Skript zur Vorlesung

Prinzipien von Programmiersprachen

SS 2012

Prof. Dr. Michael Hanus

Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Version vom 29. Juni 2012

Vorwort

In dieser Vorlesung werden grundlegende Prinzipien heutiger Programmiersprachen vorgestellt. Dabei steht die praktische Anwendung von Sprachkonzepten zur Unterstützung der Erstellung zuverlässiger Softwaresysteme im Vordergrund.
Bei der Programmierung kommt es weniger darauf an, irgendein Programm zu schreiben, das eine gegebene Aufgabe löst. Vielmehr muss das Programm so geschrieben sein, dass es verständlich und damit wartbar ist, und es muss auch an neue Anforderungen leicht anpassbar sein. Daher ist es wichtig, die für die Problemstellung geeigneten Programmiersprachen und Sprachkonstrukte zu verwenden. Leider gibt es nicht die für alle Probleme gleich gut geeignete universelle Programmiersprache. Daher ist es wichtig zu wissen, welche Sprachkonzepte für welche Problemstellungen geeignet sind. Diese Vorlesung soll hierzu einen Beitrag leisten, indem ein Überblick über wichtige Sprachkonzepte moderner Programmiersprachen gegeben wird. Dadurch werden die Studierenden in die Lage versetzt, sich einerseits schnell in unbekannte Programmiersprachen einzuarbeiten (da viele Konzepte in den verschiedenen Sprachen immer wieder vorkommen), andererseits sollen sie verschiedene Sprachen und Sprachkonzepte aufgrund iher Eignung für ein Softwareproblem kritisch beurteilen können.
Dieses Skript ist eine überarbeitete Fassung der Mitschrift, die ursprünglich von Jürgen Rienow im WS 2002/2003 in LaTeX gesetzt wurde. Ich danke Jürgen Rienow für die erste LaTeX-Vorlage und Fabian Reck und Christoph Wulf für Korrekturhinweise.

Kiel, Juni 2012 Michael Hanus

P.S.: Wer in diesem Skript keine Fehler findet, hat sehr unaufmerksam gelesen. Ich bin für alle Hinweise auf Fehler dankbar, die mir persönlich, schriftlich oder per e-mail mitgeteilt werden.

Inhaltsverzeichnis

1 Einführung

In diesem einführenden Kapitel wollen wir einige grundlegende Begriffe im Zusammenhang mit dieser Vorlesung erläutern.
Eine **Programmiersprache** ist eine Notation für Programme, d.h. eine (formale) Beschreibung für Berechnungen. In diesem Sinn ist ein **Programm** eine Spezifikation einer Berechnung und die **Programmierung** ist die Formalisierung von Algorithmen und informellen Beschreibungen.
Jede Programmiersprache basiert auf einem bestimmten **Berechnungsmodell**, welches beschreibt, wie Programme abgearbeitet werden. Daher können wir Programmiersprachen auch nach diesen Modellen klassifizieren.

Klassifikation von PS nach Berechnungsmodellen

- **Maschinensprachen:** von Neumann-Rechner (Register, Speicher,...)
- **Assembler:** wie Maschinensprachen, aber
  - Abstraktion von Befehlscode (symbolische Namen)
  - Abstraktion von Speicheradressen (symbolische Marken)
- **höhere Programmiersprachen:** Abstraktion von Maschinen, am Problem orientiert.

  Abstraktionen:
  - Ausdrücke (mathematische Notation)
  - Berechnungseinheiten (Prozeduren, Objekte, ...)
- **imperative PS:** Berechnung ≈ Folge von Zustandsänderungen.
  Zustandsänderung ≈ Zuweisung, Veränderung einer Speicherzelle (immer noch von-Neumann-Rechner!)
- **funktionale PS:**
  Programm ≈ Menge von Funktionsdefinitionen
  Berechnung ≈ Ausrechnen eines Ausdrucks
- **logische PS:**
  Programm ≈ Menge von Relationen
  Berechnung ≈ Beweisen einer Formel
- **nebenläufige PS:**
  Programm ≈ Menge von Prozessen, die miteinander kommunizieren
  Berechnung ≈ Kommunikation, Veränderung der Prozeßstruktur

Programmierparadigmen:

Art der Programmierung / Strukturierung
Häufig: bestimmt durch PS ⇒ imperatives, funktiones, logisches (Programmier-) Paradigma

Aber: ein Programmierparadigma ist nicht zwangsläufig an eine bestimmte Programmiersprache gebunden:

- funktional in imperativer PS programmieren
- imperativ in funktionaler/logischer PS programmieren (Zustände als Parameter durchreichen)

Weitere Paradigmen:

- **Strukturierte Programmierung** (Pascal)
  - Zusammenfassen von Zustandsübergangsfolgen zu Prozeduren
  - keine Sprünge („gotos considered harmful“ [4]), sondern nur: Zuweisung, Prozeduraufrufe, bedingte Anweisung, Schleifen, (Ausnahmebehandlung)
- **modulare Programmierung** (Modula-x, Ada)
  - Programm ≈ Menge von Modulen
  - Modul ≈ Schnittstelle und Implementierung
  - Benutzung eines Moduls ≈ Benutzung der Schnittstellenelemente
- **objektorientierte Programmierung (Simula, Smalltalk, C++, Eiffel, Java)**
  - Programm: Menge von Klassen ≈ Schema für Objekte
  - Objekte: Zustand und Prozeduren (≈ Modul)
  - Berechnung: Nachrichtenaustausch zwischen Objekten (≈ Prozeduraufruf)

Elemente von Programmiersprachen

- **Syntax:** Beschreibung (einer Obermenge) der zulässigen Zeichenfolgen (meistens durch reguläre Ausdrücke und kontextfreie Grammatiken).
- **Semantik:** Beschreibung der Bedeutung von Zeichenfolgen
  - *statische Semantik:* Eigenschaften von Programmen, die unabhängig von der Ausführung sind, z.B. Einschränkungen der Syntax:
    - „Bezeichner müssen deklariert werden.“
    - „Bei Zuweisung müssen die linke und rechte Seite den gleichen Typ haben.“

  - *dynamische Semantik:* Was passiert bei der Programmausführung?

- **Pragmatik:** Anwendungsaspekte der Sprache, die außerhalb der Semantikbeschreibung liegen, z.B.
  - Wie werden Gleitpunktzahlen addiert?
  - Wie soll man bestimmte Sprachelemente anwenden?

Aspekte von PS

- Sprachbeschreibung sollte formal sein ⇒ PS kann eindeutig angewendet werden. Meistens: formale Syntax, informelle Semantik (Gefahr!)
- Implementierbarkeit: PS sollte einfach zu implementieren sein, d.h.:
  - preiswerte Übersetzer (vgl. Ada!)
  - schnelle Übersetzer
  - geringere Fehleranfälligkeit von Übersetzern
  - schnelle Zielprogramme
- Anwendbarkeit:
  - keine ungewohnten Schreib- und Denkweisen
  - problemorientierte Programmierung, geringe Rücksicht auf Implementierungsanforderungen (nicht Hardware-orientiert)
  - leichte Integration mit „externer Welt“ (Betriebssystem, Graphik, Bibliotheken, andere Programme,...)
- Robustheit:
  - sicherer Programmablauf oder welche Laufzeitfehler können auftreten? (Typprüfung, Speicherverwaltung,...)
  - können Programme robust gemacht werden? (z.B. durch Fehlerbehandlung)
- Portabilität:
  - laufen die Programme auf vielen Rechnern / Rechnertypen / Betriebssystemen?
  - Vermeidung von Rechner/Betriebssystem-spezifischen Konstrukten (I/O, Parallelität)
  - besser: Bibliotheken (implementiert auf verschiedenen Rechnern / Betriebssystemen)
- Orthogonalität: Unabhängigkeit verschiedener Sprachkonstrukte
  - wenig Konstrukte

- universell kombinierbar
  (Beispiel: Typen und Funktionen orthogonal ⇒ keine Restriktionen bei Parameter- / Ergebnistypen)
- einfache (verständliche) Sprache
- einfache Implementierung
- häufig: spezielle Syntax für häufige Kombinationen („syntaktischer Zucker“)

Was stellen PS zur Verfügung?

(im Vergleich zur „nackten“ Maschine, d.h. Maschinensprache)

1. **Berechnungsmodell:** Dies kann wie die zu Grunde liegende Maschine sein, aber auch sehr verschieden davon, wie funktionale, logische oder nebenläufige Sprachen
2. **Datentypen und Operationen:** problemorientierte Datentypen, z.B. Zahlen in unterschiedlichen Ausprägungen, Zeichen (-ketten), Felder, Verbünde,...
3. **Abstraktionsmöglichkeiten:** keine Aneinanderreihung von Grundoperationen, sondern Strukturierung dieser:
   - Funktionen: Abstraktion komplexer Ausdrücke
   - Prozeduren: Abstraktion von Algorithmen
   - Datentypen: Abstraktion komplexer Strukturen
   - Objekte: Abstraktion von Strukturen, Zuständen und Operationen
4. **Programmprüfung:** Möglichkeiten, bestimmte Programmeigenschaften zur Übersetzungszeit zu prüfen und zu garantieren, wie z.B. Initialisierung von Variablen, Typprüfung, Deadlockfreiheit,...

Eine Anmerkung zur *Effizienz*: Dieser Aspekt steht nicht im Fokus von Hochsprachen, denn es ist klar, dass die Verwendung von Hochsprachen Kosten verursacht (Übersetungszeit, Laufzeit, Speicherplatz etc). Diese Kosten werden aber aufgewogen durch die schnellere Programmentwicklung, bessere Wartbarkeit, Zuverlässigkeit und Portabilität von Programmen, die in einer Hochsprache geschrieben werden. Hierzu ein Zitat von Dennis Ritchie, einem der maßgeblichen Entwickler von C und Unix: „Auch wenn Assembler schneller ist und Speicher spart, wir würden ihn nie wieder benutzen.“

Ziele und Inhalte der Vorlesung:

- Vorstellung wichtiger Sprachkonzepte und Programmierparadigmen:
  - imperative PS
  - Konzepte zur Programmierung im Großen
  - funktionale PS
  - logische PS

  - nebenläufige und verteilte Programmierung

- multilinguales Programmieren
- einfaches Erlernen neuer Programmiersprachen
- kritische Beurteilung von PS hinsichtlich ihrer Eignung für bestimmte Anwendungen
- eigener Sprachentwurf („special purpose language“ / „domain-specific languages“)

2 Grundlagen

In diesem Kapitel behandeln wir einige Grundlagen, die für alle höheren Programmiersprachen relevant sind.

2.1 Beschreibung der Syntax

Die übliche Formalismen zur Beschreibung der Syntax von Programmiersprachen sind *formale Grammatiken*:

- reguläre Grammatiken für Grundsymbole (Zahlen, Bezeichner, Schlüsselwörter,...)
- kontextfreie Grammatiken für komplexe Strukturen (Klammerstrukturen, Blöcke)
- Kontextabhängigkeiten werden häufig in der statischen Semantik festgelegt (informell oder mittels Inferenzsystemen)

Zur textuellen Notation von Grammatiken wird überwiegend die EBNF (erweiterte Backus-Naur-Form) benutzt. Daher definieren wir zunächst die Backus-Naur-Form (BNF), die erstmals für die Beschreibung von Algol-60 verwendet wurde.

Definition 2.1 (BNF (Backus-Naur-Form)) *Die Backus-Naur-Form (BNF) enthält folgende Metasymbole:*

- *"`::=`" definiert Regeln für Nichtterminalsymbole*
- *"`|`" Regelalternative*
- *"`<...>`" Nichtterminalsymbole*

Alles andere sind Terminalsymbole (die manchmal auch durch Fettdruck oder Unterstreichen gekennzeichnet werden).

Beispiel: Grammatik für Ausdrücke

```
<exp>    ::= <number> | <id> | <exp> <op> <exp> | ( <exp> )
<op>     ::= + | - | * | /
<id>     ::= <letter> | <id> <letter> | <id> <digit>
<number> ::= <digit> | <number> <digit>
<letter> ::= a | b | c | ... | z
<digit>  ::= 0 | 1 | 2 | ... | 9
```

Definition 2.2 (EBNF (Erweiterte Backus-Naur-Form)) *Die EBNF unterscheidet sich von der BNF wie folgt:*

- *Nichtterminalzeichen beginnen mit Großbuchstaben*
- *Terminalzeichen werden quotiert (z.B. `'+'`) oder ohne weitere Angaben direkt notiert, wenn eine Verwechslung mit andere Symbolen ausgeschlossen ist.*
- *Metasymbole:*
  - *"`::=`", "`|`" wie in BNF*
  - *"`(...)`": Gruppierung von Grammatikelementen*
  - *"`{...}`": beliebig viele Vorkommen (auch 0!)*
  - *"`[...]`": optionales Vorkommen (d.h. 0- oder 1-mal)*

Beispiel:

```
Exp    ::= Number | Id | Exp Op Exp | '(' Exp ')'
Op     ::= + | - | * | /
Id     ::= Letter { Letter | Digit }
Number ::= Digit { Digit }
```

Problem: Mehrdeutigkeiten von Strukturbäumen / Ableitungen.
Ableitungsbaum (vereinfacht!) für "`2*3+v`":

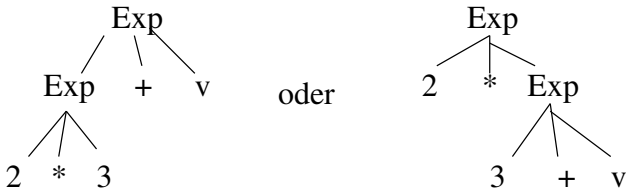


Abbildung 2.1: Mehrdeutigkeiten von Strukturbäumen / Ableitungen (verkürzte Darstellung).

Semantisch ist der Unterschied zwischen beiden Bäumen relevant!
⇒ Auflösen von Mehrdeutigkeiten:

- durch eindeutige Grammatiken ($LL(k)$, $LR(k)$: vgl. Compilerbau)
  Beispiel: Unsere Grammatik für Ausdrücke kann wie folgt in eine eindeutige Grammatik umgewandelt werden:

  ```
  Exp    ::= Term   { ( + | - ) Term }
  Term   ::= Faktor { ( * | / ) Faktor }
  Faktor ::= '(' Exp ')' | Number | Id
  ```

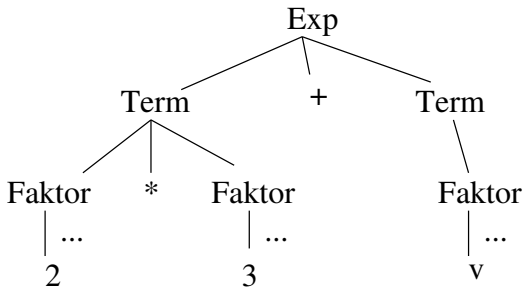


Abbildung 2.2: Auflösen von Mehrdeutigkeiten im Strukturbaum (verkürzte Darstellung).

Ableitungsbaum (vereinfacht) für "`2*3+v`":

- durch Prioritätsregeln („Punktrechnung vor Strichrechnung")
  - spezielle „Operatorpräzedenzgrammatiken"
  - Transformation von Grammatiken, um Prioritäten zu codieren (s.o.)
  - in statischer Semantik auflösen

Definition 2.3 (Syntaxdiagramm) *Ein* Syntaxdiagramm *ist eine graphische Darstellung kontextfreier Grammatiken.*

- *für jedes Nichtterminal: ein gerichteter Graph*
- *eckige Box: Nichtterminalsymbol*
- *runde Box: Terminalsymbol*
- *Ableitung für ein Nichtterminalsymbol: Graphdurchlauf von links nach rechts*

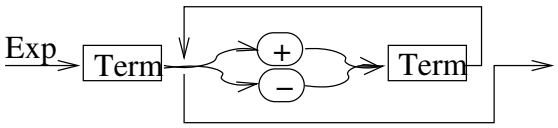


Abbildung 2.3: Beispiel für ein Syntaxdiagramm

2.2 Methoden zur Semantikbeschreibung

Bei der Semantikbeschreibung geht es darum, die Bedeutung der syntaktischen Konstrukte zu beschreiben.

- informell, umgangssprachlich: Problem: Eindeutigkeit
- besser: formale Beschreibung:
  - axiomatische Semantik: Vor- / Nachbedingungen für jedes Konstrukt der PS (Hoare-Logik)
  - denotationelle Semantik: Abbildung: syntaktisches Konstrukt $\mapsto$ semantisches Objekt (vgl. Vorlesung: „Semantik von Programmiersprachen“)
  - operational: erstelle Modell des Berechnungsraumes und beschreibe darauf in einer formalen Weise die Berechnungsabläufe

2.2.1 Strukturierte Operationale Semantik (SOS, [11])

Wir betrachten als erstes Beispiel die Beschreibung der Semantik einer einfachen imperativen Sprache, deren Syntax wie folgt definiert ist (das Nichterminalsymbol `S` steht für „Statement“):

```
S ::= skip
    | Var := Exp
    | S ; S
    | if Exp then S else S
    | while Exp do S
```

Imperative Programme manipulieren Werte von Variablen. Um dies zu beschreiben, benötigen wir einen **Zustand**, der als Abbildung $\sigma : Var \to Int \cup Bool \cup ...$ jeder Variablen einen Wert zuweist.
Ein **Berechnungsschritt** arbeitet die Anweisungen schrittweise ab und manipuliert dabei den Zustand. Daher können wir einen Berechnungsschritt formal als **Zustandsübergang**

$$\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$$

beschreiben.
Eine **Berechnung** beginnt in einem **Startzustand** (S ist das abzuarbeitende Programm)

$$\langle S, \sigma_{initial} \rangle$$

und endet in einem **Endzustand**

$$\langle \texttt{skip}, \sigma \rangle$$

Wir beschreiben die Zustandsübergangsrelation „$\to$“ über den Aufbau von S (aus diesem Grund nennt man diese Form auch *strukturierte* operationale Semantik). Hierbei nehmen wir an, dass wir eine **Interpretationsfunktion** I zum Ausrechnen von Ausdrücken in einem Zustand haben, d.h. der *Wert eines Ausdrucks e in einem gegebenen Zustand* σ wird mit

$$I[\![e]\!]\sigma$$

bezeichnet. Die genaue Definition von I erfolgt später.

Transitionsaxiome und -regeln zur Interpretation von Anweisungen:
Axiom zur Abarbeitung einer *Zuweisung*:

$$\langle v := e, \sigma\rangle \to \langle \texttt{skip}, \sigma[v/I[\![e]\!]\sigma]\rangle$$

hierbei wird durch die Notation $\sigma[v/w]$ die *Abänderung* einer Funktion σ an der Stelle v bezeichnet, d.h. dies ist eine neue Funktion, die wie folgt definiert ist:

$$\sigma[v/w])x = \begin{cases} w, & \text{falls} \quad x = v \\ \sigma(x), & \text{sonst} \end{cases}$$

Axiom und Regel zur Abarbeitung einer *Sequenz von Anweisungen*:

$$\langle \texttt{skip}; S, \sigma\rangle \to \langle S, \sigma\rangle$$

$$\frac{\langle S_1, \sigma\rangle \to \langle S_2, \sigma'\rangle}{\langle S_1; S, \sigma\rangle \to \langle S_2; S, \sigma'\rangle}$$

(d.h. die Anweisung `skip` hat keine Wirkung und bei einer Sequenz muss erst die linke Anweisung abgearbeitet werden)

Axiome zur Abarbeitung einer *bedingten Anweisung*:

$$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma\rangle \to \langle S_1, \sigma\rangle \qquad \text{falls } I[\![b]\!]\sigma = true$$

$$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma\rangle \to \langle S_2, \sigma\rangle \qquad \text{falls } I[\![b]\!]\sigma = false$$

Axiome zur Abarbeitung einer *Schleife*:

$$\langle \text{while } b \text{ do } S, \sigma\rangle \to \langle \texttt{skip}, \sigma\rangle \qquad \text{falls } I[\![b]\!]\sigma = false$$

$$\langle \text{while } b \text{ do } S, \sigma\rangle \to \langle S; \text{while } b \text{ do } S, \sigma\rangle \qquad \text{falls } I[\![b]\!]\sigma = true$$

Damit können wir eine **Berechnung** bezüglich eines Programms S_0 und einem Anfangszustand σ_0 als Folge

$$\langle S_0, \sigma_0\rangle \to \langle S_1, \sigma_1\rangle \to \langle S_2, \sigma_2\rangle \to \dots$$

definieren, wobei jeder einzelne Schritt $\langle S_i, \sigma_i\rangle \to \langle S_{i+1}, \sigma_{i+1}\rangle$ aus den obigen Axiomen und Regel folgt. Wir sprechen von einer **terminierenden Berechnung**, falls ein i existiert mit $S_i = \texttt{skip}$, ansonsten ist die Berechnung **nichtterminierend** (üblich: „Endlosschleife“).

2.2.2 Exkurs: Inferenzsysteme

Die obige operationale Semantik wurde durch Axiome und Regeln, d.h. einem *Inferenzsystem* definiert. Da Inferenzsysteme ein zentrales Mittel zur Beschreibung der statischen und dynamischen Semantik von Programmiersprachen sind, ist es wichtig, deren Bedeutung genau zu verstehen. Aus diesem Grund wiederholen wir an dieser Stelle aus der Logik die Begriffe Inferenzsysteme und Ableitbarkeit.

Definition 2.4 (Inferenzsystem) *Ein* Inferenzsystem (Deduktionssystem, Kalkül) *ist ein Paar* $IR = (BL, R)$ *mit*

- BL *(base language): Sprache, d.h. Menge von zulässigen Sätzen über einen bestimmten Alphabet*
- $R = \{R_i\}_{i \in I}$ *(*I *Indexmenge) ist eine Familie von Mengen* $R_i \subseteq BL^{n_i}$ *mit* $n_i > 0$ *(d.h.* n_i*-fache kartesische Produkte), wobei wir die beiden folgenden grundlegenden Klassen unterscheiden:*
  - $n_i = 1$*: Dann heißt* R_i Axiomenschema
  - $n_i > 1$*: Dann heißt* R_i Regel- oder Inferenzschema*. Statt* $(S_1, \ldots, S_{n_i}) \in R_i$ *schreiben wir auch:*
    $$\frac{S_1 \ldots S_{n_i - 1}}{S_{n_i}}$$
    (gelesen als: „aus $S_1 \ldots S_{n_i - 1}$ *folgt* S_{n_i}*“).*

Definition 2.5 (Ableitbare Sätze) *Die Menge aller* Folgerungen *oder* beweisbaren / ableitbaren Sätze *bzgl. eines Inferenzsystems* IR *ist die kleinste Menge* F *mit:*

- *Ist* R_i *ein Axiomenschema und* $S \in R_i$*, dann ist* $S \in F$*.*
- *Ist* R_i *Inferenzschema und* $(S_1, \ldots, S_{n_i}) \in R_i$ *und* $S_1, \ldots, S_{n_i - 1} \in F$*, dann ist auch* $S_{n_i} \in F$ *(„falls die Voraussetzungen beweisbar sind, dann ist auch die Folgerung beweisbar“).*

Übliche Einschränkungen für Inferenzsysteme:

1. Die Indexmenge I ist endlich (d.h. es gibt nur endliche viele Schemata).
2. Jedes R_i ist entscheidbar.

Unter diesen Voraussetzungen gilt:

- die Beweise sind effektiv überprüfbar (d.h. es ist entscheidbar, ob ein Beweis für eine Folgerung richtig ist)
- die Menge aller Folgerungen ist aufzählbar

Es ist allerdings in der Regel nicht entscheidbar, ob ein Satz aus dem Inferenzsystem ableitbar ist.

Beispiel:
Wir schauen uns noch einmal die oben angegebene Definition für die operationale Semantik imperativer Programme an. Diese Definition wird als Inferenzsystem wie folgt interpretiert:

- BL enthält Sätze der Form

$$\langle S_1, \sigma_1 \rangle \to \langle S_2, \sigma_2 \rangle$$

  wobei S_1, S_2 aus dem Nichtterminal `S` ableitbar ist und σ_1, σ_2 textuelle Darstellungen von Zuständen sind.

- Axiomenschema: Betrachte z.B. das Schema

$$\langle \texttt{skip}; S, \sigma \rangle \to \langle S, \sigma \rangle$$

  Hierbei sind S und σ *Schemavariablen*, d.h. sie stehen für die Menge der Anweisungen und Zustände.

- Ein Regelschema ist z.B. bei der Sequenz zu finden:

$$\frac{\langle S_1, \sigma \rangle \to \langle S_2, \sigma' \rangle}{\langle S_1; S, \sigma \rangle \to \langle S_2; S, \sigma' \rangle}$$

  Hier sind $S_1, S_2, S, \sigma, \sigma'$ Schemavariablen.

Somit ist unsere Schreibweise von Axiomen und Regeln nur eine endliche, kompakte Darstellung für eine unendliche Menge von Axiomen und Regeln. Daher werden diese auch Schemata genannt.

Behauptung: Wir wollen zeigen, dass

$$\langle \texttt{x:=1; y:=2}, \{\} \rangle \to \langle \texttt{skip; y:=2}, \{\texttt{x} \mapsto 1\} \rangle \to \langle \texttt{y:=2}, \{\texttt{x} \mapsto 1\} \rangle \to \langle \texttt{skip}, \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 2\} \rangle$$

eine korrekte Berechnung ist.

Beweis: Es gilt:
$\langle \texttt{x:=1}, \{\} \rangle \to \langle \texttt{skip}, \{\texttt{x} \mapsto 1\} \rangle$ ist ein Axiom, also ableitbar.
Die Anwendung der Regel mit diesem ableitbaren Satz als Voraussetzung ergibt:

$$\langle \texttt{x:=1; y:=2}, \{\} \rangle \to \langle \texttt{skip; y:=2}, \{\texttt{x} \mapsto 1\} \rangle$$

ist ableitbar. Damit ist der erste Schritt korrekt.
Der zweite Schritt

$$\langle \texttt{skip; y:=2}, \{\texttt{x} \mapsto 1\} \rangle \to \langle \texttt{y:=2}, \{\texttt{x} \mapsto 1\} \rangle$$

ist ein Axiom und damit auch korrekt.
Der dritte Schritt

$$\langle \texttt{y:=2}, \{\texttt{x} \mapsto 1\} \rangle \to \langle \texttt{skip}, \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 2\} \rangle$$

ist ebenfalls ein Axiom und damit auch korrekt.

2.2.3 Interpretation von Ausdrücken

In der obigen Beschreibung der Semantik von Anweisungen ist die Interpretation von Ausdrücken, d.h. die Definition bzw. Berechnung von $I[\![e]\!]\sigma$ noch offen. Aus diesem Grund definieren wir nun die Interpretation von Ausdrücken präzise mittels eines Inferenzsystems.
Die Struktur der Basissprache dieses Inferenzsystems ist etwas anders als bei Anweisungen, da wir hier nicht an den einzelnen Schritten interessiert sind, sondern nur an dem Wert eines Ausdrucks. Aus diesem Grund verwenden wir eine Form, die auch als „**Natürliche Semantik**“ (natural semantics) bezeichnet wird. Eine Semantikbeschreibung mittels in der Form der natürlichen Semantik basiert auf folgenden Ideen:

- Die Basissprache hat die Form $\sigma \vdash e : v$ („im Zustand σ hat e den Wert v“)
- Somit erfolgt eine direkte Zuordnung des vollständigen Ergebnisses der Berechnung (im Gegensatz zur Semantik von Anweisungen, wo die einzelnen Schritte zu einer Berechnungssequenz kombiniert werden müssen).
- Die Definition erfolgt über den syntaktischen Aufbau von e

Syntax der Ausdrücke (die konkrete Syntax wird hier zu einer „abstrakten Syntax“ vereinfacht, was eine übliche Methode ist):

```
Exp ::= n
      | x
      | Exp op Exp
```

Hierbei ist `n` eine Zahl, `x` eine Variable und `op` $\in \{$`+`, `*`$\}$.
Die Definition der Semantik der Ausdrücke erfolgt durch ein Inferenzsystem. Die Basissprache ist dabei $\sigma \vdash e : v$, wobei e aus `E` ableitbar ist, σ ist die textuelle Repräsentation eines Zustandes, und v ist ein Wert (eine Zahl).

$$\sigma \vdash n : n \qquad \text{wobei } n \text{ eine Zahl ist}$$

$$\sigma \vdash x : \sigma(x) \qquad \text{wobei } x \text{ eine Variable ist}$$

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma \vdash e_2 : v_2}{\sigma \vdash e_1\texttt{+}e_2 : v_1 + v_2}$$

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma \vdash e_2 : v_2}{\sigma \vdash e_1\texttt{*}e_2 : v_1 \cdot v_2}$$

Hierbei ist zu beachten, dass in der Basissprache vor dem “`:`” ein Ausdruck und dahinter eine Zahl steht. Dies bedeutet, dass in der Inferenzregel für die Addition bei $\sigma \vdash e_1\texttt{+}e_2 : v_1 + v_2$ das erste Plussymbol “`+`” ein syntaktisches Symbol ist, während das zweite Plussymbol “+” für die mathematische Funktion steht, die an dieser Stelle ausgerechnet werden muss. Dies gilt ebenso für die Multiplikationsregel, wo zur Verdeutlichung unterschiedliche Symbole gewählt wurden. Zum Beispiel ist

$$\frac{\{\} \vdash 2 * 3 : 6 \quad \{\} \vdash 3 : 3}{\{\} \vdash 2\texttt{*}3\texttt{+}3 : 9}$$

eine Beispielinstanz der Multiplikationsregel. Wenn wir „6+3“ an Stelle der „9“ geschrieben hätten, wäre dies keine korrekte Regelinstanz, da der Ausdruck „6+3“ keine Zahl ist!
Der Vorteil dieses Inferenzsystems ist die leichte Erweiterbarkeit auf komplexere Formen von Ausdrücken. Als Beispiel betrachten wir die Erweiterung um lokale Bindungen, z.B. in Form von let-Ausdrücken. Wir erweitern hierzu die Syntax um

```
Exp ::= ...
      |  let x = Exp in Exp
```

(hierbei ist `x` eine Variable).
Die Bedeutung von let-Ausdrücken kann durch eine weitere Inferenzregel festgelegt werden:

$$\frac{\sigma \vdash e_1 : v_1 \quad \sigma[x/v_1] \vdash e_2 : v_2}{\sigma : \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : v_2}$$

Diese Regel beschreibt, dass der Wert von x nur lokal gültig in e_2 ist. Z.B. ist

$$\{x \mapsto 6\} \vdash \texttt{let x=2+2 in x*x} : 16$$

ableitbar (und nicht mit dem Ergebnis 36 statt 16!).
Nun können wir die Interpretation von Ausdrücken wie folgt definierten:

Es gilt $I[\![e]\!]\sigma$ falls $\sigma \vdash e : v$ ableitbar ist.

Man beachte, dass man bei der Sprachdefinition auch mehrere Inferenzsysteme für verschiedene Aspekte oder Sprachanteile benutzen kann. In der Regel ist das jeweilige zu benutzende Inferenzsystem an der Form der abzuleitenden Sätze (d.h. der Basissprache) erkennbar.

2.2.4 Datentypen

In der bisheriger Semantik imperative Programme ist noch die Bedeutung der Basistypen (z.B. Int, Bool) und ihrer Operationen offen, d.h. es ist noch nicht genau spezifiziert, wie die semantischen Operationen o in “$\sigma \vdash e_1\ op\ e_2 : v_1\ o\ v_2$” definiert sind. Eine Methode zur *implementierungsunabhängigen* Spezifikation von Datentypen und ihren Operationen sind *abstrakte Datentypen*. „Abstrakt“ bedeutet hierbei, dass die Implementierung nicht festgelegt ist, sondern nur die Bedeutung der Operationen ist wichtig ist.

