

Programmierparadigmen

Vorwort

Dieses Skript wird/wurde im Wintersemester 2013/2014 von Martin Thoma zur Vorlesung von Prof. Dr. Snelting und Jun.-Prof. Dr. Hummel geschrieben. Dazu wurden die Folien von Prof. Dr. Snelting und Jun.-Prof. Dr. Hummel benutzt, die Struktur sowie einige Beispiele, Definitionen und Sätze übernommen.

Das Ziel dieses Skriptes ist vor allem in der Klausur als Nachschlagewerk zu dienen; es soll jedoch auch vorher schon für die Vorbereitung genutzt werden können und nach der Klausur als Nachschlagewerk dienen.

Ein Link auf das Skript ist unter martin-thoma.com/programmierparadigmen zu finden.

Anregungen, Verbesserungsvorschläge, Ergänzungen

Noch ist das Skript im Aufbau. Es gibt viele Baustellen und es ist fraglich, ob ich bis zur Klausur alles in guter Qualität bereitstellen kann. Daher freue ich mich über jeden Verbesserungsvorschlag.

Anregungen, Verbesserungsvorschläge und Ergänzungen können per Pull-Request gemacht werden oder mir per E-Mail an info@martin-thoma.de geschickt werden.

Erforderliche Vorkenntnisse

Grundlegende Kenntnisse vom Programmieren, insbesondere mit Java, wie sie am KIT in „Programmieren“ vermittelt werden, werden vorausgesetzt. Außerdem könnte ein grundlegendes Verständnis für das \mathcal{O} -Kalkül aus „Grundbegriffe der Informatik“ hilfreich sein.

Die Unifikation wird wohl auch in „Formale Systeme“ erklärt; das könnte also hier von Vorteil sein.

Die Grundlagen des Kapitels „Parallelität“ wurden in Softwaretechnik I (kurz: SWT I) gelegt.

Inhaltsverzeichnis

1	Programmiersprachen	3
1.1	Abstraktion	3
1.2	Paradigmen	5
1.3	Typisierung	7
1.4	Kompilierte und interpretierte Sprachen	10
1.5	Dies und das	10
2	Programmiertechniken	15
2.1	Rekursion	15
2.2	Backtracking	18
2.3	Funktionen höherer Ordnung	18
3	Logik	19
3.1	Prädikatenlogik erster Stufe	19
3.1.1	Symbole	19
3.1.2	Terme	20
3.1.3	Ausdrücke	21
3.1.4	1. Stufe	22
3.1.5	Freie Variablen	23
3.1.6	Metasprachliche Ausdrücke	23
3.1.7	Substitutionen	24
4	λ-Kalkül	27
4.1	Reduktionen	28
4.2	Auswertungsstrategien	29
4.3	Church-Zahlen	30
4.4	Church-Booleans	32
4.5	Weiteres	32

4.6	Fixpunktkombinator	33
4.7	Let-Polymorphismus	35
4.8	Literatur	37
5	Typinferenz	39
5.1	Beispiele	42
5.1.1	$\lambda x. \lambda y. x\ y$	42
5.1.2	Selbstapplikation	44
6	Parallelität	47
6.1	Architekturen	48
6.2	Prozesskommunikation	50
6.3	Parallelität in Java	52
6.4	Message Passing Modell	52
7	Haskell	55
7.1	Erste Schritte	55
7.1.1	Hello World	55
7.2	Syntax	56
7.2.1	Klammern und Funktionsdeklaration	56
7.2.2	if / else	57
7.2.3	Rekursion	57
7.2.4	Listen	58
7.2.5	Strings	60
7.2.6	Let und where	60
7.3	Typen	61
7.3.1	Standard-Typen	61
7.3.2	Typinferenz	61
7.3.3	type	64
7.3.4	data	64
7.4	Lazy Evaluation	64
7.5	Beispiele	65
7.5.1	Quicksort	65
7.5.2	Fibonacci	65
7.5.3	Polynome	66
7.5.4	Hirsch-Index	67

7.5.5	Lauf­längen­codierung	67
7.5.6	Intersections	68
7.5.7	Funktionen höherer Ordnung	69
7.5.8	Chruch-Zahlen	69
7.5.9	Standard Prelude	70
7.6	Weitere Informationen	71
8	Prolog	73
8.1	Erste Schritte	73
8.1.1	Hello World	73
8.2	Syntax	73
8.2.1	Arithmetik	74
8.2.2	Listen	75
8.3	Beispiele	76
8.3.1	Humans	76
8.3.2	Splits	77
8.3.3	Delete	77
8.3.4	Zebrarätsel	78
8.4	Weitere Informationen	79
9	Scala	81
9.1	Erste Schritte	81
9.1.1	Hello World	81
9.2	Vergleich mit Java	82
9.3	Syntax	83
9.4	Companion Object	84
9.5	actor	84
9.6	Beispiele	84
9.6.1	Wetter	84
9.7	Weitere Informationen	85
10	X10	87
10.1	Erste Schritte	87
10.2	Syntax	88
10.3	Datentypen	88
10.4	Beispiele	88

10.5 Weitere Informationen	88
11 C	89
11.1 Datentypen	89
11.2 ASCII-Tabelle	90
11.3 Syntax	90
11.4 Präzedenzregeln	90
11.5 Beispiele	90
11.5.1 Hello World	90
11.5.2 Pointer	91
12 MPI	95
12.1 Erste Schritte	95
12.2 Funktionen	96
12.3 Beispiele	102
12.4 Weitere Informationen	102
13 Compilerbau	103
13.1 Funktionsweise	105
13.2 Lexikalische Analyse	105
13.2.1 Reguläre Ausdrücke	105
13.2.2 Lex	106
13.3 Syntaktische Analyse	107
13.4 Semantische Analyse	107
13.5 Zwischencodeoptimierung	108
13.6 Codegenerierung	108
13.7 Literatur	109
14 Java Bytecode	111
14.1 Instruktionen	111
14.2 Weitere Informationen	112
Bildquellen	113
Abkürzungsverzeichnis	115
Ergänzende Definitionen	117

Symbolverzeichnis	121
Stichwortverzeichnis	123

1 Programmiersprachen

Definition 1

Eine **Programmiersprache** ist eine formale Sprache, die durch eine Spezifikation definiert wird und mit der Algorithmen beschrieben werden können. Elemente dieser Sprache heißen **Programme**.

Ein Beispiel für eine Sprachspezifikation ist die *Java Language Specification*.¹ Obwohl es kein guter Stil ist, ist auch eine Referenzimplementierung eine Form der Spezifikation.

Im Folgenden wird darauf eingegangen, anhand welcher Kriterien man Programmiersprachen unterscheiden kann.

1.1 Abstraktion

Wie nah an den physikalischen Prozessen im Computer ist die Sprache? Wie nah ist sie an einer mathematisch / algorithmischen Beschreibung?

Definition 2

Eine **Maschinensprache** beinhaltet ausschließlich Instruktionen, die direkt von einer CPU ausgeführt werden können. Die Menge dieser Instruktionen sowie deren Syntax wird **Befehlssatz** genannt.

Beispiel 1 (Maschinensprachen)

1) x86

¹Zu finden unter <http://docs.oracle.com/javase/specs/>

2) SPARC

Definition 3 (Assembler)

Eine Assemblersprache ist eine Programmiersprache, deren Befehle dem Befehlssatz eines Prozessor entspricht.

Beispiel 2 (Assembler)

Folgendes Beispiel stammt von https://de.wikibooks.org/wiki/Assembler-Programmierung_für_x86-Prozessoren_Das_erste_Assemblerprogramm:

```
_____ firstp.asm _____  
1  org 100h  
2  start:  
3      mov ax, 5522h  
4      mov cx, 1234h  
5      xchg cx, ax  
6      mov al, 0  
7      mov ah, 4Ch  
8      int 21h
```

Definition 4 (Höhere Programmiersprache)

Eine Programmiersprache heißt *höher*, wenn sie nicht ausschließlich für eine Prozessorarchitektur geschrieben wurde und turing-vollständig ist.

Beispiel 3 (Höhere Programmiersprachen)

Java, Python, Haskell, Ruby, TCL, ...

Definition 5 (Domänenspezifische Sprache)

Eine domänenspezifische Sprache (engl. domain-specific language; kurz DSL) ist eine formale Sprache, die für ein bestimmtes Problemfeld entworfen wurde.

Beispiel 4 (Domänenspezifische Sprache)

1) HTML

2) VHDL

1.2 Paradigmen

Eine weitere Art, wie man Programmiersprachen unterscheiden kann ist das sog. „Programmierparadigma“, also die Art wie man Probleme löst.

Definition 6 (Imperatives Paradigma)

In der *imperativen Programmierung* betrachtet man Programme als eine Folge von Anweisungen, die vorgibt auf welche Art etwas Schritt für Schritt gemacht werden soll.

Beispiel 5 (Imperative Programmierung)

In folgenden Programm erkennt man den imperativen Programmierstil vor allem an den Variablenzuweisungen:

```
int fib(int n) {  
    if (n < 0) {  
        return -1;  
    }  
  
    int fib[2] = {0, 1}, tmp;  
  
    for (; n > 0; n--) {  
        tmp = fib[1];  
        fib[1] = fib[0] + fib[1];  
        fib[0] = tmp;  
    }  
    return fib[0];  
}
```

Definition 7 (Prozedurales Paradigma)

Die prozeduralen Programmierung ist eine Erweiterung des imperativen Programmierparadigmas, bei dem man versucht die Probleme in kleinere Teilprobleme zu zerlegen.

Definition 8 (Funktionales Paradigma)

In der funktionalen Programmierung baut man auf Funktionen und ggf. Funktionen höherer Ordnung, die eine Aufgabe ohne Nebeneffekte lösen.

Beispiel 6 (Funktionale Programmierung)

Der Funktionale Stil kann daran erkannt werden, dass keine Werte zugewiesen werden:

```
fibAkk n n1 n2
  | (n == 0)  = n1
  | (n == 1)  = n2
  | otherwise = fibAkk (n - 1) n2 (n1 + n2)
fib n = fibAkk n 0 1
```

Haskell ist eine funktionale Programmiersprache, C ist eine nicht-funktionale Programmiersprache.

Wichtige Vorteile von funktionalen Programmiersprachen sind:

- Sie sind weitgehend (jedoch nicht vollständig) frei von Seiteneffekten.
- Der Code ist häufig sehr kompakt und manche Probleme lassen sich sehr elegant formulieren.

Definition 9 (Logisches Paradigma)

Das **logische Programmierparadigma** baut auf der formalen Logik auf. Man verwendet **Fakten** und **Regeln** und einen Inferenzalgorithmus um Probleme zu lösen.

Der Inferenzalgorithmus kann z. B. die Unifikation nutzen.

Beispiel 7 (Logische Programmierung)

Obwohl die logische Programmierung für Zahlenfolgen weniger geeignet erscheint, sei hier zur Vollständigkeit das letzte Fibonacci-Beispiel in Prolog:

```
fib(0, A, _, A).
fib(N, A, B, F) :- N1 is N - 1,
                   Sum is A + B,
                   fib(N1, B, Sum, F).
fib(N, F) :- fib(N, 0, 1, F).
```

1.3 Typisierung

Programmiersprachen können anhand der Art ihrer Typisierung unterschieden werden.

Definition 10 (Typisierungsstärke)

Es seien X, Y Programmiersprachen.

X heißt stärker typisiert als Y , wenn X mehr bzw. nützlichere Typen hat als Y .

Beispiel 8 (Typisierungsstärke)

Die Stärke der Typisierung ist abhängig von dem Anwendungsszenario. So hat C im Gegensatz zu Python, Java oder Haskell beispielsweise keine booleschen Datentypen.

Im Gegensatz zu Haskell hat Java keine GADTs².

Definition 11 (Polymorphie)

- a) Ein Typ heißt polymorph, wenn er mindestens einen Parameter hat.
- b) Eine Funktion heißt polymorph, wenn ihr Verhalten nicht von dem konkreten Typen der Parameter abhängt.

Beispiel 9 (Polymorphie)

In Java sind beispielsweise Listen polymorphe Typen:

```
ArrayList<String> l1 = new ArrayList<String>();  
ArrayList<Integer> l2 = new ArrayList<Integer>();
```

Entsprechend sind auf Listen polymorphe Operationen wie `add` und `remove` definiert.

Definition 12 (Statische und dynamische Typisierung)

- a) Eine Programmiersprache heißt **statisch typisiert**, wenn eine Variable niemals ihren Typ ändern kann.

²generalized algebraic data type

- b) Eine Programmiersprache heißt **dynamisch typisiert**, wenn eine Variable ihren Typ ändern kann.

Beispiele für statisch typisierte Sprachen sind C, Haskell und Java. Beispiele für dynamisch typisierte Sprachen sind Python und PHP.

Vorteile statischer Typisierung sind:

- **Performance:** Der Compiler kann mehr Optimierungen vornehmen.
- **Syntaxcheck:** Da der Compiler die Typen zur Compile-Zeit überprüft, gibt es in statisch typisierten Sprachen zur Laufzeit keine Typfehler.

Vorteile dynamischer Typisierung sind:

- Manche Ausdrücke, wie der Y-Combinator in Haskell, lassen sich nicht typisieren.

Der Gedanke bei dynamischer Typisierung ist, dass Variablen keine Typen haben. Nur Werte haben Typen. Man stellt sich also Variablen eher als Beschriftungen für Werte vor. Bei statisch typisierten Sprachen stellt man sich hingegen Variablen als Container vor.

Definition 13 (Explizite und implizite Typisierung)

Sei X eine Programmiersprache.

- a) X heißt **explizit typisiert**, wenn für jede Variable der Typ explizit genannt wird.
- b) X heißt **implizit typisiert**, wenn der Typ einer Variable aus den verwendeten Operationen abgeleitet werden kann.

Sprachen, die implizit typisieren können nutzen dazu Typinferenz.

Beispiele für explizit typisierte Sprachen sind C, C++ und Java. Beispiele für implizit typisierte Sprachen sind JavaScript, Python, PHP und Haskell.

Mir ist kein Beispiel einer Sprache bekannt, die dynamisch und explizit typisiert ist.

Vorteile expliziter Typisierung sind:

- **Lesbarkeit**

Vorteile impliziter Typisierung sind:

- **Tippfreundlicher:** Es ist schneller zu schreiben.
- **Anfängerfreundlicher:** Man muss sich bei einfachen Problemen keine Gedanken um den Typ machen.

Definition 14 (Duck-Typing und strukturelle Typisierung)

- a) Eine Programmiersprache verwendet **Duck-Typing**, wenn die Parameter einer Methode nicht durch die explizite Angabe von Typen festgelegt werden, sondern durch die Art wie die Parameter verwendet werden.
- b) Eine Programmiersprache verwendet **strukturelle Typisierung**, wenn die Parameter einer Methode nicht durch die explizite Angabe von Typen festgelegt werden, sondern explizit durch die Angabe von Methoden.

Strukturelle Typisierung wird auch *typsicheres Duck-Typing* genannt. Der Satz, den man im Zusammenhang mit Duck-Typing immer hört, ist

„When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.“

Beispiel 10 (Strukturelle Typisierung)

Folgende Scala-Methode erwartet ein Objekt, das eine Methode namens `quack` besitzt:

```
def quacker(duck:
    {def quack(value: String): String}) {
    println (duck.quack("like a duck!"))
}
```

```
Diese Funktion ist vom Typ (duck: AnyRefdef quack(value:
String): String)Unit.
```

1.4 Kompilierte und interpretierte Sprachen

Sprachen werden üblicherweise entweder interpretiert oder kompiliert, obwohl es Programmiersprachen gibt, die beides unterstützen.

C und Java werden kompiliert, Python und TCL interpretiert.

1.5 Dies und das

Definition 15 (Seiteneffekt)

Seiteneffekte sind Veränderungen des Zustandes eines Programms.

Manchmal werden Seiteneffekte auch als Nebeneffekt oder Wirkung bezeichnet. Meistens meint man insbesondere unerwünschte oder überraschende Zustandsänderungen.

Definition 16 (Unifikation)

Die Unifikation ist eine Operation in der Logik und dient zur Vereinfachung prädikatenlogischer Ausdrücke. Der Unifikator ist also eine Abbildung, die in einem Schritt dafür sorgt, dass auf beiden Seiten der Gleichung das selbe steht.

Beispiel 11 (Unifikation³)

Gegeben seien die Ausdrücke

$$A_1 = (X, Y, f(b))$$

$$A_2 = (a, b, Z)$$

Großbuchstaben stehen dabei für Variablen und Kleinbuchstaben für atomare Ausdrücke.

Ersetzt man in A_1 nun X durch a , Y durch b und in A_2 die Variable Z durch $f(b)$, so sind sie gleich oder „unifiziert“. Man erhält

$$\sigma(A_1) = (a, b, f(b))$$

$$\sigma(A_2) = (a, b, f(b))$$

mit

$$\sigma = \{X \mapsto a, Y \mapsto b, Z \mapsto f(b)\}$$

Definition 17 (Allgemeinster Unifikator)

Ein Unifikator σ heißt *allgemeinster Unifikator*, wenn es für jeden Unifikator γ eine Substitution δ mit

$$\gamma = \delta \circ \sigma$$

gibt.

