A Brief Introduction to Parallel Programming in Fortress

Sukyoung Ryu

Department of Computer Science
Korea Advanced Institute of Science and Technology

December 13, 2009

Fortress: Programmable HPC

- Ubiquitous parallelism
- Transactional synchronization
- Key language features defined by libraries
- Flexible, whitespace-aware mathematical syntax
- Strong typing with polymorphism
- Resolution of overloading based on dynamic types

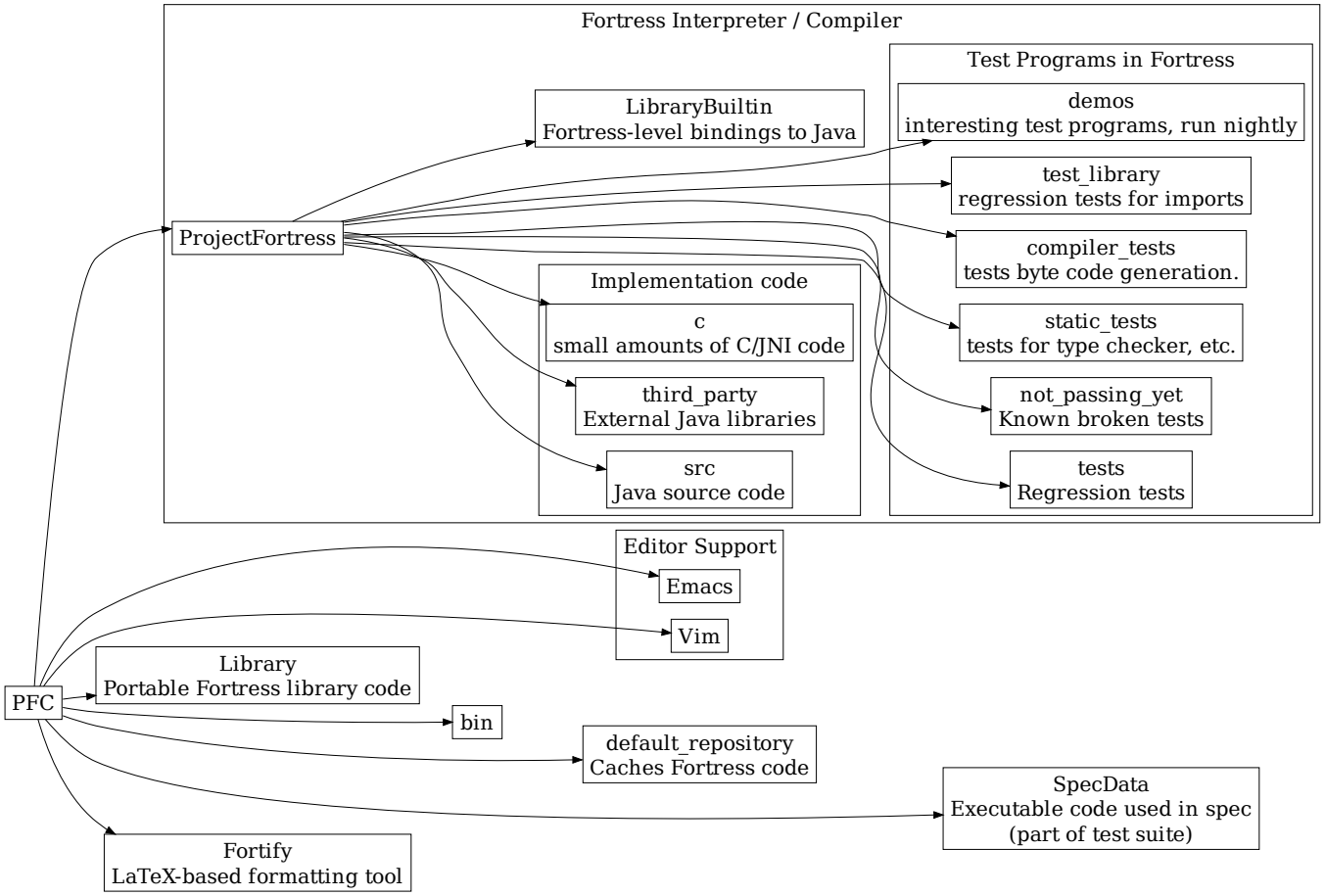
Hello World: Fortress Program #1

```
component hello
export Executable

run() = println("Hello, World!")

end
```

Hello World: Fortress Program #1

```
component hello
export Executable
run() = println("Hello, World!")
end
```

Fortress Repository Structure



Hello, World!

```
component hello
    export Executable
    run() = println("Hello, World!")
end
```

Hello, World!

```
component hello
    export Executable
    run() = println("Hello, World!")
end
```

```
api Executable
    run(): ()
end
```

