Programmierung 1

Vorlesung 20

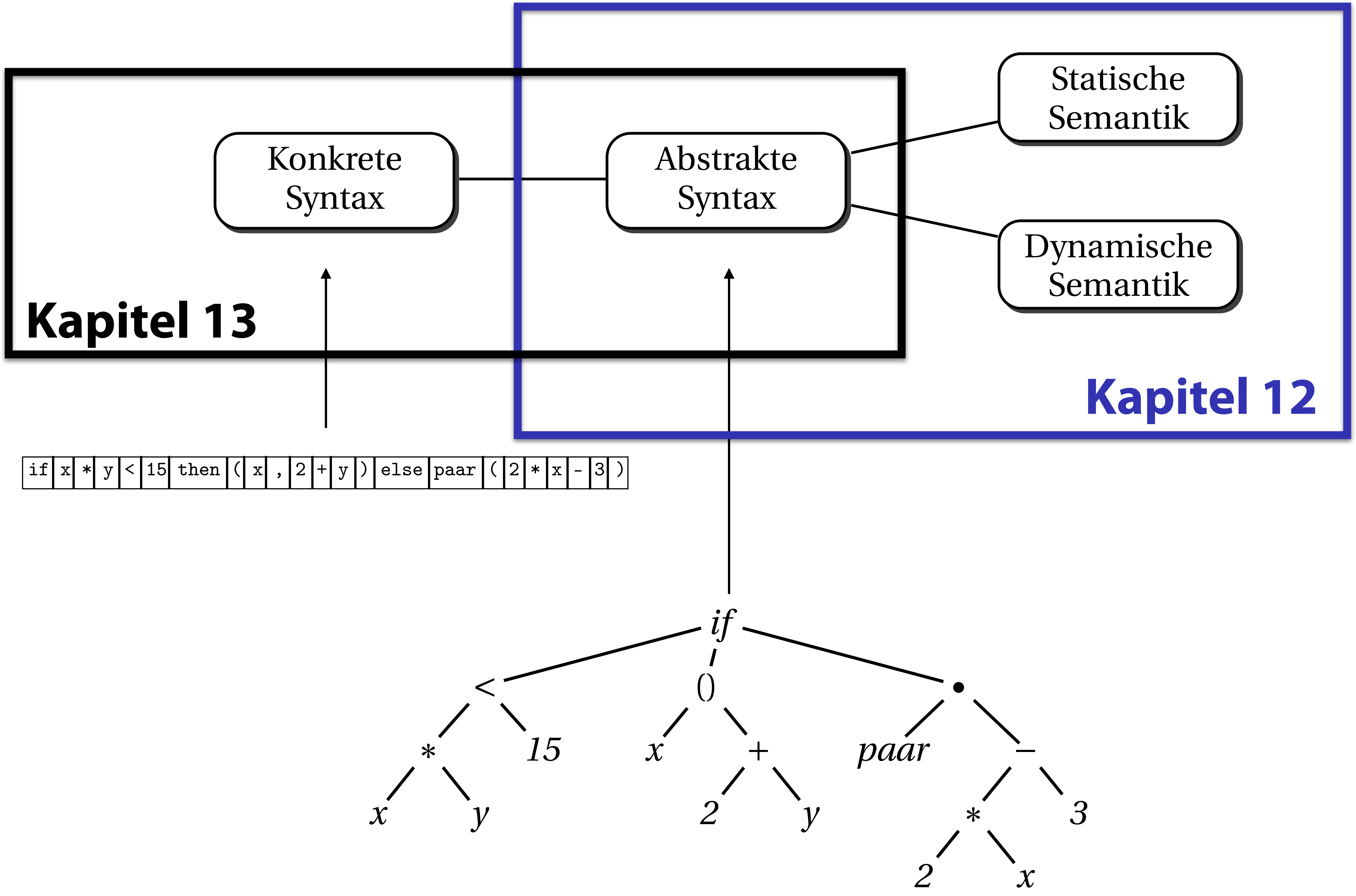
Livestream beginnt um 10:20 Uhr

Kapitel 13
Konkrete Syntax

Verarbeitungsphasen eines Interpreters

Lexikalische Analyse *Lexikalische Syntax*	Kann die eingegebene Zeichenfolge als **Folge von Wörtern** aufgefasst werden?
↓	
Syntaktische Analyse *Phrasale Syntax*	Beschreibt die Wortfolge ein **Programm**?
	Kapitel 13
↓	
Semantische Analyse *Statische Semantik*	Ist das Programm **semantisch zulässig**?
↓	
Ausführung *Dynamische Semantik*	**Führe** das Programm **aus.**
	Kapitel 12

Abstrakte Syntax



Abstrakte Syntax von F

Die abstrakte Syntax wird üblicherweise mithilfe einer schematischen Darstellung, der **abstrakten Grammatik**, definiert.

z	$\in$	$\mathbb{Z}$	Zahlen
c	$\in$	$Con = false \mid true \mid z$	Konstanten
x	$\in$	$Id = \mathbb{N}$	Bezeichner
o	$\in$	$Opr = + \mid - \mid * \mid \leq$	Operatoren
t	$\in$	$Ty = bool \mid int \mid t \to t$	Typen
e	$\in$	$Exp =$	Ausdrücke
		c	Konstante
		$\mid x$	Bezeichner
		$\mid eoe$	Operatoranwendung
		$\mid if\ e\ then\ e\ else\ e$	Konditional
		$\mid fn\ x : t \Rightarrow e$	Abstraktion
		$\mid ee$	Prozeduranwendung