Beispiel 12 (Allgemeinster Unifikator⁴)

Sei

$$C = \{ f(a, D) = Y, X = g(b), g(Z) = X \}$$

eine Menge von Gleichungen über Terme.

Dann ist

$$\gamma = [Y \dot{\mapsto} f(a, b), D \dot{\mapsto} b, X \dot{\mapsto} g(b), Z \dot{\mapsto} b]$$

ein Unifikator für C . Jedoch ist

$$\sigma = [Y \dot{\mapsto} f(a, D), X \dot{\mapsto} g(b), Z \dot{\mapsto} b]$$

der allgemeinste Unifikator. Mit

$$\delta = [D \dot{\mapsto} b]$$

gilt $\gamma = \delta \circ \sigma$.

Algorithmus 1 Klassischer Unifikationsalgorithmus

```

function UNIFY(Gleichungsmenge  $C$ )
  if  $C == \emptyset$  then
    return []
  else
    Es sei  $\{ \theta_l = \theta_r \} \cup C' == C$ 
    if  $\theta_l == \theta_r$  then
      UNIFY( $C'$ )
    else if  $\theta_l == Y$  and  $Y \notin FV(\theta_r)$  then
      UNIFY( $[Y \dot{=} \theta_r]C'$ )  $\circ [Y \dot{=} \theta_r]$ 
    else if  $\theta_r == Y$  and  $Y \notin FV(\theta_l)$  then
      UNIFY( $[Y \dot{=} \theta_l]C'$ )  $\circ [Y \dot{=} \theta_l]$ 
    else if  $\theta_l == f(\theta_l^1, \dots, \theta_l^n)$  and  $\theta_r == f(\theta_r^1, \dots, \theta_r^n)$  then
      UNIFY( $C' \cup \{ \theta_l^1 = \theta_r^1, \dots, \theta_l^n = \theta_r^n \}$ )
    else
      fail
  
```

Dieser klassische Algorithmus hat eine Laufzeit von $\mathcal{O}(2^n)$ für folgendes Beispiel:

$$f(X_1, X_2, \dots, X_n) = f(g(X_0, X_0), g(X_1, X_1), \dots, g(X_{n-1}, X_{n-1}))$$

Der *Paterson-Wegman-Unifikationsalgorithmus* ist deutlich effizienter. Er basiert auf dem *Union-Find-Algorithmus* und funktioniert wie folgt:

⁴Folie 268 von Prof. Snelting

⁴[https://de.wikipedia.org/w/index.php?title=Unifikation_\(Logik\)&oldid=116848554#Beispiel](https://de.wikipedia.org/w/index.php?title=Unifikation_(Logik)&oldid=116848554#Beispiel)

Algorithmus 2 Paterson-Wegeman Unifikationsalgorithmus

```
function UNIFY(Knoten  $p$ , Knoten  $q$ )
   $s \leftarrow \text{FIND}(p)$ 
   $t \leftarrow \text{FIND}(q)$ 
  if  $s == t$  oder  $s.\text{GETATOM} == t.\text{GETATOM}$  then
    return True
  if  $s, t$  Knoten für gleichen Funktor, mit Nachfolgern
   $s_1, \dots, s_n$  bzw.  $t_1, \dots, t_n$  then
    UNION( $s, t$ )
     $k \leftarrow 1$ 
     $b \leftarrow \text{True}$ 
    while  $k \leq n$  and  $b$  do
       $b \leftarrow \text{UNIFY}(s_k, t_k)$ 
       $k \leftarrow k + 1$ 
    return True
  if  $s$  oder  $t$  ist Variablen-Knoten then
    UNION( $s, t$ )
    return True
  return False
```

2 Programmiertechniken

2.1 Rekursion

Definition 18 (rekursive Funktion)

Eine Funktion $f : X \rightarrow X$ heißt rekursiv definiert, wenn in der Definition der Funktion die Funktion selbst wieder steht.

Beispiel 13 (rekursive Funktionen)

1) Fibonacci-Funktion:

$$\begin{aligned} fib : \mathbb{N}_0 &\rightarrow \mathbb{N}_0 \\ fib(n) &= \begin{cases} n & \text{falls } n \leq 1 \\ fib(n-1) + fib(n-2) & \text{sonst} \end{cases} \end{aligned}$$

Erzeugt die Zahlen 0, 1, 1, 2, 3, 5, 8, 13, ...

2) Fakultät:

$$\begin{aligned} ! : \mathbb{N}_0 &\rightarrow \mathbb{N}_0 \\ n! &= \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)! & \text{sonst} \end{cases} \end{aligned}$$

3) Binomialkoeffizient:

$$\begin{aligned} \binom{\cdot}{\cdot} : \mathbb{N}_0 \times \mathbb{N}_0 &\rightarrow \mathbb{N}_0 \\ \binom{n}{k} &= \begin{cases} 1 & \text{falls } k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sonst} \end{cases} \end{aligned}$$

Ein Problem von rekursiven Funktionen in Computerprogrammen ist der Speicherbedarf. Für jeden rekursiven Aufruf müssen alle Umgebungsvariablen der aufrufenden Funktion („stack frame“) gespeichert bleiben, bis der rekursive Aufruf beendet ist. Im Fall der Fibonacci-Funktion sieht der Call-Stack in Abb. 2.1 abgebildet.

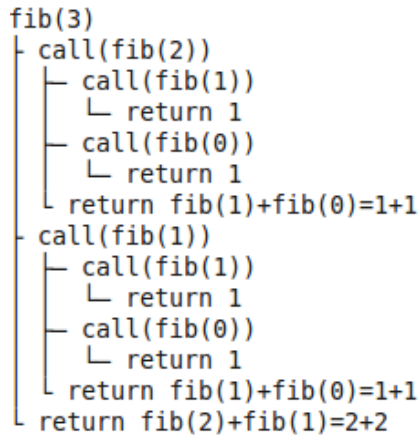


Abbildung 2.1: Call-Stack der Fibonacci-Funktion

Bemerkung 1

Die Anzahl der rekursiven Aufrufe der Fibonacci-Funktion f_C ist:

$$f_C(n) = \begin{cases} 1 & \text{falls } n = 0 \\ 2 \cdot fib(n) - 1 & \text{falls } n \geq 1 \end{cases}$$

Beweis:

- Offensichtlich gilt $f_C(0) = 1$
- Offensichtlich gilt $f_C(1) = 1 = 2 \cdot fib(1) - 1$
- Offensichtlich gilt $f_C(2) = 3 = 2 \cdot fib(2) - 1$
- Für $n \geq 3$:

$$\begin{aligned} f_C(n) &= 1 + f_C(n-1) + f_C(n-2) \\ &= 1 + (2 \cdot fib(n-1) - 1) + (2 \cdot fib(n-2) - 1) \end{aligned}$$

$$\begin{aligned}
&= 2 \cdot (\text{fib}(n-1) + \text{fib}(n-2)) - 1 \\
&= 2 \cdot \text{fib}(n) - 1
\end{aligned}$$

Mit Hilfe der Formel von Moivre-Binet folgt:

$$f_C \in \mathcal{O}\left(\frac{\varphi^n - \psi^n}{\varphi - \psi}\right) \text{ mit } \varphi := \frac{1 + \sqrt{5}}{2} \text{ und } \psi := 1 - \varphi$$

Dabei ist der Speicherbedarf $\mathcal{O}(n)$. Dieser kann durch das Benutzen eines Akkumulators signifikant reduziert werden.

TODO

Definition 19 (linear rekursive Funktion)

Eine Funktion heißt linear rekursiv, wenn in jedem Definitionszweig der Funktion höchstens ein rekursiver Aufruf vorkommt.

Definition 20 (endrekursive Funktion)

Eine Funktion heißt endrekursiv, wenn in jedem Definitionszweig der Rekursive aufruf am Ende des Ausdrucks steht. Der rekursive Aufruf darf also insbesondere nicht in einen anderen Ausdruck eingebettet sein.

Auf Englisch heißen endrekursive Funktionen *tail recursive*.

Beispiel 14 (Linear- und endrekursive Funktionen)

- 1) `fak n = if (n==0) then 1 else (n * fak (n-1))`
ist eine linear rekursive Funktion, aber nicht endrekursiv, da nach der Rückgabe von `fak (n-1)` noch die Multiplikation ausgewertet werden muss.
- 2) `fakAcc n acc = if (n==0) then acc else fakAcc (n-1) (n*acc)`
ist eine endrekursive Funktion.
- 3) `fib n = n <= 1 ? n : fib(n-1) + fib (n-2)`
ist weder linear- noch endrekursiv.

Wenn eine rekursive Funktion nicht terminiert oder wenn

2.2 Backtracking

Unter *Backtracking* versteht man eine Programmiertechnik, die (eventuell implizit) auf einem Suchbaum arbeitet und mittels Tiefensuche versucht eine Lösung zu finden.

Beispiel 15 (Backtracking)

Probleme, bei deren (vollständigen) Lösung Backtracking verwendet wird, sind:

- 1) Damenproblem
- 2) Springerproblem
- 3) Rucksackproblem

2.3 Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, die auf Funktionen arbeiten. Bekannte Beispiele sind:

- `map(function, list)`
map wendet `function` auf jedes einzelne Element aus `list` an.
- `filter(function, list)`
`filter` gibt eine Liste aus Elementen zurück, für die `function` mit `true` evaluiert.
- `reduce(function, list)`
`function` ist für zwei Elemente aus `list` definiert und gibt ein Element des gleichen Typs zurück. Nun steckt `reduce` zuerst zwei Elemente aus `list` in `function`, merkt sich dann das Ergebnis und nimmt so lange weitere Elemente aus `list`, bis jedes Element genommen wurde.
Bei `reduce` ist die Assoziativität wichtig (vgl. Seite 69)

3 Logik

3.1 Prädikatenlogik erster Stufe

Folgendes ist von http://de.wikipedia.org/wiki/Pr%C3%A4dikatenlogik_erster_Stufe

Die Prädikatenlogik erster Stufe ist ein Teilgebiet der mathematischen Logik. Sie befasst sich mit der Struktur gewisser mathematischer Ausdrücke und dem logischen Schließen, mit dem man von derartigen Ausdrücken zu anderen gelangt. Dabei gelingt es, sowohl die Sprache als auch das Schließen rein syntaktisch, das heißt ohne Bezug zu mathematischen Bedeutungen, zu definieren. [...]

Wir beschreiben hier die verwendete Sprache auf rein syntaktische Weise, das heißt wir legen die betrachteten Zeichenketten, die wir Ausdrücke der Sprache nennen wollen, ohne Bezug auf ihre Bedeutung fest.

3.1.1 Symbole

Eine Sprache erster Stufe wird aus folgenden Symbolen aufgebaut:

- $\forall, \exists, \wedge, \vee, \rightarrow, \leftrightarrow, \neg, (,), \equiv$
- sogenannte Variablensymbole v_0, v_1, v_2, \dots ,
- eine (möglicherweise leere) Menge \mathcal{C} von Konstantensymbolen,
- eine (möglicherweise leere) Menge \mathcal{F} von Funktionssymbolen,

- eine (möglicherweise leere) Menge \mathcal{R} von Relationssymbolen.

Das Komma wird hier nur als Trennzeichen für die Aufzählung der Symbole benutzt, es ist nicht Symbol der Sprache.

3.1.2 Terme

Die nach folgenden Regeln aufgebauten Zeichenketten heißen Terme:

- Ist v ein Variablensymbol, so ist v ein Term.
- Ist c ein Konstantensymbol, so ist c ein Term.
- Ist f ein 1-stelliges Funktionssymbol und ist t_1 ein Term, so ist ft_1 ein Term.
- Ist f ein 2-stelliges Funktionssymbol und sind t_1, t_2 Terme, so ist ft_1t_2 ein Term.
- Ist f ein 3-stelliges Funktionssymbol und sind t_1, t_2, t_3 Terme, so ist $ft_1t_2t_3$ ein Term.
- und so weiter für 4,5,6,...-stellige Funktionssymbole.

Ist zum Beispiel c eine Konstante und sind f und g 1- bzw. 2-stellige Funktionssymbole, so ist fgv_2fc ein Term, da er sich durch Anwendung obiger Regeln erstellen lässt: c ist ein Term, daher auch fc ; fc und v_2 sind Terme, daher auch gv_2fc und damit schließlich auch fgv_2fc .

Wir verzichten hier auf Klammern und Kommata als Trennzeichen, das heißt wir schreiben fgv_2fc und nicht $f(g(v_2, f(c)))$. Wir setzen damit implizit voraus, dass unsere Symbole derart beschaffen sind, dass eine eindeutige Lesbarkeit gewährleistet ist.

Die Regeln für die Funktionssymbole fasst man oft so zusammen:

- Ist f ein n -stelliges Funktionssymbol und sind t_1, \dots, t_n Terme, so ist $ft_1 \dots t_n$ ein Term.

Damit ist nichts anderes als die oben angedeutete unendliche Folge von Regeln gemeint, denn die drei Punkte ... gehören nicht zu den vereinbarten Symbolen. Dennoch wird manchmal von dieser Schreibweise Gebrauch gemacht.

Über den Aufbau der Terme lassen sich weitere Eigenschaften definieren. So definieren wir offenbar durch die folgenden drei Regeln rekursiv, welche Variablen in einem Term vorkommen:

- Ist v ein Variablensymbol, so sei $\text{var}(v) = \{v\}$.
- Ist c ein Konstantensymbol, so sei $\text{var}(c) = \emptyset$.
- Ist f ein n -stelliges Funktionssymbol und sind t_1, \dots, t_n Terme, so sei $\text{var}(ft_1 \dots t_n) = \text{var}(t_1) \cup \dots \cup \text{var}(t_n)$.

3.1.3 Ausdrücke

Wir erklären nun durch Bildungsgesetze, welche Zeichenketten wir als Ausdrücke der Sprache ansehen wollen.

Atomare Ausdrücke

- Sind t_1 und t_2 Terme, so ist $t_1 \equiv t_2$ ein Ausdruck.
- Ist R ein 1-stelliges Relationssymbol und ist t_1 ein Term, so ist Rt_1 ein Ausdruck.
- Ist R ein 2-stelliges Relationssymbol und sind t_1, t_2 Terme, so ist Rt_1t_2 ein Ausdruck.
- und so weiter für 3,4,5,...-stellige Relationssymbole.

Dabei gelten die oben zur Schreibweise bei Termen gemachten Bemerkungen.

Zusammengesetzte Ausdrücke

Wir beschreiben hier, wie sich aus Ausdrücken weitere gewinnen lassen.

- Ist φ ein Ausdruck, so ist auch $\neg\varphi$ ein Ausdruck.
- Sind φ und ψ Ausdrücke, so sind auch $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ und $(\varphi \leftrightarrow \psi)$ Ausdrücke.
- Ist φ ein Ausdruck und ist x eine Variable, so sind auch $\forall x\varphi$ und $\exists x\varphi$ Ausdrücke.

Damit sind alle Ausdrücke unserer Sprache festgelegt. Ist zum Beispiel f ein 1-stelliges Funktionssymbol und R ein 2-stelliges Relationssymbol, so ist $\forall v_0((Rv_0v_1 \vee v_0 \equiv fv_1) \rightarrow \exists v_2 \neg Rv_0v_2)$ ein Ausdruck, da er sich durch Anwendung obiger Regeln aufbauen lässt. Es sei noch einmal darauf hingewiesen, dass wir die Ausdrücke mittels der genannten Regeln rein mechanisch erstellen, ohne dass die Ausdrücke zwangsläufig irgendetwas bezeichnen müssten.

3.1.4 1. Stufe

Unterschiedliche Sprachen erster Stufe unterscheiden sich lediglich in den Mengen \mathcal{C} , \mathcal{F} und \mathcal{R} , die man üblicherweise zur Symbolmenge S zusammenfasst und auch die *Signatur* der Sprache nennt. Man spricht dann auch genauer von S -Termen bzw. S -Ausdrücken. Die Sprache, das heißt die Gesamtheit aller nach obigen Regeln gebildeten Ausdrücke, wird mit $L(S)$, L^S oder L_I^S bezeichnet. Bei letzterem steht die römische I für die 1-te Stufe. Dies bezieht sich auf den Umstand, dass gemäß letzter Erzeugungsregel nur über Variable quantifiziert werden kann. L_I^S sieht nicht vor, über alle Teilmengen einer Menge oder über alle Funktionen zu quantifizieren. So lassen sich die üblichen [[Peano-Axiome]] nicht in L_I^S ausdrücken, da das Induktionsaxiom eine Aussage über alle Teilmengen der natürlichen Zahlen macht. Das kann als Schwäche dieser Sprache angesehen werden, allerdings sind die Axiome

der Zermelo-Fraenkel-Mengenlehre sämtlich in der ersten Stufe mit dem einzigen Symbol \in formulierbar, so dass die erste Stufe prinzipiell für die Mathematik ausreicht.