Hello, World!

```
component hello
    export Executable
    run() = println("Hello, World!")
end
```

```
api Executable
    run(): ()
end
```

- `bin/fortress ProjectFortress/hello.fss`
- `Hello, World!`

Hello, World!

```
component hello
    export Executable
    run() = println("Hello, World!")
end
```

```
api Executable
    run(): ()
end
```

- `bin/fortress ProjectFortress/hello.fss`

Factorial

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

Factorial: Function

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  factorial(n: ZZ32) =
    if 0 <= n <= 1 then 1
    elif n > 1 then n(factorial(n-1))
    end
  run() = println(factorial 5)
end
```

Factorial: Function

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  factorial(n: ℤ32) =
    if 0 ≤ n ≤ 1 then 1
    elif n > 1 then n(factorial(n − 1))
    end
  run() = println(factorial 5)
end
```

Factorial: Juxtaposition

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  factorial(n: ℤ32) =
    if 0 ≤ n ≤ 1 then 1
    elif n > 1 then n(factorial(n − 1))
    end
  run() = println(factorial 5)
end
```

Factorial: Operator

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  opr (n: ℤ32)! =
    if 0 ≤ n ≤ 1 then 1
    elif n > 1 then n(n − 1)!
    end
  run() = println 5!
end
```

Factorial: Exception

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  opr (n: ℤ32)! =
    if 0 ≤ n ≤ 1 then 1
    elif n > 1 then n(n − 1)!
    else fail "Non-negative integer is expected."
    end
  run() = println 5!
end
```

Factorial: Assertion

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  opr (n: ℤ32)! =
    if 0 ≤ n ≤ 1 then 1
    elif n > 1 then n(n − 1)!
    else fail "Non-negative integer is expected."
    end
  run() = assert(5!, 120)
end
```

Factorial: Mutable Local Variable

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  opr (n: ℤ32)! = do
    var result: ℤ32 = 1
    for i ← 1 : n do atomic result := result i end
    result
  end
  run() = assert(5!, 120)
end
```

Factorial: Parallel For Loop

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  opr (n: ℤ32)! = do
    var result: ℤ32 = 1
    for i ← 1 : n do atomic result := result i end
    result
  end
  run() = assert(5!, 120)
end
```

Factorial: Atomic Transaction

$$n! = \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  opr (n: ℤ32)! = do
    var result: ℤ32 = 1
    for i ← 1 : n do atomic result := result i end
    result
  end
  run() = assert(5!, 120)
end
```

Factorial: While Loop

$$n! \quad = \quad \begin{cases} 1 & \text{if } 0 \leq n \leq 1 \\ n(n-1)! & \text{if } n > 1 \end{cases}$$

```
component Factorial
  export Executable
  opr (n: ℤ32)! = do
    var result: ℤ32 = 1
    var i: ℤ32 = 1
    while i ≤ n do  atomic result := result i; i += 1 end
    result
  end
  run() = assert(5!, 120)
end
```

Basics from Fortress Libraries

- Types: ZZ32, Boolean, String
- println

```
println(a:String):()
println(a:Number):()
```

- fail

```
fail[\T\](s:String):T = do
    errorPrintln("FAIL: " s)
    throw FailCalled(s)
  end
```

- assert

```
(* Assertion *)
assert(flag:Boolean): ()
assert(flag: Boolean, failMsg: String): ()
assert(x:Any, y:Any, failMsg: Any...): ()
```

Basics from Fortress Libraries

- Types: $\mathbb{Z}32$, Boolean, String
- *println*

  $println(a : \text{String}) : ()$

  $println(a : \text{Number}) : ()$
- *fail*

  $fail[\![T]\!](s : \text{String}) : T = \texttt{do}$

  $\quad errorPrintln(\text{“}\texttt{FAIL: }\text{”}\ s)$

  $\quad \texttt{throw}\ \text{FailCalled}(s)$

  $\texttt{end}$
- *assert*

  $(* \text{ Assertion } *)$

  $assert(flag : \text{Boolean}) : ()$

  $assert(flag : \text{Boolean}, failMsg : \text{String}) : ()$

  $assert(x : \text{Any}, y : \text{Any}, failMsg : \text{Any} \ldots) : ()$

Exercise: Fibonnaci

```
fib(n: ZZ32): ZZ32 =
    if n < 0
    then fail("Non-negative integer is expected.")
    elif n < 2 then n
    else fib(n-1) + fib(n-2)
    end
```

$$
\begin{array}{l}
\mathit{fib}(n\colon \mathbb{Z}32)\colon \mathbb{Z}32 = \\
\quad \texttt{if } n < 0 \\
\quad \texttt{then } \mathit{fail}(\text{“Non-negative integer is expected.”}) \\
\quad \texttt{elif } n < 2 \texttt{ then } n \\
\quad \texttt{else } \mathit{fib}(n-1) + \mathit{fib}(n-2) \\
\quad \texttt{end}
\end{array}
$$

Exercise: Histogram

