Funktionale Programmierung

Teil 1: Einführung

Programm

Übersicht

Woche	Thema	Praktika
8	Organisatorisches, historisches, Begriff der funktionalen Programmierung, Einführung F#	1
9	Funktionen und Typen I: Werte, Unions (Listen), Produkte	2
10	Funktionen und Typen II: Options, Funktionstyp, “partial application”, Currying	2,3
11	Funktionen und Typen III: Kombinatoren und höhere Funktionen	3
12	Rekursion I: Formen der Rekursion	4
13	Rekursion II: Fixpunkte	4,5
14	Rekursion III: Rekursion und Compiler Optimierungen	5

Programm

Detail

- Organisatorisches
  - Vorbemerkung
  - Ablauf
  - Leistungsnachweis
- Funktionale Programmierung - was heisst das?
  - Das imperative Modell
  - Das funktionale Modell
  - Historischer Überblick
  - Das funktionale Modell zeitgenössisch interpretiert
- Einführung in F#
  - Installation
  - F# in 60 seconds

Organisatorisches

Organisatorisches

Vorbemerkung

- Kurs über Konzepte und Grundlagen der funktionalen Programmierung
- F# (ein ML Dialekt) als "Arbeitssprache"
- Keine Vorlesung über "die Sprache F#"
- Unterlagen und weitere Informationen: OLAT

Zweigeteilter Lektionsablauf	
Theorie	Praktikum
■ +/- Frontalunterricht ■ Basiert auf Folien (OLAT) ■ Auflockerung durch Programmieraufgaben und Demos	■ Praktikas als Gruppenarbeit ■ Basiert auf Praktikumsblättern (OLAT) ■ Theorieaufgaben, Programmieraufgaben und einiges dazwischen

Organisatorisches

Leistungsnachweis

- Die Kursnote ergibt sich aus der Semesterendprüfung
- Die Prüfung beinhaltet Theoriefragen und praktische Programmieraufgaben
- Die Prüfung wird “elektronisch” (am Notebook) geschrieben 😎

Administrative und organisatorische Fragen?

Beispiel (Elemente einer Liste aufsummieren):

```
sum(L){
    l = length(L);
    i = 0;
    sum = 0;
    while i < l do
        sum = sum + L[i];
        i = i + 1;
    return sum;
    }
```

- Basiert auf Zuständen, Zustandsübergängen und deren zeitlichen Abfolge.
- Ein imperativer Algorithmus beschreibt, wie ein Problem Schritt-für-Schritt gelöst wird.
- Probleme werden in geeignete Arbeitsschritte zerlegt, welche in einer bestimmten Reihenfolge abgearbeitet werden müssen.

Beispiel (Elemente einer Liste aufsummieren):

- "Mathematisch":

$$sum(\varnothing) = 0$$
$$sum(\{a_1, \ldots, a_n\}) = a_1 + sum(\{a_2, \ldots, a_n\})$$

- Funktional (Haskell):

```
sum [] = 0
sum (a1:rest) = a1 + sum rest
```

- Funktional (F#)

```
let sum = function
    | [] -> 0
    | a1::rest -> a1 + sum rest
```

- Nahe an der mathematischen Notation (Funktionen sind mathematische Objekte)
- Definition/Beschreibung einer Funktion anstelle expliziter Anweisungen (deklarativ)
- Keine "Variablen" (im Sinn von Veränderlichen)
  - $\Rightarrow$ Kein Zustand
  - $\Rightarrow$ Keine (explizite) Zeit!

Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values.
Graham Hutton

- 1930 - 1938 (Church, Kleene, Rosser): Theoretische Fundierung der funktionalen Programmierung durch die Entwicklung des λ-Kalküls.
- 1958 (McCarthy): Einführung von LISP.
- 1966 (Landin): Mit *The next 700 programming languages* erscheint ein visionäres Papier in dem viele moderne funktionale Sprachkonzepte vorweggenommen werden (z.B. algebraische Datentypen).
- 1977 (Backus): Einführung der Sprache FP, "Function-level programming".
- 1970er: Stark typisierte funktionale Sprachen. ML (Meta Language) als Teil eines Theorem-Beweis-Programms ("Logic of Computable Functions").

- 1983-86 (Turner): Miranda als frühe Sprache mit “lazy evaluation”, algebraischen Datentypen und polymorphem Typsystem à la Milner.
- Neuere Entwicklungen: Funktionale Sprachen für Mainstream “Plattformen” wie JVM (Scala, Clojure), .Net (F#) und Web (clojurescript, purescript, elm, . . .) sowie die Übernahme funktionaler Konzepte in Mainstream-Sprachen (lambdas, closures, continuations, . . .)
- Weiteres zur Geschichte in OLAT/Lektüre/historisches

Es gibt keine feste "Definition", einige für die funktionale Programmierung zentrale Konzepte können aber identifiziert werden:

- Vollwertige Unterstützung von Funktionen (first class functions): Funktionen höherer Ordnung, λ-Terme, Currying und partielle Anwendung, ...
- Unveränderliche (immutable) Daten, Vermeidung von Nebeneffekten und daraus folgend referenzielle Transparenz
- Hoch entwickeltes Typensystem: Algebraische Datentypen, Pattern Matching, Typinferenz,...
- Rekursion: Tail-Call-Optimization

Alternativ lässt sich funktionale Programmierung auch sprachunabhängig als “Programmierstil” oder eine Menge von “Patterns” definieren.

