Heutige Themen

- *Funktionale reaktive Programmierung*: Mobile Roboter
  - Arrows: Eine Verallgemeinerung des Monadenkonzepts
- *Parsing*: Lexikalische und syntaktische Analyse
  - Kombinatorparsing
  - Monadisches Parsing

Motivation: Mobile Roboter

Mobile Roboter und die hier betrachtete Konfiguration:

Roboter haben zwei Räder, die von je einem Motor unabhängig voneinander angetrieben werden. Die relative Geschwindigkeit der Räder zueinander legt die Richtung fest, in die sich der Roboter bewegt. Bewegen sich beide Räder gleich schnell, fährt der Roboter geradeaus.

Roboter verfügen über unterschiedliche Sensoren wie etwa einen Berührungssensor, ob der Roboter auf ein Hindernis trifft, Entfernungsmesser, um die Entfernung zum nächstgelegenen Objekt in einer bestimmten Richtung zu bestimmen (in der Folge nur für vorwärts, rückwärts, rechts und links), einen Objektsensor, der die aktuelle Position aller anderen Roboter liefert und ggf. die Position weiterer beweglicher Objekte wie z.B. Bällen innerhalb einer bestimmten Entfernung vom Roboter (Objektsensoren etwa realisiert durch visuelle Subsysteme oder Kommunikationssysteme zum Austausch von Koordinaten).

Roboter haben eine eindeutige Kennung (ID) und ggf. weitere (bei Bedarf) eingeführte Fähigkeiten.

Motivation: Roboterfußball

Roboterfußball, eine typische Aufgabenstellung:

"Schreibe ein Programm "robocup soccer" wie folgt: Erzeuge mithilfe von Wandsegmenten zwei Tore an jedem Spielfeldende. Bestimme die Anzahl von Spielern pro Mannschaft und schreibe Steuerungsprogramme für diese Spieler. Zweckmäßigerweise sind die Steuerungsprogramme generisch für unterschiedliche Spielertypen, etwa für Torhüter, Stürmer, Verteidiger. Erzeuge eine initiale Welt, in der der Ball am Anstoßpunkt im Mittelkreis liegt und jeder der Spieler strategisch günstig in der jeweils eigenen Spielhälfte und außerhalb des Mittelkreises positioniert ist. Jede Mannschaft mag die gleichen oder unterschiedliche Kontrollprogramme nutzen."

Mobile Roboter: Hybride Systeme 1(2)

Hybride Systeme sind Systeme mit

- stetigen und
- diskreten

Komponenten.

Mobile Roboter: Hybride Systeme 2(2)

Mobile Roboter sind konkrete Ausprägung eines hybriden Systems:

- Unter *physikalischer* Perspektive
  - *Stetige Komponenten*: Spannungsgesteuerte Motoren, Batterien, Entfernungsmesser,...
  - *Diskrete Komponenten*: Mikroprozessoren, Berührungssensoren,...
- Unter *logischer* Perspektive
  - *Stetige* Begriffe: Radgeschwindigkeit, Ausrichtung, Entfernung von einer Wand,...
  - *Diskrete* Begriffe: Zusammenstoß mit einem anderen Objekt, Empfangen einer Nachricht, Erreichen eines Ziels,...

Robotersimulatorcode

...erhältlich über die Yampa-Leitseite unter

`http://www.haskell.org/yampa`

In der Folge einige Auszüge...

Struktur des Gesamtprogramms

```
module MyRobotShow where
  import AFrob
  import AFrobRobotSim

  main :: IO ()
  main = runSim (Just world) rcA rcB

  world :: WorldTemplate
  world = ...

  rcA :: SimbotController -- controller for simbot A's
  rcA = ...

  rcB :: SimbotController -- controller for simbot B's
  rcB = ...
```


Die Welt

```
type WorldTemplate = [ObjectTemplate]

data ObjectTemplate =
     OTBlock   { otPos  :: Position2 }  -- Square obstacle
   | OTVWall   { otPos  :: Position2 }  -- Vertical wall
   | OTHWall   { otPos  :: Position2 }  -- Horizontal wall
   | OTBall    { otPos  :: Position2 }  -- Ball
   | OTSimbotA { otRId  :: RobotId,     -- Simbot A robot
                 otPos  :: Position2,
                 otHdng :: Heading    }
   | OTSimbotB { otRId  :: RobotId,     -- Simbot B robot
                 otPos  :: Position2,
                 otHdng :: Heading    }
```

Robotersteuerung

```
type SimbotController =
          SimbotProperties -> SF SimbotInput SimbotOutput

Class HasRobotProperties i where
  rpType     :: i -> RobotType     -- Type of robot
  rpId       :: i -> RobotId       -- Identity of robot
  rpDiameter :: i -> Length        -- Distance between wheels
  rpAccMax   :: i -> Acceleration  -- Max translational acc
  rpWSMax    :: i -> Speed         -- Max wheel speed

type RobotType = String
type RobotId   = Int
```


Robotersimulation in Aktion

Laufenlassen der Robotersimulation:

```
runSim :: Maybe WorldTemplate ->
          SimbotController -> SimbotController -> IO ()
```


Robotersteuerung

```
rcA :: SimbotController
rcA rProps =
  case rrpId rProps of
    1 -> rcA1 rProps
    2 -> rcA2 rProps
    3 -> rcA3 rProps

rcA1, rcA2, rcA3 :: SimbotController
rcA1 = ...
rcA2 = ...
rcA3 = ...
```