Die **Grammatik** definiert die **Mengen** Con, Id, Opr, Ty, und Exp

z, x, o, t, e sind **Metavariablen:** sie bezeichnen Objekte der Mengen

Typdeklarationen

In Standard ML können wir die **abstrakte Syntax** einer Sprache als **Typdeklarationen** darstellen.

$$
\begin{array}{lll}
z & \in & \mathbb{Z} \\
c & \in & Con = \mathit{false} \mid \mathit{true} \mid z \\
x & \in & Id \ = \mathbb{N} \\
o & \in & Opr = + \mid - \mid * \mid \leq \\
t & \in & Ty \ = \mathit{bool} \mid \mathit{int} \mid t \to t \\
e & \in & Exp = \\
 & & \quad c \\
 & & \mid x \\
 & & \mid eoe \\
 & & \mid \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e \\
 & & \mid \mathit{fn}\ x : t \Rightarrow e \\
 & & \mid ee
\end{array}
$$

```
datatype con = False | True | IC of int
type     id  = string
datatype opr = Add | Sub | Mul | Leq

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

datatype exp =
    Con of con
  | Id  of id
  | Opr  of opr * exp * exp
  | If  of exp * exp * exp
  | Abs of id  * ty  * exp
  | App of exp * exp
```

Elaborierung

Sif $\dfrac{T \vdash e_1 : bool \quad T \vdash e_2 : t \quad T \vdash e_3 : t}{T \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : t}$

```
| elab f (If(e1,e2,e3)) =
          (case (elab f e1, elab f e2, elab f e3) of
             (Bool, t2, t3) => if t2=t3 then t2
                                else raise Error "T If1"
           |  _ => raise Error "T If2")
| elab f (Abs(x,t,e)) = Arrow(t, elab (update f x t) e)
```

Sabs $\dfrac{T[x := t] \vdash e : t'}{T \vdash \mathit{fn}\ x : t \Rightarrow e : t \to t'}$

```
  elab f (App(e1,e2)) = (case elab f e1 of
             Arrow(t',t) => if t' = elab f e2 then t
                            else raise Error "T App1"
           | _  => raise Error "T App2")
```

Sapp $\dfrac{T \vdash e_1 : t' \to t \quad T \vdash e_2 : t'}{T \vdash e_1 e_2 : t}$

```
fun update env x a y = if y=x then a else env y
```

Dynamische Semantik

D≤
$$\frac{V \vdash e_1 \rhd z_1 \qquad V \vdash e_2 \rhd z_2 \qquad z = \text{if } z_1 \le z_2 \text{ then } 1 \text{ else } 0}{V \vdash e_1 \le e_2 \rhd z}$$

Diftrue
$$\frac{V \vdash e_1 \rhd 1 \qquad V \vdash e_2 \rhd v}{V \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rhd v}$$

Diffalse
$$\frac{V \vdash e_1 \rhd 0 \qquad V \vdash e_3 \rhd v}{V \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rhd v}$$

Dabs
$$\frac{}{V \vdash \mathit{fn}\ x : t \Rightarrow e \rhd \langle x, e, V \rangle}$$

Dapp
$$\frac{V \vdash e_1 \rhd \langle x, e, V' \rangle \qquad V \vdash e_2 \rhd v_2 \qquad V'[x := v_2] \vdash e \rhd v}{V \vdash e_1\, e_2 \rhd v}$$

Evaluierung

eval: *value env* $\rightarrow$ *exp* $\rightarrow$ *value*

```
fun eval f (Con c) = evalCon c
  | eval f (Id x) = f x
  | eval f (Opr(opr,e1,e2)) = evalOpr opr (eval f e1) (eval f e2)
  | eval f (If(e1,e2,e3)) = (case eval f e1 of
                 IV 1 => eval f e2
                | IV 0 => eval f e3
                |  _   => raise Error "R If")
  | eval f (Abs(x,t,e)) = Proc(x, e, f)
  | eval f (App(e1,e2)) = (case (eval f e1, eval f e2) of
                 (Proc(x,e,f'), v) =>  eval (update f' x v) e
                | _ => raise Error "R App")
```

Zeichen und Wortdarstellung

Zeichendarstellung:

```
if x*y<15then(x,2+y)else paar(2*x-3)
```

Lexikalische Analyse
Lexikalische Syntax

Wortdarstellung:

`if`	`x`	`*`	`y`	`<`	`15`	`then`	`(`	`x`	`,`	`2`	`+`	`y`	`)`	`else`	`paar`	`(`	`2`	`*`	`x`	`-`	`3`	`)`

- Die **Zeichendarstellung** stellt eine Phrase als **Buchstabenfolge** dar. Leerzeichen (Zwischenraum, Tabulator, Zeilenwechsel) wo nötig.
- Die **Wortdarstellung** stellt eine Phrase als **Wortfolge** dar.

