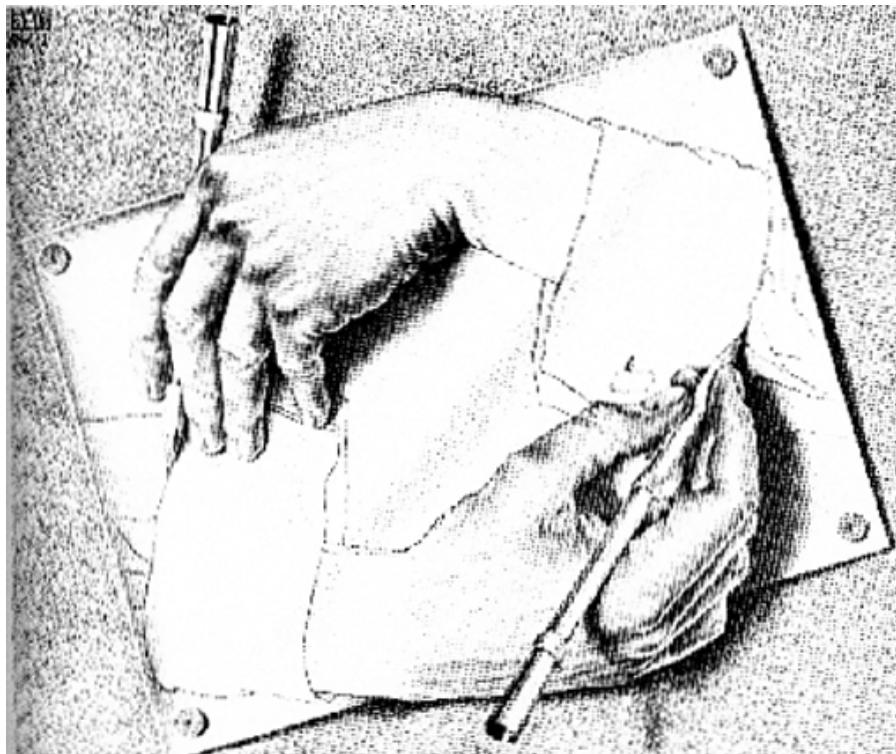
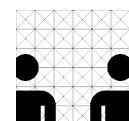


Softwareentwicklung III

Leonie Dreschler-Fischer
Prüfungsunterlagen Band 1:
Grundlagen



Universität Hamburg
Fachbereich Informatik
WS 2019/2020



Inhaltsverzeichnis

Band I: Grundlagen	1
I Grundlagen der funktionalen Programmierung	7
1 Einführung	7
1.1 Grundbegriffe der Programmierung	9
1.2 Programmiersprachen für die funktionale Programmierung	25
2 Organisatorisches	30
2.1 Kommentiertes Literaturverzeichnis	30
2.2 Modulprüfung und Übungen	37
3 Die Arbeitsumgebung	39
3.1 Das Racket-System	39
3.2 Symbolische Ausdrücke	45
II Funktionale Abstraktion	49
4 Funktionale Abstraktion	49
4.1 Namensgebung	49
4.2 Der Lambda-Abstraktor	56
5 Variablenkopus und closures	63
5.1 Freie und gebundene Variable	63
5.2 Fallunterscheidungen	64
5.3 Umgebungen und closures	68
6 Zeichen und Symbole	74
6.1 Zeichen	74
6.2 Symbole in Racket	79
6.3 Quotierung	80
III Datenabstraktion in Racket	86
7 Datenabstraktion	86
7.1 Grundlagen der Typsysteme	86

7.2 Polymorphie	92
8 Elementare Datentypen	94
8.1 Zahlen	94
8.2 Wahrheitswerte	99
8.3 Der Datentyp „char“	108
9 Listen	112
9.1 Grundoperationen auf Listen	113
9.2 Listen als rekursive Datenstrukturen	120
9.3 Listen mit Symbolen	122
9.3.1 Zugriff auf Listenelemente	123
9.3.2 Assoziationslisten	124
9.3.3 Rekursiver Abstieg zur Sprachgenerierung	125
IV Repräsentation der Datenstrukturen	134
10 Funktionale Datenabstraktion	134
11 Repräsentation und Gleichheit	136
11.1 Cons-Zellen	136
11.2 Gleichheitsprädikate für Listen	139
11.3 Rekursion über Listen	142
12 Weitere strukturierte Datentypen	145
12.1 Zeichenketten	145
12.2 Vektoren	152
12.3 Verbunde (structures)	155
12.4 Mengen	158
12.5 Streuspeicherung, Hashtables	158
12.6 Sequenzen	158
12.7 Generatoren	158
12.8 Bilder	158
V Semantik und Korrektheit	159
13 Semantik	159
13.1 Programmiersprachen: Grundbegriffe	160
13.2 Wertesemantik und Reduktionsmodelle	168
13.3 Die operationale Semantik von Racket	172

13.3.1 Die Auswertung funktionaler Ausdrücke	172
13.3.2 Auswertung von Special Form Expressions	174
14 Korrektheit und Spezifikation	184
14.1 Korrektheit	184
14.2 Spezifikation und automatischer Test	186
15 Rekursion und Induktion	210
15.1 Arten von Rekursion	211
15.2 Korrektheit rekursiver Funktionen	226
15.3 Rekursive Prozesse und Endrekursion	230
VI Der Entwurf von Funktionen	241
16 Funktionen höherer Ordnung	241
16.1 Funktionen als Werte	241
16.2 Beispiel: Funktion und Ableitung	244
16.3 Der Baukasten	246
17 Kontrollabstraktion	248
17.1 Funktionsapplikation	248
17.2 Funktionen höherer Ordnung für Listen	250
17.2.1 Abbilden (mapping)	250
17.2.2 Filtern	252
17.2.3 Falten (foldl, foldr)	254
17.3 Wiederholung und Iteration, Generatoren	257
18 Modularer Entwurf	265
18.1 Module	265
18.2 Formale Spezifikation und modularer Entwurf	269
VII Funktionale Abschlüsse	282
19 Funktionale Abschlüsse, Verknüpfung von Funktionen	282
19.1 Curry-Verfahren	282
19.2 Funktionskomposition	286
19.3 Funktionale Abschlüsse	287
19.4 Die Dylan-Funktionen	288
20 Callback functions: Closures als Ereignisroutinen	291

20.1 Graphische Bedienoberflächen	291
20.2 Das teachpack gui.rkt	295
20.3 Animation und Spiele	297
Band II: Algorithmen und Fallstudien	311
VIII Ausgewählte funktionale Algorithmen	313
21 Kombinatorische Probleme	313
21.1 Die Liste aller Unterlisten	314
21.2 Permutationen	316
21.3 Das Rucksackproblem	318
22 Rückzugsverfahren: Backtracking	320
22.1 Backtracking-Probleme	320
22.2 Das 8-Damen-Problem	321
22.3 Allgemeines backtracking-Schema	334
23 Nichtdeterminismus	344
23.1 Nichtdeterministische Suche: amb	344
23.2 Färben eines Graphen	349
IX Fallstudien	354
24 Musterabgleich (pattern matching)	355
24.1 Eliza: Ein regelbasierter Übersetzer	355
24.1.1 Pattern matching	361
24.1.2 Die Regelbasis und Dialogschleife	372
24.2 Kontrollabstraktion und Werkzeuge	391
24.3 STUDENT: Algebraische Probleme	397
25 Means-Ends-Analyse: GPS	425
25.1 Der „General Problem Solver“: GPS	425
25.2 GPS Version 2	452
26 GPS Anwendungen	468
26.1 Beispiel: Monkey and Banana	468
26.2 Beispiel: Pfade im Labyrinth	472

26.3 Beispiel: Klötzenwelt	478
X Objekte und generische Funktionen	487
27 CLOS: Objekte und generische Funktionen	487
27.1 CLOS: Klassen, generische Funktionen	487
27.2 Mehrfachvererbung und Methodenkombination	503
27.3 Ergänzungsmethoden	513
28 Entwurf eines ereignisorientierten Simulationssystems	519
28.1 Ereignisorientierte Simulation	519
28.1.1 Der Simulator	519
28.1.2 Statistische Modelle	522
28.2 Systementwurf	530
28.2.1 Das Basissystem	530
28.2.2 Das Bediensystem	547
28.3 Eine Anwendung	554
28.3.1 Mensa-Szenario	554
28.3.2 Spezialisierung von erzeugten Objekten	557
29 Objektorientierte Verarbeitungsmodelle	563
29.1 Nachrichten vs. Generics	563
29.2 Delegation	567
29.3 Backtracking mittels Delegation	568
30 Das Simulationssystem: Der Sourcecode	573
30.1 Modulstruktur	573
30.2 Das Basissystem	575
30.3 Das Anwendungssystem: Bediensystem	616
30.4 Das Mensa-Szenario	635
XI Metaprogrammierung und Programmierstile	640
31 Metaprogrammierung und Spracherweiterung	640
31.1 Programmieren mit Macros	640
31.1.1 Spracherweiterung durch Macros	640
31.1.2 Imperative Kontrollstrukturen in Racket	645
31.1.3 Macros zur Effizienzsteigerung	646
31.2 Fallstricke der Macroprogrammierung	647
31.2.1 Mehrfachauswertung	648

31.2.2 Seiteneffekte	648
31.2.3 Lexikalischer Kontext	650
32 Stromorientierte Programmierung	652
32.1 Verzögerte Auswertung	652
32.2 Ströme	652
33 Programmieren mit Zuständen	662
33.1 Caching und Memoization	662
33.2 Empfehlungen	668
33.3 Satire: Man mordet nicht nach Sprungbefehl	669
XII Relationale Programmierung	672
34 Relationale Programmierung	672
34.1 Der relationale Programmierstil	672
34.2 Die Datenbasis	677
34.3 Unifikation und Suche	681
35 Prolog als Datenbanksprache	685
35.1 Prolog als relationale Datenbank	685
35.2 Prolog als deduktive Datenbank	688
36 Prolog als Inferenzmaschine	695
36.1 Das Zebra-Rätsel	695
36.2 Severus Snapes Rätsel	700
36.3 Funktionale Sprachelemente in Prolog	705
37 Ein Prologinterpreter in Racket	708
37.1 Die Unifikation: unify.ss	708
37.2 Die Datenbasis: prologDB.ss	710
37.3 Backtracking und Interaktion: prolog.ss	715
37.4 Das Prolog-in-Scheme-Paket: prologInScheme.ss	720
XIII Zusammenfassung	723
38 Zusammenfassung und Ausblick	723
38.1 Zusammenfassung	723

38.2 Vorbereitung auf die Klausur	729
XIV Anhang	732
A Das Modul SE III: (aus dem Akkreditierungsantrag)	732
A.1 Motivation und Stellung im Gesamtprogramm	732
A.2 Lernziele und Kompetenzen	733
B Anmerkungen zu Abstraktion und Begriffsbildung	735
B.1 Begriffsbildung	735
B.2 Abstraktionsverfahren	736
C Implementation der Ströme in Racket	741
D Kurzreferenz: Racket	745
D.1 Funktionslexikon	745
D.2 CLOS	747
D.3 Prolog-in-Scheme-Lexikon	748
D.4 Unterschiede zwischen den Schemadialekten, DrScheme, Version 4.1.1	750
E Glossar und Index	757
Glossar	757
Allgemeiner Index	769
Literatur	770

Vorwort: In diesem Modul werden Sie ein weiteres Verarbeitungsmodell und den dazugehörigen Programmierstil kennenlernen — die applikative (oder funktionale) Programmierung. Es gibt im wesentlichen zwei Familien von Programmiersprachen, die für diesen Programmierstil entworfen wurden, die Lisp-Familie und die Haskell-Familie. Da wir nur ein Semester Zeit haben, können wir uns in dieser Vorlesung nicht mit beiden Sprachfamilien auseinandersetzen, sondern müssen uns leider auf eine davon beschränken. Sie werden in den Übungen zur Vorlesung die Programmiersprache *Racket* kennenlernen, einen typischen Vertreter der Lisp-Familie. Racket ist ein neuer Dialekt der Programmiersprache Scheme — eine orthogonal entworfenen Untergruppe von Common Lisp, die sich sehr gut als Ausbildungssprache eignet, da sie schnell zu erlernen ist, aber dennoch alle wichtigen Konzepte der Lisp-Programmierung unterstützt.

Eine Entwicklungsumgebung für die Programmiersprache Racket, das DrRacket-System, können Sie sich von der DrRacket-Homepage <http://racket-lang.org/>.

Diese Prüfungsunterlagen sind eine Materialsammlung, die Ihnen dabei helfen soll, die Fülle an Stoff, der in diesem Modul behandelt wird, zu strukturieren und die Übersicht zu behalten. Sie finden hier eine Übersicht und Definitionen zu den wichtigsten Fachbegriffen und Konzepten, die in der Vorlesung eingeführt werden, Verweise auf die einschlägige Literatur, Querverweise zu verwandten Themen in anderen Vorlesungen, Kopien der Vorlesungsfolien, viele Programmbeispiele, sowie Anhänge zu den verwendeten Programmiersprachen.

Nicht alle Programmbeispiele werden wir in der Vorlesung ausführlich besprechen. Teilweise sind sie als Vorlage für Ihre Übungsaufgaben gedacht. Wir haben für Sie eine Reihe von klassischen Beispielen der funktionalen Programmierung aus Standard-Lehrbüchern ([Norvig, 1992](#); [Bird and Wadler, 1988](#)) nach Scheme übertragen, so daß Sie nicht auch noch Miranda und Common Lisp lernen müssen.

Eins der besten Bücher, das je über die funktionale Programmierung geschrieben wurde, basiert glücklicherweise auf Scheme, dem Vorgänger von Racket. ([Abelson et al., 1996](#)). Die ersten vier Kapitel decken einen großen Teil des Stoff ab, den wir hier in der Vorlesung behandeln. Schauen Sie aber zum Vergleich auch einmal in die Einführung in die funktionale Programmierung von [Bird and Wadler, 1988](#) hinein. Und natürlich benötigen Sie ein Handbuch für die Sprache Racket, entweder den Programming Guide, den Sie im Internet finden <http://docs.racket-lang.org/guide/index.html>, oder eine der guten Spracheinführungen ([Felleisen et al., 2003](#)), ([Friedman and Felleisen, 1996](#)).

Zitate aus englischsprachigen Lehrbüchern wurden im Originaltext übernommen und nicht übersetzt. Ich möchte Sie nachdrücklich dazu ermuntern, englischsprachige Literatur in der ursprünglichen Fassung zu lesen und nicht auf die Übersetzungen zurückzugreifen. Spätestens im Master-Studium kommen Sie nicht mehr darum herum, mit englischer Literatur zu arbeiten. Außerdem vermeiden Sie so Probleme mit fehlerhaften oder lieblosen Übersetzungen; mache stilistisch sehr gute Bücher haben bei der Übersetzung ins Deutsche viel von ihrem Charme eingebüßt, indem beispielsweise die Wortspiele nicht mit übersetzt wurden.

Dieses ist kein Skript zur Vorlesung! Wir behandeln in diesem Modul einen sehr traditionellen Stoff der Informatik, zu dem es viele gute Lehrbücher gibt. Ich habe wenig Sinn darin gesehen, etwas nochmals neu zu schreiben, was andere schon sehr gut dargestellt haben. Insbesondere das Lehrbuch von [Felleisen et al., 2003](#) die Einführungen in die funktionale Programmierung von [Bird and Wadler, 1988](#) und [Abelson et al., 1996](#), und die Fallstudien von [Norvig, 1992](#) möchte ich Ihnen dringend ans Herz legen.

Der Leseausweis für die Informatikbibliothek ist für die Mitarbeit in diesem Modul mindestens so wichtig wie Ihr account auf den Rechenanlagen des Informatikrechenzentrums. In der Vorlesung und den Übungen und beim Durcharbeiten der empfohlenen Literatur lernen Sie Grundbegriffe und Konzepte der Informatik und des funktionalen Programmierens kennen. In den praktischen Übungen am Rechner werden Sie die Fertigkeiten erwerben, mit diesen Konzepten sicher umzugehen.

Von Informatikerinnen und Informatikern wird im Berufsleben die Fähigkeit und Bereitschaft zur Teamarbeit erwartet, und das lernt man nicht als Einzelgänger und einsamer Hacker zuhause. Schließen Sie sich daher unbedingt einer Lerngruppe an, in der Sie gemeinsam die Übungsaufgaben bearbeiten und den Vorlesungsstoff durchsprechen. Nutzen Sie die studentischen Arbeitsräume und Rechenanlagen des Fachbereichs und arbeiten Sie da, wo Sie Ihre Kommilitoninnen und Kommilitonen finden, bei uns im Informatikum.

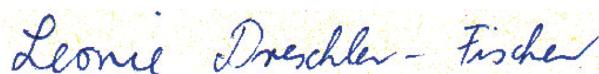
Wenn Sie Racket gemeistert haben, dann haben Sie eine gute Grundlage erworben, um in die Common Lisp Programmierung einzusteigen und die faszinierende Welt der KI-Softwaretechnik kennenzulernen. Wie Sie mit funktionaler Abstraktion den Entwurf großer Systeme meistern, können Sie bei ([Norvig, 1992](#)) nachlesen.

Achtung! Die Beispiele in diesen Unterlagen wurden alle in der Racket-Version „DrRacket, version 5.0“ erprobt. In neueren Versionen von DrRacket kann es zu Inkompatibilitäten kommen.

Sie haben sich mit dieser Vorlesung für ein vielseitiges und spannendes Teilgebiet der Programmierung entschieden. Mir hat die Vorbereitung dieser Unterlagen zur Vorlesung viel Spaß gemacht, und ich wünsche mir, daß sich meine Begeisterung für die funktionale und die relationale Programmierung auf Sie übertragen wird. Ich freue mich auf unsere Zusammenarbeit und wünsche mir, daß Sie mit Spaß, Engagement und Erfolg dabei sein werden und die Faszination der neuen Programmierstile kennenlernen und erleben können.

Viel Vergnügen mit curry, car und cdr!

Hamburg, im August 2018

A handwritten signature in blue ink that reads "Leonie Dreschler-Fischer". The signature is fluid and cursive, with some yellow decorative flourishes around the letters.

Ihre Leonie Dreschler-Fischer

Danksagung: In diese Vorlesung sind viele Anregungen aus früheren Vorlesungen zur funktionalen und relationalen Programmierung eingegangen. Ich möchte allen Kollegen danken, die mit Vorlesungsskripten und Diskussionen zum Entstehen dieser Vorlesung beigetragen haben, insbesondere Wolfgang Menzel, mit dem ich viele Jahre gemeinsam die Vorlesung „Praktische Informatik 1“ durchgeführt habe, und Benjamin Seppke, David Mosteller und Ralf Möller, mit denen ich Praktika zur funktionalen Programmierung veranstaltet habe. Eine wertvolle Hilfe waren mir auch die Diskussionen mit den Übungsleitern und die Grundstudiumsskripte der Kollegen Züllighoven, Görz und Habel zur Programmierung. Meinen ganz herzlichen Dank an alle!

Teil I

Grundlagen der funktionalen Programmierung

1 Einführung



Our design of this introductory computer-science subject reflects two major concerns: ([Abelson and Sussman, 1985, Vorwort](#))

First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute.

Second, we believe that the essential material to be addressed by a subject at this level is

- not the syntax of particular programming language constructs,
- nor clever algorithms for computing efficiently,
- nor even the mathematical analysis of algorithms and the foundations of computing,
- but rather the techniques used to control the intellectual complexity of large software systems.

- Underlying our approach to this subject is our conviction that “computer science” is not a science and *that its significance has nothing to do with computers*. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology — the study of the structure of knowledge from an imperative point of view taken by classical mathematical subjects.
 - **Mathematics** provides a framework for dealing precisely with notions of „*what is*“
 - **computation** provides a framework for dealing precisely with notions of „*how to*“.
- Abelson-Sussman-85*

Warum ein zweiter Programmierstil?

Wir wollen einen zweiten Programmierstil kennen lernen, dessen Konzeption sich grundlegend vom imperativen Programmierstil unterscheidet, und die Konsequenzen dieser Unterschiede diskutieren.

- Der funktionale Programmierstil ist mathematisch wohlfundiert,
- leicht zu erlernen,
- enthält softwaretechnisch sehr wichtige Konzepte,
- bietet nicht nur Compiler sondern auch Interpreter.

Warum Racket?

Racket ist eine Sprache der großen Lisp-Familie.

Wichtige Merkmale sind:

- Funktionale Abstraktion.
- Symbolverarbeitung.
- Dynamische Typisierung.
- Applikative, vorgezogene Auswertung.
- Metaprogrammierung, Erweiterbarkeit in der Sprache.
- Kein syntaktischer Unterschied zwischen Daten und Programm.

1.1 Grundbegriffe der Programmierung

Algorithmus

Im 1.Semester haben Sie in der Vorlesung „Softwareentwicklung I“ den Begriff **Algorithmus** als ein wesentliches Konzept der *imperativen Programmierung* kennen gelernt. Wir werden hier einen erweiterten Algorithmusbegriff einführen, der Berechnungsvorschriften in einer Vielzahl von Programmierstilen umfaßt.

Algorithmus

Definition: 1 (*Algorithmus*)

ein grundlegender Begriff der Informatik, benannt nach dem persischen Mathematiker Masa al Khowarizmi:

1. Ein Algorithmus ist ein eindeutig bestimmtes Verfahren unter Verwendung von Grundoperationen über primitiven gegebene Objekten. (**Schefe, 1985**)
2. **Alternative Definition:** Ein Algorithmus ist eine präzise, d.h. in einer festgelegten Sprache abgefaßte, endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer Verarbeitungsschritte. (**Bauer and Goos, 1982**)

Algorithmus vs. Programm

- Bauer und Goos 1982 betonen die *Beschreibung* eines Algorithmus, Schefe 1985 abstrahiert davon.
- Wir wollen den *Begriff des Algorithmus* von seiner *textuellen Beschreibung als Programm* unterscheiden:

Ein Programm muß

- präzise
- eindeutig
- endlich sein.

siehe (**Bauer and Goos, 1982**), (**Schefe, 1985**).

Fachbegriffe zu Algorithmen

Definition: 2

- Ein **Algorithmus** heißt *sequentiell*, wenn alle Schritte streng hintereinander ausgeführt werden.
- Ein Algorithmus heißt *parallel* oder *nebenläufig*, wenn einige seiner Schritte gleichzeitig, z.B. durch mehrere Ausführende, bearbeitet werden können.
- Ein Algorithmus heißt *deterministisch*, wenn alle Schritte vollständig geregelt sind, andernfalls heißt er nicht-deterministisch.
- Ein Algorithmus *terminiert*, wenn er nach endlich vielen Schritten abbricht.
- Ein A. ist *determiniert*, wenn er ein eindeutig bestimmtes Ergebnis liefert.

Anmerkung: Die Darstellung des Algorithmus-Begriffs hier wurde von Scheife, 1985 übernommen.

Verarbeitungsmodelle und Grundstile der Programmierung

Verarbeitungsmodell

Wenn wir einen **Algorithmus** formulieren, beziehen wir uns auf ein *Modell* desjenigen Automaten, der den Algorithmus ausführen wird, das *Verarbeitungsmodell*.

Definition: 3 (Verarbeitungsmodell)

Das **Verarbeitungsmodell** spezifiziert

1. die *Grundoperationen*, die ein Automat versteht und ausführen kann,
2. sowie das *Verhalten* des Automaten als Reaktion auf die Eingaben.

Imperative Programmierung: Folge von Anweisungen und Zustandsänderungen der Variablen im Speicher.

Von-Neumann Programmierung: Imperative Programmierung unter Bezug auf Anweisungsnummern.

Objektorientierte Programmierung: Manipulation und Zustandsänderungen von abstrakten Objekten.

Funktionale Programmierung: Auswertung funktionaler Ausdrücke.

Logische Programmierung: Auch relationale Programmierung genannt; es werden Beziehungen zwischen Fakten ausgewertet und die Erfüllbarkeit von logischen Klauseln überprüft.

Stromorientierte Programmierung: Datenströme, Kombinatoren und Filter, die die Datenströme verarbeiten.

☞ **Beachte:**

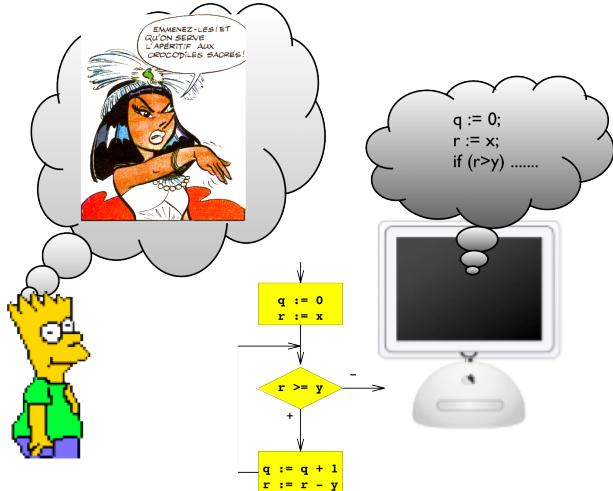
- Ein **Verarbeitungsmodell** *abstrahiert* von der konkreten, verarbeitenden Maschine.
- Der **Programmierstil** wird vom Verarbeitungsmodell bestimmt.
- **Programmiersprachen** sind jeweils für einen (oder mehrere) Programmierstile entworfen, enthalten aber in der Regel Schlupflöcher zu anderen Programmierstilen.

☞ **Beachte:**

- Der Fachbegriff „Programmierstil“ bezieht sich auf den *Modellierungsansatz* beim Programmentwurf,
- nicht auf die saubere Arbeitsmethodik, wie Namenswahl, Kommentare, sorgfältiges Testen usw.

Eine andere Sicht auf den Programmierstil: <http://de.wikipedia.org/wiki/Programmierstil>

Das imperative Verarbeitungsmodell



☞ Anmerkung: Beim *imperativen Programmieren* beschreiben wir, wie der Speicher für Daten bereitgestellt und organisiert werden soll, sowie den Prozeß, der die Daten, und damit den Zustand der Berechnung, verändert. Die imperativen Programmierung ist inzwischen durch die *imperative objekt-orientierte Programmierung* abgelöst worden, die eine moderne Form der imperativen Programmierung darstellt, bei der wir nicht mehr an die Zustandsänderung von Variablen, sondern eher an die Zustandsänderung von abstrakten *Objekten* denken.

Die von-Neumann Programmierung war die erste Form der (imperativen) Programmierung, da sie direkt auf die Strukturen der Hardware abgebildet werden konnte. Heute ist sie allerdings überholt. Mit Anweisungsnummern und expliziten Sprungbefehlen lassen sich total unstrukturierte Programme erzeugen, die auch mit Flußdiagrammen dargestellt nicht übersichtlicher werden, und dann humorvoll „Spaghetti-Code“ genannt werden. Unstrukturierte Programme sind fehleranfällig, schwer wartbar, und sie machen es den Compilern unnötig schwer, effizienten Code zu erzeugen. Diese Art der Programmierung hat nur noch ihre Berechtigung, wenn es wegen besonderer Umstände unerlässlich ist, maschinennah zu programmieren. Daß die von-Neumann-Programmierung, speziell eben die Sprunganweisung, das (Dijkstra, 1968) „GOTO“, schon in den siebziger Jahren umstritten war, zeigt eine Satire (siehe [Unterabschnitt 33.3, Seite 669](#)) aus der Computerwoche vom 11. Juni 1976.

Das imperative Verarbeitungsmodell

Merkmale

- Der Programmablauf führt zu einer Folge von *Zustandsänderungen* der Variablen oder Objekte des Programms.
- Die Zustandsänderungen werden durch Anweisungen oder Nachrichten bewirkt.
- Die Reihenfolge der Anweisungen wird durch *Kontrollstrukturen* (Schleifen, Verzweigungen, früher auch Sprünge) gesteuert.
- Korrektheitsbeweise über *Vor- und Nachbedingungen* der Anweisungen, sehr aufwendig.

Ein imperativer Multiplikationsalgorithmus

Beispiel: 4

1. Multipliziere die letzte Stelle der zweiten Zahl mit der ersten Zahl.
2. Multipliziere das Ergebnis aus 1. mit dem Stellenwert der benutzten Stelle.
3. Streiche diese, merke aber den Stellenwert der nunmehr letzten Stelle.
4. Gibt es bereits ein Ergebnis aus einer vorangegangenen Berechnung, so addiere dies zum Ergebnis des letzten Rechenschrittes und halte das Ergebnis fest, andernfalls halte das Ergebnis des Schrittes 2 fest.
5. Wenn keine Stellen der zweiten Zahl mehr vorhanden sind, liegt das Resultat mit dem Ergebnis aus dem zuletzt ausgeführten Schritt vor, andernfalls fahre bei 1. fort.

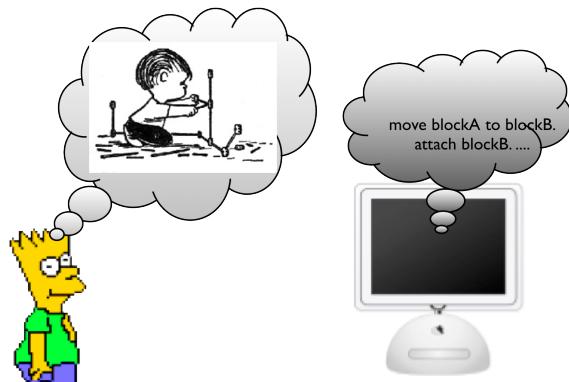
Ein typischer imperativer Algorithmus:

- Die Beschreibung ist *endlich*.
- Das Verfahren ist *allgemein*, es lässt sich auf verschiedene Zahlen, ja sogar Zahlensysteme anwenden.
- Es gibt elementare Operationen, Streichen einer Stelle, Addition zweier Zahlen.
- Überlegen Sie selbst: Ist der Algorithmus *präzise* und *eindeutig* beschrieben? d.h. haben alle Begriffe eine klare Bedeutung, z.B. *Stelle*, *Stellenwert*?

Das objektorientierte Verarbeitungsmodell

Das objektorientierte Verarbeitungsmodell gibt es als moderne Form des imperativen Verarbeitungsmodells und des funktionalen Verarbeitungsmodells. Bei diesem Verarbeitungsmodell können wir Repräsentationen der Objekte des Gegenstandsbereichs im rechnerinternen Modell so manipulieren, als hätten wir die echten physikalischen Objekte vor uns. Wir reden über Objekte und nicht über Daten. Sofern die Modellierung gut gelungen ist, sind die Programme wegen der starken Analogie zum Gegenstandsbereich sehr gut verständlich, wartbar und dank der Vererbung auch gut erweiterbar. Bei der imperativen objektorientierten Programmierung sind formale Korrektheitsbeweise allerdings genauso aufwendig, wie bei jedem anderen zustandsbehafteten Programm auch.

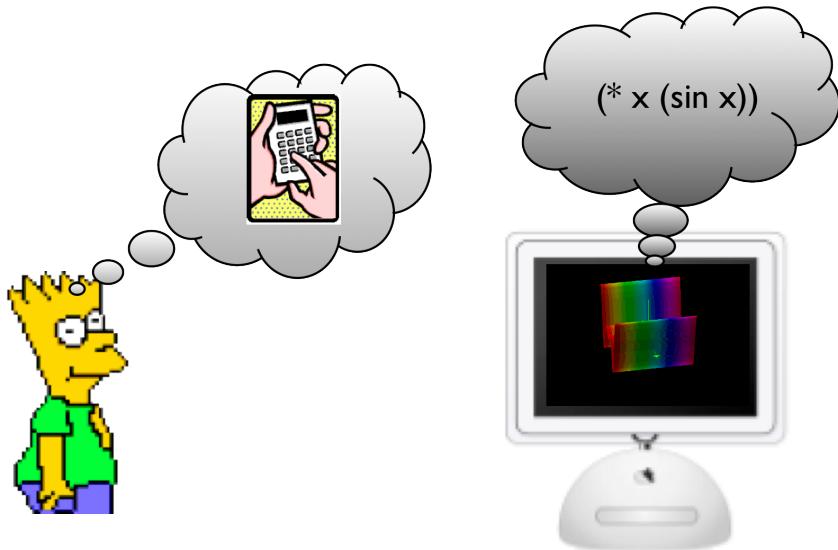
Das objektorientierte Verarbeitungsmodell



- Modellierung des Problembereichs über abstrakte Objekte
- Gut verständliche Programme
- Gute Erweiterbarkeit über Vererbung
- Für imperativen, objektorientierten Programme: Schwierige formale Korrektheitsbeweise wegen der zustandsbehafteten Programme.

Anmerkung: Wie die objektorientierte Programmierung mit der funktionalen Programmierung verbunden werden kann, werden wir im Kapitel **27** besprechen.

Das funktionale Verarbeitungsmodell



Das funktionale Verarbeitungsmodell

- Das funktionale (oder synonym applikative) Verarbeitungsmodell basiert auf der Definition von Funktionen und der Konstruktion von funktionalen Termen.
- Die Programme können meist sehr prägnant und kurz formuliert werden.
- Ein Vorteil dieses Verarbeitungsmodells ist, daß die Semantik der Programme formal definiert werden kann. Das ermöglicht es, Korrektheitsbeweise zu führen oder die Programme direkt aus formalen Spezifikationen zu entwickeln.
- Die Syntax funktionale Programmiersprachen ist einfach und leicht zu lernen.

Anmerkung: Der Name dieses Verarbeitungsmodells „applikativ“ bezieht sich darauf, daß Funktionen auf ihre Argumente angewendet werden. Es basiert auf der Definition von Funktionen und der Konstruktion von funktionalen Termen.

Ein funktionaler Multiplikationsalgorithmus

Beispiel: 5

Das Produkt zweier Zahlen ergibt sich als Summe der Produkte aus der ersten Zahl und den mit ihren Stellenwerten multiplizierten Stellen der zweiten Zahl.

Dieser Algorithmus ist

- parallel
- nicht-deterministisch
- terminierend
- determiniert
- funktional, rekursiv

Funktionale Programmiersprachen

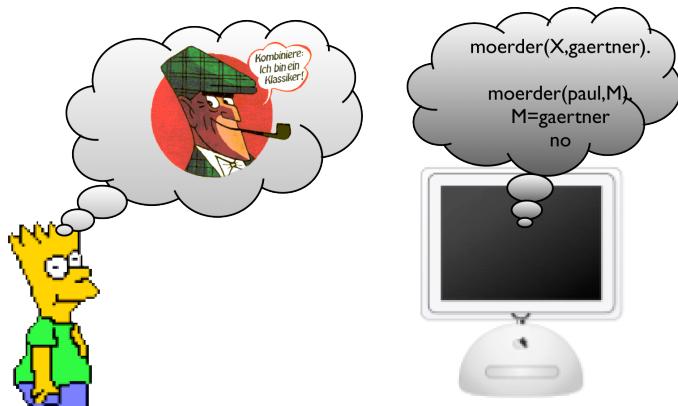
Programmierung

- Die Syntax ist extrem einfach und wenig redundant.
- Die denotationelle Semantik wird über das λ -Kalkül definiert. Die erste Implementation von Lisp, die John McCarthy mit seinen Studentinnen und Studenten durchgeführt hat, war eine direkte Implementation des λ -Kalküls (McCarthy, 1960).
- Die Ausdruckskraft der Sprachen ermöglicht den Entwurf sehr knapper Algorithmen für sehr schwierige Programmierprobleme.
- Die Syntax ist leicht zu lernen, aber die Pragmatik erfordert das Lernen der typischen Entwurfsmuster.

★ Anmerkung: Unerfahrene Programmierer werden allerdings aus Freude über die eleganten Formulierungsmöglichkeiten häufig zu sehr unverständlichen Programmkonstrukten verleitet, die bestechend aussehen, die aber ihre Schöpfer schon nach kurzer Zeit selbst nicht mehr verstehen können. ☺

Halten Sie sich an bewährte Entwurfsmuster, dann wird jeder Ihre Programme lesen können. ☺

Das gilt sinngemäß auch für die relationale Programmierung, die wir gleich besprechen werden!



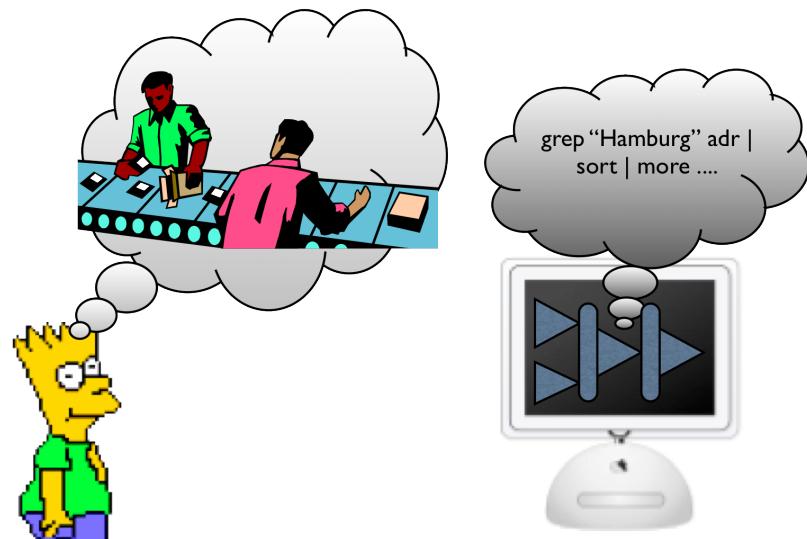
Das logische oder relationale Verarbeitungsmodell

Dieses Modell basiert auf der Vorstellung von einem Automaten, der bei einer gegebenen Menge von Fakten, Beziehungen zwischen diesen Fakten und einer gegebenen Menge von Schlußfolgerungsregeln durch Kombination dieser Daten nach strengen Regeln in einem logischen Kalkül neue Fakten herleiten und Anfragen beantworten kann. Berühmte Vertreter dieses Verarbeitungsmodells sind beispielsweise die Expertensysteme MYCIN und PROSPECTOR. MYCIN konnte bemerkenswert gut bakterielle Infektionen diagnostizieren und PROSPECTOR, ein Expertensystem zur Suche nach Bodenschätzten, hat auch solche gefunden. Wir werden dieses Verarbeitungsmodell gegen Semesterende näher besprechen, siehe Abschnitt 34 dieser Prüfungsunterlagen.

Merkmale des logischen / relationalen Verarbeitungsmodells

- Explizite Modellierung von Fakten und Beziehungen
- Vordefinierte Such- und Schlußfolgerungsmechanismen
- Einfache Syntax, leicht zu lernen
- Formal definierte Semantik
- Wie der funktionale Programmierstil sehr gut für komplexe Programmieraufgaben geeignet.

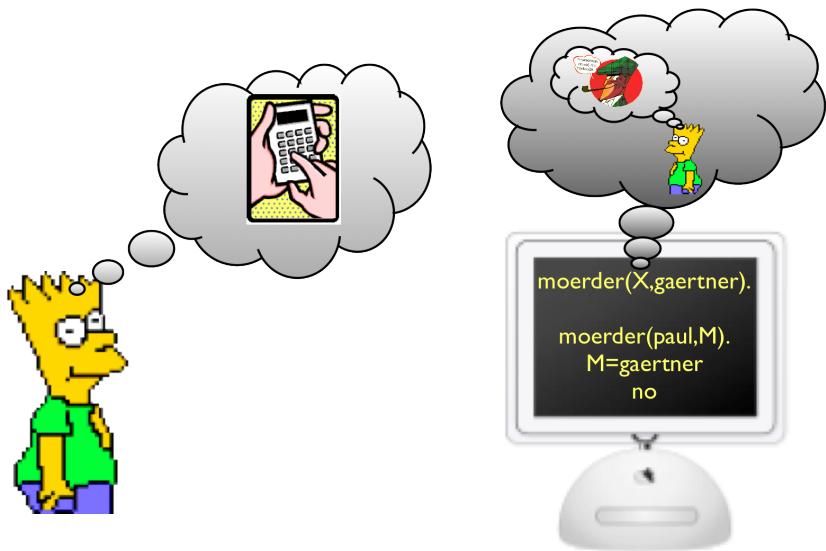
Das stromorientierte Verarbeitungsmodell



Merkmale des stromorientierten Verarbeitungsmodells

- Das stromorientierte Verarbeitungsmodell ist besonders gut geeignet für die fließbandartige Verarbeitung von (endlosen) Strömen gleichartiger Daten (Objekte).
- Programmkomponenten sind Quellen und Senken von Strömen, sowie Filter, Transformatoren und Kombinatoren.
- Beispiele: Bildverarbeitungssysteme, Unix-Shells; siehe Abschnitt:
32

Das Benutzermodell

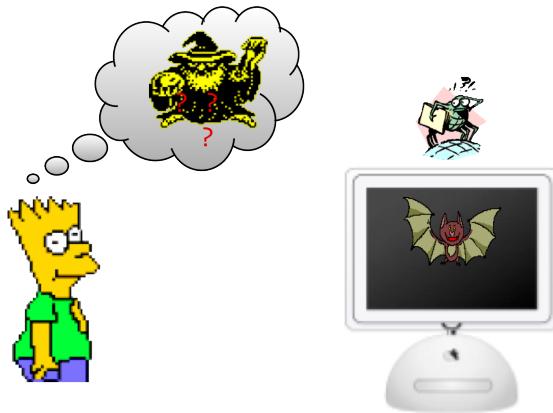


Verarbeitungsmodell vs. Benutzermodell

Im Hinblick auf die Mensch-Maschine-Kommunikation ist ein *Benutzermodell* das Gegenstück zum Verarbeitungsmodell: Beim Entwurf von interaktiver Software müssen wir darauf achten, daß wir klare Vorstellungen davon haben, wie die Benutzer mit dem Programm kommunizieren werden und wie sie die Ausgabe des Programms interpretieren. Ohne ein klares Benutzermodell wird leicht an den Bedürfnissen und Fähigkeiten der Benutzer vorbei produziert und unergonomische Software entwickelt, die durch unnötig komplizierte Interaktionsprozesse oder aus Benutzersicht unverständliches Verhalten die Anwender frustriert.

In dieser Vorlesung beschränken wir uns aber die Behandlung von Verarbeitungsmodellen und darauf, wie wir die unterschiedlichen Programmierstile softwaretechnisch einsetzen. Diese softwareergonomischen Aspekte der Programmierung werden in anderen Modulen behandelt .

Das mystische Verarbeitungsmodell



Das mystische Verarbeitungsmodell

Dieses Verarbeitungsmodell sollten Sie nicht von Ihrem Rechner haben, oder zumindest schnell wieder verlernen.

Auch bei diesem Modell sehen wir den Rechner als *black box* mit geregeltem E/A-Verhalten, aber im Unterschied zu den anderen Modellen verursachen uns die wunderbaren und (für uns) unerklärlichen und quasi magischen Fähigkeiten Ängste. Unbewußt personifizieren wir den Rechner und kommunizieren mit ihm auch emotional, wie mit einem Lebewesen. Wir verhalten uns so, als hätte der Rechner eigene Absichten und Ziele, und wir sind uns nicht sicher, ob wir ihm trauen können. Dieses kommt besonders bei unerwartetem Verhalten eines Rechners zum Ausdruck, dann wenn er sich aus unserer Sicht fehlerhaft, unkooperativ, wiederspenstig oder gar boshaft verhält. Wir fühlen uns dann wie Goethes Zauberlehrling: „Die Geister die ich rief, ich werd' sie nicht mehr los!“

Dabei ist gegen so leistungsfähige Systeme, deren Fähigkeiten geradezu magisch erscheinen, im Prinzip gar nichts einzuwenden. Wir arbeiten im Gegenteil gezielt darauf hin. Wir müssen nur sicherstellen, daß das Verhalten eines Systems mit einem wohldefinierten Verarbeitungsmodell übereinstimmt, das die Anwender und Anwenderinnen verstehen und beherrschen können. *Niemand darf für zu lange Zeit in die Rolle des Zauberlehrlings gedrängt werden, weder AnwenderInnen noch EntwicklerInnen!*

Die eine Ursache für ein nicht beherrschbares Verarbeitungsmodell ist schlechte oder schlecht dokumentierte, oder gar fehlerhafte Software, die alle, die damit arbeiten müssen, zur Verzweiflung treibt. Die Verwendung solcher Erzeugnisse ist und riskant führt zu neuen unsicheren Programmen.

Für den Umgang mit solchen gefährlichen „Produkten“ gibt es nur eine Regel — sagen Sie standhaft und entschieden „Nein!“ und arbeiten Sie mit darauf hin, daß diese vom Markt verschwinden. Als verantwortungsvolle zukünftige Informatiker oder Informatikerinnen sollten Sie niemals unsichere Software kaufen oder verwenden und schon gar nicht selber entwickeln (siehe dazu das Zitat von Hoare auf Seite [22](#)).

Die zweite, nicht weniger bedenkliche Ursache für ein mystisches Verarbeitungsmodell ist ein zu großes Vertrauen in die Fähigkeiten der Maschinen. Wir lassen uns durch die Präzision der Verarbeitung und durch die Eleganz formaler Notationen beeindrucken, die uns eine *Scheinobjektivität* suggerieren. Aber auch das ausgefeilteste formale System schützt einen nicht davor, darin großen Unsinn auszudrücken. Wir dürfen uns nicht von der maschinellen Präzision blenden lassen und niemals Resultate ungeprüft übernehmen. Das verpflichtet uns aber auch dazu, Software so zu entwerfen, daß sie validiert und die erzeugten Ergebnisse überprüft werden können. Auf keinen Fall dürfen wir es zulassen, daß unsere Programmsysteme so komplex werden, daß wir das Gefühl haben, sie selbst nicht mehr verstehen zu können und die Verantwortung für die Resultate auf den Computer abschieben (siehe [Weizenbaum, 1978](#)).



Wer die Ergebnisse unbesehen glaubt, sollte gar nicht erst anfangen, Informatik zu studieren!!!!!!

H.-H. Nagel 1972, im Skriptum zu Informatik I

Für Freunde utopischer Erzählungen: Ein besonderes Lesevergnügen zum Thema „Computer mit wirklich magischen Fähigkeiten“ sind die „Robotermärchen“ von Stanislav Lem.

Scheinobjektivität von Programmen

Ein schönes Beispiel für die Scheinobjektivität von Programmen ist das große Vertrauen in Computerhoroskope:

Ich bin Menschen begegnet, die nie ein von Hand erstelltes Horoskop glauben würden, aber den maschinell gerechneten Horoskopen vertrauen, denn diese werden doch von einer Maschine erstellt, die keine Fehler macht.

Any sufficiently advanced technology is undistinguishable from magic.

Arthur C. Clarke



Kennzeichen des mystischen Verarbeitungsmodells

- Unvorhersehbares Systemverhalten,
- überraschende verborgene Zustände,
- unreproduzierbare Fehler,
- wenig hilfreiche Fehlermeldungen,
- verwirrende, unergonomische Bedienoberflächen,
- unverständliche, konfuse Handbücher.



Solche Systeme sind *nicht beherrschbar* und sind *gefährlich!*

- Für den Umgang mit solchen gefährlichen „Produkten“ gibt es nur eine Regel — sagen Sie standhaft und entschieden „Nein!“ und arbeiten Sie mit darauf hin, daß diese vom Markt verschwinden.
- Als verantwortungsvolle zukünftige Informatikerinnen oder Informatiker sollten Sie niemals unsichere Software kaufen oder verwenden und schon gar nicht selbst entwickeln.

Eine *unzuverlässige* Programmiersprache, aus der *unzuverlässige* Programme hervorgehen, bedeutet eine sehr viel größere Gefahr für unsere Gesellschaft als unsichere Automobile, giftige Pestizide und Reaktorunfälle in Kernkraftwerken.

C. A. R. Hoare

(zitiert nach ([Schefe, 1985](#)))

★ Anmerkung: Das Zitat von Hoare bezieht sich auf die Programmiersprache Ada, die sehr umstritten ist, da lange bezweifelt wurde, ob es je möglich sein würde, korrekte Compiler für diese Sprache zu entwickeln. Auch galt die Sprache als zu komplex und schwer erlernbar. Abgesehen von diesen Bedenken war die Sprache vielen wegen ihres militärischen Ursprungs unsympathisch.

Welches ist der beste Programmierstil?

Diese Frage lässt sich nicht allgemeingültig beantworten.

- Sie sollten sich stets für einen Programmierstil entscheiden, in dem Sie die passenden Grundoperationen und Modellierungsmöglichkeiten für den Problembereich finden.
- Bei der Programmierung ist es die wichtigste Aufgabe, zunächst das Anwendungsproblem zu analysieren und die Grundoperationen und Abstraktionen festzulegen, mit denen der Algorithmus am treffendsten beschrieben werden kann.
- Und wenn keine der zur Verfügung stehenden Programmiersprachen geeignet ist?

Anwendungsspezifische Verarbeitungsmodelle

- Wenn keine der zur Verfügung stehenden Programmiersprachen die Modellierungsmöglichkeiten bietet, die Sie benötigen, dann sollten Sie erwägen, diese selbst zu schaffen:
 - Ein eigenes, problemadäquates *Verarbeitungsmodell* definieren,
 - Eine passende *Programmiersprache* definieren,
 - Eine *virtuelle Maschine* implementieren, die dieses Verarbeitungsmodell realisiert.
- Wir werden in diesem Semester einige Beispiele kennenlernen, wie wir die Programmiersprache Racket um neue Verarbeitungsmodelle erweitern.

Ausdruckskraft einer Programmiersprache

Definition: 6 (berechnungsuniversell)

Eine Programmiersprache ist *berechnungsuniversell*, wenn in ihr jeder intuitive Algorithmus formuliert werden kann.

General Purpose Programmiersprachen, wie Lisp, Prolog, Java, C usw. sind berechnungsuniverselle Programmiersprachen, aber es gibt viele Spezialsprachen.

Spezialsprachen, wie Datenbankanfragesprachen, Modellierungssprachen sind in der Regel nicht berechnungsuniversell.

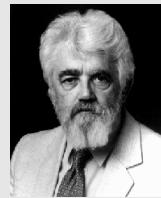
Nachweis: Beweisen Sie, daß sich in der Programmiersprache eine universelle Turingmaschine implementieren läßt.

1.2 Programmiersprachen für die funktionale Programmierung

- Es gibt zwei große Familien von funktionalen Programmiersprachen:
 1. Die *Lisp-Familie* (Symbole, dynamische Typisierung, hybrid), zu der Racket, Scheme und Common Lisp gehören,
 2. sowie die Familie der *streng-getypten*, referentiell transparenten Sprachen, mit Vertretern wie Haskell und Miranda.
- In dieser Vorlesung werden wir die applikative oder funktionale Programmierung vor allem aus der Sicht der Lisp-Familie darstellen, deren Stärke die Fähigkeit zur *Symbolverarbeitung* ist.

Entstehung von Lisp

- ▶ Die Arbeiten an Lisp wurden 1958 von *John McCarthy* und seinen Studentinnen und Studenten begonnen. Die erste Implementation war eine direkte Implementation des λ -Kalküls von *McCarthy 1960*.
- ▶ Nach FORTRAN ist Lisp die älteste Programmiersprache, die noch benutzt wird. Lisp ist immer noch eine der führenden Programmiersprachen, was neue softwaretechnische Konzepte betrifft.



John McCarthy

◀ ▶ ⟲ ⟳ ⟴ ⟵ ⟷ ⟸ ⟹ ⟻ ⟺ ⟻ ⟹ ⟷ ⟸ ⟷ ⟷ ⟷ ⟷

Der Name „Lisp“

- Der Name „Lisp“ steht für eine ganze Familie von Programmiersprachen. Wir beziehen uns hier auf Racket.
- Das Acronym „Lisp“ steht für *List processing*.
- Manche sagen, es steht für: *lots of silly insidious parentheses*. :-)

McCarthy, 1960.

Die wichtigsten Vertreter der Lisp-Familie

ANSI-Common Lisp: Ein genormter objektorientierter Lisp-Standard, der den funktionalen und objektorientierten Programmierstil unterstützt. Es gibt viele softwaretechnisch interessante Erweiterungspakete: CLIM (Common Lisp Interface Management System), MOP (Meta Object Protocol) usw.

Scheme: Ein orthogonal entworfenes Kernsystem für Lisp, speziell für Ausbildungszwecke, aber auch gut als *scripting language* geeignet. Für die Übungen werden wir den Dialekt *Racket* verwenden.

★ **Anmerkung:** Racket als Sprache für scripts: Viele Programmierprobleme lassen sich sehr elegant lösen, indem Folgen bestehender Programme und Systemkommandos als sogenannte „scripts“ kombiniert werden. Ein solches „script“ kann dann wie ein Systemkommando verwendet werden. Hierfür geeignete Programmiersprachen heißen Skriptsprachen (*scripting language*). Eine solche Sprache, aber sehr elementar, ist die Sprache der C-shell. Sehr komfortabel und beliebt ist die Sprache Perl. Auch Racket hat in den letzten Jahren große Beliebtheit als *scripting language* gefunden. Leider können wir aus Zeitgründen hier nicht darauf eingehen, wie Sie von Racket aus Systemkommandos aufrufen oder andere Programme starten können, aber denken Sie an Ihre Racket-Kenntnisse zurück, wenn Sie eine geeignete *scripting language* suchen.

