CSAR

Dev 8
Functional Programming

Vooraf

Voor vragen:

1) Steek je hand op
2) Stel ze in de chat

Wie zijn wij?

Ricardo Stam

0913788@hr.nl

Marcel Bostelaar

0917554@hr.nl

Het plan

Extra lessen i.c.m de dev videos.

We gaan ervan uit dat je de dev videos al bekeken hebt.

github.com/CSARotterdam/Dev_8_Extra_lessen

De planning

Week 1 – Lambda calculus & F# basics

Week 2 – F#; Data structuren & pattern matching

Week 3 – F#; Functie compositie & pipe opperators

Voor vandaag

- Lambda calculus
  - Extra opdrachten
- F# (refresh)
  - Uitwerking van een opdracht

Paradigmas:
OO vs Functioneel

Paradigmas: OO vs Functioneel

Paradigma's

Paradigma's zijn concepten die toegepast worden binnen het programmeren.

Een programmeer taal kan ontwikkeld zijn om één of meerdere paradigma's te ondersteunen.

Voorbeelden van paradigma's:
Imperatief
Functioneel
Object oriented

Paradigmas: OO vs Functioneel

Object oriented

Main concept:

Alles is een object.

Wat programmeren we?

Classes die objecten representeren.

Hoe wordt het programma uitgevoerd?

Objecten versturen berichten naar elkaar en voeren hierdoor verschillende acties uit en veranderen hierbij hun staat.

Wat is het resultaat?

De (final) staat van de objecten

Paradigmas: OO vs Functioneel

```python
class Player:
    def __init__(self):
        self.hp = 10

    def Attack(self, target, amount):
        target.ReduceHP(amount)

    def ReduceHP(self, amount):
        self.hp = self.hp - amount


class Enemy:
    def __init__(self):
        self.hp = 10

    def Attack(self, target, amount):
        target.ReduceHP(amount)

    def ReduceHP(self, amount):
        self.hp = self.hp - amount

a = Player()
b = Enemy()

a.Attack(b, 2)

print(b.hp)
```

Paradigmas: OOP vs Functioneel

Functioneel

Main concept:
"Alles" is een functie.

Wat programmeren we?
Functies, welke een input nemen, hiermee een actie uitvoeren en een nieuwe waarde teruggeven.

Hoe wordt het programma uitgevoerd?
Evaluatie van de functies.

Wat is het resultaat?
De (final) uitkomst van de "Main" functie.

Lambda calculus

Lambda calculus

Onderdelen

Variabele: a, b, c, ab, acd...

Functie: fun x -> t

Functie applicatie: (fun x -> t) A

Function call: t u

Lambda calculus

Wat zien we?

Functie : fun x -> t

1. Deze functie, accepteert één parameter (x)
2. De body van de functie is t.

“Leesbaar” geschreven:

fun var -> body

Lambda calculus

Wat zien we?

Functie : fun x -> t

```
def f1(x):
    t
```

Lambda calculus

Wat zien we?

Functie applicatie: (fun x -> t) A

1. fun x -> t.
2. Deze functie voeren we uit met als input A (functie applicatie)

“Leesbaar” geschreven:

(fun var -> body) input

Lambda calculus

Wat zien we?

Functie applicatie: (fun x -> t) A

```
def f1(x):
    t

f1(A)
```

Lambda calculus

Wat zien we?

Functie call: t u

1. In principe is dit functie applicatie... maar...
2. t is geen functie of variabele*
3. u is geen functie of variabele*

* - Dit is dus een **functie applicatie** of een **functie call**

Lambda calculus

Wat zien we?

Function calls: t u

```
def fun1(x):
    return lambda y: x + y


def fun2(y):
    return y + 2


t = fun1(1)
u = fun2(1)

print(t(u))
```

Lambda calculus

Evaluatie

Variabelen: a, b, c, ab, acd...

Eval (a)
==
a

Lambda calculus

Evaluatie

Variabelen: a, b, c, ab, acd...

Eval (a b)
==
a b

Lambda calculus

Evaluatie

Functie : fun x -> t

Eval (fun x -> t)
==
fun x -> t

Lambda calculus

Evaluatie

Functie applicatie : (fun x -> t) A

Eval (fun x -> t) A

==

fun x -> t ➔ t[x -> A]

Lambda calculus

Evaluatie

Functie applicatie : (fun x -> t) A

Eval (fun x -> t) A

==

fun x -> t ➔ t[x -> A]

In de body van de functie veranderen we elke x naar A.

