

Grundlagen der Programmierung 2 (1.A)

Einführung

Prof. Dr. Manfred Schmidt-Schauß

Sommersemester 2018

Geplanter Inhalt der ersten Hälfte:

- Programmieren in Haskell
 - Definitionen; Rekursion
 - Auswertung in Haskell
 - Programmieren mit Listen
 - Datenstrukturen: Bäume
 - Polymorphe Typen und Typklassen
- Compilerbau;
 - Lexer
 - Parser
 - Kombinator-Parser
 - Kode-Erzeugung; abstrakte Maschinen
 - Shift-Reduce Parser und Compiler-Generatoren

Haskell und funktionale Programmierung:

- <http://www-stud.informatik.uni-frankfurt.de/~prg2>
insbesondere das Skript zur Vorlesung
- www.haskell.org Haskell-Web-Seite
- <http://haskell.org/onlinereport/> Haskell-Doku
- Manuel Chakravarty und Gabriele Keller, Einführung in die
Programmierung mit Haskell
- Ein aktuelles Online Buch:
<https://www.manning.com/books/get-programming-with-haskell>
- Richard Bird, Introduction to Functional Programming Using Haskell
- Simon Thompson, Haskell: The Craft of Functional Programming
- Graham Hutton, Programming in Haskell (2007)

Compiler:

- J. D. Ullman, M. S. Lam, R. Sethi , A. V. Aho
Compilers: Principles, Techniques, and Tools , 2nd Edition, Pearson 2006
DE: Compiler: Prinzipien, Techniken und Werkzeuge, Pearson Studium, 2008
- Niklaus Wirth, Grundlagen und Techniken des Compilerbaus, Oldenbourg 1997

rekursive Programmierung
mit einer stark typisierten
funktionalen Programmiersprache
mit parametrischem Polymorphismus

Haskell

Haskell ist eine **moderne** Programmiersprache;
sehr weitgehende Konzepte werden erprobt und kombiniert:

- **strenge und statische Typisierung**
- **Nicht-strikte Auswertung**
 - ⇒ viele korrekte Programmtransformationen
 - ⇒ korrekte automatische Parallelisierung
 - ⇒ Test und Verifikation wird erleichtert.
- **Prozess-Programmierung und Konkurrente Auswertung**
 - ⇒ deklarative Programmierung

Wichtige Eigenschaften funktionaler Programmiersprachen

Referentielle Transparenz

Gleiche Funktion, gleiche Argumente \Rightarrow gleicher (Rückgabe-)Wert
Keine Seiteneffekte! D.h. keine Änderung von Objekten

Verzögerte Auswertung

Nur die für das Resultat notwendigen Unterausdrücke werden
(so spät wie möglich) ausgewertet.

Parametrisch Polymorphes Typsystem

Nur Ausdrücke mit Typ sind erlaubt — es gibt Typvariablen.
Das Typsystem garantiert: keine dynamischen Typfehler.

Automatische Speicherverwaltung

Anforderung und Freigabe von Speicher

OCaml: Variante von ML, eine Programmiersprache analog zu Haskell.
Aus dem Artikel von *Yaron Minsky und Stephen Weeks*: (JFP 2008)

Immutability wird gelobt: entspricht Verzicht auf Zuweisungen

Pattern Matching wird gelobt: entspricht Datentypen mit
Konstruktoren und case-Expressions

Tail-Rekursions-Optimierung wird vermisst. Das gibt es in Haskell

Facebook benutzt Haskell

zur Abwehr von Spam, Hackern und Missbrauch

Aus einem Artikel von *Simon Marlow 2016*

<https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/>

Methode: Schnittstelle zum Regelbasierten Programmieren:
Sogenannte “policies”, die oft geändert werden.

