Praktische Informatik 3: Funktionale Programmierung
Vorlesung 12 vom 17.01.17: Domänenspezifische Sprachen (DSLs)

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Domain-Specific Languages (DSLs)

- Was ist das?
- Wie macht man das?
- Wozu braucht man so etwas?

Programmiersprachen sind überall

- Beispiel 1: SQL — Anfragesprache für relationale Datenbanken

Programmiersprachen sind überall

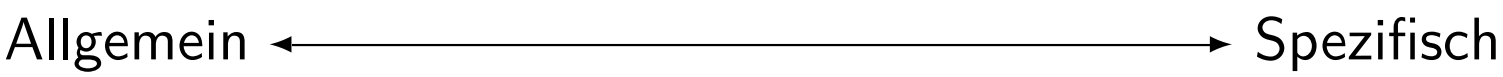
- Beispiel 1: SQL — Anfragesprache für relationale Datenbanken
- Beispiel 2: Excel — Modellierung von Berechnungen

Programmiersprachen sind überall

- Beispiel 1: SQL — Anfragesprache für relationale Datenbanken
- Beispiel 2: Excel — Modellierung von Berechnungen
- Beispiel 3: HTML oder LaTeX oder Word — Typesetting

Vom Allgemeinen zum Speziellen

- Modellierung von Problemen und Lösungen



Allgemeine Lösung: GPL

- Mächtige Sprache (Turing-mächtig)
- Große Klasse von Problemen
- Großer Abstand zum Problem
- Java, Haskell, C ...
- General purpose language (GPL)

Spezifische Lösung: DSL

- Maßgeschneiderte Sprache
- Wohldefinierte Unterklasse (Domäne) von Problemen
- Geringer Abstand zum Problem
- Domain-Specific Language (DSL)
- Als Teil einer Programmiersprache (eingebettet) oder alleinstehend (stand-alone)

DSL: Definition 1

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

(van Deursen et al., 2000)

Eigenschaften von DSLs

- Fokussierte Ausdrucksmächtigkeit
  - Turing-Mächtigkeit nicht Ziel der Sprache (aber kein Ausschlusskriterium)
  - Oftmals deutlich weniger mächtig: Reguläre Ausdrücke, Makefiles, HTML
- Üblicherweise klein (“little languages”, “micro-languages”)
- Anzahl der Sprachkonstrukte eingeschränkt und auf die Anwendung zugeschnitten
- Meist deklarativ: XSLT, Relax NG Schemas, Excel Formeln...

DSL-Beispiel: Relax NG

Adressbuchformat

```
grammar {
  start = entries
  entries = element entries { entry* }
  entry = element entry {
    attribute name { text },
    attribute birth { xsd:dateTime },
    text }
}
```

- Beschreibung von XML-Bäumen
  - Erlaubte Element-Verschachtelungen & -Reihenfolgen
  - Datentypen von Attributen & Elementwerten
- Automatische Generierung von Validatoren
- Nicht Turing-mächtig (?)

Domain-Specific Embedded Languages

- DSL direkt in eine GPL einbetten
  - Vorhandenes Ausführungsmodell und Werkzeuge
- Funktionale Sprachen eignen sich hierfür besonders gut
  - Algebraische Datentypen zur Termrepräsentation
  - Funktional $\subseteq$ Deklarativ
  - Funktionen höherer Ordnung ideal für Kombinatoren
  - Interpreter (`ghci`, `ocaml`, ...) erlauben "rapid prototyping"
  - Erweiterung zu stand-alone leicht möglich
- Andere Sprachen:
  - Java: Eclipse Modelling Framework, Xtext

Beispiel: Reguläre Ausdrücke

Ein regulärer Ausdruck ist:

- Leeres Wort ϵ
- Einzelnes Zeichen c
- Beliebiges Zeichen ?
- Sequenzierung $e_1\ e_2$
- Alternierung $e_1 \mid e_2$
- Kleene-Stern $e^* = \epsilon \mid ee^*$
- Abgeleitet:
  - Kleene-Plus $e^+ = e\ e^*$

