

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: Kurzsriptum 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_{ab}, m_{ab} und s_{an}, m_{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_{ab} , m_{ab} und s_{an} , m_{an}

Ausgabe: natürliche Zahl

Berechnung:

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

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_{ab} , m_{ab} , s_{an} , m_{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, int m_an)  
    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 $s_{\text{an}}, s_{\text{ab}}, m_{\text{an}}, m_{\text{ab}} : \text{nat}$

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

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

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 Maschinsprache 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

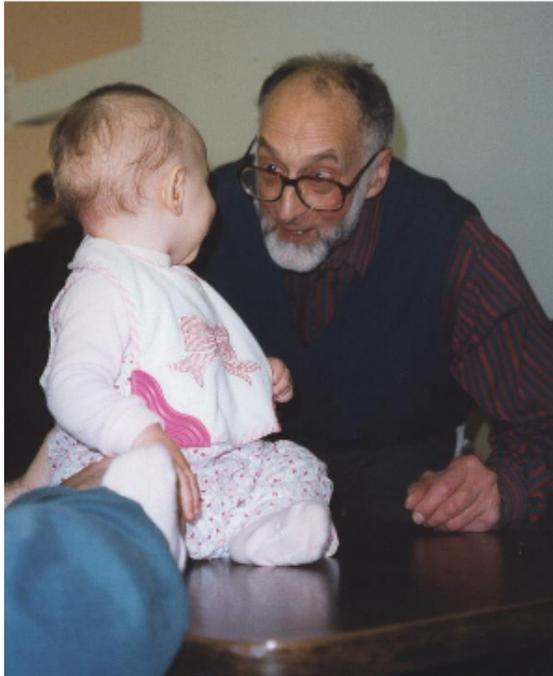
Programmiersprachen für Spezialanwendungen

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

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

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

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

end

Programmierung

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, int m_an)  
    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

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

$$f(x) = 1/x$$

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

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

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

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

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

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

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

Beispiele

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

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

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

Terminologie

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

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

Z.B.:

$$f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x, y) = x + 2y^2$$

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, \dots, A_n Mengen, so bildet man das kartesische Produkt

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

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

Zum Beispiel

$$\text{fahrzeit} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

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.

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

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

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

Typen

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

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

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

In Pseudocode schreiben wir auch

`nat, real, int, bool, A_1 * ... * A_n.`

Terme

Terme sind Ausdrücke mit Variablen (Platzhaltern).

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

Dieser Term enthält die Variablen s_{ab} , m_{ab} und s_{an} , m_{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_{ab} , m_{ab} und s_{an} , m_{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 Werte kann ein Term undefiniert sein, etwa x/y für $y = 0$.

Funktionen aus Termen

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

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

Wir bilden eine Funktion

$$\mathbf{function}(x_1, \dots, x_n)t : A_1 \times \dots \times A_n \rightarrow B$$

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

Beispiel:

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

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

Funktionstypen

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

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

Zum Beispiel ist **function**(x) x ein Term vom Typ $A \rightarrow A$ 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 $\int_0^1 e^x dx$ 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 $\int_0^\infty e^{-st} f(t) dt$ enthält die Variable s vom Typ \mathbb{R} und die Variable f vom Typ $\mathbb{R} \rightarrow \mathbb{R}$.

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:

```
function( $s_{ab}$ ,  $m_{ab}$ ,  $s_{an}$ ,  $m_{an}$ )  
  let  $zwErg = (s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab}$  in  
  let  $minProTag = 24 \cdot 60$  in  
  if  $zwErg \geq 0$  then  $zwErg$  else  $zwErg + minProTag$ 
```

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

$$<, >, \leq, \geq: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$$

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

$$f : A_1 \times \cdots \times A_n \rightarrow B$$

$$f = \mathbf{function}(x_1, \dots, x_n)t$$

schreiben wir auch

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

oder

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

Beispiele

```
function( $s_{ab}:\mathbf{nat}$ ,  $m_{ab}:\mathbf{nat}$ ,  $s_{an}:\mathbf{nat}$ ,  $m_{an}:\mathbf{nat}$ )nat  
  let  $zwErg = (s_{an} - s_{ab}) \cdot 60 + m_{an} - m_{ab}$  in  
  let  $minProTag = 24 \cdot 60$  in  
  if  $zwErg \geq 0$  then  $zwErg$  else  $zwErg + minProTag$ 
```

Rekursive Funktionen

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

Beispiel:

fakultät = **function**(n)**if** $n = 0$ **then** 1 **else** $n \cdot$ 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 undefinierterheit:

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 $f : \mathbb{N} \rightarrow \mathbb{N}$, aber $D(f) = \{0\}$.

