Dev 8
Functional Programming

Start: 15:10

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
- Generics
- Collections
  - Tuples
  - Lists
  - Records
  - Unions
- Pattern matching

Λambda calculus

Lambda calculus

Onderdelen

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

Functie: fun x -> t || λ x -> t

Functie applicatie: (fun x -> t) A || (λ x -> t) A

Function call: t u

Lambda calculus

Evaluatie

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

Eval (a)
==
a

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

Function calls: t u

t u

==

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

==

eval t' u' == v

Lambda calculus

Evaluatie

Function calls: t u

t u

==

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

==

eval t' u' == v

Lambda calculus

Bespreken van opdrachten:

(LL) 2
(SL) 1
(LL) 4
(SL) 5

Lambda calculus

Bespreken van opdrachten:

(LL) 2

```
2. (fun x -> fun y -> x y) T
x = x
t = fun y -> x y
a = T
t[x -> a] = fun y -> T y
```

Lambda calculus

Bespreken van opdrachten:

(SL) 1

```
1. (λ x -> x) (λy -> x)
x = x
t = x
a = (λy -> x)
t[x -> a] = (λy -> x)
```

Lambda calculus

Bespreken van opdrachten:

(LL) 4

```
4. (fun z -> fun a -> a z) (((fun b -> b) D) ((fun t -> t) I))

t = (fun z -> fun a -> a z)
u = (((fun b -> b) D) ((fun t -> t) I))
------
t[0] = (fun z -> fun a -> a z) => (fun z -> fun a -> a z)
t' = t[0]

u[0] = ((fun b -> b) D) ((fun t -> t) I) => D ((fun t -> t) I)
u[1] = D ((fun t -> t) I) => D I
u[2] = D I => D I
u' = u[2]
------

t' u' = (fun z -> fun a -> a z) D I

t' u'[0] = (fun z -> fun a -> a z) D I => (fun a -> a D) I
t' u'[1]= (fun a -> a D) I => I D

t' u'[2] = I D => I D
v = t' u'[2]
```

Lambda calculus

Bespreken van opdrachten:

(SL) 5

```
5. ((λ x -> x) 4) ((λ x y -> x + y) 5)

t = ((λ x -> x) 4)
u = ((λ x y -> x + y) 5)
------

t[0] = ((λ x -> x) 4) => 4
t[1] = 4 => 4
t' = t[1]

u[0] = ((λ x y -> x + y) 5) => (λ y -> 5 + y)
u[1] = (λ y -> 5 + y) => (λ y -> 5 + y)
u' = u[1]
------

t' u' = 4 (λ y -> 5 + y)

t' u'[0] = 4 (λ y -> 5 + y) => 4 (λ y -> 5 + y)
v = t' u'[0]
```

Generics

Generics

Hoe zat het ook alweer? (C#)

```
public class generic<T>
{
    private T value;

    public generic(T _value)
    {
        value = _value;
    }
}
```

```
generic stringish = new generic<string>("value");
```

```
generic intish = new generic<int>(123);
```

Generics

Creëren van een generic (functie)

- Generic types moeten met een apostrof ' beginnen.

```
let returnEersteWaarde (arg1 : 'type1) (arg2 : 'type2) : 'type1 = arg1
```

```
let returnTweedeWaarde (arg1 : 'a) (arg2 : 'b) : 'b = arg2
```

Generics

Aanroepen van een generic (functie)

```
let returnTweedeWaarde (arg1 : 'a) (arg2 : 'b) : 'b = arg2

let x = returnTweedeWaarde 1 2
let y = returnTweedeWaarde 3 "aa"
```

```
let returnTweedeWaarde (arg1 : 'a) (arg2 : 'b) : 'b = arg2

let x = returnTweedeWaarde<int, int> 1 2
let y = returnTweedeWaarde<int, string> 3 "aa"
```

Tuples

Tuples

Wat was een tuple?

- Een tuple is een groepering van (verschillende type) waarde(s).