Beispiel: Reguläre Ausdrücke

Ein regulärer Ausdruck ist:

- Leeres Wort ϵ
- Einzelnes Zeichen c
- Beliebiges Zeichen ?
- Sequenzierung $e_1\ e_2$
- Alternierung $e_1 \mid e_2$
- Kleene-Stern $e^* = \epsilon \mid ee^*$
- Abgeleitet:
  - Kleene-Plus $e^+ = e\ e^*$

Haskell-Implementierung — Signatur:

```
type RegEx

eps  :: RegEx
char :: Char→ RegEx
arb  :: RegEx
seq  :: RegEx→ RegEx→ RegEx
alt  :: RegEx→ RegEx→ RegEx
star :: RegEx→ RegEx
```

Beispiel: Reguläre Ausdrücke

Ein regulärer Ausdruck ist:

- Leeres Wort ϵ
- Einzelnes Zeichen c
- Beliebiges Zeichen ?
- Sequenzierung $e_1\ e_2$
- Alternierung $e_1 \mid e_2$
- Kleene-Stern $e^* = \epsilon \mid ee^*$
- Abgeleitet:
  - Kleene-Plus $e^+ = e\ e^*$

Haskell-Implementierung — Signatur:

```
type RegEx

eps  :: RegEx
char :: Char→ RegEx
arb  :: RegEx
seq  :: RegEx→ RegEx→ RegEx
alt  :: RegEx→ RegEx→ RegEx
star :: RegEx→ RegEx
```

Implementierung: siehe `RegExS.hs`

Regular Ausdrücke: Suche

- Wie modellieren wir mehrfache Suche?
- Signatur:

```
type RegEx =
  String→ [String]
```

- Wie modellieren wir ersetzen?

Besser: Repräsentation durch Datentypen

```
data RE = Eps
        | Chr Char
        | Str String
        | Arb
        | Seq RE RE
        | Alt RE RE
        | Star RE
        | Plus RE
        | Range [Char]
        deriving (Eq, Show)

intp :: RE→ RegEx
```

```
searchAll :: RE→ String→
                    [String]
```

Flache Einbettung vs. Tiefe Einbettung

- Flache Einbettung:
  - Domänenfunktionen direkt als Haskell-Funktionen
  - Keine explizite Repräsentation der Domänenobjekte in Haskell
- Tiefe Einbettung:
  - Repräsentation der Domänenobjekte durch Haskell-Datentyp (oder ADT)
  - Domänenfunktionen auf diesem Datentyp

Flach oder Tief?

- Vorteile flache Einbettung:
  - Schnell geschrieben, weniger 'boilerplate'
  - Flexibel erweiterbar
- Vorteile tiefe Einbettung:
  - Mächtiger: Manipulation der Domänenobjekte
  - Transformation, Übersetzung, ...
  - Bsp: Übersetzung RE in Zustandsautomaten

Beispiel: Grafik

- Erzeugung von SVG-Grafiken
- Eingebettete DSL:
  - Erste Näherung: TinySVG (modelliert nur die Daten)
  - Erweiterung: Monade Draw (Zustandsmonade)
  - Funktionen zum Zeichnen:

```
line    :: Point→ Point→ Draw ()
polygon :: [Point]→ Draw ()
```

  - "Ausführen":

```
draw :: Double→ Double→ String→ Draw ()→ IO ()
```

Beispielprogramm: Sierpińksy-Dreieck

Dreieck mit Eckpunkten zeichnen:

```
drawTriangle :: Point→ Point→ Point→ Draw ()
```