Relevanz

- Lisp ist eine weit verbreitete, relevante Programmiersprache, insbesondere auch für KI¹-Programmierung. Viele moderne softwaretechnische Prinzipien wurden und werden zuerst in Lisp oder Prolog erprobt.

If you're going to learn a language, it might as well be one with a growing literature, rather than a dead tongue. (Norvig, 1992)

¹Künstliche Intelligenz

Flexibilität von Lisp

- In Lisp ist es sehr einfach, neue Generalisierungen und die dazugehörigen Kontrollstrukturen einzuführen.
- In Lisp ist es einfach, und gängige Praxis, neue Programmiersprachen zu definieren, die einen neuen Programmierstil möglich machen.
- Lisp-Programme sind auch Lisp-Daten und können wie Daten zur Laufzeit erzeugt oder verarbeitet werden.

Einfache und schnelle Programmierung

In Lisp ist es (wie in Prolog) sehr einfach, schnell lauffähige Prototypen zu entwickeln:

- Lisp ist interaktiv.
- Lisp Programme sind kurz und bündig (concise). Sie sind nicht überladen mit low-level Details.
- Common Lisp bietet einen Baukasten von über 700 vordefinierten Funktionen.
- Zu einem Lisp System gehört eine sehr gut ausgestattete Entwicklungsumgebung: Integrierte Editoren, inkrementelle Compiler, Debugger usw.

Ein Lisp-Mythos

Ein weit verbreiteter Mythos ist, daß Lisp eine *special purpose Programmiersprache* für KI-Programmierung sei, während andere Programmiersprachen, wie Java oder C als *general purpose Programmiersprachen* angesehen werden.

- **Das Gegenteil ist richtig!** Lisp ist eine der wenigen Sprachen, die wirklich den Namen *general purpose language* verdienen,
- während Java oder C *special purpose languages* für Probleme der traditionellen EDV sind: die Sprachen bieten Formulierungsmöglichkeiten für Arithmetik, logische Ausdrücke und Datenabstraktion, aber nur sehr eingeschränkte Möglichkeiten zur funktionalen Abstraktion oder gar zur Definition neuer Kontrollstrukturen und Verarbeitungsmodelle.

Lisp und KI-Programmierung

- Lisp wird gerade deshalb zur KI-Programmierung eingesetzt, weil die Beschränkungen dieser angeblichen *general purpose* Sprachen es nicht zulassen, neue Abstraktionen, wie sie in der KI entwickelt werden, angemessen zu formulieren und neue Verarbeitungsmodelle zu implementieren.
- Lisp ist eine Standardsprache für *exploratives Programmieren* und rapid prototyping, nicht nur für KI-Anwendungen.

Lisp als general purpose Sprache

In Lisp sind viele Verarbeitungsmodelle und Programmierstile eingebettet oder können mit wenigen Erweiterungen realisiert werden:

- Imperativer und objektorientierter Programmierstil,
- funktionaler Programmierstil,
- strom-orientierter Programmierstil,
- logischer oder relationaler Programmierstil.

When new styles of programming were invented, other languages died out; Lisp simply incorporated the new styles by defining some new macros.
Peter Norvig

Das Chamäleon Lisp

- „Lisp's flexibility allows it to adapt as programming styles change, but more importantly, Lisp can adapt to your particular programming problem.
- In other languages you fit your problem to the language; with Lisp you extend the language to fit your problem.“
(Norvig, 1992)





Lisp ist eine effiziente Programmiersprache

Ein Vorurteil gegen Lisp ist, daß Lisp-Programme ineffizient wären:

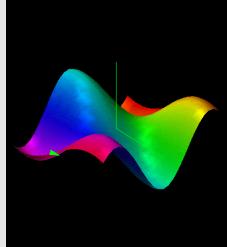
Das Gegenteil ist wahr:

- Lisp-Programme können wegen der sehr guten Programmierumgebungen sehr effizient entwickelt werden.
- Die Compiler optimieren endrekursive Funktionen (siehe Abschnitt: **15.3**).
- Für die klassischen EDV-Probleme, wie Rechnen, Organisieren, Suchen, gibt es sehr effiziente Basisfunktionen, so daß man Lisp ohne Effizienzverlust getrost für solche Aufgaben einsetzen kann, die man sonst in C, Java oder gar Maschinensprache lösen würde.
- Es zwar gibt Speicher- und Rechenzeit-aufwendige Lisp-Systeme, aber das liegt an der Komplexität der Probleme (oder an unerfahrenen Programmierern) und nicht an Lisp .
- Für besonders schwierige Probleme ist Lisp eben oftmals die beste Sprache, um diese Aufgaben überhaupt bewältigen zu können.

2 Organisatorisches

2.1 Kommentiertes Literaturverzeichnis

Kommentiertes Literaturverzeichnis



- 1 Einführung
- 2 Organisatorisches
 - Kommentiertes Literaturverzeichnis
 - Modulprüfung und Übungen
- 3 Die Arbeitsumgebung

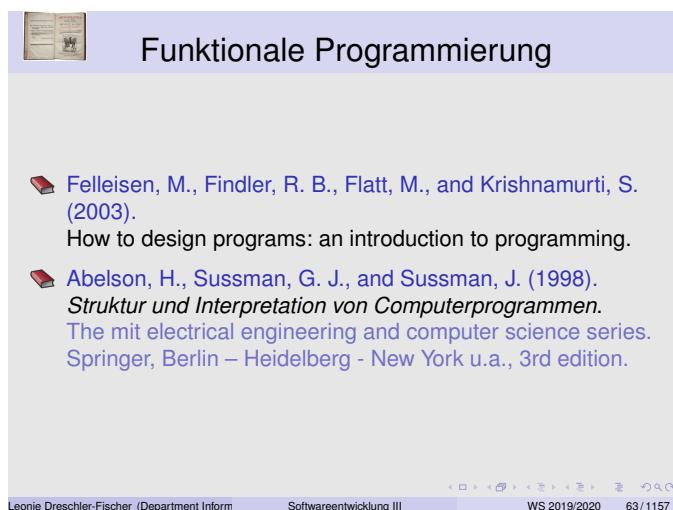
Navigation icons: back, forward, search, etc.

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 62/1157

Siehe dazu auch das Virtuelle Bücherregal zur Vorlesung:
<http://www.informatik.uni-hamburg.de/bib/vib/index.shtml.de>

Literatur zur funktionalen Programmierung

Das Buch „How to Design Programs“ von Felleisen et al., 2003 gibt eine gute Einführung in den funktionalen Programmierstil und insbesondere die Programmierung mit Racket. Es ist als erste Begegnung mit der Programmierung überhaupt gedacht und richtet sich daher insbesondere an Anfänger. Wir werden dennoch dieses Standardwerk in weiten Teilen für die Vorlesung verwenden, auch wenn Sie keine Anfänger mehr sind, da das Buch sehr gut verständlich und auf die in der Vorlesung verwendete Programmierumgebung DrRacket abgestimmt ist. Die dritte Auflage von Felleisen et al., 2003 können Sie online lesen: <http://www.htdp.org/2003-09-26/Book/>.



„How to Design Programs“ ist eine Weiterentwicklung des Klassikers von Abelson et al., 1998, aus dem wir ebenfalls einige Beispiele für die Vorlesung verwenden werden. Der Abelson-Sussman ist ein echtes „Kultbuch“ der Informatik, das jede Informatikerin oder jeder Informatiker gelesen haben sollte, gleichgültig, ob man Scheme oder Racket nun mag oder nicht. In diesem Buch finden sie die Grundkonzepte applikativen (oder funktionalen) Programmieren, so wie wir sie in dieser Vorlesung behandeln werden. Schlagen Sie bei Abelson et al., 1998 nach, wenn Sie Fragen zur Auswertung funktionaler Ausdrücke, zur Rekursion, zum Umgebungsmodell der Auswertung haben.

Achten Sie bitte bei der Suche in der Bibliothek darauf, daß Sie die passende Auflage entleihen: für die englische Ausgabe([Abelson et al., 1996](#)) die zweite Auflage und für die deutsche Ausgabe ([Abelson et al., 1998](#)) die dritte Auflage, sonst fehlt das für die Vorlesung relevante Kapitel über nicht-deterministische Algorithmen.

Es gibt zwei große Familien von funktionalen Programmiersprachen: Die Lisp-Familie, zu der Racket, Scheme und Common Lisp gehören, sowie die Familie der streng-getypten, referentiell transparenten Sprachen, mit Vertretern wie Haskell und Miranda. In der Vorlesung werden wir die applikative oder funktionale Programmierung vor allem aus der Sicht der Lisp-Familie darstellen, deren Stärke die Fähigkeit zur Symbolverarbeitung ist.

The screenshot shows a presentation slide with a light blue header containing the title 'Funktionale Programmierung: Formale Grundlagen'. Below the title is a small image of an open book. The main content area contains two entries, each with a small book icon and a reference:

- Bird, R. and Wadler, P. (1988).
Introduction to Functional Programming.
Prentice-Hall, Englewood Cliffs, New Jersey.
- Lippe, Wolfram-Manfred (2009).
Funktionale und Applikative Programmierung.
Springer-Verlag, Berlin – Heidelberg.

At the bottom of the slide, there is a navigation bar with icons for back, forward, search, and other presentation controls. The footer contains the text 'Leonie Dreschler-Fischer (Department Inform Softwareentwicklung III)' and 'WS 2019/2020 64/1157'.

„Introduction to Functional Programming“ von [Bird and Wadler, 1988](#) ist eine Einführung in die funktionale Programmierung aus der Sicht der streng typisierten Programmiersprachen am Beispiel der Programmierung in Miranda. Für diese Vorlesung wurden eine Reihe von Beispielen aus dem [Bird and Wadler, 1988](#) in die Programmiersprache Racket übertragen. In dieser Vorlesung werde ich mich gelegentlich auf den [Bird and Wadler, 1988](#) beziehen, aber nicht so oft, daß sie das Buch unbedingt kaufen müßten. Es ist zudem in großer Anzahl in der Bibliothek vorhanden, leider nur in der deutschen Übersetzung.

Das Buch von [Lippe, 2009](#) gibt eine sehr ausführliche Übersicht zu funktionalen Programmiersprachen und den formalen Grundlagen der funktionalen Programmierung, insbesondere zum Lambda-Kalkül.

Objektorientierte funktionale Programmierung

The screenshot shows a presentation slide with a light blue header bar containing the title 'OO-funktionale Programmierung'. Below the header, there is a list of two books:

- Keene, S. (1989).
Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.
Addison-Wesley.
- Graham, P. (1996).
Ansi Common Lisp.
Prentice-Hall, Englewood Cliffs, New Jersey, London.

At the bottom of the slide, there is a navigation bar with icons for back, forward, search, and other presentation controls. The footer contains the names of the author and department, the course name, the semester, and the page number.

Leonie Dreschler-Fischer (Department Inform Softwareentwicklung III WS 2019/2020 825/1157

In dieser Vorlesung werden wir auch ausführlich darüber sprechen, wie Objekte und Methoden im funktionalen Programmierstil über generische Funktionen und funktionale Abschlüsse modelliert werden können. Wir werden dazu das Common Lisp Object System CLOS kennenlernen, das auch als Erweiterung von DrRacket zur Verfügung steht.

Das Buch von Keene, 1989 ist eine kompakte, aber sehr gut lesbare Einführung in CLOS; das Buch von Graham, 1996 ist eine Einführung in die funktionale und objektorientierte Programmierung mit Common Lisp, die auch wesentliche Merkmale des CLOS-Systems beschreibt.

Relationale Programmierung

Gegen Ende des Semesters werden wir auch den relationalen (oder logischen) Programmierstil besprechen und dabei, allerdings nur sehr kurz, die Programmiersprache Prolog kennenlernen. Das Buch von [Clocksin and Mellish, 2003](#) ist eine sehr empfehlenswerte Einführung in die Grundlagen der Prolog-Programmierung, das Buch von [Sterling and Shapiro, 1994](#) beschreibt fortgeschrittene Methoden der Logikprogrammierung.

The screenshot shows a presentation slide with a light blue header bar containing the title 'Relationale Programmierung'. Below the header is a small image of a book cover. The main content area contains two entries, each with a small book icon and a blue link:

- [Clocksin, W. F. and Mellish, C. \(2003\).
Programming in Prolog.
Springer, Berlin.](#)
- [Sterling, E. and Shapiro, E. \(1994\).
The Art of Prolog : Advanced Programming Techniques.
MIT Press, Cambridge, Ma.](#)

At the bottom of the slide, there is a footer bar with the following text: Leonie Dreschler-Fischer (Department Inform), Softwareentwicklung III, WS 2019/2020, 66 / 1157. To the right of the footer bar are several small navigation icons.

Relationale Spracherweiterungen zu Lisp

Um relationales Programmieren zu üben, ohne deswegen gleich eine neue Programmierumgebung kennenzulernen, werden wir eine relationale Erweiterung von Racket benutzen, die von [Norvig, 1992](#) für die Programmiersprache Common Lisp entwickelt wurde. Für diese Vorlesung wurde der Code nach Racket portiert. Das Buch enthält eine Reihe spannender Fallstudien in funktionaler Programmierung, von denen wir einige in der Vorlesung besprechen werden.

Auch das schon erwähnte Buch von [Abelson et al., 1996](#) beschreibt, wie mittels Metaprogrammierung ein Prologinterpret in Racket programmiert werden kann.

Für die weitere Lektüre sei der „Reasoned Schemer“ (Friedman et al., 2005) empfohlen, der zeigt, wie mit wenigen neuen Sprachelementen die Sprache Scheme zu einer relationalen Programmiersprache wird.

The screenshot shows a presentation slide with a light blue header bar containing the title "Relationale Spracherweiterungen". Below the header, there is a small thumbnail image of a book cover. The main content area contains two entries, each with a small book icon and a list of details:

- Norvig, P. (1992).
Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.
Morgan Kaufmann Publishers San Mateo, CA 1992, San Mateo, Calif.
- Friedman, D. P., Byrd, W. E., and Kiselyov, O. (2005).
The Reasoned Schemer.
The MIT Press, Cambridge, MA.

At the bottom right of the slide, there is a set of small navigation icons typically used in Beamer presentations.

Handbücher und Spracheinführungen

Für Racket gibt es noch kein gedrucktes Handbuch, aber die online-Dokumentation im Hilfezentrum von DrRacket enthält einen sehr guten Guide (Flatt et al., 2010) und eine ausführliche Referenz (Flatt and PLT, 2010). Pdf-Files dieser beiden Handbücher finden Sie hier:

Guide: Racket <http://pre.plt-scheme.org/docs/pdf/guide.pdf>

Reference: Racket <http://pre.plt-scheme.org/docs/pdf/reference.pdf>

Drucken Sie sich ausgewählte Kapitel der Referenz aus – Sie sollten sie während des Programmierens und während der Klausur griffbereit haben.

Der „Little Schemer“ von ([Friedman and Felleisen, 1996](#)) ist ein Übungsbuch für Scheme, das sehr viele kleine Übungsaufgaben enthält, an denen man das Verständnis für die Sprache überprüfen kann.

Handbücher und
Spracheinführungen

Sperber, M., Dybvig, R. K., Flatt, M., and van Straaten, A. (2007).
The revised⁶ report on the algorithmic language scheme.
Im DrScheme-Helpdesk.

Friedman, D. P. and Felleisen, M. (1996).
The Little Schemer.
Mit Press Cambridge, MA, 4th edition.



Online-Materialien

- Die DrRacket-Homepage: <http://racket-lang.org/>
- Offizielle Scheme-Homepage: www.schemers.org
 - Racket-Systeme für den PC,
 - Dokumentation

2.2 Modulprüfung und Übungen

Materialien

Materialien:

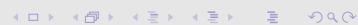
- Die Wep-page zur Vorlesung: <http://kogs-www.informatik.uni-hamburg.de/~dreschle/teaching/Lehre/Lehre.html>
- Übungsaufgaben in STINE
- Folienskript als pdf in STINE
- Racket-Module (Toolbox, Beispielprogramme):<http://kogs-www.informatik.uni-hamburg.de/~dreschle/informatik/Skripte/se3-bib.zip>

Übungen

- Erste Übung: 2. Vorlesungswoche
- Ausgabe der Übungsaufgaben in STINE
- Abgabe der Lösungen
 - per Mail bei der/dem Übungsgruppenleiter(in)
 - spätestens am Montagmittag eine Woche nach der Ausgabe der Aufgaben,
 - mit Angabe derjenigen Aufgaben, für die die Bereitschaft zum Vortragen der Lösung besteht.

Punkteschema

Nr.	Thema	Pkt.	Ausgabe Freitag	Abgabe Montag
0	Präsenzübung	0	19.10.	–
1	Notation, Funktionen	25	19.10.	29.10.
2	Namen, Symbole, Zahlen	24	26.10.	5.11.
3	Listen und Strings	30	2.11.	12.11.
4	Semantik	30	9.11.	19.11.
5	Rekursion	30	16.11.	26.11.
6	Funkt. höh. Ordnung	30	23.11.	3.12.
7	Closures	35	30.11.	10.12.
8	Baumrekursion	35	7.12.	17.12.
9	Kombinatorik	35	14.12.	7.1.
10	Gener. Funktionen	40	21.12.	14.1.
11	Methodenkombination	40	11.1.	21.1.
12	Prolog	40	18.1.	28.1.
13	Kombinatorik	(40)	28.1.	7.2.
		394		



Übungsschein

Erfolgreiche Teilnahme: Anforderungen

- regelmäßige aktive Teilnahme (Anwesenheit 85%)
- mindestens 50% der möglichen Punkte erreicht
 - Die Punktzahl pro Aufgabenblatt variiert zwischen 20 und 40, für die Aufgaben der 2. Semesterhälfte gibt es mehr Punkte.
 - Bestanden bei $355/2 = 177,5$ Punkten
- mindestens zwei Präsentationen von Lösungsvorschlägen vor der Übungsgruppe
- Teamarbeit und Abgabe der Aufgaben im Team erwünscht, max. 3 Personen pro Team

Das aktive Programmieren ist wichtig:

Peter Norvig, 1992:

„You will never become proficient in a foreign language by studying vocabulary lists. Rather, you must hear and speak (or read and write) the language to gain proficiency. The same is true for learning computer languages.“

Oder auf gut Deutsch: *Übung macht den Meister.*

Modulprüfung, Klausur

Zulassungsvoraussetzung: Erfolgreiche Teilnahme an den Übungen zu SE-III

Termine: In der vorlesungsfreien Zeit:

- Di, 18. Feb. 2020 12:30-14:30, ESA-A , Einlaß ab 9:15 Uhr
- Wiederholungsklausur: Do, 12. März 2018 09:30-11:30, ESA C, Einlaß ab 9:15 Uhr

Erlaubte Hilfsmittel: Falls Ihre Muttersprache nicht Deutsch sein sollte:
ein Wörterbuch.

(Sperber et al., 2007)

3 Die Arbeitsumgebung

3.1 Das Racket-System

Arbeitsumgebung

DrRacket: installiert auf den Linux-PCs, Macintoshs und Windows-PCs des Informatik-RZ, als freie Software für viele Plattformen verfügbar
<http://racket-lang.org/download/>.

SE3-Bibliothek mit Spracherweiterungen: installiert auf den Linux-PCs des Informatik-RZ, zu kopieren von <http://kogs-www.informatik.uni-hamburg.de/~dreschle/informatik/Skripte/se3-bib.zip>.

DrRacket

Zur Programmiersprache Racket gehört das DrRacket-Dialogsystem, dessen Bedienung Sie in den Übungen kennenlernen werden.

Die Bausteine sind:

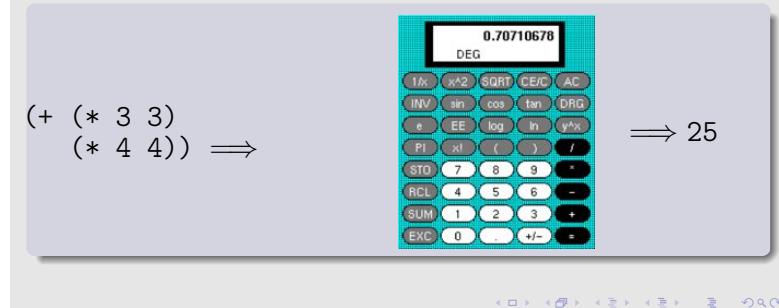
Ein Evaluator für Ausdrücke (*expressions*),

Eine Programmierumgebung zum Laden und Übersetzen des Programms, zum Ändern (Editieren) der Programme, Hilfestellung, Fehleranalyse usw.

Funktionale Ausdrücke werden vom *evaluator* ausgewertet; Kommandos werden von der *Programmierumgebung* bearbeitet.

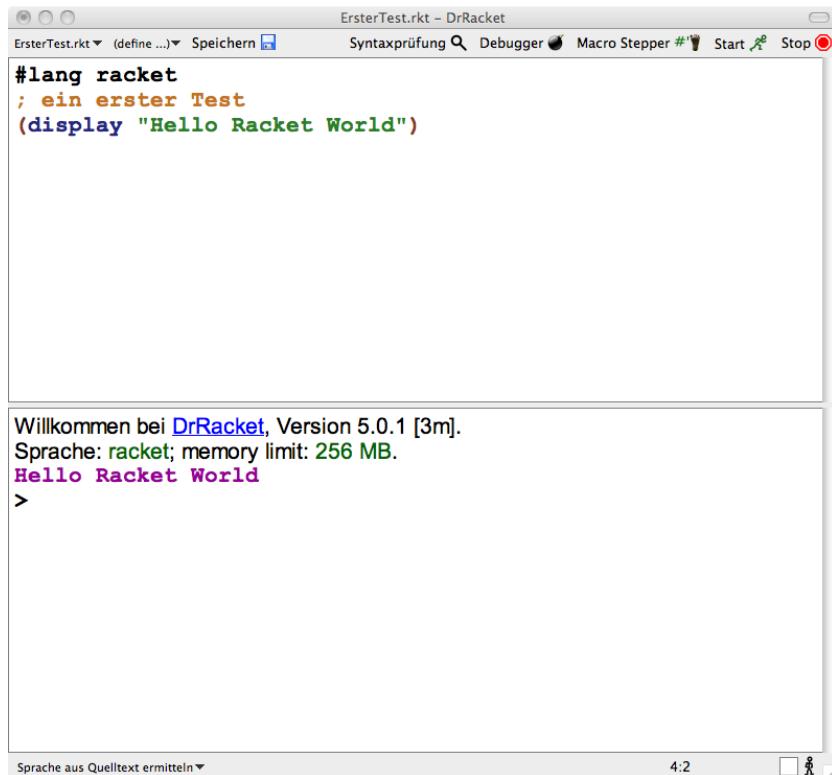
Erste Begegnung mit Racket

Wir können das Racket-System wie einen Taschenrechner benutzen: Ein Teil des Systems, der *evaluator*, wertet funktionale Ausdrücke, *expressions* genannt, aus und zeigt den Wert an:



Aufruf von DrRacket

An Linux-Workstations starten Sie DrRacket, indem Sie im Terminal-Fenster „racket“ „drracket“ eingeben:



The screenshot shows the DrRacket application window titled "ErsterTest.rkt - DrRacket". The top half of the window contains a code editor with the following Racket code:

```
#lang racket
; ein erster Test
(display "Hello Racket World")
```

The bottom half of the window is the "Evaluation Area" (Auswertungsumgebung) displaying the output of the code:

```
Willkommen bei DrRacket, Version 5.0.1 [3m].
Sprache: racket; memory limit: 256 MB.
Hello Racket World
>
```

Das DrRacket-Hauptfenster

Das Hauptfenster hat zwei Teile:

Ein Editor für Racket-Programme in der oberen Hälfte,

Eine Auswertungsumgebung in der untere Hälfte. In der Auswertungs-umgebung läuft der sogenannte toplevel, der

- Ausdrücke einliest,
- sie vom evaluator auswerten lässt
- und das Ergebnis zurückgibt.

Wahl der Sprache

Im DrRacket-System sind mehrere Racket-Dialekte implementiert, die unterschiedliche Teilmengen der Sprache bieten:

Lehre-Sprachen: Die Lehresprachen (Beginning Student, ...) schränken die Syntax ein, so daß treffendere Fehlermeldungen möglich sind.

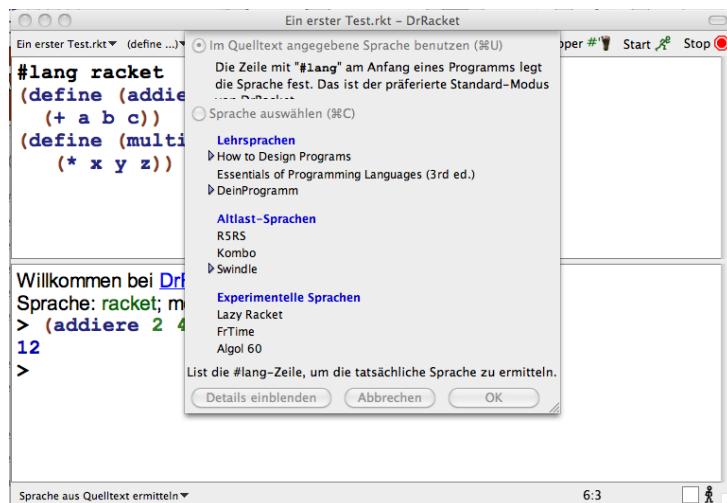
Professional Languages: Sprachen, die den vollen Sprachumfang des Revised Reports und Spracherweiterungen bieten (unter „Legacy Languages“ und „Experimental Languages“).

Professional Languages in DrRacket

- Racket
- Lazy Racket: ein Dialekt mit verzögerter Auswertung
- Standard Scheme (R5RS,R6RS)
- Swindle (Common Lisp, CLOS)
- Module: DrRacket stellt automatisch die Sprache in, die in einer Spezifikation mit #lang <language> am Anfang eines Programmmoduls deklariert wurde.

☞ Anmerkung: Zur Wahl der Sprache:

- Für die Übungen sollten Sie die Sprache auf Module einstellen die gewünschte Sprache mit #lang <language> in der Kopfzeile angeben. Wir wissen bei der Korrektur der Aufgaben dann, mit welcher Sprache Sie Ihr Programm getestet haben.
- In der ersten Semesterhälfte werden wir zunächst nur mit der Sprache „Racket“ arbeiten und
- später dann auch mit „Lazy Racket“.
- In der 2. Semesterhälfte benötigen wir auch die Spracherweiterungen von „Swindle“.



In der Auswertungsumgebung

```
Welcome to DrRacket, Version 5.0.1 [3m].  
Language: Racket.  
> 4  
4  
> (expt 2 4)  
16  
> (sin 1.5707963267949)  
1.0  
> (* (+ 2 2) (- 2 3))  
-4  
>
```



Fehlermeldungen

```
> (* 2 5)
 reference to undefined identifier: *2
> (* 2 5)
10
>
```

Wir bekommen eine Fehlermeldung, wenn der Operator oder einer der Operanden undefiniert sind. Hier ist die Funktion „*2“ unbekannt.



```
> (* pi Daumen)
 reference to undefined identifier: pi
> (modulo 2 2.3)
 modulo: expects type <integer>
as 2nd argument,
given: 2.3; other arguments were: 2
>
```

- Die Variablen „pi“ und „Daumen“ haben (noch) keinen definierten Wert
- und die Modulo-Operation ist nur auf ganze Zahlen anwendbar.

3.2 Symbolische Ausdrücke

Symbolische Ausdrücke (s-expressions)

Symbolische Ausdrücke sind entweder
atomare Formen, wie Zahlen, Zeichen, Symbole, Wahrheitswerte
zusammengesetzte Formen, wie Paare, Listen, Vektoren, Verbunde, Zeichenketten oder
funktionale Ausdrücke.

Funktionale Ausdrücke werden in *Präfixnotation* geschrieben, nach dem Schema:

```
<s-expr> ::= ( <function> { <s-expr> } ) | ...
> (< (sqrt 4) (sqrt 5)) ; √4 < √5?
#t      ; wahr (true)
> (= (sqrt 4) (sqrt 5)) ; √4 = √5?
#f      ; falsch (false)
```

Ausdrücke (s-expressions)

- Der Wert eines Ausdrucks kann nicht nur eine Zahl, sondern auch ein Objekt eines beliebigen anderen Typs sein:
 - ein Symbol, ein Buchstabe, ein Wahrheitswert
 - eine Liste, eine Zeichenkette, eine Reihung, ein Vektor,
 - eine Funktion usw.
- Ausdrücke können geschachtelt sein.

Klammerung

- Dank der Klammerung ist immer klar, wann ein Ausdruck beendet ist. Es ist keine besondere Endemarkierung nötig.
- Ein Vorteil der Klammerung und der Präfixnotation ist, daß wir Funktionen mit *variabler Zahl von Argumenten* definieren können; z.B. die arithmetischen Operatoren nehmen beliebig viele Argumente,
 - (+ 1), (+ 1 2) (+ 1 2 3 4)
 - (*), (* 1 2) (* 1 2 3 4)
- In Racket gibt es keine *Operatorpräzedenzen* wie in Java oder C.

Zur Notation

Auf den folgenden Folien wird die Auswertung der Racket-Ausdrücke durch den Evaluator abkürzend dargestellt:

> (+ 1 2) → 3

- Das linke Größerzeichen > ist die Eingabeaufforderung (prompt) von DrRacket.
- Der Pfeil → soll bedeuten: *Evaluert zu ...*

Variable Zahl von Argumenten

Sprache: racket ; memory limit: 256 MB

```
> (+ 1 2) → 3
> (+ 1 2 3) → 6
> (+ 4 5 6 7) → 22
> (+ 1) → 1
> (+) → 0
> (= 1 1 (- 2 1)) → #t
> (< 1 3 5 7) → #t
> (> 7 5 3 4) → #f
```

Sprache: Anfänger ; memory limit: 256 MB.

```
> (+ 1)

procedure +: expects at least 2
arguments , given 1: 1
>
```

☞ **Anmerkung:** Wenn Sie eine der eingeschränkten Lehrsprachen wählen, bekommen Sie eine Fehlermeldung, wenn Sie den +-Operator oder den *-Operator mit weniger als zwei Operanden aufrufen.

Formatieren

- Geschachtelte Ausdrücke sollten wir durch passendes Einrücken übersichtlich schreiben.
- Am besten überlassen wir das Einrücken dem Formatierungsprogramm im DrRacket-System.
- Der DrRacket-Editor zeigt uns auch durch Blinken an, welche Klammern zusammengehören.
- Ebenso werden syntaktische Schlüsselwörter durch unterschiedliche Farben oder Schriftarten hervorgehoben.

```
(and
  (> (sin pi)
       (sqrt e))
  (> (sqrt 6)
      (max 7 3))) ;
```

sin ist die Sinusfunktion, **sqrt** die Quadratwurzelfunktion, **max** die Maximumsfunktion, **and** das logische „und“.

Gut formatiert:

```
(or ; ( $\sqrt{5} > \sqrt{4}$ )  $\vee$  ( $\sqrt{6} > \sqrt{7}$ )
  (> (sqrt 5)
      (sqrt 4))
  (> (sqrt 6)
      (sqrt 7))) ;
```

Schlecht formatiert:

```
(or (> (sqrt 5)
        (sqrt 4)) (> (sqrt 6)
        (sqrt 7)))
```

or ist das logische „oder“.

Kommentare

In DrRacket haben Sie drei Möglichkeiten, einen Kommentar einzufügen:

1. Das Semikolon leitet einen Kommentar ein, der sich bis zum Ende der Zeile erstreckt. ; *Ab hier wird alles überlesen.*
2. Die Zeichen #/ begrenzen einen mehrzeiligen Kommentar |#.
3. Über das Special-Menü können Sie eine Kommentarbox einfügen.

Konstruktion von S-expressions

Die Ausdrücke werden aus

- elementaren Operatoren und Funktionen (Standardfunktionen genannt), die im System vordefiniert sind, aufgebaut, sowie aus
- unseren eigenen Funktionen, die
 - entweder in einem file definiert und vor der Sitzung geladen werden,
 - oder während der Sitzung im Editor-Fenster neu definiert werden.

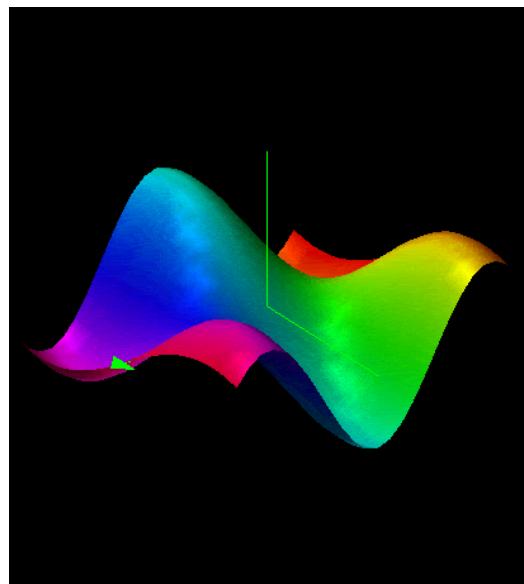
```
(define (meinCosinus x)
  (sqrt (- 1 (*
    (sin x)
    (sin x))))) ; cos α = √(1 - sin² α)
```

Teil II

Konstrukte der funktionalen Programmierung

4 Funktionale Abstraktion

4.1 Namensgebung



Die Elemente der applikativen Programmierung

Das *Verarbeitungsmodell* basiert auf dem mathematischen *Funktionsbegriff*.

Zur Programmierung benötigen wir: Eine Sprache, die bereitstellt:

- Grundfunktionen,
- Mittel zur Kombination von Termen zu Ausdrücken,
- Mittel zur Einführung zusammengesetzter Funktionen,
- (Mittel zur Beschreibung der Definitions- und Wertebereiche von Funktionen)

 **Beachte:** **Abstraktion:** Die Methode der Abstraktion wird bei der Programmierung in vielfältiger Weise eingesetzt.

- Verarbeitungsmodelle abstrahieren von der zugrunde liegenden Hardware der ausführenden Rechenanlagen.
- Anwendungsprogramme wiederum modellieren den Anwendungsbereich und abstrahieren sowohl von den konkreten Anwendungsobjekten als auch vom Verarbeitungsmodell.

Allen Anwendungen der Abstraktion ist gemeinsam, daß wir die Komplexität meistern, indem wir generalisieren. Wir bilden Klassen von Objekten, die hinsichtlich wichtiger Eigenschaften äquivalent sind. Die gemeinsamen Eigenschaften werden durch eine Äquivalenzrelation beschrieben, die die Menge der Objekte in Äquivalenzklassen partioniert. Stellvertretend für die konkreten Objekte arbeiten wir dann mit abstrakten Objekten, die aufgrund der gemeinsamen Eigenschaften für jeweils eine Äquivalenzklasse von Objekten stehen. Wir können neue Begriffe bilden, indem wir den Äquivalenzklassen Namen geben (siehe Anhang, 735).

Abstraktion in der funktionalen Programmierung

- Bei der funktionalen (applikativen) Programmierung gibt es zwei wichtige Anwendungen der Abstraktion:
 - *Funktionale Abstraktion*
 - und *Datenabstraktion*.
- Beide Arten der Abstraktion dienen dazu, typische Muster (seien es nun Daten oder funktionale Ausdrücke) zu wiederverwendbaren Bausteinen zu verallgemeinern, aus denen neue, komplexere Einheiten gebildet werden können.
- *Beide Arten der Abstraktion abstrahieren also von den Grundoperationen und primitiven Objekten des zugrundeliegenden Verarbeitungsmodells.*

Definition neuer Namen

Definition: 7 (Namensgebung)

ist ein Mittel, um sich auf (abstrakte) Objekte zu beziehen. Der Name einer Funktion steht als *black box* für das Berechnungsverfahren.

Wir sagen: Ein Name identifiziert ein Symbol, dessen Wert ein (abstraktes) Objekt ist.

Benennung ist das erste und einfachste Mittel zur Abstraktion.

... man sieht dem Wert nicht mehr an, wie er entstanden ist.

Wertzuweisung als Abstraktion

Einem Wert sieht man nicht mehr an, wie er zustande kam.

```
> (define k (* (/ 4 2) 3))    → ⊥  
> (define k (* 12 (/ 4 8)))   → ⊥  
> (define k 6)                 → ⊥  
> (* k k)                     → 36
```

Abstraktor: Die Berechnungsschritte.

Äquivalenzrelation: Wertgleichheit der Ausdrücke.

Resultat der Abstraktion: Die Konstante „6“.

Definition von Namen

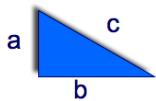
- Ein Name wird definiert, indem wir ihn in einem define–Ausdruck einführen.
- Bei der Definition muß ein Ausdruck angegeben werden, der den Wert bezeichnet, an den der Name gebunden wird.
 $(\mathbf{define} \langle \text{Name} \rangle \langle \text{Wert} \rangle)$ oder
 $(\mathbf{define} \langle \text{Name} \rangle \langle s\text{-expression} \rangle)$
- Die *special form expression* (**define**) definiert eine Variable namens $\langle \text{Name} \rangle$ und initialisiert sie mit dem Wert des angegebenen Arguments.
- Der Wert eines (**define**)-Ausdrucks ist undefiniert \perp .

Verwendung von Namen

- Namen müssen *eindeutig* sein. Daher dürfen wir für unsere Definitionen keine Namen verwenden,
 - die schon als Standardfunktionen (wie **sin**, **sqrt**),
 - syntaktische Schlüsselwörter (wie **define**, **begin**, **if**),
 - Konstanten (wie **#t**, **#f**)

oder andere Zwecke in Racket belegt sind.

- Wenn wir Namen definiert haben, können wir uns bei der Formulierung von Ausdrücken darauf beziehen,
 - sowohl im Evaluator
 - als auch im Programm,da diese Namen dann zur globalen Umgebung gehören.



Rechtwinkeliges Dreieck



Beispiel: 8

Berechne die Hypotenuse eines Dreiecks $c = \sqrt{a^2 + b^2}$:

Im Editor-Fenster von DrRacket schreiben wir:

```
#lang racket
;; Mein erstes Racket-Programm:
;; Berechne die Hypotenuse eines Dreiecks
;; nach dem Satz von Pythagoras
(define a 10)
(define b 20)
;; Jetzt können wir c-Quadrat ausrechnen
(define c
  (sqrt (+ (* a a)
            (* b b))))
```

- Sichern Sie Ihr Programm in einem File, z.B. `pythagoras.rkt`.
- Drücken Sie die Run-Taste, um die Definitionen aus dem Editor-Fenster zu laden.
- Jetzt können Sie die definierten Namen im Evaluator-Fenster verwenden.

Willkommen bei DrRacket , Version 5.0.1 [3m].

Sprache: racket ; memory limit: 256 MB.

```
> c          → 22.360679774997898  
> a          → 10  
> b          → 20  
>
```

Zur Nomenklatur

- Die special form **define** definiert eine *Variable*
- und bindet den Wert der Variablen an ein Symbol.
- Man sagt: „Der Name der Variablen *referenziert* das Symbol.“
- Die Symbole werden in einer *Symboltabelle* gespeichert.
- Wird eine Variable in einem Ausdruck referenziert, wird der Wert aus der Symboltabelle ausgelesen.

☞ Beachte: Variablen sind nicht variabel!

- In der funktionalen und relationalen Programmierung sind Variable feste Größen, entsprechend dem mathematischen Variablenbegriff.
- Sie werden bei der Definition an einen Wert gebunden, der später nicht mehr verändert werden sollte.
- Die Lehrsprachen von DrRacket verbieten es explizit, den Namen einer definierten Variablen durch ein erneutes **define** zu verändern.

Willkommen bei DrRacket , Version 5.0.1 [3m].

Sprache: Zwischenstufe ; memory limit: 256 MB.

Tests deaktiviert .

```
> (define a 2)  
> (define a 4)  
define: cannot redefine name: a
```

Racket verwendet deutlich weniger Interpunktionszeichen als andere Sprachen, beispielsweise C. Daher können viele Zeichen Teil eines Namens sein, da sie keine trennende Wirkung haben.

Lexikalische Einheiten

Namen können beliebige nicht-leere Worte über dem Alphabet der druckbaren ASCII-Zeichen sein (bis auf wenige Einschränkungen).

- Die Zeichen dürfen kein „white space“ sein, wie Leerzeichen, Tabulator oder Zeilenvorschub.
- Folgende Zeichen dürfen nicht vorkommen:
() [] { } , ' ' ; # / \ "
- Die Zeichenfolge darf keine Zahlkonstante ergeben:

Beispiele

Zulässige Namen: \$=>Euro, euro=>\$, *abc*, .a, 1a, R2D2

Unzulässige Namen: (2), 11, 1e-05

Warum sind diese Namen unzulässig?

☞ Anmerkung: Zur Notation von Definitionen in Racket

- Wir sollten keine Namen wählen, die in der Standardumgebung schon gebunden oder als Schlüsselwort (**define**, **if**, **cond** ...) festgelegt sind, denn sonst können wir die ursprünglichen Funktionen nicht mehr verwenden, da sie durch unsere neuen Definitionen verschattet werden würden.

Syntaktische Schlüsselwörter

Die folgenden Namen sind *syntaktische Schlüsselwörter* und dürfen *auf keinen Fall* durch unsere eigenen Definitionen verschattet werden:

define	do	or
and	begin	else
if	case	cond
lambda	set!	let
let*	letrec	delay
quasiquote	quote	

Interpunktionszeichen

Die Interpunktionszeichen sind:

- Öffnende und schließende, runde, geschweifte oder eckige Klammern,
- Gänsefüßchen ", quote ', backquote ',
- Leerzeichen, Zeilenende, Semikolon und Komma.

Interpunktionszeichen können nicht Teil eines Namens sein.

Groß- und Kleinschreibung

DrRacket: Variablenamen werden hinsichtlich der Groß/Kleinschreibung unterschieden:

Standard-Scheme (R5RS) dagegen unterscheidet *nicht* zwischen Groß- und Kleinschreibung,

```
Welcome to DrRacket, version 5.0.1 [3m].
```

```
Language: R5RS; memory limit: 256 MB.
```

```
> (define abc 3)      → ⊥
```

```
> (= abc aBc Abc)    → #t
```

```
Language: racket; memory limit: 256 MB.
```

```
> (define abc 3)      → ⊥
```

```
> (define abcC 4)     → ⊥
```

```
> (= abc abC)        → #f
```

```
> abc                 → 3
```

```
> abC                 → 4
```

Wenn Sie also portierbare Scheme-Programme schreiben wollen, verzichten Sie am besten auf Namen, die sich nur in der Groß-Kleinschreibung unterscheiden und halten sich dann konsequent an die einmal gewählte Schreibweise.

4.2 Der Lambda-Abstraktor

Funktionsdefinition als Abstraktion:

Für anspruchsvollere Aufgaben brauchen wir mehr als nur Ausdrücke über Standardfunktionen:

- Wir wollen eine Klasse von *äquivalenten komplexen Operationen* als Einheit auffassen,
- als ein neues, abstraktes Objekt einführen
- und der Klasse einen Namen geben, mit dem wir uns auf sie beziehen können (Referenz).

Aufgabe: Inhalt einer Kugel

Beispiel: 9

Radius = 4: Volumen = $4/3 * \pi * 4 * 4 * 4$

Radius = 5: Volumen = $4/3 * \pi * 5 * 5 * 5$

Radius = 6: Volumen = $4/3 * \pi * 6 * 6 * 6$

Radius = \square : Volumen = $4/3 * \pi * \square * \square * \square$

Für beliebige Radien haben wir immer dieselbe Termstruktur!

Strukturgleiche Ausdrücke:

Wir führen den Radius **r** als Variable ein:

```
> (define pi (* 2 (asin 1.0))) → ⊥  
> pi → 3.141592653589793  
> (* (/ 4 3) pi r r r) → 0.3568179048045239  
  
> (define r2 3) ; r2, anderer Wert für r  
> (* (/ 4 3) pi r2 r2 r2) → 113.09733552923255
```

Strukturgleichheit als Äquivalenzrelation



Alonzo Church

- ▶ Abstraktion über *Strukturgleichheit als Äquivalenzrelation*.
 - ▶ Der Abstraktor heißt **λ -Abstraktor** nach dem Church'schen λ -Kalkül. Wir erhalten als abstrakte Funktion

$$\lambda r.(4/3 * \pi * r * r * r)$$

Funktionsobjekt als Abstraktion

- Wir erhalten Funktionsobjekte als Abstraktion vieler ähnlicher Berechnungen mit unterschiedlichen Werten.
 - In Racket gibt es den λ -*Abstraktor* als Sprachelement, der uns anonyme Funktionsobjekte erzeugt.

(lambda (r) (* (/ 4 3) pi r r r))
; eine anonyme Funktion

```
> ((lambda (r) (* (/ 4 3) pi r r r))  
    4)  
→ 268.082573106329
```

λ -Notation in DrRacket

- In DrRacket können Sie anstelle des Wortes **lambda** auch ein λ -Zeichen schreiben (über das special-Menü eingeben).
 - Die Analogie zum λ -Kalkül wird dann besonders deutlich, aber Ihr Programm ist dann nicht portierbar.

Willkommen bei DrRacket, Version 5.0.1 [3m].

Sprache: racket; memory limit: 256 MB.

```

> ( $\lambda(r)(\cdot(\cdot(\cdot(\cdot 4) 3) \pi r r r))$ )
      #<procedure>; eine anonyme Funktion
> (( $\lambda(r)(\cdot(\cdot(\cdot(\cdot 4) 3) \pi r r r))$ )
    4)
--> 268.082573106329

```

Lambda-expressions

- Ein *Lambda-Ausdruck* hat die allgemeine Form

$(\lambda (\langle \text{parameters} \rangle) \langle \text{body} \rangle)$

- Ein lambda-Ausdruck ist ein *nicht-atomarer Name* für eine Funktion, so wie „**sqrt**“ oder „+“ *atomare* Namen für Funktionen sind.

```
((lambda (x)
         (+ x 2)) 4)      → 6
(sqrt 4)           → 2
```

☞ Beachte: Der λ -Abstraktor: Der λ -Abstraktor wurde von Alonzo Church mit dem λ -Kalkül eingeführt, das Sie in den Modulen zu den formalen Grundlagen der Informatik kennenlernen. Die Notation $\lambda x.(x * x)$ bedeutet, daß wir eine abstrakte Funktion erzeugt haben, die für beliebige x als Argument den Wert $x * x$ berechnet.

Beachten Sie, daß diese abstrakte Funktion noch keinen Namen hat. Wir können sie per Namensabstraktion an beliebige Namen, wie „Karlchen“, „Nikolausi“, oder sinnvoller, „Square“ binden, und sie wird doch immer, unabhängig vom Namen, dieselbe Menge von strukturgleichen Ter men darstellen.

Es ist sehr lästig, allem und jeden einen Namen geben zu müssen, selbst wenn man etwas nur einmal braucht, denn Namen müssen innerhalb der Umgebung eindeutig sein. Und zusätzlich sollen sie auch noch sinnvoll sein, damit wir sie uns merken können. Wenn wir etwas nur einmal benutzen wollen, brauchen wir keinen Namen. Wir können das Objekt direkt verwenden. Wer gibt schon jeder Nudel oder jedem Reiskorn einen Namen, bevor er diese verspeist? Wenn man zuviele Namen finden muß, ist es eher kontraproduktiv. Ein Beispiel:

```
(define t1 (* 3 7)) ;
(define t2 (* 4 6)) ;
(define ergebnis (* t1 t2)) ;
```

ist weniger klar als

```
(define ergebnis (+
                     (* 3 7)
                     (* 4 6)))
```

Auch Funktionen können den Wert eines funktionalen Ausdrucks darstellen und als Ergebnis einer Berechnung erzeugt werden. Dann brauchen wir sie nicht jedesmal zu taufen, ehe wir sie an andere Funktionen als Argumente weitergeben.

In Lisp und Racket gibt es glücklicherweise den λ -Abstraktor als Sprachelement, so daß wir Hilfsfunktionen, die nur lokal gebraucht werden, direkt in dieser Notation definieren können:

```
(lambda (x) (* x x))  
; das Beispiel von oben in Racket
```

Ursprung des Namens „lambda“

$\Lambda\lambda$

The name *lambda* comes from the mathematician Alonzo Church's notation for functions (Church 1941). Lisp usually prefers expressive names over terse Greek letters, but lambda is an exception. A better name would be make-function. **Lambda** derives from the notation in Russel and Whitehead's *Principia Mathematica*, which used a caret over bound variables: $\hat{x}(x + x)$. Church wanted a one-dimensional string, so he moved the caret in front: $\hat{x}(x + x)$. The caret looked funny with nothing below it, so Church switched to the closest thing, an uppercase lambda, $\Lambda x(x + x)$. The Λ was easily confused with other symbols, so eventually the lowercase lambda was substituted: $\lambda x(x + x)$. John McCarthy was a student of Church's at Princeton, so when McCarthy invented Lisp in 1958, he adopted the lambda notation. There were no Greek letters on the keypunches of that era, so McCarthy used (**lambda** (x) (+ x x)), and it has survived to this day.
Norvig 1992

Die erste Veröffentlichung des Lambda-Kalküls ist in (Church, 1941) nachzulesen.

Woher kommt der Name „lambda“?

- Ursprünglich hatten Russel und Whitehead gebundene Variablen durch ein ^-Zeichen markiert: $\hat{x}(x + x)$
- Das ließ sich nicht damals nicht gut als sequentieller Text schreiben, deshalb hat Church das Dach vor die Variablennamen gesetzt: $\hat{x}(x + x)$.
- Das kleine ^-Zeichen sah vor den Variablennamen nicht gut aus. Deshalb wurde es durch ein großes Λ ersetzt: $\Lambda x(x + x)$.
- Wegen der Verwechslungsgefahr mit dem logischen „und“ wurde das Zeichen schließlich durch λ ersetzt.
- Da es auf seinen Kartenlochern damals keine griechischen Zeichen gab, hat McCarthy den Namen LAMBDA einfach ausgeschrieben.

Anonyme Funktionen

- Der **lambda**-Ausdruck referenziert eine anonyme Funktion.
- Funktionen benötigen nicht unbedingt einen Namen, damit wir sie aufrufen können.
- Den **lambda**-Ausdruck können wir als komplexen Namen wie einen Namen von Standard-Funktionen verwenden:

```
> ( (lambda (r) (* (/ 4 3) pi r r r)) 4)
→ 268.082573106329
> ( (lambda (r) (* (/ 4 3) pi r r r)) 1)
→ 4.188790204786391
> (define volume
      (lambda (r) (* (/ 4 3) pi r r r)))
> (volume 3)           → 113.0973355292326
> (volume 1)           → 4.188790204786391
```

Benennung von Funktionen

Funktionen können mittels **define** benannt werden, wenn sie einen Namen brauchen,

- um sie zu dokumentieren,
- um sie mehrfach mit unterschiedlichen Argumenten aufrufen zu können
- oder um sie rekursiv aufrufen zu können.

```
> (define volume  
    (lambda (r) (* (/ 4 3) pi r r r)))  
  
> (volume 3)      → 113.0973355292326  
> (volume 1)      → 4.188790204786391
```

Eine zweite Notation zur Definition von Funktionen

```
(define square ; Die Quadrierfunktion  
    (lambda (x) (* x x)))  
  
;; äquivalente Definition  
(define (square x)  
    (* x x))
```

Die zweite Notationsform lehnt sich an der denotationellen Semantik an:

- Der Ausdruck (square x)
- ist durch (* x x) zu ersetzen.

Zur Syntax

Beide Notationsformen zur Funktionsdefinition sind äquivalent.

- Die zweite Form ist reiner „syntaktischer Zucker“, um uns das Leben zu erleichtern.
- Beide Formen werden in der Literatur verwendet, so daß Sie mit beiden Formen vertraut sein sollten.

☞ **Anmerkung:** Zur Nomenklatur: In der Sprachdefinition von Racket werden Funktionen, wie wir sie oben eingeführt haben *procedure* genannt. Funktionen im eigentlichen Sinne sind solche *procedures*, die keine Seiteneffekte erzeugen (was das bedeutet, werden wir noch ausführlich behandeln). Da wir ohnehin nur Prozeduren ohne Seiteneffekte verwenden werden, haben wir gleich den Begriff „Funktion“ eingeführt.

5 Variablenkonus und closures

5.1 Freie und gebundene Variable

Freie und gebundene Variable

```
> (define volume
  (lambda (r)
    (* (/ 4 3) pi (r r r)))
```

- **r** ist eine gebundene Variable bezüglich des λ -Ausdrucks.
- **pi** ist eine freie Variable.
- Namen gebundener Variablen sind austauschbar.

