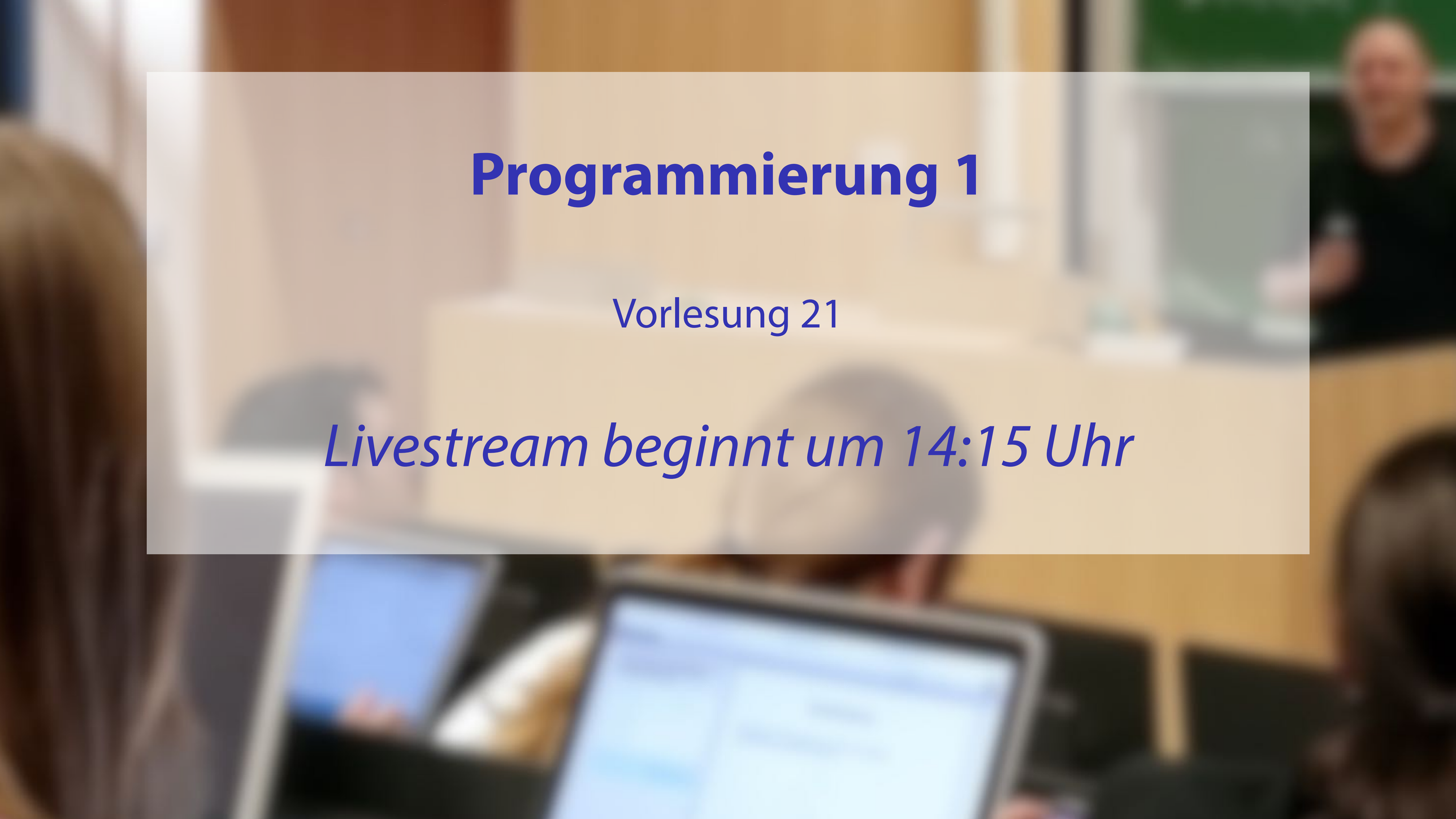


Programmierung 1

Vorlesung 21

Livestream beginnt um 14:15 Uhr

Konkrete Syntax
Teil 2

Programmierung 1

Hauptklausur

- **Haupttermin: 13. Februar 2021**
- 90 Minuten
- Klausurrelevanter Stoff bis einschließlich **Übungsblatt K**

- Um zur Hauptklausur zugelassen zu werden, müssen Sie die Mittelklausur (oder den Nachtermin der Mittelklausur) bestanden haben, sowie 50% der insgesamt erreichbaren Punkte aus den Minitests erreicht haben.