Mehr Rekursion

Die Fibonacci-Zahlen:

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

fib = function(*n*)**if** *n* = 0 **then** 1 **else if** *n* = 1 **then** 1 **else** fib(*n* - 1) + fib(*n* - 2)

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

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

Nochmal $3n+1$

$g = \mathbf{function}(n)$ **if** $n \bmod 2 = 0$ **then** $n/2$ **else** $3n + 1$
 $f = \mathbf{function}(n)$ **if** $n = 1$ **then** 0 **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 = $\{(i, j) \mid i, j \in \text{Turm}, i \neq j\}$

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

Lösung : $\mathbb{N} \times \text{Turm} \times \text{Turm} \rightarrow \text{Befehlsfolge}$

Lösung = **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($n - 1, i, k$) \wedge (i, j) \wedge 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 $\text{istPrim} : \mathbf{nat} \rightarrow \mathbf{bool}$ mit $\text{istPrim}(n) = \mathbf{true}$ gdw., n prim.

$\text{istPrim} = \mathbf{function}(n) \mathbf{if } n = 0 \vee n = 1 \mathbf{ then false else if } n = 2 \mathbf{ then true else ?}$

Primzahlen

keineTeiler = function($n:\text{nat}$, $k:\text{nat}$)bool

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

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

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

istPrim = function($n:\text{nat}$)bool

$n > 1 \wedge \text{keineTeiler}(n, 2)$

Beispiel: Binäre Suche

Man soll ein Wort im Lexikon suchen.

$\text{suche} : \text{Lexikon} \times \text{Wort} \rightarrow \mathbf{bool}$

$\text{suche}(l, w) = \text{“}w \text{ kommt in } l \text{ vor“}$

Noch nicht detailliert genug.

Wir nehmen an, es gibt eine Funktion

$\text{ntesWort} : \text{Lexikon} \rightarrow \text{Wort}$

die das n -te Wort im Lexikon liefert.

Lösung eins: alle durchprobieren

```
sucheBis = function(l, w, n:nat)bool
  if n = 0 then false else
    sucheBis(l, w, n - 1) ∨ w = ntesWort(l, n)
suche = function(l, w)
  sucheBis(l, w, 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 =  $\lfloor (i + j) / 2 \rfloor$  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))
```

Abstiegssfunktion

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

Sei

$$f : A \rightarrow B$$

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

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

$\Phi(f, x)$ 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 $A' \subseteq D(f)$.

Abstiegssfunktion

Sei nun zusätzlich $m : A \rightarrow \mathbb{N}$ 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 *Abstiegssfunktion*.

Beispiel Fakultät

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

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

Beispiel keineTeiler

keineTeiler = **function**($n:\mathbf{nat}$, $k:\mathbf{nat}$)**bool**

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

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

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

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

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, \dots 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 $m : A \rightarrow \mathbb{N}$ auch eine Abstiegsfunktion $m : A \rightarrow M$ 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:

```
ackermann = function(x:nat, y:nat)nat
  if x = 0 then y + 1 else
    if y = 0 then ackermann(x - 1, 1)
      else ackermann(x - 1, ackermann(x, y - 1))
```

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

```
gerade = function(x:nat)if x = 0 then true else ungerade(x - 1)
ungerade = function(x:nat)if x = 0 then false else gerade(x - 1)
```

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

Verschränkte Rekursion über kartesische Produkte

Man kann verschränkte Rekursion durch Produkte simulieren:

```
gerade/ungerade = function(x:nat)bool × bool
  if x=0 then (true, false) else
    let (g, u) = gerade/ungerade(x - 1) in
      (u, g)
```

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 ,
- $m : A \rightarrow M$ 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 $m(y) R m(a)$ 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

$$\begin{aligned}\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) = a\phi^{n-1} + b(-\phi)^{-n+1} + a\phi^{n-2} + \\ &b(-\phi)^{-n+2} = a\phi^n(\phi^{-1} + \phi^{-2}) + b(-\phi)^{-n}(-\phi + \phi^2) = a\phi^n + b(-\phi)^{-n}.\end{aligned}$$

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

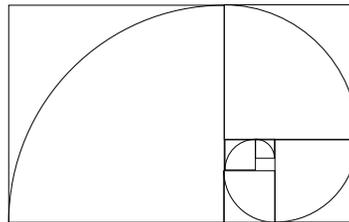
Kettenbruch und Kettenwurzel

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

$$1/\phi = \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \dots}}}}$$

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

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 $n \in P$.

^aHöhere Mächte befahlen: Linke obere Ecke schwarz malen (SIGMAR POLKE)

Exponentiation

```
exp = function(x:real, n:nat)real
  if n=0 then 1
    else if n gerade then exp(x, n div 2)2
      else exp(x, n div 2)2 · x
```

Hier ist $x \text{ div } 2$ die ganzzahlige Division, z.B., $5 \text{ div } 2 = 2$.

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

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:

$$\text{erstes} : \alpha \times \beta \rightarrow \alpha$$

$$\text{erstes} = \text{function}(x, y)x$$

$$\text{tausch} : \alpha \times \beta \rightarrow \beta \times \alpha$$

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

Polymorphe Funktionen

Man kann eine polymorphe Funktionen $f : typ \rightarrow typ'$ 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.

