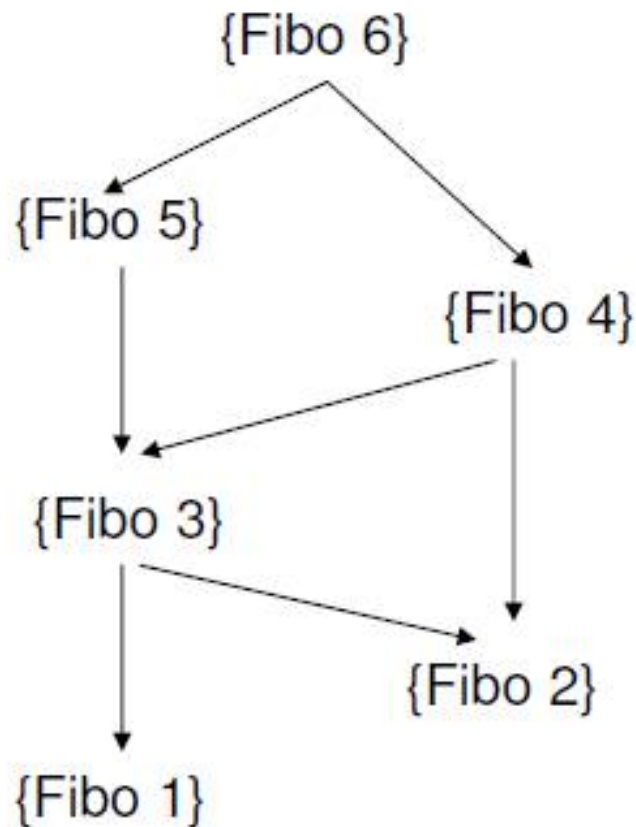
```
fun {Fibo N}  
  case {FPair N+1} of A#B then B end  
end
```

Linear Recursion



```
fun {FPair N}  
  case N of 2 then 1#1  
  [] M then case {FPair M-1}  
    of S#L then (S+L)#S end  
  end  
end
```

To Iteration



$\{FPair\ N\} = \{H^{(N-2)}\ 1\#1\}$
 $= \{FPairIt\ (N-2)\ 1\#1\}$

```
fun {H P}  
  case P of A#B then A+B#A end  
end
```

Tail-Recursive Fibonacci

```
fun {FPair N}    {FPairIt (N-2) 1#1} end
```

```
fun {FPairIt N P}
```

```
  case N of
```

```
    0 then P
```

```
    [] M then {FPairIt N-1 {H P}} end
```

```
end
```

Summary So Far

- Tupled Recursion
 - Eliminate multiple traversals
 - Eliminate redundant calls
- Eureka – find suitable tuple of calls.