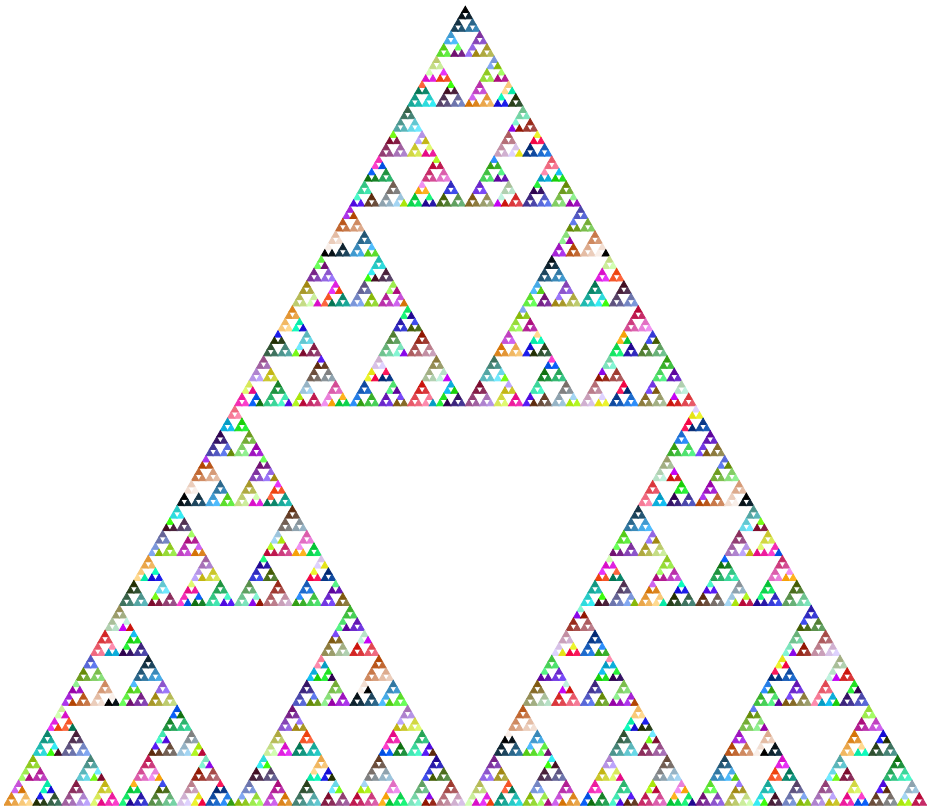
Mitte zwischen zwei Punkten:

```
midway :: Point→ Point→ Point
midway p q = 0.5 `smult` (p+q)
```

