Organisation

- Vorlesung: Di, Do
- Rechnerkennung
- Übungsbetrieb: Mo, Di, Mi
- Klausur
- Schein muss bis zum 3. Semester erworben werden, sonst durchgefallen.
- WWW Seite der Vorlesung

Inhalt der Vorlesung

- Was ist Informatik, Einführendes und Geschichtliches, der Algorithmusbegriff
- Grundkonzepte funktionaler Programmierung (Mengen, Funktionen, Terme, Typen, Rekursion, Polymorphie)
- Programmierung mit OCAML
- Datenstrukturen: Listen, Tupel, Reihungen, Bäume
- Formale Syntax und Semantik von Programmiersprachen
- Methodische Programmentwicklung, Modularisierung
- Algorithmik, Effizienz und Komplexität
- Denotationelle Semantik, Fixpunkttheorie
- Imperative Programmierung und Hoare Kalkül

Begleitliteratur

Die VL richtet sich nach F. Kröger: Kurzskriptum Informatik I, WS 02/03.

Weiterführende Literatur

- L. PAULSON, Standard ML for the working programmer, Cambridge University Press.
- Developing Applications with OCAML, O'Reilly. `caml.inria.fr/oreilly-book`
- GUMM-SOMMER: Einf. in die Informatik, Oldenbourg.
- GOOS: Vorlesungen über Informatik, Bd. I, Springer.

Kapitel 0. Einleitung

Was ist Informatik?

Von franz. *informatique* (= *information* + *mathématiques*).

Engl.: *computer science*, neuerdings auch *informatics*.

- DUDEN Informatik: Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Computern.
- Gesellschaft f. Inf. (GI): Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen.
- Association vor Computing Machinery (ACM): Systematic study of algorithms and data structures.

Teilbereiche der Informatik

- Technische Informatik
  Aufbau und Wirkungsweise von Computern
- Praktische Informatik
  Konstruktion von Informationsverarbeitungssystemen sowie deren Realisierung auf Computern
- Theoretische Informatik
  Theoretische und verallgemeinerte Behandlungen von Fragen und Konzepten der Informatik
- Angewandte Informatik
  Verbindung von Informatik mit anderen Wissenschaften

Typische Arbeitsgebiete

- Algorithmen und Datenstrukturen
- Betriebssysteme
- Bioinformatik
- Datenbanken
- Grafik
- Medieninformatik
- Programmiersprachen und Compiler
- Rechnerarchitektur
- Rechnernetze
- Robotik
- Simulation
- Softwareentwicklung
- Wirtschaftsinformatik

Kapitel I: Informationsverarbeitung durch Programme

Algorithmusbegriff

IBN MUSA AL CHWARIZMI schreibt um 900 das Lehrbuch *Kitab al jabr wal-muqābala* (Regeln der Wiedereinsetzung und Reduktion).

AL CHWARIZMI

Algorithmusbegriff

Algorithmus: Systematische, schematisch ausführbare Verarbeitungsvorschrift.

Alltagsalgorithmen: Kochrezepte, Spielregeln, schriftliches Rechnen.

Bedeutende Algorithmen: MP3-Komprimierung, RSA Verschlüsselung, Textsuche, Tomographie,. . .

Einfaches aber trotzdem wichtiges Beispiel: Berechnen der Fahrzeit eines Zuges aus Abfahrts- und Ankunftszeit.

Information und Daten

Ein Algorithmus hat ein oder mehrere **Eingaben** und **Ausgaben**, sowie möglicherweise **Zwischenergebnisse**.

Diese bezeichnet man als **Daten**, sie entsprechen **Informationen** über die reale Welt.

Innerhalb des Algorithmus werden die Daten als mathematische Objekte (z.B. Zahlen) kodiert.

Im Computer werden alle Daten als Binärzahlen kodiert.

Beispiel Fahrzeit

Die Eingabe ist hier die Abfahrtszeit und die Ankunftszeit: Wir kodieren sie jeweils als Paar von Zahlen $s_{\mathrm{ab}}, m_{\mathrm{ab}}$ und $s_{\mathrm{an}}, m_{\mathrm{an}}$, die jeweils Stunden und Minuten bezeichnen.

Die Ausgabe erfolgt in Minuten.

Andere mögliche Kodierung: Millisekunden seit dem 1.1.1970.

Algorithmus im Beispiel

Eingabe: vier natürliche Zahlen $s_{\mathrm{ab}}, m_{\mathrm{ab}}$ und $s_{\mathrm{an}}, m_{\mathrm{an}}$

Ausgabe: natürliche Zahl

Berechnung:

$$\text{Ergebnis} = (s_{\mathrm{an}} - s_{\mathrm{ab}}) \cdot 60 + m_{\mathrm{an}} - m_{\mathrm{ab}}.$$

Grundanforderungen an einen Algorithmus

- **Präzise Beschreibung:** Die Verarbeitungsvorschrift, sowie die zu verarbeitenden Daten müssen unmissverständlich aufgeschrieben sein. (nicht *Nach Gefühl Butter dazu*)
- **Effektivität:** Die Verarbeitung muss von der zugrunde liegenden Verarbeitungseinheit tatsächlich ausführbar sein. (nicht x *sei die zu* $s_{\mathrm{ab}}, m_{\mathrm{ab}}, s_{\mathrm{an}}, m_{\mathrm{an}}$ *gehörende Fahrtzeit*, auch nicht: *falls in der Dezimalbruchentwicklung von* π *vier aufeinanderfolgende 7er vorkommen*,...)

Eigenschaften von Algorithmen

- **terminierend für eine Eingabe**: jede mögliche Ausführung für diese Eingabe endet.
- **terminierend**: jede mögliche Ausführung für jede erlaubte Eingabe endet.
- **deterministisch**: die Reihenfolge und Art der auszuführenden Schritte ist eindeutig bestimmt (anderenfalls **nichtdeterministisch**)
- **determiniert**: das Ergebnis ist eindeutig durch die Eingabe bestimmt.
- **sequenziell**: die Verarbeitungsschritte werden der Reihe nach hintereinander ausgeführt
- **parallel (nebenläufig)**: gewisse Verarbeitungsschritte können gleichzeitig nebeneinander ausgeführt werden.

Arten von Algorithmen

- **Grundlegende Algorithmen** Lösen immer wieder vorkommende Grundaufgaben (kürzester Weg in einem Graphen, Sortieren, . . .) ohne direkten Bezug auf konkrete Anwendung.
- **Anwendungsalgorithmen** Lösen konkrete Aufgaben der realen Welt; deren Kenntnis (d.h. Bedeutung der Daten in der realen Welt) ist i.a. für die Entwicklung notwendig. Bestehen meist aus einem oder mehreren grundlegenden Algorithmen.

Programmierung

Ausformulierung eines Algorithmus so, dass er tatsächlich ausgeführt werden kann.

In OCAML (in Info I):

```
let fahrtzeit(s_ab,m_ab,s_an,m_an) =
       (s_an - s_ab) * 60 + m_an - m_ab;;
```

In Java (in Info II):

```
public static int fahrtzeit(int s_ab, int m_ab, int s_an, in
   return  (s_an - s_ab) * 60 + m_an - m_ab;
}
```

Fahrtzeit Algorithmus in ix386 Maschinensprache

```
pushl   %ebp
movl    %esp, %ebp
movl    16(%ebp), %eax
subl    8(%ebp), %eax
leal    (%eax,%eax,2), %edx
leal    (%edx,%edx,4), %eax
sall    $2, %eax
addl    20(%ebp), %eax
subl    12(%ebp), %eax
popl    %ebp
```

Pseudocode

- Halbformale Darstellung eines Algorithmus
- Teile können auf Deutsch oder in allgemeiner mathematischer Notation gegeben sein
- Enthält Vorbedingungen und Kommentare
- Erleichtert den Weg von Spezifikation, Idee zum echten ausführbaren Programm.

Darstellung in Pseudocode

algorithm Fahrtzeit

input $s_{\mathrm{an}}, s_{\mathrm{ab}}, m_{\mathrm{an}}, m_{\mathrm{ab}}$: **nat**

output : **nat**

pre Keine Fahrt geht über Mitternacht hinaus.

result Berechnung der Fahrtzeit in Minuten bei Abfahrt

um s_{ab} Uhr m_{ab} und Ankunft um s_{an} Uhr m_{an}.

begin

$(s_{\mathrm{an}} - s_{\mathrm{ab}}) \cdot 60 + m_{\mathrm{an}} - m_{\mathrm{ab}}$.

end

Programmiersprachen I

- Seit ca. 1970: Problemorientierte Programmiersprachen. Vorher: maschinennahe Sprachen wie Fortran, Basic oder Assembler.
- Imperative Sprachen (Pascal, C, C++, Modula, Java): Das Maschinenmodell liegt nach wie vor zugrunde. Das Programm definiert eine Folge von abzuarbeitenden Befehlen, die den Zustand der Daten verändern.
- Funktionale Sprachen (Lisp, ML (SML, OCAML), Scheme, Haskell, Miranda): Programme definieren mathematische Funktionen; es bleibt dem Compiler (Übersetzungsprogramm) überlassen, wie dies in Maschinenbefehle umgesetzt wird.

Programmiersprachen II

- Strukturierung durch Prozeduren und Funktionen (Pascal, C)
- Strukturierung durch Module: Datentypen plus zugehörige Funktionen. (Modula, SML, OCAML)
- Strukturierung durch Objekte: Daten plus zugehörige Funktionen (C++, Eiffel, Simula, Java, OCAML)

Programmiersprachen III

- Übersetzung eines ganzen Programms in Maschinensprache durch Übersetzungsprogramm (*Compiler*): Pascal, C, OCAML.
- Zeilenweise direkte Abarbeitung des Programms durch Verarbeitungsprogramm (*Interpreter*): Lisp, Shellscript, Perl, Python.
- Übersetzung eines ganzen Programms in Zwischensprache (*Bytecode*) der dann *interpretiert* wird: Java, OCAML, dotNET

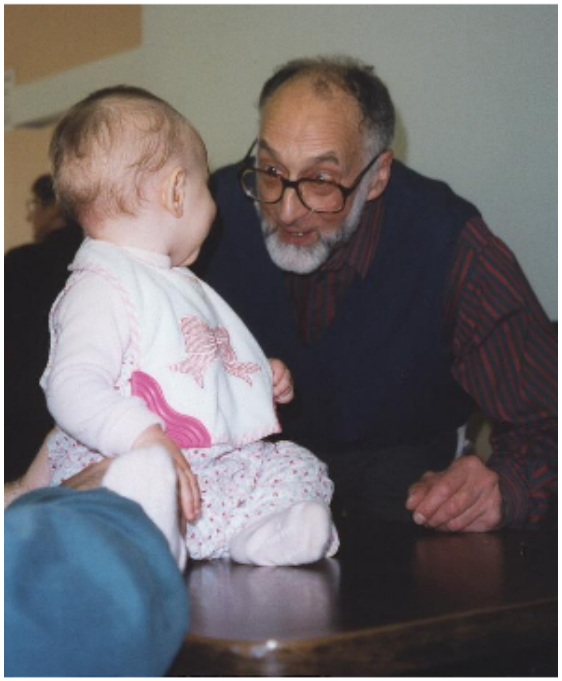
Programmiersprachen für Spezialanwendungen

- Postscript: dient der Erzeugung und Beschreibung von Dokumenten
- VRML: Beschreibung von 3D Szenen +
- HDL: Hardwarebeschreibung
- PROLOG, CHL: Beschreibung von Problemen als logische Formel. Computer sucht Lösung durch systematisches Probieren.

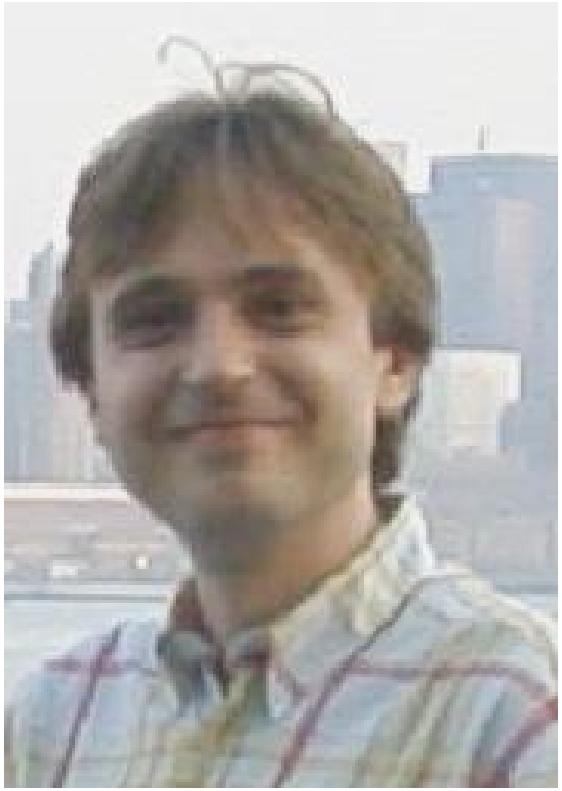
Geschichte von OCAML

- um 1973 von ROBIN MILNER in Edinburgh entwickelt als *Programmiersprache* (*MetaLanguage*) für den LCF Theorembeweiser.
- 1985 entwickelt GÉRARD HUET und andere CAML = *Categorical Abstract Machine + ML* als Implementierungssprache für den CoQ Theorembeweiser am INRIA, Frankreich.
- 1990 entwickelt XAVIER LEROY eine neue sehr effiziente Implementierung von CAML und fügt Objektorientierung hinzu. OCAML = Objective CAML.

Geschichte von OCAML



ROBIN MILNER



XAVIER LEROY

Nochmal die Uhrzeit

Jetzt möchten wir auch Fahrten über Mitternacht hinaus erlauben.

Wie muss die Ausgabe aussehen?

Wie berechnen wir die Ausgabe?

Lösung

Wir geben wiederum die Fahrzeit in Minuten aus.

Wir verwenden dieselbe Formel wie zuvor; bei negativem Ergebnis addieren wir $60 \cdot 24$.

Darstellung in Pseudocode

algorithm Fahrtzeit

input $s_{\text{an}}, s_{\text{ab}}, m_{\text{an}}, m_{\text{ab}}$: **nat**

output : **nat**

result Berechnung der Fahrtzeit in Minuten bei Abfahrt

um s_{ab} Uhr m_{ab} und Ankunft um s_{an} Uhr m_{an}.

begin

$$\text{zwErg} = (s_{\text{an}} - s_{\text{ab}}) \cdot 60 + m_{\text{an}} - m_{\text{ab}}.$$

$$\text{Ergebnis} = \begin{cases} \text{zwErg, falls zwErg} \geq 0 \\ \text{zwErg} + 60 \cdot 24, \text{ sonst} \end{cases}$$

end

Progammierung

In OCAML

```
let fahrtzeit(s_ab,m_ab,s_an,m_an) =
      let zwErg = (s_an - s_ab) * 60 + m_an - m_ab in
      if zwErg >= 0 then zwErg else zwErg + 60 * 24
```

In Java

```
public static int fahrtzeit(int s_ab, int m_ab, int s_an, in
   int zwErg = (s_an - s_ab) * 60 + m_an - m_ab;
   if (zwErg >=0)
      return zwErg;
   else
      return zwErg + 60 * 24;
}
```

In Assembler

```
        pushl   %ebp
         movl    %esp, %ebp
         movl    16(%ebp), %eax
         subl    8(%ebp), %eax
         leal    (%eax,%eax,2), %edx
         leal    (%edx,%edx,4), %eax
         sall    $2, %eax
         addl    20(%ebp), %eax
         subl    12(%ebp), %eax
         js      .L5
         popl    %ebp
.L5:
         addl    $1440, %eax
         popl    %ebp
```

Kapitel 2. Konzepte funktionaler Programmierung

Der Funktionsbegriff

A und B seien Mengen. Eine *Funktion* f von A nach B ist eine Zuordnung von *genau einem* Element $y = f(x)$ zu jedem Element x einer Teilmenge A' von A.

A' ist der *Definitionsbereich* von f.

Ist $A' \neq A$, so ist f eine partielle Funktion.

Man schreibt $A' = D(f)$.

Beispiele

$f : \mathbb{R} \to \mathbb{R}$

$f(x) = 1/x$

$D(f) = \mathbb{R} \setminus \{0\}$

$A = B =$ Endliche Folgen von 0en und 1en.

$$f(x) = \begin{cases} 0w, \text{ falls } x = 1w \text{ für ein } w \in A \\ \text{undefiniert sonst} \end{cases}$$

$D(f) = \{1w \mid w \in A\}$

$g : \mathbb{N} \to \mathbb{N}$

$$g(x) = \begin{cases} x/2, \text{ falls } x \text{ gerade} \\ 3x + 1, \text{ sonst} \end{cases}$$

$D(g) = \mathbb{N}$

Beispiele

$$f : \mathbb{N} \to \mathbb{N}$$

$$f(x) = \begin{cases} \text{das kleinste } n \in \mathbb{N} \text{ so dass } \underbrace{g(g(g(\ldots g}_{n \text{ Mal}}(x)\ldots) = 1, \text{ falls es existiert} \\ \\ \text{undefiniert sonst} \end{cases}$$

Es ist ein offenes Problem, ob $D(f) = \mathbb{N}$

Z.B.: $f(27) = 111$.

Terminologie

$f : A \to B, D(f) = A'$.

A heißt *Quelle*, B heißt *Ziel* von f.

Wenn $a \in D(f)$, so ist $f(a)$ der *Wert* der *Anwendung* von f auf das *Argument* a.

Man schreibt statt $f(a)$ manchmal auch

fa	Präfixnotation
af	Postfixnotation

Funktionen mit zwei Argumenten

Sind A_1 und A_2 Mengen, so bildet man das *kartesische Produkt*

$$A_1 \times A_2 = \{(a_1, a_2) \mid a_1 \in A_1 \text{ und } a_2 \in A_2\}$$

Ist $f : A_1 \times A_2 \to B$, so kann man f auf Paare anwenden.

Z.B.:

$$f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$f(x, y) = x + 2y^2$$

Man schreibt *nicht* $f((x, y))$.

Für solche Funktionen gibt es auch die *Infixnotation* $x\,f\,y$ für $f(x, y)$. Etwa, wenn $f = '+'$.

Eine Funktion von $A_1 \times A_2$ nach B heißt *zweistellig*.

Funktionen mit mehreren Argumenten

Sind $A_1, \ldots A_n$ Mengen, so bildet man das kartesische Produkt

$$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_i \in A_i \text{ für } i = 1 \ldots n\}$$

Die Elemente von $A_1 \times, \ldots, \times A_n$ heißen *n-Tupel* (Verallg. von Tripel, Quadrupel, Quintupel, Sextupel,...).

Zum Beispiel

$$\text{fahrtzeit} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

Solch eine Funktion heißt *n-stellig* (Vokabeln: *Stelligkeit*, *n-ary*, *arity*)

Nur äußerst selten ist $n > 6$.

Kartesische Produkte als Ziel

Es gibt auch Funktionen, die Paare oder gar n-Tupel zurückliefern.

$$\text{divmod} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$

$$D(\text{divmod}) = \mathbb{N} \times (\mathbb{N} \setminus \{0\})$$

$$\text{divmod}(a, b) = (q, r), \text{ wobei } a = qb + r \text{ und } q, r \in \mathbf{N} \text{ und } r < b$$

Typen

Typen = Mengen gleichartiger Daten, die von Funktionen verarbeitet werden.

Basistypen: $\mathbb{N}, \mathbb{R}, \mathbb{Z}, \mathbb{B} = \{\text{true}, \text{false}\}, \ldots$

Zusammengesetzte Typen: $A_1 \times \cdots \times A_n$

In Pseudocode schreiben wir auch
`nat, real, int, bool, A_1 * ... * A_n`.

Terme

Terme sind Ausdrücke mit Variablen (Platzhaltern).

Beispiel eines Terms: $(s_{\mathrm{an}} - s_{\mathrm{ab}}) \cdot 60 + m_{\mathrm{an}} - m_{\mathrm{ab}}$.

Dieser Term enthält die Variablen $s_{\mathrm{ab}}, m_{\mathrm{ab}}$ und $s_{\mathrm{an}}, m_{\mathrm{an}}$

Kennt man die Werte der Variablen, die in einem Term vorkommen, so kann man den Wert des Terms ausrechnen.

Damit der Term sinnvoll ist, muss man zu jeder Variablen angeben, aus welcher Menge ihre Werte kommen sollen.

Man bezeichnet diese Menge als den *Typ* der Variablen.

Außerdem: *Typ des Terms* = Menge aus der die Werte des Terms zu gegebenen Werten der Variablen entstammen.

Im Beispiel sollen $s_{\mathrm{ab}}, m_{\mathrm{ab}}$ und $s_{\mathrm{an}}, m_{\mathrm{an}}$ jeweils den Typ $\mathbb{N}$ haben. Der Typ des Terms für die Fahrzeit hat auch den Typ $\mathbb{N}$.

Beispiele

Sei x eine Variable vom Typ $\mathbb{R}$ und p eine Variable vom Typ $\mathbb{Z}$.

Was sind die Typen der Terme $x + x$, 1, $\lceil x \rceil$, x^p?

Beachte: für bestimmte erte kann ein Term undefiniert sein, etwa x/y für $y = 0$.

Funktionen aus Termen

Seien $x_1, \ldots, x_n$ Variablen vom Typ $A_1, \ldots, A_n$ (jeweils).

Sei t ein Term vom Typ B, der alle oder einige dieser Variablen enthält.

Wir bilden eine Funktion

$\mathbf{function}(x_1, \ldots, x_n)t : A_1 \times \cdots \times A_n \to B$

$(\mathbf{function}(x_1, \ldots, x_n)t)(a_1, \ldots, a_n) =$ Wert des Terms t, falls x_i den Wert a_i hat.

Beispiel:
fahrtzeit $= \mathbf{function}(s_{\text{ab}}, m_{\text{ab}}, s_{\text{an}}, m_{\text{an}})(s_{\text{an}} - s_{\text{ab}}) \cdot 60 + m_{\text{an}} - m_{\text{ab}}$

Man schreibt statt $\mathbf{function}(x_1, \ldots, x_n)t$ auch $\lambda(x_1, \ldots, x_n)t$ (vgl. Logo des TCS Lehrstuhls).

Funktionstypen

Mit $A_1 \times \cdots \times A_n \to B$ bezeichnet man die Menge der Funktionen von $A_1 \times \cdots \times A_n$ nach B.

Solche Mengen können als Typ von Variablen und als Typ von Termen auftreten.

Zum Beispiel ist $\mathbf{function}(x)x$ ein Term vom Typ $A \to A$ falls x eine Variable vom Typ A ist.

Freie Variablen

Der Term $\mathbf{function}(x)x$ enthält keine Variablen in dem Sinne, dass sein Wert von ihnen abhängt.

Die Variable x ist nämlich in $\mathbf{function}(x)x$ *gebunden*.

Ebenso ist x in $\int_0^1 e^x \mathrm{d}x$ gebunden.

Variablen, die “echt” in einem Term vorkommen, bezeichnet man als *freie* Variablen.

Freie Variablen in Funktionstermen

Ein Funktionsterm kann trotzdem freie Variablen enthalten:

$$\mathbf{function}(x)x + y$$

enthält die Variable y. Der Wert des Terms hängt vom Wert der Variablen y ab (und ist dann die Funktion “addiere y”).

Anderes Beispiel: der Term $\int_0^\infty e^{-st} f(t)\mathrm{d}t$ enthält die Variable s vom Typ $\mathbb{R}$ und die Variable f vom Typ $\mathbb{R} \to \mathbb{R}$.

Lokale Definitionen

Terme dürfen auch informelle deutsche Beschreibungen enthalten:

das kleinste gemeinsame Vielfache von x und y

ist ein Term mit den Variablen x und y.

$x + y$, wobei $y = 10$

ist ein Term mit der Variablen x. Die Variable y ist wiederum *gebunden*.

0, falls $x \leq 0$; 1, falls $x > 0$

ist ein Term mit der freien Variablen x.

Standardnotation

In Pseudocode (und später in der Programmierung) verwendet man gern standardisierte Notation für solche Terme.

Statt

t_1, wobei $x = t_2$

schreiben wir

let $x = t_2$ **in** t_1

Statt

t_1, falls Bedingung A und t_2 sonst

schreiben wir

if Bedingung A **then** t_1 **else** t_2

Beispiel

Erweiterte Fahrzeitberechnung:

$$\begin{array}{l}\textbf{function}(s_{\mathrm{ab}}, m_{\mathrm{ab}}, s_{\mathrm{an}}, m_{\mathrm{an}}) \\ \quad \textbf{let } \mathit{zwErg} = (s_{\mathrm{an}} - s_{\mathrm{ab}}) \cdot 60 + m_{\mathrm{an}} - m_{\mathrm{ab}} \textbf{ in} \\ \quad \textbf{let } \mathit{minProTag} = 24 \cdot 60 \textbf{ in} \\ \quad \textbf{if } \mathit{zwErg} \geq 0 \textbf{ then } \mathit{zwErg} \textbf{ else } \mathit{zwErg} + \mathit{minProTag}\end{array}$$

Beachte: im Skript werden **let** Konstrukte mit **end** abgeschlossen. Wir machen das nicht, es ist aber auch kein Fehler.

Formulierung von Bedingungen

Eine *Bedingung* ist ein Term vom Typ $\mathbb{B} = \{\text{true}, \text{falsch}\}$ (im Pseudocode geschrieben `bool`).

Zur Formulierung von Bedingungen verwenden wir z.B. die Vergleichsoperationen

$$<, >, \leq, \geq \colon \mathbb{N} \times \mathbb{N} \to \mathbb{B}$$

Für diese verwendet man die Infixnotation, also $3 < x$ statt $< (3, x)$.

Bedingungen kann man mit den logischen Operationen $\wedge, \vee, \neg$ zusammensetzen.

$x < 7 \wedge x > 8$.