Exceptions

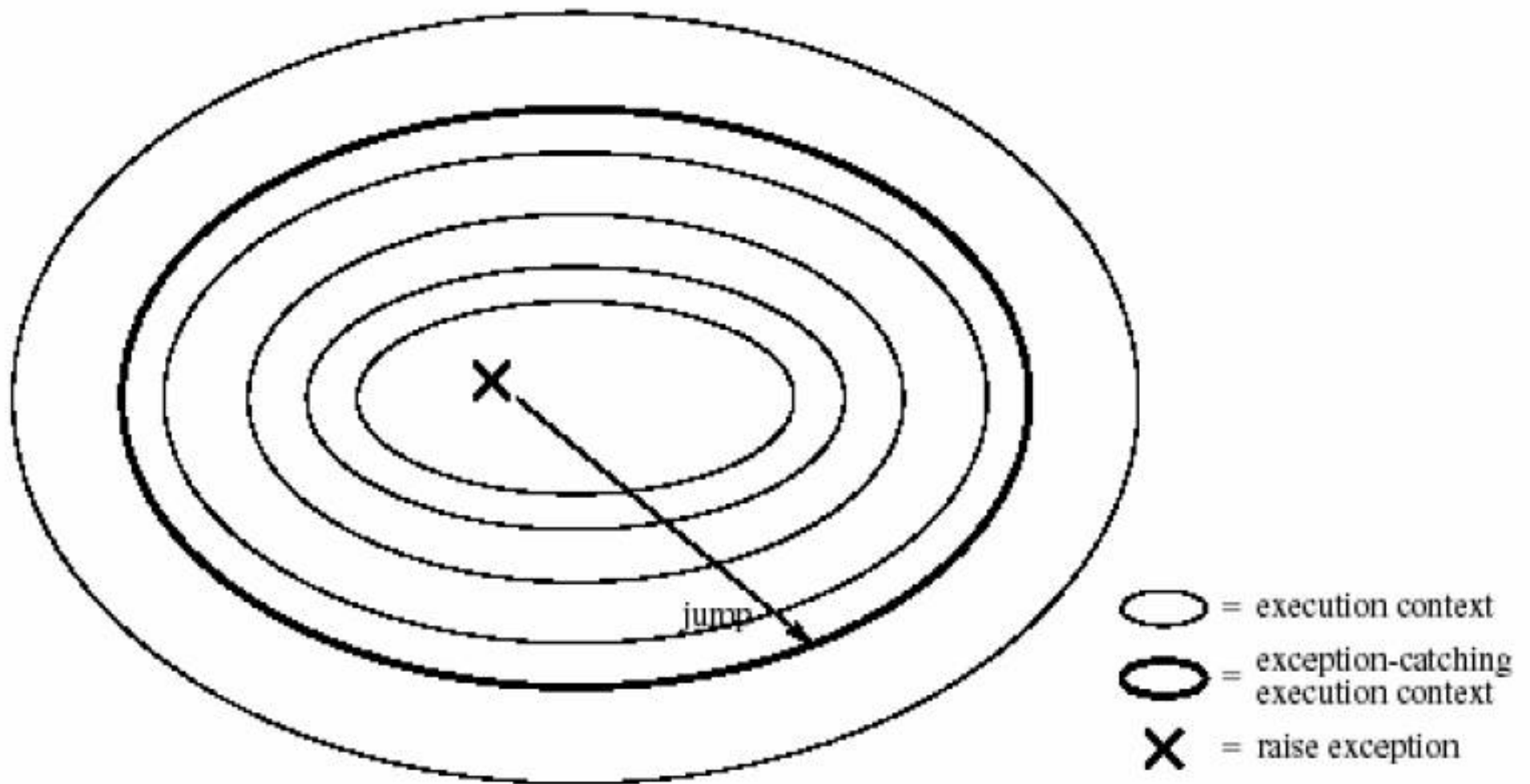
Exceptions

- Error = Actual behavior - Desired behavior.
 - Type of errors:
 - Internal: invoking an operation with an illegal type/value
 - External: opening a nonexisting file
 - Detect and handle these errors without stopping the program execution.
 - Solution - Transfer to an **exception handler**, and pass a value that describes the error.
-

Exceptions handling

- Oz program = interacting “**components**”
 - Exception causes a “**jump**” from inside the component to its boundary.
 - Able to exit arbitrarily levels of nested contexts.
 - A **context** is an entry on the semantic stack.
 - Nested contexts are created by procedure calls and sequential compositions.
-

