Recursion and F# translations

The INFDEV@HR Team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Lecture topics

- Recursion (`let-rec`)
- F# translations of lambda programs so far

Recursive functions and `let-rec`

Recursive functions and `let-rec`

Idea

- The lambda calculus has no `while` loops
- This means that we need to emulate them with recursion

Recursive functions and `let-rec`

Idea

- This is a bit of an issue
- A function is just a lambda term, which does not have a name
- If the function does not have a name, how do we call it from its own body?

Recursive functions and `let-rec`

Idea

- We can define a recursive function as a function with an extra parameter
- Calling the extra parameter will result in calling the function itself

Recursive functions and `let-rec`

For example, the factorial function becomes

$(\lambda \texttt{f n} \rightarrow \texttt{if (n} = \texttt{0) then 1 else ((f (n} - \texttt{1))} \times \texttt{n))}$

Recursive functions and `let-rec`

- We now need an external operator that handles recursive functions properly
- This must ensure that a recursive function gets itself as a parameter, in an endless chain
- This combinator is known as `fixpoint` operator

```
(λf→((λx→(f (x x))) (λx→(f (x x)))))
```

Recursive functions and let-rec

```
((fix (λf n→if (n = 0) then 1 else (f (n − 1)
    ))) 2)
```

Recursive functions and `let-rec`

$$((\texttt{fix}\ (\lambda \texttt{f}\ \texttt{n} \rightarrow \texttt{if}\ (\texttt{n} = 0)\ \texttt{then}\ 1\ \texttt{else}\ (\texttt{f}\ (\texttt{n} - 1)))\ 2)$$

$$((\underline{\texttt{fix}}\ \underline{(\lambda \texttt{f}\ \texttt{n} \rightarrow \texttt{if}\ (\texttt{n} = 0)\ \texttt{then}\ 1\ \texttt{else}\ (\texttt{f}\ (\texttt{n} - 1)))})\ 2)$$

Recursive functions and `let-rec`

$$\boxed{\texttt{((}\underline{\texttt{fix}}\ \underline{\texttt{(}\lambda\texttt{f n}\rightarrow\texttt{if (n = 0) then 1 else (f (n - 1)))}}\texttt{) 2)}}$$

Recursive functions and `let-rec`

$$((\underline{\texttt{fix}}\ \underline{(\lambda \texttt{f n} \rightarrow \texttt{if (n = 0) then 1 else (f (n - 1)))}})\ \texttt{2})$$

$$((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else (... (n - 1))) 2})$$

Recursive functions and `let-rec`

$$\boxed{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 2)}}$$

Recursive functions and `let-rec`

$\boxed{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 2)}}$

$\boxed{\underline{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 2)}}}$

Recursive functions and let-rec

$$\underline{((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else (...}\quad \texttt{(n} - \texttt{1))) 2)}}$$

Recursive functions and `let-rec`

$\underline{((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else (... (n - 1))) 2)}}$

`if (2 = 0) then 1 else (... (2 - 1))`

```
if (2 = 0) then 1 else (... (2 - 1))
```

Recursive functions and `let-rec`


```
if (2 = 0) then 1 else (... (2 - 1))
```

```
if (2 = 0) then 1 else (... (2 - 1))
```

Recursive functions and `let-rec`


```
if (2 = 0) then 1 else (... (2 - 1))
```

Recursive functions and `let-rec`


```
if (2 = 0) then 1 else (... (2 - 1))
```

```
if FALSE then 1 else (... (2 - 1))
```

Recursive functions and `let-rec`


```
if FALSE then 1 else (... (2 − 1))
```

```
if FALSE then 1 else (... (2 - 1))
```

```
if FALSE then 1 else (... (2 - 1))
```

$\underline{\texttt{if FALSE then 1 else (...} \quad \texttt{(2} - \texttt{1))}}$

Recursive functions and `let-rec`


```
if FALSE then 1 else (...  (2 - 1))
```

```
(... (2 - 1))
```

Recursive functions and `let-rec`

```
(... (2 - 1))
```

```
(... (2 - 1))
```

```
((fix (λf n→if (n = 0) then 1 else (f (n - 1)))) (2
    - 1))
```

Recursive functions and `let-rec`

$$\boxed{\texttt{((}\underline{\texttt{fix}}\ \underline{(\lambda \texttt{f n} \rightarrow \texttt{if (n = 0) then 1 else (f (n - 1)))}}\texttt{) (2 - 1))}}$$