Typannotate

Man kann Quelle und Ziel eines Funktionsterms auch in diesen aufnehmen:

Statt

$$f : A_1 \times \cdots \times A_n \to B$$
$$f = \mathbf{function}(x_1, \ldots, x_n)t$$

schreiben wir auch

$$f = \mathbf{function}(x_1{:}A_1, \ldots, x_n{:}A_n)B\ t$$

oder

$$f = \mathbf{function}(x_1{:}A_1, \ldots, x_n{:}A_n)B{:}t$$ Notation aus dem Skript

Beispiele

$$\textbf{function}(s_{\mathrm{ab}:\textbf{nat}}, m_{\mathrm{ab}}:\textbf{nat}, s_{\mathrm{an}}:\textbf{nat}, m_{\mathrm{an}}:\textbf{nat})\textbf{nat}$$

let *zwErg* $= (s_{\mathrm{an}} - s_{\mathrm{ab}}) \cdot 60 + m_{\mathrm{an}} - m_{\mathrm{ab}}$ **in**

let *minProTag* $= 24 \cdot 60$ **in**

if *zwErg* ≥ 0 **then** *zwErg* **else** *zwErg* $+$ *minProTag*

Rekursive Funktionen

Man kann eine Funktion $f : A \to B$ durch einen Term definieren, der selbst Aufrufe von f enthält.

Beispiel:

$$\text{fakultät} = \mathbf{function}(n)\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \cdot \text{fakultät}(n-1)$$

Dies bezeichnet man als *rekursive Definition*.

Wie man formell den Wert einer rekursiv definierten Funktion (kurz: rekursiven Funktion) bestimmt, sehen wir später.

Jetzt rechnen wir einfach aus:

$$\text{fakultät}(3) = 3{\cdot}\text{fakultät}(2) = 3{\cdot}2{\cdot}\text{fakultät}(1) = 3{\cdot}2{\cdot}1{\cdot}\text{fakultät}(0) = 3{\cdot}2{\cdot}1{\cdot}1 = 6$$

Beachte: Rekursion ist eine Quelle von Undefiniertheit:

Wenn

$$f = \mathbf{function}(n)\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ f(n+1)$$

dann ist $f(0) = 1$ und $f(n)$ undefiniert für $n > 0$. Also $f : \mathbb{N} \to \mathbb{N}$, aber $D(f) = \{0\}$.

Mehr Rekursion

Die Fibonacci-Zahlen:

$$\text{fib} : \mathbb{N} \to \mathbb{N}$$

$$\text{fib} = \textbf{function}(n)\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else if } n = 1 \textbf{ then } 1 \textbf{ else } \text{fib}(n-1) + \text{fib}(n-2)$$

Interpretation: $\text{fib}(n) =$ Hasenpopulation nach n Monaten unter der Annahme, dass Hasen jeden Monat einen Nachkommen haben, dies aber erst ab dem zweiten Lebensmonat.

$$\text{fib}(n) \sim (\frac{\sqrt{5}+1}{2})^n$$

Nochmal 3n+1

$$g = \textbf{function}(n)\textbf{if } n \bmod 2 = 0 \textbf{ then } n/2 \textbf{ else } 3n+1$$

$$f = \textbf{function}(n)\textbf{if } n = 1 \textbf{ then } 0 \textbf{ else } 1 + f(g(n))$$

Türme von Hanoi

Es gibt drei senkrechte Stäbe. Auf dem ersten liegen n gelochte Scheiben von nach oben hin abnehmender Größe.

Man soll den ganzen Stapel auf den dritten Stab transferieren, darf aber immer nur jeweils eine Scheibe entweder nach ganz unten oder auf eine größere legen.

Angeblich sind in Hanoi ein paar Mönche seit Urzeiten mit dem Fall $n = 64$ befasst.

Lösung

Für $n = 1$ kein Problem.

Falls man schon weiß, wie es für $n - 1$ geht, dann schafft man mit diesem Rezept die obersten $n - 1$ Scheiben auf den zweiten Stab (die unterste Scheibe fasst man dabei als “Boden” auf.).

Dann legt man die größte nunmehr freie Scheibe auf den dritten Stapel und verschafft unter abermaliger Verwendung der Vorschrift für $n - 1$ die restlichen Scheiben vom mittleren auf den dritten Stapel.

Lösung in Pseudocode

Turm $= \{1, 2, 3\}$

Befehle $= \{(i, j) \mid i, j \in$ Turm$, i \neq j\}$

Befehlsfolge $= \{\vec{b} \mid$ es gibt n so dass $\vec{B} \in$ Befehle$^n\}$

Lösung : $\mathbb{N} \times$ Turm $\times$ Turm $\to$ Befehlsfolge

Lösung $= \mathbf{function}(n, i, j)$

Liefert Befehlsfolge zum Transfer von n Scheiben von i nach j

if $i = j$ **then** leere Befehlsfolge

else let $k = 6 - i - j$ **in**

Lösung$(n-1, i, k)\hat{}(i, j)\hat{}$Lösung$(n-1, k, j)$

Technik der Einbettung

Im Beispiel mussten wir eine allgemeinere Funktion rekursiv definieren.

Versuchen wir, nur die Funktion “verschiebe von 1 nach 2” zu definieren, dann ergibt sich keine rekursive Lösung.

Häufig muss man vor einer rekursiven Lösung das Problem generalisieren.

Diese Technik bezeichnet man als *Einbettung*

Beispiele von Einbettung: Primzahlen

Man soll bestimmen, ob n eine Primzahl ist.

Gesucht istPrim : **nat** $\to$ **bool** mit istPrim$(n) =$ **true** gdw., n prim.

istPrim $=$ **function**(n)**if** $n = 0 \vee n = 1$ **then false else if** $n = 2$ **then true else** ?

Primzahlen

keineTeiler $=$ **function**(n:**nat**, k:**nat**)**bool**

pre Stellt fest, ob n keine Teiler im Bereich $k \ldots n-1$ hat

if $k \geq n-1$ **then true**

else $\neg(k \mid n) \wedge$ keineTeiler$(n, k+1)$

istPrim $=$ **function**(n:**nat**)**bool**

$n > 1 \wedge$ keineTeiler$(n, 2)$

Beispiel: Binäre Suche

Man soll ein Wort im Lexikon suchen.

suche : Lexikon $\times$ Wort $\longrightarrow$ **bool**

suche$(l, w) =$ "w kommt in l vor"

Noch nicht detailliert genug.

Wir nehmen an, es gibt eine Funktion

ntesWort : Lexikon $\longrightarrow$ Wort

die das n-te Wort im Lexikon liefert.

Lösung eins: alle durchprobieren

$$\text{sucheBis} = \mathbf{function}(l, w, n{:}\mathbf{nat})\mathbf{bool}$$

$$\quad \mathbf{if}\ n = 0\ \mathbf{then\ false\ else}$$

$$\quad\quad \text{sucheBis}(l, w, n-1) \vee w = \text{ntesWort}(l, n)$$

$$\text{suche} = \mathbf{function}(l, w)$$

$$\quad \text{sucheBis}(l, w, \text{anzahlWörter}(l))$$

Lösung zwei: binäre Suche

```
sucheVonBis = function(l, w, i:int, j:int)bool
    if i > j then false else
    if i = j then ntesWort(l, i) = w else
        let m = ⌊(i + j)/2⌋ in
        let w_m = ntesWort(l, m) in
        if w_m = w then true else
            if w kommt vor ntesWort(l, m) then
                sucheVonBis(l, w, i, m − 1)
            else sucheVonBis(l, w, m + 1, j)
suche = function(l, w)
    sucheVonBis(l, w, 1, anzahlWörter(l))
```

Abstiegsfunktion

Um festzustellen, ob eine rekursiv definierte Funktion für ein Argument definiert ist, kann man eine Abstiegsfunktion verwenden.

Sei

$$f : A \to B$$
$$f = \mathbf{function}(x)\Phi(f, x)$$

eine rekursive Definition einer Funktion $f : A \to B$.

$\Phi(f, x)$ bezeichnet hier den definierenden Term, der sowohl f, als auch x enthält.

Sei $A' \subseteq A$ eine Teilmenge von A und werde in $\Phi(f, x)$ die Funktion f nur für Argumente $y \in A'$ aufgerufen.

Sei $\Phi(f, x)$ immer definiert, wenn $x \in A'$ und $f(y)$ definiert ist für alle Aufrufe $f(y)$ in $\Phi(f, x)$.

Dann muss noch nicht unbedingt gelten $A' \subseteq D(f)$.

Abstiegsfunktion

Sei nun zusätzlich $m : A \to \mathbb{N}$ eine Funktion mit $A' \subseteq D(m)$ mit der folgenden Eigenschaft:

Im Term $\Phi(f, x)$ wird f nur für solche $y \in A'$ aufgerufen, für die gilt $m(y) < m(x)$.

Dann ist $A' \subseteq D(f)$.

Man bezeichnet so ein m als *Abstiegsfunktion*.

Beispiel Fakultät

$$\text{fakultät}(n{:}\textbf{nat}) = \textbf{if } n{=}0 \textbf{ then } 1 \textbf{ else } n \cdot \text{fakultät}(n-1)$$

Hier nehmen wir $A' = \textbf{nat}$ und $m(x) = x$.

Beispiel keineTeiler

keineTeiler $=$ **function**(n:**nat**, k:**nat**)**bool**

pre Stellt fest, ob n keine Teiler im Bereich $k \ldots n-1$ hat

if $k \geq n$ **then true**

else $\neg k \mid n \wedge$ keineTeiler$(n, k+1)$

Hier setzen wir $A' =$ **nat** $\times$ **nat** und $m(n,k) =$ **if** $k \geq n$ **then** 0 **else** $n-k$.

Wohlfundierte Relationen

Definition Sei M eine Menge. Eine Relation $R \subseteq M \times M$ ist wohlfundiert, wenn es keine unendliche Folge $a_1, a_2, a_3, \ldots$ von Elementen in M gibt sodass $a_{i+1} R a_i$ für alle $i \geq 1$.

Ist R eine wohlfundierte Relation auf einer Menge M, so kann man anstelle einer Abstiegsfunktion $m : A \to \mathbb{N}$ auch eine Abstiegsfunktion $m : A \to M$ wählen, derart dass $m(y) R m(x)$ wenn $f(y)$ in $\Phi(f, x)$ aufgerufen wird.

Beispiele

$M = \mathbb{N}$ und $xRy \Leftrightarrow x < y$.

$M = \mathbb{N}$ und $xRy \Leftrightarrow y = x + 1$

$M = \mathbb{N} \times \mathbb{N}$ und $(x_1, x_2)R(y_1, y_2) \Leftrightarrow x_1 < y_1 \vee x_1 = x_2 \wedge y_1 < y_2$.

Mit dieser wohlfundierten Relation kann man die Ackermannfunktion rechtfertigen:

$$\begin{array}{l} \text{ackermann} = \textbf{function}(x{:}\textbf{nat}, y{:}\textbf{nat})\textbf{nat} \\ \quad \textbf{if } x = 0 \textbf{ then } y + 1 \textbf{ else} \\ \quad\quad \textbf{if } y = 0 \textbf{ then } \text{ackermann}(x - 1, 1) \\ \quad\quad\quad \textbf{else } \text{ackermann}(x - 1, \text{ackermann}(x, y - 1)) \end{array}$$

Hier wählen wir die Abstiegsfunktion $m(x, y) = (x, y)$.

Verschränkte Rekursion

Manchmal rufen sich zwei Funktionen gegenseitig rekursiv auf. Das ist *verschränkte Rekursion*.

Beispiel

$$\text{gerade} = \mathbf{function}(x{:}\mathbf{nat})\mathbf{if}\ x = 0\ \mathbf{then\ true\ else}\ \text{ungerade}(x-1)$$

$$\text{ungerade} = \mathbf{function}(x{:}\mathbf{nat})\mathbf{if}\ x = 0\ \mathbf{then\ false\ else}\ \text{gerade}(x-1)$$

Es ist $\text{gerade}(x) = (x \bmod 2{=}0)$.

Verschränkte Rekursion über kartesische Produkte

Man kann verschränkte Rekursion durch Produkte simulieren:

$$\begin{array}{l}\text{gerade/ungerade} = \mathbf{function}(x{:}\mathbf{nat})\mathbf{bool} \times \mathbf{bool} \\ \quad \mathbf{if}\ x{=}0\ \mathbf{then}\ (\mathbf{true}, \mathbf{false})\ \mathbf{else} \\ \quad\quad \mathbf{let}\ (g,u) = \text{gerade/ungerade}(x-1)\ \mathbf{in} \\ \quad\quad (u,g)\end{array}$$

Stimmen die Quellen der beiden verschränkt rekursiven Funktionen nicht überein, dann muss man das Produkt der beiden Quellen nehmen.

Diese Simulation legt auch nahe, was von einer Abstiegsfunktion für verschränkt rekursive Funktionen zu fordern ist.

Induktionsbeweise

So wie man eine Funktion rekursiv definieren kann, also durch “Rückgriff” auf andere (hoffentlich schon bekannte) Funktionswerte, so kann man eine Behauptung dadurch beweisen, dass man sie für andere Fälle als bereits bewiesen voraussetzt (*rekursiver Beweis*).

Natürlich muss man dann argumentieren, dass die Kette der rekursiven Rückgriffe irgendwann abbricht, wozu sich wiederum die Abstiegsfunktion anbietet.

Ein solcher rekursiver Beweis mit Abstiegsfunktion ist ein *Induktionsbeweis*.

Induktionsprinzip

Sei

- R eine wohlfundierte Relation auf einer Menge M,
- $m : A \to M$ eine Funktion,
- $P \subseteq A$ eine Teilmenge von A.

Falls für alle $a \in A$ gilt

"a ist in P unter der Annahme, dass alle $y \in A$ mit $m(y)Rm(a)$ in P sind"

dann ist $P = A$.

Beweis des Induktionsprinzips

Äquivalente Formulierung der Bedingung:

"Falls $a \notin P$ dann existiert $y \in A$ mit $m(y)Rm(x)$ und $y \notin P$".

Ein einziges Gegenbeispiel $a \notin P$ zieht also eine unendlich lange Kette von Gegenbeispielen nach sich im Widerspruch zur Wohlfundiertheit von R.

Induktion

Oftmals ist $A = \mathbb{N}$ und $m(n) = n$ und xRy, falls $y = x + 1$.

Hier muss man $0 \in P$ ohne Voraussetzungen zeigen.

Bei $a = y + 1$ darf man aber $y \in P$ schon voraussetzen.

Beispiel

Behauptung: Jedes nur denkbare Verfahren zum Transfer von n Scheiben braucht mindestens $2^n - 1$ Befehle.

Sei P die Menge derjenigen Zahlen n für die das gilt.

$0 \in P$ ist klar, da $2^0 - 1 = 0$.

Sei jetzt $n > 0$. Irgendwann wurde die größte Scheibe verlegt. Dazu aber müssen $n - 1$ Scheiben weggeschafft worden sein (auf den Hilfsstapel). Nach Annahme kostet das mindestens $2^{n-1} - 1$ Befehle. Danach müssen die $n - 1$ Scheiben auf die größte verschafft werden: wieder $2^{n-1} - 1$ Befehle. Insgesamt also $2 \cdot 2^{n-1} - 2 + 1 = 2^n - 1$ Befehle.

Beispiel

Sei $\phi = \frac{\sqrt{5}+1}{2}$. Beachte: $\phi^2 = \phi + 1$.

Behauptung: Es ist $\text{fib}(n) = a\phi^n + b(-\phi)^{-n}$ wobei $a + b = 1$ und $a\phi - b(1/\phi) = 1$.

Sei P die Menge der n für die das wahr ist. Wir wählen $m(x) = x$ und $yRx \Leftrightarrow y < x$.

Es ist $0 \in P$ und $1 \in P$ (Nach Def. von a, b; bei Zweifel nachrechnen)

Wenn $n \geq 2$, dann
$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) = a\phi^{n-1} + b(-\phi)^{-n+1} + a\phi^{n-2} +$
$b(-\phi)^{-n+2} = a\phi^n(\phi^{-1} + \phi^{-2}) + b(-\phi)^{-n}(-\phi + \phi^2) = a\phi^n + b(-\phi)^{-n}$.

Also $n \in P$ und $P = \mathbb{N}$.

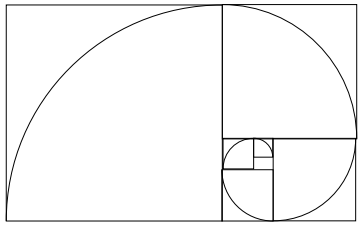
Kettenbruch und Kettenwurzel

Es ist $\phi^2 = 1 + \phi$, also $(1/\phi) = \frac{1}{1+(1/\phi)}$, also

$$1/\phi = \cfrac{1}{1 + \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\ddots}}}}}$$

Außerdem $\phi = \sqrt{1+\phi}$, also $\phi = \sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\ldots}}}}$.

Die Zahl $\phi = \frac{\sqrt{5}+1}{2}$ heißt Goldener Schnitt.



Pflasterung

Es gelte, ein $2 \times n$ Rechteck mit Dominos der Größe 2×1 zu pflastern.

Behauptung: Die Anzahl der Möglichkeiten, das zu tun beträgt $\text{fib}(n)$.

Sei P die Menge der n, für die das gilt.

Sei n fest aber beliebig vorgegeben. Wenn $n \leq 1$, dann $n \in P$.

Wenn $n > 1$, dann gibt es zwei Möglichkeiten, die linke obere Ecke[a] zu pflastern. Entweder mit einem senkrechten oder mit einem waagrechten Domino. Im ersten Fall bleibt ein Rechteck der Größe $2 \times (n-1)$ zu pflastern. Da $n - 1 \in P$ gibt es dafür $\text{fib}(n-1)$ Möglichkeiten. Im anderen Fall ist man gezwungen auf die linke untere Ecke auch ein waagrechtes Domino zu legen—es bleibt ein Rechteck der Größe $2 \times (n-2)$, welches man auf $\text{fib}(n-2)$ Arten pflastern kann.

Insgesamt hat man also $\text{fib}(n-1) + \text{fib}(n-2) = \text{fib}(n)$ Möglichkeiten und es ist $n \in P$.

[a] Höhere Mächte befahlen: Linke obere Ecke schwarz malen (SIGMAR POLKE)

Exponentiation

$$\exp = \textbf{function}(x\text{:}\textbf{real}, n\text{:}\textbf{nat})\textbf{real}$$

if $n{=}0$ **then** 1

else if n gerade **then** $\exp(x, n \text{ div } 2)^2$

else $\exp(x, n \text{ div } 2)^2 \cdot x$

Hier ist x div 2 die ganzzahlige Division, z.B., 5 div $2 = 2$.

Man beweise: $\exp(x, n) = x^n$.

Polymorphe Funktionen

Manchmal gestattet ein Term mehrere Typisierungen. Man kann ihm dann einen Typ mit Typvariablen zuweisen.

Typvariablen bezeichnen wir mit griechischen Buchstaben α, β, γ.

Ein Typ mit Typvariablen heißt *polymorpher Typ* (auch *Polytyp*).

Ein Typ ohne Typvariablen heißt auch *monomorpher Typ* (auch *Monotyp*).

Beispiele:

$$\text{erstes} : \alpha \times \beta \to \alpha$$
$$\text{erstes} = \mathbf{function}(x, y)x$$

$$\text{tausch} : \alpha \times \beta \to \beta \times \alpha$$
$$\text{tausch} = \mathbf{function}(x{:}\alpha, y{:}\beta)(y, x)$$

Polymorphe Funktionen

Man kann eine polymorphe Funktionen $f : \mathit{typ} \to \mathit{typ}'$ mit einem Argument x aufrufen, falls es eine Ersetzung der Typvariablen in *typ* gibt, sodass der Typ von x herauskommt.

Das Ergebnis des Aufrufes $f(x)$ hat den den Typ, der sich durch dieselbe Ersetzung der Typvariablen in typ' ergibt.

$$\text{tausch}(7, 13) = (13, 7) : \textbf{nat} \times \textbf{nat}$$

$$\text{tausch}(\textbf{true}, -13) = (-13, \textbf{true}) : \textbf{int} \times \textbf{bool}$$

$$\text{tausch}(3.14, \textbf{false}) = (\textbf{false}, 3.14) : \textbf{bool} \times \textbf{real}$$

Funktionen höherer Ordnung

Wir haben schon gesehen, dass Funktionen als Argumente anderer Funktionen auftreten können.

Funktionen können auch als Wert zurückgegeben werden:

$$\begin{aligned}\text{plus} = \mathbf{function}(x)\\ \quad \mathbf{function}(y)x + y\end{aligned}$$

$\text{plus}(2) = \mathbf{function}(y)y + 2$, also die Funktion “addiere zwei dazu”.

Funktionen höherer Ordnung

Allgemein kann man zu jeder Funktion

$$f_1 : typ_1 \times \cdots \times typ_n \to typ$$

eine Funktion

$$f_2 : typ_1 \to typ_2 \to \cdots \to typ_n \to typ$$

definieren durch

$$f_2 = \mathbf{function}(x_1)\mathbf{function}(x_2)\ldots\mathbf{function}(x_n)f_1(x_, \ldots, x_n)$$

Es ist

$$f_2\ x_1\ x_2\ \ldots x_n = f_1(x_1, \ldots, x_n)$$

Wir bezeichnen f_2 als das *Currying* von f_1 und f_1 als das *Uncurrying* von f_2. (Nach dem Logiker HASKELL CURRY (1900-1982))

Funktionen höherer Ordnung

CURRY (1900–1982)

SCHÖNFINKEL

Schönfinkel gilt als der eigentliche Erfinder des “Currying”.

Iteration und Komposition

$$\text{iteriere} = \mathbf{function}(f{:}\boldsymbol{\alpha} \to \boldsymbol{\alpha}, x{:}\boldsymbol{\alpha}, n{:}\mathbf{nat})\boldsymbol{\alpha}$$
$$\mathbf{if}\ n = 0\ \mathbf{then}\ x\ \mathbf{else}\ \text{iteriere}(f, f(x), n-1)$$

Es ist

$$\text{iteriere}(f, x, n) = \underbrace{f(f(f(\ldots f}_{n\text{ Mal}}(x)\ldots)$$

Andere Notation: $\text{iteriere}(f, x, n) = f^n(x)$.

$$\text{komponiere} = \mathbf{function}(g{:}\boldsymbol{\beta} \to \boldsymbol{\gamma}, f{:}\boldsymbol{\alpha} \to \boldsymbol{\beta})\boldsymbol{\alpha} \to \boldsymbol{\gamma}$$
$$\mathbf{function}(x{:}\boldsymbol{\alpha})g(f(x))$$

Andere Notation: $\text{komponiere}(g, f) = g \circ f$.

Noch ein Beispiel zur Induktion

Seien $f : A \to B$ und $g : B \to A$ beliebige Funktionen.

Für alle n ist

$$g(\text{iteriere}(\text{komponiere}(f, g), x, n)) = \text{iteriere}(\text{komponiere}(g, f), g(x), n)$$

Kapitel 3. Funktionale Programmierung in OCAML

Arbeitsweise von OCAML

OCAML hat zwei Modi:

- Interaktiver Modus: Man gibt Definitionen ein; OCAML wertet sie aus und zeigt den Wert an.
- Compilierender Modus: Man schreibt ein OCAML Programm in eine oder mehrere Datei. Der OCAML Compiler übersetzt sie und liefert ein ausführbares Programm (“EXE-Datei”).

Wir befassen uns hauptsächlich und zunächst mit dem interaktiven Modus.

OCAML-Sitzung

Eröffnen einer OCAML Sitzung durch Eingabe von `ocaml`.

Es erscheint die Ausgabe:

```
        Objective Caml version 3.06

#
```

Das Zeichen # ist ein *Prompt*. Es fordert uns auf, eine Eingabe zu tätigen.

```
# let a = 18.35;;
val a : float = 18.35
# let aquadrat = a *. a;;
val aquadrat : float = 336.7225
# let b = -0.31 /. a;;
val b : float = -0.01689373297
```

Die Eingaben nach dem Prompt wurden vom Benutzer getätigt, die mit `val` beginnenden Zeilen sind Ausgaben von OCAML.

OCAML und Xemacs

Mit `xemacs` rufen Sie den Xemacs-Editor auf.

Jede Student(in) der Informatik sollte Xemacs kennen.

Mit dem Kommando `M-x shell` machen Sie ein Terminal Fenster auf.

Dort geben Sie das Kommando

```
export TERM=xemacs;ocaml
```

ein. Dann können Sie wie gewohnt mit OCAML arbeiten, haben aber die Möglichkeit, mit `M-p` und `M-n` die letzten Eingaben wieder heraufzuholen und die Kommandozeile zu editieren.

Hinweis: die Notation `M-x` bedeutet gleichzeitiges Drücken von `Alt` und `x`.

Fließkommaarithmetik in OCAML

Der OCAML Datentyp `float` umfasst *Fließkommazahlen*.

Das sind “reelle” Zahlen, wie sie im Taschenrechner vorkommen, z.B. `17.82, 2.3e-1, -25.1`.

Die Zahl der Nachkommastellen der *Mantisse* ist beschränkt (auf ca. 15). Der *Exponent* ist auch beschränkt (auf 304).

Grund: Eine `float` Zahl muss in 64bit passen.

Die arithmetischen Operationen $+, -, \times, /$ werden in OCAML als `+. -. *. /.` geschrieben.

Konstanten des Typs `float` haben entweder einen Dezimalpunkt oder ein `e` (großes `E` ist auch erlaubt).

`10` ist kein Element des Typs `float`, sondern ein Element des Typs `int`.

Vorsicht mit Rundungsfehlern

```
# let originalpreis = 3e14;; (* 300 Billiarden Euro *)
val originalpreis : float = 3e+14
# let sonderpreis = originalpreis -. 0.05;; (* 5ct Rabatt *)
val sonderpreis : float = 3e+14
# originalpreis -. sonderpreis;;
- : float = 0.0625 (* Ups *)
#
```

Kommentare

Kommentare werden in `(*...*)` eingeschlossen.

Sie haben auf den Programmablauf keinen Einfluss, dienen aber der Verständlichkeit und der Dokumentation.

Es gibt auch das *Rauskommentieren* (*commenting out*) von derzeit nicht benötigten Programmteilen.

Leider werden oft zuwenig Kommentare geschrieben.

Leider wird oft ein Programm verändert, die Kommentare aber nicht.

Ganze Zahlen in OCAML

```
val x : int = 0
# let x = 1729;;
val x : int = 1729
# let z = 13;;
val z : int = 13
# z * x;;
- : int = 22477
#
```

Datentyp `int` = $\{-2^{30}, \ldots, 2^{30} - 1\} \approx \mathbb{Z}$.

Operationen: `+, -, *, /, mod`.

Einen Datentyp `nat` gibt es nicht!

Überlauf

```
let guthaben = 800000000;;
guthaben * 2;;
- : int = -547483648
```

Grund: 1.6Mrd $> 2^{30}$.