- Tuples != List (() **vs** [])

Tuples

Aanmaken van een tuple

- Gebruikt de comma , voor de constructie (en deconstructie)
- Kan onbeperkt aantal elementen hebben

```
let waarde = 10

let waarde2 = "voorbeeld"

let tuple = "eerste waarde","tweede waarde"

let tuple2 = "kan ook verschilende types zijn", 420

let tuple2_met_haakjes = ("kan ook verschilende types zijn", 420) //Betekend hetzelfde als tuple2
```

Tuples

Type notatie

- Type notatie is met een sterretje (‘a * ‘b)

```
let (eentuple : string*int) = "hallo",44

let (eentuple2 : string*int*bool) = "hallo",44,true
```

Tuples

Ingebouwde functies (2D tuples)

- fst – Return het eerste element uit een tuple
- Snd – Return het tweede element uit een tuple

```
let first = fst (1, 2)

let second = snd (1, 2)
```

```
let voorbeeld_fst (argument : string * int) : string = fst argument
let voorbeeld_snd (argument : string * int) : int = snd argument
```

Tuples

Deconstructie notatie

- Je kan voor alle tuples de deconstructie notatie gebruiken

  (**let item1, item2, item3 = tuple**)

```
let voorbeeldTuple = 21, "Fred"

let leeftijd, naam = voorbeeldTuple

let grooteTuple = 21, "Fred", "de Jonge", "Rotterdam"

let leeftijd2, naam2, achternaam, woonplaats = grooteTuple
```

Lists

Lists

- F# lists zijn immutable singly linked lists (dev 7)
- De lijst variable wijst altijd naar de head, de rest van de lijst is niet direct bereikbaar

Lists

Aanmaken van een list

- Lijstelementen zijn gescheiden door puntcommas ;
  (dus geen commas ,)

```
let legeLijst = []

let voorbeeldlijst = ["hallo" ; "dit zijn elementen" ; "Ze moeten allemaal hetzelfde type hebben"]
```

Lists

Toevoegen van waardes

- Met de :: operator kan je een los element 'a aan de **voorkant** van een bestaande list<'a> plakken

```
let legeLijst = []

let voorbeeldlijst = ["hallo" ; "dit zijn elementen" ; "Ze moeten allemaal hetzelfde type hebben"]

let itemToevoegen = "nieuw item" :: voorbeeldlijst

let kanNietAanEinde = voorbeeldlijst :: "nieuw item"
```

Lists

Vraag:

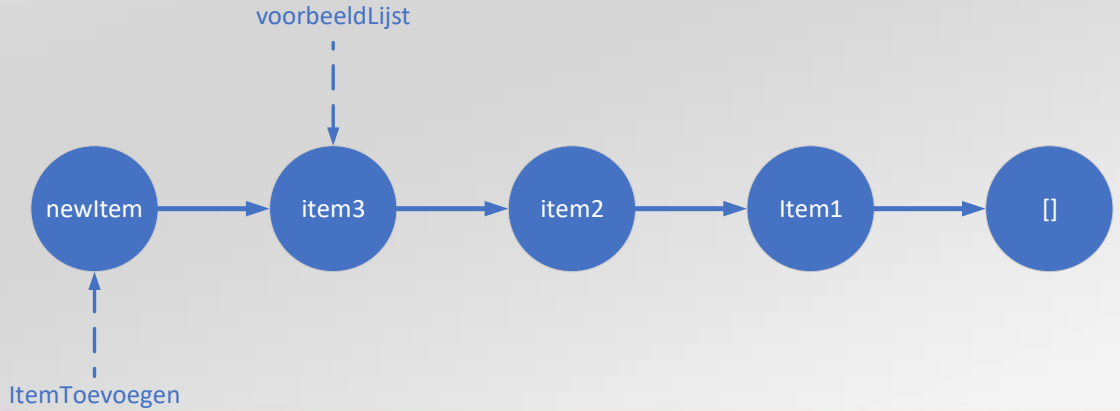
- Wanneer we de laatste regel uitvoeren, wordt hierbij een nieuwe lijst aangemaakt? En waarom (niet)?

```
let voorbeeldlijst = ["hallo" ; "dit zijn elementen" ; "Ze moeten allemaal hetzelfde type hebben"]

let itemToevoegen = "nieuw item" :: voorbeeldlijst
```