Recursive functions and let-rec

$$((\underline{\texttt{fix}}\ \underline{(\lambda \texttt{f n} \rightarrow \texttt{if (n = 0) then 1 else (f (n} - \texttt{1)))}})\ (\texttt{2} - \texttt{1}))$$

$$((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else (... (n} - \texttt{1)))}\ (\texttt{2} - \texttt{1}))$$

Recursive functions and `let-rec`


```
((λn→if (n = 0) then 1 else (... (n − 1))) (2
     − 1))
```

Recursive functions and `let-rec`

$$((\lambda n \rightarrow \texttt{if (n = 0) then 1 else (... (n - 1))) (2 - 1))}$$

$$((\lambda n \rightarrow \texttt{if (n = 0) then 1 else (... (n - 1)))} \ \underline{\texttt{(2 - 1)}}\texttt{)}$$

Recursive functions and `let-rec`


```
((λn→if (n = 0) then 1 else (... (n - 1)))
    (2 - 1))
```

Recursive functions and `let-rec`

$$((\lambda n \rightarrow \text{if } (n = 0) \text{ then } 1 \text{ else } (\ldots (n - 1)))\ \underline{(2 - 1)})$$

$$((\lambda n \rightarrow \text{if } (n = 0) \text{ then } 1 \text{ else } (\ldots (n - 1)))\ 1)$$

Recursive functions and let-rec

$$\boxed{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 1)}}$$

Recursive functions and `let-rec`

$\boxed{((\lambda n \rightarrow \texttt{if}\ (n = 0)\ \texttt{then}\ 1\ \texttt{else}\ (\ldots\ (n - 1)))\ 1)}$

$\boxed{\underline{((\lambda n \rightarrow \texttt{if}\ (n = 0)\ \texttt{then}\ 1\ \texttt{else}\ (\ldots\ (n - 1)))\ 1)}}$

Recursive functions and `let-rec`

$$\boxed{\underline{((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else (... (n - 1))) 1)}}}$$

Recursive functions and `let-rec`

$\underline{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 1)}}$

`if (1 = 0) then 1 else (... (1 - 1))`

Recursive functions and `let-rec`


```
if (1 = 0) then 1 else (... (1 - 1))
```

Recursive functions and `let-rec`


```
if (1 = 0) then 1 else (... (1 - 1))
```

```
if (1 = 0) then 1 else (... (1 - 1))
```

Recursive functions and `let-rec`

```
if (1 = 0) then 1 else (... (1 - 1))
```

Recursive functions and `let-rec`


```
if (1 = 0) then 1 else (... (1 - 1))
```

```
if FALSE then 1 else (... (1 - 1))
```

Recursive functions and `let-rec`

```
if FALSE then 1 else (... (1 - 1))
```

```
if FALSE then 1 else (... (1 − 1))
```

```
if FALSE then 1 else (... (1 − 1))
```

Recursive functions and let-rec

```
if FALSE then 1 else (...  (1 − 1))
```

Recursive functions and `let-rec`


```
if FALSE then 1 else (...  (1 − 1))
```

```
(...  (1 − 1))
```

Recursive functions and `let-rec`

```
(... (1 - 1))
```

Recursive functions and `let-rec`


```
(... (1 - 1))
```

```
((fix (λf n→if (n = 0) then 1 else (f (n - 1)))) (1
    - 1))
```

Recursive functions and `let-rec`

$\boxed{\texttt{((}\underline{\texttt{fix}}\ \underline{\texttt{(}\lambda\texttt{f n}\rightarrow\texttt{if (n = 0) then 1 else (f (n - 1)))}}\texttt{) (1 - 1))}}$

Recursive functions and let-rec

$$((\underline{\texttt{fix}}\ \underline{(\lambda \texttt{f n} \rightarrow \texttt{if (n = 0) then 1 else (f (n - 1))}})))\ (1 - 1))$$

$$((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else (... (n - 1)))}\ (1 - 1))$$

Recursive functions and let-rec

$$((\lambda \mathtt{n} \rightarrow \mathtt{if}\ (\mathtt{n} = 0)\ \mathtt{then}\ 1\ \mathtt{else}\ (\ldots\ (\mathtt{n} - 1)))\ (1 - 1))$$

Recursive functions and let-rec

$$((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else (... (n - 1))) (1 - 1))}$$

$$((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else (... (n - 1)))}\ \underline{\texttt{(1 - 1)}})$$

Recursive functions and `let-rec`

Recursion and F# translations

The INFDEV@HR Team

Introduction

Recursive functions and let-rec

Translating to F#

Conclusion

```
((λn→if (n = 0) then 1 else (... (n - 1)))
    (1 - 1))
```

Recursive functions and let-rec

$$((\lambda n \rightarrow \texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } (\ldots (n - 1))) \; \underline{(1 - 1)})$$

$$((\lambda n \rightarrow \texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } (\ldots (n - 1))) \; 0)$$