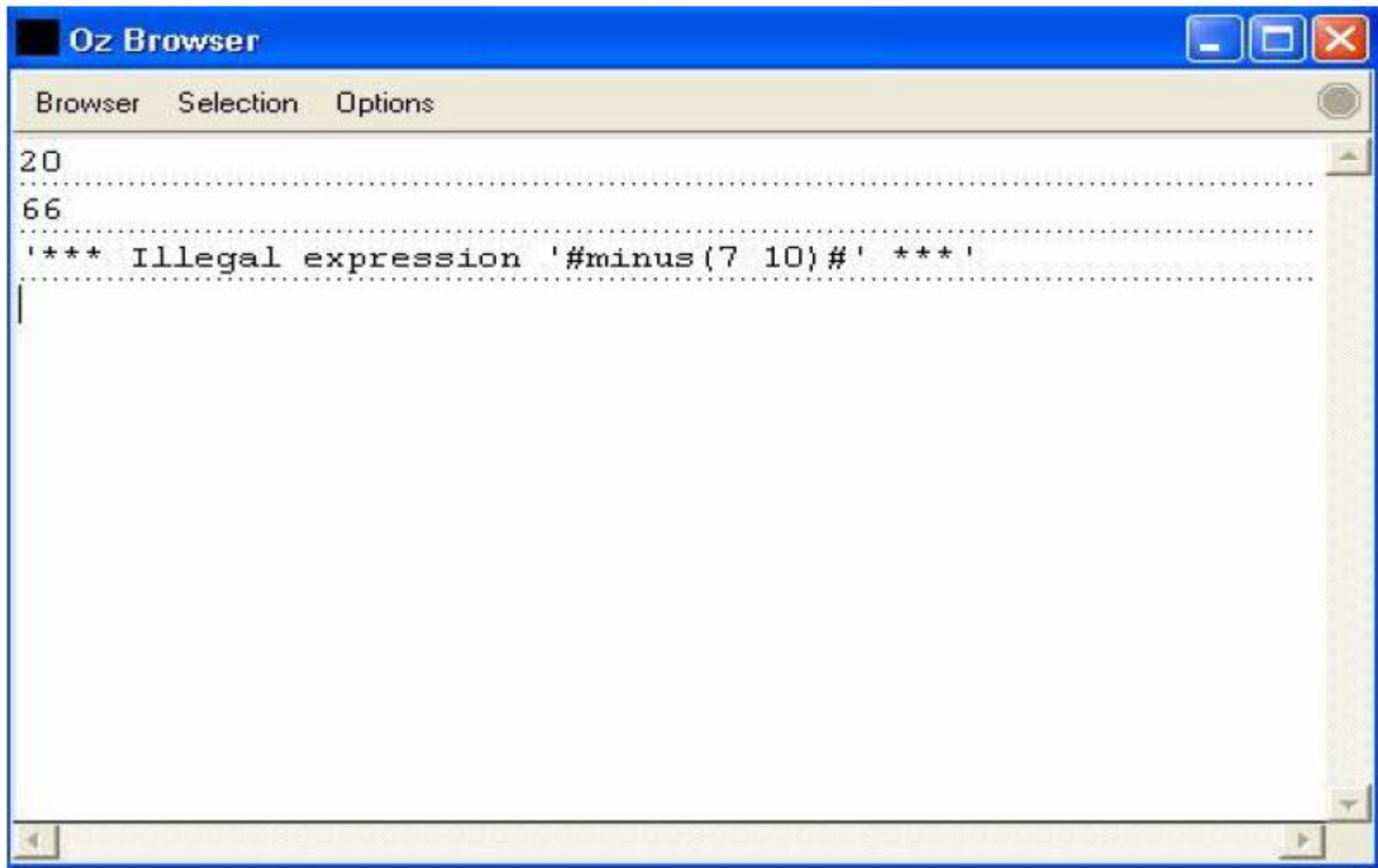
Exceptions handling



Exceptions (Example)

```
fun {Eval E}
  if {IsNumber E} then E
  else
    case E
    of plus(X Y) then {Eval X}+{Eval Y}
    [] times(X Y) then {Eval X}*{Eval Y}
    else raise illFormedExpression(E) end
    end
  end
end
try
  {Browse {Eval plus(plus(5 5) 10)}}
  {Browse {Eval times(6 11)}}
  {Browse {Eval minus(7 10)}}
catch illFormedExpression(E) then
  {Browse '*** Illegal expression '#E#' ***'}
end
```

Exceptions (Example)



The screenshot shows a window titled "Oz Browser" with a blue title bar and standard window controls. Below the title bar is a menu bar with "Browser", "Selection", and "Options". The main content area is a text editor with a dotted line border. It contains the following text:

```
20
66
'*** Illegal expression '#minus(7 10)#' ***'
```

The text is displayed in a monospaced font. The window has a vertical scrollbar on the right and a horizontal scrollbar at the bottom.