```
hist(primes: Set[\ZZ32\]): ZZ32[10] = do
    result = array1[\ZZ32,10\](0)
    for p <- primes do
        atomic result[p MOD 10] += 1
    end
    result
end
```

$$
\begin{array}{l}
\mathit{hist}(\mathit{primes}\colon \mathrm{Set}[\![\mathbb{Z}32]\!])\colon \mathbb{Z}32[10] = \texttt{do} \\
\quad \mathit{result} = \mathit{array}_1[\![\mathbb{Z}32, 10]\!](0) \\
\quad \texttt{for}\ p \leftarrow \mathit{primes}\ \texttt{do} \\
\quad\quad \texttt{atomic}\ \ \mathit{result}[p\ \texttt{MOD}\ 10] \mathrel{+}= 1 \\
\quad \texttt{end} \\
\quad \mathit{result} \\
\texttt{end}
\end{array}
$$

Libraries and Lists

Goals

- Understand APIs and libraries
- Know where to look for libraries
- Write programs using lists

Components and APIs

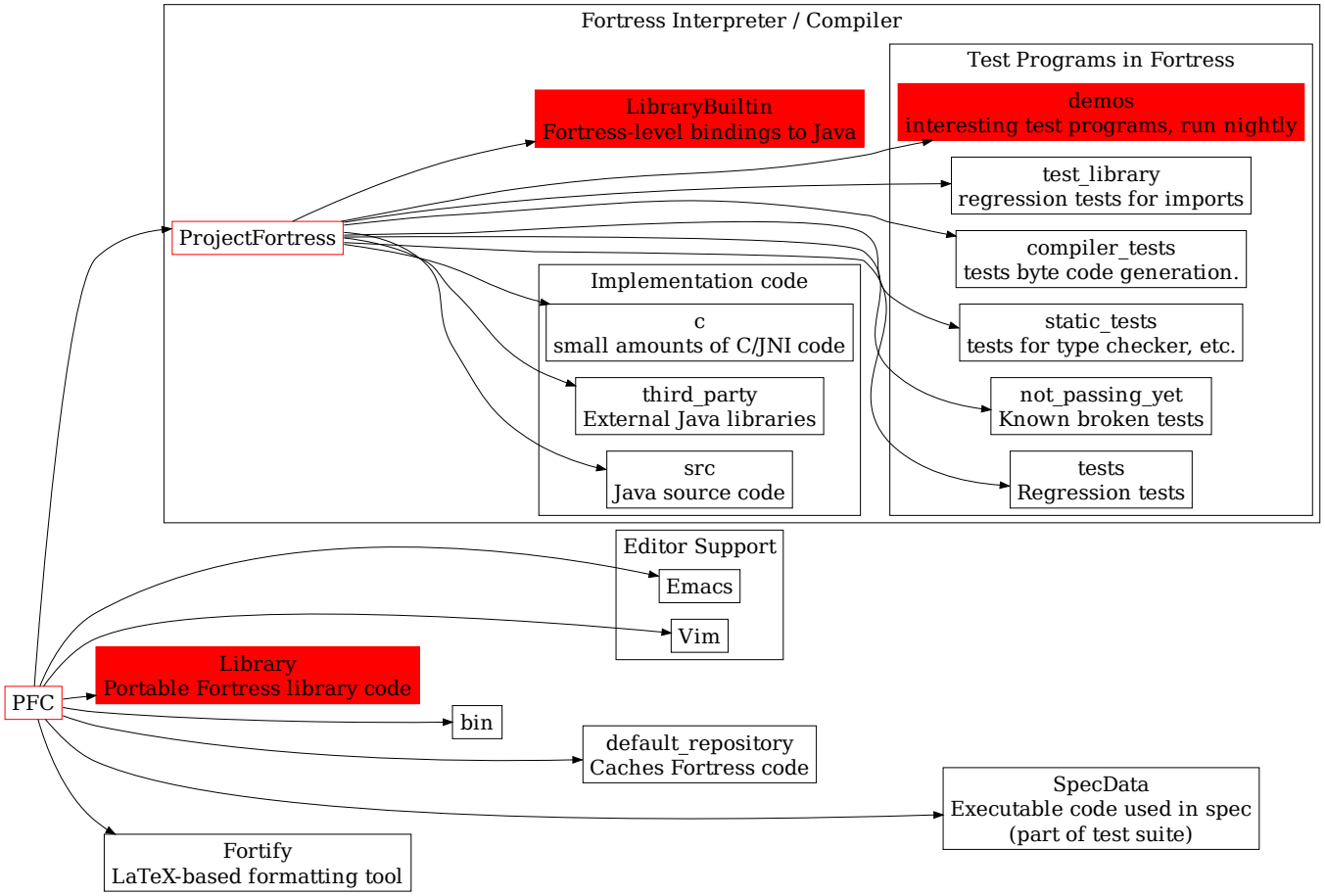
In ProjectFortress/hello.fss:

```
component hello
export Executable

run() = println("Hello, World!")

end
```

In Library/Executable.fsi:

```
api Executable
  run(): ()
end
```

Fortress Repository Structure



Some Fortress Libraries

ProjectFortress/LibraryBuiltin/FortressBuiltin.fsi

Library/FortressLibrary.fsi

Library/List.fsi

Library/Set.fsi

Library/Map.fsi

Library/IntMap.fsi

Library/Heap.fsi

Library/Sparse.fsi

Library/Relation.fsi

Library/QuickSort.fsi

Library/Shuffle.fsi

Library/CaseInsensitiveString.fsi

Library/File.fsi

Library/Format.fsi

Library/System.fsi

Library/Timing.fsi

Library/Constants.fsi

Library/Generator2.fsi

The List Library

```
trait List[\E\] extends { AnyList, LexicographicOrder[\List[\E\],E\] }
  getter extractLeft(): Maybe[\(E,List[\E\])\]
  getter extractRight(): Maybe[\(List[\E\],E)\]
  opr ||(self, other:List[\E\]): List[\E\]
  addLeft(e:E):List[\E\]
  addRight(e:E):List[\E\]
  split(): (List[\E\], List[\E\])
  zip[\F\](other: List[\F\]): Generator[\(E,F)\]
  filter(p: E -> Boolean): List[\E\]
end

opr <|[\E\] xs: E... |>: List[\E\]
opr BIG <|[\T\]|>:Comprehension[\T,List[\T\],List[\T\],List[\T\]\]
opr BIG <|[\T\] g:Generator[\T\]|>:List[\T\]
```

The List Library

```
trait List⟦E⟧ extends { AnyList, LexicographicOrder⟦List⟦E⟧, E⟧ }
  getter extractLeft(): Maybe⟦(E, List⟦E⟧)⟧
  getter extractRight(): Maybe⟦(List⟦E⟧, E)⟧
  opr || (self, other : List⟦E⟧): List⟦E⟧
  addLeft(e : E) : List⟦E⟧
  addRight(e : E) : List⟦E⟧
  split(): (List⟦E⟧, List⟦E⟧)
  zip⟦F⟧(other: List⟦F⟧): Generator⟦(E, F)⟧
  filter(p: E → Boolean): List⟦E⟧
end

opr ⟨⟦E⟧xs: E . . . ⟩: List⟦E⟧
opr BIG⟨⟦T⟧⟩ : Comprehension⟦T, List⟦T⟧, List⟦T⟧, List⟦T⟧⟧
opr BIG⟨⟦T⟧g : Generator⟦T⟧⟩ : List⟦T⟧
```

Inherited from LexicographicOrder

```
getter asString():String
getter isEmpty(): Boolean
getter nonEmpty(): Boolean
getter indexValuePairs(): Indexed[\(I,E),I\]
opr |self| : ZZ32 = self.size
opr =(self,other:List[\E\]): Boolean
opr <(self, other:List[\E\]): Boolean
opr >(self, other:List[\E\]): Boolean
opr >=(self, other:List[\E\]): Boolean
opr <=(self, other:List[\E\]): Boolean
opr [n:ZZ32]: E
opr [n:Range[\ZZ32\]]: List[\E\]
generate[\R\](r: Reduction[\R\], body: E->R): R
seq(self): SequentialGenerator[\E\]
opr IN(elt:T, self): Boolean
```

Inherited from LexicographicOrder

`getter` $asString() : \text{String}$

`getter` $isEmpty() : \text{Boolean}$

`getter` $nonEmpty() : \text{Boolean}$

`getter` $indexValuePairs() : \text{Indexed}[\![(I, E), I]\!]$

`opr` $|\texttt{self}| : \mathbb{Z}32$

`opr` $=(\texttt{self}, other : \text{List}[\![E]\!]) : \text{Boolean}$

`opr` $<(\texttt{self}, other : \text{List}[\![E]\!]) : \text{Boolean}$

`opr` $>(\texttt{self}, other : \text{List}[\![E]\!]) : \text{Boolean}$

`opr` $\geq(\texttt{self}, other : \text{List}[\![E]\!]) : \text{Boolean}$

`opr` $\leq(\texttt{self}, other : \text{List}[\![E]\!]) : \text{Boolean}$

`opr` $[n : \mathbb{Z}32] : E$

`opr` $[n : \text{Range}[\![\mathbb{Z}32]\!]] : \text{List}[\![E]\!]$

$generate[\![R]\!](r : \text{Reduction}[\![R]\!], body : E \to R) : R$

$seq(\texttt{self}) : \text{SequentialGenerator}[\![E]\!]$

`opr` $\in(elt : T, \texttt{self}) : \text{Boolean}$

Aggregate list constants