Recursive functions and `let-rec`

$$\boxed{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 0)}}$$

Recursive functions and `let-rec`

$\boxed{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 0)}}$

$\boxed{\underline{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 0)}}}$

Recursive functions and `let-rec`

$$\underline{((\lambda \mathtt{n} \rightarrow \mathtt{if}\ (\mathtt{n} = 0)\ \mathtt{then}\ 1\ \mathtt{else}\ (\ldots\ (\mathtt{n} - 1)))\ 0)}$$

Recursive functions and `let-rec`

$\underline{\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else (... (n - 1))) 0)}}$

`if (0 = 0) then 1 else (... (0 - 1))`

Recursive functions and let-rec


```
if (0 = 0) then 1 else (... (0 - 1))
```

Recursive functions and `let-rec`


```
if (0 = 0) then 1 else (... (0 - 1))
```

```
if (0 = 0) then 1 else (... (0 - 1))
```

Recursive functions and `let-rec`

$$\texttt{if}\ \underline{(0 = 0)}\ \texttt{then 1 else}\ (\ldots\ (0 - 1))$$

Recursive functions and `let-rec`

$$\texttt{if } \underline{(0 = 0)} \texttt{ then 1 else (... } (0 - 1))$$

$$\texttt{if TRUE then 1 else (... } (0 - 1))$$

Recursive functions and `let-rec`


```
if TRUE then 1 else (... (0 − 1))
```

Recursive functions and `let-rec`

```
if TRUE then 1 else (... (0 - 1))
```

```
if TRUE then 1 else (... (0 - 1))
```

$$\boxed{\underline{\texttt{if TRUE then 1 else (... (0} - \texttt{1))}}}$$

Recursive functions and let-rec

```
if TRUE then 1 else (...  (0 − 1))
```

```
1
```

We can now try our hand at a factorial computation

$$\texttt{((fix (}\lambda\texttt{f n}\rightarrow\texttt{if (n = 0) then 1 else ((f (n} - \texttt{1))} \times \texttt{n))) 2)}$$

Recursive functions and `let-rec`

$$((\texttt{fix}\ (\lambda \texttt{f}\ \texttt{n}\rightarrow \texttt{if}\ (\texttt{n} = 0)\ \texttt{then}\ 1\ \texttt{else}\ ((\texttt{f}\ (\texttt{n} - 1))\ \times\ \texttt{n})))\ 2)$$

Recursive functions and `let-rec`

$$\texttt{((fix } (\lambda\texttt{f n}\rightarrow\texttt{if (n} = \texttt{0) then 1 else ((f (n} - \texttt{1))} \times \texttt{n))) 2)}$$

$$\texttt{((}\underline{\texttt{fix}}\ \underline{\lambda\texttt{f}\rightarrow}\ \underline{\texttt{n}}\rightarrow \underline{\texttt{if (n} = \texttt{0) then 1 else ((f (n} - \texttt{1))} \times \texttt{n)}}\texttt{) 2)}$$

Recursive functions and `let-rec`

$$\boxed{\begin{array}{l}((\underline{\texttt{fix}}\ \underline{\lambda\texttt{f}\rightarrow}\ \underline{\texttt{n}}\rightarrow \\ \quad \underline{\texttt{if}\ (\texttt{n} = \texttt{0})\ \texttt{then 1 else}\ ((\texttt{f}\ (\texttt{n} - \texttt{1}))\ \times\ \texttt{n})})\ \ \texttt{2})\end{array}}$$

Recursive functions and let-rec

```
((fix λf→ n→
    if (n = 0) then 1 else ((f (n − 1)) × n)) 2)
```

```
((λn→if (n = 0) then 1 else ((... (n − 1)) ×
    n)) 2)
```

Recursive functions and `let-rec`

$$((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else ((... (n - 1))} \times \texttt{n)) 2)}$$

$$\texttt{((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else ((... (n} - \texttt{1))} \times \texttt{n)) 2)}$$

$$\texttt{(}\underline{\lambda\texttt{n}\rightarrow}\ \rightarrow\underline{\texttt{if (n = 0) then 1 else ((... (n} - \texttt{1))} \times \texttt{n)}}\ \underline{\texttt{2}}\texttt{)}$$

Recursive functions and `let-rec`

$$(\underline{\lambda \mathtt{n} \rightarrow}\ \rightarrow \underline{\mathtt{if\ (n = 0)\ then\ 1\ else\ ((\ldots\ \ (n - 1))\ \times\ n)}}\ \underline{\mathtt{2}})$$