- Vermeiden von Seiteneffekten und veränderlichen Zuständen
- Komponierbarkeit anstreben (“transform-oriented-programming”)
- Höhere Funktionen und Rekursion der Iteration vorziehen
- Abstraktion (z.B. Monade und Monoide (vgl. Map-Reduce))
- ...

Funktionale Sprachen sind Programmiersprachen, die den funktionalen Stil erleichtern oder, im Fall rein funktionaler Sprachen, sogar (weitgehend) erzwingen.

Einige modernere Sprachen nach “Funktionalität geschichtet”:

- “Rein funktional”: Haskell, Frege, Miranda, ...
- “Functional first”: Nemerle, Erlang, ML (SML, OCaml, F#), ...
- “Spezialfall Lisp-Familie”: Scheme, Clojure, ...
- “Funktional inspiriert”: Julia, Scala, Rust, ...

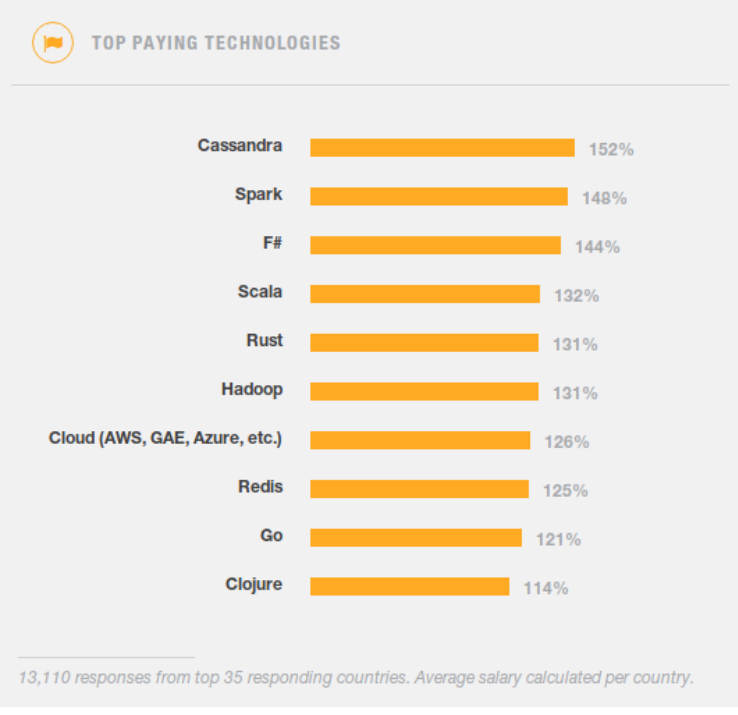
Anwendungen:

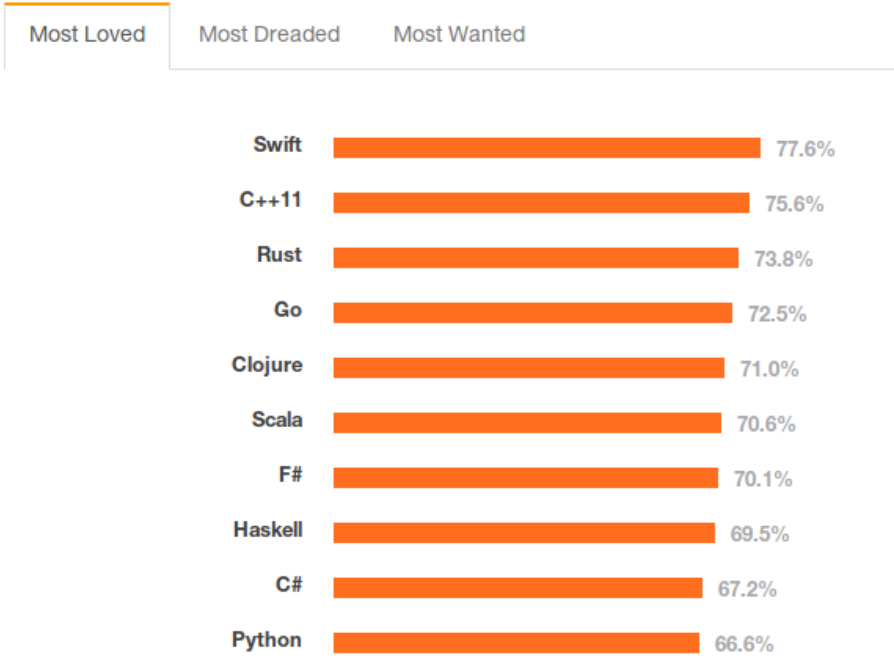
- Klassisch: Compilerbau, Theorembeweiser, KI
- Verbreitet: Numerik, Data science, Fintech
- Alles, bis hin zu GUI Programmierung (“functional reactive programming”), ist möglich (und sinnvoll).