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

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

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

Funktionen höherer Ordnung

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

Funktionen können auch als Wert zurückgegeben werden:

$$\text{plus} = \text{function}(x) \\ \text{function}(y)x + y$$

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

Funktionen höherer Ordnung

Allgemein kann man zu jeder Funktion

$$f_1 : \text{typ}_1 \times \cdots \times \text{typ}_n \rightarrow \text{typ}$$

eine Funktion

$$f_2 : \text{typ}_1 \rightarrow \text{typ}_2 \rightarrow \cdots \rightarrow \text{typ}_n \rightarrow \text{typ}$$

definieren durch

$$f_2 = \mathbf{function}(x_1)\mathbf{function}(x_2) \dots \mathbf{function}(x_n)f_1(x, \dots, x_n)$$

Es ist

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

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

iteriere = **function**($f:\alpha \rightarrow \alpha, x:\alpha, n:\mathbf{nat}$) α
if $n = 0$ **then** x **else** iteriere($f, f(x), n - 1$)

Es ist

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

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

komponiere = **function**($g:\beta \rightarrow \gamma, f:\alpha \rightarrow \beta$) $\alpha \rightarrow \gamma$
function($x:\alpha$) $g(f(x))$

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

Noch ein Beispiel zur Induktion

Seien $f : A \rightarrow B$ und $g : B \rightarrow A$ 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 `+`, `-`, `*`, `/` 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 Billionen 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 $\text{int} = \{-2^{30}, \dots, 2^{30} - 1\} \approx \mathbb{Z}$.

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

Einen Datentyp nat gibt es nicht!

Überlauf

```
let guthaben = 8000000000;;  
guthaben * 2;;  
- : int = -547483648
```

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

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 $\text{fib2 } n = (\text{fib } n, \text{fib } (n + 1))$.

