Unifikation

Definition

Sei V ein Alphabet von **Variablen**
und F ein Alphabet von **Funktionssymbolen**.

Jedem Funktionssymbol $f \in F$ ist eine natürliche Zahl $\rho(f) \geq 0$ zugeordnet, die die Zahl der Parameter der Funktion f angibt.

Funktionssymbole f mit $\rho(f) = 0$ heißen auch **Konstante**.

Mit Hilfe dieser Alphabete konstruiert man nun Ausdrücke (**Terme**) in Präfix-Notation.

Definition

Seien V und F Alphabete von Variablen und Funktionssymbolen.
Dann gilt:

(a) Jede Variable $x \in V$ ist ein Term.

(b) Ist $f \in F$ und sind $t_1, \ldots, t_{\rho(f)}$ Terme, so ist auch $f(t_1, \ldots, t_{\rho(f)})$ ein Term.

Bemerkung

Bildet man einen Term aus einer Konstanten f ($\rho(f) = 0$), schreibt man oft statt $f()$ kurz f.

Beispiel

Gegeben sei der Typ-Ausdruck

$$(char \times char) \rightarrow (int \rightarrow pointer(int \rightarrow int))$$

Das Alphabet F enthält die Funktionssymbole

$char$ und int mit $\rho(char) = \rho(int) = 0$ als Konstante,
$pointer$ mit $\rho(pointer) = 1$ und
$\rightarrow$ und $\times$ mit $\rho(\rightarrow) = \rho(\times) = 2$.

Also ist die Präfix-Notation dieses Ausdrucks:

$$\rightarrow (\times(char,char), \rightarrow (int, pointer(\rightarrow (int, int))))$$

Substitution von Termen

Definition

Eine **Substitution** ist eine Abbildung S, die jede Variable $x_i \in V$ auf einen Term t_i abbildet.
Eine Substitution wird in der Form $\{x_1 \mapsto t_1, \ldots, x_r \mapsto t_r\}$ notiert.
Variablen, die nicht in der Menge auftreten, werden durch die Substitution auf sich selbst abgebildet.

Die Substitution S_{id} bildet jede Variable auf sich selbst ab.
Es gilt offensichtlich $S_{id} = \{\}$.

Definition

Ist S eine Substitution und t ein Term, so entsteht der Term $S(t)$ aus t, indem man alle Vorkommen von Variablen in t durch den durch S zugeordneten Term ersetzt. $S(t)$ heißt auch **Instanz** von t.

Hintereinanderausführung von Substitutionen

Substitutionen können –wie Abbildungen– hintereinander ausgeführt werden.

Definition

Sind S_1 und S_2 zwei Substitutionen, so ist die zusammengesetzte Substitution $S_1 \circ S_2$ so definiert, dass für alle Argumente t gilt: $(S_1 \circ S_2)(t) = S_1(S_2(t))$.

Die $\circ$ Operation ist nicht kommutativ!

Unifikation zweier Terme

Definition

Zwei Terme t_1 und t_2 heißen **unifizierbar**, wenn es eine Substitution S gibt, so dass $S(t_1) = S(t_2)$ ist.
Eine Substitution mit dieser Eigenschaft heißt **Unifikator** von t_1 und t_2.
$S(t_1)$ heißt **Unifikation** von t_1 und t_2.

Ein Unifikator zweier Terme ist also eine Substitution, die bei Anwendung auf beide Terme dasselbe Ergebnis liefert.

Definition

Ein Unifikator U_a zweier Terme t_1 und t_2 heißt **allgemeinster Unifikator** von t_1 und t_2, wenn zu jedem Unifikator U von t_1 und t_2 eine Substitution S existiert, so dass $S \circ U_a = U$ gilt, also $U(t_1)$ eine Instanz von $U_a(t_1)$ ist.

Ein Algorithmus zur Bestimmung des allgemeinsten Unifikators

J.A. Robinson hat 1965 bewiesen, dass für zwei gegebene Terme, die unifizierbar sind, stets ein allgemeinster Unifikator existiert.

Algorithmus `unify` zur Berechnung eines allgemeinsten Unifikators

Gegeben seien zwei Terme t_1 und t_2 über einem Alphabet V von Variablen und einem Alphabet F von Funktionssymbolen.

Fall 1: Einer der beiden Terme ist eine Variable x.
Der andere Term sei t.

- Ist $t = x$, dann gib S_{id} zurück.
- Kommt x in t vor, so beende den Algorithmus:
  t_1 und t_2 sind nicht unifizierbar.
- Sonst gib die Substitution $\{x \mapsto t\}$ zurück.

Fall 2: Es sei $t_1 \equiv f(x_1, \ldots, x_n)$ und $t_2 \equiv g(y_1, \ldots, y_m)$.

- Ist $f \neq g$ oder $n \neq m$, so beende den Algorithmus:
  t_1 und t_2 sind nicht unifizierbar.
- Setze $S := S_{id}$ und führe für alle k, $1 \leq k \leq n$, aus:
  1. Berechne rekursiv einen allgemeinsten Unifikator S' von $S(x_k)$ und $S(y_k)$, also $S' = \texttt{unify}(S(x_k), S(y_k))$.
  2. Setze $S := S' \circ S$.
- Gib die Substitution S zurück.

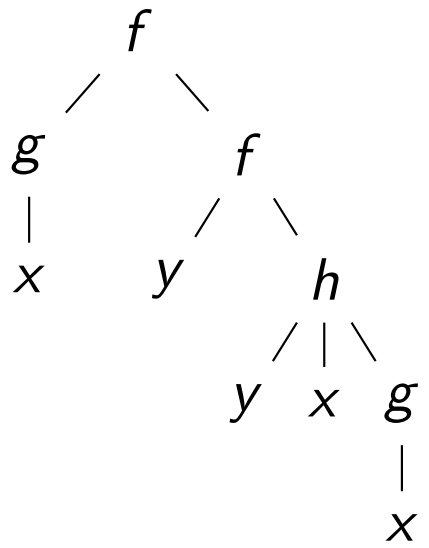
Bemerkung

Wenn man von einer **Unifikation** *zweier Typ-Ausdrücke spricht, so wird darunter meist eine allgemeinste Substitution verstanden, die beide Typ-Ausdrücke identisch macht.*