Diese Definition stellt dieselbe Funktion dar:

```
(define volume
  (lambda (rad)
    (* (/ 4 3) pi rad rad rad)))
```

Die lokale Umgebung

☞ Die Namen der *gebundenen Variablen* sind nur in der *lokalen Umgebung* der Funktion definiert und können und dürfen außerhalb nicht referenziert werden.

```
(define volume
  (lambda (rad)
    (* (/ 4 3) pi rad rad rad)))
```

☞ Beachte: Zum Variablenbegriff

- Wir bezeichnen wie in der Mathematik üblich, diejenigen Namen, die als Platzhalter für aktuelle Werte stehen, als *Variable*, auch wenn die Werte, an die diese Namen gebunden sind, nicht variieren, jedenfalls nicht während einer Funktionsanwendung.
- Der Wert einer freien Variable kann zwar von Fall zu Fall für unterschiedliche Werte stehen, aber während der Reduktion (Auswertung) eines funktionalen Ausdrucks steht er immer nur für einen bestimmten festen Wert.

☞ Anmerkung: Auch der Name einer Funktion ist syntaktisch in Racket der Name einer Variablen, deren (fester) Wert die Funktion ist.

Argumente, formale und aktuelle Parameter

Definition: 10

Argumente: An eine Funktionsdefinition gebundene Variable heißen *Parameter* oder *Argumente* einer Funktion.

Formale Parameter: In der Funktionsdefinition werden die gebundenen Variablen *formale Parameter* (*Formalparameter*) genannt.

Aktuelle Parameter: Bei der Funktionsanwendung werden die Formalparameter an feste Werte gebunden. Diese heißen die *aktuellen Parameter* (*Aktualparameter*).

Schnittstellen

- Die Abbildung zwischen *Definitions-* und *Wertebereich* einer Funktion, ihre *Signatur*, kann als *E/A-Relation* aufgefaßt werden.
- Über das E/A-Verhalten von Funktionen lassen sich neue Abstraktionsbegriffe einführen.
- Gleicher E/A-Verhalten von Funktionen als Äquivalenzrelation führt zu Klassen von Funktionen gleichen Typs.
- Wir werden beim Thema „Funktionen höherer Ordnung“ hierauf zurückkommen.

5.2 Fallunterscheidungen

Fallunterscheidungen

Häufig werden in mathematischen Funktionsdefinitionen Fallunterscheidungen gemacht, beispielsweise

$$\max 2(x, y) = \begin{cases} x \text{ falls } x \geq y \\ y \text{ sonst} \end{cases}$$

$$\text{durchy}(x, y) = \begin{cases} x/y \text{ falls } |y| > 0 \\ \infty \text{ sonst} \end{cases}$$

Bedingte Ausdrücke

In Racket können wir Fallunterscheidungen in *bedingten Ausdrücken* (engl. conditional expressions) vornehmen.

```
Language: racket;
> (define (max2 a b)
  (if (> a b) ; wenn a>b ist ,
      a ; dann a
      b)) ; andernfalls b
> (max2 3 5)           → 5
> (max2 5 3)           → 5
```

Absichern einer Division

```
Language: racket;
> (define (durchy x y)
  (if (> (abs y) 0) ; Falls y > 0,
      (/ x y) ; dann x/y
      (error "durchy: divisor = 0!")))
> (durchy 2 4) → 1/2
> (durchy 39 42) → 13/14
> (durchy 2 0)
```



durchy: divisor = 0!
>

Ausdrücke der Form

(**if** <Bedingung> <s-expr1> [<s-expr2>])
heißen *bedingte Ausdrücke*.

☞ Anmerkung: An diesem Beispiel sehen Sie, daß in Racket Rationalzahlen als exakte Brüche dargestellt werden. Solange Sie nur ganze Zahlen oder Rationalzahlen über die arithmetischen Grundoperationen verknüpfen, bleiben die Zahlen exakt, d.h., es treten keine Rundungsfehler auf.

Bedingte Ausdrücke

```
(if <test> <consequent> <alternate>)
(when <test> <consequent>)
Language: Standard (R5RS).
> (let ([x 1]
       [y 2])
   (if (< x y) (- y x) (- x y))) → 1
> (let ([x 1]
       [y 2])
```

```
( if (< x y) (display "x<y") (void))) → ⊥
x<y
>
```

Die Alternative des **if** darf in Standard-Scheme (R5RS, R6RS) und Common Lisp auch fehlen, das Resultat ist dann undefiniert, wenn der Test #f ergibt, in Racket ist das unzulässig.

Einseitige Verzweigungen

In Racket darf die Alternative des **if** *nicht* fehlen – dafür gibt es hier die einseitigen Verzweigungen **when** und **unless**.

Sprache: **Module**; *memory limit: 128 megabytes*.

```
> (let ([x 1])
    (if (> x 0) 1)) →
if: bad syntax (must have an "else" expr.)
> (let ([x 1])
    (when (> x 0) 1)) → 1
> (let ([x 1])
    (unless (> x 0) 1)) → ⊥
> (let ([x 1])
    (unless (= x 0) 1)) → 1
```

Das Konditional

```
;;; Errechne den prozentualen Steuersatz
;;; abhängig vom Einkommen.
(define (steuersatz eink)
  (cond [(< eink 10000) 0]
        [(< eink 20000) 10]
        [(< eink 40000) 30]
        [(< eink 180000) 45]
        [else 80]))
> (steuersatz 100) => 0
> (steuersatz 200000) => 80
```

Das Konditional: Syntax

```
(cond [<Bedingung1> {<s-expr>} <Resultat1>]
      [ .... ]
      [<BedingungN> {<s-expr>} <ResultatN>]
      [else {<s-expr>} <ResultatSonst>])
```

Ausdrücke der Form

[Bedingung {<s-expr>} <Resultat>]

heißen *guards* oder *Wächter*.

Auswertung von Wächtern

- Die guards werden der Reihe nach ausgewertet.
- Der Wert der ersten Bedingung, die erfüllt ist, wird als Wert des Konditionals zurückgegeben.
- Wenn die Bedingungen nicht disjunkt sind, ist daher die Reihenfolge der „guards“ wichtig, bei disjunkten Bedingungen dagegen nicht.
- Jede Konstellation von Argumenten, die auftreten kann, sollte durch einen Wächter abgedeckt sein, sonst ist die Funktion nur partiell definiert.
- Trifft keine der Bedingungen zu, wird die Auswertung mit einer Fehlermeldung abgebrochen.

„catch-all“

- **else** ist ein sogenannter „catch-all“, dessen Bedingung immer zutrifft, falls keine der vorher genannten „guards“ zutreffen sollten.
- Solche „catch-alls“ sind sinnvoll, um Fehler abzufangen, wenn bestimmte Fälle eigentlich gar nicht auftreten dürften. Wir können dann eigene spezifische Fehlermeldungen geben.
- Wir sollten „catch-alls“ aber nicht dazu missbrauchen, summarisch alle Fälle zu behandeln, über die wir nicht weiter nachdenken wollen. Für die Verständlichkeit von Programmen ist es oft besser, wenn wir alle zu unterscheidenden Fälle explizit machen.

Schleifen

- *Schleifen*, wie `while`, `dolist`, `dotimes` usw. sind typisch für das Programmieren mit *Zuständen*.
- Sie sind in der Spracherweiterung Swindle verfügbar, und wir werden später darauf eingehen, siehe Abschnitt „Objekte und generische Funktionen“
- Im reinen funktionalen Programmierstil werden Schleifen sehr elegant durch *Rekursion* oder *Funktionen höherer Ordnung* ausgedrückt, siehe Teil 3 des Skriptes.

5.3 Umgebungen und closures

Manche Funktiondefinitionen sind übersichtlicher, wenn wir sinnvolle Teilterme durch Hilfsfunktionen errechnen lassen oder an Namen binden. Da diese Namen nur lokal benötigt werden, sollten sie auch nur lokal definiert werden.

Diese Namen sind dann, genau wie die formalen Parameter der Funktion, nur in der lokalen Umgebung sichtbar und referenzierbar. **let** ist die gebräuchliche Form, um Variablen einzuführen, die nur lokal von Bedeutung sind. Die Namen sind nur innerhalb der Klammern des **let** zugreifbar. Sie verschatten Namen von Variablen, die mit gleichem Namen außerhalb des **let** eingeführt wurden.

Mit **define** eingeführte Variablen dagegen sind besondere Variablen, die global bekannt sind und auch nur verwendet werden sollten, wenn sie global benötigt werden.

Lokale Definitionen helfen dabei, die Software modular zu entwerfen. Wenn mehrere Personen an einem Programm arbeiten, brauchen sie nicht für jede Variablentaupe eine Konferenz einzuberufen, damit keine Namenskonflikte entstehen.

Lokale Definitionen

- Es ist guter Programmierstil, Funktionen und Variable, die nur lokal im Programm benötigt werden, nur lokal zu definieren.
- Lokale Definitionen werden in einer **let**-Klausel eingeführt:
- Das Schema:

```
(let ( ; Namens-Wert-Paare  
      [ <Name1> <s-expr1> ]  
      [ <NameN> <s-exprN> ] )  
      {s-expr} ) ; body
```

- Folge von auszuwertenden Ausdrücken:
- Der letzte Ausdruck definiert den Wert des **let**.

Lokale Variable mit „let“

```
> ( let ([x 40]           ; variable x
         [y (+ 1 1)]) ; variable y
      (+ x y))       ; body
=> 42
> ( let* ([x 6]
          [y (* x x)])
      (+ x y))
=> 42

(define (f x y)
  ( let ([a (/ (+ x y) 2)] ; a = (x+y)/2
        [b (* x x)])        ; b = x*x
    (* (+ a 1) (+ b 2)))) ; Resultat: (a+1)*(b+2)

(define (f2 x z)
  ( let* ([square
          (lambda (y) (* y y))]
         ; lokale Funktion square
         [a (square x)])        ; a = x*x
        (if (> x 10) (+ x a)
            (- x a))))
```

Special form operator let

- Lokale Variable werden direkt bei der Definition an Werte gebunden.
- Die Reihenfolge der Bindung ist undefiniert.
- Die so eingeführten Namen sind nur innerhalb des **let** sichtbar.
- Nach der Variablenbindung werden sequentiell die Ausdrücke des Rumpfes (body) ausgewertet.
- Der Wert eines **let** ist der Wert des letzten Ausdrucks des „body“. Die Auswertung der anderen Ausdrücke wirkt nur durch Seiteneffekte, z.B. Ausgabe auf Files.
- **let**-expressions können geschachtelt sein.

Special form operator „let“

- In einem **let** können mehrere Variablen definiert werden.
- Bei der Definition einer Variablen darf nicht auf die anderen gleichzeitig definierten Variablen Bezug genommen werden.
- Wenn die Bindungen sequentiell in Abhängigkeit voneinander vorzunehmen sind, dann muß die Form **let*** genommen werden.
- Im folgenden Beispiel wäre mit **let** die Variable **x** noch nicht gebunden, wenn **y** berechnet werden soll.

```
> ( let* ([x 6]
           [y (* x x)])
      (+ x y))
```

Lambda und lokale Variable

- **let** lässt sich durch **lambda** ersetzen.
- Die **let**-variante ist aber übersichtlicher.

```
> ( let ([x 40]
           [y (+ 1 1)])
      (+ x y)) => 42
```

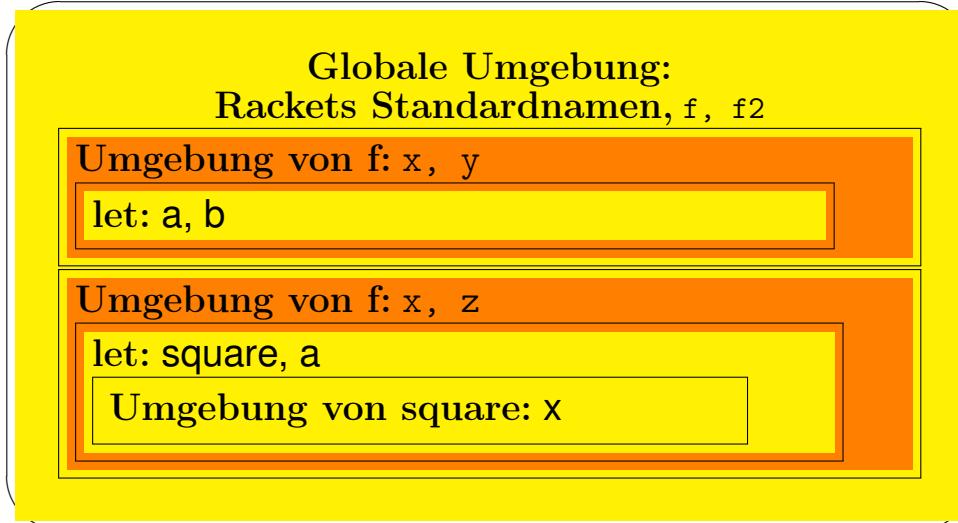
```
> (( lambda (x y)
               (+ x y))
    40 (+ 1 1)) => 42
```

Blockstruktur und Umgebungen

```
(define (f x y)
  (let ([a (/ (+ x y) 2)] ; a = (x+y)/2
        [b (* x x)]) ; b = x*x
    (* (+ a 1) (+ b 2)))) ; Resultat: (a+1)*(b+2)

(define (f2 x z)
  (let* ([square
         (lambda (y) (* y y))]
        ; lokale Funktion square
         [a (square x)]) ; a = x*x
    (if (> x 10) (+ x a)
        (- x a))))
```

Umgebungen (environments)



Die Umgebungen: In welchen Umgebungen sind in diesem Beispiel welche Namen gültig?

In allen lokalen Umgebungen gibt es Variable mit Namen `x`. Es sind jedoch völlig verschiedene Variable, die nur zufällig gleich heißen. Die in `f`, `f2` und `square` definierten Namen haben nur dort (und in den dort lokal definierten Funktionen) Gültigkeit.

In der Umgebung von `square` sind alle Namen aus der umfassenden Umgebung von `f2` gültig. Wir könnten uns beispielsweise auf `z` oder `a` beziehen. Auch das `x` aus `f2` gehört zur Umgebung von `square`, kann aber nicht referenziert werden, da es durch das lokalere `x` aus `square` *verschattet* wird.

Definition: 11 (*Umgebung*)

Eine Menge von Zuordnungen zwischen Namen und Objekten heißt Umgebung (environment).

Globale Umgebung: Die vom Racket-System vordefinierten Namen, sowie die Namen, die wir im toplevel definiert haben. Die globale Umgebung kann während der Sitzung laufend durch neue Definitionen erweitert werden.

Lokale Umgebung: Die während der Auswertung eines (Teil)-Ausdrucks lokal gültige Umgebung, die in den umschließenden Ausdrücken definiert ist.

☞ Umgebung und Abschluß

Definition: 12 (*Funktionaler Abschluß, engl. closure*)

☞ Jede Funktionsdefinition geschieht im Kontext einer aktuellen Umgebung.

☞ Die Einheit aus textueller Definition und Bindungsumgebung nennt man *funktionalen Abschluß* oder *Closure, funktionaler Abschluß*.

☞ Anmerkung: Closure is a computer science term with a precise but hard-to-explain meaning. Closures are implemented in Perl as anonymous subroutines with lasting references to lexical variables outside their own scopes. These lexicals magically refer to the variables that were around when the subroutine was defined (deep binding).

Closures make sense in any programming language where you can have the return value of a function be itself a function, as you can in Perl. Note that some languages provide anonymous functions but are not capable of providing proper closures: the Python language, for example. For more information on closures, check out any textbook on functional programming. Scheme is a language that not only supports but encourages closures. (aus Perl Programmers Reference Guide PERLFAQ7(1)) Die Grafik auf der folgenden Folie stammt aus ([Zwittlinger, 1981](#)), einer recht amüsanten Einführung in die Programmiersprache PASCAL und die strukturierte Programmierung.



Sichtbarkeit einer Variablen

Definition: 13 (*Sichtbarkeit, engl. scope*)

- Die Sichtbarkeit legt den Programmreich fest, in dem eine Variable textuell referenziert werden kann.
- Der Abschnitt eines Programms, in dem eine Variable sichtbar ist, heißt Block.
- Eine lexikalische Variable in einem inneren Block macht eine lexikalische Variable des umfassenden Blocks lokal im inneren Block unsichtbar, wenn beide Variablen denselben Namen haben. Dieses wird *Ver-schatten* genannt.

Lebensdauer



Definition: 14 (Lebensdauer, engl. extent)

Die Lebensdauer bestimmt, wie lange eine Variable während der Ausführung eines Programms referenziert werden kann.

- *Variablen der globalen Umgebung* leben unbegrenzt.
- Normalerweise *lebt* eine *lexikalische Variable* nur solange der Block aktiv ist, in dem sie eingeführt wurde.

Besonderheiten von Racket

Lebensdauer und Sichtbarkeit sind in Racket anders geregelt als in klassischen blockorientierten Programmiersprachen, wie Java oder C.

Variable: Durch closures können *lexikalische Variable* außerhalb des definierenden Blocks sichtbar sein und länger leben als der definierende Block.

Objekte: Die Lebensdauer (extent) von *Objekten* ist prinzipiell unbegrenzt. Ein Objekt (Zahl, Liste, Funktion usw.) lebt solange, wie es ein Teil einer Datenstruktur ist oder an eine Variable gebunden ist.

Aus dem Revised Report: All objects created in the course of a Scheme computation, including procedures and continuations, have unlimited extent. No Scheme object is ever destroyed. The reason that implementations of Scheme do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation. Other languages in which most objects have unlimited extent include APL and other Lisp dialects.

6 Zeichen und Symbole

6.1 Zeichen

Symbol



Alice

4 Funktionale Abstraktion
5 Variablenkopus und closures
6 Zeichen und Symbole

- Zeichen
- Symbole in Racket
- Quotierung

7 Listen

Zeichen

Zeichen: Die Basis der maschinellen Informationsverarbeitung und Nachrichtenübertragung bilden frei vereinbarte diskrete Schrift- und Zahlzeichen.

Alphabet: In der Programmierung werden die Zeichen als Worte über dem Alphabet Σ einer formalen Sprache L gebildet.

Zeichen sind neutral und haben keine Bedeutung

Beachte:

Zeichen sind zunächst neutral und haben keine Bedeutung.

- ☞ Für die Nachrichtenübertragung und Informationsverarbeitung muß den Zeichen per Vereinbarung eine *Bedeutung* zugeordnet werden.
- ☞ Für die maschinelle Informationsverarbeitung ist es wichtig, daß die Bedeutung eines Nachricht *eindeutig* und *effizient* ermittelt werden kann.

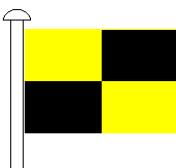
Symbole

Definition: 15 (*Symbol*)

- In der Semiotik (Bedeutungslehre der Zeichen) werden Symbole als Paar aus einem Zeichen und der Bedeutung des Zeichens definiert.
 - Die Bedeutung eines Symbols hängt von der Verwendung ab und wird frei vereinbart.
- ☞ Eine große Stärke von Lisp und Racket ist Fähigkeit zur die Symbolverarbeitung.

Mehrdeutigkeit von Zeichen

- Dasselbe Zeichen kann in unterschiedlichen Kontexten eine unterschiedliche Bedeutung haben, beispielsweise das Zeichen Kopfschütteln kann
 - einmal „ja“
 - oder einmal „nein“ bedeuten.



Die Flagge „L“ Die Flagge „L“ steht einmal für den Buchstaben „L“, aber wenn sie von Fahrzeugen des öffentlichen Dienstes gezeigt wird, bedeutet dieses die Aufforderung „Anhalten!“.

- Umgekehrt können viele Symbole dieselbe Bedeutung haben — das Morsezeichen „L“ (· – · ·), als Schallsignal oder Folge von Lichtblitzen gegeben, bedeutet ebenfalls „Anhalten“.
- Alle diese Zeichen können für die Zahl „vier“ stehen:

$$4, \quad \text{IV}, \quad \textcircled{4}, \quad \{\rightarrow \boxtimes \clubsuit \bowtie\}, \quad 3 + 1$$

Anmerkung: Die Bedeutung von Zeichen beruht vollkommen auf der Vereinbarung zwischen den Anwendern und der Konvention der Verwendung. Dieser Prozeß der Kodierung wird sehr amüsant von Lewis Carroll dargestellt (Carroll, 1951), dessen Erzählungen um „Alice im Wunderland“ viele, viele Spielereien mit Zeichen und Symbolen enthalten:

Zur Bedeutung von Zeichen



„I don't know what you mean by ‘glory’,” Alice said.

Humpty Dumpty smiled contemptuously.

„Of course you don’t — till I tell you. I meant ‘there’s a nice knock-down argument for you!’“ „But ‘glory’ doesn’t mean ‘a nice knock-down argument,’“ Alice objected.

„When I use a word,“ Humpty Dumpty said, in a rather scornful tone, „it means just what I chose it to mean — neither more nor less.“



“The question is,” said Alice, “whether you *can* make words mean so many different things.”

“The question is,” said Humpty Dumpty, “which is to be the master — that’s all.”

Alice was too much puzzled to say anything; so after a minute Humpty Dumpty began again. „...

Impenetrability! That’s what I say!”

“Would you tell me please,” said Alice, “what that means?”

“Now you talk like a reasonable child,” said Humpty Dumpty, looking very much pleased. “I meant by ‘impenetrability’ that we’ve had enough of the subject, and it would be just as well you’d mention what you mean to do next, as I suppose you don’t mean to stop here all the rest of your life.”

“That’s a great deal to make one word mean,” Alice said in a thoughtful tone.

“When I make a word do a lot of work like that,” said Humpty Dumpty, “I always pay it extra.”

‘Oh!’ said Alice. She was too much puzzled to make any other remark.



So long and thanks for all the fish. It is an important and popular fact that things are not always what they seem. For instance, on the planet earth, man had always assumed that he was more intelligent than dolphins because he had achieved so much — the wheel, New York, wars and so on — while all the time dolphins had ever done was muck about in the water having a good time. But conversely, the dolphins had always believed that they were far more intelligent than man — for precisely the same reasons.

Curiously enough, the dolphins had long known of the impending destruction of the planet earth and had made many attempts to alert mankind to the danger; but most of their communications were misinterpreted as amusing attempts to punch footballs or whistle for tidbits, so they eventually gave up and left earth by their own means shortly before the Vogons arrived.

The last dolphin message was misinterpreted as a surprisingly sophisticated attempt to do a double-backward somersault through a hoop while whistling the “Star-Spangled Banner,” but in fact the message was this: *So long and thanks for all the fish.*

In fact there was only one species on the planet more intelligent than dolphins, and they spent a lot of their time in behavioral laboratories running around inside wheels and conducting frighteningly elegant and subtle experiments on man. The fact that once again man completely misinterpreted this relationship was entirely according to these creatures’ plans.
aus (Adams, 1981, Kap. 23)

☞ **Anmerkung:** Oft werden die Begriffe „Zeichen“ und „Symbol“ nicht streng getrennt oder, je nach Schule, unterschiedlich verwendet.

- In der *Informatik* haben Zeichen als Elemente eines Alphabets nicht unbedingt alle eine Bedeutung — im Gegenteil, die Menge der sinnlosen Zeichen, die mit Computerhilfe erzeugt werden können, ist unendlich groß. Durch Kodierung werden einem Teil der verfügbaren Zeichen Begriffe zugeordnet. So entstehen Symbole, die dann eine Bedeutung tragen.
- In der *Semiotik* dagegen werden überhaupt nur solche physischen Phänomene als Zeichen betrachtet, die für Menschen etwas bedeuten, so daß ein Zeichen immer Bedeutung trägt. Der semiotische Zeichenbegriff beschränkt sich also auf solche Zeichen, die Ausdruck eines Symbols sind. Daher wird oft der Begriff „Zeichen“ wie der Symbol-Begriff verwendet.

6.2 Symbole in Racket

Symbole in Lisp und Racket

Definition: 16 (*Racket-Symbole*)

Symbole sind Objekte, die genau dann identisch sind, wenn ihr Name gleich geschrieben wird.

- Daher ist in Racket jede Variable auch ein Symbol.
- Aber auch jeder Name, den wir verwenden, identifiziert ein Symbol, ohne daß wir gleich eine Variable definieren müßten.

Funktionen für Symbole

- Die Funktion `symbol?` ist ein Typprädiat für Symbole.
- Die Funktion `eqv?` kann Symbole vergleichen.
- `string->symbol` und `symbol->string` erzeugen ein Symbol aus einem String oder wandeln den Namen eines Symbols in einen String.

Beispiele

```
> (define Racket 1)      →      ⊥
> (define java 1)        →      ⊥
> (symbol? Racket)      →      #f
> (symbol? 'Racket)     →      #t
> Racket                →      1
> 'Racket               →      Racket
> (eqv? Racket java)   →      #t
> (eqv? 'Racket 'java) →      #f
> (symbol->string 'java) →      "java"
> (string->symbol "java") →      java
> (symbol? (string->symbol "java")) →      #t
>
```

6.3 Quotierung

Symbolverarbeitung

- Anders als in imperativen Sprachen können Symbole selbst, und nicht nur ihre Werte, Argumente einer Funktion und Teil einer Datenstruktur sein.
- Wir können beispielsweise Listen oder Vektoren von Symbolen verarbeiten.
- Wir benötigen daher eine Notation, um angeben zu können,
 - ob wir ein *Symbol*
 - oder seinen *Wert* meinen.

Quotierung

Quotierung kennen wir auch in der geschriebenen natürlichen Sprache: **Auswertung**

de re: Der *Pfeiffer* ist ein frecher Lümmel.

de dictu: „*Pfeiffer*“ schreibt sich mit drei „f“.

- Die Standarfunktion **quote** blockiert die Evaluierung und gibt ihr Argument wörtlich zurück.
- **quote** kann abkürzend durch das Quotierungszeichen „`“ dargestellt werden.

Beispiele

Language: Standard (R5RS).

```
> (define pi 3.141592653589793d0) → ⊥
> pi → 3.141592653589793
> 'pi → pi
> ''pi → ''pi
> (+ 2 2) → 4
> '(+ 2 2) → (+ 2 2)
> 'Auto → auto
> Auto
```



```
reference to undefined identifier: auto
> (eval 'pi) → 3.141592653589793
> (eval '(+ 2 2)) → 4
>
```

Der Evaluator: eval

Im Interpreter-toplevel läuft die *read–eval–print–loop*, eine interaktive Schleife,

- ein prompt druckt,
- einen Ausdruck einliest,
- den Ausdruck im Kontext der aktuellen Bindungsumgebung auswertet (**eval**),
- das Ergebnis druckt.

Alle Teile dieser Schleife können durch geeignete Funktionen parametrisiert werden.

- ☞ Den evaluator **eval** können wir als Funktion direkt aufrufen, um Ausdrücke auszuwerten.

Symbole als Werte

Häufig haben wir es nicht nur mit einer einfachen Beziehung (Symbol \Leftrightarrow Wert) zu tun.

- Der Wert eines Symbols kann ein Symbol sein, dessen Wert wiederum ein Symbol ist usw.
- Wir können Symbole als Namen von abstrakten Werten betrachten, für die sie stehen.

In diesem Sinne *referenzieren* Symbole andere Größen, die wiederum andere Größen bezeichnen können.

Beispiel

Language: Standard (R5RS).

```
> (define name 'susi)      → ⊥  
> (define nameVonName 'name) → ⊥  
> nameVonName → name  
> (eval nameVonName) → susi
```



“... The name of the song is called ‘Haddock’s Eyes.’ ” “Oh, that’s the name of the song, isn’t it?” Alice said, trying to feel interested.

“No, you don’t understand,” the knight said, looking a little vexed. “That’s what the name is *called*. The name really is ‘The Aged Aged Man.’ ”

“Then I ought to have said ‘That’s what the *song* is called?’ ” Alice corrected herself. “No, you oughtn’t: that’s quite another thing! The *song* is called ‘Ways and Means’: but that’s only what it is *called*, you know!”

“Well, what *is* the song then?” Alice said, who was by this time completely bewildered. “I was coming to that,” the knight said. “The song really *is* ‘Asitting On A Gate’: and the tune’s my own invention.” **Drück mich!**

A set of small, light-colored navigation icons typically found in Beamer presentations, including symbols for back, forward, search, and table of contents.

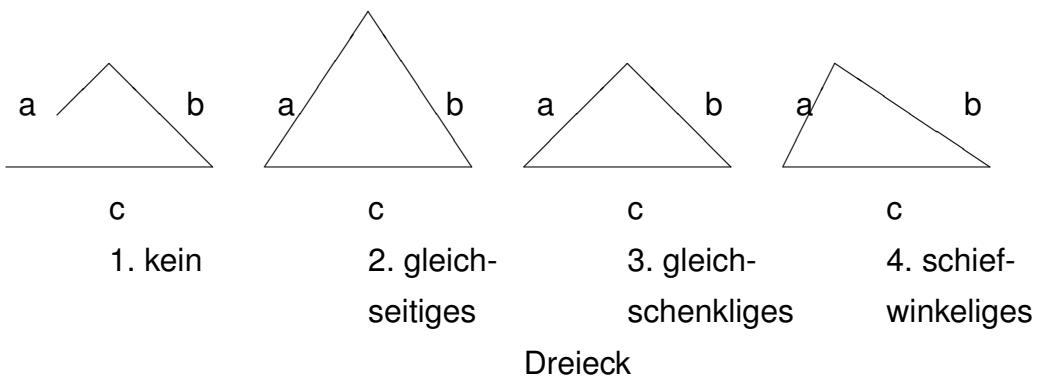
(Carroll, 1951)

Modellierung mittels Racket-Symbolen

Beispiel: 17 (The Name of the Song)

```
> (define Ways_and_Means  
      'Asitting_on_a_gate)  
     → ⊥  
> (define The_Aged_Aged_Man 'Ways_and_Means)  
     → ⊥  
> (define Haddock_Eyes 'The_Aged_Aged_Man)  
     → ⊥  
> Haddock_Eyes  
     → the_aged_aged_man  
> (eval Haddock_Eyes)  
     → ways_and_means  
> (eval (eval Haddock_Eyes))  
     → asitting_on_a_gate
```

Beispiel: Dreiecke



mit den Seitenlängen $a \leq b \leq c$

Beispiel: Dreiecke, Symbol als Konstanten

Gegeben sei ein Dreieck mit Seitenlängen $a \leq b \leq c$. Welcher Typ liegt vor?

```
; ;; a <= b <= c
(define (analyse a b c)
  (cond [(< (+ a b) c) 'kein-Dreieck]
        [(= a c) 'gleichseitig]
        [(= a b) 'gleichschenklig]
        [(= b c) 'gleichschenklig]
        [else 'schiefwinklig]))
> (analyse 1 2 2.5) → schiefwinklig
> (analyse 2 2 7) → kein-dreieck
> (analyse 1 1 1) → gleichseitig
> (analyse 1 1 2) → gleichschenklig
```

Selektionen

Wähle eine Anweisung in Abhängigkeit von einem Wert aus:

Beispiel: 18

```
(case <expr> [(<keys>) <expr1> ...]
  ...
  [(<keys>) <exprn1> ...] )
> (define Tier 'Tiger) => ⊥
> (case Tier
  [(Hund) 'wauwau]
  [(Katze Tiger) 'miau]
  [(Kuh) 'muh])      —> 'miau
```

Auswertung eines „case“

```
(case <expr> [(<keys>) <expr1> ...]
  ...
  [(<keys>) <exprn1> ...])
```

- Der Ausdruck wird der Reihe nach mit den verschiedenen Schlüssel-Werten verglichen.
- Sobald ein Vergleich erfolgreich ist, werden die Resultat-Ausdrücke der betreffenden Klausel der Reihe nach ausgewertet und der Wert der letzten s-expression als Ergebnis zurückgegeben. Zum Vergleich wird die Funktion eqv? verwendet.
- Als *catch-all* kann die letzte Klausel mit **else** beginnen.

Beispiele

```
(case (* 2 3)
  [(2 3 5 7) 'prime]
  [(1 4 6 8 9) 'composite])
  —> composite
(case (* 4 4)
  [(2 3 5 7) 'prime]
  [(1 4 6 8 9) 'composite])
  —> unspecified
(case 'm
  [(a e i o u) 'vowel]
  [(w y) 'semivowel]
  [else 'consonant])
  —> consonant
```

Namenskonventionen

Da Typdeklaration in Racket unüblich sind, sollte man die Namen so wählen, daß der Typ der Objekte deutlich wird: Folgende Konventionen haben sich durchgesetzt:

Prädikate: Der Name eines *Prädikates* (eine Funktion, die Wahrheitwerte zurückgibt), sollte mit einem Fragezeichen enden, z.B. member?, symbol?.

Modifikatoren: Der Name einer Funktion, die den Wert eines Objekts *verändert* (mutation procedure) endet mit einem Ausrufungszeichen, z.B. set-car!, string-set!.

Konversionsfunktionen: Die Namen von Funktionen, die Objekte eines Typs als Argument nehmen und ein analoges Objekt eines anderen Typs zurückgeben, enthalten “ \rightarrow ”, z.B. char \rightarrow integer, list \rightarrow string.

Teil III

Datenabstraktion in Racket

7 Datenabstraktion

7.1 Grundlagen der Typsysteme



Inhaltsverzeichnis



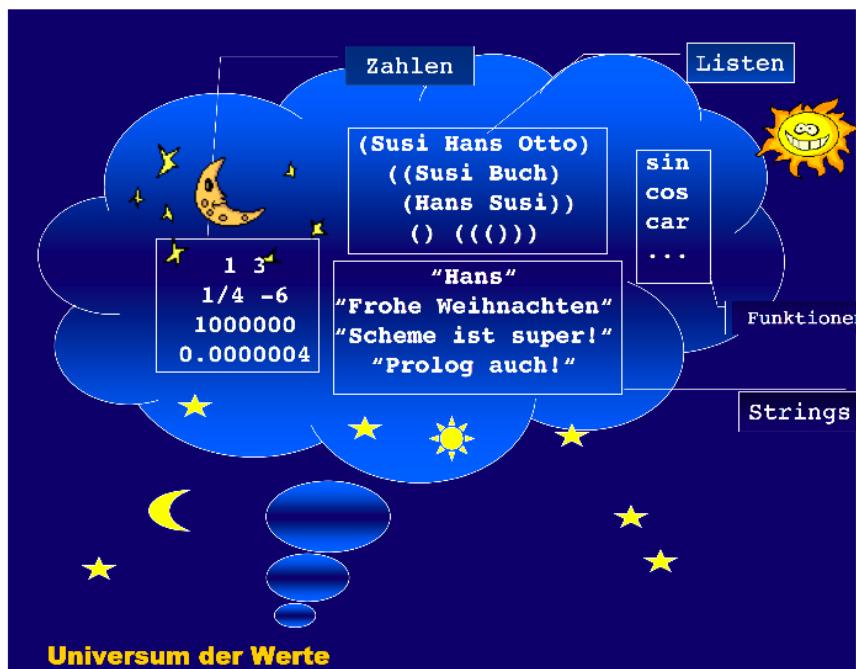
Typ der Argumente

Nicht jede Funktion ist auf jeden Wert anwendbar; Beispiel:

Language: Standard (R5RS).

```
> (define apfel 1) ; eine Zahl
> (define birne '(1, 2, apfel)) ; eine Liste
> (define banane #t) ; ein Wahrheitswert
> (+ apfel apfel)
2
> (+ apfel birne)
2
+: expects type <number> as 2nd argument,
given: (1 ,2 ,apfel); other arguments were: 1
>
```

Das Universum der Werte



Datenabstraktion:

- Wir wenden die Abstraktionsmethode auf das unendlich große Universum der möglichen Werte an und bilden *Äquivalenzklassen von Werten*, auf die dieselben Operationen (Funktionen) anwendbar sind.
- Jede solche Klasse bildet einen eigenen *Datentyp*.

Definition: 19 (*Datentyp*)

Ein *Datentyp* (kurz Typ) bestimmt

- die Menge der Werte, die eine Konstante angehört oder die von einer Variablen oder einem Term angenommen werden können, sowie die
- anwendbaren Operationen oder Funktionen auf diesen Werten. (*Wirth, 1986*, S. 20)

Klassen von Typen:

Elementare Typen: Zahlen (number), Wahrheitswerte (boolean), Zeichen (**char**).

Strukturierte Typen: Listen, Mengen, Zeichenketten, Verbunde.

Funktionstypen: Klassen von Funktionen äquivalenter Signatur, z.B. die Funktionen **sin**, **cos**, **sqrt** sind typgleich und bilden alle Zahlen auf Zahlen ab.

Polymorphe Typen: Typen von polymorphen Objekten, d.h. Objekten, die unterschiedliche speziellere Typen annehmen können, z.B. die leere Liste.

Generische Typen: Klassen von strukturgeglichen Datentypen.

Konkrete und abstrakte Datentypen

Konkrete Datentypen: Die *elementaren Typen* werden von einer Programmiersprache direkt bereitgestellt.

Für die *strukturierten Typen* gibt es

- *Konstruktoren*, mit denen wir Werte dieses Typs aus den Werten anderer Typen zu komplexen Aggregaten zusammensetzen können, und
- *Akzessoren*, mit denen wir die elementaren Bestandteile solcher Aggregate referenzieren können.

Abstrakte Datentypen: Wir können eigene Datentypen entwerfen, indem wir Akzessoren und Konstruktoren definieren. Solche Datentypen heißen *abstrakte Datentypen*.

Kardinalität eines Datentyps

Definition: 20 (Kardinalität)

- Die Mächtigkeit der Wertemenge eines Datentyps T heißt **Kardinalität** von T abgekürzt: $\text{card}(T)$.
- Satz: Bei strukturierten Typen ist die Kardinalität das Produkt der Kardinalitäten der Typen der Komponenten.
- Die Kardinalität ist ein Maß für die Menge an Speicherplatz, die für die Repräsentation der Werte dieses Typs nötig ist: Der Speicherbedarf $S(T)$ in [bit] für einen Wert vom Typ T ist mindestens

$$S(T) = \log_2(\text{card}(T)) = \text{ld}(\text{card}(T))$$

- Um die Werte eines Typs T mit $\text{card}(T) = 2$ zu repräsentieren, reicht beispielsweise ein Bit.

Typen von Funktionen

Wir können eine Funktion f als Zuordnungsregel interpretieren, die die Werte des Datentyps A eindeutig den Werten eines zweiten Typs B zuordnet.

- A heißt *Quelltyp* und B *Zieltyp* der Funktion.
- Wir schreiben: $f :: A \rightarrow B$, was bedeutet, daß die Funktion f , als Wert betrachtet, den Typ $A \rightarrow B$ hat.
- $f :: A \rightarrow B$ übernimmt Werte vom Typ A als Argument und gibt Werte vom Typ B als Resultat zurück, z.B.

`sin:: number → number` oder `odd:: number → boolean`.

Da in Racket die Typen der Argumente einer Funktion nicht explizit deklariert werden, sollten Sie an kritischen Programmstellen solche Zusicherungen als Kommentare einfügen.

Typsysteme

Definition: 21 (Latente und manifeste Typsysteme)

- Eine Programmiersprache hat ein *manifestes Typsystem*, wenn für die Variablen bei der Deklaration Typen spezifiziert werden müssen.

Manifeste Typsysteme sind typisch für die imperativen Programmiersprachen (C, Java usw.)

- Eine *Programmiersprache* hat ein *latentes Typsystem*, wenn die Typen der Variablen aus den Typen der an diese Variablen gebundenen Werte abgeleitet werden:

dynamisch zur Laufzeit des Programms wie in Racket und Common Lisp oder

statisch zur Übersetzungszeit wie in Haskell und Miranda.

Synonyme

- *Latente Typisierung* wird auch
 - *dynamische Typisierung* oder
 - “*weak typing*” genannt,
- *manifeste Typisierung* auch
 - *statische Typisierung* oder
 - “*strong typing*”.

Das Typsystem von Racket

- Die Sprachen der Lisp-Familie haben ein latentes Typsystem.
- Normalerweise deklarieren wir die Typen von Variablen in Sprachen der Lisp-Familie nicht, ganz anders als in imperativen Sprachen, wie Pascal oder C.
- In Racket hat dennoch jedes Objekt und jeder Wert sehr wohl einen wohldefinierten Typ, der die darauf anwendbaren Operationen festlegt, aber nicht unbedingt die Variablen, an die diese Werte gebunden sind.

☞ **Anmerkung:** Typdeklarationen: Gelegentlich sollten wir aber auch bei der Lisp-Programmierung die Typen der Variablen deklarieren. Diese Deklarationen dokumentieren, wie unsere Funktionen zu verwenden sind, indem sie die zulässigen Argumente formal spezifizieren. Sie sind so ein wichtiger Teil der Spezifikation unserer Software. Eine Typdeklaration ist besonders dann sinnvoll, wenn wir auf einen effizienten Code angewiesen sind. Ein guter Compiler kann den Code optimieren, wenn durch eine Deklaration die Typen der Variablen schon zur Übersetzungszeit bekannt sind.

Die Basistypen von Racket sind disjunkt.

Satz: 22 (*Disjunktheit der Typen*)

Kein Objekt erfüllt (in Racket) mehr als eins der folgenden Typpräädikate:

boolean?	pair?
symbol?	number?
char?	string?
vector?	port?
procedure?	

7.2 Polymorphie

Generische Funktionen

Manche Funktionen besitzen mehrere Typen – sie sind *polymorph* (*vielgestaltig*):

- Beispielsweise die *Identitätsfunktion* `id`, die einen beliebigen Wert auf sich selbst abbildet, ist für Elemente jeden Datentyps anwendbar:
- Solche Funktionen heißen *generische Funktionen*.

```
> (define (id x) x)      → ⊥
> (id (+ 2 3))          → 5
> (id #t)                → #t
> (id id)               → #<Procedure ID>
```

Typvariable

- Um den Typ einer generischen Funktion spezifizieren zu können, verwendet man *Typvariable*, die für einen festen, aber unbekannten Typ stehen.
- Typvariable werden mit *griechischen Buchstaben* bezeichnet.
- Gleiche Buchstaben bezeichnen denselben unbekannten Datentyp.

```
odd  ::  number → boolean
id   ::  α → α
equal? ::  α, β → boolean
```

Formen von Polymorphie

Strukturgleichheit: Die Struktur ist gleich, aber der Basistyp unterschiedlich: Tabellen, Funktionen...

Spezialisierung, Generalisierung: Polymorphe Werte haben oft eine *Hierarchie von Typen*:

Ein Sperling ist ein

- Fink,
- ein Singvogel,
- ein Vogel,
- ein Wirbeltier,
- ein Tier,
- ein Lebewesen usw.

In Racket sind die Zahlen polymorph: Eine natürliche Zahl ist sowohl eine Rationalzahl als auch eine komplexe Zahl.

8 Elementare Datentypen

Datenstrukturen



Inhaltsverzeichnis

8.1 Zahlen

Zahlen (number)

- In Racket gibt es eine Hierarchie von Typen für Zahlen: *number*, *complex*, *real*, *rational*, *integer*.
- Für jeden Zahlentyp gibt es ein Typprädiat zum prüfen, ob ein Objekt ein Wert von diesem Typ ist: *number?*, *complex?*, *real?*, *rational?*, *integer?*.
- Der Wert der größten *integer*-Zahl ist implementationsabhängig. Von der Sprachdefinition her ist keine Beschränkung vorgesehen.

Beispiele

(complex? 3+4i)	→ #t
(complex? 3)	→ #t
(real? 3)	→ #t
(real? -2.5+0.0i)	→ #t; Imaginärteil=0
(real? #e1e10)	→ #t; exakte Zahl 10^{10}
(rational? 6/10)	→ #t
(rational? 6/3)	→ #t
(integer? 3+0i)	→ #t
(integer? 3.0)	→ #t
(integer? 8/4)	→ #t

Standardfunktionen auf Zahlen

(+ m n)	Addition von m und n
(- m n)	Subtraktion von m und n
(* m n)	Multiplikation von m und n
(/ m n)	Division von m durch n
(abs n)	Absolutbetrag von n
(ceiling n)	Der kleinste Integer-Wert $\geq n$
(floor n)	Der größte Integer-Wert $\leq n$
(round n)	Runden zum nächsten integer-Wert.
(truncate n)	Abschneiden der Nachkommastellen
(expt n k)	n^k
(sqrt n)	\sqrt{n}
(min n ...)	Das Minimum der Argumente
(max n ...)	Das Maximum der Argumente
(exp n)	e^n
(log n)	$\log_e n$
(sin n)	$\sin(n)$
(cos n)	$\cos(n)$
(asin n)	$\arcsin(n)$
(acos n)	$\arccos(n)$
(tan n)	$\tan n$
(atan n)	$\arctan n$
(quotient n k)	ganzzahlige Division von n und k
(remainder n k)	Rest der ganzzahligen Division von n und k mit dem Vorzeichen von n
(modulo n k)	Rest der ganzzahligen Division von n und k mit dem Vorzeichen von k

Rackets Standardoperationen auf Zahlen

>	<	=	<=	>=
+	-		*	
quotient	remainder		modulo	
max	min		abs	
numerator	denominator		gcd	
lcm	floor		ceiling	
truncate	round		rationalize	
expt				

Exakte und inexakte Zahlenformate

Racket kennt zwei Arten, um Zahlen zu repräsentieren:

Exakte Repräsentation: Präfix #e

Ganze Zahlen und Rationalzahlen werden in der exakten Darstellung so repräsentiert, daß bei arithmetischen Operationen keine Rundungsfehler und Wertebereichsüberläufe auftreten.

Inexakte Repräsentation: Präfix #i

Speicherökonomische Repräsentation, bei der Rundungsfehler auftreten können.

```

> (exact? 1.0)      → #f
> (exact? #e1.0)    → #t
> (exact? (/ 1 3)) → #t
> (/ 1 3)          → 1/3

```

Beispiel: Fakultät

```

00000000000000000000000000000000
> (exact->inexact (fak 200)) → +inf.0 ; +∞
> (exact->inexact (fak 100))
→ 9.332621544394415e+157

```

Genauigkeit: Da die Zahlendarstellung nicht immer exakt ist, sind die errechneten Werte mathematisch gleichwertiger Ausdrücke nicht immer gleich. Beispielsweise muß nicht immer gelten, daß $(x*y)/y = x$ ist. Wir können uns weder auf das Assoziativgesetz noch das Distributivgesetz verlassen, da es bei großen Zahlen zu Auslöschenungen kommen kann. Bei der Addition von sehr großen zu sehr kleinen Zahlen kann es zu Auslöschenungen kommen:

Obwohl für die Addition das Assoziativgesetz gelten sollte, hängt das Ergebnis der Addition von der Klammerung ab. Dieser Effekt ist immer dann kritisch, wenn wir in einem Term gleichzeitig sehr große und sehr kleine Operanden haben. Wir müssen beim Programmieren also wissen, welche Klammerung angenommen wird, wenn wir selbst keine spezifiziert haben. Dank der Präfixnotation gibt es aber in Racket meistens keine Mehrdeutigkeiten. In einer Programmiersprache dagegen, die Infix-Notation verwendet (Java, C, Miranda), müssen Sie genau die Assoziativität der Operatoren und die Vorrangregeln beachten.

Zur Genauigkeit

```
(define (hoch x n)
  (if (= 0 n) 1
      (* x (hoch x (- n 1)))))
(define zehnHoch100 (hoch 10 100))
```

```
> zehnHoch100 →  
1000000000000000000000000000000000000000000000000  
0000000000000000000000000000000000000000000000000  
000000000000000000000000000000000000000000000000
```

```
(define zehnHoch100inexact  
  (exact->inexact zehnHoch100))  
> zehnHoch100inexact → 1e+100
```

Auslöschen

```
> (+ 1
     zehnHoch100
     (- zehnHoch100))
1
> (+ 1
     zehnHoch100inexact
     (- zehnHoch100inexact))
0.0
>
```

☞ Anmerkung: In Standard-Scheme R5RS, R6RS, Racket und Swindle sind die IEEE-Zahlen `+inf.0`, `-inf.0` und `+nan.0` definiert. Diese Symbole stehen für die real-Zahlen $+\infty$, $-\infty$ und *not a number* und können bei real-Operationen als Resultat einer Division durch Null auftreten. Für Zahlen vom Typ **rational** ist die Division durch Null nicht zulässig.

Spezielle Zahlen: ∞

Sprache: **Module**; memory limit: 128 megabytes.

> (/ 1.0 0.0) → +inf.0 ; +∞

$\geq (-1.0 \ 0.0) \rightarrow -\infty$

> (/ 0.0 0.0) → +nan.0 ; not a number

$\geq (\text{ / } 1 \text{ } 0) \rightarrow \text{ /: division by zero}$

```
> (< (/ -1.0 0.0) (/ 1.0 0.0))
```

→ #t

```
> (+ (/ 1.0 0.0) (/ -1.0 0.0))
```

→ +nan.0

```
> (+ (/ 1.0 0.0) (/ 1.0 0.0))
```

→ +inf.0

```
> (+ (/ -1.0 0.0) (/ -1.0 0.0))
```

→ -inf.0

>

8.2 Wahrheitswerte

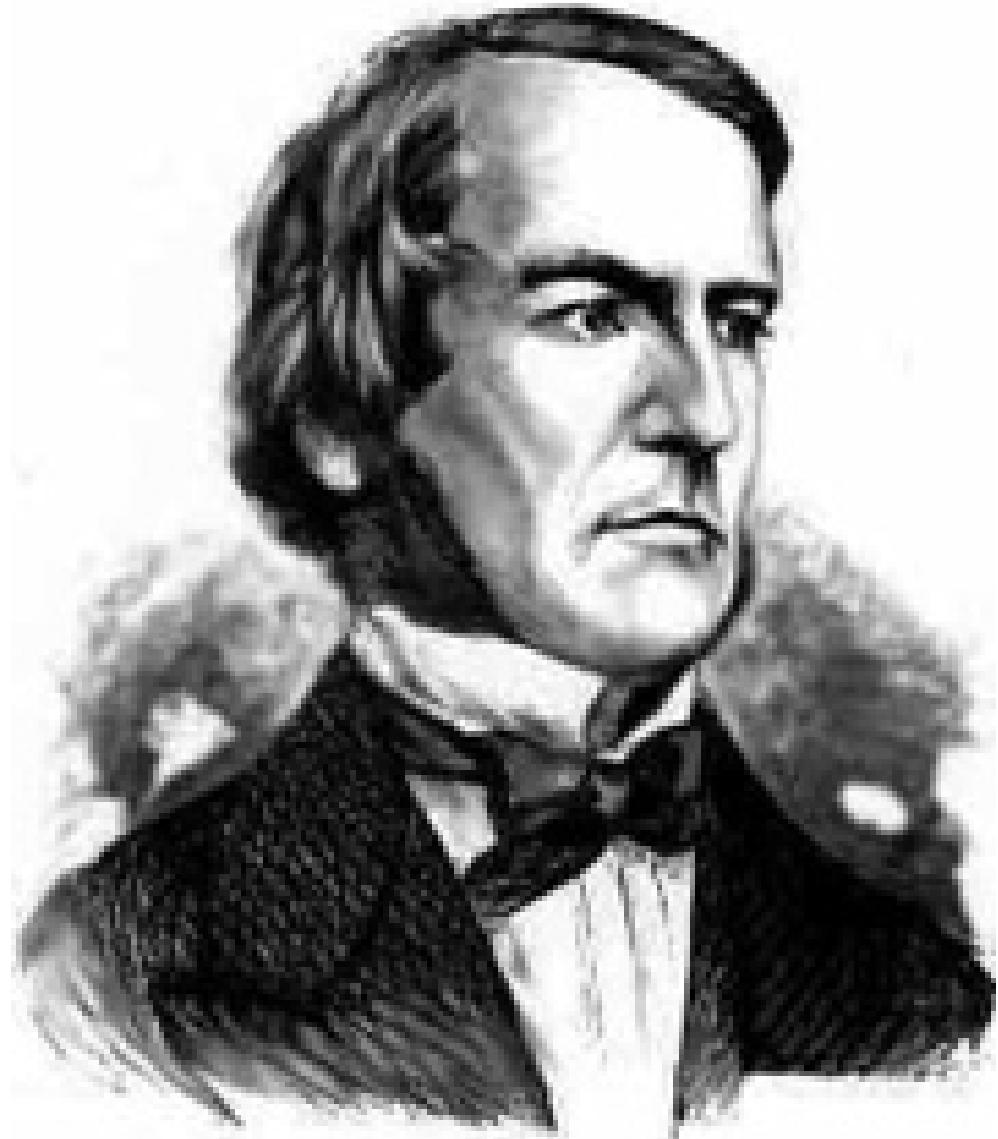
Die Vergleichsoperatoren

- Die Vergleichsprädikate $>$, \geq , \leq , $<$ definieren (partielle) Ordnungsrelationen und eine Äquivalenzrelation $=$ über den Wertemengen der Zahlen.
- Auf nicht-numerische Datentypen sind diese Funktionen nicht anwendbar!
Für andere Datentypen gibt es spezielle Ordnungsprädikate und Vergleichsfunktionen.

```
> (< 1 3 5)  →#t
> (< "auto" "Auto")

Error: Bad arg. type in: (< "auto" "Auto")
> (string<? "auto" "Auto")  →#f
```

Der Datentyp boolean



George Boole

- Wahrheitswerte bilden einen Datentyp von nur zwei Elementen, mit den zwei kanonische Repräsentationen:

#t, #f; card (boolean) = 2.