Lists

Vraag: Wanneer we de laatste regel uitvoeren, wordt hierbij een nieuwe lijst aangemaakt? En waarom?

```
let voorbeeldlijst = ["hallo" ; "dit zijn elementen" ; "Ze moeten allemaal hetzelfde type hebben"]

let itemToevoegen = "nieuw item" :: voorbeeldlijst
```



Records

Records

- Een record is een verzameling waardes met genaamde velden
- Geen constructor functie
- Type definition is met dubbele punt :
- Instance creatie is met equals teken =

```
type voorbeeldRecord = {fieldName1: string; fieldName2: int}

let instanceVanVoorbeeld = {fieldName1 = "dit is de string"; fieldName2 = 42}

type genericRecord<'a> = {veld1: 'a; veld2: int}

let (instance1Generic : genericRecord<string>) = {veld1 = "met string"; veld2 = 42}
let (instance2Generic : genericRecord<int list>) = {veld1 = [1;2;3;4;] ; veld2 = 42}
```

Records

Aanroepen van velden

- Velden roep je aan met puntnotatie net als in C#

```
type voorbeeldRecord = {fieldName1: string; fieldName2: int}
let instance = {fieldName1 = "dit is de string"; fieldName2 = 42}

let veld1 = instance.fieldName1
let veld2 = instance.fieldName2
```

Records

“Aanpassen” van een record

- Je kan records makkelijk kopieren met 1 of meer veranderde velden met "with"

```
type voorbeeldRecord = {fieldName1: string; fieldName2: int}

let instanceVanVoorbeeld = {fieldName1 = "dit is de string"; fieldName2 = 42}

let verander1veld = {instanceVanVoorbeeld with fieldName2 = 420}

type voorbeeldRecordLarge =  {fieldName1: string; fieldName2: int ;fieldName3: int ;fieldName4: int}
let instance = {fieldName1= "test"; fieldName2= 3; fieldName3= 3; fieldName4= 3}
let verandertInstance = {instance with fieldName1 = "hallo"; fieldName3 = 55}
```

Discriminated unions

Discriminated Unions

- Een discriminated union maakt het mogelijk om een keuze uit meerdere mogelijke types in 1 variabele te hebben
- Ook kan je mogelijkheden met dezelfde datatypen een ander label geven
- Je kan ook dataloze mogelijkheden maken

```
type discriminatedUnionVoorbeeld =
    | EersteMogelijkheid of int
    | TweedeMogelijkheid of string
    | DerdeMogelijkheid of string*int
    | VierdeMogelijkheid of int
    | VijfdeMogelijkheid of int //Meerdere mogelijkheden mogen hetzelfde type hebben
    | MogelijkheidZonderWaarde
    | MogelijkheidZonderWaarde2 //Mogelijkheden zonder waarde kunnen ook
```

Discriminated Unions

- Voorbeeld gebruik in een usecase:

```
type telefoon = {merk: string}

type inhoudPakketje =
    | Telefoon of telefoon
    | Laptop of laptop
    | Pen of pen
    | Boek of boek
    | GrafischeKaart of grafischeKaart
    | Niks

let (pakketjeMetTelefoon : inhoudPakketje) = Telefoon {merk = "samsung"}
```

Pattern matching

Pattern Matching

- Maakt het mogelijk om discriminated unions en lijsten te verwerken.

```
let VertelVerhaal (persoon : string*int) :string =
    match persoon with
    | ("Albert Einstein", leeftijd) ->
        "Albert Einstein is een hele slimme man van " + (string(leeftijd)) + " jaar oud"
    | (naam, 65) -> naam + " mag eindelijk met pensioen"
    | ("Fred", _) -> "Deze man heet fred, zijn leeftijd gooi ik weg"
    | (naam, leeftijd) -> naam + " is " + (string(leeftijd)) + " jaar oud"
```