3.1.5 Freie Variablen

Weitere Eigenschaften von Ausdrücken der Sprache L_I^S lassen sich ebenfalls rein syntaktisch definieren. Gemäß dem oben beschriebenen Aufbau durch Bildungsregeln definieren wir die Menge $\text{frei}(\varphi)$ der im Ausdruck φ frei vorkommenden Variablen wie folgt:

- $\text{frei}(t_1 \equiv t_2) = \text{var}(t_1) \cup \text{var}(t_2)$
- $\text{frei}(Rt_1 \dots t_n) = \text{var}(t_1) \cup \dots \cup \text{var}(t_n)$
- $\text{frei}(\neg\varphi) = \text{frei}(\varphi)$
- $\text{frei}(\varphi \wedge \psi) = \text{frei}(\varphi) \cup \text{frei}(\psi)$ und genauso für $\vee, \rightarrow, \leftrightarrow$
- $\text{frei}(\forall x\varphi) = \text{frei}(\varphi) \setminus \{x\}$
- $\text{frei}(\exists x\varphi) = \text{frei}(\varphi) \setminus \{x\}$

Nicht-freie Variable heißen *gebundene Variable*. Ausdrücke φ ohne freie Variable, das heißt solche mit $\text{frei}(\varphi) = \emptyset$, nennt man *Sätze*. Sämtliche in obigem motivierenden Beispiel angegebenen Axiome der geordneten abelschen Gruppen sind bei entsprechender Übersetzung in die Sprache $L_I^{\{0,+, -, \leq\}}$ Sätze, so zum Beispiel $\forall v_0 \forall v_1 + v_0 v_1 \equiv + v_1 v_0$ für das Kommutativgesetz.

3.1.6 Metasprachliche Ausdrücke

Das gerade gegebene Beispiel $\forall v_0 \forall v_1 + v_0 v_1 \equiv + v_1 v_0$ als Symbolisierung des Kommutativgesetzes in der Sprache $L_I^{\{0,+, -, \leq\}}$ zeigt, dass die entstehenden Ausdrücke oft schwer lesbar sind. Daher kehrt der Mathematiker, und oft auch der Logiker, gern zur klassischen Schreibweise $\forall x, y : x + y = y + x$ zurück. Letzteres ist aber

kein Ausdruck der Sprache $L_I^{\{0,+, -, \leq\}}$ sondern nur eine Mitteilung eines solchen Ausdrucks unter Verwendung anderer Symbole einer anderen Sprache, hier der sogenannten [[Metasprache]], das heißt derjenigen Sprache, in der man über $L_I^{\{0,+, -, \leq\}}$ spricht. Aus Gründen der besseren Lesbarkeit lässt man auch gern überflüssige Klammern fort. Das führt nicht zu Problemen, solange klar bleibt, dass man die leichter lesbaren Zeichenketten jederzeit zurückübersetzen könnte.

3.1.7 Substitutionen

Häufig werden in der Mathematik Variablen durch Terme ersetzt. Auch das lässt sich hier rein syntaktisch auf Basis unserer Symbole erklären. Durch folgende Regeln legen wir fest, was es bedeuten soll, den Term t für eine Variable x einzusetzen. Wir folgen dabei wieder dem regelhaften Aufbau von Termen und Ausdrücken. Die Ersetzung wird als $[\]_x^t$ notiert, wobei die eckigen Klammern weggelassen werden dürfen.

Für Terme s wird die Einsetzung s_x^t wie folgt definiert:

- Ist v ein Variablensymbol, so ist v_x^t gleich t falls $v = x$ und v sonst.
- Ist c ein Konstantensymbol, so ist $c_x^t := c$.
- Sind f ein n -stelliges Funktionssymbol und t_1, \dots, t_n Terme, so ist $[ft_1 \dots t_n]_x^t := ft_1^t \dots t_n^t$.

Für Ausdrücke schreiben wir eckige Klammern um den Ausdruck, in dem die Substitution vorgenommen werden soll. Wir legen fest:

- $[t_1 \equiv t_2]_x^t := t_1^t \equiv t_2^t$
- $[Rt_1 \dots t_n]_x^t := Rt_1^t \dots t_n^t$
- $[\neg \varphi]_x^t := \neg[\varphi]_x^t$
- $[(\varphi \vee \psi)]_x^t := ([\varphi]_x^t \vee [\psi]_x^t)$ und genauso für $\wedge, \rightarrow, \leftrightarrow$

- $[\exists x\varphi]_x^t := \exists x\varphi$; analog für den Quantor \forall
- $[\exists y\varphi]_x^t := \exists y[\varphi]_x^t$ falls $x \neq y$ und $y \notin \text{var}(t)$; analog für den Quantor \forall
- $[\exists y\varphi]_x^t := \exists u[\varphi]_{y\ x}^{u\ t}$ falls $x \neq y$ und $y \in \text{var}(t)$, wobei u eine Variable sei, die nicht in φ oder t vorkommt, zum Beispiel die erste der Variablen v_0, v_1, v_2, \dots , die diese Bedingung erfüllt. Die analoge Festlegung wird für \forall getroffen.

Bei dieser Definition wurde darauf geachtet, dass Variablen nicht unbeabsichtigt in den Einflussbereich eines Quantors geraten. Falls die gebundene Variable x im Term auftritt, so wird diese zuvor durch eine andere ersetzt, um so die Variablenkollision zu vermeiden.

Definition 21 (Freie Variable)

Eine Variable, die nicht gebunden ist, heißt frei.

Beispiel 16 (Freie Variablen¹)

In dem Ausdruck $(\lambda x \rightarrow xy)$ ist y eine freie Variable.

Definition 22 (Kombinator)

Ein Kombinator ist eine Funktion oder Definition ohne freie Variablen.

Beispiel 17 (Kombinatoren²)

- 1) $\lambda a \rightarrow a$
- 2) $\lambda a \rightarrow \lambda b \rightarrow a$
- 3) $\lambda f \rightarrow \lambda a \rightarrow \lambda b \rightarrow fba$

¹Quelle: http://www.haskell.org/haskellwiki/Free_variable

²Quelle: <http://www.haskell.org/haskellwiki/Combinator>

4 λ -Kalkül

Der λ -Kalkül (gesprochen: Lambda-Kalkül) ist eine formale Sprache. In diesem Kalkül gibt es drei Arten von Termen T :

- Variablen: x
- Applikationen: (TS)
- Lambda-Abstraktion: $\lambda x.T$

In der Lambda-Abstraktion nennt man den Teil vor dem Punkt die *Parameter* der λ -Funktion. Wenn etwas dannach kommt, auf die die Funktion angewendet wird so heißt dieser Teil das *Argument*:

$$\underbrace{(\lambda \underbrace{x}_{\text{Parameter}}.x^2)}_{\text{Parameter}} \underbrace{5}_{\text{Argument}} = 5^2$$

Beispiel 18 (λ -Funktionen)

- 1) $\lambda x.x$ heißt Identität.
- 2) $(\lambda x.x^2)(\lambda y.y + 3) = \lambda y.(y + 3)^2$
- 3) $(\lambda x.(\lambda y.yx)) ab$
 $\Rightarrow (\lambda y.ya)b$
 $\Rightarrow ba$

In Beispiel 18.3 sieht man, dass λ -Funktionen die Argumente von Links nach rechts einziehen.

Die Funktionsapplikation sei linksassoziativ. Es gilt also:

$$a \ b \ c \ d = ((a \ b) \ c) \ d$$

Definition 23 (Gebundene Variable)

Eine Variable heißt gebunden, wenn sie der Parameter einer λ -Funktion ist.

Definition 24 (Freie Variable)

Eine Variable heißt *frei*, wenn sie nicht gebunden ist.

Satz 4.1

Der untypisierte λ -Kalkül ist Turing-Äquivalent.

4.1 Reduktionen

Definition 25 (Redex)

Eine λ -Term der Form $(\lambda x.t_1)t_2$ heißt Redex.

Definition 26 (α -Äquivalenz)

Zwei Terme T_1, T_2 heißen α -Äquivalent, wenn T_1 durch konsistente Umbenennung in T_2 überführt werden kann.

Man schreibt dann: $T_1 \stackrel{\alpha}{=} T_2$.

Beispiel 19 (α -Äquivalenz)

$$\lambda x.x \stackrel{\alpha}{=} \lambda y.y$$

$$\lambda x.xx \stackrel{\alpha}{=} \lambda y.yy$$

$$\lambda x.(\lambda y.z(\lambda x.zy)y) \stackrel{\alpha}{=} \lambda a.(\lambda x.z(\lambda c.zx)x)$$

Definition 27 (β -Äquivalenz)

Eine β -Reduktion ist die Funktionsanwendung auf einen Redex:

$$(\lambda x.t_1) \ t_2 \Rightarrow t_1[x \mapsto t_2]$$

Beispiel 20 (β -Äquivalenz)

$$\text{a) } (\lambda x. x) y \xRightarrow{\beta} x[x \mapsto y] = y$$

$$\text{b) } (\lambda x. x (\lambda x. x))(y z) \xRightarrow{\beta} (x (\lambda x. x))[x \mapsto y z](y z)(\lambda x. x)$$

Definition 28 (η -Äquivalenz¹)

Die Terme $\lambda x. f x$ und f heißen η -Äquivalent, wenn $x \notin FV(f)$ gilt.

Man schreibt: $\lambda x. f x \stackrel{\eta}{=} f$.

Beispiel 21 (η -Äquivalenz²)

$$\lambda x. \lambda y. f z x y \stackrel{\eta}{=} \lambda x. f z x$$

$$f z \stackrel{\eta}{=} \lambda x. f z x$$

$$\lambda x. x \stackrel{\eta}{=} \lambda x. (\lambda x. x) x$$

$$\lambda x. f x x \stackrel{\eta}{\neq} f x$$

4.2 Auswertungsstrategien

Definition 29 (Normalenreihenfolge)

In der Normalenreihenfolge-Auswertungsstrategie wird der linkeste äußerste Redex ausgewertet.

Definition 30 (Call-By-Name)

In der Call-By-Name Auswertungsreihenfolge wird der linkeste äußerste Redex reduziert, der nicht von einem λ umgeben ist.

Die Call-By-Name Auswertung wird in Funktionen verwendet.

Haskell verwendet die Call-By-Name Auswertungsreihenfolge zusammen mit „sharing“. Dies nennt man *Lazy Evaluation*. Ein Spezialfall der Lazy-Evaluation ist die sog. *Kurzschlussauswertung*. Das bezeichnet die Lazy-Evaluation von booleschen Ausdrücken.

Was ist sharing?

Definition 31 (Call-By-Value)

In der Call-By-Value Auswertung wird der linke Redex reduziert, der nicht von einem λ umgeben ist und dessen Argument ein Wert ist.

Die Call-By-Value Auswertungsreihenfolge wird in C und Java verwendet. Auch in Haskell werden arithmetische Ausdrücke in der Call-By-Name Auswertungsreihenfolge reduziert.

4.3 Church-Zahlen

Im λ -Kalkül lässt sich jeder mathematische Ausdruck darstellen, also insbesondere beispielsweise auch $\lambda x.x + 3$. Aber „3“ und „+“ ist hier noch nicht das λ -Kalkül.

Zuerst müssen wir uns also Gedanken machen, wie man natürliche Zahlen $n \in \mathbb{N}$ darstellt. Dafür dürfen wir nur Variablen und λ verwenden. Eine Möglichkeit das zu machen sind die sog. *Church-Zahlen*.

Dabei ist die Idee, dass die Zahl angibt wie häufig eine Funktion f auf eine Variable z angewendet wird. Also:

- $0 := \lambda f z.z$
- $1 := \lambda f z.fz$
- $2 := \lambda f z.f(fz)$
- $3 := \lambda f z.f(f(fz))$

Auch die gewohnten Operationen lassen sich so darstellen.

Beispiel 22 (Nachfolger-Operation)

$$\text{succ} := \lambda n f z.f(nfz)$$

$$= \lambda n. (\lambda f. (\lambda z. f(n f z)))$$

Dabei ist n die Zahl.

Will man diese Funktion anwenden, sieht das wie folgt aus:

$$\begin{aligned} \text{succ } 1 &= (\lambda n f z. f(n f z)) 1 \\ &= (\lambda n f z. f(n f z)) \underbrace{(\lambda f z. f z)}_n \\ &= \lambda f z. f(\lambda f z. f z) f z \\ &= \lambda f z. f(f z) \\ &= 2 \end{aligned}$$

Beispiel 23 (Vorgänger-Operation)

$$\begin{aligned} \text{pair} &:= \lambda a. \lambda b. \lambda f. f a b \\ \text{fst} &:= \lambda p. p(\lambda a. \lambda b. a) \\ \text{snd} &:= \lambda p. p(\lambda a. \lambda b. b) \\ \text{next} &:= \lambda p. \text{pair}(\text{snd } p) (\text{succ}(\text{snd } p)) \\ \text{pred} &:= \lambda n. \text{fst}(n \text{ next}(\text{pair } c_0 c_0)) \end{aligned}$$

Beispiel 24 (Addition)

$$\text{plus} := \lambda m n f z. m f(n f z)$$

Dabei ist m der erste Summand und n der zweite Summand.

Beispiel 25 (Multiplikation)

$$\begin{aligned} \text{times} &:= \lambda m n f. m \ s \ (n \ f \ z) \\ &\stackrel{\eta}{=} \lambda m n f z. n(m s) z \end{aligned}$$

Dabei ist m der erste Faktor und n der zweite Faktor.

Beispiel 26 (Potenz)

$$\begin{aligned}\text{exp} &:= \lambda b e. e b \\ &\stackrel{\eta}{=} \lambda b e f z. e b f z\end{aligned}$$

Dabei ist b die Basis und e der Exponent.

4.4 Church-Booleans

Definition 32 (Church-Booleans)

True wird zu $c_{\text{true}} := \lambda t. \lambda f. t$.

False wird zu $c_{\text{false}} := \lambda t. \lambda f. f$.

Hiermit lässt sich beispielsweise die Funktion `is_zero` definieren, die True zurückgibt, wenn eine Zahl 0 repräsentiert und sonst False zurückgibt:

$$\text{is_zero} = \lambda n. n (\lambda x. c_{\text{False}}) c_{\text{True}}$$

4.5 Weiteres

Satz 4.2 (Satz von Church-Rosser)

Wenn zwei unterschiedliche Terme a und b äquivalent sind, d.h. mit Reduktionsschritten beliebiger Richtung ineinander transformiert werden können, dann gibt es einen weiteren Term c , zu dem sowohl a als auch b reduziert werden können.

4.6 Fixpunktkombinator

Definition 33 (Fixpunkt)

Sei $f : X \rightarrow Y$ eine Funktion mit $\emptyset \neq A = X \cap Y$ und $a \in A$.

a heißt **Fixpunkt** der Funktion f , wenn $f(a) = a$ gilt.

Beispiel 27 (Fixpunkt)

- 1) $f_1 : \mathbb{R} \rightarrow \mathbb{R}; f(x) = x^2 \Rightarrow x_1 = 0$ ist Fixpunkt von f , da $f(0) = 0$. $x_2 = 1$ ist der einzige weitere Fixpunkt dieser Funktion.
- 2) $f_2 : \mathbb{N} \rightarrow \mathbb{N}$ hat ganz \mathbb{N} als Fixpunkte, also insbesondere unendlich viele Fixpunkte.
- 3) $f_3 : \mathbb{R} \rightarrow \mathbb{R}; f(x) = x + 1$ hat keinen einzigen Fixpunkt.
- 4) $f_4 : \mathbb{R}[X] \rightarrow \mathbb{R}[X]; f(p) = p^2$ hat $p_1(x) = 0$ und $p_2(x) = 1$ als Fixpunkte.

Definition 34 (Kombinator)

Ein Kombinator ist eine Abbildung ohne freie Variablen.

Beispiel 28 (Kombinatoren³)

- 1) $\lambda a. a$
- 2) $\lambda a. \lambda b. a$
- 3) $\lambda f. \lambda a. \lambda b. f \ b \ a$

Definition 35 (Fixpunkt-Kombinator)

Sei f ein Kombinator, der $f \ g = g \ (f \ g)$ erfüllt. Dann heißt f **Fixpunktkombinator**.

Insbesondere ist also $f \ g$ ein Fixpunkt von g .

Definition 36 (Y-Kombinator)

Der Fixpunktkombinator

$$Y := \lambda f. (\lambda x. f \ (x \ x)) \ (\lambda x. f \ (x \ x))$$

heißt Y -Kombinator.

³Quelle: <http://www.haskell.org/haskellwiki/Combinator>

Beh.: Der Y -Kombinator ist ein Fixpunktkombinator.

Beweis: ⁴

Teil 1: Offensichtlich ist Y ein Kombinator.

Teil 2: z. Z.: $Y f \Rightarrow^* f (Y f)$

$$\begin{aligned}
 Y f &= (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) f \\
 &\Rightarrow^\beta (\lambda x. f (x x)) (\lambda x. f (x x)) \\
 &\Rightarrow^\beta f ((\lambda x. f (x x)) (\lambda x. f (x x))) \\
 &\Rightarrow^\beta f (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) f) \\
 &= f (Y f)
 \end{aligned}$$

■

Definition 37 (Turingkombinator)

Der Fixpunktkombinator

$$\Theta := (\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y))$$

heißt **Turingkombinator**.

Beh.: Der Turing-Kombinator Θ ist ein Fixpunktkombinator.

Beweis: ⁵

Teil 1: Offensichtlich ist Θ ein Kombinator.