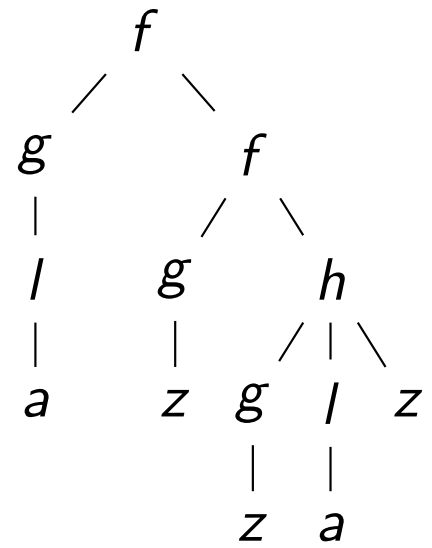
Beispiel: Unifikation

$$t = f(g(x), f(y, h(y, x, g(x)))) \quad \text{und}$$
$$u = f(g(l(a)), f(g(z), h(g(z), l(a), z)))$$

$t =$



$u =$

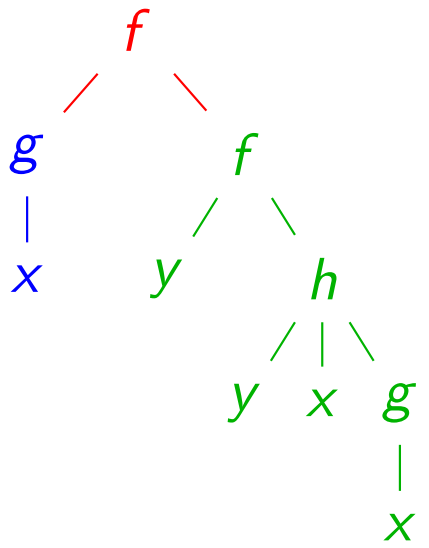


Beispiel: Unifikation

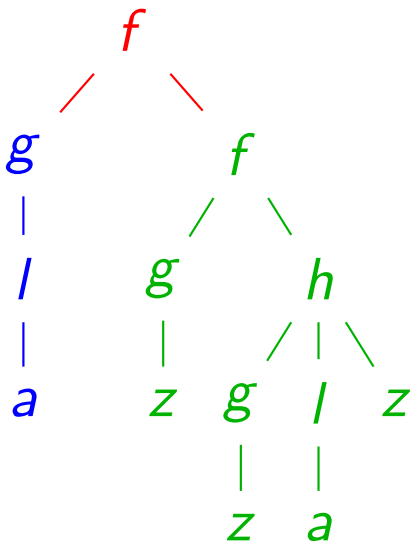
Fallunterscheidung: Ist einer der beiden Terme eine Variable? Nein (Fall 2).

Stimmen Funktionssymbole und Anzahl der Parameter überein? Ja.

$t =$



$u =$



Dann müssen die jeweiligen (ersten und zweiten) Parameter unifiziert werden.

Beispiel: Unifikation: 1. Parameter

Fallunterscheidung: Ist einer der beiden Terme eine Variable? Nein (Fall 2).

Stimmen Funktionssymbole und Anzahl der Parameter überein? Ja.

$t_1 =$ 

$u_1 =$ 

Dann müssen die jeweiligen (ersten) Parameter unifiziert werden.

Beispiel: Unifikation: 1.1. Parameter

Fallunterscheidung: Ist einer der beiden Terme eine Variable? Ja (Fall 1).

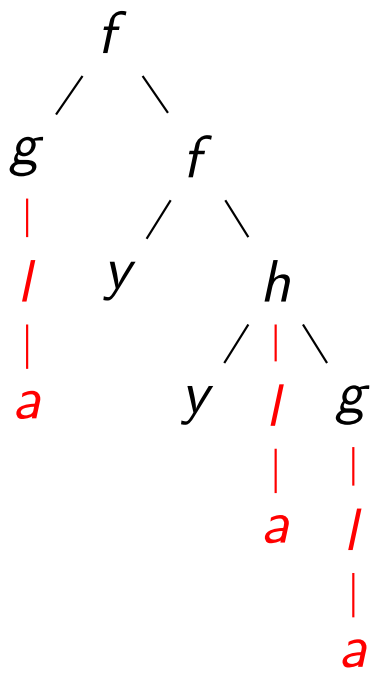
Kommt die Variable im anderen Term vor (*occur check*)? Nein.

$t_{1,1} =$ x $\qquad$ $u_{1,1} =$ $l(a)$

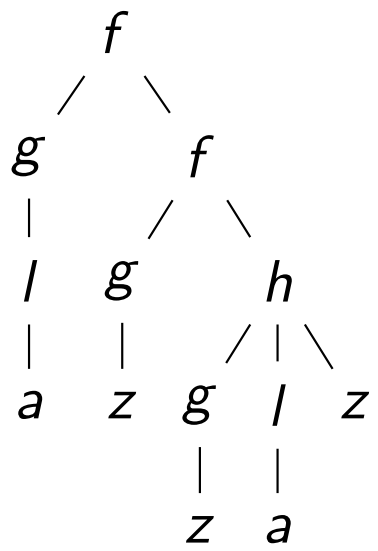
Dann erhalten wir die Substitution $S_1 = \{x \mapsto l(a)\}$ und wenden diese an.

Beispiel: Unifikation nach erster Substitution

$S_1(t) =$



$S_1(u) =$

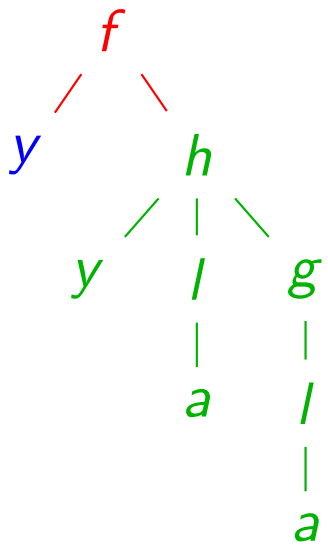


Beispiel: Unifikation 2. Parameter

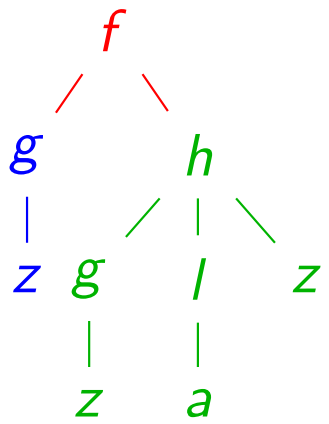
Fallunterscheidung: Ist einer der beiden Terme eine Variable? Nein (Fall 2).