Roboteraktionen: Steuerungsprogramme 1(2)

Ein stationärer Roboter:

```
rcStop :: SimbotController
rcStop _ = constant (mrFinalize ddBrake)
```

Ein sich blind mit konstanter Geschwindigkeit bewegender Roboter:

```
rcBlind1 _ = constant (mrFinalize $ ddVelDiff 10 10)
```

Ein sich blind mit halber Maximalgeschwindigkeit bewegender Roboter:

```
rcBlind2 rps =
  let max = rpWSMax rps
  in constant (mrFinalize $ ddVelDiff (max/2) (max/2))
```

Roboteraktionen: Steuerungsprogramme 2(2)

Ein sich zu vorgegebener Geschwindigkeit maximal schnell drehender Roboter:

```
rcTurn :: Velocity -> SimbotController
rcTurn vel rps =
  let vMax = rpWSMax rps
      rMax = 2 * (vMax - vel) / rpDiameter rps
  in constant (mrFinalize $ ddVelTR vel rMax)
```


Roboterklassen 1(5)

- I.a. gibt es unterschiedlich ausgestattete Typen von Robotern (2 Räder, 3 Räder, Kamera, Sonar, Lautsprecher, Blinklichter, etc.)
- Die Ausstattungsvariante eines Roboters wird durch seine Ein- und Ausgabetypen festgelegt.

Diese sind in den *Input* und *Output Classes* nebst ihren zugehörigen Funktionen festgelegt...

Roboterklassen 2(5)

Input Classes und zugehörige Funktionen:

```
class HasRobotStatus i where
  rsBattStat :: i -> BatteryStatus  -- Current battery status
  rsIsStuck  :: i -> Bool           -- Currently stuck or not

data BatteryStatus = BSHigh | BSLow | BSCritical
     deriving (Eq, Show)

-- derived event sources:
rsBattStatChanged  :: HasRobotStatus i => SF i (Event BatteryStatus)
rsBattStatLow      :: HasRobotStatus i => SF i (Event ())
rsBattStatCritical :: HasRobotStatus i => SF i (Event ())
rsStuck            :: HasRobotStatus i => SF i (Event ())
```


Roboterklassen 3(5)

```
class HasOdometry where
  odometryPosition :: i -> Position2  -- Current position
  odometryHeading  :: i -> Heading    -- Current heading

class HasRangeFinder i where
  rfRange    :: i -> Angle -> Distance
  rfMaxRange :: i -> Distance

-- derived range finders:
rfFront :: HasRangeFinder i => i -> Distance
rfBack  :: HasRangeFinder i => i -> Distance
rfLeft  :: HasRangeFinder i => i -> Distance
rfRight :: HasRangeFinder i => i -> Distance
```

Roboterklassen 4(5)

```
class HasAnimateObjectTracker i where
  aotOtherRobots :: i -> [(RobotType, Angle, Distance)]
  aotBalls       :: i -> [(Angle, Distance)]

class HasTextualConsoleInput i where
  tciKey :: i -> Maybe Char

tciNewKeyDown :: HasTextualConsoleInput i =>
                            Maybe Char -> SF i (Event Char)
tciKeyDown    :: HasTextualConsoleInput i => SF i (Event Char)
```


Roboterklassen 5(5)

Output Classes und zugehörige Funktionen:

```
class MergeableRecord o => HasDiffDrive o where
  ddBrake :: MR o                              -- Brake both wheels
  ddVelDiff :: Velocity -> Velocity -> MR o -- Set wheel velocities
  ddVelTR   :: Velocity -> RotVel   -> MR o -- Set vel. and rot.

class MergeableRecord o => HasTextConsoleOutput o where
  tcoPrintMessage :: Event String -> MR o
```


Die Klasse `Arrow`

```
class Arrow a where
  pure  :: (b -> c) a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

Bem.:

- `pure` erlaubt die Einbettung gewöhnlicher Funktionen in die Klasse `Arrow`
- (>>>) zur Komposition von Berechnungen
- `first` hat als Analogon auf dem Niveau gewöhnlicher Funktionen die Funktion `firstfun f = \(x,y) -> (f x, y)`

Gesetze für Instanzen von `arrow`

```
Identitaet                        pure id >>> f = f
Identitaet                        f >>> pure id = f
Assoziativitaet          (f >>> g) >>> h = f >>> (g >>> h)
functor-composition        pure (g . f) = pure f >>> pure g
extension                  first (pure f) = pure (f x id)
functor                  first (f >>> g) = first f >>> first g
exchange        first f >>> pure (id x g) = pure (id x g) >>> first f
unit                 first f >>> pure fst = pure fst >>> f
association  first (first f) >>> pure assoc = pure assoc >>> first f
```

Hinweis: Hier und in der Folge steht `x` für $\times$.

Hilfsfunktionen

```
(x) :: (a -> a') -> (b -> b') -> (a,b) -> (a',b')
(f x g) (a,b) = (f a, g b)

assoc :: ((a,b),c) -> (a,(b,c))
assoc~(~(x,y),z) = (x,(y,z))

second :: Arrow a => a b c -> a (d,b) (d,c)
second f = pure swap >>> first f >>> pure swap

swap :: (a,b) -> (b,a)
swap~(x,y) = (y,x)
```


Beispiel

Gewöhnliche Funktionen als Instanz der Klasse `Arrow`:

```
instance Arrow (->) where
  pure f  = f
  f >>> g = g . f
  first f = f x id
```

Bem.:

- Alternativ lässt sich `first` im obigen Beispiel folgendermaßen definieren: `first f = \(b,d) -> (f b, d)`

Weitere abgeleitete Funktionen

...im Zusammenhang mit der Klasse `Arrow`:

```
(***)   :: Arrow a => a b c -> a b' c' -> a (b.b') (c,c')
f *** g = first f >>> second g

(&&&)   :: Arrow a => a b c -> a b c' -> a b (c,c')
f &&& g = pure (\b -> (b,b)) >>> f *** g

idA :: Arrow a => a b b
idA = pure id
```


Hintergrund und Motivation 1(6)

Berechnungsbegriffe

```
add       :: (b -> Int) -> (b -> Int) -> (b -> Int)
add f g z = f z + g z
```

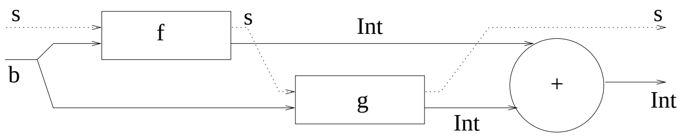
Hintergrund und Motivation 2(6)

Verallgemeinerung von add auf *Zustandstransformatoren*:

```
type State s i o = (s,i) -> (s,o)

addST :: State s b Int -> State s b Int -> State s b Int
addST f g (s,z) = let (s',x)  = f (s,z)
                      (s'',y) = g (s',z)
                  in (s'',x+y)
```

Illustration:



Hintergrund und Motivation 3(6)

Verallgemeinerung von add auf *Nichtdeterminismus*:

```
type NonDet i o = i -> [o]

addND :: NonDet b Int -> NonDet b Int -> NonDet b Int
addND f g z = [ x+y | x <- f z, y <- g z ]
```


Hintergrund und Motivation 4(6)

Verallgemeinerung von add auf *Abbildungstransformatoren*:

```
type MapTrans s i o = (s -> i) -> (s -> o)

addMT :: MapTrans s b Int -> MapTrans s b Int -> MapTrans s b Int
addMT f g m z = f m z + g m z
```


Hintergrund und Motivation 5(6)

Verallgemeinerung von add auf *einfache Automaten*:

```
newtype Auto i o = A (i -> (o, Auto i o))

addAuto :: Auto b Int -> Auto b Int -> Auto b Int
addAuto (A f) (A g) = A (\z -> let (x,f') = f z
                                   (y,g') = g z
                               in (x+y), addAuto f' g'))
```

...Modellierung synchroner Schaltkreise.

Hintergrund und Motivation 6(6)

- Funktionen und Programme beinhalten häufig Komponenten, die “funktionsartig”, aber “nicht einfach Funktionen” sind.
- `Arrows` definieren eine gemeinsame Schnittstelle, um mit dem “Berechnungsbegriff” solcher funktionsartigen Komponenten umgehen zu können.
- Monaden sind als Spezialfall in `Arrows` enthalten.
- Wie Monaden erlauben `Arrows`, Programme sinnvoll zu strukturieren.

Zurück zu den Beispielen 1(7)

- Den vorausgegangenen Beispielen ist gemeinsam, dass es einen Typ $A \rightsquigarrow B$ von *Berechnungen* gibt, wobei Eingaben vom Typ A in Ausgaben vom Typ B transformiert werden.
- `Arrows` liefern eine hinreichend allgemeine Schnittstelle, um diese Gemeinsamkeiten einheitlich zu beschreiben und in einer Klasse zu kapseln.

Zurück zu den Beispielen 2(7)

Implementierung der vorausgegangenen Beispiele mithilfe der `Arrow`-Klasse...

```
newtype State s i o = ST ((s,i) -> (s,0))

newtype NotDet i o  = ND (i -> [o])

newtype MapTrans s i o = MT ((s -> i) -> (s -> o))

newtype Auto i o = A (i -> (o, Auto i o))
```


Zurück zu den Beispielen 3(7)

Zustandstransformatoren...

```
instance Arrow (State s) where
  pure f        = ST (id x f)
  ST f >>> ST g = ST (g . f)
  first (ST f)  = ST (assoc . (f x id) . unassoc)

unassoc            :: (a,(b,c)) -> ((a,b),c)
unassoc~(x,~(y,z)) = ((x,y),z)
```

Zurück zu den Beispielen 4(7)

Nichtdeterminismus...

```
instance Arrow NonDet where
  pure f        = ND (\b -> [f b])
  ND f >>> ND g = ND (\b -> [d | c <- f b, d <- g c])
  first (ND f)  = ND (\(b,d) -> [(c,d) | c <- f b])
```


Zurück zu den Beispielen 5(7)

Abbildungstransformatoren...

```
instance Arrow (MapTrans s) where
  pure f        = MT (f .)
  MT f >>> MT g = MT (g . f)
  first (MT f)  = MT (zipMap . (f x id) . unzipMap)

zipMap     :: (s -> a, s -> b) -> (s -> (a,b))
zipMap h s = (fst h s, snd h s)

unzipMap   :: (s -> (a,b)) -> (s -> a, s -> b)
unzipMap h = (fst . h, snd . h)
```


Zurück zu den Beispielen 6(7)

Einfache Automaten...