Bei *Überlauf* wird einfach ganz unten (bei -2^{30}) wieder angefangen. Vgl. “Asteroids”.

Funktionen in OCAML

```
let f = function (x:float) -> (1./.x : float);;
val f : float -> float = <fun>
```

Typannotate sind (meistens) unnötig.

```
# let f = function x -> 1./.x;;
val f : float -> float = <fun>
# let mittel = function (x,y) -> (x +. y)/. 2.;;
val mittel : float * float -> float = <fun>
# let loesung = function (a,b,c) ->
      (-.b +. sqrt(b*.b -. 4.*.a*.c))/.(2.*.a);;
val loesung : float * float * float -> float = <fun>
#   loesung(1.,-1.,-1.);;
- : float = 1.61803398875
```

Alternative Notationen für Funktionen

Statt

```
# let f = function x -> function y -> function z -> x+y+z;;
```

auch

```
# let f = fun x y z -> x+y+z;;
val f : int -> int -> int -> int = <fun>
```

oder

```
# let f x y z = x+y+z;;
```

Natürlich auch mit nur einem Argument:

```
# let f(x,y,z) = x + y + z;;
```

Rekursion

```
# let rec fak = function n ->
        if n = 0 then 1 else n * fak (n-1);;
val fak : int -> int = <fun>
# fak 10;;
- : int = 3628800
# fak (-1);;
Stack overflow during evaluation (looping recursion?).
```

Alternativnotation

```
# let rec fak n =
        if n = 0 then 1 else n * fak (n-1);;
```

Die Fibonaccizahlen mit floats

```
# let rec fib n =
    if n = 0 || n = 1 then 1. else fib(n-1)+.fib(n-2);;
val fib : int -> float = <fun>
```

Alternative Definition

```
# let rec fib2 n = if n=0 then (1.,1.) else
                   let (u,v)=fib2(n-1) in (v,u+.v);;
val fib2 : int -> float * float = <fun>
```

Es ist `fib2` $n = (\texttt{fib}\, n, \texttt{fib}\, (n+1))$.

Beweis durch Induktion.

```
# fib 40;;
```

Braucht mehrere Minuten

```
# fib2 40;;
```

Braucht nur ein paar Millisekunden. Warum?

Verschränkte Rekursion

```
# let rec gerade x = if x = 0 then true else ungerade (x-1)
  and ungerade x = if x = 0 then false else gerade (x-1);;
val gerade : int -> bool = <fun>
val ungerade : int -> bool = <fun>
# gerade 10;;
- : bool = true
```

Mehrere `let`-Definitionen (insbesondere) rekursive können mithilfe von `and` zu einer einzigen zusammengefasst werden.

Bezeichner

Bezeichner (engl.: *identifier*) dienen der Bezeichnung von Abkürzungen (Definitionen) und Variablen.

Sie müssen in OCAML mit einem Kleinbuchstaben oder dem *underscore* (_) beginnen.

Danach dürfen beliebige Buchstaben, Zahlen, sowie die Symbole _ (Underscore) und `'` (*Hochkomma*, engl. *quote*) benutzt werden.

Legale Bezeichner: `eins`, `x789`, `zahlen_wert`, `gueteMass`, `__DEBUG`, `x'`

Illegale Bezeichner: `Eins`, `1a`, `zahlen-wert`, `GueteMass`.

Ich rate von der Verwendung von Umlauten und ß in Bezeichnern ab.

Schlüsselwörter

Bestimmte Wörter, die in Sprachkonstrukten verwendet werden sind auch nicht Bezeichner erlaubt. Dazu gehören z.B.:

```
and else false fun function if in let
```

Eine vollständige Liste findet sich in der OCAML Dokumentation.

```
# let else = 9;;
Syntax error
# let let = 1;;
Syntax error
# let fun = 8;;
Syntax error
#
 let false = 0;;
This expression has type int but is here used with type boo
```

Polymorphie und Funktionsparameter

```
# let komponiere g f = fun x -> g(f x);;
val komponiere : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun
# let rec iteriere f n = if n=0 then (fun x->x) else
      komponiere f (iteriere f (n-1));;
val iteriere : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Beachte: Typvariablen beginnen mit Hochkomma (`'a`, `'b`,...)

Zeichenketten

Zeichenketten bilden den Datentyp `string`.

```
# let vorname = "Ibn Musa";;
val vorname : string = "Ibn Musa"
# let nachname = "Al Chwarizmi";;
# let name = vorname ^ nachname;;
val name : string = "Ibn MusaAl Chwarizmi"
# let name = vorname ^ " " ^ nachname;;
val name : string = "Ibn Musa Al Chwarizmi"
# String.length name;;
- : int = 21
```

Die Funktion `String.length` bestimmt die Länge einer Zeichenkette,

die Funktion ^ (infixnotiert) verkettet zwei Zeichenketten.

Die Verkettung heißt auch *Konkatenation* (von lat. *catena*=Kette).

Zeichen

Eine Zeichenkette ist aus Zeichen zusammengesetzt.

Die Zeichen (engl. *character*) bilden den Datentyp `char`.

Zeichenkonstanten werden in Hochkommata eingeschlossen:

```
# let z1 = 'a';;
val z1 : char = 'a'
# let z2 = '#';;
val z2 : char = '#'
# let z3 = '%';;
val z3 : char = '%'
#
```

Zeichen sind Buchstaben (`A-Za-z`), Ziffern (`0-9`), druckbare Sonderzeichen. Am besten nur

```
^!"$%&/()=?{[]}\''~@-_.,;:#+*|<>.+
```

Vorsicht mit ßäöü, usw.

Nicht druckbare Zeichen

Außerdem gibt es nicht druckbare Zeichen wie “Zeilenumbruch” (*newline*).

In OCAML notiert man dieses Zeichen `\n`.

Den “Rückschrägen” (*backslash*) notiert man `\\`.

Das Hochkomma notiert man `\'`.

Das Anführungszeichen notiert man `\"`.

Vergleichsoperationen

Die Vergleichsoperationen `<, >, =, <=, >=` haben in OCAML den Typ

```
'a * 'a -> bool
```

Bei `int, float` bezeichnen sie die übliche Ordnung.

Bei `bool` gilt `false < true`.

Bei `char` gilt die ASCII-Ordnung, siehe [Kröger, S.33], also etwa `'/' < '0' < 'Q' < 'q' < 'r' < '{'`.

Bei `string` gilt die *lexikographische Ordnung*.

Bei allen anderen Typen genügt es zu wissen, dass `<` eine *totale Ordnung* ist.

Bei Funktionstypen ist die Verwendung von Vergleichsoperationen ein Fehler.

Lexikografische Ordnung

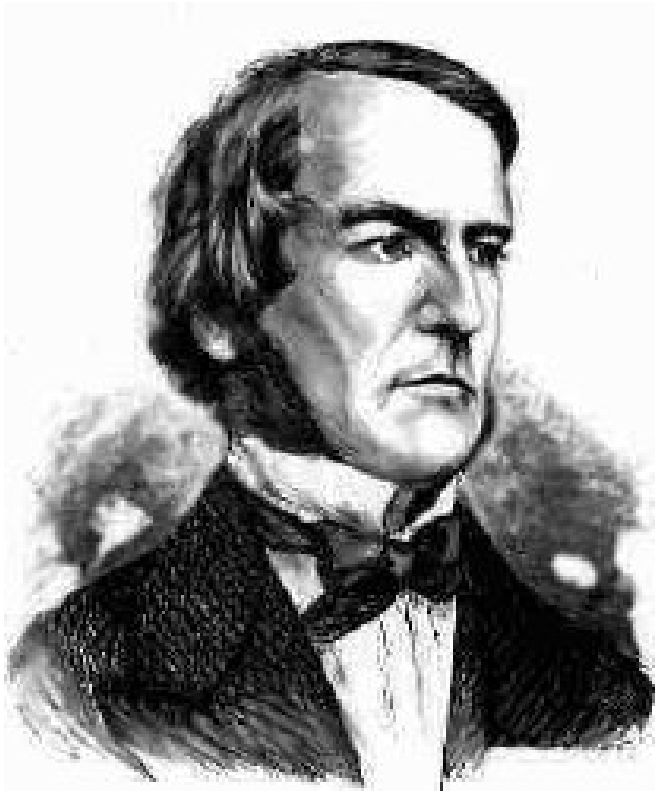
Seien $u = u_1, \ldots, u_m$ und $v = v_1, \ldots, v_n$ Zeichenketten (also $u_i, v_j \in$ `char`.

Es gilt $u < v$ gdw., entweder $m = 0$ und $n > 0$ oder $u_1 < v_1$ oder $u_1 = v_1$ und $u_2, \ldots, u_m < v_2, \ldots, v_n$.

Z.B.

```
"AAAAAAA" < "Anderer Schluesseldienst"
"Martin" < "Martina"
"Lexikografisch" < "Lexikon"
```

Boole'sche Operationen



GEORGE BOOLE 1815–1864

Pseudocode	OCAML
$\wedge$	`&&`
$\vee$	`||`
$\neg$	`not`

Schaltjahre sind durch 4 teilbar aber nicht durch 100 es sei denn durch 400.
Übung: Man schreibe einen Ausdruck, der `true` ist gdw. `jahr` ein Schaltjahr ist.

Zusammenfassung Basistypen, Basisfunktionen

Arithmetische Operationen

`+, -, *`	`int*int->int`	
`/`	`int*int->int`	Ganzzahlige Division
`mod`	`int*int->int`	Rest. Infixnotiert.
`+., -., *.,/.`	`float*float->float`	

Vergleichsoperationen

`=,<,>,<=,>=,<>`	`'a*'a->bool`	Undefiniert für Funktionstypen
Boolesche Operationen		`<>`: ungleich
`not`	`bool->bool`	
`||,&&`	`bool*bool->bool`	

Konkatenation

`^`	`string*string->string`

Für weitere Operationen siehe OCAML Standard Library

Dateieingaben

Sie können eine Folge von OCAML Eingaben auch in eine Datei schreiben.

Dabei dürfen Sie die `;;` weglassen.

Diese Datei können Sie dann mit

`# #use` *Dateiname*`;;`

laden. Der Effekt ist derselbe, als hätten Sie alle Eingaben von Hand getätigt.

Es empfiehlt sich die Dateiendung `.ml`

Kapitel 3.3 Syntaxdefinitionen

Syntax

Syntax: Festlegung des Satzbaus.

Beispiele syntaktisch falscher deutscher Sätze:

Kai liest eine Buch. Buch lesen Kai. Kai pr1&. Kai liest ein Buch, weil ihr ist langweilig.

Beispiele syntaktisch falscher OCAML-Phrasen (Frasen?):

```
let let x = 3 in 2;;
let if = 1;;
2 + * 3;;
2 : 3;;
```

Semantik

Semantik: Festlegung der Bedeutung eines Satzes.

Semantische Fragen im Deutschen: Worauf bezieht sich ein Relativpronomen? Welchen Einfluss haben Fragepartikel wie “eigentlich”, “denn”? Wann verwendet man welche Zeitform?

Semantische Fragen bei OCAML: Was ist der Wert von Ausdrücken wie

```
let x = 1 in let y = x in let x = 2 in y;;
let x = 1./.0. in 2;;
let rec f x = f x in if true then 1 else f 0;;
```

Grundfrage der Semantik von Programmiersprachen: Welche Wirkung hat ein syntaktisch korrektes Programm?

Aus *historischen Gründen* werden Fragen der Typüberprüfung auch der Semantik zugerechnet.

Formale Syntax

- Ein *Alphabet* ist eine endliche Menge, deren Elemente *Symbole* genannt werden.
- Eine *Zeichenkette* (auch *Wort*) über einem Alphabet Σ ist eine endliche Folge von Elementen $\sigma_1, \ldots, \sigma_n$ von Σ, wobei $n \geq 0$. Man schreibt ein Wort als $\sigma_1\sigma_2 \ldots \sigma_n$. Der Fall $n = 0$ bezeichnet das *leere Wort* geschrieben ε.
- Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet. Zu zwei Wörtern $w = \sigma_1 \ldots \sigma_n$ und $w' = \sigma'_1 \ldots \sigma'_m$ bildet man die *Verkettung* (Konkatenation) $ww' = \sigma_1 \ldots \sigma_n\sigma'_1 \ldots \sigma'_m$. Es ist $\varepsilon w = w\varepsilon = w$ und $(ww')w'' = w(w'w'')$.
- Eine *formale Sprache* ist eine Teilmenge von Σ^*.

Beispiele

$\Sigma = \{a, b\}$.

Wörter über Σ: $aaba, baab, bababa, baba, \varepsilon, bbbb$.

Sprachen über Σ: $\emptyset$, $\{a, b\}$, $\{a^n b^n \mid n \in \mathbb{N}\}$.

$\Sigma = \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, \texttt{e}, \texttt{-}, \texttt{+}, \texttt{.}, \texttt{E}\}$.

Wörter über Σ: `-1E98`, `--2e--`, `32.e`

Sprachen über Σ: Syntaktisch korrekte `float` Konstanten,
$\{\texttt{e}^n \mid n$ gerade$\}$.

$\Sigma = \{\texttt{0}, \ldots, \texttt{9}, \texttt{if}, \texttt{then}, \texttt{let}, \ldots\}$

Sprache über Σ: alle syntaktisch korrekten OCAML Phrasen.

Beispiel: OCAML-Bezeichner

- Bezeichner sind Zeichenketten über dem Alphabet $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}, \texttt{0}, \ldots, \texttt{9}, \texttt{'}, \texttt{\_}\}$. Bezeichner müssen mit einem Kleinbuchstaben oder dem Zeichen _ beginnen.
- Ein *Buchstabe* ist ein *Kleinbuchstabe* oder ein *Großbuchstabe*
- Ein *Kleinbuchstabe* ist ein Zeichen $\texttt{a}, \ldots, \texttt{z}$.
- Ein *Großbuchstabe* ist ein Zeichen $\texttt{A}, \ldots, \texttt{Z}$.
- Eine *Ziffer* ist ein Zeichen $\texttt{0}, \ldots, \texttt{9}$.

Formale Definition als BNF-Grammatik

BNF = *Backus-Naur Form*

$$
\begin{array}{lcl}
\langle \textit{Bezeichner} \rangle & ::= & \{\langle \textit{KlBuchst} \rangle \mid \_\}\{\langle \textit{Buchst} \rangle \mid \langle \textit{Ziffer} \rangle \mid \texttt{'} \mid \_\}^* \\
\langle \textit{Buchst} \rangle & ::= & \langle \textit{KlBuchst} \rangle \mid \langle \textit{GrBuchst} \rangle \\
\langle \textit{KlBuchst} \rangle & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{g} \mid \texttt{h} \mid \texttt{i} \mid \texttt{j} \mid \texttt{k} \mid \texttt{l} \mid \texttt{m} \mid \texttt{n} \mid \texttt{o} \\
 & & \mid \texttt{p} \mid \texttt{q} \mid \texttt{r} \mid \texttt{s} \mid \texttt{t} \mid \texttt{u} \mid \texttt{v} \mid \texttt{w} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \\
\langle \textit{GrBuchst} \rangle & ::= & \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{G} \mid \texttt{H} \mid \texttt{I} \mid \texttt{J} \mid \texttt{K} \mid \texttt{L} \mid \texttt{M} \mid \texttt{N} \mid \texttt{O} \\
 & & \mid \texttt{P} \mid \texttt{Q} \mid \texttt{R} \mid \texttt{S} \mid \texttt{T} \mid \texttt{U} \mid \texttt{V} \mid \texttt{W} \mid \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \\
\langle \textit{Ziffer} \rangle & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{array}
$$

`float`-Literale

Literal = Konstante in einer Programmiersprache.

$$
\begin{array}{lcl}
\langle \textit{float-Literal} \rangle & ::= & [\langle \textit{Vorzeichen} \rangle]\langle \textit{Mantisse} \rangle[\langle \textit{Exponent} \rangle] \\
\langle \textit{Vorzeichen} \rangle & ::= & \texttt{-} \mid \texttt{+} \\
\langle \textit{Mantisse} \rangle & ::= & \{\langle \textit{Ziffer} \rangle\}^{+}[\texttt{.}\{\langle \textit{Ziffer} \rangle\}^{*}] \\
\langle \textit{Exponent} \rangle & ::= & \{\texttt{e} \mid \texttt{E}\}[\langle \textit{Vorzeichen} \rangle]\{\langle \textit{Ziffer} \rangle\}^{+}
\end{array}
$$

Zusätzliche *Kontextbedingung*: Entweder ein Dezimalpunkt, oder ein `e` oder ein `E` muss vorhanden sein.

Übung: man verbessere die BNF Darstellung so, dass diese Kontextbedingung wegfallen kann.

Syntax der BNF

Eine BNF-Grammatik ist ein Quadrupel $G = (\Sigma, V, S, P)$.

- Σ ist die Menge der Terminalsymbole, meist in `Courier` gesetzt.
- V ist die Menge der Nichtterminalsymbole, meist in spitze Klammern gesetzt. Im Beispiel:
  $V = \{\langle \textit{float-Literal} \rangle, \langle \textit{Mantisse} \rangle, \langle \textit{Vorzeichen} \rangle, \langle \textit{Exponent} \rangle\}$.
- $S \in V$ ist ein ausgezeichnetes Nichtterminalsymbol, das *Startsymbol*. Im Beispiel: $S = \langle \textit{float-Literal} \rangle$.
- P ist eine endliche Menge von *Produktionen* der Form $X ::= \delta$, wobei δ eine *BNF-Satzform*, s.u., ist.

BNF-Satzformen

- Jedes Symbol in $V \cup \Sigma$ ist eine BNF Satzform.
- Sind $\gamma_1, \ldots, \gamma_n$ BNF-Satzformen, so auch $\gamma_1 \mid \cdots \mid \gamma_n$ (*Auswahl*).
- Sind $\gamma_1, \ldots, \gamma_n$ BNF-Satzformen, so auch $\gamma_1 \ldots \gamma_n$ (*Verkettung*).
- Ist γ eine BNF Satzform, so auch $\{\gamma\}$ (*Klammerung*).
- Ist γ eine BNF Satzform, so auch $\{\gamma\}^*$ (*Iteration*).
- Ist γ eine BNF Satzform, so auch $\{\gamma\}^+$ (*nichtleere Iteration*).
- Ist γ eine BNF Satzform, so auch $[\gamma]$ (*Option*).

Eine Satzform der Gestalt $\gamma_1 \mid \cdots \mid \gamma_n$ muss immer geklammert werden, es sei denn, sie tritt unmittelbar als rechte Seite einer Produktion auf.

Diese Folie wurde gestrichen, die folgende sinngemäß veränddert.

Semantik der BNF

Sei $G = (\Sigma, V, S, P)$ und $X \in V$ ein Nichtterminalsymbol. Ein Wort w ist aus X herleitbar ("ist ein X", "ist in $\mathcal{L}(X)$"), wenn man es aus X durch die folgenden Ersetzungsoperationen erhalten kann:

- Falls $X ::= \gamma_1 \mid \cdots \mid \gamma_n$ eine Produktion ist, so darf man ein Vorkommen von X durch eines der γ_i ersetzen.
- Ein Vorkommen von $\{\gamma_1 \mid \cdots \mid \gamma_n\}$ darf man durch eines der γ_i ersetzen.
- Ein Vorkommen von $\{\gamma\}^*$ darf man durch $\overbrace{\{\gamma\}\{\gamma\}\ldots\{\gamma\}}^{n\text{-mal}}$ mit $n \geq 0$ ersetzen.
- Ein Vorkommen von $\{\gamma\}^+$ darf man durch $\overbrace{\{\gamma\}\{\gamma\}\ldots\{\gamma\}}^{n\text{-mal}}$ mit $n > 0$ ersetzen.
- Ein Vorkommen von $\{\gamma\}$ darf man durch γ ersetzen, wenn γ nicht von der Form $\gamma_1 \mid \cdots \mid \gamma_n$ ist.
- Ein Vorkommen von $[\gamma]$ darf man durch $\{\gamma\}$ ersetzen, oder ersatzlos streichen.

Beispiel

$\langle$*float-Literal*$\rangle \to [\langle$*Vorzeichen*$\rangle]\langle$*Mantisse*$\rangle[\langle$*Exponent*$\rangle] \to$
$\langle$*Mantisse*$\rangle[\langle$*Exponent*$\rangle] \to \{\langle$*Ziffer*$\rangle\}^+[\texttt{.}\{\langle$*Ziffer*$\rangle\}^*][\langle$*Exponent*$\rangle] \to$
$\{\langle$*Ziffer*$\rangle\}^+\texttt{.}\{\langle$*Ziffer*$\rangle\}^*[\langle$*Exponent*$\rangle] \to$
$\{\langle$*Ziffer*$\rangle\}^+\texttt{.}\{\langle$*Ziffer*$\rangle\}\{\langle$*Ziffer*$\rangle\}[\langle$*Exponent*$\rangle] \to$
$\{\langle$*Ziffer*$\rangle\}\texttt{.}\{\langle$*Ziffer*$\rangle\}\{\langle$*Ziffer*$\rangle\}[\langle$*Exponent*$\rangle] \to \texttt{2.71}\langle$*Exponent*$\rangle \to$
$\texttt{2.71}\{\texttt{e} \mid \texttt{E}\}[\langle$*Vorzeichen*$\rangle]\{\langle$*Ziffer*$\rangle\}^+ \to$
$\texttt{2.71E}\langle$*Vorzeichen*$\rangle\{\langle$*Ziffer*$\rangle\}\{\langle$*Ziffer*$\rangle\}\{\langle$*Ziffer*$\rangle\} \to$
$\texttt{2.71E}\{\texttt{-} \mid \texttt{+}\}\texttt{001} \to \texttt{2.71E-001}$

Also $\texttt{2.71E-001} \in \mathcal{L}(\langle$*float-Literal*$\rangle)$.

Kontextbedingungen

Manche syntaktische Bedingungen lassen sich mit BNF nur schwer oder gar nicht formulieren.

Man gibt daher manchmal zusätzliche *Kontextbedingungen* an, denen die syntaktisch korrekten Wörter zusätzlich genügen müssen.

Beispiele:

- Bezeichner dürfen nicht zu den Schlüsselwörtern gehören wie z.B. `let`, `if`, etc.
- float-Literale müssen `.`, `e`, oder `E` enthalten.

Andere Bedingungen, wie korrekte Typisierung oder rechtzeitige Definition von Bezeichnern werden, wie schon gesagt, der Semantik zugerechnet.

Varianten

Häufig wird statt $\{\gamma\}^*$ nur $\{\gamma\}$ geschrieben. Für die Klammerung verwendet man dann runde Klammern.

Die spitzen Klammern zur Kennzeichnung der Nichtterminalsymbole werden oft weggelassen.

Steht kein `Courier` Zeichensatz zur Verfügung, so schließt man die Terminalsymbole in “Anführungszeichen” ein.

Ableitungsbäume

Ableitungen in einer BNF lassen sich grafisch durch *Ableitungsbäume* darstellen.

Beispiele von Ableitungsbäumen finden Sie in [Kröger].

Diese Ableitungsbäume sind für die Festlegung der Semantik von Bedeutung.

Ein *Parser* berechnet zu einem vorgegebenen Wort einen Ableitungsbaum, falls das Wort in $\mathcal{L}(S)$ ist, und erzeugt eine Fehlermeldung, falls nicht.

Diese Aufgabe bezeichnet man als *Syntaxanalyse*.

Syntaxdiagramme

Man kann BNF Syntaxdefinitionen auch grafisch durch *Syntaxdiagramme* darstellen.

Terminalzeichen werden als Kreise dargestellt.

Für jedes Nichtterminalzeichen X (als Kasten dargestellt) ein Diagramm. Eine Zeichenreihe aus $\mathcal{L}(X)$ erhält man, indem man im Diagramm für X einen beliebigen Weg vom Anfang zum Ausgang wählt, dabei jedes auftretende Terminalzeichen aufsammelt und jedes auftretende Nichtterminalzeichen Y durch ein entsprechend gefundenes Wort aus $\mathcal{L}(Y)$ ersetzt.

Siehe [Kröger] für Beispiele von Syntaxdiagrammen.

OCAML-Sprachdefinition (Fragment)

⟨*toplevel-command*⟩ ::= `let` ⟨*let-binding*⟩{ `and` ⟨*let-binding*⟩}*`;;`
| `let rec` ⟨*let-binding*⟩{ `and` ⟨*let-binding*⟩}*`;;`
| ⟨*expression*⟩`;;`

⟨*expression*⟩ ::= ⟨*constant*⟩
| `(`⟨*expression*⟩`)`
| ⟨*expression*⟩⟨*infix-op*⟩⟨*expression*⟩
| `if` ⟨*expression*⟩ `then` ⟨*expression*⟩ `else` ⟨*expression*⟩
| `let` ⟨*let-binding*⟩{ `and` ⟨*let-binding*⟩}* `in` ⟨*expression*⟩
| `let rec`⟨*let-binding*⟩{ `and` ⟨*let-binding*⟩}* `in` ⟨*expression*⟩
| `function` ⟨*ident*⟩ `->` ⟨*expression*⟩
| {⟨*expression*⟩}$^+$
| ⟨*expression*⟩`,`⟨*expression*⟩

⟨*infix-op*⟩ ::= `+` | `-` | `*` | `/` | `+.` | `-.` | `*.` | `/.` | `&&` | `||` | `mod`
| `<` | `>` | `<>` | `<=` | `>=` | `<>`

⟨*let-binding*⟩ ::= {⟨*ident*⟩}$^+$ `=` ⟨*expression*⟩

Parsergeneratoren

Erinnerung: *Parser* : $\Sigma^* \to$ *Ableitungsbäume* $\cup$ *Fehlermeldungen*.

Ein *Parsergenerator* erzeugt aus einer BNF-Grammatik automatisch einen Parser.

Der bekannteste Parsergenerator heißt “yacc” (*yet another compiler-compiler*). Er erzeugt aus einer BNF-Grammatik einen in der Programmiersprache C geschriebenen Parser.

Für OCAML gibt es “ocamlyacc”. Es wird ein OCAML-Programm erzeugt.

Geschichte

Grammatikformalismen, die syntaktisch korrekte Wörter durch einen Erzeugungsprozess definieren (wie die BNF) heißen *generative Grammatiken*.

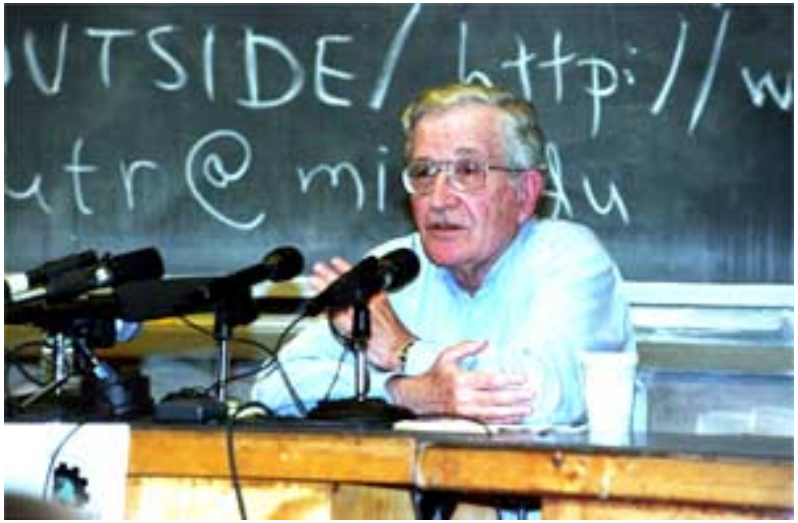
Sie gehen auf den Sprachforscher NOAM CHOMSKY (1928–) zurück.

Eine *kontextfreie Chomsky-Grammatik* ist eine BNF-Grammatik ohne die Konstrukte $\{\ \}^*, [\ ], |, \{\ \}^+$. Man kann jede BNF-Grammatik durch eine kontextfreie Chomsky-Grammatik simulieren.

Chomsky betrachtet u.a. auch kontextsensitive Grammatiken.

Die Backus-Naur-Form wurde von den Informatikern JOHN BACKUS und PETER NAUR entwickelt, die die Bedeutung von Chomskys generativen Grammatiken für Programmiersprachensyntax erkannten.

Noam Chomsky



NOAM CHOMSKY

$$
\begin{array}{lcl}
\langle \mathit{Bezeichner} \rangle & ::= & \langle \mathit{KlBuchst} \rangle \langle \mathit{Folge} \rangle \\
\langle \mathit{Bezeichner} \rangle & ::= & \_ \langle \mathit{Folge} \rangle \\
\langle \mathit{Folge} \rangle & ::= & \langle \mathit{Zeichen} \rangle \langle \mathit{Folge} \rangle \\
\langle \mathit{Folge} \rangle & ::= & \varepsilon \\
\langle \mathit{Zeichen} \rangle & ::= & ' \\
\langle \mathit{Zeichen} \rangle & ::= & \_ \\
\langle \mathit{Zeichen} \rangle & ::= & \langle \mathit{Buchst} \rangle \\
\langle \mathit{Zeichen} \rangle & ::= & \langle \mathit{Ziffer} \rangle \\
\langle \mathit{Buchst} \rangle & ::= & \langle \mathit{KlBuchst} \rangle \\
\langle \mathit{Buchst} \rangle & ::= & \langle \mathit{GrBuchst} \rangle
\end{array}
$$

Kapitel 3.4 Termauswertung

Präzedenz- und Assoziationsregeln

Operator	Präzedenz	Assoziativität	Erklärung
Funktionsapplikation		links	
`-`	10	—	Präfix-Minus
`**`	9	rechts	Potenzierung bei `float`
`*. /. * / mod`	8	links	
`+. -. + -`	7	links	
`^`	6	rechts	Verkettung
`< > <= >= <> =`	5	links	
`not`	4	—	Präfix-Verneinung
`&&`	3	links	
`||`	2	links	
`,`	1	—	Paarbildung, *in praxi* meis geklammert
`if let fun function`	0	—	Quasi-Präfixoperatoren

Was bedeutet das?

Dass `* /` oberhalb von `+ -` steht entspricht der Punkt-vor-Strich Regel.

Vergleichsoperatoren unterhalb von arithmetischen Operatoren bedeutet, dass z.B. `i - j < 7` bedeutet `(i - j) < 7` und nicht etwa `i - (j < 7)` was ein Typfehler wäre.

Das Funktionsapplikation über allem steht, bedeutet, dass man z.B.: $2 \times f(x)$ in OCAML als `2 * f x` schreiben kann.

Dass `if let fun` ganz unten stehen, bedeutet, dass z.B.