```
component listExample
export Executable


run(): () = do
    empty = <| |>
    println(empty.asString)
end

end
/.../listExample.fss:6:13-16:
    Operator <| |> is not defined.
```

```
opr <|[\E\] xs:E...|>

getter asString(): String
```

Aggregate list constants

```
component listExample
export Executable
import List.{ opr <| |> }

run(): () = do
    empty = <| |>
    println(empty.asString)
end

end

<| |>
```

```
opr <|[\E\] xs:E...|>

getter asString(): String
```

Aggregate list constants

```
component listExample
export Executable
import List.{...}

run(): () = do
    empty = <| |>
    println(empty)
end

end

<| |>
```

```
opr <|[\E\] xs:E...|>

getter asString(): String
```

Aggregate list constants

```
component listExample
export Executable
import List.{...}

run(): () = do
    empty = <| |>
    println("Answer: " empty)
end

end
```

String juxtaposition concatenates strings.

```
Answer: <| |>
```

Type information for list aggregates

```
empty = <| |>
strings = <| "Hello", "There" |>
println(empty || strings)
```

```
opr ||(self,
       other:List[\E\])
```

List append operator

```
<|Hello, There|>
```

Type information for list aggregates

```
empty : List[\String\] = <| |>
strings : List[\String\] = <| "Hello", "There" |>
println(empty || strings)
```

```
/.../listExample.fss:7:1-22:
RHS expression type ArrayList[\BOTTOM\] is not
assignable to LHS type List[\String\]
Context:
/.../listExample.fss:7:1-22:
/.../listExample.fss:7:1-22:
```

Type information for list aggregates

```
empty : List[\String\] = <|[\String\] |>
strings : List[\String\] = <| "Hello", "There" |>
println(empty || strings)
```

```
/.../listExample.fss:8:1-24:
RHS expression type ArrayList[\FlatString\] is not
assignable to LHS type List[\String\]
Context:
/.../listExample.fss:8:1-24:
/.../listExample.fss:8:1-24:
```

Type information for list aggregates

```
empty : List[\String\] = <|[\String\] |>
strings : List[\String\] = <|[\String\] "Hello",
                                       "There" |>
println(empty || strings)
```

```
<|Hello, There|>
```

Specify types in list aggregates in order to avoid type errors later

Parallel List Filter

```
noHello(xs: List[\String\]): List[\String\] =
    if |xs| = 0 then
        <|[\String\] |>
    elif |xs| = 1 then
        if xs[0] = "Hello" then
            <|[\String\] |>
        else
            xs
        end
    else
        (ys, zs) = xs.split()
        noHello(ys) || noHello(zs)
    end

strings = <|[\String\] "Hello", "there",
           "how", "are", "you", "?" |>
println(noHello(strings))
```

```
<|there, how, are, you, ?|>
```

Parallel List Filter

```
noHello(xs: List[\String\]): List[\String\] =
    if |xs| = 0 then
        <|[\String\] |>
    elif |xs| = 1 then
        if xs[0] = "Hello" then
            <|[\String\] |>
        else
            xs
        end
    else
        (ys, zs) = xs.split()
        noHello(ys) || noHello(zs)
    end
```

```
opr |self|: ZZ32
```

Recursive subdivision using `split()`.

Using case

```
noHello(xs: List[\String\]): List[\String\] =
    case |xs| of
        0 => xs
        1 => if xs[0] = "Hello" then
                 <|[\String\] |>
             else
                 xs
             end
        else =>
             (ys,zs) = xs.split()
             noHello(ys) || noHello(zs)
    end
```

Only computes `|xs|` once.

AND and OR

```
noHello(xs: List[\String\]): List[\String\] =
    if |xs| = 0 OR
       |xs| = 1 AND xs[0] = "Hello" then
        <|[\String\] |>
    elif |xs| = 1 then
        xs
    else
        (ys, zs) = xs.split()
        noHello(ys) || noHello(zs)
    end
```

Fails when xs is empty!

Shortcut AND: and OR:

```
noHello(xs: List[\String\]): List[\String\] =
    if |xs| = 0 OR:
       |xs| = 1 AND: xs[0] = "Hello" then
        <|[\String\] |>
    elif |xs| = 1 then
        xs
    else
        (ys, zs) = xs.split()
        noHello(ys) || noHello(zs)
    end
```

Use trailing :
for short circuit
evaluation

Extracting a Sublist

