Fortgeschrittene Programmierung
Vorlesung
WS 09,10; SS 12–14, 16–19, 21–26

Johannes Waldmann, HTWK Leipzig

16. April 2026

– Typeset by FoilTEX –

Einleitung

Programmierung im Studium bisher

- 1. Sem: Modellierung (formale Spezifikationen (von konkreten und abstrakten Datentypen))
- 1./2. Sem Grundlagen der (AO) Programmierung
  - imperatives Progr. (Programm ist Folge von Anweisungen, bewirkt Zustandsänderung)
  - strukturiertes P. (genau ein Eingang/Ausgang je Teilp.)
  - objektorientiertes P. (Interface $=$ abstrakter Datentyp, Klasse $=$ konkreter Datentyp)
- 2. Sem: Algorithmen und Datenstrukturen (Spezifikation, Implementierung, Korrektheit, Komplexität)
- 3. Sem: Softwaretechnik (industrielle Softwareproduktion)

Worin besteht jetzt der Fortschritt?

- *deklarative* Programmierung

  (Programm *ist* ausführbare Spezifikation)

- insbesondere: *funktionale* Programmierung

  Def: Programm berechnet *Funktion*
  f : Eingabe $\mapsto$ Ausgabe,

  (kein Zustand, keine Zustandsänderungen)

- – Daten (erster Ordnung) sind Bäume
  – Programm ist Gleichungssystem
  – Programme sind auch Daten (höherer Ordnung)

- ausdrucksstark, sicher, effizient, parallelisierbar

Formen der deklarativen Programmierung

- funktionale Programmierung: `foldr (+) 0 [1,2,3]`

```
foldr f z l = case l of
    [] -> z ; (x:xs) -> f x (foldr f z xs)
```

- logische Programmierung: `append(A,B,[1,2,3]).`

```
append([],YS,YS).
append([X|XS],YS,[X|ZS]):-append(XS,YS,ZS).
```

- Constraint-Programmierung

```
(set-logic QF_LIA) (set-option :produce-models true
(declare-fun a () Int) (declare-fun b () Int)
(assert (and (>= a 5) (<= b 30) (= (+ a b) 20)))
(check-sat) (get-value (a b))
```

Definition: Funktionale Programmierung

- Rechnen $=$ Auswerten von Ausdrücken (Termbäumen)
- Dabei wird ein *Wert* bestimmt

  und es gibt keine (versteckte) *Wirkung*.

  (engl.: side effect, dt.: Nebenwirkung)
- Werte können sein:
  - "klassische" Daten (Zahlen, Listen, Bäume...)

    ```
    True :: Bool, [3.5, 4.5] :: [Double]
    ```
  - Funktionen (Sinus, ...)

    ```
    [sin, cos] :: [Double -> Double]
    ```
  - Aktionen (Datei lesen, schreiben, ...)

    ```
    readFile "foo.text" :: IO String
    ```

Softwaretechnische Vorteile

. . . der funktionalen Programmierung

- Beweisbarkeit: Rechnen mit Programmen
  wie in der Mathematik mit Termen

- Sicherheit: es gibt keine Nebenwirkungen
  und Wirkungen sieht man bereits am Typ

- Aussdrucksstärke, Wiederverwendbarkeit: durch
  Funktionen höherer Ordnung (sog. Entwurfsmuster)

- Effizienz: durch Programmtransformationen im Compiler,

- Parallelität: keine Nebenwirkungen $\Rightarrow$ keine *data races*,
  fktl. Programme sind *automatisch parallelisierbar*

Beispiel Spezifikation/Test

```
import Test.LeanCheck

append :: forall t . [t] -> [t] -> [t]
append [] y = y
append (h : t) y = h : (append t y)

associative f =
   \ x y z -> f x (f y z) == f (f x y) z
commutative f = \ x y -> ...

test = check (associative (append @Bool))
```

Übung: Kommutativität (formulieren und testen)

Beispiel Verifikation

```
app :: forall t . [t] -> [t] -> [t]
app [] y = y
app (h : t) y = h : (app t y)

Lemma: app x (app y z) .=. app (app x y) z

Proof by induction on List x
  Case []
    To show: app [] (app y z) .=. app (app [] y) z
  Case h:t
    To show: app (h:t) (app y z) .=. app (app (h:t) y
    IH: app t (app y z) .=. app (app t y) z
```