Teil 2: z. Z.: $\Theta f \Rightarrow^* f (\Theta f)$

Sei $\Theta_0 := (\lambda x. \lambda y. y (x x y))$. Dann gilt:

$$\begin{aligned}
 \Theta f &= ((\lambda x. \lambda y. y (x x y)) \Theta_0) f \\
 &\Rightarrow^\beta (\lambda y. y (\Theta_0 \Theta_0 y)) f \\
 &\Rightarrow^\beta f (\Theta_0 \Theta_0 f) \\
 &= f (\Theta f)
 \end{aligned}$$

■

⁴Quelle: Vorlesung WS 2013/2014, Folie 175

⁵Quelle: Übungsblatt 6, WS 2013/2014

4.7 Let-Polymorphismus

⁶Das Programm $P = \text{let } f = \lambda x. 2 \text{ in } f (f \text{ true})$ ist eine polymorphe Hilfsfunktion, da sie beliebige Werte auf 2 abbildet. Auch solche Ausdrücke sollen typisierbar sein.

Die Kodierung

$$\text{let } x = t_1 \text{ in } t_2$$

ist bedeutungsgleich mit

$$(\lambda x. t_2) t_1$$

Das Problem ist, dass

$$P = \lambda f. f(f \text{ true}) (\lambda x. 2)$$

so nicht typisierbar ist, da in

$$\text{ABS} \frac{f : \tau_f \vdash f (f \text{ true}) : \dots}{\vdash \lambda f. f (f \text{ true}) : \dots}$$

müsste

$$\tau_f = \text{bool} \rightarrow \text{int}$$

und zugleich

$$\tau_f = \text{int} \rightarrow \text{int}$$

in den Typkontext eingetragen werden. Dies ist jedoch nicht möglich. Stattdessen wird

$$\text{let } x = t_1 \text{ in } t_2$$

als neues Konstrukt im λ -Kalkül erlaubt.

Definition 38 (Typschema)

Ein Typ der Gestalt $\forall \alpha_1. \forall \alpha_2. \dots \forall \alpha_n. \tau$ heißt **Typschema**.

Es bindet freie Variablen $\alpha_1, \dots, \alpha_n$ in τ .

⁶WS 2013 / 2014, Folie 205ff

Beispiel 29 (Typschema)

Das Typschema $\forall\alpha. \alpha \rightarrow \alpha$ steht für unendlich viele Typen und insbesondere für folgende:

- 1) $\text{int} \rightarrow \text{int}, \text{bool} \rightarrow \text{bool}, \dots$
- 2) $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}), \dots$
- 3) \dots

Definition 39 (Typschemainstanziierung)

Sei τ_2 ein Nicht-Schema-Typ. Dann heißt der Typ

$$\tau[\alpha \mapsto \tau_2]$$

eine **Instanziierung** vom Typschema $\forall\alpha. \tau$ und man schreibt:

$$(\forall\alpha. \tau) \succeq \tau[\alpha \mapsto \tau_2]$$

Beispiel 30 (Typschemainstanziierung)

Folgendes sind Beispiele für Typschemainstanziierungen:

- 1) $\forall\alpha. \alpha \rightarrow \alpha \succeq \text{int} \rightarrow \text{int}$
- 2) $\forall\alpha. \alpha \rightarrow \alpha \succeq (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$
- 3) $\text{int} \succeq \text{int}$

Folgendes sind keine Typschemainstanziierungen:

- 1) $\alpha \rightarrow \alpha \not\succeq \text{int} \rightarrow \text{int}$
- 2) $\alpha \not\succeq \text{bool}$
- 3) $\forall\alpha. \alpha \rightarrow \alpha \not\succeq \text{bool}$

Zu Typschemata gibt es angepasste Regeln:

$$\text{VAR} \frac{\Gamma(x) = \tau' \quad \tau' \succeq \tau}{\gamma \vdash x : \tau}$$

und

$$\text{ABS} \frac{\Gamma, x : \tau_1 \vdash t : \tau_2 \quad \tau_1 \text{ kein Typschema}}{\Gamma \vdash \lambda x. t : \tau_1 \rightarrow \tau_2}$$

Folie 208ff

4.8 Literatur

- <http://c2.com/cgi/wiki?FreeVariable>
- <http://www.lambda-bound.com/book/lambdacalc/node9.html>

5 Typinferenz

Definition 40 (Datentyp)

Ein *Datentyp* oder kurz *Typ* ist eine Menge von Werten, mit denen eine Bedeutung verbunden ist.

Beispiel 31 (Datentypen)

- `bool` = { `True`, `False` }
- `char` = vgl. Seite 90
- `intHaskell` = $[-2^{29}, 2^{29} - 1] \cap \mathbb{N}$
- `intC90` = $[-2^{15} - 1, 2^{15} - 1] \cap \mathbb{N}^1$
- `float` = siehe IEEE 754
- Funktionstypen, z. B. `int` \rightarrow `int` oder `char` \rightarrow `int`

Hinweis: Typen sind unabhängig von ihrer Repräsentation. So kann ein `bool` durch ein einzelnes Bit repräsentiert werden oder eine Bitfolge zugrunde liegen.

Auf Typen sind Operationen definiert. So kann man auf numerischen Typen eine Addition (+), eine Subtraktion (-), eine Multiplikation (*) und eine Division (/) definieren.

Ich schreibe hier bewusst „eine“ Multiplikation und nicht „die“ Multiplikation, da es verschiedene Möglichkeiten gibt auf Gleitpunktzahlen Multiplikationen zu definieren. So kann man beispielsweise die Assoziativität unterschiedlich wählen.

Beispiel 32 (Multiplikation ist nicht assoziativ)

In Python 3 ist die Multiplikation linksassoziativ. Also:

¹siehe ISO/IEC 9899:TC2, Kapitel 7.10: Sizes of integer types <limits.h>

```
>>> 0.1*0.1*0.3
0.00300000000000000005
>>> (0.1*0.1)*0.3
0.00300000000000000005
>>> 0.1*(0.1*0.3)
0.003
```

Definition 41 (Typvariable)

Eine Typvariable repräsentiert einen Typen.

Hinweis: Üblicherweise werden kleine griechische Buchstaben $(\alpha, \beta, \tau_1, \tau_2, \dots)$ als Typvariablen gewählt.

Genau wie Typen bestimmte Operationen haben, die auf ihnen definiert sind, kann man sagen, dass Operationen bestimmte Typen, auf die diese Anwendbar sind. So ist

$$\alpha + \beta$$

für numerische α und β wohldefiniert, auch wenn α und β boolesch sind oder beides Strings sind könnte das Sinn machen. Es macht jedoch z. B. keinen Sinn, wenn α ein String ist und β boolesch.

Die Menge aller Operationen, die auf die Variablen angewendet werden, nennt man **Typkontext**. Dieser wird üblicherweise mit Γ bezeichnet.

Das Ableiten einer Typisierung für einen Ausdruck nennt man **Typinferenz**. Man schreibt: $\vdash (\lambda x.2) : \alpha \rightarrow \text{int}$.

Bei solchen Ableitungen sind häufig viele Typen möglich. So kann der Ausdruck

$$\lambda x.2$$

Mit folgenderweise typisiert werden:

- $\vdash (\lambda x.2) : \text{bool} \rightarrow \text{int}$
- $\vdash (\lambda x.2) : \text{int} \rightarrow \text{int}$

- $\vdash (\lambda x.2) : \text{Char} \rightarrow \text{int}$
- $\vdash (\lambda x.2) : \alpha \rightarrow \text{int}$

In der letzten Typisierung stellt α einen beliebigen Typen dar.

Definition 42 (Typsystem $\Gamma \vdash t : T$)

Ein Typkontext Γ ordnet jeder freien Variable x einen Typ $\Gamma(x)$ durch folgende Regeln zu:

$$\text{CONST} : \frac{c \in \text{Const}}{\Gamma \vdash c : \tau_c}$$

$$\text{VAR} : \frac{\Gamma(x) = \tau}{\Gamma \vdash c : \tau}$$

$$\text{ABS} : \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$$

$$\text{APP} : \frac{\Gamma \vdash t_1, \tau_2 \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 t_2 : \tau}$$

Dabei ist der lange Strich kein Bruchstrich, sondern ein Symbol der Logik das als **Schlussstrich** bezeichnet wird. Dabei ist der Zähler als Voraussetzung und der Nenner als Schlussfolgerung zu verstehen.

Definition 43 (Typsubstitution)

Eine *Typsubstitution* ist eine endliche Abbildung von Typvariablen auf Typen.

Für eine Menge von Typsubstitutionen wird üblicherweise σ als Symbol verwendet. Man schreibt also beispielsweise:

$$\sigma = [\alpha_1 \mapsto \text{bool}, \alpha_2 \mapsto \alpha_1 \rightarrow \alpha_1]$$

Definition 44 (Lösung eines Typkontextes)

Sei t eine beliebige freie Variable, $\tau = \tau(t)$ ein beliebiger Typ
 σ eine Menge von Typsubstitutionen und Γ ein Typkontext.

(σ, τ) heißt eine Lösung für (Γ, t) , falls gilt:

$$\sigma\Gamma \vdash t : \tau$$

5.1 Beispiele

Im Folgenden wird die Typinferenz für einige λ -Funktionen durchgeführt.

5.1.1 $\lambda x. \lambda y. x y^2$

Gesucht ist ein Typ τ , sodass sich $\vdash \lambda x. \lambda y. x y : \tau$ mit einem Ableitungsbaum nachweisen lässt. Es gibt mehrere solche τ , aber wir suchen das allgemeinste. Die Regeln unseres Typsystems (siehe Seite 41) sind *syntaxgerichtet*, d. h. zu jedem λ -(Teil)-Term gibt es genau eine passende Regel.

Für $\lambda x. \lambda y. x y$ wissen wir also schon, dass jeder Ableitungsbaum von folgender Gestalt ist. Dabei sind α_i Platzhalter:

$$\text{ABS} \frac{\text{ABS} \frac{\text{VAR} \frac{(x:\alpha_2, y:\alpha_4) (x)=\alpha_6}{x:\alpha_2, y:\alpha_4 \vdash x:\alpha_6} \quad \text{VAR} \frac{(x:\alpha_2, y:\alpha_4) (y)=\alpha_7}{x:\alpha_2, y:\alpha_4 \vdash y:\alpha_7}}{x : \alpha_2, y : \alpha_4 \vdash x y : \alpha_5}}{x:\alpha_2 \vdash \lambda y. x y : \alpha_3}}{\vdash \lambda x. \lambda y. x y : \alpha_1}$$

Das was wir haben wollen steht am Ende, also unter dem unterstem Schlussstrich. Dann bedeutet die letzte Zeile

$$\vdash \lambda x. \lambda y. x y : \alpha_1$$

²Lösung von Übungsblatt 6, WS 2013 / 2014

Ohne (weitere) Voraussetzungen lässt sich sagen, dass der Term

$$\lambda x. \lambda y. x y$$

vom Typ α_1 ist.

Links der Schlussstriche steht jeweils die Regel, die wir anwenden. Also entweder ABS, VAR, CONST oder APP.

Nun gehen wir eine Zeile höher:

$$x : \alpha_2 \vdash \lambda y. x y : \alpha_3$$

Diese Zeile ist so zu lesen: Mit der Voraussetzung, dass x vom Typ α_2 ist, lässt sich syntaktisch Folgern, dass der Term $\lambda y. x y$ vom Typ α_3 ist.

Hinweis: Alles was in Zeile i dem \vdash steht, steht auch in jedem „Nenner“ in Zeile $j < i$ vor jedem einzelnen \vdash .

Folgende Typgleichungen C lassen sich aus dem Ableitungsbaum ablesen:

$$\begin{aligned} C = & \{ \alpha_1 = \alpha_2 \rightarrow \alpha_3 \} \\ & \cup \{ \alpha_3 = \alpha_4 \rightarrow \alpha_5 \} \\ & \cup \{ \alpha_6 = \alpha_7 \rightarrow \alpha_5 \} \\ & \cup \{ \alpha_6 = \alpha_2 \} \\ & \cup \{ \alpha_7 = \alpha_4 \} \end{aligned}$$

Diese Bedingungen (engl. *Constraints*) haben eine allgemeinste Lösung mit einem allgemeinsten Unifikator σ_C :

$$\sigma_C = [\alpha_1 \dot{\rightarrow} (\alpha_4 \rightarrow \alpha_5) \rightarrow \alpha_4 \rightarrow \alpha_5,$$

$$\begin{aligned}
&\alpha_2 \dot{\rightarrow} \alpha_4 \rightarrow \alpha_5, \\
&\alpha_3 \dot{\rightarrow} \alpha_4 \rightarrow \alpha_5, \\
&\alpha_6 \dot{\rightarrow} \alpha_4 \rightarrow \alpha_5, \\
&\alpha_7 \dot{\rightarrow} \alpha_4]
\end{aligned}$$

Hinweis: Es gilt $(\alpha_4 \rightarrow \alpha_5) \rightarrow \alpha_4 \rightarrow \alpha_5 = (\alpha_4 \rightarrow \alpha_5) \rightarrow (\alpha_4 \rightarrow \alpha_5)$

Also gilt: Der allgemeinste Typ von $\lambda x. \lambda y. x \ y$ ist $\sigma_C(\alpha_1) = (\alpha_4 \rightarrow \alpha_5) \rightarrow \alpha_4 \rightarrow \alpha_5$.

5.1.2 Selbstapplikation³

Im Folgenden wird eine Typinferenz für die Selbstapplikation, also

$$\lambda x. x \ x$$

durchgeführt.

Zuerst erstellt man den Ableitungsbaum:

$$\text{ABS} \frac{\text{APP} \frac{\text{VAR} \frac{(x:\alpha_2) \ (x)=\alpha_5}{x:\alpha_2 \vdash x:\alpha_5} \quad \text{VAR} \frac{(x:\alpha_2) \ (x)=\alpha_4}{x:\alpha_2 \vdash x:\alpha_4}}{x:\alpha_2 \vdash x \ x : \alpha_3}}{\vdash \lambda x. x \ x : \alpha_1}$$

Dies ergibt die Constraint-Menge

$$C = \{ \alpha_1 = \alpha_2 \rightarrow \alpha_3 \} \quad \text{ABS-Regel} \quad (5.1)$$

$$\cup \{ \alpha_5 = \alpha_4 \rightarrow \alpha_3 \} \quad \text{APP-Regel} \quad (5.2)$$

$$\cup \{ \alpha_5 = \alpha_2 \} \quad \text{Linke VAR-Regel} \quad (5.3)$$

$$\cup \{ \alpha_4 = \alpha_2 \} \quad \text{Rechte VAR-Regel} \quad (5.4)$$

Aus Gleichung (5.3) und Gleichung (5.4) folgt:

$$\alpha_2 = \alpha_4 = \alpha_5$$

³Lösung von Übungsblatt 6, WS 2013 / 2014

Also lässt sich Gleichung (5.2) umformulieren:

$$\alpha_2 = \alpha_2 \rightarrow \alpha_3$$

Offensichtlich ist diese Bedingung nicht erfüllbar. Daher ist die Selbstapplikation nicht typisierbar. Dies würde im Unifikationsalgorithmus (vgl. Algorithmus 1) durch den *occur check* festgestellt werden.

6 Parallelität

Systeme mit mehreren Prozessoren sind heutzutage weit verbreitet. Inzwischen sind sowohl in Desktop-PCs als auch Laptops, Tablets und Smartphones „Multicore-CPUs“ verbaut. Daher sollten auch Programmierer in der Lage sein, Programme für mehrere Kerne zu entwickeln.

Parallelverarbeitung kann auf mehreren Ebenen statt finden:

- **Bit-Ebene:** Werden auf 32-Bit Computern `long long`, also 64-Bit Zahlen, addiert, so werden parallel zwei 32-Bit Additionen durchgeführt und das carry-flag benutzt.
- **Anweisungs-Ebene:** Die Ausführung von Anweisungen in der CPU besteht aus mehreren Phasen (Instruction Fetch, Decode, Execution, Write-Back). Besteht zwischen aufeinanderfolgenden Anweisungen keine Abhängigkeit, so kann der Instruction Fetch-Teil einer zweiten Anweisung parallel zum Decode-Teil einer ersten Anweisung geschehen. Das nennt man Pipelining. Man spricht hier auch von *Instruction Level Parallelism* (ILP)
- **Datenebene:** Es kommt immer wieder vor, dass man in Schleifen eine Operation für jedes Objekt eines Containers (z. B. einer Liste) durchführen muss. Zwischen den Anweisungen verschiedener Schleifendurchläufe besteht dann eventuell keine Abhängigkeit. Dann können alle Schleifenaufrufe parallel durchgeführt werden.
- **Verarbeitungsebene:** Verschiedene Programme sind unabhängig von einander.

Gerade bei dem letzten Punkt ist zu beachten, dass echt parallele Ausführung nicht mit *verzahnter Ausführung* zu verwechseln ist. Auch bei Systemen mit nur einer CPU und einem Kern kann man gleichzeitig den Browser nutzen und einen Film über eine Multimedia-Anwendung laufen lassen. Dabei wechselt der Scheduler sehr schnell zwischen den verschiedenen Anwendungen, sodass es sich so anfühlt, als würden die Programme echt parallel ausgeführt werden.

Weitere Informationen zu Pipelining gibt es in der Vorlesung „Rechnerorganisation“ bzw. „Digitaltechnik und Entwurfsverfahren“ (zu der auch ein exzellentes Skript angeboten wird). Informationen über Scheduling werden in der Vorlesung „Betriebssysteme“ vermittelt.