- Der Datentyp heißt boolean, nach dem Mathematiker George Boole.
- Funktionen, die als Resultat einen Wahrheitswert liefern, heißen *Prädikate*
- Wahrheitswerte sind das Resultat von Vergleichsoperationen oder logischen Operationen \neg, \vee, \wedge .

Besonderheiten von Racket

- In bedingten Ausdrücken **cond**, **if**, **and**, **or**, **not** kann jedes Objekt als Wahrheitwert verwendet werden.
- Nur **#f** wirkt als Wahrheitwert *falsch*, jedes andere Objekt wird als **#t** gewertet.
- Achtung: In anderen Lisp-Dialekten wird gelegentlich auch die leere Liste oder das Symbol **nil** als **#f** gewertet, das ist in Racket nicht so!
- **#t** und **#f** evaluieren zu sich selbst und erfordern kein quote.

Beispiele

#t	→ #t
#f	→ #f
' #f	→ #f
(not #t)	→ #f
(not 3)	→ #f
(not (list 3))	→ #f
(not #f)	→ #t
(not '())	→ #f
(not (list))	→ #f
(not 'nil)	→ #f

Beispiel

- > (**if** 'auto 'fahren 'laufen) → fahren



- > (**if** (**not** 'auto) 'fahren 'laufen) → laufen



Elementare Vergleichsprädikate

```
> (= 2 3)           → #f
> (= #t #f)

  Error: Bad argument type ...

> (equal? #t #t)   → #t
> (equal? #t #f)   → #f
> (equal? (> 3 2)
           (> 7 6))   → #t
> (define x 6)
> (and (< x 10)
        (> x 4))   → #t
> (not (= x 3))   → #t
> (or (= x 2)
       (< x 8))   → #t
> #t               → #t
> #f               → #f
> '#f              → #f
> (equal? #f '#f)  → #t
```

= :: number, number → boolean
equal? :: α, β → boolean

- Die Vergleichsfunktion equal? ist polymorph und kann auf Werte beliebigen Typs angewendet werden.
- Die Vergleichsfunktion “=” dagegen ist nur auf Zahlen anwendbar.
- Die Konstanten #t, #f evaluieren zu sich selbst und brauchen nicht quotiert zu werden.

Die logischen Operatoren:

- Racket kennt die drei logischen Funktionen \neg , \wedge , und \vee zur Verknüpfung von Wahrheitswerten.
- Da diese mathematischen Zeichen üblicherweise nicht auf der Tastatur vorkommen, werden die folgenden Ersatzdarstellungen verwendet:

Operation	mathematisch	in Racket
Konjunktion	\wedge	and
Disjunktion	\vee	or
Negation	\neg	not

Die logischen Operatoren sind *special form operators* und *nicht strikt*. Zur Stricktheit siehe Definition 47

Verknüpfungstabellen

p	q	(and p q)	(or p q)	(not p)
#f	#f	#f	#f	#t
#f	#t	#f	#t	#t
#f	\perp	#f	\perp	#t
#t	#f	#f	#t	#f
#t	#t	#t	#t	#f
#t	\perp	\perp	#t	#f
\perp	*	\perp	\perp	\perp

Zur Auswertung logischer Ausdrücke

- Die Argumente werden von links nach rechts ausgewertet. Die Auswertung wird beendet, sobald der Resultatwert feststeht.
- Die nicht-ausgewerteten Argumente können undefiniert sein.
 - Die Disjunktion (**or** a₁ ... a_n) ist wahr, sobald ein Argument zu **#t** evaluiert, **#f** sonst.
 - Die Konjunktion (**and** a₁ ... a_n) ist falsch, sobald ein Argument zu **#f** evaluiert, **#t** sonst.
 - Die Negation negiert den Wahrheitswert.

Anmerkung: Racket reduziert die logischen Operatoren wie andere special form expressions auch nach der Strategie der *verzögerten Auswertung* (*lazy evaluation*). Das Gegenstück ist die *vorgezogene Auswertung* (*eager evaluation*), wie sie für Funktionen angewendet wird. (siehe Kapitel „Semantik“).

Beispiele

```
> (define x 0)           → ⊥
> (and (> x 0)
     (> (/ 1 x) 0.2))   → #f
> (define xx 3)          → ⊥
> (and (> xx 0)
     (> (/ 1 xx) 0.2))  → #t
> (define xxx 'ein-symbol)
                           → ein-symbol
> (and (number? xxx) ; Reihenfolge wichtig!
         (< xxx 7))      → #f
> (and (< xxx 7)(number? xxx))
```



Error: Bad argument type

Beispiel: Schaltjahre

Beispiel: 23 (Wann ist y ein Schaltjahr? Variante 1:)

```
(define (leap? y)
  (cond [(= 0 (remainder y 100))
            (= 0 (remainder y 400))]
            ; durch 400 teilbar
            [else
             (= (remainder y 4) 0)]))
```



```
> (leap? 2000) → #t
> (leap? 1999) → #f
> (leap? 1000) → #f
> (leap? 1964) → #t
> (leap? 1963) → #f
```

Anmerkung: Wenn die Jahreszahl glatt durch hundert teilbar ist, ist das Jahr nur ein Schaltjahr, sofern die Jahreszahl auch durch 400 teilbar ist. Sonst ist alle vier Jahre ein Schaltjahr.

Kalender und Schaltjahre: Ein Kalender ist eine Einteilung der Zeit nach astronomischen Gesichtspunkten. Es gibt viele Kalender, doch alle benutzen bestimmte, natürliche Zeitabschnitte als Grundlage.

Der kleinste natürliche Zeitabschnitt aller Kalender ist der Tag. Der nächste größere Zeitabschnitt ist der synodische Monat, die Zeit zwischen zwei gleichen Mondphasen, der bis auf wenige Ausnahmen allen alten Kalendern zugrunde gelegt wurde.

Der synodische Monat umfaßt aber keine ganze Zahl von Tagen, sondern 29,5306 Tage. Man läßt daher Monate unterschiedlicher Länge aufeinander folgen, wobei die nächstliegende Einteilung diejenige in Monate mit abwechselnd 29 und 30 Tagen ist. 12 solcher Monate, je 6 zu 29 und 30 Tagen ergeben einen Zeitraum von 354 Tagen, wogegen ein Mondjahr aus zwölf synodischen Monaten 354,367 Tage umfaßt.

In einem Mondkalender sind also Schalttage nötig. Das Mondjahr ist unabhängig vom Sonnenlauf (etwa 11 Tage kürzer) und der Jahresbeginn wandert durch die Jahreszeiten.

Sonnenkalender orientieren sich an der Wiederkehr der Jahreszeiten, dem tropischen Jahr von 365,2422 Tagen, dem Zeitraum zwischen zwei Durchgängen der Sonne durch den Frühlingspunkt. Das Sonnenjahr ist auch Grundlage unseres Kalenders, des *Gregorianischen Kalenders*. Beim Vorläufer dieses Kalenders, dem *Julianische Kalender*, gab es alle vier Jahre ein Schaltjahr mit 366 Tagen, so daß daraus eine mittlere Jahreslänge von 365,25 Tagen resultierte. Da das Julianische Jahr gegenüber dem tropischen Jahr etwas zu lang war, verschob sich der Jahresbeginn gegenüber den Jahreszeiten. Bis zum 16. Jahrhundert war dieser Fehler auf 10 Tage angewachsen. Papst Gregor XIII ordnete eine Kalenderreform an, wobei zunächst auf den 4. Oktober 1582 unmittelbar der 15. Oktober folgte. Die alte Schaltregel, nach der alle vier Jahre ein Schaltjahr mit 29 Februartagen auftritt, wurde durch den Zusatz ergänzt, daß alle die Jahre mit einer vollen Jahrhundertzahl, bei der die Division der Jahreszahl mit 400 nicht aufgeht, als Gemeinjahr zu zählen seien (1900, 2100). Die mittlerer Länge des gregorianischen Jahres ist damit 365,2425 Tage. Erst seit etwa 607 u. Z. werden die Jahre „vor Christi Geburt“ bzw. „nach Christi Geburt“ gezählt, heute vielfach auch „vor unserer Zeitrechnung“ bzw. „unserer Zeitrechnung“ (abg. v. u. Z., u. Z.).

Eine geniale Konstruktion ist der alte chinesische Kalender, der mittels Schaltmonaten das Sonnenjahr und Mondjahr in Einklang bringt.

Informatiker und Informatikerinnen sollten das *Julianische Datum*, abgekürzt J.D., kennen, da es das Rechnen mit Daten sehr erleichtert. Dieser einfachste aller Kalender wurde von J. Scaliger 1582 vorgeschlagen und besteht einfach in einer fortlaufenden Numerierung der Tage. So lassen sich leicht Zeitdifferenzen berechnen, die sich über mehrere Jahre erstrecken. Tagesbeginn ist dabei 12^h Weltzeit. Gezählt wird ab 1. Jan. 4713 v.u.Z. Der 1. Jan. 1970, 13^h MEZ hat das J.D. 2 440 588,00 und der 1. Jan. 1980 das J.D. 2 444 240,00. (Weigert and Zimmermann, 1971)

Beispiel: Schaltjahre

Beispiel: 24 (Wann ist y ein Schaltjahr? Variante 2:)

Der bedingte Ausdruck kann durch logische Operatoren ersetzt werden:

```
(define (leap2? y)
  (or
    (and (= (remainder y 4) 0)
          (not (= (remainder y 100) 0)))
         (= (remainder y 400) 0)))
  > (leap2? 2000) → #t
```

8.3 Der Datentyp „char“

Der ASCII-Code, 7-bit und 8-bit:

0	NUL	32	SP	64	Ø	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	,	71	G	103	g
8	BS	40	(72	H	104	h
9	HT	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Ä	196
Ö	214
Ü	220
ß	223
ä	228
ö	246
ü	252

Zeichen und Zeichenketten

- Damit Programme auch Texte verarbeiten und erzeugen können, gibt es den Datentyp **char**.
- Dieser Datentyp repräsentiert die Zeichen des Alphabets des 7-Bit-ASCII-Zeichensatzes².
- Der Datentyp **string** (Zeichenketten) ist ein strukturierter Datentyp, der Folgen von Zeichen, die Worte über diesem Alphabet repräsentiert.

Hiermit können ganze Dokumente (z.B. Programme) eingelesen, repräsentiert und verarbeitet werden.

Zu den Zeichen gehören die

sichtbaren druckbaren Zeichen: Buchstaben, Ziffern, Interpunktionszeichen usw.

Formatierungszeichen: Leerzeichen (SP), Zeilenvorschub (LF), Wagenrücklauf (CR), Seitenvorschub (FF), Glöckchen (BEL)

Steuerzeichen: DC1, ACK, NAK, ... Diese dienen der Kommunikation mit dem E/A-Gerät und werden von den Gerätetreibern interpretiert.

Werte vom Typ char

- Werte vom Typ **char** werden durch den Präfix #\ kenntlich gemacht.
 - #\ a bedeutet das ASCII-Zeichen „a“,
 - a die Variable a,
 - 'a das wörtliche Symbol a.
- Einige Formatierungszeichen sind als Konstanten vordefiniert:
 - #\space (Leerzeichen) und
 - #\newline (Neue Zeile).

Codierung der Zeichen

Jedem ASCII-Zeichen ist ein eindeutiger Code-Wert zugeordnet. Diese können wir nutzen,

- um eine Ordnung auf den Zeichen zu definieren,

²Auf den Sun-Workstations auch des 8-Bit-ASCII-Zeichensatzes

(**char<?** #\A #\Z) → **#t**

- um Zeichen zu erzeugen, die nicht auf der Tastatur einzugeben sind, z.B. BEL=(integer->char 7),
- um die Arten von Zeichen — Ziffern, Buchstaben usw. zu unterscheiden:

(**char>?** #\A #\1) → **#t**

- ASCII-Zeichen können direkt durch die dreistellige Oktalzahl des ASCII-Codes angegeben werden.

#\077 → #\?

Operationen auf Zeichen

(char->integer x): Der ASCII-Code des Zeichens

(integer->char x): Das ASCII-Zeichen mit code x

```
> (char->integer #\A) → 65
> (char->integer #\077) → 63
> (char->integer #\?) → 63
> (char->integer #\8) → 56
> (char->integer #\newline) → 10
> (char? #\A) → #t
> (char? 'a) → #f
```

char->integer ist die *Umkehrfunktion* zu integer->char.

```
> (integer->char
    (char->integer #\A)) → #\A
> (integer->char
    (+ 1
       (char->integer #\A))) → #\B
```

Unicode-Zeichen

Unicode-Zeichen werden durch #\u, gefolgt von vier Hexadezimalzahlen, angegeben.

- Ordnungsrelationen und Groß/Kleinwandlung wirken wie bei den ASCII-Zeichen.

```
> #\u03BB → #\λ
> #\λ → #\λ
(integer->char
  (+ 1 (char->integer #\μ))) → #\ν
> "30,00\20AC" → "€30,00"
```

Funktionen auf Zeichen

```
> (char-downcase #\A) → #\a
> (char-upcase #\?) → #\?
> (char-downcase
  (char-upcase #\?)) → #\?
> (char-alphabetic? #\A) → #t
> (char-numeric? #\0) → #t
> (char-whitespace? #\newline) → #t
```

9 Listen

Teil II

Konstrukte der funktionalen Programmierung

- 4 Funktionale Abstraktion
- 5 Variablenkopus und closures
- 6 Zeichen und Symbole
- 7 Listen

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 180 / 1157



- 4 Funktionale Abstraktion
- 5 Variablenkopus und closures
- 6 Zeichen und Symbole
- 7 Listen
 - Grundoperationen auf Listen
 - Listen als rekursive Datenstrukturen
 - Listen mit Symbolen

Navigation icons: back, forward, search, etc.

9.1 Grundoperationen auf Listen

Listen

Definition: 25 (*Liste*)

Eine Liste ist eine linear geordnete Sammlung von Werten oder Objekten.

In Racket können die Elemente einer Liste Objekte von unterschiedlichem Typ sein.

- Sind alle Elemente vom gleichen Typ und betrachtet man die Elemente des Basistyps als Zeichen eines Alphabets Σ , so lassen sich Listen mathematisch als Worte über Σ beschreiben.
- Eine andere nützliche mathematische Sicht auf Listen ist es, diese als Mengen zu betrachten (nur, wenn keine Elemente doppelt vorkommen).
- Für beide Interpretationen gibt es in Racket Standardfunktionen.

Konstruktion von Listen

Endliche Listen können

- durch Aufzählung der Elemente direkt notiert
- oder mittels der Funktion **list** konstruiert werden:

'(1 2 3) ; eine Liste von Zahlen	→ (1 2 3)
(list 1 'bus (* 2 21) '(1 2)) ; Unterlisten	→ (1 bus 42 (1 2))
(length '(4 62 3 42))	→ 4
() ; die leere Liste	→ ()
(length '())	→ 0
(null? '(1 2 3))	→ #f ; Liste leer?
(null? '())	→ #t

Die leere Liste

- Die leere Liste `'()` ist *polymorph*, denn sie ist leer im Hinblick auf jeden speziellen Typ.
- Die Liste `'(())` dagegen ist nicht leer, da sie das Element `()` enthält; sie ist also eine Liste der Länge 1.
- Wenn Sie Listen direkt durch Aufzählung der Werte notieren, ist das quote-Zeichen wichtig, sonst wird die Liste als funktionaler Ausdruck gelesen.
- Die leere Liste hat einen singulären Datentyp: Sie erfüllt keins der Basistyppräädikate:

`boolean? pair? symbol? number?`
`char? string? vector? port? procedure?`

Quotierung

- Wenn Sie Listen direkt durch Aufzählung der Werte notieren, ist das quote-Zeichen wichtig, sonst wird die Liste als funktionaler Ausdruck gelesen.

```
> (sin 3)           → 0.1411200080598672
    ; Aufruf der Funktion sin
> '(sin 3)          → (sin 3)
    ; Liste mit den Elementen sin und 3
> (length '(sin 3)) → 2
```



Auch in Racket müssen die öffnenden und schließenden Klammern immer gut aufeinander abgestimmt sein!

Konkatenation von Listen: append

Zwei Listen können mit der *append*-Funktion zu einer längeren Liste zusammengesetzt werden.

> (*append* '(1 2) '(3 4)) → (1 2 3 4)

Satz: 26 (Neutrales Element)

Werden die Listen als Worte über einem Alphabet Σ betrachtet, dann entspricht diese Operation der Zusammensetzung mit dem neutralen Element „'()“.

$$(\text{append} \ xs \ '()) = (\text{append} \ '() \ xs) = xs$$

Assoziativität

Satz: 27 (Assoziativität von *append*)

Die Konkatenation ist assoziativ:

$$\begin{aligned} & (\text{append} \ (\text{append} \ xs \ ys) \ zs) \\ &= (\text{append} \ xs \ (\text{append} \ ys \ zs)) \end{aligned}$$

```
> (define xs '(2 3 4))
> (define ys '(a b c))
> (append xs '())
→ (2 3 4)
> (equal? (append xs '()))
      (append '() xs))
→ #t
> (append xs ys)
→ (2 3 4 a b c)
```

Mengentheoretische Interpretation

- Wenn wir zwei Listen m_1 und m_2 als Mengen M_1 und M_2 interpretieren, dann entspricht die **append**-Operation der *Vereinigung* zweier Mengen:

$$(\text{append } m_1 \ m_2) \equiv M_1 \cup M_2$$

- Die Standardfunktion (**member** x xs) testet, ob ein Wert x Element der Liste ist, also $x \in xs$.

Länge einer Liste \equiv Mächtigkeit der Menge

- Die Länge einer endlichen Liste ist die Zahl der Elemente. Die Standardfunktion **length** bestimmt die Länge einer Liste.

Semiprädikate

- **Semiprädikate** sind Prädikate, deren Resultat wie ein Wahrheitswert verwendet werden kann, aber nicht unbedingt vom Typ boolean ist.

- Die Funktion **member**

- gibt **#f** zurück, wenn das gesuchte Element nicht Element der Liste ist,
- andernfalls die Teilliste, ab der das Element zum ersten Mal in der Liste auftaucht.

```
> (define xs '(1 2 3 a 4))
> (member 5 xs)           → #f
> (member 'a xs)          → (a 4)
> (if (member 'a xs) 'a '()) → a
> (member '(1 2) '(1 2 3)) → #f
> (member '(1 2)
           '( 1 (1 2) 3))   → ((1 2) 3)
```

Elementweise Konstruktion einer Liste

- Die Funktion „**cons**“ fügt am Anfang einer Liste ein Element ein:

```
> (cons 1 '())
> (cons 1
           (cons 1 '()))
> (cons 1
```

```
(cons 1
      (cons 1
            '()))
> (pair? (cons 1 '()))
      → #t
```

- Die Aufzählung der Elemente in runden Klammern ist nur eine syntaktische Kurzform für die elementweise Konstruktion mit “**cons**”.
- Syntaktisch heißt das Resultat eines **cons** “pair”, ein *Paar*.

Paare

- Die **cons**-Funktion ist ein allgemeiner Konstruktor für *Paare* - der zweite Parameter muß nicht unbedingt eine Liste sein.

```
> (cons x y)           → ( x . y )
> (pair? '( x . y ))  → #t
```

- Paare, deren zweites Element eine Liste ist, werden in der vereinfachten Listennotation dargestellt — syntaktischer Zucker.

```
> '(a . (b . (c . (d . ()))))
      → (a b c d )
```

- Der *dot-Operator* „.“ ist eine Infix-Notation für den cons-Operator.

Paare versus Listen

- Auch eine Liste mit nur einem Element ist ein Paar:

```
> (pair? '(a)) → #t
> (equal? '(a) (cons 'a '()))
> (equal? '(a) '( a . ())) → #t
```

- Ein Paar ist nur dann eine Liste, wenn das zweite Element des Paares eine Liste ist.

```
> (list? (cons 'a '())) → #t
> (list? (cons 'a 'b)) → #f

> (cons 'a (cons 'b '())) → (a b)
> (list? (cons 'a (cons 'b '())))) → #t

> (cons 'a (cons 'b 'c)) → (a b . c)
> (list? (cons 'a (cons 'b 'c)))) → #f
```

Zerlegung einer Liste

Die Akzessoren *car* und *cdr* geben Zugriff auf die Bestandteile einer Liste:

- *car* liefert den *Kopf* der Liste, das erste Element,
- und *cdr* den *Schwanz (Körper)* der Liste, die Liste aller Elemente außer dem ersten.
- Es gelten die Bedingungen:

$$\begin{aligned} (\text{car } (\text{cons } x \ xs)) &= x \\ (\text{cdr } (\text{cons } x \ xs)) &= xs \\ (\text{car } '()) &= (\text{cdr } '()) = \perp \\ xs &= (\text{cons } (\text{car } xs) \ (\text{cdr } xs)) \end{aligned}$$

Akzessoren

Kurzform	steht für
(caar xs)	(car(car xs))
(caaar xs)	(car(car(car xs)))
(caaaaar xs)	(car(car(car(car(xs))))))
(cddr xs)	(cdr(cdr xs))
(cdddrr xs)	(cdr(cdr(cdr(xs))))
(cdddddr xs)	(cdr(cdr(cdr(cdr(xs))))))
(cadr xs)	(car(cdr xs))
(cdadr xs)	(cdr(car(cdr xs)))

Historische Randbemerkung: car und cdr Die ungewöhnlichen Namen *car* und *cdr*. stammen noch von John McCarthys erster Implementation von Lisp auf einer IBM 704 und sind die Namen der Maschinenbefehle, mit denen das Paar aus Kopf und Schwanz dereferenziert wurde. Listen wurden (und werden meistens) so implementiert, dass jeder Aufruf von *cons* eine Datenstruktur anlegt, eine sogenannte *cons-Zelle*, die Referenzen auf den Kopf und Schwanz enthält. In der ersten Lisp-Implementation war eine *cons-Zelle* genau ein Speicherwort, und je ein Halbwort davon enthielt die Adresse von Kopf und Schwanz. Die Maschinenbefehle *car* (*contents of address register*) und *cdr* (*contents of decrement register*) lieferten sehr effizient genau die Teile der Liste.

Es gab, und gibt immer noch, eine ganze Familie von Funktionen mit solchen kryptischen Namen:

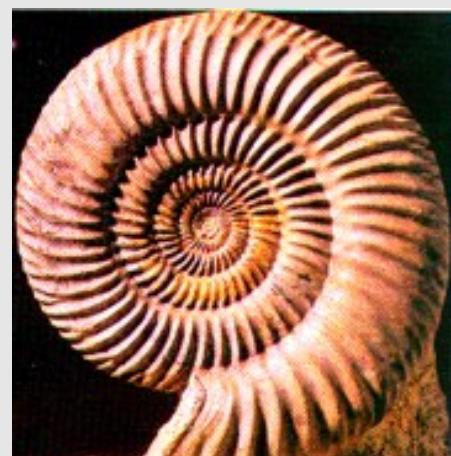
(cadr y): kurz für (car (cdr y)), entspricht dem 2. Element der Liste.

(caddr y): kurz für (car (cdr (cdr y))), dem 3. Element.

cxr: x ist eine beliebige Folge von bis zu vier as oder bs, wobei jedes a für einen Aufruf von car und b für cdr steht.

(cddddr y) ist dann (cdr (cdr (cdr (cdr y))))

In Common Lisp und Racket gibt es jetzt äquivalente Standardfunktionen mit treffenderen Namen wie first, second usw., aber die traditionellen Funktionsnamen leben in friedlicher Koexistenz weiter, denn die Mehrheit der Lisp-Gemeinde in Scheme hat sich noch nicht von den traditionellen Namen trennen können.



Das Gehäuse eines Ammoniten hat eine rekursive, listenartige Struktur. Die Abbildung zeigt einen Ammoniten aus Frankreich (Parkinsonia parkinsoni, 15cm, Jura-Bajoc, Bayeux, Frankreich).

Hier ist ein Film, der die rekursive Konstruktion zeigt:
<http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/SE3-Movies/Pages/Ammonit.html>

9.2 Listen als rekursive Datenstrukturen

Listen als rekursive Datenstrukturen

Wenn wir



- ▶ die „**cons**“-Funktion zur Konstruktion von Listen verwenden,
- ▶ und die Standardfunktionen car und cdr für den Zugriff,

können wir den Datentyp Liste **rekursiv** definieren:

Definition (Liste)

Eine Liste ist

- ▶ entweder die leere Liste () oder
- ▶ die Konstruktion (**cons** x xs) aus dem Kopf x und einer Liste xs, dem Körper.

Die rekursive Struktur der Liste spiegelt sich häufig in rekursiven Listen-Funktionen wider.



Rekursive Konstruktion und Selektion





Einen Film der Schachtelpuppen finden Sie hier:

http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/SE3-Movies/Pages/Schachtelpuppe_1.html

http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/SE3-Movies/Pages/Schachtelpuppe_2.html

Zur Konstruktion von Listen

list: Die list -Funktion nimmt beliebig viele Argumente beliebigen Typs und verknüpft sie in der angegebenen Reihenfolge zu einer Liste:

```
> (define Ada '(Ada Countess of Lovelace))  
> (list 1 'Ada Ada)  
    →(1 ADA (ADA COUNTESS OF LOVELACE))
```

cons: Die cons-Funktion konstruiert eine neue Liste aus einem Element und einer Liste, wobei das Element zum Kopf der neuen Liste wird und die angegebene Liste zum Körper der neuen Liste. Und genau wie in Prolog können wir so rekursiv beliebig lange Listen konstruieren.

Eine andere Sicht auf `cons` ist, daß zwei Elemente durch `cons` zu einem *Paar* verbunden werden. Wir konstruieren Paare aus Kopf und Schwanz. Mit dieser Sichtweise ist es auch sinnvoll, als zweites Argument keine Liste sondern Elemente beliebigen Typs zu übergeben. Das Resultat ist dann ein Wertpaar, in Racket und Lips wegen der externen Repräsentation mit Klammern und Punkt *dotted pair* genannt. Beispiel: (`(cons 1 'Birne)`) evaluiert zu: `(1 . BIRNE)`. Listen von Wertpaaren werden gerne dazu verwendet, Relationen zu repräsentieren.

`append`: Diese Funktion verbindet zwei Listen zur Zusammensetzung der beiden Listen.

Auf den folgenden Folien wird als ein Beispiel zur Verwendung von Listen und Symbolen eine kleine Datenbasis mit Namen berühmter Informatiker modelliert. Dieses Beispiel ist direkt von [Norvig, 1992](#) übernommen.

9.3 Listen mit Symbolen



Listen mit Symbolen

```
Welcome to DrRacket, version 4.2.3 [3m].
Language: Module; memory limit: 256 megabytes.
> (define Ada
      '(Ada Countess of Lovelace)) → ⊥
> (cons 'Lady Ada)
      → (Lady Ada Countess of Lovelace)
> (cons 'Lady 'Ada) → (Lady . Ada)
      ; ein dotted pair
> (append '(Lady Ada)
      '(Countess of Lovelace)) →
(Lady Ada Countess of Lovelace)
> (list 'Lady Ada) →
(Lady (Ada Countess of Lovelace))
```

9.3.1 Zugriff auf Listenelemente

Zugriff auf Listenelemente

```
Welcome to DrRacket, version 4.2.3 [3m].  
Language: Module; memory limit: 256 megabytes.  
> (define Ada '(Ada Countess of Lovelace))  
      → ⊥  
> (length Ada) → 4  
> (car Ada) → Ada  
> (cdr Ada) → (Countess of Lovelace)  
> (cadr Ada) → Countess  
> (cdddr Ada) → (Lovelace)  
> (cadddr Ada) → Lovelace
```

Definition von Funktionen auf Listen

Beispiel: 28 (Parsing von Namen.)

```
> (define (last-name name)  
    ;;; Select the last name of a name  
    ;;; represented as a list."  
    (car (reverse name)))  
  
> (last-name '(Charles Babbage)) → Babbage  
> (define Ada '(Ada Countess of Lovelace))  
> (last-name Ada) → Lovelace  
> (last-name  
    '(Rear Admiral Grace Murray Hopper))  
      → Hopper  
> (last-name '(Blaise Pascal)) → Pascal  
> (last-name '(Eratosthenes)) → Eratosthenes
```

Anmerkung: Die Funktion **reverse** dreht eine Liste um.

```
> (define (first-name name)  
    ;;; select the first name from a name  
    ;;; represented as a list.  
    (car name))  
  
> (first-name '(Charles Babbage)) → charles  
> (first-name '(alonzo church)) → alonzo
```

- *first-name* macht nichts anderes, als die Funktion `car` anzuwenden.
- Es ist dennoch sinnvoll, eine eigene Funktion für die Datenstruktur „Name“ einzuführen — Datenabstraktion!

Zur Datenabstraktion Wenn wir eigene Datenstrukturen einführen, dann ist es zwar sinnvoll, die Grundoperationen der Sprache für die Programmierung der Akzessorfunktionen zu verwenden, aber wir sollten diese nie direkt verwenden, sondern immer eigene Zugriffsfunktionen einführen. Das erleichtert es uns, die Repräsentation notfalls zu ändern. Das Programm ist unabhängig von der speziellen Repräsentation der Daten, in unserem Beispiel also unabhängig von der Repräsentation der Namen als Listen '(Lady Ada Countess of Lovelace). Wenn wir die Repräsentation in Zukunft durch Zeichenketten ersetzen wollen, dann brauchen wir nur die beiden Funktionen `first-name` und `last-name` zu ändern, aber *nicht das Anwendungsprogramm!* Wenn wir konsequent auf unsere Datenstrukturen nur über spezielle Akzessorfunktionen und Konstruktorfunktionen zugreifen, dann definieren wir über diese Funktionen einen neuen *abstrakten Datentyp*. Zusätzlich sollten wir noch ein Typprädiat `is-name?` und einen Konstruktor `make-name` vorsehen.

9.3.2 Assoziationslisten



Eine „Datenbasis“ von Namen berühmter Informatikerinnen und Informatiker: Eine Liste von Listen

```
> (define *computer-scientists*
'( (Charles Babbage)
  (Alonzo Church)
  (Freiherr Gottfried Wilhelm von Leibniz)
  (Rear Admiral Grace Murray Hopper)
  (Lady Ada Countess of Lovelace)
  (John McCarthy)
  (John von Neumann)
  (Alan Turing)
  (Blaise Pascal))) → ⊥
```

Suche in der Datenbasis:

- Mithilfe der Standardfunktion **assoc** lassen sich Assoziationslisten (Listen von Paaren) durchsuchen:

(**assoc** <key> <list-of-lists>)

- **assoc** sucht das erste Paar, dessen erstes Element gleich dem Schlüssel ist.

```
> (assoc 'Alan *computer-scientists*)
      → (Alan Turing)
> (last-name
  (assoc 'Alan *computer-scientists__))
      → Turing
```

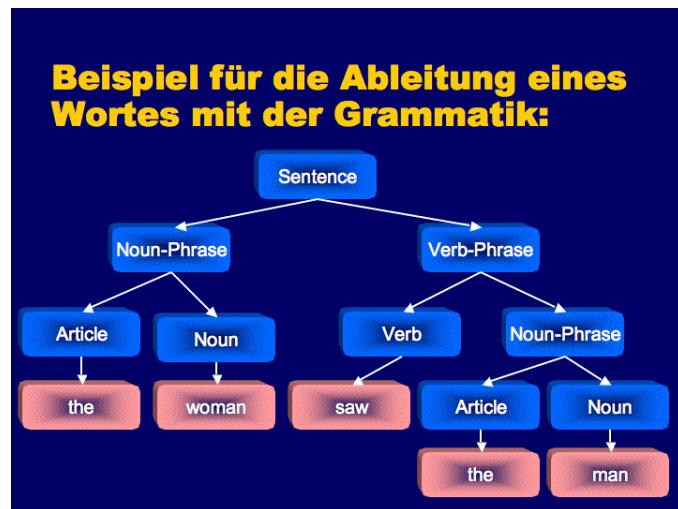
9.3.3 Rekursiver Abstieg zur Sprachgenerierung

Sprachgenerierung

Beispiel: 29 (Eine simple generative Syntax für englische Sätze:)

```
<Sentence>      ::= <Noun-Phrase> <Verb-Phrase>
<Noun-Phrase>   ::= <Article> <Noun>
<Verb-Phrase>   ::= <Verb> <Noun-Phrase>
<Article>       ::= the | a | ...
<Noun>          ::= man | ball | woman | table ...
<Verb>           ::= hit | took | saw | liked ...
```

Der Ableitungsbaum



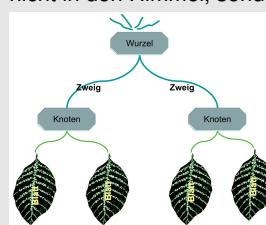
Definition: 30 (Ableitungsbaum)

Rekursive Definition: Ein **Ableitungsbaum** ist entweder

- ein einzelner Knoten, der mit einem *Terminalzeichen* markiert ist (ein Blatt des Baumes)
- oder ein Knoten, markiert mit einer *metalinguistischen Variablen* und gefolgt von und verbunden mit den Elementen einer endlichen, geordneten Menge von (Teil-)Bäumen.
- Ein Knoten heißt *Wurzel*, wenn er keine übergeordneten Knoten hat.

Ableitungsbäume

Für Informatikerinnen und Informatiker wachsen die Bäume nicht in den Himmel, sondern stehen Kopf:



- ▶ Die Wurzel ist oben.
- ▶ Die Blätter sind unten.

Drei Welten, M.C. Escher 1955



Aufgabe:

Gesucht: Ein Programm, das zufällige Sätze der englischen Sprache erzeugt.

Direkter Lösungsansatz: Jede Regel wird durch eine Funktion repräsentiert.

Rekursiver Abstieg:

Dieses Verfahren wird das Verfahren des *rekursiven Abstiegs* genannt, da rekursive Regeln zu rekursiven Funktionsaufrufen führen.

Funktionale Repräsentation der Grammatik in Racket

Beispiel: 31 (Rekursiver Abstieg)

Die Terminalsymbole werden zu Racket-Symbolen.

Die Sätze der Sprache werden zu Listen.

Die Regeln werden zu Funktionen.

Die Konkatenation der Phrasen erfolgt durch die **append**-Funktion.

Repräsentation der Regeln als Funktionen

```
(define (sentence)
        (append (noun-phrase) (verb-phrase)))
(define (noun-phrase)
        (append (Article) (Noun)))
(define (verb-phrase)
        (append (Verb) (noun-phrase)))
(define (Article)
        (one-of '(the a)))
(define (Noun)
        (one-of '(man ball woman table)))
(define (Verb)
        (one-of '(hit took saw liked)))
```

- Die von der Grammatik erzeugte Sprache ist *endlich*, da keine rekursiven Regeln vorkommen.

Zufälligen Auswahl von Wörtern

```
(define (one-of set)
        ;;; Pick one element of set,
        ;;; and make a list of it.
        (list (random-elt set)))
(define (random-elt choices)
        ;;; Choose an element from a list
        ;;; at random.
        (list-ref choices
                    (random (length choices))))
```

- $(\text{random } n)$ liefert eine zufällige Zahl x , mit $0 \leq x < n$
- $(\text{list-ref } liste \ n)$ liefert das n -te Element von $liste$, wobei die Zählung bei 0 beginnt.

Das Modul „tools-module.rkt“

Selbstdefinierte Funktionen, auf die wir häufig zurückgreifen werden, haben wir für Sie in einer Bibliothek se3–bib zusammengestellt, siehe:

- http://kogs-www.informatik.uni-hamburg.de/~dreschle/teaching/Uebungen_Se_III/Uebungen_Se_III.html
- Im file „se3–bib/tools–module.rkt“ finden Sie die am häufigsten benutzten Definitionen,
 - die Sie in Ihre Programme kopieren können
 - oder als Modul zu Ihren Definitionen dazu laden können:
`(require se3–bib/tools–module)`
- Die Beispielprogramme finden Sie in der se3–bib im Verzeichnis
... /se3–bib/Integrationstest.

Language: **Module**; *memory limit: 256 megabytes*.

```

> (sentence)      →(the woman hit a ball)
> (sentence)      →(the ball liked the woman)
> (noun–phrase)   →(a woman)
> (verb–phrase)   →(saw a woman)
> (Noun)          →(ball)
> (sentence)      →(the woman saw the ball)
> (sentence)      →(the ball hit a table)
> (sentence)      →(the man liked the ball)
> (sentence)      →(a table took a man)

(define writeln
  (lambda args
    (for-each display args)
    (newline)))

(define (random–elt choices)
  ;;; Choose an element from a list at random
  (list-ref choices (random (length choices))))
```

Aufruf-Trace

Willkommen bei DrRacket, Version 5.0.1 [3m].
 Sprache: racket; *memory limit: 256 MB*.

```

> (require racket/trace)
> (trace sentence noun–phrase verb–phrase
           Article Noun one–of random–elt)
   →(sentence noun–phrase verb–phrase
              Article Noun one–of random–elt)
> (noun–phrase)
| (noun–phrase)
| (Article)
| (a)
| (Noun)
| (ball)
|(a ball)  →(a ball)
> (untrace sentence noun–phrase)

> (sentence)
|(sentence)
| (noun–phrase)
```

```

| (( Article )
| |( the )
| |( Noun )
| |( table )
| (the table )
| (verb-phrase)
| |( noun-phrase )
| | ( Article )
| | |( a )
| | |( Noun )
| | |( woman )
| |( a woman )
| ( liked a woman )
|( the table liked a woman )
    →(the table liked a woman)

```

Die Funktion `trace` ist eine nützliche Hilfe bei der Fehlersuche: Sie protokolliert alle Aufrufe der überwachten Funktionen und die Argumente, die übergeben werden.

Wenn wir das Protokoll wieder abstellen wollen, dann rufen wir `untrace` auf, mit den jeweiligen Funktionen als Argument, deren Aufrufe nicht mehr protokolliert werden sollen.

`trace` ist ein weiteres gutes Beispiel dafür, wie nützlich es ist, wenn Funktionen eine variable Zahl von Argumenten haben können. Wir können so mehrere Funktionen gleichzeitig verfolgen lassen.

Wir sollten generell nur Funktionen „tracen“ lassen, die wir selbst programmiert haben. Wenn wir einen `trace` für eine Standardfunktion bestellen, kann es sehr gut eine Funktion sein, die von der `trace`-Funktion selbst auch aufgerufen wird, so daß wir unbeabsichtigt eine Rekursion auslösen.

eval und **trace** sind nicht in allen Scheme-Implementationen als Standardfunktionen implementiert. In scm-Scheme und DrRacket läßt sich **trace** nicht auf **eval** anwenden, in Common Lisp dagegen schon.

Erweiterung: Adjektive und Präpositionen

Beispiel: 32 (Zusätzliche Regeln: Adjektive und Präpositionen)

```

<Sentence>      ::= <Noun-Phrase> <Verb-Phrase>
<Noun-Phrase>   ::= <Article> <Adj*> <Noun> <PP*>
<Adj*>          ::= ∅ | <Adj> <Adj*>

```

```

<PP*>      ::= ∅ | <PP> <PP*>
<PP>        ::= <Prep> <Noun-Phrase>
<Adj>        ::= big | little | blue | green | ...
<Prep>       ::= to | in | by | with | ...
<Verb-Phrase> ::= <Verb> <Noun-Phrase>
<Article>    ::= the | a | ...
<Noun>       ::= man | ball | woman | table ...
<Verb>        ::= hit | took | saw | liked ...

```



Zur Notation

- Neu hinzugekommen, bzw. geändert wurden die Regeln 2–7.
- In dieser Notation stellt \emptyset das leere Wort dar; ein Komma trennt mehrere Alternativen, und der Stern (*) bedeutet nichts Besonderes.
- Es besteht aber die Konvention, daß *metalinguistische Variable*, deren Namen mit einem * enden, für gar kein Zeichen oder beliebig viele Wiederholungen stehen. Diese Notation heißt *Kleene-Stern Notation*, nach dem Mathematiker Stephen Cole Kleene.
- Die neu definierte Sprache hat unendlich viele Worte, denn die rekursiven Regeln können unendlich viele und beliebig lange verschiedene Teilworte erzeugen.

```

(define (Adj*)
  (if (= (random 2) 0)
      '()
      (append (Adj) (Adj*))))
(define (PP*)
  (if (random-elt '#t #f)
      (append (PP) (PP*)) '()))
(define (noun-phrase)
  (append (Article) (Adj*) (Noun) (PP*)))
(define (PP) (append (Prep) (noun-phrase)))
(define (Adj)
  (one-of
   '(big little blue green adiabatic)))
(define (Prep) (one-of '(to in by with on)))
>(sentence) →(a woman by a man hit a table)

```

- > (**sentence**) →(a woman in the woman
by the man with the man liked the
woman in a adiabatic man in a man
in a woman with a table on the ball
on a little woman to a big blue
green ball)

- > (**sentence**) →(a man took the big man to the
big big woman with a big green man on the blue
little green table to a ball to the man on A
blue man on the ball on the man by a woman
with the green man with the green big little
little little adiabatic adiabatic table to a
big table in the ball with a green man by the
woman in the green man with the table with a
adiabatic ball in a little little
green man on a woman)

Anmerkungen: Diese Beispiele zeigen, daß mit Phrasenstrukturgrammatiken zwar gut die Syntax einer Sprache, ihre äußere Form, beschrieben werden kann, aber nur wenig von der Semantik, der Bedeutung, der Sprache erfaßt wird. Man kann in wohlgeformten Sätzen den größten Unsinn produzieren und bleibt doch syntaktisch korrekt.

Teilweise ließe sich die Semantik besser berücksichtigen, wenn die Regeln kontextabhängig formuliert würden, so daß beispielsweise das Prädikat "hit" nicht mit dem Subjekt "the table" kombiniert werden könnte. Kontextsensitive Grammatiken, die das ausdrücken können, werden Sie in den Modulen der formalen Grundlagen der Informatik kennenlernen.

Teil IV

Repräsentation der Datenstrukturen

10 Funktionale Datenabstraktion

Funktionale Datenabstraktion

- Datenobjekte sind in einer funktionalen Programmiersprache nicht unbedingt erforderlich.
- Zahlen und Listen lassen sich durch *Closures* repräsentieren.

Beispiel: 33 (Closure als Liste)

```
> (define (my-cons head tail)
  (lambda (message)
    (case message
      [(h) head]
      [(t) tail])))
> (define xs (my-cons 1 2)) → xs
> (xs 'h) → 1
> (xs 't) → 2
> (define (my-car xs) (xs 'h))
> (my-car xs) → 1
```

☞ Beachte: Funktionen höherer Ordnung: my-cons, my-car und my-cdr sind Funktionen höherer Ordnung. my-cons erzeugt als Resultat einen funktionalen Abschluß, der in seiner Umgebung die Werte von head und tail speichert. Die Funktion gibt auf Anfrage entweder den head oder den tail als Wert der Funktion zurück, je nach dem, welchen Wert der Parameter message hat. Die Akzessor-Funktionen my-car, my-cdr rufen die als Argument übergebene Funktion jeweils mit der passenden Nachricht auf. Bei dieser Repräsentation haben sich unsere Daten in aktive Datenobjekte verwandelt, die Nachrichten verstehen und die gewünschte Verarbeitung selbst durchführen. Wir wenden keine Funktionen auf die Daten an, sondern senden ihnen Daten, nämlich Nachrichten.

Sie sehen aber auch, daß beide Sichtweisen durch zwei einzeilige Funktionen ganz leicht ineinander überführt werden können und das message-passing für uns verborgen werden kann. Viele sehen daher das für die objektorientierte Programmierung typische „message“ passing eher als syntaktischen Zucker denn als eigenen Programmierstil an.

Im folgenden Kapitel werden wir sehen, was mit Funktionen höherer Ordnung noch so alles möglich ist und sie als wirklich mächtiges Abstraktionsmittel kennenlernen, mit dem wir wohlstrukturierte und flexible Programme entwerfen können.

11 Repräsentation und Gleichheit

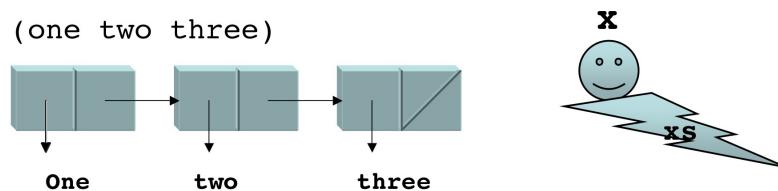
11.1 Cons-Zellen

Repräsentation von Listen

- Listen sind in Racket *dynamische* Datenstrukturen, d.h. sie können zur Laufzeit ihre Struktur verändern.
- Daher sind sie so repräsentiert, daß es leicht ist, Teillisten zu entfernen oder einzufügen.

Jede Liste wird durch zwei *Referenzen* repräsentiert:

- Eine Referenz, die das erste Element, den Kopf, bezeichnet,
- und eine Referenz, die den Schwanz (oder den Rest) der Liste bezeichnet.



Anmerkungen zur Repräsentation: Da wegen der rekursiven Definition einer Liste der Schwanz wiederum eine Liste ist, finden wir hier auch wieder zwei Referenzen, die Referenz auf das zweite Element der Liste und den Schwanz des Schwanzes, usw. Wer schon Pascal oder C kennt, kann sich diese Referenzen als Pointer vorstellen, in Java entsprechend als Referenzen auf die beiden Objekte Kopf und Rest.

Die beiden Referenzen auf Kopf und Schwanz werden in den schon erwähnten cons-Zellen gespeichert, die meist in Form von Kästchen dargestellt werden, wobei die Referenzen als Pfeile auf die referenzierten Strukturen gezeichnet werden.

Eine Liste referenziert ihre erste **cons**-Zelle, d.h. der Wert einer Liste ist die cons-Zelle. Beide Referenzen in der Zelle können mit **set!** oder anderen Modifikatoren geändert werden, wenn man imperativ programmiert.

Repräsentation und Gleichheit

Wann sind zwei Listen gleich?

- Wenn sie *dasselbe Objekt* im Speicher sind?
- Wenn sie *gleich aufgebaut* sind (mit denselben Elementen)?
- Wenn sie *gleich gedruckt* werden?

Beispiel:

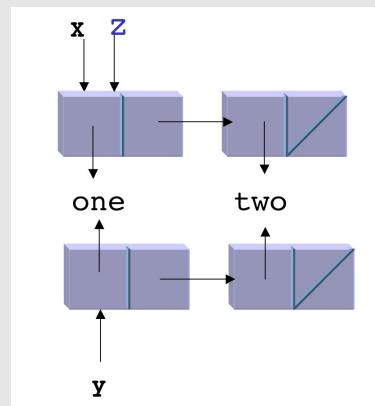
Listen mit gemeinsamen Elementen

```
(define x '(one two)) ; x → (one two)
(define y '(one two)) ; y → (one two)
(define z x)           ; z → (one two)
```

Problem: 34 (Frage:)

- Sind x und y identisch ($\text{eq? } x \ y$)?
- Sind x und z identisch ($\text{eq? } x \ z$)?
- Sind x und y gleich ($\text{equal? } x \ y$)?

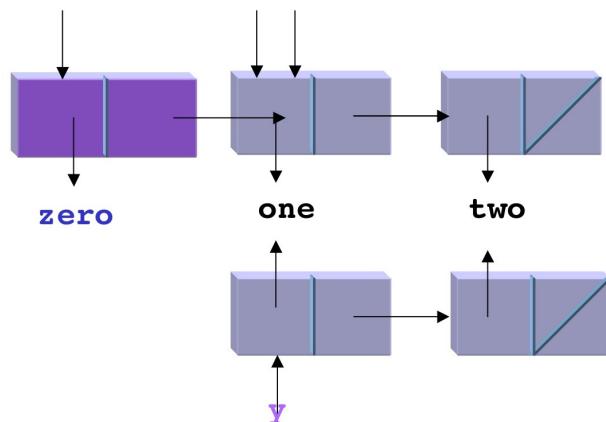
Zur Gleichheit und Identität



- ▶ x und y sind **gleich**, aber nicht **identisch**.
- ▶ Sie enthalten dieselben Elemente, verwenden aber unterschiedliche cons-Zellen.
- ▶ x und z dagegen sind **identisch**.

Die cons-Operation

(cons 'zero x) x z



Was bewirkt ein Aufruf von cons?

- Ein **cons**-Aufruf erzeugt genau eine cons-Zelle.
- Diese Zelle enthält die Referenzen auf Kopf und Körper der neuen Liste.
- Der Körper dieser neuen Liste ist *identisch* mit der Liste, die als zweites Element an **cons** übergeben wurde.
- Wenn wir mit Modifikatoren (z.B. **set!**) Teile von Strukturen ändern wollen, dann brauchen wir eine Möglichkeit, um zu testen, ob wir ein Original oder eine Kopie haben.
- Racket kennt daher verschiedene Funktionen, um die Gleichheit oder Identität zu testen.

11.2 Gleichheitsprädikate für Listen

Grade von Gleichheit

All animals are equal, but some animals are more equal than others.

George Orwell, Animal Farm, 1945

Die wichtigsten Gleichheitsprädikate:

`eq?`: Wahr für identische Objekte.

`eqv?`: Wahr für identische Objekte oder gleiche Zahlen und Wahrheitsweite.

`equal?`: Wahr für Objekte, die `eqv?` sind oder Listen, Mengen und Strings, mit Elementen, die `eqv?` sind.

`=`: Wahr für gleiche **numerische** Werte.

`string=?`: Wahr für gleiche **Zeichenketten**.



Auswertung von Gleichheitsprädikaten

x	y	eq?	eqv?	equal?
'x	'x	#t	#t	#t
1	1	⊥	#t	#t
'(x)	'(x)	#f	#f	#t
"xy"	"xy"	⊥	⊥	#t
"Xy"	"xY"	⊥	⊥	#f
1	#e1.0	⊥	#t	#t
1	2	⊥	#f	#f

- Daneben gibt es typspezifische Gleichheitsprädikate.

☞ **Anmerkung:** Nur Zahlen (`number?`) und Zeichen (`char?`) unterscheiden sich `eq?` und `eqv?`. Das liegt daran, daß identische Zahlen und Zeichen implementationabhängig mehrfach im Speicher repräsentiert sein können. Für Zahlen und Zeichen sollte zum Test auf Identität unbedingt `eqv?` verwendet werden.

☞ Anmerkung: Funktionen (procedures) dürfen in manchen Sprachen, z.B. Common Lisp nicht mit equal? verglichen werden. Warum wohl nicht?



Familien von Suchfunktionen

- Funktionen, die Datenstrukturen nach bestimmten Kriterien durchsuchen, gibt es oft in mehreren Ausprägungen, je nach dem, welches Gleichheitsprädikat sie verwenden.
- Die Endung des Funktionsnamens weist auf das verwendete Gleichheitsprädikat hin.
- Beispiele:

(member x xs): ist x Element der Liste xs?

(assoc x xss): Suche die Unterliste von xss, die mit x beginnt.

member

- Die Funktionen der *member*-Familie vergleichen elementweise die Elemente der Liste mit dem gesuchten Element
- und geben den ersten Listenrumpf zurück, der mit dem Element beginnt,
- #f, falls das Element nicht gefunden wird.

member vergleicht mittel equal?.

memv vergleicht mittel eqv?.

memq vergleicht mittel eq?.

Beispiel: member

Language: racket ;

```
> (member '(Alan Turing) *computer-scientists*)
  → ((Alan Turing) (Blaise Pascal))
> (member '(Madonna) *computer-scientists*)
  → #f
> (memv '(Alan Turing) *computer-scientists*)
  → #f
> (memq '(Alan Turing) *computer-scientists*)
  → #f
> (memv 'Turing '(Alan Turing) )
  → (Turing)
```

assoc

- *assoc* erwartet als Argumente einen Suchschlüssel und eine Liste von Paaren oder Listen.
- Die Funktionen der *assoc*-Familie vergleichen elementweise die Köpfe der Unterlisten (Paare) mit dem Suchschlüssel
- und geben die erste Unterliste zurück, die mit dem Schlüssel beginnt.
- #f, falls das Element nicht gefunden wird.

assoc vergleicht mittel equal?.

assv vergleicht mittel eqv?.

assq vergleicht mittel eq?.

