Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

# Haskell, Do You Read Me?

## Constructing and Composing Efficient Top-down Parsers at Runtime

Marcos Viera  Doaitse Swierstra  Eelco Lempsink

Instituto de Computación, Universidad de la República, Uruguay
Dept. of Information and Computing Sciences, Utrecht University

September 25, 2008

# Symptoms

$$
\begin{aligned}
&\textbf{data}\ T1 = T1 :<: T1 \\
&\qquad\qquad \mid\ T1 :>: T1 \\
&\qquad\qquad \mid\ C \\
&\qquad\quad \textbf{deriving}\ (Read, Show) \\
&\textbf{infixl}\ 5\ :<: \\
&\textbf{infixr}\ 6\ :>: \\
&x \qquad\quad :: T1 \\
&x \ = C :<: C :<: C
\end{aligned}
$$

```
*Main> x
(C :<: C) :<: C
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

# Symptoms

$$
\begin{aligned}
&\textbf{data}\ T1 = T1 :<: T1 \\
&\qquad\qquad \mid\ T1 :>: T1 \\
&\qquad\qquad \mid\ C \\
&\qquad\quad \textbf{deriving}\ (Read, Show) \\
&\textbf{infixl}\ 5\ :<: \\
&\textbf{infixr}\ 6\ :>: \\
&x, x' \quad :: T1 \\
&x \ = C :<: C :<: C \\
&x' = (read \circ show)\ \$\ C :<: C :<: C
\end{aligned}
$$

```
*Main> x'
(C :<: C) :<: C
```

# Symptoms

$$
\begin{aligned}
&\textbf{data}\ T1 = T1 :<: T1 \\
&\qquad\qquad \mid\ T1 :>: T1 \\
&\qquad\qquad \mid\ C \\
&\qquad\quad \textbf{deriving}\ (Read, Show) \\
&\textbf{infixl}\ 5\ :<: \\
&\textbf{infixr}\ 6\ :>: \\
&x, x', x'' :: T1 \\
&x\ \ = C :<: C :<: C \\
&x' = (read \circ show)\ \$\ C :<: C :<: C \\
&x'' = read\ \texttt{"C :<: C :<: C"}
\end{aligned}
$$

```
*Main> x''
*** Exception: Prelude.read: no parse
```

# Symptoms

```
data T1 = T1 :<: T1
        |  T1 :>: T1
        |  C
        deriving (Read, Show)
infixl 5 :<:
infixr 6 :>:
x, x', x'' :: T1
x   = C :<: C :<: C
x'  = (read . show) $ C :<: C :<: C
x'' = read "C :<: C :<: C"
```

Ideally, you should be able to *read* every valid constant expression.

# Parentheses

```
*Main> time (read "C" :: T1)
C
CPU Time: 0 ms
```

# Parentheses

```
*Main> time (read "C" :: T1)
C
CPU Time: 0 ms

*Main> time (read "(((((C)))))" :: T1)
C
CPU Time: 74 ms
```

# Parentheses

```
*Main> time (read "C" :: T1)
C
CPU Time: 0 ms

*Main> time (read "(((((C)))))" :: T1)
C
CPU Time: 74 ms

*Main> time (read "((((((C))))))" :: T1)
C
CPU Time: 389 ms
```

# Parentheses

```
*Main> time (read "C" :: T1)
C
CPU Time: 0 ms

*Main> time (read "(((((C)))))" :: T1)
C
CPU Time: 74 ms

*Main> time (read "((((((C))))))" :: T1)
C
CPU Time: 389 ms

*Main> time (read "(((((((C)))))))" :: T1)
C
CPU Time: 1753 ms
```

# Breadth-first Parsing

The language which is actually recognised by the generated *read* function is described by the non left-recursive grammar:

| | |
|---|---|
| $T1\ (n) \rightarrow\ T1\ (6)$ ":<:" $T1\ (6)$ | $(n \leqslant 5)$ |
| $\mid\ T1\ (7)$ ":>:" $T1\ (7)$ | $(n \leqslant 6)$ |
| $\mid$ "C" | $(n \leqslant 10)$ |
| $\mid$ "(" $T1\ (0)$ ")" | $(n \leqslant 10)$ |

# Breadth-first Parsing

The language which is actually recognised by the generated $read$ function is described by the non left-recursive grammar:

| | |
|---|---|
| $T1\ (n) \rightarrow\ T1\ (6)$ ":<:" $T1\ (6)$ | $(n \leqslant 5)$ |
| $\mid\ T1\ (7)$ ":>:" $T1\ (7)$ | $(n \leqslant 6)$ |
| $\mid$ "C" | $(n \leqslant 10)$ |
| $\mid$ "(" $T1\ (0)$ ")" | $(n \leqslant 10)$ |

Three parallel parsers are started up for the first '(', and so on recursively.

# Common Left-factors

Unfortunately the problem also shows up for more reasonable expressions such as $C$ :>: ($C$ :>: ($C$ :>: ...)).

We remove the conditions, and encode them in the non-terminals.

$$
\begin{array}{lll}
T1\ (0..5) & \rightarrow & T1\ (6)\ \texttt{":<:"}\ T1\ (6) \mid T1\ (6) \\
T1\ (6) & \rightarrow & T1\ (7)\ \texttt{":>:"}\ T1\ (7) \mid T1\ (7) \\
T1\ (7..10) & \rightarrow & \texttt{"C"} \\
 & & \mid\ \texttt{"("}\ T1\ (0)\ \texttt{")"}
\end{array}
$$

# Common Left-factors

Unfortunately the problem also shows up for more reasonable expressions such as $C$ :>: ($C$ :>: ($C$ :>: ...)).

We remove the conditions, and encode them in the non-terminals.

$$
\begin{array}{lll}
T1\ (0..5) & \rightarrow & T1\ (6)\ \texttt{":<:"}\ T1\ (6) \mid T1\ (6) \\
T1\ (6) & \rightarrow & T1\ (7)\ \texttt{":>:"}\ T1\ (7) \mid T1\ (7) \\
T1\ (7..10) & \rightarrow & \texttt{"C"} \\
 & & \mid\ \texttt{"("}\ T1\ (0)\ \texttt{")"}
\end{array}
$$

We see that some alternatives start with the same non-terminal symbol.

# The Problem

- Derived $read$ functions treat all operators as being *non-associative*, despite their declared associativities and precedences.

# The Problem

- Derived $read$ functions treat all operators as being *non-associative*, despite their declared associativities and precedences.
- Derived $show$ functions generate the needed extra parentheses, in order to make $read \circ show$ work.

# The Problem

- Derived $read$ functions treat all operators as being *non-associative*, despite their declared associativities and precedences.
- Derived $show$ functions generate the needed extra parentheses, in order to make $read \circ show$ work.
- These extra parentheses make parsing take exponential time.

# The Problem

- Derived $read$ functions treat all operators as being *non-associative*, despite their declared associativities and precedences.
- Derived $show$ functions generate the needed extra parentheses, in order to make $read \circ show$ work.
- These extra parentheses make parsing take exponential time.
- Common left-factors have a similar effect.

# How Does the Problem Arise?