6.1 Architekturen

Es gibt zwei Ansätze, wie man Parallelrechner entwickeln kann:

- **Gemeinsamer Speicher:** In diesem Fall kann jeder Prozessor jede Speicherzelle ansprechen. Dies ist bei Multicore-CPUs der Fall.
- **Verteilter Speicher:** Es ist auch möglich, dass jeder Prozessor seinen eigenen Speicher hat, der nur ihm zugänglich ist. In diesem Fall schicken die Prozessoren Nachrichten (engl. *message passing*). Diese Technik wird in Clustern eingesetzt.

Eine weitere Art, wie man Parallelverarbeitung klassifizieren kann, ist anhand der verwendeten Architektur. Der der üblichen, sequentiellen Art der Programmierung, bei der jeder Befehl nach einander ausgeführt wird, liegt die sog. **Von-Neumann-Architektur** zugrunde. Bei der Programmierung von parallel laufenden Anwendungen kann man das **PRAM-Modell** (kurz für *Parallel Random Access Machine*) zugrunde legen. In diesem Modell geht man von ei-

ner beliebigen Anzahl an Prozessoren aus, die über lokalen Speicher verfügen und synchronen Zugriff auf einen gemeinsamen Speicher haben.

Anhand der **Flynn'schen Klassifikation** können Rechnerarchitekturen in vier Kategorien unterteilt werden:

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMI	MIMD

Dabei wird die Von-Neumann-Architektur als *SISD-Architektur* und die PRAM-Architektur als *SIMD-Architektur* klassifiziert. Es ist so zu verstehen, dass ein einzelner Befehl auf verschiedene Daten angewendet wird.

Bei heutigen Multicore-Rechnern liegt MIMD vor.

MISD ist nicht so richtig sinnvoll.

Definition 45 (Nick's Class)

Nick's Class (in Zeichen: \mathcal{NC}) ist die Klasse aller Probleme, die im PRAM-Modell in logarithmischer Laufzeit lösbar sind.

Beispiel 33 (Nick's Class)

Folgende Probleme sind in \mathcal{NC} :

- 1) Die Addition, Multiplikation und Division von Ganzzahlen,
- 2) Matrixmultiplikation, die Berechnung von Determinanten und Inversen,
- 3) ausschließlich Probleme aus \mathcal{P} , also: $\mathcal{NC} \subseteq \mathcal{P}$

Es ist nicht klar, ob $\mathcal{P} \subseteq \mathcal{NC}$ gilt. Bisher wurde also noch kein Problem $P \in \mathcal{P}$ gefunden mit $P \notin \mathcal{NC}$.

6.2 Prozesskommunikation

Die Prozesskommunikation wird durch einige Probleme erschwert:

Definition 46 (Wettlaufsituation)

Ist das Ergebnis einer Operation vom zeitlichen Ablauf der Einzeloperationen abhängig, so liegt eine Wettlaufsituation vor.

Beispiel 34 (Wettlaufsituation)

Angenommen, man hat ein Bankkonto mit einem Stand von 2000 Euro. Auf dieses Konto wird am Monatsende ein Gehalt von 800 Euro eingezahlt und die Miete von 600 Euro abgeboben. Nun stelle man sich folgende beiden Szenarien vor:

t	Prozess 1: Lohn	Prozess 2: Miete	Kontostand
1	Lade Kontostand	Lade Kontostand	2000
2	Addiere Lohn		2000
3	Speichere Kontostand		2800
4		Subtrahiere Miete	2800
5		Speichere Kontostand	1400

Dieses Problem existiert nicht nur bei echt parallelen Anwendungen, sondern auch bei zeitlich verzahnten Anwendungen.

Definition 47 (Semaphore)

Eine Semaphore $S = (c, r, f, L)$ ist eine Datenstruktur, die aus einer Ganzzahl, den beiden atomaren Operationen $r =$ „reservieren probieren“ und $f =$ „freigeben“ sowie einer Liste L besteht.

r gibt entweder Wahr oder Falsch zurück um zu zeigen, ob das reservieren erfolgreich war. Im Erfolgsfall wird c um 1 verringert. Es wird genau dann Wahr zurück gegeben, wenn c positiv ist. Wenn Wahr zurückgegeben wird, dann wird das aufrufende Objekt der Liste hinzugefügt.

f kann nur von Objekten aufgerufen werden, die in L sind. Wird f von $o \in L$ aufgerufen, wird o aus L entfernt und c um

eins erhöht.

Semaphoren können eingesetzt werden um Wettlaufsituationen zu verhindern.

Definition 48 (Monitor)

Ein Monitor $M = (m, c)$ ist ein Tupel, wobei m ein Mutex und c eine Bedingung ist.

Monitore können mit einer Semaphore, bei der $c = 1$ ist, implementiert werden. Monitore sorgen dafür, dass auf die Methoden der Objekte, die sie repräsentieren, zu jedem Zeitpunkt nur einmal ausgeführt werden können. Sie sorgen also für *gegenseitigen Ausschluss*.

Beispiel 35 (Monitor)

Folgendes Beispiel von [https://en.wikipedia.org/w/index.php?title=Monitor_\(synchronization\)&oldid=596007585](https://en.wikipedia.org/w/index.php?title=Monitor_(synchronization)&oldid=596007585) verdeutlicht den Nutzen eines Monitors:

```
monitor class Account {
    private int balance := 0
    invariant balance >= 0

    public method boolean withdraw(int amount)
        precondition amount >= 0
    {
        if balance < amount:
            return false
        else:
            balance := balance - amount
            return true
    }

    public method deposit(int amount)
        precondition amount >= 0
    {
        balance := balance + amount
    }
}
```

```
    }  
}
```

6.3 Parallelität in Java

Java unterstützt mit der Klasse `Thread` und dem Interface `Runnable` Parallelität.

Interessante Stichwörter sind noch:

- `ThreadPool`
- `Interface Executor`
- `Interface Future<V>`
- `Interface Callable<V>`

6.4 Message Passing Modell

Das Message Passing Modell ist eine Art, wie man parallel laufende Programme schreiben kann. Dabei tauschen die Prozesse Nachrichten aus um die Arbeit zu verteilen.

Ein wichtiges Konzept ist hierbei der *Kommunikator*. Ein Kommunikator definiert eine Gruppe von Prozessen, die mit einander kommunizieren können. In dieser Gruppe von Prozessen hat jeder Prozesse einen eindeutigen *Rang*, den sie zur Kommunikation nutzen.

Die Grundlage der Kommunikation bilden *send* und *receive* Operationen. Prozesse schicken Nachrichten an andere Prozesse, indem sie den eindeutigen Rang und einen *tag* angeben, der die Nachricht identifiziert.

Wenn ein Prozess mit einem einzigen weiteren Prozess kommuniziert, wird dies *Punkt-zu-Punkt-Kommunikation* genannt.

Wenn ein Prozess allen anderen eine Nachricht schickt, nennt man das *Broadcast*.

7 Haskell

Haskell ist eine funktionale Programmiersprache, die von Haskell Brooks Curry entwickelt und 1990 in Version 1.0 veröffentlicht wurde.

Wichtige Konzepte sind:

1. Funktionen höherer Ordnung
2. anonyme Funktionen (sog. Lambda-Funktionen)
3. Pattern Matching
4. Unterversorgung
5. Typinferenz

Haskell kann mit „Glasgow Haskell Compiler“ mittels `ghci` interpretiert und mittels

7.1 Erste Schritte

Haskell kann unter www.haskell.org/platform/ für alle Plattformen heruntergeladen werden. Unter Debian-Systemen ist das Paket `ghc` bzw. `haskell-platform` relevant.

7.1.1 Hello World

Speichere folgenden Quelltext als `hello-world.hs`:

```
1 main = putStrLn "Hello, World!"
```

Kompiliere ihn mit `ghc -o hello hello-world.hs`. Es wird eine ausführbare Datei erzeugt.

Alternativ kann es direkt mit `runghc hello-world.hs` ausgeführt werden.

7.2 Syntax

7.2.1 Klammern und Funktionsdeklaration

Haskell verzichtet an vielen Stellen auf Klammern. So werden im Folgenden die Funktionen $f(x) := \frac{\sin x}{x}$ und $g(x) := x \cdot f(x^2)$ definiert:

```
f :: Floating a => a -> a
f x = sin x / x
```

```
g :: Floating a => a -> a
g x = x * (f (x*x))
```

Die Funktionsdeklarationen mit den Typen sind nicht notwendig, da die Typen aus den benutzten Funktionen abgeleitet werden.

Zu lesen ist die Deklaration wie folgt:

```
[Funktionsname] :: [Typendefinitionen] =>
                    Signatur
```

T. Def. Die Funktion `f` benutzt als Parameter bzw. Rückgabewert einen Typen. Diesen Typen nennen wir `a` und er ist vom Typ `Floating`. Auch `b`, wasweisich oder etwas ähnliches wäre ok.

Signatur Die Signatur liest man am einfachsten von hinten:

- f bildet auf einen Wert vom Typ a ab und
- f hat genau einen Parameter a

Gibt es Funktionsdeklarationen, die bis auf Wechsel des Namens und der Reihenfolge äquivalent sind?

7.2.2 if / else

Das folgende Beispiel definiert den Binomialkoeffizienten (vgl. Beispiel 13.3)

```
binom :: (Eq a, Num a, Num a1) => a -> a -> a1
binom n k =
    if (k==0) || (k==n)
    then 1
    else binom (n-1) (k-1) + binom (n-1) k
```

Das könnte man auch mit sog. Guards machen:

```
binom :: (Eq a, Num a, Num a1) => a -> a -> a1
binom n k
    | (k==0) || (k==n) = 1
    | otherwise       = binom (n-1) (k-1)
                      + binom (n-1) k
```

7.2.3 Rekursion

Die Fakultätsfunktion wurde wie folgt implementiert:

$$fak(n) := \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot fak(n) & \text{sonst} \end{cases}$$

```
fak :: (Eq a, Num a) => a -> a
fak n = if (n==0) then 1 else n * fak (n-1)
```

Diese Implementierung benötigt $\mathcal{O}(n)$ rekursive Aufrufe und hat einen Speicherverbrauch von $\mathcal{O}(n)$. Durch einen **Akkumulator** kann dies verhindert werden:

```
fakAcc :: (Eq a, Num a) => a -> a -> a
fakAcc n acc = if (n==0)
                then acc
                else fakAcc (n-1) (n*acc)

fak :: (Eq a, Num a) => a -> a
fak n = fakAcc n 1
```

7.2.4 Listen

- `[]` erzeugt die leere Liste,
- `[1,2,3]` erzeugt eine Liste mit den Elementen 1,2,3
- `:` wird **cons** genannt und ist der Listenkonstruktor.
- `list !! i` gibt das *i*-te Element von `list` zurück.
- `head list` gibt den Kopf von `list` zurück, `tail list` den Rest:

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> tail []
*** Exception: Prelude.tail: empty list
Prelude> tail [1]
[]
Prelude> head [1]
1
Prelude> null []
True
Prelude> null [[]]
False
```

- `last [1,9,1,3]` gibt 3 zurück.
- `length list` gibt die Anzahl der Elemente in `list` zurück.
- `maximum [1,9,1,3]` gibt 9 zurück (analog: `minimum`).
- `null list` prüft, ob `list` leer ist.
- `take 3 [1,2,3,4,5]` gibt `[1,2,3]` zurück.
- `drop 3 [1,2,3,4,5]` gibt `[4,5]` zurück.
- `reverse [1,9,1,3]` gibt `[3,1,9,1]` zurück.
- `elem item list` gibt zurück, ob sich `item` in `list` befindet.

Beispiel in der interaktiven Konsole

```
Prelude> let mylist = [1,2,3,4,5,6]
Prelude> head mylist
1
Prelude> tail mylist
[2,3,4,5,6]
Prelude> take 3 mylist
[1,2,3]
Prelude> drop 2 mylist
[3,4,5,6]
Prelude> mylist
[1,2,3,4,5,6]
Prelude> mylist ++ sndList
[1,2,3,4,5,6,9,8,7]
```

List-Comprehensions

List-Comprehensions sind kurzschreibweisen für Listen, die sich an der Mengenschreibweise in der Mathematik orientieren. So

entspricht die Menge

$$\begin{aligned} myList &= \{ 1, 2, 3, 4, 5, 6 \} \\ test &= \{ x \in myList \mid x > 2 \} \end{aligned}$$

in etwa folgendem Haskell-Code:

```
Prelude> let mylist = [1,2,3,4,5,6]
Prelude> let test = [x | x <- mylist, x>2]
Prelude> test
[3,4,5,6]
```

Beispiel 36 (List-Comprehension)

Das folgende Beispiel zeigt, wie man mit List-Comprehensions die unendliche Liste aller pythagoreischen Tripels erstellen kann:

```
triples :: [(Integer, Integer, Integer)]
triples = [(x,y,z) | z <- [1..],
                    x <- [1..z],
                    y <- [1..z],
                    z^2 == x^2 + y^2
                    ]
```

7.2.5 Strings

- Strings sind Listen von Zeichen:
tail "ABCDEF" gibt "BCDEF" zurück.

7.2.6 Let und where

```
>>> let f = 3; g = f where f = 7
>>> f
3
>>> g
7
```

7.3 Typen

7.3.1 Standard-Typen

Haskell kennt einige Basis-Typen:

- **Int**: Ganze Zahlen. Der Zahlenbereich kann je nach Implementierung variieren, aber der Haskell-Standard garantiert, dass das Intervall $[-2^{29}, 2^{29} - 1]$ abgedeckt wird.
- **Integer**: beliebig große ganze Zahlen
- **Float**: Fließkommazahlen
- **Double**: Fließkommazahlen mit doppelter Präzision
- **Bool**: Wahrheitswerte
- **Char**: Unicode-Zeichen

Des weiteren gibt es einige strukturierte Typen:

- Listen: z. B. $[1, 2, 3]$
- Tupel: z. B. $(1, 'a', 2)$
- Brüche (Fractional, RealFrac)
- Summen-Typen: Typen mit mehreren möglichen Repräsentationen

7.3.2 Typinferenz

In Haskell werden Typen aus den Operationen geschlossenfolgert. Dieses Schlussfolgern der Typen, die nicht explizit angegeben werden müssen, nennt man **Typinferenz**.

Haskell kennt die Typen aus Abb. 7.1.

Ein paar Beispiele zur Typinferenz:

```
Prelude> let x = \x -> x*x
```

```
Prelude> :t x
```

```
x :: Integer -> Integer
```

```
Prelude> x(2)
```

```
4
```

```
Prelude> x(2.2)
```

```
<interactive>:6:3:
```

```
No instance for (Fractional Integer)
```

```
arising from the literal '2.2'
```

```
Possible fix: add an instance declaration for  
(Fractional Integer)
```

```
In the first argument of 'x', namely '(2.2)'
```

```
In the expression: x (2.2)
```

```
In an equation for 'it': it = x (2.2)
```

```
Prelude> let mult = \x y->x*y
```

```
Prelude> mult(2,5)
```

```
<interactive>:9:5:
```

```
Couldn't match expected type 'Integer' with  
actual type '(t0, t1)'
```

```
In the first argument of 'mult', namely '(2, 5)'
```

```
In the expression: mult (2, 5)
```

```
In an equation for 'it': it = mult (2, 5)
```

```
Prelude> mult 2 5
```

```
10
```

```
Prelude> :t mult
```

```
mult :: Integer -> Integer -> Integer
```

```
Prelude> let concat = \x y -> x ++ y
```

```
Prelude> concat [1,2,3] [3,2,1]
```

```
[1,2,3,3,2,1]
```

```
Prelude> :t concat
```

```
concat :: [a] -> [a] -> [a]
```

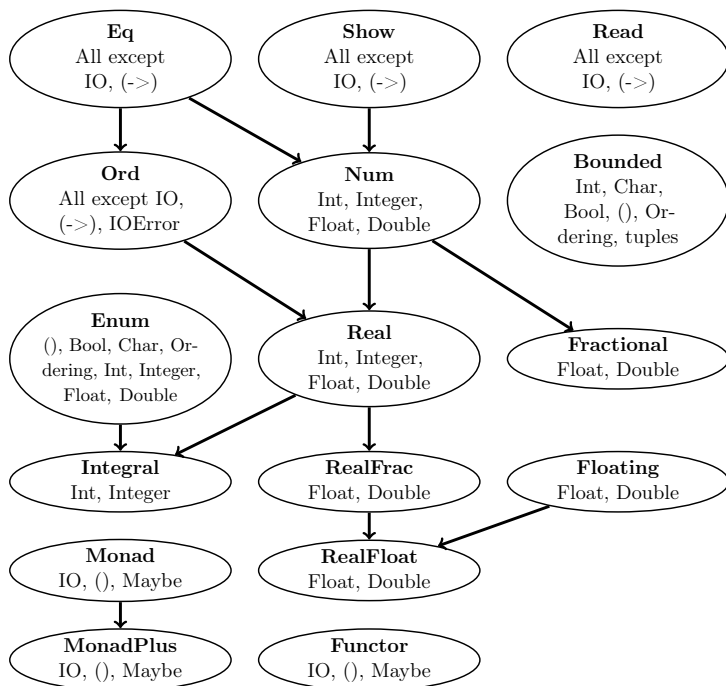


Abbildung 7.1: Hierarchie der Haskell Standardklassen

7.3.3 type

Mit `type` können Typsynonyme erstellt werden:

```
type Prenom = String
type Age = Double
type Person = (Prenom, Age)
type Friends = [Person]
type Polynom = [Double]
```

7.3.4 data

Mit dem Schlüsselwort `data` können algebraische Datentypen erzeugt werden:

```
data Bool    = False | True
data Color   = Red | Green | Blue | Indigo | Violet
data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Point   = Point Float Float deriving (Show)
data Tree t  = Node t  [Tree t]
```

7.4 Lazy Evaluation

Haskell wertet Ausdrücke nur aus, wenn es nötig ist.

Beispiel 37 (Lazy Evaluation)

Obwohl der folgende Ausdruck einen Teilausdruck hat, der einen Fehler zurückgeben würde, kann er aufgrund der Lazy Evaluation zu 2 evaluiert werden:

```
_____ lazy-evaluation.hs _____
1 g a b c
2   | c > 0      = b
3   | otherwise = a
4
```



```
5 main = do
6   print (g (1/0) 2 3)
```

7.5 Beispiele

7.5.1 Quicksort

```
----- qsort.hs -----
1 qsort []      = []
2 qsort (p:ps) = (qsort (filter (\x -> x<=p) ps))
3               ++ p:(qsort (filter (\x -> x> p) ps))
```

- Die leere Liste ergibt sortiert die leere Liste.
- Wähle das erste Element p als Pivotelement und teile die restliche Liste ps in kleinere und gleiche sowie in größere Elemente mit `filter` auf. Konkateniere diese beiden Listen mit `++`.

Durch das Ausnutzen von Unterversorgung lässt sich das ganze sogar noch kürzer schreiben:

```
----- qsort.hs -----
1 qsort []      = []
2 qsort (p:ps) = (qsort (filter (<=p) ps))
3               ++ p:(qsort (filter (> p) ps))
```

7.5.2 Fibonacci

```
----- fibonacci.hs -----
1 fib n
2   | (n == 0) = 0
3   | (n == 1) = 1
4   | otherwise = fib (n - 1) + fib (n - 2)
```

```

1 fibAkk n n1 n2
2   | (n == 0)  = n1
3   | (n == 1)  = n2
4   | otherwise = fibAkk (n - 1) n2 (n1 + n2)
5 fib n = fibAkk n 0 1

```

```

1 fib = 0 : 1 : zipWith (+) fibs (tail fibs)

```

```

1 fib 0 = 0
2 fib 1 = 1
3 fib n = fib (n - 1) + fib (n - 2)

```

Die unendliche Liste alle Fibonacci-Zahlen, also der Fibonacci-*Stream* wird wie folgt erzeugt:

```

1 fibs :: [Integer]
2 fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

```

7.5.3 Polynome

```

1 type Polynom = [Double]
2
3 add :: Polynom -> Polynom -> Polynom
4 add a [] = a
5 add [] a = a
6 add (x:xs) (y:ys) = (x+y) : add xs ys
7

```

```

8 eval :: Polynom -> Double -> Double
9 eval [] x = 0
10 eval (p:ps) x = p + x * (eval ps x)
11 -- alternativ:
12 eval p x = foldr (\element rest -> element+x*rest) 0 p
13
14 deriv :: Polynom -> Polynom
15 deriv [] = []
16 deriv p = zipWith (*) [1..] (tail p)

```

7.5.4 Hirsch-Index

Parameter: Eine Liste L von Zahlen aus \mathbb{N}

Rückgabe: $\max \{ n \in \mathbb{N} \mid n \leq \| \{ i \in L \mid i \geq n \} \| \}$

```

_____ hirsch-index.hs _____
1 import Data.List --sort und reverse
2
3 hIndex :: (Num a, Ord a) => [a] -> a
4 hIndex l = helper (reverse (sort l)) 0
5     where helper [] acc = acc
6           helper (z:ls) acc
7               | z > acc    = helper ls (acc + 1)
8               | otherwise = acc
9
10 -- Alternativ
11 hindex1 = length . takeWhile id .
12           zipWith (<=) [1..] . reverse . sort
13 hindex2 = length . takeWhile \(i, n) -> n >= i) .
14           zip [1..] . reverse . sort

```

7.5.5 Lauflängencodierung

```

1  splitWhen :: (a -> Bool) -> [a] -> ([a], [a])
2  splitWhen _ [] = ([], [])
3  splitWhen p (x:xs)
4      | p x      = ([], x:xs)
5      | otherwise = let (ys, zs) = splitWhen p xs
6                      in (x:ys, zs)
7  -- >>> splitWhen even [1,2,3]
8  -- ([1],[2,3])
9
10 group :: Eq a => [a] -> [[a]]
11 group [] = []
12 group (x:xs) = let (group1, rest) = splitWhen (/=x) xs
13                 in (x:group1) : group rest
14
15 encode :: Eq a => [a] -> [(a, Int)]
16 encode xs = map (\x -> (head x, length x)) (group xs)
17
18 decode [] = []
19 decode ((x,n):xs) = replicate n x ++ decode xs
20 -- alternativ
21 decode = concat . (map (\(x,n)->replicate n x))

```

7.5.6 Intersections

```

1  module Intersect where
2  intersect :: (Ord t) => [t] -> [t] -> [t]
3  intersect a [] = []
4  intersect [] a = []
5  intersect (x:xs) (y:ys)
6      | x == y = x : intersect xs ys
7      | x < y  = intersect xs (y:ys)
8      | y > x  = intersect (x:xs) ys

```

```

9
10 intersectAll :: (Ord t) => [[t]] -> [t]
11 intersectAll (l:ls) = (foldr intersect l) ls
12 intersectAll []      = undefined
13
14 multiples n = [n*k | k <- [1..]]
15 commonMultiples a b c =
16     intersectAll [ multiples n | n <- [a,b,c]]

```

7.5.7 Funktionen höherer Ordnung

```

_____ folds.hs _____
1 summer :: [Int] -> Int
2 summer = foldr (-) 0
3
4 summel :: [Int] -> Int
5 summel = foldl (-) 0
6
7 main :: IO ()
8 main = do
9     print (summer [1,2,3])
10    -- 0-(1-(2-3)) = 0-(1-(-1)) = 2
11    print (summel [1,2,3])
12    -- ((0-1)-2)-3 = -6

```

7.5.8 Church-Zahlen

```

_____ church.hs _____
1 type Church t = (t -> t) -> t -> t
2
3 int2church :: Integer -> Church t
4 int2church 0 s z = z

```

```

5 int2church n s z = int2church (n - 1) s (s z)
6
7 church2int :: Church Integer -> Integer
8 church2int n = n (+1) 0

```

7.5.9 Standard Prelude

Hier sind die Definitionen einiger wichtiger Funktionen:

```

map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs

```

```

zipWith      :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
    = z a b : zipWith z as bs
zipWith _ _ _    = []

```

```

zip          :: [a] -> [b] -> [(a,b)]
zip          = zipWith (,)

```

```

unzip        :: [(a,b)] -> ([a],[b])
unzip        = foldr (\(a,b) ~ (as,bs) -> (a:as,b:bs)

```

```

foldl        :: (a -> b -> a) -> a -> [b] -> a
foldl f z []    = z
foldl f z (x:xs) = foldl f (f z x) xs

```

```

foldr        :: (a -> b -> b) -> b -> [a] -> b
foldr f z []    = z

```

```
foldr f z (x:xs) = f x (foldr f z xs)
-----
```

```
take                :: Int -> [a] -> [a]
take n _           | n <= 0 = []
take _ []          = []
take n (x:xs)      = x : take (n-1) xs
-----
```

```
splitAt             :: Int -> [a] -> ([a],[a])
splitAt n xs        = (take n xs, drop n xs)
-----
```

7.6 Weitere Informationen

- hackage.haskell.org/package/base-4.6.0.1: Referenz
- haskell.org/hoogle: Suchmaschine für das Haskell-Manual
- wiki.ubuntuusers.de/Haskell: Hinweise zur Installation von Haskell unter Ubuntu

8 Prolog

Prolog ist eine Programmiersprache, die das logische Programmierparadigma befolgt.

Eine interaktive Prolog-Sitzung startet man mit `swipl`.

In Prolog definiert man Terme.

8.1 Erste Schritte

8.1.1 Hello World

Speichere folgenden Quelltext als `hello-world.pl`:

```
_____ hello-world.hs _____  
1 :- initialization(main).  
2 main :- write('Hello World!'), nl, halt.  
_____
```

Kompiliere ihn mit `gplc hello-world.pl`. Es wird eine ausführbare Datei erzeugt.

8.2 Syntax

In Prolog gibt es Prädikate, die Werte haben. Prädikate werden immer klein geschrieben. So kann das Prädikat `farbe` mit den Werten `rot`, `gruen`, `blau`, `gelb` - welche auch immer klein geschrieben werden - wie folgt definiert werden:

```
farbe(blau).
farbe(gelb).
farbe(gruen).
farbe(rot).
```

- Terme werden durch `,` mit einem logischem **und** verknüpft.
- Ungleichheit wird durch `=` ausgedrückt.

So ist folgendes Prädikat `nachbar(X, Y)` genau dann wahr, wenn X und Y Farben sind und $X \neq Y$ gilt:

```
nachbar(X, Y) :- farbe(X), farbe(Y), X \= Y.
```

8.2.1 Arithmetik

Die Auswertung arithmetischer Ausdrücke muss in Prolog explizit durch `is` durchgeführt werden:

```
?- X is 5-2*5.
X = -5.
```

Dabei müssen alle Variablen, die im Term rechts von `is` vorkommen, instanziiert sein:

```
?- X is 3^2.
X = 9.
```

```
?- Y is X*X.
```

```
ERROR: is/2: Arguments are not sufficiently
instantiated
```

```
?- X is X+1.
```

```
ERROR: is/2: Arguments are not sufficiently
instantiated
```

Arithmetische Ausdrücke können mit `==`, `=`, `<`, `<=`, `>`, `>=` verglichen werden.

Beispiel 38 (Arithmetik in Prolog¹)

```

1) even(0) .
   even(X) :- X>0, X1 is X-1, odd(X1) .

   odd(1) .
   odd(X) :- X>1, X1 is X-1, even(X1) .

2) fib(0,0) .
   fib(1,1) .
   fib(X,Y) :- X>1,
               X1 is X-1, X2 is X-2,
               fib(X1,Y1), fib(X2,Y2),
               Y is Y1+Y2 .

```

8.2.2 Listen

Das Atom `[]` ist die leere Liste.

Mit der Syntax `[K|R]` wird eine Liste in den Listekopf `K` und den Rest der Liste `R` gesplittet:

```

?- [X|Y] = [1,2,3,4,5] .
X = 1,
Y = [2, 3, 4, 5] .

```

Einen Test `member(X, Liste)`, der `True` zurückgibt wenn `X` in `Liste` vorkommt, realisiert man wie folgt:

```

member(X, [X|R]) .
member(X, [Y|R]) :- member(X,R) .

```

¹WS 2013 / 2014, Folie 237f

Eine Regel `append(A, B, C)`, die die Listen `A` und `B` zusammenfügt und als Liste `C` speichert, kann wie folgt erstellt werden:

```
append([], L, L) .
append([X|R], L, [X|T]) :- append(R, L, T) .
```

Die erste Regel besagt, dass das Hinzufügen der leeren Liste zu einer Liste `L` immer noch die Liste `L` ist.

Die zweite Regel besagt: Wenn die Liste `R` und `L` die Liste `T` ergeben, dann ergibt die Liste, deren Kopf `X` ist und deren Rumpf `R` ist zusammen mit der Liste `L` die Liste mit dem Kopf `X` und dem Rumpf `T`.

Übergibt man `append(X, Y, [1, 2, 3, 4, 5])`, so werden durch Reerfüllung alle Möglichkeiten durchgegangen, wie man die Liste `[1, 2, 3, 4, 5]` splitten kann.

8.3 Beispiele

8.3.1 Humans

Erstelle folgende Datei:

```
human.pro
```

```
1 human(bob) .
2 human(socrates) .
3 human(antonio) .
```

Kompiliere diese mit

```
$ swipl -c human.pro
% library(swi_hooks) compiled into pce_swi_hooks
%                0.00 sec, 2,224 bytes
% human.pro compiled 0.00 sec, 644 bytes
% /usr/lib/swi-prolog/library/listing compiled into
%                prolog_listing 0.00 sec, 21,648 bytes
```

Dabei wird eine `a.out` Datei erzeugt, die man wie folgt nutzen kann:

```
$ ./a.out
```

```
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
software, and you are welcome to redistribute it under certain
conditions. Please visit http://www.swi-prolog.org for
```

```
For help, use ?- help(Topic). or ?- apropos(Word).
```

```
?- human(socrates).
true.
```

8.3.2 Splits

```
_____ splits.pl _____
1 splits(L, ([], L)).
2 splits([X|L], ([X|S], E)) :- splits(L, (S, E)).
_____
```

Dieses skript soll man `swipl -f test.pl` aufrufen. Dann erhält man:

```
? splits([1,2,3], Res).
Res = ([], [1,2,3]) ;
Res = ([1], [2,3]) ;
Res = ([1,2], [3]) ;
Res = ([1,2,3], []) ;
No
```

8.3.3 Delete

```
remove([(X,A)|L],X,[(X,ANew)|L]) :- A>0, ANew is A-1.
remove([X|L],Y,[X|L1]) :- remove(L,Y,L1).
```

8.3.4 Zebrarätsel

Folgendes Rätsel wurde von <https://de.wikipedia.org/w/index.php?title=Zebrar%C3%A4tsel&oldid=126585006> entnommen:

1. Es gibt fünf Häuser.
2. Der Engländer wohnt im roten Haus.
3. Der Spanier hat einen Hund.
4. Kaffee wird im grünen Haus getrunken.
5. Der Ukrainer trinkt Tee.
6. Das grüne Haus ist direkt rechts vom weißen Haus.
7. Der Raucher von Altem-Gold-Zigaretten hält Schnecken als Haustiere.
8. Die Zigaretten der Marke Kools werden im gelben Haus geraucht.
9. Milch wird im mittleren Haus getrunken.
10. Der Norweger wohnt im ersten Haus.
11. Der Mann, der Chesterfields raucht, wohnt neben dem Mann mit dem Fuchs.
12. Die Marke Kools wird geraucht im Haus neben dem Haus mit dem Pferd.
13. Der Lucky-Strike-Raucher trinkt am liebsten Orangensaft.
14. Der Japaner raucht Zigaretten der Marke Parliaments.
15. Der Norweger wohnt neben dem blauen Haus.

Wer trinkt Wasser? Wem gehört das Zebra?

_____ zebraraetsel.pro _____

```
1 Street=[Haus1,Haus2,Haus3],
2 mitglied(haus(rot,_,_),Street),
3 mitglied(haus(blau,_,_),Street),
4 mitglied(haus(grün,_,_),Street),
5 mitglied(haus(rot,australier,_),Street),
6 mitglied(haus(_,italiener,tiger),Street),
7 sublist(haus(_,_,eidechse),haus(_,chinese,_),Street),
8 sublist(haus(blau,_,_),haus(_,_,eidechse),Street),
9 mitglied(haus(_,N,nilpferd),Street).
```

TODO

8.4 Weitere Informationen

- wiki.ubuntuusers.de/Prolog: Hinweise zur Installation von Prolog unter Ubuntu

9 Scala

Scala ist eine objektorientierte und funktionale Programmiersprache, die auf der JVM aufbaut und in Java Bytecode kompiliert wird. Scala bedeutet scalable language.

Mit sog. „actors“ bietet Scala eine Unterstützung für die Entwicklung parallel ausführender Programme.

Weitere Materialien sind unter <http://www.scala-lang.org/> und <http://www.simplyscala.com/> zu finden.

9.1 Erste Schritte

Scala kann auf Debian-basierten Systemen durch das Paket `scala` installiert werden. Für andere Systeme stehen auf <http://www.scala-lang.org/download/> verschiedene Binärdateien bereit.

9.1.1 Hello World

Interaktiv

```
$ scala
Welcome to Scala version 2.9.2 [...]\n\nscala> println("Hello world")
Hello world
```

Es kann mit `./scala-test.scala` Scala funktioniert ausgeführt werden.

Kompiliert

```
_____ hello-world.scala _____  
1 object HelloWorld {  
2     def main(args: Array[String]) {  
3         println("Hello World!")  
4     }  
5 }
```

Dieses Beispiel kann mit `scalac hello-world.scala` kompiliert und mit `scala HelloWorld` ausgeführt werden.

9.2 Vergleich mit Java

Scala und Java haben einige Gemeinsamkeiten, wie den Java Bytecode, aber auch einige Unterschiede.

Gemeinsamkeiten

- Java Bytecode
- Keine Mehrfachvererbung
- Statische Typisierung
- Scopes

Unterschiede

- Java hat Interfaces, Scala hat traits.
- Java hat primitive Typen, Scala ausschließlich Objekte.
- Scala benötigt kein `;` am Ende von Anweisungen.
- Scala ist kompakter.
- Java hat `static`, Scala hat `object` (Singleton)

Weitere Informationen hat Graham Lea unter <http://tinyurl.com/scala-hello-world> zur Verfügung gestellt.

9.3 Syntax

In Scala gibt es sog. *values*, die durch das Schlüsselwort `val` angezeigt werden. Diese sind Konstanten. Die Syntax ist der UML-Syntax ähnlich.

