

# **Advanced Functional Programming**

2010-2011, periode 2

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

November 22, 2011

# 3. Advanced parser Combinators





A DSL is a programming language tailored for a particular application domain, which captures precisely the semantics of the application domain:

A DSL is a programming language tailored for a particular application domain, which captures precisely the semantics of the application domain:

- Lex / Yacc (lexing and parsing)
- LATEX(for document mark-up)
- ► Tcl/Tk (GUI scripting)
- MatLab (numerical computations)

#### Advantages of using a DSL are that programs:

- are easier to understand
- ► are easier to write
- are easier to maintain
- can be written by non-programmers

#### Advantages of using a DSL are that programs:

- are easier to understand
- are easier to write
- are easier to maintain
- can be written by non-programmers

#### Disadvantages:

- High start-up cost
  - design and implementation of a new language is hard
- ► Lack of "general purpose" features (e.g. abstraction)
- ▶ Little tool support





# 3.1 Embedded Domain Specific Languages



### **Domain Specific Embedded Languages**

#### Embed a DSL as a library in a general purpose host language:

- ▶ inherit general purpose features from host language
  - abstraction mechanism
  - type system
- inherit compilers and tools
- good integration with host language
- many DSL's can easily be used together!



### **Domain Specific Embedded Languages**

#### There are however also disadvantages:

- Constrained by host language
  - Syntax
  - Type system
- ► Implementation shines through

Haskell is a very suitable host for DSEL's:

► Polymorphism

Haskell is a very suitable host for DSEL's:

- ► Polymorphism
- ► Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.

Haskell is a very suitable host for DSEL's:

- ► Polymorphism
- ▶ Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- ► Lazy evaluation:

Haskell is a very suitable host for DSEL's:

- Polymorphism
- ► Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- ► Lazy evaluation:
  - allows for writing programs that analyse and transform themselves

#### Haskell is a very suitable host for DSEL's:

- ► Polymorphism
- ► Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- ► Lazy evaluation:
  - allows for writing programs that analyse and transform themselves
  - gives you partial evaluation for free

#### Haskell is a very suitable host for DSEL's:

- ► Polymorphism
- ▶ Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:
  - allows for writing programs that analyse and transform themselves
  - gives you partial evaluation for free
- ▶ Infix syntax allows you to make programs look nice

#### Haskell is a very suitable host for DSEL's:

- ► Polymorphism
- ▶ Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:
  - allows for writing programs that analyse and transform themselves
  - gives you partial evaluation for free
- ▶ Infix syntax allows you to make programs look nice
- ► List and monad comprehensions

#### Haskell is a very suitable host for DSEL's:

- ► Polymorphism
- ▶ Higher-order functions: results can be functions decribing the *denotational semantics* of an expression.
- Lazy evaluation:
  - allows for writing programs that analyse and transform themselves
  - gives you partial evaluation for free
- ▶ Infix syntax allows you to make programs look nice
- List and monad comprehensions
- ▶ Type classes for passing extra arguments in an implicit way



# **Examples**

- Parser combinators
- Pretty printing libraries
- HaskelIDB for implicitly generating SQL
- QuickCheck
- GUI libraries
- WASH/CGI for describing HTML pages
- Haskore for describing music
- Agent based systems
- ► Financial combinators for describing financial products

# **General Purpose Programming Language**

Old idea:

A general purpose programming language should make it easy to write any kind of program.

# **General Purpose Programming Language**

Old idea:

A general purpose programming language should make it easy to write any kind of program.

New idea:

A general purpose programming language should make it possible to write very powerful libraries; even if this is a substantial effort.

#### We will:

show an example of a combinator language

#### We will:

- show an example of a combinator language
- show how it can be extended

#### We will:

- show an example of a combinator language
- show how it can be extended
- ▶ show how we can produce smarter implementations

#### We will:

- show an example of a combinator language
- show how it can be extended
- show how we can produce smarter implementations
- show how e can emded "compilers". i.e. how we can analyse and transform the EDSL

# What are parser combinators

- ▶ a collection basic parsing functions that recognise a piece of input
- ▶ a collection of combinators that build new parsers out of existing ones

# What are parser combinators

- a collection basic parsing functions that recognise a piece of input
- a collection of combinators that build new parsers out of existing ones

For the time being parsers are just Haskell functions, but we will extend on that later.