CYP `https://github.com/noschinl/cyp`,

ist vereinfachte Version

von Isabelle `https://isabelle.in.tum.de/`

Beispiel Parallelisierung (Haskell)

```
-- Länge der Collatz-Folge
collatz :: Int -> Int
collatz x = if x <= 1 then 0
  else 1 + collatz (if even x then div x 2 else 3*x+1
-- Summe der Längen
main :: IO ()
main = print $ sum
    $ map collatz [1 .. 10^7]
```

wird parallelisiert durch *Strategie-Annotation*:

```
import Control.Parallel.Strategies
...
main = print $ sum
  $ withStrategy (parListChunk (10^5) rseq)
  $ map collatz [1 .. 10^7]
```

Beispiel Parallelisierung (C#, PLINQ)

- Die Anzahl der 1-Bits einer nichtnegativen Zahl:

```
Func<int,int>f =
    x=>{int s=0; while(x>0){s+=x%2;x/=2;}return s;}
```

- $\sum_{x=0}^{2^{26}-1} f(x)$ `Enumerable.Range(0,1<<26).Select(f).Sum()`
- automatische parallele Auswertung, Laufzeitvergleich:

```
Time(()=>Enumerable.Range(0,1<<26).Select(f).Sum())
Time(()=>Enumerable.Range(0,1<<26)
            .AsParallel().WithDegreeOfParallelism(4)
                                    .Select(f).Sum())
```

vgl. *Introduction to PLINQ* `https://msdn.microsoft.com/en-us/library/dd997425(v=vs.110).aspx`

Softwaretechnische Vorteile

. . . der statischen Typisierung

> The language in which you write profoundly affects the design of programs written in that language.
>
> For example, in the OO world, many people use UML to sketch a design. In Haskell or ML, one writes type signatures instead. Much of the initial design phase of a functional program consists of writing type definitions.
>
> **Unlike UML, though, all this design is incorporated in the final product, and is machine-checked throughout.**

Simon Peyton Jones, in: Masterminds of Programing, 2009;

`http://shop.oreilly.com/product/9780596515171.do`

Deklarative Programmierung in der Lehre

- funktionale Programmierung: diese Vorlesung
- logische Programmierung: in *Grundl. Künstl. Intell.*
- Constraint-Programmierung: in Wpf *Formale Methoden und Werkzeuge* (folgendes Sem.)

Beziehungen zu weiteren LV: Voraussetzungen

- Bäume, Terme (Modellierung, Alg.+DS)
- Logik (Grundlagen TI, Softwaretechnik)

Anwendungen:

- Softwarepraktikum
- weitere Sprachkonzepte in *Prinzipien v. Programmiersprachen*

- *Programmverifikation*

Konzepte und Sprachen

Funktionale Programmierung ist ein *Konzept*.
Realisierungen:

- in prozeduralen Sprachen:
  - Unterprogramme als Argumente (in Pascal)
  - Funktionszeiger (in C)
- in OO-Sprachen: Befehlsobjekte
- Multi-Paradigmen-Sprachen:
  - Lambda-Ausdrücke in C#, Scala, Clojure
- funktionale Programmiersprachen (LISP, ML, Haskell)

Die Erkenntnisse sind sprachunabhängig.

- A good programmer can write LISP in any language.
- Learn Haskell and become a better Java programmer.

Gliederung der Vorlesung

- Terme, Termersetzungssysteme, algebraische Datentypen, Pattern Matching, Persistenz
- Funktionen (polymorph, höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
- Typklassen zur Steuerung der Polymorphie

  (Anwendung: automatische Testdatenerzeugung)
- Bedarfsauswertung, unendl. Datenstrukturen
- Konstruktorklassen (Functor, Applicative, Monad)
- Collections (endliche Mengen, Abbildungen, Folgen)

  als Anwendung vorher gezeigter Konzepte

Anwendungen dieser Konzepte

- algebraische Datentypen, Pattern Matching, Termersetzungssysteme

  Scale: case class, Java: Entwurfsmuster Kompositum, immutable objects (`record`), das Datenmodell von Git
- Funktionen (höherer Ordnung), Lambda-Kalkül, Rekursionsmuster