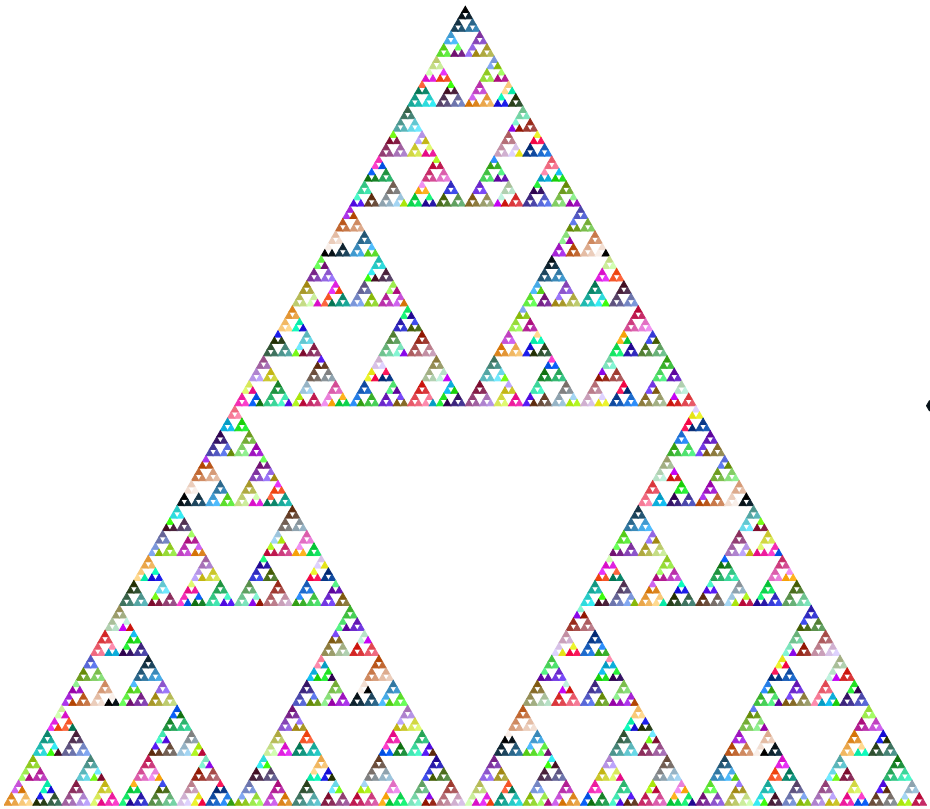
Sierpińksy-Dreieck rekursiv

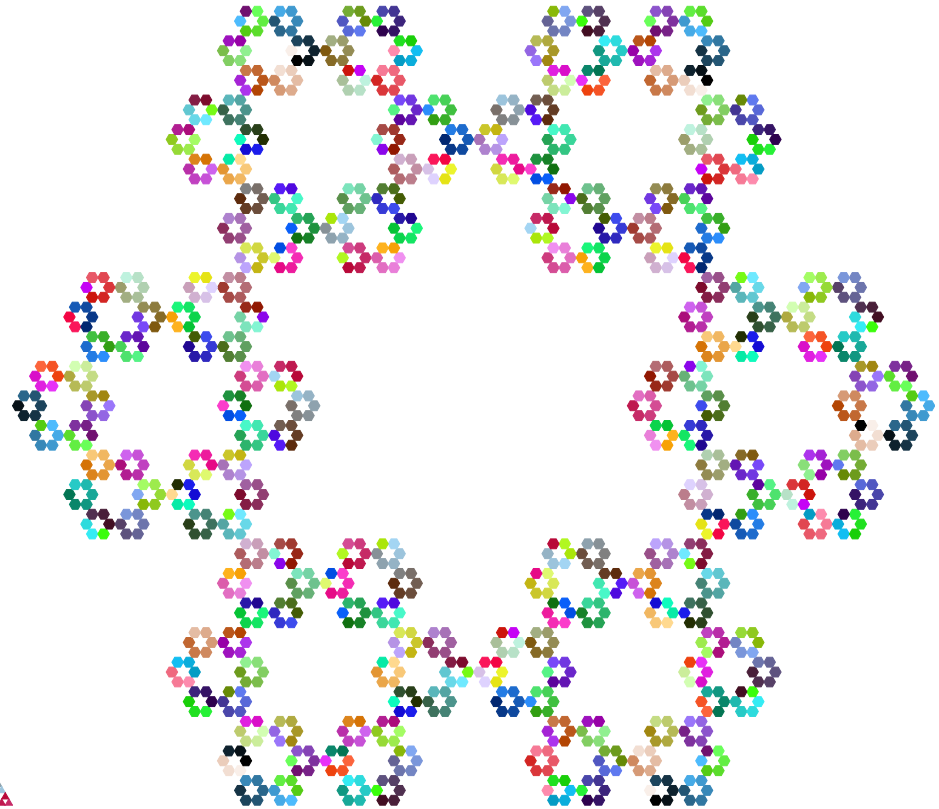
```
spTri :: Double→ Int→ Draw ()
spTri sz limit = sp3 a b c 0 where
  h = sz∗ sqrt 3/4
  a = Pt 0 (−h); b = Pt (−sz/2) h; c= Pt (sz/2) h
  sp3 :: Point→ Point→ Point→ Int→ Draw ()
  sp3 a b c n
    | n ≥ limit = drawTriangle a b c
    | otherwise = do
        let ab = midway a b; bc = midway b c; ca = midway c a
        sp3 a ab ca (n+1); sp3 ab b bc (n+1); sp3 ca bc c (n+1)
```