Pattern Matching

- Maakt het mogelijk op basis van specifieke waarden data anders te verwerken

```
let VertelVerhaal (persoon : string*int) :string =
    match persoon with
    | ("Albert Einstein", leeftijd) ->
        "Albert Einstein is een hele slimme man van " + (string(leeftijd)) + " jaar oud"
    | (naam, 65) -> naam + " mag eindelijk met pensioen"
    | ("Fred", _) -> "Deze man heet fred, zijn leeftijd gooi ik weg"
    | (naam, leeftijd) -> naam + " is " + (string(leeftijd)) + " jaar oud"
```

Pattern Matching

- Deconstructie gebruikt dezelfde syntax als constructie

```
let VertelVerhaal (persoon : string*int) :string =
    match persoon with
    | ("Albert Einstein", leeftijd) ->
        "Albert Einstein is een hele slimme man van " + (string(leeftijd)) + " jaar oud"
    | (naam, 65) -> naam + " mag eindelijk met pensioen"
    | ("Fred", _) -> "Deze man heet fred, zijn leeftijd gooi ik weg"
    | (naam, leeftijd) -> naam + " is " + (string(leeftijd)) + " jaar oud"
```

Pattern Matching

- Match regels worden van boven naar beneden verwerkt. Zet specifiekere matchregels altijd boven minder specifieke.

```
let BijzondereLeeftijd (leeftijd : int) : bool =
    match leeftijd with
    | 65 -> true
    | 69 -> true
    | 420 -> true
    | 18 -> true
    | _ -> false
```

Pattern Matching

- Alle mogelijke waardes moeten gecovered worden door een matchregel

```
let BijzondereLeeftijd (leeftijd : int) : bool =
    match leeftijd with
    | 65 -> true
    | 69 -> true
    | 420 -> true
    | 18 -> true
    | _ -> false
```

Pattern Matching

- Wildcard underscore _ kan als laatste matchregel gebruikt worden, maar haalt hiermee compile checks voor alle mogelijke inputs weg (dus doe dit voorzichtig)

```
let BijzondereLeeftijd (leeftijd : int) : bool =
    match leeftijd with
    | 65 -> true
    | 69 -> true
    | 420 -> true
    | 18 -> true
    | _ -> false
```

Pattern Matching

- Discriminated unions kunnen alleen via pattern matching verwerkt worden
- Binnen deze matches kan je ook op specifieke subwaarden matchen

Pattern Matching

```
type voorbeeldUnion =
    | EersteMogelijkheid of int
    | TweedeMogelijkheid of string
    | DerdeMogelijkheid of string*int

let matchHet x : string =
    match x with
    | EersteMogelijkheid intwaarde -> intwaarde.ToString()
    | TweedeMogelijkheid stringwaarde -> stringwaarde
    | DerdeMogelijkheid (stringwaarde, intwaarde) -> stringwaarde + ", " + intwaarde.ToString()

let matchHet2 x : string =
    match x with
    | EersteMogelijkheid 420 -> "Deze regel matcht alleen cases van eerstewaarde met de waarde 420"
    | EersteMogelijkheid intwaarde -> intwaarde.ToString()
    | TweedeMogelijkheid stringwaarde -> stringwaarde
    | DerdeMogelijkheid (stringwaarde, 42) -> stringwaarde + ", hier matchde ik specifiek 42"
    | DerdeMogelijkheid (stringwaarde, intwaarde) -> stringwaarde + ", " + intwaarde.ToString()
```

Pattern Matching

- Lijsten kunnen op specifieke aantallen elementen worden gematched

```
let lijstenMatchen (lijst : 'a list) =
    match lijst with
    | [] -> "Lege lijst"
    | [x] -> "lijst met 1 item erin: " + x.ToString()
    | [x; y] -> "lijst met 2 items erin: " + x.ToString() + ", " + y.ToString()
    | head :: tail -> "een lijst met ten minste 1 waarde en een staart met 0 of meer items"

let minimaleLijstVerwerking (lijst : 'a list) =
    match lijst with
    | [] -> "Lege lijst"
    | head :: tail -> "1 Waarde " + head.ToString() + " en een staart van 0 of meer items"
```