Zusatztutorien

1. 01.02. 16:00 **Konstruktortypen**
2. 02.02. 16:00 **Bäume**
3. 04.02. 16:00 **Mengenlehre + Mathematische Prozeduren**
4. 05.02. 16:00 **Induktion**

— vorlesungsfreie Zeit —

5. 08.02. 10:00 **Laufzeit**
6. 08.02. 14:00 **Statische und Dynamische Semantik**
7. 09.02. 10.00 **Konkrete Syntax**

bitte im CMS (unverbindlich) anmelden

Nachklausurtutorium

- **voraussichtlich 01.03. — 10.03.2021**

- intensive Vorbereitung auf die Nachklausur

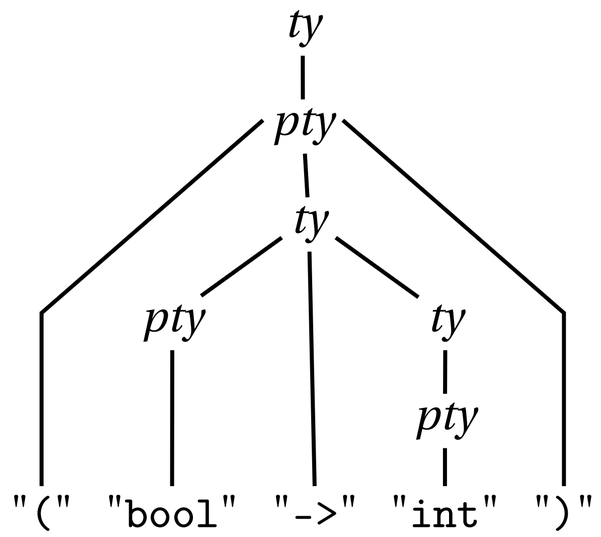
- wird von Tutor:innen *aus dem aktuellen Semester* gehalten

Ein Lexer für Typen von F

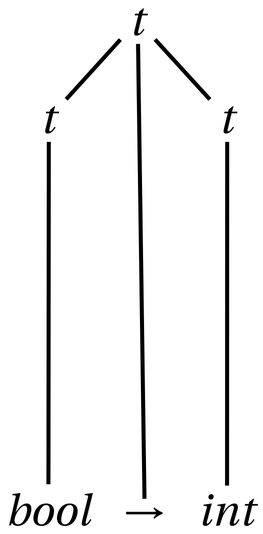
```
exception Error of string

datatype token = BOOL | INT | ARROW | LPAR | RPAR

fun lex nil = nil
  | lex (#" ":: cr) = lex cr
  | lex (#"\t":: cr) = lex cr
  | lex (#"\n":: cr) = lex cr
  | lex (#"b":: #"o":: #"o":: #"l":: cr) = BOOL:: lex cr
  | lex (#"i":: #"n":: #"t":: cr) = INT:: lex cr
  | lex (#"-":: #">":: cr) = ARROW:: lex cr
  | lex (#"(":: cr) = LPAR:: lex cr
  | lex (#")":: cr) = RPAR:: lex cr
  | lex _ = raise Error "lex"
```

val lex : char list → token list

Konkrete und abstrakte Ableitungen



konkrete Ableitung



abstrakte Ableitung

$$ty ::= pty \mid pty \texttt{"->"} ty$$
$$pty ::= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"}$$

$$t \in Ty = bool \mid int \mid t \to t$$

Parsing

Seien eine **konkrete** Grammatik und eine Kategorie *A* gegeben.

- Ein **Prüfer für *A*** ist eine Prozedur, die für eine Wortfolge **entscheidet,** ob es sich um einen **Satz** gemäß *A* handelt.
- Ein **Parser für *A*** ist ein Prüfer für *A*, der, falls es sich um einen Satz gemäß *A* handelt, eine **Baumdarstellung** gemäß einer **abstrakten Syntax** für die dargestellte Phrase liefert.

Idee: algorithmische Interpretation von Grammatiken durch **rekursiven Abstieg** (**RA**). (Engl.: **recursive descent**):
Wir gehen von links nach rechts durch die Wortfolge und bauen dabei die Baumdarstellung auf.

RA-Tauglichkeit

- Eine konkrete Grammatik heißt RA-tauglich, falls für die algorithmische Interpretation ihrer Gleichungen gilt:
  - falls es zu einer Rekursion kommt, wurde die Argument-liste **um mindestens ein Wort** verkürzt, und
  - falls zwischen mehreren Alternativen gewählt werden muss, kann die Wahl **immer aufgrund des ersten Wortes** der Argumentliste erfolgen.

Beispiele:

seq ::= “0” | “1” *seq* | “2” *seq seq*	RA-tauglich
a ::= “0” | *a* | “2” *a a*	nicht RA-tauglich
b ::= “0” | *b* “*1*” | *b b* “*2*”	nicht RA-tauglich

Prüfer für Typen

- **konkrete Grammatik**:

$$ty ::= pty \mid pty \texttt{"->"}\ ty$$

$$pty ::= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"}$$

Grammatik ist nicht RA-tauglich!