```
infix 5 :+:
infix 6 :*:

data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
          deriving Read
t2 :: T2 (T2 Int)
t2 = read "(3 :*: C2) :*: C2"
```

The function *read* is a member of the class *Read*:

# How Does the Problem Arise?

```
infix 5 :+:
infix 6 :*:

data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
          deriving Read
t2 :: T2 (T2 Int)
t2 = read "(3 :*: C2) :*: C2"
```

The function *read* is a member of the class *Read*:

- *read* functions are elements in dictionaries

# How Does the Problem Arise?

```
infix 5 :+:
infix 6 :*:

data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
          deriving Read
t2 :: T2 (T2 Int)
t2 = read "(3 :*: C2) :*: C2"
```

The function *read* is a member of the class *Read*:

- *read* functions are elements in dictionaries
- **instance**-declarations compose new dictionaries out of existing dictionaries at run-time

# How Does the Problem Arise?

```
infix 5 :+:
infix 6 :*:

data T2 a = T2 a :+: T2 a
          | a    :*: T2 a
          | C2
          deriving Read
t2 :: T2 (T2 Int)
t2 = read "(3 :*: C2) :*: C2"
```

The function *read* is a member of the class *Read*:

- *read* functions are elements in dictionaries
- **instance**-declarations compose new dictionaries out of existing dictionaries at run-time
- hence *read* functions are to be composed at run-time

# The Bad News

- Bottom-up parsers do not compose at all, and all perform an analysis of the complete grammar (YACC, Happy)
- Top-down parsers do not compose efficiently for arbitrary grammars, and may lead to left-recursive parsers if no care is taken:

$$\begin{array}{l} \textbf{data}\ T1\ a = a \quad\quad\ \ :*: Int\ \textbf{deriving}\ Read \\ \textbf{data}\ T2 \quad = T1\ T2 :+: Int\ \textbf{deriving}\ Read \end{array}$$

# The Bad and the Good News

- Bottom-up parsers do not compose at all, and all perform an analysis of the complete grammar (YACC, Happy)