### **Elementary Parsers**

All libraries are based on at least the following four basic combinators:

### **Elementary Parsers**

All libraries are based on at least the following four basic combinators:

	Haskell
alternative	p < > p
composition	p <*> q
terminals	pSym 's'
empty string	pSucceed v



A parser takes a string and returns result of type a

 $\mathsf{String} \to \mathsf{a}$ 

A parser may be ambiguous, and there may be many possible ways of recognizing a value of type a:

 $\mathsf{String} \to [\mathsf{a}]$ 

A parser may not consume the whole input, and return also the unused part of the input:

 $\mathsf{String} \to [(\mathsf{a},\mathsf{String})]$ 

Why should we only accept character strings? Everything goes, as long as we have equality defined for the tokens:

$$\mathsf{Eq}\;\mathsf{s}\Rightarrow [\mathsf{s}]\rightarrow [(\mathsf{a},[\mathsf{s}])]$$

Why should we only accept character strings? Everything goes, as long as we have equality defined for the tokens:

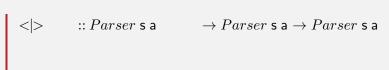
Eq 
$$s \Rightarrow [s] \rightarrow [(a,[s])]$$

Which we give a name:

**type** 
$$Parser$$
 s  $a = Eq s \Rightarrow [s] \rightarrow [(a,[s])]$ 

### Types of the Elementary Combinators

### **Types**



Try to remember these types. Knowing the types is half the work when programming in Haskell.

### Types of the Elementary Combinators

#### **Types**

```
<\mid>> :: Parser s a \longrightarrow Parser s a \longrightarrow Parser s a 
 <math>\Rightarrow Parser s b \longrightarrow Parser s (a, b)
```

Try to remember these types. Knowing the types is half the work when programming in Haskell.

### Types of the Elementary Combinators

### **Types**

Try to remember these types. Knowing the types is half the work when programming in Haskell.

4□▶
4□▶
4□▶
4□▶
4□▶
4□
9
0

### Types of the Elementary Combinators

#### **Types**

Try to remember these types. Knowing the types is half the work when programming in Haskell.

### Types of the Elementary Combinators

#### **Types**

Try to remember these types. Knowing the types is half the work when programming in Haskell.



### **Types of the Elementary Combinators**

#### **Types**

Try to remember these types. Knowing the types is half the work when programming in Haskell.

## 3.2 Simple "List of Succeses" Implementation





$$pFail \_ = []$$





$$pFail _ = []$$
  
 $pSucceed v inp = [(v, inp)]$ 



$$pFail \_ = []$$
 $pSucceed v inp = [(v, inp)]$ 
 $(p < |> q) inp = p inp + q inp$ 

```
\begin{array}{ll} pFail \ \_ &= [\,] \\ pSucceed \ v \ inp = [\,(v,inp)\,] \\ (p < \mid > q) & inp = p \ inp \ + q \ inp \\ pSym \ a & inp = \mathbf{case} \ inp \ \mathbf{of} \\ & (s:ss) \rightarrow \mathbf{if} \quad a == s \\ & \quad \mathbf{then} \ [\,(s,ss)\,] \\ & \quad \mathbf{else} \ [\,] \\ & \quad [\,] \quad \rightarrow [\,] \end{array}
```

```
pFail = []
pSucceed v inp = [(v, inp)]
(p < |> q) inp = p inp + q inp
pSym a inp = case inp of
                   (s:ss) \rightarrow if a == s
                             then [(s, ss)]
                             else []
                   | | \rightarrow []
(p \ll q) inp = [(a2b a, rr)]
                    |(a2b, r) \leftarrow p inp
                   (a, rr) \leftarrow q r
```