```
instance Arrow Auto where
  pure f      = A (\b -> (f b, pure f))
  A f >>> A g = A (\b -> let (c,f') = f b
                             (d,g') = g c
                         in (d, f' >>> g')))
  first (A f) = A (\(b,d) -> let (c,f') = f b
                             in ((c,d),first f'))
```


Zurück zu den Beispielen 7(7)

Generalisierung

Betrachte folgenden allgemeinen Kombinator:

```
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = f &&& g >>> pure (uncurry (+))
```

Beobachtung:

- Jede der betrachteten Varianten von `add` ergibt sich als Spezialisierung von `addA` mit dem entsprechenden `arrow`-Typ.

Weitere Beobachtungen

- `Arrow`-Kombinatoren operieren auf "Berechnungen", nicht auf Werten. Sie sind argumentfrei (engl. point-free) im Unterschied zum "Normalfall" funktionaler Programmierung.
- Analog zum monadischen Fall erlaubt eine `do`-artige Notationsvariante einen programmiertechnisch oft einfacheren und suggestiveren Umgang mit `Arrow`-Operationen (siehe Literaturhinweise am Ende), wohingegen die punktfreie Variante für beweistechnische überlegungen vorteilhafter ist.

Zurück zu mobilen Robotern

Der Kreis schließt sich...

```
SF a b    = Signal a -> Signal b

Signal a  = Time -> a

type Time = Double
```

...`SF` ist Instanz der Klasse `Arrow`.

Bem.: Werte vom Typ `SF` sind Signaltransformatoren. Signaltransformatoren werden hier als Signalfunktionen (engl. *signal functions*) bezeichnet; daher die Kurzform `SF`.

Literaturhinweise 1(6)

Speziell für diesen Vorlesungsteil...

- Paul Hudak, Antony Courtney, Herik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. Summer School on Advanced Functional Programming 2002, LNCS 2638, Springer-Verlag, 159 - 187, 2003.
- John Hughes. *Generalising Monads to Arrows*. Science of Computer Programming 37, 67 - 111, 2000.
- Ross Paterson. *A New Notation for Arrows*. In Proceedings of the 6th ACM SIGPLAN'01 International Conference on Functional Programming (ICFP 2001), 229 - 240, 2001.

Literaturhinweise 2(6)

Zu den Wurzeln funktionaler reaktiver Programmierung i.a...

- Fran

  Conal Elliot, Paul Hudak. *Functional Reactive Animation*. In Proceedings of the International Conference on Functional Programming (ICFP 1997), 263 - 273, 1997.

  Conal Elliot. *Functional Implementations of Continuous Modeled Animation*. In Proceedings of PLILP/ALP'98, Springer-Verlag, 1998.

Literaturhinweise 3(6)

Zu den Wurzeln funktionaler reaktiver Programmierung i.a...

- FRP

  John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. *Yale FRP User's Manual*. Department of Computer Science, Yale University, January 2001. `http://www.haskell.org/frp/manual.html`

  Zhanyong Wan, Paul Hudak. *Functional Reactive Programming from First Principles*. In Proceedings of the ACM SIGPLAN'00 Conference on Programming Languages Design and Implementation (PLDI 2000), ACM Press, 2000.

  Henrik Nilsson, Antony Courtney, John Peterson. *Functional Reactive Programming, Continued*. In Proceedings of the ACM SIGPLAN'02 Haskell Workshop, October 2002.

Literaturhinweise 4(6)

Weitere Arbeiten und Sprachen zu funktionaler reaktiver Programmierung für dezidierte Anwendungsbereiche...

- FAL

  Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

- Frob

  Izzet Pembeci, Henrik Nilsson, Gregory Hager. *Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages*. In Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'02), October 2002.

  John Peterson, Gregory Hager, Paul Hudak. *A Language for Declarative Robotic Programming*. In Proceedings of the International Conference on Robotics and Automation, 1999.

Literaturhinweise 5(6)

Weitere Arbeiten und Sprachen zu funktionaler reaktiver Programmierung für dezidierte Anwendungsbereiche...

- Fvision

  Alastair Reid, John Peterson, Gregory Hager, Paul Hudak. *Prototyping Real-Time Vision Systems: An Experiment in DSL Design*. In Proceedings of the International Conference on Software Engineering, May 1999.

- Fruit

  Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop, September 2001.

Literaturhinweise 6(6)

Weitere Arbeiten zu Real-Time FRP und Event-Driven FRP...

- Real-Time FRP

  Zhanyong Wan, Walid Taha, Paul Hudak. *Real-Time FRP*. In Proceedings of the 6th ACM SIGPLAN'01 International Conference on Functional Programming (ICFP 2001), ACM Press.

  Zhanyong Wan. *Functional Reactive Programming for Real-Time Embedded Systems*. PhD Thesis, Department of Computer Science, Yale University, December 2002.

- Even-Driven FRP

  Zhanyong Wan, Walid Taha, Paul Hudak. *Event-Driven FRP*. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), ACM Press, January 2002.

Lexikalische und syntaktische Analyse

- ...im folgenden kurz *Parsing*

...eine typische Demonstrationsanwendung funktionaler Programmierung mit langer Tradition.

Eine frühe Arbeit ist etwa...

- W. Burge. *Recursive Programming Techniques*, Addison-Wesley, 1975.

Parsing – Implementierungsvarianten

Zwei Spielarten...

- *Kombinator-Parsing*
  $\leadsto$ *recursive descent* Parsing
  - Graham Hutton. *Higher-Order Functions for Parsing*. Journal of Functional Programming 2(3):323-343, 1992.
- *Monadisches Parsing*
  - Graham Hutton, Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.

Literaturhinweis

Die folgende Darstellung orientiert sich i.w. an...

- Kapitel 17
  Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.
- Graham Hutton, Erik Meijer. *Monadic Parsing in Haskell*. Journal of Functional Programming 1(1), 1993.

Parsing informell

Das Grundproblem...

- Lies eine Folge von Objekten vom Typ `a` und
- extrahiere daraus ein(e Liste von) Objekt(en) vom Typ `b`.

Beispiel: Parsen von Ausdrücken

Betrachte...

- Ausdrücke