Definition 2.6 (Abstrakte Datentypen (ADTs)) *Ein ADT besteht aus einer* ***Signatur*** Σ*, was eine Menge von* ***Operatoren*** $f : \tau_1, \ldots, \tau_n \to \tau \in \Sigma$ *ist, wobei* $\tau, \tau_1, \ldots, \tau_n$ *Typen sind. Der Operator* f *bildet Elemente aus* $\tau_1, \ldots, \tau_n$ *auf ein Element aus* τ *ab. Spezialfall:* $n = 0$*:* $c :\to \tau$*: Dann nennen wir* c *eine* ***Konstante****.*

Beispiel: Wahrheitswerte

```
datatype Bool
  true, false :            → Bool
  ¬           : Bool       → Bool
  ∨, ∧        : Bool, Bool → Bool
  =, ≠ , <, > : Int, Int   → Bool
end
```

Beispiel: ganze Zahlen

```
datatype Int
  ...,-2,-1,0,1,2,... : → Int
  +,-,*,div,mod : Int, Int → Int
end
```

Nun können wir (wohlgeformte) Ausdrücke über Datentypen definieren. Dazu nehmen wir an, dass X eine Menge getypter Variablen (disjunkt zu Operatoren und Konstanten) ist, d.h. $x : \tau \in X$ bedeutet „x ist Variable vom Typ τ“

Definition 2.7 (Ausdrücke / Terme) *Die Menge aller Ausdrücke / Terme wird mit $T(\Sigma, X)$ bezeichnet. Ein Ausdruck / Term vom Typ τ kann folgendes sein:*

- ***Variable:*** $x \in T(\Sigma, X)$*, falls* $x : \tau \in X$
- ***Konstanten:*** $c \in T(\Sigma, X)$*, falls* $c :\rightarrow \tau \in \Sigma$
- ***zusammengesetzter Term:*** $f(t_1, \ldots, t_n) \in T(\Sigma, X)$*, falls* $f : \tau_1, \ldots, \tau_n \rightarrow \tau \in \Sigma$ *und* $t_i \in T(\Sigma, X)$ *vom Typ* τ_i *(für alle* $i \in \{1, \ldots, n\}$*)*
- ***Grundterm:*** *Term ohne Variablen*

Beispiele:
Terme vom Typ Int:

-2 $\quad 1 + 3 * 4$ (Infixdarstellung für $+(1, *(3, 4))$) $\quad 3 \bmod x$, falls $x : Int \in X$

Terme vom Typ Bool:

`true` $\quad$ `false` $\vee$ `true` $\quad x \neq \wedge x \neq 1$

Bisher haben wir nur die Syntax von Ausdrücken definiert. Was ist aber die Bedeutung dieser Ausdrücke. Intuitiv sollte $1 + 2$ die gleiche Bedeutung wie 3 haben. Zu diesem Zweck definieren wir die Bedeutung durch Gleichungen, um von einer konkreten Implementierung zu abstrahieren:

Definition 2.8 (Gleichung) *Eine* ***Gleichung*** $t_1 = t_2$ *ist ein Paar von Termen gleichen Typs. Falls in einer Gleichung Variablen vorkommen, bezeichnet diese ein Schema, das für alle Gleichungen steht, bei denen Variablen durch passende Werte ersetzt werden können (dies bezeichnet man auch als* ***Instanz*** *des Schemas).*

Beispiel:
Gleichungen für Bool (mit $b : Bool \in X$)

$$\begin{aligned} \neg true &= false \\ \neg false &= true \\ true \wedge b &= b \\ false \wedge b &= false \\ true \vee b &= true \\ false \vee b &= b \end{aligned}$$

Gleichungen für Int (mit $x : Int \in X$)

$$\begin{aligned} 0 + x &= x \\ x + 0 &= x \\ 1 + 1 &= 2 \\ 1 + 2 &= 3 \\ &\vdots \end{aligned}$$

Ausrechnen von Gleichheiten: Wir stellen die üblichen Gleichungsaxiome als Inferenzregeln dar:
Reflexivität:

$$x = x$$

Symmetrie:

$$\frac{x = y}{y = x}$$

Transitivität:

$$\frac{x = y \quad y = z}{x = z}$$

Kongruenz:

$$\frac{x_1 = y_1 \quad \ldots \quad x_n = y_n}{f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)}$$

(für alle Operatoren f)

Axiom:

$$t_1 = t_2$$

(falls $t_1 = t_2$ Gleichung aus der Spezifikation ist, d.h. Gleichungen des ADT sind Axiomenschemata)

Beispiel: Wir wollen zeigen, dass $\neg(true \wedge false) = true$ ableitbar ist:

$$\frac{\dfrac{true \wedge false = false \text{ (Axiom)}}{\neg(true \wedge false) = \neg(false) \text{ (Kongruenz)}} \qquad \neg false = true \text{ (Axiom)}}{\neg(true \wedge false) = true \text{ (Transitivität)}}$$

Somit können wir das **Ausrechnen** von Ausdrücken mit Gleichheitsinferenzen beschreiben. Ein Problem dabei ist, dass es nicht klar ist, was das Berechnungsergebnis sein soll. Z.B. ist auch

$$\neg(true \wedge false) = \neg(false \wedge true)$$

ableitbar, wobei intuitiv der Ausdruck nicht ausgerechnet wurde, weil die rechte Seite vereinfacht werden kann. Um „vereinfachte“ Terme zu charakterisieren, definieren wir den Begriff der Konstruktoren:

Definition 2.9 (Konstruktoren) *Eine Menge C von Konstruktoren eines ADT τ ist eine kleinste Menge C von Operationen mit dem Ergebnistyp τ, so dass für alle Grundterme t vom Typ τ gilt: es existiert ein Grundterm s mit:*

- *$t = s$ ist ableitbar, und*
- *s enthält nur Operationen aus C.*

Beispiel:
Der ADT Bool hat die Konstruktoren $true$ und $false$.
Der ADT Int hat die Konstruktoren ...,-2,-1,0,1,2,...

Intuition:

- Konstruktoren erzeugen Daten
- andere Operationen sind **Selektoren** oder **Verknüpfungen**
- Ausrechnen: Finden eines äquivalenten („gleichen“) Konstruktorterms

Diese Definition ist unabhängig von einer konkreten Implementierung, aber dies ist noch keine konstruktive Rechenvorschrift („Ausrechnen“ nur aufzählbar, aber nicht entscheidbar).

Beispiel: Listen von ganzen Zahlen

```
datatype IntList
  nil  : → IntList
  cons : Int,IntList → IntList
  head : IntList → Int
  tail : IntList → IntList
  empty: IntList → Bool
equations a:Int, l:IntList
  head(cons(a,l))  =  a
  tail(cons(a,l))  =  l
  empty(nil)       =  true
  empty(cons(a,l)) =  false
end
```

Die Konstruktoren des ADT `IntList` sind `nil` und `cons`. Nach unserer Definition wären formal auch `head` und `tail` Konstruktoren, aber `head(nil)` oder `tail(nil)` wird üblicherweise mit einem Fehler identifiziert und ist daher kein Wert des ADT.

Beispiel: Listen von Wahrheitswerten: diese könnte man `IntList` definieren, wobei überall `Int` durch `Bool` ersetzt wird:

```
datatype BoolList
  nil  : → BoolList
  cons : Bool,BoolList → BoolList
  head : BoolList → Bool
  tail : BoolList → BoolList
  empty: BoolList → Bool
equations a:Bool, l:BoolList
  head(cons(a,l))  =  a
  tail(cons(a,l))  =  l
  empty(nil)       =  true
  empty(cons(a,l)) =  false
end
```

Wie man hierbei sieht, ist die Definition von Listen unabhängig von der Art der Elemente. Daher ist es sinnvoll, vom Typ der Elemente zu abstrahieren und diesen als Parameter des ADT zuzulassen. In diesem Fall spricht man auch von einem **parametrisierten ADT** $\tau(\alpha_1, \ldots, \alpha_n)$, wobei die Variablen α_i **Typparameter** sind, die in der ADT-Definition anstelle von Typen vorkommen können. Eine **Instanz eines parametrisierten ADTs** wird mit $\tau(\tau_1, \ldots, \tau_n)$ bezeichnet, wobei $\tau_1, \ldots, \tau_n$ sind. Diese Instanz steht für die textuelle Ersetzung jedes α_i durch τ_i in der ADT-Definition.
Beispiel: Parametrisierte Listen `List(`α`)`

```
datatype List(α)
  nil  : → List(α)
  cons : α,List(α) → List(α)
  head : List(α) → α
  tail : List(α) → List(α)
  empty: List(α) → Bool
equations a:α, l:List(α)
  head(cons(a,l))  =  a
  tail(cons(a,l))  =  l
  empty(nil)       =  true
  empty(cons(a,l)) =  false
end
```

Datentypen in Programmiersprachen

Allgemein: Ein Typ in einer PS entspricht einer Implementierung eines ADT. Sind alle Konstruktoren Konstanten, so heißt dieser Typ **Grundtyp**, ansonsten **zusammengesetzter Typ**.

Viele funktionale Programmiersprachen erlauben eine direkte Implementierung vieler ADTs:

- Die Typdefinition erfolgt durch Angabe der Konstruktoren
- direkte Angabe der Gleichungen
  (dies ist allerdings nur mit folgender Einschränkung möglich: in der linken Seite $f(t_1, \ldots, t_n)$ darf f kein Konstruktor sein und $t_1, \ldots, t_n$ dürfen nur Konstruktoren oder Variablen enthalten)

Beispiel: Parametrisierte Listen in Haskell:

```
data List a = Nil | Cons a (List a)

empty Nil = True
empty (Cons x xs) = False

head (Cons x xs) = x

tail (Cons x xs) = xs
```

2.3 Bindungen und Blockstruktur

Programme bestehen aus bzw. manipulieren verschiedene Einheiten ("entities"), z.B. Variablen, Prozeduren, Klassen, ... Die konkreten Eigenschaften dieser Programmeinheiten werden durch **Attribute** beschrieben.

Beispiele:

- Variablen: (Name, Adresse, Typ, Wert)
  Die Adresse ist wichtig z.B. für sharing, z.B. bei `var`-Parametern (Pascal)
  zusätzlich könnte z.B. Gültigkeitsbereich o.ä. auch als Attribut relevant sein.
- Prozeduren: (Name, Parametertypen, Übergabemechanismen, ...)
- Klassen: (Name, Instanzvariablen, Klassenvariablen, Methoden, ...)

Die konkrete Zuordnung von Programmeinheiten zu Attributen ist abhängig von der jeweiligen Programmierspreche. Unter einer **Bindung** verstehen wir die Festlegung bestimmter Attribute.

Unterschiede bei konkreten Programmiersprachen:

- Welche Einheiten gibt es?

- Welche Attribute haben diese?
- Wann werden die Attributwerte definiert? (**Bindungszeit**)

Bei der Bindung unterschieden wir die

Statische Bindung: Attributwert liegt zur Übersetzungszeit fest (z.B. Typ bei getypter Syntax, Wert bei Konstanten)

Dynamische Bindung: Attributwert ist erst zur Laufzeit festgelegt (z.B. Werte von Variablen).

Beachte: statisch/dynamisch kann für jedes Attribut eventuell unterschiedlich sein.

Beispiel:

statisch getypte PS: Typen von Bezeichnern liegen zur Übersetzungszeit fest (z.B. Pascal, Modula, Java).

dynamisch getypte PS: Typen der Bezeichner werden erst zur Laufzeit bekannt (z.B. Lisp, Prolog, SmallTalk, Scriptsprachen wie Tcl, PHP, Ruby, ...)

ungetypte PS: Variablen haben kein Attribut „Typ“ (z.B. Assembler, z.T. auch C)

Beispiel:
in Modula-3: `FOR i:=1 TO 10 DO ...`
Hier hat i den Typ INTEGER.

in Scheme: `(define (p x) (display x))`
Hier ist der Typ von `x` erst zur Laufzeit bekannt, z.B. im Aufruf "`(p 1)`" hat `x` den Typ Zahl, währen im Aufruf `(p (list 1 2))` `x` den Typ List hat

ungetypt: `x:=65; printchar(x)` $\rightsquigarrow$ kein Typ für `x`

Dynamische Bindungen sind eventuell **veränderbar**:

Definition 2.10 (Seiteneffekt) *Ein* Seiteneffekt *ist die Veränderung einer existierenden Bindung. In einer imperativen Sprache ist dies z.B. die Veränderung von Wertbindungen von Variablen. In deklarativen Sprachen gibt es keine Wertänderung, sie sind* ***seiteneffektfrei****.*

Definition 2.11 (Gültigkeitsbereich) *Der* Gültigkeitsbereich *(*scope*) ist eine Folge von Anweisungen / Ausdrücken, in denen Bindungen einer bestimmten Einheit im Programm gültig oder sichtbar sind.*

Beispiel:
Programmiersprache C:
Prozeduren: im gesamten Programm sichtbar
Variablen: Unterschied:
(a) global (im gesamten Programm sichtbar) oder
(b) lokal (definiert in Prozedur, auch Parameter, nur dort sichtbar)

Algol, Pascal: beliebig verschachtelte Gültigkeitsbereiche

Definition 2.12 (Block) *Ein* Block *ist eine Folge von Anweisungen / Ausdrücken, in denen lokal deklarierte Einheiten sichtbar sind.*

Beispiel: Gesamtprogramm, Prozedurrümpfe, Methodendefinitionen in Klassen, For-Schleifen

Sichtbarkeitsregeln einer Programmiersprache beschreiben, welche Bindungen wo sichtbar sind
Die üblich Sichtbarkeitsregel ist: Ein Bezeichner x ist sichtbar an einer Stelle, falls x in diesem (lokaler Bezeichner) oder einem umfassenden Block (globaler Bezeichner) vereinbart wurde.

Definition 2.13 (Umgebung) *Menge aller sichtbaren Bezeichner und deren Bindungen (an einer Stelle)*

Definition 2.14 (Blockstrukturierte Programmiersprache) *PS mit explizitem oder implizitem Konzept von Blöcken*

Beispiele:
Assembler: keine Blöcke
C: nur 2 Hierarchien
Ada: beliebige Schachtelungen

Beispiel: Blöcke in Ada: eingeleitet durch `declare`

```
B: declare a: Real;
           b: Boolean;
   begin
     C: declare b,c: Float;  -- b verdeckt b aus B!
        begin
          c:=a+b;  -- a aus B, b,c aus C
        end C;
     b := (a=3);  -- b aus B
   end B;
```

Prinzip: verwende die innerste umgebende Deklaration eines Bezeichners. Dies erscheint klar bei statischen Blöcken, aber bei Prozeduraufrufen ist dies nicht so einfach. Betrachten wir dazu folgendes Beispiel:

```
declare
  a: Integer;

  procedure p1 is
  begin
    print(a);  -- (*)
  end;
```

```
  procedure p2 is
    a: Integer;  -- lokal in p2 deklariert
  begin
    a:=0;
    p1;
  end;

begin
  a:=42;
  p2;
end;
```

Die Frage ist nun, was an der Stelle `(*)` ausgegeben wird, was gleichbedeutend ist mit der Frage, welche Deklaration von `a` an der Stelle `(*)` im Aufruf von `p1` gemeint ist. Eine eindeutige Antwort ist nicht möglich, denn dies ist abhängig von der Bindungsregel der Programmiersprache. Hier gibt es zwei Möglichkeiten:

Lexikalische (statische) Bindung von Bezeichnern zu Deklarationen (**lexical / static scoping**): Die Bindung stammt aus der Umgebung, in der diese Prozedur (z.B. `p1`) deklariert ist (d.h. die genaue Bindung ist aus dem Programmtext, also statisch, festgelegt).
⇒ in `(*)` ist mit `a` die äußere Deklaration gemeint: Ausgabe: 42

Dynamische Bindung von Bezeichnern zu Deklarationen (**dynamic scoping**): Die Bindung stammt aus der Umgebung, in der diese Prozedur aufgerufen wurde, d.h. die genaue Bindung wird erst dynamisch zur Laufzeit festgelegt.
⇒ in `(*)` ist mit `a` die Deklaration in `p2` gemeint: Ausgabe: 0

Lexikalische Bindung:

- einfacher zu überschauen
- weniger fehleranfällig
- prüfbar durch Compiler
- überwiegend verwendet in PS (Ausnahmen: APL, Snobol-4, Lisp, Emacs-Lisp)

3 Imperative Programmiersprachen

Hauptaspekte:

- Variablen und Zuweisungen
- Standard-Datentypen
- Kontrollabstraktionen
- Prozeduren
- Ausnahmebehandlung

Diese fünf Hauptaspekte sind *Kern jeder imperativen Programmiersprache*. Als Programmiersprache für konkrete Beispiele betrachten wir im folgenden die Sprache Java, wenn dies nicht anders angegeben ist.

3.1 Variablen

Im imperativen Programmiersprachen (Achtung: dies ist in funktionalen oder logischen Programmiersprachen anders!) sind **Variablen** Behälter ($\approx$ Speicherzelle), die Werte aufnehmen können. Insbesondere kann man Werte und **verändern**.
Auf Grund dieser Überlegungen könnte man Variablen als Exemplare des folgenden ADTs modellieren, der insbesondere definiert, dass einmal überschriebene Werte nicht wieder gelesen werden können (der Parameter α definiert den Typ der Werte, den die Variable aufnehmen kann):

```
datatype Variable(α)
  new   : → Variable(α)
  write : α,Variable(α) → Variable(α)
  read  : Variable(α) → α
equations a,b:α, v:Variable(α)
  write(a, write(b,v)) = write (a,v)
  read(write(a,v))     = a
end
```

Die erste Gleichung beschreibt, dass alte Werte einer Variablen überschrieben werden.

Diese Modellierung ist allerdings unzureichend, da in vielen Programmiersprachen die Referenz auf Variablen wichtig ist (z.B. bei Zeigern oder `var`-Parametern von Prozeduren). Aus diesem Grund wählen wir folgendes Variablenmodell:

Eine **Variable** ist ein Tripel (Referenz, Typ, Wert), wobei „Referenz“ die Bezeichnung für den Behälter der Variablen, „Typ“ der Typ der Inhalte des Behälters und „Wert“ der eigentliche Inhalt ist.

Die Bindung eines Namens an eine Variable entspricht dann der Bindung des Variablennamens an ein solches Tripel.

Beispiel: Betrachte die Variablendeklaration

```
int i;
```

Effekt: Der Name „i“ wird z.B. gebunden an das Tripel `(ref99, int, ?)`, wobei `ref99` eine eindeutige (neue) Referenz ist. `?` ist bei C undefiniert, aber bei Java der Wert 0.

Operationales Modell imperativer Sprachen

Wir wollen nun ein präzises operationales Modell für imperative Sprachen angeben, das insbesondere den Referenzaspekt von Variablen genau modelliert. Zu diesem Zweck müssen wir zwei Arten von Bindungen, die bei Variablen in imperativen Sprachen relevant sind, modellieren:

1. Name $\mapsto$ (Referenz,Typ): Diese Bindungen werden in einer **Umgebung** aufgesammelt.
2. Referenz $\mapsto$ Wert: Diese Bindungen werden in einem **Speicher** aufgesammelt.

Wir modellieren dies daher wie folgt:

- **Umgebung** E: Dies ist eine Liste von Paaren $x : (ref, typ)$, wobei x ein Name ist (später werden auch noch anderen Arten Bindungen hinzugefügt).
- **Speicher** M: Dies ist eine Abbildung

  $$\text{Referenzen} \rightarrow Int \cup Bool \cup \ldots$$

  wobei Referenzen den Speicheradressen entsprechen (z.B. $\mathbb{N}$) und der Zielbereich die Menge aller möglichen Werte ist.

Die wichtigste Grundoperation auf einer Umgebung ist das *Auffinden von Bindungen*, das wir durch folgendes Inferenzsystem definieren:

Axiom:

$$E; x : \beta \vdash^{lookup} x : \beta$$

(hierbei bezeichnet “;” die Listenkonkatenation)

Regel:

$$\frac{E \vdash^{lookup} x : \beta}{E; y : \gamma \vdash^{lookup} x : \beta} \qquad \text{falls } x \neq y$$

Hierdurch modellieren wir die Suche nach dem letzten Eintrag, d.h. die Umgebung wird als Stack verwaltet.

Imperative Sprachen basieren auf folgendem Verarbeitungsprinzip:

- Deklarationen verändern die Umgebung
- Anweisungen verändern den Speicher

Um dieses Prinzip durch ein Inferenzsystem zu beschreiben, wählen wir die folgende *Basissprache zur Beschreibung von Deklarationen und Anweisungen*:

$$\langle E, M\rangle \; \alpha \; \langle E', M'\rangle$$

Falls ein solches Element ableitbar ist, interpretieren wir dies wie folgt: durch Abarbeitung von α wird die Umgebung E in E' und der Speicher M in M' überführt.

Beispiel: Deklarationen ganzzahliger Variablen in Java:

$$\langle E, M\rangle \texttt{ int x; } \langle E; x : (l, int), M[l/0]\rangle \qquad \text{mit } l \in free(M)$$

Dieses Axiom ist intuitiv wie folgt zu interpretieren:

- erzeuge eine **neue** Bindung für x mit Referenz l
- l („location“) ist eine freie Speicherzelle, d.h. die **Menge aller freien Speicherzellen** $free(M)$ ist wie folgt definiert:

$$free(M) = \{l \mid l \,\text{Referenz} \wedge M(l) \text{ ist undefiniert}\}$$

- initialisiere Wert von `x` mit 0

Analog: Deklaration boolescher Variablen:

$$\langle E, M\rangle \texttt{ boolean x } \langle E; x : (l, boolean), M[l/false]\rangle \qquad \text{mit } l \in free(M)$$

In diesen Regeln haben wir festgelegt, dass Variablenwerten initialisiert werden. Manche Programmiersprachen lassen diese Initialisierung offen, was aber problematisch ist, denn dadurch können sich Programme in mehreren Läufen nichtdeterministisch verhalten, fehlerhafte Zeiger existieren u.ä. Um dies zu vermeiden, ist manchmal auch eine *Deklaration mit Initialisierung* möglich, die wir wie folgt beschreiben können:

$$\langle E, M\rangle \texttt{ int x = } n \; \langle E; x : (l, int), M[l/n]\rangle$$

wobei n eine Zahl und $l \in free(M)$ ist.

Beim Umgang mit Variablen ist es wichtig zu unterscheiden zwischen

- Referenz (Behälter, **L-Wert, l-value**)
- aktueller Wert (**R-Wert, r-value**)

der Variablen. Dies ist auch relevant bei der Parameterübergabe bei Prozeduren. L/R bezieht sich auf die Seiten einer Zuweisung.

Da nicht alle Ausdrücke einen L-Wert haben, wohl aber einen R-Wert, benötigen wir unterschiedliche Inferenzsysteme zum Ausrechnen von L/R-Werten. Die Berechnung von R-Werten kann analog zur Wertberechnung von Ausdrücken (vgl. Kapitel 2.2.3) erfolgen, wobei wir hier nur noch das Nachschlagen von Variablendeklarationen berücksichtigen:

$$\frac{E \vdash^{lookup} x : (l, \tau)}{\langle E, M \rangle \vdash^{R} x : M(l)}$$

$$\frac{}{\langle E, M \rangle \vdash^{R} n : n} \quad \text{falls } n \text{ Zahl}$$

$$\frac{\langle E, M \rangle \vdash^{R} e_1 : v_1 \quad \langle E, M \rangle \vdash^{R} e_2 : v_2}{\langle E, M \rangle \vdash^{R} e_1 \texttt{+} e_2 : v_1 + v_2}$$

$$\frac{\langle E, M \rangle \vdash^{R} e_1 : v_1 \quad \langle E, M \rangle \vdash^{R} e_2 : v_2}{\langle E, M \rangle \vdash^{R} e_1 \texttt{*} e_2 : v_1 \cdot v_2}$$

Die Berechnung von L-Werten erfolgt analog. Dies ist aber eingeschränkter, da z.B. `x+3` keinen L-Wert hat. Zunächst einmal hat jede Variable einen L-Wert:

$$\frac{E \vdash^{lookup} x : (l, \tau)}{\langle E, M \rangle \vdash^{L} x : l}$$

Die Regeln für L-Werte werden später noch erweitert.

Zuweisung

Die Zuweisung ist die elementare Operation, die mit Variablen in imperativen Sprachen assoziiert ist. Hierbei findet man hauptsächliche die folgenden syntaktischen Darstellungen:

$\langle exp_1 \rangle$ = $\langle exp_2 \rangle$ (C, Java)

$\langle exp_1 \rangle$:= $\langle exp_2 \rangle$ (Pascal, Modula)

Einschränkung (statische Semantik): $\langle exp_1 \rangle$ muss einen L-Wert haben.
Beispiel: Konstanten haben nur R-Werte (ebenso Wertparameter bei Prozeduren, siehe später). In Java werden Konstanten durch das Schlüsselwort `final` deklariert:

```
static final int mean = 42;
```

Hierdurch wird die Bindung des Namens "`mean`" an die `int`-Konstante 42 definiert. Aus diesem Grund ist die Zuweisung

```
mean = 43;
```

ein Programmfehler. Um dies zu formalisieren, tragen wir Konstanten als Paar (Typ,Wert) in die Umgebung ein:

Deklaration eine Konstanten:

$$\langle E, M\rangle \texttt{ static final int } x\texttt{=}n \ \langle E; x : (int, n), M\rangle$$

Benutzung einer Konstanten (nur als R-Wert!):

$$\frac{E \vdash^{lookup} x : (\tau, v)}{\langle E, M\rangle \vdash^{R} x : v}$$

Damit können wir nun die **Semantik der Zuweisung** wie folgt festlegen:

- Berechne L-Wert von exp_1: l
- Berechne R-Wert von exp_2: v
- Schreibe v in Behälter l, d.h. ändere den Speicher so, dass $M(l) = v$

Formal:

$$\frac{\langle E, M\rangle \vdash^{L} exp_1 : l \quad \langle E, M\rangle \vdash^{R} exp_2 : v}{\langle E, M\rangle \ exp_1 = exp_2 \ \langle E, M[l/v]\rangle}$$

Weitere Forderung an die Zuweisung: *Typkompatibilität* der linken und rechten Seite. Für $e_1 = e_2$ kann das bedeuten:

- e_1 und e_2 haben gleiche Typen (strenge Forderung)
- meistens wird jedoch nur gefordert: Der Typ von e_2 muss in den Typ von e_1 konvertierbar sein (implizite Konvertierung).

Beispiel: Ganze Zahlen sind in Gleitkommazahlen konvertierbar:

```
float x;
int i=99;
x=i; // x hat den Wert 99.0
```

Umgekehrt kann man `float` nach `int` nur mit einem Genauigkeitsverlust konvertieren. Aus diesem Grund ist dies häufig unzulässig (z.B. in Java). Erlaubt ist dies nur mit expliziter Konvertierung („type cast“):

```
double x=1.5;
int i;
i=x; // unzulaessig
i=(int)x; // type cast, i hat den Wert 1
```

Typkonvertierungen sind von der jeweiligen Programmiersprache abhängig:

- C: Wahrheitswerte $\approx$ ganze Zahlen: $0 \approx$ `false`, $\neq 0 \approx$ `true`

- Java: `boolean` und `int` sind nicht konvertierbar:

```
boolean b; int i;
i=b; // unzulaessig
```

Formalisierung der Typkonvertierung

- berechne Typ von linker und rechter Seite (z.B. durch ein neues Inferenzsystem $\vdash^T$)
- füge, falls notwendig und erlaubt, Konvertierungsfunktionen ein

$\rightsquigarrow$ Übung

Abkürzungen für häufige Zuweisungsarten in Programmiersprachen:

- `x=x+3;`
  Modula-3: `INC(x,3);`
  C, Java: `x+=3;`
- allgemein: "`var op= expr`" $\approx$ "`var = var op expr`"
- "`x++`" entspricht "`x += 1`"
- "`x--`" entspricht "`x -= 1`"

3.2 Standarddatentypen

Ein Datentyp in einer Programmiersprache bezeichnet einen Wertebereich für Variablen. In einer streng getypten Programmiersprache enthält eine Variablendeklaration einen Typ, so dass diese Variable nur Werte des angegebenen Typs aufnehmen kann.
Man spricht von einem **Grundtyp**, wenn alle Konstruktoren Konstanten sind. Übliche Grundtypen sind Wahrheitswerte, Zeichen oder Zahlen.
Grundtypen in Java:

Typ	Werte	Initialwert
`boolean`	`true, false`	`false`
`char`	16bit-Unicode-Zeichen	`\u0000`
`byte`	8bit ganze Zahl mit Vorzeichen	0
`short`	16bit ganze Zahl mit Vorzeichen	0
`int`	32bit ganze Zahl mit Vorzeichen	0
`long`	64bit ganze Zahl mit Vorzeichen	0
`float`	32bit-Gleitkommzahl	`0.0f`
`double`	64bit-Gleitkommazahl	0.0

Weitere Grundtypen in Programmiersprachen (nicht in Java):

Aufzählungstypen: Wertebereich: endliche Menge von Konstanten
Modula-3:

```
TYPE Color = { Red, Green, Blue, Yellow };
VAR c: Color;
```

Ausschnittstypen: Wertebereich: Teilmenge aufeinanderfolgender Werte eines anderen Typs
Modula-3:

```
VAR day: [1..31];
day := 25;  (* ok *)
day := day + 10; (* Addition ok, Zuweisung fehlerhaft *)
```

Zusammengesetze Typen: Werte sind nicht elementar, sondern haben eine innere Struktur. Zusammengesetze Typen bieten insbesondere Operationen zum Zugriff auf Teilwerte.

Felder

Seien $I_1, \ldots, I_n$ Indexmengen. Dann heißt die Abbildung $A : I_1, \ldots, I_n \to \tau$ ein **Feld** mit Indexmengen $I_1, \ldots, I_n$ und Elementtyp τ.
Andere Sichtweise: A ist eine indizierte Sammlung von $|I_1| \cdots |I_n|$ Elementen.
Basisoperation auf einem Feld: Zugriff auf ein Feldelement: $A[i_1, \ldots, i_n]$ mit $i_j \in I_j$ für alle $j \in \{1, \ldots, n\}$.

Häufige Einschränkungen:

- Die Indexmengen sind endliche Ausschnitte aus den natürlichen Zahlen
- Fortran: Indexuntergrenze ist immer 1.
- C, C++, Java: Indexuntergrenze ist immer 0.

Zur Modellierung von Feldvariablen müssen wir den *Typ eines Feldes* festlegen. Dieser beinhaltet auf jeden Fall den Typ der Elemente und die Anzahl der Indizes, nicht aber unbedingt die Indexmengen selbst, da diese eventuell zur Übersetzungszeit noch nicht bekannt sind. Wir unterscheiden daher:

Statisches Feld: Die Indexgrenzen sind statisch bekannt und gehören zum Typ ⇒ Anzahl der Elemente zur Compilezeit bekannt.

Dynamisches Feld: Indexgrenzen werden erst zur Laufzeit bekannt ⇒ gehören nicht zum Typ

Flexibles Feld: Indexangaben zur Laufzeit veränderbar (seltenes Konzept, z.B. Algol-68).
Zeiger in C: Benutzung wie flexible Felder

Felddeklarationen in verschiedenen Programmiersprachen:

- Modula-2: `VAR a: ARRAY [1..100] OF REAL;`
- Fortran: `DOUBLE a(100,100); // 2-dimensionales Feld`
- C: `double a[100,100];`
- Java: `double[][] a = new double[100,100];`
  In Java wird die Deklaration (linke Seite) von der Erzeugung des Feldes (rechte Seite) getrennt. Die Zahl bei der Erzeugung bedeutet immer die Anzahl der Elemente in der Dimension, d.h. bei `[100]` sind die Indizes im Intervall `[0 .. 99]`.

Felder in Java:

- immer dynamisch
- müssen explizit erzeugt werden
- Erzeugung auch durch „initializers“ möglich:

  ```
  int[] pow2 = {1,2,4,8,16,32};
  ```

- Deklaration und Erzeugung sind unabhängig voneinander:

  ```
  int[][] a;
  ...
  a = new int[42][99];
  ```

- Zugriff durch Angabe der Indizes: `a[21][50]`
- Beachte: Unterschied zwischen L/R-Werten:

  ```
  a[i][j] = a[i+1][j+1];
  ```

  Der Ausdruck “`a[i][j]`” auf der linken Seite bezeichnet die Referenz auf das Element mit dem Index `(i,j)`. Dagegen bezeichnet der Ausdruck “`a[i+1][j+1]`” auf der rechten Seiten den Inhalt eines Feldelementes.

Präzisierung von Feldern:

Deklaration eines Feldes:

$$\langle E, M\rangle\ \tau \underbrace{[]\ldots[]}_{n-mal} x\ \langle E; x : array(n, \tau), M\rangle$$

Erzeugung eines Feldes:

$$\frac{E \vdash^{lookup} x : array(n, \tau) \quad \langle E, M\rangle \vdash^{R} e_1 : d_1\ \ldots\ \langle E, M\rangle \vdash^{R} e_n : d_n}{\langle E, M\rangle\ x = new\ \tau[e_1]\ldots[e_n]\ \langle E; x : array(l, [d_1, \ldots, d_n], \tau), M'\rangle}$$

wobei $a = d_1 \cdots d_n$, $l, l+1, \ldots, l+a-1 \in free(M)$ und, falls i der Initialwert von τ ist, $M' = M[l/i][l+1/i] \ldots [l+a-1/i]$

Somit wird beim Erzeugen der notwendige Platz für die einzelnen Feldelemente reserviert und die Anfangsadresse und Dimensionen bei x eingetragen (eigentlich müsste der Eintrag von x in der Umgebung E *verändert* werden, aber der Einfachheit halber fügen wir den veränderten Eintrag nur hinzu)

Zugriff auf ein Feldelement:

$$\frac{E \vdash^{lookup} x : array(l, [d_1, \ldots, d_n], \tau) \quad \langle E, M\rangle \vdash^R e_1 : i_1 \ \ldots \ \langle E, M\rangle \vdash^R e_n : i_n}{\langle E, M\rangle \vdash^L x[e_1] \ldots [e_n] : l_x}$$

$$\frac{E \vdash^{lookup} x : array(l, [d_1, \ldots, d_n], \tau) \quad \langle E, M\rangle \vdash^R e_1 : i_1 \ \ldots \ \langle E, M\rangle \vdash^R e_n : i_n}{\langle E, M\rangle \vdash^R x[e_1] \ldots [e_n] : M(l_x)}$$

mit $l_x = l + d_n \cdots d_2 \cdot i_1 + d_n \cdots d_3 \cdot i_2 + \cdots + d_n \cdot i_{n-1} + i_n$.

Dies ist eine vereinfachte Darstellung des Feldzugriffs. In Java ist es auch möglich, nicht alle Indizes anzugeben. In diesem Fall ist das Zugriffsergebnis ein Feld über restliche Indizes.

Spezialfall: **Zeichenketten**
Zeichenketten werden in vielen Sprachen als Feld von Zeichen dargestellt, d.h. könnte der Typ `String` identisch zu `char[]` sein (dies ist z.B. in C der Fall, wobei das letzte Zeichen das Nullzeichen `\000` ist)

In Java gilt dagegen:

- `String` ist ein eigener Typ (verschieden von `char[]`).
- Strings sind nicht veränderbar (stattdessen: `StringBuffer`)
- Stringkonstanten haben eine feste Syntax:

  ```
  String s = "Java";
  ```

  Hierbei ist `"Java"` eine Stringkonstante.

- Zur Konkatenation von Strings gibt es einen vordefinierten Operator "`+`":

  ```
  System.out.println(s+"2000");
     ⇝    Java2000
  ```

  ⇒ automatische Konvertierung von Werten zu Strings durch "`+`"

Verbunde

Ein Verbund ist das kartesische Produkt von Typen.
Wichtiger Aspekt: Man erhält Zugriff auf einzelne Komponenten über einen Namen für jede Komponente.