#### Left Factorisation I

It is not a good idea to have parsers that have alternatives starting with the same element:

$$p = f < $> a < *> b$$
  
 $<|> g < $> a < *> c$ 

#### Left Factorisation I

It is not a good idea to have parsers that have alternatives starting with the same element:

$$p = f <\$> a <\!\!\!*> b$$
   
 <|> g <\mathref{\$>\$} a <\!\!\!\*> c

So we define:

$$flip f x y = f y x$$

[Faculty of Science Information and Computing Sciences



#### Left Factorisation II

We want to recognise expressions with as result a value of the type:

```
data Expr = Lambda Id Expr | App Expr Expr | TypedExpr Type Expr
```

#### Left Factorisation II

We want to recognise expressions with as result a value of the type:

```
\begin{tabular}{lll} \mbox{\bf data} \ \mbox{Expr} &= \mbox{Lambda} & \mbox{Id} & \mbox{Expr} \\ &| \mbox{App} & \mbox{Expr} & \mbox{Expr} \\ &| \mbox{TypedExpr} & \mbox{Type} & \mbox{Expr} \\ \end{tabular}
```

#### Left Factorisation II

We want to recognise expressions with as result a value of the type:

$$\begin{split} \mathsf{pExpr} &= & pChainl \; (pSucceed \; \mathsf{App}) \; \mathsf{pFactor} \\ &? ( & \mathsf{TypedExpr} \\ &<\$ \; \; \mathsf{pTok} \; "::" \\ &<\!\! *> \; \mathsf{pType} \end{split}$$



#### 3.3 Monadic Parsers





## The Chomsky Hierarchy

#### The Chomsky hierarchy:

- Regular
- Context-free
- Context-sensitive
- ► Recursively enumerable

It is well known that context free grammars have limited expressibility.



```
\label{eq:character} \begin{array}{ll} \mathsf{times} & :: \mathsf{Int} \to Parser \; \mathsf{Char} \; \mathsf{a} \to Parser \; \mathsf{Char} \; \mathsf{a} \\ 0 \; \text{`times'} \; \mathsf{p} = pSucceed \; [ \, ] \\ \mathsf{n} \; \text{`times'} \; \mathsf{p} = (:) < \! \mathsf{p} > \! \mathsf{p} < \! \mathsf{m} - 1) \; \mathsf{`times'} \; \mathsf{p} \\ \end{array}
```

```
times :: Int \rightarrow Parser Char a \rightarrow Parser Char a 0 'times' p = pSucceed [] n 'times' p = (:) <$> p <*> (n - 1) 'times' p abc n = n <$ (n 'times' a) <* (n 'times' b) <* (n 'times' c)
```

```
times :: Int \rightarrow Parser Char a \rightarrow Parser Char a 0 'times' p = pSucceed [] n 'times' p = (:) <$> p <> (n - 1) 'times' p abc n = n <$ (n 'times' a) <* (n 'times' b) <* (n 'times' c) ABC = foldr (<|>) pFail [abc n | n \leftarrow 0 . .]
```

```
times :: \operatorname{Int} \to Parser \operatorname{Char} \operatorname{a} \to Parser \operatorname{Char} \operatorname{a} 0 'times' \operatorname{p} = pSucceed [] \operatorname{n} 'times' \operatorname{p} = (:) < > \operatorname{p} > \operatorname{m} > (\operatorname{n} - 1) 'times' \operatorname{p} > \operatorname{abc} \operatorname{n} = \operatorname{n} <  \operatorname{n} 'times' \operatorname{a} > \operatorname{m} > \operatorname{m}
```

We admit that this is not very efficient, but left factorisation is not so easy since the corresponding context free grammar is infinite.

Wouldn't it be nice if we could start by just recognizing a sequence of a s, and then use the result to enforce the right number of b s and c s?