Lexikalische Syntax für Typen von F

- 5 **Wörter:** “`bool`”, “`int`”, “`->`”, “`(`”, “`)`”
- direkt aufeinanderfolgend oder getrennt durch **Leerzeichen:**
  - Zwischenraum “ “
  - Tabulator “`\t`”
  - Zeilenwechsel “`\n`”
- **Beispiele:**
  - lexikalisch zulässig:

```
"(int->int) -> int"     ⇝    "("  "int"  "->"  "int"  ")"  "->"  "int"

"  intbool->int  "      ⇝    "int"  "bool"  "->"  "int"
```

  - lexikalisch unzulässig:
    “`intboolbo0l`”
    “`bit`”

Lexikalische Syntax

- Eine wohldefinierte **lexikalische Syntax** ist eine Funktion, die Zeichenfolgen auf Wortfolgen abbildet.
- Die lexikalische Syntax ist typischerweise **nicht injektiv.**
- Ein **Lexer** ist eine Prozedur, die
  - **prüft,** ob eine **Zeichenfolge** lexikalisch **zulässig** ist und
  - sie gegebenenfalls in eine **Wortfolge übersetzt.**

```
lex (explode "(int->bool)->int")
```

[LPAR, INT, ARROW, BOOL, RPAR, ARROW, INT] : token list

```
lex (explode "  intbool->int  ")
```