Beispiel: Verbunde in Modula-3

```
TYPE Point = RECORD
               x,y: REAL   (* x und y sind Komponentennamen *)
             END;

VAR pt: Point;
pt.x := 0.0;  (* Selektion der Komponente x von pt mit . *)
```

Verbunde in C:

```
typedef struct {double x,y} Point;

Point pt;
pt.x=0.0;
```

Java: keine expliziten Verbunde, sondern Klassen:

```
class Point {
    double x,y;
}
...
Point pt = new Point();
pt.x = 0.0;
```

Vereinigungstypen

Ein Vereinigungstyp bezeichnet die Vereinigung der Werte verschiedener Typen. In manchen Sprachen treten diese auch als „variante Verbunde“ auf.
Beispiel: Variante Verbunde in C:

```
typedef struct {
  int utype;
  union {
    int ival;
    float fval;
    char cval;
  } uval;
} UT;

UT v;
```

Hier dient die Komponente `utype` dazu, die konkreten Varianten dieser Verbundsstruktur zu unterscheiden, denn in der Komponenten `uval` ist zu jedem Zeitpunkt nur eine der Varianten `ival`, `fval` oder `cval` vorhanden. Z.B. erhalten wir durch `v.uval.ival` einen

`int`-Wert und durch `v.uval.fval` einen `float`-Wert. Zu beachten ist aber, dass die Komponenten `ival` und `fval` **an der gleichen Stelle** gespeichert werden:

Speicherstruktur:

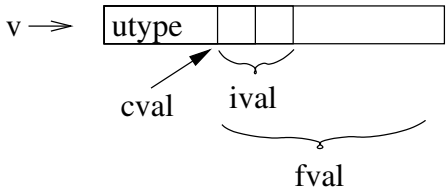


Abbildung 3.1: Speicherstruktur eines varianten Verbundes

Aus diesem Grund muss der Programmierer kontrollieren (z.B. mittels `utype`), welche Variante jeweils aktuell ist. Ein Problem kann sich leicht durch falsche Variantenbenutzung ergeben:

```
v.uval.fval = 3.14159;
v.uval.ival = 0;
```

Hierdurch wird ein Teil des `float`-Wertes 3.14159 mit 0 überschrieben. Der sich ergebende Wert von `v.uval.fval` ist implementierungsabhängig. Eine bessere und sichere Alternative zu varianten Verbunden sind Klassen und Unterklassen, bei dem das System den korrekten Zugriff kontrolliert (vgl. Kapitel 4).

Mengen

In einigen Programmiersprachen kann man Mengen definieren und darauf spezielle Mengenoperationen anwenden.
Beispiel: Modula-3

```
TYPE Color = {Blue, Green, Red, Yellow};
     ColorSet = SET OF Color;
```

Für Variablen vom Typ `ColorSet` sind nun die üblichen Mengenoperationen vordefiniert. Da es für Mengen je nach Anwendungsfall unterschiedlich effiziente Implementierungen gibt, sind bei den meisten modernen Sprachen Mengen nicht mit einer speziellen Syntax vordefiniert, sondern man benutzt stattdessen spezielle Bibliotheken.

Listen

Listen sind Sequenzen beliebiger Länge und bilden in logischen und funktionalen Sprachen die wichtigste vordefinierte Datenstruktur. Eine Liste ist entweder leer oder besteht aus einem Element (Listenkopf) und einer Restliste. Listen sind in imperativen Sprachen mit Verbünden und Zeigern definierbar. Aus diesem Grund betrachten wir zunächst einmal Zeigerstrukturen.

Zeiger

Ein Zeiger ist ein Verweis auf einen Behälter, in dem ein Wert gespeichert ist. Somit ist der Wert (R-Wert) einer Zeigervariablen immer ein L-Wert.
Die wichtigste Operation auf Zeigervariablen ist das **Dereferenzieren**, was den Übergang von einem L-Wert zu dem dort gespeicherten Wert bezeichnet.
Beispiel: Zeiger in C:

```
int *ip; // Wert von ip: Zeiger auf einen Behaelter, in dem ein int steht

int i;

ip = &i; // Adresse (L-Wert) der Variablen i

i = *ip; // dereferenzierter Wert von ip
```

Beispiel: Zeiger in Pascal:
Zeigerdeklaration: `var p: ^`τ`;`
Dereferenzieren: `p^`

Da jeder Datentyp einen Initialwert haben sollte, sollte dies auch für Zeiger gelten. Aus diesem Grund ist es üblich, einen ausgezeichneten Zeigerwert zu definieren: **der leere Zeiger**, der auf kein Objekt verweist.

Beispiel:
Pascal: `nil`
C: `NULL`
Java: `null`

Hier ergibt sich ein Problem: beim Dereferenzieren muss der Zeiger auf einen definierten Behälter zeigen. Somit ist das Dereferenzieren von leeren Zeigern nicht möglich und führt zu einem Laufzeitfehler.
Fortsetzung des Beispiels von oben:

```
ip = NULL;
i = *ip; // Laufzeitfehler
```

Die **Semantik von Zeigern** ist einfach definierbar mit L/R-Werten:

Deklaration:

$$\langle E, M\rangle \quad \tau * x \quad \langle E; x : (l, *\tau), M[l/NULL]\rangle$$

wobei $l \in free(M)$

Dereferenzieren:

$$\frac{\langle E, M\rangle \vdash^R e : l}{\langle E, M\rangle \vdash^L *e : l}$$

Beispiel:

```
*ip = 3;  // Setze den Behaelter, auf den ip verweist, auf 3.
ip = addr // Setze den Wert von ip (d.h. eine Adresse oder L-Wert) auf addr.
```

$$\frac{\langle E, M\rangle \vdash^R e : l}{\langle E, M\rangle \vdash^R *e : M(l)} \qquad \text{falls } l \neq NULL$$

Adressoperator:

$$\frac{\langle E, M\rangle \vdash^L e : l}{\langle E, M\rangle \vdash^R \&e : l}$$

Beispiel: Definition von Listen in C:

```
struct List {
  int elem;          // Listenelemente: Zahlen
  struct List *next; // Restliste: Zeiger auf den Listenrest
};

struct List *head;
head = (struct List *) malloc(sizeof(struct List)); // neue Listenstruktur
(*head).elem = 3;  // Kurzschreibweise: head->elem = 3;
```

Somit wird die leere Liste durch den leeren Zeiger `NULL` repräsentiert, wogegen eine nichtleere Liste ein Zeiger auf einen `List`-Verbund ist.

Zeigerarithmetik: In manchen Sprachen (Assembler, C) werden Referenzen (L-Werte) mit ganzen Zahlen identifiziert. Als Konsequenz sind dann arithmetische Operationen auf Referenzen erlaubt, z.B. "`(&i)+4`".
Dies ist eventuell relevant für die Systemprogrammierung, aber:

- Code wird undurchschaubar
- fehleranfällig (keine Kontrolle durch Laufzeitsystem)
- man erhält nicht-portablen Code

$\Rightarrow$ sollte in höheren PS nicht erlaubt sein.

Alternativen: Modula-3, JNI (Java Native Language Interface): klare Trennung von höherer Programmierung und low-level Systemprogrammierung.

Allgemeine Nachteile von Zeigern und Zeigerprogrammierung:

- fehleranfällig: existieren die Objekte, auf die ein Zeiger verweist? (dangling pointers)
- Zeiger verlangen eine explizite Handhabung des Dereferenzierens (z.B. C: `**p`, doppelte Verweise)
- Zeigerarithmetik (wenn das zulässig ist) erlaubt keine Laufzeitprüfungen

- Zeiger sind unstrukturiert („pointers are gotos“)
- häufig bieten diese Programmiersprachen keine automatische Speicherverwaltung (sondern: `malloc`, `free`) $\rightsquigarrow$ fehleranfällig

Lösung in funktionalen, logischen und objektorientierte Sprachen (Smalltalk, Eiffel, Java):

- nur Referenzen auf Objekte
- keine explizite Unterscheidung Referenz $\leftrightarrow$ Wert
- implizites Dereferenzieren

Beispiel: Listen in Java

```
class List {
  int elem;
  List next;
}
...
List head;
head = new List(); // Erzeugung neuer Listenstruktur
head.elem = 3;
```

Nachteile dieses allgemeinen Ansatzes:

- immer Indirektion durch Zeiger
- alle Werte nur durch Dereferenzieren erreichbar
- überflüssig für Grunddatentypen (`int`, `char`,...)

Daher in Java: Grunddatentypen direkt zugreifbar, strukturierte Typen immer über Referenzen

3.3 Kontrollabstraktion

Ein Programmablauf in einer imperativen Programmiersprache ist im wesentlichen eine Folge von Zuweisungen. Da es wichtig ist, dass Programme lesbar und wartbar sein, müssen auch Zuweisungsfolgen strukturiert werden (keine `goto`s, vgl. [4]).
Lösung:

- strukturierte Programmierung (ohne `goto`)
- Kontrollabstraktionen: (sinnvolle) Zusammenfassung von häufigen Ablaufschemata zu Einheiten

In folgenden diskutieren wir typische Kontrollabstraktionen imperativer Sprachen.

Sequentielle Komposition

Wenn S_1 und S_2 Anweisungen sind, dann ist auch die *sequentielle Komposition* $S_1; S_2$ eine Anweisung. Hier wird S_1 und dann S_2 ausgeführt.
Formal:

$$\frac{\langle E, M\rangle\ S_1\ \langle E', M'\rangle \qquad \langle E', M'\rangle\ S_2\ \langle E'', M''\rangle}{\langle E, M\rangle\ S_1; S_2\ \langle E'', M''\rangle}$$

Anmerkungen:

- In Java können Deklarationen mit Anweisungen gemischt werden
- Häufig erlaubt: Zusammenfassung von Anweisungen zu **Blöcken**
  (Java: $\{S_1; S_2; ...; S_n;\}$)
  In diesem Fall sind die Deklarationen innerhalb des Blocks außerhalb unsichtbar:

$$\frac{\langle E, M\rangle\ S\ \langle E', M'\rangle}{\langle E, M\rangle\ \{S\}\ \langle E, M'\rangle}$$

- Benutzung des Semikolons: Hier existieren zwei unterschiedliche Sichtweisen:
  1. "`;`" bezeichnet die sequentielle Komposition und steht zwischen Anweisungen ($\approx$ Pascal). Dies führt zu folgender Blocksyntax:

     ```
     begin s1; s2; ... ; sn end
     ```

     Hier darf hinter der letzten Anweisung `sn` kein Semikolon stehen!
  2. "`;`" kennzeichnet das Ende einer Anweisung und steht hinter jeder Anweisung ($\approx$ Java). Dies führt zu folgender Blocksyntax:

     ```
     { s1; s2; ... sn; }
     ```

  Manchmal: beide Sichtweisen, z.B. in Modula-3.

Bedingte Anweisungen

Typische Form bedingter Anweisungen (Fallunterscheidungen), z.B. in Ada:

```
if bexp then S1 else S2 endif
```

wobei `bexp` ein boolescher Ausdruck und `S1` und `S2` Anweisungsfolgen sind. Die `if-endif`-Klammerung vermeidet hierbei Mehrdeutigkeiten („dangling else"), die bei einem fehlenden `else`-Teil auftreten können:
Java:

```
if (bexp) S1 else S2
```

wobei `S1` und `S2` Anweisungen sind und der `else`-Teil auch fehlen kann. Beispiel:

```
x = 0;
y = 1;
if (y<0)
  if (y>-5)
    x=3;
else x=5;
```

Wozu gehört das "`else`"? Falls dies zum ersten `if` gehört, gilt am Ende `x=5`. Falls dies zum zweiten `if` gehört, gilt am Ende `x=0`.

Üblich (auch in Java) ist die folgende Konvention:

> Ein `else` gehört immer zum letztmöglichen `if`, welcher keinen `else`-Zweig hatte.

In diesem Beispiel gehört das `else` also zum zweiten `if`, d.h. die Einrückungen sind falsch gewählt.
Ein Zweifelsfällen kann (und sollte) man immer Klammern setzen:

```
x = 0;
y = 1;
if (y<0)
  {if (y>-5)
    x=3;
   else x=5;}
```

```
x = 0;
y = 1;
if (y<0)
 {if (y>-5)
    x=3; }
else x=5;
```

Bedeutung bedingter Zuweisungen:

$$\frac{\langle E, M\rangle \vdash^R bexp : true \quad \langle E, M\rangle\ S_1\ \langle E', M'\rangle}{\langle E, M\rangle\ if\ (bexp)\ S_1\ else\ S_2\ \langle E', M'\rangle}$$

$$\frac{\langle E, M\rangle \vdash^R bexp : true \quad \langle E, M\rangle\ S_1\ \langle E', M'\rangle}{\langle E, M\rangle\ if\ (bexp)\ S_1\ \langle E', M'\rangle}$$

$$\frac{\langle E, M\rangle \vdash^R bexp : false \quad \langle E, M\rangle\ S_2\ \langle E', M'\rangle}{\langle E, M\rangle\ if\ (bexp)\ S_1\ else\ S_2\ \langle E', M'\rangle}$$

$$\frac{\langle E, M\rangle \vdash^R bexp : false}{\langle E, M\rangle\ if\ (bexp)\ S_1\ \langle E, M\rangle}$$

Varianten von bedingten Anweisungen:

- "`elseif`". (Ada):

  ```
  if b1 then S1
  elseif b2 then S2
  elseif b3 then S3
  else S4
  endif
  ```

Dies ist äquivalent zu

```
if b1 then S1
else
  if b2 then S2
  else
    if b3 then S3 else S4 endif
  endif
endif
```

- `case`/`switch`: Fallunterscheidung über den Wert eines Ausdrucks (in Java-Syntax):

  ```
  switch (exp) {
    case v1: S1; break;
    case v2: S2; break;
    ...
    case vn: Sn; break;
    default: Sn+1; }
  ```

  Dies ist äquivalent zu

  ```
  vc = exp; // vc ist eine neue Variable
  if (vc=v1) S1
  else if (vc=v2) S2
       else ...
       else if (vc=vn) Sn
            else Sn+1;
  ```

  Allerdings ist es üblich, dass der Compiler eine effizientere Übersetzung wählt (z.B. müssen die Werte v_i konstant sein, so dass man dann eine Sprungtabelle generieren kann, die das Auffinden der Alternative in konstanter Zeit ermöglicht).

Schleifen

Schleifen ermöglichen ein wiederholtes Ausführen von Anweisungen. Semantisch entsprechen Schleifen einer Rekursion (in der denotationellen Semantik werden Schleifen als kleinste Fixpunkte einer Semantiktransformation interpretiert).
Prinzipiell ist eine `while`-Schleife (und Fallunterscheidungen) für ein Programm ausreichend.

Java-Syntax für `while`-Schleifen:

```
while (B) S
```

Intuitiv: „Solange `B` wahr ist, führe `S` aus.“
Formal: “`while (B) S`” ist äquivalent zu “`if (B) {S; while (B) S}`”

Aus dieser Überlegung können wir die Semantik der `while`-Schleife durch folgende abgeleitete Inferenzregeln definieren:

$$\frac{\langle E, M\rangle \vdash^R B : false}{\langle E, M\rangle\ while\ (B)\ S\ \langle E, M\rangle}$$

$$\frac{\langle E, M\rangle \vdash^R B : true \qquad \langle E, M\rangle\ S;\ while\ (B)\ S\ \langle E', M'\rangle}{\langle E, M\rangle\ while\ (B)\ S\ \langle E', M'\rangle}$$

Varianten von Schleifen:

- Repeat-Schleifen: Führe die Anweisung mindestens einmal aus:
  - Pascal: `repeat S until B`
  - Java: `do S while (B)`

  Die letzte Form ist äquivalent zu "`S; while (B) S`", während in der `repeat`-Form die Bedingung negiert ist.

- Zählschleifen (meistens in Verbindung mit Feldern):
  Führe bei der Schleifenanweisung einen Zähler mit. Üblicherweise werden Zählschleifen mit dem Schlüsselwort "`for`" eingeleitet, aber es gibt sehr viele Varianten, z.B. in Java:

  ```
  for (initstat; boolexpr; incrstat) stat
  ```

  Dies ist äquivalent zu:

  ```
  { initstat;
    while (boolexpr) {
       stat
       incrstat;
    }
  }
  ```

  Typische Anwendung: Aufsummieren von Feldelementen:

  ```
  s=0;
  for (int i=0; i<fieldlen; i++) {
     s=s+a[i];
  }
  ```

- Endlosschleifen: Modula-3: "`loop S end`"
  Dies entspricht "`while (true) S`"
  Endlosschleifen können bei Serveranwendungen vorkommen, aber in der Regel sind sie nur sinnvoll, falls es die Möglichkeit gibt, die Schleife mit einem Aussprung zu verlassen:

- Modula-3: `exit`: verlasse innerste Schleife und mache danach weiter.
- Java: `break`: analog, aber auch mit Marken, um mehrere Schleifen zu verlassen, z.B.:

```
search:   // Sprungmarke
for (...) {
  while (...) {
    ...
    break search;
    /* verlasse damit die while- und for-Schleife auf einmal. */
    ...
  }
}
```

3.4 Prozeduren und Funktionen

Im Allgemeinen sind Komponenten von Programmen oder logisch zusammengehörige Programmteile durch folgende Elemente charakterisiert:

- Eingabewerte
- Berechnungsteil
- Ausgabewerte

Prozedurale Abstraktion bedeutet, dass man solchen Programmteilen einen Namen gibt und den Berechnungsteil dann als „black box“ betrachtet.

Aspekte von Prozeduren:

- logisch zusammenhängende Berechnung
- wiederverwendbares Berechnungsmuster
- manchmal Unterscheidung zwischen Funktionen und Prozeduren:
  - Funktionen: berechnen ein Ergebnis
  - Prozeduren: berechnen kein explizites Ergebnis, sondern sie haben im wesentlichen nur Seiteneffekte (möglichst nur auf Parameter, s.u.)
- Vorsicht: auch Funktionen können Seiteneffekte haben:
  - schlechter Programmierstil
  - schwer überschaubar
  - Reihenfolge wird bei der Auswertung von Ausdrücken relevant

  besser:

- verbiete Seiteneffekte in Funktionen
- übergebe alle zu verändernde Daten als Parameter (z.B. call-by-reference)

Benutzung von Prozeduren:

- (echte) Prozeduren: Prozeduraufruf $\approx$ Anweisung
- Funktionen: Funktionsaufruf in Ausdrücken

Prozedurdeklaration: Übliches Schema: definiere

- Typ der Parameter
- evtl. Übergabemechanismus für Parameter (s.u.)
- Ergebnistyp (bei Funktionen)
- Prozedurenrumpf (Berechnungsteil, Block)

Um die Bedeutung von Prozeduren genauer festzulegen, benutzen wir die folgende Pseudonotation für Prozeduren und Funktionen:

$$function\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) :\ \tau\ \texttt{is}\ S$$

mit $\tau_1, \ldots, \tau_n$ Parametertypen, τ Ergebnistyp und S Rumpf der Prozedur.

Bei Funktionen erfolgt im Rumpf üblicherweise die Festlegung des Ergebniswertes auf zwei mögliche Arten:

- durch eine spezielle Anweisung: `return(e)`, wobei `e` der Ergebniswert dieses Funktionsaufrufes ist.
- durch eine Zuweisung an den Prozedurnamen: f `:=e`. Hierbei wird der Prozedurname f wie eine lokale Variable im Rumpf behandelt.

Anwendung einer Prozedur:

- Bei einer echten Prozedur (d.h. der Ergebnistyp τ fehlt bei der Deklaration) ist der Prozeduraufruf $f(e_1, \ldots, e_n)$ syntaktisch eine Anweisung.
- Bei einer Funktion mit Ergebnistyp τ kann der Aufruf $f(e_1, \ldots, e_n)$ in Ausdrücken vorkommen, wo ein Wert vom Typ τ erwartet wird.
- Operationales Vorgehen:
  1. Deklariere $x_1, \ldots, x_n$ als (lokale) Variablen mit Initialwerten $e_1, \ldots, e_n$
  2. Führe den Rumpf S aus
  3. Bei Funktionen: ersetze den Funktionsaufruf durch das berechnete Ergebnis

Im folgenden werden wir diese Semantik durch entsprechende Erweiterungen unseres Inferenzsystems formalisieren.

Prozedurdeklaration:

$\langle E, M\rangle \;\; function\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \texttt{ is } S \;\; \langle E; f() : func(x_1{:}\tau_1, \ldots, x_n{:}\tau_n, \tau, S, E), M\rangle$

Anmerkungen:

- Eine Prozedurdeklaration hat keinen Effekt auf den Speicher.
- Die aktuelle Umgebung E der Deklaration muss wegen der lexikalischen Bindung bei der Prozedur gespeichert werden (vgl. Prozeduraufruf).
- τ kann bei einer echten Prozedur auch fehlen. In diesem Fall schreiben wir *proc* statt $func$.

Anwendung einer echten Prozedur:

$$\frac{E \vdash^{lookup} p() : proc(x_1{:}\tau_1, \ldots, x_n{:}\tau_n, S, E') \qquad \begin{array}{c}\langle E, M\rangle \vdash^R e_1 : v_1 \\ \vdots \\ \langle E, M\rangle \vdash^R e_n : v_n\end{array} \qquad \langle E'', M\rangle \;\; S \;\; \langle E''', M'\rangle}{\langle E, M\rangle \;\; p(e_1, \ldots, e_n) \;\; \langle E, M'\rangle}$$

wobei

$$E'' = E'; p() : proc(x_1{:}\tau_1, \ldots, x_n{:}\tau_n, S, E'); x_1 : (\tau_1, v_1); \ldots; x_n : (\tau_n, v_n)$$

(das Eintragen der Prozedurdeklaration $p()$ in E'' ist wichtig, um rekursive Aufrufe zu ermöglichen)

Funktionsanwendung:

$$\frac{E \vdash^{lookup} f() : func(x_1{:}\tau_1, \ldots, x_n{:}\tau_n, \tau, S, E') \qquad \begin{array}{c}\langle E, M\rangle \vdash^R e_1 : v_1 \\ \vdots \\ \langle E, M\rangle \vdash^R e_n : v_n\end{array} \qquad \langle E'', M'\rangle \;\; S \;\; \langle E''', M''\rangle}{\langle E, M\rangle \vdash^R f(e_1, .., e_n) : M''(l)}$$

wobei

$$E'' = E'; f() : func(x_1{:}\tau_1, \ldots, x_n{:}\tau_n), \tau, S, E'); x_1 : (\tau_1, v_1); \ldots; x_n : (\tau_n : v_n); f : (l, \tau)$$

mit $l \in free(M)$, $M' = M[l/i]$, wobei i der Initialwert von Typ τ ist

Anmerkungen:

- **Parameterübergabe**: zunächst müssen die *aktuellen* Parameterwerte ausgerechnet werden (mittels $\langle E, M\rangle \vdash^R e_j : v_j$), um dann die *formalen* Parameter x_j als Konstante mit den entsprechenden Werten $(x_j : (\tau_j, v_j))$ vor Ausführung des Prozedurrumpfes zu deklarieren (die hier spezifizierte Übergabeart ist also "call by value", s.u.)

- Der Rumpf S wird in der Umgebung der Deklaration abgearbeitet (lexikalische Bindung!), wobei die Deklaration der Prozedur für eventuelle rekursive Aufrufe eingeführt wird. Eine Modellierung einer dynamischen Bindung wie in Lisp wäre auch einfach möglich: dazu müsste man im Aufruf die aktuelle Umgebung lassen, d.h. E' durch E ersetzen.

- Bei Funktionen haben wir angenommen, dass diese keine Seiteneffekte haben (d.h. wir ignorieren den veränderten Speicher M''). Wenn wir auch mögliche Seiteneffekte von Funktionen modellieren wollten, müssten wir den veränderten Speicher in den Inferenzregeln für $\vdash^R$ berücksichtigen, wodurch wir allerdings eine feste Auswertungsreihenfolge bei Ausdrücken festlegen müssten.

- Ergebnisrückgabe bei Funktionen: Hierzu haben wir den Funktionsnamen f als lokale Variable deklariert ($f : (l, \tau)$). Damit können wir beide Arten der Funktionswertrückgabe modellieren:
  - Falls in S die Zuweisung f `=` e vorkommt, wird hierdurch der Rückgabewert festgelegt.
  - Falls in S die Anweisung `return(`e`)` vorkommt, interpretiere wir diese als "f `=` e`; break`"

  Beachte, dass es hierzu wichtig ist, in der Umgebung zwischen der Funktionsdeklaration $f()$ und der lokalen Variablen f zu unterscheiden!

- **formale Parameter:** hier haben wir diese als Konstante deklariert, d.h. Zuweisungen an diese im Rumpf S sind nicht erlaubt.

  Manchmal (z.B. in Java) werden formale Parameter als lokale Variablen (deren Initialwert der Wert des aktuellen Parameters ist) eingeführt. Dies könnten wir auch einfach modellieren, in dem wir statt $x_j : (\tau_j, v_j)$ die formalen Parameter durch $x_j : (l_j, \tau_j)$ und $M[l_j/v_j]$ mit $l_j \in free(M)$ einführen.
  Konsequenz: Zuweisungen an formale Parameter x_j sind im Rumpf S möglich, allerdings haben diese keine globale Auswirkungen.

- **Rekursion**: problemlos möglich, da Prozedurdeklaration zur Umgebung des Rumpfes gehört (in älteren Sprachen wie Cobol oder Fortran war dies verboten)

- **indirekte Rekursion**: p ruft q auf und q ruft p auf.

  Dies haben wir aus Vereinfachungsgründen weggelassen. Dies könnte jedoch auf zwei Arten ermöglicht werden:

  1. `forward`-Deklarationen: deklariere zunächst nur den Kopf der Prozedur, später dann den Rumpf (dies wurde z.B. in der Programmiersprache Pascal eingeführt).
  2. Sammle alle Deklarationen eines Blocks in einer Umgebung und speichere diese Umgebung in den Prozedurdeklarationen (dies ist üblich in modernen Programmiersprachen).

- **Schlüsselwortparameter** (z.B. Ada): Parameterübergabe nicht in der Reihenfolge der angegebenen formalen Parameter, sondern beliebig mit den Namen der formalen Parameter. Z.B. werden durch

  $$f(e_1, \ldots, e_k, x_{i_1} = e_{i_1}, \ldots, x_{i_l} = e_{i_l})$$

  die ersten $e_1, \ldots, e_k$ an die ersten k formalen Parameter übergeben und die restlichen Parameter direkt mittels der Namen der formalen Parameter übergeben.

  Vorteil: nicht alle Parameter müssen übergeben werden, sondern man kann default-Werte für einige Parameter verwenden.

Syntax von Prozeduren in Java (hier spricht man von „Methoden“ statt Prozeduren)

`public static` τ f`(`$\tau_1\, x_1, \ldots, \tau_n\, x_n$`) { `$S$` }`

Hierbei bedeutet `public`, dass der Prozedurname nach außen bekannt ist und `static`, dass die Prozedur auch ohne Objekte existiert (vgl. 4). Im Rumpf einer Funktion wird der Rückgabewert mit “`return` e`;`” festgelegt.

Beispiel: Aufsummieren aller Elemente eines Feldes:

```
class XY {
  ...
  public static int sum(int[] a, int n) {
    int i, s=0;
    for (i=0; i<=n; i++) s += a[i];
    return s;
  }
  ...
    int[] pow2 = {1,2,4,8};
    System.out.println("Summe: "+sum(pow2,3));
  ...
}
```

Parameterübergabemechanismen

In unserer bisherigen Modellierung sind die formalen Parameter lokale Objekte. Damit können die Parameter nicht verwendet werden, um globale Änderungen zu bewirken. Dies bedeutet, dass Prozeduren nur globale Effekte haben, in dem diese globale bekannte Objekte verändern. Dies führt aber leicht zu undurchschaubaren Seiteneffekten von Prozeduren, die man nur durch ansehen des Prozedurrumpfes erkennen könnte, was aber der Idee der prozeduralen Abstraktion widerspricht. Wünschenswert ist es, dass alle Effekt einer Prozedur durch den Prozedurkopf, d.h. die Parameter kontrolliert werden. Dies ist aber nur möglich, wenn man mittels der Parameter auch globale Effekte bewirken kann. Zu diesem Zweck gibt es in verschiedenen Programmiersprachen eine Reihe von Parameterübergabemechanismen, die wir im folgenden vorstellen.

Wertaufruf (call-by-value): Dies ist der zuvor beschriebene Übergabemechanismus:

- berechne R-Wert des aktuellen Parameters
- initialisiere formalen Parameter mit diesem Wert
- Typeinschränkung: Typ des R-Wertes muss an den Typ des formalen Parameters anpassbar sein
- Nachteil: da Wertparameter bei der Übergabe kopiert werden, kann dies bei großen Datenstrukturen (z.B. Felder) aufwändig sein

Ergebnisaufruf (call-by-result):

- aktueller Parameter muss L-Wert haben
- formaler Parameter ist zunächst eine lokale Variable in dieser Prozedur
- bei Prozedurende: kopiere den Wert des formalen Parameters in den aktuellen Parameter

Beispiel: Ada:

```
procedure squareRoot(x: in real; y: out real) is ...
```

Hierbei wird `x` per Wertaufruf ("`in`") und `y` per Ergebnisaufruf ("`out`") übergeben

Aktuelle Aufrufe dieser Prozedur:

```
squareRoot (2.0+z, r) -- ok;
squareRoot (z, 2.0+r) -- nicht ok
```

Wert-Ergebnisaufruf (call-by-value-result): Dies bezeichnet die Kombination der beiden vorhergehenden Übergabearten:

- aktueller Parameter muss L-Wert haben
- initialisiere formalen Parameter mit R-Wert des aktuellen Parameters
- bei Prozedurende: rückkopieren wie bei Ergebnisaufruf

Referenzaufruf (call-by-reference):

- aktueller Parameter muss L-Wert haben
- formaler Parameter entspricht Zeiger auf aktuellen Parameter
- Initialisierung: $\cdots \langle E, M\rangle \vdash^L e_j : l_j \quad \cdots \quad E'' = ...x_j : ref(l_j, \tau_j)$
- ref-Objekte werden wie Zeiger behandelt, aber mit impliziter Dereferenzierung bei ihrer Benutzung:

$$\frac{E \vdash^{lookup} x : ref(l, \tau)}{\langle E, M\rangle \vdash^L x : l}$$

$$\frac{E \vdash^{lookup} x : ref(l, \tau)}{\langle E, M\rangle \vdash^R x : M(l)}$$

- bei typisierten Sprachen muss der Typ des aktuellen Parameters **äquivalent** zum Typ des formalen Parameters sein

  Zum Begriff der **Typäquivalenz**:
  Hierfür gibt es unterschiedliche Definitionen in verschiedenen Programmiersprachen:

  - **Namensgleichheit:** Typen sind äquivalent $\Leftrightarrow$ Typnamen identisch
  - **Strukturgleichheit:** Typen sind äquivalent falls diese strukturell gleich sind. Z.B. können Verbundtypen mit unterschiedliche Verbundnamen aber gleichen Komponententypen äquivalent sein.

  In den meisten Programmiersprachen ist die Namensgleichheit üblich. In diesem Fall sind im Beispiel

  ```
  type ziffer = [0..9];
  type note   = [0..9];
  ```

  die Typen `ziffer` und `note` zwar strukturell gleich, aber bezüglich der Namensgleichheit nicht typäquivalent.

- Potenzielles Problem beim Referenzaufruf:
  Aliasing: dies bedeutet, dass es unterschiedliche Namen für gleiche Objekte (L-Werte) geben kann

  ```
  procedure p(x,y: in out integer) is B
  ```

  Dann steht beim Aufruf "`p(i,i)`" sowohl `x` als auch `y` für die gleiche Speicherzellen, was die Gefahr von unüberschaubaren Seiteneffekten hat. Wenn z.B. in B die Anweisungen

  ```
  x := 0;
  y := 1;
  while y>x do ...
  ```

  vorkommen, dann ist bei dem Aufruf "`p(i,i)`" die Bedingung `y>x` nie erfüllt, was vielleicht vom Implementierer dieser Prozedur nicht intendiert ist.

Der Referenzaufruf kann benutzt werden, um z.B. eine universelle Prozedur `swap` zum Vertauschen des Inhaltes zweier Variablen zu definieren (hier: Modula-2, Kennzeichnung des Referenzaufrufs durch "`var`"):

```
procedure swap(var x: integer; var y: integer);
  var z: integer;
begin
  z := x; x := y; y := z
end swap;
⋮
```

```
i := 2; a[2] := 42;
swap(i,a[i]);
```

Nach dem Aufruf von `swap` hat `i` den Wert 42 und `a[2]` den Wert 2.

Namensaufruf (call-by-name): Algol-60, Simula-67

- formale Parameter werden im Rumpf *textuell* durch aktuelle Parameter ersetzt. Hierbei werden jedoch Namenskonflikte gelöst (im Gegensatz zur einfachen Makro-Expansion)

  Beispiel für einen Namenskonflikt:

  ```
  int i; int[] a;
  procedure p(int x) is int i; i=x; end;
  ⋮
  p(a[i]);
  ```

  Das "`int i`" in der Prozedur und das `i` in der Parameterübergabe verursachen einen Namenskonflikt, wenn man im Rumpf der Prozeudr `x` einfach durch `a[i]` ersetzen würde.
  Lösung: Benenne in `p` **vor dem Aufruf** lokale Variablennamen so um, dass diese nicht in Konflikt mit den übergebenen aktuellen Parameter stehen.

- Implementierung des Namensaufrufs:
  - erzeuge für jeden aktuellen Parameter Prozeduren, die die L- und R-Werte dieses Parameters berechnen.
  - ersetze jedes Vorkommen eines formale Namenparameters im Rumpf durch die entsprechende Parameterprozeduraufrufe
- Somit werden die aktuellen Parameter bei *jedem* Zugriff auf die entsprechenden formalen Parameter ausgewertet! Dies ermöglicht die folgende Berechnung, die auch als **Jensen-Trick** aus Algol-60 bekannt ist (hierbei werden Wertparameter durch "`value`" gekennzeichnet; alle anderen Parameter sind immer Namenparameter):

  ```
  real procedure sum (i,n,x,y);
    value n;
    integer i,n;
    real x,y;
  begin
    real s; s := 0;
    for i:=1 step 1 until n do s := s+x*y;
    sum := s;
  end;
  ```

Aufruf: `sum(i,n,a[i],b[i])` $\rightsquigarrow$ `a[1]*b[1]+···+a[n]*b[n]`
(wegen wiederholter Auswertung von `a[i]` und `b[i]`!)

Prozeduren als Parameter:

- in manchen Sprachen zulässig:

```
procedure p(x: int; procedure r(y: int)) is ... r(...) ... end;
```

- Parameterübergabe: call-by-value/-reference, d.h. übergeben wird die (Code-) Adresse der aktuellen Prozedur (und die Umgebung bei lexikalischer Bindung!), die nicht veränderbar ist.
- Pascal: typunsicher: Parametertypen von Prozeduren als Parametern sind unbekannt und werden daher nicht überprüft.
- Modula-3: typsicher und Zuweisung an Prozedurvariablen (z.B. Prozeduren in Records) sind zulässig
- Eine umfassende Behandlung erfolgt in den funktionalen Programmiersprachen (vgl. Kapitel 5).