Stimmen Funktionssymbole und Anzahl der Parameter überein? Ja.

$S_1(t_2) =$



$S_1(u_2) =$



Dann müssen die jeweiligen (ersten und zweiten) Parameter unifiziert werden.

Beispiel: Unifikation: 2.1. Parameter

Fallunterscheidung: Ist einer der beiden Terme eine Variable? Ja (Fall 1).

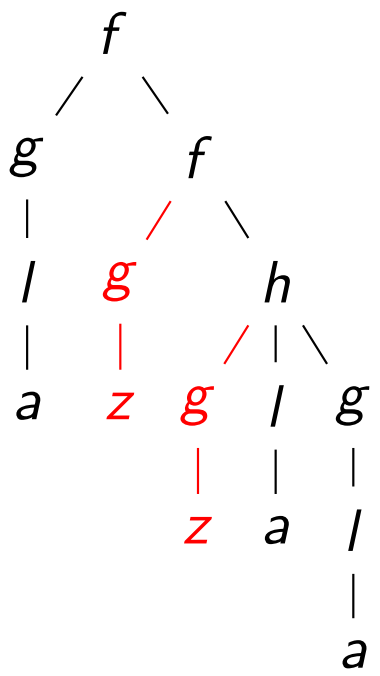
Kommt die Variable im anderen Term vor (*occur check*)? Nein.

$S_1(t_{2,1}) =$ y $\qquad$ $S_1(u_{2,1}) =$ $g(z)$

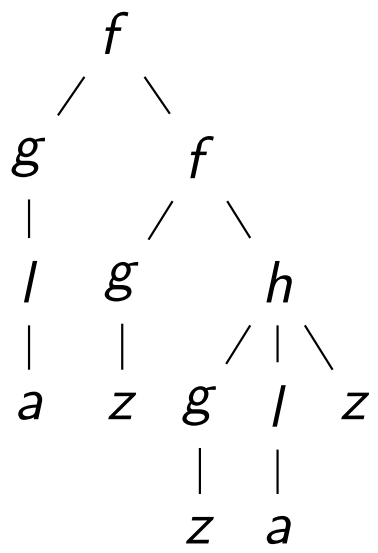
Dann erhalten wir die Substitution $S_2 = \{y \mapsto g(z)\}$ und wenden diese an.

Beispiel: Unifikation nach zweiter Substitution

$S_2(S_1(t)) =$



$S_2(S_1(u)) =$

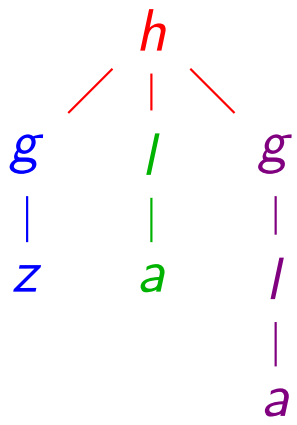


Beispiel: Unifikation 2.2. Parameter

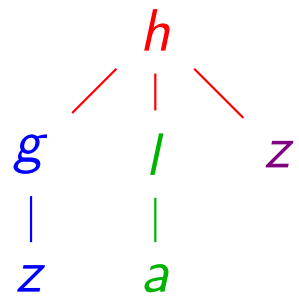
Fallunterscheidung: Ist einer der beiden Terme eine Variable? Nein (Fall 2).

Stimmen Funktionssymbole und Anzahl der Parameter überein? Ja.

$S_2(S_1(t_{2,2})) =$



$S_2(S_1(u_{2,2})) =$



Dann müssen die jeweiligen (ersten, zweiten und dritten) Parameter unifiziert werden.
Die ersten und die zweiten Parameter liefern nur triviale Substitutionen.

Beispiel: Unifikation: 2.2.3. Parameter

Fallunterscheidung: Ist einer der beiden Terme eine Variable? Ja (Fall 1).

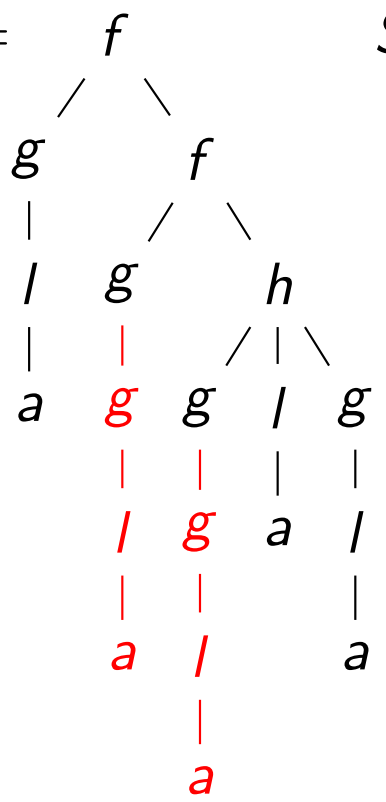
Kommt die Variable im anderen Term vor (*occur check*)? Nein.

$S_2(S_1(t_{2,2,3})) =$ $g(I(a))$ $\quad$ $S_2(S_1(u_{2,2,3})) =$ z

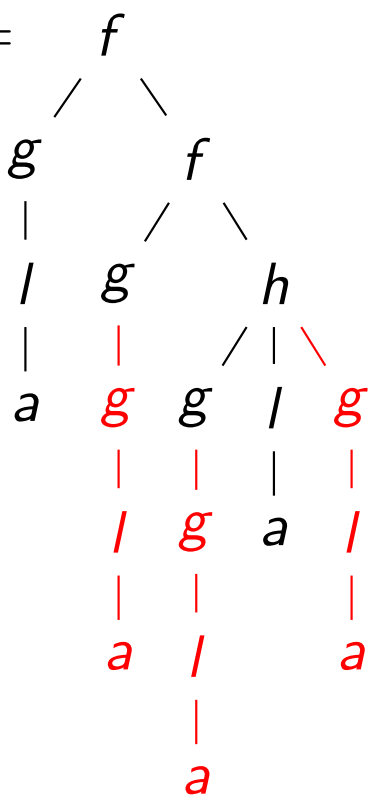
Wir erhalten die Substitution $S_3 = \{z \mapsto g(I(a))\}$ und wenden diese an.