Vorteile:

Pur funktional und streng getypt

Concurrency (Nebenläufigkeit)

automatisch in Concurrent Haskell

Leichte Code-Aktivierung (nach Änderungen)

Performanz


Grundprinzipien: des funktionalen Programmierens

- Definition von Funktionen `quadrat x = x*x`
- Aufbau von Ausdrücken:
Anwendung der Funktion auf Argumente,
die wieder Ausdrücke sein können. `3*(quadrat 5)`
- programminterne Kommunikation:
Nur der **Wert** von Ausdrücken wird bei der
Auswertung zurückgegeben. `75`
- Funktionen können Datenobjekte sein
- Trennung von IO und Auswertung

Darstellung von Quell-Code (Source-code) auf den Folien und Skript:

```
quadrat x = x*x
```

Darstellung von Benutzereingabe und Interpreterausgabe auf Folien und im Skript:

```
*Main> 2+2   
4
```

Wir verwenden den Interpreter **GHCi**





Siehe `www.haskell.org`

Einfacher Download und Installation auf verschiedenen Systemen

Siehe Hilfestellungen auf der Professur-Webseite.
und die zahlreichen Informationen auf Blatt Nr. 0.


Simon Peyton Jones (Microsoft Research) und **Simon Marlow**
die wichtigsten Forscher und Weiterentwickler des
GHC: (Glasgow Haskell Compiler).
(Simon Marlow: bis März 2013:)

Online-Report <http://www.haskell.org/onlinereport>
Aufruf: ghci

```
prompt > ghci   
  < Einige Zeilen Infos  >  
Prelude> :h   
  < Hilfe-Menu  >  
Prelude> :t True   
True :: Bool  
Prelude> :set +s 
```

(druckt den Typ des Ausdrucks True)
(Option s für Statistik gesetzt)

Module im Interpreter verwenden:

```
Prelude> :m +Data.Char +Numeric 
```


- **ganze Zahlen**
0,1,-3 Typ: **Int**
 n mit $|n| \leq 2^{31} - 1 = 2147483647$
 - **beliebig lange ganze Zahlen**
11122399387141 Typ: **Integer**,
 - **rationale Zahlen**
3%7 Typ: **Ratio**
 - **Gleitkommazahlen**
3.456e+10 Typ: **Floating**
 - **Zeichen**
'a' Typ: **Char**
 - **Datenkonstrukturen**
True, False Typ: **Bool**
- Diese nennen wir auch *Basiswerte* (bis auf **Floating**)

- **Arithmetische Operatoren:** $+, -, *, /$
(ein) Typ: $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
- **Arithmetische Vergleiche:** $==, <=, < \dots$
(ein) Typ: $\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$
- **Logische Operatoren:** $\&\&, ||, \text{not}$
(ein) Typ: $\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

Definition eines Polynoms, z.B.: $x^2 + y^2$:

```
quadratsumme x y = quadrat x + quadrat y
```

Auswertung:

```
... *Main> quadratsumme 3 4   
25
```


TYP	Beispiel-Ausdruck
Int	3
Integer	123
Float	1.23e45
Double	1.23e45
Integer -> Integer -> Integer	(+)
Integer -> Integer	quadrat
Integer -> Integer -> Integer	quadratsumme

Funktions-Typen:

(Typ von Argument 1) -> (Typ von Argument 2) -> ... -> Ergebnistyp

Beispiel

Die Ausgabe des Typs für die Addition (+):

```
Prelude> :t (+)   
(+) :: (Num a) => a -> a -> a
```

D.h.: Für alle Typen `a`, die man als numerisch klassifiziert hat,
d.h. die in der Typklasse `Num` sind,
hat `(+)` den Typ `a -> a -> a`

Zum Beispiel gilt:

`(+)` :: `Integer -> Integer -> Integer`

`(+)` :: `Double -> Double -> Double`

$\langle \text{FunktionsDefinition} \rangle ::= \langle \text{Funktionsname} \rangle \langle \text{Parameter} \rangle^* = \langle \text{Funktionsrumpf} \rangle$
 $\langle \text{Funktionsrumpf} \rangle ::= \langle \text{Ausdruck} \rangle$
 $\langle \text{Ausdruck} \rangle ::= \langle \text{Bezeichner} \rangle \mid \langle \text{Zahl} \rangle$
 $\quad \mid (\langle \text{Ausdruck} \rangle \langle \text{Ausdruck} \rangle)$
 $\quad \mid (\langle \text{Ausdruck} \rangle)$
 $\quad \mid (\langle \text{Ausdruck} \rangle \langle \text{BinInfixOp} \rangle \langle \text{Ausdruck} \rangle)$
 $\langle \text{Bezeichner} \rangle ::= \langle \text{Funktionsname} \rangle \mid \langle \text{Datenkonstruktorname} \rangle$
 $\quad \mid \langle \text{Parameter} \rangle \mid \langle \text{BinInfixOp} \rangle$
 $\langle \text{BinInfixOp} \rangle ::= * \mid + \mid - \mid /$

Argumente einer Funktion: *formale Parameter*.