Einordnung der Parameterübergabemechanismen:

- viele Programmiersprachen bieten mehrere Übergabemechanismen für Prozedurparameter an, z.B.
  - Pascal/Modula: Wert- und Referenzaufruf
  - Ada: Wert- (`in`), Ergebnis- (`out`), Wert/Ergebnis- (`in out`) und Referenzaufruf für Felder
  - Fotran: Referenzaufruf für Variablen, sonst Wertaufruf
  - C: nur Wertaufruf
  - Java: nur Wertaufruf, aber beachte: Objektvariablen (Felder, Verbünde) sind Referenzen, daher wird bei Aufruf nur die Referenz übergeben, aber das Objekt selbst wird nicht kopiert

```
public static void first99(List f) {
    f.elem = 99; // globale Aenderung
    f = null;    // lokale Aenderung
}
⋮
List head = new List();
head.elem = 3;
first99(head);  // danach gilt head.elem = 99
```

- Namensaufruf (Algol-60, Simula-67): eher historisch wichtig

- heute: überwiegend Wert- und Referenzaufruf:
  Wertaufruf ≈ Eingabeparameter
  Referenzaufruf ≈ Ausgabeparameter bzw. große Eingabeobjekte
- Beachte: Wertaufruf für Zeiger oder Referenzen entspricht (fast) einem Referenzaufruf
- Funktionale Programmiersprachen bieten auch den „faulen Aufruf“ (call-by-need) als Übergabeart (vgl. Kapitel 5).

3.5 Ausnahmebehandlung

Unter einer **Ausnahme** verstehen wir eine Situation, bei der der normale Programmablauf nicht fortgeführt werden kann und daher unterbrochen wird. Man spricht auch von **Laufzeitfehler** oder **exception**.

Beispiele:

- Division durch 0
- Zugriff auf Feldwerte mit undefiniertem Index
- Dereferenzierung eines Zeigers mit Wert NULL
- Öffnen einer nicht vorhandenen Datei
- ...

Zuverlässige Programme sollten dadurch nicht abstürzen! (z.B. Programme zur Kraftwerkssteuerung, Flugzeugsteuerung etc.)
Konsequenz: Eine gute Programmiersprache muss Konstrukte zur Behandlung von Ausnahmen anbieten (solche wurden zuerst in PL/I eingeführt).

Das übliche Konzept zum Arbeiten mit Ausnahmen gliedert sich in zwei Teile:

Auslösung von Ausnahmen:

- durch das Laufzeitsystem (Feldindex, ..., s.o.)
- durch das Betriebssystem (Dateien, ...)
- durch den Programmierer (falsche Eingaben, ...)

mittels einer speziellen Anweisung “`raise <Fehlertyp>`” (z.B. könnte der Fehlertyp “division by zero” beschreiben)

Behandlung von Ausnahmen durch

- speziellen Anweisungsteil in Blöcken oder Prozeduren (z.B. Ada), der nur bei Ausnahmen in diesem Block oder Prozedur ausgeführt wird
- spezielle Anweisung mit Ausnahmebehandler (z.B. Java, ähnlich wie Block (s.o.), aber als spezielle Anweisung)

Kontrollfluss beim Auftreten einer Ausnahme:

- falls für die Anweisung / Block / Prozedur, in der die Ausnahme auftritt, ein Ausnahmebehandler vorhanden ist: führe Anweisungen in diesem Behandler aus
- sonst: verlasse diesen Block / Prozedur und löse Ausnahme im nächsten umgebenden Block bzw. Prozedur aus (evtl. bis zum Hauptprogramm → Programmabbruch)

⇒ operationelle Semantik ≈ wiederholtes `break` + `raise`

Wo wird nach der Ausnahmebehandlung weitergemacht? Hier bieten sich zwei Alternativen an:

- an der Stelle, wo der Fehler auftrat („resumption model“, PL/I).
  Hierbei ergibt sich allerdings ein Problem: Der Ausnahmebehandler kennt in der Regel nicht die Stelle, wo der Fehler auftrat ⇒ Wie kann man *eine* Behandlung für unterschiedliche Ursachen erreichen? Außerdem ist es manchmal schwierig, den Fehler wirklich zu „reparieren“, um sinnvoll weiter zu machen.
- Daher heute üblich: beende aktuellen Block / Prozedur nach Ausnahmebehandlung („termination model“) und mache im umgebenden Block bzw. aufrufender Prozedur normal weiter.

Beispiel: Ausnahmebehandlung bei Ada

- Programmierer kann Ausnahmen definieren und auslösen
- Ausnahmebehandlung ist Teil jedes Blocks
- Behandlung spezieller Ausnahmetypen
- „termination model“

Beispielprogramm:

```
⋮
Invalid: exception; -- Deklaration einer Ausnahme
begin
  ...
  if Data<0 then raise Invalid; end if;
  ...
exception -- Beginn der Ausnahmebehandlung in diesem Block
  when Constraint_Error => Put ("Error - data out of range");
  when Invalid          => Put ("Error - negative value used");
  when others           => Put ("Some other error occured");
    -- others ist ein Schluesselwort (default-Behandlung)
end;
```

Beispiel: Ausnahmebehandlung in Java

- Programmierer kann Ausnahmen (spezielle Objekte) deklarieren und auslösen
- Ausnahmebehandlung in spezieller Anweisung (`try / catch / finally`)
- spezielle Behandlung verschiedener Ausnahmetypen
- jede Prozedur (Methode) muss mögliche Ausnahmen behandeln oder explizit deklarieren, z.B.

  ```
  ... p(...) throws exception1, exception2,...
  ```

  sonst: Compilerfehler (bis auf Standardausnahmen wie `RuntimeException`, d.h. arithmetische Fehler, Feldfehler etc.)
- „termination model“

Syntax der **Ausnahmebehandlung**:

```
try { <normale Berechnung> }
catch (ExceptionTypeA e1) { <Behandlungsblock dieses Ausnahmetyps,
                             wobei e1 Objekt der Ausnahme ist> }
catch (ExceptionTypeB e2) { <Behandlungsblock> }
⋮
finally { <Anweisungen, die immer (mit oder ohne Ausnahme)
           am Ende ausgefuehrt werden: Aufraeumanweisungen
           wie Schliessen von Dateien etc.> }
```

Die `catch`- oder `finally`-Teile können auch fehlen.

Auslösen von Ausnahmen mittels `throw`:

```
throw new InvalidException ("negative value");
```

Beispielprogramm:

```
⋮
public static void main(String[] args) {
    int i;
    // Pruefe, ob Programm mit Integer-Argumenten aufgerufen wurde:
    try {
        i = Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException e) { // Argument fehlt
        System.out.println ("Bitte Integer-Argument angeben!");
        return; // beende Hauptprogramm
    } catch (NumberFormatException e) { // Argument ist keine Zahl
```

```
        System.out.println ("Argument muss eine ganze Zahl sein!");
        return;
    }
    p(i); // mache mit der normalen Verarbeitung weiter
    ⋮
}
```

4 Sprachmechanismen zur Programmierung im Großen

In diesem Kapitel wollen wir Sprachmechanismen betrachten, die in Programmiersprachen dazu dienen, große Programme oder Programmsysteme zu erstellen. Da der Begriff „groß“ ungenau ist, betrachten wir einige *Merkmale großer Programme*:

- Programm von mehreren Personen erstellt
- prozedurale Abstraktion zur Strukturierung alleine unzureichend, daher:
- Unterteilung der Daten und zugehöriger Operationen zu Einheiten, z.B.: Flugreservierungssystem: Flüge, Flugzeuge, Wartelisten,...
- Aufteilung des Systems in Module
- separate Programmierung dieser Module

Mit den bisherigen Sprachmechanismen (prozedurale Abstraktion) ist dies nur unzureichend realisierbar. Notwendig für solchen großen Systeme ist:

- Unterteilung des Namensraumes
- Kontrolle / Lösung von Namenskonflikten (lokal frei wählbare Namen)

Die hier vorgestellten Techniken sind weitgehend unabhängig von einem bestimmten Programmierparadigma, daher behandeln wir dies in einem eigenen Kapitel. Für konkrete Beispiele wählen wir aber trotzdem die imperative Programmierung, aber viele Konzepte sind auch übertragbar auf funktionale, logische oder nebenläufige Programmiersprachen.

4.1 Module und Schnittstellen

Ein **Modul** ist eine Programmeinheit mit einem Namen und folgenden Eigenschaften:

- logisch oder funktional abgeschlossen (für eine bestimmte Aufgabe zuständig)
- **Datenkapselung:** beinhaltet Daten und Operationen auf diesen Daten
- **Datenunabhängigkeit:** Darstellung der Daten ist unwichtig, d.h. die Darstellung kann auch ausgetauscht werden, ohne dass dies Konsequenzen für den Benutzer des Moduls hat (außer, dass die implementierten Operationen eventuell schneller oder langsamer ablaufen)

- **Informationsverbergung** (information hiding): Die Implementierung dieser Daten und Operationen ist außerhalb des Moduls nicht bekannt.

Somit entspricht ein Modul etwa einem ADT, wobei in imperativen Programmiersprachen das Modul noch einen lokaler Zustand enthalten kann.

Somit hat ein Modul zwei Sichten:

1. **Implementierungssicht:** Wie sind die Daten / Operationen realisiert?
2. **Anwendersicht:** Welche (abstrakten) Daten / Operationen stellt das Modul zur Verfügung? ⇒ **Schnittstelle** des Moduls

Anmerkungen:

- Die Anwender eines Moduls sollen nur die Schnittstelle kennen, d.h. sie dürfen nur die Namen aus der Schnittstelle verwenden ($\rightsquigarrow$ information hiding)
- Ein Modul kann durchaus mehrere Schnittstellen haben ($\rightarrow$ Modula-3, Java)

Beispiel: Modul Table:

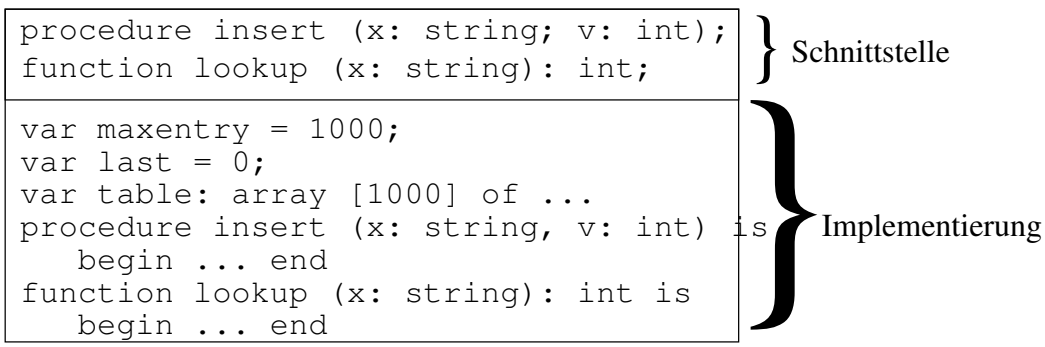


Abbildung 4.1: Beispielhafter Aufbau eines Moduls

Module unterteilen den Namensraum in

- sichtbare Namen (hier: `insert`, `lookup`)
- verborgene Namen (hier: `maxentry`, `last`, `table`)

Ein **Modulsystem**

- prüft den korrekten Zugriff auf Namen
- vermeidet Namenskonflikte (z.B. sollten identische lokale Namen in verschiedenen Modulen keine Probleme bereiten)
- unterstützt häufig die **getrennte Übersetzung** von Modulen:

  - Übersetzung und Prüfung eines Moduls, auch wenn nur die Schnittstellen (und nicht die Implementierung) der benutzten anderen Module bekannt sind ⇝ unabhängige Softwarenentwicklung
  - Erzeugung eines ausführbaren Gesamtprogramm durch einen Binder (Linker)
- unterstützt **Bibliotheken** oder **Pakete**: Zusammenfassung mehrere Module zu größeren Einheiten (z.B. Numerikpaket, Fensterbibliothek, Graphikpakete, ...)

Beispiel: Modula-2:
Schnittstellendefinition:

```
DEFINITION MODULE Table;
  EXPORT QUALIFIED insert, lookup; (* Liste aller sichtbaren Bezeichner *)
  PROCEDURE insert (x: String; v: INTEGER);
  FUNCTION lookup (x: String): INTEGER;
END Table.
```

Implementierung:

```
IMPLEMENTATION MODULE Table;
  VAR maxentry, last: INTEGER;
  VAR table: ARRAY [1..maxentry] OF ...;

  PROCEDURE insert (x: String; v: INTEGER);
  BEGIN
    ⋮ (* Implementierungscode *)
  END insert;

  FUNCTION lookup (x: String): INTEGER;
  BEGIN
    ⋮ (* Implementierungscode *)
  END lookup;

END Table.
```

Benutzung (Import) eines Moduls:

```
IMPORT Table;
```

oder

```
FROM Table IMPORT insert, lookup;
```

Dadurch sind von nun an die Namen `Table.insert` und `Table.lookup` sichtbar und können wie lokale definierte Prozeduren verwendet werden.

Die **modulare Programmierung** ist also durch folgendes Vorgehen charakterisiert:

- Modul $\approx$ ADT
- Programm $\approx$ Menge von Modulen
- gebe in den Schnittstellen nur die wirklich notwendigen Bezeichner bekannt.
- Programmausführung: führe den Rumpf des **Hauptmoduls** aus

4.2 Klassen und Objekte

Ein Modul in einer imperativen Sprachen kann als ADT mit einem Zustand aufgefasst werden. Wenn so ein Modul importiert wird, wird nur genau eine Instanz (ein Modul existiert ja nur einmal) importiert. Dies hat den Nachteil, dass nicht mehrere Instanzen eines Moduls (z.B. mehrere Tabellen) gleichzeitig verwendet werde können.
Eine Lösung hierzu wurde schon früh in der Sprache Simula-67 vorgestellt: Fasse ein Modul als normalen Datentyp auf, von dem man mehrere Instanzen bilden kann (z.B. mehrere Variablendeklarationen vom Typ dieses Moduls). In diesem Fall spricht man von einer **Klasse**. Eine Klasse

- ist ein Schema für Module (hier: **Objekte** genannt),
- entpricht einem Datentyp, und
- unterstützt Instanzenbildung

Wir sprechen von einer **klassen- bzw. objektbasierte Sprache**, wenn die Programmiersprache die Definition und Instanzbildung von Klassen unterstützt.
Eine **objektorientierte (OO) Sprache** ist objektbasiert und unterstützt das Prinzip der Vererbung (vgl. Kap. 4.3).
Eine Klassenvereinbarung hat die Form

$$\texttt{class C } \{m_1; \ldots, m_k\}$$

Anmerkungen:

- `C` ist der Name der Klasse, der verwendbar wie ein Datentyp ist, z.B. in einer Variablendeklaration:

  ```
  C x,y,z;
  ```

- $m_1, \ldots, m_k$: **Merkmale** der Klasse (in Java: **Mitglieder** bzw. **members**).
  Hierbei ist jedes m_i eine
  - **Attributvereinbarung** (Java: **field**) der Form

    $$\tau\, x_1, \ldots, x_n;$$

– **Methodenvereinbarung** der Form

$$\tau\ \texttt{p}(\tau_1\, x_1, \ldots, \tau_n\, x_n)\{\ldots\}$$

Dies entspricht im wesentlichen einer Funktion oder Prozedur. Im Rumpf von `p` kann auf alle Merkmale der Klasse `C` zugegriffen werden.

- Instanzen der Klasse `C` können mittels

```
new C()
```

erzeugt werden. Hierdurch wird also eine neue Instanz der Klasse `C` erzeugt, d.h. ein neues Objekt, das alle Merkmale von `C` enthält.

- Zugriff auf Merkmale eines Objektes `x`: Punktnotation: `x.`m_i
  - Falls m_i ein Attribut ist, dann bezeichnet dies den L- bzw. R-Wert des Attributs
  - Falls m_i eine Methode ist, dann bezeichnet dies einen Prozeduraufruf mit `x` als zusätzlichem Parameter.

    Bei Abarbeitung des Rumpfes: ersetze jedes Vorkommen eines Merkmals m_j durch `x.`m_j

    Dynamische Sichtweise (à la Smalltalk): „sende Nachricht m_i an Objekt `x`“

- andere Auffassung: Objekt = Verbund + Operationen darauf + information hiding
- Zur Kontrolle des “information hiding” erlaubt Java die Spezifikation der Sichtbarkeit durch optionale Schlüsselwörter vor jedem Merkmal:
  - `public`: überall sichtbar
  - `private`: nur in der Klasse sichtbar
  - `protected`: nur in Unterklassen und Programmcode im gleichen Paket sichtbar („package“, vgl. Kapitel 4.6)
  - ohne Angabe: nur im Programmcode des gleichen Pakets sichtbar

Beispiel: Punkte im 2-dimensionalen Raum:

```
class Point {
    public double x,y;

    public void clear() {
        x=0;
        y=0;
    }

    public double distance(Point that) { // distance to that Point
```

```
        double xdiff = x-that.x;  // (1)
        double ydiff = y-that.y;
        return Math.sqrt(xdiff*xdiff+ydiff*ydiff);
    }

    public void moveBy(double x, double y) {
        this.x += x;  // (2)
        this.y += y;
    }
}
```

Anmerkungen:

(1): Hier bezeichnet `x` das Attribut des Objektes, für das die Methode `distance` aufgerufen wird. Dagegen bezeichnet `that.x` das entsprechende Attribut des Parameterobjektes

(2): Das Schlüsselwort `this` bezeichnet immer das Objekt, für das der momentane Aufruf erfolgt, d.h. "`this.x`" bezeichnet das Attribut dieses Objektes. Dagegen bezeichnet `x` den Parameter der Methode `moveBy`. In der Sprache Smalltalk wird das Schlüsselwort `self` statt `this` verwendet.

In diesem Beispiel wäre die Verwendung von "`this`" vermeidbar durch Umbenennung der Methodenparameter. Im Allgemeinen ist "`this`" allerdings notwendig, um z.B. das eigene Objekt als Parameter an andere Methoden zu übergeben.

Anwendung der Klasse `Point`:

```
Point p;
p = new Point();
p.clear();
p.x = 80.0;
p.moveBy(1200.0, 1024.0);
// Nun ist p.x == 1280.0 und p.y == 1024.0
```

Operationale Semantik von Objekten:

Wir können die operationale Bedeutung von Klassendeklarationen und Objektverwendung relativ einfach durch Inferenzregeln definieren.

Klassendeklaration (hier ignorieren wir die Sichtbarkeitsregeln)

$$\langle E, M\rangle \quad \texttt{class C } \{m_1; \ldots; m_k;\} \quad \langle E; C : class\{m_1, \ldots, m_k, E\}, M\rangle$$

Die Speicherung der momentanen Umgebung E ist notwendig, um bei späteren Methodenaufrufen die korrekte Deklarationsumgebung zur Verfügung zu haben.

Objektdeklaration ($\approx$ Referenzvariable):

$$\langle E, M\rangle \quad \texttt{C x} \quad \langle E; x : (l, C), M[l/\texttt{null}]\rangle$$

mit $l \in free(M)$.
Hier wird die Objektvariable `x` wie eine Referenz auf ein Objekt vom Typ `C` behandelt.

Objekterzeugung:

$$\frac{E \vdash^{lookup} x : (l, C) \qquad E \vdash^{lookup} C : class\{\tau_1\, a_1, \ldots, \tau_n\, a_n, \langle Methoden\rangle, E'\}}{\langle E, M\rangle \quad \texttt{x = new C()} \quad \langle E, M[l/l'][l'/i_1]\ldots[l'+n-1/i_n]\rangle}$$

wobei $l', \ldots, l'+n-1 \in free(M)$ und i_j Initialwert vom Typ τ_j
(d.h. reserviere Speicher für alle Attriute, aber nicht für die Methoden)

Zugriff auf ein Attribut:

$$\frac{E \vdash^{lookup} x : (l, C) \qquad E \vdash^{lookup} C : class\{\tau_1\, a_1, \ldots, \tau_n\, a_n, \ldots\}}{\langle E, M\rangle \vdash^{L} x.a_i : M(l) + i - 1}$$

$$\frac{E \vdash^{lookup} x : (l, C) \qquad E \vdash^{lookup} C : class\{\tau_1\, a_1, \ldots, \tau_n\, a_n, \ldots\}}{\langle E, M\rangle \vdash^{R} x.a_i : M(M(l) + i - 1)}$$

Methodenaufruf `x.m(...)`:
Dieser wird im Prinzip wie ein Prozeduraufruf behandelt (daher sparen wir uns eine detaillierte Definition), aber mit folgenden Unterschieden:

- Die Umgebung für die Methodenabarbeitung ist die Umgebung der Klassendeklaration, d.h. die Umgebung, die bei der Deklaration der Klasse gespeichert wurde (s.o.) einschließlich der Deklaration der Klasse selbst, damit auf alle Klassenmerkmale korrekt zugegriffen werden kann.
- Deklariere vor der Abarbeitung des Rumpfes in der lokalen Umgebung:
  `this` $: (l, C)$ falls $E \vdash^{lookup} x : (l, C)$
- Ersetze im Rumpf alle sichtbaren Vorkommen von Merkmalen (d.h. alle Attribute und Methoden) `m` dieser Klasse durch `this.m`
- Die Parameterübergabe ist in Java immer der Wertaufruf, d.h. formale Parameter werden wie lokale Variablen behandelt.

Weitere Konzepte in Klassen und Objekten

Konstruktoren:

Methoden zur Initialisierung von Objekten

- gleicher Name wie Klasse (aber kein Ergebnistyp)

- zusätzliche Parameter (auch mehrere Konstruktoren sind für eine Klasse erlaubt, falls deren Parameter unterschiedlich sind)
- werden nach der Objekterzeugung (`new`) implizit aufgerufen, wobei die aktuellen Parameter die bei "`new`" übergebenen Parameter sind

Beispiel:

```
class Point {
    ...
    Point (double x, double y) {
      this.x=x;
      this.y=y;
    }
}

Point p;
p = new Point();          // nur Erzeugung
p = new Point(1.0,2.0);   // Erzeugung mit Konstruktoraufruf
```

Statische Attribute und Methoden:

(auch: **Klassenattribute und -methoden** genannt)

Falls das Schlüsselwort "`static`" vor Attributen oder Methoden steht, dann gehört dieses Merkmal zur Klasse und nicht zu Objekten diese Klasse, d.h.

- es wird nur einmal repräsentiert (auch ohne Existenz von Objekten)
- der Zugriff erfolgt außerhalb der Klasse durch `<Klassenname>.m`

Beispiel:

```
class Ident {
    public int nr;

    private static int nextid = 0;

    public static int currentnr () {
      return nextid;
    }

    Ident () { // Konstruktor
      nr = nextid;
      nextid++;
    }
}
```

Effekt: jedes `Ident`-Objekt erhält bei seiner Erzeugung eine neue Nummer, die fortlaufend hochgezählt wird.
Zugriff auf den aktuellen Zählerstand: `Ident.currentnr()`

Konstanten und endgültige Methoden:

Falls das Schlüsselwort `final` vor Merkmalen steht, dann ist dieses Merkmal nicht weiter veränderbar. Somit gilt:

- `final` Attribute entsprechen Konstanten

```
class Math {
   ⋮
   static final double PI = 3.1415926535......;
   ⋮
}
```

- `final` Methoden sind nicht redefinierbar in Unterklassen (→ später)
- `final class C {...}`: alle Methoden sind "`final`"

Methode `main`:

Beim Start eines Java-Programms mittels

```
> java C <Parameter>
```

erfolgt ein Aufruf der Methode `main` aus der Klasse `C`. Zu diesem Zweck muss in der Klasse `C` die `main`-Methode wie folgt deklariert sein:

```
class C {
   ⋮
   public static void main(String[] args) {
     ...
   }
}
```

Hierbei bedeutet:

`public`: Die Methode ist öffentlich bekannt.

`static`: `main` ist eine Klassenmethode und kann somit ohne Existenz eines Objektes aufgerufen werden.

`void`: `main` hat keinen Rückgabewert.

`args` ist die Liste der Aufrufparameter.

Rein OO Sprachen:

Konzeptuell ist es elegant, wenn alle Datentypen durch Klassen definiert sind („alles sind Objekte“). Sprachen mit dieser Sichtweise werden auch als rein objektorientierte Sprachen bezeichnet, wie dies z.B. in Smalltalk der Fall ist.
Vorteile:

- einheitliche Sichtweise (wichtig z.B. bei Generizität)
- uniformer Zugriff auf alle Objekte (alles sind Referenzen)

Nachteile:

- Overhead bei Grunddatentypen (int, bool, ...), da dies immer Referenzen sind.
- Sichtweise ist manchmal unnatürlich, z.B. bei Arithmetik in Smalltalk:
  `3 + 4` ≈ sende an das Objekt “`3`” die Nachricht “`+`” mit dem Parameter “`4`”

Ausweg (C++, Java): gemischt OO Sprachen:
unterscheide zwischen einfachen Grunddatentypen (int, bool, ...) und Klassen

Nachteil: Prozeduren, die Objekte (Referenzen) als Parameter verlangen, kann man nicht mit einfachen Datentypen aufrufen.

Lösung in Java: für jeden primitiven Typ existiert eine entsprechende Klasse, die Objekte dieses Typs repräsentiert.

Beispiel: Ganze Zahlen: Klasse `Integer`

```
Integer obj = new Integer (42); // erzeuge Objekt, das Zahl enthaelt
if (ob.intValue() == ...) ... // Wertzugriff
```

Weiterer Aspekt dieser Typklassen: Sammlung wichtiger Prozeduren, die für die „tägliche Programmierung“ recht nützlich sind:

- `obj.toString()` ⇝ Konvertierung zu Stringdarstellung, z.B. `"42"`
- `Integer.parseInt("42")` ⇝ Konvertierung eines Strings in eine Zahl: `42`

Sichtbarkeit von Attributen

Bei unserer Definition der Klasse `Point` sind die Attribute `x` und `y` für alle sichtbar. Hierdurch ergibt sich das potenzielle Problem, dass jeder Benutzer diese ändern könnte.

Lösung: definiere Attribute als “`private`”, was aus softwaretechnischer Sicht i.d.R. sinnvoll ist. Z.B. sind in Smalltalk grundsätzlich alle Attribute immer privat (aber in Unterklassen sichtbar), so dass der Zugriff auf Attribute immer über Methoden erfolgt. Dies kann man natürlich auch in Java mit entsprechenden Sichtbarkeitsdeklarationen erreichen:

```
class Point {
   private double x,y;

   public double x() {
     return x;
   }

   public double y() {
     return y;
   }
   ⋮
}
```

Effekt:

- Zugriff auf Attribut über die entsprechende Methode:

  ```
  p.x()
  ```

- direktes Setzen ist nicht möglich:

  ```
  p.x = 80.0; ⇝ Compilerfehler
  ```

4.3 Vererbung

Wichtiges Prinzip bei der Softwareentwicklung: **Wiederverwendung** existierender Programmteile

Mit den bisherigen Methoden gibt es dazu zwei Möglichkeiten:

- Prozeduren: Diese können benutzt werden, falls der Programmcode weitgehend gleich ist und sich nur durch einige Parameter unterscheidet.
- „cut/copy/paste“: Kopiere Programmcode und modifiziere diesen: Dies mag gut für die Bezahlung nach „lines of code“ sein, ist aber äußerst schlecht für die Lesbarkeit und Wartbarkeit von Programmen.

Objektorientierte Programmiersprachen bieten eine weiter strukturierte Lösung an: **Vererbung**
Hierunter versteht man die Übernahme von Merkmalen einer **Oberklasse** A in eine **Unterklasse** B. Zusätzlich kann man in B neue Merkmale hinzufügen oder Merkmale abändern (überschreiben). Wichtig ist aber, dass man nur diese Änderungen in B beschreiben muss, was zu einer übersichtlichen Struktur führt.
Beispiel: Pixel (Bildschirmpunkte) sind Punkte und haben eine Farbe.

A
↓
B

Abbildung 4.2: Vererbungsstruktur

```
class Pixel extends Point {  // Pixel erbt von Point
    Color color;

    public void clear () {
        super.clear();
        color = null;
    }
}
```

Anmerkungen:

- "`extends Point`": Erbe alle Merkmale von `Point`. Damit ist `Pixel` eine **Unterklasse** von `Point` und hat die Attribute `x`, `y` (von `Point`) und `color`.
- "`clear()`": diese Methode existiert auch in Point und wird vererbt, aber in `Pixel` wird diese redefiniert. Ist also `p` ein `Pixel`-Objekt, dann bezeichnet "`p.clear()`" die neu definierte Methode.
- "`super.clear():`" falls Methoden überschrieben werden, ist die vererbte Methode gleichen Namens nicht mehr zugreifbar. Aus diesem Grund gibt es das Schlüsselwort "`super`", mit dem man auf Methoden der Oberklasse verweisen kann, d.h. `super.clear()` bezeichnet die `clear`-Methode der Oberklasse `Point`.

Beachte: Unterschied zwischen „überschreiben“ und „überladen“:

Überschreiben: Redefinition geerbter Methoden (gleicher Name und gleiche Parametertypen) ⇒ geerbte Methode nicht mehr zugreifbar (Ausnahme: `super`)

Überladen: Definition einer Methode mit gleichem Namen aber unterschiedlichen Parametertypen (auch innerhalb einer Klasse) ⇒ beide Methoden (je nach Typ der aktuellen Parameter) sind zugreifbar

Beispiel:

```
class O {
    int id(int x) {
        return x;
    }
    double id(double x) {
        return -x;
```

```
    }
  }
```

$\Rightarrow$ `new O().id(1)` $\rightsquigarrow$ 1
$\Rightarrow$ `new O().id(1.0)` $\rightsquigarrow$ -1.0

Überladung (overloading) ist unabhängig von OO Programmiersprache, z.B. existiert dieses Konzept auch in Ada, funktionalen Sprachen etc. Ein klassischer Fall von Überladung ist die Operation "`+`":

- `1+3`: Addition auf ganze Zahlen
- `1.0+3.0`: Addition auf Gleitkommazahlen
- `"af"+"fe"`: Stringkonkatenation

Bei der Vererbung sollte man zwei konzeptionelle Sichtweisen unterscheiden:

Modulsichtweise: Durch Vererbung wird ein Modul (Oberklasse) erweitert bzw. modifiziert, d.h. Vererbung $\approx$ Programmmodifikation (wobei nur die Modifikation aufgeschrieben wird).

Typsichtweise: durch Vererbung wird ein **Untertyp** B (ein speziellerer Typ) der Oberklasse A definiert, d.h. B-Objekte können überall eingesetzt werden, wo A-Objekte verlangt werden: „B ist ein A“ (B is-a A) (aber mit speziellen Eigenschaften)

- Viele OOPS unterstützen beide Sichtweisen
- Typsichtweise ist problematisch beim Überschreiben von Methoden, denn dadurch kann die Unterklasse semantisch völlig verschieden von der Oberklassen sein. Beachte: Falls B ein Untertyp von A ist, sollte jedes Element aus B auch ein Element aus A sein.
- **Semantische Konformität** (ist jedes B-Element auch ein A-Element?) ist im allgemeinen unentscheidbar. Aus diesem Grund wird die Konformität abgeschwächt zu einer leichter überprüfbaren Konformität:
  B ist **konform** zu A :$\Leftrightarrow$ B-Objekte sind überall anstelle von A-Objekten verwendbar, ohne dass es zu Typfehlern kommt.

Wie können sichere Typsysteme für OO Sprachen aussehen, die die Konformität garantierten?
Hierzu kann man sich mit folgender Approximation begnügen:
B ist konform zu A, falls für alle Merkmale m von A ein Merkmal m von B existiert mit:

- Falls m ein Attribut vom Typ τ in A ist, dann ist m ein Attribut vom Typ τ in B
- Falls m eine Methode in A mit Typ $\tau_1, \ldots, \tau_n \to \tau$, dann ist m eine Methode in B mit dem Typ $\tau'_1, \ldots, \tau'_n \to \tau'$ und

– τ_i ist Untertyp von τ_i' (**Kontravarianz** der Argumente)
– τ' ist Untertyp von τ (**Kovarianz** der Ergebnisse)

Pragmatische Lösungen:

- Smalltalk: keine Konformitätsprüfung (da es keine Parametertypen gibt) $\rightsquigarrow$ Laufzeitfehler
- Eiffel: kovariante Argumenttypen, Prüfung durch Binder, Typunsicherheiten sind bekannt
- Java: bei Überschreibung konforme Argumenttypen (d.h. gleich), sonst wird dies als Überladung interpretiert (dies ist eine vereinfachte Darstellung: in Wirklichkeit ist alles überladen und die Auswahl einer Methode erfolgt über ein „most specific match"-Algorithmus)

Beispiel für eine Objekthierarchie: Klassen für primitive Datentypen in Java:

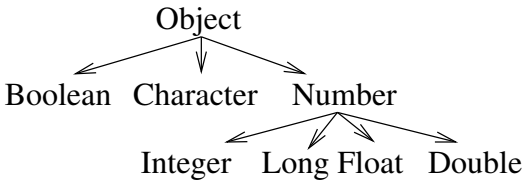


Abbildung 4.3: Objekthierarchie in Java.

Die Klasse `Object` ist dabei Urklasse aller Objekte, d.h. jede Klasse ohne "`extends`" ist automatisch Unterklasse von `Object`. Dies ist ganz nützlich, um Polymorphie auszudrücken. Hierbei bedeutet **Polymorphie**, dass Objekte oder Methoden mehrere Typen haben können.

Beispiel: Feld mit beliebigen Elementen:

```
Object[] a = new Object[100];
a[1] = new Point();
a[2] = new Integer(42);
a[3] = new Pixel();
```

Beachte: alle Elemente in `a` haben den Typ `Object`. Dies bedeutet, dass z.B. `a[1].clear()` nicht zulässig ist.
Falsch dies gewünscht ist, müssen explizite Typkonversionen (**type cast**) eingefügt werden:

```
((Point)a[1]).clear()
```

Hierbei beinhaltet `(Point)` eine Laufzeitprüfung, ob `a[1]` ein `Point`-Objekt ist.
Somit führt z.B. `(Point)a[2]` zu einem Laufzeitfehler.