```
data Expr = Lit Int | Var Name | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod

Op Mul (Op Add (Lit 2) (Lit 3)) (Lit 3)
                        entspricht  ((2+3)*3)
```

Zu lösende (Parsing-) Aufgabe...

- Einlesen von Ausdrücken der Form ((2+3)*3) und Ausgeben des zugehörigen Ausdrucks vom Typ `expr`

Vorüberlegungen 1(2)

Welchen Typ sollte eine Parsing-Funktion haben?

```
type BSParse1 a b = [a] -> b

-- Parser  Eingabe      Erwartete Ausgabe
bracket    "(xyz"  -->  '('
number     "234"   -->  2 oder 23 oder 234 ?
bracket    "234"   -->  kein Ergebnis, Fehlschlag?
```

Zu beantwortende Fragen hierbei: Wie soll sich der Parser verhalten im Falle...

- ...mehrerer Ergebnisse
- ...eines Fehlschlags?

Vorüberlegungen 2(2)

```
type BSParse2 a b = [a] -> [b]

-- Parser  Eingabe      Erwartete Ausgabe
bracket    "(xyz"  -->  ['(']
number     "234"   -->  [2, 23, 234]
bracket    "234"   -->  []
```

Zu beantwortende Frage hierbei:

- Was passiert mit der restlichen Eingabe?

Typ des Parsers 1(2)

Ergebnis der Vorüberlegungen...

```
type Parse a b = [a] -> [(b,[a])]

-- Parser  Eingabe      Erwartete Ausgabe

bracket    "(xyz"  -->  [('(', "xyz")]
number     "234"   -->  [(2,"34"), (23,"4"), (234,"")]
bracket    "234"   -->  []
```

Bemerkung:

- Ausgabe mehrerer Analyseergebnisse erlaubt die Analyse auch mehrdeutiger Grammatiken
  $\leadsto$ *list of successes* technique

Typ des Parsers 2(2)

Vereinbarung:

- *Ausgabe der leeren Liste* ...repräsentiert Fehlschlag der Analyse
- *Ausgabe einer nichtleeren Liste* ...repräsentiert Erfolg der Analyse; jedes Element der Liste ist dann ein Paar, dessen erste Komponente das erkannte Objekt (token) ist und dessen zweite Komponente die noch nicht gelesene Eingabe ist

Grundlegende/Basis-Parser 1(3)

- Primitive, eingabeunabhängige Parser

```
-- Der nie erfolgreiche Parser
none :: Parse a b
none inp = []

-- Der stets erfolgreiche Parser
succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

Bemerkung: Der `succeed`-Parser konsumiert seine Eingabe nicht. In BNF-Notation entspricht er dem ε-Symbol für das leere Wort.

Grundlegende/Basis-Parser 2(3)

- Erkennung einzelner Objekte (token)...

```
token :: Eq a => a -> Parse a a
token t (x:xs)
  | t == x     =  [(t,xs)]
  | otherwise  =  []
token t []     =  []
```

- Erkennung einzelner Objekte mit einer bestimmten qualifizierenden Eigenschaft...

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x        =  [(x,xs)]
  | otherwise  =  []
spot p []      =  []
```


Grundlegende/Basis-Parser 3(3)

Anwendung:

```
bracket = token '('
dig     = spot isDigit

isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')
```

Beachte: ...`token` mittels `spot` ausdrückbar

```
token t = spot (== t)
```

Zusammengesetzte/Kombinierte Parser 1(4)

...liefern komplexe(re) Parser

⇝ *Combinator Parsing*

- Alternativen

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp
```

Beispiel:

```
(bracket 'alt' dig) "234"  -->  [] ++ [(2,"34")]
```


Zusammengesetzte/Kombinierte Parser 2(4)

- Sequenzen/Hintereinanderausführung

```
infixr 5 >*>
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp,
                                (z,rem2) <- p2 rem1 ]
```


Zusammengesetzte/Kombinierte Parser 3(4)

Beispiel:

```
wegen  number "24(" --> [(2,"4("), (24,"(")]  zunaechst

(number >*> bracket) "24("
   -->  [((y,z),rem2) | (y,rem1) <- [(2,"4("), (24,"(")],
                         (z,rem2) <- bracket rem1 ]
   -->  [((2,z),rem2) | (z,rem2) <- bracket "4(" ] ++
        [((24,z),rem2) | (z,rem2) <- bracket "(" ]

   -->  [] ++ [((24,z),rem2) | (z,rem2) <- bracket "(" ]

wegen  bracket "("  -->  [('(',"")]  somit insgesamt

   -->  [((24,z),rem2) | (z,rem2) <- [('(',"")] ]
   -->  [ ((24,'('), "" ]
```