Beispiele für assoc

Language: racket .

```
> (assoc 'Alan *computer-scientists*) → (Alan Turing)
> (assq 'Alan *computer-scientists*) → (Alan Turing)
> (assv 'Alan *computer-scientists*) → (Alan Turing)
> (define *computerStrings*
  '( ("Charles" "Babbage")
    ("Alan" "Turing")
    ("Blaise" "Pascal")))
> (assoc "Alan" *computerStrings*)
("Alan" "Turing")
> (assv "Alan" *computerStrings*) → #f
> (assq "Alan" *computerStrings*) → #f
```

11.3 Rekursion über Listen

Rekursion über Listen: Länge einer Liste

- Die *leere Liste* hat die Länge 0.
- Die Länge einer *nicht-leeren Liste* xs ist: $1 + \text{Länge des Schwanzes}$, also $1 + (\text{my-length}(\text{cdr } \text{xs}))$

```
> (define (my-length xs)
  (if (null? xs) 0
      (+ 1 (my-length (cdr xs)))))  
> (my-length '(1 2)) → 2
```

member?: _____

Ist item ein Element der Liste xs?

- item ist nicht Element der *leeren Liste*.
- In einer *nicht-leeren Liste* ist item enthalten, wenn
 - (car xs) = item
 - oder (member? item (cdr xs)) gilt,
- also entweder der *Kopf* gleich dem gesuchten Element ist
- oder das gesuchte Element im *Schwanz* vorkommt.

```
> (define (member? item xs)
  (if (null? xs) #f
      (or (equal? item (car xs))
          (member? item (cdr xs)))))  
> (member? 2 '(5 4 7 2 4)) → #t  
> (member? 2 '(5 4)) → #f
```

Anmerkungen: Das hier definierte Praedikat member? ist ein echtes **Prädikat**, in dem Sinne, daß es die Wahrheitswerte #t, f als Resultat zurückgibt. Die Standardfunktion **member** dagegen ist in Racket wie in allen anderen Lisp-Dialektlen ein **Semipräädikat**. Ist das erste Argument nicht in der Liste enthalten, dann ist das Resultat #f, aber wenn das Element in der Liste gefunden wird, ist das Resultat die erste Teilliste, die mit dem gesuchten Element beginnt, also:

```
(member 3 '(1 2 3 4 5 6)) → (3 4 5 6)
```

Namensanalyse: Variante 2



```
> (define (first-name name)
  (car name))
> (first-name
  '(Freiherr Gottfried Wilhelm von Leibniz))
  → Freiherr
> (first-name
  '(Rear Admiral Grace Murray Hopper))
  → Rear
```

first-name ist noch nicht sehr treffend formuliert: Titel und Vornamen werden nicht unterschieden.

Beispiel: 35 (Var. 2: Rekursion über den Namen)

Language: racket .

```
> (define *titles*
  '( Freiherr Rear Admiral Lady von Herr Frau
    Mr. Mrs. Miss Sir Madam Dr. Prof. ))
> (define (first-name2 name)
  ; ; select the first name from a name
  (if (member (car name) *titles*)
    (first-name2 (cdr name))
    (car name)))
> (first-name2 '(Lady Ada Lovelace))
  → Ada
> (first-name2
  '( Freiherr Gottfried Wilhelm von Leibniz))
  → Gottfried
```

Gesucht: Das n-te Element von xs.

Beispiel: 36 (Rekursion: Indizierung einer Liste)

- Das nullte Element ist der Kopf der Liste.
- Das n -te Element von `xs` ist das $(n-1)$ -te Element von `(cdr xs)`.
- Fehler: wenn $n > \text{Länge der Liste}$.

```
(define (list-ref xs n)
  ;;; element n of list xs, zero indexed
(cond
  [(<= (length xs) n)
   (error "list-ref:_Index" n
          "out_of_range:" xs)]
  [(zero? n) (car xs)]
  [else (list-ref (cdr xs) (- n 1))]))
```

Beachte: Der Zugriff dauert umso länger, je länger die Liste ist.
`list-ref` ist eine Standardfunktion von Racket.

Zur Stilistik

- Es ist hilfreich, wenn die Namen von Variablen auf deren Typ schließen lassen.
- Wir bezeichnen daher
 - Listenelemente mit `x`, `y`, `z`.
 - Listen mit `xs`, `ys`, `zs`.
 - Listen von Listen mit `xss`, `yss`, `zss`.

Codeerzeugung zur Laufzeit

Mit Listen können wir *dynamisch* zur Laufzeit neuen Code erzeugen und auch ausführen:

- Ein Racket-Ausdruck (s-expression) ist syntaktisch eine Liste,
- lässt sich wie eine Liste zur Laufzeit konstruieren
- und mit **eval** auswerten.

```
> (list 1 2 3) → (1 2 3) ; eine Liste
> (define a (list * 3 3))
> a → '#<procedure:> 3 3)
> (eval a) → 9
```

12 Weitere strukturierte Datentypen

12.1 Zeichenketten

Zeichenketten (strings)

- Eine Folge von Zeichen heißt *Zeichenkette (string)*.
- Strings lassen sich auf ihre lexikographische Ordnung hin vergleichen, lassen sich drucken, lassen sich verknüpfen.
- Strings werden mit Gänsefußchen markiert:
- Achtung! Ein einzelnes Zeichen hat einen anderen Typ als ein String der Länge 1.

```
> (string<? "Racket" "Common_Lisp")    → #f
> (string<? "Jo" "Johannah")           → #t
> (string? #\A)                      → #f
> (string? "a")                      → #t
```

Konkatenation von Zeichenketten

Die Funktion `string-append` fügt zwei Zeichenketten zu einer zusammen, indem sie die Elemente der zweiten Liste hinter die erste hängt: mathematisch entspricht das der Zusammensetzung zweier Worte einer formalen Sprache.

```
> (string-append "Auto" "bus") → "Autobus"
> (string-length "a")          → 1
> (list->string '(#\a))      → "a"
> (string->list "hallo")
  → (#\h #\a #\l #\l #\o )
> (substring "123456" 2 4)   → "34"
```

Stringkonstanten und Stringvariable

Zeichenketten (strings) sind in Racket entweder

- *konstant* (immutable), wenn sie direkt als Konstante notiert werden,
- oder *modifizierbar* (mutable), wenn sie mit `make-string` erzeugt werden.

```

> (immutable? "123")
#t
(define s (make-string 5 #\.))
> s → "....."
> (immutable? s)
#f
> (string-set! s 2 #\λ)
> s → "...λ..."

```

Beispiel: Bei der Ausgabe von Tabellen oder anderen Textstücken möchte man oft gerne steuern können, wo genau die Werte gedruckt werden. Meist soll ein Wert entweder

linksbündig *zentriert oder* *rechtsbündig*
gedruckt werden.

Wir werden drei Funktionen `ljustify`, `rjustify`, `cjustify` definieren, die einen String als Argument erhalten und links und/oder rechts so mit Leerzeichen auffüllen, daß er passend gedruckt wird.

Ein weiteres Argument wird die Zahl der Druckstellen in der Spalte sein. Wir nehmen zur Vereinfachung an, daß alle Zeichen dieselbe Breite haben (Proportionalschrift verboten!).

Alle drei Funktionen sind als partielle Funktionen definiert und geben \perp als Wert zurück, wenn der String länger ist als der zur Verfügung stehende Platz.

Wir benötigen noch eine Funktion (`space n`), die uns eine Zeichenkette von n Leerzeichen erzeugt. Zur Übung empfohlen!

Die Funktion (`string-length x`) gibt die Länge einer Zeichenkette zurück. Sie berechnet für die Menge der Worte w über dem Alphabet der ASCII-Zeichen die die Länge des Wortes w , also $|w|$. (*substring string von bis*) berechnet das Teilwort des strings, von Position *von* bis *bis*.

Textformatierung

Beispiel: 37 (Spaltengenaues Einrücken, Druckbreite n:)

```

(define (ljustify n s)
  (let ([len (string-length s)])
    (if (>= n len)
        ; string to short, pad with blanks
        (string-append
         s
         (make-string (- n len) #\space)))
        s))

```

```

    (space (- n len)))
; string to long, trim at the end
(substring s 0 n)
)))
> (ljustify 20 "Halli_Hallo!")
→ "Halli_Hallo!_____"
(define (rjustify n s)
  (let ([len (string-length s)])
    (if (>= n len)
        ; string to short, pad with blanks
        (string-append
          (space (- n len))
          s)
        ; string to long, trim at the start
        (substring s (- len n) len)
      )))
> (rjustify 20 "Halli_Hallo!")
→ "_____Halli_Hallo!"
(define (cjustify n s)
  (let* ([len (string-length s)]
         [left (quotient (- n len) 2)]
         [right (- n left len)])
    (if (>= n len)
        (string-append
          (space left) s (space right))
        (substring s 0 n))))
> (cjustify 20 "hallihallo!")
→ "____hallihallo!_____"

```

Beispiel: Konversion Zahl — Text

Beispiel: 38 (number- cleartext)

Gelegentlich müssen Zahlen auch mal in Textform ausgegeben werden, beispielsweise auf Schecks oder Überweisungsformularen.

Wir wollen eine Funktion *number->cleartext* entwerfen, die uns natürliche Zahlen in strings wandelt:

```

> (number->cleartext 555050)
→ five hundred and fifty-five thousand
and fifty

```

```
> (number->cleartext 1)  
→ one
```

Beobachtung:

- Die Zahlen 1 … 19 werden unsystematisch gebildet.
- Die Zahlen 20 … 99 werden systematisch
 - aus Einerstelle
 - und Zehnerstelle kombiniert.
- Die Zahlen 100 bis 999 werden
 - aus einer Hunderterstelle
 - und einer zweistelligen Zahl konstruiert.
- Die Zahlen 1000 bis 999999 werden
 - aus einer Hunderterangabe,
 - der Kennung „thousand“
 - und einer weiteren Hunderterangabe gebildet.

Lösungsansatz:

Drei Gruppen von Stützfunktionen:

Konvertierung: Zerlegung in Ziffern, Konversion zweistelliger, dreistelliger und sechsstelliger Zahlen:

digit , convert2 , convert3

Kombinatoren: Zusammenfassen von Teilgruppen zu einem String:

combine2 , combine3 , combine6 , link

Füllworte: and

Wir werden das Problem modular lösen und zunächst mit den Zahlen von 1 bis 99 beginnen. Die zweistellige Zahl wird von der Funktion digit in ein Paar von Ziffern konvertiert, die dann von convert2 gewandelt werden.

```

;;; Wandlung von Zahlen < 100
(define units '("one" "two" "three" "four"
                "five" "six" "seven"
                "eight" "nine"))
(define teens  '("ten" "eleven" "twelve"
                  "thirteen" "fourteen"
                  "fifteen" "sixteen"
                  "seventeen" "eighteen"
                  "nineteen"))
(define tens   '("twenty" "thirty" "forty"
                  "fifty" "sixty" "seventy"
                  "eighty" "ninety"))

```

Zweistellige Zahlen

```

(define (digit n pos)
  ; digit: number number → number
  ; Ziffer von n an Pos. pos
  (let ([n-ziffern
        (remainder n (expt 10 pos))])
    (quotient n-ziffern (expt 10 (- pos 1)))))
> (digit 123 1)      → 3
> (digit 123 3)      → 1

(define (convert2 n)
  ; convert2: number → string
  (combine2 (digit n 2)
            (digit n 1)))

(define (combine2 d-tens d-units)
  ; natural → string
  (cond [(= 0 d-tens)
          (list-ref units (- d-units 1))]
         [ (= 1 d-tens)
          (list-ref teens d-units)]
         [ (= 0 d-units)
          (list-ref tens (- d-tens 2))]
         [else
          (string-append
            (list-ref tens (- d-tens 2))
            "_"
            (list-ref units (- d-units 1))))]))

```

```

> (convert2 1)      → "one"
> (convert2 10)     → "ten"
> (convert2 39)     → "thirty-nine"
> (convert2 42)     → "forty-two"

```

Dreistellige Zahlen

Wandlung der Zahlen von 1 bis 999: Zerlegung in die führende Ziffer und die beiden letzten Stellen.

```

(define (convert3 n)
  ; convert3: number → string
  (combine3 (digit n 3)
            (remainder n 100)))
> (convert3 999)
  → "nine_hundred_and_ninety-nine"
> (convert3 111)
  → "one_hundred_and_eleven"
> (convert3 42)
  → "forty-two"

(define (combine3 d-hundreds d-belowhundred)
  ; combine3: number number → string
  (cond [(= 0 d-hundreds)
          (convert2 d-belowhundred)]
        [ (= 0 d-belowhundred)
          (string-append
            (list-ref units (- d-hundreds 1))
            "_hundred")]
        [else (string-append
                  (list-ref units (- d-hundreds 1))
                  "_hundred_and_"
                  (convert2 d-belowhundred))]))
> (convert3 999)
  → "nine_hundred_and_ninety-nine"

```

Wandlung von 6-stelligen Zahlen:

Zerlegung in Tausender-Stellen und Hunderter-Stellen.

```

(define (number->cleartext n)
  (if (< n 1000000)
    (combine6
      (quotient n 1000)
      (remainder n 1000)))

```

```

        " "))

(define (link h)
  (if (< h 100) "_and_" ""))
> (number->cleartext 5012)
  → "five_thousand_and_twelve"
> (number->cleartext 5112)
  → "five_thousand_one_hundred_and_twelve"

(define (combine6 thousands hundreds)
  (cond [(= 0 thousands)(convert3 hundreds)]
        [(= 0 hundreds)
         (string-append (convert3 thousands)
                       "_thousand")]
        [else
         (string-append (convert3 thousands)
                       "_thousand" (link hundreds)
                       (convert3 hundreds))]))
> (number->cleartext 1111)
  → "one_thousand_one_hundred_and_eleven"

> (number->cleartext 42042)
  → "forty-two_thousand_and_forty-two"

```

Sprachausgabe

- Unter Mac OS kann englischer Text mit dem Speech Synthesis Manager als gesprochene Sprache ausgegeben werden.
- Dazu muß das shell-Kommando „say“ aufgerufen werden.

say [-v<voice>] [-o out.aiff]
[-f<file> | <string>]

- <voice>: Die Sprechstimme: Victoria, Fred, Princess, Deranged usw.
- <file> oder <string> Der zu sprechende Text.

system.rkt

- Um in Racket shell-Kommandos ausführen zu können, benötigen Sie das Bibliotheksmodul system.rkt.

```
(require racket/system)
(define canSpeak?
  (equal? (system-type) 'macosx))

(define (say text-to-say voice)
  (let ([theCommand
        (string-append
         "say -v \" " voice " \" "
         text-to-say " \" ")])
    (if canSpeak?
        (system theCommand)
        (display text-to-say))))
```

Beispiel

- Im Modul se3-bib/macos-module.rkt sind die beiden Funktionen canSpeak? und say schon vordefiniert:

Sprache: **Module**; *memory limit: 128 megabytes*.

```
(require se3-bib/macos-module)
> (say "Hello_World!" "Victoria")
   → say -v "Victoria" "Hello_World!" #t
> (say "Hello_World!" "Princess")
   → say -v "Princess" "Hello_World!" #t
>
```

12.2 Vektoren

Auszug aus dem DrScheme-Helpdesk: ([Flatt, 2006](#)) Vectors are heterogeneous structures whose elements are indexed by integers. A vector typically occupies less space than a list of the same length, and the average time required to access a randomly chosen element is typically less for the vector than for the list.

The length of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The valid indexes of a vector are the exact non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Vectors are written using the notation #(obj ...). For example, a vector of length 3 containing the number zero in element 0, the list (2 2 2 2) in element 1, and the string "Anna" in element 2 can be written as following:

```
#(0 (2 2 2 2) "Anna")
```

Note that this is the external representation of a vector, not an expression evaluating to a vector. Like list constants, vector constants must be quoted:

```
'#(0 (2 2 2 2) "Anna")
      → #(0 (2 2 2 2) "Anna")
```

library procedure: (vector obj ...)

Returns a newly allocated vector whose elements contain the given arguments. Analogous to list.

```
(vector 'a 'b 'c) → #(a b c)
```

procedure: (make-vector k fill)

Returns a newly allocated vector of k elements. If a second argument is given, then each element is initialized to fill. Otherwise the initial contents of each element is unspecified.

procedure: (vector-length vector)

Returns the number of elements in vector as an exact integer.

procedure: (vector-ref vector k)

k must be a valid index of vector. Vector-ref returns the contents of element k of vector.

```
(vector-ref '#(1 1 2 3 5 8 13 21)
            5) → 8
(vector-ref '#(1 1 2 3 5 8 13 21)
            (let ((i (round (* 2 (acos -1)))))
              (if (inexact? i)
                  (inexact->exact i)
                  i)))
            → 13
```

procedure: (vector-set! vector k obj)

k must be a valid index of vector. Vector-set! stores obj in element k of vector. The value returned by vector-set! is unspecified.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 ('("Sue" "Sue")))
  vec)
            → #(0 ("Sue" "Sue") "Anna")
(vector-set! '#(0 1 2) 1 "doe")
            → error ; constant vector
```

library procedure: (vector->list **vector**)

```
library procedure: (list->vector list)
```

Vector->list returns a newly allocated list of the objects contained in the elements of vector. List->vector returns a newly created vector initialized to the elements of the list list.

```
(vector->list '#(dah dah didah))  
          → (dah dah didah)  
(list->vector '(dididit dah))  
          → #(dididit dah)
```

Vektoren

- Vektoren sind *heterogene* Strukturen, die über natürliche Zahlen indiziert werden können.
- Das erste Element eines Vektors hat den Index „0“.
- Der Zugriff auf ein Element ist bei einem Vektor i.A. schneller als bei einer rekursiven Liste.
- Dafür können Vektoren im Gegensatz zu Listen (in Racket) nicht dynamisch vergrößert oder verkleinert werden.
- Verwenden Sie daher
 - Vektoren, wenn Sie wahlfreien Zugriff (random access) auf die Elemente einer Datenstruktur brauchen, und
 - Listen, wenn die Struktur Ihrer Daten flexibel sein muß.

```
> '#(0 (2 2 2 2) "Anna")  
          ; eine Vektorkonstante  
          → #(0 (2 2 2 2) "Anna")  
> (vector 'a 'b 'c)      → #(a b c)  
> (make-vector 4 "a")   → #4("a")  
> (vector-ref '#(1 1 2 3 5 8 13 21)  
           5)   → 8  
> (vector->list '#(dah dah didah))  
          → (dah dah didah)  
> (list->vector '(dididit dah))  
          → #(dididit dah)  
> (vector? '#(1 2 3))   → #t
```

12.3 Verbunde (structures)

Verbunde (structures)

- *Verbunde (structures)* sind algebraische Datentypen, die aus einer endlichen Anzahl von *Feldern* unterschiedlichen Typs zusammengesetzt sind.
- Der Wertebereich eines Verbundes ist das Kreuzprodukt der Wertebereiche der Basistypen.
- Die Kardinalität ist daher das Produkt der Kardinalitäten der Basistypen.

```
(define-struct <s> (<field> ...))  
(define-struct address (street number))
```

Konstruktor, Selektor, Modifikator

Die Deklaration einer *structure* erzeugt automatisch:

1. Ein Typprädiat: <typename>?,
2. einen Konstruktor: make-<typename>,
3. Selektoren für die Felder: <typename>-<fieldname>,
4. Modifikatoren für die Felder: set-<typename>-<fieldname>!

Beispiel: Adressen

```
(define-struct address (street number))  
> (define PrimeMinisterAdr  
     (make-address "Downing_Street" 10)) → ⊥  
> (address? PrimeMinisterAdr) → #t  
> (address-street PrimeMinisterAdr)  
   → "Downing_Street"  
> (address-number PrimeMinisterAdr)  
   → 10  
> (set-address-street!  
    PrimeMinisterAdr "Strand")  
> (set-address-number!  
    PrimeMinisterAdr 3)
```

([Flatt, 2006](#)) A structure type is a record datatype composing a number of fields. A structure, an instance of a structure type, is a first-class value that contains a value for each field of the structure type. A structure instance is created with a type-specific constructor procedure, and its field values are accessed and changed with type-specific selector and setter procedures. In addition, each structure type has a predicate procedure that answers #t for instances of the structure type and #f for any other value.

Defining Structure Types

A new structure type can be created with one of four define-struct forms:

```
(define-struct s (field ...) [inspector-expr])  
(define-struct (s t) (field ...) [inspector-expr])
```

where s, t, and each field are identifiers. The latter form is described in section 4.2. The optional inspector-expr, which should produce an inspector or #f, is explained in section 4.5.

A define-struct expression with n fields defines $4 + 2n$ names:

- * struct:s, a structure type descriptor value that represents the new datatype. This value is rarely used directly.
- * make-s, a constructor procedure that takes n arguments and returns a new structure value.
- * s?, a predicate procedure that returns #t for a value constructed by make-s (or the constructor for a subtype; see section 4.2) and #f for any other value.
- * s-field, for each field, an accessor procedure that takes a structure value and extracts the value for field.
- * set-s-field!, for each field, a mutator procedure that takes a structure and a new field value. The field value in the structure is destructively updated with the new value, and void is returned.
- * s, a syntax binding that encapsulates information about the structure type declaration. This binding is used to define subtypes (see section 4.2). It also works with the shared and match forms (see Chapter 39 and Chapter 24 in PLT MzLib: Libraries Manual). For detailed information about the expansion-time information stored in s, see section 12.6.4.

Each time a define-struct expression is evaluated, a new structure type is created with distinct constructor, predicate, accessor, and mutator procedures. If the same define-struct expression is evaluated twice, instances created by the constructor returned by the first evaluation will answer #f to the predicate returned by the second evaluation.

Examples:

```
(define-struct cons-cell (car cdr))
(define x (make-cons-cell 1 2))
(cons-cell? x) ; => #t
(cons-cell-car x) ; => 1
(set-cons-cell-car! x 5)
(cons-cell-car x) ; => 5

(define orig-cons-cell? cons-cell?)
(define-struct cons-cell (car cdr))
(define y (make-cons-cell 1 2))
(cons-cell? y) ; => #t
(cons-cell? x) ; => #f
; cons-cell? now checks for a different type
(orig-cons-cell? x) ; => #t
(orig-cons-cell? y) ; => #f
```

The let-struct form binds structure identifiers in a lexical scope; it does not support an inspector-expr.

```
(let-struct s (field ...)
  body-expr ...1)
(let-struct (s t) (field ...)
  body-expr ...1)
```

Für die folgenden Datentypen siehe das Handbuch:

- 12.4 Mengen**
- 12.5 Streuspeicherung, Hashtables**
- 12.6 Sequenzen**
- 12.7 Generatoren**
- 12.8 Bilder**

Teil V

Semantik und Korrektheit

13 Semantik

Semantik und Korrektheit



- 11 Semantik
 - Programmiersprachen: Grundbegriffe
 - Wertesemantik und Reduktionsmodelle
 - Die operationale Semantik von Racket
- 12 Korrektheit und Spezifikation
- 13 Rekursion und Induktion

☞ **Anmerkung:** Dieses und die folgenden Kapitel basieren auf dem Buch von [Bird and Wadler, 1988](#). Wir ersparen es uns daher, das Buch auf jeder Seite neu zu zitieren. Die Beispiele im [Bird and Wadler, 1988](#) sind in einer Notation formuliert, die der Programmiersprache Miranda sehr ähnlich ist. Für diese Vorlesung haben wir diese Beispiele nach Racket übersetzt, soweit das möglich war.

Wenn Sie planen, den [Bird and Wadler, 1988](#) als Grundlage für die Vorlesung zu kaufen, dann nehmen Sie unbedingt die englische Originalausgabe. In der deutschen Ausgabe ist nicht nur der Text ins Deutsche übertragen, sondern auch alle Standardfunktionen haben schöne neue deutsche Namen erhalten: `show` heißt dort `zeige`, `zip2` heißt `reissverschluß`, `map` heißt `abbild` usw. Das ist zwar eine konsequente Anwendung des Prinzips der funktionalen Abstraktion und der Namensabstraktion, aber sehr lästig für die praktische Programmierung, denn wenn Sie die Beispiele in Miranda nachprogrammieren wollen, müssen Sie bei jedem Beispiel erraten, wie die Funktion wohl in Miranda heißen mag.

Wenn sie sich intensiver mit den Aspekten der formalen Semantik von Programmiersprachen auseinandersetzen möchte, können Sie zur Vertiefung im Lehrbuch von [Winskel, 1993](#) nachlesen.

13.1 Programmiersprachen: Grundbegriffe

Programmiersprachen: Grundbegriffe

- Programmiersprachen dienen als
 - Notationssysteme für Programme (Algorithmen)
 - Werkzeuge zur Beschreibung von Problemen
- Es gibt hunderte von Programmiersprachen; sie unterscheiden sich bezüglich
 - **Verarbeitungsmodell**
 - * Abstraktionsebenen
 - * Konzepte
 - Syntax, *Semantik von Programmiersprachen* und *Pragmatik*,
 - Anwendungsgebiet.

Warum nicht die natürlichen Sprachen für die Programmierung?

Natürliche Sprachen sind

- mehrdeutig, sowohl lexikalisch als auch strukturell,
Lexikalische Mehrdeutigkeit liegt vor, wenn Zeichen keine eindeutige Bedeutung haben — Beispiel „Heft“: Schulheft oder Messerheft.
Strukturelle Mehrdeutigkeit liegt vor, wenn die grammatischen Strukturen eines Satzes nicht eindeutig ist.
- unpräzise,
- nicht stabil in der Bedeutung,
- aufwendig zu analysieren.

Das folgende Beispiel (Raphael, 1976) zeigt sowohl strukturelle als auch lexikalische Mehrdeutigkeiten:

„Time flies like an arrow.“

Automatisch generierte Interpretationen

(Harvard-Sprachsystem, 1960):



Die Zeit fliegt wie ein Pfeil.



Messe die Zeit von pfeilgleichen Fliegen.



Messe, so wie ein Pfeil, die Zeit von Fliegen.



Zeitfliegen mögen einen Pfeil.



Mehrdeutigkeiten

Lexikalische Mehrdeutigkeiten:

time: Die Zeit, messe die Zeit

flies: fliegt, Fliegen

like: mögen, gleich wie, so wie

Strukturelle Mehrdeutigkeiten:

flies: Ein Subjekt oder das Objekt (Fliegen), Teil einer Nominalphrase (Zeitfliegen) oder Teil des Prädikats (fliegt).

like: Ebenso mehrdeutig in der Struktur.

Ein Beispiel für strukturelle Mehrdeutigkeit

Englisch: I saw the man on the hill with the telescope.

Deutsch: Ich sah den Mann auf dem Berg mit dem Fernrohr.

Mehrdeutigkeiten:

- Wo ist das Fernrohr? Wie groß ist es?
- Habe ich den Mann von ferne gesehen oder ihn getroffen?
- Wer hat das Fernrohr? Der Mann oder ich?
- Wer war auf dem Berg? Ich, der Mann oder beide?

Diese Mehrdeutigkeiten sind für die menschliche Kommunikation kein Problem. Wir werten Sprache immer im Kontext unserer Erfahrungen und unserer Vorstellungen über sinnvolle und unsinnige Aussagen aus.

Wir Menschen sind auch gut darin, unvollständige Sätze oder grammatisch falsche Sätze zu verstehen. Gelegentlich verwenden wir solche Konstrukte bewußt als stilistisches Mittel; siehe beispielsweise den alten Werbespruch: „Ich bin zwei Öltanks.“

Umgekehrt können wir syntaktisch einwandfreie natürlich sprachliche Texte erzeugen, die keine Bedeutung haben, siehe beispielsweise Lewis Carrolls „Jabberwocky“ (Carroll, 1951).

- Für die Kommunikation mit Maschinen ist es bedeutend praktischer, eigene Sprachen zu haben, deren Bedeutung stabil ist (und keinen kulturellen Entwicklungen unterliegt) und vor allem mit begrenztem Aufwand vereinbart, eindeutig beschrieben und eindeutig ermittelt werden kann. Zum wohldefinierten Umfang der Sprache sollten nur solche Äußerungen gehören, die sowohl grammatisch korrekt sind als auch eine wohldefinierte Bedeutung haben.
- Unsere natürlichen Sprachen sind meist redundant, d.h. kleine Variationen im Satz verändern den Sinn nicht wesentlich. Programmiersprachen dagegen haben oftmals sehr viel weniger Redundanz, so daß meist ein einzelnes verändertes Zeichen zu einem völlig anderen Sinn führt. Beispielsweise löscht das Unix-Shell-Kommando `rm *.tmp` alle Files mit der Kennung `tmp`. Nimmt man bei der Eingabe nicht schnell genug den Finger von der `Shift`-Taste und tippt versehentlich: `rm *>tmp`, dann werden *alle* Files im Verzeichnis gelöscht, unabhängig von der Kennung, und in “`tmp`” steht das Protokoll dazu.

Ein syntaktisch korrekter, aber sinnloser Text

Jabberwocky



*'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome rath outgrabe.
"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"*

Lewis Carroll

Navigation icons: back, forward, search, etc.

He took his vorpal sword in hand:
Long time the manxome foe he sought –
So rested he by the Tumtum tree,
And stood awhile and thought.

And, as in uffish thought he stood,
The Jabberwock, with eyes of flame
Came whiffling through the tulgey wood,
And burbled as it came!

One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with his head
He went galumphing back.

"And hast thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!"
He chortled in his joy.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome rath outgrabe.

Programmiersprache vs. natürliche Sprache

Eine Programmiersprache muß im Gegensatz zur natürlichen Sprache präzise und eindeutig sein.

Eine präzise Definition der Semantik von Programmiersprachen ist notwendig, damit

- die EntwicklerInnen von Compilern und Interpretern eindeutige Vorgaben haben
- und es möglich ist, das Verhalten von Programmen und deren Ausgaben eindeutig vorherzusagen und
- die Korrektheit der Programme und der Werkzeuge zur Programmierung zu beweisen.

Programmiersprachen sind formale Sprachen mit einer wohldefinierten Syntax und Semantik. Die Verwendung wird durch die Pragmatik geregelt.

Programmiersprache

Eine **Programmiersprache** über einem Alphabet Σ besteht aus

- einer *formalen Sprache* $L \subseteq \Sigma^*$, der Menge der syntaktisch richtigen Programme,
- einer Vorschrift, die syntaktisch richtigen Programmen eine Bedeutung zuordnet, der *Semantik*,
- und Vereinbarungen zur richtigen Verwendung der Sprache, der *Pragmatik*.

Anmerkung: **zur Pragmatik**

- Im Gegensatz zu Syntax und Semantik wird die Pragmatik nicht formal spezifiziert, sondern in Handbüchern informell beschrieben.
- Wichtige Aspekte der Pragmatik sind
 - das Verarbeitungsmodell und der Programmierstil,
 - Entwurfsmuster für typische Lösungen,
 - Konventionen zur Dokumentation.

Programmiersprache

Definition: 39 (Programmiersprache)

Seien Σ, Δ und Φ drei Alphabete:

- Σ das Programmalphabet,
- Δ das Eingabealphabet und
- Φ das Resultatalphabet.

Eine Programmiersprache ist ein Paar (L, f) .

L ist eine formale Sprache über Σ und

$f : M \rightarrow \Phi$ ist eine berechenbare Funktion mit $M \subseteq L \times \Delta^*$, die also für gewisse $x \in L$ und gewisse $y \in \Delta^*$ einen Wert $z = f(x, y)$ mit $z \in \Phi^*$ besitzt.

f heißt *Semantik*.

— Verfahren, um die Semantik zu definieren: —

Denotationelle Semantik: Eine deklarative Beschreibung der Semantik, bei der ein Programm als eine statische Beschreibung von Algorithmen und Werten gesehen wird. Die Spezifikation ist völlig unabhängig von speziellen Implementationen.

Operationale Semantik: Die Bedeutung der primitiven Grundoperationen oder das Verfahren zur Berechnung der Semantik, und damit die Ausführung, wird spezifiziert.

So wird über die Operationen das *Verhalten* des Programms, der erzeugte Prozeß, eindeutig festgelegt.

Semantik von Programmiersprachen

Definition 39 betont, daß es einen Algorithmus geben muß, der die Bedeutung eines Programms eindeutig ermitteln und das Programm ausführen kann. Das ist die *operationale Semantik*.

Für prozedurale Programmiersprachen (Java, C usw.) wird die Semantik üblicherweise nur über die operationale Semantik spezifiziert.

Für deklarative Programmiersprachen, deren Syntax sich an mathematische Kalküle anlehnt (Prolog, Racket, Miranda usw.), wird eine denotationelle Semantik im entsprechenden Kalkül spezifiziert.

☞ Anmerkung: Verwechseln Sie bitte nicht den deklarativen Programmierstil im deklarativen (logischen) Verarbeitungsmodell mit den deklarativen Programmiersprachen. Racket und Miranda werden zu den deklarativen Programmiersprachen gezählt, aber diese Sprachen sind nicht für das deklarative (logische) Verarbeitungsmodell entworfen.

Operationale und denotationelle Semantik

Sprache	denotationelle Semantik	Operationale Semantik
Java	intuitiv	Java virtual machine
Racket	λ -Kalkül	Der evaluator eval
Prolog	Logik	Unifikation und Suche

Bezugstransparenz

Definition: 40 (*Bezugstransparenz*)

Eine Sprache, bei der die denotationelle und die operationale Semantik zusammenfallen, nennt man *referentiell transparent* bzw. *deklarativ*.

- Referentielle Transparenz bedeutet insbesondere, daß das Berechnungsresultat vom Zustand der (abstrakten) Maschine unabhängig ist.
- Die „Bezugstransparenz“ oder „Referentielle Transparenz“ wird auch *Konfluenz* oder nach den Entdeckern „Church-Rosser-Eigenschaft“ genannt.

Widersprüche in der Semantik

Die denotationelle und operationale Semantik sind nicht notwendigerweise identisch:

- Auch wenn die denotationelle Semantik eindeutig ist, kann es sein, daß der Algorithmus der operationalen Semantik die Bedeutung nicht berechnen kann.
- Die denotationelle Semantik läßt es zu, prinzipiell unentscheidbare Probleme zu formulieren.

Hybride Sprachen: Schlupflöcher

- Sprachelemente, die Schlupflöcher in andere Programmierstile bieten, können dazu führen, daß die denotationelle Semantik und operationale Semantik nicht mehr identisch sind,
 - imperative Sprachelemente in funktionalen Programmiersprachen (Seiteneffekte),
 - funktionale Sprachelemente in relationalen Programmiersprachen.

☞ **Anmerkung: Semantik:** Während ihres Studiums wird Ihnen der Begriff „Semantik“ in unterschiedlichen Bedeutungen begegnen, beispielsweise unterscheidet sich die Verwendung des Begriffs in der mathematischen Logik von der hier eingeführten in der Theorie der Programmiersprachen. Und außerhalb der Informatik gibt es noch viele andere Verwendungen, deren Erklärung ein eigenes Studium erfordern würde.

Die Semantik ist eine fachübergreifende Wissenschaft, die die Beziehung zwischen Sprache und Welt untersucht. Zentrale Themen sind die Bedeutung sprachlicher Ausdrücke sowie die kommunikative Funktion von Sprache. Semantische Untersuchungen spielen nicht nur in der Informatik eine Rolle, sondern vor allem auch in der Philosophie, Linguistik, Anthropologie, in der Künstlichen Intelligenz und der Kognitionswissenschaft (siehe [Strube, 1996](#)).

☞ **Beachte:** Wenn wir hier die Begriffe „Zeichen, Symbol, Semantik“ verwenden, dann meinen wir ihre engere Bedeutung in der Theorie der Programmiersprachen und beziehen uns ausschließlich auf die formale Spezifikation einer Bedeutung für algorithmische Sprachen, auch wenn wir zur Veranschaulichung Beispiele aus dem Bereich der menschlichen Kommunikation heranziehen.

13.2 Wertesemantik und Reduktionsmodelle

Wertesemantik: Der Wert eines Ausdrucks

- Ein funktionaler Ausdruck oder Term wird einzig und allein dazu verwendet, einen *Wert* zu bezeichnen.

Definition: 41 (Bedeutung eines Terms)

Die Bedeutung eines Ausdrucks ist der Wert, den er bezeichnet (denotiert).

- Ein Ausdruck hat *keine andere Wirkung*, als daß er einen Berechnungsprozeß auslöst, der diesen Ausdruck auf seinen Wert reduziert.
- Es gibt keine versteckten Wirkungen (Seiteneffekte), d.h. konkret, daß durch die Auswertung eines Terms die Umgebung des Terms nicht beeinflußt wird.

Wie wird der Wert ermittelt?

Das Substitutionsmodell

- Der Wert eines Ausdrucks hängt nur von den Werten der Ausdrücke ab, aus denen er sich zusammensetzt.
- Da nur der Wert eines Ausdrucks relevant ist, da er ja sonst keine Wirkung hat, können wir einen Ausdruck jederzeit in anderen Termen durch Terme ersetzen, die denselben Wert denotieren.
- $4 * 4 + 3 * 3 = 4 * 4 + 9 = \sqrt{256} + \sqrt{81} = 25$.
- Diese Eigenschaft haben wir schon als „Bezugstransparenz“ oder „Referentielle Transparenz“ kennengelernt.

Reduktion nach dem Substitutionsmodell

- Wir nutzen die Church-Rosser-Eigenschaft, um Ausdrücke auf einfachere Ausdrücke zu reduzieren und so ihren Wert zu ermitteln.
- Dieses Modell der Auswertung heißt *Substitutionsmodell*, da wir Terme durch andere Terme mit gleichem Wert ersetzen.

Definition: 42 (Der Wert eines Terms)

Der Wert eines Ausdrucks ist Wert des einfachsten äquivalenten Terms.

Definition: 43 (Normalform)

Ein Term ist kanonisch oder in Normalform, wenn er nicht weiter reduziert werden kann.

☞ Beachte: Zur Semantik von Racket Racket ist keine reine funktionale Sprache. Racket ermöglicht auch die imperative Programmierung, indem die Sprache Modifikatoren vorsieht, mit denen der Zustand von Objekten geändert werden kann. Daher kann die Semantik von Racket nicht alleine durch die Wertesemantik erklärt werden. Wir werden daher zunächst Racket als eine referentiell transparente Sprache benutzen und diejenigen Sprachelemente ignorieren, die Seiteneffekte erzeugen (alle Modifikatoren, also alle Sprachelemente, deren Namen mir einem „!“ enden.)

☞ Anmerkung:

- Die Begriff *Evaluierung*, *Vereinfachung* und *Reduktion* werden synonym verwendet.
- Wir müssen zwischen den Werten eines Ausdrucks und den Ausdrücken, die sie denotieren, unterscheiden. Der einfachste äquivalente Term ist noch nicht der Wert eines Ausdrucks, sondern auch dieser denotiert nur einen Wert. Das soll uns aber nicht weiter verwirren, da wir bei der Programmierung die Werte ja auch nicht abstrakt angeben können, sondern immer eine kanonische Repräsentation angeben; wir können beispielsweise nicht den abstrakten Wert „vier“ manipulieren, wohl aber eine kanonische Repräsentation „4“ oder „IV“. Für die Manipulation von Werten können wir uns eben nur auf Repräsentationen beziehen. Auch Racket gibt uns als Ergebnis der Termreduktion eine kanonische Repräsentation des Wertes aus.
- Das Konzept einer kanonischen Form hängt sowohl von der Syntax der Terme als auch von der Definition der Reduktionsregeln ab. Manche Werte haben keine kanonische Darstellungen, und andere wieder keine endlichen. Die Zahl π können wir wohl durch einen Algorithmus beliebig genau berechnen lassen, aber wir können sie nie als Dezimalzahl vollständig hinschreiben, genauso wenig wie die Zahl ∞ (unendlich).

Innere Reduktion

- Mit der Definition

(**define** (**square** x)(* x x))

können wir reduzieren:

Variante 1:

$$\begin{aligned} (\text{square } (+ 3 4)) &\longrightarrow (\text{square } 7) ; (+) \\ &\longrightarrow (* 7 7) ; (\text{square }) \\ &\longrightarrow 49 ; (*) \end{aligned}$$

- Diese Art der Reduktion heißt *innere Reduktion*, weil die Terme von innen nach außen reduziert werden.

Äußere Reduktion

Variante 2:

$$\begin{aligned} (\text{square } (+ 3 4)) &\longrightarrow (* (+ 3 4) (+ 3 4)) ; \text{square} \\ &\longrightarrow (* 7 (+ 3 4)) ; (+) \\ &\longrightarrow (* 7 7) ; (+) \\ &\longrightarrow 49 ; (*) \end{aligned}$$

Diese Art der Reduktion heißt *äußere Reduktion*, weil die Terme von außen nach innen reduziert werden.

 **Beachte:**

Satz: 44 (Reihenfolgeunabhängigkeit des Wertes)

Wenn die *Bezugstransparenz* gewährleistet ist, dann ist das Ergebnis unabhängig von der Reduktionsreihenfolge und die *innere Reduktion* und *äußere Reduktion* führen zum selben Ergebnis.

- Die Reduktionsreihenfolge bestimmt aber den Aufwand der Berechnung.
- Von der Reduktionsreihenfolge kann es abhängen, ob überhaupt ein Wert ermittelt werden kann (z.B. bei partiellen Funktionen).
- Miranda und Haskell verwenden die äußere Reduktion, Racket die innere.

Reduktionsaufwand

Äußere Reduktion: Die Äußere Reduktion ist ungünstig, wenn ein Parameter mehrfach benötigt wird, da er so mehrfach ausgewertet wird.

```
(define (hoch4 x) (* x x x x))
```

Innere Reduktion: Die innere Reduktion ist ungünstig, wenn die Argumente der Funktion gar nicht zur Berechnung des Resultats benötigt werden.

```
(define (konstant x y z) 1)
```

Die Begriffe innere und äußere Reduktion beziehen sich darauf, in welcher *Reihenfolge* die Terme des Ausdrucks bei der **Auswertung** substituiert werden. Daneben wird die Reduktionsstrategie auch daraufhin unterschieden, ob der Ausdruck vollständig oder nur teilweise reduziert wird, um den Wert zu ermitteln. Wir unterscheiden zwischen **vorgezogener** und **verzögter Auswertung**.



Vorgezogene Auswertung

Definition: 45 (eager evaluation)

Man spricht von

- **vorgezogener Auswertung** (engl. eager evaluation)
- oder **applikativer Auswertung** (engl. applicative order evaluation),

wenn alle Argumente einer Funktion ausgewertet werden, bevor die Funktion darauf angewendet wird (Racket, Common Lisp).



Verzögerte Auswertung

Definition: 46 (lazy evaluation)

Man spricht von

- **verzögerte Auswertung** (engl. lazy evaluation)
- oder **Normalform der Auswertung** (engl. normal order evaluation),

wenn die Auswertung aller Argumente verzögert wird, bis sich zeigt, daß sie für die Reduktion des Ausdrucks erforderlich sind (Miranda, Haskell, Lazy Racket).

13.3 Die operationale Semantik von Racket

13.3.1 Die Auswertung funktionaler Ausdrücke

S-Expressions: Operationale Semantik

Racket-Ausdrücke werden nach dem *Substitutionsmodell* ausgewertet.

Zahlen: Der Wert einer Zahl (1, 2.3, ...) ist die Zahl selbst.

Bezeichner: Der Wert eines Bezeichners (Namens) ist der Wert, an den er lexikalisch und dynamisch gebunden ist.

Zusammengesetzte Ausdrücke werden durch Substitution zu atomaren Ausdrücken reduziert. Dieser Vorgang ist also *rekursiv*.

Special form expressions werden anders ausgewertet als reguläre funktionale Ausdrücke.

Anmerkung: Über Bindungen von Namen werden wir später noch ausführlicher sprechen. Racket ist nicht immer referentiell transparent.

Reduktionsstrategie für funktionale Ausdrücke

Die Auswertung ist *applikativ*:

- Zunächst werden die Argumente evaluiert. Die Reihenfolge ist undefined.
- Der erste Teilausdruck hinter der öffnenden Klammer wird zuletzt ausgewertet und das Ergebnis als Funktion auf die ausgewerteten Argumente angewendet.
Hinter einer öffnenden Klammer erwartet Racket *immer* eine Funktion (oder special form)!
- Die Strategie wird *rekursiv* auf Teilausdrücke angewendet.
- Die Reduktionsstrategie ist also: **Strikte**, innere Reduktion + vorgezogene Auswertung.

Modifikatoren

Mit **set!** können wir Variablen einen Wert geben. Das Ausrufungszeichen im Namen weist darauf hin, daß die Funktion Seiteneffekte verursacht.

```

> (define Wurzel 1)      → ⊥
> (set! Wurzel (sqrt 4)) → ⊥
> wurzel                  → 2.0
> (* Wurzel Wurzel)       → 4.0
> (set! Wurzel (sqrt 8)) → ⊥
> (* Wurzel Wurzel)       → 8.0

```

- Modifikatoren verletzen die **Bezugstransparenz**.
- Wir sollten sie daher nur wohlüberlegt einsetzen.

☞ **Anmerkung:** An diesen einfachen Beispielen sehen sie, daß Racket im Gegensatz zu klassischen imperativen Sprachen, wie C, Pascal, Java wirklich interaktiv ist. Wir können im toplevel, im interaktiven Dialog, alle Definitionen und Namensbindungen ändern, ohne dazu ein Programm editieren und neu laden zu müssen.

☞ **Beachte: Seiteneffekte:** Alle Funktionen, deren Namen mit einem Ausrufungszeichen enden, ändern den Zustand der Berechnung und können Seiteneffekte hervorrufen. Ihre Verwendung verletzt die referentielle Transparenz. Die Semantik eines Terms kann nicht mehr nur allein durch den denotierten Wert beschrieben werden. Operationale und denotationelle Semantik fallen nicht mehr zusammen. Gewissenhafter Programmierer verwenden diese Konstrukte daher sehr sparsam und nur nach sorgfältiger Überlegung, denn ihr Einsatz erschwert die Korrektheitsbeweise und erhöht damit die Fehleranfälligkeit des Programms. In manchen reinen funktionalen Programmiersprachen (Miranda, Haskell) gibt es daher keine Funktionen mit Seiteneffekten.

Set! kann sehr nützlich sein, wenn Sie nach Programmfehlern suchen – ein Programm „debuggen“, da Sie so in einfacher Weise Testdaten schaffen können.



★ **Anmerkung: Entwanzen** Fehlersuche wird „debugging“ genannt; wir „entwanzen“ das Programm. Der Ausdruck wird Grace Murray Hopper zugeschrieben, die ihn einer Anekdote nach geprägt hat, als sie nach tagelanger Fehlersuche an einer der damals noch sehr teuren und empfindlichen Rechenanlagen ein verkohltes Insekt mittels einer Pinzette aus dem Prozessor präparierte.

Über die Art des Insekts, das sie triumphierend präsentierte, sind die Quellen unterschiedlicher Meinung: Die meisten nennen das Insekt „moth“ (Falter), aber ich habe auch schon Versionen gehört, nach denen es sich um eine Heuschrecke oder eine Kakerlake gehandelt haben soll. Und nach anderen Quellen, war es nicht einmal Grace Hopper, die diesen Begriff geprägt hat ... (siehe den „Jargon-File“):

<http://www.huis.hiroshima-u.ac.jp/Computer/Jargon/LexiconEntries/Bug.html>)

Grace Murray Hopper (1906–1992) ist neben Lady Ada Lovelace die große Pionierin der Informatik. Sie hat für die Eckert-Mauchly Computer Corp. den ersten Compiler geschaffen, der symbolische Ausdrücke in Maschinensprache übersetzt hat und die ersten Programmabibliotheken entwickelt. Abwechselnd mit ihrer Tätigkeit als Forscherin hat sie immer wieder aktiven Dienst bei der Marine versehen, zuletzt im Range eines „Rear Admiral“. 1997 hat die Navy einen Zerstörer nach Grace Hopper benannt. Mehr zu Grace Hopper finden Sie unter

http://www.navsea.navy.mil/hopper/hopper_grace.html,
http://www.norfolk.navy.mil/chips/grace_hopper/file2.htm

13.3.2 Auswertung von Special Form Expressions

Special Form Expressions

- Ausdrücke, die mit **define**, **quote**, **if** oder anderen speziellen Operatoren beginnen, werden anders ausgewertet als reguläre funktionale Racket-expressions.
- Diese Ausdrücke heißen *special form expressions*, kurz *special forms*,
- und die Operatoren **define**, **quote** usw. *special form operators*.

Warum ist eine Abweichung von der applikativen Reduktion, strikte Auswertung der Argumente von links nach rechts, nötig?

quote

```
(define zahlen (quote (1 2 3))
(define zahlen2 '(1 2 3))
```

- Quote soll ja gerade verhindern, daß das Argument ausgewertet wird.
- Wenn eval auf den *special form operator* **quote** oder das Quotierungszeichen stößt, wird das Argument unausgewertet als Resultat zurückgegeben.

define, set!

```
(define zahlen (quote (1 2 3))
(define zahlen2 '(1 2 3))
```

- Auch **define** und **set!** dürfen das erste Argument nicht wie üblich auswerten,
- da das zu definierende Symbol ja eventuell noch nicht an einen Wert gebunden ist und die Evaluierung des Symbols einen Laufzeitfehler auslösen könnte.

if, case, cond

```
(if <bedingung>
  <then-klausel>
  <else-klausel>)
```

- Bei bedingten Ausdrücken ist erst klar, welche der Alternativen ausgewertet werden darf, wenn die Bedingung getestet wurde.
- Die special form expressions **if**, **cond** und **case** prüfen daher zunächst die Bedingungen und werten dann gezielt nur die gewünschte Alternative aus.

Special Forms: Beispiele

```
> (define dont-eval '(/ 1 0))
      → ⊥
> (define x 0)      → ⊥
> (if (= x 0) 0 (/ 1 x))
      → 0
> (set! x 4)       → ⊥
> (if (= x 0) 0 (/ 1 x))
      → 0.25
```

Wichtige special forms

Special Form Operators

define	Definiere eine Variable
set!	Binde eine Variable
let , let*	Binde lokale Variable
case, cond	Fallunterscheidung
if	Bedingter Ausdruck
quote (')	Nehme Daten wörtlich
delay	Verzögere die Auswertung

Sequenzen

begin erzwingt die Auswertung in der angegeben Reihenfolge;

- nützlich zum Erzeugen von Druckausgabe,
- zum Erzeugen von files,
- zum Einlesen von Eingaben usw.

(**begin** <expression1> <expression2> ... ,...)

```
> (define x 0)
  (begin (set! x 5)
    (+ x 1))      →  6

> (begin (display "4_plus_1_equals_")
  (display (+ 4 1)))
4 plus 1 equals 5      →  ⊥
```



Der undefinierte Wert

Denotationelle Semantik

Manche Terme können nicht zu einem wohldefinierten Wert reduziert werden, auch wenn sie syntaktisch wohlgeformt sind, siehe beispielsweise:

```
> (cond [#f 1])      →  ⊥
```

- Damit jeder syntaktisch wohldefinierte Term ausnahmslos einen Wert hat, führen wir *in der denotationellen Semantik* für Terme mit undefiniertem Wert den speziellen Wert \perp (gesprochen „bottom“) ein.
- Dieses ermöglicht uns, auch *partielle Funktionen* zu untersuchen.

Implementationsabhängigkeiten

Der Wert einiger Ausdrücke ist im Revised Report als undefined spezifiziert. Das bedeutet, daß bei der Implementation von Racket keine bestimmten Resultate für diese Ausdrücke garantiert werden müssen. Das Resultat ist implementationsabhängig.

SCM-Scheme kennzeichnet den undefinierten Wert konsequent mit #<unspecified>, xscheme und UBM-Scheme geben je nach Sprachkonstrukt unterschiedliche Resultatwerte zurück: den Namen der definierten Variablen bei einem define, die leere Liste in anderen Fällen, Fehlermeldungen usw.

DrRacket gibt in manchen Fällen eine leere Ausgabe zurück.

Repräsentation des undefinierten Wertes

Beachte:

- Sie dürfen im Programm niemals darauf bauen, daß der undefinierte Wert eine bestimmte Repräsentation hat.
- Implementationsabhängigkeiten sind immer Schwachstellen im Programm, die Ihnen bei Versionsänderungen viel Ärger bereiten können.

Undefinierter Wert vs. leerer Wert

In Racket gibt es zwei Varianten des undefinierten Wertes:

1. einen für beabsichtigte Undefiniertheit: #<void>
2. und einen für fehlerhafte Undefiniertheit: #<undefined>

Der leere Wert #<void>:

Dieser Wert ist das Resultat von Ausdrücken, die kein Resultat berechnen sollen und nur durch Seiteneffekte wirken, wie Display, define.

Der undefinierte Wert #<undefined>:

Das ist der Wert von uninitialisierten Variablen.

Der leere Wert: void

Der leere Wert kann in Racket

- definiert erzeugt, abgefragt und an Variablen gebunden werden.
- Die Funktion *void* nimmt beliebig viele Argumente und gibt den leeren Wert zurück.
- Das Prädikat *void?* prüft, ob ein Wert leer ist.
- Der leere Wert (als alleiniges Resultat) wird als leeres Wort gedruckt.