Dynamische Bindung

Bei Vererbung ist zur Compilationszeit i.allg. die Bindung einer Methodennamens zu einer Methode in einer Klasse nicht berechenbar. Dies ist eine Konsequenz der Polymorphie. Die Zuordnung des Methodennamens zu einer Methode wird auch als **dynamische Bindung** von Methoden bezeichnet.
Beispiel:

```
Point p;
⋮
p = new Pixel(); // zulaessig, da Pixel auch ein Point ist
p.clear(); // Methode in Pixel, da p ein Pixel ist!
p.moveBy(1.0,1.0) // Methode in Point, da Pixel diese ererbt
```

Operational bedeutet dies: Suche in Klassenhierarchie, beginnend bei der Klasse des aktuellen Objektes, nach der ersten Definition dieser Methode. Hierbei gibt es allerdings eine Ausnahme: Ein **Konstruktor** ist eine Methode der Klasse, die **nicht** vererbt wird. Früheres Beispiel:

```
class Point {
  ⋮
  Point (double x,y) { ... }
}
```

Wie kann man die Funktionalität des `Point`-Konstruktors in der Unterklasse nutzbar machen? Ebenfalls mittels `super`, wobei `super` dann für den Konstruktor der Oberklasse steht (analog: `this` ≈ Konstruktor dieser Klasse):

```
class Pixel extends Point {
   ⋮
   Pixel (double x, double y, Color c) {
      super(x,y); // Aufruf des Point-Konstruktors mit x,y
      color=c;
   }
}
```

Wichtig: `super(...)` bzw. `this(...)` muss die erste Anweisung im Rumpf sein. Falls diese fehlt, erfolgt ein impliziter Aufruf von `super()` (d.h. es wird der Konstruktor der Oberklasse ohne Argumente aufgerufen).

Reihenfolge der Konstruktorabarbeitung/Objektinitialisierung:

1. Führe Konstruktor der Oberklasse aus
2. Initialisiere Attribute entsprechend ihrer Initialwerte
3. Führe Rumpf des Konstruktors aus

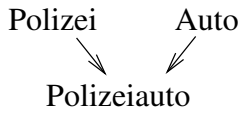
Diese Reihenfolge ist evtl. relevant bei Seiteneffekten in Methodenaufrufen von Konstruktoren.

4.4 Schnittstellen

Unter der **Schnittstelle einer Klasse** verstehen wir die Menge aller anwendbaren Merkmale dieser Klasse.
Falls die Schnittstelle keine Attribute enthält, entspricht dies einem ADT (da nur die Operationen sichtbar sind).
Vererbung entspricht dann der Erweiterung von Schnittstellen.

Mehrfachvererbung bedeutet, dass aus mehreren Oberklassen vererbt wird. Hierbei können allerdings eine Probleme auftreten:

- Konflikte bei namensgleichen Merkmalen in Oberklassen
  (Lösung: explizite Beseitigung der Konflikte: C++, Eiffel)
- Mehrfachvererbung kann leicht zu einer falschen Problemmodellierung führen:



  Dies ist eine falsche Modellierung: ein Polizeiauto ist keine Polizei, sondern nur ein Auto.
- Smalltalk, Java: nur Einfachvererbung erlaubt
- Java: Mehrfachvererbung nur für Schnittstellen (s.u.)

Abstrakte Klassen

Wichtig ist bei einem guten Entwurf, logisch zusammengehörige Teile zusammenzufassen und diese möglichst wiederzuverwenden.
Häufig Situation: Module / Klassen haben gemeinsame Teile, sind aber nicht Untertyp einer gemeinsamen realen Oberklasse. Hier kann man sich durch das Entwurfsprinzip der **Faktorisierung** behelfen: verschiebe gemeinsame Anteile in eine gemeinsame Oberklasse. Falls diese Oberklasse keine Instanzen hat, spricht man von einer **abstrakten Klasse**.
Charakteristik abstrakter Klassen: einige Merkmale sind bekannt, andere Merkmale werden in Unterklassen definiert.

Beispiel: Benchmark-Klasse: abstrakt, da konkreter Benchmark noch unbekannt

```
abstract class Benchmark { // abstract: enthaelt mind. eine abstrakte Methode
    abstract void benchmark(); // abstract: der Rumpf ist unbekannt
```

```
    public long repeat(int count) {
        long start = System.currentTimeMillis();
        for (int i=0; i<count; i++) benchmark ();
        return (System.currentTimeMillis() - start);
    }
}
```

Beachte: der Aufruf "`new Benchmark()`" ist nicht erlaubt!

Als Beispiel betrachten wir einen konkreten Benchmark zur Zeitmessung für Methodenaufrufe:

```
class MethodBenchmark extends Benchmark {
    void benchmark() { } // leerer Rumpf, d.h. nur Aufruf wir gemessen

    public static void main(String[] args) {
        int count = Integer.parseInt(args[0]);
        long time = new MethodBenchmark().repeat(count);
        System.out.println ("msecs: "+time);
    }
}
```

Merkmale abstrakter Klassen:

- Teile der Implementierung sind bekannt, andere nicht
- es existieren keine Objekte dieser Klasse
- falls alle Teile unbekannt, aber Merkmalsnamen/-typen bekannt ⇒ benutze Interfaces in Java

Interfaces in Java

Technik zur besseren Abstraktion / Modularisierung und zur Realisierung von Mehrfachvererbung

Interface (in Java)

- abstrakte Klasse nur mit abstrakten Methoden (und Konstanten)
- jede Klasse kann *beliebig viele* Interfaces implementieren (d.h. sie muss für jede Interface-Methode eine Methode gleichen Namens und Typs enthalten)

Beispiel: Interfaces für Tabellen:

```
interface Lookup {  // statt abstract class
    Object lookup(String name);
}
```

```
interface Insert {
    void insert(String name, Object value);
}
```

Interfaces können wie abstrakte Klasse benutzt werden:

```
void processValues (String[] names, Lookup table) {
    // table ist Objekt einer Klasse, die Lookup implementiert
    ...
    table.lookup(...);
    ...
}
```

Konkrete Implementierung einer Tabelle:

```
class MyTable implements Lookup, Insert {
    private String[] names;
    private Object[] values;
    ... // Implementierung von lookup() + insert()
}
```

Anmerkung: da Interfaces nichts implementieren, werden die Konflikte, die bei Mehrfachvererbung auftreten können, vermieden.

4.5 Generizität

Wir haben gesehen, dass Klassen und Module ADTs entsprechen. In Kapitel 2.2.4 haben wir als Verallgemeinerung auch *parametrisierte ADTs* kennengelernt. Gibt es auch ein ähnliches Konzept für parametrisierte Klassen oder Module in Programmiersprachen? Tatsächlich sind entsprechende Konzepte zur **Generizität** in verschiedenen Programmiersprachen entwickelt worden. In OO Programmiersprachen sind (Typ)Parameter Objekttypen. Betrachten wir dazu Listen mit beliebigen Objekten als Elementen:

```
class List {
    Object elem;
    List next;
}
```

Da der Elementtyp `Object` ist, können die Listen beliebige Objekte als Elemente enthalten.
Der Nachteil ist allerdings, dass damit keine *uniformen Listen* (z.B. Integer- oder Character-Listen, etc.) garantiert werden können, wodurch es leicht zu Programmierfehlern kommen kann. Um dies zu vermeiden, gibt es in existierenden Programmiersprachen verschiedene Ansätze:

Ada, Modula-3: Module mit Parametern:

```
GENERIC MODULE List(Elem)
  ⋮
  ... Elem.type ...
  ⋮

MODULE IntList = List(IntegerElement)
END IntList;
```

Eiffel erlaubt Klassen mit Parametern:

```
class List[T] ...
```

wobei `T` ein Typparameter ist.

Java hat ab Version 5 eine Erweiterung um Generizität (**generics**):

```
class List<etype> {
    etype elem;
    List<etype> next;
}
```

Instanzbildung: Angabe des konkreten Typs:

```
List<String> names;
```

Funktionale Programmiersprachen (Haskell, SML) erlauben parametrischen Polymorphismus bei Datentypen und Funktionen (→ Kapitel 5).

4.6 Pakete (packages)

- Strukturierung von Klassen zu größeren Einheiten (Pakete/packages in Java)
- Jede Klasse gehört zu einem Paket: Angabe zu Beginn des Programms:

  ```
  package mh.games.tetris;
  ```

- Jede Klasse eines Pakets ist ansprechbar über Paketnamen (Vermeidung von Namenskonflikten)

  ```
  java.util.Date now = new java.util.Date();
  ```

  Hierbei ist `java.util` der Paketname und `Date` der Name der Klassen in diesem Paket.

- Import kompletter Pakete: Sichtbarmachen der Paketklassen

```
import java.util.Date; // oder java.util.*: alle Klassen
⋮
Date now = new Date();
```

- Viele Pakete sind in der Java API vorhanden, z.B.:
  `java.applet`: alles für Applet-Programmierung
  `java.awt`: Graphik, Fenster etc.

5 Funktionale Programmiersprachen

Imperative Programmiersprachen basieren auf der *Zuweisung* als einfachste Anweisung. Da jede Zuweisung ein *Seiteneffekt* auf Objektwerten ist, kann man die imperative Programmierung auch als „*seiteneffektbasierte Programmierung*“ interpretieren. Dies führt häufig zu Problemen:

- Effekte von Programmeinheiten (z.B. Prozeduren, Funktionen) sind schwer kontrollierbar
- Es können viele kleine Fehler durch falsche Reihenfolgen (`i++`, ...) passieren.
- Die Auswertungsreihenfolge ist auch im Detail sehr relevant.

Beispiel: Betrachten wir ein C-Programm:

```
x=3;
y=(++x)*(x--);
```

Hier ist das Ergebnis von `y` abhängig von der Auswertungsreihenfolge des `*`-Operators.

Beispiel: Fakultätsfunktion imperativ:

```
int fac(int n) {
    int z=1;
    int p=1;
    while (z<n+1) {
        p=p*z;
        z++;
    }
    return p;
}
```

Schon bei diesem kleinen Programm gibt es viele mögliche Fehlerquellen:

- Initialisierung: `z=0` oder `z=1`? Ebenso für `p`?
- Abbruchbedingung: `z<n+1` oder `z<n` oder `z`$\leq$`n`?
- Reihenfolge bei Zuweisungen: `z++` hinter oder vor `p=p*z`?

Dagegen ist die mathematische Definition wesentlich klarer und daher weniger fehleranfällig:

$fac(0) = 1$
$fac(n) = n * fac(n-1)$ `falls` $n > 0$

Funktionale Programmiersprachen:

- keine Zuweisung, keine Seiteneffekte
- nur Ausdrücke und Funktionen
- außerdem: Funktionen sind „Werte 1. Klasse “ (first class values / citizens), d.h. sie können selbst Argumente oder Ergebnisse sein ($\rightsquigarrow$ Funktionen höherer Ordnung)

Wichtiges Prinzip rein funktionaler Programmiersprachen:

Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab (und nicht vom Zeitpunkt der Auswertung!)

Dieses Prinzip wird auch als **referentielle Transparenz** bezeichnet.

Konsequenz:

- Wert von `x+y` hängt nur vom Wert von `x` bzw. `y` ab
- Variablenänderung (wie z.B. `(++x)*(x--)`) unzulässig
- Variablen sind Platzhalter für Werte (wie in der Mathematik) und keine Namen für veränderbare Speicherzellen
- Funktionen haben keine Seiteneffekte

Ansonsten könnte man leicht Programme konstruieren, bei denen die Reihenfolge der Auswertung arithmetischer Ausdrücke relevant ist:

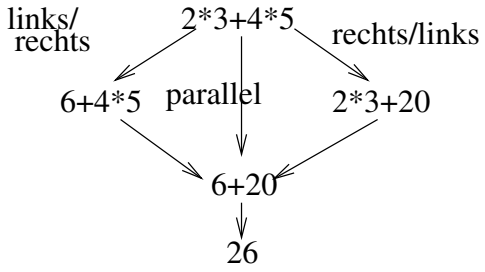
```
var y,z: int;

function f(x: int): int is
begin
   z:=2;
   return (2*x)
end;

⋮
z:=1;
y:=f(2)*z;
```

$\Rightarrow$ Links-Rechts-Auswertung der Multiplikation: `y=8`
$\Rightarrow$ Rechts-Links-Auswertung der Multiplikation: `y=4`

Vorteile der referentiellen Transparenz:

- mathematisches Substitutionsprinzip (ersetze Ausdruck durch seinen Wert) gültig

- einfache operationale Semantik („Gleiches durch Gleiches ersetzen“)
- ermöglicht einfache Optimierung, Transformation, Verifikation von Programmen (z.B. oben: ersetze `f(2)` durch `4` im Programmcode ist eine unzulässige Optimierung)
- flexible Auswertungsreihenfolge:



  $\Rightarrow$ einfache Ausnutzung paralleler Rechnerarchitekturen
- lesbare, zuverlässige, weniger fehleranfällige Programme
  (da Reihenfolgen bei Zuweisung nicht relevant sind)

Betrachten wir hierzu ein weiteres Beispiel: **Sortieren mittels Quicksort**
Die klassische Formulierung in einer imperativen Sprache sieht etwa wie folgt aus:

```
procedure qsort(l,r: index);
var i,j: index; x,w: item
begin
   i := l;  j := r;
   x := a[(l+r) div 2];
   repeat
      while a[i] < x do i := i+1;
      while x < a[j] do j := j-1;
      if i <= j then
      begin w := a[i]; a[i] := a[j]; a[j] := w;
         i := i+1; j := j-1
      end
   until i > j;
   if l < j then qsort(l,j);
   if i < r then qsort(i,r);
end
```

Hier kann man sehr viele Fehler machen, z.B. bei der Initialisierung, bei der Wahl der Abbruchbedingungen, der Reihenfolge der Zuweisung etc. Dies führt dazu, dass man es wohl kaum schafft, die erste Version des Programms fehlerfrei aufzuschreiben. Dagegen ist die Formulierung von Quicksort als funktionales Programm viel klarer und auch weniger fehleranfällig:

```
qsort [] = []
qsort (x:l) = (qsort (filter (<x) l))
              ++ [x]
              ++ (qsort (filter (>=x) l))
```

Im folgenden schauen wir uns die Syntax und Semantik funktionaler Programmiersprachen genauer an, um die Bedeutung dieses Programms genau zu verstehen.

5.1 Syntax funktionaler Programmiersprachen

Als konkrete funktionale Programmiersprache betrachten wir die Sprache Haskell, die heutzutage als Standard für eine moderne rein funktionale Sprache gelten kann.

Ein **funktionales Programm** ist eine Menge von Funktionsdefinitionen (und Datentypdeklarationen, später)
Eine **Funktionsdefinition** ist ähnlich wie in imperativen Sprachen, allerdings ist der Rumpf nur ein Ausdruck (weil es keine Seiteneffekte gibt!).

Konkret erfolgt die Definition einer Funktion durch eine **Gleichung** oder **Regel**:

`f` x_1 `...` x_n `= e`

Hier ist `f` der Funktionsname, x_1,...,x_n die Parameter der Funktion und `e` der Rumpf der Funktion.

Beispiel:

```
square x = x*x
```

Beachte:

- Anwendung durch Hintereinanderschreibung:
  statt `f(x1,...,xn)` hier `f x1 ... xn` oder auch `(f x1 ... xn)`
- übliche Infixschreibweise bei (mathematische) Operatoren zulässig (z.B. `3*4`)
- Rechnen: ersetze Aufruf durch Rumpf, wobei formale Parameter durch aktuelle Argumente ersetzt werden:

  `square 5` $\rightarrow$ `5*5` $\rightarrow$ `25`

- auch mehrere Alternativen mit Bedingungen möglich:

```
f x1 ... xn | b1 = e1
            | b2 = e2
            ⋮
            | bn = en
```

(wobei `bn` auch `otherwise` sein kann, was für `True`, also immer erfüllt steht)

Beispiel:

```
fac n | n ==0 = 1
      | n > 0 = n * fac (n-1)
```

Alternativ:

```
fac n = if n==0 then 1 else n*fac(n-1)
```

Lesbarere Alternative: **musterorientierte Definitionen**

- mehrere Gleichungen für eine Funktion
- Muster (Datenterme) statt Variablen als formale Parameter
- Bedeutung: wende Regel an, falls Muster „passt“

Vorteil des musterorientierten Stils:

- kurze, prägnante Definitionen
- leicht lesbar
- leicht verifizierbar

Als Beispiel betrachten wir die Funktion `and` zur Konjunktion boolescher Werte:

- musterorientiert:

```
and True  x = x
and False x = False
```

- mit Bedingungen:

```
and x y | x == True  = y
        | x == False = False
```

- mit Fallunterscheidung:

```
and x y = if x == True then y else False
```

Datentypen: elementare Typen wie üblich vordefiniert:

- Wahrheitswerte: `Bool`, Konstanten: `True`, `False`
- Ganze Zahlen: `Int` (oder `Integer` für beliebig große Zahlen), Konstanten: 0,1,-2,...
- Gleitkommazahlen: `Float`, Konstanten: `0.0`, `1.5e-2`,...

- Zeichen: `Char`, Konstanten: `'a'`, `'b'`, `'\n'`,...
- Zeichenketten: `String` (Liste von `Char`), z.B. `"abcd"`

Besonderheit funktionaler Sprachen: einfache Definition neuer **(algebraischer) Datentypen** durch Aufzählung der Konstruktoren ($\approx$ Vereinigungstypen), wobei auch Parametrisierung erlaubt ist.
Allgemeine Form:

`data` $T\ a_1\ \ldots\ a_n$ `=` $C_1\ \tau_{11}\ \ldots\ \tau_{1m_1}$ `|` $\ldots$ `|` $C_k\ \tau_{k1}\ \ldots\ \tau_{km_k}$

Hier sind

- T der Name des neuen Datentyps (Typname)
- $a_1, \ldots, a_n$ Typvariablen, d.h. die Parameter des generischen Datentyps
- $C_1, \ldots, C_k$ die Konstruktor des Datentyps
- $\tau_{i1}, \ldots, \tau_{im_i}$ die Argumenttypen des Konstruktors C_i, die aufgebaut sind aus anderen Typnamen und $a_1, \ldots, a_n$

Mittels algebraischer Datentypen können wir verschiedene andere Typstrukturen definieren:

- Aufzählungstypen:

```
data Bool = True | False
data Color = Red | Yellow | Blue | Green
```

- Parametrisierte Listen:

```
data List a = Nil | Cons a (List a)
```

  In Haskell sind diese mit einer bestimmten Syntax vordefiniert:
  Wir schreiben "`[`τ`]`" statt "`List` τ" und "`:`" statt "`Cons`", wobei "`:`" als rechtsassoziativer Infixoperator definiert ist. So können wir die Liste mit den Elementen 1,2,3 in Haskell wie folgt schreiben:

```
1:(2:(3:[]))   1:2:3:[]    [1,2,3]
```

- Binärbäume:

```
data BTree a = Leaf a | Node (BTree a) (BTree a)
```

  z.B. `Node (Leaf 1) (Node (Leaf 2) (Leaf 3)) :: BTree Int`
  "`::`" bedeutet: „hat den Typ“

- Wir können auch allgemeine Bäume definieren, wobei die Knoten nun Listen von Teilbäumen besitzen können:

```
data Tree a = Leaf a | Node [Tree a]
```

- Tupel sind vordefiniert:
  $(e_1, \ldots, e_n)$`::`$(\tau_1, \ldots, \tau_n)$ falls e_i`::`τ_i für $i = 1 \ldots, n$ gilt
  z.B.: `(1,True,'a')::(Int,Bool,Char)`

Allgemeine Funktionsdefinition

Allgemein können in Haskell Funktionen durch mehrere(!) Gleichungen der Form

```
f t1 ... tn  | b1 = e1
             | b2 = e2
             :
             | bn = en
```

definiert werden. Hierbei ist

- `f`: Funktionsname
- $t_1, \ldots, t_n$: Muster, bestehend aus Konstruktoren und Variablen, wobei jede Variable höchstens einmal vorkommt
- $b_1, \ldots, b_n$: boolsche Ausdrücke
- $e_1, \ldots, e_n$: beliebige Ausdrücke, bestehend aus Variablen, Konstruktoren, Funktionsaufrufe, wobei die b_i und e_i nur Variablen aus $t_1, \ldots, t_n$ enthalten dürfen.

Beispiel: Konkatenation von Listen

```
conc []     ys = ys
conc (x:xs) ys = x : (conc xs ys)
```

(vordefiniert in Haskell als Infixoperator "`++`")

Beachte:

- musterorientierter Stil: keine Erhöhung der Berechnungsstärke, sondern klarere Programmstruktur
- Muster in linken Regelseiten entsprechen Prädikaten (hat das Argument diese Form?) und Selektoren (Zugriff auf Teilargumente)
  $\Rightarrow$ Muster vereinfachen Funktionsdefinitionen

Beispiel: Konkatenation ohne Muster: Notwendige Hilfsfunktionen (Selektoren)
`head xs`: erstes Element von `xs`
`tail xs`: Rest von `xs` (ohne `head xs`)
Nun können wir `conc` so definieren:

```
conc xs ys = if xs==[]
             then ys
             else (head xs) : (conc (tail xs) ys)
```

Hier ist die Definition ohne Muster noch relativ einfach. Dies kann aber bei mehreren Regeln aufwändig werden:

```
or True  x     = True
or x     True  = True
or False False = False
```

kann man auch schreiben als

```
or x y = if x==True
         then True
         else if y==True
              then True
              else if x==False && y==False
                   then False
                   else error "..."  -- undefiniert, kommt nicht vor
```

Eine weitere Übersetzungsalternative ist die Verwendung von `case`-Ausdrücken (dies wird intern in Haskell benutzt). Allgemein:

`case` e `of`
$C_1\ x_{11} \ldots x_{1r_1} \rightarrow e_1$
$\vdots$
$C_k\ x_{k1} \ldots x_{kr_k} \rightarrow e_k$

wobei e ein Ausdruck vom Typ τ, $C_1, \ldots, C_k$ Konstruktoren des gleichen Datentyps τ und x_{ij} Selektorvariablen sind (diese können in e_i benutzt werden)

Vorteile von `case`-Ausdrücken gegenüber `if-then-else`:

- implizite Selektoren durch die Selektorvariablen x_{ij}
- mehr als zwei Alternativen sind möglich

Beispiel:

```
conc xs ys = case xs of
               []   → ys
               z:zs → z : cons zs ys
```

Funktionale Programmiersprachen haben einen „Pattern-Matching-Compiler“:

- Übersetzung mehrerer Gleichungen für eine Funktion f in eine Gleichung der Form

  $$f\ x_1 \ldots x_n = e$$

  wobei in e auch `case`-Ausdrücke vorkommen können (vgl. `conc`).

- Strategie des Pattern Matching:
  - für jede Regel Muster von links nach rechts testen
  - alle Regeln von oben nach unten ausprobieren
  - erste passende Regel „feuert“

5.2 Operationale Semantik

Das Grundprinzip der Auswertung bei funktionalen Programmiersprachen ist die Ersetzung „Gleiches durch Gleiches“ mittels Anwendung von Funktionsgleichungen/Regeln von links nach rechts.
Aus diesem Grund ist es wichtig festzulegen, wann eine Regel anwendbar ist. Da in Regeln auch Muster vorkommen können, verlangt dieser Begriff einige Vorbereitungen.

Ein **Ausdruck** ist eine Variable x oder eine Anwendung $(f\ e_1 \ldots e_n)$ wobei f eine Funktion oder ein Konstruktor ist und $e_1, \ldots, e_n$ wiederum Ausdrücke sind ($n \geq 0$). Wir ignorieren hier erst einmal die Forderung nach wohlgetyptheit, denn darauf gehen wir ein einem späteren Kapitel ein.

Eine **Position** ist eine Liste natürlicher Zahlen, die einen Teilausdruck identifiziert.

Ein **Teilausdruck** $e|_p$ („e an der Stelle p“), wobei p eine Position im Ausdruck e ist, ist formal wie folgt definiert:

$$e|_\epsilon = e \qquad (\epsilon\text{: leere Liste, Wurzelposition})$$

$$(f\ e_1 \ldots e_n)|_{i \cdot p} = e_i|_p$$

Beispiel: Ausdruck und Positionen: $\underbrace{(}_{\epsilon}\ \texttt{and}\ \underbrace{\texttt{True}}_{1}\ \underbrace{(}_{2}\ \texttt{and}\ \underbrace{\texttt{x}}_{2.1}\ \underbrace{\texttt{y}}_{2.2}\ ))$

Die **Ersetzung** eines Teilausdrucks in einem Ausdruck e an der Position p durch einen neuen Teilausdruck t wird durch $e[t]_p$ notiert. Formal ist dies wie folgt definiert:

$$e[t]_\epsilon = t$$

$$(f\ e_1 \ldots e_n)[t]_{i \cdot p} = (f\ e_1 \ldots e_{i-1}\ e_i[t]_p\ e_{i+1} \ldots e_n)$$

Eine **Substitution** ist eine Ersetzung von Variablen durch Ausdrücke, wobei gleiche Variablen durch gleiche Ausdrücke ersetzt werden. Wir notieren Substitutionen in der Form

$$\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

Formalisierung der Anwendung einer Substitution auf einen Ausdruck:

$$\sigma(x_i) = t_i \quad (i = 1, \ldots, k)$$

$$\sigma(x) = x \quad \text{falls } x \neq x_i \quad \forall i = 1, \ldots, k$$

$$\sigma(f\, e_1 \ldots e_n) = f\ \sigma(e_1) \ldots \sigma(e_n)$$

Damit haben wir nun alle Begriffe bereitgestellt, um einen Ersetzungsschritt („ersetze Gleiches durch Gleiches“) zu definieren. Ein **Ersetzungsschritt**, geschrieben $e \to e'$, ist möglich, falls $l = r$ eine Funktionsgleichung ist, p eine Position in e und σ eine Substitution mit $\sigma(l) = e|_p$ und $e' = e[\sigma(r)]_p$. In diesem Fall heißt $e|_p$ auch **Redex** (reducible expression) von e.

Eine funktionale Berechnung ist eine Folge von Ersetzungsschritten. Wenn der sich ergebende Ausdruck keine definierten Funktionen mehr enthält, ist diese erfolgreich:

Eine **erfolgreiche Berechnung** eines Ausdrucks e ist eine endliche Folge

$$e_1 \to e_2 \to \cdots \to e_n$$

wobei e_n keine definierten Funktionen enthält. In diesem Fall heißt e_n auch Wert von e_1:

$$e_n = value(e_1)$$

Bisher haben wir nur Funktionsregeln ohne Bedingungen betrachtet. Im allgemeinen Fall sind einige Erweiterungen einzubeziehen:

1. Regeln für `if-then-else`: `if-then-else`-Ausdrücke können wir als Funktionsanwendung in mixfix-Schreibweise auffassen, die durch folgende Regeln definiert sind:

   ```
   if True  then x else y = x
   if False then x else y = y
   ```

2. Regeln für `case` (Regelschema, keine wirklichen Funktionsregeln!):

   ```
   case (C x1 ... xn) of
     ⋮
     C x1 ... xn  → e
     ⋮                    = e
   ```

3. Bedingte Regeln: Transformiere

l `|` b_1 `=` r_1
⋮
`|` b_k `=` r_k

in unbedingte Gleichung:

l `= if` b_1 `then` r_1 `else`
`if` b_2 `then` r_2 `else`
⋮
`if` b_k `then` r_k `else error ...`

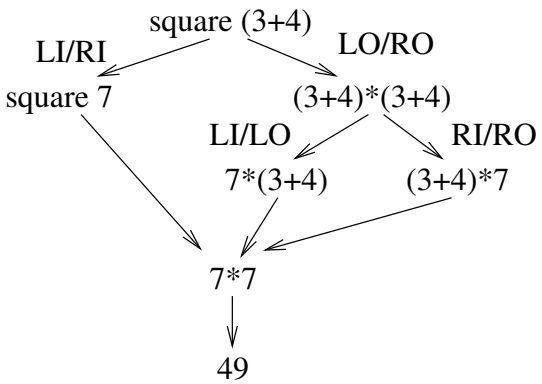
4. Primitive Funktionen: diese können wir so interpretieren, dass sie durch eine unendliche Menge von Gleichungen definiert sind, wie z.B.

```
0+0 = 0          3<5 = True
⋮                ⋮
3+4 = 7          3==5 = False
⋮                ⋮
```

Im Allgemeinen gibt es mehrere mögliche Berechnungen für einen gegebenen Ausdruck. Betrachten wir z.B. die Funktion

```
square x = x*x
```

Dann gibt es für den Ausdruck "`square (3+4)`" folgende Berechnungen:



Eine **Auswertungsstrategie** legt für jeden Ausdruck den zu ersetzenden Redex fest. Wichtige Auswertungsstrategien sind z.B.

- **Innermost:** der ersetzte Redex enthält keine anderen Redexe

- **Leftmost Innermost (LI):** der ersetzte Redex ist der linkeste innerste
- **Rightmost Innermost (RI):** der ersetzte Redex ist der rechteste innerste
- **Outermost:** der ersetzte Redex ist nicht Teil eines anderen Redex
- **Leftmost Outermost (LO):** der ersetzte Redex ist der linkeste äußerste
- **Rightmost Outermost (RO):** der ersetzte Redex ist der rechteste äußerste

Strikte funktionale Sprachen (Lisp, Scheme, SML): LI-Strategie außer `if-then-else`, `case` (diese werden outermost ausgewertet); dies entspricht der Parameterübergabe call-by-value.

Nicht-strikte funktionale Sprachen (Miranda, Gofer, Haskell): LO-Strategie; dies entspricht der Parameterübergabe call-by-name

Im obigen Beispiel führen alle Strategien zum gleichen Ergebnis. Dies muss nicht immer so sein, allerdings gilt:
Konfluenz / Eindeutigkeit der Werte: Falls Ableitungen

$$e \to \cdots \to e_1$$

und

$$e \to \cdots \to e_2$$

existieren, wobei e_1, e_2 Werte sind, dann gilt: $e_1 = e_2$.

Voraussetzung hierfür ist: Alle Regeln $l = r$ und $l' = r'$ sind

- **nicht überlappend**: $\sigma(l) \neq \sigma(l')$ $\forall$ Substitutionen σ, oder
- **trivial überlappend**: falls σ existiert mit $\sigma(l) = \sigma(l')$, dann ist auch $\sigma(r) = \sigma(r')$.

Spielt damit die Strategie keine Rolle? Doch, denn die LO-Strategie kann Werte berechnen für Ausdrücke, bei denen LI nicht terminiert.
Beispiel:

```
f x = f (x+1)
p x = True
```

LI: `p (f 0)` $\to$ `p (f (0+1))` $\to$ `p (f 1)` $\to$ `p (f (1+1))` $\to$ `p (f 2)` $\to$ `p (f (2+1))` ...
LO: `p (f 0)` $\to$ True

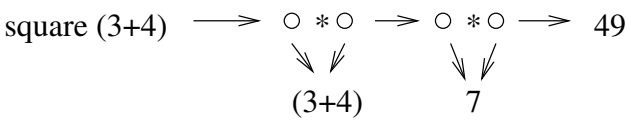
Ist LO damit grundsätzlich besser als LI? Leider nicht, denn:

1. LO ist aufwändiger zu implementieren, da aktuelle Parameter unausgewerte Ausdrücke sein können (vgl. Diskussion zu call-by-name in Kapitel 3.4).
   Beachte: LI $\approx$ call-by-value, LO $\approx$ call-by-name

2. LO kann Teilausdrücke mehrfach auswerten (z.B. "`3+4`" in obiger Ableitung)

Den letzten Nachteil kann man verbessern durch die „faule / verzögerte Auswertung (lazy evaluation, call by need). Die **faule Auswertung** basiert auf der folgenden Idee:

1. Berechne einen Teilausdruck erst dann, wenn er benötigt wird (z.B. von primitive Funktionen, if-then-else, case, ...)
2. Berechne Teilausdruck nicht komplett, sondern nur bis zu einer **Kopfnormalform**: dies ist ein Ausdruck, bei dem an der Wurzel keine definierte Funktion sondern ein Konstruktor steht, z.B. "`0 : (conc [1] [2])`" ("`:`" ist hier die Wurzel).
3. Berechne jeden Ausdruck **höchstens einmal**. Implementierung durch **Graphdarstellung** der Ausdrücke (→ Graphreduktionsmaschinen).



Vorteile der faulen Auswertung:

- Vermeidung überflüssiger Auswertungen
  (→ optimale Reduktion, [5])
- Rechnen mit unendlichen Datenstrukturen
  (→ Modularisierung: Trennung von Daten und Kontrolle)

Beispiel: Liste aller natürlichen Zahlen: `(from 0)` wobei

```
from n = n : from (n+1)
```

Erste n Elemente einer Liste l (Kontrolle): `(take` n l`)`, wobei

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

Modulare Kombination: `take 2 (from 0)` $\rightsquigarrow$ `[0,1]`

Schauen wir uns beim letzten Beispiel die faule Auswertung etwas genauer an:

```
take 2 (from 0)
→ take 2 (0 : from (0+1))
→ 0 : take (2-1) (from (0+1))
→ 0 : take 1 ((0+1): from ((0+1)+1)
→ 0 : (0+1) : take (1-1) (from ((0+1)+1))
→ 0 : (0+1) : []
→ 0 : 1 : []
```

Effekt: obwohl der „Erzeuger“ `from` und der „Verbraucher“ `take` unabhängig implementiert sind (Modularität!), laufen diese miteinander verschränkt ab.