- Top-down parsers do not compose efficiently for arbitrary grammars, and may lead to left-recursive parsers if no care is taken:

$$\begin{array}{l} \textbf{data}\ T1\ a = a \quad\quad\ \ : * : Int\ \textbf{deriving}\ Read \\ \textbf{data}\ T2 \quad = T1\ T2 : + : Int\ \textbf{deriving}\ Read \end{array}$$

- Grammars can be composed!

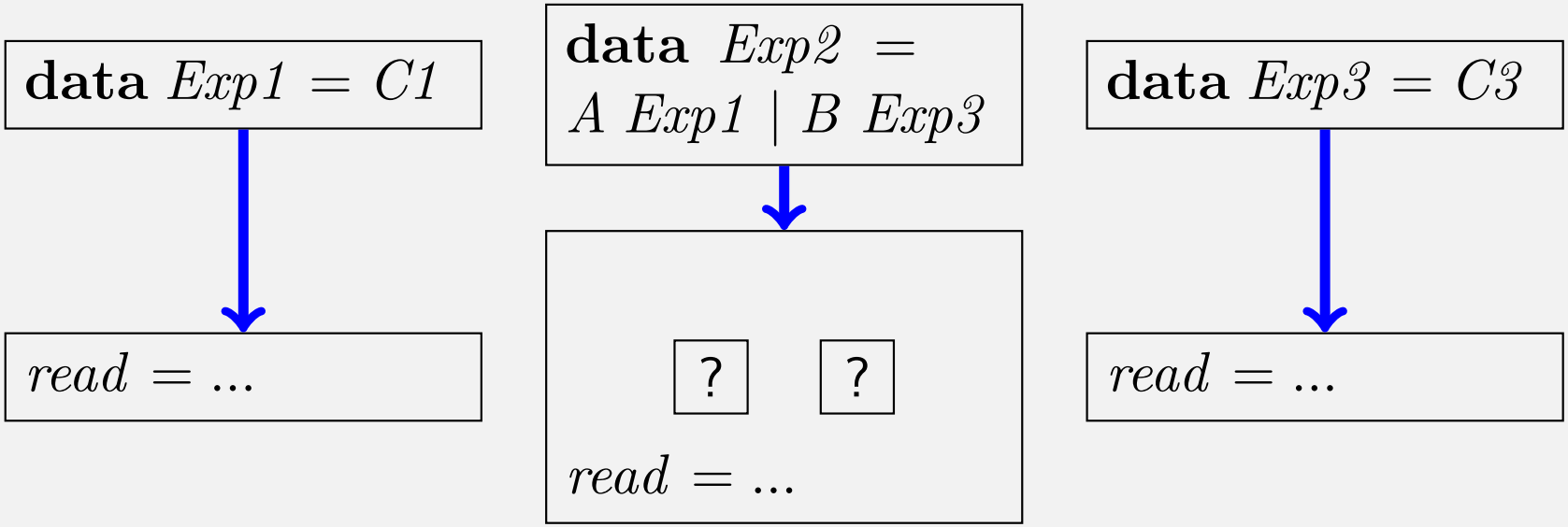
# Using Grammars instead of Parsers

$$\boxed{\textbf{data}\ Exp1 = C1}$$

$$\boxed{\begin{array}{l}\textbf{data}\ Exp2 = \\ A\ Exp1 \mid B\ Exp3\end{array}}$$

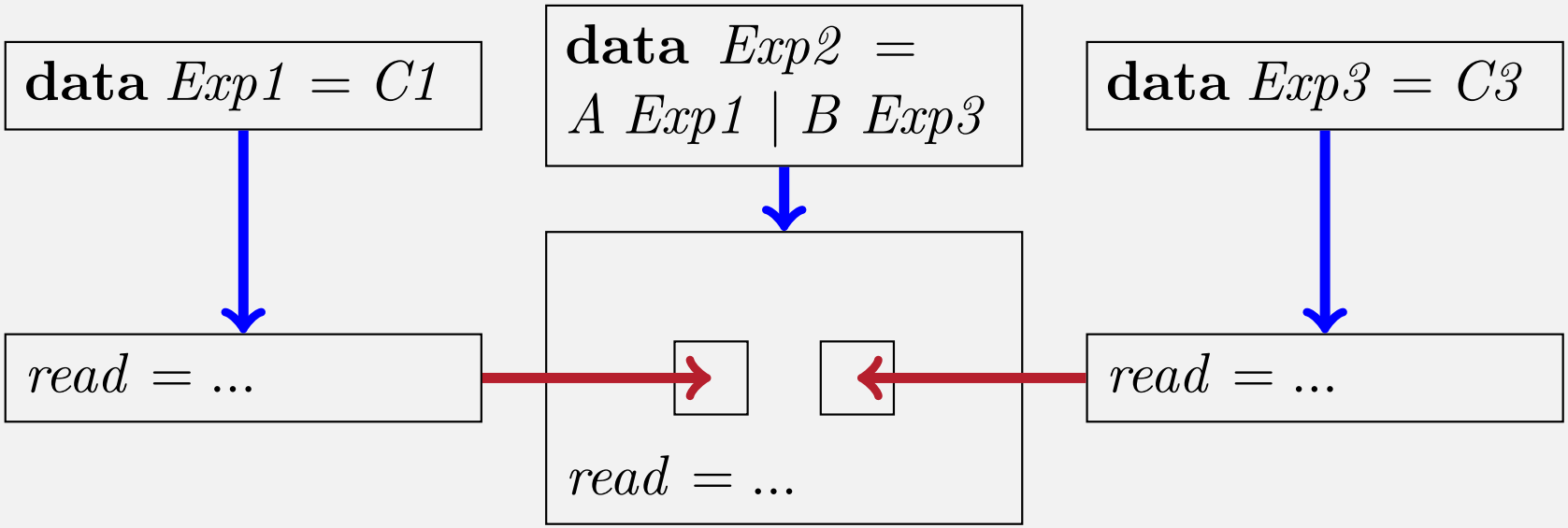
$$\boxed{\textbf{data}\ Exp3 = C3}$$

# Using Grammars instead of Parsers



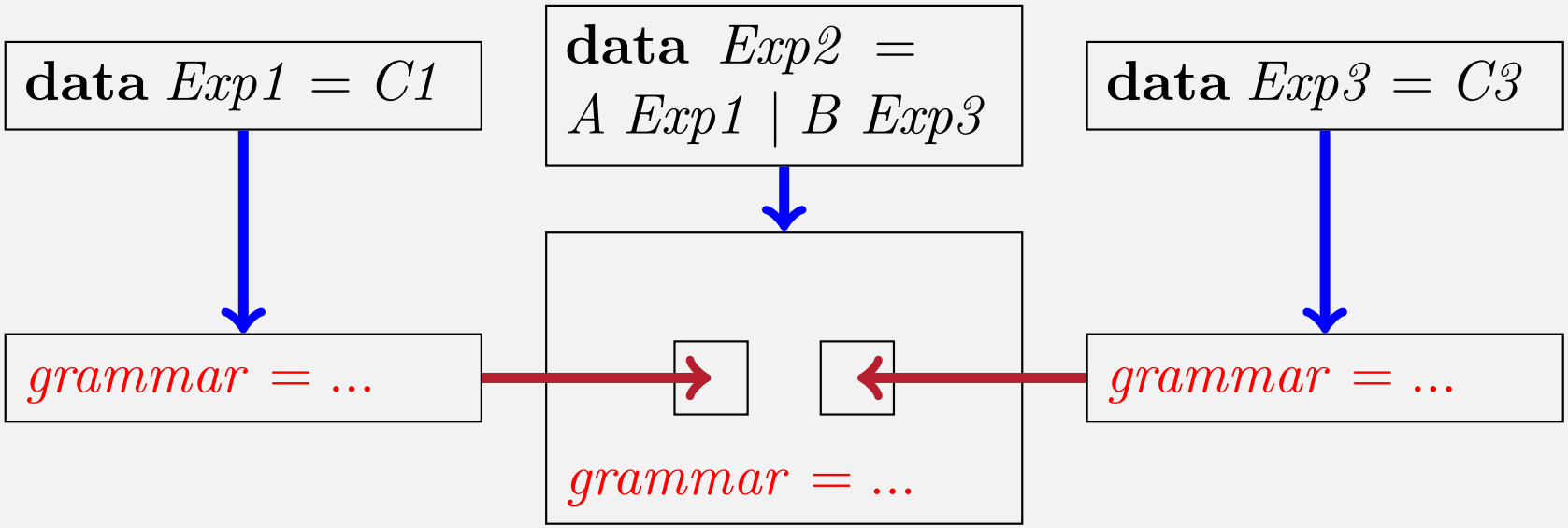

derive

# Using Grammars instead of Parsers



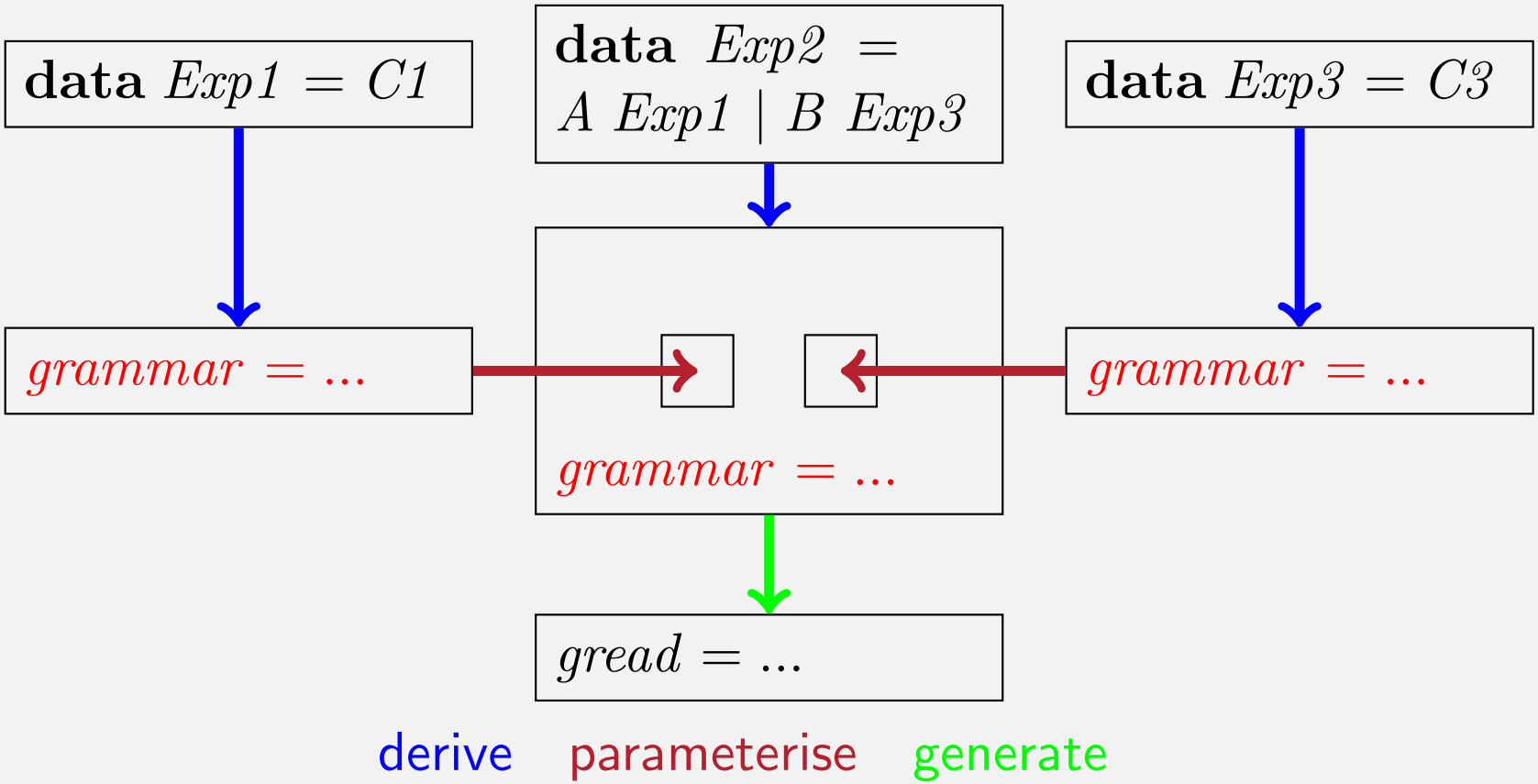

derive parameterise

# Using Grammars instead of Parsers




derive parameterise

# Using Grammars instead of Parsers



# The Class *Gram*

Instead of the class *Read* we introduce:

```
class Gram a where
  grammar :: DGrammar a
```

where *DGrammar* is a data type describing grammatical structures, including information about precedences.

# The Class $Gram$

Instead of the class $Read$ we introduce:

```
class Gram a where
  grammar :: DGrammar a
```

where $DGrammar$ is a data type describing grammatical structures, including information about precedences.

Note that it is labelled with a type $a$, which is the data type described by a value of type $DGrammar\ a$.

# The Class *Gram*

Instead of the class *Read* we introduce:

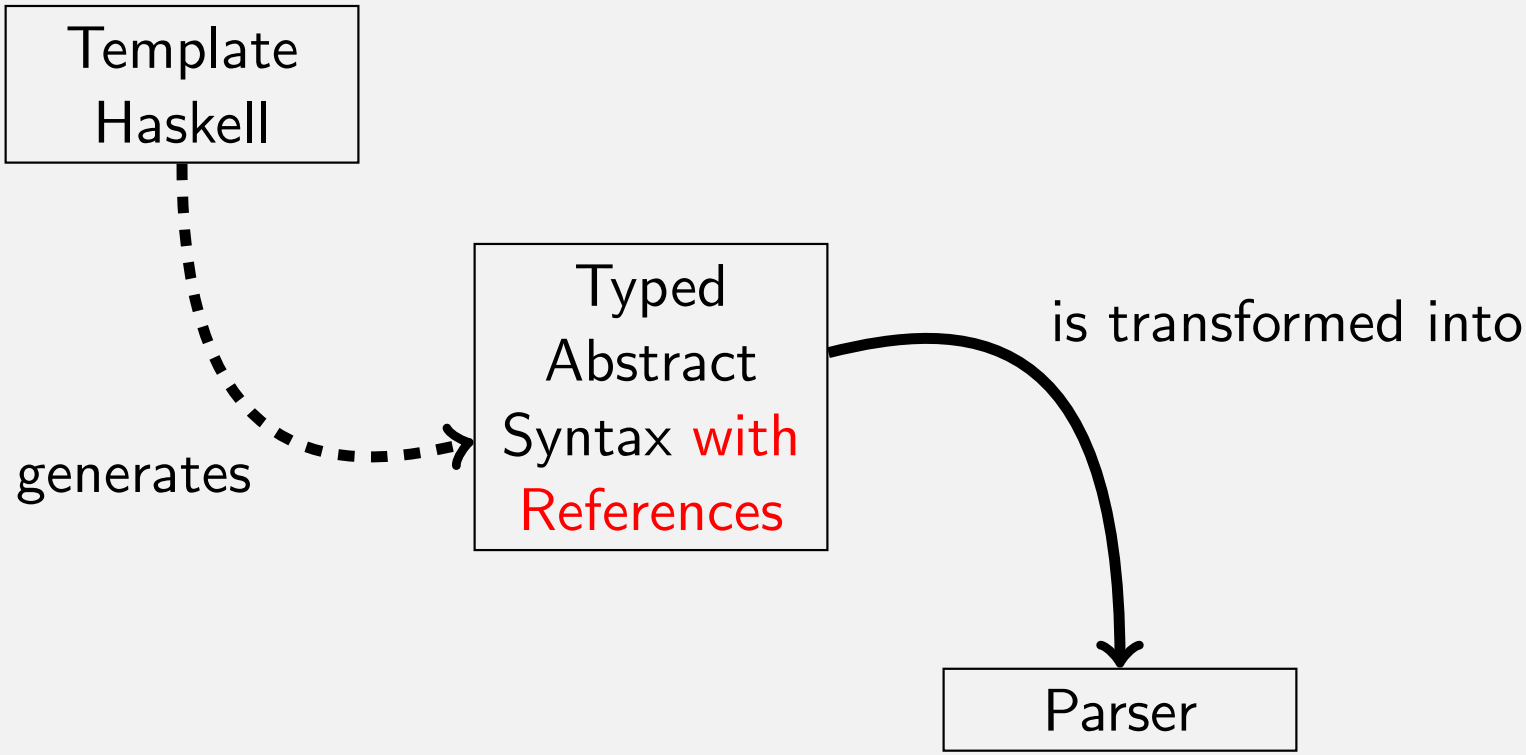
```
class Gram a where
  grammar :: DGrammar a
```

where *DGrammar* is a data type describing grammatical structures, including information about precedences.

Note that it is labelled with a type $a$, which is the data type described by a value of type $DGrammar\ a$.
Now we can, just as for *read* define:

```
read  :: Read a  => String -> a
gread :: Gram a => String -> a
```

# Generating parsers from Data Types



# The Steps to be Taken

group Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.

# The Steps to be Taken

**group** Combine pieces of grammar together, introduce extra non-terminals to represent the precedences.

**leftcorner** Remove possible left-recursion by applying the *Left-Corner Transform*

# The Steps to be Taken

| | |
|---|---|
| group | Combine pieces of grammar together, introduce extra non-terminals to represent the precedences. |
| leftcorner | Remove possible left-recursion by applying the *Left-Corner Transform* |
| leftfactoring | Combine common prefixes of alternatives |

# The Steps to be Taken

| | |
|---|---|
| group | Combine pieces of grammar together, introduce extra non-terminals to represent the precedences. |
| leftcorner | Remove possible left-recursion by applying the *Left-Corner Transform* |
| leftfactoring | Combine common prefixes of alternatives |
| compile | Map the resulting $Grammar$ onto a parser |

# The Steps to be Taken

| | |
|---|---|
| group | Combine pieces of grammar together, introduce extra non-terminals to represent the precedences. |
| leftcorner | Remove possible left-recursion by applying the *Left-Corner Transform* |
| leftfactoring | Combine common prefixes of alternatives |
| compile | Map the resulting $Grammar$ onto a parser |
| parse | Use the parser to read a value |

# The Steps to be Taken

| | |
|---|---|
| group | Combine pieces of grammar together, introduce extra non-terminals to represent the precedences. |
| leftcorner | Remove possible left-recursion by applying the *Left-Corner Transform* |
| leftfactoring | Combine common prefixes of alternatives |
| compile | Map the resulting $Grammar$ onto a parser |
| parse | Use the parser to read a value |

# The Steps to be Taken

| | |
|---|---|
| group | Combine pieces of grammar together, introduce extra non-terminals to represent the precedences. |
| leftcorner | Remove possible left-recursion by applying the *Left-Corner Transform* |
| leftfactoring | Combine common prefixes of alternatives |
| compile | Map the resulting $Grammar$ onto a parser |
| parse | Use the parser to read a value |

$$
\begin{aligned}
&gread :: Gram\ a \Rightarrow String \rightarrow a \\
&gread = (parse \circ compile \quad \circ leftfactoring \\
&\qquad\qquad \circ leftcorner \circ group)\ grammar
\end{aligned}
$$

# Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals

# Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals
- We want to be able to inspect and transform the grammar

# Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals
- We want to be able to inspect and transform the grammar
- We want to inspect the underlying graph structure

# Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals
- We want to be able to inspect and transform the grammar
- We want to inspect the underlying graph structure
- Of which the nodes are labelled with different types

# Types Abstract Syntax with Explicit References

- Right hand sides of productions contain references to non-terminals
- We want to be able to inspect and transform the grammar
- We want to inspect the underlying graph structure
- Of which the nodes are labelled with different types
- So we use heterogeneous collections, i.e. we use nested cartesian products, henceforth called $Env$-ironments

# References and Environments I

We introduce natural numbers, labelled with a type $a$ describing what is referred to, and a list of types $env$ describing the structure in which this $a$ labelled object lives:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Ref}\ a\ \mathit{env}\ \textbf{where}\\
&\quad \mathit{Zero} :: \qquad\qquad\quad \mathit{Ref}\ a\ (a, \mathit{env})\\
&\quad \mathit{Suc}\ \ :: \mathit{Ref}\ a\ \mathit{env}' \to \mathit{Ref}\ a\ (x, \mathit{env}')\\
&\textbf{data}\ \mathit{Equal}\ a\ b\ \textbf{where}\\
&\quad \mathit{Eq} :: \mathit{Equal}\ a\ a\\
&\mathit{match} :: \mathit{Ref}\ a\ \mathit{env} \to \mathit{Ref}\ b\ \mathit{env} \to \mathit{Maybe}\ (\mathit{Equal}\ a\ b)\\
&\mathit{match}\ \mathit{Zero}\ \ \ \mathit{Zero}\ \ \ \ = \mathit{Just}\ \mathit{Eq}\\
&\mathit{match}\ (\mathit{Suc}\ x)\ (\mathit{Suc}\ y) = \mathit{match}\ x\ y\\
&\mathit{match}\ \_\ \ \ \_\ \ \ \ = \mathit{Nothing}
\end{aligned}
$$

# References and Environments II

$$
\begin{array}{l}
\textbf{data}\ \mathit{Env}\ t\ \mathit{use}\ \mathit{def}\ \textbf{where} \\
\quad \mathit{Empty} :: \mathit{Env}\ t\ \mathit{use}\ () \\
\quad \mathit{Cons} \quad :: t\ a\ \mathit{use} \to \mathit{Env}\ t\ \mathit{use} \qquad \mathit{def}' \\
\qquad\qquad\qquad\qquad \to \mathit{Env}\ t\ \mathit{use}\ (a, \mathit{def}')
\end{array}
$$

$t\ a\ use$ :: a term of type $t$, describing a value of type $a$ contains references pointing into an environment labelled by $use$. The parameter $def$ describes the values actually existing in the $Env$. If $use$ equals $def$ the environment is closed.

# *DGrammar*

$$
\begin{aligned}
&\textbf{data}\ DGrammar\ a \\
&\quad = \forall env.DGrammar\ (Ref\ a\ env) \\
&\qquad\qquad\qquad\qquad (Env\ DGram\ env\ env) \\
&\textbf{data}\ DGram\ a\ env = DGD\ (DLNontDefs\ a\ env) \\
&\qquad\qquad\qquad\qquad\quad \mid\ DGG\ (DGrammar\ a)
\end{aligned}
$$

$$
\textbf{newtype}\ DRef\ a\ env = DRef\ (Ref\ a\ env, Int)
$$

$$
\begin{aligned}
&\textbf{newtype}\ DLNontDefs\ a\ env \\
&\quad = DLNontDefs\ [(DRef\ a\ env, DProductions\ a\ env)]
\end{aligned}
$$

# Continued ..

```
newtype DProductions a env
    = DPS {unDPS :: [DProd a env]}
```

```
data DProd a env where
   DSeq  :: DSymbol b env → DProd (b → a) env
                          → DProd a env
   DEnd :: a              → DProd a env
```

```
data DSymbol a env where
   DNont :: DRef a env → DSymbol a env
   DTerm :: Token      → DSymbol Token env
```

# Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[
                    _Exp                    ":+:"
                    _Exp

,
                    "C"



]
```

# Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[
                    _Exp                    ":+:"
                    _Exp

,
                    "C"
          ,         "("             _Exp
                    ")"

]
```

# Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[ dNont (_Exp   ) .#. dTerm ":+:" .#.
  dNont (_Exp   )

, dTerm "C"
, dTerm "(" .#. dNont (_Exp   ) .#.
  dTerm ")"

]
```

# Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:

[
    DPS [dNont (_Exp, 6) .#. dTerm ":+:" .#.
         dNont (_Exp, 7)                    ]

,
    DPS [dTerm "C"
        ,dTerm "(" .#. dNont (_Exp, 0) .#.
         dTerm ")"                      ]

]
```

# Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:
```

```
[   (DRef (_Exp, 6)
    , DPS [dNont (_Exp, 6) .#. dTerm ":+:" .#.
           dNont (_Exp, 7)                    ]
    )
,   (DRef (_Exp, 10)
    , DPS [dTerm "C"
          , dTerm "(" .#. dNont (_Exp, 0) .#.
            dTerm ")"                       ]
    )
]
```

# Typed Abstract Syntax

```
data Exp = Exp :+: Exp
         | C
infixl 6 :+:

[   (DRef (_Exp, 6)
    , DPS [dNont (_Exp, 6) .#. dTerm ":+:" .#.
           dNont (_Exp, 7) .#. dEnd plus]
    )
,   (DRef (_Exp, 10)
    , DPS [dTerm "C" .#. dEnd (const C)
          , dTerm "(" .#. dNont (_Exp, 0) .#.
            dTerm ")" .#. dEnd parenT]
    )
]
plus e1 _ e2 = e2 :+: e1
```