```
if i < j then 0 else x - y + z
```

als

```
if i < j then 0 else (x - y + z)
```

und nicht etwa

```
(if i < j then 0 else x) - y + z
```

Werte und Umgebungen

Wir wollen formal beschreiben, was der Wert eines OCAML-Ausdrucks ist.

Um den Wert eines Ausdrucks anzugeben, muss man die Werte der in ihm enthaltenen freien Variablen kennen.

Eine Zuweisung von Werten an freie Variablen heißt *Umgebung*.

Die *Semantik* eines OCAML-Ausdrucks ist also eine Abbildung von Umgebungen auf Werte.

Umgebungen

Eine *Umgebung* ist eine Menge von *Bindungen* $<a, w>$ wobei a ein Bezeichner ist und w ein Wert.

Während einer OCAML Sitzung wird eine Umgebung (die *aktuelle Umgebung*) sukzessive aufgebaut.

Zu Beginn ist die aktuelle Umgebung leer.

Bei der Eingabe `let` x`=` t`;;` wird der Term t in der aktuellen Umgebung ausgewertet.

Ist die Auswertung undefiniert, so entsteht eine Fehlermeldung, bzw. falls Nichttermination der Grund ist, wird nichts ausgegeben.

Anderenfalls wird die Bindung $<x, w>$ der aktuellen Umgebung hinzugefügt, wobei w der berechnete Wert von t ist.

Eine frühere Bindung der Form $<x, w'>$ wird dabei entfernt.

Die aktuelle Umgebung ist also zu jeder Zeit eine *endliche partielle Funktion* von Bezeichnern nach Werten.

Beispiel

Welche Umgebung liegt nach folgenden Eingaben vor?

```
let x = 2;;
let x = x + 1;;
let y = x + 2;;
let f = function z -> z - x;;
let a = f + 1;;
let a = f (f(x)+1);;
```

Auswertung von Termen

Sei U eine Umgebung und t ein Term.

Der Wert $W^U(t)$ des Terms t in der Umgebung U ist wie folgt rekursiv definiert:

- Ist c eine Konstante, so ist $W^U(c) = c$,
- Ist x ein Bezeichner, so ist $W^U(x) = w$, falls $<x, w> \in U$.
  Anderenfalls ist $W^U(x)$ undefiniert.
- Ist *op* ein Infixoperator aber nicht `||`, `&&`, welcher eine Basisfunktion $\oplus$ bezeichnet.

  Sind $W^U(t_1)$ und $W^U(t_2)$ beide definiert, so ist
  $W^U(t_1\ op\ t_2) = W^U(t_1) \oplus W^U(t_2)$.

  Ist auch nur einer der beiden undefiniert, so ist $W^U(t_1\ op\ t_2)$ undefiniert.

  Einstellige Basisfunktionen wie `not` und `-` sind analog.

Auswertung von Funktionstermen

Notation:
$U + \{<x_1, w_1>, \ldots, <x_n, w_n>\} = U' \cup \{<x_1, w_1>, \ldots, <x_n, w_n>\}$,
wobei U' aus U durch Entfernen aller eventuell vorhandenen Bindungen von $x_1, \ldots, x_n$ entsteht.

- Sei t eine Funktionsanwendung der Form $t_1\ t_2$. Es sei $W^U(t_1)$ die Funktion f und $W^U(t_2) = w$. Ist auch nur eines der beiden undefiniert, so ist $W^U(t)$ auch undefiniert. Ansonsten ist $W^U(t) = f(w)$. Dies kann trotzdem noch undefiniert sein, falls $w \notin D(f)$.
- Sei $t =$ `function(`$x_1, \ldots, x_n$`) ->` t'. Im Falle $n = 1$ auch ohne Klammern. Es ist $W^U(t)$ diejenige Funktion, die $(w_1, \ldots, w_n)$ auf $W^{U+\{<x_1,w_1>,\ldots,<x_n,w_n>\}}(t')$ abbildet. Beachte: $W^U(t)$ ist immer definiert, könnte aber die nirgends definierte Funktion sein.
- Die alternativen Notationen für Funktionsdefinitionen (`fun`, `let`) haben analoge Bedeutung.

Auswertung von `if` und `let`

- Sei t ein bedingter Term der Gestalt `if` t_1 `then` t_2 `else` t_3. Ist $W^U(t_1) = \mathit{true}$, so ist $W^U(t) = W^U(t_2)$. Beachte: $W^U(t_3)$ kann dann undefiniert sein.

  Ist $W^U(t_1) = \mathit{false}$, so ist $W^U(t) = W^U(t_3)$. Beachte: $W^U(t_2)$ kann dann undefiniert sein.

  In allen anderen Fällen ist $W^U(t)$ undefiniert.

- Sei t von der Form `let` $(x_1, \ldots, x_n)$ `=` t_1 `in` t_2 im Falle $n = 1$ auch ohne Klammern.

  Sei $W^U(t_1)$ definiert und von der Form $W^U(t_1) = (w_1, \ldots, w_n)$, also ein n-Tupel von Werten. Dann ist $W^U(t) = W^{U+\{<x_1,w_1>,\ldots,<x_n,w_n>\}}(t_2)$. Ansonsten ist $W^U(t)$ undefiniert. Beachte: $W^U(t_1)$ muss auf jeden Fall definiert sein, selbst wenn eines der oder gar alle x_i nicht in t_2 vorkommen.

Tupel und Boole'sche Operatoren

- Sei $t = (t_1, \ldots, t_n)$. Seien weiter $W^U(t_1) = w_1, \ldots, W^U(t_n) = w_n$ allesamt definiert. Dann ist $W^U(t) = (w_1, \ldots, w_n)$.
- $W^U(t_1 \texttt{ || } t_2) = W^U(\texttt{if } t_1 \texttt{ then true else } t_2)$
- $W^U(t_1 \texttt{ \&\& } t_2) = W^U(\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else false})$.

Zur Beachtung: Die Semantikdefinition im Skript [Kröger] ist teilweise nicht ganz richtig; maßgeblich sind daher die Folien[a].

Zur Beachtung: $t_1 \texttt{ || } t_2$ kann definiert sein, auch wenn t_2 undefiniert ist. Auf jeden Fall muss aber t_1 definiert sein.

[a] Kann sein, dass die Folien auch nicht ganz richtig sind, maßgeblich ist daher der gesunde Menschenverstand :-)

Beispiel

$$
\begin{aligned}
& W^{\{<\mathtt{x},1>\}}(\mathtt{let\ x\ =\ 2\ in\ if\ x{=}2\ then\ 0\ else\ 1}) \\
= \quad & W^{\{<\mathtt{x},2>\}}(\mathtt{if\ x{=}2\ then\ 0\ else\ 1}) \\
= \quad & W^{\{<\mathtt{x},2>\}}(\mathtt{0}) \\
= \quad & 0
\end{aligned}
$$

Semantik von `let rec`

Eine rekursive `let`-Bindung

$$\texttt{let rec } f\ x \texttt{ = } t$$

bindet f an die durch

$$F(w) = W^{U+\{<x,w>,<f,F>\}}(t)$$

rekursiv definierte Funktion.

Das gilt analog für Phrasen mit `let rec ...in` und für Funktionen mit mehreren Argumenten.

Beispiel

```
let rec fakt n = if n = 0 then 1 else n * fakt(n-1)
```

Sei $F = W^{\emptyset}(\texttt{fakt})$.

Es ist

$$
\begin{aligned}
& F(3) \\
= \; & W^{\{<\texttt{fakt},F>,<\texttt{n},3>\}}(\texttt{if n = 0 then 1 else n * fakt(n-1)}) \\
= \; & W^{\{<\texttt{fakt},F>,<\texttt{n},3>\}}(\texttt{n * fakt(n-1)}) \\
= \; & W^{\{<\texttt{fakt},F>,<\texttt{n},3>\}}(\texttt{n}) \cdot W^{\{<\texttt{fakt},F>,<\texttt{n},3>\}}(\texttt{fakt(n-1)}) \\
= \; & 3 \cdot F(W^{\{<\texttt{fakt},F>,<\texttt{n},3>\}}(\texttt{n} - 1)) \\
= \; & 3 \cdot F(2) \\
= \; & W^{\{<\texttt{fakt},F>,<\texttt{n},2>\}}(\texttt{if n = 0 then 1 else n * fakt(n-1)}) \\
= \; & \cdots = 6
\end{aligned}
$$

Abstrakte Syntax

Die Definition von W bezieht sich auf *abstrakte Syntax*: Wir setzen voraus, dass der äußerste Operator eines verschachtelten Terms bekannt ist. Strenggenommen ist somit W auf *Herleitungsbäumen* (Ausgabe des Parsers) definiert.

Wie sollte man sonst z.B.

$$W^U(\texttt{2 * 3 - 1})$$

verstehen? Als $W^U(\texttt{2 * 3}) - W^U(\texttt{1})$ oder $W^U(\texttt{2}) \cdot W^U(\texttt{3 - 1})$?

Im Interpreter oder Compiler werden tatsächlich die Herleitungsbäume ausgewertet.

Strikte Auswertung

Die Tatsache, dass alle Argumente eines Funktionsaufrufes definiert sein müssen, damit der Funktionsaufruf terminiert, bezeichnet man als *strikte Auswertung*.

In anderen Programmiersprachen gibt es die *verzögerte Auswertung*, wo nicht benutzte Terme auch nicht ausgewertet werden. Z.B.: in Haskell

```
f x y = y + 1
g x = g x + 1
f (g 0) 3
----> 4
```

In Haskell wird zudem ein einmal ausgewerter Term als Wert abgespeichert und nicht nochmal ausgewertet. Beispiel:

```
u + u where u = f (g 0) 4
```

Partielle Korrektheit

Verhält sich ein Algorithmus richtig für alle Eingaben, für die er terminiert, so spricht man von *partieller Korrektheit*.

Terminiert er außerdem für alle interessierenden Eingaben, so liegt *totale Korrektheit* vor.

Häufig kann man partielle Korrektheit unabhängig von der totalen Korrektheit und mit anderen Methoden zeigen.

Partielle Korrektheit rekursiver Funktionen

Seien A, B Mengen und $R \subseteq A \times B$ eine Relation. Sei A' eine Teilmenge von A. Wir möchten zu jedem $x \in A'$ ein $y \in B$ berechnen, sodass xRy gilt.

Satz von der partiellen Korrektheit: Sei

$$f = \mathbf{function}(x)\Phi(f, x)$$

In Φ befinde sich f nicht im Geltungsbereich einer weiteren Funktionsabstraktion außer $\mathbf{function}(x)$ (also z.B. nicht $\Phi(f, x) = \mathbf{function}(y)f(x)(y)$).

Um zu zeigen, dass für alle $x \in A' \subseteq A$ entweder $f(x)$ undefiniert ist oder $xRf(x)$ gilt, genügt es, folgendes nachzuweisen:

- ist $x \in A'$, so werden in $\Phi(f, x)$ nur Aufrufe $f(x')$ mit $x' \in A'$ getätigt.
- Für alle $x \in A'$ gilt $xR\Phi(f, x)$ unter der Annahme, dass $x'Rf(x')$ für alle in $\Phi(f, x)$ getätigten Aufrufe $f(x')$ von f.

Beispiel

$f = \textbf{function}(x, y)$

if $y = 0$ **then** 0 **else**

if y gerade **then** $2 \cdot f(x, y/2)$ **else** $\quad 2f(x, \lceil y/2 \rceil) + x$

Man zeige: für alle $x, y \in \mathbb{N}$ gilt: falls $f(x, y)$ definiert ist, dann ist $f(x, y) = xy$.

$f = \textbf{function}(x)$

if $x = 0 \vee x = 1$ **then** 0 **else**

if x gerade **then** $f(x/2)$ **else** $f(3x + 1)$

Man zeige: für alle $x \in D(f)$ ist $f(x) = 0$.

McCarthy's Funktion

$$f = \textbf{function}(x{:}\textbf{int})\textbf{int}$$
$$\quad \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } f(f(x+11))$$

$$f(91) = f(f(102)) = f(92) = f(f(103)) = f(93) = \cdots = f(101) = 91$$

$$f(70) = f(f(81)) = f^3(92) = f^4(103) = f^4(93) = f^4(104) = f^3(94) =$$
$$f^4(105) = f^3(95) = f^4(106) = f^3(96) = f^4(107) = f^3(97) =$$
$$f^4(108) = f^3(98) = f^4(109) = f^3(99) = f^4(110) = f^3(100) =$$
$$f^4(111) = f^3(91) = \cdots = 91$$

Man beweise: falls $x \in D(f)$, so gilt
$f(x) = \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } 91.$

Denotationelle Semantik

Die W-Funktion weist OCAML-Ausdrücken abstrakte mathematische Werte zu.

Etwa Funktionsausdrücken mathematische Funktionen, also (unendliche) Mengen von Paaren.

Man bezeichnet diese Art der Semantikgebung als *denotationelle Semantik*.

Vorteile der denotationellen Semantik:

Implementierungsdetails werden versteckt, mathematische Beweismethoden (Gleichungsschließen, Abstiegsfunktion, Induktion, Satz von der partiellen Korrektheit) werden verfügbar gemacht.

Operationelle Semantik

Manchmal ist die denotationelle Semantik zu abstrakt:

- Effizienzbetrachtungen
- Seiteneffekte wie Ein-/Ausgabe
- Verständnisschwierigkeiten bei mathematisch nicht einschlägig gebildeten Personen.

Die *operationelle Semantik* beschreibt die Semantik eines Ausdrucks durch Rechenregeln, die sukzessive angewandt werden. Wert eines Ausdrucks ist dann das Endergebnis der Rechnung.

Problem: Was soll das Ergebnis eines Funktionsausdrucks sein?

Antwort: Der Funktionsausdruck selber, wobei allerdings die Werte freier Variablen gemerkt werden müssen.

Beispiel

```
let x = 1;;
let f = function y -> x + y;;
let x = 2;;
```

Hier sollte `f` den Wert "`fun y -> y + x`, wobei `x`=1" haben.

Solch ein Paar aus einem Funktionsausdruck und einer Umgebung (die freie Variablen des Funktionsausdrucks bindet) bezeichnet man als *Closure*.

Operationelle Semantik formal

Definition *Werte* und *Umgebungen* (im Sinne der operationellen Semantik) werden wie folgt definiert:

- Eine OCAML-Konstante ist ein Wert
- Eine *Umgebung* ist ein Menge von Bindungen $<x, w>$ von Bezeichnern an Werte.
- Ist e ein OCAML-Ausdruck, x ein Bezeichner und U eine Umgebung, so ist $(\texttt{fun}\ \ x\texttt{->}e, U)$ ein Wert.
- Ist e ein OCAML-Ausdruck, f und x Bezeichner und U ein Umgebung, so ist $(f = \texttt{fun}\ \ x\texttt{->}e, U)$ ein Wert.

Beispiel

Habe x den Wert 7.

Nach den Deklarationen

```
let x = 7;;
let rec f = fun y -> if y=0 then x else f (y-1);;
let g = f;;
```

hat `g` den Wert
(`f` $=$ `fun y->if y=0 then x else f (y-1)`, $\{<$`x`$, 7>\}$).

Auswerterelation

Wir schreiben $U, e \to w$ um zu sagen, dass in der Umgebung U der Ausdruck e als Endergebnis den Wert w hat.

Diese *Auswerterelation* wird formal durch die folgenden Regeln definiert.

Auswerteregeln

- ist c eine Konstante, so gilt $U, c \to c$
- ist x ein Bezeichner und $<x, w> \in U$, so gilt $U, x \to w$.
- für Funktionsausdrücke gilt U, `fun` x`->`$e \to$ (`fun` x`->`e, U'), wobei U' die Einschränkung von U auf die freien Variablen von e ist.

Auswerteregeln

- ist *op* ein Infixoperator aber nicht `||`, `&&`, welcher eine Basisfunktion $\oplus$ bezeichnet so gilt folgendes: wenn $U, e_1 \to w_1$ und $U, e_2 \to w_2$, dann $U, e_1\ op\ e_2 \to w_1 \oplus w_2$.

  Natürlich müssen w_1, w_2 im Definitionsbereich von $\oplus$ liegen.

Einstellige Basisfunktionen werden analog behandelt.

Fallunterscheidung

- Gilt $U, e_1 \to$ *true* und $U, e_2 \to w_2$, so auch $U, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to w_2$.
- Gilt $U, e_1 \to$ *false* und $U, e_3 \to w_3$, so auch $U, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to w_3$.
- Die Infixoperatoren `&&`, `||` werden analog behandelt.

Applikation

- Gilt $U, e_1 \to w_1$ mit $w_1 = (\texttt{fun}\ \ x\texttt{->}e, U')$ und $U, e_2 \to w_2$, so ist (in einer Nebenrechnung) ein Wert w mit $U' + \{<x, w_2>\}, e \to w$ zu bestimmen. Es ist dann $U, e_1\ e_2 \to w$.
- Gilt $U, e_1 \to w_1$ mit $w_1 = (f\texttt{=fun}\ \ x\texttt{->}e, U')$ und $U, e_2 \to w_2$, so ist zunächst (in einer Nebenrechnung) ein Wert w mit $U' + \{<x, w_2>, <f, w_1>\}, e \to w$ zu bestimmen. Es ist dann $U, e_1\ e_2 \to w$.

Bindung

- Ist $U, e_1 \to w_1$, so ist zunächst (in einer Nebenrechnung) ein Wert w_2 mit $U + \{<x, w_1>\}, e_2 \to w_2$ zu bestimmen. Es ist dann U, `let` x`=`e_1 `in` $e_2 \to w_2$.
- Es gilt U, `let rec` f`=fun` x`->`e_1 `in` $e_2 \to w$, falls $U + \{<f, (f$`=fun` x`->`$e_1, U')>\}, e_2 \to w$ wobei U' die Einschränkung von U auf die freien Variablen von e_1 ist.

Beispiel

Wir wollen

```
let x=1+0 in let rec f = fun y->if y=0 then x else f(y-1) in
        let x = 2 in f (x+x)
```

in der leeren Umgebung auswerten.

Es ist $\emptyset, \texttt{1} + \texttt{0} \to 1$ also muss

```
let rec f = fun y->if y=0 then x else f(y-1) in
        let x = 2 in f (x+x)
```

in der Umgebung $U_1 = \{<\texttt{x}, 1>\}$ auszuwerten. Das aber bedeutet, in der Umgebung

$U_2 = \{<\texttt{x}, 1>, <\texttt{f}, (\texttt{f=fun y->if y=0 then x else f(y-1)}, U_1)>\}$

den Ausdruck `let x=2 in f (x+x)` auszuwerten. Dies schließlich

Beispiel

bedeutet, in der Umgebung

$U_3 = \{<\texttt{x}, 2>, <\texttt{f}, (\texttt{f=fun y->if y=0 then x else f(y-1)}, U_1)>\}$

den Ausdruck `f (x+x)` auszuwerten.

Es gilt $U_3, \texttt{x+x} \to 4$ und
$U_3, \texttt{f} \to (\texttt{f=fun y->if y=0 then x else f(y-1)}, U_1)$, also
müssen wir in der Umgebung

$U_4 = \{<x, 1>, <\texttt{f}, (\texttt{f=fun y->if y=0 then x else f(y-1)}, U_1)>, <y, 4>$

den Ausdruck

```
if y=0 then x else f(y-1)
```

auswerten. Der Ausdruck `y=0` hat den Wert *false*, es gilt also in U_4 den Ausdruck `f 2` auszuwerten...

Zusammenfassung Operationelle Semantik

- Die operationelle Semantik ordnet Umgebungen und Ausdrücken *Werte* zu. Diese Werte sind syntaktisch definiert.
- Rekursion wird durch wiederholte Auswertung, statt durch Rekursion auf der Meta-ebene definiert.
- Die operationelle Semantik liegt näher an der tatsächlichen Implementierung als die denotationelle Semantik.
- Denotationelle Semantik ist günstiger für mathematische Beweismethoden (Gleichungsschließen, Induktion, Satz von der partiellen Korrektheit)

Zusammenhang operationelle u. denotationelle Semantik

Satz: Genau dann ist $W^{\emptyset}(e)$ definiert, wenn es einen Wert w gibt mit $\emptyset, e \to w$.

(Falls e vom Typ `int` ist, folgt daraus, dass beide Semantiken denselben Wert liefern. Wieso?)

Satz: Gilt $W^U(e_1) = W^U(e_2)$ für alle U, so sind e_1 und e_2 bezüglich der operationellen Semantik nicht voneinander zu unterscheiden, d.h., erhält man e_4 aus e_3 durch Ersetzen eines oder mehrerer Vorkommen von e_1 durch e_2, dann gilt $\emptyset, e_3 \to w$ für ein w genau dann wenn $\emptyset, e_4 \to w'$ für ein w'. Ist der Typ von e_3 und e_4 gleich `int`, so gilt $w = w'$.

Bemerkung: die Umkehrung des Satzes gilt nicht: es gibt ununterscheidbare Ausdrücke, die doch nicht die gleiche denotationelle Semantik haben.

Kapitel 3.5 Typüberprüfung

Wozu Typen

ROBIN MILNER: "*Well-typed programs do not go wrong*".

Dynamischer Typfehler: Die Auswertung scheitert wegen falscher Operanden. Beispiele: $\mathit{true} \oplus 8$ oder $5\ 8$ (Applikation).

Die *statische Typüberprüfung* weist solche Programme bereits vor deren Ausführung zurück.

Allerdings werden auch manche "gute Programme" zurückgewiesen:

```
if true then 1 else 5 8
```

"Irgendwie" stimmt aber doch etwas nicht mit diesem Programm!

Statische Typüberprüfung definiert eine intuitiv begreifbare Teilmenge derjenigen Programme, die keine dynamischen Typfehler exhibieren.

ML Folklore: "*If it typechecks, it works*".

Typüberprüfung

Eine *Typaussage* (auch *Typisierungsurteil*, engl.: *typing judgment*) ist eine Aussage der Form

$$\Gamma \triangleright e : \mathit{typ}$$

wobei e ein OCAML-Ausdruck ist, *typ* ein OCAML-Typ ist und Γ eine Menge von Bindungen der Form x_i:typ_i ist, welche den Bezeichnern in e Typen zuweist. Dabei darf kein Bezeichner in Γ zwei verschiedene Typen zugewiesen bekommen.

Solch eine Menge von Bindungen heißt *Typzuweisung* (engl.: *type assignment*, auch *typing context*).

Bedeutung: "Unter der Annahme, dass die Bezeichner die in Γ angegebenen Typen haben, hat e den Typ *typ*."

Beispiel:

$$\texttt{a:int} \triangleright \texttt{fun x -> a + 2 * x} : \texttt{int->int}$$

Typisierungsregeln

Typisierungsaussagen werden formal hergeleitet aus *Typaxiomen* mithilfe von *Typisierungsregeln* (engl.: *typing rules*).

- Typaxiome sind elementare Typaussagen wie etwa $\emptyset \triangleright \texttt{7} : \texttt{int}$ oder $\emptyset \triangleright \texttt{7e1} : \texttt{float}$.
- Typisierungsregeln haben die Form $P_1, \ldots, P_m \vdash K$, wobei die P_i und K Typaussagen sind. Die P_i heißen *Prämissen*, K heißt *Konklusion* der Regel.

  Bedeutung: Gelten die P_i, so soll auch K gelten.
- Eine *Herleitung* einer Typaussage A ist eine Folge $A_1, \ldots, A_n$ von Typaussagen derart, dass $A_n = A$ und jedes A_i entweder ein Typaxiom ist, oder Konklusion einer Typisierungsregel, deren Prämissen unter den $A_1, \ldots, A_{i-1}$ sind.

Typisierungsregel für Paare

$$\Gamma_1 \rhd e_1 : typ_1$$
$$\Gamma_2 \rhd e_2 : typ_2$$
$$\vdash \Gamma_1 \cup \Gamma_2 \rhd (e_1, e_2) : typ_1 * typ_2$$

wobei Γ_1 und Γ_2 *kompatibel* sein müssen.

Definition: Γ_1 und Γ_2 sind kompatibel, wenn $\Gamma_1 \cup \Gamma_2$ eine Typzuweisung ist, d.h. aus $x{:}typ \in \Gamma_1$ und $x{:}typ' \in \Gamma_2$ folgt $typ_1 = typ_2$.

Bemerkung: Die obige Typisierungsregel ist eigentlich ein *Regelschema*, das für unendlich viele einzelne Typisierungsregeln steht: eine für jedes konkret gegebene $\Gamma_1, \Gamma_2, e_1, e_2, typ_1, typ_2$, sodass die Kompatibilitätsbedingung erfüllt ist.

Typisierungsregel für Arithmetik

$$\Gamma_1 \rhd e_1 : \texttt{int}$$
$$\Gamma_2 \rhd e_2 : \texttt{int}$$
$$\vdash \Gamma_1 \cup \Gamma_2 \rhd e_1 \; op \; e_2 : \texttt{int}$$

wobei $op \in \{\texttt{+}, \texttt{*}, \texttt{/}, \texttt{div}, \texttt{mod}\}$ und Γ_1, Γ_2 kompatibel.

Analoge Regeln gibt es für `float` und `bool`.

In Zukunft erwähnen wir die Kompatibilitätsbedingung nicht mehr explizit.

Typisierungsregel für Fallunterscheidung

$$\Gamma_1 \triangleright e_1 : \texttt{bool}$$
$$\Gamma_2 \triangleright e_2 : \mathit{typ}$$
$$\Gamma_3 \triangleright e_3 : \mathit{typ}$$
$$\vdash$$
$$\Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \triangleright \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \mathit{typ}$$

Beachte: beide Zweige der Fallunterscheidung müssen denselben Typ haben.

Typaxiome für Bezeichner

$$\{x{:}typ\} \triangleright x : typ$$

Hierbei ist x ein Bezeichner und *typ* ein Typ.

Typisierungsregel für Funktionen

$$\Gamma \cup \{x{:}\mathit{typ}\} \triangleright e : \mathit{typ}'$$

$$\vdash$$

$$\Gamma \triangleright \texttt{function}\ x\texttt{->}e : \mathit{typ}\texttt{->}\mathit{typ}'$$

Beispiel

(1) $\emptyset \triangleright 2 : \texttt{int}$

(2) $\{\texttt{a} : \texttt{int}\} \triangleright \texttt{a} : \texttt{int}$

(3) $\{\texttt{x} : \texttt{int}\} \triangleright \texttt{x} : \texttt{int}$

(4) $\{\texttt{x} : \texttt{int}\} \triangleright \texttt{2*x} : \texttt{int}$

(5) $\{\texttt{a} : \texttt{int}, \texttt{x} : \texttt{int}\} \triangleright \texttt{a+2*x} : \texttt{int}$

(6) $\{\texttt{a} : \texttt{int}\} \triangleright \texttt{function x->a+2*x} : \texttt{int->int}$

Typisierungsregel für Applikation

$$\Gamma_1 \rhd e_1 : \mathit{typ}\texttt{->}\mathit{typ}'$$

$$\Gamma_2 \rhd e_2 : \mathit{typ}$$

$$\vdash$$

$$\Gamma_1 \cup \Gamma_2 \rhd e_1\ e_2 : \mathit{typ}'$$

Typisierungsregel für `let`

$$\frac{\Gamma_1 \rhd e_1 : \mathit{typ} \qquad \Gamma_2 \cup \{x{:}\mathit{typ}\} \rhd e_2 : \mathit{typ}'}{\Gamma_1 \cup \Gamma_2 \rhd \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : \mathit{typ}'}$$

Beachte: Es ist nicht ausgeschlossen, dass x in Γ_1 gebunden ist.

Typisierungsregel für `let rec`

$$\frac{\Gamma_1 \cup \{f{:}typ\} \triangleright e_1 : typ \qquad \Gamma_2 \cup \{f{:}typ\} \triangleright e_2 : typ'}{\Gamma_1 \cup \Gamma_2 \triangleright \texttt{let rec } f\texttt{=}e_1 \texttt{ in } e_2 : typ'}$$

Hier muss *typ* ein Funktionstyp sein und e_1 ein Funktionsausdruck sein.

Beachte: allgemeinere Formen, wie `let rec f x = ...` sind nicht erfasst.

Die vollständigen Typisierungsregeln für OCAML umfassen mehrere engbedruckte Seiten!

Polymorphie

Die meisten Ausdrücke gestatten mehrere Typisierungen. Etwa:

$\{$`x:int`, `f:int->int`$\}$ $\triangleright$ `f x` : `int`

$\{$`x:int*int`, `f:int*int->int`$\}$ $\triangleright$ `f x` : `int`

$\{$`x:'a`, `f:'a->'a`$\}$ $\triangleright$ `f x` : `'a`

$\{$`x:'a`, `f:'a->'b`$\}$ $\triangleright$ `f x` : `'b`

Die letztgenannte Typisierung ist die allgemeinstmögliche: jede andere Typisierung von `f x` lässt sich aus dieser durch Einsetzen von Typen für die Typvariablen `'a, 'b` erhalten.

Man bezeichnet diese Typisierung als die *prinzipale Typisierung* des Ausdrucks.

Das Einsetzen von Typen für Typvariablen nennt man *Instanzierung* (früher *Instantiierung*) der Typvariablen.

Instanzierungsregel

$$\frac{\Gamma \triangleright e : \mathit{typ}}{\vdash \Gamma[\alpha_1 := \mathit{typ}_1, \ldots, \alpha_m := \mathit{typ}_m] \triangleright e : \mathit{typ}[\alpha_1 := \mathit{typ}_1, \ldots, \alpha_m := \mathit{typ}_m]}$$

Hierbei bezeichnet $[\alpha_1 := \mathit{typ}_1, \ldots, \alpha_m := \mathit{typ}_m]$ die simultane Ersetzung jedes Vorkommens von α_i durch typ_i.

(1) $\{$`x:'a, f:'a->'b`$\} \triangleright$ `f x` $:$ `'b`

(2) $\{$`x:'a, f:'a->'a`$\} \triangleright$ `f x` $:$ `'a`

(3) $\{$`x:float, f:float->int`$\} \triangleright$ `f x` $:$ `int`

Hier ist (1) eine Instanz der Applikationsregel. (2) entsteht aus (1) durch Anwenden der Instanzierungsregel mit [`'a` $:=$ `'a`, `'b` $:=$ `'a`] und (3) entsteht aus (1) mit [`'a` $:=$ `float`, `'b` $:=$ `int`].

Natürlich hätte man (2), (3) auch direkt mit der Applikationsregel bekommen können.

Typinferenz

Das OCAML System kann zu beliebig vorgegebenem (geschlossenem) Ausdruck die prinzipale Typisierung automatisch bestimmen (oder eine Fehlermeldung ausgeben, wenn es überhaupt keine Typisierung des Ausdrucks gibt.)

Dies bezeichnet man als *Typinferenz*.

Beispiel:

```
# function x -> function f -> f x;;
- : 'a -> ('a -> 'b) -> 'b = <fun>
# function x -> x x;;
                  ^

This expression has type 'a -> 'b but is here used with type 'a
#
```

Unifikation

Die Typinferenz erfolgt rekursiv über den Termaufbau.

Aus den prinzipalen Typen für die Teilausdrücke eines Ausdrucks bestimmt man den prinzipalen Typ für den Ausdruck, indem man auf die jeweiligen prinzipalen Typen der Teilausdrücke die allgemeinstmögliche Instanzierung anwendet, sodass sie zusammenpassen.

Beispiel: Habe e_1 den prinzipalen Typ `'a*('b->'c) -> ('b->'a)` und e_2 den prinzipalen Typ`('a->'a)*(int->float)`, so hat e_1 e_2 den prinzipalen Typ `int->'a->'a`.

Unifikation

Allgemein: hat der geschlossene Term e_1 den prinzipalen Typ typ_1 `->`typ_2 und hat der geschlossene Term e_2 den prinzipalen Typ typ_1', so muss man die allgemeinstmögliche Instanzierung $\sigma = [\alpha_1 := typ_1, \ldots, \alpha_m := typ_m]$ finden, sodass $typ_1\sigma = typ_2\sigma$ wird. Der prinzipale Typ von $e_1\ e_2$ ist dann $typ_2\sigma$.

Man darf dabei (ggf. nach Umbenennung) voraussetzen, dass typ_1 und typ_2 keine Variablen gemeinsam haben.

Solch ein σ heißt allgemeinster Unifikator von typ_1 und typ_2, das Auffinden desselben heißt *Unifikation*.

Gibt es keinen Unifikator oder hat e_1 keinen Funktionstyp, so existiert gar kein Typ für $e_1\ e_2$ und ein Typfehler liegt vor.

Let-Polymorphie

Bemerkung: Die Typisierungsregeln für `let` und `let rec` sind in OCAML noch etwas allgemeiner als hier dargestellt:

```
# let f = function x->x in (f 0, f 1E1);;
- : int * float = (0, 10.)
```

Nach unseren Typregeln ist `(f 0, f 1E1)` nicht typisierbar.

```
# (fun f -> (f 0, f 1e1)) (fun x -> x);;
                         ^^^
This expression has type float but is here used with type int
```

Wir behandeln toplevel Deklarationen nicht als Variablen sondern als Abkürzungen.

Z.B.: nach der toplevel Deklaration

```
let f = fun x -> x;;
```

hat `f` den Typ `'a->'a` und kann mit verschiedenen Instanzierungen verwendet werden.

Kapitel 3.6 Ausnahmen und Mustervergleich

Ausnahmen

Mit

```
exception A;;
```

wird eine *Ausnahme* namens A (engl.: *exception*) deklariert.

Der Ausdruck

```
raise A
```

bewirkt dann, dass die Berechnung an dieser Stelle abgebrochen wird.

Dieser Ausdruck hat den Typ `'a`, kann also an beliebiger Stelle auftreten.

Beispiel

```
# exception Fehler;;
exception Fehler
# let rec fakt n = if n<0 then raise Fehler else
       if n=0 then 1 else n * fakt (n-1);;
val fakt : int -> int = <fun>
# fakt 3;;
- : int = 6
# fakt (-7);;
Exception: Fehler.
```

Achtung: Namen von Ausnahmen beginnen immer mit Großbuchstaben.

Auffangen von Ausnahmen

Man kann eine Ausnahme durch das `try with` Konstrukt auffangen:

```
function x -> try string_of_int(fakt x) with
      Fehler -> "Falsches Argument!"
```

Syntax und Typisierung

⟨*expression*⟩ ::= ...

| `raise`⟨*Ident*⟩

| `try` ⟨*expression*⟩ `with` ⟨*ident*⟩ `->` ⟨*expression*⟩

{`|` ⟨*Ident*⟩ `->` ⟨*expression*⟩}*

Man kann mehrere verschiedene Ausnahmen in einem `try with` abfangen.

Typisierung: Die Typen aller `with`-Klauseln müssen mit dem Typ des Audrucks nach `try` übereinstimmen. Dieser gemeinsame Typ ist dann der Typ des gesamten try-with Konstrukts.

Übung: Man schreibe eine Typisierungsregel für try-with

Denotationelle Semantik von Ausnahmen

Die W-Funktion bildet Ausdrücke und Umgebungen ab auf $\mathcal{W} \cup \mathcal{E}$, wobei $\mathcal{W}$ die Menge der Werte ist (Konstanten und Funktionen) und $\mathcal{E}$ die Menge der Ausnahmen ist.

Im Beispiel: $\mathcal{E} = \{\texttt{Fehler}\}$.

Die semantische Funktion (W) muss nun Ausnahmen "durchreichen", z.B.:

$$W^U(t_1 \ \mathit{op} \ t_2) = \textbf{if } W^U(t_1) = A \in \mathcal{E} \textbf{ then } A \textbf{ else}$$
$$\textbf{if } W^U(t_2) = A' \in \mathcal{E} \textbf{ then } A' \textbf{ else } W^U(t_1) \oplus W^U(t_2)$$

Die Semantik von try-with (mit einer Ausnahmeklausel) ist wie folgt definiert:

$$W^U(\texttt{try } e_1 \texttt{ with } A \texttt{ -> } e_2) = \textbf{if } W^U(e_1) = A \in \mathcal{E} \textbf{ then } W^U(e_2) \textbf{ else } W^U(e_1)$$

Die operationelle Semantik kann man analog erweitern.

Simulation von Ausnahmen

Man kann Ausnahmen wie folgt simulieren

Ausnahme `Fehler` entspricht einem bestimmten Wert, z.B. -1.

```
let rec fakt n = if n<0 then -1 else
      if n=0 then 1 else n * fakt (n-1);;
```

```
function x -> let res = fakt x in
     if res = -1 then "Falsches Argument!"
     else string_of_int res
```

Nachteile: Umständlicher, nicht von vornherein klar, welche Werte den Ausnahmen entsprechen.

Ausnahmen mit Werten

Man kann einer Ausnahme einen Wert beigeben:

```
exception Fehler of int ;;
```

Hier steht ein Typ

```
let rec fakt x = if x < 0 then raise (Fehler x) else
      if x = 0 then 1 else x * fakt (x-1);;
let f x = try string_of_int (fakt x)
          with Fehler z -> "fakt: falsches Argument: " ^
                           string_of_int z
```

Test:

```
# f (-27);;
- : string = "fakt: falsches Argument: -27"
```

Man kann Syntax und Semantik auf Ausnahmen mit Werten erweitern.

Ein Fehler bei der Vorbereitung der Folien

Was ist hier falschgemacht worden?

```
(* Fehler und fakt wie gehabt *)
let f x = let res = fakt x in try string_of_int res with
                Fehler z -> "fakt: falsches Argument: " ^
                            string_of_int z
# f (-19);;
Exception: Fehler -19.
```

Eingebaute Ausnahmen

```
# (fun x->x) <= (fun x-> x);;
Exception: Invalid_argument "equal: functional value".
# String.sub "op" (-1);;
- : int -> string = <fun>
# String.sub "op" (-1) 9;;
Exception: Invalid_argument "String.sub".
# 5/0;;
Exception: Division_by_zero.
```

Die Deklarationen

```
Exception Invalid_argument of string
Exception Division_by_zero
```

werden also automatisch zu Beginn vorgenommen.

(Dazu noch viele andere!)

Zusammenfassung Ausnahmen

- Ausnahmen dienen der Behandlung von Fehlern.
- Mit `raise` A wird Ausnahme A ausgelöst.
- Ausnahmen werden durch alle Konstrukte (außer try-with) durchgereicht.
- Mit `try with` können Ausnahmen aufgefangen werden.
- Ausnahmen können Werte beinhalten, auf die in try-with zugegriffen werden kann.
- Es gibt eingebaute Ausnahmen, wie "`Invalid_argument`".

Mustervergleich

Man kann Werte mit einem vorgegebenen *Muster* vergleichen (engl.: *pattern matching*).

```
let rec fakt n = match n with
   0 -> 1
|  n -> n * fakt (n-1)
```

Weitere Möglichkeit:

```
let rec fakt n = match n with
    n when n<0 -> raise (Fehler n)
|   0 -> 1
|   n -> n * fakt (n-1)
```

Allgemein:

`match` ⟨*expression*⟩ `with` ⟨*pattern*⟩ `->` ⟨*expression*⟩
{`|` ⟨*pattern*⟩ `->` ⟨*expression*⟩}*

Weitere Beispiele

```
let rec ack x = match x with
          (0,n) -> n + 1
        | (m,0) -> ack(m-1,1)
        | (m,n) -> ack(m-1,ack(m,n-1))
let roemisch x = match x with
           0 -> "" | 1 -> "I" | 5 -> "V" | 10 -> "X" | 50 -> "L"
         | 100 -> "C" | 500 -> "D" | 1000 -> "M"
```

Form der Muster

$$
\begin{array}{lcl}
\langle \mathit{pattern} \rangle & ::= & \langle \mathit{ident} \rangle \\
& | & \langle \mathit{const} \rangle \\
& | & \_ \\
& | & \texttt{(} \langle \mathit{pattern} \rangle \{ \texttt{,} \ \langle \mathit{pattern} \rangle \}^* \texttt{)} \\
& | & \langle \mathit{pattern} \rangle \ \texttt{when} \ \langle \mathit{expression} \rangle
\end{array}
$$

Kontextbedingung: eine `when`Klausel darf nur ganz außen stehen, also nicht `(0,m when m>0)`. Stattdessen: `(0,m) when m>0`.

Später lernen wir noch andere Muster kennen.

Bedeutung der Muster

Gegeben ein Wert w und ein Muster p. Es kann eine der folgenden beiden Möglichkeiten auftreten:

- w passt nicht auf das Muster p
- w passt auf das Muster p; das *Ergebnis* dieses Abgleichs ist eine *Umgebung*, die alle Bezeichner in p an Werte bindet.

Wann passt welches Muster auf w?

- x passt immer; das Ergebnis ist $\{<x, w>\}$
- _ (underscore, *wildcard*) passt immer; das Ergebnis ist $\{\}$
- `(`p_1`,` $\ldots$ `,`p_m`)` passt, wenn $w = (w_1, \ldots, w_m)$ ist und w_i jeweils auf p_i passt (mit Ergebnis U_i. Ergebnis ist dann $U_1 + \cdots + U_m$.
  Bemerkung: kein Bezeichner darf in zwei verschiedenen p_i auftauchen.
- p `when` e passt, wenn erstens w auf p passt (mit Ergebnis U) und zweitens $W^U(e) =$ *true*. Ergebnis ist dann U.

Bedeutung von match-with

Sei

$$e = \texttt{match}\ e'\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{->}\ e_n$$

Wir wollen $W^U(e)$ bestimmen.

Sei $w = W^U(e)$ und sei p_{i_0} das **erste** Muster aus $p_1, \ldots, p_n$, auf das w passt, mit Ergebnis U_{i_0}. Gibt es kein solches Muster, so ist $W^U(e)$ die Ausnahme `Match_failure`.

Ansonsten ist $W^U(e) = W^{U+U_{i_0}}(e_{i_0})$.

Zur Beachtung: `Match_failure` ist so deklariert:

```
exception Match_failure of string * int * int
```

Tritt `Match_failure`(f, i, j) auf, so bedeutet das, dass der Mustervergleich in Datei f zwischen den Positionen i und j aufgetreten ist. In Xemacs springt man mit M-x goto-char an eine Position.

Typisierung von match-with

Damit `match` e' `with` p_1 `->` e_1 `|` $\ldots$ `|` p_n `->` e_n (unter Typzuweisung Γ) typkorrekt ist, muss zunächst e' einen Typ `typ'` haben, d.h., $\Gamma \triangleright e' : \mathit{typ'}$.

Sodann ist für jedes Muster p_i eine Typzuweisung Γ_i zu bestimmen, sodass $\Gamma + \Gamma_i \triangleright p_i : \mathit{typ'}$.

Es muss dann einen Typ *typ* geben, sodass für alle i gilt: $\Gamma + \Gamma_i \triangleright e_i :$ `typ`.

Gelingt all das, so gilt $\Gamma \triangleright e : \mathit{typ}$.

Übung: Man formalisiere das als Typisierungsregel.

Spezialformen

Statt

```
fun x -> match x with ...
```

kann man auch

```
function ...
```

schreiben.

Beispiel:

```
function 0 -> 1 | 1 -> 0 | x -> x-1
```

In `let` und `fun` Ausdrücken dürfen auch Muster verwendet werden, aber keine Alternativen (“|”):

```
let plus_bruch (z1,n1) (z2,n2) = (z1*n2 + z2*n1, n1*n2)
let zaehler = fun (z,_) -> z
let hauptnenner u v = let (_,n) = plus_bruch u v in n
```

Kapitel 4: Strukturierte Daten

Rechenstrukturen

Ein *Rechenstruktur* ist eine Menge von Datentypen zusammen mit einer Menge von Operationen (= Funktionen, auch nullstellige, also Konstanten) auf diesen Typen.

Ist ein Datentyp *typ* speziell ausgezeichnet, so spricht man von der *Rechenstruktur von typ*.

Beispiel: Die Rechenstruktur von **bool** hat als Operationen die Konstanten *true*, *false*, sowie $\wedge, \vee, \neg$.

In OCAML ist diese Rechenstruktur fest eingebaut.

Die Rechenstruktur von **nat** ist dagegen nicht fest eingebaut, kann aber durch den Typ `int` realisiert werden.

Verbunde (*Records*)

Zunächst ein Beispiel: Mit

```
type studierende = {name:string; alter:int}
```

definieren wir einen Typ von Studierenden.

```
# let maier1 = {name = "Maier"; alter = 23};;
val maier1 : studierende = {name = "Maier"; alter = 23}
# let maier2 = {name = "Maier"; alter = 22};;
val maier2 : studierende = {name = "Maier"; alter = 22}
# let string_of_studierende x = x.name ^ ", " ^
            string_of_int x.alter ^ " Jahre alt.";;
val string_of_studierende : studierende -> string = <fun>
# string_of_studierende maier2;;
- : string = "Maier, 22 Jahre alt."
```

Weitere Beispiele

```
type complex = {re:float; im:float}
type rational = {num:int; denum:int}
type point = {x:float; y:float}
type name_t = {vorname:string; initial:char; nachname:string}
type datum = {tag: int; monat: int; jahr: int}
type mitarbeiter = {name: name_t;
                    mitarb_nr: int;
                    gehalt: float;
                    dienstantritt: datum}
```

Allgemein

Sind $t, f_1, \ldots, f_n$ Bezeichner und $typ_1, \ldots, typ_n$ Typen, so wird durch

$$\texttt{type } t \texttt{ = } \{f_1 \texttt{:} typ_1 \texttt{;} \ldots \texttt{;} f_n \texttt{:} typ_n\}$$

ein neuer Typ t definiert, dessen Werte von der Form

$$\{f_1 \texttt{=} w_1 \texttt{;} \ldots \texttt{;} f_n \texttt{=} w_n\}$$

sind, wobei jeweils w_i ein Wert des Typs typ_i ist.

Bemerkung: Wir haben nicht formal definiert, was ein “Wert eines Typs” ist und werden das auch nicht tun.

Dieser Typ t heißt *Verbundtyp* oder auch *Recordtyp*. Die f_i heißen *Felder* des Verbundtyps.

Typisierungsregeln für Verbunde

Falls $t = \{x_1 : typ_1 ; \ldots ; x_n : typ_n\}$ deklariert wurde, so hat man folgende Regel für Konstruktion von Verbunden:

$$\Gamma \triangleright e_1 : typ_1, \ldots, \Gamma \triangleright e_n : typ_n \vdash \Gamma \triangleright \{f_1 = e_1 ; \ldots ; f_n = e_n\} : t$$

Außerdem folgende Regel für die Selektion von Komponenten:

$$\Gamma \triangleright e : t \vdash \Gamma \triangleright e.f_i : typ_i$$

Achtung: Jeder Verbundtyp muss vorher deklariert werden. Hat man mehrere Verbundtypen mit überlappenden Feldern deklariert, so treten zunächst bizarre Phänomene auf. Versuchen Sie, die empirisch zu erklären!

Varianten (ja die Dinger heißen so)

```
type figur = Kreis of point * float
           | Rechteck of point * point
           | Dreieck of point * point * point
let flaeche = function
          Kreis(m,r) -> r *. r *. 4.*.atan 1.
        | Rechteck(p1,p2) ->
                abs_float((p1.x -. p2.x)*.(p1.y -.  p2.y))
        | Dreieck(p1,p2,p3) ->
                let ux = p1.x -. p2.x in
                let uy = p1.y -. p2.y in
                let vx = p3.x -. p2.x in
                let vy = p3.y -. p3.y in
                abs_float((ux *. vy -. vx *. uy) /. 2.)
```

Benutzung

```
val flaeche : figur -> float = <fun>
# let a = {x=0.; y=0.};;
val a : point = {x = 0.; y = 0.}
# let b = {x=1.; y=1.};;
val b : point = {x = 1.; y = 1.}
# let c = {x=0.; y=1.};;
val c : point = {x = 0.; y = 2.}
# flaeche (Dreieck(a,b,c));;
- : float = 0.5
# let k = Kreis(a, 2.0);;
val k : figur = Kreis ({x = 0.; y = 0.}, 2.)
# flaeche k;;
- : float = 12.5663706144
#
```

Allgemein

Konstruktoren sind wie Bezeichner definiert, beginnen aber mit einem Großbuchstaben. Bsp.: `Kreis`, `Dreieck`, `Datum`, `X`, `Zz_`

Ist t ein Bezeichner und sind $F_1, \ldots, F_n$ Konstruktoren und sind $typ_1, \ldots, typ_n$ Typen, so wird durch

$$\texttt{type } t \texttt{ = } F_1 \texttt{ of } typ_1 \texttt{|} \ldots \texttt{|} F_n \texttt{ of } typ_n$$

ein neuer Typ t definiert, dessen Werte von der Form $F_i(w_i)$ sind, wobei $i \in \{1, \ldots, n\}$ und w_i ein Wert des Typs typ_i ist.

Beispiel: `type t = A of int | B of int`

Die Werte des Typs `t` sind
$\{\mathtt{A}(0), \mathtt{B}(0), \mathtt{A}(1), \mathtt{B}(1), \mathtt{A}(-1), \mathtt{B}(-1), \mathtt{A}(2), \mathtt{B}(-2), \mathtt{A}(3), \mathtt{B}(-3), \ldots\}$.

"Mathematisch" ist das dasselbe, wie
$\{(0,0), (1,0), (0,1), (1,1), (0,-1), (1,-1), \ldots\}$.

Der Typ t heißt *Variante* (en.: *variant*). Die F_i sind die *Konstruktoren von* t.

Zusätzlicher Sonderfall

Es gibt auch Konstruktoren ohne Argumente:

```
type t = A of int | B of int | C
```

Hier sind die Werte
$\{\mathsf{C}, \mathsf{A}(0), \mathsf{B}(0), \mathsf{A}(1), \mathsf{B}(1), \mathsf{A}(-1), \mathsf{B}(-1), \mathsf{A}(2), \mathsf{B}(-2), \mathsf{A}(3), \mathsf{B}(-3), \ldots\}$.
Oder "mathematisch"
$\{(2,0), (0,0), (1,0), (0,1), (1,1), (0,-1), (1,-1), \ldots\}$.

Weitere Beispiele:

```
type farbe = Rot | Gruen | Blau | RGB of int*int*int
type traffic_light = Red | Amber | Green
```

Eine Variante mit ausschließlich konstanten Konstrukten (wie `traffic_light`) heißt *Aufzählungstyp*.

Verwendung von Varianten

Ist in der Variante t ein Konstruktor F `of` *typ* vorhanden so hat man folgende Typisierungsregel:

$$\Gamma \triangleright e : \mathit{typ} \quad \vdash \quad \Gamma \triangleright F\ e : t$$

Ist p ein Muster, so ist auch Fp ein Muster. Ein Wert w passt auf das Muster Fp, wenn gilt $w = F(w')$ und w' auf p passt. Liefert dieser Vergleich Ergebnis U, so ist U das Ergebnis des Vergleichs von w mit $F(w)$.

Beispiel: der Wert $\mathsf{RGB}(181, 158, 12)$ passt auf das Muster $\mathsf{RGB}(r, g, \_)$. Ergebnis ist $\{<\mathsf{r}, 181>, <\mathsf{g}, 158>\}$.

Listen

Ist A eine Menge, so ist A **list** $= \bigcup_{n \geq 0} A^n$ die Menge der *Listen* über A.

Fasst man A als Alphabet auf, so ist A **list** $= A^*$.

Man notiert Listen als $[a_1; \ldots; a_n]$ anstatt $(a_1, \ldots, a_n)$ oder $a_1 \cdots a_n$.

Die leere Liste wird mit **nil** oder $[]$ bezeichnet.

Ist $a \in A$ und $l = [a_1; \ldots; a_n] \in A$ **list**, so bezeichnet **cons**(a, l) die Liste $[a; a_1; \ldots; a_n]$.

Man schreibt auch $a :: l$ für **cons**(a, l).

Die Verkettung von Listen l_1 und l_2 schreiben wir $l_1 \mathbin{@} l_2$. Es gilt

$$[] \mathbin{@} l_2 = l_2$$
$$(a :: l) \mathbin{@} l_2 = a :: (l \mathbin{@} l_2)$$

Beispiele

```
# let x = 17::[];;
val x : int list = [17]
# let y = 9 :: 2 :: x;;
val y : int list = [9; 2; 17]
# x = y;;
- : bool = false
# x @ y;;
- : int list = [17; 9; 2; 17]
# [x;y];;
- : int list list = [[17]; [9; 2; 17]]
# [(9,2); (3,5)];;
- : (int * int) list = [(9, 2); (3, 5)]
# [2;true];;
Characters 3-7:
  [2;true];;
     ^^^^
This expression has type bool but is here used with type int
```

Neue Muster für Listen

Auf das Muster ...

- `[]` passt die leere Liste **nil**,
- `[`p_1`;` ... `;`p_n`]` passt die Liste $[w_1; \ldots; w_n]$, wenn w_i auf p_i passt.
- p_1`::`p_2 passt $w_1 :: w_2$, wenn w_i auf p_i für $i = 1, 2$ passt.

Grundalgorithmen für Listen

```
let rec length = function
        [] -> 0
    | _::l -> 1 + length l

let rec enthalten = function
      ([],x) -> false
    | (h::t,x) -> x=h || enthalten(t,x)

let rec rev = function
        [] -> []
    | h::t -> rev t @ [h]

let hd (h::t) = h

let tl (h::t) = t

let null l = l = []
```

Effizientes Spiegeln

```
let rec rev_aux = function
     ([], acc) -> acc
   | (h::t,acc) -> rev_aux(t,h::acc)

let rev2 l = rev_aux(l,[])
```

Man erprobe `rev(mklist 10000)` und `rev2(mklist 10000)` wobei `mklist` n eine Liste der Länge n ist.

Man zeige mit dem Satz von der partiellen Korrektheit, dass gilt:

$$\texttt{rev\_aux}(l, acc) = \texttt{rev}(l) \mathbin{@} acc$$

Sortieren durch Einfügen

Eine Liste $[x_1; \ldots; x_n]$ heißt *sortiert*, wenn $x_1 \leq x_2 \leq \cdots \leq x_n$.

```
let rec insertel = function
   (a, []) -> [a]
 | (a, h::t) -> if a <= h then a::h::t else h :: insertel(a,t)

let rec inssort = function
      [] -> []
   | h::t -> insertel(h,sort l)
```

Ist l sortiert, so auch `insertel`(a,l) und es enthält dieselben Elemente wie $a :: l$.

Für beliebiges l ist `inssort`(l) sortiert und enthält dieselben Elemente wie l.

Anwenden einer Funktion auf alle Elemente einer Liste

```
let rec map f l = match l with
        [] -> []
      | h::t -> f h :: map f t
```

Beispiel:

```
# map (fun x -> x * x) [1;2;3;4];;
- : int list = [1; 4; 9; 16]
# map string_of_int [1;2;3;4];;
- : string list = ["1"; "2"; "3"; "4"]
# map (map (fun x -> x*x)) [[1;2]; [3]; [4;5;6]];;
- : int list list = [[1; 4]; [9]; [16; 25; 36]]
```

Weitere Funktionen als Übung

Zerlegen in zwei Teile

```
val split : ('a - > bool) -> 'a list -> 'a list * 'a list
```

Verflachen

```
val flatten : 'a list list -> 'a list
```

Falten

```
val fold_right : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

`fold_right` $f\ x\ [x_1; \ldots; x_n] = f\ x_1 (f\ x_2\ \ldots (f\ x_n\ x)) \ldots)$

Beispiel

Kontovorgänge liegen als Liste von floats vor.

Buchungsbetrag $\leq$ 1000EUR : Gebühr 30 Cent

Buchungsbetrag $>$ 1000EUR : Gebühr 50 Cent

Man schreibe eine Funktion, die die Gesamtgebühr bestimmt.

Zunächst direkt, dann mit `fold_right`.

Weitere Anwendungsbeispiele in [Kröger] und der OCAML Bibliothek.

Stapel

Stapel (Keller, *stack*): Endliche Listen mit Basisfunktionen **nil**, **cons**, **hd**, **tl**, **null** (hier meist bezeichnet als **empty**, **push**, **top**, **pop**, **isempty**).

Idee: Man legt Dinge auf dem Stapel ab und bekommt sie in der Reihenfolge "*last-in-first-out*" (LIFO) zurück.

Der Stapel ist eine häufig verwendete Datenstruktur. Man kann mit ihm z.B. Rekursion simulieren (wird hier nicht behandelt).

Schlange

Schlangen: Endliche Folgen mit den Basisfunktionen **nil**, **hd**, **tl**, sowie **enter** mit Bedeutung **enter**$(l, x) = l@[x]$.

Idee: Dinge werden in die Schlange eingereiht und in der Reihenfolge *first-in-first-out* (FIFO) wieder entnommen.

Schlangen werden sowohl als echte Warteschlangen, als auch als Hilfsdatenstruktur in bestimmten Algorithmen verwendet.

Bemerkung: Die naive Implementierung von Schlangen als Listen ist ineffizient, da die Basisfunktion **enter** Zeit proportional zur Länge der Schlange verbraucht.

Reihungen

Reihungen (Vektoren, *Arrays*): Endliche Folgen mit den Basisfunktionen **init**, **dim**, **get**, **update** mit den Bedeutungen

$$\textbf{init}(n, x) = \underbrace{[x; x; \ldots; x]}_{n \text{ Einträge}}$$

$$\textbf{dim}([x_1; \ldots; x_n]) = n$$

$$\textbf{get}([x_1; \ldots; x_n], i) = x_i$$

$$\textbf{update}([x_1; \ldots; x_n], i, x) = [x_1; \ldots; x_{i-1}; x; x_{i+1}; \ldots; x_n]$$

Im Falle unpassender Indizes sind diese Operationen undefiniert, z.B.: **get**$([x_1; x_2], 3)$ oder **init**$(-2, 0.0)$.

Wir schreiben α **vect** für den Typ der Reihungen mit Einträgen vom Typ α.

Man kann Reihungen als OCAML-Listen implementieren; bei großer Dimension ist das aber ineffizient.

Binäre Suche mit Reihungen

Die folgenden Funktionen setzen voraus, dass die Reihung l aufsteigend sortiert ist.

$$\begin{array}{l}
\text{sucheVonBis} = \textbf{function}(l\text{:}\textbf{string vect}, w\text{:}\textbf{string}, i\text{:}\textbf{int}, j\text{:}\textbf{int})\textbf{bool} \\
\quad \textbf{if } i > j \textbf{ then false else} \\
\quad \textbf{if } i = j \textbf{ then get}(l,i) = w \textbf{ else} \\
\quad\quad \textbf{let } m = \lfloor (i+j)/2 \rfloor \textbf{ in} \\
\quad\quad \textbf{let } w_m = \textbf{get}(l,m) \textbf{ in} \\
\quad\quad \textbf{if } w_m = w \textbf{ then true else} \\
\quad\quad\quad \textbf{if } w < \textbf{get}(l,m) \textbf{ then} \\
\quad\quad\quad\quad \text{sucheVonBis}(l,w,i,m-1) \\
\quad\quad\quad\quad \textbf{else } \text{sucheVonBis}(l,w,m+1,j) \\
\text{suche} = \textbf{function}(l\text{:}\textbf{string vect}, w\text{:}\textbf{string})\text{sucheVonBis}(l,w,1,\textbf{dim}(l))
\end{array}$$

Binärbäume

Definition: Sei A eine Menge. Die Menge $A^\triangle$ der *Binärbäume über A* ist induktiv definiert wie folgt:

1. $A^\triangle$ enthält den *leeren Binärbaum* τ
2. Sind x in A und $l, r \in A^\triangle$, so ist das 3-Tupel $(x, l, r) \in A^\triangle$

Man kann die induktive Definition wie folgt umgehen: Die Menge $A^\triangle_n$ der *Höhe höchstens n* ist rekursiv (über n) definiert durch

1. $A^\triangle_0 = \{\tau\}$
2. $A^\triangle_{n+1} = A^\triangle_n \cup \{(x, l, r) \mid x \in A, l \in A^\triangle_n, r \in A^\triangle_n\}$.

Es ist dann $A^\triangle = \bigcup_{n \geq 0} A^\triangle_n$. Die *Höhe* eines Binärbaumes $t \in A^\triangle$ ist das kleinste n, sodass $t \in A^\triangle_n$.

Terminologie

x heißt *Wurzel* oder *Wurzelbeschriftung* von (x, l, r).

l, r heißen *linker, bzw. rechter Unterbaum* von (x, l, r). Ein Binärbaum der Form (x, τ, τ) heißt *Blatt*. Ein von τ verschiedener Binärbaum heißt *nichtleer*.

Die *Knoten* und *Teilbäume* eines Binärbaums sind rekursiv definiert wie folgt:

1. τ hat keine Knoten und nur τ als Teilbaum.
2. Die Knoten von (x, l, r) sind x und die Knoten von l und die Knoten von r. Die Teilbäume von (x, l, r) sind (x, l, r) und die Teilbäume von l und von r.

Beispiel

$t = (6, (3, (2, \tau, \tau), (8, \tau, (5, \tau, \tau))), (8, (4, \tau, \tau), \tau))$.

Knoten von t: $\{6, 3, 2, 8, 5, 4\}$.

Teilbäume von t: $\{t, (3, (2, \tau, \tau), (8, \tau, (5, \tau, \tau))),$
$(2, \tau, \tau), (8, \tau, (5, \tau, \tau)), (5, \tau, \tau), (8, (4, \tau, \tau), \tau)), (4, \tau, \tau), \tau\}$

Realisierung in OCAML

Binärbäume sind nicht fest eingebaut, können aber leicht als *rekursive Variante* definiert werden.

```
type 'a bintree = Empty | Build of 'a * 'a bintree * 'a bintree
```

Nun können wir Binärbäume bilden und Funktionen darauf definieren:

```
let t = Build(6,Build(3,Build(2,Empty,Empty),
                        Build(8,Empty,Build(5,Empty,Empty))),
                Build(8,Build(4,Empty,Empty),Empty)
              )
let root (Build(x,_,_)) = x (* unvollst Mustervergleich *)
let left (Build(_,l,_)) = l
let right (Build(_,_,r)) = r
let isempty t = (t = Empty)
let rec knotanz t = if isempty t then 0 else
         1 + knotanz (left t) + knotanz (right t)
let rec hoehe t = if isempty t then 0 else
         1 + max (hoehe (left t))  (hoehe (right t))
```

Binärbäume als Rechenstruktur

Der Datentyp `'a bintree` und die Operationen `Empty, Build, isempty, left, right` bilden die *Rechenstruktur der Binärbäume*.

In [Kröger, S62] wird diese zusätzlich auf der Pseudocodeebene eingeführt.

Weitere Grundalgorithmen als Übung

Gleichheit zweier Binärbäume:

```
bbeq : ’ a bintree * ’a bintree -> bool
```

Suchen eines Datenelements in einem Binärbaum:

```
enthalten2 : ’ a * ’a bintree -> bool
```

Linearisierungen

Im folgenden definieren wir drei Funktionen

```
linvor, linsym, linnach
```

jeweils vom Typ

```
'a bintree -> 'a list
```

welche die Knoten eines Baums in einer bestimmten Reihenfolge als Liste berechnen.

Vorordnung

```
let rec linvor = function
    Empty -> []
  | Build(a,l,r) -> a :: linvor l @ linvor r

let t = Build(6,Build(3,Build(2,Empty,Empty),
                        Build(8,Empty,Build(5,Empty,Empty))),
                Build(8,Build(4,Empty,Empty),Empty)
              )
linvor t;;
- : int list = [6; 3; 2; 8; 5; 8; 4]
```

Symmetrische Ordnung

```
let rec linsym = function
    Empty -> []
  | Build(a,l,r) -> linsym l @ [a] @  linsym r

let t = Build(6,Build(3,Build(2,Empty,Empty),
                        Build(8,Empty,Build(5,Empty,Empty))),
                Build(8,Build(4,Empty,Empty),Empty)
              )
linsym t;;
- : int list = [2; 3; 8; 5; 6; 4; 8]
```

Nachordnung

```
let rec linnach = function
    Empty -> []
  | Build(a,l,r) -> linnach l @ linnach r @ [a]

let t = Build(6,Build(3,Build(2,Empty,Empty),
                        Build(8,Empty,Build(5,Empty,Empty))),
                Build(8,Build(4,Empty,Empty),Empty)
              )
linnach t;;
- : int list = [2; 5; 8; 3; 4; 8; 6]
```

Anwendung: Repräsentation von Termen

```
let t = Build("-",
              Build("+",
                    Build("10",Empty,Empty),
                    Build("*",Build("x",Empty,Empty),
                              Build("85",Empty,Empty))),
              Build("+",
                    Build("124",Empty,Empty),
                    Build("y1",Empty,Empty)))
linnach t;;
- : string list =
    ["10"; "x"; "85"; "*"; "+"; "124"; "y1"; "+"; "-"]
```

Terme lassen sich als Bäume repräsentieren. Die Nachordnung entspricht der Postfixnotation (vgl.: HP Taschenrechner)

Bessere Repräsentation von Termen

Nachteile der vorigen Repräsentation: Sehr viel “`Empty`”, auch syntaktisch falsche Terme haben Repräsentation:
“`Build("x",Build(...),Build(...)`”

Besser ist die Verwendung einer speziellen rekursiven Variante:

```
type op = Plus | Minus | Mal | Geteilt
type expr = Var of string | Num of int | Op of op * expr * expr

let t = Op(Minus,Op(Plus,Num 10,Op(Mal,Var "x", Num 85)),
                 Op(Plus,Num 124, Var "y1"))
```

Auswertung solcher Terme

Repräsentiere Umgebung als *Liste* von Paaren Variable-Zahl, z.B.:

$$\{<\texttt{"x"}, 9>, <\texttt{"y"}, 0>, <\texttt{"z"}, 3>\} \mapsto [(\texttt{"x"}, 9); (\texttt{"y"}, 0); (\texttt{"z"}, 3)]$$

Einfügen mit `::`

Auslesen mit

```
List.assoc : 'a -> ('a * 'b) list -> 'b
```

Eine so verwendete Liste von Paaren heißt *Assoziationsliste*.

Rekursive Auswertung

```
let rec eval t env = match t with
    Num x -> x
|   Var x -> List.assoc x env
|   Op(op,t1,t2) -> let v1 = eval t1 env in
                    let v2 = eval t2 env in
                    (match op with
                        Plus -> v1 + v2
                    |   Minus -> v1 - v2
                    |   Mal -> v1 * v2
                    |   Geteilt -> v1 / v2)
```

Parsing

```
type token = TokNum of int | TokL | TokR |
             TokOp of op | TokVar of string
```

Übung: Man definiere eine (rekursive) Funktion

```
parse : token list -> token list * expr
```

derart, dass aus `parse l = (rest,e)` folgt, dass `l = anfang @ rest` und `anfang` den Ausdruck `e` bezeichnet. Außerdem soll `rest` so kurz wie möglich gewählt werden.

Beginnt `l` nicht mit der Darstellung eines Ausdrucks, so wird die (selbstdefinierte) Ausnahme `ParseError` ausgelöst.

Beispiel:

```
parse [TokNum 13;TokNum 13] = ([TokNum 13], Num 13)
parse [TokNum 2;TokOp Plus;TokL;TokNum 3;TokOp Mal;TokNum 5;TokR]=
      ([], Op (Plus, Num 2, Op (Mal, Num 3, Num 5)))
parse [TokR] --> ParseError
```

Binärer Suchbaum

Definition: Sei t binärer Baum mit Knoten aus `int`. Der Baum t heißt *binärer Suchbaum* (*binary search tree, BST*), wenn $t = \tau$ oder $t = (a, l, r)$ und

- Für jeden Knoten x von l gilt $x \leq a$.
- Für jeden Knoten x von r gilt $a \leq r$.
- l und r sind selbst wiederum binäre Suchbäume.

Effizienteres Suchen in binären Suchbäumen

Folgende Funktion stellt fest, ob ein Knoten in einem binären Suchbaum enthalten ist:

```
let rec enthaltenBST(x,t) = match t with
        Empty -> false
|       Build(a,l,r) -> x=a ||
                  (if x<=a then enthaltenBST(x,l)
                           else enthaltenBST(x,r))
```

Es werden nur die Knoten auf dem Pfad von der Wurzel zum gesuchten Element, bzw. bis zu einem Blatt angeschaut.

Dagegen werden bei `enthalten2` *alle* Knoten angeschaut.

Dafür funktioniert aber `enthaltenBST` nur für binäre Suchbäume.

Kapitel 5: Methodisches Programmieren

Beispiel einer komplexen Anwendung

In einer Firma sollen durch ein Programm Quittungen erstellt werden:

Fa. L. Kaufgut *12.1.2004*

Ladenstr. 17

77777 Handelsstadt

QUITTUNG

Wir bestätigen, von Herrn P. Schulze

EUR 218,37 (zweihundertachtzehn)

erhalten zu haben.

Diese Quittung wurde maschinell erstellt und trägt keine Unterschriften.

Beispiel einer komplexen Anwendung

Aufgabe: Bei Eingabe von Datum, Kundenname und Betrag soll eine Quittung in dieser Form erstellt werden.

Modellierung:

Kundenname:	`string`
Quittungstext:	`string`
Datum:	`string` oder `datum_t` (benutzerdefiniert)
Betrag:	`float`

Grobgliederung in Teilaufgaben

- Erzeugung der Wortdarstellung zum Zahlungsbetrag,
- Einsetzen der Eingaben und der erzeugten Wortdarstellung in den festen Quittungstext,
- Erstellung der endgültigen Fassung gemäß gewünschtem Layout. Evtl. Zeilenumbruch erforderlich.
- Eingabe der Parameter, Drucken der Quittung.

Modulkonzept

Ein *Modul* ist eine Menge von Datentypen zusammen mit einer Menge von Funktionen auf diesen (und anderen) Typen und evtl. noch weiteren Bestandteilen.

Rechenstrukturen sind insbesondere Module.

Im allgemeinen enthält ein Modul interne Bestandteile, die von außen nicht verwendbar sein sollen. Diejenigen Bestandteile, die von anderen verwendbar sind, heißen *Schnittstelle*.

Modularisierung bezeichnet die Zerlegung einer Aufgabe in Teile und Festlegung der für die einzelnen Teile zu erstellenden Schnittstellen.

Modularisierung im Beispiel

Module für die Datentypen `string` und `float`, dazu:

`Wortdarst`	Erzeugung der Wortdarstellung zum Betrag
`Trennen`	Korrekte Trennung von Wortdarstellungen
`Quittung`	Einsetzen der Eingaben, Erstellung der endg. Quittung

Schnittstelle von `Wortdarst`

konvert $=$ **function**(x:**float**) :

pre $x \geq 0$

result *konvert*(x) $=$ Wortdarstellung von x

Centanteile werden nicht berücksichtigt

Verwendung eines Moduls

Die Schnittstelle definiert die nach außen sichtbaren Datentypen und Funktionen mit ihren Typen.

Die Implementierung der Funktionen wird nicht sichtbar gemacht.

Von außen benutzt man die Funktionen allein aufgrund ihrer Spezifikation.

Spezifikation = Informelle oder formale Beschreibung der Funktion.

Vorteile der Modularisierung

- Strukturierung großer Programmsysteme
- Abkapselung: Implementierung des Moduls von dessen Verwendung getrennt.
- Verstecken von Hilfsfunktionen: Nur explizit in der Schnittstelle aufgeführte Funktionen und Typen sind von außen zu verwenden. Vermeidung von Namenskonflikten
- Änderungen überschaubar durchführbar
- Einzelimplementierungen unabhängig voneinander.

Arten der Modularisierung

Es gibt unterschiedliche Leitlinien für die Abgrenzung von Modulen:

- Problemorientierung: Zusammenfassung von Algorithmen die ein bestimmtes Teilproblem lösen.
- Datenorientierung: Zusammenfassung geeigneter Datentypen und zugehörige Funktionen.
- Funktionsorientierung: Zusammenfassung funktional verwandter Algorithmen, z.B.: “Statistikbibliothek”.

Objektorientierung (2. Semester) versucht, diese Arten der Modularisierung miteinander zu verbinden.

Programmentwicklung

Die Modulstruktur liegt nur selten von vornherein statisch vor. Stattdessen:

- Weitere Aufteilung eines Moduls in Teilmodule.
- Zusammenfassung bisher getrennter Module in eins, z.B.: `Trennen` in `Wortdarst` integrieren.
- Erweitern von Schnittstellen
- Veränderung der Spezifikation von Schnittstellen

Modularisierung in OCAML

Schnittstellen werden in OCAML durch *Signaturdeklarationen* festeglet,

Module werden in *Strukturdeklarationen* beschrieben.

Beispiel:

```
module type WortdarstSig = sig
val konvert : float -> string
end

module Wortdarst : WortdarstSig = struct
let pi = 3.141
let konvert x = "*** Rechnungsbetrag ***" (* muss verbessert werde
end
```

Zugriff auf Module

Jedes Modul hat eine Signatur; ist keine explizit angegeben, so wird eine solche automatisch erstellt.

Auf die Komponente x des Moduls M wird mit $M.x$ zugegriffen, z.B.: ist

```
Wortdarst.konvert : float -> string
```

Beachte: `Wortdarst.pi` ist nicht definiert.

Durch die Deklaration `open` M wird der Modul M geöffnet, man kann dann die Bestandteile der Schnittstelle direkt verwenden. Die "versteckten" Bestandteile, wie `pi` aber nicht.

Vordefinierte Module

Im OCAML System sind zahlreiche Module bereits vordefiniert, so z.B.:
`List`, `String`, `Array`, `Graphics`, ...

Übung

Man definiere OCAML-Signaturen für die folgenden Rechenstrukturen:

- Stapel
- Binärbaum
- Reihung
- Schlange
- Ring (Menge mit Konstanten 0, 1 und Operationen $+, \times$)
- Endliche Funktionen (implementierbar etwa mit Assoziationslisten)
- Endliche Mengen

Anwendung: Breitenordnung

Man gebe die Knoten eines Binärbaums in der *Breitenordnung* aus, d.h., Knoten mit kleinerer Höhe zuerst, bei Knoten gleicher Höhe die weiter links stehenden zuerst.

```
let t = Build(6,Build(3,Build(2,Empty,Empty),
                         Build(8,Empty,Build(5,Empty,Empty))),
              Build(8,Build(4,Empty,Empty),Empty)
             )
linbreit t;;
- : int list = [6; 3; 8; 2; 8; 4; 5]
```

Lösung: man schreibe eine allgemeinere Funktion, die die Knoten einer *Schlange* von Bäumen in der Breitenordnung ausgibt.

Bemerkung: ersetzt man “Schlange” durch “Stapel”, so erhält man eine effiziente Implementierung der Vorordnung.