Zusammengesetzte/Kombinierte Parser 4(4)

- Transformation/Modifikation

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```

Beispiel:

```
(digList 'build' digsToNum) "21a3"
   --> [ (digsToNum x,rem) | (x,rem) <- digList "21a3" ]
   --> [ (digsToNum x,rem) | (x,rem) <-
                             [("2","1a3"),("21","a3")]]
   --> [ (digsToNum "2", "1a3"), (digsToNum "21", "a3") ]
   --> [ (2,"1a3"), (21,"a3") ]
```

Bsp.: Parsen einer Liste von Objekten

...gegeben einen Parser zum Erkennen eines einzelnen Objektes

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry (:)))
```


Zusammenfassung und Zwischenfazit

...über das Kombinieren von Parsern (*parser combinators*)

- Parser im obigen Stil sind strukturell der Form von BNF-Grammatiken ähnlich. Die Operatoren der BNF-Grammatik haben auf Parserseite eine direkte Entsprechung in Form einer Funktion höherer Ordnung.
- Diese Funktionen höherer Ordnung *kombinieren* einfache(re) Parser zu komplexe(re)n neuen Parsern.
- Man spricht daher von *kombinierenden Formen* (engl. *combining forms*) oder kurz von *Kombinatoren* (engl. *combinators*) (vgl. dazu z.B. Graham Hutton. *Higher-Order Functions for Parsing*.

Die Parser-Fkt. im Überblick 1(4)

```
-- Sequenzoperator
infixr 5 >*>

-- Parser-Typ
type Parse a b = [a] -> [(b,[a])]

-- Eingabeunabhaengige Parser
none :: Parse a b
none inp = []

succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```


Die Parser-Fkt. im Überblick 2(4)

```
-- Erkennen einzelner Objekte
token :: Eq a => a -> Parse a a
token t = spot (==t)

-- Erkennen einzelner Objekte mit bestimmter Eigenschaft
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x        =  [(x,xs)]
  | otherwise  =  []
spot p []      =  []
```

Die Parser-Fkt. im Überblick 3(4)

```
-- Alternativen
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp

-- Sequenzen
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp, (z,rem2) <- p2 rem1 ]

-- Transformation/Modifikation
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```


Die Parser-Fkt. im Überblick 4(4)

```
-- Anwendungsbeispiel
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry (:)))
```


Anwendung: Zurück zum Ausgangsbeispiel

Wir betrachten Ausdrücke der Form...

```
data Expr = Lit Int | Var Name | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod

Op Add (Lit 2) (Lit 3)     entspricht   2+3
```

...wobei weiter die folgende Vereinbarung gelte:

- *Literale* ...67, ~89, etc., mit ~ für einstelliges Minus
- *Namen* ...die Kleinbuchstaben von 'a' bis 'z'
- *Anwendungen der binären Operationen* ...$+, *, -, /, \%$, wobei % für die modulo-Operation steht und / die ganzzahlige Division bezeichnet.

Ein Parser für Ausdrücke 1(3)

```
parser :: Parse Char Expr
parser = litParse 'alt' nameParse 'alt' opExpParse

nameParse :: Parse Char Expr
nameParse = spot isName 'build' Name

isName :: Char -> Bool
isName x = ('a' <= x && x <= 'z')
```

Ein Parser für Ausdrücke 2(3)

```
opExpParse
  = (token '(' >*>
     parser    >*>
     spot isOp >*>
     parser    >*>
     token ')')
     'build' makeExpr

litParse
  = ((optional (token '~')) >*>
     (neList (spot isDigit))
    'build' (charlistToExpr . uncurry (++))
```


Ein Parser für Ausdrücke 3(3)

Verwendete Hilfsfunktionen wie...

- `isOp`
- `charlistToExpr`
- ...

bleiben noch zu definieren.

Top-level Parser

```
topLevel :: Parse a b -> [a] -> b
topLevel p inp
  = case results of
      [] -> error ``parse unsuccessful''
      _  -> head results
    where
    results = [ found | (found, []) <- p inp ]
```


Zusammenfassung und Fazit 1(2)

Parser der Form...

```
type Parse a b = [a] -> [(b,[a])]

none :: Parse a b
succeed :: b -> Parse a b
spot :: (a -> Bool) -> Parse a a
alt :: Parse a b -> Parse a b -> Parse a b
>*> :: Parse a b -> Parse a c -> Parse a (b,c)
build :: Parse a b -> (b -> c) -> Parse a c
topLevel :: Parse a b -> [a] -> b
```

...unterstützen besonders die Konstruktion sog. *recursive descent* Parser.

Zusammenfassung und Fazit 2(2)

Folgende Spracheigenschaften werden besonders ausgenutzt...

- *Funktionen höherer Ordnung* ...`Parse a b` ist von einem Funktionstyp, somit sind alle Parserkombinatoren Funktionale
- *Polymorphie* ...siehe wieder Typ von `Parse a b`, obige Parserkombinatoren sind für andere (Token-) und Zieltypen unmittelbar wiederverwendbar
- *Lazy evaluation* ... *"on demand"*-Erzeugung der möglichen Parsegänge, automatisches backtracking

Monadisches Parsing

```
newtype Parser a = Parser (String -> [(a,String)])
```

Vereinbarung auch hier:

- *Ausgabe der leeren Liste* ...repräsentiert Fehlschlag der Analyse
- *Ausgabe einer nichtleeren Liste* ...repräsentiert Erfolg der Analyse; jedes Element der Liste ist dann ein Paar, dessen erste Komponente das erkannte Objekt (token) ist und dessen zweite Komponente die noch nicht gelesene Eingabe ist

Grundlegende/Basis-Parser

- Erkennung eines einzelnen Zeichens...