Beispiel: Liste aller Fibonacci-Zahlen: $n_0 = 0 \quad n_1 = 1 \quad n_k = n_{k-1} + n_{k-2}\ (k > 1)$
Idee: Listenerzeuger hat zwei Parameter, die beiden vorigen Fibonacci-Zahlen:

```
fibgen n1 n2 = n1 fibgen n2 (n1 + n2)

fibs = fibgen 0 1  -- 0:1:1:2:3:5:8:13:...

take 6 fibs  ⇝ [0,1,1,2,3,5]
```

Formalisierung der faulen Auswertung

(nach Launchbury, POPL '93 [6])

Der bisherige Formalismus der Termersetzung ist für die Darstellung der faulen Auswertung unzureichend, da die „höchstens einmalige“ Auswertung eine graphähnliche Darstellung verlangt. Aus diesem Grund stellen wir nun Terme durch Zuordnung einer Variablen für jeden Teilausdruck dar, z.B. wird

$$\texttt{f(}\underbrace{\texttt{g}}_{x}\ \underbrace{\texttt{1}}_{y}\texttt{)}\ \underbrace{\texttt{2}}_{z} \qquad \text{umgeformt in} \qquad \texttt{y = 1, z = 2, x = g y, f x z}$$

Dies Darstellung ist ausdrückbar mittels `let`-Ausdrücken, was ein Standardkonstrukt in funktionalen Sprachen ist:

`let` *decls* `in` *exp*

Beispiel:

```
f x y = let a = 2+x
            b = x*y
        in a*y + a*b}
```

Beachte: Hier wurde die **Layout-Regel** zur einfacheren Darstellung lokaler Deklarationen verwendet. Die Layout-Regel basiert auf folgenden Konventionen:

- alle lokalen Deklarationen nach `let` beginnen in der gleichen Spalte
- falls eine neue Zeile rechts davon beginnt, handelt es sich um Fortsetzung der vorigen Zeile

Der Vorteil der Layout-Regel ist, dass Trennsymbole wie “`;`” überflüssig sind und durch ein lesbares Design ersetzt werden.

Mittels `let` können alle Ausdrücke in eine „flache Form“ transformiert werden:

e^* ist die **flache Form** von e:

- $x^* = x \qquad \forall\, Variablen\, x$
- $c^* = c \qquad \forall\, Konstanten\, c$
- $(f\ e_1 \ldots e_n)^* = \texttt{let}\ \ x_1 = e_1^*$
  $\vdots$
  $x_n = e_n^*$
  $\texttt{in}\ f\ x_1 \ldots x_n$

  wobei $x_1, \ldots, x_n$ neue Variablennamen und f eine definierte Funktion oder ein Konstruktor ist.

Optimierung der Übersetzung:

1. In $(f\ e_1 \ldots e_n)^*$ muss man nur neue Variablen für nicht-variable Argumente einführen.
2. Transformiere geschachtelte `let`s:
   `let x = (let y=e in e') in ...` $\Rightarrow$

```
let y=e
    x=e' in ...
```

Somit:

```
(f (g 1) y 2)* = let  x1 = 1
                      x2 = g x1
                      x3 = 2
                 in f x2 y x3
```

Bei der flachen Form sind also alle Funktions- und Konstruktorargumente Variablen. Diese Variablen werden als Speicherzellen oder Graphknoten betrachtet:

Speicher M: Variablen $\rightarrow$ Ausdrücke

Annahme: jede Funktion f ist durch eine einzige Gleichung

$$f\ x_1 \ldots x_n = e \qquad \text{mit } e^* = e$$

definiert, d.h alle Muster sind übersetzt in `if-then-else` bzw. `case` und der Rumpf ist in flacher Form.

Unter dieser Annahme ist das operationale Modell der faulen Auswertung leicht definierbar durch Inferenzregeln im Stil der natürlichen Semantik:

Die Basissprache ist hier:

$$M : e \Downarrow M' : e'$$

„Im Speicherzustand M wird der Ausdruck e reduziert zu e', wobei der Speicher zu M' modifiziert wird."

Wichtig: alle Ausdrücke sind in flacher Form. Unter diesen Annahmen können wir die faule Auswertung durch das folgende Regelsystem definieren:

Konstante / Konstruktor:

$$M : C\ x_1 \ldots x_n \Downarrow M : C\ x_1 \ldots x_n$$

mit C Konstruktor, $n \geq 0$, d.h. der Ausdruck ist in Kopfnormalform.

Primitive Funktionen ($+$,$-$,$*$,/,...):

$$\frac{M : e_1 \Downarrow M' : v_1 \quad M' : e_2 \Downarrow M'' : v_2}{M : e_1 \oplus e_2 \Downarrow M'' : v_1 \oplus v_2}$$

Funktionsanwendung:

$$\frac{M : \sigma(e) \Downarrow M' : e'}{M : f\ y_1 \ldots y_n \Downarrow M' : e'}$$

falls $f\ x_1 \ldots x_n = e$ Regel für f und $\sigma = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$.

`case`:

$$\frac{M : e \Downarrow M' : C\ y_1 \ldots y_k \quad M' : \sigma(e') \Downarrow M'' : e''}{M : \texttt{case}\ e\ \texttt{of}\ \ \ldots C\ x_1 \ldots x_k \to e' \ldots \Downarrow M'' : e''}$$

mit $\sigma = \{x_1 \mapsto y_1, \ldots, x_k \mapsto y_k\}$.

Variable:

$$\frac{M : e \Downarrow M' : e'}{M : x \Downarrow M'[x/e'] : e'}$$

falls $M(x) = e$.

Let:

$$\frac{M[y_1/\sigma(e_1)] \ldots [y_n/\sigma(e_n)] : \sigma(e) \Downarrow M' : e'}{M : \texttt{let}\ \ x_1 = e_1 \ldots x_n = e_n\ \texttt{in}\ \ e \Downarrow M' : e'}$$

mit $\sigma = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$, wobei $y_1, \ldots, y_n$ neue („frische“) Variablen sind.

Anmerkungen:

- Die erste Regel spezifiziert, dass Ausdrücke nur bis zur Kopfnormalform ausgewertet werden.
- `case` + Funktionsanwendung: nur Umsetzen der Variablen ($\approx$ Referenzen)
- Variable: hier wird wirklich der Speicher modifiziert (ersetze Ausdruck durch sein Ergebnis) $\Rightarrow$ faule Auswertung!
- `let`-Regel entspricht dem Speichern der lokalen Ausdrücke

Die Regel für Variablen kann verbessert werden, um manchmal unendliche Schleifen zu erkennen:

Beispiel: Die Auswertung von "`let x=1+x in x`" führt zu einer Endlosableitung (beziehungsweise: es existiert keine endliche Ableitung bezüglich unseres Inferenzsystems).

Hierzu modifizieren die Variablenregel und führen wir eine neue Regel ein:

Variable':

$$\frac{M : e \Downarrow M' : e'}{M[x/e] : x \Downarrow M'[x/e'] : e'}$$

Loop:

$$M : x \Downarrow \langle \text{Fehlschlag: Schleife} \rangle$$

falls $M(x)$ undefiniert ist.
Dann liefert die Auswertung des obigen Ausdrucks einen Fehlschlag! In Haskell wird dies auch als **black hole** bezeichnet.

5.3 Funktionen höherer Ordnung

Beispiel: Sortieren einer Zahlenliste mittels „Insertion Sort" (Sortieren durch Einfügen)
Sortiertes Einfügen eines Elementes in einer Zahlenliste:

```
insert x []     = [x]
insert x (y:ys) = if x<y then x : y : ys
                         else y : insert x ys
```

Sortieren durch Einfügen:

```
isort []     = []
isort (x:xs) = insert x (isort xs)

> isort [3,1,2]  ⇝ [1,2,3]
```

Nachteil dieser Lösung: sie ist zu speziell:

- nur Sortieren von Zahlenlisten
- nur aufsteigendes Sortieren

Absteigendes Sortieren: Kopiere Programmcode und ersetze "`x<y`" durch "`x>y`"

- gut bei Bezahlung nach "lines of code"
- schlecht aus SW-technischer Sicht

Lösung: Parametrisiere `isort` über ein Vergleichsprädikat $\rightsquigarrow$

Eine **Funktion höherer Ordnung** ist eine Funktion, die eine andere Funktion als Parameter oder Ergebnis hat.

In unserem Beispiel könnte `isort` auch eine Funktion höherer Ordnung sein, die zwei Parameter hat: ein Vergleichsprädikat `p` und eine zu sortierende Liste `xs`:

```
isort p []     = []
isort p (x:xs) = insert x (isort p xs)
  where insert x []     = [x]
        insert x (y:ys) = if p x y then x : y : ys
                                   else y : insert x ys
```

Nun können wir `isort` flexibel einsetzen:

```
isort (<) [3,1,2]  ⇝ [1,2,3]
isort (>) [3,1,2]  ⇝ [3,2,1]
```

Weitere nützliche Funktionen höherer Ordnung:

- Transformation einer Liste durch Anwendung einer Funktion auf jedes Listenelement:

  ```
  map f []     = []
  map f (x:xs) = f x : map f xs
  ```

  Anwendung:

  ```
  map square [1,2,3]  ⇝ [1,4,9]
  map inc    [1,2,3]  ⇝ [2,3,4]
  ```

  (mit `inc x = x+1`)

  Falls `inc` nur einmal benutzt wird, ist es eigentlich überflüssig, diese Funktion explizit zu definieren. Daher kann man alternativ auch **anonyme Funktionen** bzw. **λ-Abstraktionen** benutzen:

  ```
  map (\x  → x+1) [1,2,3]}  ⇝ [2,3,4]
  ```

  Hierbei bezeichnet "`\x -> x+1`" eine Funktion mit `x` als Argument und Ergebnis `x+1`

  Daher ist

  ```
  inc x = x+1
  ```

  äquivalent zu

```
inc = \x  → x+1
```

Weitere Alternative:

```
inc = (+) 1
```

- "`(+)`" ist eine Funktion, die zwei Argumente erwartet (die Klammern sind notwendig, da "`+`" ein Infixoperator ist)
- "`(+) 1`" ist eine Funktion, die noch ein Argument erwartet (**partielle Applikation**)

Voraussetzung hierfür ist, dass "`(+)`" nicht ein Paar von Zahlen erwartet, sondern erst eine Zahl und danach die andere Zahl. Beachte hierzu den Unterschied in den Typen:

```
plus' (x,y) = x+y
```

Hier hat `plus'` den Typ `plus' :: (Int,Int) -> Int`

Wenn wir dagegen die „Curry-Schreibweise" verwenden:

```
plus x y = x+y
```

Hier hat `plus` den Typ `plus :: Int -> (Int -> Int)`

Beachte: "`f x`" steht für die Anwendung von `f` auf `x`. Die Applikation ist dabei immer linksassoziativ:

```
f x y  ≈ (f x) y
```

Somit:

- `plus` nimmt ein Argument (eine Zahl) und liefert eine Funktion, die ein weiteres Argument nimmt und dann die Summe liefert.
- "`(+) 1`" ist eine Funktion, die ein Argument nimmt und dazu 1 addiert.

Die Curry-Schreibweise ist zunächst ungewohnt, da in vielen Programmiersprachen die Tupel-Schreibweise üblich ist. Sie ermöglicht aber durch partielle Applikation universeller einsetzbare Programme.

- Filtern bestimmter Elemente einer Liste:

```
filter p []     = []
filter p (x:xs) | p x = x:filter p xs
                | otherwise = filter p xs
```

Mit dieser Definition hat `filter` den Typ

```
filter :: (a → Bool) → [a] → [a]
```

(hierbei ist `a` eine Typvariable die für einen beliebigen Typ, vgl. Kapitel 5.4)

```
filter (\x → x<3) [1,3,4,2]  ⇝ [1,2]
```

Es gibt eine spezielle Syntax für partielle Applikationen bei Infixoperation:

```
(3<)   ≈ (\x → 3<x)
(<3)   ≈ (\x → x<3)
```

Obiges Beispiel:

```
filter (<3) [1,3,4,2]  ⇝ [1,2]
```

Anwendung: Quicksort:

```
qsort []     = []
qsort (x:xs) = qsort (filter (<x) xs) ++ x : qsort (filter (>=x) xs)
```

- Akkumulation von Listenelementen:

  `foldr` $\oplus$ z $[x_1,\ldots,x_n]$ $\rightsquigarrow$ $x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus z)\ldots))$

  Hierbei ist $\oplus$ eine binäre Verknüpfung und z kann als „neutrales“ Element (oder auch Startwert) der Verknüpfung aufgefasst werden.

  ```
  foldr :: (a → b → b) → b → [a] → b
  foldr f z []     = z
  foldr f z (x:xs) = f x (foldr f z xs)
  ```

  Damit kann man die folgenden Funktionen einfach definieren:

  `sum = foldr (+) 0` (Summe aller Listenelemente)

  `prod = foldr (*) 1` (Produkt aller Listenelemente)

  `concat = foldr (++) []` (Konkatenation einer Liste von Listen, `++` ist die Listenkonkatenation)

  Produkt der ersten 10 positiven Zahlen: “`prod (take 10 (from 1))`”

Kombinator-Stil: Programmierung durch Kombination von Basisfunktionen (Programmiersprachen basierend hierauf: APL, FP)

Kontrollstrukturen als Funktionen höherer Ordnung

Eine while-Schleife zur Veränderung eines Wertes besteht aus den folgenden Komponenten:

- Bedingung (Prädikat für den Wert)
- Transformation bisheriger Wert $\mapsto$ neuer Wert
- Anfangswert

Beispiel: kleinste 2-Potenz größer als 10000:

```
x:=1;              -- Anfangswert
while x <= 10000   -- Bedingung
do x:=2*x          -- Transformation
```

Die Kontrollstruktur der while-Schleife können wir auch als Funktion höherer Ordnung definieren:

```
while :: (a  → Bool)  → (a  → a)  → a  → a
while p f x | p x       = while p f (f x)
            | otherwise = x
```

Obiges Beispiel: "`while (<=10000) (2*) 1`" $\rightsquigarrow$ `16384`

Anmerkungen

- neue Kontrollstrukturen können ohne Spracherweiterung eingeführt werden (z.B. `until` ist analog in Haskell vordefiniert)
- einfache Basissprache
- flexible anwendungsorientierte Erweiterungen

5.4 Typsysteme

Betrachten wir noch einmal einige Begriffe aus Kapitel 2.3:

Statisch getypte Programmiersprache (manchmal auch streng getypte Programmiersprache): Jeder Bezeichner hat einen Typ, und dieser liegt zur Compilezeit fest.

Die **Semantik eines Typs** ist dabei die Menge möglicher Werte.

Beispiel:
`Int`: (endliche Teil-)Menge der ganzen Zahlen
`Bool`: {`True`, `False`}
`Int -> Bool`: Menge aller Funktionen, die ganze Zahlen auf Wahrheitswerte abbilden.

Typkorrektheit: Ein Ausdruck e ist **typkorrekt** / **wohlgetypt**, wenn

- e den Typ τ hat und
- falls die Auswertung von e den Wert v ergibt, dann gehört v zum Typ τ

Beispiel:
`3+6*7` hat den Typ `Int`

`3+False` ist nicht typkorrekt (Auswertung liefert „error“ und dies gehört zu keinem Typ)

`map (+1) [1,3,'a',5]` ist nicht typkorrekt

`map (+1) [1,3,4,5]` hat den Typ `[Int]`

Ein Programm heißt typkorrekt, wenn alle Ausdrücke, Funktionen, etc. typkorrekt sind.

Bei einer statisch getypte Programmiersprache gilt (sollte gelten!):
zulässige Programme sind typkorrekt, d.h. es gibt keine Typfehler zur Laufzeit
([9]: “well-typed programs do not go wrong”)

Vorteile statisch getypter Programmiersprachen:

- Programmiersicherheit
- Produktivitätsgewinn: Compiler meldet Typfehler (potentielle Laufzeitfehler)
- Laufzeiteffizienz: keine Typprüfung zur Laufzeit
- Dokumentation: Typen $\approx$ partielle Spezifikation

Nachteile:

- Typkorrektheit ist i.allg. unentscheidbar $\Rightarrow$ Einschränkung der zulässigen Programme
- Beispiel: “`map (+1) (take 2 [1,2,True])`” ist typkorrekt, aber i.a. unzulässig in einer statisch getypten Programmiersprache

Da aber die Vorteile die Nachteile überwiegen, sind viele moderne Programmiersprachen statisch getypt.

Kriterien für gute Typsysteme:

- Sicherheit (wohlgetypt $\Rightarrow$ keine Laufzeitfehler)
- Flexibilität (möglichst wenig Einschränkungen)
- Benutzung (Typangaben weglassen, falls diese nicht notwendig sind)

Am besten wird dies erreicht in funktionalen Programmiersprachen durch

- Polymorphismus ($\rightsquigarrow$ Flexibilität)
- Typinferenz ($\rightsquigarrow$ Benutzung)

Polymorphismus bedeutet, dass Objekte (Funktionen) mehrere Typen haben können.

Hierbei unterscheiden wir zwei Arten von Polymorphismus:

Ad-hoc-Polymorphismus: Unterschiedliches Verhalten der Objekte auf unterschiedlichen Typen (üblich: Overloading)
Beispiel: "`+`" in Java:
`1+2`: Addition auf ganzen Zahlen
`"ab"+"cd"`: Konkatenation von Strings

Parametrischer Polymorphismus: Gleiches Verhalten auf allen Typen. Typischerweise enthalten Typen hierbei Typparameter (vgl. Kapitel 2.2.4: parametrisierte ADTs)
Beispiel: Länge einer Liste:

```
length []     = 0
length (x:xs) = 1 + length xs
```

Typ:

```
length :: [a]  → Int
```

wobei `a` ein Typparameter ist, der durch einen beliebigen anderen Typ ersetzt werden kann
⇒ `length` ist anwendbar auf `[Int]`, `[(Float,Bool)]`,...
⇒ "`length [0,1] + length [True,False]`" ist typkorrekt.

Parametrischer Polymorphismus ist

- nicht vorhanden in vielen imperativen PS (wie Pascal, Modula, Java, ...)
- besonders wichtig bei Funktionen höherer Ordnung (vgl. die Typen von `filter`, `foldr`)

Typinferenz bezeichnet die Herleitung von (möglichst allgemeinen) Typen für Objekte, so dass das Programm typkorrekt ist. Dies beinhaltet:

- Herleitung der Parametertypen:
  Falls "`f :: Int -> Bool -> Int`" und `f x y = ...`, dann hat `x` den Typ `Int` und `y` hat den Typ Bool.
- Herleitung von Funktionstypen: nur Regeln für `length` hinschreiben ⇒ automatische Typherleitung
  (dies ist besonders angenehm und sinnvoll bei lokalen Deklarationen)

Ein **Typsystem**

- definiert die Sprache der Typen
- definiert „wohlgetypt" (was im allgemeinen eine konstruktive Einschränkung von „typkorrekt" ist)

- legt evtl. auch Typinferenz fest (dann gibt es oft weitere Einschränkungen)

Im folgenden:

- Typsystem à la Hindley/Milner [3]
  (beachte: Haskell hat ein komplexeres Typsystem mit Typklassen)
- wir definieren nur wohlgetypt, nicht aber die Typinferenz
  (dies wird z.B. in der Vorlesung über Deklarative Programmierung behandelt)

Wir definieren zunächst die **Sprache der parametrisierten Typen** (**Typausdrücke**)

`Type ::= TVar` (Typvariablen, Haskell: Bezeichner beginnend mit Kleinbuchstaben)
`| Basetype`
`| Typecon { Type }`

`Basetype ::= Bool | Int | Float | Char` usw.

`Typecon ::=` Typkonstruktoren aus `data`-Deklarationen, z.B.:

- `Tree, BTree`
- `List` (Haskell: $[\tau]$ statt "`List` τ")
- `Tuple` (Haskell: (τ_1, τ_2) statt "`Tuple` τ_1 τ_2")
- `Func` (Haskell: τ_1 `->` τ_2 statt "`Func` τ_1 τ_2")

Parametrischer Polymorphismus: lasse Substitutionen auf Typvariablen zu („**Typsubstitution**“ analog zu Substitutionen auf Ausdrücken, vgl. Kapitel 5.2)

```
length :: [a]  →  Int
```

hat auch die Typen:

```
[Int]  →  Int              Typsubstitution: {a ↦ Int}
[(Char,Float)]  →  Int     Typsubstitution: {a ↦ (Char,Float)}
```

Damit die Eigenschaft „wohlgetypt“ entscheidbar ist, sind Einschränkungen notwendig. Hier ist die wesentliche Einschränkung, dass die Bildung von Typinstanzen (d.h. Anwendung von Typsubstitutionen) innerhalb von Funktionsregeln bei Typen von Funktionsparametern *nicht* erlaubt ist.
Beispiel:

```
f :: (a  →  a)  →  (a  →  a)
f g = g g
```

Diese Funktion ist prinzipiell typkorrekt, falls im Rumpf "`g g`"

- das erste `g` den Typ "`(a->a) -> (a->a)`" hat (d.h. die Typsubstitution {`a` ↦ `(a->a)`} angewendet wird)

- das zweite `g` den Typ "`(a->a)`" hat.

Hierzu müsste also der Typ des Parameters `g` innerhalb der Regel auf verschiedene Arten substituiert werden. Dies ist allerdings aus Entscheidbarkeitsgründen nicht zugelassen, d.h. in Haskell ist diese Funktion **nicht wohlgetypt.**

Um dies genau zu beschreiben, erweitern wir die Typsprache um **Typschemata**:

`Typescheme ::= ∀{TVar}.Type`

(beachte: das triviale Typschema $\forall.\tau$ entspricht dem Typ τ)

Beispiel: `∀a.[a] -> Int` ist ein Typschema für die Funktion `length`

Beachte: In Haskell stehen Funktionstypen

`f :: τ`

immer für das Typschema

`f :: ` $\forall a_1, \ldots, a_n.\tau$

wobei $a_1, \ldots, a_n$ alle Typvariablen aus τ sind.

Wichtig ist nun, dass von Typschemata Instanzen gebildet werden können:

τ ist eine **generische Instanz** des Typschemas $\forall a_1, \ldots, a_n.\tau'$ $(n \geq 0)$, falls eine Typsubstitution $\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$ existiert mit $\tau = \sigma(\tau')$.

Nun ist die Wohlgetyptheit mittels eines Inferenzsystems einfach definierbar. Hierzu betrachten wir eine **Typannahme** als Zuordnung A von Namen zu Typschemata. Das Raten einer korrekten Typannahme ist gerade die Aufgabe der Typinferenz, die wir hier nicht weiter betrachten.

Inferenzsystem zur **Wohlgetyptheit von Ausdrücken**:

Axiom:

$$\frac{}{A \vdash x :: \tau}$$

falls τ eine generische Instanz von $A(x)$ ist (dies drückt genau den parametrischen Polymorphismus aus!).

Applikation:

$$\frac{A \vdash e_1 :: \tau_1 \to \tau_2 \quad A \vdash e_2 :: \tau_1}{A \vdash e_1\ e_2 :: \tau_2}$$

Abstraktion:

$$\frac{A[x/\tau] \vdash e :: \tau'}{A \vdash \backslash x \to e :: \tau \to \tau'}$$

wobei τ ein Typausdruck ist.

Bedingung:

$$\frac{A \vdash e_1 :: \mathit{Bool} \quad A \vdash e_2 :: \tau \quad A \vdash e_3 :: \tau}{A \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: \tau}$$

Let-Deklaration:

$$\frac{A[x/\tau] \vdash e_1 :: \tau \quad A[x/\sigma] \vdash e_2 :: \tau'}{A \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 :: \tau'}$$

wobei τ ein Typausdruck ist, $\sigma = \forall a_1, \ldots, a_n.\tau$, a_i kommt in τ aber nicht in A vor (dies spezifiziert die polymorphe Verwendung lokaler Funktionen).

Eine Gleichung $f\ t_1 \ldots t_n = e$ ist wohlgetypt bzgl. A:

1. $A(f) = \forall a_1 \ldots a_n.\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau$
2. Es existieren Typausdrücke $\tau'_1, \ldots, \tau'_k$ für die Variablen $x_1, \ldots, x_k$ in $t_1, \ldots, t_n$ mit: Sei $A' = A[x_1/\tau'_1, \ldots, x_k/\tau'_k]$. Dann sind $A' \vdash t_i :: \tau_i$ $(i = 1, \ldots, n)$ und $A' \vdash e :: \tau$ ableitbar.

Gleichungen mit Bedingungen, d.h. $l \mid b = r$ können wir hier auffassen als

l `= if` b `then` r `else` $\bot$

mit $A(\bot) = \forall a.a$

Ein Programm ist wohlgetypt falls alle Gleichungen wohlgetypt bzgl. einer festen Typannahme A sind.
Die Konsequenz von Punkt 2 ist, dass, wie schon oben erwähnt, bestimmte wohlgetypte Ausdrücke als nicht typisierbar betrachtet werden:

```
funsum f l1 l2 = f l1 + f l2
```

Dann ist der Ausdruck `(funsum length [1,2] "ab")` nicht wohlgetypt, obwohl dieser typkorrekt sein könnte, falls

`funsum ::` $\forall a, b.(\forall c.c \to$ `Int`$) \to a \to b \to$ `Int`

aber Typschemata sind als Typen von Parametern nicht zulässig.

Beispiel:

```
twice f x = f (f x)
```

ist wohlgetypt mit der Typannahme $A($`twice`$) = \forall a.(a \to a) \to a \to a$ und geeigneten Annahmen für `f` und `x` ($\to$ Übung).

Die Aufgabe der **Typinferenz** ist es, eine geeignete Typannahme zu raten bzw. konstruktiv zu finden. Dies ist

- unproblematisch für Parameter
- schwieriger für Typschemata (Typen der Funktionen), daher fordert man dann weitere Einschränkungen, die allerdings nur bei komplexen Beispielen relevant werden.

Beispiel:

```
f :: [a]  →  [a]
f x = if length x == 0 then fst (g x x) else x

g :: [a]  →  [b]  →  ([a],[b])
g x y = (f x, f y)

h = g [3,4] [True, False]

fst (x,y) = x   -- first
```

- Mit Typangaben für `f` und `g` ist das Beispiel wohlgetypt.
- Ohne Typangaben für `f` und `g` ist das Beispiel nicht wohlgetypt, da der Typ `g :: [a] -> [a] -> ([a][a])` hergeleitet wird.

Prinzip bei der Typinferenz: innerhalb von rekursiven Aufrufen / Definitionen haben Funktionen kein Typschema, sondern einen Typ. (Andernfalls wäre die Typinferenz unentscheidbar.)

5.5 Funktionale Konstrukte in imperativen Sprachen

Charakteristika funktionaler PS:

- referentielle Transparenz
- algebraische Datentypen und Pattern Matching
- Funktionen höherer Ordnung
- parametrischer Polymorphismus
- lazy-Auswertung

Der erste und letzte Punkt ist schwer mit imperativen Programmiersprachen zu kombinieren, da dies ein anderes Auswertungsprinzip erfordert. Die anderen Aspekte sind oder können teilweise in imperativen Programmiersprachen integriert werden. Z.B. wurde schon früh mit der Sprache Pizza[1] [10] ein Ansatz vorgestellt, wie diese Aspekte in Java integriert werden können.

Der **parametrische Polymorphismus** wurde als Generizität in Java 5 eingeführt (vgl. Kapitel 4.5).

Funktionen höherer Ordnung existieren (eingeschränkt) in vielen imperativen Programmiersprachen, z.B. in Pascal (typunsicher, da keine Parametertypen angegeben werden), C (als Funktionszeiger).

[1] `http://pizzacompiler.sourceforge.net/`

Die Programmiersprache Scala[2] ist ein Ansatz, Eigenschaften objektorientierter und funktionaler Sprachen zu integrieren. Scala unterstützt Funktionen höherer Ordnung, Pattern Matching und auch (eingeschränkt) lazy Auswertung, allerdings nicht in der reinen Form funktionaler Sprachen. Somit können wir festhalten:

- Prinzipiell ist die Integration einiger Aspekte funktionaler Programmierung auch in imperative Sprachen möglich.
- Die Vorteile funktionaler Programmiersprachen (Verständlichkeit durch referentielle Transparenz und musterorientierte Regeln) ist nur wirklich ausnutzbar in rein funktionalen Programmiersprachen.

[2] `http://www.scala-lang.org`

6 Logische Programmiersprachen

„Rechnen“ in verschiedenen Programmiersprachen:

- Imperative Programmiersprachen: Folge von Variablenmanipulationen (Zuweisungen)
- Funktionale Programmiersprachen: Folge von Ersetzungen (Reduktion zur Normalform)
- Logische Programmiersprachen: Folge von Beweisschritten (in einem logischen Kalkül) („Ist eine Aussage wahr bzgl. der eingegebenen Regeln?“)

Im folgenden werden wir alle Beispiele in der Programmiersprache **Prolog** angeben. Prolog ist zwar schon eine recht alte Sprache (die Ursprünge gehen zurück auf das Jahr 1972), aber es ist ein Standard in der logischen Programmierung und als ISO-Standard auch industriell relevant.
Die Notation ist im Gegensatz zu Haskell mehr logikorientiert, d.h. statt $(f\ e_1 \dots e_n)$ schreibt man in Prolog $f(e_1, \dots, e_n)$.

6.1 Einführung

Zunächst geben wir eine kurze Einführung in die Struktur logischer Programme.

Ein **Programm** ist Menge von Prädikaten (Relationen).

Ein **Prädikat** ist definiert durch Fakten und Implikationen / Regeln. Daher nennt man dies manchmal auch regelorientierte Programmierung, die für Anwendungen in der KI (Künstliche Intelligenz) wichtig ist.

Ein **Literal** ist die Anwendung eines Prädikats auf Argumente (Terme). Intuitiv entspricht dies einer Aussage.

Beispiel: Wir wollen Familienbeziehungen als logisches Programm modellieren. Die betrachteten Prädikate (Aussagen) sind dabei:

Vater: `vater(v,k)` :⇔ `v` ist Vater von `k`
Großvater: `grossvater(g,e)` :⇔ `g` ist Großvater von `e`

Die Definitionen hierfür bilden das Prolog-Programm:

```
vater(fritz,thomas).  % Fritz ist Vater von Thomas.
vater(thomas,maria).
vater(thomas,anna).
```

```
grossvater(G,E) :- vater(G,V), vater(V,E).
% G Grossvater von E, falls G Vater von V und V Vater von E ist.
```

Einige Erläuterungen zur Syntax von Prolog:

- Kommentare: beginnen mit `%` und erstrecken sich bis zum Zeilenende
- Variablennamen: beginnen mit einem Großbuchstaben (`G,V,E`)
- Literal: `<Prädikatname>(<arg1>,...,<argn>)`
  Beachte: Zwischen dem Prädikatnamen und der danach öffnenden Klammer darf *kein* Leerzeichen stehen!
- Faktum: `<Literal>.`
  Ein Literal ist eine immer wahre Aussage. Es muss mit einem Punkt am Ende abgeschlossen werden.
- Implikation: `<Lit> :- <Lit1>,...,<Litn>.`
  Das Symbol "`:-`" kann man als Implikation "$\Leftarrow$" lesen, und jedes Komma entsprecht einer Konjunktion "$\wedge$". Wichtig ist auch hier, dass die Regel mit einem Punkt am Ende abgeschlossen wird. Intuitiv bedeutet diese Regel: „Falls `<Lit1>` und ... und `<Litn>` wahr sind, dann ist auch `<Lit>` wahr."
- Die Argumente von Literalen sind **Terme** (dies entspricht Ausdrücken oder Datentermen in funktionalen Sprachen ohne definierte Funktionen). Diese sind zusammengesetzt aus Variablen, Konstanten, Zahlen, **Funktoren** ($\approx$ Datenkonstruktoren aus funktionalen Sprachen), wie z.B. `datum(1,januar,J)` (hier ist "`datum`" ein Funktor).
- Listen sind wie in Haskell definiert, allerdings ist die Notation etwas anders: in Prolog schreibt man `[H|T]` statt `(h:t)` bzw. `[E1,E2,E3|T]` statt `(e1:e2:e3:t)`
- Eine **Klausel** ist ein Faktum oder eine Regel / Implikation.

Logikprogramme entsprechen logischen Formeln, z.B. entspricht die Regel

```
grossvater(G,E) :- vater(G,V), vater(V,E)
```

der logischen Formel

$$\forall G \forall E \forall V \ (grossvater(G,E) \leftarrow vater(G,V) \wedge vater(V,E))$$

Eine **Anfrage** ist eine Konjunktion von Literalen. Intuitiv bedeutet eine Anfrage, dass wir daran interessiert sind, ob diese Literale logisch aus dem Programm folgen. In Anfrage sind auch Variablen erlaubt. In diesem Fall sollen dann Werte berechnet/erraten werden, so dass für diese Werte die Anfrage logisch aus dem Programm folgt.