Wouldn't it be nice if we could start by just recognizing a sequence of a s, and then use the result to enforce the right number of b s and c s?

instance Monad  $(Parser \ s)$  where  $p \ (\ggg) \ q = \lambda inp \rightarrow [(qv, rr) \mid (pv, r) \leftarrow p \quad inp \\ \quad , \ (qv, rr) \leftarrow q \ pv \ r$   $\qquad \qquad \qquad ]$   $return \ v = \lambda inp \rightarrow [(v, inp)]$ 

Wouldn't it be nice if we could start by just recognizing a sequence of a s, and then use the result to enforce the right number of b s and c s?

イロトイクトイミトイミト ま めのべ

Wouldn't it be nice if we could start by just recognizing a sequence of a s, and then use the result to enforce the right number of b s and c s?

```
instance Monad (Parser s) where
   p \gg q = \lambda inp \rightarrow [(qv, rr) \mid (pv, r) \leftarrow p]
                                                               inp
                                        (qv, rr) \leftarrow q pv r
   return v = \lambda inp \rightarrow [(v, inp)]
                :: Parser Char Int
as
               = length \langle \$ \rangle pList (pSym 'a')
as
               = n < $ (n 'times' b) < * <math>(n 'times' c)
bc n
ABC
               = do n \leftarrow as
                       bc n
```



#### 4. Problems with "List of Successes"





[Faculty of Science Information and Computing Sciences]

▶ If your input does not conform to the language recognized by the parser all you get as a result is: [].

- ▶ If your input does not conform to the language recognized by the parser all you get as a result is: [].
- ▶ It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.



- ▶ If your input does not conform to the language recognized by the parser all you get as a result is: [].
- ▶ It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.
- ▶ There is no indication of where things went wrong.

- ▶ If your input does not conform to the language recognized by the parser all you get as a result is: [].
- ▶ It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.
- ▶ There is no indication of where things went wrong.

- ▶ If your input does not conform to the language recognized by the parser all you get as a result is: [].
- ▶ It may take quite a while before you get this negative result, since the backtracking may try all other alternatives at all positions.
- ▶ There is no indication of where things went wrong.

These problem have been cured in both Parsec and the uu-parsinglib library. The latter does this:

- without much overhead
- no need for help from the programmer
- ▶ without stopping; most errors can be found in a single run



### **Left Recursions**

As with any top-down parsing method, having left-recursive parsers is no good

- You get non-terminating parsers
- ► You get no error messages





### **Left Recursions**

As with any top-down parsing method, having left-recursive parsers is no good

- You get non-terminating parsers
- You get no error messages

This problem can be partially been cured as we will see later.



### **Problems with Space Consumption**

- ► A complete result has to be constructed before any part of it is returned
- ► The complete input is present in memory as long as no parse has been found
- ► Efficiency may depend critically on the ordering of the alternatives, and thus on how the grammar was written

For all of these problems we have found solutions.



### **5.** Solving the problems



[Faculty of Science Information and Computing Sciences]

## Replace depth-first by breath-first

In order to replace depth-first by breadth-first we have to run all possible parses in parallel.

### Replace depth-first by breath-first

In order to replace depth-first by breadth-first we have to run all possible parses in parallel.

In order to make things not too complicated we start by defining recognisers first. A recogniser is a parser that does not return a result.

### Replace depth-first by breath-first

In order to replace depth-first by breadth-first we have to run all possible parses in parallel.

In order to make things not too complicated we start by defining recognisers first. A recogniser is a parser that does not return a result.

In order to be able to inspect the progress made during the recognition-process we let recognisers return a list of Steps.

```
data Steps = Step Steps
| Fail
| Done
```