Pattern Matching

- Lijst kan ook met de :: operator gematched worden, waarin tail de lijst zonder het eerste element is.

```
let lijstenMatchen (lijst : 'a list) =
    match lijst with
    | [] -> "Lege lijst"
    | [x] -> "lijst met 1 item erin: " + x.ToString()
    | [x; y] -> "lijst met 2 items erin: " + x.ToString() + ", " + y.ToString()
    | head :: tail -> "een lijst met ten minste 1 waarde en een staart met 0 of meer items"

let minimaleLijstVerwerking (lijst : 'a list) =
    match lijst with
    | [] -> "Lege lijst"
    | head :: tail -> "1 Waarde " + head.ToString() + " en een staart van 0 of meer items"
```

Pattern Matching

- Een lege lijst en een head :: tail is nodig om alle cases the coveren

```
let lijstenMatchen (lijst : 'a list) =
    match lijst with
    | [] -> "Lege lijst"
    | [x] -> "lijst met 1 item erin: " + x.ToString()
    | [x; y] -> "lijst met 2 items erin: " + x.ToString() + ", " + y.ToString()
    | head :: tail -> "een lijst met ten minste 1 waarde en een staart met 0 of meer items"

let minimaleLijstVerwerking (lijst : 'a list) =
    match lijst with
    | [] -> "Lege lijst"
    | head :: tail -> "1 Waarde " + head.ToString() + " en een staart van 0 of meer items"
```

Pattern Matching

- Lijsten, net als andere constructies, kunnen op specifieke waarden gematched worden

```
let lijstenMatchen2 (lijst : int list) =
    match lijst with
    | [] -> "leeg"
    | [420] -> "Alleen maar 420"
    | 420 :: 420 :: tail -> "2x 420 en dan een tail"
    | 420 :: tail -> "1x 420 en dan een tail"
    | head :: tail -> "geen 420 maar wel een andere waarde"
```

Pattern Matching

- Records (en vrijwel alles) kunnen ook op basis van hun inhoud gematched worden

```
type persoon = {voornaam: string; achternaam: string;}

let veranderJoostNaarBram (persoon : persoon) : persoon =
    match persoon with
    | {voornaam = "Joost"; achternaam = x} ->
        {voornaam = "Bram"; achternaam = x}
    | x -> x
```

Option

Option

- Een belangrijke discriminated union is option
- Wordt gebruikt om afwezigheid van waarde te modeleren (IPV null)
- Ingebouwde F# versie is zonder hoofdletter

```
type Option<'a> =
    | Some of 'a
    | None

let Niet420 (getal : int) : Option<int> =
    match getal with
    | 420 -> None
    | x -> Some x
```

Option

- Hieronder volgt een korte illustratie van hoe men option kan gebruiken

```
type Option<'a> =
    | Some of 'a
    | None

let Niet420 (getal : int) : Option<int> =
    match getal with
    | 420 -> None
    | x -> Some x
```

```
let rec listMap (functie : 'a -> 'b) (lijst : 'a list) : 'b list =
    match lijst with
    | [] -> []
    | head :: tail -> functie head :: listMap functie tail

let rec alleenSomeValues (lijst : Option<'x> list) : 'x list =
    match lijst with
    | [] -> []
    | Some y :: tail -> y :: alleenSomeValues tail
    | None :: tail -> alleenSomeValues tail

let (voorbeeldlijst : int list) = [420;555;420;420;54;3;223;44;55]
let (eerstestap : Option<int> list) = listMap Niet420 voorbeeldlijst
let (resultaat : int list) = alleenSomeValues eerstestap
```

Vragen?

Opdracht

Opdracht

- Maak records: cirkel met radius, rechthoek met lengte en breedte
- Maak een discriminated union "vorm" met rechthoek en cirkel er in
- Maak een functie die de oppervlakte van een rechthoek berekent
- Maak een functie die de oppervlakte van een cirkel berekent
- Maak een functie die de oppervlakte van een vorm berekent
- Maak een functie die een lijst met vormen accepteerd en alleen de cirkels teruggeeft
- Doe hetzelde voor rechthoeken

Dank voor jullie aandacht en tijd!

Tot volgende week