[INT, BOOL, ARROW, INT] : token list

Ein Lexer für Typen von F

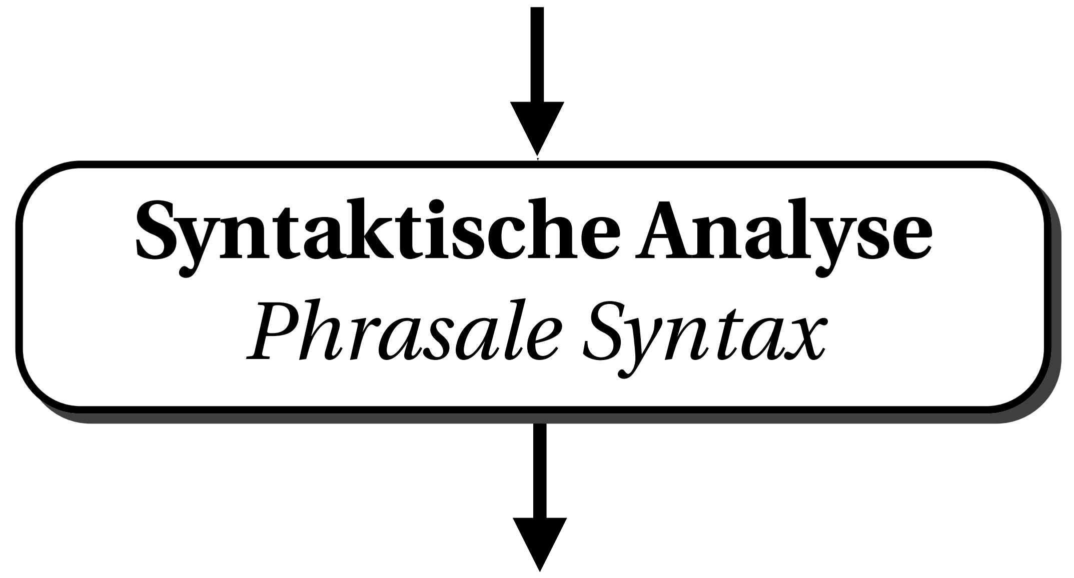
```
exception Error of string

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

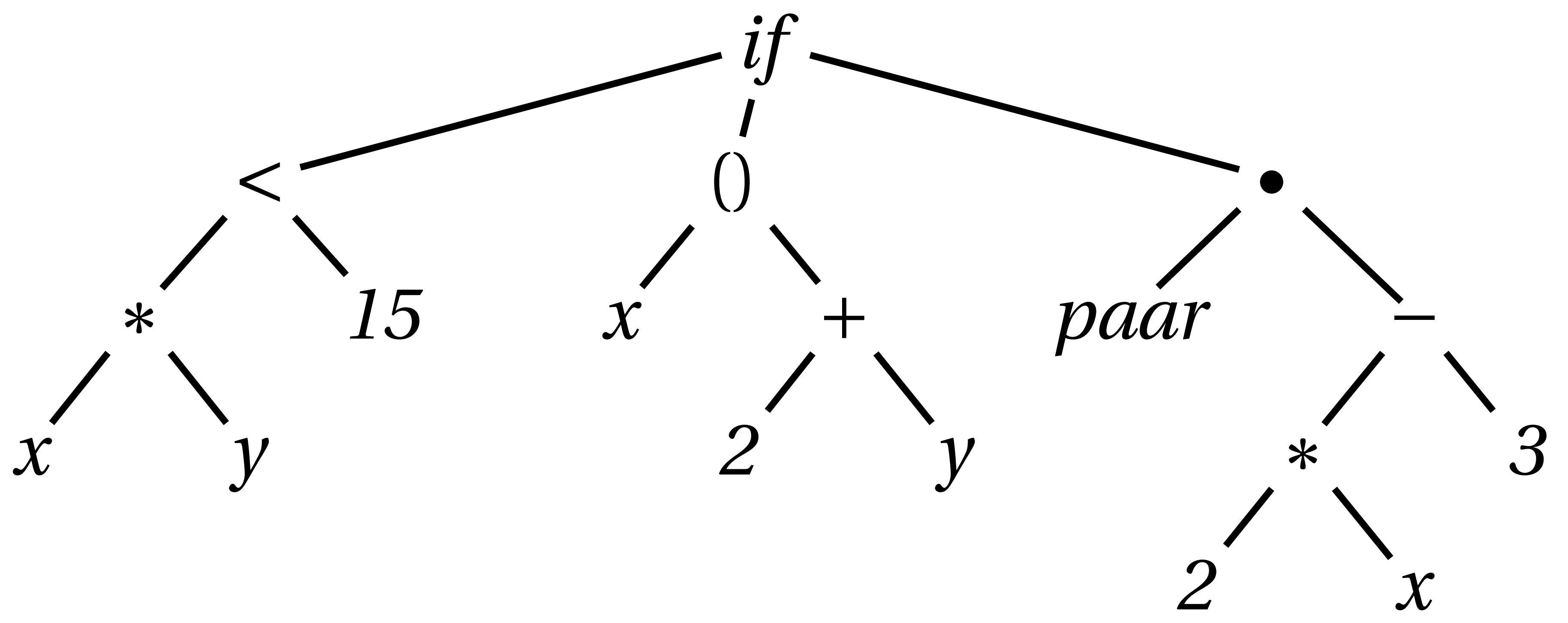
Baumdarstellung

Wortdarstellung:

if	x	*	y	<	15	then	(	x	,	2	+	y	)	else	paar	(	2	*	x	-	3	)



Baumdarstellung:



Phrasale Syntax

Die **phrasale Syntax** beschreibt, wie die Bäume der **abstrakten Syntax** durch **Wortfolgen** darzustellen sind.

Beispiel:

- **Abstrakte Grammatik** der Typen von F:

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

- Dazu korrespondierende **konkrete Grammatik:**

$$\begin{aligned} ty &::= \quad pty \mid pty\ \texttt{"->"}\ ty \\ pty &::= \quad \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"} \end{aligned}$$

Konkrete Grammatik

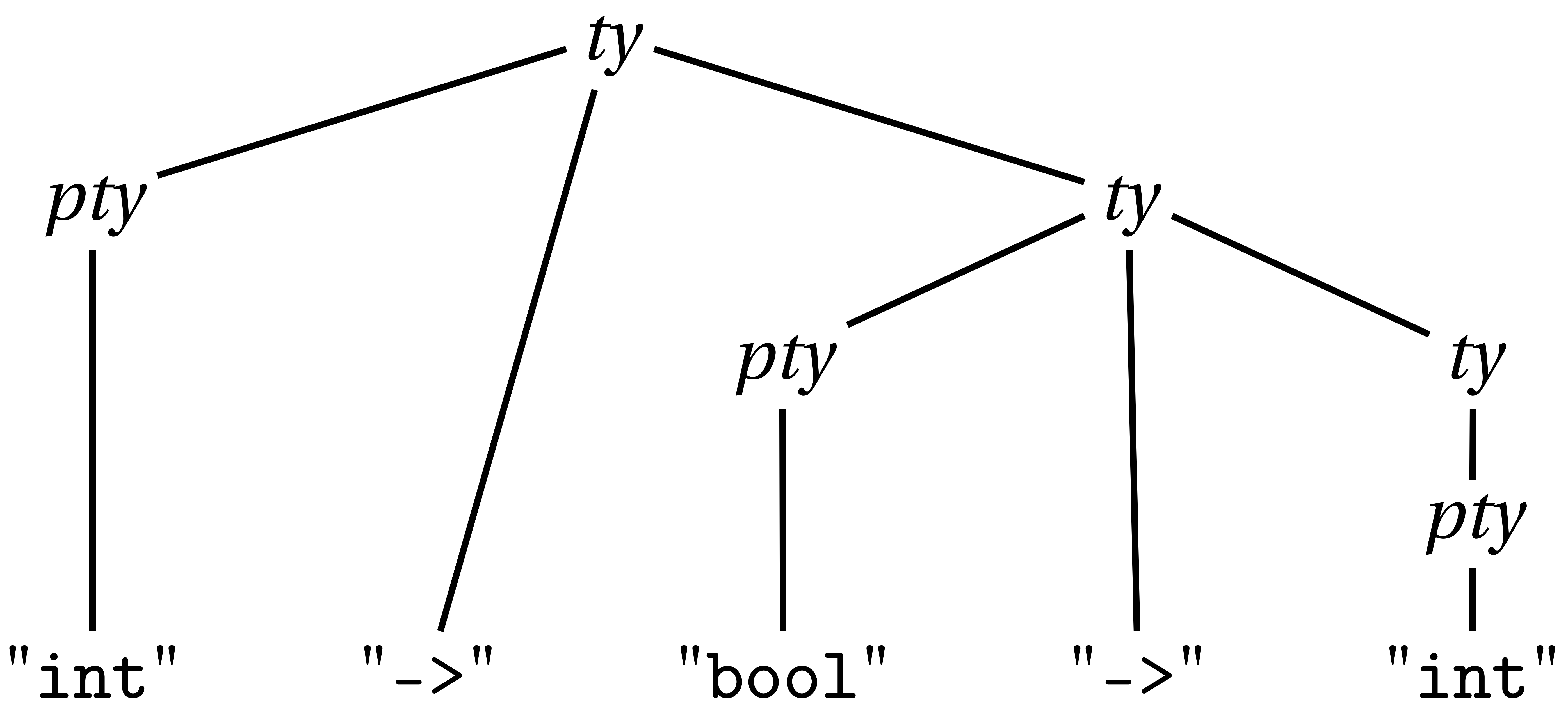
$$\begin{aligned} ty &::= pty \mid pty \, \texttt{"->"} \, ty \\ pty &::= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("} \, ty \, \texttt{")"} \end{aligned}$$