# Typed Abstract Syntax

```
instance Gram Exp where
  grammar = DGrammar _0 envExp

envExp :: Env DGram (Exp, ()) (Exp, ())
envExp = consD (nonts _0) Empty
  where
    nonts _Exp = DLNontDefs
      [   (DRef (_Exp, 6)
          , DPS [dNont (_Exp, 6) .#. dTerm ":+:" .#.
                 dNont (_Exp, 7) .#. dEnd plus]
          )
      ,   (DRef (_Exp, 10)
          , DPS [dTerm "C" .#. dEnd (const C)
                 , dTerm "(" .#. dNont (_Exp, 0) .#.
                 dTerm ")" .#. dEnd parenT]
          )
      ]
    plus e1 _ e2 = e2 :+: e1
```

# An Intermediate result

$$
\begin{array}{lll}
A & \rightarrow & \texttt{"C1"}\ A\_C1 \mid \texttt{"("}\ A\_\ ( \\
A\_A & \rightarrow & \texttt{":<:"}\ B\ A\_A \mid \texttt{":<:"}\ B \\
A\_B & \rightarrow & A\_A \mid \epsilon \\
A\_C & \rightarrow & \texttt{":>:"}\ B\ A\_B \mid A\_B \\
A\_C1 & \rightarrow & A\_C \\
A\_\ ( & \rightarrow & A\ \texttt{")"}\ A\_C \\
B & \rightarrow & \texttt{"C1"}\ B\_C1 \mid \texttt{"("}\ B\_\ ( \\
B\_C & \rightarrow & \texttt{":>:"}\ B \mid \epsilon \\
B\_C1 & \rightarrow & B\_C \\
B\_\ ( & \rightarrow & A\ \texttt{")"}\ B\_C \\
C & \rightarrow & \texttt{"C1"}\ C\_C1 \mid \texttt{"("}\ C\_\ ( \\
C\_C1 & \rightarrow & \epsilon \\
C\_\ ( & \rightarrow & A\ \texttt{")"}
\end{array}
$$

1. We have introduced new non-terminals
2. Old non-terminals have new productions

# The Transformations

All the transformations can be expressed in terms of an arrow-like type:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Trafo}\ m\ t\ a\ b = \\
&\quad \mathit{Trafo}\ (\forall \mathit{env1}.m\ \mathit{env1} \rightarrow \\
&\qquad\quad \exists \mathit{env2}. \\
&\qquad\quad (m\ \mathit{env2} \\
&\qquad\quad ,\forall s.\quad a\ s \rightarrow T\ \mathit{env2}\ s \rightarrow \mathit{Env}\ t\ s\ \mathit{env1} \rightarrow \\
&\qquad\qquad\quad (b\ s,\quad T\ \mathit{env1}\ s\ ,\ \mathit{Env}\ t\ s\ \mathit{env2}) \\
&\qquad\quad )
\end{aligned}
$$