Beispiel: Unifikation nach dritter Substitution

$S_3(S_2(S_1(t))) =$



$S_3(S_2(S_1(u))) =$



Das Unifikationsergebnis lautet daher:

```
f(g(l(a)), f(g(g(l(a))), h(g(g(l(a))), l(a), g(l(a)))))
```

Um diesen Unifikationsprozess formaler darstellen zu können, wird die Unifikationsaufgabe entsprechend dem Algorithmus `unify` in Teilaufgaben zerlegt.

- Man schreibt die zu unifizierenden Terme untereinander und vergleicht die Terme von links nach rechts.
- Stößt man bei dem Vergleich auf einen Unterschied, so betrachtet man die unterschiedlichen Teilterme.
- Lassen sich diese Teilterme unifizieren, so notiert man die gefundene Substitution und wendet diese auf die Terme an und fährt mit dem Vergleich fort bis man, sofern möglich, die Terme unifiziert hat.
- Die bis dahin bestimmte Substitution ist der gesuchte Unifikator.

Beispiel

Die zu unifizierenden Terme t und u sind

$$
\begin{array}{rcl}
t & = & \texttt{f(g(x), f(y, h(y, x, g(x))))} \\
u & = & \texttt{f(g(l(a)), f(g(z), h(g(z), l(a), z)))}
\end{array}
$$

$$S_1 = \{x \mapsto l(a)\}$$

$$
\begin{array}{rcl}
S_1(t) & = & \texttt{f(g(l(a)), f(y, h(y, l(a), g(l(a)))))} \\
S_1(u) & = & \texttt{f(g(l(a)), f(g(z), h(g(z), l(a), z)))}
\end{array}
$$

$$S_2 = \{y \mapsto g(z)\}$$

$$
\begin{array}{rcl}
S_2(S_1(t)) & = & \texttt{f(g(l(a)), f(g(z), h(g(z), l(a), g(l(a)))))} \\
S_2(S_1(u)) & = & \texttt{f(g(l(a)), f(g(z), h(g(z), l(a), z)))}
\end{array}
$$

$$S_3 = \{z \mapsto g(l(a))\}$$

Beispiel

$$
\begin{array}{rcl}
S_2(S_1(t)) & = & \texttt{f(g(l(a)), f(g(z), h(g(z), l(a), g(l(a)))))} \\
S_2(S_1(u)) & = & \texttt{f(g(l(a)), f(g(z), h(g(z), l(a), z)))} \\
\hline
S_3 & = & \{z \mapsto g(l(a))\} \\
\hline\hline
S_3(S_2(S_1(t))) & = & \texttt{f(g(l(a)), f(g(g(l(a))), h(g(g(l(a))), l(a), g(l(a)))))} \\
S_3(S_2(S_1(u))) & = & \texttt{f(g(l(a)), f(g(g(l(a))), h(g(g(l(a))), l(a), g(l(a)))))}
\end{array}
$$

Der allgemeinste Unifikator berechnet sich damit zu:

$$
\begin{array}{rcl}
S & = & S_3 \circ S_2 \circ S_1 \\
& = & \{z \mapsto g(l(a))\} \circ \{y \mapsto g(z)\} \circ \{x \mapsto l(a)\} \\
& = & \{z \mapsto g(l(a))\} \circ \{x \mapsto l(a),\ y \mapsto g(z)\} \\
& = & \{x \mapsto l(a),\ y \mapsto g(g(l(a))),\ z \mapsto g(l(a))\}
\end{array}
$$

Anwendung in der Programmiersprache Haskell

In Haskell muss der Programmierer den Typ seiner Funktionen nicht selbst deklarieren, sondern ein ausgefeiltes Typ-System berechnet automatisch den Typ eines eingegebenen Ausdrucks.

Beispiel

```
ac(oper,init,seq) = if null(seq) then init
                    else oper(head(seq), ac(oper,init,tail(seq)))
```

so berechnet das System den Typ als:

$$ac :: ((t, t1) \to t1, t1, [t]) \to t1$$

wobei der Typ-Ausdruck in unserer Notation so aussieht:

$$(\alpha \times \beta \to \beta) \times \beta \times list(\alpha) \to \beta$$

Typ-Ausdrücke für Standard-Funktionen

Der Typ-Ausdruck jeder Standard-Funktion ist dem Typ-System bekannt, z.B. hat die

- Funktion `null` den Typ-Ausdruck $\mathit{list}(\alpha) \to \mathit{bool}$,
- Funktion `head` den Typ-Ausdruck $\mathit{list}(\alpha) \to \alpha$.
- Funktion `tail` den Typ-Ausdruck $\mathit{list}(\alpha) \to \mathit{list}(\alpha)$.

Daraus schließt das Typ-System, welcher Typ-Ausdruck der Funktion `ac` zuzuordnen ist.

Bei jeder späteren Anwendung der Funktion `ac` kann dann der korrekte Aufruf der Funktion überprüft werden.

Typ-Ausdrücke für Standard-Funktionen

Beispiel

Typ-Inferenz für die Haskell-Funktion

```
length(ls) = if null(ls) then 0
             else length(tail(ls)) + 1
```

Definiert wird eine Funktion `length` mit einem Parameter `ls`.

Das Prädikat `null` testet, ob eine Liste leer ist.
Die Funktion `tail` liefert eine Liste ohne sein erstes Listenelement zurück.

Die Funktion `length` berechnet also die Länge einer Liste.
Ihr wird daher der Typ-Ausdruck $\mathit{list}(\alpha) \rightarrow \mathit{int}$ zugeordnet.

Bestimmung des Typ-Ausdrucks

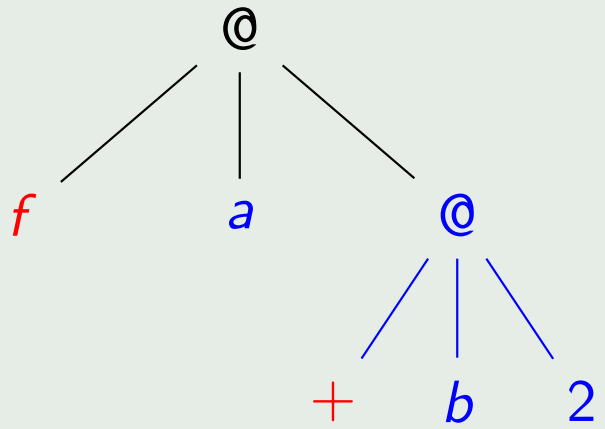
Wie ermittelt das Typ-System den Typ-Ausdruck?