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 bool
```

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 ß§ä, 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 $u = u_1, \dots, u_m$ und $v = v_1, \dots, v_n$ Zeichenketten (also $u_i, v_j \in \text{char}$).

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

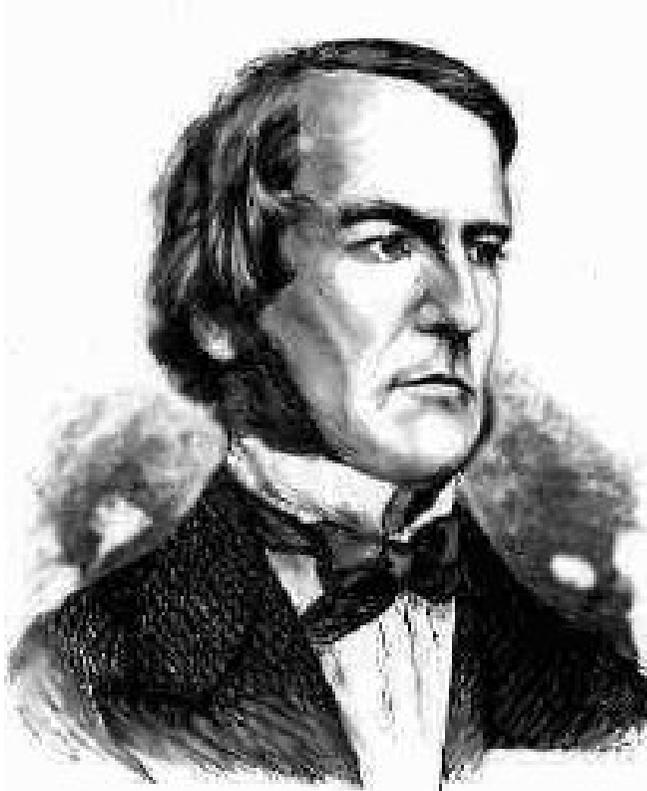
Z.B.

"AAAAAAA" < "Anderer Schluesseldienst"

"Martin" < "Martina"

"Lexikografisch" < "Lexikon"

Boole'sche Operationen



GEORGE BOOLE 1815–1864

| Pseudocode | OCAML |
|------------|-------------------------|
| \wedge | <code>&&</code> |
| \vee | <code> </code> |
| \neg | <code>not</code> |

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 prl &. 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 $\sigma_1, \dots, \sigma_n$ von Σ , wobei $n \geq 0$. Man schreibt ein Wort als $\sigma_1\sigma_2 \dots \sigma_n$. Der Fall $n = 0$ bezeichnet das *leere Wort* geschrieben ε .
- Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet. Zu zwei Wörtern $w = \sigma_1 \dots \sigma_n$ und $w' = \sigma'_1 \dots \sigma'_m$ bildet man die *Verkettung* (Konkatenation) $ww' = \sigma_1 \dots \sigma_n \sigma'_1 \dots \sigma'_m$. 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, ε , bbbb*.

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

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

Wörter über Σ : *-1E98, --2e--, 32.e*

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

$\Sigma = \{0, \dots, 9, \text{if, then, let, } \dots \}$

Sprache über Σ : alle syntaktisch korrekten OCAML Phrasen.

Beispiel: OCAML-Bezeichner

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

Formale Definition als BNF-Grammatik

BNF = *Backus-Naur Form*

$\langle \text{Bezeichner} \rangle ::= \{ \langle \text{KlBuchst} \rangle \mid - \} \{ \langle \text{Buchst} \rangle \mid \langle \text{Ziffer} \rangle \mid ' \mid - \}^*$

$\langle \text{Buchst} \rangle ::= \langle \text{KlBuchst} \rangle \mid \langle \text{GrBuchst} \rangle$

$\langle \text{KlBuchst} \rangle ::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o$
 $\mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$

$\langle \text{GrBuchst} \rangle ::= A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O$
 $\mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z$

$\langle \text{Ziffer} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

float-Literale

Literal = Konstante in einer Programmiersprache.

$$\langle \text{float-Literal} \rangle ::= [\langle \text{Vorzeichen} \rangle] \langle \text{Mantisse} \rangle [\langle \text{Exponent} \rangle]$$
$$\langle \text{Vorzeichen} \rangle ::= - \mid +$$
$$\langle \text{Mantisse} \rangle ::= \{ \langle \text{Ziffer} \rangle \}^+ [\cdot \{ \langle \text{Ziffer} \rangle \}^*]$$
$$\langle \text{Exponent} \rangle ::= \{ e \mid E \} [\langle \text{Vorzeichen} \rangle] \{ \langle \text{Ziffer} \rangle \}^+$$

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:
$$V = \{\langle \text{float-Literal} \rangle, \langle \text{Mantisse} \rangle, \langle \text{Vorzeichen} \rangle, \langle \text{Exponent} \rangle\}.$$
- $S \in V$ ist ein ausgezeichnetes Nichtterminalsymbol, das *Startsymbol*.
Im Beispiel: $S = \langle \text{float-Literal} \rangle$.
- 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 $\gamma_1, \dots, \gamma_n$ BNF-Satzformen, so auch $\gamma_1 \mid \dots \mid \gamma_n$ (*Auswahl*).
- Sind $\gamma_1, \dots, \gamma_n$ BNF-Satzformen, so auch $\gamma_1 \dots \gamma_n$ (*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 $\gamma_1 \mid \dots \mid \gamma_n$ muss immer geklammert werden, es sei denn, sie tritt unmittelbar als rechte Seite einer Produktion auf.

Diese Folie wurde gestrichen, die folgende sinngemäß verändert.

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 $\overbrace{\{\gamma\}\{\gamma\}\dots\{\gamma\}}^{n\text{-mal}}$ mit $n \geq 0$ ersetzen.
- Ein Vorkommen von $\{\gamma\}^+$ darf man durch $\overbrace{\{\gamma\}\{\gamma\}\dots\{\gamma\}}^{n\text{-mal}}$ 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

$\langle \text{float-Literal} \rangle \rightarrow [\langle \text{Vorzeichen} \rangle] \langle \text{Mantisse} \rangle [\langle \text{Exponent} \rangle] \rightarrow$
 $\langle \text{Mantisse} \rangle [\langle \text{Exponent} \rangle] \rightarrow \{ \langle \text{Ziffer} \rangle \}^+ [\cdot \{ \langle \text{Ziffer} \rangle \}^*] [\langle \text{Exponent} \rangle] \rightarrow$
 $\{ \langle \text{Ziffer} \rangle \}^+ \cdot \{ \langle \text{Ziffer} \rangle \}^* [\langle \text{Exponent} \rangle] \rightarrow$
 $\{ \langle \text{Ziffer} \rangle \}^+ \cdot \{ \langle \text{Ziffer} \rangle \} \{ \langle \text{Ziffer} \rangle \} [\langle \text{Exponent} \rangle] \rightarrow$
 $\{ \langle \text{Ziffer} \rangle \} \cdot \{ \langle \text{Ziffer} \rangle \} \{ \langle \text{Ziffer} \rangle \} [\langle \text{Exponent} \rangle] \rightarrow 2.71 \langle \text{Exponent} \rangle \rightarrow$
 $2.71 \{ e \mid E \} [\langle \text{Vorzeichen} \rangle] \{ \langle \text{Ziffer} \rangle \}^+ \rightarrow$
 $2.71E \langle \text{Vorzeichen} \rangle \{ \langle \text{Ziffer} \rangle \} \{ \langle \text{Ziffer} \rangle \} \{ \langle \text{Ziffer} \rangle \} \rightarrow$
 $2.71E \{ - \mid + \} 001 \rightarrow 2.71E-001$

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

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)

$\langle \text{toplevel-command} \rangle ::= \text{let } \langle \text{let-binding} \rangle \{ \text{and } \langle \text{let-binding} \rangle \}^* ; ;$
| $\text{let rec } \langle \text{let-binding} \rangle \{ \text{and } \langle \text{let-binding} \rangle \}^* ; ;$
| $\langle \text{expression} \rangle ; ;$

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

$\langle \text{infix-op} \rangle ::= + \mid - \mid * \mid / \mid +. \mid -. \mid *. \mid /. \mid \&\& \mid || \mid \text{mod}$
| $< \mid > \mid <> \mid <= \mid >= \mid <>$

$\langle \text{let-binding} \rangle ::= \{ \langle \text{ident} \rangle \}^+ = \langle \text{expression} \rangle$

Parsergeneratoren

Erinnerung: *Parser* : $\Sigma^* \rightarrow \text{Ableitungsbäume} \cup \text{Fehlermeldungen}$.