- ty und pty werden als **syntaktische Kategorien** bezeichnet.
- Die Wortdarstellungen gemäß einer syntaktischen Kategorie bezeichnen wir als die Sätze (gemäß) der Kategorie.

  Beispiel: `int->int` ist ein Satz gemäß ty, aber
  kein Satz gemäß pty.
- Jeder **Satz einer konkreten Grammatik** kann aus mindestens einer Kategorie der Grammatik abgeleitet werden.
  Die baumartige Darstellung einer solchen Ableitung heißt **Syntaxbaum.**

Syntaxbaum

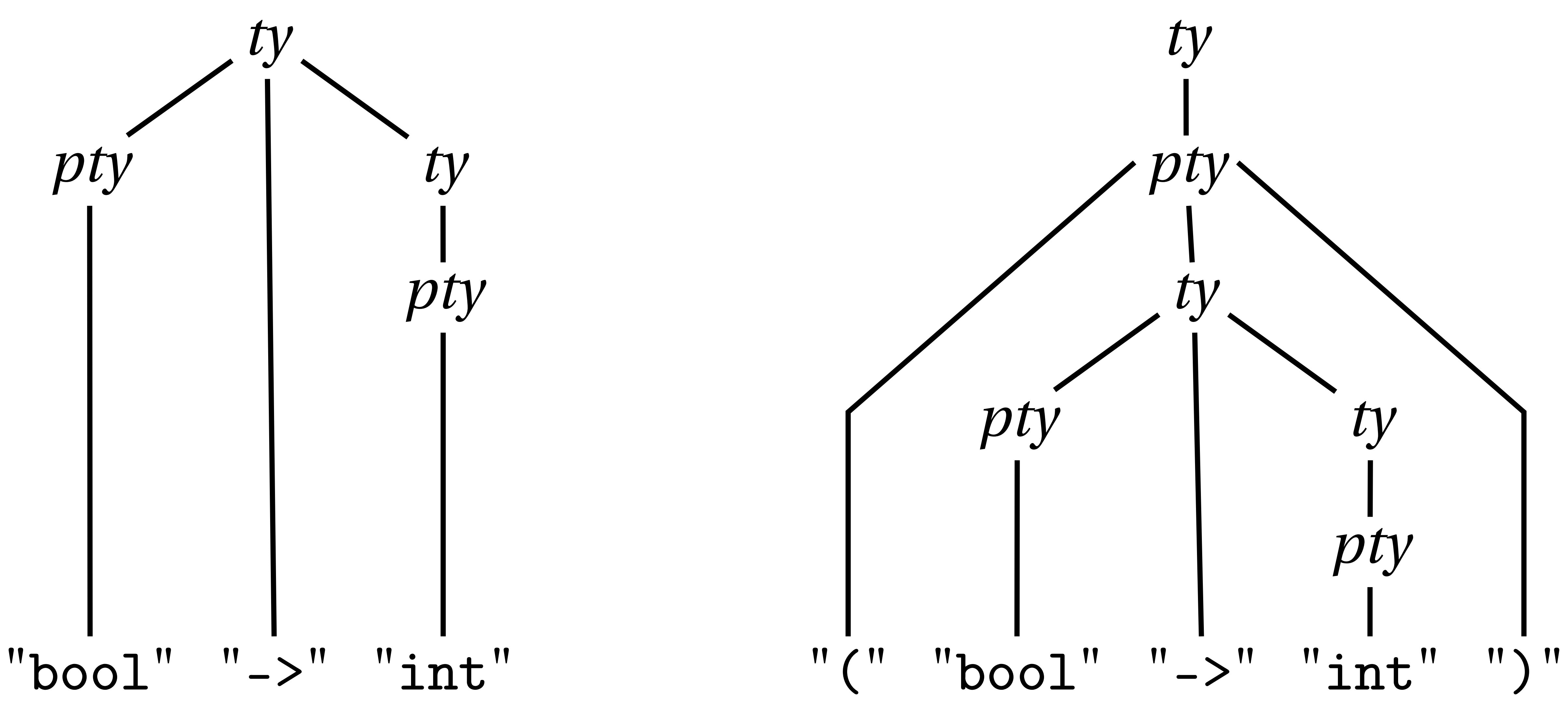
Ableitung des Satzes `int->bool->int` aus der Kategorie ty:



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

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

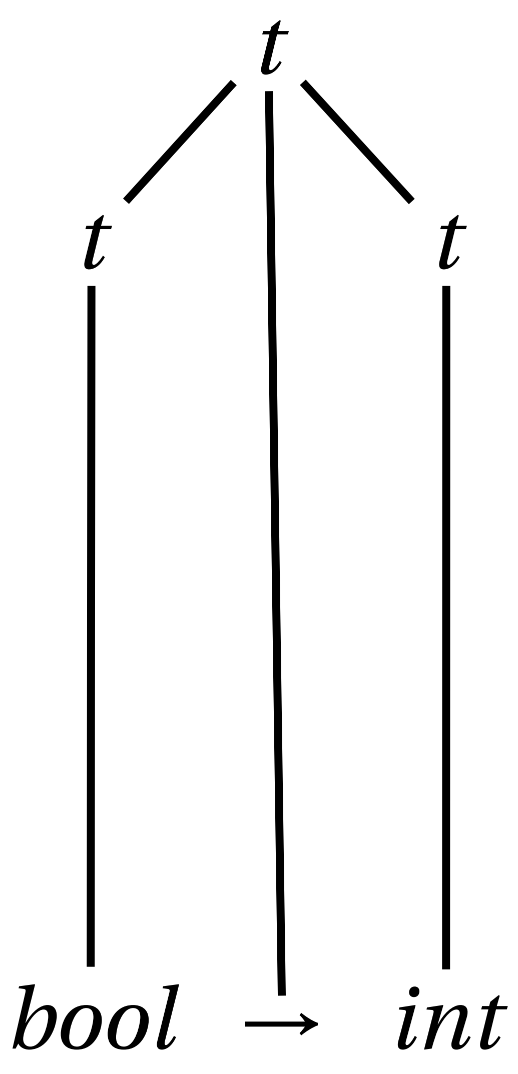
Konkrete und abstrakte Ableitungen



konkrete Ableitungen

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

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



abstrakte Ableitung