```
val name: type = value
```

Variablen werden durch das Schlüsselwort `var` angezeigt:

```
var name: type = value
```

Methoden werden mit dem Schlüsselwort `def` erzeugt:

```
def name(parameter: String): Unit = { code ... }
```

Klassen werden wie folgt erstellt:

```
class Person (  
    val firstName: String,  
    var lastName: String,  
    age: Int) {  
    println("This is the constructor.")  
  
    def sayHi() = println("Hello world!")  
}
```

und so instanziiert:

```
val anna = new Person("anna", "bern", 18)  
anna.sayHi()
```

Listen können erstellt und durchgegangen werden:

```
val list = List("USA", "Russia", "Germany")  
for(country <- list)  
    println(country)
```

9.4 Companion Object

Ein Companion Object ist ein Objekt mit dem Namen einer Klasse oder eines Traits. Im Gegensatz zu anderen Objekten / Traits hat das Companion Object Zugriff auf die Klasse.

9.5 actor

Definition 49 (Aktor)

Ein *Aktor* ist ein Prozess, der Nebenläufig zu anderen Aktoren läuft. Er kommuniziert mit anderen Aktoren, indem er Nachrichten austauscht.

Das folgende Wetter-Beispiel zeigt, wie man Aktoren benutzen kann.

9.6 Beispiele

9.6.1 Wetter

Das folgende Script sendet parallel Anfragen über verschiedene ZIP-Codes an die Yahoo-Server, parst das XML und extrahiert die Stadt sowie die Temperatur:

```
_____ weather.scala _____  
1 import scala.io._  
2 import scala.xml.{Source => Source2, _}  
3 import scala.actors._  
4 import Actor._  
5  
6 def getWeatherInfo(woeid: String) = {  
7     val url = "http://weather.yahooapis.com/forecastrss?  
8     val response = Source.fromURL(url).mkString
```

```
9      val xmlResponse = XML.loadString(response)
10      println(xmlResponse \\ "location" \\ "@city",
11              xmlResponse \\ "condition" \\ "@temp")
12  }
13
14  val caller = self
15
16  for(id <- 2391271 to 2391279) {
17      actor{ getWeatherInfo(id.toString) }
18  }
19
20  for(id <- 2391271 to 2391279) {
21      receiveWithin(5000) {
22          case msg => println(msg)
23      }
24  }
```

9.7 Weitere Informationen

- <http://docs.scala-lang.org/style/naming-conventions.html>

10 X10

X10 ist eine objektorientierte Programmiersprache, die 2004 bei IBM entwickelt wurde.

X10 nutzt das PGAS-Modell:

Definition 50 (PGAS¹)

PGAS (partitioned global address space) ist ein Programmiermodell für Mehrprozessorsysteme und massiv parallele Rechner. Dabei wird der globale Adressbereich des Arbeitsspeichers logisch unterteilt. Jeder Prozessor bekommt jeweils einen dieser Adressbereiche als lokalen Speicher zugeteilt. Trotzdem können alle Prozessoren auf jede Speicherzelle zugreifen, wobei auf den lokalen Speicher mit wesentlich höherer Geschwindigkeit zugegriffen werden kann als auf den von anderen Prozessoren.

10.1 Erste Schritte

Als erstes sollte man x10 von <http://x10-lang.org/x10-development/building-x10-from-source.html?id=248> herunterladen.

Dann kann man die `bin/x10c++` zum erstellen von ausführbaren Dateien nutzen. Der Befehl `x10c++ hello-world.x10` erstellt eine ausführbare Datei namens `a.out`.

`hello-world.x10`

¹<https://de.wikipedia.org/wiki/PGAS>

```
// file HelloWorld.x10
public class HelloWorld {
    public static def main(args:Rail[String]) {
        x10.io.Console.OUT.println("Hello, World");
    }
}
```

10.2 Syntax

10.3 Datentypen

Byte, UByte, Short, UShort, Char, Int, UInt, Long, ULong, Float, Double, Boolean, Complex, String, Point, Region, Dist, Array

10.4 Beispiele

10.5 Weitere Informationen

- <http://x10-lang.org/>

11 C

C ist eine imperative Programmiersprache. Sie wurde in vielen Standards definiert. Die wichtigsten davon sind:

- C89 wird auch ANSI C genannt.
- C90 wurde unter ISO 9899:1990 veröffentlicht. Es gibt keine bedeutenden Unterschiede zwischen C89 und C90, nur ist das eine ein ANSI-Standard und das andere ein ISO-Standard.
- C99 wurde unter ISO 9899:1999 veröffentlicht.
- C11 wurde unter ISO 9899:2011 veröffentlicht.

11.1 Datentypen

Die grundlegenden C-Datentypen sind

Typ	Größe
char	1 Byte
int	4 Bytes
float	4 Bytes
double	8 Bytes
void	0 Bytes

zusätzlich kann man `char` und `int` noch in `signed` und `unsigned` unterscheiden. Diese werden *Modifier* genannt.

In C gibt es keinen direkten Support für Booleans.

11.2 ASCII-Tabelle

11.3 Syntax

11.4 Präzedenzregeln

A „[name] is a...“

B.1 parenthesis **()**

B.2 postfix operators:

B.2.1 **()** „...function returning...“

B.2.2 **[]** „...array of...“

B.3 prefix operator: ***** „...pointer to...“

B.4 prefix operator ***** and **const** / **volatile** modifier:
 „...[modifier] pointer to...“

B.5 **const** / **volatile** modifier next to type specifier:
 „...[modifier] [specifier]“

B.6 type specifier: „...[specifier]“

```
static unsigned int* const *(*next)();
```

11.5 Beispiele

11.5.1 Hello World

Speichere den folgenden Text als `hello-world.c`:

```
_____ hello-world.c _____
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, World\n");
```

```
6     return 0;  
7 }
```

Compiliere ihn mit `gcc hello-world.c`. Es wird eine ausführbare Datei namens `a.out` erzeugt.

11.5.2 Pointer

```
1 #include <stdio.h>  
2  
3 int arr[] = {0,1,2,3,4,5};  
4  
5 int main() {  
6     printf("%i %i", arr[0], (&arr[3])[0]);  
7     return 0;  
8 }
```

Die Ausgabe hier ist 0 3.

Dez.	Z.	Dez.	Z.	Dez.	Z.	Dez.	Z.
0		32		64	@	96	'
1		33	!	65	A	97	a
2		34	"	66	B	98	b
3		35	#	67	C	99	c
4		36	\$	68	D	100	d
5		37	%	69	E	101	e
6		38	&	70	F	102	f
7		39	'	71	G	103	g
8		40	(72	H	104	h
9		41)	73	I	105	i
10		42	*	74	J	106	j
11		43	+	75	K	107	k
12		44	,	76	L	108	l
13		45	-	77	M	109	m
14		46	.	78	N	110	n
15		47	/	79	O	111	o
16		48	0	80	P	112	p
17		49	1	81	Q	113	q
18		50	2	82	R	114	r
19		51	3	83	S	115	s
20		52	4	84	T	116	t
21		53	5	85	U	117	u
22		54	6	86	V	118	v
23		55	7	87	W	119	w
24		56	8	88	X	120	x
25		57	9	89	Y	121	y
26		58	:	90	Z	122	z
27		59	;	91	[123	{
28		60	<	92	\	124	
29		61	=	93]	125	}
30		62	>	94	^	126	~
31		63	?	95	_	127	DEL

A	next	next is a
B.3	*	... pointer to...
B.1	()	...
B.2.1	()	... a function returning...
B.3	*	... pointer to...
B.4	*const	... a read-only pointer to...
B.6	static unsigned int	... static unsigned int.

12 MPI

Message Passing Interface (kurz: MPI) ist ein Standard, der den Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Computersystemen beschreibt.

12.1 Erste Schritte

```
hello-world.c
#include <stdio.h>
#include <mpi.h>
int main (int argc, char** args) {
    int size;
    int myrank;
    MPI_Init(&argc, &args);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("Hello world, I have rank %d out of %d.\n",
        myrank, size);
    MPI_Finalize();
    return 0;
}
```

Das wird `mpicc hello-world.c` kompiliert.

Mit `mpirun -np 14 scripts/mpi/a.out` werden 14 Kopien des Programms gestartet.

12.2 Funktionen

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

Liefert die Größe des angegebenen Kommunikators; dh. die Anzahl der Prozesse in der Gruppe.

Parameter

- **comm**: Kommunikator (handle)
- **size**: Anzahl der Prozesse in der Gruppe von comm

Beispiel

```
#include "mpi.h"

int          size;
MPI_Comm     comm;
...
MPI_Comm_size(comm, &size);
...
```

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

Bestimmt den Rang des rufenden Prozesses innerhalb des Kommunikators.

Der Rang wird von MPI zum Identifizieren eines Prozesses verwendet. Die Rangnummer ist innerhalb eines Kommunikators eindeutig. Dabei wird stets von Null beginnend durchnummeriert. Sender und Empfänger bei Sendeoperationen oder die Wurzel bei kollektiven Operationen werden immer mittels Rang angegeben.

Parameter

- **comm**: Kommunikator (handle)

- **rank**: Rang des rufenden Prozesses innerhalb von `comm`

Beispiel

```
#include "mpi.h"

int          rank;
MPI_Comm     comm;

...
MPI_Comm_rank(comm, &rank);
if (rank==0) {
    ... Code fur Prozess 0 ...
}
else {
    ... Code fur die anderen Prozesse ...
}
```

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

Senden einer Nachricht an einen anderen Prozeß innerhalb eines Kommunikators. (Standard-Send)

Parameter

- **buf**: Anfangsadresse des Sendepuffers
- **count**: Anzahl der Elemente des Sendepuffers (nichtnegativ)
- **datatype**: Typ der Elemente des Sendepuffers (handle)
- **dest**: Rang des Empfängerprozesses in `comm` (integer)
- **tag**: message tag zur Unterscheidung verschiedener Nachrichten; Ein Kommunikationsvorgang wird durch ein Tripel (Sender, Empfänger, tag) eindeutig beschrieben.

- **comm**: Kommunikator (handle)

Beispiel

```
#include "mpi.h"
...
int signal, i, numprocs, me;
MPI_Status stat;
MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD,
               &numprocs);
if (me==ROOT) {
    ...
    for (i=1; i<numprocs; i++) {
        MPI_Send(&signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
    ...
else {
    MPI_Recv(&sig, 1, MPI_INT, ROOT, MPI_ANY_TAG,
            MPI_COMM_WORLD, &stat);
    ...
}
```

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

Empfangen einer Nachricht (blockierend)

Parameter

- **buf**: Anfangsadresse des Empfangspuffers
- **status**: Status, welcher source und tag angibt (MPI_Status)
- **count**: Anzahl der Elemente im Empfangspuffer (nichtnegativ)

- **datatype**: Typ der zu empfangenden Elemente (handle)
- **source**: Rang des Senderprozesses in comm oder MPI_ANY_SOURCE
- **tag**: message tag zur Unterscheidung verschiedener Nachrichten Ein Kommunikationsvorgang wird durch ein Tripel (Sender, Empfänger, tag) eindeutig beschrieben. Um Nachrichten mit beliebigen tags zu empfangen, benutzt man die Konstante MPI_ANY_TAG.
- **comm**: Kommunikator (handle)

Beispiel

```
#include "mpi.h"

int          msglen, again=1;
void         *buf;
MPI_Datatype datatype
MPI_Comm     comm;
MPI_Status   status;

...
while (again) {
    MPI_Probe(ROOT, MPI_ANY_TAG, comm, &status);
    MPI_Get_count(&status, datatype, &msglen);
    buf=malloc(msglen*sizeof(int));
    MPI_Recv(buf, msglen, datatype, status.MPI_SOURCE,
             status.MPI_TAG, comm, &status);
    ...
}
...
```

```
int MPI_Reduce(const void *sendbuf, void *recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
```

Führt eine globale Operation **op** aus; der Prozeß „root“ erhält das Resultat.

Parameter

- **sendbuf**: Startadresse des Sendepuffers
 - **count**: Anzahl der Elemente im Sendepuffer
 - **datatype**: Datentyp der Elemente von sendbuf
 - **op**: auszuführende Operation (handle)
 - **root**: Rang des Root-Prozesses in comm, der das Ergebnis haben soll
 - **comm**: Kommunikator (handle)
-

`MPI_Bcast(buffer, count, datatype, root, comm)`

Sendet eine Nachricht vom Prozess `root` an alle anderen Prozesse des angegebenen Kommunikators.

Parameter

- **buffer**: Startadresse des Datenpuffers
 - **count**: Anzahl der Elemente im Puffer
 - **datatype**: Datentyp der Pufferelemente (handle)
 - **root**: Wurzelprozeß; der, welcher sendet
 - **comm**: Kommunikator (handle)
-

`MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, root, comm)`

Verteilt Daten vom Prozess `root` unter alle anderen Prozesse in der Gruppe, so daß, soweit möglich, alle Prozesse gleich große Anteile erhalten.

Parameter

- **sendbuf**: Anfangsadresse des Sendepuffers (Wert ist lediglich für 'root' signifikant)
- **sendcount**: Anzahl der Elemente, die jeder Prozeß geschickt bekommen soll (integer)
- **sendtype**: Datentyp der Elemente in sendbuf (handle)
- **recvcount**: Anzahl der Elemente im Empfangspuffer. Meist ist es günstig, `recvcount = sendcount` zu wählen.
- **recvtype**: Datentyp der Elemente des Empfangspuffers (handle)
- **root**: Rang des Prozesses in comm, der die Daten versendet
- **comm**: Kommunikator (handle)

Beispiel

```
#include "mpi.h"
```

```
int myid;
```

```
int recvbuf[DATASIZE], sendbuf[DATA_SIZE];
```

```
...
```

```
/* Minimum bilden */
```

```
MPI_Reduce(sendbuf, recvbuf, DATA_SIZE, MPI_INT, MPI_MIN,
           0, MPI_COMM_WORLD);
```