Recursive functions and let-rec

$(\underline{\lambda n \rightarrow} \rightarrow \underline{\texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } ((\ldots \ (n - 1)) \times n)} \ \underline{2})$

$\texttt{if } (2 = 0) \texttt{ then } 1 \texttt{ else } ((\ldots \ (2 - 1)) \times 2)$

Recursive functions and `let-rec`

$\boxed{\texttt{if (2 = 0) then 1 else ((... (2 - 1))} \times \texttt{2)}}$

Recursive functions and `let-rec`

$\texttt{if (2 = 0) then 1 else ((... (2 - 1))} \times \texttt{2)}$

$\texttt{if } \underline{\texttt{(2 = 0)}} \texttt{ then 1 else ((... (2 - 1))} \times \texttt{2)}$

Recursive functions and `let-rec`

```
if (2 = 0) then 1 else ((... (2 - 1)) × 2)
```

Recursive functions and `let-rec`

$\texttt{if } \underline{(2 = 0)} \texttt{ then 1 else } ((\ldots\ (2 - 1)) \times 2)$

$\texttt{if FALSE then 1 else } ((\ldots\ (2 - 1)) \times 2)$

Recursive functions and `let-rec`

$$\boxed{\texttt{if FALSE then 1 else ((... (2 - 1))} \times \texttt{2)}}$$

Recursive functions and let-rec

$\boxed{\texttt{if FALSE then 1 else ((... (2} - \texttt{1))} \times \texttt{2)}}$

$\boxed{\underline{\texttt{if FALSE then 1 else ((... (2} - \texttt{1))} \times \texttt{2)}}}$

Recursive functions and let-rec

$\underline{\texttt{if FALSE then 1 else ((... (2 - 1)) } \times \texttt{ 2)}}$

Recursive functions and `let-rec`

$\boxed{\underline{\texttt{if FALSE then 1 else ((...} \;\; \texttt{(2} - \texttt{1))} \times \texttt{2)}}}$

$\boxed{\texttt{((...} \;\; \texttt{(2} - \texttt{1))} \times \texttt{2)}}$

Recursive functions and let-rec

$$\boxed{((\ldots\ (2\ -\ 1))\ \times\ 2)}$$

$((\ldots \; (2 - 1)) \times 2)$

$(((\underline{\texttt{fix}\ \lambda f \rightarrow\ n \rightarrow}$
$\underline{\texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } ((f\ (n - 1)) \times n)}) \; (2 - 1)) \times 2)$

Recursive functions and `let-rec`

$$(((\underline{\texttt{fix}}\ \underline{\lambda \texttt{f}\rightarrow}\ \underline{\texttt{n}}\rightarrow \underline{\texttt{if}\ (\texttt{n} = 0)\ \texttt{then}\ 1\ \texttt{else}\ ((\texttt{f}\ (\texttt{n} - 1))\ \times\ \texttt{n}}))\ (2 - 1))\ \times\ 2)$$

Recursive functions and `let-rec`

$$\texttt{(((}\underline{\texttt{fix}}\ \underline{\lambda\texttt{f}\rightarrow}\ \underline{\texttt{n}}\rightarrow\ \underline{\texttt{if (n = 0) then 1 else ((f (n - 1))} \times \texttt{n)}}\texttt{) (2 - 1))} \times \texttt{2)}$$

$$\texttt{(((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else ((... (n - 1))} \times \texttt{n)) (2 - 1))} \times \texttt{2)}$$

Recursive functions and let-rec

$$(((\lambda n \rightarrow \texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } ((\ldots\ (n - 1)) \times n))\ (2 - 1)) \times 2)$$

Recursive functions and `let-rec`

$(((\lambda n \rightarrow \texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } ((\ldots (n - 1)) \times n)) (2 - 1)) \times 2)$

$(((\lambda n \rightarrow \texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } ((\ldots (n - 1)) \times n)) \underline{(2 - 1)}) \times 2)$

Recursive functions and `let-rec`

$$\boxed{\texttt{(((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else ((... (n - 1)) × n))}\ \underline{\texttt{(2 - 1)}}\texttt{) × 2)}}$$

Recursive functions and `let-rec`

$$(((\lambda n \rightarrow \texttt{if}\ (n = 0)\ \texttt{then}\ 1\ \texttt{else}\ ((\ldots\ (n - 1)) \times n))\ \underline{(2 - 1)})\ \times\ 2)$$

$$(((\lambda n \rightarrow \texttt{if}\ (n = 0)\ \texttt{then}\ 1\ \texttt{else}\ ((\ldots\ (n - 1)) \times n))\ 1)\ \times\ 2)$$