```
noHello(xs: List[\String\]): List[\String\] =
    if |xs| >= 1 AND: xs[0] = "Hello" then
        noHello(xs[ 1: ])
    elif |xs| <= 1 then
        xs
    else
        (ys, zs) = xs.split()
        noHello(ys) || noHello(zs)
    end
```

Index with a range

`l:u` or `l#size`

extracts a sublist

Cdring Down Sequentially

```
noHello(xs: List[\String\]): List[\String\] =
    if (hd,tl) <- xs.extractLeft then
        tl' = noHello(tl)
        if hd = "Hello" then
            tl'
        else
            tl'.addLeft(hd)
        end
    else
        xs
    end
getter extractLeft(): Maybe[\(E,List[\E\])\]
```

Not recommended!

Similar, but Parallel

```
noHello(xs: List[\String\]): List[\String\] =
    if (hd,tl) <- xs.extractLeft then
        (ys,zs) = tl.split()
        tl' = noHello(ys) || noHello(zs)
        if hd = "Hello" then
            tl'
        else
            tl'.addLeft(hd)
        end
    else
        xs
    end
```

Has useful parallelism

Use Built-in Filter Method

```
noHello(xs: List[\String\]): List[\String\] =
            xs.filter(fn (x) => x =/= "Hello")
```

- One line beats many!
- The parallelism is taken care of

Exercise: Merge Sort

```
sort(a: List[\ZZ32\]): List[\ZZ32\] =
    if |a| <= 1 then
        a
    else
        (b,c) = a.split()
        (sb,sc) = (sort(b),sort(c))
        merge(sb,sc)
    end
```

Parallel Merge

```
merge(xs:List[\ZZ32\], ys:List[\ZZ32\]):
                                        List[\ZZ32\] =
    if |xs| <= 1 then
        baseCase(xs,ys)
    elif |ys| <= 1 then
        baseCase(ys,xs)
    else
        x_m = |xs| DIV 2
        y_m = indexLarger(xs[x_m], ys)
        merge(xs[0#x_m], ys[0#y_m]) ||
            merge(xs[x_m#], ys[y_m#])
    end
```

Base Case of Merge

```
baseCase(xs:List[\ZZ32\], ys:List[\ZZ32\]):
                                        List[\ZZ32\] =
    if |xs| = 0 then
        ys
    else
        y_m = indexLarger(xs[0], ys)
        ys[0#y_m] || xs || ys[y_m#]
    end
```

indexLarger: Binary Search

```
indexLarger(x:ZZ32, xs:List[\ZZ32\]): ZZ32 = do
    (start: ZZ32, fin: ZZ32) := (0, |xs|)
    while fin > start do
        mid = (start + fin) DIV 2
        if x < xs[mid] then
            fin := mid
        else
            start := mid+1
        end
    end
    fin
  end
```

Comprehensions and Reductions

Goals

- Write list comprehensions
- Write reductions

List filter

```
noHello(xs: List[\String\]): List[\String\] = do
    r : List[\String\] := <|[\String\] |>
    for x <- xs do
        if x =/= "Hello" then
            atomic r ||= <|[\String\] x |>
        end
    end
    r
  end
```

Lists generate their elements in parallel

```
<|are, how, there, you, ?|>
```

List filter

```
noHello(xs: List[\String\]): List[\String\] = do
    r : List[\String\] := <|[\String\] |>
    for x <- seq(xs) do
        if x =/= "Hello" then
            r ||= <|[\String\] x |>
        end
    end
    r
  end
```

Can use seq on any parallel generator

```
<|there, how, are, you, ?|>
```

List filter

```
noHello(xs: List[\String\]): List[\String\] = do
    r : List[\String\] := <|[\String\] |>
    for x <- seq(xs), x=/="Hello" do
        r ||= <|[\String\] x |>
    end
    r
  end
```

```
<|there, how, are, you, ?|>
```

List filter

$$
\begin{array}{l}
\mathit{noHello}(\mathit{xs}\colon \mathrm{List}[\![\mathrm{String}]\!])\colon \mathrm{List}[\![\mathrm{String}]\!] = \texttt{do} \\
\quad r : \ \mathrm{List}[\![\mathrm{String}]\!] := \langle [\![\mathrm{String}]\!] \rangle \\
\quad \texttt{for}\ x \leftarrow \mathit{seq}(\mathit{xs}), x \neq \text{“}\texttt{Hello}\text{”}\ \ \texttt{do} \\
\quad\quad r \mathbin{\|}= \langle [\![\mathrm{String}]\!] x \rangle \\
\quad \texttt{end} \\
\quad r \\
\texttt{end}
\end{array}
$$

Formatting improves readability!