Resultat: Sierpińsky-Dreieck und Schneeflocke



Resultat: Sierpińsky-Dreieck und Schneeflocke





Erweiterung: Transformation

- Allgemein: Transformation von Grafiken

```
xform :: (Graphics→ Graphics)→ Draw()→ Draw()
```

- Speziell:
  - Rotation um einen Punkt:

```
rotate :: Point→ Double→ Draw ()→ Draw ()
```

  - Skalierung um einen Faktor:

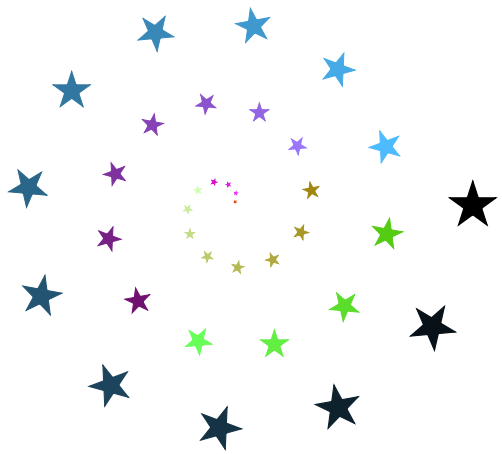
```
scale :: Double→ Draw()→ Draw ()
```

  - Verschiebung um einen Vektor (Punkt):

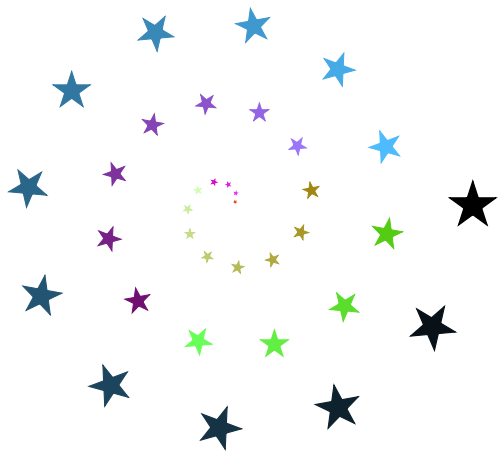
```
translate :: Point → Draw () → Draw ()
```

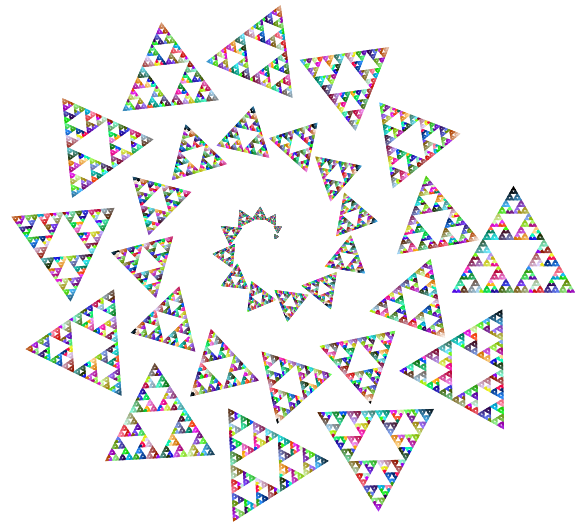
Beispiele: Verschiebung und Skalierung

Beispiele: Verschiebung und Skalierung



Beispiele: Verschiebung und Skalierung





Weitere Abgrenzung

Programmierschnittstellen (APIs)

- Etwa jUnit: `assertTrue()`, `assertEquals()` Methoden & `@Before`, `@Test`, `@After` Annotationen
- Funktionsnamen spiegeln ebenfalls Domänenvokabular wider
- Gängige Sprachen (Java, C/C++) erschweren weitere Abstraktion: Syntaxerweiterungen, Konzepte höherer Ordnung
- Imperative Programmiersprache vs. deklarative DSL

Weitere Abgrenzung

Programmierschnittstellen (APIs)