Recursive functions and `let-rec`

$$\texttt{(((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else ((... (n} - \texttt{1))} \times \texttt{n)) 1)} \times \texttt{2)}$$

$$(((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else ((... (n} - \texttt{1))} \times \texttt{n)) 1)} \times \texttt{2)}$$

$$\texttt{((}\underline{\lambda \texttt{n} \rightarrow}\ \rightarrow \underline{\texttt{if (n = 0) then 1 else ((... (n} - \texttt{1))} \times \texttt{n)}}\ \underline{\texttt{1}}\texttt{)} \times \texttt{2)}$$

Recursive functions and `let-rec`

$$\texttt{((}\underline{\lambda\texttt{n}\rightarrow}\ \rightarrow\underline{\texttt{if (n = 0) then 1 else ((... (n - 1)) }\times\texttt{ n)}}\ \underline{\texttt{1}}\texttt{) }\times\texttt{ 2)}$$

Recursive functions and let-rec

((λn→ →if (n = 0) then 1 else ((... (n − 1)) × n) 1) × 2)

(if (1 = 0) then 1 else ((... (1 − 1)) × 1) × 2)

Recursive functions and `let-rec`


```
(if (1 = 0) then 1 else ((... (1 − 1)) × 1) ×
    2)
```

Recursive functions and `let-rec`

$$\texttt{(if (1 = 0) then 1 else ((... (1 - 1))} \times \texttt{1)} \times \texttt{2)}$$

$$\texttt{(if } \underline{\texttt{(1 = 0)}} \texttt{ then 1 else ((... (1 - 1))} \times \texttt{1)} \times \texttt{2)}$$

Recursive functions and `let-rec`

$$\boxed{\texttt{(if } \underline{\texttt{(1 = 0)}} \texttt{ then 1 else ((... (1 - 1))} \times \texttt{1)} \times \texttt{2)}}$$

Recursive functions and `let-rec`

$\texttt{(if } \underline{\texttt{(1} = \texttt{0)}} \texttt{ then 1 else ((... (1} - \texttt{1))} \times \texttt{1)} \times \texttt{2)}$

$\texttt{(if FALSE then 1 else ((... (1} - \texttt{1))} \times \texttt{1)} \times \texttt{2)}$

$$\texttt{(if FALSE then 1 else ((... (1} - \texttt{1))} \times \texttt{1)} \times \texttt{2)}$$

Recursive functions and `let-rec`

$$\texttt{(if FALSE then 1 else ((... (1} - \texttt{1))} \times \texttt{1)} \times \texttt{2)}$$

$$\texttt{(}\underline{\texttt{if FALSE then 1 else ((... (1} - \texttt{1))} \times \texttt{1)}} \times \texttt{2)}$$

Recursive functions and `let-rec`

$$\texttt{(}\underline{\texttt{if FALSE then 1 else ((...}\quad\texttt{(1} - \texttt{1))} \times \texttt{1)}} \times \texttt{2)}$$

Recursive functions and `let-rec`

$$\boxed{(\underline{\texttt{if FALSE then 1 else ((... (1} - \texttt{1))} \times \texttt{1)}} \times \texttt{2)}}$$

$$\boxed{\texttt{(((... (1} - \texttt{1))} \times \texttt{1)} \times \texttt{2)}}$$

Recursive functions and `let-rec`

$$\boxed{(((\ldots (1 - 1)) \times 1) \times 2)}$$

Recursive functions and `let-rec`

$\texttt{(((...\ (1\ -\ 1))\ \times\ 1)\ \times\ 2)}$

$\texttt{((((}\underline{\texttt{fix}}\ \underline{\lambda\texttt{f}\rightarrow}\ \underline{\texttt{n}}\rightarrow$
$\underline{\texttt{if (n = 0) then 1 else ((f (n - 1))}\ \times\ \texttt{n)}}\texttt{) (1 -}$
$\texttt{1))}\ \times\ \texttt{1)}\ \times\ \texttt{2)}$

Recursive functions and `let-rec`

$$((((\underline{\texttt{fix}}\ \underline{\lambda \texttt{f} \rightarrow}\ \underline{\texttt{n}}\rightarrow \underline{\texttt{if (n = 0) then 1 else ((f (n - 1))} \times \texttt{n)}})\ \texttt{(1 - 1))} \times \texttt{1)} \times \texttt{2)}$$