```
<|there, how, are, you, ?|>
```

Using a comprehension

```
noHello(xs: List[\String\]): List[\String\] =
    <|[\String\] x | x <- xs, x=/="Hello" |>
```

$$noHello(xs\colon \mathrm{List}[\![\mathrm{String}]\!])\colon \mathrm{List}[\![\mathrm{String}]\!] = \langle [\![\mathrm{String}]\!]x \mid x \leftarrow xs, x \neq \text{“}\texttt{Hello}\text{”} \rangle$$

```
<|there, how, are, you, ?|>
```

Containment

```
hasHello(xs: List[\String\]): Boolean =
  do
    r: Boolean := false
    for x <- xs, x="Hello" do
        r := true
    end
    r
  end

run(): () = do
  strings = <|[\String\] "Hello", "there",
              "how", "are", "you", "?" |>
  println(hasHello(strings))
end
```

```
true
```

Containment

```
hasHello(xs: List[\String\]): Boolean =
  do
    r: Boolean := false
    for x <- xs, x="Hello" do
        r := true
    end
    r
  end
```

No need for atomic here

true

Non-local Control Flow

```
hasHello(xs: List[\String\]): Boolean =
  label found
    for x <- xs, x="Hello" do
        exit found with true
    end
    false
  end
```

`label` and `exit` allow early return

`true`

Reduction

```
hasHello(xs: List[\String\]): Boolean =

    BIG OR[x <- xs] x="Hello"
```

Reduction operation combines values from individual iterations

$$hasHello(xs\colon \mathrm{List}[\![\mathrm{String}]\!])\colon \mathrm{Boolean} = \bigvee_{x \leftarrow xs} x = \text{“}\texttt{Hello}\text{”}$$

Use Built-In Operator

```
hasHello(xs: List[\String\]): Boolean =

    "Hello" IN xs
```

Defined for every generator

$$hasHello(xs\colon \mathrm{List}[\![\mathrm{String}]\!])\colon \mathrm{Boolean} =$$

$$\text{“}\texttt{Hello}\text{”} \in xs$$

Overlap

```
over(xs:List[\String\], ys:List[\String\]):Boolean =

    BIG OR[x <- xs] x IN ys
```

$$over(xs : \mathrm{List}[\![\mathrm{String}]\!], ys : \mathrm{List}[\![\mathrm{String}]\!]) : \mathrm{Boolean} = \bigvee_{x \leftarrow xs} x \in ys$$

Multigenerator Reductions

```
over(xs:List[\String\], ys:List[\String\]):Boolean =

    BIG OR[x <- xs, y <- ys] x=y
```

$$over(xs : \mathrm{List}[\![\mathrm{String}]\!], ys : \mathrm{List}[\![\mathrm{String}]\!]) : \mathrm{Boolean} = \bigvee_{\substack{x \leftarrow xs \\ y \leftarrow ys}} x = y$$

Fortress Type System

Traits are like Java™ interfaces

- Name may be used as a type
- May be generic (having static parameters)
- Multiple inheritance
- AND may contain method definitions
- AND may constrain subtyping relationships
  - `comprises` clause
  - `excludes` clause
- DO NOT have fields

Sample Trait

```
trait List⟦E⟧
    extends { AnyList, LexicographicOrder⟦List⟦E⟧, E⟧ }
    excludes { Number, HasRank, String }
  getter asString() : String = ...
end
```

Sample Trait

```
trait List⟦E⟧
    extends { AnyList, LexicographicOrder⟦List⟦E⟧, E⟧ }
    excludes { Number, HasRank, String }
  getter asString() : String = ...
end
```

- Generic: `List⟦E⟧`
- Multiple inheritance:
  `extends { AnyList, LexicographicOrder⟦List⟦E⟧, E⟧ }`
- `excludes` clause: `excludes { Number, HasRank, String }`
- Method definition:
  `getter asString() : String = ...`
- DO NOT have fields

Objects are Like Java™ classes

- Name may be used as a type
- May be generic (having static parameters)
- May contain field declarations
- May contain method definitions
- May be parameterized
  - If so, the parameters define a constructor function
  - If not, it's a "singleton" object
- CANNOT be extended

Sample Object

```
object FailCalled(s: String)
    extends UncheckedException
  toString(): String = "FAIL: " s
end

object DivisionByZero
    extends UncheckedException
  toString: String = "FAIL: Division by 0"
end
```

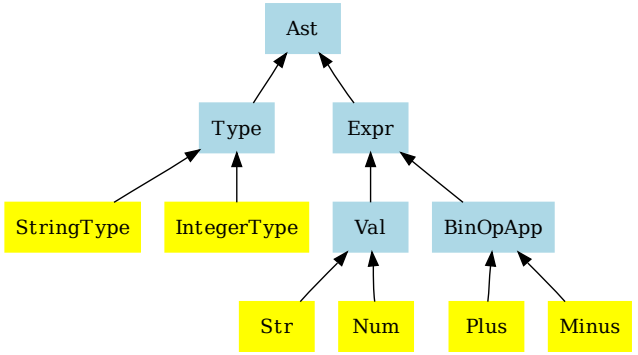
Sample Object