```
item :: Parser Char
item  = Parser (\cs -> case cs of
                          ""     -> []
                          (c:cs) -> [(c,cs)])
```

Vgl.: `item` vs. `token`

Die Parser-Monade

Erinnerung: Die Klasse Monade...

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b
```

Beachte: `Parser` ist ein Typkonstruktor. Dies ermöglicht...

```
instance Monad Parser where
   -- Der stets erfolgreiche Parser
   return a = Parser (\cs -> [(a,cs)])
   -- Sequenzen/Hintereinanderausf\"uhrung
   p >>= f  = Parser (\cs -> concat [parse (f a) cs' |
                                     (a,cs') <- parse p cs])
```

Vgl.: `return` vs. `succeed` und `(>>=)` vs. `infixr`

Eigenschaften von `return` und `(>>=)`

Wie für Instanzen der Klasse `Monad` gefordert, lässt sich zeigen...

```
            return a >>= f  =  f a
              p >>= return  =  p
p >>= (\a -> (f a >>= g))   =  (p >>= (\a -> f a)) >>= g
```

Erinnerung:

- Obige Eigenschaften sind für jede Instanz der Klasse `Monad` gefordert, nicht nur für den Spezialfall der Parser-Monade
  - ...`return` ist Links- und Rechtseins für `(>>=)`
    ⇝ ...erlaubt einige Parser einfacher anzugeben
  - ...`(>>=)` ist assoziativ
    ⇝ ...erlaubt Klammern bei Hinterausführung von Parsern zu unterdrücken

Typischer Aufbau eines Parsers 1(2)

...mit Hilfe des `(>>=)`-Operators

```
p1 >>= \a1 ->
p2 >>= \a2 ->
...
pn >>= \an ->
f a1 a2 ... an
```

Intuition:

- Wende Parser `p1` an und nenne dessen Resultat `a1`
- Wende anschließend Parser `p2` an und nenne dessen Resultat `a2`
- ...
- Wende schließlich Parser `pn` an und nenne dessen Resultat `an`
- Kombiniere abschließend die Teilergebnisse durch Anwendung einer geeigneten Funktion `f`

Typischer Aufbau eines Parsers 2(2)

Die `do`-Notation erlaubt folgende elegantere Notation...

```
do a1 <- p1
   a2 <- p2
   ...
   an <- pn
   f a1 a2 ... an
```

Alternativ in einer Zeile notiert...

```
do {a1 <- p1; a2 <- p2; ...; an <- pn; f a1 a2 ... an}
```


Sprech- und Schreibweisen

Ausdrücke der Form

- `ai <- pi` heißen *Generatoren*
  (da sie Werte für die Variablen `ai` erzeugen)

Bemerkung:

Ein Generator der Form `ai <- pi` kann durch

- `pi` ersetzt werden, wenn der erzeugte Wert nicht weiter benötigt wird

Beispiel

```
p :: Parser (Char,Char)
p  = do {c <- item; item; d <- item; return (c,d)}
```

Informell: Parser `p`...

- liest drei Zeichen
- verwirft davon das zweite Zeichen und
- gibt das erste und dritte Zeichen als Paar zurück

Parser-Erweiterungen 1(2)

Monade mit *Null* und `plus`...

```
class Monad m => MonadZero m where
   zero :: m a

class MonadZero m => MonadPlus m where
   (++) :: m a -> m a -> m a
```


Parser-Erweiterungen 2(2)

```
instance MonadZero Parser where
   zero   = Parser (\cs -> [])

instance MonadPlus Parser where
   p ++ q = (\cs -> parse p cs ++ parse q cs)
```


Einfache Eigenschaften 1(2)

Es lässt sich zeigen...

```
     zero ++ p  =  p
     p ++ zero  =  p
p ++ (q ++ r)   =  (p ++ q) ++ r
```

Bem.: Obige Eigenschaften sind für jede Monade mit *Null* und `plus` gefordert

Informell:

- ...`zero` ist Links- und Rechtseins für `(++)`
- ...`(++)` ist assoziativ

Einfache Eigenschaften 2(2)

Speziell für die Parser-Monade lässt sich zeigen...

```
             zero >>= f  =  zero
        p >>= const zero  =  zero
          (p ++ q) >>= f  =  (p >>= f) ++ (q >>= f)
 p >>= (\a -> f a ++ g a)  =  (p >>= f) ++ (p >>= g)
```

Informell:

- ...`zero` ist Links- und Rechtsnull für (>>=)
- ...(>>=) distribuiert über (++)

Deterministische Auswahl

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p ++ q) cs of
                             []     -> []
                             (x:xs) -> [x])
```

Beachte:

- (+++) zeigt dasselbe Verhalten wie (++), liefert jedoch höchstens ein Resultat ab
- (+++) erfüllt alle der vorher genannten Eigenschaften von (++)

Weitere Parser

Erkennung...

- einzelner Objekte mit einer bestimmten qualifizierenden Eigenschaft

  ```
  sat   :: (Char -> Bool) -> Parser Char
  sat p = do {c <- item; if p c then return c else zero}
  ```

- einzelner Objekte

  ```
  char   :: Char -> Parser Char
  char c = sat (c ==)
  ```

- von Ziffernfolgen, Klein- und Großbuchstaben, etc.
  ...analog zu `char`

Vgl.: `sat` und `char` vs. `spot` und `token`

Rekursions-Kombinatoren 1(3)

Parser können oft rekursiv definiert werden...