```
> (void 1 2 3) →  
> (void? (void 1 2 3)) → #t  
> (define x (void 1 2 3)) →  
> x →  
> (void? x) → #t  
> (list (void) (void) (void))  
→ (#<void> #<void> #<void>)
```

Der Typ des leeren Wertes

Der leeren Wert hat einen singulären Datentyp:

```
> (null? (void)) → #f  
> (list? (void)) → #f  
> (number? (void)) → #f  
> (symbol? (void)) → #f  
> (void? (display "Hallo")) → #t  
Hallo  
> (void? (when #f 1)) → #t  
> (void? (when #t 1)) → #f  
> (void?  
  (if (< pi 3) 'gross (void)))  
→ #t
```

⌚ Anmerkung: Ein Auszug aus dem PLT-Programming Guide:

A constant that prints as #<undefined> is used as the result of a reference to a local binding when the binding is not yet initialized. Such early references are not possible for bindings that correspond to procedure arguments, let bindings, or let* bindings; early reference requires a recursive binding context, such as letrec or local defines in a procedure body. Also, early references to top-level and module-level bindings raise an exception, instead of producing #<undefined>. For these reasons, #<undefined> rarely appears.

Der undefinierte Wert: undefined

Der Wert #<undefined> tritt selten auf:
nur dann, wenn auf lokale, ungebundene Variable zugegriffen wird.

```
> (define (strange)
  (define x x)
  x)
> (strange) → #<undefined>
```

Striktheit einer Funktion

Definition: 47

<1->[Striktheit einer Funktion] Eine Funktion f heißt **strikt**, wenn $f(\perp) = \perp$, d.h. die Funktion hat nur für definierte Argumente einen definierten Wert.

Definition: 48

<2->[Strikte Semantik] Eine Programmiersprache hat eine strikte **Semantik von Programmiersprachen**, wenn alle definierbaren Funktionen strikt sind.

Das Gegenteil heißt nicht-strikte Semantik. (*Bird and Wadler, 1988*)

Strikte Auswertung

Definition: 49 (Strikte Auswertung)

(oder Reduktion v. Termen:) Eine Programmiersprache verwendet eine **strikte** Auswertung, wenn alle Argumente einer Funktion vor der Anwendung der Funktion ausgewertet werden, wie beispielsweise in Racket und Common Lisp.

- Strikte Auswertung ist die Kombination von vorgezogener Auswertung (eager evaluation) und innerer Reduktion.

Striktheit in Racket

- Alle Funktionen werden in Racket grundsätzlich **strikt** ausgewertet.
- **Racket** hat nicht-strikte *special form expressions*,

if , cond, case, and, or usw.

- **Miranda, Haskell** und **Lazy Racket** dagegen haben eine nicht-strikte Semantik. Diese Sprachen verwenden generell die *äußere Reduktion* und verzögern die Auswertung von Teilausdrücken, bis deren Wert benötigt werden.

Vorteile nicht-strikter Semantik

- Beweise werden einfacher, weil Ausnahmen nicht berücksichtigt werden müssen: Mit der Definition

```
(define (three x) 3)
```

gilt: für alle x, insbesondere auch für $x = \perp$:

```
(+ 2 (three x)) = 5
```

Wir können in Beweisen *immer* „3“ für (three x) substituieren, unabhängig vom Wert von x.

In Racket dagegen gilt

```
(three  $\perp$ ) =  $\perp$ .
```

Vorteil 2: Kontrollstrukturen

Beispiel: 50

Mit strikter Auswertung kann nicht zwischen Alternativen gewählt werden.

```
(define (my-if wenn dann sonst)
  (cond [wenn dann]
          [else sonst]))
> (define x 0) → x
> (my-if (> x 0) ; strikte Auswertung
          (/ 1 x) 'nenner-null)

          /: division by zero
> (if (> x 0) ; nicht strikt
          (/ 1 x)
          'nenner-null) → nenner-null
```

Nachteile nicht-strikter Sprachen

- Äußere Reduktion kann zu aufwendigen Mehrfachberechnungen führen.
- Verzögerte Auswertung erfordert einen größeren Verwaltungsaufwand.
- Nicht-strikte Auswertung ist fatal, wenn die **Bezugstransparenz** nicht gewährleistet ist (Seiteneffekte, Programmzustände), da wir uns dann darauf verlassen können müssen, ob und wann Terme ausgewertet werden.

Problem

- + Nicht-strikte Semantik und verzögerte Auswertung sind meist sehr angenehm und gelegentlich unverzichtbar.
- Beide können zu erhöhtem Rechenaufwand führen und sind gefährlich bei bestimmten Programmierstilen, beispielsweise der imperativen oder objektorientierten Programmierung.
- *Eigentlich müssen wir beim Programmieren selbst bestimmen können, welche Reduktionsstrategie wir anwenden wollen.*

Der Kompromiß in Miranda und Racket

Lisp und Racket verwenden generell eine strikte Semantik und bietet Kontrollstrukturen, um verzögerte Auswertung und nicht-strikte Funktionen punktuell und gezielt erzwingen zu können.

Wenn wir es nicht explizit verhindern, können wir darauf vertrauen, daß jedes Argument ausgewertet wird.

Miranda und Haskell dagegen verwenden generell eine nicht-strikte Semantik und bieten die Funktionen `force` und `seq`, um die Auswertung von Termen zu erzwingen und die Reihenfolge der Auswertung bestimmen zu können — nützlich beispielsweise bei Laufzeitmessungen.

Lazy Racket

Die experimentelle Racket-Sprache „Lazy Racket“ verwendet generell die verzögerte Auswertung.

- Keine Laufzeitfehler bei undefinierten Argumenten, wenn diese nicht benötigt werden.

```
#lang lazy
(define (immer3 x)
  3)

> (immer3 (/ 1 0)) → 3
```

Lazy Racket

Funktionen als Kontrollstrukturen

```
#lang lazy

(define (my-if wenn dann sonst)
  (cond [wenn dann]
        [else sonst]))
```

☞ In Lazy-Racket können Funktionen als Kontrollstrukturen wirken.

```
> (define x 0)
> (my-if (not (zero? x))
           (/ 10 x) (void)) → ⊥
>
```

In Lazy-Racket können wir sogar potentiell unendliche Datenstrukturen definieren und Rekursion ohne Abbruchbedingung schreiben.

Lazy Racket: Unendliche Listen

Beispiel: 51 (Natürliche Zahlen)

Eine Liste als unendlicher Strom der natürlichen Zahlen:

- Die Elemente der Liste werden erst berechnet, wenn auf ein konkretes Element zugegriffen wird.
- Die Struktur #<promise> beschreibt die Berechnung.

```
#lang lazy
(define (natsAbN n)
  (cons n (natsAbN (+ n 1)))) 

> (list-ref (natsAbN 1) 1000) → 1001
> (natsAbN 1) → (1 . #<promise>)
```

Erzwingen der Berechnung

```
(define (echo xs)
  (when (pair? xs)
    (display (car xs))
    (display " ")
    (echo (cdr xs)))))

> (echo (natsAbN 1)) →
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49 50 51
52 53 54 55 56 57 58 59 60 61 62 63 ...
user break
```

14 Korrektheit und Spezifikation

14.1 Korrektheit

Korrektheit von Programmen

Definition: 52 (*Korrektheit*)

Ein Programm heißt *korrekt*, wenn es genau die Spezifikation erfüllt, also für alle Argumente des Definitionsbereichs die korrekten Resultate berechnet. (Züllighoven, 1995)

Definition: 53 (*Partielle Korrektheit*)

Ein Programm heißt *partiell korrekt*, wenn es, sofern es terminiert, korrekte Resultate liefert.

Definition: 54 (*Totaler Korrektheit*)

Ein Programm heißt *total korrekt*, wenn es für alle Eingaben mit korrekten Resultaten terminiert.

Sicherstellung der Korrektheit

- Formaler Korrektheitsbeweis
- Vollständiger Test
- Systematische Konstruktion des Programms (Programmsynthese, Programmtransformation)

☞ Beachte: **Formaler Korrektheitsbeweis:** Wir erstellen zunächst ein Programm und beweisen anschließend die Korrektheit. Das Beweis-Verfahren ist hier „bottom-up“: Wenn wir unser Programm modular aufgebaut haben, werden wir zunächst die Korrektheit der Basismodule beweisen und dann darauf aufbauend die Korrektheit komplexerer Module.

Jede Funktion (jedes Programm), die wir verwenden, muß dazu mit Zusicherungen (assertions) versehen werden. Zusicherungen sind Aussagen über die Argumente und die Resultate, die das Verhalten einer Funktion oder eines Programms formal spezifizieren.

Die Beweise werden meistens zweiteilig durchgeführt. Wir beweisen zuerst die partielle Korrektheit und unabhängig davon das Terminieren. Bei allen Beweisen gehen wir davon aus, daß die Systemfunktionen ihre Spezifikationen erfüllen und daß die Argumente, die übergeben werden, korrekt sind.

Je klarer die Struktur unseres Programms ist, desto einfacher wird der Beweis. Spaghetti-Code, dessen Korrektheit wir nicht beweisen können, sollten wir gleich wieder löschen, denn solcher Code zeigt uns, daß wir das Problem, das wir lösen wollten, noch gar nicht richtig verstanden haben. Wir sollten ein paar Entspannungübungen machen und nochmal gründlich nachdenken, ehe wir wieder zu programmieren anfangen.

Vollständiger Test

Ein experimenteller Korrektheitsbeweis; wir erproben systematisch alle typischen Konstellationen von Eingabewerten, um durch vollständige Erfassung aller möglichen Fälle experimentell die Korrektheit zu überprüfen. Es ist aber dennoch formal zu beweisen, daß unser Test alle Fälle abdeckt.

Ein Test ist immer nötig, auch wenn wir die Korrektheit formal bewiesen haben, denn es kommt häufig vor, daß beim Korrektheitsbeweis dieselben Denkfehler gemacht werden, die man schon beim Programmieren gemacht hat. Außerdem können sich im Zusammenspiel zwischen Systemumgebung und Programm Fehler zeigen, die von den Spezifikationen nicht abgedeckt sind.

Systematische Konstruktion des Programm

Das Programm wird mit einem systematischen Verfahren hergeleitet, bei dem nur korrekte Programme entstehen können. Bei der Programmsynthese wird die Spezifikation automatisch in ein Programm umgewandelt; bei der Programmtransformation wird ein korrektes Programm in ein neues korrektes Programm mit anderen Eigenschaften umgewandelt.

☞ Beachte: Alle diese Ansätze (außer dem zweiten) zielen darauf ab, fehlerhafte Programme gar nicht erst entstehen zu lassen, also Fehlervermeidung anstelle von Fehlersuche. Niemand sollte damit angeben, Tage und Nächte mit der Suche nach Fehlern verbracht zu haben, die man hätte vermeiden können. Dagegen können wir sehr, sehr stolz sein, wenn ein solide entworfenes Programm auf Anhieb korrekte Resultate liefert.

Debugging is a disgrace!

John McCarthy, 1983

14.2 Spezifikation und automatischer Test

Zusicherungen (assertions)

- Jede nicht-triviale Funktion sollte mit Zusicherungen (assertions) spezifiziert werden, die die korrekte Verwendung und Funktionalität beschreiben.

Vorbedingungen: Zusicherungen für die Argumente spezifizieren, welche Bedingungen die Argumente erfüllen müssen.

Nachbedingungen: Zusicherungen an das Resultat spezifizieren den Wert des Resultats.

- Die **Bezugstransparenz** ermöglicht es, die Zusicherungen an das Resultat durch einen Korrektheitsbeweis *allein aus den Zusicherungen* an die Argumente abzuleiten.

Überprüfen der Argumente

- Wenn Sie beweisen können, daß eine Funktion nur mit korrekten Argumenten aufgerufen werden kann (Vertragsmodell), ist es überflüssig, die Argumente zu überprüfen.
- Wenn die Argumente auf unsichereren Wegen übergeben werden (Benutzereingabe, Einlesen von Files usw.), ist es sinnvoll, die Argumente bei der Übergabe automatisch daraufhin zu überprüfen, ob sie die Spezifikation erfüllen.

Common Lisp Mit der Funktion *assert* können die Zusicherungen funktional spezifiziert und automatisch überprüft werden.

Scheme (R5RS): bisher keine assert-Funktion (selbst implementieren).

DrRacket: Verträge können mittels *contract* Objekten annotiert werden.

Das Vertragsmodell in DrRacket

Objekte der Klasse *contract* berechnen Prädikate über Werte:

- Sie werden mit speziellen Konstruktoren erzeugt
- und mit special form expressions an zu überwachende Ausdrücke und Variable gebunden.
- Verträge können an
 - Variable (flat contract),
 - die Argumente und Resultate von Funktionen,
 - die importierten und exportierten Objekte eines Moduls
 - und die Felder von Verbunden und Objekten gebunden werden.
- Die Verträge werden automatisch überprüft, wenn auf den überwachten Wert zugegriffen wird.

flat contracts für Variable

Willkommen bei DrRacket, Version 5.0.1 [3m].

Sprache: racket; memory limit: 256 MB.

```
> (define/contract
  *x* ; definiere Variable *x*
  (between/c 0.0 3) ; Vertrag: 0 <= *x* <= 3
  (* (random) 5)) ; Initialisierung

... se3-bib/Beispiele/contractExamples.rkt:3.4:
(definition *x*) broke the contract
  (between/c 0.0 3) on *x*;
expected <(between/c 0.0 3)>,
given: 4.028033627623459
```

Weitere Verträge

```
(define/contract *z*
  (flat-contract odd?) 8)

(define/contract *zz*
  (flat-named-contract "ungerade" odd?) 8)

(define contListeUngerade
  (listof ; Liste mit ungeraden Zahlen
  (flat-named-contract "ungerade" odd?)))
```

```
(define/contract  
  *xs* contListeUngerade '(1 3 5 8 19))
```

Semiprädikate als Vertrag

Jedes einstellige Semiprädikat kann als Vertrag verwendet werden.

Beispiel: 55 (Ein Vertrag für ungerade Werte:)

```
(define/contract *z* odd? ; z ist ungerade  
  8)  
... /contractExamples.rkt:6.17:  
(definition *z*) broke the contract  
          odd? on *z*;  
expected <odd?>, given: 8
```

Weitere Konstruktoren für Verträge

```
(define/contract *zz*  
  (flat-contract odd?) 9)  
  
(define/contract *zzz*  
  (flat-named-contract "ungerade" odd?) 11)  
  
(define contListeUngerade  
  (listof ; Liste mit ungeraden Zahlen  
    (flat-named-contract "ungerade" odd?)))  
  
(define/contract  
  *xs* contListeUngerade '(1 3 5 8 19))  
→ (definition *xs*) broke the contract  
  (listof "ungerade") on *xs*;  
expected <ungerade>, given: 8
```

Vorteile von Verträgen

Verträge vs. Typdeklarationen

Mit Verträgen können beliebige Bedingungen geprüft werden, nicht nur die korrekte Typ-Signatur einer Funktion.

Verträge vs. bedingte Ausdrücke

Ein Vertrag wird nur einmal spezifiziert, wird aber bei jedem Zugriff auf das überwachte Objekt automatisch überprüft.

Kontrakte für die Signatur von Funktionen

Funktionen,

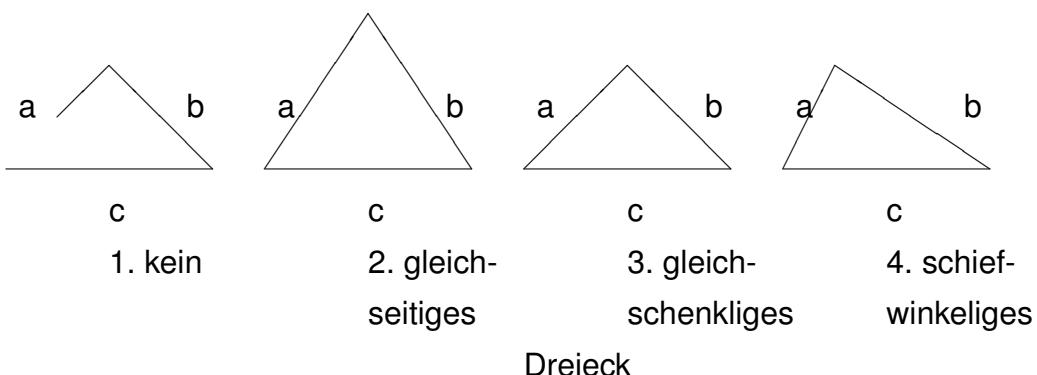
- die nur ein Resultat zurückgeben,
- die eine feste Zahl von Argumenten haben,
- und die nicht-rekursiv sind,

können mit dem \rightarrow -Operator spezifiziert werden:

```
(define/contract
  nusy      ; Funktionsname
  ( $\rightarrow$  number? symbol?) ; Kontrakt
  (lambda (n) 'eins)) ; die Funktion
  ; oder in Infix-Notation
(define/contract
  nuboint
  (integer? boolean? .  $\rightarrow$  . integer?)
  (lambda (n w) 1))
```

Der \rightarrow Konstruktor definiert einen Kontrakt für die Signatur einer Funktion, indem für jedes Argument und für das Resultat ein Kontrakt angegeben wird. nusy wird so als eine Funktion definiert, die Zahlen auf Symbole abbildet, nuboint nimmt zwei Argumente vom Typ integer und boolean und bildet diese auf integer-Zahlen ab. Die Argumente von \rightarrow können nicht nur Prädikate sein, sondern beliebige Kontrakte.

Beispiel: Dreiecke



mit den Seitenlängen $a \leq b \leq c$

Abhängigkeiten zwischen Argumenten

```
; ;; a <= b <= c
(define/contract analyse
  (->d ([a number?]
          [b (and/c number? (>=/c a) (=/c b))])
        [c number?])
    ())
  [result symbol?])
(lambda (a b c)
  (cond [(< (+ a b) c) 'kein-Dreieck]
        [(= a c) 'gleichseitig]
        [(= a b) 'gleichschenklig]
        [(= b c) 'gleichschenklig]
        [else 'schiefwinklig])))
```

Kontrakt für Aufzählungstypen

```
(define/contract analyse2
  (->d ((a number?))
        (b (and/c number? (>=/c a) (=/c b)))
        (c number?))
    ())
  (list (lambda (r)
          (member r
                  '(kein-Dreieck
                    gleichseitig
                    gleichschenklig
                    schiefwinklig))))
    (lambda (a b c)
      (cond [(< (+ a b) c) 'kein-Dreieck]
            [(= a c) 'gleichseitig]
            [(= a b) 'gleichschenklig]
            [(= b c) 'gleichschenklig]
            [else 'schiefwinklig]))))
```

contract.rkt: Contracts

To load: (require racket/contract)

Rackets contract.rkt library defines new forms of expression that specify contracts and new forms of expression that attach contracts to values.

This section describes three classes of contracts: contracts for flat values (described in section 12.1), contracts for functions (described in section 12.2), and contracts for objects and classes (described in section 12.4).

In addition, this section describes how to establish a contract, that is, how to indicate that a particular contract should be enforced at a particular point in the program (in section 12.5).

Flat Contracts A contract for a flat value can be a predicate that accepts the value and returns a boolean indicating if the contract holds.

(flat-contract predicate) ;*FLAT-CONTRACT*

Constructs a contract from predicate.

(flat-named-contract type-name predicate)
;*FLAT-CONTRACT*

For better error reporting, a flat contract can be constructed with flat-named-contract, a procedure that accepts two arguments. The first argument must be a string that describes the type that the predicate checks for. The second argument is the predicate itself.

any/c ;*FLAT-CONTRACT*

any/c is a flat contract that accepts any value.

If you are using this predicate as the result portion of a function contract, consider using any instead. It behaves the same, but in that one restrictive context has better memory performance.

none/c ;*FLAT-CONTRACT*

none/c is a flat contract that accepts no values.

(or/c contract ...) ;*OR/C*

or/c accepts any number of predicates and higher-order contracts and returns a contract that accepts any value that any one of the contracts accepts, individually.

If all of the arguments are predicates or flat contracts, it returns a flat contract. If only one of the arguments is a higher-order contract, it returns a contract that just checks the flat contracts and, if they don't pass, applies the higher-order contract.

If there are multiple higher-order contracts, or/c uses contract-first-order-passes? to distinguish between them. More precisely, when an or/c is checked, it first checks all of the flat contracts. If none of them pass, it calls contract-first-order-passes? with each of the higher-order contracts. If only one returns true, or/c uses that contract. If none of them return true, it signals a contract violation. If more than one returns true, it signals an error indicating that the or/c contract is malformed.

or/c tests any values by applying the contracts in order, from left to right, with the exception that it always moves the non-flat contracts (if any) to the end, checking them last.

(and/c contract ...) CONTRACT

and/c accepts any number of contracts and returns a contract that checks that accepts any value that satisfies all of the contracts, simultaneously.

If all of the arguments are predicates or flat contracts, and/c produces a flat contract.

and/c tests any values by applying the contracts in order, from left to right.

(not/c flat-contract) FLAT-CONTRACT

not/c accepts a flat contracts or a predicate and returns a flat contract that checks the inverse of the argument.

(=/c number) FLAT-CONTRACT

=/c accepts a number and returns a flat contract that requires the input to be a number and equal to the original input.

(>/c number) FLAT-CONTRACT

>/c accepts a number and returns a flat contract that requires the input to be a number and greater than or equal to the original input.

(</c number) FLAT-CONTRACT

</c accepts a number and returns a flat contract that requires the input to be a number and less than or equal to the original input.

(between/c number number) FLAT-CONTRACT

between/c accepts two numbers and returns a flat contract that requires the input to be between the two numbers (or equal to one of them).

(>/c number) FLAT-CONTRACT

>/c accepts a number and returns a flat contract that requires the input to be a number and greater than the original input.

(</c number) FLAT-CONTRACT

</c accepts a number and returns a flat contract that requires the input to be a number and less than the original input.

(integer-in number number) FLAT-CONTRACT

integer-in accepts two numbers and returns a flat contract that recognizes if integers between the two inputs, or equal to one of its inputs.

(real-in number number) FLAT-CONTRACT

real-in accepts two numbers and returns a flat contract that recognizes real numbers between the two inputs, or equal to one of its inputs.

natural-number/c FLAT-CONTRACT

natural-number/c is a contract that recognizes natural numbers (i.e., an integer that is either positive or zero).

(string/len number) FLAT-CONTRACT

string/len accepts a number and returns a flat contract that recognizes strings that have fewer than that number of characters.

false/c FLAT-CONTRACT

false/c is a flat contract that recognizes #f.

printable/c FLAT-CONTRACT

printable/c is a flat contract that recognizes values that can be written out and read back in with write and read.

(one-of/c value ...) FLAT-CONTRACT

one-of/c accepts any number of atomic values and returns a flat contract that recognizes those values, using eqv? as the comparison predicate. For the purposes of one-of/c, atomic values are defined to be: characters, symbols, booleans, null keywords, numbers, void, and undefined.

(symbols symbol ...) FLAT-CONTRACT

symbols accepts any number of symbols and returns a flat contract that recognizes those symbols.

(is-a?/c class-or-interface) FLAT-CONTRACT

is-a?/c accepts a class or interface and returns a flat contract that recognizes if objects are subclasses of the class or implement the interface.

(implementation?/c interface) FLAT-CONTRACT

implementation?/c accepts an interface and returns a flat contract that recognizes if classes are implement the given interface.

(subclass?/c class) FLAT-CONTRACT

subclass?/c accepts a class and returns a flat-contract that recognizes classes that are subclasses of the original class.

(listof flat-contract) FLAT-CONTRACT

listof accepts a flat contract (or a predicate which is converted to a flat contract) and returns a flat contract that checks for lists whose elements match the original flat contract.

(list-immutableof contract) CONTRACT

list-immutableof accepts a contract (or a predicate which is converted to a flat contract) and returns a contract that checks for immutable lists whose elements match the original contract. In contrast to listof, list-immutableof accepts arbitrary contracts, not just flat contracts.

Beware, however, that when a value is applied to this contract, the result will not be eq? to the input.

(vectorof flat-contract) FLAT-CONTRACT

vectorof accepts a flat contract (or a predicate which is converted to a flat contract via flat-contract) and returns a predicate that checks for vectors whose elements match the original flat contract.

(vector-immutableof contract) CONTRACT

vector-immutableof accepts a contract (or a predicate which is converted to a flat contract) and returns a contract that checks for immutable lists whose elements match the original contract. In contrast to vectorof, vector-immutableof accepts arbitrary contracts, not just flat contracts.

Beware, however, that when a value is applied to this contract, the result will not be eq? to the input.

(vector/c flat-contract ...) FLAT-CONTRACT

vector/c accepts any number of flat contracts (or predicates which are converted to flat contracts via flat-contract) and returns a flat-contract that recognizes vectors. The number of elements in the vector must match the number of arguments supplied to vector/c and the elements of the vector must match the corresponding flat contracts.

(vector-immutable/c contract ...) CONTRACT

`vector-immutable/c` accepts any number of contracts (or predicates which are converted to flat contracts via `flat-contract`) and returns a contract that recognizes vectors. The number of elements in the vector must match the number of arguments supplied to `vector-immutable/c` and the elements of the vector must match the corresponding contracts.

In contrast to `vector/c`, `vector-immutable/c` accepts arbitrary contracts, not just flat contracts. Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

(`box/c flat-contract`) FLAT-CONTRACT

`box/c` accepts a flat contract (or predicate that is converted to a flat contract via `flat-contract`) and returns a flat contract that recognizes for boxes whose contents match `box/c`'s argument.

(`box-immutable/c contract`) CONTRACT

`box-immutable/c` one contracts (or a predicate that is converted to a flat contract via `flat-contract`) and returns a contract that recognizes boxes. The contents of the box must match the contract passed to `box-immutable/c`.

In contrast to `box/c`, `box-immutable/c` accepts an arbitrary contract, not just a flat contract. Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

(`cons/c flat-contract flat-contract`) FLAT-CONTRACT

`cons/c` accepts two flat contracts (or predicates that are converted to flat contracts via `flat-contract`) and returns a flat contract that recognizes cons cells whose car and cdr correspond to `cons/c`'s two arguments.

(`cons-immutable/c contract contract`) CONTRACT

`cons-immutable/c` accepts two contracts (or predicates that are converted to flat contracts via `flat-contract`) and returns a contract that recognizes immutable cons cells whose car and cdr correspond to `cons-immutable/c`'s two arguments. In contrast to `cons/c`, `cons-immutable/c` accepts arbitrary contracts, not just flat contracts.

Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

(`list/c flat-contract ...`) FLAT-CONTRACT

`list/c` accepts an arbitrary number of flat contracts (or predicates that are converted to flat contracts via `flat-contract`) and returns a flat contract that recognizes for lists whose length is the same as the number of arguments to `list/c` and whose elements match those arguments.

(`list-immutable/c contract ...`) CONTRACT

`list-immutable/c` accepts an arbitrary number of contracts (or predicates that are converted to flat contracts via `flat-contract`) and returns a contract that recognizes for lists whose length is the same as the number of arguments to `list-immutable/c` and whose elements match those contracts.

In contrast to `list/c`, `list-immutable/c` accepts arbitrary contracts, not just flat contracts. Beware, however, that when a value is applied to this contract, the result will not be `eq?` to the input.

`(syntax/c flat-contract)` FLAT-CONTRACT

`syntax/c` accepts a flat contract and produces a flat contract that recognizes syntax objects whose contents match the argument to `syntax/c`.

`(struct/c struct-name flat-contract ...)` FLAT-CONTRACT

`struct/c` accepts a struct name and as many flat contracts as there are fields in the named struct. It returns a contract that accepts instances of that struct whose fields match the given contracts.

`(flat-rec-contract name flat-contract ...)` SYNTAX

Each `flat-rec-contract` form constructs a flat recursive contract. The first argument is the name of the contract and the following arguments are flat contract expressions that may refer to name.

As an example, this contract:

```
(flat-rec-contract sexp
  (cons/c sexp sexp)
  number?
  symbol?)
```

is a flat contract that checks for (a limited form of) s-expressions. It says that an `sexp` is either two `sexp` combined with `cons`, or a number, or a symbol.

Note that if the contract is applied to a circular value, contract checking will not terminate.

`(flat-murec-contract ([name flat-contract ...] ...)`
`body ...)` SYNTAX

The `flat-murec-contract` form is a generalization of `flat-rec-contracts` for defining several mutually recursive flat contracts simultaneously.

Each of the names is visible in the entire `flat-murec-contract` and the result of the final body expression is the result of the entire form.

Note that if the contract is applied to a circular value, contract checking will not terminate.

Function Contracts This section describes the contract constructors for function contracts. This is their shape:

```

contract-expr ::==
| (case-> arrow-contract-expr ...)
| arrow-contract-expr

arrow-contract-expr ::==
| (-> expr ... expr)
| (-> expr ... any)
| (-> expr ... (values expr ...))

| (->* (expr ...) (expr ...))
| (->* (expr ...) any)
| (->* (expr ...) expr (expr ...))
| (->* (expr ...) expr any)

| (->d expr ... expr)
| (->d* (expr ...) expr)
| (->d* (expr ...) expr expr)

| (->r ((id expr) ...) expr)
| (->r ((id expr) ...) any)
| (->r ((id expr) ...) (values (id expr) ...))
| (->r ((id expr) ...) id expr expr)
| (->r ((id expr) ...) id expr any)
| (->r ((id expr) ...) id expr (values (id expr) ...))

| (->pp ((id expr) ...) pre-expr expr res-id post-expr)
| (->pp ((id expr) ...) pre-expr any)
| (->pp ((id expr) ...) pre-expr (values (id expr) ...))
| (->pp-rest ((id expr) ...) id expr pre-expr expr
| (->pp-rest ((id expr) ...) id expr pre-expr any)
| (->pp-rest ((id expr) ...) id expr pre-expr
| (->pp-rest ((id expr) ...) values (id expr) ...)) post-expr)

```

```

| (opt-> (expr ...) (expr ...) expr)
| (opt->* (expr ...) (expr ...) any)
| (opt->* (expr ...) (expr ...) (expr ...))

```

where `expr` is any expression.

`(-> expr ...)` SYNTAX

`(-> expr ... any)` SYNTAX

The `->` contract is for functions that accept a fixed number of arguments and return a single result. The last argument to `->` is the contract on the result of the function and the other arguments are the contracts on the arguments to the function. Each of the arguments to `->` must be another contract expression or a predicate. For example, this expression:

`-(integer? boolean? . -> . integer?)`

is a contract on functions of two arguments. The first must be an integer and the second a boolean and the function must return an integer. (This example uses MzScheme's infix notation so that the `->` appears in a suggestive place; see section 11.2.4 in PLT MzScheme: Language Manual).

If `any` is used as the last argument to `->`, no contract checking is performed on the result of the function, and tail-recursion is preserved. Except for the memory performance, this is the same as using `any/c` in the result.

The final case of `->` expressions treats values as a local keyword - that is, you may not return multiple values to this position, instead if the word `values` syntactically appears in the in the last argument to `->` the function is treated as a multiple value return.

`(->* (expr ...) (expr ...))` SYNTAX

`(->* (expr ...) any)` SYNTAX

`(->* (expr ...) expr (expr ...))` SYNTAX

`(->* (expr ...) expr any)` SYNTAX

The `->*` expression is for functions that return multiple results and/or have rest arguments. If two arguments are supplied, the first is the contracts on the arguments to the function and the second is the contract on the results of the function. These situations are also covered by `->`.

If three arguments are supplied, the first argument contains the contracts on the arguments to the function (excluding the rest argument), the second contains the contract on the rest argument to the function and the final argument is the contracts on the results of the function. The final argument can be any which, like `->` means that no contract is enforced on the result of the function and tail-recursion is preserved.

`(->d expr ...)` SYNTAX

`(->d* (expr ...) expr))` SYNTAX

`(->d* (expr ...) expr expr)` SYNTAX

The `->` and `->d*` contract constructors are like their d-less counterparts, except that the result portion is a function that accepts the original arguments to the function and returns the range contracts. The range contract function for `->d*` must return multiple values: one for each result of the original function. As an example, this is the contract for `sqrt`:

```
(number?
 . ->d .
 (lambda (in)
 (lambda (out)
 (and (number? out)
 (< (abs (- (* out out) in)) 0.01))))
```

It says that the input must be a number and that the difference between the square of the result and the original number is less than 0.01.

`(->r ([id expr] ...) expr)` SYNTAX

The `->r` contract allows you to build a contract where the arguments to a function may all depend on each other and the result of the function may depend on all of the arguments.

Each of the ids names one of the actual arguments to the function with the contract. Each of the names is available to all of the other contracts. For example, to define a function that accepts three arguments where the second argument and the result must both be between the first, you might write:

`(->r ([x number?] [y (and/c (>/c x) (</c z))] [z number?])
 (and/c number? (>/c x) (</c z)))`

`(->r ([id expr] ...) any)` SYNTAX

This variation on $\rightarrow r$ does not check anything about the result of the function, which preserves tail recursion.

$(\rightarrow r ([id\ expr]\ \dots)\ (\mathbf{values}\ [id\ expr]\ \dots))$ SYNTAX

This variation on $\rightarrow r$ allows multiple value return values. The ids for the domain are bound in all of the exprs, but the ids for the range (the ones inside values) are only bound in the exprs inside the values.

As an example, this contract:

$(\rightarrow r ()\ (\mathbf{values}\ [x\ number?]\ [y\ (and/c\ (>=/c\ x)\ (<=/c\ z))]\ [z\ number?]))$

matches functions that accept no arguments and that return three numeric values that are in ascending order.

$(\rightarrow r ([id\ expr]\ \dots)\ id\ expr\ expr)$ SYNTAX

$(\rightarrow r ([id\ expr]\ \dots)\ id\ expr\ any)$ SYNTAX

$(\rightarrow r ([id\ expr]\ \dots)\ id\ expr\ (\mathbf{values}\ [id\ expr]\ \dots))$ SYNTAX

These three forms of the $\rightarrow r$ contract are just like the previous ones, except that the functions they matches must accept arbitrarily many arguments. The extra id and the expr just following it specify the contracts on the extra arguments. The value of id will always be a list (of the extra arguments).

$(\rightarrow pp\ ([id\ expr]\ \dots)\ pre-expr\ expr\ res-id\ post-expr)$ SYNTAX

$(\rightarrow pp\ ([id\ expr]\ \dots)\ pre-expr\ any)$ SYNTAX

$(\rightarrow pp\ ([id\ expr]\ \dots)\ pre-expr\ (\mathbf{values}\ [id\ expr]\ \dots)\ post-expr)$ SYNTAX

$(\rightarrow pp-rest\ ([id\ expr]\ \dots)\ id\ expr\ pre-expr\ expr\ res-id\ post-expr)$ SYNTAX

$(\rightarrow pp-rest\ ([id\ expr]\ \dots)\ id\ expr\ pre-expr\ any)$ SYNTAX

```
(->pp-rest ([ id expr] ...) id expr pre-expr
  (values [id expr] ...) post-expr)      SYNTAX
```

These six shapes of $\rightarrow\text{pp}$ match up to the six shapes of $\rightarrow\text{r}$ forms explained above, with the addition that the extra pre- and post-condition expressions must not evaluate to #f.

If the pre-condition evaluates to #f, the caller is blamed and if the post-condition expression evaluates to #f the function itself is blamed.

The argument variables are bound in the pre-expr and the post-expr and the variables in the values result clauses are bound in the post-expr.

Additionally, the variable res-id is bound to the result in the first $\rightarrow\text{pp}$ case and in the first $\rightarrow\text{pp-rest}$ case.

```
(case-> arrow-contract-expr ...)    CONTRACT-CASE->
```

The case-> expression constructs a contract for case-lambda function. It's arguments must all be function contracts, built by one of

\rightarrow , $\rightarrow\text{d}$, $\rightarrow*$, or $\rightarrow\text{d}*$.

```
(opt-> (req-contracts ...) (opt-contracts ...)
  res-contract))      SYNTAX
```

```
(opt->* (req-contracts ...) (opt-contracts ...)
  (res-contracts ...))      SYNTAX
```

```
(opt->* (req-contracts ...) (opt-contracts ...) any)
  SYNTAX
```

The opt-> expression constructs a contract for an opt-lambda function. The first arguments are the required parameters, the second arguments are the optional parameters and the final argument is the result. The req-contracts expressions, the opt-contracts expressions, and the res-contract expressions can be any expression that evaluates to a contract value.

Each opt-> expression expands into case->.

The opt->* expression constructs a contract for an opt-lambda function. The only difference between opt-> and opt->* is that multiple return values are permitted with opt->* and they are specified in the last clause of an opt->* expression. A result of any means any value or any number of values may be returned, and the contract does not inhibit tail-recursion.

Lazy Data-structure Contracts Typically, contracts on data structures can be written using flat contracts. For example, one might write a sorted list contract as a function that accepts a list and traverses it, ensuring that the elements are in order. Such contracts, however, can change the asymptotic running time of the program, since the contract may end up exploring more of a function's input than the function itself does. To circumvent this problem, the define-contract-struct form introduces contract combinators that are lazy that is, they only verify the contract holds for the portion of some data structure that is actually inspected. More precisely, a lazy data structure contract on a struct is not checked until a selector extracts a field of a struct.

The form

```
(define-contract-struct struct-name (field ...))
```

is like the corresponding define-struct, with two differences: it does not define field mutators and it does define two contract constructors: struct-name/c and struct-name/dc. The first is a procedure that accepts as many arguments as there are fields and returns a contract for struct values whose fields match the arguments. The second is a syntactic form that also produces contracts on the structs, but the contracts on later fields may depend on the values of earlier fields. Its syntax is:

```
(struct-name/dc field-spec ...)
```

where each field-spec is one of the following two lines:

```
[field contract-expr]
[field (field ...) contract-expr]
```

In each case, the first field name specifies which field the contract applies to, and the fields must be specified in the same order as the original define-contract-struct. The first case is for when the contract on the field does not depend on the value of any other field. The second case is for when the contract on the field does depend on some other fields, and the field names in middle second indicate which fields it depends on. These dependencies can only be to fields that come earlier in the struct.

As an example consider this module:

```

(module product mzscheme
  (require (lib "contract.ss"))

(define-contract-struct kons (hd tl))

;; sorted-list/gt : number -> contract
;; produces a contract that accepts
;; sorted kons-lists whose elements
;; are all greater than 'num'.
(define (sorted-list/gt num)
  (or/c null?
        (kons/dc [hd (>=/c num)]
                 [tl (hd) (sorted-list/gt hd)])))

;; product : kons-list -> number
;; computes the product of the values
;; in the list. if the list contains
;; zero, it avoids traversing the rest
;; of the list.
(define (product l)
  (cond
    [(null? l) 1]
    [else
      (if (zero? (kons-hd l))
          0
          (* (kons-hd l)
             (product (kons-tl l))))])))

(provide kons? make-kons kons-hd kons-tl)
(provide/contract
  [product (-> (sorted-list/gt -inf.0) number?)]))

```

It provides a single function, `product` whose contract indicates that it accepts sorted lists of numbers and produces numbers. Using an ordinary flat contract for sorted lists, the `product` function cannot avoid traversing having its entire argument be traversed, since the contract checker will traverse it before the function is called. As written above, however, when the `product` function aborts the traversal of the list, the contract checking also stops, since the `kons/dc` contract constructor generates a lazy contract.

Attaching Contracts to Values There are three special forms that attach contract specification to values: `provide/contract`, `define/contract`, and `contract`.

(`provide/contract p/c-item ...`) SYNTAX

`p/c-item` is one of
 (`struct identifier ((identifier contract-expr) ...)`)
 (`struct (identifier identifier)`
 `((identifier contract-expr) ...)`)
 (`rename id id contract-expr`)
 (`id contract-expr`)

A `provide/contract` form can only appear at the top-level of a module (see section 5 in PLT MzScheme: Language Manual). As with `provide`, each identifier is provided from the module. In addition, clients of the module must live up to the contract specified by `contract-expr`.

The `provide/contract` form treats modules as units of blame. The module that defines the provided variable is expected to meet the positive (covariant) positions of the contract. Each module that imports the provided variable must obey the negative (contra-variant) positions of the contract.

Only uses of the contracted variable outside the module are checked. Inside the module, no contract checking occurs.

The `rename` form of a `provide/contract` exports the first variable (the internal name) with the name specified by the second variable (the external name).

The `struct` form of a `provide/contract` clause provides a structure definition. Each field has a contract that dictates the contents of the fields.

If the `struct` has a parent, the second `struct` form (above) must be used, with the first name referring to the `struct` itself and the second name referring to the parent `struct`. Unlike `define-struct`, however, all of the fields (and their contracts) must be listed. The contract on the fields that the sub-`struct` shares with its parent are only used in the contract for the sub-`struct`'s maker, and the selector or mutators for the super-`struct` are not provided.

Note that the struct definition must come before the provide clause in the module's body.

```
(define/contract id contract-expr init-value-expr)  
SYNTAX
```

The define/contract form attaches the contract contract-expr to init-value-expr and binds that to id.

The define/contract form treats individual definitions as units of blame. The definition itself is responsible for positive (co-variant) positions of the contract and each reference to id (including those in the initial value expression) must meet the negative positions of the contract.

Error messages with define/contract are not as clear as those provided by provide/contract because define/contract cannot detect the name of the definition where the reference to the defined variable occurs. Instead, it uses the source location of the reference to the variable as the name of that definition.

```
(contract contract-expr to-protect-expr  
positive-blame negative-blame) SYNTAX  
  
(contract contract-expr to-protect-expr  
positive-blame negative-blame contract-source) SYNTAX
```

The contract special form is the primitive mechanism for attaching a contract to a value. Its purpose is as a target for the expansion of some higher-level contract specifying form.

The contract form has this shape:

```
(contract expr to-protect-expr  
positive-blame negative-blame contract-source)
```

The contract expression adds the contract specified by the first argument to the value in the second argument. The result of a contract expression is the result of the to-protect-expr expression, but with the contract specified by contract-expr enforced on to-protect-expr. The expressions positive-blame and negative-blame must be symbols indicating how to assign blame for positive and negative positions of the contract specified by contract-expr. Finally, contract-source, if specified, indicates where the contract was assumed. It must be a syntax object specifying the source location of the location where the contract was assumed. If the syntax object wraps a symbol, the symbol is used as the name of the primitive whose contract was assumed. If absent, it defaults to the source location of the contract expression.

contract? PREDICATE

The procedure contract? returns #t if its argument is a contract (ie, constructed with one of the combinators described in this section).

flat-contract? PREDICATE

This predicate returns true when its argument is a contract that has been constructed with flat-contract (and thus is essentially just a predicate).

(flat-contract-predicate value) SELECTOR

This function extracts the predicate from a flat contract.

(contract-first-order-passes? contract value) PROCEDURE

Returns a boolean indicating if the first-order tests of contract pass for value.

If it returns #f, the contract is guaranteed not to hold for that value; if it returns #t, the contract may or may not hold. If the contract is a first-order contract, a result of #t guarantees that the contract holds.

(make-none/c sexp-name) PROCEDURE

Makes a contract that accepts no values, and reports the name sexp-name when signaling a contract violation.

(contract-violation->string [violation-renderer]) PROCEDURE

This is a parameter that is used when constructing a contract violation error. Its value is procedure that accepts six arguments: the value that the contract applies to, a syntax object representing the source location where the contract was established, the names of the two parties to the contract (as symbols) where the first one is the guilty one, an s-expression representing the contract, and a message indicating the kind of violation. The procedure then returns a string that is put into the contract error message. Note that the value is often already included in the message that indicates the violation.

(recursive-contract contract) SYNTAX

Unfortunately, the standard contract combinators (like `->`, etc) evaluate their arguments eagerly, leading to either references to undefined variables or infinite loops, while building recursive contracts.

The recursive-contract form delays the evaluation of its argument until the contract is checked, making recursive contracts possible.

Testfälle

- Wenn Sie eine Funktion spezifiziert haben, sollten Sie Testfälle definieren, die das spezifizierte Verhalten überprüfen.
- Überprüfen Sie insbesondere die Grenzwerte der Definitionsbereiche.
- In DrRacket können Sie die Testfälle direkt als Annotationen in den Quelltext einfügen.
- Die Testfälle können dann automatisch bei jeder Änderung des Programms überprüft werden.
- Die Testfälle werden in den Lehresprachen über das Scheme-Menue (Tests aktivieren, Tests deaktivieren) ein- und ausgeschaltet.
- In der Sprache Racket werden die Tests durch den Aufruf von `(test)` ausgeführt.

Special Form Operators für Test Cases

Prüfe auf Gleichheit: check-expect

(**check-expect** *expr1* *expr2*)

Prüfe, ob der Ausdruck *expr1* zu *expr2* evaluiert.

Im Definitionsfenster:

```
(require test-engine/racket-tests)
(define a 1)
(define b 2)
(check-expect b (* 2 a))
(test)
```

Im Interaktionsfenster:

```
Willkommen bei DrRacket, Version 5.0.1 [3m].
Sprache: racket; memory limit: 256 MB.
The only test passed!
```

>

Ein fehlgeschlagener Testfall

Beispiel: 56 (Testfall: nicht erfolgreich)

Im Definitionsfenster:

```
(define a 1)
(check-expect (* 3 a) 4)
```

Im Test-Results-Fenster:

```
Willkommen bei DrRacket, Version 5.0.1 [3m].
Sprache: racket; memory limit: 256 MB.
Ran 1 check.
0 checks passed.
Actual value 3 differs from 4, the expected value.
At line 4 column 0
```

Prüfe die Genauigkeit: check-within

(**check-within** *expr1* *expr2* *expr3*)

Prüfe, ob der numerische Ausdruck *expr1* zu *expr2* evaluiert.

expr3 ist ein Toleranzwert für die Genauigkeit.

Im Definitionsfenster:

```
(check-expect (sin pi) 0.0)
(check-within (sin pi) 0.0 0.1e-14)
(test)
```

Im Interaktionsfenster:

```
check-expect cannot compare inexact numbers.
Try (check-within test 0.0 range).
> (check-within (sin pi) 0.0 0.1e-14)
(test)
The only test passed!
```

Special Form Operators für Test Cases

Prüfe die Fehlermeldung: check-error

```
(check-error expr expr)
```

Prüfe, ob der Ausdruck *expr1* die erwartete Fehlermeldung *expr2* signalisiert.

expr2 muß zu einer Zeichenkette evaluieren.

Im Definitionsfenster:

```
(check-error (/ 1 0)
            "/:_division_by_zero")
```

Beispiel: 57 (Testfälle für my-length)

```
(require test-engine/racket-tests)
(define (my-length xs)
  (cond [(not (list? xs))
         (error 'xs "keine_Liste")]
        [(null? xs) 0]
        [else
         (+ 1 (my-length (cdr xs))))])

(check-expect (my-length '()) 0)
(check-expect (my-length '(1 2 3)) 3)
(check-error (my-length "12")
             "xs:_keine_Liste")
→ All 3 tests passed!
```

15 Rekursion und Induktion

Rekursion



- 11 Semantik
- 12 Korrektheit und Spezifikation
- 13 Rekursion und Induktion
 - Arten von Rekursion
 - Korrektheit rekursiver Funktionen
 - Rekursive Prozesse und Endrekursion

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 356 / 1157

Literaturhinweise zu Rekursion und Funktionen höherer Ordnung:
Für dieses Kapitel über **Rekursion** haben wir auf eine ganze Reihe von Lehrbüchern zurückgreifen müssen: es gibt zwar viele gute Darstellungen, aber nicht alle beziehen sich dabei auf die Programmierung in Racket.

Der Baukastenansatz: Diesen Abschnitt können Sie am besten bei [Hölyer, 1991](#) nachlesen.

Wir haben für Sie die Miranda-Funktionen `until` und `iterate` nach Racket portiert.

Funktionen höherer Ordnung werden sehr gut von [Abelson et al., 1998](#) eingeführt. Die hier vorgestellten Beispiele `sqrt`, `newton` sind von [Bird and Wadler, 1988](#) übernommen und von uns in Racket reimplementiert worden.

Die Dylan-Funktionen: Die Implementationen der Funktionen `curry`, `compose`, `disjoin` usw. sind aus den Common Lisp Versionen von [Graham, 1996](#) abgeleitet.

Rekursion und Induktion: Für dieses Kapitel haben wir die Darstellung aus dem A1-Skript von [Züllighoven, 1995](#) übernommen.

Rekursive Algorithmen finden Sie in vielen Lehrbüchern: Kombinatorische Funktionen (Bird and Wadler, 1988), backtracking (Springer and Friedman, 1989; Bird and Wadler, 1988), Musterabgleich, Eliza (Norvig, 1992).

Effizienz: Die Betrachtungen zur Effizienz (Datenmüll, Memo-Funktionen) stehen bei Norvig, 1992.

Verzögerte Auswertung: Die konzeptuellen Grundlagen zur verzögerten Auswertung und zum stromorientierten Programmierstil finden Sie bei Bird and Wadler, 1988 und Abelson et al., 1998. Bei Abelson et al., 1998 finden Sie auch die Implementation von Strömen in Racket.

Rekursion in anderen Domänen: Eine vergnügliche, aber dennoch sehr fundierte Betrachtung von rekursiven Strukturen in der Logik, in den rekursiven Bildern von Escher und in den Fugen von Bach finden Sie bei Hofstadter, 1980. Bei diesem Buch ist auch die Übersetzung ins Deutsche sehr gelungen.

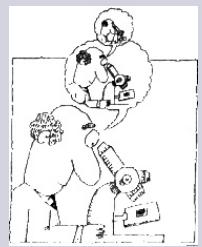
Programmbibliothek: Eine Bibliothek aller Funktionen, die keine Racket-Standardfunktionen sind, finden Sie in der se3-bib im Modul tools-module.rkt. Die Beispielprogramme finden Sie ebenfalls in der se3-bib im Unterverzeichnis Integrationstest

15.1 Arten von Rekursion

Wann verwenden wir rekursive Funktionen?

- **Rekursive** Funktionen werden **verwendet**, wenn ein schwieriges Problem durch eine Folge von Vereinfachungen auf ähnliche, aber leichtere Probleme zurückgeführt werden kann, bis ein trivialer Fall erreicht wird.
- **Rekursion** wird verwendet, wenn unendliche Prozesse oder Datenstrukturen in endlicher Form beschrieben werden sollen.
- Rekursive Funktionen werden zur Verarbeitung rekursiver Datenstrukturen verwendet.
- Wir unterscheiden zwischen rekursiven *Definitionen* und rekursiven *Prozessen*.

Rekursive Definition



Definition (Rekursive Definition)

Eine Definition ist **rekursiv**, wenn sie auf den gerade zu definierenden Begriff Bezug nimmt.

- Das Wort „Rekursion“ ist aus dem lateinischen Wort „recurrere“ (zurücklaufen) abgeleitet, da bei der Auswertung einer solchen Definition wiederholt zum Anfang zurückgegangen werden muß.
 - Eine rekursivee Definition muß immer einen *trivialen Fall* enthalten, der nicht mehr auf die Definition Bezug nimmt.

Die Menge der natürlichen Zahlen

Definition: 58 (*Die Menge der natürlichen Zahlen*)

Die Menge der natürlichen Zahlen \mathcal{N} ist rekursiv definiert durch das Peano-sche Axiomensystem.

1. 1 ist eine natürliche Zahl.
 2. Jede Zahl $a \in \mathcal{N}$ hat einen bestimmten Nachfolger $a' \in \mathcal{N}$.
 3. Es gibt keine Zahl mit dem Nachfolger 1.
 4. Aus $a' = b'$ folgt $a = b$.
 5. Jede Menge M von natürlichen Zahlen, welche die Zahl 1 und zu jeder Zahl a auch den Nachfolger a' enthält, enthält alle natürlichen Zahlen ($M = \mathcal{N}$).

nach (Meschkowski, 1971)

☞ Anmerkung: Diese und die folgenden Definitionen sind von Züllighoven, 1995 übernommen.

Minimalrekursive Funktion

Definition: 59 (Minimal-rekursive Funktionsdefinition)

Eine Funktionsdefinition heißt **Rekursion: Minimal rekursiv**, wenn sie auf der rechten Seite der definierenden Gleichung

- nur aus einem elementaren Fall
- und einer rekursiven Verwendung der Funktion besteht.
- Der *elementare Fall* beschreibt die *Abbruchbedingung* und das Funktionsergebnis.

```
(define (add x y)
  (if (= y 0) x ; elementarer Fall
    (add (+ x 1) (- y 1)))) ; rek. Verw.
```

Rekursionsschema

- $x + y = x$, falls $y=0$
- $x + y = x + 1 + y - 1$

Ein Trace

Language: racket.