Beispiel: Gegeben sei unser obiges Prolog-Programm. Wenn wir dies in das Prolog-System geladen haben, können wir folgende Anfragen stellen:

```
?- vater(fritz,thomas).
yes
?- vater(fritz,anna).
no
?- grossvater(fritz,anna).
yes
?- grossvater(fritz,E). % Welche Enkel E hat Fritz?
E=maria  ;  % Eingabe von ";" entspricht der Suche nach weiteren Loesungen
E=anna
```

Wie wir sehen, sucht das Prolog-System nach passenden Lösungen und es kann durchaus auch mehrere Lösungen geben.
Wichtig ist festzuhalten, dass es in Prolog keine festen Ein-/Ausgabeargumente gibt (d.h. Prolog kann bidirektional rechnen), weil man bei jedem Argument eine Variable in der Anfrage einsetzen kann:

```
?- grossvater (G,anna). % Welche Grossvaeter hat Anna?
G=fritz
```

Unterschied zu funktionalen Programmiersprachen:

- Raten von passenden Werten
- (nichtdeterministische) Suche nach Lösungen
- keine festen Ein-/Ausgabeargumente: mit der Definition einer Funktion hat man automatisch auch immer die Umkehrfunktion bzw. -relation zur Verfügung

Analog zur funktionalen Programmierung können wir in Prolog auch musterorieniert Programmieren. Als Beispiel betrachten wir ein Prädikat zur Listenkonkatenation:
`append(L1,L2,L3)` :⇔ `L3` ist Konkatenation von `L1` und `L2`

```
append([],L,L).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Daran können wir ein Schema zur Übersetzung funktionaler Programme in relationale Programme erkennen:

- Füge ein zusätzliches Ergebnisargument hinzu:
  eine n-stellige Funktion wird übersetzt in ein $(n+1)$-stelliges Prädikat
- Herausziehen geschachtelter Funktionsaufrufe:
  „`X=f(`$\underbrace{\texttt{f(Y)}}_{Z}$`)`“ $\rightsquigarrow$ `f(Y,Z), f(Z,X)` (`Z` ist eine neue Variable).
- relationale Version ist flexibler einsetzbar. Beispiele:
  Präfix abspalten:

```
?- append([1,2],X,[1,2,3,4]).
X=[3,4]
```

Letztes Element:

```
?- append(_,[X],[1,2,3,4]).
X=4
```

Hier bezeichnet "_" eine anonyme Variable, an deren Wert man nicht interessiert ist.

6.2 Operationale Semantik

Wir wollen nun die operationale Semantik von Prolog erläutern, d.h. wir werden sehen, wie ein Prolog-System in der Lage ist, die oben gezeigten Schlussfolgerungen zu ziehen. Prolog berechnet Schlussfolgerungen und beweist damit Aussagen mit Hilfe des sogenannten Resolutionsprinzips. Zunächst einmal betrachten wir dieses Prinzip in einer vereinfachten Form:

Vereinfachtes Resolutionsprinzip: Um ein Literal L zu beweisen, suche eine zu L passende Regel L`:-`$L_1, \ldots, L_n$ und beweise $L_1, \ldots, L_n$ (falls $n = 0$, d.h. die Regel ist ein Faktum, dann ist L bewiesen).

Probleme:

- Regel passt evtl. nicht genau ⇒ Unifikation (Beispiel: `vater(fritz,K)`)
- evtl. passen mehrere Regeln ⇒ Nichtdeterminismus, Suche

Unifikation: „Gleichmachen" von Literalen/Termen durch Substitution der Variablen in *beiden* Literalen/Termen.
Dagegen ist das "pattern matching" aus funktionalen Sprachen nur die Anpassung *eines* Terms an einen anderen. Dies ist einer der wesentlichen Unterschiede zwischen funktionaler und logischer Programmierung: Beim Anwenden von Regeln wird in logischen Sprachen Unifikation statt Pattern Matching verwendet. Hierdurch wird das Finden von Lösungen und die bidirektionale Anwendung von Relationen ermöglicht. Wir wollen nun den Begriff des Unifikators formal definieren.

Definition 6.1 (Unifikator) *Ein Unifikator für zwei Terme t_1, t_2 ist eine Substitution σ mit $\sigma(t_1) = \sigma(t_2)$. In diesem Fall heißen t_1 und t_2* ***unifizierbar****.*
Ein ***allgemeinster Unifikator (most general unifier, mgu)*** *für t_1, t_2 ist ein Unifikator σ für t_1, t_2 mit: falls σ' auch ein Unifikator für t_1, t_2 ist, dann exisitiert eine Substitution φ mit $\sigma' = \varphi \circ \sigma$ („mgu subsumiert alle anderen Unifikatoren")*

Beispiel: Seit $t_1 = p(a, Y, Z)$ und $t_2 = p(X, b, T)$
$\sigma_1 = \{X \mapsto a, Y \mapsto b, Z \mapsto c, T \mapsto c\}$ ist ein Unifikator, aber nicht mgu.
$\sigma_2 = \{X \mapsto a, Y \mapsto b, Z \mapsto T\}$ ist ein mgu

Fragen:

- Exisitiert immer ein mgu für unifizierbare Terme?
- Wie können mgu's berechnet werden?

Ein Antwort darauf finden wir in [12]:

> Für unifizierbare Terme existiert immer ein mgu, der effektiv berechenbar ist.

Im folgenden geben wir nicht den Algorithmus von Robinson an, sondern zeigen, wie ein mgu durch Transformation von Termgleichungen berechnet werden kann (nach [8])

Sei E eine Menge von Termgleichungen
(initial ist $E = \{t_1 = t_2\}$, falls wir t_1 und t_2 unifizieren wollen)

Transformation zur mgu-Berechnung (hier steht x für eine Variable):

Eliminate:

$$\{x = x\} \cup E \;\Rightarrow\; E$$

Decompose:

$$\{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\} \cup E \;\Rightarrow\; \{s_1 = t_1, \ldots, s_n = t_n\} \cup E$$

Clash:

$$\{f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)\} \cup E \;\Rightarrow\; fail$$

falls $f \neq g$ oder $n \neq m$.

Swap:

$$\{f(s_1, \ldots, s_n) = x\} \cup E \;\Rightarrow\; \{x = f(s_1, \ldots, s_n)\} \cup E$$

Replace:

$$\{x = t\} \cup E \;\Rightarrow\; \{x = t\} \cup \sigma(E)$$

falls x eine Variable ist, die in t nicht vorkommt, $x \neq t$ und $\sigma = \{x \mapsto t\}$

Occur check:

$$\{x = t\} \cup E \;\Rightarrow\; fail$$

falls x eine Variable ist, die in t vorkommt und $x \neq t$.

Die letzte Regel ist notwendig, um nicht unifizierbare Terme zu entdecken:
Zum Beispiel ist $\{x = f(x)\}$ nicht unifizierbar.
Mit der letzten Regel gilt: $\{x = f(x)\} \Rightarrow fail$

Notation: $E \Rightarrow^* E' \;:\Leftrightarrow\; E \Rightarrow E_1 \Rightarrow E_2 \Rightarrow \cdots \Rightarrow E'$ (transitiver Abschluss)

Satz 6.1 *Falls $\{t_1 = t_2\} \Rightarrow^* \{x_1 = s_1, \ldots, x_n = s_n\}$ gilt und keine weitere Regel anwendbar ist, dann ist $\sigma = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ ein mgu für t_1 und t_2.*
Andernfalls gilt $\{t_1 = t_2\} \Rightarrow^* fail$ und dann sind t_1 und t_2 nicht unifizierbar.

Beispiel: Sei $\{p(a, Y, Z) = p(X, b, X)\}$
Decompose $\Rightarrow \{a = X, Y = b, Z = X\}$
Swap $\Rightarrow \{X = a, Y = b, Z = X\}$
Replace $\Rightarrow \{X = a, Y = b, Z = a\}$
Damit ist $\sigma = \{X \mapsto a, Y \mapsto b, Z \mapsto a\}$ ist mgu.

Damit können wir nun einen allgemeinen Resolutionsschritt definieren:

Definition 6.2 (SLD-Resolutionsschritt) *Sei*

`?-` $L_1, \ldots, L_m$ `.`

die aktuelle Anfrage und

H `:-` $B_1, \ldots, B_n$ `.`

eine Klausel ($n \geq 0$)*, wobei* L_i *und* H *unifizierbar sind mit einem mgu* σ*. Dann heißt die neue Anfrage*

`?-` $\sigma(L_1, \ldots, L_{i-1}, B_1, \ldots, B_n, L_{i+1}, \ldots, L_m)$ `.`

Resolvente *aus der aktuellen Anfrage und der Klausel mit mgu* σ*.*

Anmerkungen:

S: Selektionsregel S wählt ein L_i aus der aktuellen Anfrage aus (in Prolog: $i = 1$, d.h. das linke Literal wird zuerst bewiesen)

L: Lineare Resolution, d.h. verknüpfe immer eine Anfrage mit einer Klausel und erhalte eine neue Anfrage. Es gibt auch eine allgemeine Resolution, bei der auch Programmklauseln verknüpft werden, die auch eine allgemeinere Form haben können.

D: Definite Klauseln, d.h. die linke Seite enthält immer nur eine Literal.

Eine **SLD-Ableitung** ist eine Folge $A_1, A_2, \ldots$ von Anfragen, wobei A_{i+1} eine Resolvente aus A_i und einer **Variante** einer Programmklausel (mit neuen Variablen) ist. Das Betrachten von Varianten ist notwendig, denn sonst wäre bei einem Faktum "`p(X)`" die Anfrage "`?- p(f(X)).`" nicht beweisbar.
Eine SLD-Ableitung ist

- **erfolgreich**, wenn die letzte Anfrage leer ist (ohne Literale): dann ist alles bewiesen und die Substitution der Anfragevariablen ist die berechnete Lösung.
- **fehlgeschlagen**, wenn die letzte Anfrage nicht leer ist und keine weiteren Regeln anwendbar sind.
- **unendlich:**, wenn die SLD-Ableitung eine unendliche Folge ist. Dies entspricht einer Endlosschleife.

Beispiel:

```
?- grossvater(fritz,E).
```
$\sigma_1 = \{\texttt{G} \mapsto \texttt{fritz}\}$
```
?- vater(fritz,V), vater(V,E).
```
$\sigma_2 = \{\texttt{V} \mapsto \texttt{thomas}\}$
```
?- vater (thomas,E).
```
$\sigma_3 = \{\texttt{E} \mapsto \texttt{maria}\}$
```
?- .
```

Damit ist die berechnete Lösung: `E = maria`

Satz 6.2 (Korrektheit und Vollständigkeit der SLD-Resolution) *Sei S eine beliebige Selektionsregel.*

1. ***Korrektheit:*** *Falls eine erfolgreiche SLD-Ableitung eine Substitution σ berechnet $\Rightarrow$ σ ist „logisch korrekte" Antwort.*
2. ***Vollständigkeit:*** *Falls σ eine „logisch korrekte" Antwort für eine Anfrage ist, dann existiert(!) eine erfolgreiche SLD-Ableitung mit einer berechneten Antwort σ' und eine Substitution φ mit $\sigma = \varphi \circ \sigma'$ (d.h. es werden allgemeinere Antworten brechnet)*

Problem: die Vollständigkeit ist nur eine Existenzaussage, aber wie findet man die erfolgreichen SLD-Ableitung unter allen SLD-Ableitungen? Eine Möglichkeit ist, alle möglichen SLD-Ableitungen zu untersuchen, aber in welcher Reihenfolge? Eine faire und sichere Strategie wäre, alle SLD-Ableitungen gleichzeitig zu untersuchen:

- parallel („ODER"-Parallelismus)
- Breitensuche im Baum aller Ableitungen (SLD-Baum)

Dies ist aber aufwändig. Daher wird in Prolog auf diese sichere Strategie verzichtet. Stattdessen wird ein **Backtracking-Verfahren** verwendet (was einer Tiefensuche im SLD-Baum aller Ableitungen entspricht). Informell kann dieses Verfahren wie folgt beschrieben werden:

- Falls mehrere Klauseln anwendbar sind, nehme die textuell erste passende Klausel.
- Falls man in einer Sackgasse landet (wo keine Regel anwendbar ist), gehe zur letzten Alternative zurück und probiere die textuell nächste Klausel.

Ein Problem dieser Strategie ist, dass man eventuell keine Lösung bei existierenden endlosen Ableitungen erhält:

```
p :- p.
p.

?- p.   ⇒   ?- p.   ⇒   ?- p.   ⇒ ···
```

Formalisierung von Prologs Backtracking-Strategie

Wir wollen nun die soeben informell beschriebene Backtracking-Strategie präzise beschreiben. Dazu verwenden wir die folgenden Notationen:

$l_1 \wedge \cdots \wedge l_n$: Anfrage mit Literalen $l_1, \ldots, l_n$

true: leere Anfrage (hierbei entspricht ein Faktum einer Regel der Form `p :-` *true*)

++: Konkatenation auf Listen (wobei die Argumente auch einzelne Elemente statt einelementige Listen sein können)

[ ]: leere Liste

Die Basissprache des Inferenzsystems für die Backtracking-Strategie enthält Aussagen der Form

$$cs, \sigma \vdash g : \overline{\sigma}$$

wobei:
cs ist eine Liste von Klauseln, die noch zum Beweis des ersten Literals von g benutzt werden können
σ ist die bisher berechnete Antwort
g ist die zu beweisende Anfrage
$\overline{\sigma}$ ist die Liste der berechneten Antworten für g

Inferenzregeln (hierbei bezeichnet P das gesamte Programm, also eine Folge von Klauseln):

keine Klausel vorhanden:

$$\frac{}{[\,], \sigma \vdash g : [\,]}$$

Anfrage bewiesen:

$$\frac{}{cs, \sigma \vdash \mathit{true} : [\sigma]}$$

Literal bewiesen:

$$\frac{cs, \sigma \vdash g : \overline{\sigma}}{cs, \sigma \vdash \mathit{true} \wedge g : \overline{\sigma}}$$

Klauselanwendung:

$$\frac{P', \varphi \circ \sigma \vdash \varphi(b \wedge g) : \overline{\sigma_1} \qquad cs, \sigma \vdash l \wedge g : \overline{\sigma_2}}{(a \texttt{ :- } b) +\!+ cs, \sigma \vdash l \wedge g : \overline{\sigma_1} +\!+ \overline{\sigma_2}}$$

wobei φ ein mgu für a und l ist und P' ist eine Variante des Programms P (mit jeweils neuen Variablen).

Klausel nicht anwendbar:

$$\frac{cs, \sigma \vdash l \wedge g : \overline{\sigma}}{(a \texttt{ :- } b) +\!+ cs, \sigma \vdash l \wedge g : \overline{\sigma}}$$

wobei a und l nicht unifizierbar sind.

Die Regel „Klauselanwendung“ formalisiert das „Backtracking“ bei einer Klauselanwendung: es werden die Antworten mit der ersten Klausel berechnet und dahinter dann die Antworten mit den restlichen Klauseln.

Anwendung des Inferenzsystems: $\overline{\sigma}$ ist die Liste der berechneten Antworten für die Anfrage G, falls $p, \{\} \vdash G : \overline{\sigma}$ ableitbar ist.

Beispiel: Das Programm P enthalte folgende Klauseln:

```
p(a) :- true.
p(b) :- true.
q(X) :- p(X).
```

Dann ist $P, \{\} \vdash$ `q(Z)` $: [\{$`Z` $\mapsto$ `a`$\}, \{$`Z` $\mapsto$ `b`$\}]$ ableitbar (wobei wir hier die Substitution der Variablen `X` weggelassen haben).

Problem: Bezüglich dieses Inferenzsystems ist nichts ableitbar bei

- unendlich vielen Antworten
- endlich vielen Antworten gefolgt von unendlicher Berechnung

Mögliche Lösung: erweitere das Inferenzsystem, um nur die maximal ersten n Antworten zu berechnen ($\rightarrow$ Übung).

6.3 Erweiterungen von Prolog

Prolog ist prinzipiell eine berechnungsuniverselle Programmiersprachen. Trotzdem sind verschiedene Erweiterungen möglich, die praktisch auch nützlich sind. Die wichtigsten Erweiterungen wollen wir im folgenden kurz diskutieren.

Negation

Im bisherigen Sprachumfang konnten keine negativen Bedingungen in Regeln formuliert werden, obwohl dies manchmal nützlich wäre, wie das folgende Beispiel zeigt (logische Programme mit Negation werden manchmal auch als **normale Programme** bezeichnet):

Zwei Mengen `X` und `Y` sind verschieden:

```
verschieden(X,Y) :- element(Z,X), ¬element(Z,Y).
```

Problem: wie kann man mit „¬“ beweistechnisch umgehen?

Eine echte prädikatenlogische Negation ist aufwändig, da dann eine *lineare* Resolution nicht mehr möglich wäre. Daher wird die Negation in Prolog als **Negation als Fehlschlag** (**negation as failure**) interpretiert. Dies wird in Prolog als `\+` statt ¬ notiert, d.h.

```
verschieden(X,Y) :- element(Z,X), \+ element(Z,Y).
```

Interpretiert wird "`\+` l" als: falls alle Beweise für l fehlgeschlagen, dann ist "`\+` l" beweisbar.

Inferenzregel für "negation as failure":

$$\frac{P', \sigma \vdash l : [\,]}{cs, \sigma \vdash \texttt{\textbackslash+}\, l : [\sigma]}$$

Hierbei ist P' eine Variante des Programms P mit jeweils neuen Variablen.
Wird die SLD-Resolution um diese Regel erweitert, spricht man auch von **SLDNF-Resolution**.

Bedingungen für die Korrektheit der SLDNF-Resolution findet man z.B. in [7]. Zusammengefasst ist die SLDNF-Resolution im Sinne der Logik korrekt unter folgenden Bedingungen:

1. Interpretiere Implikationen in Regeln als Äquivalenzen. Z.B. wird das Programm

   ```
   p(a).
   p(b).
   ```

   als Formel $\forall x : p(x) \leftrightarrow (x = a \vee x = b)$ interpretiert, Man spricht dann auch von der „**Vervollständigung**“ (completion) des Programms.

2. Beweise `\+` l nur, falls l variablenfrei ist (d.h. man muss eine flexible Selektionsregel anwenden).

Die Notwendigkeit der letzten Einschränkung soll durch ein kleines Beispiel gezeigt werden:

```
p(a).
q(b).
```

Die Anfrage sei "`?- \+ p(X), q(X).`"

- Falls "`\+ p(X)`" direkt selektiert würde, wäre "`p(X)`" beweisbar und damit "`\+ p(X)`" nicht beweisbar, wodurch die Anfrage nicht beweisbar wäre.
- Falls der Beweis von "`\+ p(X)`" zunächst zurückgestellt wird: beweise "`q(X)`", was zu der Lösung {`X`↦`b`} führt, und beweise dann die zunächst zurückgestellte Aussage "`\+ p(b)`", was nun erfolgreich möglich ist. Damit erhalten wir als Gesamtlösung {`X`↦`b`}.

Erweiterte Programme

In erweiterten Programmen dürfen Regeln der Form l `:-` b vorkommen, wobei b eine beliebige prädikatenlogische Formel (d.h. mit Disjunktionen, Negation, Quantoren) ist:

dies ist manchmal praktisch, erhöht aber nicht die Ausdrucksmächtigkeit, da erweiterte Programme in normale Programme transformierbar sind [7]. Trotzdem werden erweiterte Programme für Datenbankanwendungen betrachtet.

Flexible Berechnungsregeln

Die Standardberechnungsregel von Prolog ist, dass alle Literale von links nach rechts bewiesen werden. Hierzu hat man auch verschiedene Modifikationen betrachtet:

Prolog mit Koroutinen: verzögere den Beweis von Literalen, bis bestimmte Bedingungen erfüllt sind (z.B. Literal variablenfrei). Dies ist nützlich zur korrekten Implementierung der SLDNF-Resolution, kann aber auch verwendet werden, um bestimmte Endlosableitungen in Programmen zu vermeiden.

Andorra-Prolog: beweise zuerst „deterministische“ Literale (bei denen höchstens eine Klausel passt). Dies führt häufig zu einer Verkleinerung des Suchraums. Kombiniert wird dieses Prinzip dann noch mit einer parallelen Auswertung:

Parallele logische Sprachen: Bei der Erweiterung um parallele Auswertung kann man folgende Arten unterscheiden:

- **UND-Parallelismus:** beweise mehrere Literale in einer Anfrage gleichzeitig (unabhängig oder abhängig bzgl. gleicher Variablen in verschiedenen Literalen)
- **ODER-Parallelismus:** probiere gleichzeitig verschiedene Alternativen (Regeln) aus (dies entspricht einer Breitensuche im SLD-Baum aller Ableitungen).
- **Committed Choice:** erweitere Klauselrümpfe um eine Bedingung (“guard”):

  ```
  <head> :- <guard> | <body>
  ```

  Falls mehrere Klauseln zu einem Literal passen: wähle nicht-deterministisch („paralleles Raten“) eine, bei der die Bedingung `<guard>` beweisbar ist, und ignoriere andere Alternativen (diese Prinzip ist aus logischer Sicht unvollständig, aber es ist gut geeignet zur Programmierung nebenläufiger Systeme).

Constraints („Einschränkungen“)

Der grundlegende Lösungsmechanismus in logischen Sprachen ist die Unifikation, was dem Lösen von Gleichungen zwischen Termen entspricht. Der Nachteil hiervon ist, dass Gleichungen rein syntaktisch gelöst werden, ohne dass ein „Ausrechnen“ stattfindet:

```
?- X=3+2.
X=3+2
```

Wünschenswert wäre hier aber die Lösung `X=5`.

```
?- 5=X+2.
no
```

Wünschenswert wäre hier aber die Lösung `X=3`.
Um letzteres zu erreichen, könnte man außer der Termgleichheit (Strukturgleichheit) auch die Gleichheit modulo interpretierten Funktionen und weitere Prädikate für spezielle Strukturen erlauben. Dies führt zu der Erweiterung der **Logikprogrammierung mit Constraints** (**Constraint Logic Programming, CLP**):

- erlaube statt Termen auch Constraint-Strukturen (d.h. Datentypen mit einer speziellen festgelegten Bedeutung)
- ersetze die Unifikation durch spezielle Lösungsalgorithmen für diese Constraints

Beispiel: CLP($\mathcal{R}$)

- Constraint-Struktur: reelle Zahlen (und Terme) und arithmetische Funktionen
- Constraints: Gleichungen / Ungleichungen zwischen arithmetischen Ausdrücken
- Lösungsalgorithmen: Gauß'sches Eliminationsverfahren (Gleichungen), Simplexmethode (Ungleichungen) und Unifikation (Terme)

Mit SICStus-Prolog:

```
?- use_module(library(clpr)).
?- {X=3+2}.
X=5.0
?- {5=3+X}
X=2.0
```

Als konkretes Beispiel betrachten wir die Hypothekenberechnung. Dafür sind die folgenden Parameter relevant:

H: Hypothek-Gesamtbetrag

L: Laufzeit (in Monaten)

Z: monatlicher Zinssatz

B: am Ende ausstehender Betrag

MR: monatliche Rückzahlung

Ein Programm, das diese Parameter in die richtige Relation setzt, sieht wie folgt aus:

```
:- use_module(library(clpr)).

hypothek(H,L,Z,B,MR) :- {L>0,L=<1,B=H*(1+Z*L)-L*MR}.
hypothek(H,L,Z,B,MR) :- {L>1}, hypothek(H*(1+Z)-MR,L-1,Z,B,MR).
```

Damit haben wir ein recht universell einsetzbares Programm, um verschiedene Hypothekenprobleme zu berechnen:
Monatliche Rückzahlung einer Hypothek?

```
?- hypothek(100000,180,0.01,0,MR).
MR=1200.68
```

Zeitdauer zur Finanzierung einer Hypothek?

```
?- hypothek(100000,L,0.01,0,1400).
L=125.901
```

Relation zwischen den Parametern `H,B,MR`?

```
?- hypothek(H,180,0.01,B,MR).
MR=0.012*H-0.002*B
```

Weitere Constraint-Strukturen:

- Boolsche Ausdrücke: diese können beim Hardwareentwurf und -verifikation eingesetzt werden
- **Endliche Bereiche:** diese haben zahlreiche Anwendungen bei Planungsaufgaben und Optimierungsproblemen (Operations Research), z.B. bei der Personalplanung, Containerverladung, Flottenplanung, Produktionsplanung, etc. Dies ist daher eine der praktisch wichtigsten Erweiterungen von Prolog.

  Das typische Vorgehen bei der Programmierung mit endlichen Bereichen ist wie folgt:

  1. Definiere endlichen(!) Wertebreich der Variablen („domain“).
  2. Definiere Constraints / Randbedingungen der Variablen.
  3. Definiere Aufzählungsverfahren / Backtracking-Strategie für Variablen („labeling“).

  Ohne dies weiter zu vertiefen, geben wir ein Beispiel hierfür an. Wir wollen das folgende kryptoarithmetisches Puzzle lösen:

  $$\begin{array}{r} \text{SEND} \\ +\ \text{MORE} \\ \hline \text{MONEY} \end{array}$$

  Hierbei steht jeder Buchstabe für eine Ziffer, und unterschiedliche Buchstaben für unterschiedliche Ziffern. Mit CLP(FD) (d.h. Constraint Logic Programming over Finite Domains) kann man dies Problem z.B. in SICStus-Prolog wie folgt lösen:

  ```
  % Lade den Constraint-Loeser fuer endliche Bereiche:
  :- use_module(library(clpfd)).

  sum(L) :-
  ```

```
    L=[S,E,N,D,M,O,R,Y],
    domain(L,0,9), % Wertebereich festlegen
    S#>0,          % Constraints festlegen
    M#>0,
    all_different(L),
                1000*S+100*E+10*N+D
              + 1000*M+100*O+10*R+E
      #= 10000*M+1000*O+100*N+10*E+Y,
    labeling([],L). % Variablenwerte aufzaehlen

?- sum([S,E,N,D,M,O,R,Y]).
D=7,E=5,M=1,N=6,O=0,R=8,S=9,Y=2
```

6.4 Datenbanksprachen

Eine relationale Datenbank ist im Prinzip eine Menge von Fakten ohne Variablen und ohne Funktoren. Eine Anfrage à la SQL ist eine prädikatenlogische Formel, die in eine Prolog-Anfrage mit Negation transformiert werden kann.
Aus diesem Grund ist es sinnvoll und konzeptuell einfach möglich, relationale Datenbanken in Prolog-Systeme einzubetten. Der Vorteil einer solchen Einbettung ist die Erweiterung der Funktionalität eines Datenbanksystems durch Methoden der Logikprogrammierung. Dies führt zu

Deduktive Datenbanken: Dies sind Mischungen aus relationale Datenbanken und Prolog, wobei meist aber nur eine Teilsprache namens **DATALOG** (Prolog ohne Funktoren) betrachtet wird.

Vorteil: man muss nur Basisrelationen in der Datenbank speichern und kann abgeleitete Relationen als Programm darstellen. Hierbei ist es wichtig anzumerken, dass das Programm auch rekursiv sein kann, wodurch die Mächtigkeit im Vergleich zu SQL erhöht wird.

Beispiel: Wir betrachten folgende Basisrelationen:
Lieferanten-Teile: `supp_part(Supp,Part)`
Produkt-Teile: `prod_part(Prod,Part)`

Nun wollen wir eine abgeleitete Relation definieren: Produkt-Lieferant `prod_supp`

```
prod_supp(P,S) :- prod_part(P,Part), supp_part(S,Part).
prod_supp(P,S) :- prod_part(P,Part), prod_supp(Part,S).
                  % Ein Produkt kann andere Produkte enthalten!
```

Dies ist eine rekursive Relation, die so nicht direkt in SQL formulierbar ist.

Aspekte von DATALOG:

- Anfrageoptimierung: effizientes Finden von Antworten (top-down/bottom-up-Auswertung)

- Integritätsprüfung: Integritätsbedingungen sind logische Formeln über Datenbank-Relationen, deren Erfüllbarkeit nach jeder Datenbank-Änderung (effizient!) geprüft werden muss.

7 Sprachkonzepte zur nebenläufigen und verteilten Programmierung

Bisher haben wir nur Programmiersprachen mit sequentiellen Berechnungsprinzipien betrachtet. Durch die starke Vernetzung von Rechnern und auch mehreren CPUs in einzelnen Rechnern wird das Rechnen in Prozessen und Netzwerken immer wichtiger. In diesem Kapitel wollen wir hierzu geeignete Programmiersprachen und Sprachkonzepte betrachten.

7.1 Grundbegriffe und Probleme

Unter einem **Prozess** verstehen wir den Ablauf eines sequentiellen Programms, d.h. ein Prozess enthält alle dafür notwendigen Informationen, wie Programmzähler, Speicher, u.ä. Somit müssen wir folgende Begriffe auseinanderhalten:

- Programm: statische Beschreibung möglicher Abläufe
- Prozess: ein dynamischer Ablauf eines Programms

Der Ablauf eines Programms / Prozesses kann neue Prozesse erzeugen. Falls ein Programm mehrere Prozesse beschreibt, sprechen wir auch von einem **Thread** („Faden“): darunter verstehen wir einen Ablauf in einem Programm (z.B. die Veränderung eines Programmzählers). Eine **multi-threaded language / system** enthält in der Regel mehrere Threads. In diesem Fall sprechen wir auch von **multi-processing**.

Motivation für Mehrprozessprogrammierung:

- Erhöhung der Effizienz (mehrere CPUs)
- natürliche Problemabstraktion:
  - physikalisch verteiltes System (z.B. Internet)
  - mehrere Benutzer, Simulationen,...

Begriffsabgrenzung (dies ist leider nicht ganz einheitlich):

Nebenläufiges System: enthält mehrere Threads / Prozesse

Verteiltes System: Prozesse laufen auf unterschiedlichen Prozessoren, die oft weit auseinderliegen aber mit einem Netzwerk verbunden sind

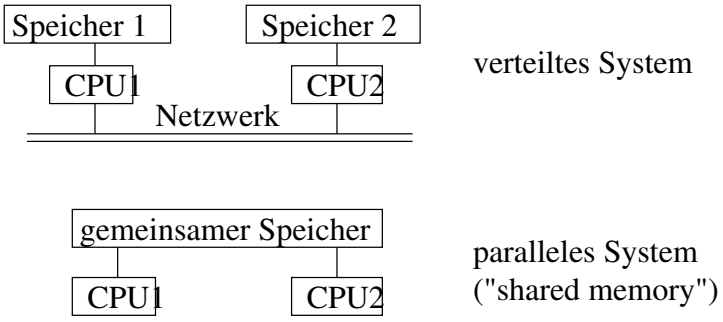


Abbildung 7.1: Vergleich verteiltes / paralleles System.

Paralleles System: wie verteilt, aber Prozessoren enger gekoppelt (z.B. gemeinsamer Speicher, schneller Bus) $\Rightarrow$ Parallelrechner

Nebenläufigkeit / Verteiltheit: Dies ist eine logische Eigenschaft, die durch die Programmiersprache explizit unterstützt werden muss.

Parallelität: Dies ist eine Eigenschaft der Implementierung, die durch eine Programmiersprache unterstützt werden kann, aber auch implizit, z.B. durch einen parallelisierenden Compiler, realisiert wird.

Allerdings: Die Probleme bei der Programmierung sind in beiden Fällen ähnlich.

Probleme bei Nebenläufigkeit:

1. Prozesse laufen nicht unabhängig ab.
2. Prozesse kopieren und tauschen Daten aus.
3. Prozesse greifen auf gemeinsame Datenbasis zu.
4. Prozesse haben Abhängigkeiten (Ergebnis eines Prozesss wird von anderen Prozessen genutzt).
5. Prozesse müssen auf andere Prozesse warten.
6. Prozesse müssen evtl. auf einem einzigen Prozessor ablaufen (interleaving).

2.+3. $\rightsquigarrow$ Kommunikation
4.+5. $\rightsquigarrow$ Synchronisation
6. $\rightsquigarrow$ Scheduling (hier nicht weiter betrachtet, da dies eine typische Aufgabe des Betriebssystems ist)

Kommunikation und Synchronisation hängen häufig eng zusammen, da oft durch Kommunikation auch synchronisiert wird.

Wichtiger Mechanismus zur Lösung dieser Probleme:

Gegenseitiger Ausschluß

Beispiel: Zwei Prozesse verändern gemeinsame Daten:

```
P1: ⋮              P2: ⋮
    x=x*2;             x=x+1;
    ⋮                  ⋮
```

Falls initial `x=0` gilt, sollte nach Ablauf von P1 und P2 `x=1` oder `x=2` gelten, je nach, welche Zuweisungsinstruktion eher ausgeführt wird. Aber eine Zuweisung wie `x=x*2` oder `x=x+1` ist keine Elementaroperation, sondern `x=x+1` besteht z.B. aus den Maschinenanweisungen

```
load x in a;
incr a;
store a in x
```

Somit ist auch folgende Ausführung möglich (Zeitdiagramm):

```
P1            x=0    P2
⋮                    ⋮
load:  a=0           ...
                     load: a=0
mult2: a=0
                     incr: a=1
              x=1    store
store         x=0
```

Lösung: Gegenseitiger Ausschluss der Modifikation von `x` in P1 und P2. Man sagt auch, dass "`x=x+1`" ein **kritischer Bereich** ist.

Ein einfacher Mechanismus zur Realisierung des gegenseitigen Ausschlusses sind **Sperren** (**locks**) auf gemeinsame Ressourcen.

Beispiel:

```
p1:
    ⋮
    lock(x);
    x:=x*2;
    unlock(x);
    ⋮
```

`lock(x):` Prüfe, ob Objekt `x` gesperrt:
Falls ja: warte, bis `x` nicht gesperrt.

Falls nein: Sperre `x`
Diese Operation ist atomar (nicht unterbrechbar wie z.B. eine Zuweisung).

`unlock(x):` entsperre `x` (und aktiviere evtl. wartenden Prozess)

Hierdurch wird das obige Problem vermieden, aber es ergeben sich auch neue Probleme, die wir am klassischen Beispiel der essenden Philosophen erläutern wollen.

Beispiel: Essende Philosophen (Dining Philosophers)

- 5 Philosophen (Prozesse, $P_1, \ldots, P_5$) essen und denken abwechselnd
- es befindet sich jeweils ein Stäbchen S_i zwischen den Philosophen P_i und P_{i+1} ($\approx$ gemeinsame Ressource)
- P_i benötigt zum Essen seine beiden Nachbarstäbchen S_i und S_{i+1}. Da die Philosophen im Kreis am Tisch sitzen, soll gelten: $S_6 \equiv S_1$.

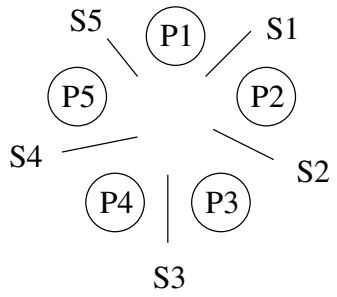


Abbildung 7.2: Struktur der essenden Philosophen

Je nach Programmierung können folgende Probleme auftreten:

Verklemmung (Deadlock)
Das Programm für den Philosophen P_i lautet wie folgt ($i = 1, \ldots, n$):

```
loop
   lock(S_i);
   lock(S_{i+1});
   esse;
   unlock(S_i);
   unlock(S_{i+1});
   denke;
end
```

Falls alle P_i gleichzeitig `lock(`S_i`)` ausführen, sind alle Philosophen blockiert.

Blockade (Livelock)
Kein Deadlock, aber kein Prozess macht Fortschritte.

Hierbei ist der Code wie oben, aber nach `lock(`S_i`)` wird nun `unlock(`S_i`)` ausgeführt, falls S_{i+1} gesperrt ist.
Hierdurch können eventuell alle Philosophen P_i in die folgende Schleife geraten:

```
lock(Si);
unlock(Si);
lock(Si);
unlock(Si);
⋮
```

Fairness
Jeder Philosoph P_i, der essen möchte, soll dies irgendwann tun können.
Unfaire Lösung: Lasse nur P_1 und P_3 essen.
Ebenfalls unfair(!): Lasse reihum P_1, P_2, P_3, P_4, P_5 essen (dies ist unfair, denn die Philosophen haben verschieden viel Hunger und brauchen unterschiedlich lange zum Essen).

Busy Waiting
Falls `lock(`S_i`)` ausgeführt werden soll, prüfe immer wieder, ob Sperre auf S_i noch vorhanden ist. Diese Prüfschleife belastet den Prozessor und sollte vermieden werden. Wie man dies machen kann, wird im nächsten Abschnitt erläutert.

7.2 Sprachkonstrukte zum gegenseitigen Ausschluß

Man kann Busy Waiting vermeiden durch:

Semaphore (Dijkstra 1968): Konzeptuell ist ein Semaphor eine nichtnegative ganzzahlige Variable mit einer Prozesswarteschlange und zwei Operationen:

`P` („passeren“) oder „wait“:
: falls Semaphor > 0, erniedrige diesen um 1; sonst stoppe die Ausführung des aufrufenden Prozesses (d.h. trage diesen in die Prozesswarteschlange des Semaphors ein).

`V` („vrijgeven“) oder „signal“:
: falls Prozesse in der Warteschlange warten, aktiviere einen, sonst erhöhe den Semaphor um 1.

Der Initialwert eines Semaphors ist 1 (dann sprechen wir auch von einem **binäreren Semaphor**) oder $n > 1$) (falls n Prozesse in den kritischen Bereich gleichzeitig eintreten können sollen).

Wichtig: `P`/`V` sind **unteilbare** Operationen und werden durch das Betriebssystem realisiert.

Mit Semaphoren können wir kritische Bereiche absichern, indem wir `P` zu Beginn und `V` am Ende benutzen.

Beispiel: s: binärer Semaphor

```
⋮
P(s);
x := x+1;  // kritischer Bereich
V(s);
⋮
```

Semaphore in Programmiersprachen:

- Algol 68: erste Programmiersprache mit eingebauten Semaphoren
- C: nutzt Unix-Bibliotheken für Semaphore (`sys/sem.h`)
  Erzeugung neuer Prozesse durch Kopieren (`fork`) eines existierenden

Nachteil von Semaphoren:

- low-level ($\approx$ Assembler)
- fehleranfällig, insbesondere bei der Programmierung von komplexen, gemeinsam benutzten Datenstrukturen

Idee (Hoare 1974): höhere Abstraktion zur Synchronisation:

Ein **Monitor** ist ein Modul / Objekt, welches den synchronisierten Zugriff auf Daten organisiert. Ein Monitor besteht aus:

- lokalen Daten
- Operationen auf diesen Daten
- Initialisierungsteil
- Mechanismus zur Suspension („delay“) und zum Aufwecken („continue“) von Operationen / Prozessen

Wichtig:

- der Zugriff auf lokale Daten erfolgt nur über die Operationen des Monitors ($\approx$ ADT)
- nur ein Prozess kann gleichzeitig in den Monitor eintreten (d.h. eine Operation aufrufen)

Beispiel: Puffer in Concurrent-Pascal:

```
type buffer =
  monitor
    var contents: array[1..n] of ...;
        num: 0..n;  { number of elements }
        sender, receiver: queue;
```

```
  procedure entry append (item: ...);
  begin
    if num=n then delay(sender); { buffer is full }
    ⋮ { insert in buffer }
    continue(receiver);
  end;

  procedure entry remove (var item: ...);
  begin
    if num=0 then delay(receiver);
    ⋮ { take one item }
    continue(sender);
  end;
begin
  num := 0; ...
end;
```

Anmerkungen:

- `delay(Q)` fügt aufrufenden Prozess in die Warteschlange `Q` ein
- `continue(Q)` aktiviert einen Prozess aus der Warteschlange `Q`
- `entry` markiert eine von außen aufrufbare Monitor-Operationen

Elegantes Konzept (alle kritischen Bereiche und Datenstrukturen sind in einem ADT gekapselt), aber trotzdem wurde dies eher selten in Programmiersprachen verwendet (z.B. in Concurrent Pascal, Mesa). Man findet aber konzeptuell ähnliche Konzepte in Programmiersprachen, z.B. bietet Java ein Monitor-ähnliches Konzept an (siehe Kaptiel 7.3).

Rendezvouz: Kommunikation und Synchronisation von Prozessen durch koordiniertes Abarbeiten einer Prozedur
Dieses Synchronisationskonzept findet man in den Sprachen Ada (und auch in Concurrent C).
Die Grundidee des Rendezvouz ist eine Client-Server-Kommunikation

- Server: bietet einen oder mehrere Einstiegspunkte mit Parametern zum Aufruf an und wartet auf einen Aufruf
- Client: ruft diese Einstiegspunkte auf und wartet bei dem Aufruf (d.h. er blockiert), bis dieser akzeptiert und abgearbeitet ist (⇒ „Rendezvouz“).

Sprachkonstrukte hierfür (in Ada):

- Einstiegspunkt:

```
accept <entryname + parameter> do
  <body>
end
```

- Aufruf:

```
<servername>.<entryname + parameter>
```

Zeitlicher Ablauf:

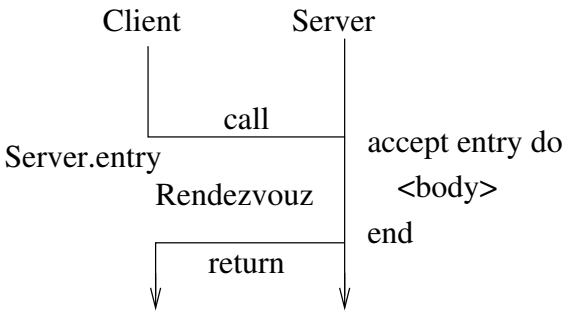


Abbildung 7.3: Ablauf Rendezvouz.

- Der Aufruf kann vor `accept` oder auch umgekehrt erfolgen, da beide Partner bis zur Synchronisation warten.
- Ein Server kann mehrere Clients bedienen.
- Ein Server kann mehrere Einstiegspunkte anbieten. Hierzu gibt es die allgemeinere Form:

```
select
  when <condition> => accept ... end; ...
or
  when <condition> => accept ... end; ...
end select;
```

Unterschiede zu Monitoren:

- Der Server ist kein eigenständiges Modul, das gemeinsame Datenstrukturen verwaltet.
- Jeder Prozess (in Ada: `task`) kann mittels einer `accept`-Anweisung zu einem Server werden.

- Server/Client-Funktion nicht global fixiert (im Gegensatz zu einem Monitor, der immer ein Server ist), sondern kann dynamisch variieren.
- Es gibt eine Warteschlange pro Einstiegspunkt.

Beides sind daher unterschiedliche aber hohe Konzepte (im Gegensatz zu Semaphoren) zur Synchronisation.

Beispiel: Puffer in Ada mittels Rendevouz-Konzept

```
task Buffer is
  entry Append(item: in INTEGER);
  entry Remove(item: out INTEGER);
end;

task body Buffer is
  contents: array (1..n) of INTEGER;
  num: INTEGER range 0..n := 0;
  ipos, opos: INTEGER range 1..n := 1;
 begin loop
  select
    when num<n =>
      accept Append(item: in INTEGER) do
        contents(ipos) := item;
      end;
    ipos := (ipos mod n)+1; num := num+1;
  or
    when num>0 =>
      accept Remove(item: out INTEGER) do
        item := contents(opos);
      end;
    opos := (opos mod n)+1; num := num-1;
  end select;
 end loop;
end Buffer;
```

Anmerkungen:

- Die Abarbeitung eines `accept` ist ähnlich zu einem Prozeduraufruf mit üblicher Parameterübergabe, aber der Prozeduraufruf des Client wird im Server abgearbeitet.
- Erzeuger-Client: Server-Aufruf `Buffer.Append(v)`
- Verbraucher-Client: Server-Aufruf `Buffer.Remove(v)`
- Das Rendezvouz ist erst möglich, falls die „when“-Bedingung erfüllt ist.

Nachrichten (message passing): Kommunikation und Synchronisation zwischen Prozessen durch das Versenden von Nachrichten über Kanäle, meistens vom Betriebssystem unterstützt

Unterschiede gibt es bei diesem Konzept bei

- der Struktur der Kanäle (symmetrisch/asymmetrisch)
- Synchronisation

Vorteile:

- keine kritischen Bereiche, da Prozesse konzeptuell getrennt (keine gemeinsamen Speicherbereiche)
- flexible Kommunikationsstrukturen ($1:1$, $1:n$, $n:1$, $n:m$)

Modelle zum Nachrichtenaustausch:

- **Punkt-zu-Punkt:** Ein Sender (Prozess) schickt eine Nachricht an einen Empfänger (Prozess).
  - **symmetrisch:** Empfänger und Sender kennen sich gegenseitig mit Namen, dadurch kann der Empfänger direkt antworten (z.B. Telefon)
  - **asymmetrisch:** Nachrichtenfluss nur in eine Richtung (z.B. „pipes“ in Unix) Fall es viele Sender gibt, muss der Empfänger die eingehenden Nachrichten zwischenspeichern: jeder Empfänger hat einen Empfangspuffer, auch **mailbox** oder **„port“** genannt.
  - **synchrone Kommunikation:** Sender wartet, bis der Empfänger die Nachricht akzeptiert hat (häufig auf Hardwareebene, z.B. in Occam zur Programmierung von Transputern)
  - **asynchrone Kommunikation:** Sender arbeitet nach dem Versand einer Nachricht direkt weiter, die Empfängerbestätigung muss explizit programmiert werden (häufig bei Netzwerken und many-to-one-Kommunikation)
- **Rendevouz:** s.o.; im Wesentlichen synchroner Nachrichtenaustausch zwischen Sender und Empfänger
- **Prozedurfernaufrufe (remote procedure call, RPC, RMI (Java)):** ähnlicher Mechanismus wie Rendevouz, jedoch analoge Behandlung wie lokaler Prozeduraufruf:
  - einfache Einbettung in existierende Programmiersprachen
  - „einfache“ Portierung existierender sequentieller Programme in nebenläufigen Kontext (jedoch komplexere Fehlersituationen möglich!)
- **Gruppenkommunikation (broadcasting, multicasting):** ein Prozess kann Nachrichten direkt an eine Menge von Empfängerprozessen schicken

Programmierunterstützung dieser Konzepte:

- durch Betriebssystem und/oder Bibliotheken
  $\Rightarrow$ im wesentlichen keine Änderung der Basissprache
- durch spezielle Konstrukte in Programmiersprachen:
  Beispiel: Occam: Assembler für Transputer (eng gekoppelte Parallelrechner); synchrone Kommunikation (Punkt-zu-Punkt) über Kanäle

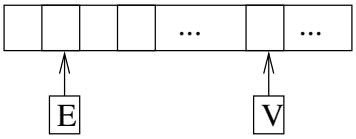
```
CHAN OF BYTE input, output -- Kanaldeklaration
SEQ
  input?x    -- empfange Wert x von Kanal input
  output!x   -- sende den Wert x ueber den Kanal output
```

Zusammenfassung: Programmiersprachen müssen zur nebenläufigen Programmierung folgendes bereitstellen:

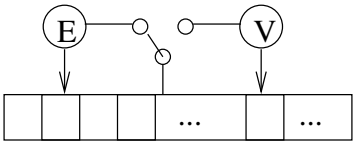
1. Konstrukte zur Erzeugung neuer Prozesse
   (Ada: `task`, C/Unix: `fork`, Occam: `par`,...)
2. Konstrukte zur Kommunikation / Synchronisation zwischen Prozessen

Veranschaulichung von Synchronisationsmechanismen für einen Puffer (hierbei ist E ein Erzeuger und V ein Verbraucher):

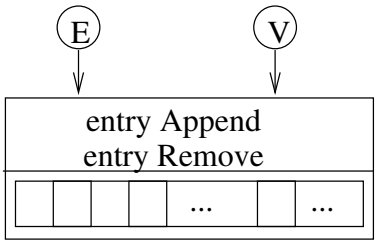
Direktzugriff ohne Synchronisation:



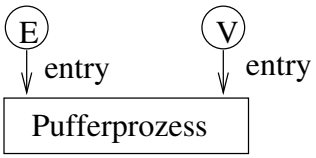
Semaphor:



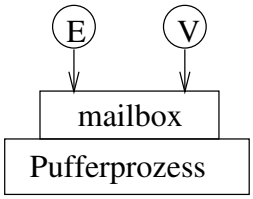
Monitor:



Rendezvous:



Nachrichten:



7.3 Nebenläufige Programmierung in Java

Java wurde ursprünglich als Programmiersprache für das Internet eingeführt und enthält daher Möglichkeiten zur nebenläufigen Programmierung:

1. Standardklasse `Thread` für Prozesse ⇒ definiere eigene Prozesse als Unterklasse von `Thread`
2. Synchronisation durch Monitor-ähnliches Konzept: jedes Objekt kann als Monitor agieren, wenn alle Attribute `private` und alle Methoden `synchronized` sind.

Prozesse in Java: Threads

Analog zu anderen Basisklassen wie `String` gibt es eine Klasse `Thread`, deren Objekte ablaufbare Prozesse sind. Diese entsprechen nicht unbedingt Betriebssystem-Prozessen, da die JVM Threads selbst verwalten kann.
Wichtige Methoden der Klasse `Thread` sind:

`start()`: startet einen neuen Thread, wobei in diesem die Methode `run()` ausgeführt wird.

`run()`: Methode, die ein Thread ausführt. Diese wird üblicherweise in Unterklassen redefiniert. Wenn die Ausführung von `run()` terminiert, wird der Thread beendet.

`sleep(long ms)`: stoppe Thread für `ms` Millisekunden

Beispiel: Druckprozess

```
class PrintThread extends Thread {
    public void run () {
      <Anweisungen zum Drucken>
    }
}
```

Starten des Druckprozesses:

```
⋮
new PrintThread().start();
⋮
```

Weitere Möglichkeit: implementiere das Interface `Runnable` (dieses enthält nur die Methode `run()`) und erzeuge einen Thread mit dem Konstruktor `Thread(Runnable target)`:

```
class PrintSpooler implements Runnable {
    public void run() { ... };
}
```

Starten des Druckprozesses:

```
⋮
new Thread(new PrintSpooler()).start();
⋮
```

Synchronisation in Java

Die Synchronisation in Java ist verankert in der Klasse `Object`: jedes Objekt kann synchronisieren und enthält eine Sperre (lock). Auf die kann man allerdings nicht explizit zugreifen, sondern es gibt eine Synchronisationsanweisung:

```
synchronized (expr) statement
```

Bedeutung dieser Anweisung:

1. Werte `expr` zu einem Objekt `o` aus.
2. Falls `o` nicht gesperrt ist, sperre `o`, führe `statement` aus, entsperre `o`.
3. Falls `o` vom gleichen Prozess gesperrt ist, führe `statement` aus.
4. Falls `o` von einem anderen Prozess gesperrt ist, warte, bis `o` entsperrt ist.

Anstelle der Synchronisation einzelner Anweisungen/Blöcke kann man auch (strukturierter!) Methoden synchronisieren:

```
synchronized} τ method(...) {
  Rumpf
}
```

Dies entspricht dann folgender Definition:

```
τ method(...) {
  synchronized(this) { Rumpf }
}
```

d.h. der Rumpf der Methode `method` wird nur ausgeführt, falls man die Sperre auf dem Objekt `o` bekommt oder schon hat.

Damit können wir einen Monitor-orientierten Stil realisieren (dies ist auch empfehlenswert wegen seiner guten Programmstruktur!):

- alle Attribute sind als `private` deklariert
- alle nicht `private`-Methoden sind `synchronized`

Effekt: nur ein Prozess kann gleichzeitig Attribute eines Objektes verändern.

Jedoch ist auch noch notwendig: **Suspension** von Prozessen, bis bestimmte Bedingungen erfüllt sind (vgl. das Puffer-Beispiel). Zu diesem Zweck hat in Java jedes Objekt eine Prozesswarteschlange, die zwar nicht direkt zugreifbar ist, aber die mit folgenden Methoden beeinflusst werden kann:

`o.wait()`: suspendiere den aufrufenden Prozess und gebe gleichzeitig die Sperre auf das Objekt `o` frei

`o.notify()`: aktiviere einen(!) Prozess, der mit `o.wait()` vorher suspendiert wurde (dieser Prozess muss sich wieder um die frei gegebenen Sperre bewerben!)

`o.notifyAll()`: aktiviere alle diese Prozesse

Typische Anwendung dieser Methoden innerhalb von synchronisierten(!) Objektmethoden:

- `wait()` in `while`-Schleife, bis die Bedingung zur Ausführung erfüllt ist:

```
synchronized void doWhen() {
    while (!<bedingung>) wait();
    <Hier geht es richtig los>
}
```

- `notify()` durch Prozess, der Objektzustand so verändert, dass ein wartender Prozess evtl. etwas machen und daher aktiviert werden kann.

```
synchronized void change() {
    <Zustandsaenderung>
    notify(); // auch: notifyAll()
}
```

Wichtig: die Reihenfolge ist beim Aktivieren nicht festgelegt (d.h. es ist keine wirkliche Warteschlange!), daher sollte `notifyAll()` verwendet werden, falls möglicherweise mehrere Prozesse warten.

Beispiel: Einen *synchronisierter Puffer* könnten wir in Java wie folgt implementieren:

```
class Buffer {
    private int n;          // Pufferlaenge
    private int[] contents; // Pufferinhalt
    private int num, ipos, opos = 0;

    public Buffer (int size) {
        n = size;
        contents = new int[size];
    }

    public synchronized void append (int item) throws InterruptedException {
        while (num==n) wait();
        contents[ipos] = item;
        ipos = (ipos+1)%n;
        num++;
        notify(); // oder: notifyAll()
    }

    public synchronized int remove () throws InterruptedException {
        while (num==0) wait();
        int item = contents[opos];
        opos = (opos+1)%n;
        num--;
        notify(); // oder: notifyAll()
        return item;
    }
}
```

Prozesse, die für einige Zeit inaktiv sind (die z.B. mittels `sleep()` oder `wait()` auf Ereignisse warten), können die Ausnahme `InterruptedException` werfen, falls diese durch einen Interrupt unterbrochen werden. Aus diesem Grund muss die `InterruptedException` bei Benutzung von `sleep()` oder `wait()` auch behandelt oder deklariert werden. Üblicherweise kann die Behandlung dieser Ausnahmen durch direkte Beendigung erfolgen (s.u.), aber man könnte auch noch offene Dateien schließen oder ähnliche „Aufräumaktionen“ durchführen.
Ein Erzeuger für einen Puffer könnte dann so aussehen:

```
class Producer extends Thread {
    private Buffer b;
```

```
    public Producer (Buffer b) {
        this.b=b;
    }

    public void run() {
        try {
            for (int i=1; i<=10; i++) {
                System.out.println("into buffer: "+i);
                b.append(i);
                sleep(5);
            }
        }
        catch (InterruptedException e)
          { return; }  // terminate this thread
    }
}
```

Ein Verbraucher für einen Puffer könnte die folgende Form haben:

```
class Consumer extends Thread {
    private Buffer b;

    public Consumer (Buffer b) {
        this.b=b;
    }

    public void run() {
        try {
            for (int i=1; i<=10; i++) {
                System.out.println("from buffer: "+b.remove());
                sleep(20);
            }
        }
        catch (InterruptedException e)
          { return; } // terminate this thread
    }
}
```

Das Starten der Erzeugers und Verbrauchers kann wie folgt passieren:

```
Buffer b = new Buffer(4);
new Consumer(b).start();
new Producer(b).start();
```

7.4 Synchronisation durch Tupelräume

Die bisherigen Synchronisationsmechanismen waren Client/Server-orientiert, d.h. es wurden typischerweise zwei Partner/Prozesse synchronisiert.
Ein alternativer Ansatz ist das Konzept **Linda** [2]: hier erfolgt die Kommunikation durch Austausch von Tupeln
Das grundlegende Modell ist dabei wie folgt charakterisiert:

- ein zentraler Speicher: **Tupelraum** ≈ Multimenge von Tupeln
- viele unabhängige Prozesse, die Tupel in den Tupelraum einfügen oder auslesen.
- sprachunabhängig: Linda ist ein Kommunikationsmodell, das zu verschiedenen Sprechen hinzugefügt werden kann. So gibt es C-Linda, Fortran-Linda, Scheme-Linda, Java-Linda (Java Spaces / Jini),...

Linda basiert auf folgenden Grundoperationen:

`out(t):` füge ein Tupel `t` in den Tupelraum ein, wobei ein Tupel eine geordnete Liste von Elementen ist. Vor dem Einfügen werden die einzelnen Elemente entsprechend ihrer Bedeutung in der jeweiligen Programmiersprachen ausgewertet.
Beispiel: `out("hallo",1+3,6.0/4.0)` $\rightsquigarrow$ Tupel `("hallo",4,1.5)` in den Tupelraum einfügen

`in(t)`: Entferne Tupel `t` aus dem Tupelraum, falls es vorhanden ist; sonst warte, bis `t` in den Tupelraum eingefügt wird. `t` kann auch ein Muster sein, wobei einige Argumente Variablen sind. In diesem Fall wird ein passendes Tupel gesucht und die Variablen entsprechend instantiiert.
Beispiel in C-Linda: `in("hallo",?i,?f)` $\rightsquigarrow$ entferne obiges Tupel und setze `i=4` und `f=1.5`

`rd(t)`: wie `in(t)`, aber ohne Entfernung des Tupels

`eval(t)`: erzeugt einen neuen Prozess, der dann `t` (nach Auswertung) in den Tupelraum einfügt.
Unterschied zu `out(t)`: Auszuwertendes Objekt kann auch eine Funktion sein, die als Ergebniswert ein Tupel hat.

`inp(t), rdp(t)` (manchmal): wie `in(t)` oder `rd(t)`, aber ohne Blockade, falls das Tupel nicht vorhanden ist (sondern z.B. `false` als Ergebnis)

Beispiel: Zwei Prozesse, die abwechselnd aktiv werden („Ping-Pong-Spieler“), in C-Linda:

```
ping(...)
{ while(...)
  { out("ping");  /* spiele ping */
    in("pong");   /* warte auf pong */
```

```
  }
}

pong(...)
{ while(...)
  { in("ping");
    out("pong");
  }
}
```

Hauptprogramm:

```
eval(ping(...)); eval(pong(...));
```

Auch Datenstrukturen sind durch Tupel darstellbar: wir können einen n-elementigen Vektor $V = (e_1, \ldots, e_n)$ als n-Tupel darstellen:

("V",1,e_1)
⋮
("V",n,e_n)

Vorteil: direkter Zugriff auf einzelne Komponenten möglich, z.B. Multiplikation der 3. Komponente mit der Zahl 3:

```
in("V",3,?e);
out("V",3,3*e);
```

Anwendung: Verteilung und parallele Bearbeitung komplexer Datenstrukturen.
Beispiel: ein Rechenprozess für jede Komponente:

```
for (i=1; i<=n; i++) eval(f(i)); /* ein Prozess fuer jede Komponente */
```
⋮
```
f(int i)
{ in("V",i,?v);
  out("V",i,compute(v));
}
```

Dies ist relevant, falls `compute` aufwändig zu berechnen ist.

Das Linda-Modell erlaubt die verständliche Formulierung vieler Synchronisationsprobleme, wie folgende Beispiele zeigen.

Beispiel: „Dining Philosophers“: eine Implementierung in Linda:

```
phil(int i)
{ while(true)
  { think();
```

```
    in("stab",i);
    in("stab",(i+1)%5);
    eat();
    out("stab",i);
    out("stab"(i+1)%5);
  }
}
```

Initialisierung:

```
for(i=0; i<5; i++)
{ out("stab",i);
  eval(phil(i));
}
```

Beispiel: Client-Server-Kommunikation:
Idee: Zur Zuordnung der Antworten zu den Anfragen nummeriere alle Anfragen durch (dies entspricht dem Vergeben einer Auftragsnummer und Zuordnung dieser Nummer zu den Ergebnissen):

```
server()
{ int i=1;
  out("serverindex",i);
  while(true)
  { in("request",i,?req);
    ⋮
    out("response",i,result);
    i++;
  }
}

client()
{ int i;
  in("serverindex",?i); /* hole Auftragsnummer */
  out("serverindex",i+1);
  ⋮
  out("request",i,req); /* in req steht der eigentliche Auftrag */
  in("response",i,?result);
  ⋮
}
```

7.5 Nebenläufige logische Programmierung: CCP

Logische Programmiersprachen haben ein hohes Potenzial für die parallele Implementierung:

- UND-Parallelismus
- ODER-Parallelismus

Allerdings ist in diesen Fällen die Parallelität nicht sichtbar für den Programmierer, sondern nur ein Implementierungsaspekt.

Dies ist anders bei „nebenläufigen logischen Programmiersprachen“:

- enthalten explizite Konstrukte zur Nebenläufigkeit (z.B. Committed Choice, vgl. Kapitel 6.3)
- erstmalig entwickelt im Rahmen von Japans „Fifths Generation Computing Project“
- realisiert in vielen verschiedenen Sprachen
- weiterentwickelt von Saraswat ([13]) zu CCP-Modell

CCP: **Concurrent Constraint Programming**

- Erweiterung von CLP um Nebenläufigkeit
- Synchronisation durch Gültigkeit von Constraints
- Prinzip: nebenläufige Agenten (≈ Prozesse) arbeiten auf gemeinsamen **Constraint-Speicher** (CS)
- jeder Agent:
  - wartet auf Information in CS („**ask**“), oder
  - fügt neue (konsistente) Information zu CS hinzu („**tell**“)

  ⇒ Linda (ohne „in“) mit Constraints statt Tupel
- CS wird mit Information angereichert
  ⇒ Reihenfolge der Agentenbearbeitung kann andere Agenten nicht blockieren
  ⇒ (automatische) Deadlockvermeidung

Beispiel: „Maximum-Agent“:

```
max(X,Y,Z) := X <= Y | Y=Z
            □ X >= Y | X=Z
```

Hier:

- links von “`|`”: „ask“-Constraints (≈ “`rd`” in Linda)

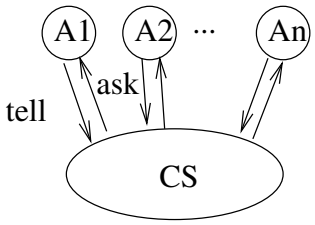


Abbildung 7.4: Skizze der Agenten und des CS

- rechts von "`|`": „tell"-Constraints (≈ "`out`" in Linda)
- □ kennzeichnet Alternativen (committed choice)

Agent `max(X,Y,Z)`

- wartet, bis Relation zwischen `X` und `Y` aus CS bekannt, und
- wählt dann **eine** passende Alternative und fügt Gleichheitsconstraint zu CS hinzu.

Mögliche Situationen („⊨" bezeichnet die logische Implikation aus dem Constraint-Speicher):
`CS` ⊨ `X=1` und `CS` ⊨ `Y=3`: füge `Z=3` zu CS hinzu
`CS` ⊨ `X=Y+2`: füge `X=Z` zu CS hinzu

Beispiel: Nichtdeterministischer Strommischer: mischt Ströme (Listen), sobald Teile bekannt sind

```
merge(S1,S2,S) := S1=[] | S=S2
                □ S2=[] | S=S1
                □ S1=[E|ST1] | S=[E|ST], merge(ST1,S2,ST)
                □ S2=[E|ST2] | S=[E|ST], merge(S1,ST2,ST)
```

Mit diesem Mischer kann man Kommunikationsstrukturen zwischen mehreren Agenten, die ask/tell verwenden, aufbauen (daher ist CCP auch geeignet für die Implementierung von Multiagentensystemen). Eine solche Struktur wollen wir kurz skizzieren:

Beispiel: Client/Server-Programmierung:
Der Client/Erzeuger hat folgende Programmstruktur:

```
client(S,...) := ... | compute(E), S=[E|S1], client(S1,...)
```

Der Server/Verbraucher hat diese Programmstruktur:

```
server(S,...) := S=[E|S1] | process(E), ..., server(S1,...)
```

Dann können wir einen Server mit zwei Clients wie folgt bauen:

```
client(S1,...), client(S2,...), merge(S1,S2,S), server(S,...)
```

7.6 Nebenläufige funktionale Programmierung: Erlang

Es gibt viele verschiedene Ansätze zur Erweiterung von funktionalen Programmiersprachen um Nebenläufigkeit. Hier betrachten wir allerdings nur die Sprache Erlang [1].

Erlang:

- entwickelt von der Firma Ericsson zur Programmierung von Telekommunikationssystemen
- Anwendungsbereich: komplexe, verteilte Systeme
- entstanden als Mischung logischer und strikter funktionaler Sprachen, die um ein Prozesskonzept erweitert wurde

Erlang-Programme:

- Menge von Modulen, jedes Modul ist eine Menge von Funktionsdefinitionen
- Datentypen: Zahlen, Atome, Pids (process identifiers), Tupel, Listen
- Pattern matching bei Funktionsdefinitionen und(!) Kommunikation

Beispiel: Listenkonkatenation in Erlang:

```
append([],    Ys) -> Ys;
append([X|Xs],Ys) -> [X|append(Xs,Ys)].
```

Syntax: Prolog-ähnlich: Variablen beginnen mit Großbuchstaben, alles andere mit Kleinbuchstaben
Erweiterung: Tupel: {a,1,b} (Tupel mit Komponenten `a`, `1` und `b`)

Prozesse: hierzu bietet Erlang drei Konstrukte an:

```
spawn(<module>,<function>,<arglist>)
```

Dies erzeugt einen neuen Prozess, der die Berechnung "`<function>(<arglist>)`" durchführt. Das Ergebnis dieses `spawn`-Aufrufs ist ein Pid (process identifier).

Wichtig: jeder Prozess hat eine Nachrichtenwarteschlange (mailbox), die er lesen kann und in die andere schreiben können, in dem sie diesem Prozess Nachrichten schicken (s.u.).

```
<Pid>!<msg>
```

Dieses Konstrukt sendet eine Nachricht (einen Wert) `<msg>` an den Prozess `<Pid>`

```
receive
  <msg_1> -> <action_1>;
  ...
  <msg_n> -> <action_n>
end
```

Hierbei ist <msg$_i$> das Muster einer Nachricht, die in der mailbox gesucht wird, und <action$_i$> eine Sequenz von Funktionsaufrufen.
Bedeutung: suche in der Nachrichtenwarteschlange (mailbox) die erste Nachricht, die zu einem <msg$_i$> passt (und zu keinem <msg$_j$> mit $j < i$) und führe <action$_i$> aus.

Beispiel: Ein einfacher echo-Prozess:

```
- module(echo)        % Modulname
- export([start/0]).  % exportierte Funktionen

start() -> spawn(echo, loop, []).  % starte echo-Prozess

loop() -> receive
            {From,Msg} -> % empfange Paar von Pid und Inhalt
               From!Msg,  % sende Inhalt zur"uck
               loop()
          end.
```

Beispiellauf:

```
⋮
Id = echo:start(),
Id!{self(),hallo},  % self: pid des eigenen Prozesses
⋮
```

Beispiel: Ping-Pong in Erlang: Hier verwenden wir Ping/Pong-Nachrichten statt Tupel wie in Linda):

```
- module(pingTest).          % Modulname
- export([main/1, pong/0]).  % exportierte Funktionen

% n-mal Ping:
ping(Pid,0) -> okay;
ping(Pid,N) -> Pid!{self(),ping},
               receive
                 pong -> ping(Pid,N-1)
               end.

pong() -> receive
            {Pid,ping} -> Pid!pong,
                          pong()
          end.

main(N) -> Pid = spawn(pingTest, pong, []),  % starte pong-Prozess
           ping(Pid,N).
```

Wichtige Aspekte von Erlang:

- einfache Verarbeitung von Nachrichten durch Pattern matching
- kompakte Definition von Funktionen
- massive Codereduktion (auf 10% - 20%)
- **Codeaustausch** im laufenden(!) Betrieb möglich:
  Ersetze dazu im obigen `echo`-Programm den rekursiven Aufruf "`loop()`" durch "`echo:loop()`". Effekt:
  - benutze im rekursiven Aufruf die aktuelle Version der Funktion `loop()` im Modul `echo`
  - falls das Modul `echo` verändert und neu compiliert wird, so wird beim nächsten Aufruf die neue veränderte Version benutzt.
- Übergang zu verteilter Programmierung einfach:
  starte Prozesse auf anderen „Knoten" (Rechnern), wobei am Code der Kommunikation nichts geändert werden muss.

8 Ausblick

Wir haben in dieser Vorlesung viele verschieden Sprachkonzepte und -konstrukte kennengelernt. Allerdings ist die Entwicklung der Programmiersprachen so umfangreich, dass natürlich nicht alle Aspekte behandelt werden konnten. Aus diesem Grund wollen wir am Ende noch einmal kurz beleuchten, was *nicht behandelt* werden konnte:

Skriptsprachen: (Perl, Tcl, Awk, Bourne-Shell, PHP, JavaScript, Ruby,...)

- Sprachen zur einfachen und schnellen Kombination existierender Software-Komponenten
- meistens: ungetypt, interpretiert
- schlecht geeignet für große Softwareentwicklung
- viele Konzepte aus anderen Sprachen, z.B. JavaScript: Funktionen als Objekte, anonyme Funktionen

Visuelle Sprachen: 2D-Sprachen zur Programmierung

- ähnlich wie Skriptsprachen, einfacher Zusammenbau von Programmen aus existierenden Komponenten
- graphische Notation
- häufig nur für spezielle Anwendungen (z.B. Entwurf von Benutzerschnittstellen)

Anwendungsspezifische Sprachen:

- Struktur ähnlich wie bei existierenden Programmiersprachen, aber spezielle Konstrukte für bestimmte Anwendungen
- häufig auch erreichbar durch anwendungsspezifische Module/Bibliotheken in existierenden Programmiersprachen

Literaturverzeichnis

[1] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[2] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] E.W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[5] G. Huet and J.-J. Levy. Call by need computations in non-ambiguous linear term rewriting systems. Rapport de recherche no. 359, INRIA, 1979.

[6] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[7] J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

[8] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[9] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[10] M. Odersky and P. Wadler. Pizza into java: Translating theory into practice. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 146–159, 1997.

[11] G.D. Plotkin. A structural approach to operational semantics. Technical report daimi fn-19, Aarhus University, 1981.

[12] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[13] V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.