Lambda calculus

Voorbeeld evaluatie: Functie applicatie

Abstracte omschrijving: Eval (fun x -> t) A == fun x -> t ➔ t[x -> A]

Opdracht: (fun f -> f f g) Z

x = f
t = f f g
A = Z
t[x -> A] = Z Z g

Lambda calculus

Opdracht evaluatie (5 min): Functie applicatie

Abstracte omschrijving: Eval (fun x -> t) A == fun x -> t ➔ t[x -> A]

Opdracht: (fun a -> fun -> b -> a b) (fun x -> x)

x =
t =
A =
t[x -> A] =

Lambda calculus

Evaluatie

Function calls: t u

t u

==

(eval t == t' && eval u == u')

==

eval t' u' == v

Lambda calculus

Even een stapje terug..

Functie call: t u

1. In principe is dit functie applicatie... maar...
2. t is geen functie of variabele*
3. u is geen functie of variabele*

* - Dit is dus een **functie applicatie** of een **functie call**

Lambda calculus

Evaluatie

(eval t == t' && eval u == u')

==

eval t' u' == v

Wanneer we een functie applicatie tegenkomen waarbij t en/of u verder kunnen worden gereduceerd doen we dit eerst.

t'/u' zijn dus de gereduceerde vormen van t/u.

Lambda calculus

Evaluatie

(eval t == t' && eval u == u')

==

eval t' u' == v

De gerduceerde vormen van de t' u' geeft ons de eind waarde v.

Lambda calculus

Voorbeeld evaluatie: Functie call

Abstracte omschrijving: t u == (eval t == t' && eval u == u') == eval t' u' == v

Opdracht: ((fun x -> x) 4) ((fun y -> y) 5)

t = ??
u = ??

Lambda calculus

Voorbeeld evaluatie: Functie call

((fun x -> x) 4) ((fun y -> y) 5)

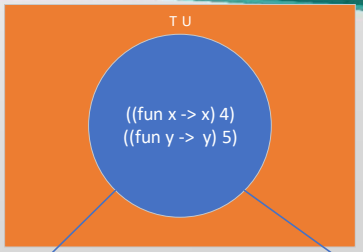
t = ((fun x -> x) 4)
u = ((fun y -> y) 5)
t' = 4
u' = 5
v = 4 5

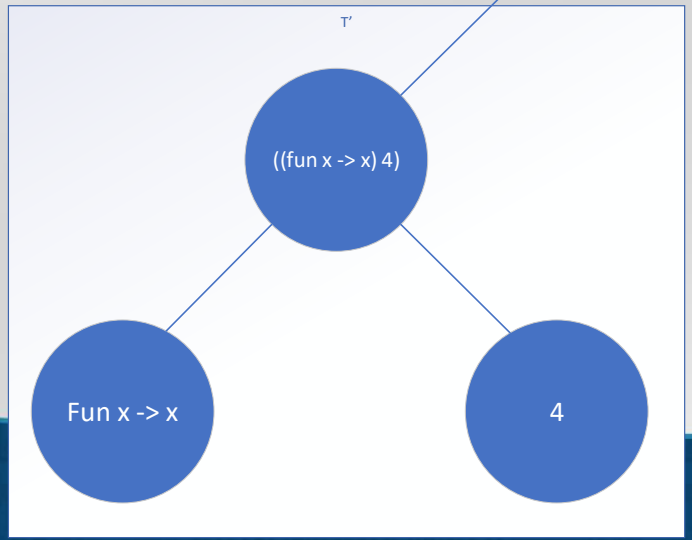
Lambda calculus

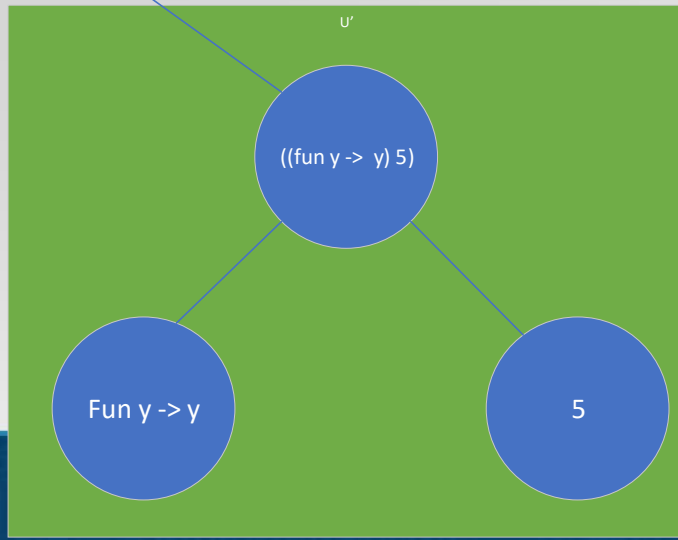
Voorbeeld evaluatie: Functie call

((fun x -> x) 4) ((fun y -> y) 5)

t = ((fun x -> x) 4)
u = ((fun y -> y) 5)
t' = 4
u' = 5
v = 4 5







Lambda calculus

Opdracht evaluatie (5 min): Functie applicatie

Abstracte omschrijving: t u == (eval t == t' && eval u == u') == eval t' u' == v

Opdracht: ((fun x -> fun y -> y x) 2) ((fun z -> fun x-> x + z) 5)

Currying

Lambda calculus

Currying

Een functie die X parameters accepteerd is om om te schrijven naar X functies die elk één parameter accepteren.