```
...
```

12.3 Beispiele

12.4 Weitere Informationen

- <http://mpitutorial.com/>
- <http://www.open-mpi.org/>
- <http://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/>

13 Compilerbau

Wenn man über Compiler redet, meint man üblicherweise „vollständige Übersetzer“:

Definition 51

Ein **Compiler** ist ein Programm C , das den Quelltext eines Programms A in eine ausführbare Form übersetzen kann.

Jedoch gibt es verschiedene Ebenen der Interpretation bzw. Übersetzung:

1. **Reiner Interpretierer**: TCL, Unix-Shell
2. **Vorübersetzung**: Java-Bytecode, Pascal P-Code, Python¹, Smalltalk-Bytecode
3. **Laufzeitübersetzung**: JavaScript²
4. **Vollständige Übersetzung**: C, C++, Fortran

Zu sagen, dass Python eine interpretierte Sprache ist, ist in etwa so korrekt wie zu sagen, dass die Bibel ein Hardcover-Buch ist.³

Reine Interpretierer lesen den Quelltext Anweisung für Anweisung und führen diese direkt aus.

Bild

¹Python hat auch `.pyc`-Dateien, die Python-Bytecode enthalten.

²JavaScript wird nicht immer zur Laufzeit übersetzt. Früher war es üblich, dass JavaScript nur interpretiert wurde.

³Quelle: stackoverflow.com/a/2998544, danke Alex Martelli für diesen Vergleich.

Bei der *Interpretation nach Vorübersetzung* wird der Quelltext analysiert und in eine für den Interpretierer günstigere Form übersetzt. Das kann z. B. durch

- Zuordnung Bezeichnergebrauch - Vereinbarung
- Transformation in Postfixbaum
- Typcheck, wo statisch möglich

geschehen. Diese Vorübersetzung ist nicht unbedingt maschinen-nah.

Bild

Die *Just-in-time-Compiler* (kurz: JIT-Compiler) betreiben Laufzeitübersetzung. Folgendes sind Vor- bzw. Nachteile von Just-in-time Compilern:

- schneller als reine Interpretierer
- Speichergewinn: Quelle kompakter als Zielprogramm
- Schnellerer Start des Programms
- Langsamer (pro Funktion) als vollständige Übersetzung
- kann dynamisch ermittelte Laufzeiteigenschaften berücksichtigen (dynamische Optimierung)

Moderne virtuelle Maschinen für Java und für .NET nutzen JIT-Compiler.

Bei der *vollständigen Übersetzung* wird der Quelltext vor der ersten Ausführung des Programms *A* in Maschinencode (z. B. x86, SPARC) übersetzt.

Bild

Was
ist
hier
ge-
meint?

13.1 Funktionsweise

Üblicherweise führt ein Compiler folgende Schritte aus:

1. Lexikalische Analyse
2. Syntaktische Analyse
3. Semantische Analyse
4. Zwischencodeoptimierung
5. Codegenerierung
6. Assemblieren und Binden

13.2 Lexikalische Analyse

In der lexikalischen Analyse wird der Quelltext als Sequenz von Zeichen betrachtet. Sie soll bedeutungstragende Zeichengruppen, sog. *Tokens*, erkennen und unwichtige Zeichen, wie z. B. Kommentare überspringen. Außerdem sollen Bezeichner identifiziert und in einer *Stringtabelle* zusammengefasst werden.

Beispiel 39

Beispiel erstellen

13.2.1 Reguläre Ausdrücke

Beispiel 40 (Regulärere Ausdrücke)

Folgender regulärer Ausdruck erkennt Float-Konstanten in C nach ISO/IEC 9899:1999 §6.4.4.2:

$$((0| \dots |9)^*.(0| \dots |9)^+)|((0| \dots |9)^+.)$$

Satz 13.1

Jede reguläre Sprache wird von einem (deterministischen) endlichen Automaten akzeptiert.

TODO: Bild einfügen

Zu jedem regulären Ausdruck im Sinne der theoretischen Informatik kann ein nichtdeterministischer Automat generiert werden. Dieser kann mittels Potenzmengenkonstruktion⁴ in einen deterministischen Automaten überführen. Dieser kann dann mittels Äquivalenzklassen minimiert werden.

Alle Schritte beschreiben

13.2.2 Lex

Lex ist ein Programm, das beim Übersetzerbau benutzt wird um Tokenizer für die lexikalische Analyse zu erstellen. Flex ist eine Open-Source Variante davon.

Eine Flex-Datei besteht aus 3 Teilen, die durch %% getrennt werden:

Definitionen: Definiere Namen

%%

Regeln: Definiere reguläre Ausdrücke und
zugehörige Aktionen (= Code)

%%

Code: zusätzlicher Code

<code>x</code>	Zeichen 'x' erkennen
<code>"xy"</code>	Zeichenkette 'xy' erkennen
<code>\</code>	Zeichen 'x' erkennen (TODO)
<code>[xyz]</code>	Zeichen x , y oder z erkennen
<code>[a - z]</code>	Alle Kleinbuchstaben erkennen
<code>[- z]</code>	Alle Zeichen außer Kleinbuchstaben erkennen
<code>x y</code>	x oder y erkennen
<code>(x)</code>	x erkennen
<code>x*</code>	0, 1 oder mehrere Vorkommen von x erkennen
<code>x+</code>	1 oder mehrere Vorkommen von x erkennen
<code>x?</code>	0 oder 1 Vorkommen von x erkennen
<code>{Name}</code>	Expansion der Definition Name
<code>\t, \n, \rq</code>	Tabulator, Zeilenumbruch, Wagenrücklauf erkennen

Reguläre Ausdrücke in Flex

13.3 Syntaktische Analyse

In der syntaktischen Analyse wird überprüft, ob die Tokenfolge zur kontextfreien Sprache gehört. Außerdem soll die hierarchische Struktur der Eingabe erkannt werden.

Ausgegeben wird ein **abstrakter Syntaxbaum**.

Beispiel 41 (Abstrakter Syntaxbaum)

TODO

Warum
kon-
text-
frei?

Was
ist ge-
meint?

13.4 Semantische Analyse

Die semantische Analyse arbeitet auf einem abstrakten Syntaxbaum und generiert einen attribuierten Syntaxbaum.

Sie führt eine kontextsensitive Analyse durch. Dazu gehören:

⁴<http://martin-thoma.com/potenzmengenkonstruktion/>

- **Namensanalyse:** Beziehung zwischen Deklaration und Verwendung
- **Typanalyse:** Bestimme und prüfe Typen von Variablen, Funktionen, ...
- **Konsistenzprüfung:** Wurden alle Einschränkungen der Programmiersprache eingehalten?

Beispiel 42 (Attributeriter Syntaxbaum)

TODO


13.5 Zwischencodeoptimierung

Hier wird der Code in eine sprach- und zielunabhängige Zwischensprache transformiert. Dabei sind viele Optimierungen vorstellbar. Ein paar davon sind:

- **Konstantenfaltung:** Ersetze z. B. $3 + 5$ durch 8.
- **Kopienfortschaffung:** Setze Werte von Variablen direkt ein
- **Code verschieben:** Führe Befehle vor der Schleife aus, statt in der Schleife
- **Gemeinsame Teilausdrücke entfernen:** Es sollen doppelte Berechnungen vermieden werden
- **Inlining:** Statt Methode aufzurufen, kann der Code der Methode an der Aufrufstelle eingebaut werden.

13.6 Codegenerierung

Der letzte Schritt besteht darin, aus dem generiertem Zwischencode den Maschinencode oder Assembler zu erstellen. Dabei muss folgendes beachtet werden:

- **Konventionen:** Wie werden z. B. im Laufzeitsystem Methoden aufgerufen?
- **Codeauswahl:** Welche Befehle kennt das Zielsystem?
- **Scheduling:** In welcher Reihenfolge sollen die Befehle angeordnet werden?
- **Registerallokation:** Welche Zwischenergebnisse sollen in welchen Prozessorregistern gehalten werden?
- **Nachoptimierung** 

13.7 Literatur

Ich kann das folgende Buch empfehlen:

Compiler - Prinzipien, Techniken und Werkzeuge. Alfred V. Aho, Monica S. Lam, Ravi Sethi und Jeffry D. Ullman. Pearson Verlag, 2. Auflage, 2008. ISBN 978-3-8273-7097-6.

Es ist mit über 1200 Seiten zwar etwas dick, aber dafür sehr einfach geschrieben.

14 Java Bytecode

Definition 52 (Bytecode)

Der Bytecode ist eine Sammlung von Befehlen für eine virtuelle Maschine.

Bytecode ist unabhängig von realer Hardware.

Definition 53 (Heap)

Der dynamische Speicher, auch Heap genannt, ist ein Speicherbereich, aus dem zur Laufzeit eines Programms zusammenhängende Speicherabschnitte angefordert und in beliebiger Reihenfolge wieder freigegeben werden können.

Activation Record ist ein *Stackframe*.

14.1 Instruktionen

Beschreibung	int	float
Addition	iadd	fadd
Element aus Array auf Stack packen	iaload	faload
Element aus Stack in Array speichern	iastore	fastore
Konstante auf Stack legen	iconst_<i>	fconst_<f>
Divide second-from top by top	idiv	fdiv
Multipliziere die obersten beiden Zahlen des Stacks	imul	fmul

14.2 Weitere Informationen

- <http://cs.au.dk/~mis/dOvs/jvmspec/ref-Java.html>

Bildquellen

Abb. ?? S^2 : Tom Bombadil, tex.stackexchange.com/a/42865

Abkürzungsverzeichnis

AST Abstrakter Syntaxbaum (Abstract Syntax Tree)

Beh. Behauptung

Bew. Beweis

bzgl. bezüglich

bzw. beziehungsweise

ca. circa

d. h. das heißt

DEA Deterministischer Endlicher Automat

etc. et cetera

ggf. gegebenenfalls

mgu most general unifier

sog. sogenannte

Vor. Voraussetzung

vgl. vergleiche

z. B. zum Beispiel

z. z. zu zeigen

Ergänzende Definitionen

Definition 54 (Quantoren)

- a) $\forall x \in X : p(x)$: Für alle Elemente x aus der Menge X gilt die Aussage p .
- b) $\exists x \in X : p(x)$: Es gibt mindestens ein Element x aus der Menge X , für das die Aussage p gilt.
- c) $\exists! x \in X : p(x)$: Es gibt genau ein Element x in der Menge X , sodass die Aussage p gilt.

Definition 55 (Prädikatenlogik)

Eine Prädikatenlogik ist ein formales System, das Variablen und Quantoren nutzt um Aussagen zu formulieren.

Definition 56 (Aussagenlogik)

TODO

Definition 57 (Grammatik)

Eine (formale) **Grammatik** ist ein Tupel (Σ, V, P, S) wobei gilt:

- (i) Σ ist eine endliche Menge und heißt **Alphabet**,
- (ii) V ist eine endliche Menge mit $V \cap \Sigma = \emptyset$ und heißt **Menge der Nichtterminale**,
- (iii) $S \in V$ heißt das **Startsymbol**
- (iv) $P = \{ p : I \rightarrow r \mid I \in (V \cup \Sigma)^+, r \in (V \cup \Sigma)^* \}$ ist eine endliche Menge aus **Produktionsregeln**

Man schreibt:

- $a \Rightarrow b$: Die Anwendung einer Produktionsregel auf a ergibt b .
- $a \Rightarrow^* b$: Die Anwendung mehrerer (oder keiner) Produktionsregeln auf a ergibt b .
- $a \Rightarrow^+ b$: Die Anwendung mindestens einer Produktionsregel auf a ergibt b .

Beispiel 43 (Formale Grammatik)

Folgende Grammatik $G = (\Sigma, V, P, A)$ erzeugt alle korrekten Klammerausdrücke:

- $\Sigma = \{ (,) \}$
- $V = \{ \alpha \}$
- $s = \alpha$
- $P = \{ \alpha \rightarrow () \mid \alpha\alpha|(\alpha) \}$

Definition 58 (Kontextfreie Grammatik)

Eine Grammatik (Σ, V, P, S) heißt **kontextfrei**, wenn für jede Produktion $p : I \rightarrow r$ gilt: $I \in V$.

Definition 59 (Sprache)

Sei $G = (\Sigma, V, P, S)$ eine Grammatik. Dann ist

$$L(G) := \{ \omega \in \Sigma^* \mid S \Rightarrow^* \omega \}$$

die Menge aller in der Grammatik ableitbaren Wörtern. $L(G)$ heißt Sprache der Grammatik G .

Definition 60

Sei $G = (\Sigma, V, P, S)$ eine Grammatik und $a \in (V \cup \Sigma)^+$.

- a) \Rightarrow_L heißt **Linksableitung**, wenn die Produktion auf das linkeste Nichtterminal angewendet wird.
- b) \Rightarrow_R heißt **Rechtsableitung**, wenn die Produktion auf das rechteste Nichtterminal angewendet wird.

Beispiel 44 (Links- und Rechtsableitung)

Sie G wie zuvor die Grammatik der korrekten Klammerausdrücke:

$$\begin{array}{ll}
 \alpha \Rightarrow_L \alpha\alpha & \iff \alpha \Rightarrow_R \alpha\alpha \\
 \Rightarrow_L \alpha\alpha\alpha & \Rightarrow_R \alpha\alpha\alpha \\
 \Rightarrow_L ()\alpha\alpha & \Rightarrow_R \alpha\alpha() \\
 \Rightarrow_L ()(\alpha)\alpha & \Rightarrow_R \alpha(\alpha)() \\
 \Rightarrow_L ()(())\alpha & \Rightarrow_R \alpha(())() \\
 \Rightarrow_L ()(()()) & \Rightarrow_R ()(())()
 \end{array}$$

Definition 61 (LL(k)-Grammatik)

Sei $G = (\Sigma, V, P, S)$ eine kontextfreie Grammatik. G heißt LL(k)-Grammatik für $k \in \mathbb{N}_{\geq 1}$, wenn jeder Ableitungsschritt durch die linkesten k Symbole der Eingabe bestimmt ist.

Ein LL-Parser ist ein Top-Down-Parser, der die Eingabe von links nach rechts liest und versucht, eine Linksableitung der Eingabe zu berechnen. Ein LL(k)-Parser kann k Token vorausschauen, wobei k als *Lookahead* bezeichnet wird.

Was ist die Eingabe einer Grammatik?

Satz .1

Für linksrekursive, kontextfreie Grammatiken G gilt:

$$\forall k \in \mathbb{N} : G \notin \text{SLL}(k)$$

Symbolverzeichnis

Reguläre Ausdrücke

\emptyset	Leere Menge
ϵ	Das leere Wort
α, β	Reguläre Ausdrücke
$L(\alpha)$	Die durch α beschriebene Sprache
$L(\alpha \beta)$	$L(\alpha) \cup L(\beta)$
L^0	Die leere Sprache, also $\{\epsilon\}$
L^{n+1}	Potenz einer Sprache. Diese ist definiert als $L^n \circ L$ für $n \in \mathbb{N}_0$
$\alpha^+ = L(\alpha)^+$	$\bigcup_{i \in \mathbb{N}} L(\alpha)^i$
$\alpha^* = L(\alpha)^*$	$\bigcup_{i \in \mathbb{N}_0} L(\alpha)^i$

Logik

$\mathcal{M} \models \varphi$	Semantische Herleitbarkeit Im Modell \mathcal{M} gilt das Prädikat φ .
$\psi \vdash \varphi$	Syntaktische Herleitbarkeit Die Formel φ kann aus der Menge der Formeln ψ hergeleitet werden.

Weiteres

- \perp Bottom
- \Downarrow TODO?
- \succeq Typschemainstanziierung

Stichwortverzeichnis

Äquivalenz

Alpha (α), 28

Beta (β), 28

Eta (η), 29

Ableitungsbaum, 42

Ableitungsregel, *siehe* Produktionsregel

Activation Record, *siehe* Stack-frame

actor, *siehe* Akteur

Akkumulator, 58

Akteur, 84

Alphabet, 117

Analyse

lexikalische, 105

semantische, 107

syntaktische, 107

Assembler, 4

Ausdrücke

reguläre, 105

Aussagenlogik, 117

Backtracking, 18

Befehlssatz, 3

Binomialkoeffizient, 15

Broadcast, 53

Bytecode, 111

C, 89–91

Call-By-Name, 29

Call-By-Value, 30

char, 89

Church-Booleans, 32

Companion Object, 84

Compiler, 103

Just-in-time, 104

Compilerbau, 103–109

concat, 67

cons, 58

Constraints, 43

data, 64

Datentyp, 39

algebraischer, 64

Datentypen, 89

def, 83

Duck-Typing, 9

even, 75

Fakultät, 15

fib, 75

Fibonacci, 65

Fibonacci-Funktion, 15

filter, 18
Fixpunkt, 33
Fixpunkt-Kombinator, 33
Flex, *siehe* Lex
Flynn'sche Klassifikation, 49
foldl, 69
foldr, 66, 69
Folds, 69
Funktion
 endrekursive, 17
 linear rekursive, 17
 rekursive, 15

Grammatik, 117
 Kontextfreie, 118
group, 67
Guard, 57

Haskell, 55–71
Heap, 111
Hirsch-Index, 67

ILP, 47
int, 89
Intersections, 68

Java Bytecode, 111–112
JIT, *siehe* Just-in-time Compiler

Kombinator, 25, 33
Kommunikator, 52
Kurzschlussauswertung, 29

Laufgrößencodierung, 67
Lazy Evaluation, 64
let, 60

let-Polymorphismus, 35
Lex, 106–107
Linksableitung, 118
List-Comprehension, 59, 68
LL(k)-Grammatik, 119
Lookahead, 119

map, 18
Maschinensprache, 3
member, 75
message passing, 48
MIMD, 49
MISD, 49
Modifier, 89
Monitor, 51
MPI, 95–102
MPI_Bcast, 100
MPI_Comm_rank, 96
MPI_Comm_size, 96
MPI_Recv, 98
MPI_Reduce, 99
MPI_Scatter, 100
MPI_Send, 97

NC, *siehe* Nick's Class
Nebeneffekt, *siehe* Seiteneffekt
Nichtterminal, 117
Nick's Class, 49
Normalenreihenfolge, 29
Num, 67

Ord, 67

Parallelität, 47–53
Paterson-Wegman-Unifikationsalgorithmus
 12
PGAS, 80

- Pipelining, 47
- Polymorphie, 7
- Polynome, 66
- Prädikatenlogik, 117
- Präzedenzregeln, 90
- PRAM-Modell, 48
- Produktionsregel, 117
- Programm, 3
- Programmiersprache, 3
 - höhere, 4
- Programmierung
 - funktionale, 5
 - imperative, 5
 - logische, 6
 - prozedurale, 5
- Prolog, 73–79
- Punkt-zu-Punkt-Kommunikation,
 - 52
- Quantor, 117
- Race-Condition, *siehe* Wettlaufsituation
- Rang, 52
- Rechtsableitung, 118
- Redex, 28
- reduce, 18
- Reduktion, 28–29
 - Alpha (α), 28
 - Beta (β), 28
 - Eta (η), 29
- Rekursion, 15–17
- Scala, 81–85
- Schlussstrich, 41
- Seiteneffekt, 10
- Selbstapplikation, 44
- Semaphore, 50
- Short-circuit evaluation, 29
- signed, 89
- SIMI, 49
- SISD, 49
- SPARC, 4
- Speicher
 - dynamischer, 111
- split, 76
- splitWhen, 67
- Sprache, 118
 - domänenspezifische, 4
- Startsymbol, 117
- Stream, 66
- Stringtabelle, 105
- Syntaxbaum
 - abstrakter, 107
 - attributeriter, 107
- tail, 66
- tail recursive, 17
- Token, 105
- Turingkombinator, 34
- Typ, *siehe* Datentyp
- type, 64
- Typinferenz, 40, 61
- Typisierung
 - dynamische, 7
 - explizite, 8
 - implizite, 8
 - statische, 7
 - strukturelle, 9
- Typisierungsstärke, 7
- Typkontext, 40
- Typschema, 35
- Typschemainstanziierung, 36

Typsubstitution, 41

Typvariable, 40

Unifikation, 10

Unifikator

 allgemeinster, 11

Union-Find-Algorithmus, 12

unsigned, 89

Unterversorgung, 65

val, 83

var, 83

Variable

 freie, 25, 28

 gebundene, 28

verzahnt, 48

Von-Neumann-Architektur, 48

Wettlaufsituation, 50

where, 60

Wirkung, *siehe* Seiteneffekt

X10, 87–88

x86, 3

Y-Kombinator, 33

zip, 70

zipWith, 66, 66, 70