Recursive functions and `let-rec`

$$((((\underline{\texttt{fix}}\ \underline{\lambda \texttt{f}\rightarrow}\ \underline{\texttt{n}\rightarrow \texttt{if (n} = \texttt{0) then 1 else ((f (n} - \texttt{1))} \times \texttt{n)}})\ \texttt{(1} - \texttt{1))} \times \texttt{1)} \times \texttt{2)}$$

$$((((\lambda \texttt{n}\rightarrow \texttt{if (n} = \texttt{0) then 1 else ((... (n} - \texttt{1))} \times \texttt{n))}\ \texttt{(1} - \texttt{1))} \times \texttt{1)} \times \texttt{2)}$$

Recursive functions and `let-rec`

$$\texttt{((((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else ((... (n - 1))} \times \texttt{n)) (1 - 1))} \times \texttt{1)} \times \texttt{2)}$$

Recursive functions and `let-rec`


```
((((λn→if (n = 0) then 1 else ((... (n − 1))
   × n)) (1 − 1)) × 1) × 2)
```

```
((((λn→if (n = 0) then 1 else ((... (n − 1))
   × n)) (1 − 1)) × 1) × 2)
```

Recursive functions and `let-rec`

Recursion and F# translations

The INFDEV@HR Team

Introduction

Recursive functions and let-rec

Translating to F#

Conclusion

$$\texttt{((((}\lambda\texttt{n}\rightarrow\texttt{if (n = 0) then 1 else ((... (n - 1))} \times \texttt{ n))} \underline{\texttt{(1 - 1)}}\texttt{)} \times \texttt{ 1)} \times \texttt{ 2)}$$

Recursive functions and let-rec

$$((((\lambda n \rightarrow \texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } ((\ldots (n - 1)) \times n)) \ \underline{(1 - 1))} \times 1) \times 2)$$

$$((((\lambda n \rightarrow \texttt{if } (n = 0) \texttt{ then } 1 \texttt{ else } ((\ldots (n - 1)) \times n)) \ 0) \times 1) \times 2)$$

Recursive functions and `let-rec`

$$((((\lambda \texttt{n} \rightarrow \texttt{if (n = 0) then 1 else ((... (n} - \texttt{1))} \times \texttt{n)) 0)} \times \texttt{1)} \times \texttt{2)}$$

Recursive functions and `let-rec`

$$((((\lambda \mathtt{n} \rightarrow \mathtt{if}\ (\mathtt{n} = 0)\ \mathtt{then}\ 1\ \mathtt{else}\ ((\ldots\ (\mathtt{n} - 1)) \times \mathtt{n}))\ 0) \times 1) \times 2)$$

$$(((\underline{\lambda \mathtt{n} \rightarrow}\ \rightarrow \underline{\mathtt{if}\ (\mathtt{n} = 0)\ \mathtt{then}\ 1\ \mathtt{else}\ ((\ldots\ \ (\mathtt{n} - 1)) \times \mathtt{n})}\ \underline{0}) \times 1) \times 2)$$

Recursive functions and `let-rec`

$$(((\underline{\lambda \mathtt{n} \rightarrow}\ \rightarrow \underline{\mathtt{if}\ (\mathtt{n} = 0)\ \mathtt{then}\ 1\ \mathtt{else}\ ((\ldots\ (\mathtt{n} - 1))\ \times\ \mathtt{n})}\ \underline{0})\ \times\ 1)\ \times\ 2)$$

Recursive functions and let-rec

$$(((\underline{\lambda n \rightarrow} \rightarrow \underline{\text{if } (n = 0) \text{ then } 1 \text{ else } ((\ldots\ (n - 1)) \times n)}\ \underline{0}) \times 1) \times 2)$$

$$((\text{if } (0 = 0) \text{ then } 1 \text{ else } ((\ldots\ (0 - 1)) \times 0) \times 1) \times 2)$$

Recursive functions and let-rec

```
((if (0 = 0) then 1 else ((... (0 − 1)) × 0)
    × 1) × 2)
```

Recursive functions and `let-rec`

$$((\texttt{if}\ (0 = 0)\ \texttt{then}\ 1\ \texttt{else}\ ((\ldots\ (0 - 1)) \times 0) \times 1) \times 2)$$

$$((\texttt{if}\ \underline{(0 = 0)}\ \texttt{then}\ 1\ \texttt{else}\ ((\ldots\ (0 - 1)) \times 0) \times 1) \times 2)$$

Recursive functions and `let-rec`

$$\boxed{\texttt{((if } \underline{\texttt{(0 = 0)}} \texttt{ then 1 else ((... (0 - 1))} \times \texttt{0)} \times \texttt{1)} \times \texttt{2)}}$$