(Fun y z -> t) == (fun y -> fun z -> t)

Lambda calculus

Uncurrying

Wanneer je X functies hebt welke elk één parameter accepteren kunnen deze worden omgeschreven naar een functie die X parameters accepteerd.

(fun y -> fun z -> t) == (Fun y z -> t)

Lambda calculus

(un)currying

```
def a(x, y, z):
    return x + y + z


print(a(1, 1, 1))

def b(x):
    def c(y):
        def d(z):
            return x + y + z
        return d
    return c


print(b(1)(1)(1))
```

Shadowing

Lambda calculus

Shadowing

De innerscope, overschrijft de outerscope. (local vs global scope)

Lambda calculus

Shadowing

```
def a(x, y, z):
    return x + y + z


print(a(1, 2, 3))  # => 6

def b(x):
    def c(y):
        def d(x):
            return x + y + x
        return d
    return c


print(b(1)(2)(3))  # => 8
```

Lambda calculus

Shadowing

```
def b(x):
    def c(y):
        def d(x):
            return x + y + x
        return d
    return c


print(b(1)(2)(3))  # => 8
```

=> (((fun x y x -> x + y + x) 1) 2) 3

Intro to F#

Syntax cheatsheet: https://pastebin.com/fXH0RpWe

Intro to F#

Top to bottom compilation

```
let keer2plus2 = fun x -> (keer2 x) + 2

let keer2 = fun x -> x * 2
```

Werkt niet

```
let keer2 = fun x -> x * 2

let keer2plus2 = fun x -> (keer2 x) + 2
```

Let & Fun

Intro to F#

Let zet een **waarde**

Fun maakt een **functie**

Maar in principe doen ze hetzelfde

Intro to F#

Let zet een **waarde**

Fun maakt een **functie**

```
let voorbeeld = fun x -> x + 2

let voorbeeld_ x = x + 2
```

Intro to F#

Let zet een **waarde**

Fun maakt een **functie**

```
let voorbeeld1 = fun x -> fun y -> x + y

let voorbeeld2 = fun x y -> x + y

let voorbeeld3 x = fun y -> x + y

let voorbeeld4 x y = x + y
```

Intro to F#

Let zet een **waarde**

Fun maakt een **functie**

```
let voorbeeld1 = fun x -> fun y -> fun z -> x + y + z

let voorbeeld2 = fun x y z -> x + y + z

let voorbeeld3 x y z = x + y + z
```

Intro to F#

Shadowing

```
let voorbeeld = fun x -> fun y -> x + y
```



```
def first(x):
    def second(y):
        return x + y
    return second
```

```
let voorbeeld2 = fun x -> fun x -> x + x
```

```
def first(x):
    def second(x):
        return x + x
    return second
```

If / Else

Intro to F#

```
let ifelsevoorbeeld nummer = if nummer < 0 then nummer * -1 else nummer

let ifelsevoorbeeld2 nummer =
    if nummer < 0
    then nummer * -1
    else nummer
```

Expressie

Type annotation

Type annotation

Handig om te gebruiken, niet **per se** nodig.

Helpt om duidelijkheid te geven over je functie.

```
let voorbeeld1 x y = x + y

let voorbeeld2 (x : int) (y : int) : int = x + y
```

Recursion

Recursion

F#

```fsharp
let rec factorial nummer =
    if nummer = 1
    then 1
    else nummer * (factorial (nummer - 1))
```

C#

```csharp
static int factorial(int value)
{
    if (value == 1)
        return 1;
    else
        return value * factorial(value - 1);
}
```

Base case

Recursive case

Recursion

```
let rec factorial nummer =
    if nummer = 1
    then 1
    else nummer * (factorial (nummer - 1))

let rec nummerstotN (N : int) (start : int) : string =
    if start < N
    then (start.ToString()) + " " + (nummerstotN N (start + 1))
    else start.ToString()
```

Meer voorbeelden

Voor en nadelen F#

Pro

- Geen verborgen staat
- Functies geven altijd dezelfde uitkomst
- Functies zijn direct vervangbaar
- Goede ondersteuning recursieve datatypen / tree structures

Voor en nadelen F#

Con

- Mogelijk minder performance
- Moeilijk toe te passen op data met referenties naar elkaar
- Slechte ondersteuning / valkuilen als je toch met staat gaat werken

Dank voor jullie aandacht en tijd!

Tot volgende week