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

Affinität

- Eine konkrete Grammatik ist **affin** zu einer abstrakten Grammatik, wenn
  - jede konkrete Ableitung durch **genau eine** abstrakte Ableitung simuliert werden kann, und
  - jede abstrakte Ableitung durch **mindestens eine** konkrete Ableitung simuliert werden kann.
- Eine konkrete Grammatik heißt **eindeutig**, wenn
  - es zu jedem Satz einer Kategorie **höchstens eine** Ableitung gibt, die den Satz aus der Kategorie ableitet.

Beispiel:

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

ist eindeutig und affin zur abstrakten Grammatik für Typen von F.

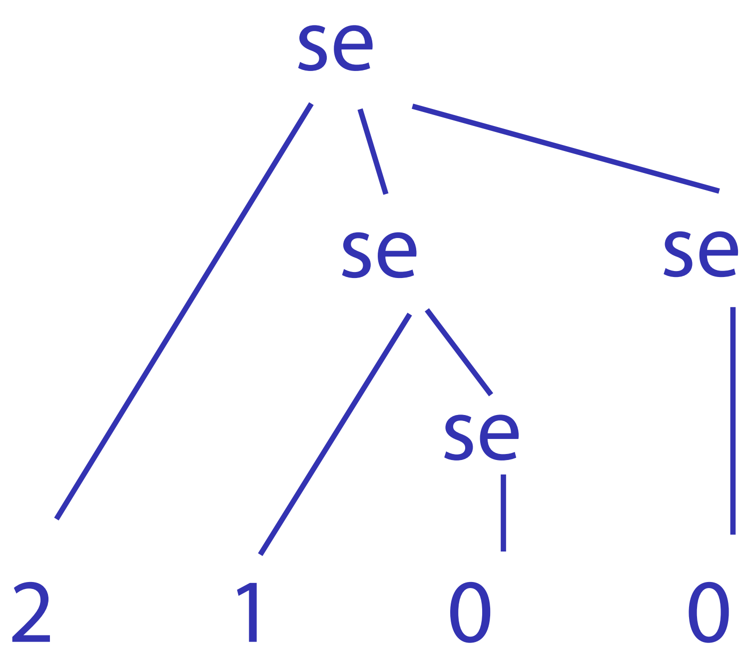
Parsing

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.

Rekursiver Abstieg



- **Beispielgrammatik:**

$$seq \;::=\; \texttt{"0"} \mid \texttt{"1"}\ seq \mid \texttt{"2"}\ seq\ seq$$

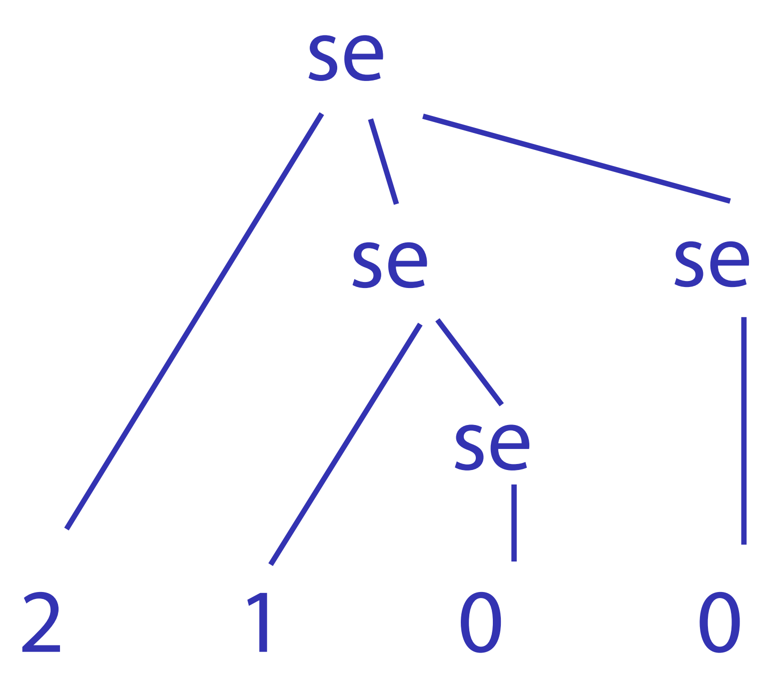
- **Algorithmische Interpretation durch rekursiven Abstieg:**