```
-- Parsen einer Zeichenreihe
string        :: String -> Parser String
string ""      = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}

-- Parsen wiederholter Anwendungen von Parser p
many   :: Parser a -> Parser [a]
many p = many1 p +++ return []

many1   :: Parser a -> Parser [a]
many1 p = do {a <- p; as <- many p; return (a:as)}
```

Rekursions-Kombinatoren 2(3)

```
-- wie many mit eingestreuten Anwendungen des Parsers sep
sepby        :: Parser a -> Parser b -> Parser [a]
p 'sepby' sep = (p 'sepby1' sep) +++ return []

sepby1        :: Parser a -> Parser b -> Parser [a]
p 'sepby1' sep = do a  <- p
                    as <- many (do {sep; p})
                    return (a:as)
```


Rekursions-Kombinatoren 3(3)

```
chainl       :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op a = (p 'chainl1' op) +++ return a

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p 'chainl1' op = do {a <- p; rest a}
                 where
                    rest a = (do f <- op
                                 b <- p
                                 rest (f a b))
                             +++ return a
```


Lexikalische Kombinatoren

Geeignete Kombinatoren erlauben eine lexikalische Analyse (token-Erkennung) zu unterdrücken...

```
-- Parsen einer Zeichenreihe mit Leerstellen und Zeilenumbruechen
space :: Parser String
space  = many (sat isSpace)

-- Parsen eines Tokens mit Hilfe des Parsers p
token  :: Parser a -> Parser a
token p = do {a <- p; space; return a}

-- Parsen eines Symboltokens
symb   :: String -> Parser String
symb cs = token (string cs)

-- Anwenden eines Parsers p, entfernen fuehrender Leerstellen
apply  :: Parser a -> String -> [(a,String)]
apply p = parse (do {space; p}]
```


Beispiel: Parsen von Ausdrücken 1(3)

Grammatik:

```
expr   ::=  expr addop term | term
term   ::=  term mulop factor | factor
factor ::=  digit | (expr)
digit  ::=  0 | 1 | ... | 9

addop  ::=  + | -
mulop  ::=  * | /
```

Beispiel: Parsen von Ausdrücken 2(3)

Parsen und evaluieren von Ausdrücken zu ihrem ganzzahligen Wert...

```
expr  :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)

expr   = term   'chainl1' addop
term   = factor 'chainl1' mulop
factor = digit +++ do {symb "("; n <- expr; symb ")"; return n}
digit  = do {x <- token (sat isDIgit); return (ord x - ord '0')}

addop  = do {symb "+"; return (+)} +++ do {symb "-"; return (-)}
mulop  = do {symb "*"; return (*)} +++ do {symb "/"; return (div)}
```


Beispiel: Parsen von Ausdrücken 3(3)

Beispiel:

```
apply expr " 1 - 2 * 3 + 4 "
                  -->   [(-1,"")] wie gewuenscht
```


Weiterführende und vertiefende Literatur 1(3)

Zum Kombinator-Parsing...

- J. Fokker. *Functional Parsers*. In: *Advanced Functional Programming, First International Summer School*, Springer, LNCS 925 (1995), 1-23.
- S. Hill. *Combinators for Parsing Expressions*. Journal of Functional Programming 6:445-463, 1996.
- P. Koopman, R. Plasmeijer. *Efficient Combinator Parsers*. In Proceedings of Implementation of Functional Languages, Springer, LNCS 1595 (1999), 122-138.

Weiterführende und vertiefende Literatur 2(3)

Zu fehlerkorrigierenden Parsern...

- P. Wadler. *How to Replace Failure with a List of Successes*, in: Functional Programming Languages and Computer Architectures, Springer, LNCS 201 (1985), 113 - 128.
- D. Swierstra, P. Azero Alcocer. *Fast, Error Correcting Parser Combinators: A Short Tutorial*. In Proceedings SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, Springer, LNCS 1725 (1999), 111-129.
- D. Swierstra, L. Duponcheel. *Deterministic, Error Correcting Combinator Parsers*. In: *Advanced Functional Programming, Second International Spring School*, Springer, LNCS 1129 (1996), 184-207.

Weiterführende und vertiefende Literatur 3(3)

Zu Parser-Bibliotheken...

- Daan Leijen, Erik Meijer. *Parsec: A Practical Parser Library*. Electronic Notes in Theoretical Computer Science 41(1), 2001.

- A. Gill, S. Marlow. *Happy – The Parser Generator for Haskell*. University of Glasgow, 1995.
`http://www.haskell.org/happy`

Nächster (und letzter) Vorlesungstermin im SS 2005...

- Di, 28.06.2005, 14 - 16 Uhr, CompLang-Bibliothek