Faculty of Science

# 5.1 Recognisers



### Recognisers

**type** Recogniser 
$$s = ([s] \rightarrow \mathsf{Steps}) \rightarrow ([s] \rightarrow \mathsf{Steps})$$



[Faculty of Science

### Recognisers

$$\textbf{type} \; \mathsf{Recogniser} \; \mathsf{s} = ([\mathsf{s}] \to \mathsf{Steps}) \to ([\mathsf{s}] \to \mathsf{Steps})$$

$$\xrightarrow{p} \xrightarrow{Steps}$$

$$p \ll q = \lambda k \text{ inp} \rightarrow p (q k) \text{ inp}$$



[Faculty of Science





イロトイクトイミトイミト ヨ かなべ

$$p \ll q = \lambda k \text{ inp} \rightarrow p (q k) \text{ inp}$$





$$p \ll q = \lambda k \text{ inp} \rightarrow p (q k) \text{ inp}$$



4日▶4畳▶4畳▶4畳▶ 畳 夕久@

# **Basic Recognisers**

```
pSucceed \ v \ k \ inp = k \ inp-- \ pSucceed \ v = id -- pSucceed = const pSymb \ a \ k \ inp = case \ inp \ of (s:ss) \rightarrow if \ s == a \ then \ Step \ (k \ ss) else Fail [] \rightarrow Fail parse p = p \ (\lambda inp \rightarrow if \ null \ inp \ then \ Done \ else \ Fail)
```

# **Basic Recognisers**

```
\begin{split} pSucceed \; \mathsf{v} \; \mathsf{k} \; \mathsf{inp} &= \mathsf{k} \; \mathsf{inp}\text{--} \; \mathsf{pSucceed} \; \mathsf{v} \; = \; \mathsf{id} \\ &\quad \mathsf{--} \; \mathsf{pSucceed} \; = \; \mathsf{const} \\ \mathsf{pSymb} \; \mathsf{a} \; \mathsf{k} \; \mathsf{inp} &= \; \mathsf{case} \; \mathsf{inp} \; \mathsf{of} \\ &\quad (\mathsf{s} : \mathsf{ss}) \to \mathsf{if} \; \mathsf{s} \coloneqq \mathsf{a} \; \mathsf{then} \; \mathsf{Step} \; (\mathsf{k} \; \mathsf{ss}) \\ &\quad \mathsf{else} \; \mathsf{Fail} \\ [] &\quad \to \mathsf{Fail} \\ \\ \mathsf{parse} \; \mathsf{p} &= \; \mathsf{p} \; (\lambda \mathsf{inp} \to \mathsf{if} \; \mathsf{null} \; \mathsf{inp} \; \mathsf{then} \; \mathsf{Done} \; \mathsf{else} \; \mathsf{Fail}) \end{split}
```

► For each successfully recognised symbol we insert a Step in the result.

# **Basic Recognisers**

```
pSucceed \ v \ k \ inp = k \ inp-- \ pSucceed \ v = id -- pSucceed = const pSymb \ a \ k \ inp = case \ inp \ of (s:ss) \rightarrow if \ s == a \ then \ Step \ (k \ ss) else Fail [] \rightarrow Fail parse p = p \ (\lambda inp \rightarrow if \ null \ inp \ then \ Done \ else \ Fail)
```

- ► For each successfully recognised symbol we insert a Step in the result.
- If there is no remaining input at the end of the parse we mark this by making Done the last element of the result. In all other cases we Fail.



We now define the <|> operator, which uses a function best:

$$\mathbf{p} < \mid > \mathbf{q} = \lambda \mathbf{k} \ \mathsf{inp} \rightarrow \mathbf{p} \ \mathbf{k} \ \mathsf{inp} \ \mathsf{`best'} \ \mathbf{q} \ \mathbf{k} \ \mathsf{inp}$$

(Step ps) 'best' (Step qs) = Step (bs 'best' qs)

Fail 'best' qs = qs

ps 'best' Fail = ps

Done 'best' Done = error "ambiguous grammar"

We now define the <|> operator, which uses a function best:

$$p < |> q = \lambda k \text{ inp} \rightarrow p \text{ k inp 'best' } q \text{ k inp}$$

$$(Step ps) 'best' (Step qs) = Step (bs 'best' qs)$$
Fail 'best' qs = qs
$$ps \quad 'best' Fail = ps$$
Done 'best' Done = error "ambiguous grammar"

pattern matching drives the computation

We now define the <|> operator, which uses a function best:

```
\begin{array}{lll} p < \mid > q = \lambda k \; inp \to p \; k \; inp \; \text{`best'} \; q \; k \; inp \\ (Step \; ps) \; \text{`best'} \; (Step \; qs) = Step \; (bs \; \text{`best'} \; qs) \\ Fail & \; \text{`best'} \; qs & = qs \\ ps & \; \text{`best'} \; Fail & = ps \\ Done & \; \text{`best'} \; Done & = error \; "ambiguous \; grammar" \end{array}
```

- pattern matching drives the computation
- ► the function best already returns part of its result by inspecting an initial part of its arguments!