```
> (define (add x y)
  (if (= y 0) x
    (add (+ x 1) (- y 1))))
> (require racket/trace))
> (trace add)
(add)
> (add 3 4)
|(add 3 4)
|(add 4 3)
|(add 5 2)
|(add 6 1)
|(add 7 0)
|7
→ 7
```

☞ Anmerkung: Rekursion ohne „define“: Müssen Funktionen einen Namen haben, damit wir sie rekursiv aufrufen können? *Nein*.

Mithilfe des λ -Abstraktors können wir sogar rekursive Funktionen definieren, ohne daß wir ihnen bei der Definition mit `define` einen Namen geben müssen. Der Trick besteht darin, die rekursiv zu rufende Funktion als Argument an eine andere Funktion zu übergeben, so daß der Name des formalen Parameters für den rekursiven Aufruf verwendet werden kann.

Denksportaufgabe: Welche Funktion wird hier berechnet?

```
> ((lambda (n)
  ((lambda (fact-iter)
    (fact-iter fact-iter 1 1))
   (lambda (f-i product counter)
     (if (> counter n)
         product
         (f-i f-i
           (* counter product)
           (+ counter 1))))))
```

4) \longrightarrow 24

Lineare Rekursion

Definition: 60 (Lineare Rekursion)

Eine Funktionsdefinition, die sich auf der rechten Seite der definierenden Gleichung in jeder Fallunterscheidung selbst nur einmal verwendet, heißt *Rekursion: linear-rekursiv*.

```
; Abbilden einer Liste
(define (my-map f xs)
  (if (null? xs)
      '()
      (cons (f (car xs))
            (my-map f (cdr xs)))))
```

> (my-map sqrt '(1 4 9 16 25))
 \longrightarrow (1.0 2.0 3.0 4.0 5.0)

my-map: Ein Trace

```
; Abbilden einer Liste
(require racket/trace)
(trace my-map)
> (my-map sqrt '(1 4 9 16))
(my-map sqrt '(1 4 9 16))
|(my-map sqrt (1 4 9 16))
```

```

| (my-map sqrt (4 9 16))
| |(my-map sqrt (9 16))
| | |(my-map sqrt (16))
| | | |(my-map sqrt ())
| | | |()
| | | |(4)
| | | |(3 4)
| | | |(2 3 4)
|(1 2 3 4) → (1 2 3 4)

```

my-map: Ein Trace

```

> (trace my-map sqrt)
> (my-map sqrt '(1 4 9))
|(my-map sqrt (1 4 9))
| (sqrt 1)
| 1
| (my-map sqrt (4 9))
| |(sqrt 4)
| 2
| |(my-map sqrt (9))
| | |(sqrt 9)
| | 3
| | |(my-map sqrt ())
| | |()
| | |(3)
| | |(2 3)
|(1 2 3) → (1 2 3)

```



Typische Fälle

Lineare Rekursion

Rekursion über natürliche Zahlen: Entsprechend den **Peanoschen Axiomen** verwenden wir die Definition über die Nachfolger-Relation:

- Der elementare Fall ist die 1 (oder 0).
- Die Rekursion geht über die Nachfolger:

$$f(n + 1) = g(f(n))$$

Rekursion über Listen:

- Elementarer Fall: die leere Liste `'()`.
- Die Rekursion setzt an der Zusammensetzung mit dem *cons*-Operator an:

$$(\text{f} \ (\text{cons} \ x \ xs)) = (\text{cons} \ (g \ x) \ (\text{f} \ xs))$$

Allgemeine Rekursion

Die **allgemeinen Rekursionen** umfassen

- die linearen und
- die nicht-linearen Rekursionen.
- Wichtige allgemeine Rekursionen sind:
 - **indirekte Rekursion**,
 - **baumartige Rekursion**,
 - **geschachtelte Rekursion**.

Indirekte Rekursion

Definition: 61 (Indirekte Rekursion)

Eine rekursivee Definiton heißt **indirekt** oder **verschränkt**, wenn zwei oder mehrere Definitionen sich wechselseitig **rekursiv** verwenden.

```
; Ist x geradzahlig oder ungeradzahlig?
(define (odd? x)
  (cond [(< x 0) (odd? (abs x))]
        [(= 0 x) #f]
        [else (even? (- x 1))]))
(define (even? x)
  (cond [(= 0 x) #t]
        [else (odd? (- x 1))]))
```

odd?: Ein Trace

```
(require racket/trace)
(define (odd? x)
  (cond [(< x 0) (odd? (abs x))]
        [(= 0 x) #f]
        [else (even? (- x 1))]))
(define (even? x)
  (cond [(= 0 x) #t]
        [else (odd? (- x 1))]))
(trace odd? even?) —→ (odd? even?)
> (odd? 5)
|(odd? 5)
|(even? 4)
|(odd? 3)
|(even? 2)
|(odd? 1)
|(even? 0)
|#t      —→ #t
```

Baumartige Rekursion



Definition: 62 (Baumartige Rekursion)

Eine rekursive Definition ist *baumartig* wenn in der Definition in einer Fallunterscheidung mehrfach auf die Definition Bezug genommen wird.

Beispiel: 63 (Fibonacci-Zahlen)

```
(define (fib n)
  (cond [(= n 0) 0]
          [(= n 1) 1]
          [else (+ (fib (- n 1))
                     (fib (- n 2)))]))
```

fib: Ein Trace

```
(define (fib n)
  (cond [(= n 0) 0]
          [(= n 1) 1]
          [else (+ (fib (- n 1))
                     (fib (- n 2))))])
> (trace fib) → (fib)
  (fib 4)
 | (fib 4)
 | | (fib 3)
 | | | (fib 2)
 | | | | (fib 1) → 1
 | | | | (fib 0) → 0
 | | | 1
 | | | | (fib 1) → 1
 | | | 2
 | | | | (fib 2)
 | | | | | (fib 1) → 1
 | | | | | (fib 0) → 0
 | | | | 1
 | | | | | 3 → 3
```

Hier wird ein Binärbaum rekursiv konstruiert:

<http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/SE3-Movies/Pages/Binbaum.html>

Und hier wächst ein Wald von Ternärbäumen:

<http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/SE3-Movies/Pages/TernaerbaumWald.html>

Anmerkung: Die Fibonacci-Zahlen:

Die Fibonacci-Zahlen $\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34 \dots\}$ haben viele interessante Eigenschaften. Beispielsweise wachsen die Zahlen Fib_n mit steigendem n exponentiell an. Man kann zeigen, daß Fib_n diejenige natürliche Zahl ist, die den kleinsten Abstand von $\Phi^n/\sqrt{5}$ hat. Φ ist der *goldene Schnitt*, mit

$$\Phi = (1 + \sqrt{5})/2 \approx 1.6180$$

und der definierenden Gleichung:

$$\Phi^2 = \Phi + 1$$

Das exponentielle Wachstum der Fibonacci-Zahlen wird häufig verwendet, um das Wachstum einer Population zu beschreiben. Eine humorvolle Schilderung eines solchen exponentiellen Wachstums findet sich in Theodor Storms Gedicht von den „Maikätzchen“.



Von Katzen

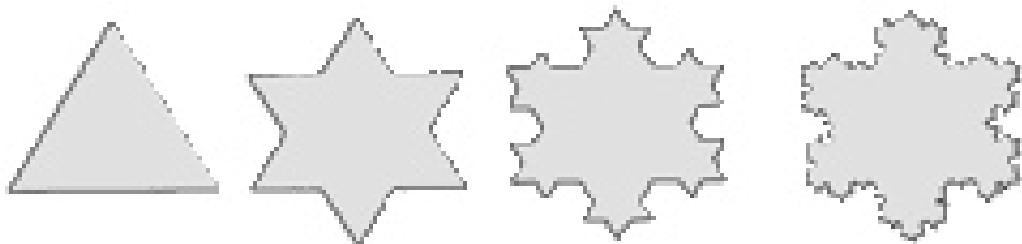
Vergangnen Maitag brachte meine Katze
Zur Welt sechs allerliebste Kätzchen,
Maikätzchen, alle weiß mit schwarzen Schwänzchen,
Fürwahr, es war ein zierlich Wochenbettchen!
Die Köchin aber — Köchinnen sind grausam,
Und Menschlichkeit wächst nicht in der Küche –
Die wollte von den Sechsen fünf ertränken,
Fünf weiße, schwarzgeschwänzte Maienkätzchen
Ermorden wollte dies verruchte Weib.
Ich half ihr heim! — Der Himmel segne
Mir meine Menschlichkeit! Die lieben Kätzchen,
Sie wuchsen auf und schritten binnen kurzem
Erhobnen Schwanzes über Hof und Herd;
Ja, wie die Köchin auch ingrimmig dreinsah,
Sie wuchsen auf, und nachts vor ihrem Fenster
Probierten sie die allerliebsten Stimmchen,
Ich aber, wie ich sie so wachsen sahe,
Ich pries mich selbst und meine Menschlichkeit. –



Ein Jahr ging um, und Katzen sind die Kätzchen,
Und Maitag ist's! — Wie soll ich es beschreiben,
Das Schauspiel, das sich jetzt vor mir entfaltet!
Mein ganzes Haus, vom Keller bis zum Giebel,
Ein jeder Winkel ist ein Wochenbettchen!
Hier liegt das eine, dort das andre Kätzchen,
In Schränken, Körben, unter Tisch und Treppen,
Die Alte gar — nein, es ist unaussprechlich,
Liegt in der Köchin jungfräulichem Bette!
Und jede, jede von den sieben Katzen
Hat sieben, denkt euch! sieben junge Kätzchen,
Maikätzchen, alle weiß mit schwarzen Schwänzchen,
Die Köchin rast, ich kann der blinden Wut
Nicht Schranken setzen dieses Frauenzimmers;
Ersäufen will sie alle neunundvierzig!
Mir selber, ach mir läuft der Kopf davon —
O Menschlichkeit, wie soll ich Dich bewahren!
Was fang ich an mit sechsundfünfzig Katzen! —

Theodor Storm

Baumrekursion: Die Koch'sche Schneeflocke

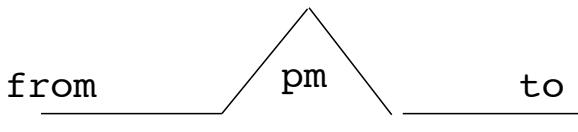


- Ein Bild mit baumartig rekursiver Struktur.
- An jedes *Dreieck* werden rekursiv *drei kleinere Dreiecke* angefügt.
- Der Aufwand wächst exponentiell: In jedem Iterationsschritt vervierfacht sich die Zahl der Linien.

Iteration i	Zahl der Linien $= 3 \times 4^i$	CPU-Zeit MacBook Pro
0	3	0.12 ms
1	12	0.486 ms
2	48	1.946 ms
3	192	7.7868 ms
4	768	31.147 ms
5	3 072	124.58 ms
10	3 145 728	2.1263 min
20	3 298 534 883 328	4.23920 Jahre
36	1.4167099e+22	1.21381 Weltalter
40	3.6267774e+24	310.73675 Weltalter
50	3.802951e+30	325 831 102.9 Weltalter
100	4.820814e+60	4.1303e+38 Weltalter

☞ Anmerkung: Für die obige Tabelle wurde ein Weltalter von ca. 15 Milliarden Jahren seit dem Urknall angenommen.

Der Generator



- Jede Gerade, begrenzt durch die Eckpunkte „from“ und „to“, wird durch vier Geradenstücke ersetzt:
 - from –p1/3,
 - p1/3 –pm,
 - pm –p2/3,
 - p2/3–to,

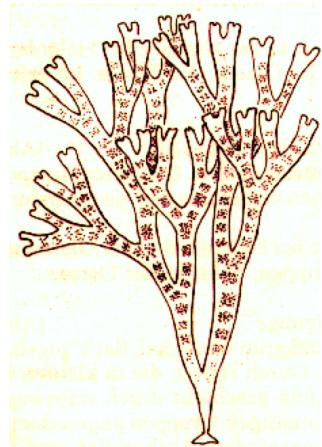
```
(define (snowflake-line from to iter color)
  ; draw a Koch-snowflake from point "from" to point "to"
  ; return the number of segments
  (if (= iter 0)
      (begin (draw-solid-line from to color) 1)
      ; split the line into 4 segments
      (let* ([p1/3 ( interpolate from to 1/3)]
             [p2/3 ( interpolate from to 2/3)]
             [pm ( interpolate from to 1/2)])
        [tipOffset
         (normal from to
                ; the offset of the tip of the new triangle
                (* (distance from to) 1/3 (sqrt 3/4)))
         [tip (translate-pos pm tipOffset)])
        (+ (snowflake-line p1/3 (- iter 1) color)
            (snowflake-line p1/3 tip (- iter 1) color)
            (snowflake-line tip p2/3 (- iter 1) color)
            (snowflake-line p2/3 to (- iter 1) color)))
      )))
  ;;; siehe se3-bib/fractals-module.rkt
```

Hier ist eine Animation des Wachstums der Koch'schen Schneeflocke und ein ganzes fraktales Schneegestöber, erzeugt durch zufällige affine Transformationen.

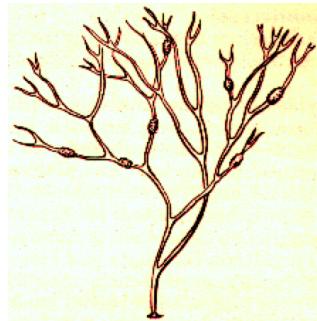
<http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/SE3-Movies/Pages/Snowflake.html>

☞ Anmerkung: Die Funktionen interpolate, normal, distance, translate sind aus dem Modul my-vector-graphics.rkt in der se3-bib library importiert.

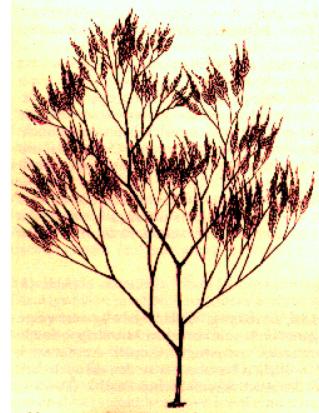
Baumrekursion in der Natur: Algen



Gabelzunge (*Dictyota dichotoma*)



Polyedes rotundus



Polysiphonia elongata

Viele Algen verzweigen sich regelmäßig wie ein Binärbaum (dichotom verzweigt).

Baumrekursion in der Natur: Liliengewächse



Köcherbaum



Drachenbaum

Geschachtelte Rekursion

Definition: 64 (Geschachtelte Rekursion)

Eine Rekursion ist geschachtelt, wenn die Funktion in der rekursiven Verwendung selbst als Argument mitgegeben wird.

Beispiel: 65 (Ackermann-Funktion)

```
; nur fuer kleine x, y: x<=3, y<=3
(define (ack x y)
  (cond [(= 0 y) 0]
        [= 0 x) (* 2 y)]
        [= 1 y) 2]
        [else (ack (- x 1)
                    (ack x (- y 1))))]))
```

Beispiel: 66 (Ein effizienter Modulo-Algorithmus)

nach ([Züllighoven, 1995](#))

Definition: $\text{mod}(a, b) = a, a < b$

$\text{mod}(a, b) = \text{mod}(a - b, b), a \geq b$

Es gilt: $\text{mod}(a, b) = \text{mod}(\text{mod}(a, 2 * b), b), a \geq 2 * b$

```
(define (modRek a b)
  (cond [(< a b) a]
        [(and (<= b a) (< a (* 2 b)))
         (- a b)]
        [else (modRek (modRek a (* 2 b)) b)])))
```

mod: Ein Trace

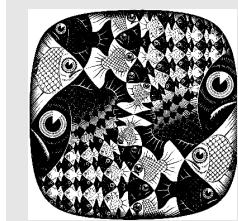
```
(trace modRek)
(modRek 55 3)
|(modRek 55 3)
| (modRek 55 6)
| |(modRek 55 12)
| | |(modRek 55 24)
| | | |(modRek 55 48)
| | | |7
| | | |(modRek 7 24)
| | | |7
```

```
|  |( modRek 7 12)
|  |7
|  (modRek 7 6)
|  1
|( modRek 1 3)
|1  →1
```

☞ **Anmerkung:** Im Eliza-Programm (siehe Abschnitt 24.1) werden Sie einige weitere nützliche Beispiele für geschachtelte Rekursion kennenlernen.

15.2 Korrektheit rekursiver Funktionen

Korrektheit rekursiver Funktionen



11 Semantik

12 Korrektheit und Spezifikation

13 Rekursion und Induktion

- Arten von Rekursion
- Korrektheit rekursiver Funktionen
- Rekursive Prozesse und Endrekursion

< > < > < > < > < > < >

Vollständige Induktion

Das 5. Peanosche Axiom heißt auch das Prinzip der **vollständigen Induktion**. Eine häufig verwendete Fassung des Prinzips lautet:

Induktionsverankerung: Eine Aussage $A(n)$ sei richtig für die Zahl 1.

Induktionsschritt: Zeige, daß aus der Induktionsannahme $A(k)$ für eine natürliche Zahl k stets die Richtigkeit von $A(k')$ für den *Nachfolger* k' folgt.

Induktionsschluß: Die Aussage $A(n)$ gilt dann für *alle* natürlichen Zahlen $n \in \mathbb{N}$.

Korrektheit linear rekursiver Funktionen

- Bei linearen Rekursionen über *natürliche Zahlen* können wir das Prinzip der **vollständigen Induktion direkt** für den Beweis verwenden.
- Bei linearen Rekursionen über andere Strukturen (Listen, Folgen, Reihen...)
 - bilden wir die Folge der Rekursionsschritte bijektiv auf die natürlichen Zahlen ab (wir numerieren die Schritte)
 - und führen den Induktionsbeweis *indirekt* über die zugeordneten Nummern.

Potenziieren, x^n

Beispiel: 67 (Potenzieren, lineare Rekursion über den Exponenten.)

```
(define (power x n)
  ; (and (integer? n) (>= n 0) (number? x))
  ; (number? result)
  (if (= n 0)
      1
      (* x (power x (- n 1)))))
```

Terminieren von power

Beweis: 68

Hilfsdefinitionen: $k = n - 1$, Induktion über k

Induktionsverankerung: $k=-1, n=0$: (`power x 0`) terminiert, da keine weitere Funktionsanwendung erfolgt.

$k=1, n=2$: (`power x 1`) terminiert, da (`power x 0`) terminiert und nur zwei weitere Multiplikationen mit n erforderlich sind.

Die Annahme ist für $k=1$ richtig.

Induktionsschritt von k auf $k+1$:

Annahme: (`power x k`), $k = n - 1$ terminiert.

Dann muß nur noch die Multiplikation mit x erfolgen, die auch terminiert. Also terminiert dann auch die rekursive Anwendung für den Nachfolger $k+1=n$.

Induktionsschluß: `power` terminiert für alle $k \in \mathcal{N}$.

Terminiert mymap?

Beispiel: 69 (Induktion über die Länge n der Liste)

```
(define (my-map f xs)
  (if (null? xs) '()
    (cons (f (car xs))
          (my-map f (cdr xs)))))
```

Beweis: 70

Hilfsdefinitionen: $k = (\text{length } xs)$, Induktion über k ,

Annahme: f terminiert für alle Argumente.

Induktionsverankerung: $(\text{my-map } f '())$ terminiert, da keine weitere Funktionsanwendung erfolgt. Die Annahme ist für $k=0$ richtig.

$(\text{my-map } f '(x))$ terminiert für beliebige x , da $(\text{cons } (f x) (\text{my-map } f '()))$ terminiert. Die Annahme ist für $k=1$ richtig.

Induktionsschritt: Sei $(\text{cons } x \ xs)$ eine Liste der Länge $k+1$. Wir postulieren, daß $(\text{my-map } f \ xs)$ terminiert.

Da nach dem Terminieren von $(\text{my-map } f \ xs)$ nur noch die **cons**-Funktion und f aufgerufen werden, terminiert auch der rekursive Aufruf für Listen der Länge $k+1$.

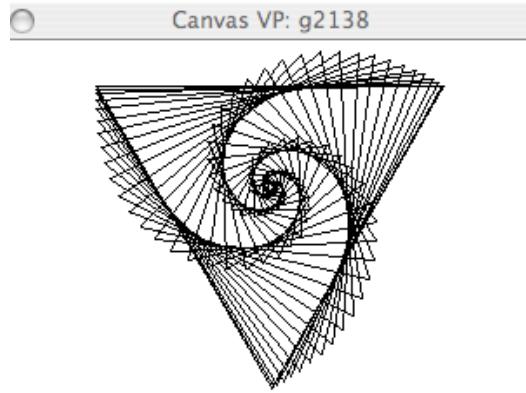
Induktionsschluß: my-map terminiert für Listen jeder Länge $k \in \mathcal{N}$.

Terminieren einer Rekursion: Bedingungen



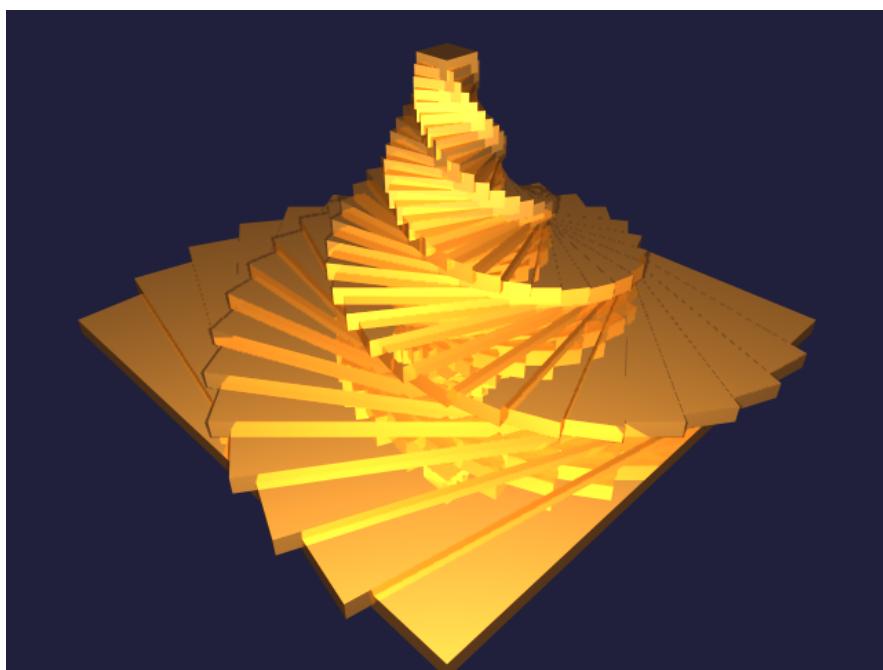
- ▶ Jeder Rekursionsschritt sollte die Argumente dichter an die Abbruchbedingung bringen, z.B.
 - ▶ ein Zahlenargument inkrementieren
 - ▶ oder ein Listenargument verkürzen.
- ▶ Die Abbruchbedingung muß so formuliert werden, daß man nicht über das Ziel hinausschießen kann.

Ein linear-rekursiver Prozeß



Rekursiv gedrehte und skalierte Polygone, siehe: Modul lilies-module.rkt

Ein linear rekursiver Prozeß, Povray-Animation



[http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/
SE3-Movies/Pages/Pyramide.html](http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/SE3-Movies/Pages/Pyramide.html)

15.3 Rekursive Prozesse und Endrekursion

Rekursive Prozesse

- Funktionale Ausdrücke definieren nicht nur den *Wert*, für den sie stehen, sondern auch einen *Berechnungsprozeß*, der diesen Wert ermittelt.
- Rekursive Ausdrücke können sehr unterschiedlich aufwendige Berechnungsprozesse auslösen. Geringe Unterschiede in der Definition können den Unterschied zwischen einem effizienten Programm und völlig unbrauchbaren Programmen ausmachen.

Ein linear rekursiver Prozeß: Nachklappern

Beispiel: 71 (Länge einer Liste, allgemein rekursiv)

```
(define (my-length xs)
  (if (null? xs) 0
      (+ 1
          (my-length (cdr xs)))))

(my-length '(a b c))
|(my-length (a b c))
|+ 1 (my-length (b c))
||+ 1 (my-length (c))
||+ 1 (my-length ())
|| 0
|| 1
|| 2
| 3       → 3
```

☞ **Anmerkung:** In alten Programmiersprachen, wie FORTRAN IV, war Rekursion nicht zugelassen, weil das Vorurteil herrschte, daß Rekursion ineffizient, unnötig und nur für Theoretiker interessant sei. Teilweise ging dieses Vorurteil darauf zurück, daß oftmals aus Übermut und Freude an den eleganten Formulierungen Rekursion eingesetzt wurde, wo sie nichts zu suchen hatte. Bei Rechenzeit-kritischen numerischen Anwendungen ist es wirklich nicht angemessen, beispielweise Determinanten rekursiv auszurechnen, wenn man keinen Compiler hat, der Endrekursion (siehe Definition 72) effizient behandeln kann. Andererseits gibt es relevante Probleme, die nur mit rekursiven Algorithmen zu lösen sind. In diesem Fall müssen wir in einer Sprache, die Rekursion nicht zuläßt, die entsprechenden Prozesse mühsam von Hand nachbauen.

Aufwand einer Rekursion

Der Aufwand, den ein rekursiver Prozeß verursacht, hat zwei Aspekte.

1. *Rechenaufwand*: In jedem Rekursionsschritt muß in diesem Beispiel ein Zwischenergebnis um „1“ erhöht werden. Dieser Rechenaufwand ist proportional zur Länge der Eingabeliste.
2. *Speicherplatz*: Mit den Additionen kann erst begonnen werden, wenn das Ende der Liste erreicht ist. Bis das der Fall ist, müssen alle partiellen Berechnungen zurückgestellt werden.

Zur Beschreibung der noch ausstehenden Berechnungen muß Speicherplatz bereitgestellt werden. Auch dieser Platzbedarf ist bei einer linearen Rekursion proportional zur Zahl der Rekursionsschritte.

Aufrufkeller und Nachklappern

- Die zurückgestellten Berechnungen werden in umgekehrter Reihenfolge *last in, first out* ausgeführt, wenn die Funktion aus der rekursiven Verschachtelung zurückkehrt.
- Daher werden sie typischerweise in einem Kellerspeicher (stack) abgelegt.
- Dieses Abschließen der Berechnungen heißt Nachklappern.
- Nachklappern kann vermieden werden, wenn eine Funktion so formuliert wird, daß alle Berechnungen erfolgt sind, ehe der rekursive Aufruf erfolgt, siehe Beispiel „add“.

Endrekursion

Definition: 72 (Endrekursion)

- Rekursive Funktionen, bei denen das Ergebnis der Rekursion nicht mehr mit anderen Termen verknüpft werden muß, heißen *endrekursiv* (engl. tail-recursion).
- Die zugehörigen Prozesse heißen *iterative Prozesse*.

Satz: 73 (Minimalrekursiv und endrekursiv:)

Minimalrekursive Funktionen sind endrekursiv.

Endrekursion in Lisp und Racket

☞ **Garantierte Optimierung in Racket:**

In Racket oder Common Lisp übersetzt der Compiler endrekursive Funktionen automatisch in eine echte *Iteration*, so daß der Rechenzeit- und Speicherbedarf für die rekursiv definierte Funktion nicht größer ist, als wenn sie direkt mit einer Iterationsschleife definiert worden wären.

Länge einer Liste, endrekursiv

Beispiel: 74 (length-acc, ohne Nachklappern)

```
(define (length_acc xs acc)
  (if (null? xs)
      acc
      (length_acc (cdr xs)
                  (+ acc 1))))
```



```
(define (my_length_acc xs)
; verborgen des Akkumulators
  (length_acc xs 0))
```

Trace: Länge einer Liste, endrekursiv

Beispiel: 75 (my-length-acc, ohne Nachklappern)

```
> (my_length_acc '(1 2 3 4 5))
|(my_length_acc (1 2 3 4 5))
|/(length_acc (1 2 3 4 5) 0)
|/(length_acc (2 3 4 5) 1)
|/(length_acc (3 4 5) 2)
```

```

|((length_acc (4 5) 3)
|((length_acc (5) 4)
|((length_acc () 5)
|5 → 5

```

Wandlung in Endrekursion, Akkumulator

- Um eine Funktion in eine endrekursive Form zu überführen, wird (meist) ein zusätzlicher Parameter, ein Akkumulator, benötigt, der die bisherigen Zwischenergebnisse sammelt (akkumuliert).
- Mit Erreichen der Abbruchbedingung steht dann das Ergebnis fest.
- Um den Akkumulator zu verbergen und korrekt zu initialisieren, sollte man eine einhüllende Funktion definieren.

Ein baumartig-rekursiver Prozeß

Bei unserer Definition der Fibonacci-Zahlen entsteht ein baumartiger Prozeß, da viele Aufrufe zu zwei rekursiven Aufrufen führen, die wiederum zwei Aufrufe auslösen, usw.

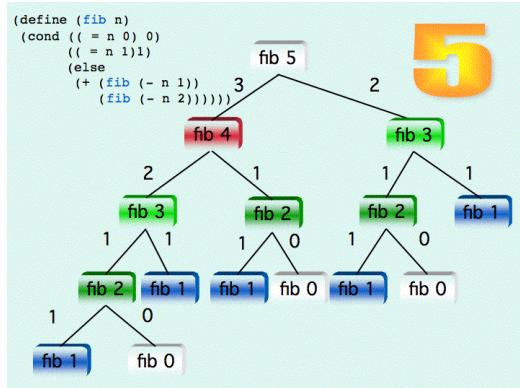
```

(define (fib n)
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [else (+ (fib (- n 1))
                  (fib (- n 2))))])

```

Beobachtung

- Die baumrekursive Variante zur Berechnung der Fibonacci-Zahlen macht unnötige Doppelberechnungen.
- Der Aufwand wächst exponentiell mit n .
- Wir führen Akkumulatoren ein, um die Zwischenergebnisse zu speichern.
- Diese Formulierung ist endrekursiv und der Prozeß linear in n .



Fibonacci-Zahlen, endrekursiv

Beispiel: 76 (Fibonacci-Zahlen, endrekursiv)

```
(define (fib-acc n a1 a2)
  (cond [ (= n 0) a1]
        [ (= n 1) a2]
        [else
          (fib-acc (- n 1)
                     a2
                     (+ a1 a2))])))
(define (fibE n)(fib-acc n 0 1))
```

Trace: Fibonacci-Zahlen, endrekursiv

```
> (fibE 5)
|(fib-acc 5 0 1)
|(fib-acc 4 1 1)
|(fib-acc 3 1 2)
|(fib-acc 2 2 3)
|(fib-acc 1 3 5)
|5
5
```



Datenmüll und Effizienz

Auch die Konstruktion von Datenstrukturen erfordert Rechenaufwand. Jede cons-Operation legt eine cons-Zelle an und verbraucht Speicherplatz. Die cons-Zellen von Listen, die wir nicht mehr erreichen können, stellen Datenmüll dar.

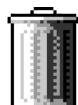
Wenn Racket Platz benötigt, werden nicht mehr benötigte Speicherplätze wieder freigegeben (recycled). Dieser Vorgang heißt *garbage collection* und ist sehr aufwendig, denn für jeden freigegebenen Speicherplatz muß festgestellt werden, ob sich noch Referenzen im Programm auf die Daten beziehen. Der Weg zu effizienten Racket-Programmen geht jedenfalls ganz umweltbewußt über die Müllvermeidung und nicht über das Wiederverwenden.

Sehr müllintensiv sind beispielsweise rekursive Funktionen mit `append`. Wenn wir in jedem Rekursionsschritt etwas an das Zwischenergebnis anhängen, wird immer der Anfang kopiert — mit neuen `cons`-Zellen, die im nächsten Rekursionsschritt auf den Müll wandern. Listen sollten nach Möglichkeit immer mit `cons` konstruiert werden, weil wir dann in jedem Schritt nur eine `cons`-Zelle brauchen.



Freispeicherverwaltung

- Wenn Racket Platz benötigt, werden nicht mehr benötigte Speicherplätze wieder freigegeben.
- Dieser Vorgang heißt „*garbage collection*“ und ist sehr aufwendig, denn für jeden freigegebenen Speicherplatz muß festgestellt werden, ob sich noch Referenzen im Programm auf die Daten beziehen.
- Der Weg zu effizienten Racket-Programmen geht jedenfalls ganz umweltbewußt über die *Müllvermeidung* und nicht über das Wiederverwenden (recycling).

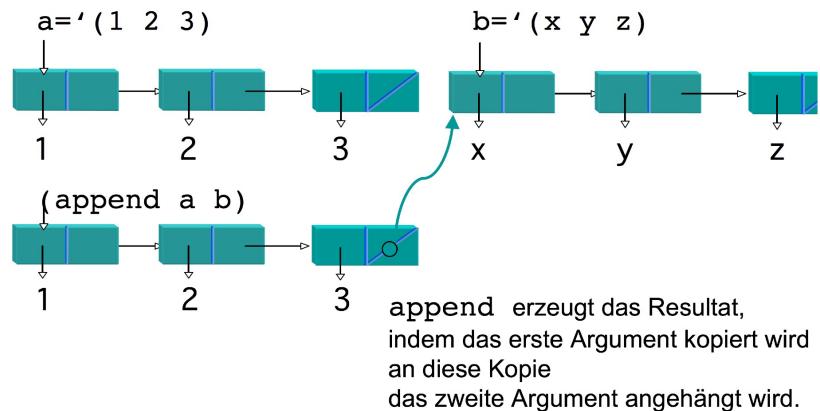


Entstehung von Datenmüll

```
> (define garbage1  
    "„Alles Müll!“)  
> (define garbage2  
    '(und noch mehr Muell!))  
> (set! garbage1 'wegdamit)  
  
> (set! garbage2 'auchweg)
```

Der String „garbage1“ und die Liste „garbage2“ sind jetzt nicht mehr zugreifbar und belasten unnötig den Speicher.

append a b



Müllintensiv: append

(Kopf | und Schwanz) → (Schwanz und | Kopf)

```
> (define (my-reverse xs)
  (if (null? xs) '()
      (append (my-reverse (cdr xs))
              (list (car xs)))))
```

- Das erste Argument von `append` wandert bei jedem Rekursionsschritt auf den Müll,
- denn `append` erzeugt das Resultat,
 - indem das erste Argument *kopiert* wird
 - und an diese Kopie das zweite Argument angehängt wird.

```

> (trace append my-reverse) → (append ...
> (my-reverse '(M A I S))
| (my-reverse (M A I S))
| (my-reverse (A I S))
| ((my-reverse (I S)))
| | (my-reverse (S))
| | | (my-reverse ())
| | | ()
| | | (append () (S))
| | | (S)
| | | ((append (S) (I)))
| | | (S I)
| | | (append (S I) (A))
| | | (S I A)
| | (append (S I A) (M))
| | (S I A M)
(S I A M)

```

Messen des Aufwands

```

(define (nats n)
; eine Liste mit n natürlichen Zahlen
(if (<= n 0) '()
(cons n (nats (- n 1)))))

(define (timereverse n)
; Laufzeitmessung für my-reverse
(time (my-reverse (nats n)))
#t)

>(timereverse 10000) → #t
cpu time: 6019 real time: 6083 gc time: 4056
>(timereverse 30000) → #t
cpu time: 55321 real time: 57118
gc time: 37594

```

☞ Anmerkung: Die Funktion **time** drückt die Rechenzeit für eine s-expression aus.

Die Funktion timereverse gibt als Resultat #t zurück, um die Ausgabe der langen Resultatlisten zu unterdrücken.

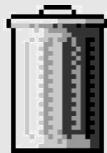
Aufwand: Spiegeln einer Liste mit „append“

Länge	100	1 000	10 000	20 000
CPU (ms)	0	24	7233	23831
gc (ms)	0	0	5249	15935
Garbage Zellen	5 050	500 500	50 005 000	200 010 000

- ☞ Der Rechenaufwand und der Speicheraufwand für cons-Zellen wachsen mit dem Quadrat der Listenlänge.
- ☞ Für die rekursive Konstruktion von Listen sollte daher möglichst *cons* und nicht *append* verwendet werden.



Umgang mit Datenmüll



☞ Konstruierte Datenobjekte sind **Müll**, wenn es keine Referenzen mehr darauf gibt, denn die Programme können nicht mehr darauf zugreifen.



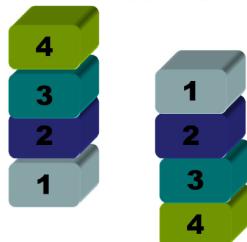
☞ Racket sammelt Müll automatisch ein und „recycled“ die Speicherelemente. Der Vorgang heißt **garbage collection** und ist rechenaufwendig.



☞ Wir sollten beim Entwurf von Funktionen den anfallenden Müll beachten und umweltbewußten Algorithmen den Vorzug geben, die wenig *garbage* erzeugen.

< > < > < > < > < > < > < > < >

Reverse, effizient mit „cons“
Idee: Umstapeln



reverse, effizient

Die akkumulierenden, endrekursiven Fassungen sind oft auch diejenigen, bei denen am wenigsten *garbage* entsteht.

```
> (define (oeko-reverse xs)
  (reverse-akk '() xs))

> (define (reverse-akk akku xs)
  (if (null? xs) akku
    (reverse-akk
      (cons (car xs) akku)
      (cdr xs))))
```

Spiegeln einer Liste mit „cons“

Aufwand: akkumulierend mit cons

Länge	1 000	10 000	100 000	1 000 000
CPU-Zeit (ms)	0	5	43	430
gc-Zeit (ms)	0	0	0	0

- ☞ Der Rechenaufwand wächst *linear* mit der Listenlänge.
- ☞ Der Speicheraufwand für cons-Zellen-Garbage ist constant = 0.
- ☞ Für die rekursive Konstruktion von Listen sollte daher möglichst *cons* und nicht *append* verwendet werden.

Bei einer Liste der Länge 1000 hat die Version mit *append* eine halbe Million cons-Zellen verschwendet, während die akkumulierende Version nur soviele cons-Zellen verbraucht hat, wie das Resultat zur Repräsentation benötigt, und die Rechenzeit war wesentlich kürzer.

Bei *my-reverse* ist der Zeit- und Speicheraufwand für eine 10-mal längere Liste um den Faktor 100 gestiegen, während er für *oeko-reverse* nur um den Faktor 10 gestiegen ist. Generell steigt der Aufwand für *myreverse* ungefähr proportional zum Quadrat der Listenlänge, während der Aufwand bei *oeko-reverse* linear mit der Listenlänge zunimmt. Je länger die Listen werden, desto vorteilhafter wird die endrekursive Funktion mit *cons* gegenüber der Version mit *append*. *my-reverse* hat mehr Zeit mit der garbage collection (cpu time (gc)) als mit dem eigentlichen Rechnen (cpu time (non-gc)) verbracht.

Sie sehen an diesem Beispiel, daß Racket-Code sehr effizient sein kann, aber daß wir auch sehr genau wissen müssen, wodurch Rechen- und Speicheraufwand verursacht wird.

Hier wächst ein Wald von blühenden Binärbäumen mit Blättern:

<http://kogs-www.informatik.uni-hamburg.de/~dreschle/Animationen/SE3-Movies/Pages/BinbaumBlueten.html>

Teil VI

Der Entwurf von Funktionen

16 Funktionen höherer Ordnung

16.1 Funktionen als Werte

Funktionen als Werte



M.C. Escher

- 14 Funktionen höherer Ordnung
 - Funktionen als Werte
 - Beispiel: Funktion und Ableitung
 - Der Baukasten
- 15 Kontrollabstraktion
- 16 Modularer Entwurf

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 426 / 1157

Funktionen höherer Ordnung

Definition: 77 (Funktionen höherer Ordnung)

Funktionen höherer Ordnung (high order functions) sind Funktionen,

- die Funktionen als *Argumente* erhalten
- oder als *Wert* zurückgeben.

Funktionen als Werte

- Jede Funktion ist in Racket ein Wert mit einem wohldefinierten Typ.
- Funktionen können die *Argumente* von anderen Funktionen sein:

(**sort** '(3 5 5 4 6 2) <)
→ (2 3 4 5 5 6)

- Funktionen können der Wert eines Ausdrucks sein:

```
> (define x 3)
> (if (> x 3) sin cos)
   → #<primitive:cos>
```

- Funktionen können als Werte Teile von Datenstrukturen sein, z.B. Elemente einer Liste

```
'( sin , cos , sqrt ).
```

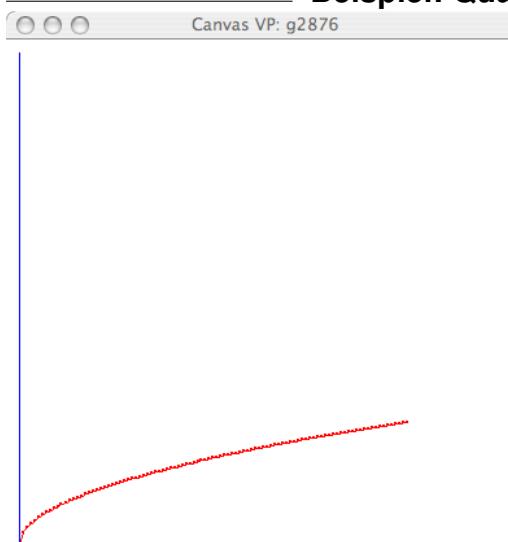
Beispiel: Zeichnen von Kurven

- In Rackets Standard-Modul *graphing.rkt* wird die Funktion *graph-fun* bereitgestellt:

```
(require htdp/graphing)
(graph-fun <procedure> <color>)
```

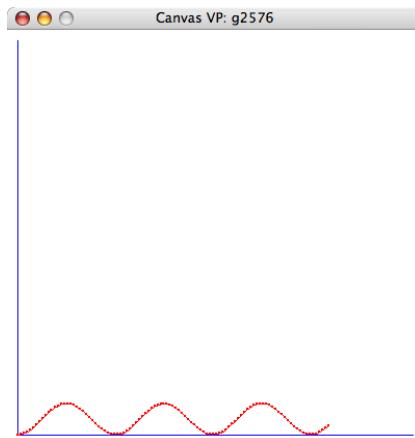
- *graph-fun* zeichnet die Kurve der Funktion <procedure> in der angegeben Farbe
- in den rechten oberen Quadranten des kartesischen Koordinatensystems, $0 < x < 10.0, 0 < y < 10$.
- *graph-fun* ist eine Funktion höherer Ordnung, da ihr Argument eine Funktion ist.

Beispiel: Quadratwurzel



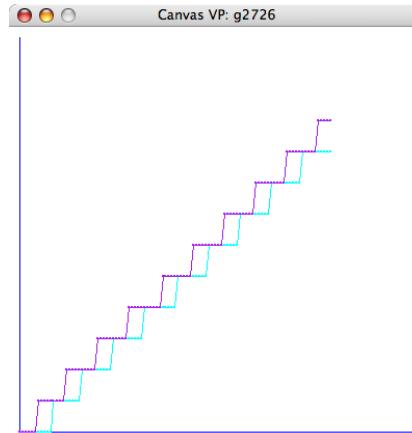
```
> ( graph-fun sqrt 'red) → #t
```

Beispiel: Zeichnen einer anonymen Funktion



```
> ( graph-fun  
  (lambda (x) ;  $\sin^2 x$   
    (* (sin x)  
       (sin x)))  
  'red) → #t
```

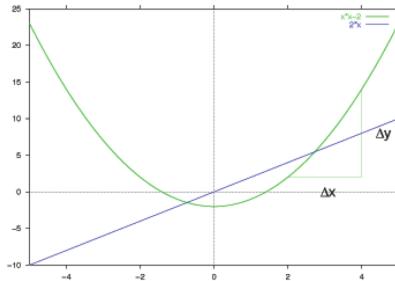
Beispiel: round und truncate



```
(require htdp/graphing)  
> ( graph-fun truncate 'cyan) → #t  
> ( graph-fun round 'purple) → #t
```

16.2 Beispiel: Funktion und Ableitung

Numerische Ableitung einer Funktion



Beispiel: 78 (Generieren einer Ableitungsfunktion)

Wir berechnen numerisch die Ableitung von f an der Stelle x und approximieren

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x}$$

```
(define (deriv f)
  ; die Ableitungsfunktion f'
  (let ([dx 0.00001])
    (lambda (x)
      (/ (- (f (+ x dx)) (f x))
          dx))))  
  
(define sinAbl (deriv sin)); sin'  
  
> sinAbl → #<procedure>
> (sinAbl 0) → 0.9999999999833332
> (define sinAblAbl (deriv sinAbl))
> (sinAblAbl 0.0)
-1.00000082740371e-05
```

Anmerkung:

- Der Wert des Terms `(deriv f)` ist die Ableitungsfunktion f' ,
- nicht der Wert der Ableitung an einer bestimmten Stelle.

Beispiele

Language: racket .

```
> pi → 3.141592653589793
> (deriv sin) → #<procedure>
> ((deriv sin) 0) → 0.9999999999833332
> ((deriv sin) (/ pi 2))
   → -5.000000413701855e-06
> (define (square x)
  (* x x))
> ((deriv square) 1) → 2.00001000001393
> ((deriv square) 3) → 6.00009999951316
> ((deriv (lambda (x)
  (* -4 x))) 5)
   → -3.999999998485687
```

Die Ableitung einer Funktion

Die Funktion deriv ist eine typische Funktion *höherer Ordnung* (higher order function).

- Ihr *Argument* f ist eine Funktion, nämlich die Funktion, deren Ableitungsfunktion f' wir benötigen.
- Der berechnete *Wert* ist ebenfalls eine Funktion, die erste Ableitung f' des Arguments f .
- Um den Wert der Ableitung an einer bestimmten Stelle x zu erfahren, müssen wir f' auf x anwenden.
- Wir können f' wie jede andere Funktion in den Ausdrücken verwenden, z.B. $((\text{deriv } \sin) 0)$, berechne die Ableitung der Sinus-Funktion für $x = 0$.

16.3 Der Baukasten

Der Baukasten-Ansatz



- 14 Funktionen höherer Ordnung
 - Funktionen als Werte
 - Beispiel: Funktion und Ableitung
 - Der Baukasten
- 15 Kontrollabstraktion
- 16 Modularer Entwurf

Idiome

- Die *Idiome* gehören zur *Pragmatik* der Programmiersprache – wie wird die Sprache verwendet?
- Idiome sind Standardlösungen für typische Probleme, z.B. das Durchsuchen einer Liste.
- Zum Beherrschung eines Programmierstils und einer Programmiersprache gehört es daher auch, daß man diese Standardlösungen für typische Aufgaben kennenlernt.
- Um gut Schach spielen zu können, reicht es ja auch nicht, die Regeln zu kennen, sondern wir müssen typische Stellungen und Strategien erkennen können.

Vorteile von Standardlösungen und Idiomen

- Andere können unsere Programme besser und schneller verstehen, wenn sie mit den Idiomen vertraut sind.
- Die Eigenschaften der Standardlösungen (Effizienz, Zusicherungen) sind wohlbekannt, und wir können darauf sicher aufbauen.
- Das macht Korrektheitsbeweise einfacher und Programme robuster.

- Die Programmierung geht zügiger, wenn wir nicht jedesmal das Rad neu erfinden.

 **Anmerkung:** Als Informatikerin oder Informatiker sollten Sie Standardprobleme der Programmierung auf einen Blick erkennen können, etwa so, wie ein erfahrerer Hausarzt eine gängige Kinderkrankheit wie Masern oder Keuchhusten auf einen Blick erkennen kann. Diese Routine bekommen Sie durch das sorgfältige Studium vieler vorbildlicher Programme, indem Sie bei erfahrenen Informatikerinnen und Informatikern in die Lehre gehen und indem Sie selbst viel programmieren, aber sinnvoll. Verwenden Sie also lieber Ihre Arbeitszeit auf die konzeptuelle Seite der Programmierung, die Architektur und den soliden Entwurf des Programms, nicht auf Effekthascherei oder das Herausschinden der letzten Nanosekunden an Laufzeit.

In der Vorlesung können wir Ihnen leider nur sehr kurze Beispielprogramme vorstellen. Diese „Spielprogramme“ können aber nicht gut vermitteln, wie wirkungsvoll und nützlich die funktionale Abstraktion als softwaretechnische Methode beim Entwurf großer Systeme ist. Viele gute Beispiele für die Anwendung des funktionalen Programmierstils als Entwurfsmethode finden Sie bei [Norvig, 1992](#). Hier sehen Sie, wie Funktionen höherer Ordnung effektvoll eingesetzt werden können, um eine modulare und flexible Systemarchitektur zu entwerfen.

Elementare Idiome der funktionalen Programmierung werden wir uns auf den folgenden Seiten anschauen.

17 Kontrollabstraktion

17.1 Funktionsapplikation

Funktionsapplikation (apply)

Problem: 79 (Ein häufiges Problem:)

- Gegeben sei
 - eine Liste von n Werten
 - sowie eine Funktion f von n Argumenten.
- Wir wollen die Elemente der Liste als Argumente für die Funktion f verwenden: Sei f beispielsweise `max`:

```
> (define data '(2 4 7 3 5 1.))
> (max (car data) (cadr data) (caddr data)...)
> (apply max data)           → 7.0
> (apply < '(1 2 3 3 4))   → #f
> (apply <= '(1 2 3 3 4))  → #t
```

`apply` bindet die Elemente einer Liste an die Argumente einer Funktion.

Ein Vertrag für sort

```
(define contListofNumber
  (listof (flat-contract number?)))

(define contSorted<=
  (flat-contract
    (lambda (xs)
      (apply <= xs)))))

(define/contract
  sort<
  (contListofNumber . -> . contSorted<= )
  (lambda (nums)
    (sort nums <)))
```

Konstruktoren für Verträge

- Viele Konstruktoren für Verträge sind **Funktionen höherer Ordnung** mit Prädikaten als Argument, z.B.

```
(flat-contract number?)

(flat-named-contract type-name predicate?)
```

Implementation von eval

Wenn Ihr Racket-System **eval** nicht kennt, können Sie funktionale Ausdrücke so auswerten:

```
> (define (my-eval expr)
  (apply (car expr) (cdr expr)))

> (define expr (list max 1 2 3 0))
   ; Liste aus der Funktion max und Werten
> (my-eval expr) → 3
```

17.2 Funktionen höherer Ordnung für Listen

17.2.1 Abbilden (mapping)



Abilden (mapping)

Problem: 80

Eine Folge oder Menge von Werten soll in einheitlicher Weise transformiert werden, so daß die Anzahl und Anordnung der Elemente erhalten bleiben.

Lösung: 81

- Repräsentation der Folge oder Menge als Liste
- und Transformation der Liste mit einer „mapping-function“: map.

$$\begin{array}{rcl} x : & (& 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \dots &) \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & f(x) = 2^x \\ 2^x : & (& 1 \quad 2 \quad 4 \quad 8 \quad 16 \quad 32 \quad \dots &) \end{array}$$

Die Standardfunktion „map“

- Die Standardfunktion **map** wendet eine Funktion auf jedes Element einer Liste an
- und gibt als Resultat eine Liste der abgebildeten Werte zurück.

```
(define (zwei-hoch n) (expt 2 n))
> (map zwei-hoch '(0 1 2 3 4 5 6 7))
  → (1 2 4 8 16 32 64 128)
> (map last-name *computer-scientists*)
  → (Babbage Church Leibniz Hopper Lovelace
      McCarthy Neumann Turing Pascal)
> (map first-name2 *computer-scientists*)
  → (Charles Alonzo Gottfried Grace Ada
      John John Alan Blaise)
```

Variable Zahl von Argumenten

- *map* kann mehrere Listen parallel abbilden:
- Die Listen müssen gleich lang sein.
- Die Stelligkeit der Funktion muss gleich der Zahl der Listen sein.

```
> (map + '( 1 2 3 4)
      '( 10 20 30 40)
      '(100 200 300 400))
   → (111 222 333 444)
```

Variable Zahl von Argumenten

```
> (map list
    (map first-name2 *computer-scientists*)
    (map last-name *computer-scientists*))
((Charles Babbage)
 (Alonzo Church)
 (Gottfried Leibniz)
 (Grace Hopper)
 (Ada Lovelace)
 (John McCarthy)
 (John Neumann)
 (Alan Turing)
 (Blaise Pascal))
```

„cons“ versus „list“

```
(map cons
  (map first-name2 *computer-scientists*)
  (map last-name *computer-scientists*))
((Charles . Babbage)
 (Alonzo . Church)
 (Gottfried . Leibniz)
 (Grace . Hopper)
 (Ada . Lovelace)
 (John . McCarthy)
 (John . Neumann)
 (Alan . Turing)
 (Blaise . Pascal))
```

17.2.2 Filtern

Problem: 82

Aus Folge oder Menge von Werten sollen diejenigen Werte ausgewählt werden, die eine bestimmte Bedingung erfüllen.

Lösung: 83

- Repräsentation der Folge oder Menge als Liste
- und Filtern der Liste mit einer Filter-Funktion.

$$\begin{array}{ccccccccc} (& 2 & 3 & 4 & 5 & 6 & 7 & \dots &) \\ & \downarrow & \downarrow & \perp & \downarrow & \perp & \downarrow & \text{Primzahl?} \\ (& 2 & 3 & & 5 & & 7 & \dots &) \end{array}$$

- Die Funktion `filter` nimmt ein *Prädikat* und eine Liste als Argument
- und bildet die Teilliste aller Elemente, die das Prädikat erfüllen.