Zunächst wird der Ausdruck geparst und ein Syntaxbaum aufgebaut.

Jede **Anwendung einer Funktion** wird im Syntaxbaum durch einen speziellen Knoten „@“ dargestellt.
Das erste Kind dieses Knotens stellt die Funktion,
jedes weitere Kind einen Parameter der Funktion dar.

Beispiel

Syntaxbaum für den Haskell-Ausdruck `f(a,b+2)`:



Berechnung des Typ-Ausdrucks

1. Allen Blättern des Syntaxbaums werden Typ-Ausdrücke zugeordnet:
   - (Vor-)Definierte Namen erhalten ihren bekannten Typ-Ausdruck.
   - Jedem neuen Namen wird eine neue Typ-Variable zugeordnet.
2. Dann versucht das Typ-System „bottom up“, allen internen @-Knoten des Syntaxbaums ebenfalls einen Typ-Ausdruck zuzuordnen:
   - Sei u der Typ-Ausdruck des ersten Kindes (der Funktion) eines @-Knotens.
   - Bilde aus den Typ-Ausdrücken $\beta_1 \ldots \beta_k$ der restlichen k Kinder (der Parameter) des @-Knotens sowie dem Typ-Ausdruck γ für das Funktionsergebnis einen neuen Typ-Ausdruck $v = \beta_1 \times \beta_2 \ldots \times \beta_k \to \gamma$.
   - Unifiziere u und v und berücksichtige die dafür notwendigen Substitutionen bei allen nachfolgenden Unifikationen für diesen Syntaxbaum.
   - Ordne dem @-Knoten den Typ-Ausdruck für γ zu.

Beispiel der Typ-Inferenz in Haskell

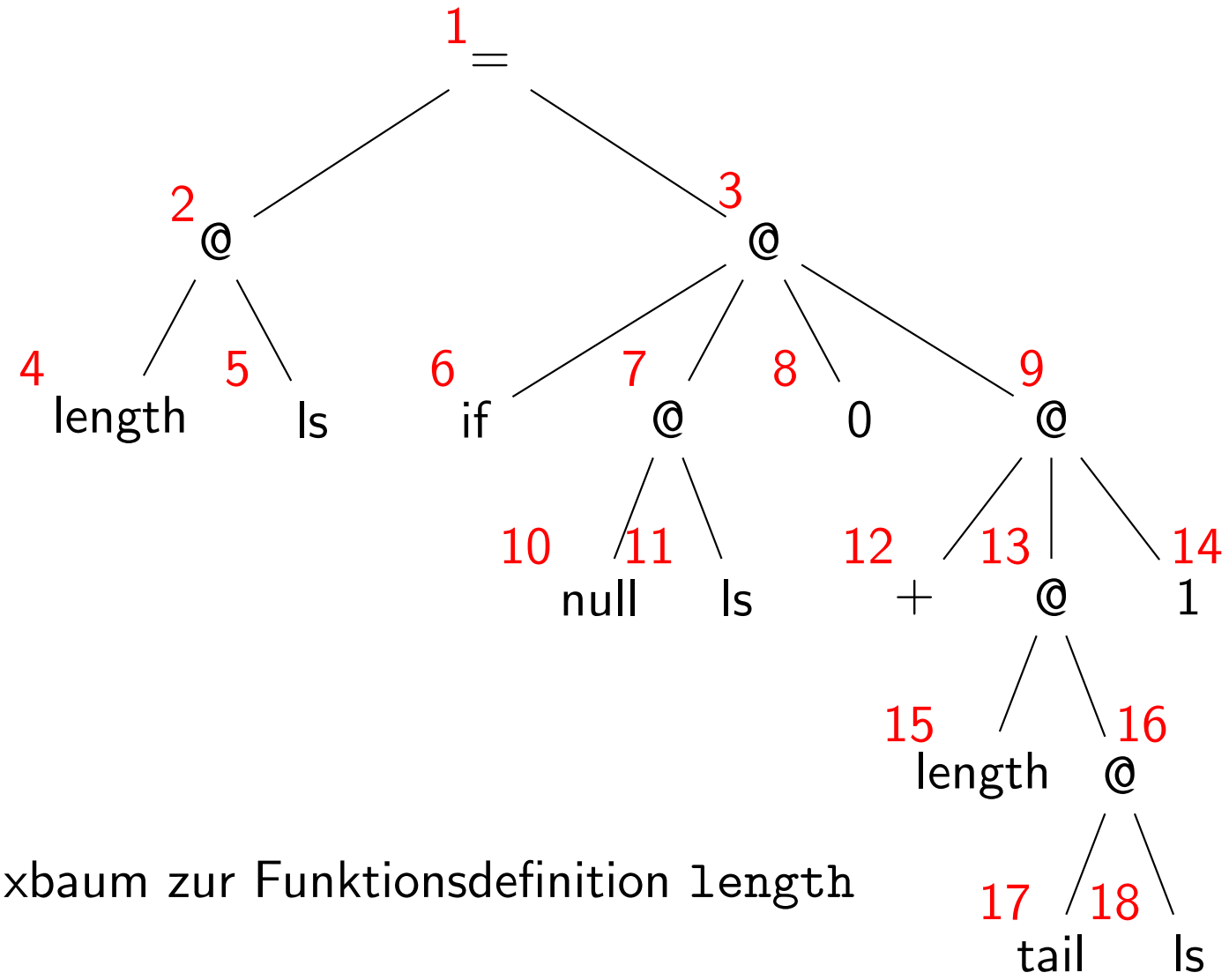
Gegeben sei:

```
length(ls) = if null(ls) then 0
             else length(tail(ls)) + 1
```

Für folgende Funktionen sind die Typ-Ausdrücke vorgegeben:

$$
\begin{aligned}
0 &: \mathit{int} \\
1 &: \mathit{int} \\
+ &: \mathit{int} \times \mathit{int} \to \mathit{int} \\
\texttt{if} &: \mathit{bool} \times \alpha \times \alpha \to \alpha \\
\texttt{null} &: \mathit{list}(\alpha) \to \mathit{bool} \\
\texttt{tail} &: \mathit{list}(\alpha) \to \mathit{list}(\alpha)
\end{aligned}
$$