Überschattung

- Variablen (Funktionsparameter, let-gebundene Variablen, etc.) haben einen bestimmten *Gültigkeitsbereich* in dem sie unter ihrem Namen verwendbar sind.
- Ausnahme: innerhalb ihres Gültigkeitsbereichs können Variablen durch andere Variablen des gleichen Namens *verschattet* (auch *überschattet*) werden.
- Die in Kapitel 3 eingeführte Semantik trägt diesen Umständen bereits Rechnung.
- Parameter einer übergeordneten Funktion bleiben in einer untergeordneten Funktion sicht- und verwendbar. Man muss sie daher nicht explizit übergeben. [Kröger] bezeichnet das als *Parameterunterdrückung*.

Beispiel: Suche in Reihungen

```
let enthalten1(x,a) =
   let rec enthallg(x,k,a) =
          if k>dim(a) then false
          else a=get(x,k) || enthallg(x,k+1,a)
   in enthallg(x,1,a)
let enthalten1'(x,a) =
   let rec enthallg k =
          if k>dim(a) then false
          else a=get(x,k) || enthallg(k+1)
   in enthallg 1
```

Datenabstraktion

Wird ein Typ in einer Schnittstelle (Signatur) nur deklariert, aber nicht definiert, so kann er von außen nur über die Operationen der Schnittstelle verarbeitet werden. Folgerungen:

- Die Implementierung des Typs kann später durch eine andere ersetzt werden. Beispiel: `Stapel0` ersetzt durch `Stapel1`.
- *Invarianten* des Typs können garantiert werden: `Stapel1.t` enthält nur solche Paare `(s,l)`, bei denen `List.length s = l`.

Ein Typ, dessen Definition von außen nicht sichtbar ist, heißt *abstrakter Datentyp*. Die Verwendung solcher heißt *Datenabstraktion*.

Separate Compilation

Sei `datei.ml` eine Datei mit ML-Definitionen.

Mit dem Befehl

```
ocamlc -c datei.ml
```

wird eine Datei `datei.cmo` erzeugt, die mit `#load "datei.ml";;` geladen werden kann.

Der Effekt ist derselbe wie wenn

```
module Datei = struct
```

Inhalt von `datei.ml`

```
end
```

eingegeben worden wäre. Es wird also ein Modul des Namens `Datei` erzeugt, dessen Komponenten die von `datei.ml` sind.

Beispiel

Datei `abc.ml` enthalte

```
let a = 0
let b = 1
let c = 2
```

Wir kompilieren mit `ocamlc -c abc.ml` und erhalten `abc.cmo`.
Folgende Sitzung ist möglich:

```
# #load "abc.cmo"
# Abc.a;;
- : int = 0
# Abc.b;;
- : int = 1
# Abc.c;;
- : int = 2
#
```

Schnittstellen

Sei `datei.mli` eine Datei, die ML Schnittstellenkomponenten, wie `val x : int` enthält.

Mit dem Befehl

```
ocamlc -c datei.mli
```

wird eine Datei `datei.cmi` erzeugt.

Wird nunmehr `ocamlc -c datei.ml` ausgeführt, so hat `#load "datei.cmo"`denselben Effekt wie

```
module type DATEI = sig
```

Inhalt von `datei.mli`

```
end
module Datei : DATEI = struct
```

Inhalt von `datei.ml`

```
end
```

Beispiel

Datei `abc.ml` wie zuvor.

Datei `abc.mli` enthalte nur `val a:int`.

Wir kompilieren mit `ocamlc -c abc.mli` gefolgt von `ocamlc -c abc.ml`. Folgende Sitzung ist möglich:

```
# #load "abc.cmo"
# Abc.a;;
- : int = 0
# Abc.b;;
  ^^^^^

Unbound value Abc.b
```

Ausführbare ML-Programme

Mit `ocamlc -o main datei.ml` wird eine ausführbare Datei `main` ("EXE-Datei") erzeugt. Der Effekt ist der des Auswertens aller Definitionen in `datei.ml`.

Natürlich macht das nur Sinn, wenn `datei.ml` Ausdrücke mit Seiteneffekten, wie die Grafikfunktionen oder `print_string` enthält:

Enthalte Datei `datei.ml` den folgenden Code

```
let _ = print_string "Hello, world!\n"
```

Compilieren mit

```
ocamlc -o main datei.ml
```

gefolgt von

```
./main
```

gibt `Hello, world!` aus.

Binden

Verwendet `datei.ml andere Dateien, z.B.: \verb`datei1.ml+ und `datei2.ml`, die man interaktiv mit `#load` hinzuladen würde, so muss man die entsprechenden `.cmo` Dateien bei der Kompilierung aufführen:

```
ocamlc -o main  datei1.cmo datei2.cmo datei.ml
```

oder auch

```
ocamlc -o main  datei1.cmo datei2.cmo datei.cmo
```

Die `.cmo` Dateien werden *gebunden* (*linked*).

Größeres Beispiel

Das Projekt der heutigen Übung besteht aus den Dateien

```
syntax.mli lex.mli norm.ml parse.mli graph.mli visualise.mli
lex.ml provided.cma visualise.ml main.ml
```

Dabei ist `provided.cma` ein Archiv, welches die Funktionen in `graph.mli` und `parse.mli` implementiert.

Sie müssen `norm.ml lex.ml visualise.ml` schreiben und mit `provided.cma` und `main.ml` zum ausführbaren Hauptprogramm zusammenbinden.

Aufgaben der Module

- `syntax.mli` Datentypen für arithmetische Ausdrücke und “Tokens”
- `parse.mli` Eine Funktion, die eine Tokenliste auf Ausrdücke abbildet. Eine Ausnahme.
- `lex.mli` Eine Funktion, die Strings auf Tokenlisten abbildet. Eine Ausnahme.
- `graph.mli` Eine Funktion, die Gegenstände zeichnet (Gegenstand = Strecke, Textbaustein). Eine Funktion, die die Größe eines Strings in Pixeln bestimmt.
- `visualise.mli` Eine Funktion, die einen Ausdruck auf eine Liste von Gegenständen abbildet (Baumdarstellung).
- `norm.mli` Eine Funktion, die Ausdrücke normalisiert.
- `main.ml` Ein Hauptprogramm, das einen String einliest, ggf. normalisiert und dann als Baum ausgibt.

Kompilieren

Soll alles kompiliert werden, so sind folgende Befehle auszuführen:

```
ocamlc -c syntax.mli
ocamlc -c lex.mli
ocamlc -c lex.ml
ocamlc -c visualise.mli
ocamlc -c visualise.ml
ocamlc -c norm.mli
ocamlc -c norm.ml
ocamlc -o main.ml
ocamlc -o main provided.cma visualise.cmo norm.cmo lex.cmo m
```

Ändert man eine Datei, so muss die entsprechende Datei neu kompiliert werden, ändert man eine Schnittstelle (`.mli`), so müssen alle Schnittstellen und Programme, die davon abhängen, neu kompiliert werden.

Makefiles

Das Unix-Werkzeug `make` erleichtert solche Kompilierungsaufgaben:

Man schreibt ein für alle Mal alle Aufgaben und deren Abhängigkeiten in eine Datei `Makefile`.

Danach genügt der Befehl `make` um nach einer Änderung die entsprechenden Neukompilierungen vorzunehmen.

Form des Makefiles

Ein Makefile enthält (neben anderen Komponenten) *Regeln* der Form.

```
datei1: datei2 datei3
        befehl1
        befehl2
```

Die Regel besagt folgendes:

- `datei1` hängt von `datei2`, `datei3` ab: haben `datei2`, `datei3` späteres Änderungsdatum als `datei1`, so ist `datei1` nicht mehr gültig und muss neu erstellt werden.
- Die Befehle[a] `befehl1` und `befehl2` erstellen `datei1` neu.

Wird `make datei1` aufgerufen, so wird geprüft, ob `datei2`, `datei3` aktuell sind (mithilfe entsprechender Regeln im Makefile). Wenn nein, dann werden zunächst diese "gemacht". Anschließend wird geprüft, ob `datei1` aktuell ist und ggf. mithilfe von `befehl1`, `befehl2` neu "gemacht".

[a] Achtung, vor jedem Befehl, wie `befehl1` und `befehl2` muss ein Tabulator stehen.

Ein Makefile für unser Projekt

```
main: visualise.cmo norm.cmo lex.cmo main.cmo
    ocamlc -o main provided.cma visualise.cmo norm.cmo lex.cmo main.cmo

visualise.cmi: visualise.mli
    ocamlc -c visualise.mli

visualise.cmo: syntax.cmi visualise.cmi
    ocamlc -c visualise.ml

lex.cmi: lex.mli
    ocamlc -c lex.mli

norm.cmi: norm.mli
    ocamlc -c norm.mli

lex.cmo: lex.cmi syntax.cmi
    ocamlc -c lex.ml

norm.cmo: norm.cmi syntax.cmi
    ocamlc -c norm.ml
```

Nachteil dieser Form

- Makefile enthält viele Redundanzen
- Wird die Modulstruktur geändert, so ändern sich ggf. die Abhängigkeiten und das Makefile muss geändert werden

Makefile mit impliziten Regeln und Abkürzungen

```
MAIN_OBJS = visualise.cmo norm.cmo lex.cmo main.cmo
.SUFFIXES: .ml .mli .cmo .cmi
.ml.cmo:
        ocamlc -c $<
.mli.cmi:
        ocamlc -c $<
main: $(MAIN_OBJS)
    ocamlc -o main $(MAIN_OBJS)
lex.cmi: syntax.cmi
norm.cmi: parse.cmi
norm.cmo: norm.cmi
visualise.cmi: graph.cmi syntax.cmi
lex.cmo: syntax.cmi lex.cmi
main.cmo: graph.cmi lex.cmi parse.cmi visualise.cmi
visualise.cmo: graph.cmi visualise.cmi
visualise.cmx: graph.cmx visualise.cmi
```

Erklärung

- Die implizite Regel

```
.ml.cmo:
        ocamlc -c $<
```

  besagt, wie eine `.cmo` Datei aus einer `.ml` Datei gemacht wird. `$<` steht für die `.ml` Datei, `$@` steht für die `.cmo` Datei (brauchen wir hier nicht).

- Die Abhängigkeiten sind weiter unten ohne Befehle angegeben.
- `$(MAIN_OBJS)` ist ein weiter oben definierte Abkürzung.
- Der Block mit den Abhängigkeiten kann mit `ocamldep *.mli *.ml` automatisch erzeugt werden.

“Professionelles” Makefile für das gesamte Projekt

```
OCAMLC=ocamlc
OCAMLOPT=ocamlopt
OCAMLDEP=ocamldep
OCAMLYACC=ocamlyacc
INCLUDES=
OCAMLFLAGS=$(INCLUDES)
PROVIDED_OBJS=parsen.cmo parse.cmo graph.cmo
MAIN_OBJS=provided.cma visualise.cmo lex.cmo main.cmo
DEPEND += parsen.ml

main: $(MAIN_OBJS)
        $(OCAMLC) -o main $(OCAMLFLAGS) $(MAIN_OBJS)

provided.cma: $(PROVIDED_OBJS)
        $(OCAMLC) -a -o provided.cma $(OCAMLFLAGS) graphics.cma $(PROVIDED_OBJS)

.SUFFIXES: .ml .mli .cmo .cmi .cmx .mly

.ml.cmo:
        $(OCAMLC) $(OCAMLFLAGS) -c $<
.mli.cmi:
        $(OCAMLC) $(OCAMLFLAGS) -c $<
.ml.cmx:
        $(OCAMLOPT) $(OCAMLOPTFLAGS) -c $<
.mly.ml:
        $(OCAMLYACC) $<
clean:
        rm -f main;rm -f *.cm[iox];rm parsen.ml;rm parsen.mli
depend: $(DEPEND)
        $(OCAMLDEP) $(INCLUDES) *.mli *.ml > .depend
include .depend
```

Was man über `make` wissen muss

- `make` ist eine Möglichkeit, größere Projekte zeitsparend und fehlerfrei zu kompilieren.
- Kooperiert mit Abhängigkeitsgenerator (`ocamldep`)
- Abhängigkeitsgeneratoren gibt es genauso für C, C++.
- `make` funktioniert unabhängig von der Programmiersprache und ist nicht auf Kompilierbefehle beschränkt.
- Für manche Programmiersprachen gibt es spezielle Kompilationsmanager (NJ-SML, Visual C++,etc), die `make` ersetzen, wenn nur in dieser Sprache gearbeitet wird. Für Projekte mit mehreren Sprachen braucht man dann doch wieder `make`.
- Man sollte wissen, dass `make` existiert; ggf. Dokumentation studieren.

Kapitel 6: Effiziente Algorithmen

Begriffsklärungen

Algorithmus ist `effizient`, wenn seine Ausführung möglichst wenig Aufwand verursacht.

Aufwand: Rechenzeit, Speicherplatz, Anzahl bestimmter Elementaroperationen.

Die *Komplexität* eines Algorithmus gibt den Aufwand als Funktion der Eingabe oder deren *Größe* an.

Bei Angabe der Komplexität als Funktion der Eingabegröße unterscheidet man

- *Best case:* Aufwand bei am günstigsten gewählter Eingabe zu fester Größe
- *Worst case:* Aufwand bei am ungünstigsten gewählter Eingabe zu fester Größe
- *Average case:* Erwartungswert des Aufwands bei Eingaben einer festen Größe bzgl. einer bestimmten Verteilung.

Beispiel: `insertel`

```
let rec insertel = function
   (a, []) -> [a]
 | (a, h::t) -> if a <= h then a::h::t else h :: insertel(a,t)
```

Anzahl der Auswertungsschritte von `insertel(a,l)` als Funktion von $n =$ `length(l)`.

- Best case: 4 (Element `a` wird am Anfang eingefügt)
- Worst case: $3n$ (Element `a` wird am Ende eingefügt)
- Average case: $1.5 \cdot n$ (Element `a` wird "in der Mitte" eingefügt)

Größenordnungen

Oft interessiert man sich nur für die *Größenordnung* der Komplexität: konstant, linear, quadratisch, exponentiell, ...

Man gibt diese mit der O-Notation an:

Sei $f : \mathbb{N} \to \mathbb{N}$:

$$O(f) = \{g \mid \exists N.\exists c > 0.\forall N \geq n.g(n) \leq c \cdot f(n)\}$$

Insbesondere: $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ existiert:$g \in O(f)$.

Notationen

Man verwendet oft folgende Abkürzungen:

- $f(n)$ statt f, z.B.: $2n^2 \in O(n^2)$ statt $\mathbf{function}(n)2n^2 \in O(\mathbf{function}(n)n^2)$,
- $g = O(f)$ statt $g \in O(f)$, z.B.: $n^2 = O(2^n)$,
- $f + O(g)$ statt $\{h \mid \exists u \in O(g).h = f + u\}$, z.B.: $\sum_{i=1}^{n} i^k = \frac{1}{k+1} n^{k+1} + O(n^k)$.

Beispiele

- $2n^2 = O(n^2)$
- $n^{1000} = O(2^n)$
- $\log(n) = O(n)$
- $1000/n = O(1)$
- Best case Laufzeit von `insertel` $= O(1)$
- Average case Laufzeit von `insertel` $= O(n)$
- Worst case Laufzeit von `insertel` $= O(n)$
- Worst case Laufzeit von `inssort` $= O(n^2)$

Entwicklung effizienter Programme

Wie kommt man zu effizienten Programmen?

- Effiziente Algorithmen
- Effiziente Datenstrukturen
- Einbeziehung des Auswerteprozesses, z.B.: günstige Rekursionsschemata, Vermeidung von “append”.

Der dritte Punkt lässt sich relativ schematisch umsetzen und wird mehr und mehr durch Fortschritte in der Compilertechnik automatisiert.

Die ersten beiden Punkte erfordern Kreativität, Fachwissen und Erfahrung.

Beispiel für effizienten Algorithmus

Wir haben gesehen, dass Sortieren durch Einfügen quadratische Komplexität hat.

Wir betrachten jetzt ein rekursives Verfahren, welches sowohl praktisch, als auch theoretisch (asymptotisch) bessere Laufzeit aufweist.

Sortieren durch Mischen

Um eine Liste der Länge n zu sortieren:

- Teile man die Liste in zwei gleichgrosse Hälften
- Sortiere jede Hälfte durch rekursiven Aufruf (bei Länge ≤ 1 ist natürlich nichts zu tun)
- Füge die beiden sortierten Hälften “im Reißverschlussverfahren” zusammen.

In OCAML

```
let rec split = function
    [] -> [],[]
  | [a] -> ([a],[])
  |  a::b::l -> let u,v = split l in
          a::u,b::v

let rec merge = function
    ([],l) -> l
  | (l,[]) -> l
  | ((a::t as x), (b::u as y)) ->
      if a <= b then a::merge(t,y) else b :: merge(x,u)

let rec mergesort = function
      [] -> []
  |   [a] -> [a]
  |   l -> let u,v = split l in
```

In OCAML

```
let u1 = mergesort u in
let v1 = mergesort v in
merge(u1, v1)
```

Empirische Laufzeitbestimmung

Die Funktion

```
Unix.gettimeofday : unit -> float
```

liefert die Zeit in Sekunden, die zwischen 1.1.1970 und dem Aufruf verstrichen ist.

Die Funktion

```
Random.int : int -> int
```

liefert einen “zufälligen” Integer im Bereich $0 \ldots n$.

Profiler

```
let profile f s =
  let before = Unix.gettimeofday() in
  let _ = f() in
  let after = Unix.gettimeofday() in
  print_string (s ^ ": " ^ string_of_float (after-.before) ^
                          " Sekunden.\n")
```

Der Aufruf `profile f s` wertet `f` aus und gibt die dafür verbrauchte Zeit zusammen mit dem String `s` aus.

Damit können wir die Laufzeit von `inssort` und `mergesort` abhängig von der Eingabe empirisch bestimmen.

Effizienz der Rekursion

- Wir wollen die Effizienz rekursiver Funktionsdefinitionen studieren.
- Eine bestimmte Art der Rekursion (“*repetitive Rekursion*”) ist besonders platzeffizient.
- Es ist manchmal möglich, rekursive Funktionen in repetitiv rekursive Form zu bringen
- Es gibt Formen der Rekursion, für die solch eine Transformation nur unter Aufbietung zusätzlichen Hilfsspeichers in derselben Größe wie die resultierende Ersparnis möglich ist.

Rekursion im Rechner

Die Auswertung rekursiver Funktionen erfolgt im Rechner im Prinzip so wie in der operationellen Semantik.

In Abwesenheit lokaler und höherstufiger Funktionen können wir näherungsweise die sukzessive Ersetzung von Funktionen durch ihre Definition verwenden ("*Substitutionsmodell*")

Auswertung der Paritätsfunktion

$$\mathit{gerade}(n) = \textbf{if } n = 0 \textbf{ then true else not}(\mathit{gerade}(n-1))$$

$$\mathit{gerade}(4) =$$
$$\textbf{not}(\mathit{gerade}(3)) =$$
$$\textbf{not}(\textbf{not}(\mathit{gerade}(2))) =$$
$$\textbf{not}(\textbf{not}(\textbf{not}(\mathit{gerade}(1)))) =$$
$$\textbf{not}(\textbf{not}(\textbf{not}(\textbf{not}(\mathit{gerade}(0))))) = \textbf{true}$$

Platzverbrauch von $\mathit{gerade}(n)$ ist $O(n)$ aber nicht $O(1)$.

Repetitiv rekursive Version der Parität

$$\mathit{gerade2}(n) = \mathit{gerade2\_aux}(n, \textbf{true})$$

$$\mathit{gerade2\_aux}(n, a) = \textbf{if } n = 0 \textbf{ then } a \textbf{ else } \mathit{gerade2\_aux}(n-1, \textbf{not}(a))$$

$$\mathit{gerade2}(4) = \mathit{gerade2\_aux}(4, \textbf{true}) =$$

$$\mathit{gerade2\_aux}(3, \textbf{not}(\textbf{true})) =$$

$$\mathit{gerade2\_aux}(3, \textbf{false}) =$$

$$\mathit{gerade2\_aux}(2, \textbf{true}) =$$

$$\mathit{gerade2\_aux}(1, \textbf{false}) =$$

$$\mathit{gerade2\_aux}(0, \textbf{true}) = \textbf{true}$$

Platzverbrauch von $\mathit{gerade2}(n) = O(1)$.

Zusammenhang der beiden Funktionen

$$\mathit{gerade2\_aux}(n, a) = \textbf{if } a \textbf{ then } \mathit{gerade}(n) \textbf{ else not}(\mathit{gerade}(n))$$

Beweis durch Satz von der partiellen Korrektheit + Abstiegsfunktion.

Beispiel: Fakultät

$$\mathit{fakt}(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \cdot \mathit{fakt}(n-1)$$

$$\begin{aligned}
&\mathit{fakt}(4) = \\
&4 \cdot \mathit{fakt}(3)) = \\
&4 \cdot (3 \cdot \mathit{fakt}(2)) = \\
&4 \cdot (3 \cdot (2 \cdot \mathit{fakt}(1))) = \\
&4 \cdot (3 \cdot (2 \cdot (1 \cdot \mathit{fakt}(0)))) = \\
&4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) = 24
\end{aligned}$$

Der “Computer” kann nicht wissen, dass man die Multiplikationen auch umklammern und vereinfachen kann: die gesamte Multiplikationsaufgabe bleibt stehen, bis endlich $\mathit{fakt}(0)$ ausgewertet wird.

$\rightarrow$ Platzverbrauch von $\mathit{fakt}(n)$ ist $O(n)$ aber nicht $O(1)$.

Repetitiv rekursive Version der Fakultät

$$\mathit{fakt2}(n) = \mathit{fakt2\_aux}(n, 1)$$

$$\mathit{fakt2\_aux}(n, a) = \textbf{if } n = 0 \textbf{ then } a \textbf{ else } \mathit{fakt2\_aux}(n - 1, a \cdot n))$$

$$\mathit{fakt2}(4) = \mathit{fakt2\_aux}(4, 1) =$$

$$\mathit{fakt2\_aux}(3, 1 \cdot 4) =$$

$$\mathit{fakt2\_aux}(3, 4) =$$

$$\mathit{fakt2\_aux}(2, 12) =$$

$$\mathit{fakt2\_aux}(1, 24) =$$

$$\mathit{fakt2\_aux}(0, 24) = 24$$

Platzverbrauch hier $O(1)$.

Zusammenhang der beiden Funktionen

$$\mathit{fakt\_aux}(n, a) = a \cdot \mathit{fakt}(n)$$

Beweis durch Satz von der partiellen Korrektheit + Abstiegsfunktion.

Repetitive Rekursion: Definition

Eine rekursive Definition

$$f(x) = \Phi(f, x)$$

heißt *repetitiv rekursiv* (auch *endständig rekursiv, iterativ rekursiv, endrekursiv, tail recursive*), wenn im Rumpf Φ höchstens ein rekursiver Aufruf von f getätigt wird und dessen Ergebnis dann bereits den Wert von $\Phi(f, x)$ liefert.

Merke: Bei der Auswertung repetitiv rekursiver Funktionen entsteht kein zusätzlicher Platzbedarf für die Verwaltung der Rekursion.

Lineare Rekursion

Eine rekursive Definition heißt *linear*, wenn im Rumpf der Definition höchstens ein rekursiver Aufruf anfällt.

Sehr häufig lassen sich linear rekursive Definitionen durch *Einbettung* in repetitiv rekursive Form bringen.

Dies resultiert dann in beträchtlicher Platz- und Zeitersparnis.

Beispiel: Fakultät, Parität.

Repetitive Rekursion und Iteration

Eine repetitiv rekursive Funktionsdefinition der Form

$$f(x) = \textbf{if } p(x) \textbf{ then } a(x) \textbf{ else } f(b(x))$$

(p beliebiges Prädikat, a, b beliebige Funktionen) kann man auch "imperativ" wie folgt auswerten:

Schreibe die Eingabe in die Speicherstelle x

Solange p auf den Inhalt von x nicht zutrifft, wiederhole folgendes:

Bestimme $y := b(\text{Inhalt von } x)$

Ersetze den Inhalt von x durch y

Gib als Ergebnis $a(\text{Inhalt von } x)$ zurück.

Dasselbe in C

Imperative Programmiersprachen unterstützen diesen Stil, z.B. “C”:

```
f(x) {
   while(!p(x)) {
      x = b(x);
   }
   return a(x);
}
```

Vor- und Nachteile des imperativen Stils

- Imperativer Stil ist zunächst verständlicher
- Funktionale Definition lassen sich durch Gleichungen spezifizieren und verifizieren
- Im imperativen Stil muss man über Zustand der Variablen zu bestimmten Zeitpunkten sprechen, was komplizierter ist.
- In der Effizienz unterscheidet sich die imperative Version nicht von der repetitiv rekursiven Version.

Nichtlineare Rekursion

Sei $f(x) = \Phi(f, x)$ eine rekursive Definition mit Abstiegsfunktion m.

Finden im Rumpf Φ bis zu a rekursive Aufrufe statt (Etwa Fibonacci, Mergesort, Hanoi: $a = 2$), so erzeugt die vollständige Auswertung von $f(x)$ bis zu $a^{m(x)}$ rekursive Aufrufe.

Fibonacci, Hanoi: $m(x) = x \rightarrow$ bis zu 2^x rekursive Aufrufe

Mergesort: $m(x) = O(\log(\mathit{length}(x))) \rightarrow O(\mathit{length}(x)$ rekursive Aufrufe.

Fazit: Bei nichtlinearer Rekursion Laufzeit im Auge behalten.

Fibonacci als repetitive Rekursion

Manchmal lassen sich auch nichtlineare Rekursionen als repetitive R. umschreiben:

$$\mathit{fib}(n) = \textbf{if } n \leq 1 \textbf{ then } 1 \textbf{ else } \mathit{fib}(n-1) + \mathit{fib}(n-2)$$

Definiere

$$\mathit{fib\_aux}(n, a, b) = \textbf{if } n = 0 \textbf{ then } a \textbf{ else } \mathit{fib\_aux}(n-1, b, a+b)$$

Es gilt $\mathit{fib\_aux}(n, \mathit{fib}(k), \mathit{fib}(k+1)) = \mathit{fib}(n+k)$

also $\mathit{fib}(n) = \mathit{fib\_aux}(n, 1, 1)$.

Dies ist ein Spezialfall der *dynamischen Programmierung*, ein allgemeines Optimierungsschema für nichtlineare Rekursionen $\rightarrow$ 'VL "Effiziente Algorithmen".

Echte nichtlineare Rekursion

Bei Mergesort, Linearisierung von Bäumen, Hanoi, lässt sich die nichtlineare Rekursion nur “künstlich” vermeiden, indem man Hilfsdatenstrukturen einführt, z.B. Stapel, die genauso viel Platz verbrauchen, wie die Verwaltung der ursprünglichen Rekursion.