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*.

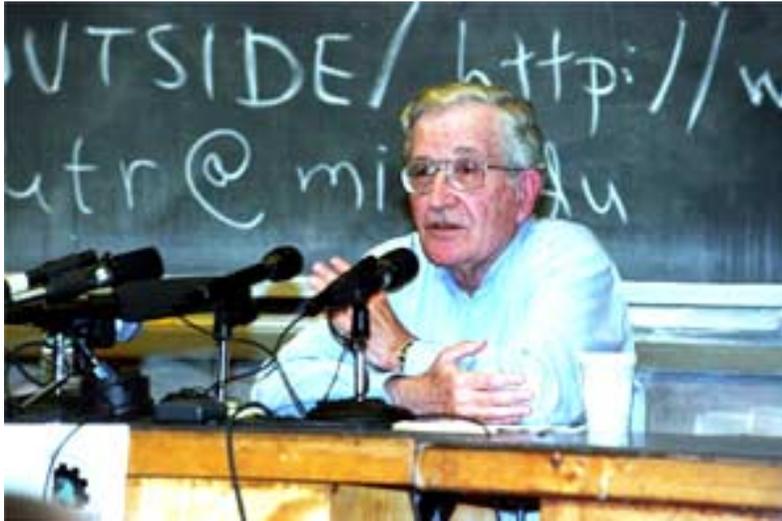
Sie gehen auf den Sprachforscher NOAM CHOMSKY (1928–) zurück.

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

$\langle \text{Bezeichner} \rangle ::= \langle \text{KlBuchst} \rangle \langle \text{Folge} \rangle$

$\langle \text{Bezeichner} \rangle ::= - \langle \text{Folge} \rangle$

$\langle \text{Folge} \rangle ::= \langle \text{Zeichen} \rangle \langle \text{Folge} \rangle$

$\langle \text{Folge} \rangle ::= \varepsilon$

$\langle \text{Zeichen} \rangle ::= ' \quad '$

$\langle \text{Zeichen} \rangle ::= -$

$\langle \text{Zeichen} \rangle ::= \langle \text{Buchst} \rangle$

$\langle \text{Zeichen} \rangle ::= \langle \text{Ziffer} \rangle$

$\langle \text{Buchst} \rangle ::= \langle \text{KlBuchst} \rangle$

$\langle \text{Buchst} \rangle ::= \langle \text{GrBuchst} \rangle$

Kapitel 3.4 Termauswertung

Präzedenz- und Assoziationsregeln

| Operator | Präzedenz | Assoziativität | Erklärung |
|-----------------------------|-----------|----------------|---|
| <i>Funktionsapplikation</i> | | 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, <i>in praxi</i> meist 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* $\langle a, w \rangle$ 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 $\langle x, w \rangle$ der aktuellen Umgebung hinzugefügt, wobei w der berechnete Wert von t ist.

Eine frühere Bindung der Form $\langle x, w' \rangle$ 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 $\langle x, w \rangle \in U$.
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

$$W^U(t_1 \text{ op } t_2) = W^U(t_1) \oplus W^U(t_2).$$

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

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

Auswertung von Funktionstermen

Notation:

$$U + \{ \langle x_1, w_1 \rangle, \dots, \langle x_n, w_n \rangle \} = U' \cup \{ \langle x_1, w_1 \rangle, \dots, \langle x_n, w_n \rangle \},$$

wobei U' aus U durch Entfernen aller eventuell vorhandenen Bindungen von x_1, \dots, 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 $t = \text{function}(x_1, \dots, x_n) \rightarrow t'$. Im Falle $n = 1$ auch ohne Klammern. Es ist $W^U(t)$ diejenige Funktion, die (w_1, \dots, w_n) auf $W^{U + \{ \langle x_1, w_1 \rangle, \dots, \langle x_n, w_n \rangle \}}(t')$ 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 $W^U(t_1) = \text{true}$, so ist $W^U(t) = W^U(t_2)$. Beachte: $W^U(t_3)$ kann dann undefiniert sein.

Ist $W^U(t_1) = \text{false}$, 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 $(x_1, \dots, x_n) = t_1$ in t_2` im Falle $n = 1$ auch ohne Klammern.

Sei $W^U(t_1)$ definiert und von der Form $W^U(t_1) = (w_1, \dots, w_n)$, also ein n -Tupel von Werten. Dann ist

$W^U(t) = W^{U+\{\langle x_1, w_1 \rangle, \dots, \langle x_n, w_n \rangle\}}(t_2)$. 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 $t = (t_1, \dots, t_n)$. Seien weiter $W^U(t_1) = w_1, \dots, W^U(t_n) = w_n$ allesamt definiert. Dann ist $W^U(t) = (w_1, \dots, w_n)$.
- $W^U(t_1 \ || \ t_2) = W^U(\text{if } t_1 \text{ then true else } t_2)$
- $W^U(t_1 \ \&\& \ t_2) = W^U(\text{if } t_1 \text{ then } t_2 \text{ else false})$.

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

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

^aKann sein, dass die Folien auch nicht ganz richtig sind, maßgeblich ist daher der gesunde Menschenverstand :-)

Beispiel

$$\begin{aligned} & W\{\langle x, 1 \rangle\}(\text{let } x = 2 \text{ in if } x=2 \text{ then } 0 \text{ else } 1) \\ = & W\{\langle x, 2 \rangle\}(\text{if } x=2 \text{ then } 0 \text{ else } 1) \\ = & W\{\langle x, 2 \rangle\}(0) \\ = & 0 \end{aligned}$$

Semantik von `let rec`

Eine rekursive `let`-Bindung

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

bindet f an die durch

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

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

Es ist

$$\begin{aligned} & F(3) \\ = & W\{\langle \text{fakt}, F \rangle, \langle n, 3 \rangle\}(\text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fakt}(n-1)) \\ = & W\{\langle \text{fakt}, F \rangle, \langle n, 3 \rangle\}(n * \text{fakt}(n-1)) \\ = & W\{\langle \text{fakt}, F \rangle, \langle n, 3 \rangle\}(n) \cdot W\{\langle \text{fakt}, F \rangle, \langle n, 3 \rangle\}(\text{fakt}(n-1)) \\ = & 3 \cdot F(W\{\langle \text{fakt}, F \rangle, \langle n, 3 \rangle\}(n-1)) \\ = & 3 \cdot F(2) \\ = & W\{\langle \text{fakt}, F \rangle, \langle n, 2 \rangle\}(\text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fakt}(n-1)) \\ = & \dots = 6 \end{aligned}$$

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.

$$W^U(2 * 3 - 1)$$

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

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 ausgewerteter 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

$f = \mathbf{function}(x, y)$
if $y = 0$ **then** 0 **else**
if y gerade **then** $2 \cdot f(x, y/2)$ **else** $2f(x, \lceil y/2 \rceil) + x$

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

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

$f = \text{function}(x:\text{int})\text{int}$

if $x > 100$ **then** $x - 10$ **else** $f(f(x + 11))$

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

$$\begin{aligned} f(70) &= f(f(81)) = f^3(92) = f^4(103) = f^4(93) = f^4(104) = f^3(94) = \\ f^4(105) &= f^3(95) = f^4(106) = f^3(96) = f^4(107) = f^3(97) = \\ f^4(108) &= f^3(98) = f^4(109) = f^3(99) = f^4(110) = f^3(100) = \\ f^4(111) &= f^3(91) = \dots = 91 \end{aligned}$$

Man beweise: falls $x \in D(f)$, so gilt

$$f(x) = \text{if } x > 100 \text{ then } x - 10 \text{ else } 91.$$

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 $y \rightarrow y + x$, 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 $\langle x, w \rangle$ von Bezeichnern an Werte.
- Ist e ein OCAML-Ausdruck, x ein Bezeichner und U eine Umgebung, so ist $(\text{fun } x \rightarrow e, U)$ ein Wert.
- Ist e ein OCAML-Ausdruck, f und x Bezeichner und U ein Umgebung, so ist $(f = \text{fun } x \rightarrow e, U)$ 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>}).
```