Zugehöriger Syntaxbaum mit Typ-Ausdrücken



Syntaxbaum zur Funktionsdefinition `length`

Beispiel: Unifikation zur Typbestimmung: Blätter

Betrachte Blätter:
Vordefinierte Typ-Ausdrücke eintragen:

Nr	Sprach-Ausdruck	Typ-Ausdruck
4	`length`	
5	`ls`	
6	`if`	$\texttt{bool} \times \alpha_i \times \alpha_i \to \alpha_i$
8	`0`	`int`
10	`null`	$\texttt{list}(\alpha_n) \to \texttt{bool}$
11	`ls`	
12	`+`	$\texttt{int} \times \texttt{int} \to \texttt{int}$
14	`1`	`int`
15	`length`	
17	`tail`	$\texttt{list}(\alpha_t) \to \texttt{list}(\alpha_t)$
18	`ls`	

Beispiel: Unifikation zur Typbestimmung: Blätter

Betrachte Blätter:
Für fehlende Typ-Ausdrücke Typ-Variablen eintragen:

Nr	Sprach-Ausdruck	Typ-Ausdruck
4	`length`	β
5	`ls`	γ
6	`if`	$\texttt{bool} \times \alpha_i \times \alpha_i \to \alpha_i$
8	`0`	`int`
10	`null`	$\texttt{list}(\alpha_n) \to \texttt{bool}$
11	`ls`	γ
12	`+`	$\texttt{int} \times \texttt{int} \to \texttt{int}$
14	`1`	`int`
15	`length`	β
17	`tail`	$\texttt{list}(\alpha_t) \to \texttt{list}(\alpha_t)$
18	`ls`	γ

Beispiel: Unifikation zur Typbestimmung: @-Knoten

Betrachte innere Knoten bottom-up:
Ggf. neue Typ-Variable für das Funktionsergebnis.

Nr	Sprach-Ausdruck	Typ-Ausdruck	zu unifizieren	Substitutionen
4	`length`	β	$u = \beta$	
5	`ls`	γ		
2	`length(ls)`	δ	$v = \gamma \to \delta$	$\beta \mapsto \gamma \to \delta$

v entsteht als Funktion vom kartesischen Produkt der Parametertypen zum Ergebnistyp.

Alle Substitutionen müssen auf nachfolgende Typ-Ausdrücke angewendet werden!

Beispiel: Unifikation zur Typbestimmung: @-Knoten

Betrachte innere Knoten bottom-up:

Evtl. steht der Typ für das Funktionsergebnis schon fest!

Nr	Sprach-Ausdruck	Typ-Ausdruck	zu unifizieren	Substitutionen
4	`length`	β	$u = \beta$	
5	`ls`	γ		
2	`length(ls)`	δ	$v = \gamma \to \delta$	$\beta \mapsto \gamma \to \delta$
10	`null`	$\texttt{list}(\alpha_n) \to \texttt{bool}$	$u = \texttt{list}(\alpha_n) \to \texttt{bool}$	
11	`ls`	γ		
7	`null(ls)`	`bool`	$v = \gamma \to \texttt{bool}$	$\gamma \mapsto \texttt{list}(\alpha_n)$

Beispiel: Unifikation zur Typbestimmung: @-Knoten

Betrachte innere Knoten bottom-up:

Substitutionen berücksichtigen!

Nr	Sprach-Ausdruck	Typ-Ausdruck	zu unifizieren	Substitutionen
4	`length`	β	$u = \beta$	
5	`ls`	γ		
2	`length(ls)`	δ	$v = \gamma \to \delta$	$\beta \mapsto \gamma \to \delta$
10	`null`	$\texttt{list}(\alpha_n) \to \texttt{bool}$	$u = \texttt{list}(\alpha_n) \to \texttt{bool}$	
11	`ls`	γ		
7	`null(ls)`	`bool`	$v = \gamma \to \texttt{bool}$	$\gamma \mapsto \texttt{list}(\alpha_n)$
17	`tail`	$\texttt{list}(\alpha_t) \to \texttt{list}(\alpha_t)$	$u = \texttt{list}(\alpha_t) \to \texttt{list}(\alpha_t)$	
18	`ls`	$\gamma = \texttt{list}(\alpha_n)$		
16	`tail(ls)`	$\texttt{list}(\alpha_t)$	$v = \texttt{list}(\alpha_n) \to \texttt{list}(\alpha_t)$	$\alpha_t \mapsto \alpha_n$

Beispiel: Unifikation zur Typbestimmung: @-Knoten

Betrachte innere Knoten bottom-up:

Manchmal liefert die Unifikation keine oder mehrere Substitutionen.

Nr	Sprach-Ausdruck	Typ-Ausdruck	zu unifizieren	Substitutionen
4	`length`	β	$u = \beta$	
5	`ls`	γ		
2	`length(ls)`	δ	$v = \gamma \to \delta$	$\beta \mapsto \gamma \to \delta$
10	`null`	$\texttt{list}(\alpha_n) \to \texttt{bool}$	$u = \texttt{list}(\alpha_n) \to \texttt{bool}$	
11	`ls`	γ		
7	`null(ls)`	`bool`	$v = \gamma \to \texttt{bool}$	$\gamma \mapsto \texttt{list}(\alpha_n)$
17	`tail`	$\texttt{list}(\alpha_t) \to \texttt{list}(\alpha_t)$	$u = \texttt{list}(\alpha_t) \to \texttt{list}(\alpha_t)$	
18	`ls`	$\gamma = \texttt{list}(\alpha_n)$		
16	`tail(ls)`	$\texttt{list}(\alpha_t)$	$v = \texttt{list}(\alpha_n) \to \texttt{list}(\alpha_t)$	$\alpha_t \mapsto \alpha_n$
15	`length`	$\beta = \texttt{list}(\alpha_n) \to \delta$	$u = \texttt{list}(\alpha_n) \to \delta$	
16	`tail(ls)`	$\texttt{list}(\alpha_t) = \texttt{list}(\alpha_n)$		
13	`length(tail(ls))`	δ	$v = \texttt{list}(\alpha_n) \to \delta$	–