  Lambda-Ausdrücke in C#, Entwurfsmuster Besucher

  Codequalität, code smells, Refaktorisierung
- Typklassen zur Steuerung der Polymorphie: Interfaces
- Bedarfsauswertung, unendl. Datenstrukturen

  Iteratoren, Ströme, LINQ
- Functor, Applicative, Monad: `map`, `flatMap`

Literatur (allgemein)

- wissenschaftliche Quellen zur aktuellen Forschung und Anwendung der funktionalen Programmierung
  - Journal of Functional Programming (CUP) `https://www.cambridge.org/core/journals/journal-of-functional-programming`
  - Intl. Conference Functional Programming (ACM SIGPLAN) `https://www.icfpconference.org/`
  - Intl. Workshop Trends in Functional Programming in Education `https://wiki.tfpie.science.ru.nl/`

- `https://haskell.org/` (Sprachstandard, Werkzeuge, Bibliotheken, Tutorials),

Literatur (speziell diese VL)

- Skript aktuelles Semester `https://www.imn.htwk-leipzig.de/~waldmann/lehre.html`
- How I Teach Functional Programming (WFLP 2017) `https://www.imn.htwk-leipzig.de/~waldmann/talk/17/wflp/`
- Kriterium für Haskell-Tutorials und -Lehrbücher:
  - wo werden `data` (benutzerdefinerte algebraische Datentypen) und `case` (pattern matching) erklärt?

    Je später, desto schlechter!

Alternative Quellen

- – Q: Aber in Wikipedia/Stackoverflow steht, daß . . .
  – A: Na und.
- Es mag eine in Einzelfällen nützliche Übung sein, sich mit dem Halbwissen von Nichtfachleuten auseinanderzusetzen. (Aber `https://xkcd.com/386/`)
- In VL und Übung verwenden und diskutieren wir die durch Dozenten/Skript/Modulbeschreibung vorgegebenen Quellen (Lehrbücher, referierte Original-Artikel, Standards zu Sprachen und Bibliotheken)
- . . . gilt entsprechend für Ihre Bachelor- und Master-Arbeit.
- Wikipedia: benutzen—ja (um Primärquellen zu finden), zitieren—nein (ist keine wissenschaftliche Quelle).

Organisation der LV

- jede Woche eine Vorlesung, eine Übung
- Hausaufgaben
  - gruppenweise: markierte Aufgaben aus dem Skript: anmelden (Wiki), diskutieren (Issue-Tracker), vorrechnen (in der jeweils nächsten Übung)
  - individuell (jeweils 2 Wochen Bearbeitungszeit)
    `https://autotool.imn.htwk-leipzig.de/new/`
- Prüfungszulassung: regelmäßiges und erfolgreiches Bearbeiten der Übungsaufgaben.
  - Vorrechnen: 3 mal,
  - Autotool: 50 Prozent der Pflicht-Aufgaben,
- Prüfung: Klausur 120 min (am Computer), keine Hilfsmittel

Übungen KW 15 (vor Vorlesung)

- Arbeiten im Pool: Shell, `$PATH`, `ghci`, vgl. `https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`

```
$ cabal update
#   $HOME/.config/cabal/config  editieren (-haddock
$ cabal install --lib leancheck
$ ghci
GHCi, version 9.14.1: https://www.haskell.org/ghc/
ghci> import Test.LeanCheck
ghci> check $ \ p q -> (p && q) == (q && p)
ghci> :doc check
```

- `ssh-keygen`, `.ssh/id_ed25519.pub` $\Rightarrow$ `gitlab.dit`, `git clone`
- wenn Zeit ist: autotool 15-1,

Übungen

- Informationen zur VL: `https://www.imn.htwk-leipzig.de/~waldmann/lehre.html`

- digitale Selbstverteidigung: Browser und Suchmaschine datenschutzgerecht auswählen und einstellen.

  Das Geschäftsmodell der Überwachungswirtschaft ist es, Ihren Bildschirmplatz, und damit Ihre Aufmerksamkeit und Ihre Lebenszeit an Anzeigenkunden zu verkaufen. Um dabei höhere Erlöse zu erzielen, wird Ihr Verhalten vermessen, gespeichert, vorhergesagt und beeinflußt. Die dazu angelegten Personenprofile erlauben eine umfassende privatwirtschaftliche und staatliche Überwachung. Diese soll verschleiert, verharmlost und

legalisiert werden.