- **Abhilfe:** Formulierung mit **Optionalklammern [ ]**

$$ty ::= pty\ [\texttt{"->"}\ ty]$$

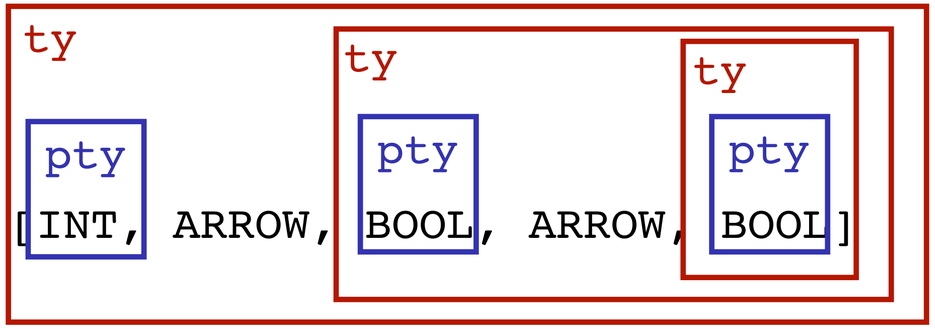
$$pty ::= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"}$$

Prüfe für *ty* zunächst gemäß *pty*.

Falls noch ein **Rest** bleibt der mit "**–>**" beginnt,
prüfe nochmals gemäß ty.
Falls nicht: fertig.

Prüfer für Typen

```
fun ty ts = case pty ts of ARROW::tr => ty tr | tr => tr
and pty (BOOL::tr) = tr
  | pty (INT::tr) = tr
  | pty (LPAR::tr) = (case ty tr of RPAR::tr => tr
                          | _ => raise Error "RPAR")
  | pty _ = raise Error "pty"
```

Parser für Typen

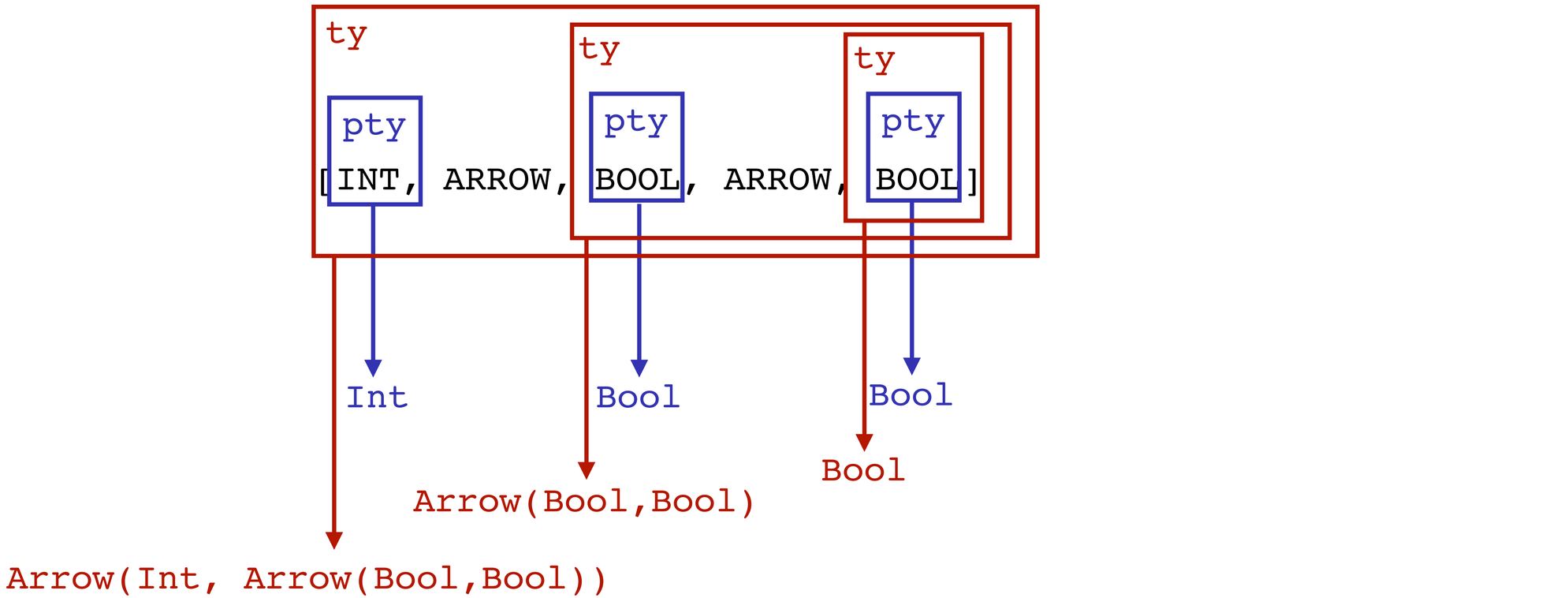
```
datatype ty = Bool | Int | Arrow of ty * ty

fun ty ts = (case pty ts of
                (t, ARROW::tr) => let val (t',tr') = ty tr
                                  in  (Arrow(t,t'), tr') end
              | s => s)
and pty (BOOL::tr) = (Bool,tr)
  | pty (INT::tr) = (Int,tr)
  | pty (LPAR::tr) = (case ty tr of
                        (t,RPAR::tr') => (t,tr')
                      | _ => raise Error "pty")
  | pty _ = raise Error "pty"
```