Exceptions. `try` and `raise`

- **`try`:** creates an exception-catching context together with an exception handler.
 - **`raise`:** jumps to the boundary of the innermost exception-catching context and invokes the exception handler there.
 - **`try <S> catch <X> then <S>1 end:`**
 - if `<S>` does not raise an exception, then execute `<S>`.
 - if `<S>` raises an exception, then the (still ongoing) execution of `<S>` is aborted. All information related to `<S>` is popped from the semantic stack. Control is transferred to `<S>1`, passing it a reference to the exception in `<X>`.
-

Exceptions. Full Syntax

- A **try** statement can specify a **finally** clause which is always executed, whether or not the statement raises an exception.
- **try** $\langle S \rangle_1$ **finally** $\langle S \rangle_2$ **end**
is equivalent to:
- **try** $\langle S \rangle_1$
 catch X **then**
 $\langle S \rangle_2$
 raise X **end**
end
 $\langle S \rangle_2$
where an identifier x is chosen that is not free in $\langle S \rangle_2$

Exceptions. Full Syntax (Example)

- An example with `catch` and `finally`.

- **try**

```
{ProcessFile F}
```

```
catch X then
```

```
{Browse '*** Exception '#X#
```

```
  ' when processing file ***' }
```

```
finally {CloseFile F} end
```

- Similar with two nested `try` statements!

System Exceptions

- Raised by Mozart system
 - `failure`: attempt to perform an inconsistent bind operation in store (“unification failure”);
 - `error`: run-time error inside a program, like type or domain errors;
 - `system`: run-time condition in the environment of the Mozart, like failure to open a connection between two processes.
-

System Exceptions (Example)

```
functor
import
  Browser
define
  fun {One} 1 end
  fun {Two} 2 end
  try
    {One}={Two}
  catch
    failure(...) then
      {Browser.browse 'We caught the failure'}
    end
  end
end
```

Summary

- Recursion vs Iteration
 - Tupled Recursion
 - Exceptions
-

Reverse

- Reversing a list
- How to reverse the elements of a list

`{Reverse [a b c d]}`

returns

`[d c b a]`

Reversing a List

- Reverse of nil is nil
- Reverse of $X|X_r$ is Z , where
reverse of X_r is Y_r , and
append Y_r and $[X]$ to get Z

$\{Rev\ [a\ b\ c\ d]\} = [d\ c\ b\ a]$
 $\{Rev\ a|[b\ c\ d]\} = \{Append\ \{Rev\ [b\ c\ d]\}\ [a]\} = [d\ c\ b\ a]$
 $\{Rev\ b|[c\ d]\} = \{Append\ \{Rev\ [c\ d]\}\ [b]\} = [d\ c\ b]$
 $\{Rev\ c|[d]\} = \{Append\ \{Rev\ [d]\}\ [c]\} = [d\ c]$
 $\{Rev\ d|nil\} = \{Append\ \{Rev\ nil\}\ [d]\} = [d]$
 nil

Question

- What is correct

`{Append {Reverse Xr} X}`

or

`{Append {Reverse Xr} [X]}`

Naive Reverse Function

```
fun {NRev Xs}  
  case Xs of  
    nil then nil  
    [] X|Xr then {Append {NRev Xr} [X]}  
  end  
end
```


Question

- What is the problem with the naive reverse?
- Possible answers
 - not tail recursive
 - Append is costly:
 - there are $O(|L1|)$ calls

```
fun {Append L1 L2}  
  case L1 of  
    nil then L2  
    [] H|T then H|{Append T L2}  
  end  
end
```