We now define the <|> operator, which uses a function best:

```
p < |> q = \lambda k \text{ inp} \rightarrow p \text{ k inp 'best' } q \text{ k inp}
(Step ps) \text{ 'best' } (Step qs) = Step (bs 'best' qs)
Fail \quad \text{'best' } qs = qs
ps \quad \text{'best' } Fail = ps
Done \quad \text{'best' } Done = error \text{ "ambiguous } grammar\text{"}
```

- pattern matching drives the computation
- the function best already returns part of its result by inspecting an initial part of its arguments!
- effectively all recognisers are driven so they run in parallel or die: we have achieved a breadth-first recognition.



◆□▶◆御▶◆団▶◆団▶ 団 めの◎

# **5.2** History Parsers



### **Extending the Steps data type**

We extend the Steps data type so it can hold a computed result (using a GADT and an existential type, to which we will come back later).

#### data Steps a where

 $\mathsf{Step} \; :: \qquad \mathsf{Progress} \to \mathsf{Steps} \; \mathsf{a} \to \mathsf{Steps} \; \mathsf{a}$ 

Apply ::  $\forall a \ b.(b \rightarrow a) \rightarrow \mathsf{Steps} \ b \rightarrow \mathsf{Steps} \ a$ 

Fail :: . . .

Faculty of Science

### **Extending the Steps data type**

We extend the Steps data type so it can hold a computed result (using a GADT and an existential type, to which we will come back later).

#### data Steps a where

$$\begin{array}{lll} \mathsf{Step} & :: & \mathsf{Progress} \to \mathsf{Steps} \ \mathsf{a} \to \mathsf{Steps} \ \mathsf{a} \\ \mathsf{Apply} & :: \forall \mathsf{a} \ \mathsf{b}.(\mathsf{b} \to \mathsf{a}) \ \to \mathsf{Steps} \ \mathsf{b} \to \mathsf{Steps} \ \mathsf{a} \\ \mathsf{Fail} & :: \dots \end{array}$$

The Progress field describes how much progress we made in the input (i.e. how much of the input was consumed by this step)

# **Computing a result**

We now adapt our code so we compute a result on the fly, and change the parser type into:

$$\begin{tabular}{ll} \textbf{newtype} \ \mathsf{HP} \ \mathsf{s} \ \mathsf{a} \\ &= & \mathsf{HP} \ (\forall \mathsf{r}. (\mathsf{a} \to \mathsf{st} \to \mathsf{Steps} \ \mathsf{r}) \to \mathsf{st} \to \mathsf{Steps} \ \mathsf{r}) \\ \end{tabular}$$

# **Computing a result**

We now adapt our code so we compute a result on the fly, and change the parser type into:

$$\label{eq:heat_problem} \begin{split} \text{newtype HP s a} & = & \text{HP } (\forall r. (a \rightarrow \mathsf{st} \rightarrow \mathsf{Steps} \ r) \rightarrow \mathsf{st} \rightarrow \mathsf{Steps} \ r) \end{split}$$

$$\xrightarrow{h} \xrightarrow{a}$$

$$p$$
Steps r

#### best

We adapt the function which compares two alternatives.