```
fun test (0::tr) = tr
  | test (1::tr) = test tr
  | test (2::tr) = test (test tr)
  | test _ = raise Error "test"
```

$val\ test : int\ list \to int\ list$

Die Prozedur test ist ein **Prüfer** für seq.
test ts prüft, ob ts mit einem **Satz** gemäß seq **beginnt.**
Wenn **ja:** liefert Restliste. Wenn **nein:** wirft Ausnahme.

Beispiel



$$seq \ ::= \ \texttt{"0"} \mid \texttt{"1"}\ seq \mid \texttt{"2"}\ seq\ seq$$

```
fun test (0::tr) = tr
  | test (1::tr) = test tr
  | test (2::tr) = test (test tr)
  | test _ = raise Error "test"
```

val test : int list → int list

```
test [2,1,0,0,0]
      = test ( test [1,0,0,0] )
      = test ( test [0,0,0] )
      = test [0,0]
      = [0]
```

RA-Tauglichkeit

- Eine konkrete Grammatik heißt RA-tauglich, falls für die algorithmische Interpretation ihrer Gleichungen gilt:
  - falls es zu einer Rekursion kommt, wurde die Argumentliste **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

Vom Baum zum Satz

- **konkrete Grammatik:**

  $$seq \ ::= \ \texttt{"0"} \mid \texttt{"1"}\ seq \mid \texttt{"2"}\ seq\ seq$$

- **Darstellung von Bäumen:**

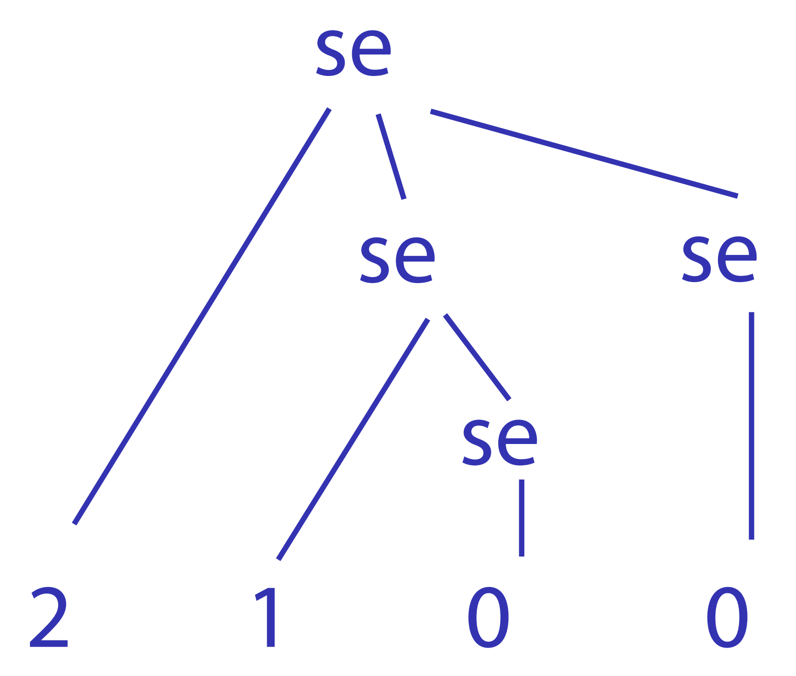
```
datatype tree = A | B of tree | C of tree * tree
```

- **Vom Baum zum Satz:**

```
fun rep A = [0]
  | rep (B t) = 1 :: rep t
  | rep (C(t,t')) = 2 :: rep t @ rep t'
```

  $rep : tree \to int\ list$

Beispiel



$$seq \ ::= \ "0" \mid "1" \ seq \mid "2" \ seq \ seq$$

```
datatype tree = A | B of tree | C of tree * tree
fun rep A = [0]
  | rep (B t) = 1 :: rep t
  | rep (C(t,t’)) = 2 :: rep t @ rep t’
```

$rep : tree \rightarrow int\ list$

```
rep (C(B A, A)) =
      = 2:: rep(B A) @ rep A
      = 2:: 1::rep A @ [0]
      = [2,1,0,0]
```

Vom RA-Prüfer zum RA-Parser

- **RA-Prüfer:**

```
datatype tree = A | B of tree | C of tree * tree
```

```
fun test (0::tr) = tr
  | test (1::tr) = test tr
  | test (2::tr) = test (test tr)
  | test _ = raise Error "test"
```

val test : int list → int list

- **RA-Parser:**

```
fun parse (0::tr) = (A, tr)
  | parse (1::tr) = let val (s,ts) = parse tr in (B s, ts) end
  | parse (2::tr) = let val (s,ts) = parse tr
                        val (s', ts') = parse ts
                    in  (C(s,s'), ts') end
  | parse _ = raise Error "parse"
```

*val parse : int list → tree * int list*

Beispiel

```
fun parse (0::tr) = (A, tr)
  | parse (1::tr) = let val (s,ts) = parse tr in (B s, ts) end
  | parse (2::tr) = let val (s,ts) = parse tr
                        val (s', ts') = parse ts
                    in  (C(s,s'), ts') end
  | parse _ = raise Error "parse"
```