Beispiel: Unifikation zur Typbestimmung: @-Knoten

Nr	Sprach-Ausdruck	Typ-Ausdruck	zu unifizieren	Substitutionen
4	`length`	β	$u = \beta$	
5	`ls`	γ		
2	`length(ls)`	δ	$v = \gamma \to \delta$	$\beta \mapsto \gamma \to \delta$
10	`null`	$\texttt{list}(\alpha_n) \to \texttt{bool}$	$u = \texttt{list}(\alpha_n) \to \texttt{bool}$	
11	`ls`	γ		
7	`null(ls)`	`bool`	$v = \gamma \to \texttt{bool}$	$\gamma \mapsto \texttt{list}(\alpha_n)$
17	`tail`	$\texttt{list}(\alpha_t) \to \texttt{list}(\alpha_t)$	$u = \texttt{list}(\alpha_t) \to \texttt{list}(\alpha_t)$	
18	`ls`	$\gamma = \texttt{list}(\alpha_n)$		
16	`tail(ls)`	$\texttt{list}(\alpha_t)$	$v = \texttt{list}(\alpha_n) \to \texttt{list}(\alpha_t)$	$\alpha_t \mapsto \alpha_n$
15	`length`	$\beta = \texttt{list}(\alpha_n) \to \delta$	$u = \texttt{list}(\alpha_n) \to \delta$	
16	`tail(ls)`	$\texttt{list}(\alpha_t) = \texttt{list}(\alpha_n)$		
13	`length(tail(ls))`	δ	$v = \texttt{list}(\alpha_n) \to \delta$	–
12	`+`	$\texttt{int} \times \texttt{int} \to \texttt{int}$	$u = \texttt{int} \times \texttt{int} \to \texttt{int}$	
13	`length(tail(ls))`	δ		
14	`1`	`int`		
9	`length(tail(ls)) + 1`	`int`	$v = \delta \times \texttt{int} \to \texttt{int}$	$\delta \mapsto \texttt{int}$
6	`if`	$\texttt{bool} \times \alpha_i \times \alpha_i \to \alpha_i$	$u = \texttt{bool} \times \alpha_i \times \alpha_i \to \alpha_i$	
7	`null(ls)`	`bool`		
8	`0`	`int`		
9	`length(tail(ls)) + 1`	`int`		
3	`if(...)`	α_i	$v = \texttt{bool} \times \texttt{int} \times \texttt{int} \to \alpha_i$	$\alpha_i \mapsto \texttt{int}$
2	`length(ls)`	$\delta = \texttt{int}$		
3	`if(...)`	$\alpha_i = \texttt{int}$		
1	`=`	`True`		

Die Funktion `length` hat den Typ-Ausdruck: $\beta = \texttt{list}(\alpha_n) \to \texttt{int}$

Teil VII

Programmierparadigmen: Funktionale Programmierung

Funktionale Programmierung

Funktionale Programmierung arbeitet mit **Ausdrücken** und **mathematischen Funktionen**.

Ihre Einfachheit ermöglicht ein besseres Verständnis eines Programms und der Programmiersprache.

Funktionale Programme werden meist schneller entwickelt, sind kürzer sowie leichter zu warten.

Funktionen und Variablen – ohne Nebenwirkungen

Mathematische Funktionen liefern für gleiche Argumente –unabhängig vom Zeitpunkt der Auswertung– stets den gleichen Funktionswert (**referentielle Transparenz**), haben also keine Nebenwirkungen.

Das vereinfacht viele Programmtransformationen,
z.B. $f(x) + f(x) = 2 \cdot f(x)$, und
erleichtert die Überprüfung der Korrektheit eines Programms.

Die Bedeutung eines (rein) funktionalen Programms hängt daher nicht von der Reihenfolge ihrer Definitionen ab.
Funktionale Sprachen können daher eine beliebige Reihenfolge erlauben.

Der Begriff **Variable** wird hier mathematisch benutzt,
also anders als bei imperativen Sprachen, wo eine Variable für einen Behälter steht, der durch eine Wertzuweisung neue Werte aufnimmt.

einfache Parallelisierung

Funktionale Programme lassen sich –da ohne Nebenwirkungen– einfach **parallelisieren**:

- Alle Argumente einer Funktion können parallel ausgewertet werden.
- Bei einer Bedarfsauswertung kann sogar die Berechnung der Argumente einer Funktion parallel zur Berechnung des Funktionsrumpfes erfolgen.

Die Möglichkeiten zur Parallelisierung sind also schnell erkannt, die anderen Probleme einer effizienten parallelen Implementierung (z.B. gute Granularität, Lokalität, Prozessorauslastung) aber ähnlich schwierig wie bei imperativen Sprachen.

Ändern von Datenstrukturen

Funktionale Sprachen betrachten eine **Datenstruktur als Ganzes**.

Imperative Sprachen erlauben **selektive Änderungen**, z.B. das Überschreiben einer Feldkomponente oder das Umsetzen eines Zeigers:

```
A[i] := x          oder          p^.next := q
```

Dies begünstigt Fehler: z.B. vorzeitiges Überschreiben von Feldelementen, Kappen noch benötigter Zeiger.

In funktionalen Sprachen werden dagegen neue Datenstrukturen gebildet:

```
let A' = update(A,i,x)    oder    let L' = q:L
```

Der Speicherverbrauch bleibt durch automatische Speicherbereinigung in Grenzen, aber der Zeitaufwand für das Kopieren bleibt – als Preis für die geringere Fehleranfälligkeit.

Eine gute Optimierung erkennt daher, ob eine Datenstruktur nur einmal benötigt wird und deshalb selektiv geändert werden darf.

Rekursive Datenstrukturen

Funktionale Programmiersprachen bieten **rekursiv definierte Typen** an und verweisen **Zeiger** damit auf die Ebene der Sprach-Implementierung.

Während in imperativen Sprachen die Datenstruktur **Feld** häufig ist, benutzt man in funktionalen Sprachen oft den rekursiven Datentyp **Liste**.

Listen haben zwei Datenkonstruktoren:

- `[ ]` erzeugt die leere Liste und
- `:` verlängert eine Liste um ein Element, und zwar am Anfang.

`[1,2,3]` beschreibt dann kurz die Liste `1:(2:(3:[ ]))`.