```
object FailCalled(s: String)
    extends UncheckedException
  toString(): String = "FAIL: " s
end

object DivisionByZero
    extends UncheckedException
  toString: String = "FAIL: Division by 0"
end
```

- Parameterized: FailCalled(*s*: String)
- Method definition: *toString*(): String = "FAIL: " *s*
- Singleton object: DivisionByZero
- Field definition: *toString*: String = "FAIL: Division by 0"

Sample Traits and Objects

```
trait Ast comprises { Type, Expr }
  getter asString(): String
end

trait Type extends Ast
    comprises { StringType, IntegerType }
    excludes Expr
end
object StringType extends Type end
object IntegerType extends Type end

trait Expr extends Ast comprises { Val, BinOpApp } excludes Type end
trait Val extends Expr comprises { Str, Num }
  getter asValue(): Object
end
object Str(string: String) extends Val
  getter asString() = string
  getter asValue() = string
end
```



Dynamic overloading in Fortress (I)

- Multiple functional declarations with the same name

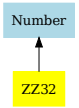
  $foo(a\colon \mathrm{Number}, b\colon \mathbb{Z}32)\colon \mathbb{Z}32$

  $foo(a\colon \mathbb{Z}32, b\colon \mathrm{Number})\colon \mathbb{Z}32$

- Overloading is chosen based on run-time argument types

  $foo(3.5, 7)$

  $foo(3, 7.9)$



Dynamic overloading in Fortress (I)

- Multiple functional declarations with the same name

  $foo(a\colon \mathrm{Number}, b\colon \mathbb{Z}32)\colon \mathbb{Z}32$

  $foo(a\colon \mathbb{Z}32, b\colon \mathrm{Number})\colon \mathbb{Z}32$

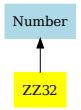
- Overloading is chosen based on run-time argument types

  $foo(3.5, 7)$

  $foo(3, 7.9)$

  $foo(3.5, 7.9)$

  $foo(3, 7)$



- Check for ambiguity at compile time (when declared)

  $foo(a\colon \mathrm{Number}, b\colon \mathbb{Z}32)\colon \mathbb{Z}32$

  $foo(a\colon \mathbb{Z}32, b\colon \mathrm{Number})\colon \mathbb{Z}32$

  $foo(a\colon \mathbb{Z}32, b : \mathbb{Z}32)\colon \mathbb{Z}32$

Dynamic overloading in Fortress (II)

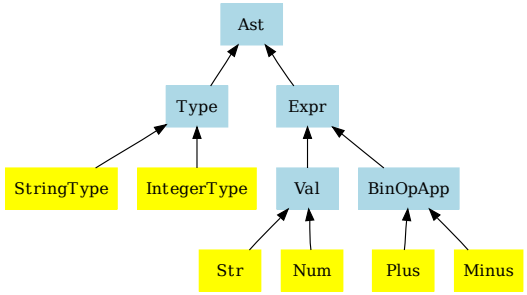
- A set of overloaded declarations is legal if for any pair among the set:
  - one declaration is more specific than the other; or
  - they are provably disjoint; or
  - another declaration covers the overlap between them.

$$foo(a\colon \mathrm{Number}, b\colon \mathbb{Z}32)\colon \mathbb{Z}32$$
$$foo(a\colon \mathbb{Z}32, b\colon \mathrm{Number})\colon \mathbb{Z}32$$
$$foo(a\colon \mathbb{Z}32, b : \mathbb{Z}32)\colon \mathbb{Z}32$$

Dynamic overloading in Fortress (II)

- A set of overloaded declarations is legal if for any pair among the set:
  - one declaration is more specific than the other; or
  - they are provably disjoint; or
  - another declaration covers the overlap between them.
- Both `excludes` and `comprises` clauses are useful here.

$eval\colon \mathrm{Expr} \to \mathrm{Object}$
$eval(v\colon \mathrm{Val})\colon \mathrm{Object}$
$eval(b\colon \mathrm{Plus})\colon \mathrm{Object}$
$eval(b\colon \mathrm{Minus})\colon \mathrm{Object}$



For More Information

- Join the Fortress community:

  http://projectfortress.sun.com/

- Check out and compile our svn repository
- Join the fortress-language mailing list
- Read and edit the Wiki

Sukyoung Ryu

sryu@cs.kaist.ac.kr

http://projectfortress.sun.com