Hier ist also keine echte Verbesserung möglich. In diesen Fällen ist die nichtlineare Rekursion eine vernünftige Lösung.

Grundsätzlich ist im Frühstadium der Softwareentwicklung eine klare rekursive Lösung vorzuziehen.

Bei modularer Planung kann diese später ggf. durch eine effizientere Lösung ersetzt werden.

Erinnerung

Eine rekursive Definition kann auch deshalb ineffizient sein, weil jeder einzelne Verarbeitungsschritt aufwendig ist, nicht weil die Zahl der rekursiven Aufrufe selbst zu groß wäre.

Beispiele: Sortieren durch Einfügen, ineffiziente Version des Umdrehens einer Liste mit “append” (Folie 221).

Hier haben wir jeweils $O(n)$ rekursive Aufrufe, deren jeder aber Aufwand $O(n)$ verursacht, also Gesamtaufwand $O(n^2)$.

Effiziente Datenstrukturen

Oft liegt der Schlüssel zu einem effizienten Alorithmus in der Wahl de geeigneten Datenstruktur.

Als Beispiel betrachten wir nochmal die Binären Suchbäume (BST), Folie 245.

Wir haben gesehen, dass das Suchen eines Elementes im BST t Zeit $O(\text{Höhe}(t))$ erfordert.

Einfügen und Löschen in BST

Man schreibe rekursive OCAML-Funktionen

```
insert : 'a -> 'a bintree -> 'a bintree
delete : 'a -> 'a bintree -> 'a bintree
```

die ein Element in einen BST einfügen, bzw. entfernen.

Einfügen

```
let rec insert x t = match t with
  Empty -> Build(x,Empty,Empty)
| Build(a,l,r) -> if x<=a then Build(a,insert x l,r)
                          else Build(a,l,insert x r)
```

Entfernen

```
let rec delete x t = match t with
  Build(a,l,r) -> if x<a then Build(a,delete x l,r)
                  else if x>a then Build(a,l,delete x r)
                  else if l=Empty then r
                  else if r=Empty then l
                  else let m,l' = remove_max l in
                  Build(m,l',r)
```

Hilfsfunktion `remove_max`

```
let rec remove_max t = match t with
  Build(a,l,Empty) -> a,l
| Build(a,l,r) -> let m,r' = remove_max r in
                  (m,Build(a,l,r'))
```

Knotenanzahl und Höhe

Sei n die Knotenanzahl eines Baumes t und h seine Höhe.

Im besten Fall gilt $n = 2^h - 1$, also $h = O(\log n)$.

Im schlechtesten Fall gilt $n = h$, also $h = O(n)$.

Suchen, Einfügen, Löschen haben Komplexität $O(h)$, also $O(\log n)$ bzw. $O(n)$ je nach Art von t.

Zwei extreme Beispiele

```
let t1 = insert 6 (insert 5 (insert 4 (insert 3
         (insert 2 (insert 1 (insert 0 Empty))))))
let t2= insert 6 (insert 4 (insert 1 (insert 0
         (insert 5(insert 1 (insert 3 Empty))))))
```

`t1` hat maximale Höhe (6) und `t2` hat minimale Höhe (3)

Eine Menge von Bäumen $\mathcal{B}$ heißt *balanciert*, wenn

$$\max\{\text{Höhe}(t) \mid \texttt{knotanz}(t) \leq n, t \in \mathcal{B}\} = O(\log(n))$$

Für balancierte BST mit n Knoten erfordern Einfügen, Löschen, Suchen Zeit $O(\log(n))$.

Rotationen

Man kann Bäume balanciert halten durch systematisches Anwenden der folgenden beiden Rotationen auf Teilbäume Verlauf des Einfügens und Löschens.

```
(* rotate_left : 'a bintree -> 'a bintree *)
let rotate_left = function
    Build(a,l,Build(b,m,r)) -> Build(b,Build(a,l,m),r)
(* rotate_right : 'a bintree -> 'a bintree *)
let rotate_right = function
    Build(a,Build(b,l,m),r) -> Build(b,l,Build(a,m,r))
```

Übung: man begründe, dass die Rotationen die BST-Eigenschaft erhalten.

Um effizient feststellen zu können, welche Rotation angewendet werden soll, muss man in den Bäumen zusätzliche Information speichern, z.B.: Differenz der Knotenanzahl zwischen linkem und rechten Teilbäumen. Näheres: Info II.

Kapitel 7: Semantik und Fixpunkttheorie

Operationelle Semantik von Listen und Bäumen

Sei $\mathcal{L}$ eine *unendliche* Menge von *Adressen* (im Speicher). Z.B.: $\mathcal{L} = \mathbb{N}$.

Die *Werte* i.S.d. operationellen Semantik (vgl. Folie 157) werden erweitert um

- Jede Adresse $\ell \in \mathcal{L}$ ist ein Wert.

Halde

Eine *Halde* (engl.: *heap*)[a] ist eine endliche Funktion von Adressen ($\mathcal{L}$) auf Werte.

Ist h eine Halde, so bezeichnet $\mathrm{dom}(h)$ die (endliche) Teilmenge von $\mathcal{L}$, für die h definiert ist.

$h[\ell \mapsto v]$ ist wie üblich durch

$$h[\ell \mapsto v](\ell') = \begin{cases} v, \text{ wenn } \ell = \ell' \\ h(\ell'), \text{ sonst} \end{cases}$$

definiert. Beachte: $\mathrm{dom}(h[\ell \mapsto v] = \mathrm{dom}(h) \cup \{\ell\}$.

[a] Halde (heap) ist ein "Teekesselchen": Man verwendet den Begriff auch für eine bestimmte baumartige Datenstruktur; das hat mit der Verwendung hier *nichts* zu tun.

Operationelle Semantik mit Halde

Zur Modellierung dynamischer Datenstrukturen erweitern wir das *Format* der operationellen Semantik um Halden wie folgt:

$$U, h, e \to w, h'$$

bedeutet, dass in der Umgebung U und Halde h der Ausdruck e den Wert w hat. Zudem verändert sich durch die Auswertung die Halde zu h'.

Wie zuvor wird dieses *Urteil* durch Auswerteregeln definiert:

Auswerteregeln

- ist c eine Konstante, so gilt $U, h, c \to c, h$. (Halde verändert sich nicht.)
- ist x ein Bezeichner und $<x, w> \in U$, so gilt $U, x, h \to w, h$.
- für Funktionsausdrücke gilt $U, h,$ `fun` x`->`$e \to ($`fun` x`->`$e, U'), h$, wobei U' die Einschränkung von U auf die freien Variablen von e ist.

Auswerteregeln

- ist *op* ein Infixoperator aber nicht `||`, `&&`, welcher eine Basisfunktion $\oplus$ bezeichnet so gilt folgendes: $U, h, x \mathit{op} y \to U(x) \oplus U(y), h$.

  Anwendung von *op* auf geschachtelte Ausdrücke kann man durch Einfügen von `lets` auf diesen Fall reduzieren.

Einstellige Basisfunktionen werden analog behandelt.

Bindung

- Um w, h' mit $U, h,$ `let` x`=`e_1 `in` $e_2 \to w, h'$ zu bestimmen, muss zunächst w_1, h_1 mit $U, h, e_1 \to w_1, h_1$ ermittelt werden. Sodann sind w_2, h_2 mit $U + \{<x, w_1>\}, h_1, e_2 \to w_2, h_2$ zu bestimmen. Es ist dann $U, h,$ `let` x`=`e_1 `in` $e_2 \to w_2, h_2$.

Konstruktoren

- $U, h, [] \to \ell, h[\ell \mapsto (0,0,0)]$ wobei $\ell \notin \mathrm{dom}(h)$.
- $U, h, x::y \to \ell, h[\ell \mapsto (1, U(x), U(y))$, wobei $\ell \notin \mathrm{dom}(h)$

Konstruktoren anderer (rekursiver) Varianten werden analog behandelt.

Mustervergleich (einfachster Fall)

Sei $e = \texttt{match}\ \ x\ \ \texttt{with []->}e_1\texttt{|}y\texttt{::}z\texttt{->}e_2$.

Um h', w mit $U, h, e \to w, h'$ zu bestimmen, gehe man wie folgt vor:

- Falls $U(z) = \ell$ und $h(\ell) = (0, 0, 0)$, so bestimme man w_1, h_1 mit $U, h, e_1 \to w_1, h_1$. Es ist dann $U, h, e \to w_1, h_1$.
- Falls $U(z) = \ell$ und $h(\ell) = (1, u, v)$, so bestimme man w_2, h_2 mit $U[y{\mapsto}u, z{\mapsto}v], h, e_2 \to w_2, h_2$. Es ist dann $U, h, e \to w_2, h_2$.

Beispiel

Sei $e_1 =$ `7::3::10::[]`.

Es gilt $\emptyset, h, e_1 \to \ell_4, h_1$, wobei

$$h_1 = h[\ell_4 \mapsto (1, 7, \ell_3), \ell_3 \mapsto (1, 3, \ell_2), \ell_2 \mapsto (1, 10, \ell_1), \ell_1 \mapsto (0, 0, 0)]$$

und $\ell_1, \ell_2, \ell_3, \ell_4$ paarw. verschieden und nicht in $\mathrm{dom}(h)$.

Sei $e_2 =$ `9::123::[]`. Es gilt $\emptyset, h_1, e_1 \to \ell_7, h_2$, wobei

$$h_2 = h_1[\ell_7 \mapsto (1, 9, \ell_6), \ell_6 \mapsto (1, 123, \ell_5), \ell_5 \mapsto (0, 0, 0)$$

und ℓ_5, ℓ_6, ℓ_7 paarw. verschieden und nicht in
$\mathrm{dom}(h_1) = \mathrm{dom}(h) \cup \{\ell_1, \ell_2, \ell_3\}$.

Beispiel

Es sei `append` wie üblich definiert:

```
let rec append l1 l2 = match l1 with []->l2|y::z->y::append z l2
```

Es sei $U = \emptyset[\texttt{l1} \mapsto \ell_3, \texttt{l2} \mapsto \ell_7]$.

Es gilt $U, h \to$ `append l1 l2` $\to \ell_{10}, h_3$, wobei

$$h_3 = h_2[\ell_{10} \mapsto (1, 7, \ell_9), \ell_9 \mapsto (1, 3, \ell_8), \ell_8 \mapsto (1, 10, \ell_7)]$$

Beispiel

Sei $e =$

```
let l1 = 7::3::10::[] in
let l2 = 9::123::[] in
append l1 l2
```

Es gilt

$$\emptyset, h, e \to \ell_{10} h_3$$

Die Adressen $\ell_1, \ell_2, \ell_3, \ell_4$ sind in $\mathrm{dom}(h_3)$, aber ihre Inhalte sind nicht mehr zugreifbar. Es wäre wünschenswert, zu setzen

$$\emptyset, h, e \to \ell_{10}, h_4$$

wobei $\mathrm{dom}(h_4) = \mathrm{dom}(h_3) \setminus \{\ell_1, \ell_2, \ell_3, \ell_4\}$ und $h_4(\ell) = h_3(\ell)$ für alle $\ell \in \mathrm{dom}(h_4)$.

Garbage collection

In der operationellen Semantik wird die Halde immer größer und enthält immer mehr unerreichbare Adressen.

In praktischen Implementierungen werden unerreichbare Adressen regelmäßig aus der Halde entfernt.

Diesen Prozess bezeichnet man als *garbage collection* (Müllabfuhr).

Dazu werden ausgehend von der aktuellen Umgebung alle erreichbaren Adressen markiert und anschließend alle nichtmarkierten Adressen gelöscht.

Konkrete Implementierung und Formalisierung dieses Prozesses: Spezialvorlesung im Hauptstudium.

Garbage collection gibt es in funktionalen und manchen objektorientierten Programmiersprachen, z.B.: Java.

Denotationelle Semantik, genauer

Die denotationelle Semantik wie in Kap 3.4, weist noch einige Ungenauigkeiten auf:

- Was sind überhaupt “Werte”?
- Was bedeutet “. . . die durch . . . definierte rekursive Funktioni. . . ”?
- Warum “gibt” es rekursiv definierte Funktionen?
- Wie harmoniert Rekursion mit Funktionen höherer Ordnung?

Beispiel

```
let rec fix phi = fun x -> phi (fix phi) x
(* fix :  ((’a -> ’b) -> ’a -> ’b) -> ’a -> ’b*)
let fakt = fix (fun f x -> if x=0 then 1 else x*f(x-1))
(* fakt : int -> int *)
```

Hier liefert `fakt 3` das Ergebnis 6, wie erwartet.

Definiert man aber

```
let rec fix phi = phi (fix phi)
```

so terminiert schon

```
let fakt = fix (fun f x -> if x=0 then 1 else x*f(x-1))
```

nicht.

Durch die denotationelle Semantik wird dies nicht zufriedenstellend erklärt.

Partielle Ordnung

Eine binäre Relation $\sqsubseteq$ auf einer Menge A heißt *partielle Ordnung*, wenn für alle $x, y, z \in A$ folgendes gilt:

- $x \sqsubseteq x$ (Reflexivität)
- $x \sqsubseteq y, y \sqsubseteq x \rightarrow x = y$ (Antisymmetrie)
- $x \sqsubseteq y, y \sqsubseteq z \rightarrow x \sqsubseteq z$ (Transitivität)

Beispiele

- $A = \mathbb{N}, x \sqsubseteq y \leftrightarrow x \leq y$
- $A = \mathbb{N}, x \sqsubseteq y \leftrightarrow x \geq y$
- A beliebig, $x \sqsubseteq y \leftrightarrow x = y$ (dies nennt man die *diskrete*[a] *Ordnung*)
- $A = \mathfrak{P}(X), x \sqsubseteq y \leftrightarrow x \subseteq y$
- $A = X \cup \{\bot\}, x \sqsubseteq y \leftrightarrow x = \bot \vee x = y$

Es ist üblich, die partielle Ordnung durch die *Trägermenge* zu bezeichnen und immer das Symbol $\sqsubseteq$ für die Relation zu verwenden.

[a] Im Englischen gibt es *discrete* und *discreet*. Hier meinen wir ersteres.

Begriffe

Seien X, Y partielle Ordnungen. Alle betrachteten Funktionen seien total.

- Eine Folge $(x_i)_{i\in\mathbb{N}}$ von Elementen $x_i \in X$ heißt *ω-Kette*, kurz *Kette*, wenn $x_i \sqsubseteq x_{i+1}$ für alle i.
- $f : X \to Y$ heißt *monoton*, wenn $x \sqsubseteq x'$ impliziert $f(x) \sqsubseteq f(x')$.
- Sei $U \subseteq X$. Ein Element $x \in X$ heißt *kleinste obere Schranke* oder *Supremum* von U, wenn gilt
  - $u \sqsubseteq x$ für alle $u \in U$,
  - Falls $u \sqsubseteq y$ für alle $u \in U$, so gilt $x \sqsubseteq y$
- $x \in X$ ist *kleinstes Element*, wenn $x \sqsubseteq y$ für alle $y \in X$.

Sätze

- Sind x, x' beides kleinste Elemente, so folgt $x = x'$. Man bezeichnet das kleinste element, wenn es existiert, mit dem Symbol $\bot$.
- Sind x, x' beides kleinste obere Schranken von $U \subseteq X$, so folgt $x = x'$. Man notiert das Supremum von U, wenn es existiert, mit $\sup U$ oder $\bigsqcup_{x \in U} x$.

Vollständigkeit

Eine partielle Ordnung X heißt *vollständig* (bzgl. ω-Ketten), wenn für jede ω-Kette $(x_i)_i$ das Supremum von $\{x_i \mid i \in \mathbb{N}$ existiert.

Man schreibt $\bigsqcup_i x_i$ für dieses Supremum.

Eine solche Ordnung heißt auf E. ω-complete partial order, kurz *ωcpo* oder noch kürzer *cpo*.

Bemerkung: Man kann den Begriff der Vollständigkeit auf *gerichtete Mengen* verallgemeinern, siehe [Kröger]. Wir brauchen das hier nicht.

Beispiele

Alle auf Folie 337 gegebenen Beispiele sind cpos.

Seien X, Y cpos. Die folgenden Konstruktionen liefern cpos.

- $Y_\perp$ definiert durch $Y_\perp = Y \cup \{\perp\}$ und $y \sqsubseteq y'$, falls $y = \perp$ oder $y \sqsubseteq y'$ in Y
- $[X \to Y] = \{f : X \to X \mid f \text{ stetig}\}$ mit $f \sqsubseteq g$, wenn $f(x) \sqsubseteq g(x)$ für alle $x \in X$.
-
- $X \times Y$ mit $(x, y) \sqsubseteq (x', y')$, wenn $x \sqsubseteq x'$ und $y \sqsubseteq y'$.
- X^* mit $[x_1; \ldots; x_m] \sqsubseteq [y_1; \ldots y_n]$, wenn $m = n$ und $x_i \sqsubseteq y_i$ für $i = 1, \ldots, m$.

Stetige Funktionen

Die Projektionen $X \times Y \to X$, $X \times Y \to Y$ sind stetig.

Sind $f : Z \to X$ und $g : Z \to Y$ stetig, so auch $h(z) = (f(z), g(z))$.

Die Applikation $(X \to Y) \times X \longrightarrow$ ist stetig.

Ist $f : X \times Y \to Z$ stetig, so ist $h : X \longrightarrow Y \to Z$ gegeben durch $h(x) = \mathbf{function}(y) f(x, y)$ wohldefiniert (liefert also stetige Funktionen) und außerdem selbst stetig.

Fixpunkte

Eine monotone Funktion $f : X \to Y$ heißt *stetig* (bzgl. ω-Ketten), wenn für jede Kette x_i gilt:

$$f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$$

Man beachte, dass $f(x_i)$ wegen der Monotonie eine Kette bildet.

Fixpunkte

Sei $f : X \to X$ stetige Funktion. Ein Element $x \in X$ heißt *kleinster Fixpunkt* von f, wenn $f(x) = x$ und wenn immer $f(y) = y$, so folgt $x \leq y$.

Bemerkung: der kleinste Fixpunkt (falls er existiert) ist eindeutig bestimmt.

Fixpunktsatz von Kleene: Hat X ein kleinstes Element $\bot$, so hat jede stetige Funktion $f : X \to X$ einen kleinsten Fixpunkt, nämlich $x := \bigsqcup_i f^i(\bot)$.

Für diesen gilt zusätzlich folgendes: gilt $f(y) \sqsubseteq y$ für ein $y \in X$, so folgt $x \sqsubseteq y$.

Man schreibt $\mathrm{fix}(f)$ für den kleinsten Fixpunkt von f.

Satz: Die Funktion $\mathrm{fix} : (X \to X) \longrightarrow X$ ist selbst stetig.

Denotationelle Semantik mit cpos

Wir interpretieren nur wohltypisierte OCAML-Phrasen.

Zur Vereinfachung betrachten wir nur monomorphe Typen (keine Typvariablen).

Jedem Typen τ ordnen wir eine cpo $W(\tau)$ zu gemäß folgender Setzung:

$$W(\texttt{int}) = \{\text{Integer Konstanten}\} \text{ mit diskreter Ordnung}$$

$$W(\text{andere Basistypen}) = \text{analog}$$

$$W(\tau_1 \ \texttt{->} \ \tau_2) = W(\tau_1) \to W(\tau_2)_\perp$$

$$W(\tau_1 \ \texttt{*} \ \tau_2) = W(\tau_1) \times W(\tau_2)$$

$$W(\tau \ \texttt{list}) = W(\tau)^*$$

Andere Typformer werden ähnlich behandelt.

CPO der Umgebungen

Sei Γ eine Typzuweisung (endliche Funktion von Bezeichnern auf Typen).

Eine Umgebung U für Γ weist jedem Bezeichner in $\mathrm{dom}(\Gamma)$ ein Element $U(x) \in W(\Gamma(x))$ zu.

Die Umgebungen für Γ bilden eine cpo $\mathrm{Env}(\Gamma)$ mit $U \sqsubseteq U'$, wenn $U(x) \sqsubseteq U(x)$ für alle $x \in \mathrm{dom}(\Gamma)$.

Semantik von OCAML Phrasen

Zu jeder Programmphrase $\Gamma \triangleright t : \tau$ definieren wir eine stetige Funktion

$$W(t) : \mathrm{Env}(\Gamma) \to W(\tau)_\perp$$

durch Rekursion über den Aufbau von e wie folgt.

Wir schreiben wie üblich $W^U(t)$ statt $W(t)(U)$.

Auswerteregeln

- Ist c eine Konstante, so ist $W^U(c) = c$,
- Ist x ein Bezeichner, so ist $W^U(x) = w$, falls $<x, w> \in U$. Beachte: es muss x in $\mathrm{dom}(U)$ sein.
- Ist *op* ein Infixoperator aber nicht `||`, `&&`, welcher eine Basisfunktion $\oplus$ bezeichnet.

  Sind $W^U(t_1)$ und $W^U(t_2)$ beide ungleich $\perp$, so ist $W^U(t_1 \; op \; t_2) = W^U(t_1) \oplus W^U(t_2)$. Ansonsten ist das Ergebnis $\perp$.

  Einstellige Basisfunktionen wie `not` und `-` sind analog.

Auswertung von Funktionstermen

- Sei t eine Funktionsanwendung der Form $t_1\ t_2$. Es sei $W^U(t_1)$ die Funktion f und $W^U(t_2) = w$. Ist auch nur eines der beiden gleich $\bot$, so ist $W^U(t) = \bot$.

  Ansonsten ist $W^U(t) = f(w)$. Dies kann trotzdem noch $= \bot$ sein, da ja $W(\tau_1\texttt{->}\tau_2) = W(\tau_1 \to W(\tau_2)_\bot$.

- Sei $t = \texttt{function}\ \ x\ \ \texttt{->}\ \ t'$ und $\Gamma \rhd t : \tau_1\texttt{->}\tau_2$.

  Es ist $W^U(t)$ diejenige Funktion, die $w \in W(\tau_1)$ auf $W^{U+\{<x,w>\}}(t') \in W(\tau_2)_\bot$ abbildet.

- Die alternativen Notationen für Funktionsdefinitionen (`fun`, `let`) haben analoge Bedeutung.

Auswertung von `if` und `let`

- Sei t ein bedingter Term der Gestalt `if` t_1 `then` t_2 `else` t_3. Ist $W^U(t_1) = \mathit{true}$, so ist $W^U(t) = W^U(t_2)$. Beachte: $W^U(t_3)$ kann dann $= \bot$ sein.

  Ist $W^U(t_1) = \mathit{false}$, so ist $W^U(t) = W^U(t_3)$. Beachte: $W^U(t_2)$ kann dann $= \bot$ sein.

  Ist dagegen $W^U(t_1) = \bot$, so auch $W^U(t)$.

- Sei t von der Form `let` x `=` t_1 `in` t_2.

  Sei $W^U(t_1) = w \neq \bot$. Dann ist $W^U(t) = W^{U+\{<x,w>\}}(t_2)$.
  Ansonsten ist $W^U(t) = \bot$.

Tupel und Boole'sche Operatoren

- Sei $t = (t_1, \ t_2)$. Seien weiter
  $W^U(t_1) = w_1 \neq \perp W^U(t_2) = w_2 \neq \perp$. Dann ist $W^U(t) = (w_1, w_2)$.
  Ansonsten ist $W^U(t) = \perp$.
- $W^U(t_1 \ \texttt{||} \ t_2) = W^U(\texttt{if} \ t_1 \ \texttt{then true else} \ t_2)$
- $W^U(t_1 \ \texttt{\&\&} \ t_2) = W^U(\texttt{if} \ t_1 \ \texttt{then} \ t_2 \ \texttt{else false})$.

Semantik von `let rec`

Eine rekursive `let`-Bindung

$$\texttt{let rec } f\ x \texttt{ = } t$$

bindet f an die Funktion

$$\mathrm{fix}(\mathbf{function}F.\mathbf{function}x.W^{U+\{<x,w>,<f,F>\}}(t))$$

Das gilt analog für Phrasen mit `let rec ...in` und für Funktionen mit mehreren Argumenten.

Man beachte, dass $W(\tau_1 \texttt{->} \tau_2)$ ein kleinstes Element besitzt, nämlich $\mathbf{function}\ w.\bot$. Somit existiert dieser kleinste Fixpunkt.

Beispiel: Fakultät

```
let rec fakt n = if n = 0 then 1 else n * fakt(n-1)
```

Sei $F = W^{\emptyset}(\texttt{fakt})$.

Es ist $F = \mathrm{fix}(\Phi)$, wobei

$$\Phi(f) = \textbf{function}\ w.W^{U+\{<\texttt{n},w>,<\texttt{fakt},f>\}}(\texttt{if n = 0 then 1 else n * fakt(n-1)})$$

Es gilt

$$\Phi(f) = \textbf{function}w.\textbf{if}\ w = 0\ \textbf{then}\ 1\ \textbf{else}\ w \cdot f(w-1)$$

Bestimmung des Fixpunkts

Wir berechnen $\Phi^i(\bot)$, wobei $\bot \in \mathbb{Z} \to \mathbb{Z}_\bot$ die Funktion "konstant $\bot$" ist.

Es ergibt sich folgende Wertetabelle:

	< 0	0	1	2	3	4	5	> 5
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	
$\Phi(\bot)$	$\bot$	1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\Phi^2(\bot)$	$\bot$	1	1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\Phi^3(\bot)$	$\bot$	1	1	2	$\bot$	$\bot$	$\bot$	$\bot$
$\Phi^4(\bot)$	$\bot$	1	1	2	6	$\bot$	$\bot$	$\bot$
$\Phi^5(\bot)$	$\bot$	1	1	2	6	24	$\bot$	$\bot$

Wir sehen, dass $\bigsqcup_i \Phi^i(\bot) = \mathbf{function} w.w!$.

Operationelle Adäquatheit

Es gilt der folgende Satz:

Haben zwei Terme die gleiche denotationelle Semantik, so kann man in einem beliebigen Programm den einen Term durch den anderen ersetzen, ohne das Verhalten des Programms im Sinne der operationellen Semantik zu verändern.

Insofern stellt die operationelle Semantik eine korrekte Implementierung der denotationellen Semantik dar.

Will man umgekehrt die operationelle Semantik als “maßgeblich” auffassen, so bedeutet der Satz, dass die denotationelle Semantik ein korrektes Schlussprinzip für Austauschbarkeit von Programmteilen liefert.