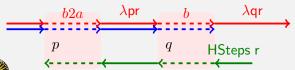
```
\begin{array}{lll} best'::Steps \ b \rightarrow Steps \ b \rightarrow Steps \ b \\ Fail \dots `best'` & \dots & = Fail \dots \\ Fail \dots `best'` & r & = r \\ I & `best'` & Fail \dots = I \\ Step \ n \ I `best'` Step \ m \ r \\ \mid n = m = Step \ n \ (I `best'` \ r) \\ \mid n < m = Step \ n \ (I `best'` \ Step \ (m - n) \ r) \\ \mid n > m = Step \ m \ (Step \ (n - m) \ I `best'` \ r) \end{array}
```

# History parsers are Functor and Applicative

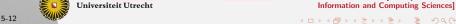
```
instance Functor (T st) where
   fmap f (HP ph) = HP (\lambda k \rightarrow ph (k \circ f))
instance Applicative (HP state) where
   HP ph \ll \sim (HP qh)
               = HP (\lambda k \rightarrow ph (\lambda pr \rightarrow qh (\lambda qr \rightarrow k (pr qr))))
   pure a = HP (\$a)
instance Alternative (T state) where
   HP ph \langle | \rangle HP gh = HP (\lambdak inp \rightarrow ph k inp 'best' gh k inp)
                           = HP (\lambda k \text{ inp} \rightarrow noAlts)
   empty
```

### History parsers are Functor and Applicative

```
instance Functor (T st) where
   fmap f (HP ph) = HP (\lambda k \rightarrow ph (k \circ f))
instance Applicative (HP state) where
   HP ph \ll \sim (HP qh)
               = HP (\lambda k \rightarrow ph (\lambda pr \rightarrow qh (\lambda qr \rightarrow k (pr qr))))
   pure a = HP (\$a)
instance Alternative (T state) where
   HP ph \langle | \rangle HP gh = HP (\lambdak inp \rightarrow ph k inp 'best' gh k inp)
                           = HP (\lambda k \text{ inp} \rightarrow noAlts)
   empty
```

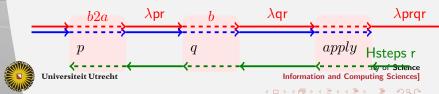


[Faculty of Science Information and Computing Sciences]



# History parsers are Functor and Applicative

```
instance Functor (T st) where
   fmap f (HP ph) = HP (\lambda k \rightarrow ph (k \circ f))
instance Applicative (HP state) where
   HP ph \ll \sim (HP qh)
               = HP (\lambda k \rightarrow ph (\lambda pr \rightarrow qh (\lambda qr \rightarrow k (pr qr))))
   pure a = HP (\$a)
instance Alternative (T state) where
   HP ph \langle | \rangle HP gh = HP (\lambdak inp \rightarrow ph k inp 'best' gh k inp)
                           = HP (\lambda k \text{ inp} \rightarrow noAlts)
   empty
```



### History parsers are IsParser

```
pSym \ \mathsf{a} = \mathsf{HP} \ (
    \lambda k \text{ inp } \rightarrow \text{case inp of}
                    (s:ss) \rightarrow if s == a then Step (k s ss)
                                    else Fail...
                    [] \rightarrow Fail
parse (HP p) inp =
     = fst \circ eval \circ p (\lambdaa rest \rightarrow if null rest
                                                then Apply (\lambda f \rightarrow (a, f)) Fail
                                                else . . . )
```

1. a recognised symbol inserts its value into the history



### History parsers are IsParser

```
pSym \ \mathsf{a} = \mathsf{HP} \ (
    \lambda k \text{ inp } \rightarrow \text{case inp of}
                    (s:ss) \rightarrow if s == a then Step (k s ss)
                                    else Fail...
                    [] \rightarrow Fail
parse (HP p) inp =
     = fst \circ eval \circ p (\lambdaa rest \rightarrow if null rest
                                                then Apply (\lambda f \rightarrow (a, f)) Fail
                                                else . . . )
```

- 1. a recognised symbol inserts its value into the history
- 2. at the end we place the computed result in the push a Sciences Universiteit Utrecht Information and Computing Sciences



### helper functions

```
eval (Step steps) = eval steps

eval (Apply f steps) = f (eval steps)
```

4日▶4畳▶4畳▶4畳▶ 畳 夕久@