Siehe auch

- OS Überwachungskapitalismus `https://www.imn.htwk-leipzig.de/~waldmann/talk/19/ubkap/`,
- VL Informatik (Nebenfach) `https://www.imn.htwk-leipzig.de/~waldmann/edu/ws21/inf/folien/#(11)`

- Benutzung Rechnerpool (ssh, tmux, ghci) `https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`

- Beispiel Funktionale Programmierung

```
$ /usr/local/waldmann/opt/ghc/latest/bin/gh
```

```
ghci> length $ takeWhile (== '0') $ reverse
```

- Typ und Wert von Teilausdrücken feststellen, z.B.

  ```
  ghci> :set +t
  ghci> foldr (*) 1 [1..100 :: Integer]
  ```

- Beachte polymorphe numerische Literale.
  (Auflösung der Polymorphie durch Typ-Annotation.)
  Warum ist 100 Fakultät als `Int` gleich 0?
- Welches ist der Typ der Funktion `takeWhile`? Beispiel:

  ```
  odd 3 ==> True ; odd 4 ==> False
  takeWhile odd [3,1,4,1,5,9] ==> [3,1]
  ```

- ersetze in der Lösung `takeWhile` durch andere Funktionen des gleichen Typs (suche diese mit Hoogle), erkläre Semantik

- typische Eigenschaften dieses Beispiels (nachmachen!)
  statische Typisierung, Schachtelung von Funktionsaufrufen, Funktion höherer Ordnung, Benutzung von Funktionen aus Standardbibliothek (anstatt selbstgeschriebener).
- schlechte Eigenschaften (vermeiden!)
  Benutzung von Zahlen und Listen (anstatt anwendungsspezifischer Datentypen) vgl. `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list/`

- Haskell-Entwicklungswerkzeuge

  - Compiler, REPL: ghci (Fehlermeldungen, Holes)
  - API-Suchmaschine `https://www.haskell.org/hoogle/`

  - Editor: Emacs `https://xkcd.com/378/`,
    IDE? gibt es, brauchen wir (in dieser VL) nicht
    `https://hackage.haskell.org/package/haskell-language-server`

- Softwaretechnik im autotool: `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/se/`

Aufgaben (allgemeines)

- benutzen Sie gitlab.dit zur Koordinierung: (einmalig) Einteilung in Dreiergruppen, (wöchentlich) Bearbeitung der Aufgaben. Benutzen Sie Wiki und Issues mit sinnvollen Titeln/Labeln. Schließen Sie erledigte Issues.

- Jede der markierten Aufgabe kann in jeder Übung aufgerufen werden (Bsp: Aufg. 3 in den INB-Übungen und in der MIB-Übung) Es kann dann eine vorher gemeinsam (von mehreren Gruppen) vorbereitete Lösung präsentiert werden—die aber von jedem einzelnen Präsentator auch verstanden sein sollte.

- Auch die nicht markierten Aufgaben können in den Übungen diskutiert werden—wenn dafür Zeit ist.

Aufgaben

SS 26: Aufgabe 1, 4, 5

1. Digitale Selbstverteidigung

(a) Welche Daten gibt Ihr Browser preis?
Starten Sie in einer Konsole den Befehl
`nc -l -p 9999`
(Konsole soll sichtbar bleiben)
Rufen Sie im Browser die Adresse
`http://localhost:9999` auf, beobachten Sie die Ausgabe in der Konsole.
Wie (personen)spezifisch ist diese Information?

(b) Wie können weitere Informationen extrahiert werden?
Verwenden Sie
`https://www.eff.org/press/releases/`

`test-your-online-privacy-protection-effs-p`
(Electronic Frontier Foundation, 2015–)

(c) Stellen Sie Firefox datenschutzgerecht ein. (Das beginnt mit der Default-Startseite!)
Zeigen Sie die Benutzung von temporary containers, von Profilen (z.B. ein Profil für Browsing im Screen-Share).
Führen Sie Browser-Plugins `uMatrix`, `uBlockOrigin` vor.