Auswertrelation

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

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

Auswerteregeln

- ist c eine Konstante, so gilt $U, c \rightarrow c$
- ist x ein Bezeichner und $\langle x, w \rangle \in U$, so gilt $U, x \rightarrow w$.
- für Funktionsausdrücke gilt $U, \text{fun } x \rightarrow e \rightarrow (\text{fun } x \rightarrow e, U')$, 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 $U, e_1 \rightarrow w_1$ und $U, e_2 \rightarrow w_2$, dann $U, e_1 op e_2 \rightarrow w_1 \oplus w_2$.

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

Einstellige Basisfunktionen werden analog behandelt.

Fallunterscheidung

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

Applikation

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

Bindung

- Ist $U, e_1 \rightarrow w_1$, so ist zunächst (in einer Nebenrechnung) ein Wert w_2 mit $U + \{ \langle x, w_1 \rangle \}, e_2 \rightarrow w_2$ zu bestimmen. Es ist dann $U, \text{let } x=e_1 \text{ in } e_2 \rightarrow w_2$.
- Es gilt $U, \text{let rec } f=\text{fun } x \rightarrow e_1 \text{ in } e_2 \rightarrow w$, falls $U + \{ \langle f, (f=\text{fun } x \rightarrow e_1, U') \rangle \}, e_2 \rightarrow w$ 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 $\emptyset, 1 + 0 \rightarrow 1$ 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 $U_1 = \{\langle x, 1 \rangle\}$ auszuwerten. Das aber bedeutet, in der Umgebung

$$U_2 = \{\langle x, 1 \rangle, \langle f, (f = \text{fun } y \rightarrow \text{if } y=0 \text{ then } x \text{ else } f(y-1)), U_1 \rangle\}$$

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

Beispiel

bedeutet, in der Umgebung

$$U_3 = \{ \langle x, 2 \rangle, \langle f, (f = \text{fun } y \rightarrow \text{if } y=0 \text{ then } x \text{ else } f(y-1)), U_1 \rangle \}$$

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

Es gilt $U_3, x+x \rightarrow 4$ und

$U_3, f \rightarrow (f = \text{fun } y \rightarrow \text{if } y=0 \text{ then } x \text{ else } f(y-1)), U_1$, also

müssen wir in der Umgebung

$$U_4 = \{ \langle x, 1 \rangle, \langle f, (f = \text{fun } y \rightarrow \text{if } y=0 \text{ then } x \text{ else } f(y-1)), U_1 \rangle, \langle y, 4 \rangle \}$$

den Ausdruck

$\text{if } y=0 \text{ then } x \text{ else } f(y-1)$

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 $\emptyset, e \rightarrow w$.