*val ty : token list → ty * token list*

*val pty : token list → ty * token list*

Beispiel

```
fun ty ts = (case pty ts of
                (t, ARROW::tr) => let val (t',tr') = ty tr
                                  in  (Arrow(t,t'), tr') end
              | s => s)
```



Optional: Hilfsprozeduren

```
datatype ty = Bool | Int | Arrow of ty * ty

fun ty ts = (case pty ts of
                 (t, ARROW::tr) => let val (t',tr') = ty tr
                                   in  (Arrow(t,t'), tr') end

               | s => s)
and pty (BOOL::tr) = (Bool,tr)
  | pty (INT::tr) = (Int,tr)
  | pty (LPAR::tr) = (case ty tr of
                          (t,RPAR::tr') => (t,tr')
                        | _ => raise Error "pty")
  | pty _ = raise Error "pty"
```

extend (t,tr) ty Arrow

match (ty tr) RPAR

- *extend (a,ts) p f* setzt den Parser *p* auf die Wortfolge *ts* an und erhält so einen zweiten Typen *a'.* Liefert Paar aus *f(a,a')* und Restfolge.
- *match (a,ts) t* prüft ob *t* als erstes Wort in *ts* erscheint und liefert in diesem Fall das Paar aus a und der Restfolge *tl ts*.

Links-klammernde Phrasen

- Typen klammern **rechts:**
  parse[INT, ARROW, BOOL, ARROW, BOOL]
  = (Arrow(Int, Arrow(Bool, Bool), [])

- **Gedankenexperiment:** Wie würde man **links-klammernde Typen** parsen?
  parse'[INT, ARROW, BOOL, ARROW, BOOL]
  = (Arrow(Arrow(Int,Bool), Bool), [])

- **Problem: Konkrete Grammatik** für links-klammernde Typen ist **nicht RA-tauglich**

  ty ::= *pty* [“->” *ty*] → *ty* ::= [*ty* “->”] *pty*

  pty ::= “bool” | “int” | “(“ *ty* “)”) → *pty* ::= “bool” | “int” | “(“ *ty* “)”)

  Neue Grammatik
  ist nicht RA-tauglich!

Links-klammernde Typen

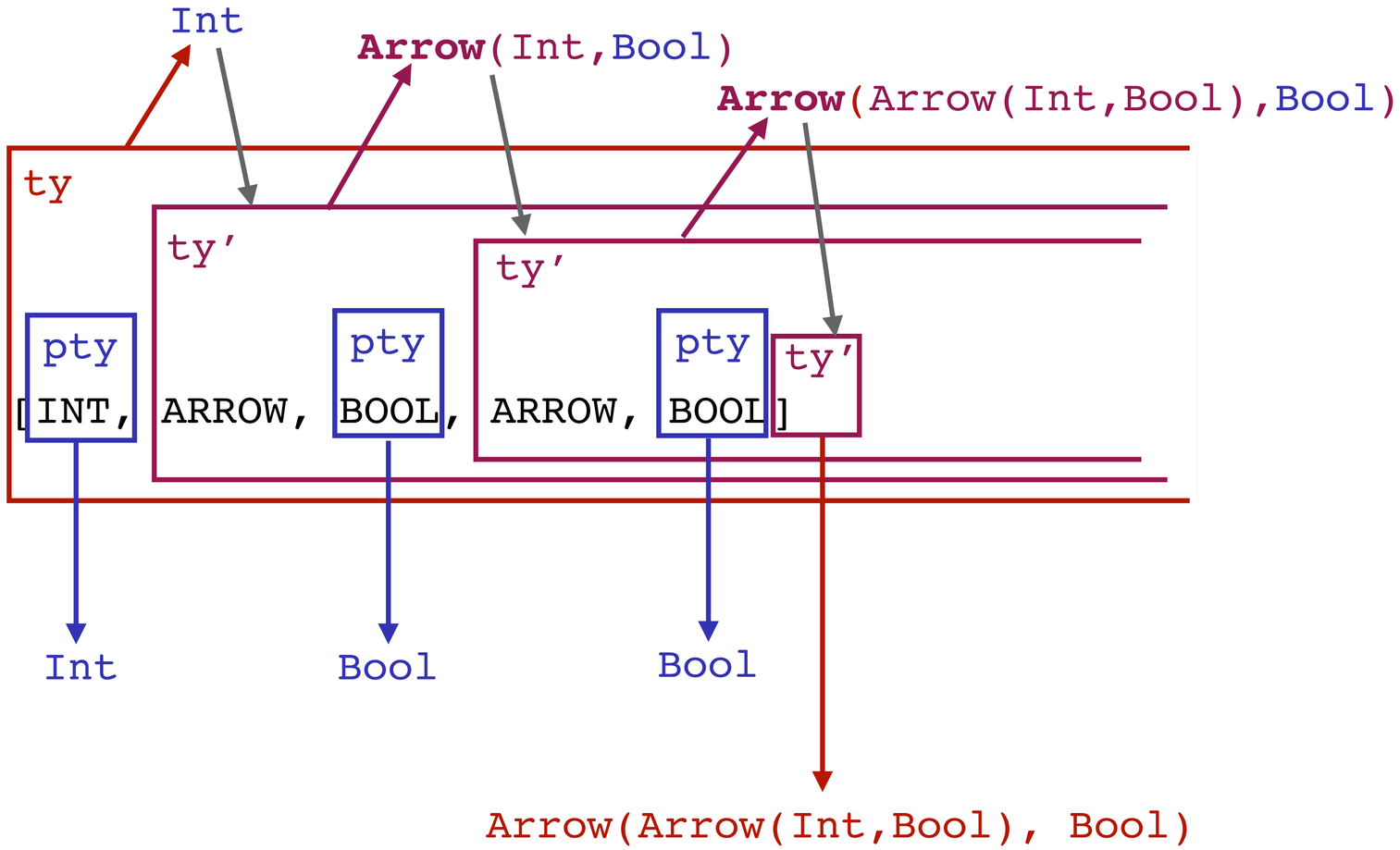
- **Problem: Konkrete Grammatik** für links-klammernde Typen ist **nicht RA-tauglich**