# Results I

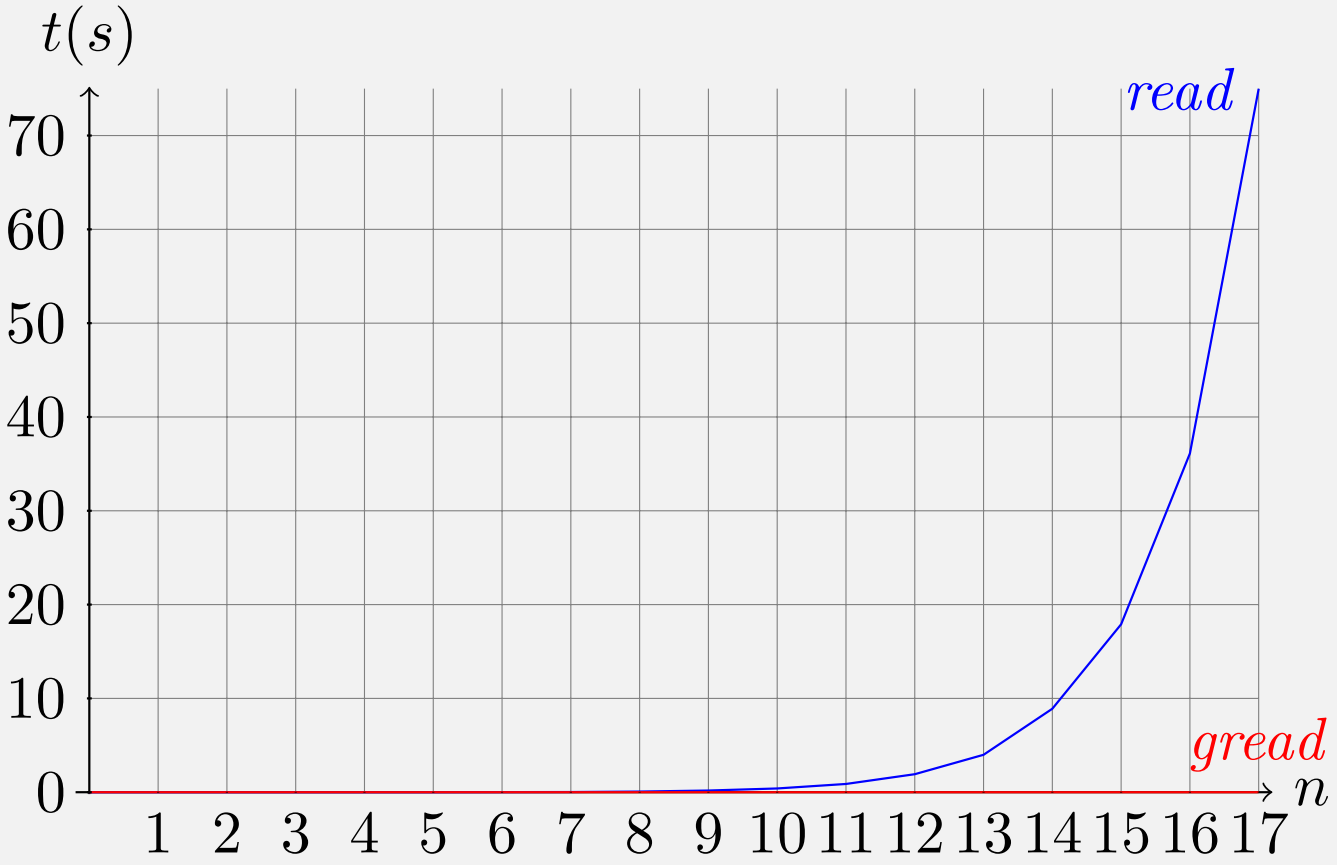
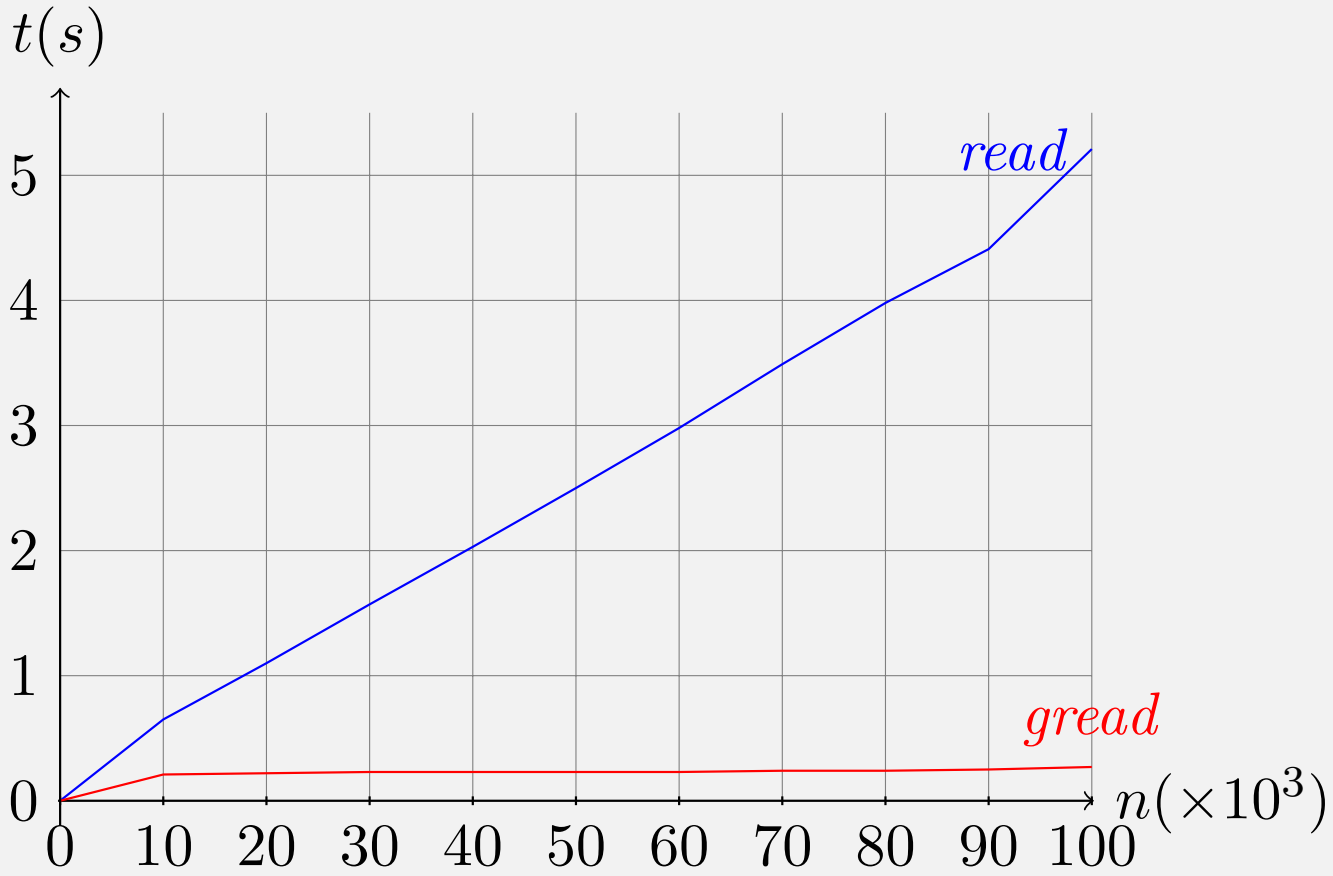



Figure: Execution times of reading $C :> (C :>: ...)$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

# Reading a Large Data Type




Overhead is very small, and that thanks to the use of the UU-parsers also parse times do hardly increase.

# Why is this so complicated ...

1. The problem is complicated

# Why is this so complicated ...

1. The problem is complicated
2. We do in  350 lines more than Bison (10.000 lines) is doing

# Why is this so complicated ...

1. The problem is complicated
2. We do in  350 lines more than Bison (10.000 lines) is doing
3. Extra constructors are needed because we need existentials

# Why is this so complicated ...

1. The problem is complicated
2. We do in  350 lines more than Bison (10.000 lines) is doing
3. Extra constructors are needed because we need existentials
4. If we have lazy evaluation, we also want it at the type level!

# Why is this so complicated ...

1. The problem is complicated
2. We do in  350 lines more than Bison (10.000 lines) is doing
3. Extra constructors are needed because we need existentials
4. If we have lazy evaluation, we also want it at the type level!

$$
\begin{array}{l}
f :: \forall a.(a \rightarrow \exists b\ (b, a, b \rightarrow b \rightarrow Int)) \\
\textbf{let}\ (b, a, g) = f\ b \\
\textbf{in}\ \ g\ b\ a
\end{array}
$$

# To Take Home

- The transformation library has been used unmodified for all the transformations
- The library can be used for any collection of abstract syntax trees, which contain references to each other, and of which the structure has to be inspected