Recursive functions and `let-rec`

$$((\texttt{if}\ \underline{(0 = 0)}\ \texttt{then}\ 1\ \texttt{else}\ ((\ldots\ (0 - 1)) \times 0) \times 1) \times 2)$$

$$((\texttt{if}\ \texttt{TRUE}\ \texttt{then}\ 1\ \texttt{else}\ ((\ldots\ (0 - 1)) \times 0) \times 1) \times 2)$$

Recursive functions and let-rec


```
((if TRUE then 1 else ((... (0 − 1)) × 0) ×
    1) × 2)
```

Recursive functions and `let-rec`

$((\texttt{if TRUE then 1 else }((\ldots\ (0 - 1)) \times 0) \times 1) \times 2)$

$((\underline{\texttt{if TRUE then 1 else }((\ldots\ (0 - 1)) \times 0)} \times 1) \times 2)$

Recursive functions and `let-rec`

$$(\underline{(\texttt{if TRUE then 1 else }((\ldots\ (0 - 1)) \times 0)} \times 1) \times 2)$$

Recursive functions and `let-rec`

$$((\underline{\texttt{if TRUE then 1 else ((... (0} - \texttt{1))} \times \texttt{0)}} \times \texttt{1)} \times \texttt{2)}$$

$$((\texttt{1} \times \texttt{1)} \times \texttt{2)}$$

Recursive functions and `let-rec`


```
((1 × 1) × 2)
```

Recursive functions and `let-rec`

$$((1 \times 1) \times 2)$$

$$(\underline{(1 \times 1)} \times 2)$$

Recursive functions and `let-rec`

$$(\underline{(1 \times 1)} \times 2)$$

Recursive functions and `let-rec`

$$(\underline{(1 \ \times \ 1)} \ \times \ 2)$$

$$(1 \ \times \ 2)$$

Recursive functions and let-rec

$$(1 \times 2)$$

Recursive functions and `let-rec`

(1×2)

$\underline{(1 \times 2)}$

Recursive functions and `let-rec`

$$\boxed{\underline{(1 \times 2)}}$$

Recursive functions and `let-rec`

$\underline{(1 \times 2)}$

2

Translating to F#

Translating to F#

Overview

- Each and every one of the constructs we have seen so far has a direct translation in F#[a]
- All constructs have exactly the same behaviour as in the lambda calculus, but with a slightly less mathematical syntax for ASCII keyboards
- A few operators are menemonically friendlier or just plain more readable, but the essence remains exactly the same
- F# is indentation-sensitive, like Python: pay a lot of attention to how terms are indented!

[a] Haskell is a bit different, so we leave it for later

Integers, booleans, floats, strings have the usual meaning, both in the lambda calculus, F#, and the languages you are used to:

```
((2 + 3) - 4)
```

```
(true && false)
```

Translating to F#

- Conditionals behave just like in the lambda calculus
- This means that they return the evaluation of either of the two branches
- This differs from imperative languages, where we just jump into either of the two branches

```
if (true && false) then
  0
else
  1
```

Translating to F#

Functions look very similar, with `fun` instead of λ

```
(fun x f -> (f x))
```

Translating to F#

Functions look very similar, with `fun` instead of λ

```
(fun x f -> (f x))
```

Just like function application

```
(((fun x f -> (f x)) 3) (fun x -> (3 + x)))
```

Translating to F#

We can give names to functions, and code becomes much prettier as a result

```
let apply =
  fun x f -> (f x)
((apply 3) (fun x -> (3 + x)))
```

Translating to F#

We can also give names to recursive functions by using `let rec` instead of `let`, and code becomes much more readable than with `fix`

```
let rec fact =
  fun n ->
    if (n = 0) then
      1
    else
      ((fact (n - 1)) * n)

(fact 2)
```

Translating to F#

We can define tuples by just putting a comma between the values, with or without nesting for more than two values is done for us

```
(1, true)
```

Pairs

- Nested tuples are the same as in the lambda calculus, and of course they also work in F#
- They are, in essence, pairs of pairs, such as: `(1,(2,(3,4)))` or `(((1,2),3),4)`, which are not the same
- Non-nested tuples are a slight extension for practical ease of work in F#
- They are, in essence, tuples with as many elements as we want, such as: `(1,2,3,4)`
- All the examples in this slide are valid in F#, but are not identical objects

Translating to F#

- Functions such as π_1 and pi_2, which both extract one item of a pair, also exist in F#
- They are called, respectively, `fst` and `snd`

```
(fst (1, true))
```

```
(snd (1, true))
```

Translating to F#

- Of course F#, just like the lambda calculus, imposes no limitation on composition
- Why not build a tuple of functions, then? Why not indeed!