ty ::= *pty* [“->” *ty*]
pty ::= “bool” | “int” | “(“ *ty* “)”)

~~*ty* ::= [*ty* “->”] *pty*~~
~~*pty* ::= “bool” | “int” | “(“ *ty* “)”)~~
Neue Grammatik ist nicht RA-tauglich!

Trick:
Wir lassen die **konkrete** Grammatik **rechts-klammernd** (Sätze bleiben gleich!), **bauen** die **abstrakte** Darstellung aber **links-klammernd.**

Idee


Int
Arrow(Int,Bool)
Arrow(Arrow(Int,Bool),Bool)
ty
ty'
ty'
pty
[INT,
ARROW,
pty
BOOL,
ARROW,
pty
BOOL
ty'
]
Int
Bool
Bool
Arrow(Arrow(Int,Bool), Bool)

Parser für links-klammernde Typen

- **Konkrete Grammatik** (wie bisher, rechts klammernd):
  ty ::= *pty* [“->” *ty*]
  pty ::= “bool” | “int” | “(“ *ty* “)”
- (Äquivalente) **Grammatik mit Hilfskategorie:**
  ty ::= *pty ty'*
  ty' ::= [“->” *pty ty'*]
  pty ::= “bool” | “int” | “(“ *ty* “)”
- **Parser:** *ty'* hat als zusätzliches Argument
  abstrakte Darstellung der bisher gelesenen Teilfolge

```
fun ty ts =  ty' (pty ts)
and ty' (t, ARROW::tr)
          = let val (t', tr') = pty tr
                in ty' (Arrow(t,t'),tr') end
  | ty' s = s;
```

zusätzliches Argument: abstrakte Darstellung der bereits gelesenen Teilfolge

Beispiel (links-klammernde Typen)

```
fun ty ts =  ty' (pty ts)
and ty' (t, ARROW::tr)
          = let val (t', tr') = pty tr
                in ty' (Arrow(t,t'),tr') end
  | ty' s = s;
```

zusätzliches Argument: abstrakte Darstellung der bereits gelesenen Teilfolge

```
ty[INT, ARROW, BOOL, ARROW, BOOL]
  = ty' (pty [INT, ARROW, BOOL, ARROW, BOOL])
  = ty' (Int, [ARROW, BOOL, ARROW, BOOL])
  = let val (t', tr') = pty [BOOL, ARROW, BOOL]
    in ty' (Arrow(Int,t'),tr') end
  = ty' (Arrow(Int,Bool), [ARROW,BOOL])
  = let val (t', tr') = pty [BOOL]
    in ty' (Arrow(Arrow(Int,Bool),t'),tr') end
  = ty' (Arrow(Arrow(Int,Bool), Bool), [])
  = (Arrow(Arrow(Int,Bool), Bool), [])
```

Arithmetische Ausdrücke

$z \in \mathbb{Z}$

$x \in Id$

$e \in Exp = z \mid x \mid e + e \mid e * e$

Phrasale Syntax

$$
\begin{aligned}
exp &::= [exp\ \texttt{"+"}]\ mexp \\
mexp &::= [mexp\ \texttt{"*"}]\ pexp \\
pexp &::= num \mid id \mid \texttt{"("}\ exp\ \texttt{")"}
\end{aligned}
$$