2. zu: E. W. Dijkstra: *Answers to Questions from Students of Software Engineering* (Austin, 2000) (EWD 1035)
   - „putting the cart before the horse“
     – übersetzen Sie wörtlich ins Deutsche,
     – geben Sie eine entsprechende idiomatische

Redewendung in Ihrer Muttersprache an,
– wofür stehen *cart* und *horse* hier konkret?

3. sind die empfohlenen exakten Techniken der Programmierung für große Systeme anwendbar?
   Erklären Sie „lengths of . . . grow not much more than linear with the lengths of . . .“.
   - Welche Längen werden hier verglichen?

   Modellieren Sie das System als Graph, die Knoten sind die Komponenten, die Kanten sind deren Beziehungen (direkte Abhängigkeiten).
   - Welches asymptotische Wachstum ist bei undisziplinierter Entwicklung des Systems zu befürchten?
   - Welche Graph-Eigenschaft impliziert den linearen

Zusammenhang?

- Wie gestaltet man den System-Entwurf, so daß diese Eigenschaft tatsächlich gilt? Welchen Nutzen hat das für Entwicklung und Wartung?

4. Lesen Sie E. W. Dijkstra: *On the foolishness of natural language programming“*
   `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html`
   und beantworten Sie
   - womit wird “einfaches Programmieren” fälschlicherweise gleichgesetzt?
   - welche wesentliche Verbesserung brachten höhere Programmiersprachen, welche Eigenschaft der Maschinensprachen haben sie trotzdem noch?

- warum sollte eine Schnittstelle *narrow* sein?
- welche formalen Notationen von Vieta, Descartes, Leibniz, Boole sind gemeint? (jeweils: Wissenschaftsbereich, (heutige) Bezeichnung der Notation, Beispiele)
- warum können Schüler heute das lernen, wozu früher nur Genies in der Lage waren?
- Übersetzen Sie den Satz “the naturalness of . . . obvious”.

Geben Sie dazu jeweils an:

- die Meinung des Autors, belegt durch konkrete Textstelle und zunächst wörtliche, dann sinngemäße Übersetzung
- Beispiele aus Ihrer Erfahrung

5. Über ein Monoid $(M, \circ, 1)$ mit Elementen $a, b \in M$ (sowie eventuell weiteren) ist bekannt: $a^2 = b^2 = (ab)^2 = 1$.
   Dabei ist ab eine Abkürzung für $a \circ b$ und a^2 für aa, usw.
   - Geben Sie ein Modell mit $1 \neq a \neq b \neq 1$ an.
   - Überprüfen Sie $ab = ba$ in Ihrem Modell.
   - Leiten Sie $ab = ba$ aus den Monoid-Axiomen und gegebenen Gleichungen ab.

   Das ist eine Übung zur Wiederholung der Konzepte *abstrakter* und *konkreter* Datentyp sowie *Spezifikation*.

6. im Rechnerpool live vorführen:
   - ein Terminal öffnen
   - `ghci` starten (in der aktuellen Version), Fakultät von 100 ausrechnen

- Datei `F.hs` mit Texteditor anlegen und öffnen, Quelltext `f = ...` (Ausdruck mit Wert `100!`) schreiben, diese Datei in `ghci` laden, `f` auswerten

Dabei wg. Projektion an die Wand:

Schrift 1. groß genug und 2. schwarz auf weiß.

Vorher Bildschirm(hintergrund) aufräumen, so daß bei Projektion keine personenbezogenen Daten sichtbar werden. Beispiel: `export PS1="$ "` ändert den Shell-Prompt (versteckt den Benutzernamen).

Vorführung der Aufgaben *auf Pool-Rechner*

Daten

Wiederholung: Terme

- (Prädikatenlogik) *Signatur* Σ ist Menge von Funktionssymbolen mit Stelligkeiten

  ein Term t in Signatur Σ ist
  - Funktionssymbol $f \in \Sigma$ der Stelligkeit k
    mit Argumenten $(t_1, \ldots, t_k)$, die selbst Terme sind.