Anzahl der Argumente: *Stelligkeit* der Funktion: $(ar(f))$

Die Nichtterminale

$\langle \text{Funktionsname} \rangle$, $\langle \text{Parameter} \rangle$, $\langle \text{Bezeichner} \rangle$, $\langle \text{Datenkonstruktorname} \rangle$
sind Namen (z.b. „quadrat“)

Aus der Haskell-Dokumentation (ohne Farben)

<http://www.hck.sk/users/peter/HaskellEx.htm>

<http://www.haskell.org/onlinereport/exps.html#sect3.2>

<code>exp10 -> \ apat1 ... apatn -> exp</code>	(lambda abstraction, $n \geq 1$)
<code> let decls in exp</code>	(let expression)
<code> if exp then exp else exp</code>	(conditional)
<code> case exp of { alts }</code>	(case expression)
<code> do { stmts }</code>	(do expression)
<code> fexp</code>	
<code>fexp -> [fexp] aexp</code>	function application)
<code>aexp -> qvar</code>	(variable)
<code> gcon</code>	(general constructor)
<code> literal</code>	
<code> (exp)</code>	(parenthesized expression)
<code> (exp1 , ... , expk)</code>	(tuple, $k \geq 2$)
<code> [exp1 , ... , expk]</code>	(list, $k \geq 1$)
<code> [exp1 [, exp2] .. [exp3]]</code>	(arithmetic sequence)

`quadratsumme x y = (quadrat x) + (quadrat y)`

Zeichenfolge

im Programm

`quadratsumme`

`x`

`y`

`=`

`(quadrat x) + (quadrat y)`

`+`

`quadrat x`

Name in der Grammatik

(man sagt auch: Nichtterminal)

⟨Funktionsname⟩

⟨Parameter⟩

⟨Parameter⟩

= gleiches Zeichen wie in Grammatik

⟨Funktionsrumpf⟩

hier ⟨Ausdruck⟩ der Form

⟨Ausdruck⟩ + ⟨Ausdruck⟩

binärer Infix-Operator

Anwendung: `quadrat` ist ein Ausdruck
und `x` ist ein Ausdruck

Ein *Haskell-Programm* ist definiert als

- Eine Menge von Funktionsdefinitionen
- Eine davon ist die Definition der Konstanten `main`.

Ohne `main`: Sammlung von Funktionsdefinitionen oder ein Modul

Prelude: vordefinierte Funktionen, Typen und Datenkonstruktoren

Module / Bibliotheken: z.B. Data.List : siehe Dokumentation

Präfix, Infix, Prioritäten: ist möglich für Operatoren

Konventionen zur Klammerung:

$s_1 \ s_2 \dots s_n \equiv ((\dots (s_1 \ s_2) \ s_3 \dots) \ s_n)$

Funktionsdefinitionen:

- formale Parameter müssen verschiedenen sein;
- keine undefinierten Variablen im Rumpf!

Weitere Trennzeichen: "{", "}" Semikolon "; "

Layout-sensibel: bewirkt Klammerung mit {, }.

Syntax: `if <Ausdruck> then <Ausdruck> else <Ausdruck>`

„if“, „then“, „else“ sind reservierte **Schlüsselworte**

Der erste Ausdruck ist eine **Bedingung** (Typ Bool)

Typisierung: `if Bool ... then typ else typ`

Syntax: `if <Ausdruck> then <Ausdruck> else <Ausdruck>`

„if“, „then“, „else“ sind reservierte **Schlüsselworte**

Der erste Ausdruck ist eine **Bedingung** (Typ Bool)

Typisierung: `if Bool ... then typ else typ`

```
(if 1 then 1 else 2)
```

ergibt einen (Typ-)Fehler

Die Infixoperatoren

`==, <, >, <=, >=, /=`

haben u.a. den Haskell-Typ: `Integer -> Integer -> Bool`

Achtung: `=` ist reserviert für Funktionsdefinitionen und `let`

Die Infixoperatoren

`==, <, >, <=, >=, /=`

haben u.a. den Haskell-Typ: `Integer -> Integer -> Bool`

Achtung: `=` ist reserviert für Funktionsdefinitionen und `let`

Boolesche Ausdrücke

sind kombinierbar mit `not, ||, &&` (nicht, oder, und)

Konstanten sind `True`, `False`.