Funktionale Programmierung...

- ...ist (momentan?) in Mode.
- ...nimmt immer mehr Einzug in traditionelle Sprachen.
- ...ist in der Industrie gefragt (und wird gut bezahlt).
- ...**macht spass!**.

Stackoverflow 2015 Developer Survey:





Gründe, die für F# als Unterrichtssprache sprechen:

- Alle wesentlichen Eigenschaften von funktionalen Sprachen vorhanden (functional first language)
- Interaktive Programmierumgebung (FSI)
- Plattformunabhängig (Linux, Windows, BSD, Mac) und open-source
- Produktiv eingesetzte Sprache

Empfohlene Plattformen:

- Visual studio (Win)
- Xamarin Studio (Win & OS X[1])
- Atom Editor + mono + ionide (Win & Linux)
- VS Code + mono + ionide (Win & Linux)

weitere Informationen **hier**.

[1] Nicht getestet.

Aufgabe

Installieren Sie eine zu Ihrem System passende Plattform für F# (mindestens einen Compiler und fsi)

Aufgabe

Besuchen Sie die Seite **diesen** Link, kopieren Sie den angezeigten Quellcode in den Editor Ihrer Wahl und speichern Sie das Ganze als .fsx Datei. Gehen Sie anschliessend durch den Code (und die Kommentare) und denken Sie daran, dass Sie mit Alt + Enter interaktiv markierte Codesegmente ausführen können.

- Grundsätzlich werden wir während des ganzen Kurses (mit wenigen ausnahmen) mit Skripten (.fsx) arbeiten.
- Wenn Sie Beispiele implementieren oder eigenen Code schreiben, dann nutzen Sie den REPL!
- Die Folien folgen der Notation:

```
// hier steht Code
let x = 5
```

```
> hier steht eine Rueckmeldung von fsi
val x : int = 5
```

```
let myString = "hello"    //no types needed
```

heisst aber nicht, dass keine Typen vorhanden[2] sind. Die Typen werden automatisch abgeletiet[3].

```
val myString : string = "hello"
```

Inkonsistente Ausdrücke werden nicht ausgewertet:

```
> "hello" + 3;;
error FS0001: The type 'int' does not match...
```

[2] F# ist stark und statisch typisiert. Bezeichner von der Form 'a stehen für generische ("polymorphe") Typen.

[3] Diesen Mechanismus wird als "type-inference" bezeichnet.

Rekursive Funktionsdeklarationen erfordern das `rec` Schlüsselwort:

```
let rec exp2 x =
    if x < 1 then 1
    else 2 * exp2 (x-1)
```

Funktionen können partiell angewendet werden:

```
let add x y z = x + y + z

let add10 x = add 0 10 x

let sub x y = add 0 x -y
```

Zu beachten ist dabei, dass die Funktionsanwendung linksassoziativ ist. Ein Ausdruck von der Form '`F x y z`' entspricht daher '`((F x) y) z`' und nicht etwa '`F (x (y z))`'.

Pattern-Matches können an Bedingungen (guards) geknüpft werden:

```
let isEvenAndNonzero x =
    match x with
    | 0 -> false
    | x when x % 2 = 0 -> true
    | x -> false
```

Funktionen können auch Argumente (und wie wir bereits gesehen haben Rückgabewerte) von anderen Funktionen[4] sein:

```
let compose f g x = f (g x)
```

Die Funktion `compose` ist vom Typ:

```
('a -> 'b)  ->  ('c -> 'a)  ->  ('c -> 'b)
```

Funktion als Argument — Funktion als Argument — Funktion als Rückgabe

[4] Man spricht in dem Zusammenhang von Funktionen höherer Ordnung.

Funktionen können anonym, als λ-Ausdrücke[5] notiert werden. Die Notation entspricht ungefähr der mathematischen Schreibweise $x \mapsto f(x)$ und wird in F# wie folgt umgesetzt:

```
//denotes a function f with f x = 5 * (x + 10)
compose (fun x -> 5 * x) (fun x -> x + 10)
```

[5] λ = "lambda"

Anonyme Funktionen und Pattern-Matchings lassen sich mit dem `function` Schlüsselwort zusammenfassen:

```
function
| 1 -> "Eins"
| 2 -> "Zwei"
| _ -> "viel"
```

entspricht

```
fun x ->
    match x with
    | 1 -> "Eins"
    | 2 -> "Zwei"
    | _ -> "viel"
```

Durch Klammern von Funktionsnamen, kann man Funktionen definieren, die in der Infix-Notation ausgewertet werden können. Erlaubte Namen für Infix-Funktionen sind alle Zeichenketten über "!%&*+-./<=>@^|?":

```
let rec (^) x y =
    if y < 1 then 1
    else x * (^) x (y-1)
```

```
> 2 ^ 10;;
val it : int = 1024
```