Lexikalische Syntax

$$
\begin{aligned}
word &::= \texttt{"+"} \mid \texttt{"*"} \mid \texttt{"("} \mid \texttt{")"} \mid num \mid id \\
num &::= [\texttt{"\~{}"}]\ pnum \\
pnum &::= digit\ [pnum] \\
digit &::= \texttt{"0"} \mid \cdots \mid \texttt{"9"} \\
id &::= letter\ [id] \\
letter &::= \texttt{"a"} \mid \cdots \mid \texttt{"z"} \mid \texttt{"A"} \mid \cdots \mid \texttt{"Z"}
\end{aligned}
$$

Lexer

```
datatype token = ADD | MUL | LPAR | RPAR | ICON of int | ID of string

fun lex nil = nil
  | lex (#" ":: cr) = lex cr
  | lex (#"\t":: cr) = lex cr
  | lex (#"\n":: cr) = lex cr
  | lex (#"+":: cr) = ADD:: lex cr
  | lex (#"*":: cr) = MUL:: lex cr
  | lex (#"(":: cr) = LPAR:: lex cr
  | lex (#")":: cr) = RPAR:: lex cr
  | lex (#"~":: c:: cr) = if Char.isDigit c then lexInt ~1 0 (c::cr)
                          else raise Error "~"
  | lex (c::cr) = if Char.isDigit c then lexInt 1 0 (c::cr)
                  else if Char.isAlpha c then lexId [c] cr
                  else raise Error "lex"
```

Lexer

```
  | lex (#"~":: c:: cr) = if Char.isDigit c then lexInt ~1 0 (c::cr)
                          else raise Error "~"
  | lex (c::cr) = if Char.isDigit c then lexInt 1 0 (c::cr)
                  else if Char.isAlpha c then lexId [c] cr
                  else raise Error "lex"
and lexInt s v cs = if null cs orelse not(Char.isDigit (hd cs))
                    then ICON(s*v) :: lex cs
                    else lexInt s (10*v+(ord(hd cs)-ord#"0")) (tl cs)
```

Vorzeichen

bisher gelesener Wert

```
and lexId cs cs' = if null cs' orelse not(Char.isAlpha (hd cs'))
                   then ID(implode(rev cs)) :: lex cs'
                   else lexId (hd cs' ::cs) (tl cs')
```

bisher gelesene Zeichen (reversiert)

Was ist das Ergebnis von

lex (explode "abc123+234a")?