  $\mathrm{Term}(\Sigma) =$ Menge der Terme über Signatur Σ
- (Graphentheorie) ein Term ist ein gerichteter, geordneter, markierter Baum
- (Datenstrukturen)
  - Funktionssymbol $=$ Konstruktor, Term $=$ Baum

Beispiele: Signatur, Terme

- Signatur: $\Sigma = \{Z/0, S/1, f/2\}$
- Elemente von $\mathrm{Term}(\Sigma)$:

  $Z(), \quad S(S(Z())), \quad f(S(S(Z())), Z())$
- Abkürzung: das leere Argument-Tupel (die Klammern) nach nullstelligen Symbolen weglassen, $f(S(S(Z)), Z)$
- Signatur: $\Gamma = \{E/0, A/1, B/1\}$
- Elemente von $\mathrm{Term}(\Gamma)$: ...
- Bezeichnung: für Signatur Σ und $k \in \mathbb{N}$:

  Σ_k bezeichnet Menge der Symbole aus Σ mit Stelligkeit k

  $\Sigma_0 = \{Z\}, \Sigma_1 = \{S\}, \Sigma_2 = \{f\},$

  $\Gamma_0 = \{E\}, \Gamma_1 = \ldots, \Gamma_2 = \ldots$

Abmessungen von Termen

- die Größe: ist Funktion $|\cdot| : \mathrm{Term}(\Sigma) \to \mathbb{N}$ mit
  - für $f \in \Sigma_k$ gilt $|f(t_1, \ldots, t_k)| = 1 + |t_1| + \ldots + |t_k|$

  die Größe eines Terms ist der Nachfolger der Summe der Größen seiner Kinder
- Bsp: $|S(S(Z()))| = 1 + |S(Z())| = 1 + 1 + |Z()| = 1 + 1 + 1$
  $|f(S(S(Z())), Z()| = \ldots$
- die Höhe: ist Funktion height : $\mathrm{Term}(\Sigma) \to \mathbb{N}$:

  für $t = f(t_1, \ldots, t_k)$ gilt
  - wenn $k = 0$, dann height$(t) = 0$
  - wenn $k > 0$, dann

    height$(t) = 1 + \max($height$(t_1), \ldots,$ height$(t_k))$

Induktion über Termaufbau (Beispiel)

- Satz: $\forall t \in \mathrm{Term}(\{a/0, b/2\}) : |t| \equiv 1 \pmod 2$ (die Größe ist ungerade)
- Beweis durch Induktion über den Termaufbau:
  - IA (Induktions-Anfang): $t = a()$
    Beweis für IA: $|t| = |f()| = 1 \equiv 1 \pmod 2$
  - IS (I-Schritt): $t = b(t_1, t_2)$
    zu zeigen ist: IB (I-Behauptung): $|t| \equiv 1 \pmod 2$
    dabei benutzen: IV (I-Voraussetzung) $|t_1| \equiv |t_2| \equiv 1 \pmod 2$
    Beweis für IS:
    $|t| = |b(t_1, t_2)| = 1 + |t_1| + |t_2| \equiv 1 + 1 + 1 \equiv 1 \pmod 2$
- Bezeichnung: das heißt IV, und nicht I-Annahme, damit es nicht mit I-Anfang verwechselt wird

Algebraische Datentypen (benannte Notation)

- Beispiel: Deklaration des Typs

```
data Foo = Con {bar :: Int, baz :: String}
    deriving Show
```

- Bezeichnungen:
  - `Foo` ist Typname
  - `Con` ist Konstruktor
  - `bar, baz` sind Komponenten-Namen des Konstruktors
  - `Int, String` sind Komponenten-Typen
- Beispiel: Konstruktion eines Datums dieses Typs

```
Con { bar = 3, baz = "hal" } :: Foo
```

der Ausdruck (vor dem `::`) hat den Typ `Foo`

Algebraische Datentypen (positionelle Not.)