Translating to F#

- Of course F#, just like the lambda calculus, imposes no limitation on composition
- Why not build a tuple of functions, then? Why not indeed!

```
((fun x -> (1 + x)), (fun x -> (2 * x)))
```

Translating to F#

- Of course F#, just like the lambda calculus, imposes no limitation on composition
- Why not build a tuple of functions, then? Why not indeed!

```
((fun x -> (1 + x)), (fun x -> (2 * x)))
```

Could we also build a tuple of tuples of numbers and functions returning tuples of functions?

Translating to F#

- Of course F#, just like the lambda calculus, imposes no limitation on composition
- Why not build a tuple of functions, then? Why not indeed!

```
((fun x -> (1 + x)), (fun x -> (2 * x)))
```

Could we also build a tuple of tuples of numbers and functions returning tuples of functions?

Yes, but we are not going to do it :)

Translating to F#

- F# also offers built-in discriminated unions
- Functions such as `inl` and `inr`, which both embed one item into the union, also exist in F#
- They are called, respectively, `Choice1Of2` and `Choice2Of2`

```
(Choice1Of2 1)
```

```
(Choice2Of2 true)
```

Translating to F#

We can, of course, perform matches on discriminated unions

```
match (Choice1Of2 1) with
| Choice1Of2 x ->
    (Choice1Of2 x)
| Choice2Of2 y ->
  (Choice2Of2 y)
```

Translating to F#

We can, of course, perform matches on discriminated unions

```
match (Choice1Of2 1) with
| Choice1Of2 x ->
   (Choice1Of2 x)
| Choice2Of2 y ->
  (Choice2Of2 y)
```

```
let i =
  match (Choice1Of2 1) with
  | Choice1Of2 x ->
     x
  | Choice2Of2 y ->
    0

(i * 2)
```

Translating to F#

Discriminated unions, and their corresponding matches, can be nested as deep as we need

```
match (Choice1Of2 (Choice2Of2 true)) with
| Choice1Of2 x ->
   match x with
  | Choice1Of2 x ->
     (Choice1Of2 x)
  | Choice2Of2 y ->
    (Choice2Of2 y)

| Choice2Of2 y ->
  (Choice2Of2 y)
```

Translating to F#

Structural equality

- Composable data types make it possible for F# to build a series of useful functions automatically
- These functions allow automatic comparison of data structures with the same shape (for example, tuples with tuples or lists with lists)
- Comparison can work with values made up of primitive expressions, tuples, unions, lists, records, and any composition of them
- With the only exclusion of functions: functions, or composite values containing functions, may not be compared

Translating to F#

Interoperability

- One of the practical strengths of F# is its ability to fully interoperate with the .Net framework
- Any feature or library available, even if written in other languages like C#, can be used from inside F#
- Moreover, any F# library can be used from other languages like C#
- This means that applications could be the internal logic in F#, and the UI and database store in C#

Translating to F#

Interoperability

- For example, we can call functions such as `System.Console.WriteLine`, `System.Console.ReadLine`, `System.Int32.Parse`, etc.
- This gives a whole other dimension of usefulness to F#
- Ranging from games to web applications, the language knows no intrinsic limitation
- This also comes with full IDE, autocompletion, and debugger support in Visual Studio and a few other IDE's
- Scala is to some extent comparable

Interoperability

A short F# demo might be in order

Translating to F#

Forbidden features

- F# is a hybrid language
- It also contains OO features, mutability (variables), interfaces, classes, inheritance, etc.
- You may only use these features in the assignments if there is no alternative to access a library you need
- Mutability may only be used in the main program, which then calls regular functions that only use the lambda calculus derived core
- Failure to comply will result in an automatic insufficient grade

About types

- A major difference between F# and the lambda calculus is that F# is, in fact, a statically typed language
- Even though the language does not require type declarations like Java or C#, the compiler will still ensure that the types make sense
- This means that we will get compiler errors because we mix types
- Code like `fst 1` will not compile, for example, because 1 is not a pair and therefore `fst` cannot be applied to it
- We will cover the type system of F# in the next lecture

Homework

Homework

Recursion and F# translations
The INFDEV@HR Team
Introduction
Recursive functions and `let-rec`
Translating to F#
Conclusion

Ways to exercise

- We have added a ton of F# homework, with solutions
- It is all on GitHub
- It is not mandatory, but it is a good idea to do it until you feel more sure
- You may discuss it during one of the two practicums

Conclusion

Conclusion

Recap

- The lambda calculus can be translated, term to term, into F#
- F# is therefore just a practical lambda calculus with a series of handy extensions and slightly more readable

Practicum begins now

This is it!

The best of luck, and thanks for the attention!