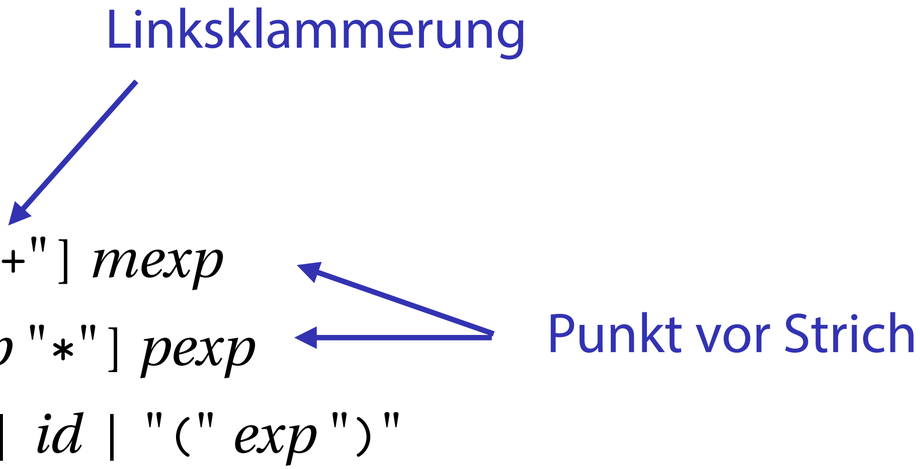
A: [ID "abc123+234a"] ✗

B: [ID "abc123+", ID "234a"] ✗

C: [ID "abc123", ADD, ICON 234, ID "a"] ✗

D: [ID "abc", ICON 123, ADD, ICON 234, ID "a"] ✓

Phrasale Syntax

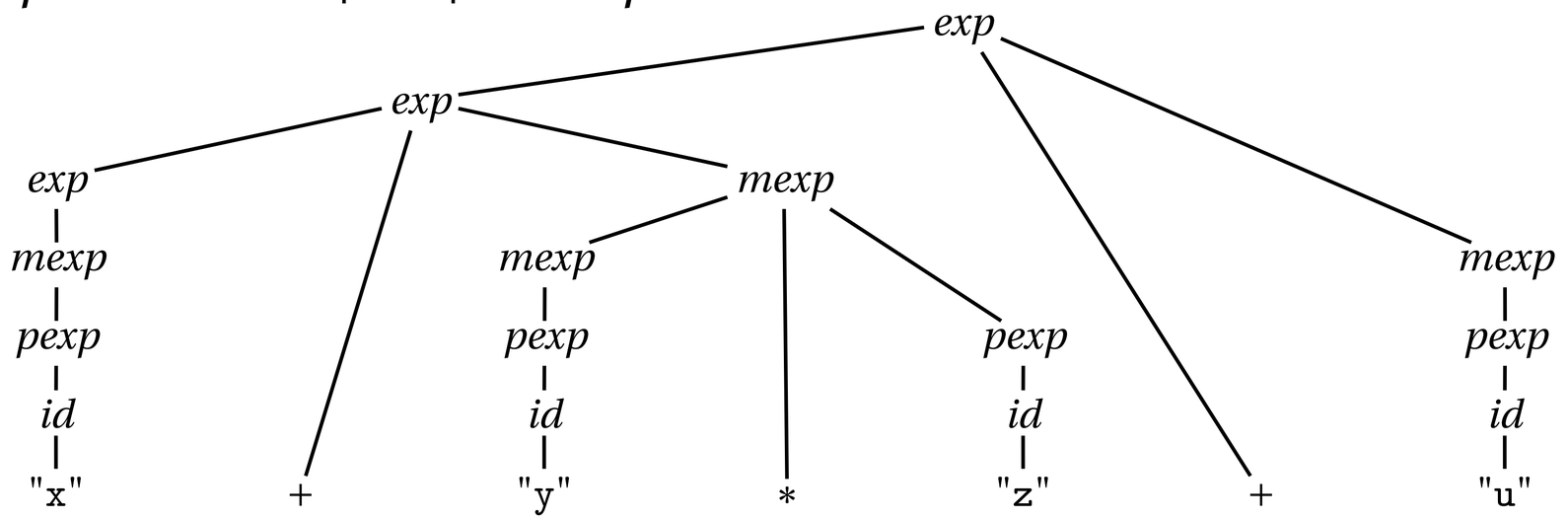


$$exp \;::=\; [\,exp\ \texttt{"+"}\,]\ mexp$$
$$mexp \;::=\; [\,mexp\ \texttt{"*"}\,]\ pexp$$
$$pexp \;::=\; num \mid id \mid \texttt{"("}\ exp\ \texttt{")"}$$

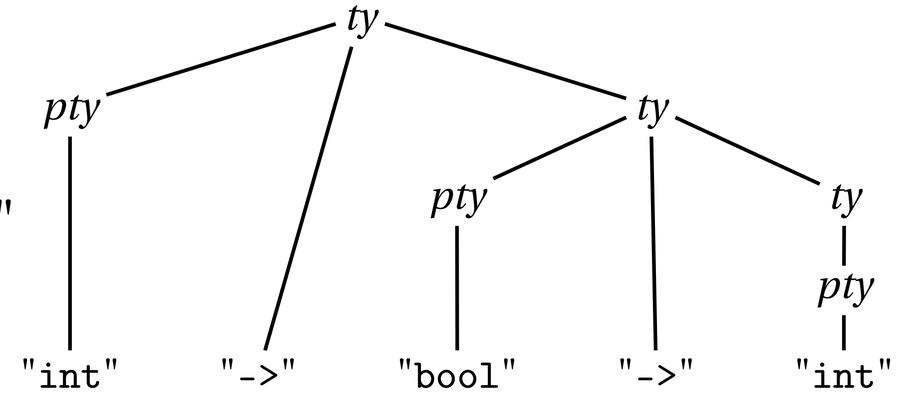
vgl. Rechtsklammerung

$$ty \;::=\; pty\ [\,\texttt{"->"}\ ty\,]$$
$$pty \;::=\; \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"}$$

$$exp \;::=\; [\,exp\ \texttt{"+"}\,]\ mexp$$
$$mexp \;::=\; [\,mexp\ \texttt{"*"}\,]\ pexp$$
$$pexp \;::=\; num \mid id \mid \texttt{"("}\ exp\ \texttt{")"}$$



$$ty \;::=\; pty\ [\,\texttt{"->"}\ ty\,]$$
$$pty \;::=\; \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"}$$



Phrasale Syntax der arithmetischen Ausdrücke

links-rekursiv

links-rekursiv

$$
\begin{aligned}
exp &::= [\, exp \text{ "+"} \,]\ mexp \\
mexp &::= [\, mexp \text{ "*"} \,]\ pexp \\
pexp &::= num \mid id \mid \text{"("}\ exp\ \text{")"}
\end{aligned}
$$

nicht RA tauglich!

Trick für links-Klammerung:

Um die Grammatik RA-tauglich zu machen ersetzen wir die links-rekursiven Gleichungen durch rechts-rekursive Gleichungen.

(Das ändert nichts an den Sätzen der Grammatik)

rechts-rekursiv

$$
\begin{aligned}
exp &::= mexp\ [\text{"+"}\ exp] \\
mexp &::= pexp\ [\text{"*"}\ mexp]
\end{aligned}
$$

RA tauglich!