- Beispiel: Deklaration des Typs

```
data Foo = Con Int String
```

- Bezeichnungen:
  - `Foo` ist Typname
  - `Con` ist zweistelliger Konstruktor
    ...mit anonymen Komponenten
  - `Int`, `String` sind Komponenten-Typen
- Beispiel: Konstruktion eines Datums dieses Typs

```
Con 3 "hal" :: Foo
```

- auch ein Konstruktor mit benannten Komponenten kann positionell aufgerufen werden

Datentyp mit mehreren Konstruktoren

- Beispiel (selbst definiert)

```
data T = A { foo :: Bool }
       | B { bar :: Ordering, baz :: Bool }
  deriving Show
```

- Bespiele (in Standardbibliothek (Prelude) vordefiniert)

```
data Bool = False | True
data Ordering = LT | EQ | GT
```

- Konstruktion solcher Daten:

```
False :: Bool
A { foo = False } :: T ; A False :: T
B EQ True :: T
```

Mehrsortige Signaturen

- (bisher) einsortige Signatur

  ist Abbildung von Funktionssymbol nach Stelligkeit

- (neu) mehrsortige Signatur
  - Menge von Sortensymbolen $S = \{S_1, \ldots\}$
  - msS ist Abb. von Funktionssymbol nach Typ
  - *Typ* ist Element aus $S^* \times S$
    Folge der Argument-Sorten, Resultat-Sorte

Bsp.: $S = \{Z, B\}, \Sigma = \{0 \mapsto ([], Z), p \mapsto ([Z, Z], Z),$
$e \mapsto ([Z, Z], B), a \mapsto ([B, B], B)\}$.

- $\mathrm{Term}(\Sigma, B)$ (Terme dieser Signatur mit Sorte B): ...

Rekursive Datentypen

- Konstruktoren mit benannten Komponenten

```
data Tree = Leaf {}
    | Branch { left :: Tree , right :: Tree }
```

- mit anonymen Komponenten

```
data Tree = Leaf | Branch Tree Tree
```

- Objekte dieses Typs erzeugen, Bsp:

```
Leaf :: Tree; Branch (Branch Leaf Leaf) Leaf :: Tree
```

- Bezeichnung `data Tree = ... | Node ...` ist falsch (irreführend), denn sowohl äußere Knoten (Leaf) als auch innere Knoten (Branch) *sind* Knoten (Node)
- Ü: die data-Dekl. für $S = \{Z, B\}$, $\Sigma = \{0 \mapsto ([], Z), p \mapsto ([Z, Z], Z), e \mapsto ([Z, Z], B), a \mapsto ([B, B], B)\}$.

Daten mit Baum-Struktur

- mathematisches Modell: Term über Signatur
- programmiersprachliche Bezeichnung: *algebraischer Datentyp* (die Konstruktoren bilden eine Algebra)
- praktische Anwendungen:
  - Formel-Bäume (in Aussagen- und Prädikatenlogik)
  - Suchbäume (in VL Algorithmen und Datenstrukturen, in `java.util.TreeSet<E>`)
  - DOM (Document Object Model) `https://www.w3.org/DOM/DOMTR`
  - JSON (Javascript Object Notation) z.B. für AJAX `https://www.ecma-international.org/publications/standards/Ecma-404.htm`

Übung Terme

- Geben Sie die Signatur des Terms $\sqrt{a \cdot a + b \cdot b}$ an.
- Bestimmen Sie $|\sqrt{a \cdot a + b \cdot b}|$, height$(\sqrt{a \cdot a + b \cdot b})$.
- Geben Sie ein Element $t \in \mathrm{Term}(\{f/1, g/3, c/0\})$ an mit $|t| = 5$ und height$(t) \leq 2$.
- die Menge $\mathrm{Term}(\{f/1, g/3, c/0\})$ wird realisiert durch den Datentyp

  ```
  data T = F T | G T T T | C deriving Show
  ```

  deklarieren Sie den Typ in ghci, erzeugen Sie o.g. Term t (durch Konstruktoraufrufe)
- Holes (Löcher) in Ausdrücken als Hilfsmittel bei der

Programmierung durch schrittweises Verfeinern

```
ghci> data T = A Bool | B T deriving Show
ghci> A _

<interactive>:2:3: error:
    • Found hole: _ :: Bool
    • In the first argument of ‘A’, namely ‘_’
      In the expression: A _
      In an equation for ‘it’: it = A _
    • Relevant bindings include ...
      Valid hole fits include ...
        False :: Bool
        True :: Bool
        ...
```

Hausaufgaben

Allgemeine Hinweise zu Arbeit und Präsentation im Pool:

- beachten Sie `https://www.imn.htwk-leipzig.de/~waldmann/etc/pool/` (`PATH` und ggf. `LD_LIBRARY_PATH`)

- Schrift schwarz auf weiß! Vernünftige Schriftgröße (Control-Plus)!!