Beispiel: `3.0 <= x && x < 5.0`

Kalender und Schaltjahre

Aufgabe: Berechne ob n ein Schaltjahr ist:

Bedingungen: Wenn n durch 4 teilbar, dann ist es ein Schaltjahr,
ansonsten ist n kein Schaltjahr.
Aber wenn es durch 100 teilbar ist, dann nicht.
Aber wenn es durch 400 teilbar ist, dann ist es
doch ein Schaltjahr.

Kalender und Schaltjahre

Aufgabe: Berechne ob n ein Schaltjahr ist:

Bedingungen: Wenn n durch 4 teilbar, dann ist es ein Schaltjahr,
ansonsten ist n kein Schaltjahr.
Aber wenn es durch 100 teilbar ist, dann nicht.
Aber wenn es durch 400 teilbar ist, dann ist es
doch ein Schaltjahr.

Erweiterung: Gilt erst nach dem Jahr 1582.
(Start des Gregorianischen Kalenders)
Wenn $n \leq 1582$, dann Abbruch.

Aufgabe: Berechne ob n ein Schaltjahr ist:

Bedingungen: Wenn n durch 4 teilbar, dann ist es ein Schaltjahr, ansonsten ist n kein Schaltjahr.
Aber wenn es durch 100 teilbar ist, dann nicht.
Aber wenn es durch 400 teilbar ist, dann ist es doch ein Schaltjahr.

Aufgabe: Berechne ob n ein Schaltjahr ist:

Bedingungen: Wenn n durch 4 teilbar, dann ist es ein Schaltjahr, ansonsten ist n kein Schaltjahr.
Aber wenn es durch 100 teilbar ist, dann nicht.
Aber wenn es durch 400 teilbar ist, dann ist es doch ein Schaltjahr.

Als Ausdruck: $n \bmod 4 == 0$

Aufgabe: Berechne ob n ein Schaltjahr ist:

Bedingungen: Wenn n durch 4 teilbar, dann ist es ein Schaltjahr,
ansonsten ist n kein Schaltjahr.
Aber wenn es durch 100 teilbar ist, dann nicht.
Aber wenn es durch 400 teilbar ist, dann ist es
doch ein Schaltjahr.

Als Ausdruck:
$$n \% 4 == 0$$
$$\&\& \text{ not } (n \% 100 == 0)$$

Aufgabe: Berechne ob n ein Schaltjahr ist:

Bedingungen: Wenn n durch 4 teilbar, dann ist es ein Schaltjahr,
ansonsten ist n kein Schaltjahr.
Aber wenn es durch 100 teilbar ist, dann nicht.
Aber wenn es durch 400 teilbar ist, dann ist es
doch ein Schaltjahr.

Als Ausdruck:

```
n % 4 == 0  
&& not (n % 100 == 0)  
|| (n % 400 == 0)
```


$3n + 1$ -Funktion `drn (.)`

Eingabe n : positive ganze Zahlen,

Ausgabe: 1, wenn `drn` Funktion terminiert.

Operation: Wenn $n = 1$, dann ausgeben.

Wenn n durch 2 teilbar, dann `drn` ($n/2$) aufrufen.

Sonst: `drn` ($3 * n + 1$) aufrufen.

$3n + 1$ -Funktion `drn (.)`

Eingabe n : positive ganze Zahlen,

Ausgabe: 1, wenn `drn` Funktion terminiert.

Operation: Wenn $n = 1$, dann ausgeben.

 Wenn n durch 2 teilbar, dann `drn ($n/2$)` aufrufen.

 Sonst: `drn ($3 * n + 1$)` aufrufen.

Beispiel: `drn 3`: 3, 10, 5, 16, 8, 4, 2, 1

Offenes Problem: Es ist nicht bekannt, ob die $3n + 1$ -Funktion immer terminiert.

Conway-Funktionen: Verallgemeinerung auf Teilbarkeit durch andere Zahlen.