(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 $\emptyset, e_3 \rightarrow w$ für ein w genau dann wenn $\emptyset, e_4 \rightarrow w'$ 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: $true \oplus 8$ 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

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

wobei e ein OCAML-Ausdruck ist, typ ein OCAML-Typ ist und Γ eine Menge von Bindungen der Form $x_i:\text{typ}_i$ 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:

```
a:int  $\triangleright$  fun x -> a + 2 * x : int->int
```

Typisierungsregeln

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

- Typaxiome sind elementare Typaussagen wie etwa $\emptyset \triangleright 7 : \text{int}$ oder $\emptyset \triangleright 7e1 : \text{float}$.
- Typisierungsregeln haben die Form $P_1, \dots, P_m \vdash K$, 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, \dots, 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, \dots, A_{i-1} sind.

Typisierungsregel für Paare

$$\Gamma_1 \triangleright e_1 : typ_1$$

$$\Gamma_2 \triangleright e_2 : typ_2$$

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

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

$$\Gamma_1 \triangleright e_1 : \text{int}$$
$$\Gamma_2 \triangleright e_2 : \text{int}$$
$$\vdash \Gamma_1 \cup \Gamma_2 \triangleright e_1 \text{ op } e_2 : \text{int}$$

wobei $op \in \{+, *, /, \text{div}, \text{mod}\}$ 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

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

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

$$\Gamma \cup \{x:typ\} \triangleright e : typ'$$
$$\vdash$$
$$\Gamma \triangleright \text{function } x \rightarrow e : typ \rightarrow typ'$$

Beispiel

- (1) $\emptyset \triangleright 2 : \text{int}$
- (2) $\{a : \text{int}\} \triangleright a : \text{int}$
- (3) $\{x : \text{int}\} \triangleright x : \text{int}$
- (4) $\{x : \text{int}\} \triangleright 2 * x : \text{int}$
- (5) $\{a : \text{int}, x : \text{int}\} \triangleright a + 2 * x : \text{int}$
- (6) $\{a : \text{int}\} \triangleright \text{function } x \rightarrow a + 2 * x : \text{int} \rightarrow \text{int}$

Typisierungsregel für Applikation

$$\Gamma_1 \triangleright e_1 : \text{typ} \rightarrow \text{typ}'$$
$$\Gamma_2 \triangleright e_2 : \text{typ}$$
$$\vdash$$
$$\Gamma_1 \cup \Gamma_2 \triangleright e_1 e_2 : \text{typ}'$$

Typisierungsregel für let

$$\Gamma_1 \triangleright e_1 : \text{typ}$$

$$\Gamma_2 \cup \{x:\text{typ}\} \triangleright e_2 : \text{typ}'$$

⊢

$$\Gamma_1 \cup \Gamma_2 \triangleright \text{let } x=e_1 \text{ in } e_2 : \text{typ}'$$

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

Typisierungsregel für `let rec`

$$\Gamma_1 \cup \{f:typ\} \triangleright e_1 : typ$$

$$\Gamma_2 \cup \{f:typ\} \triangleright e_2 : typ'$$

⊢

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

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:

$$\{x:\text{int}, f:\text{int}\rightarrow\text{int}\} \triangleright f \ x : \text{int}$$
$$\{x:\text{int}*\text{int}, f:\text{int}*\text{int}\rightarrow\text{int}\} \triangleright f \ x : \text{int}$$
$$\{x:'a, f:'a\rightarrow'a\} \triangleright f \ x : 'a$$
$$\{x:'a, f:'a\rightarrow'b\} \triangleright f \ x : 'b$$

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 *Instanziierung* (früher *Instantiierung*) der Typvariablen.

Instanziierungsregel

$$\Gamma \triangleright e : \text{typ}$$
$$\vdash \Gamma[\alpha_1 := \text{typ}_1, \dots, \alpha_m := \text{typ}_m] \triangleright e : \text{typ}[\alpha_1 := \text{typ}_1, \dots, \alpha_m := \text{typ}_m]$$

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

(1) $\{x: 'a, f: 'a \rightarrow 'b\} \triangleright f \ x : 'b$

(2) $\{x: 'a, f: 'a \rightarrow 'a\} \triangleright f \ x : 'a$

(3) $\{x:\text{float}, f:\text{float} \rightarrow \text{int}\} \triangleright f \ x : \text{int}$

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

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 $'a * ('b \rightarrow 'c) \rightarrow ('b \rightarrow 'a)$ und e_2 den prinzipalen Typ $('a \rightarrow 'a) * (int \rightarrow float)$, so hat $e_1 e_2$ den prinzipalen Typ $int \rightarrow 'a \rightarrow 'a$.