- Etwa jUnit: `assertTrue()`, `assertEquals()` Methoden & `@Before`, `@Test`, `@After` Annotationen
- Funktionsnamen spiegeln ebenfalls Domänenvokabular wider
- Gängige Sprachen (Java, C/C++) erschweren weitere Abstraktion: Syntaxerweiterungen, Konzepte höherer Ordnung
- Imperative Programmiersprache vs. deklarative DSL

Skriptsprachen

- JavaScript, PHP, Lua, Tcl, Ruby werden für DS-artige Aufgaben verwendet
  - HTML/XML DOM-Manipulation
  - Game Scripting, GUIs, ...
  - Webprogrammierung (Ruby on Rails)
- Grundausrichtung: programmatische Erweiterung von Systemen

Beispiel: Hardware Description Languages

- Ziel: Funktionalität von Schaltkreisen beschreiben
- Einfachster Fall:

```
and :: Bool→ Bool→ Bool
or  :: Bool→ Bool→ Bool
```

- Moderne Schaltkreise sind etwas komplizierter ...

CλaSH

- Modellierung und Simulation von Schaltkreisen in Haskell
- Typ Signal α für synchrone sequentielle Schaltkreise
- Rekursion für Feedback
- Simulation des Verhalten des Schaltkreises möglich
- Generiert VHDL, Verilog, SystemVerilog, und Testdaten

- Verwandt: Chisel (in Scala), Bluespec (kommerziell), Lava (veraltet)

Beispiel: SQL

- SQL-Anfragen werden in Haskell modelliert, dann übersetzt und an DB geschickt
- Vorteil: typsicher, ausdrucksstark
- Wie modelliert man das Ergebnis? $\rightarrow$ Abbildung Haskell-Typen auf DB
- Haskell: Opaleye
- Scala: Slick

Vorteile der Verwendung von DSLs

- Ausdruck von Problemen/Lösungen in der Sprache und auf dem Abstraktionslevel der Anwendungsdomäne
- Notation matters: Programmiersprachen bieten oftmals nicht die Möglichkeit, Konstrukte der Domäne angemessen wiederzugeben
- DSL-Lösungen sind oftmals selbstdokumentierend und knapp
- Bessere (automatische) Analyse, Optimierung und Testfallgenerierung von Programmen
  - Klar umrissene Domänensemantik
  - eingeschränkte Sprachmächtigkeit $\Rightarrow$ weniger Berechenbarkeitsfallen
- Leichter von Nicht-Programmierern zu erlernen als GPLs

Nachteile der Verwendung von DSLs

- Hohe initiale Entwicklungskosten
- Schulungsbedarf
- Sprachdesign ist eine äußerst schwierige und komplexe Angelegenheit, deren Aufwand nahezu immer unterschätzt wird
- Fehlender Tool-Support
  - Debugger
  - Generierung von (Online-)Dokumentation
  - Statische Analysen, ...
- Effizienz: Interpretation ggf. langsamer als direkte Implementierung in GPL

Zusammenfassung

- DSL: Maßgeschneiderte Sprache für wohldefinierten Problemkreis
- Vorteile: näher am Problem, näher an der Lösung
- Nachteile: Initialer Aufwand
- Klassifikation von DSLs:
  - Flache vs. tiefe Einbettung
  - Stand-alone vs. embedded
- Nächste Woche: Scala — eine Einführung.

Literatur


Koen Claessen and David Sands.
Observable sharing for functional circuit description.
In P. S. Thiagarajan and R. Yap, editors, *Advances in Computing Science – ASIAN'99*, volume 1742 of *LNCS*, pages 62–73, 1999.


Paul Hudak.
Building domain-specific embedded languages.
ACM Comput. Surv., 28, 1996.


Marjan Mernik, Jan Heering, and Anthony M. Sloane.
When and how to develop domain-specific languages.
ACM Comput. Surv., 37(4):316–344, 2005.


Arie van Deursen, Paul Klint, and Joost Visser.
Domain-specific languages: an annotated bibliography.
SIGPLAN Not., 35(6):26–36, 2000.