```
parse[2,1,0,0,0]
  = let val (s,ts) = parse [1,0,0,0]
                   = let val (s,ts) = parse [0,0,0]
                                    = (A, [0,0])
                      in (B s, ts) end
                   = (B A, [0,0])
        val (s',ts') = parse [0,0]
                     = (A, [0])
     in (C(s,s'), ts') end
   = (C(B A, A), [0])
```

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 [ ]**

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

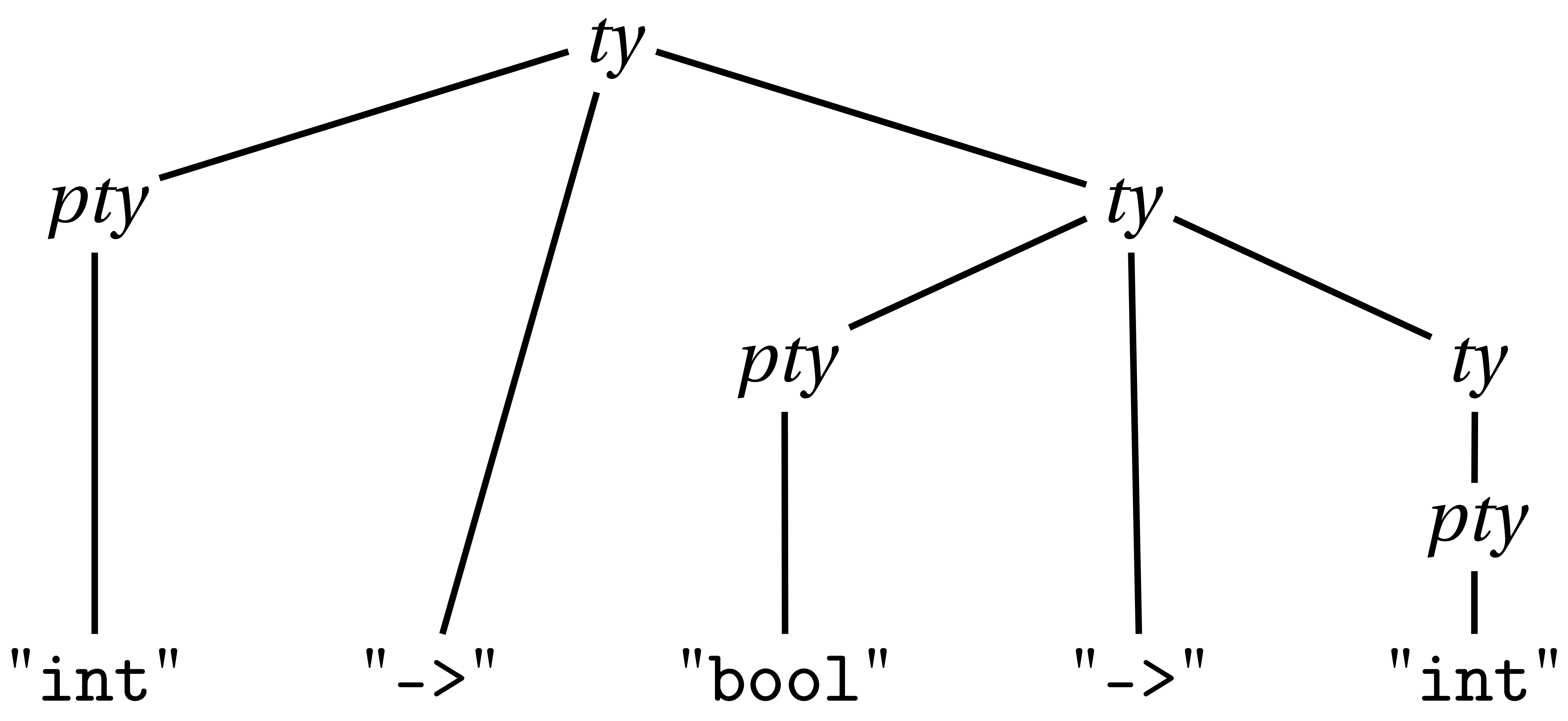
$$pty \;::=\; \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("} \; ty \; \texttt{")"}$$

Prüfe für ty zunächst gemäß pty.

Falls noch ein **Rest** bleibt der mit “**->**” beginnt,
prüfe nochmals gemäß ty.

Falls nicht: fertig.

Beispiel



$$
\begin{aligned}
ty &::= pty\ [\ \texttt{"->"}\ ty\ ] \\
pty &::= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ ty\ \texttt{")"}
\end{aligned}
$$

Prüfer für Typen

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

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

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

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"
```

val ty : token list → token list

val pty : token list → token list

Prüfe für ty zunächst gemäß pty.

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

Falls nicht: fertig.

Beispiel

```
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"
```

```
ty[INT, ARROW, BOOL, ARROW, BOOL]
  = case pty [INT, ARROW, BOOL, ARROW, BOOL] of
              ARROW::tr => ty tr | tr => tr

  = case [ARROW, BOOL, ARROW, BOOL] of
              ARROW::tr => ty tr | tr => tr

  = ty [BOOL, ARROW, BOOL]

  = case pty [BOOL, ARROW, BOOL] of
              ARROW::tr => ty tr | tr => tr

  = case [ARROW, BOOL] of
              ARROW::tr => ty tr | tr => tr

  = ty [BOOL] = case pty [] of … = []
```

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)
```

```
ty[INT, ARROW, BOOL, ARROW, BOOL]
  = case pty [INT, ARROW, BOOL, ARROW, BOOL] of
   (t,ARROW::tr) => (t, ARROW::tr) => let val (t', tr') = ty tr
                                      in (Arrow(t,t'),tr') end
              | s => s
  = case (Int, [ARROW, BOOL, ARROW, BOOL]) of
              (t,ARROW::tr) => let val (t', tr') = ty tr
                               in (Arrow(t,t'),tr') end
              | s => s
  = let val (t', tr') = ty [BOOL, ARROW, BOOL]
    in (Arrow(Int,t'),tr') end
  = let val (t', tr') = (Arrow(Bool, Bool), [])
    in (Arrow(Int,t'),tr') end
  = (Arrow(Int, Arrow(Bool, Bool)), [])
```

www.prog1.saarland