Cost of Naive Reverse

- Suppose a recursive call $\{\text{NRev } Xs\}$

- where $\{\text{Length } Xs\} = n$

- assume cost of $\{\text{NRev } Xs\}$ is $c(n)$

number of function calls

- then $c(0) = 0$

$$c(n) = c(\{\text{Append } \{\text{NRev } Xr\} [X]\}) + c(n-1)$$

$$= (n-1) + c(n-1)$$

$$= (n-1) + (n-2) + c(n-3) = \dots = n-1 + (n-2) + \dots + 1$$

- this yields: $c(n) = \frac{n(n-1)}{2}$

- For a list of length n , NRev uses approx. n^2 calls!

Doing Better for Reverse

- Use an accumulator to capture currently reversed list
- Some abbreviations
 - `{IR Xs}` `for {IterRev Xs}`
 - `Xs ++ Ys` `for {Append Xs Ys}`

Computing NRev

{NRev [a b c]}	=
{NRev [b c]}++[a]	=
({NRev [c]}++[b])++[a]	=
(({NRev nil}++[c])++[b])++[a]	=
((nil++[c])++[b])++[a]	=
([c]++[b])++[a]	=
[c b]++[a]	=
[c b a]	

Computing IterRev (IR)

`{IR [a b c] nil}` =

`{IR [b c] a|nil }` =

`{IR [c] b|a|nil}` =

`{IR nil c|b|a|nil}` =

`[c b a]`

■ The general pattern:

`{IR X|Xr Rs} ⇒ {IR Xr X|Rs}`

Why is Iteration Possible?

Associative Property

$$\begin{aligned} \{\text{Append } \{\text{Append RL } [a] \} [b] \} \\ = \{\text{Append RL } \{\text{Append } [a] [b] \} \} \end{aligned}$$

More Generally

$$\begin{aligned} \{\text{Append } \{\text{Append RL } [a] \} \text{Acc}\} \\ = \{\text{Append RL } \{\text{Append } [a] \text{Acc}\} \} \\ = \{\text{Append RL } a | \text{Acc} \} \end{aligned}$$

IterRev Intermediate Step

```
fun {IterRev Xs Ys}
  case Xs of
    nil then Ys
  [] X|Xr then {IterRev Xr X|Ys}
  end
end
```

- Is tail recursive now

IterRev Properly Embedded

```
local
  fun {IterRev Xs Ys}
    case Xs
    of nil    then Ys
    [] X|Xr  then {IterRev Xr X|Ys}
    end
  end
in
  fun {Rev Xs} {IterRev Xs nil} end
end
```

State Invariant for IterRev

- Unroll the iteration a number of times, we get:

$$\{ \text{IterRev } [X_1 \dots X_n] \ W \}$$

=

$$\{ \text{IterRev } [X_{i+1} \dots X_n] \ [X_i \dots X_1] ++ W \}$$

Reasoning for IterRev and Rev

■ Correctness:

$\{\text{Rev } Xs\}$ is $\{\text{IterRev } Xs \text{ nil}\}$

■ Using the state invariant, we have:

$$\begin{aligned} & \{\text{IterRev } [X_1 \dots X_n] \text{ nil}\} = \\ &= \{\text{IterRev nil } [X_n \dots X_1]\} \\ &= [X_n \dots X_1] \end{aligned}$$

■ Thus: $\{\text{Rev } [X_1 \dots X_n]\} = [X_n \dots X_1]$

■ Complexity:

■ The number of calls for $\{\text{IterRev } L \text{ nil}\}$, where list L has N elements, is $c(N)=N$

Summary So Far

- Use accumulators
 - yields iterative computation
 - find state invariant
 - Loop = Tail Recursion and is a special case of general recursion.
 - Exploit both kinds of knowledge
 - on how programs execute (abstract machine)
 - on application/problem domain
-