Automatische Speicherverwaltung

In funktionalen Sprachen muss sich der Programmierer nicht um die Anforderung und Freigabe von Speicherplatz kümmern;
die Implementierung der Sprache übernimmt die Speicherverwaltung.

Da Speicherplatz

- für jede Anwendung eines k-stelligen Konstruktors mit $k>0$ und
- jeden verzögerten Ausdruck

angefordert wird, ist eine gute Speicherverwaltung der Halde Voraussetzung für eine effiziente funktionale Sprache.

Muster in Funktionsdefinitionen

Der **Mustervergleich** ist eine kompakte Variante der Fallunterscheidung.
Formale Parameter werden durch Muster ersetzt.
Ein **Muster** ist ein Ausdruck mit Variablen und Datenkonstruktoren.

Die aktuellen Parameter werden dann mit diesen Mustern verglichen.

Passt der Wert jedes aktuellen Parameters auf sein Muster, so werden die Variablen im Muster an die entsprechenden Komponenten der aktuellen Parameter gebunden.

Beispiel

Wertet man den Ausdruck `f` $[2,3]$ aus bzgl. der Definition

$$\texttt{f}\ [1,x,y] = x+y$$
$$\texttt{f}\ [2,z] \quad = z+7\ ,$$

so passt der Wert $[2,3]$ auf das Muster $[2,z]$ der zweiten Alternative; daher wird die Variable z an die Komponente 3 gebunden und das Funktionsergebnis ist 10.

Funktionale: Funktionen höherer Stufe

Viele funktionale Programmiersprachen erlauben **Funktionale** (Funktionen höherer Stufe).
Sprachen werden danach unterteilt in **Sprachen erster Stufe** und **Sprachen höherer Stufe**.

Sprachen höherer Stufe betrachten Funktionen als Werte und damit als **gleichberechtigte Datenobjekte** (*first class citizens*).
Dann können Funktionen auch als Ergebnis eines Ausdrucks oder einer Funktion, als Parameter oder als Teil einer Datenstruktur auftreten.

Beispiel

Die **Komposition** zweier Funktionen ($f \;.\; g$) hat zwei Funktionen f, g als Parameter und eine Funktion als Ergebnis.

Funktionale: Funktionen höherer Stufe

Mit höheren Funktionen (wie `map`, `filter`, `zip`) kann man häufige Rekursionsmuster beschreiben.

Daher werden diese –z.T. vordefinierten– Funktionen oft verwendet, was zu besserer Lesbarkeit von Programmen und guter Modularisierung führt.

Als Parameter eines Funktionals müssen Funktionen nicht unbedingt einen Namen bekommen.
Man erzeugt dann eine anonyme Funktion durch einen Lambda-Ausdruck.

Bedarfsauswertung

Auch nach der **Auswertungsstrategie** können funktionale Sprachen eingeteilt werden:

- Bei **strikter Auswertung** (*strict evaluation*, *call by value*) wird jedes Argument einer Funktion ausgewertet, bevor die Funktion aufgerufen wird – wie oft in imperativen Sprachen.
- Bei **Bedarfsauswertung** (*lazy evaluation*, *call by need*) wird jedes Argument der Funktion unausgewertet als Ausdruck übergeben und erst berechnet, wenn der Wert benötigt wird.

Strikte Auswertung braucht weniger Speicherplatz und ist leichter mit imperativen Erweiterungen kombinierbar.

Die Bedarfsauswertung ermöglicht **zyklische** oder **unendliche Datenstrukturen**.

Typisierung

Schließlich unterscheidet man funktionale Sprachen nach ihrer **Typisierung**.

Alle verbreiteten funktionalen Sprachen sind stark typisiert, d.h. alle Typfehler werden erkannt.

Einige Sprachen überprüfen Typen aber erst zur Laufzeit (**dynamische Typprüfung**);

andere Sprachen erledigen die vollständige Typprüfung bereits zur Übersetzungszeit (**statische Typprüfung**).

Typisierung

Typen werden in funktionalen Sprachen als Terme dargestellt.
Basistypen sind atomare Terme, wie `Int`, `Bool`.
Strukturierte Typen werden durch Typterme beschrieben, z.B. eine Liste von `Int`-Werten durch `[Int]`.

Der Typ eines Ausdrucks kann auch durch ein Typschema mit Typvariablen gegeben sein.
Eine **Typvariable** kann bei verschiedenen Benutzungen mit unterschiedlichen Typen instantiiert werden.
Ein Typ(schema) kann durch Deklaration angegeben oder durch **Typinferenz** aus den Anwendungsstellen der Funktion abgeleitet werden.

In Haskell brauchen Typen nicht deklariert zu werden.
Dann schließt das System aus dem Gebrauch der Variablen auf mögliche Typen und prüft, ob alle Vorkommen dieser Variablen konsistent sind.

Vergleich funktionaler Sprachen

		Strikte Auswertung	**Bedarfsauswertung**
Dynamische Typprüfung	**Erste Stufe**	Mathematica, Erlang	
	Höhere Stufe	Lisp, Scheme, APL, FP	SASL
Statische Typprüfung	**Erste Stufe**	SISAL	Id
	Höhere Stufe	ML, SML, Caml, Hope	Miranda, Haskell

Klassifikation funktionaler Programmiersprachen

Einige funktionale Sprachen orientieren sich an bestimmten Anwendungen:

- ML, SML und Hope werden für Theorembeweiser eingesetzt,
- Erlang für verteilte Telekommunikationsanwendungen (Ericsson);
- SISAL und Id für paralleles Hochleistungsrechnen.
- Mathematica ist Teil eines Systems für symbolische Algebra.
- XSLT beschreibt und transformiert XML-Daten.

Haskell-Compiler

Beispielsprache für diesen Abschnitt ist Haskell 2010
`www.haskell.org/onlinereport/haskell2010/`

Der **Glasgow Haskell Compiler** (`GHC`) kann unter
`www.haskell.org/platform` heruntergeladen werden.
Er enthält u.a. den interaktiven Interpretierer `GHCi`.

Wichtige Kommandos sind:
`<expression>` wertet einen einzeiligen Ausdruck aus

`:?` liefert die Liste der Kommandos

`:info <name>` informiert über das Objekt `name`

`:load <filename>` lädt den angegebenen Modul,
Haskell-Dateien enden auf `.hs`

`:reload` lädt den (geänderten) Modul erneut

`:quit` beendet den Lauf