  Gleichzeitig sichtbar (d. h.: keine Verdeckungen, Umschaltungen): Aufgabenstellung, Programmtext, Ausgabe/Fehlermeldungen.

  Wenn der Desktop-Hintergrund sichtbar ist—wurde Platz verschenkt!!!

Die Arbeitsfläche wird vollständig ausgenutzt durch Tiling (Kacheln), Bsp. `https://xmonad.org/` (Spencer Janssen, Don Stewart, Jason Creigh et al., 2007–)

SS 26: 2, 4, 5

1. (Pflicht-Aufgaben im autotool beachten.)

2. Geben Sie einen Typ T (eine `data`-Deklaration) an, der alle Terme der einsortigen Signatur $\Sigma = \{E/0, F/2, G/3\}$ enthält.

   Konstruieren Sie Elemente dieses Typs.

   Geben Sie $t \in \mathrm{Term}(\Sigma)$ an mit

   - height$(t) = 2$ und $|t|$ möglichst klein
   - height$(t) = 2$ und $|t|$ möglichst groß

3. Geben Sie einen Typ (eine `data`-Deklaration) mit genau 71 Elementen an. Sie können weitere Data-Deklarationen benutzen. Minimieren Sie die Gesamt-Anzahl der Konstruktoren. Bsp:

   ```
   data Bool = False | True ; data T = X Bool Bool
   ```

   dieser Typ `T` hat 4 Elemente, benutzt ingesamt 3 Konstruktoren (`False,True,X`)

4. Beweisen Sie $\forall \Sigma : \forall t \in \mathrm{Term}(\Sigma) :$ height$(t) \leq |t| - 1$.

   durch Induktion über den Term-Aufbau.

   - Induktions-Anfang: $t = f()$ (nullstelliges Symbol f)
   - Induktions-Schritt:
     $t = f(t_1, \ldots, t_k)$ (k-stelliges Symbol f, für $k > 0$)
     dabei Induktions-Voraussetzung: die Behauptung gilt

für $t_1, \ldots, t_k$.
Induktions-Behauptung: . . . für t.

Für welche Terme t gilt Gleichheit? Wo sieht man das im Beweis?

5. wieviele Elemente des Datentyps
`data T = L | B T T` haben . . .

- die Größe 9
- die Größe ≤ 9
- die Höhe $0, 1, 2, 3$, Zusatz: größere konkrete Werte, allgemein (Formel)

Sie müssen diese Elemente nicht alle einzeln angeben.

Bestimmen sie ihre Anzahl durch dynamische Programmierung (von Hand).

Aufgaben autotool (Beispiele)

1. einen Term in einer mehrsortigen Signatur angeben (Programmiersprache: Java)

```
Gesucht ist ein Ausdruck vom Typ int
in der Signatur
    Foo e;
    String g;
    static char a ( Bar x, String y, String z );
    static Bar b ( Foo x );
    static int c ( int x, String y );
    static int d ( char x, Foo y, String z );
    static Foo f ( int x );
    static String h ( Foo x, String y, char z );
```

Lösungsansatz: `d (a (b (e), ...), ...)`

2. Fill holes (_) (replace with one item)
and ellipsis (...) (replace with several items)
in the following, so that the claim at the top becon

For this exercise, each data declaration can only re
not to itself or later (it cannot be recursive).

```
size_of (T) == 71 where
  { data Bool =  False | True
  ; data Col =  R | G | B
  ; data S =  ...
  ; data T =  ...
  }
```

Plan (vorläufig)

- KW 15: Einführung
- KW 16: Daten (Terme, algebraische Datentypen)
- KW 17: Programme (TRS, Pattern matching)
- KW 18: Beweise, Induktion (cyp)
- KW 19: Funktionen als Daten (λ - dyn. Semantik)
- KW 21: Polymorphie (λ - stat. Semantik)
- KW 22: Rekursive polymorphe algebraische Datentypen
- KW 23: Rekursionmuster
- KW 24: eingeschränkte Polymorphie (Schnittstellen)
- KW 25: Strictness, Bedarfsauswertung
- KW 26: Datenströme (Iteratoren)
- KW 27:
- KW 28: Zusammenfassung, Ausblick