Hilfskategorien

$$
\begin{aligned}
exp &::= mexp\ [\text{"+"}\ exp] \\
mexp &::= pexp\ [\text{"*"}\ mexp]
\end{aligned}
$$

Hilfskategorien ändern nichts an den Sätzen der Grammatik
(aber vereinfachen das Schreiben des Parsers):

$$
\begin{aligned}
exp &::= mexp\ exp' \\
exp' &::= [\text{"+"}\ mexp\ exp'\ ] \\
mexp &::= pexp\ mexp' \\
mexp' &::= [\text{"*"}\ pexp\ mexp'\ ]
\end{aligned}
$$

Parser für arithmetische Ausdrücke

```
fun exp ts = exp' (mexp ts)
and exp' (e, ADD::tr) = let val (e', tr') = mexp tr
                        in exp' (Sum(e,e'),tr') end
  | exp' s = s
and mexp ts = mexp' (pexp ts)
and mexp' (e, MUL::tr) = let val (e', tr') = pexp tr
                         in mexp' (Pro(e,e'), tr') end
  | mexp' s = s
and pexp (ICON z :: tr) = (Con z, tr)
  | pexp (ID x :: tr) = (Id x, tr)
  | pexp (LPAR :: tr) = case exp tr of
                          (e',(RPAR::tr')) => (e', tr')
                        | _ => raise Error "pexp";
```

zusätzliches Argument: abstrakte Darstellung der bereits gelesenen Teilfolge

zusätzliches Argument: abstrakte Darstellung der bereits gelesenen Teilfolge

Konkrete Syntax für F

$$
\begin{aligned}
ty &::= pty\ [\texttt{"->"}\ ty] \\
pty &::= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"} \\
exp &::= \texttt{"if"}\ exp\ \texttt{"then"}\ exp\ \texttt{"else"}\ exp \\
&\mid \texttt{"fn"}\ id\ \texttt{":"}\ ty\ \texttt{"=>"}\ exp \\
&\mid aexp\ [\texttt{"<="}\ aexp] \\
aexp &::= [aexp\ (\texttt{"+"} \mid \texttt{"-"})]\ mexp \quad \longleftarrow \text{links-rekursiv} \\
mexp &::= [mexp\ \texttt{"*"}]\ sexp \quad \longleftarrow \text{links-rekursiv} \\
sexp &::= [sexp]\ pexp \quad \longleftarrow \text{links-rekursiv} \\
pexp &::= \texttt{"false"} \mid \texttt{"true"} \mid num \mid id \mid \texttt{"("}\ exp\ \texttt{")"}
\end{aligned}
$$

Kapitel 14
Datenstrukturen

Strukturen

- Mit einer **Struktur** fassen wir verschiedene Objekte zusammen.
- Eine **Strukturdeklaration** ist eine Folge von Deklarationen:

structure $\langle Bezeichner \rangle$ = struct $\langle Deklaration \rangle$ ... $\langle Deklaration \rangle$ end

Beispiel:

```
structure S = struct
    val a = 4
    fun f x = x+1
end
```

structure S : {val a : int, val f : int → int}

Die **Felder** der Struktur, *a* und *f,* sind außerhalb der Strukturdeklaration über die **zusammengesetzten Bezeichner** *S.a* und *S.f* verfügbar.

Beispiel

```
structure S = struct
    val a = 4
    exception E
    datatype t = A | B
    structure T = struct val a = a+3 end
end
```

```
S.T.a
```

7 : int

```
fun switch S.A = S.B
  | switch S.B = S.A
```

val switch : S.t → S.t

Implementierung von Datenstrukturen

- Strukturdeklarationen sind dazu da, die **Implementierung** einer Datenstruktur zu **kapseln**.

Beispiel: (Endliche) Mengen von ganzen Zahlen

```
structure ISet = struct
    type set = int list
    fun set xs = xs
    fun union xs ys = xs@ys
    fun elem ys x = List.exists (fn y => y=x) ys
    fun subset xs ys = List.all (elem ys) xs
end
```

Verbergen von Implementierungsdetails

- Problem: Die **Darstellungsgleichheit** (Gleichheit von Listen) stimmt nicht mit der **abstrakten Gleichheit** (Gleichheit von Mengen) überein!
- Lösung: **Verbergen von Implementierungsdetails** durch einen **Signaturconstraint:**

```
signature ISET = sig
    type set
    val set    : int list -> set
    val union  : set -> set -> set
    val subset : set -> set -> bool
end
```

Nur die in dem Signaturconstraint **angegebenen Felder** sind für den Benutzer der Signatur **sichtbar.**

Verbergen von Implementierungsinformation

```
signature ISET = sig
    type set
    val set    : int list -> set
    val union  : set -> set -> set
    val subset : set -> set -> bool
end

structure ISet :> ISET = struct
    type set = int list
    fun set xs = xs
    fun union xs ys = xs@ys
    fun elem ys x = List.exists (fn y => y=x) ys
    fun subset xs ys = List.all (elem ys) xs
end
```

abstrakter Typ
Gleichheitstest wird verborgen.
(sichtbar bei eqtype)

Typannahmen
werden überprüft

Implementierung darf allgemeiner sein als Signatur.

polymorphe Implementierung

elem ist **verborgen**

www.prog1.saarland