Unifikation

Allgemein: hat der geschlossene Term e_1 den prinzipalen Typ $typ_1 \rightarrow typ_2$ und hat der geschlossene Term e_2 den prinzipalen Typ typ'_1 , so muss man die allgemeinstmögliche Instanzierung $\sigma = [\alpha_1 := typ_1, \dots, \alpha_m := typ_m]$ 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 allgemeinsten 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

$$\begin{aligned} \langle expression \rangle & ::= \dots \\ & \quad | \text{ raise } \langle Ident \rangle \\ & \quad | \text{ try } \langle expression \rangle \text{ with } \langle ident \rangle \rightarrow \langle expression \rangle \\ & \quad \quad \quad \{ | \langle Ident \rangle \rightarrow \langle expression \rangle \}^* \end{aligned}$$

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

Typisierung: Die Typen aller `with`-Klauseln müssen mit dem Typ des Ausdrucks 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: $\mathcal{E} = \{\text{Fehler}\}$.

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

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

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

$$W^U(\text{try } e_1 \text{ with } A \rightarrow e_2) = \mathbf{if } W^U(e_1) = A \in \mathcal{E} \mathbf{ then } W^U(e_2) \mathbf{ else } W^U(e_1)$$

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 begeben:

```
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

$$\begin{aligned} \langle pattern \rangle & ::= \langle ident \rangle \\ & | \langle const \rangle \\ & | - \\ & | (\langle pattern \rangle \{ , \langle pattern \rangle \}^*) \\ & | \langle pattern \rangle \text{ when } \langle expression \rangle \end{aligned}$$

Kontextbedingung: eine *when*Klausel darf nur ganz außen stehen, also nicht $(0 , m \text{ when } m > 0)$. Stattdessen: $(0 , m) \text{ when } m > 0$.

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 $\{ \langle x, w \rangle \}$
- $_$ (underscore, *wildcard*) passt immer; das Ergebnis ist $\{ \}$
- (p_1, \dots, p_m) passt, wenn $w = (w_1, \dots, w_m)$ ist und w_i jeweils auf p_i passt (mit Ergebnis U_i . Ergebnis ist dann $U_1 + \dots + U_m$.
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

$$e = \text{match } e' \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$$

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

Sei $w = W^U(e)$ und sei p_{i_0} das **erste** Muster aus p_1, \dots, 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 $\text{match } e' \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$ (unter Typzuweisung Γ) typkorrekt ist, muss zunächst e' einen Typ typ' haben, d.h., $\Gamma \triangleright e' : \text{typ}'$.

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

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

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

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;
                    diensttritt: datum}
```

Allgemein

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

$$\text{type } t = \{f_1 : typ_1 ; \dots ; f_n : typ_n\}$$

ein neuer Typ t definiert, dessen Werte von der Form

$$\{f_1 = w_1 ; \dots ; f_n = w_n\}$$

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

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

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, \dots, F_n Konstruktoren und sind typ_1, \dots, typ_n Typen, so wird durch

$$\text{type } t = F_1 \text{ of } typ_1 \mid \dots \mid F_n \text{ of } typ_n$$

ein neuer Typ t definiert, dessen Werte von der Form $F_i(w_i)$ sind, wobei $i \in \{1, \dots, n\}$ 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

$\{A(0), B(0), A(1), B(1), A(-1), B(-1), A(2), B(-2), A(3), B(-3), \dots\}$.

“Mathematisch” ist das dasselbe, wie

$\{(0, 0), (1, 0), (0, 1), (1, 1), (0, -1), (1, -1), \dots\}$.

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

$\{C, A(0), B(0), A(1), B(1), A(-1), B(-1), A(2), B(-2), A(3), B(-3), \dots\}$.

Oder “mathematisch”

$\{(2, 0), (0, 0), (1, 0), (0, 1), (1, 1), (0, -1), (1, -1), \dots\}$.

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:

$$\Gamma \triangleright e : typ \vdash \Gamma \triangleright F e : t$$

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 $\text{RGB}(181, 158, 12)$ passt auf das Muster $\text{RGB}(r, g, -)$.
Ergebnis ist $\{ \langle r, 181 \rangle, \langle g, 158 \rangle \}$.

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 $[a_1; \dots; a_n]$ anstatt (a_1, \dots, a_n) oder $a_1 \cdots a_n$.

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

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

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

Die Verkettung von Listen l_1 und l_2 schreiben wir $l_1 @ l_2$. Es gilt

$$[] @ l_2 = l_2$$

$$(a :: l) @ l_2 = a :: (l @ l_2)$$

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**,
- $[p_1 ; \dots ; p_n]$ passt die Liste $[w_1 ; \dots ; w_n]$, 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:

$$\text{rev_aux}(l, \text{acc}) = \text{rev}(l) @ \text{acc}$$

Sortieren durch Einfügen

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

```
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 $\text{insertel}(a,l)$ und es enthält dieselben Elemente wie $a :: l$.

Für beliebiges l ist $\text{inssort}(l)$ 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]]
```