```
(define (filter p? xs)
  (cond [(null? xs) '()]
        [(p? (car xs))
         (cons (car xs)
               (filter p? (cdr xs)))]
        [else (filter p? (cdr xs))]))
> (filter odd? '(1 3 5 4 3 5 2 7))
  → (1 3 5 3 5 7) ; ungerade Zahlen
> (filter pair? '(auto (bus) (2 3) haus))
  → ((bus) (2 3)) ; Unterlisten
> (filter boolean? '( 1 2 haus #t pi #f))
  → (#t #f) ; Wahrheitswerte
```

`filter` ist eine Standardfunktion von Racket.

Algebraische Eigenschaften von filter

Für Terme mit `filter` gelten die folgenden Zusicherungen (ohne Beweis):

$$\begin{aligned}
 & (\text{filter } p \text{ (append } xs \text{ } ys)) \\
 & = (\text{append } (\text{filter } p \text{ } xs) (\text{filter } p \text{ } ys)) \\
 & (\text{filter } p \text{ (filter } q \text{ } xs)) \\
 & = (\text{filter } q \text{ (filter } p \text{ } xs))
 \end{aligned}$$

Satz: 84 (Eigenschaften von filter)

- *filter ist also distributiv bezüglich der Konkatenation von Listen.*
- *Mehrfaches Filtern ist unabhängig von der Reihenfolge, solange unsere Funktionen referentiell transparent sind!*

17.2.3 Falten (foldl, foldr)

Problem: 85

Die Elemente einer Folge von Werten sollen

- paarweise mit demselben Operator verknüpft
- und zu einem Wert reduziert werden.

Lösung: 86

- Repräsentation der Folge als Liste
- und Reduzieren der Liste mit einer Faltungsfunktion *foldl* oder *foldr*.

$$\begin{array}{ccccccccc} (& 0 & 1 & 2 & 3 & 4 & 5 & 6 \cdots &) \\ & \curvearrowright \\ (\in & +0 & +1 & +2 & +3 & +4 & +5 & +6 \cdots &) \end{array}$$

Falten

- Die Funktion *foldl* nimmt die drei Argumente:
 1. einen Operator f,
 2. einen Startwert (neutrales Element) e (Resultat im Fall der leeren Liste)
 3. sowie eine (oder mehrere) zu faltende Liste(n).
- und verknüpft die Listenelemente von links nach rechts.
- Der übergebene Operator (Funktion) f ist zweistellig
 - und nimmt als erstes Argument das jeweilige Listenelement
 - und als letztes Argument das bisherige Resultat oder den Startwert.

☞ Anmerkung: In Common Lisp heißt die Funktion „*foldl*“ *reduce*. Werden an *foldl* mehrere Listen übergeben, so muß die Stelligkeit von f um eins größer sein als die Anzahl der Listen.

Beispiele:

```

> (foldl + 0 '(1 2 3) ) ; Summe der Werte → 6
> (foldl * 1 '(1 2 3) ) ; Produkt der Werte → 6
> (foldl min 1 '(1 2 3) ) ; Minimum der Werte → 1
> (foldl + 0
    (map (lambda (x) 1)
          '(1 2 3)) ); length
→ 3

```

foldl: Endrekursive Reduktion

```

(define (myfoldl f seed xs )
  ; falte die Liste xs unter Verwendung
  ; der Funktion f und
  ; des neutralen Elements seed.
  (cond [(null? xs) seed]
        [else
         (myfoldl f
                   (f (car xs) seed)
                   (cdr xs)
                   ))]))

```

- Diese Faltungsfunktion ist *endrekursiv*.
- Die Argumente werden von links nach rechts im Argument seed verknüpft, das hier als **Akkumulator** wirkt.

Ein trace

```

> (require racket/trace)
> (trace myfoldl)
> (myfoldl cons () '(3 2 1) )
> (myfoldl #<procedure:cons> () (3 2 1))
> (myfoldl #<procedure:cons> (3) (2 1))
> (myfoldl #<procedure:cons> (2 3) (1))
> (myfoldl #<procedure:cons> (1 2 3) ())
<(1 2 3) → '(1 2 3)
> (untrace myfoldl)

```

☞ Ein Reduzieren mit **cons** bewirkt eine *Umkehrung* der Liste.

Allgemeinrekursive Reduktion

```
(define (myfoldr f seed xs)
  ; allgemein rekursive Reduktion
  (if (null? xs)
    seed
    (f (car xs)
      (myfoldr f seed (cdr xs))))))
> (myfoldr + 0 '(1 2 3 4) ) → 10
> (myfoldr cons '() '(4 3 2 1) )
   → (4 3 2 1)
```

- Die Standardfunktion foldr ist wie myfoldr *linear rekursiv*.
- Die Argumente werden beim rekursiven Aufstieg von rechts nach links mit dem vorläufigen Ergebnis verknüpft;
- einen **Akkumulator** gibt hier es nicht.

Sprache: racket ; memory limit: 256 MB.

```
> (myfoldr cons '() '(4 3 2 1) )
> (myfoldr #<procedure:cons> () (4 3 2 1))
> (myfoldr #<procedure:cons> () (3 2 1))
> > (myfoldr #<procedure:cons> () (2 1))
> > > (myfoldr #<procedure:cons> () (1))
> > > > (myfoldr #<procedure:cons> () ())
< < <()
< < (1)
< <(2 1)
< (3 2 1)
<(4 3 2 1) → '(4 3 2 1)
>
```

Nicht-kommutative Operatoren

Reduktionsreihenfolge

Die Reduktionsreihenfolge ist wichtig,

- wenn der Reduktionsoperator nicht kommutativ ist, denn das Ergebnis kann unterschiedlich sein,
z.B. bei **cons**, **-**, **expt**
- und wenn die Liste sehr lang ist: die allgemein rekursive Fassung (foldr) baut unnötig einen tiefen Aufrufkeller auf.

☞ Anmerkung: foldl und foldr sind Standardfunktionen in Racket. Die hier nachprogrammierten Fassungen myfoldl, myfoldr finden Sie in der se3-bib im Verzeichnis se3-bib/Beispiele/kapitel6highOrder.rkt. <http://kogs-www.informatik.uni-hamburg.de/~dreschle/informatik/Skripte/se3-bib.zip>

17.3 Wiederholung und Iteration, Generatoren

Aufzählen einer Folge

Problem: 87

Aufzählen und Sammeln: Gegeben sei eine Funktion f der natürlichen Zahlen.

Gesucht: Die Folge der n Bilder $\{f(0), f(1) \dots f(n-1)\}$.

Lösung: 88

Die Funktion (`build-list n f`) berechnet die Liste der n Funktionsresultate $f(0), f(1), \dots f(n-1)$.

```
> (build-list 5 (lambda (x) (* x x)))
→ '(0 1 4 9 16); Aufzählen der Quadrate
```

Wiederholung (Iteration)

Problem: 89

Iterative Anwendung einer Funktion auf das Resultat der letzten Berechnung, beispielsweise zur Berechnung von Folgen und Reihen.

Lösung: 90

Verwendung des Bausteins `iterate` oder Rekursion.

$$\text{powers_of_two} = (1, \overset{(2*)}{\underset{\curvearrowright}{2}}, \overset{(2*)}{\underset{\curvearrowright}{4}}, \overset{(2*)}{\underset{\curvearrowright}{8}}, \overset{(2*)}{\underset{\curvearrowright}{16}}, \dots)$$

Die Parameter von iterate

- **iterate** nimmt als Argumente
 - eine Funktion,
 - ein Ende-Prädikat
 - und einen Startwert.
- Die Funktion wird zunächst auf den Startwert und dann immer wieder auf das Resultat angewendet, bis das Resultat das Ende-Prädikat erfüllt.
- Das Ergebnis ist die Liste der Funktionsanwendungen.

```
(define powers-of-two
  (iterate (lambda (x) (* 2 x)) ; Iterator
           (lambda (x) (> x 64)) ; Ende Test
           1))
> powers-of-two
  → (1 2 4 8 16 32 64 128)
```

Beispiel

```
(define (naturals n)
; die Liste der natuerlichen Zahlen 1..n
  (iterate add1
           (lambda (x) (≥ x n))
           1))
> (naturals 7) → (1 2 3 4 5 6 7)
> (define (spaces n)
  (foldl
    string-append " "
    (map (lambda (x) " ")
          (naturals n)))
    ))
> (spaces 6) → "       "
> (define (spaces2 n) ; einfacher
  (make-string n \space))
```

Übungsaufgabe: Definieren Sie die Funktion (naturals n) mithilfe von build-list .

☞ Anmerkung:

Reduktor und Generator

iterate und **foldl/r** sind als Iteratoren über Listen *orthogonale* Funktionen:

- **foldl** und **foldr** sind *Reduktoren*, die eine Liste mithilfe eines Operators rekursiv zu einem Wert reduziert.
- **iterate** ist ein *Generator*, der aus einem Startwert rekursiv die Elemente einer Liste generiert.

☞ Anmerkung: **iterate** ist leider keine Standard-Funktion in Racket, so daß wir sie selbst implementieren müssen. Für das Programm wurde ein neues Sprachelement benutzt: **letrec**. Wenn eine lokale Funktion *rekursiv* ist, dann kann sie nicht mit **let** definiert werden, da sie während der Definition ja noch nicht definiert ist und daher nicht auf sich selbst Bezug nehmen darf. Mit **letrec** ist das aber zulässig. Die Hilfsfunktion **iteration** ist nötig, um im Parameter *result* die Resultat-Liste zu konstruieren. Bei jedem rekursiven Aufruf wird diese Resultat-Liste um ein Element verlängert — das Resultat wird akkumuliert. Daher heißen solche Parameter *Akkumulatoren*. Wenn die Abbruchbedingung erreicht wird, wird der Akkumulator als Resultat zurückgegeben. Vorher wird er mit **reverse** gespiegelt, weil die zuletzt berechneten Werte im Akkumulator vorne an stehen.

Implementation von **iterate**

Variante 1: rekursiv

```
(define (iterate f end? seed)
  ; apply f to seed and repeat the process
  ; on the sequence of results (f (f .. seed)))
  ; until (end? (f(f .. seed))) is satisfied.
  ; return the last value.
(letrec ([iteration
          (lambda (result nth-term)
            (let ([new-term (f nth-term)])
              (if (end? nth-term) result
                  (iteration
                    (cons new-term result)
                    new-term))))]
        (reverse
          (iteration (list seed) seed))))
```

Zusammenfassung:

Funktionen höherer Ordnung

Funktionen als Argument:

Funktionen höherer Ordnung zur Kontrollabstraktion, die den Ablauf steuern:

Applikation: apply

Abilden: map

Filtern: filter

Falten: foldl, foldr

Iteration: iterate, (gen-iterate), iter-until

Verträge: flat-contract

Generatoren und Sequenzen

Erzeugung einer Generatorfunktion: generator ist eine Funktion höherer Ordnung, die Generatorfunktionen erzeugt.

```
(require racket/generator)
(generator () body ...)
```

Der Generator läuft in einer endlosen Schleife.

- Bei jedem *yield* unterbricht er und gibt einen Wert zurück.
- Beim nächsten Aufruf setzt er an der Unterbrechungsstelle fort bis zum nächsten *yield*.

Beispiel: 91 (Natürliche Zahlen)

Ein Generator, der die natürlichen Zahlen aufzählt:

```
(define (next x)
  (yield x)
  (next (+ 1 x)))
(define nats
  (generator ()
    (next 1)))
> (nats) → 1
> (nats) → 2
> (nats) → 3
> (nats) → 4
```

2. Implementation von iterate

Variante 2: Mit Generator

Typischer für Racket ist die Implementation von **iterate** mithilfe von Generatoren und Sequenzen.

2 Schritte:

Wir definieren zwei Stützfunktionen:

fgenerator: Verwende den Konstruktor **generator**, um einen Generator für die Folge init , (f init), (f(f init)) ... zu erzeugen.

take-generator-until: Sammele die generierten Elemente in einer Liste, bis das Endeprädikat erfüllt ist (mittels in-producer, for-list)

Stützfunktion fgenerator

Variante 1: rekursiv

```
(require racket/generator)
(define (fgenerator f init)
  (generator ()
    (letrec ; lokale Funktion floop
      ([floop
        (lambda (x)
          (yield x)
          (floop (f x)))]
       (floop init))))
  > (define potenzen
      ; ein Generator fuer Zweierpotenzen
      (fgenerator (lambda (x) (* x x)) 2))
  > (potenzen) → 2
  > (potenzen) → 4
```

Stützfunktion fgenerator

Variante 2: iterativ

```
(require racket/generator)
(define (fgenerator f init)
  (generator ()
    (do ([x init (f x)])
        (#f) ; stop?-expr
```

```

        ( yield x ))))
> (define potenzen
  ; ein Generator fuer Zweierpotenzen
  (fgenerator (lambda (x) (* x x)) 2))
> (potenzen) → 2
> (potenzen) → 4

```

Sequenzen

Der Datentyp *Sequenz* (*sequence*) umfaßt alle geordneten Sammlungen von Werten, z.B:

- Listen
- Vektoren
- Hashtabellen
- Strings, ...

Über Sequenzen kann mit *for*-Schleifen iteriert werden.

```
(for ([i '(1 2 3)])
  (display i))
→ 123
```

Sequenzen können durch einen *Generator* aufgezählt werden:

(sequence->generator <sequence>)

Stützfunktion *take-generator-until*:

Sammeln der generierten Werte

In der Schleife *for/list* werden die Elemente der Sequenz nacheinander an die Variable item gebunden und in einer Liste gesammelt.

in-producer erzeugt aus seq eine Teilsequenz, die endet, wenn das erste Element von seq das Prädikat end? erfüllt.

```

(define (take-generator-until seq end?)
  (for/list
    ([item (in-producer seq end?)])
    item))
> (take-generator-until
  (sequence->generator (in-naturals 3))
  (lambda (x) (> x 7)) → '(3 4 5 6 7)

```

- (`(in-naturals 3)`) ist die Sequenz der natürlichen Zahlen ab 3,
- `sequence->generator` erzeugt aus einer Sequenz einen Generator, der die Sequenz aufzählt.

gen-iterate: Iteration mit Generatoren

```
(define (gen-iterate f end? seed)
  (let ([fgen (fgenerator f seed)])
    (take-generator-until fgen end?)))
```

; Beispiel: Liste von Einsen

```
> (gen-iterate
  (lambda (xs) (cons 1 xs)))
  (lambda (xs) (> (length xs) 3))
  '()) → '(() (1) (1 1) (1 1 1))
```

`fsequence`, `take-seq-until` und `gen-iterate` finden Sie ebenfalls in der se3-bib im tools-module.rkt.

Wiederholung II

Problem: 92

Iterative Anwendung einer Funktion auf das Resultat der letzten Berechnung,

- aber wir sind nicht an der *Folge der Werte* interessiert,
- sondern nur am *Endwert*.

Lösung: 93

Verwendung des Bausteins `iter-until` oder Rekursion.

`powers_of_two` = (1, $\overset{(2*)}{\curvearrowright}$ 2, $\overset{(2*)}{\curvearrowright}$ 4, $\overset{(2*)}{\curvearrowright}$ 8, $\overset{(2*)}{\curvearrowright}$ 16, ...)

`iter-until` in tools-module.rkt ist eine Reimplementation der Funktion `until` aus Miranda. In Racket gibt es die Funktion `until` nicht, aber in Swindle ist ein `until` implementiert, das einen Namenskonflikt bewirken würde. Das `until` in Swindle wirkt nur durch Seiteneffekte und gibt den undefinierten Wert zurück.

Rekursive Implementation von `iter-until`

```

(require racket/gui) ; fuer bell, Systemglocke

(define (iter-until f end? seed)
  (if (end? seed) seed
      (iter-until f end? (f seed)))))

(define (klingeling2 wieoft) ; Beispiel
  (iter-until
    (lambda (x) (bell) (add1 x))
    (lambda (x) (= x wieoft))
    0)
  (void))
> (klingeling 60) →

```

Zusammenfassung: Werkzeugkasten

Bisher: Funktionen höherer Ordnung zur Kontrollabstraktion, die den Ablauf steuern:

Applikation: apply

Abilden: map

Filtern: filter

Falten: reduce, reduce-right

Iteration: iterate, iter-until

Jetzt: Einschub: Spezifikation und Modularisierung

Dann: Funktionen höherer Ordnung zur Verknüpfung von Funktionen: Werkzeuge zur Konstruktion von Parametern für map, filter usw.

18 Modularer Entwurf

18.1 Module

The screenshot shows a presentation slide with the following content:

- Section title: Modularer Entwurf
- List items:
 - 14 Funktionen höherer Ordnung
 - 15 Kontrollabstraktion
 - 16 Modularer Entwurf
 - Module
 - Formale Spezifikation und modularer Entwurf

At the bottom of the slide, there is a navigation bar with icons for back, forward, search, and other presentation controls.

Module in DrRacket

Module dienen der Strukturierung eines Programms in separate, abgeschlossene Namensräume.

- In Racket ist jeder File, der mit „#lang sprache“ beginnt, automatisch ein Modul.
- Module dürfen nicht geschachtelt werden.

„Nesting ist for birds!“ Brinch-Hansen

Evaluation und Exekution

- Wenn eine Moduldeklaration evaluiert wird,
 - wird die Syntax expandiert und das Modul übersetzt,
 - aber nicht exekutiert.
- Der Modulkörper wird erst ausgeführt, wenn das Modul über *require* importiert wird.
- Jedes Modul wird höchstens einmal ausgeführt, auch wenn es mehrfach über *require* importiert wird.
- Im Modul definierte Namen sind nur lokal sichtbar; sie können mit **provide** außerhalb sichtbar gemacht werden.

provide

```
(provide <provide-spec> ...)

<provide-spec> is one of
  <identifier>
  (rename <local-identifier>
          <export-identifier>)
  (struct <struct-identifier>
          (<field-identifier> ...))
  (all-from <module-name>)
  (all-from-except <module-name><identif. >... .)
  (all-defined)
  (all-defined-except <identifier> ...)
  (prefix-all-defined <prefix-identifier>)
  (prefix-all-defined-except
    <prefix-identifier> <identifier> ...)
  (protect <provide-spec> ...)
```

The identifier form exports the (imported or defined) identifier from the module.

The (rename local-identifier export-identifier) form exports local-identifier from the module with the external name export-identifier; other modules importing from this one will see export-identifier instead of local-identifier.

The (struct struct-identifier (field-identifier ...)) form exports the names that (define-struct struct-identifier (field-identifier ...)) generates.

The (all-from module-name) form exports all of the identifiers imported from the named module, using their local names.

The (all-from-except module-name identifier ...) form is similar, except that the listed imported identifiers are not exported.

The (all-defined) form exports all of the identifiers defined (not imported) in the module.

The (all-defined-except identifier ...) form is similar, except that the listed defined identifiers are not exported.

The (prefix-all-defined prefix-identifier) and (prefix-all-defined-except prefix-identifier identifier ...) forms are like all-defined and all-defined-except, but prefix-identifier is prefixed onto each defined identifier for its external name.

The (protect provide-spec ...) form is like the sequence of individual provide-specs, but the provided identifiers are protected (see section 9.4); the provide-specs must not contain another protect form, an all-from form, or an all-from-except form, and they must not name any identifier that is imported into the providing module, instead of defined within the module.

The scope of all imported identifiers covers the entire module body, as does the scope of any identifier defined within the module body. See section 12.3.5 for additional information concerning macro-generated definitions, require declarations, and provide declarations.

An identifier can be defined by a definition or import at most once, except than an identifier can be imported multiple times if each import is from the same module.

All exports must be unique. A module body cannot contain free variables.

A module is not permitted to mutate an imported variable with set!. However, mutations to an exported variable performed by its defining module are visible to modules that import the variable.

At syntax-expansion time, expressions and definitions within a module are partially expanded, just enough to determine whether the expression is a definition, syntax definition, import, export, or a non-definition. If a partially expanded expression is a syntax definition, the syntax transformer is immediately evaluated and the syntax name is available for expanding successive expressions.

Import expressions are treated similarly, so that imported syntax is available for expansion following its import. (The ordering of syntax definitions does not affect the scope of the syntax names; a transformer for A can produce expressions containing B, while the transformer for B produces expressions containing A, regardless of the order of declarations for A and B. However, a syntactic form that produces syntax definitions must be defined before it is used.)

The begin form at the top level for a module body works like begin at the top level, so that the sub-expressions are flattened out into the module's body.

At run time, expressions and definitions are evaluated in order as they appear within the module. Accessing a (non-syntax) identifier before it is initialized signals a run-time error, just like accessing an undefined global variable.

Beispiel: tools-module.rkt

```
#lang racket
(provide ; high order functions
```

```

conjoin disjoin always never
some some? every
iter-until iterate ; recursive
gen-iterate ; with generators
fgenerator
take-generator-until
...)
(require mzlib/defmacro; fuer define-macro
  swindle/extra; fuer amb
  racket/generator; fuer sequenzen
  (only-in racket/gui bell)) ; fuer bell

```

Beispiel: fractals-module.rkt

```

#lang racket
(provide fractals-demo fractal
  snowflake-line snowflake
  snowflake-invers snowflake-both
  all-snowflakes
  quad-koch-line quadkoch
  quadkoch-invers quadkoch-both
  quadkoch-all-iterations )
(require 2htdp/image
  2htdp/universe
  lang/posn
  se3-bib/macos-module
  se3-bib/tools-module
  se3-bib/demo-gui-module
  se3-bib/my-vector-graphics)

```

Zusicherungen für den Import/Export

- Im Top-level eines Moduls können Verträge für exportierte Objekte spezifiziert werden:
- Ein *provide/contract* macht wie *provide* einen Namen außerhalb des Moduls sichtbar und spezifiziert zusätzlich Zusicherungen.
- Die Zusicherungen werden nur außerhalb des Moduls geprüft.

```

(provide/contract <p/c-item> ...)
<p/c-item> is one of
  (struct <id> ; structure name
    ((<id> <contract-expr>) ...)); Felder
  ...

```

```
(rename <id> <id> <contract-expr>)
(<id> contract-expr)
```

☞ Anmerkung: Die Form (struct <id> ...) dient zur Spezifikation und zum Export von Verbundtypen. Der Verbundtyp muß vor der provide/-contract Anweisung schon definiert sein.

Die Form (rename <id> ...) exportiert einen Bezeichner unter einem neuen Namen.

18.2 Formale Spezifikation und modularer Entwurf

Das folgende Beispiel wurde von [Bird and Wadler, 1988](#) übernommen.

Spezifikation: Quadratwurzel einer Zahl

Beispiel: 94

Wir programmieren eine Funktion (msqrt x), die die Quadratwurzel von x berechnet: Die mathematische Spezifikation für msqrt:

$$\text{msqrt}(x) \geq 0 \wedge \text{msqrt}(x)^2 = x \quad (1)$$

oder schwächer:

$$\text{msqrt}(x) \geq 0 \wedge |\text{msqrt}(x)^2 - x| < \epsilon, \epsilon > 0 \quad (2)$$

☞ Anmerkung: Wir haben die Funktion `msqrt` und nicht `sqrt` getauft, da es in Racket die Standardfunktion `sqrt` schon gibt, deren Namen wir nicht überladen wollen.

Diese Spezifikation ist noch keine *berechenbare Funktion*. Wir müssen noch einen Algorithmus finden, der die Wurzeln ermittelt.

Spezifikation und Implementation

Definition: 95 (Implementation)

- Ein *Algorithmus*, der eine gegebene Spezifikation erfüllt, wird *Implementation* der Spezifikation genannt.
- Der Vorgang des Implementierens heißt *Implementierung*.
- Generell ist stets formal zu beweisen, daß eine Implementation ihre Spezifikation erfüllt.

☞ Anmerkung:

- Die erste Variante der Spezifikation von msqrt ist sehr streng, da sie nicht die begrenzte Genauigkeit der Arithmetik für float-Zahlen berücksichtigt. Für die Irrationalzahl

$$\sqrt{2} = 1.4142135623 \dots$$

müssen unendlich viele Stellen berechnet werden, was unmöglich wäre.

Man könnte zeigen, daß ein Algorithmus die Spezifikation erfüllt, indem er sich der gewünschten Präzision immer genauer annähert, aber hier werden wir die Spezifikation abschwächen.

Wir fordern:

$$\begin{aligned}(\text{msqrt } x) &\geq 0 \\ |(\text{msqrt } x)^2 - x| &< \text{eps} \\ \text{eps} &> 0\end{aligned}$$

Beispielrechnung: $\text{eps} = 0.0001$, $x = 2$

$$\begin{aligned}1.4141 * 1.4141 &= 1.99967881 \\ 1.4142 * 1.4142 &= 1.99996164 \\ 1.4143 * 1.4143 &= 2.00024449\end{aligned}$$

$$(\text{msqrt } 2) = 1.4142$$

Das Newton-Verfahren

Newton-Verfahren

Das *Newton-Verfahren* ist iteratives ein Näherungsverfahren, das solange immer bessere Näherungswerte liefert, bis eine gewünschte Genauigkeit erreicht ist.

- Sei y_n ein Näherungswert für \sqrt{x} .
- Dann ergibt sich ein besserer Näherungswert y_{n+1} :

$$y_{n+1} = \frac{y_n + x/y_n}{2}$$

$$y_{n+1} = (y_n + x/y_n)/2$$

Mit $x = 2$ und $y_0 = x$ erhalten wir:

$$\begin{array}{lll} y_0 & = & 2 \\ y_1 & = & (2 + 2/2)/2 \\ y_2 & = & (1.5 + 2/1.5)/2 \\ y_3 & = & (1.4167 + 2/1.4167)/2 \\ \vdots & & \vdots \end{array}$$

Teilaufgaben bei der Wurzelberechnung:

Iterationsregel: (`improve y`) $\rightarrow (y + \frac{x}{y})/2$

Abbruchbedingung: (`good-enough? y`) $\rightarrow \text{abs}(y^2 - x) < \text{eps}$; $\text{eps} = 0.0003$

Wiederholschleife: `iter-until`

msqrt mit lokalen Definitionen:

```
(define (msqrt x)
  (let* ([eps 0.00001]
         [good-enough?
          (lambda (y)
            (< (abs (- (sqr y) x))
                eps))])
    [improve
     (lambda (y)
       (/ (+ y (/ x y))
          2))])
  (iter-until
   improve good-enough? x)))
> (msqrt 4)    → 2 1/10761680
```

msqrt mit inexakter Ausgabe:

```
(define (msqrt2 x)
  (let* ([eps 0.00001]
         [good-enough?
          (lambda (y)
            (< (abs (- (sqr y) x))
                eps))])
```

```

[ improve
  (lambda (y)
    (/ (+ y (/ x y))
      2))])
(exact->inexact
  (iter-until
    improve good-enough? x)))
> (msqrt2 4) → 2.00000009292 ...

```

Spezifikation mit Kontrakt

- Wir wollen die spezifizierten Eigenschaften als „*contract*“ zusichern.
- Da der Vertrag eine Beziehung zwischen dem Argument *x* und dem Resultat *out* darstellt, benötigen wir die Form $\rightarrow d$.
- Um das Prädikat *good-enough?* und die Genauigkeit *eps* auch für die Spezifikation nutzen können, dürfen diese nicht mehr lokal in *msqrt* sein.
- Wir definieren daher global:

```

(define (good? x wurzelx)
  (let ([eps 0.00001])
    (< (abs (- (sqr wurzelx) x))
        eps)))

```

Der Vertrag

```

(define/contract ; der Vertrag
  msqrt3
  (→d ([x (≥/c 0)])
    (r (and/c
      number?
      (lambda (wu) (good? wu x)))))
  (lambda(x)
    (let ([good-enough?
          (lambda (wurzelx) ; x an good? binden
            (good? x wurzelx))])
      [improve
        (lambda (wurzelx)
          (/ (+ wurzelx (/ x wurzelx))
              2))])
    (iter-until

```

```

        improve good-enough? x))))
> (msqrt3 2) → 1 169/408
> (exact->inexact (msqrt3 2)) → 1.4142156862745099

```

Den Vertrag testen

```

> (msqrt3 -1)
 ... broke the contract (->d ((x ...)) () (y ...))
on msqrt3; expected <(>/c 0)>, given: -1
> (msqrt3 0) → 0
> (msqrt3 4) → 2 1/10761680
> (exact->inexact (msqrt3 4)) → 2.0000000929222947
> (msqrt3 4.0) → 2.0000000929222947
> (msqrt3 'vier')
 ... broke the contract (->d ((x ...)) () (y ...))
on msqrt3; expected <(>/c 0)>, given: 'vier
>

```

☞ Anmerkung: Modularer Entwurf

Die Funktion msqrt ist aus drei Teilverträgen zusammengesetzt:

- good-enough? und improve werden nur lokal benötigt. Wir haben diese Hilfsfunktionen daher lokal definiert.
- Die lokalen definierten Funktionen können auf den Parameter x zugreifen, so daß sie nur einen Parameter benötigen.
- Dieser Programmierstil heißt modular, denn wir konstruieren komplexe Einheiten aus einfacheren Modulen.
- Wenn die Module gut gewählt wurden, sind solche Programme leicht zu verstehen und zu verändern.
- Eine Stärke dieser Vorgehensweise ist es auch, daß Module leicht wiederverwendet werden können.

Das werden wir an diesem Beispiel zeigen, indem wir das Programm verallgemeinern.

☞ **Anmerkung: Das allgemeine Newton-Verfahren:** Wir wollen das Newton-Verfahren in seiner allgemeinen Form zur Suche von Nullstellen (Wurzeln) einer Funktion implementieren:

Gegeben sei eine Gleichung $g(x) = a$. Gesucht seien diejenigen Werte für x , die diese Gleichung erfüllen, die Lösungen oder Wurzeln der Gleichung. Im Newton-Verfahren wird die Aufgabe umformuliert als die Suche nach den Nullstellen einer Funktion f ,

$$f(x) = g(x) - a = 0$$

die in einem Iterationsverfahren approximiert werden.

Beim Newton-Verfahren setzen wir voraus, daß die Funktion f für alle Werte von x definiert und stetig differenzierbar ist. An der als existent angenommenen Nullstelle α sei zwar $f(\alpha) = 0$, aber die Ableitung $f'(\alpha) \neq 0$.

Weiterhin wird eine Schätzung x_0 als Startwert benötigt, die genügend nahe bei α liegt.

Sind diese Voraussetzungen erfüllt, liefert das Verfahren eine Folge von Werten x_0, x_1, \dots , die gegen α konvergiert, mit der Iterationsformel:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Das Newton-Verfahren

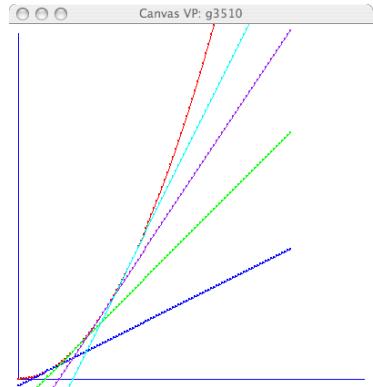
Suche nach Nullstellen einer Funktion



Sir Isaac Newton

- ▶ Hinter der Iterationsformel steckt die Annahme, daß f in der Nähe der Nullstelle durch eine Gerade, nämlich die Tangente an f im Punkt $(x_n, f(x_n))$, approximiert werden kann.
- ▶ Als neue Schätzung für die Nullstelle wird die Nullstelle der Tangente genommen.
- ▶ Diese Schnittstelle mit der x-Achse existiert nicht, wenn die Tangente parallel zur X-Achse ist ($f'(x_n) = 0$).

$$f(x) = 0.25 \times x^2 \text{ und einige Tangenten}$$



☞ Anmerkung: Die Graphik mit den Tangenten an die Funktion $f(x) = 0.25 \times x^2$ entstand mittels der bekannten Funktionen *deriv* und *graphfun*:

```
(define (tangente f x)
  ; die Tangente ist eine Gerade
  ; mit der Steigung m=(deriv f)
  ; und dem Achsenabschnitt c=(f x)-m*x
  (lambda (xx)
    (let* ([m ((deriv f) x)]
          [c (-(f x) (* m x))])
      (+ (* m xx) c)))

; Kurve mit Tangenten zeichnen
(define (zeicheTangente f x farbe)
  ;zeichne die Tangente von f an der Stelle x in farbe
  (graph-fun (tangente f x) farbe))

(define (vieleTangenten f)
  ;zeichne f und mehrere Tangenten
  (graph-fun f 'red)
  (zeicheTangente f 1.0 'blue)
  (zeicheTangente f 2.0 'green)
  (zeicheTangente f 3.0 'purple)
  (zeicheTangente f 4.0 'cyan))

(define (viertelxqua x)
  (* 0.25 x x))
(vieleTangenten viertelxqua)
```

Allgemeines Newton-Verfahren:

Wurzeln einer Funktion

Beispiel: 96

Sei y_n ein Näherungswert für die Nullstelle einer Funktion f , dann ist

$$y_n - \frac{f(y_n)}{f'(y_n)}$$

eine bessere Näherung.

Mit $f(x) = x^2 - a$ und $f'(x) = 2x$

$$y_n - \frac{f(y_n)}{f'(y_n)} = y_n - \frac{y_n^2 - a}{2y_n} = (y_n + a/y_n)/2$$

```
(define (newton f)
  ;; returns an iterator for f:
  ;; Find a root of f, start with first-guess.
  (lambda (first-guess)
    (let* ((eps 0.00001)
           (good-enough?
             (lambda (y)
               (< (abs (f y)) eps)))
           (improve
             (lambda (y)
               (- y (/ (f y)
                         ((deriv f) y))))))
      (iter-until
        improve good-enough? first-guess))))
```

☞ Anmerkung: Die Funktion newton liefert diejenige Funktion, die ausgehend von einem Startwert für das Funktionsargument von newton eine Nullstelle sucht.

Um eine Nullstelle für f zu berechnen, brauchen wir nur das Resultat von (newton f) auf einen Startwert anzuwenden:

```
((newton f) startwert)
```

Beispiele

```
(define (n-sqrt x) ;  $\sqrt{x}$ 
  ((newton      ; solving:  $y*y-x = 0.$ 
    (lambda (y)
      (- (square y) x)))
   x))
(define (cubrt x) ;  $\sqrt[3]{x}$ 
  ((newton      ; solving  $y*y*y-x=0.$ 
    (lambda (y) (- (expt y 3) x)))
   x))
(define (root x n);  $\sqrt[n]{x}$ 
  (((newton
    (lambda (y) (- (expt y n) x)))
   x)))
```

Testfälle

```
> (n-sqrt 4)           → 2.0000000944921563
> (cubrt 27)          → 3.0000000020413817
> (cubrt
  (expt 1.5 3)) → 1.500006487422155
```

☞ Anmerkung: Dieses Programm ist allgemeiner als das vorherige. Wir können jetzt für beliebige (stetig differenzierbare) Funktionen Nullstellen suchen.

- Wir müssen uns allerdings jeweils überzeugen, daß unsere Approximation der Ableitung gut genug ist.
- Wir haben ein schönes Beispiel für die Verwendung von *Funktionen höherer Ordnung*.
 - deriv ist eine Funktion, die eine Funktion und einen Wert x als Argumente erhält und an der Stelle x differenziert. deriv f ist die Ableitungsfunktion von f, deriv g die Ableitung von g.
 - newton f ist als Funktion definiert, die eine Funktion in einem Argument x als Wert liefert, die als einziges Argument noch den Startwert benötigt.

☞ Beachte: Spezifikationen: An diesem Beispiel (`sqrt`) haben wir zum ersten Mal die Leistung eines zu schreibenden Programms formal spezifiziert. Eine Spezifikation ist der erste wichtige Schritt für einen systematischen Programmertwurf. Der zweite Schritt besteht dann darin, eine Implementation zu finden, die diese Spezifikation erfüllt, und der dritte Schritt ist der Beweis, daß die Implementation korrekt ist, entweder durch einen formalen Korrektheitsbeweis oder durch systematisches Testen.

Beim funktionalen Programmieren ist es oftmals möglich, die Implementation systematisch aus der Spezifikation abzuleiten, indem die Spezifikation schematisch in ein Programm transformiert wird. Dieses ist möglich, weil funktionale Programmiersprachen sich stark an die mathematischen Formulierungen anlehnen, die für Beweise verwendet werden.

Richtiges Spezifizieren ist eine schwierige und anspruchsvolle Aufgabe. Die Kunst besteht darin, alle wesentlichen Anforderungen sehr präzise zu formulieren, aber nichts unnötiges zu fordern. Unnötige Anforderungen können dazu führen, daß womöglich gar keine Lösung für das Problem mehr existiert oder sich der Aufwand für die Berechnung immens erhöht. Ein häufiger Fehler beim Spezifizieren besteht darin, daß Bedingungen, die einem selbstverständlich erscheinen, nicht explizit gemacht werden und dann überraschenderweise in der Implementation verletzt werden. So kommt es oft dazu, daß Programme im Sinne der Spezifikation zwar völlig korrekt sind, sie aber leider doch nicht das tun, was beabsichtigt war.

Wer ein zu lösendes Problem schlecht oder zu vage spezifiziert, sollte sich nicht beklagen, wenn er mit der Antwort nichts anfangen kann. Und schon sehr schlichte Spezifikationen können erstaunlich lange Rechenprozesse auslösen, wie hier humorvoll von Adams, 1981 beschrieben . . . Außerdem können Sie hier erfahren, warum so viele Programmbeispiele immer wieder die Zahl 42 errechnen.

The Ultimate Question of Life, the Universe, and Everything ☆
There are of course many problems connected with life, of which some of the most popular are *Why are people born? Why do they die? Why do they want to spend so much of the intervening time wearing digital watches?*

Many millions of years ago a race of hyperintelligent pandimensional beings (whose physical manifestation in their own pandimensional universe is not dissimilar to our own) got so fed up with the constant bickering about the meaning of life which used to interrupt their favorite pastime of Brockian Ultra Cricket (a curious game which involved suddenly hitting people for no readily apparent reason and then running away) that they decided to sit down and solve the problem once and for all.

And to this end they built themselves a stupendous super computer which was so amazingly intelligent that even before its data banks had been connected up it had started from "I think therefore I am" and got as far as deducing the existence of rice pudding and income tax before anyone managed to turn it off.

It was the size of a small city. . . .

The subtlest of hums indicated that the massive computer was now in total active mode. After a pause it spoke to them in a voice rich, resonant and deep.

It said: "What is the great task for which I, Deep Thought, the second greatest computer in the Universe of Time and Space, have been called into existence?" . . .

"O Deep Thought computer," he said, "the task we have designed you to perform is this. We want you to tell us. . . .", he paused, "the Answer!"

"The Answer?" said Deep Thought. "the Answer to what?"

"Life!" urged Fook.

"The Universe!" said Lunkwill.

"Everything!" they said in chorus.

Deep Thought paused for a moments's reflection.

"Tricky," he said finally.

"But can you do it?"

"Yes," said Deep Thought, "I can do it."

"There is an answer?" said Fook with breathless excitement.

"A simple answer?" added Lunkwill.

"Yes," said Deep Thought. "Life, the Universe, and Everything. There is an answer. But," he added, "I'll have to think about it." . . .

The hum in the room suddenly increased as several ancillary bass driver units, mounted in sedately carved and varnished cabinet speakers around the room, cut in to give Deep Thought voice a little more power.

"All I wanted to say," bellowed the computer, "is that my circuits are now irrevocably committed to calculating the answer to the Ultimate Question of Life, the Universe, and Everything." He paused and satisfied himself that he now had everyone's attention, before continuing more quietly. "But the program will take me a little while to run."

Fook glanced impatiently at his watch. "How long?" he said.

"Seven and a half million years," said Deep Thought. . . .

Seven and a half million years later . . .

There was a moment's expectant pause while panels slowly came to life on the front of the console. Lights flashed on and off experimentally and settled down in a businesslike pattern. A soft low hum came from the communication channel.

"Good morning," said Deep Thought at last.

"Er . . . good morning, O Deep Thought," said Loonquawl nervously, "do you have . . . er, that is . . ."

"An answer for you?" interrupted Deep Thought majestically. "Yes. I have."

The two man shivered with expectancy. Their waiting had not been in vain.

"There really is one?" breathed Plouchg.

"There really is one," confirmed Deep Thought.

"To Everything? To the great Question of Life, The Universe, and Everything?"

"Yes. . . ."

"Though I don't think," added Deep Thought, "that you're going to like it."

"Doesn't matter!" said Plouchg. "We must know it! Now."

"Now?" inquired Deep Thought.

"yes! Now. . . ."

"All right," said the computer, and settled into silence again. The two man fidgeted. The tension was unbearable. "You're really not going to like it," observed Deep thought. "Tell us!"

"All right," said Deep Thought. "The answer to the Great Question. . . ."

"Yes. . . .!"

"Of Life, the Universe and Everything. . . ." said Deep Thought.

"Yes. . . .!"

"Is. . . ."

"Yes. . . . !!! . . . ?"

"Forty-two", said Deep Thought, with infinite majesty and calm.

It was a long long time before anyone spoke. Out of the corner of his eye Plouchg could see the sea of tense expectant faces down in the square outside.

"We're going to get lynched, aren't we?" he whispered.

"It was a tough assignment," said Deep Thought mildly.

"Forty-two!" yelled Loonquawl. "Is that all you've got to show for seven and a half million years of work?"

"I checked it very thoroughly," said the computer, "and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."

"But it was the Great Question! The Ultimate Question of Life, the Universe and Everything," howled Loonquawl.

"Yes," said Deep Thought with the air of one who suffers fools gladly, "but what actually *is* it?"

A slow stupefied silence crept over the men as they stared at the computer and then at each other.

"Well, you know, it's just Everything..." offered Phuochg weakly.

"Exactly!" said Deep Thought. "So once you do know what the question actually is, you'll know what the answer means."

"Oh, terrific," muttered Plouchg, flinging aside his notebook and wiping away a tiny tear.

"Look, all right, all right," said Loonqual, "can you just *tell* us the question?"

"The Ultimate Question?"

"Yes!"

"Of Life, the Universe and Everything?"

"Yes!"

Deep Thought pondered for a moment.

"Tricky," he said.

"But can you do it?" cried Loonqual.

Deep Thought pondered this for another long moment.

Finally: "No," he said firmly.

Both mans collapsed onto their chairs in despair.

"But I'll tell you who can," said Deep Thought.

They both looked up sharply. "Who? Tell us!"...

"I speak of none but the computer that is to come after me," intoned Deep Thought, his voice regaining its accustomed declamatory tones. "A computer whose merest operational parameters I am not worthy to calculate – and yet I will design it for you. A computer that can calculate the Question to the Ultimate Answer, a computer of such infinite and subtle complexity that organic life itself shall form part of its operational matrix. And you yourselves shall take on new forms and go down into the computer to navigate its ten-million year program! Yes! I shall design this computer for you. And I shall name it also unto you. And it shall be called ...the Earth."

Pouchg gaped at Deep Thought. "What a dull name," he said. ([Adams, 1981](#))

Teil VII

Funktionale Abschlüsse

19 Funktionale Abschlüsse, Verknüpfung von Funktionen

19.1 Curry-Verfahren

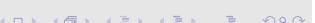
Grundlagen der funktionalen Programmierung



17 Funktionale Abschlüsse, Verknüpfung von Funktionen

- Curry-Verfahren
- Funktionskomposition
- Funktionale Abschlüsse

18 Callback functions: Closures als Ereignisroutinen



☞ Beachte: An vielen Beispielen war gut zu sehen, daß wir uns die Definition von kleinen Hilfsfunktionen ersparen können, wenn wir diese Funktion nur lokal benötigen. Wir konstruieren einfach eine geeignete Hilfsfunktion als lambda-Ausdruck. Wenn sich allerdings die Lambdas häufen, können die Ausdrücke leicht unübersichtlich werden.

Hier sind dann die Verknüpfungsfunktionen aus Dylan sehr praktisch. Einige davon sind jetzt auch Standardfunktionen in Racket; die fehlenden haben wir in Anlehnung an die Darstellung bei (Graham, 1996, Seite 110) nach Racket übertragen und Ihnen im Modul `tools-module.rkt` zur Verfügung gestellt.

Das Curry-Verfahren

Definition: 97 (Das Curry-Verfahren (currying))

Das **Curry-Verfahren** führt die Anwendung mehrstelliger Funktionen auf ihre Argumente auf eine Folge von Anwendungen von einstelligen Funktionen zurück.

- Das Verfahren ist nach dem Logiker H. B. Curry benannt, der es von ([Schönfinkel, 1977](#)) übernommen hat.
- Das Curry-Verfahren ist eine interessante Art der funktionalen Abstraktion und ein sehr flexibles Mittel zur funktionalen Programmierung.

Partielle Anwendung von Funktionen

Die Grundidee des Curry-Verfahrens ist die partielle Anwendung von Funktionen.

```
> (max 3 6) → 6  
> (max 3 8) → 8
```

Wir abstrahieren wieder strukturgleiche Terme mit der *curry*-Funktion und bilden die Funktion (curry **max** 3)

```
(define (curry f x)  
  ;; curry left arg to the function f  
  (lambda (y) (f x y)))  
> ((curry max 3) 10) → 10
```

☞ **Anmerkung:** Zur partiellen Anwendung von Funktionen Aus einer Funktion mit mehr als einem Argument können wir eine Menge von neuen Funktionen erzeugen, indem wir das erste Argument (oder die ersten Argumente) an einen Wert binden. Dieses leistet die Funktion *curry*.

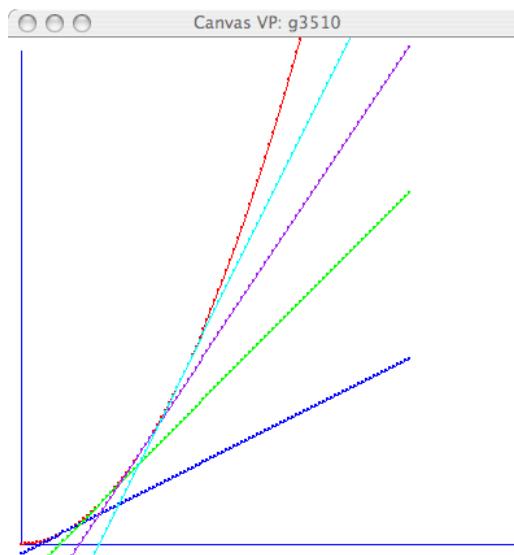
Wir binden also beispielsweise die Funktion *max* an den Aktualparameter 3 und erhalten als Resultat die Funktion (curry *max* 3). Diese Funktion hat nur noch einen formalen Parameter und berechnet das Maximum aus 3 und dem anderen Argument: ((curry **max** 3) 8) → 8.

- Das Currying als Mittel zur funktionalen Abstraktion sollten wir beherrschen, weil es ein nützliches und elegantes Ausdrucksmittel der funktionalen Programmierung ist und Spaß macht, wenn man elegant damit jonglieren kann.

Beispiele für currying

```
#lang racket
( curry + 1)      ;  $x+1$ , Inkrementfunktion
( curry / 1)       ;  $1/x$ , Reziprokwertfunktion
( curryr / 2)     ;  $x/2$ , Halbierfunktion
( curryr * 2)     ;  $x*2$ , Verdopplungsfunktion
( curryr expt 2) ;  $x^2$ , Quadrierfunktion
( curry - 0)       ;  $(-x)$ , Negation
( curry = 0)       ;  $x=0?$ 
> (map ( curry > 0) '( 1 2 4 -6 -4 1))
→ '(#f #f #f #t #t #f)
> (map ( curry * 2) '( 1 2 4 -6 -4 1))
→ '(2 4 8 -12 -8 2)
```

$$f(x) = 0.25 \times x^2 \text{ und einige Tangenten}$$



Zeichnen einer Liste von Kurven

```
(define (vieleTangenten2 f)
  ; zeichne f und mehrere Tangenten
  (graph-fun f 'red)
  (map ( curry zeicheTangente f)
    '(1 2 3 4)
    '(blue green purple cyan)) )
```

☞ **Anmerkung:** Bei mehrsteligen Funktionen ist es nützlich, wenn mehr als ein Argument auf einmal an die Funktion gebunden werden kann. Wenn die Funktion nicht kommutativ ist (beispielsweise $>$, $-$), dann möchte man auch wählen können, ob die Argumente von links oder von rechts an die formalen Parameter gebunden werden sollen.

Die Curry-Funktionen in Racket nehmen deshalb beliebig viele Argumente.

- *curry* bindet die Argumente von links nach rechts,
- *curryr* von rechts nach links.

Da wir nicht näher darauf eingehen werden, wie man Funktionen definiert, die keine festgelegte Zahl von Argumenten haben, folgen die Definitionen für die Funktionen hier ohne Kommentar.

```
(define curry
  (lambda args ;function name and args to be curried
    (let ([f (car args)]
           [curried-args (cdr args)])
      (lambda not-curried-args
        (apply f
          (append
            curried-args
            not-curried-args))))))

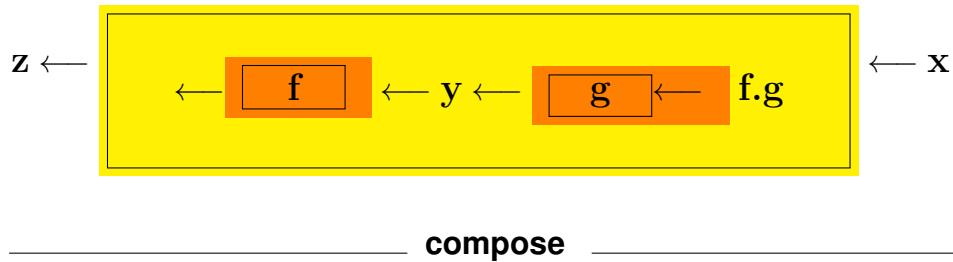
(define curryr
  ;;; curry an arbitrary number of args
  ;;; (as the last args) to a function of several args.
  (lambda args ;function name and args to be curried
    (let ([f (car args)]
           [curried-args (cdr args)])
      (lambda not-curried-args
        (apply f
          (append
            not-curried-args
            curried-args))))))
```

19.2 Funktionskomposition

Definition: 98 (*Funktionskomposition*)

Die *Funktionskomposition* $f \cdot g$ ist die Verknüpfung der beiden Funktionen f und g zu einer neuen Funktion h , die beide Funktionen nacheinander ausführt:

$$f \cdot g(x) = f(g(x))$$



compose

```
(define (compose f g)
  ;; compose two functions in one argument.
  (lambda (x) (f (g x))))  
  
(compose cdr cdr)      ; cddr
(curry compose not)     ; complement
```

compose ist *polymorph* und auf alle Paare von einstelligen Funktionen anwendbar, bei denen der Quelltyp der einen Funktion mit dem Zieltyp der anderen Funktion übereinstimmt.

Beispiele

Satz: 99 (Assoziativität von *compose*)

Funktionskomposition ist assoziativ.

$$(f \cdot g) \cdot h = f \cdot (g \cdot h)$$

Viele Definitionen lassen sich klarer schreiben; vergleiche:

Sprache: racket; *memory limit: 256 MB*
; gegeben Funktionen f_1, f_2, f_3

```
(define (f x)
  (f1 (f2 (f3 x)))) ; oder einfacher
(define f (compose f1 f2 f3))
(define (wa x) (sqrt (abs x)))
(define wu (compose sqrt abs))
```

19.3 Funktionale Abschlüsse

Zur Erinnerung: Umgebungen und closures

- Wenn wir Funktionen als Werte von Funktion zu Funktion weiterreichen, dann besteht dieser Wert nicht nur aus der textuellen Definition der Funktion, sondern auch aus einer Beschreibung der Bindungen der Namen.
- Diese Einheit nennt man *funktionalen Abschluß* (**Closure**, **funktionaler Abschluß**), siehe Definition 12.
- Ein funktionaler Abschluß ist eine Funktion zusammen mit den Namensbindungen der lokalen Umgebung, in der die Funktion definiert wurde.

Closure als Datenkapsel

Beachte:

- Funktionskomposition und Currying sind nur möglich, weil die übergebenen Funktionen im funktionalen Abschluß gespeichert werden.
- In Programmiersprachen, die keine Abschlüsse kennen, können Funktionsobjekte keine Daten kapseln, und Funktionen höherer Ordnung sind nur beschränkt nutzbar.

Beispiel

```
(compose  
  (deriv sin)  
  (deriv tan))
```

- Beim ersten Abschluß ist in der lokalen Umgebung, in der dieser erzeugt wurde, der **Formalparameter f** an den Wert **sin** gebunden,
- beim zweiten Abschluß an den Wert **tan**.
- Beide Abschlüsse kennen und behalten ihre persönliche Variable **f**, auch wenn sie eventuell in einer Umgebung ausgewertet werden, wo es andere Bindungen für den Namen **f** gibt.

Algebraische Eigenschaften von compose und curry

Satz: 100 (Distributivität von map)

Für Terme mit *compose* gelten die folgenden Zusicherungen (ohne Beweis):

```

(curry map (compose f g))
  = (compose (curry map f)) (curry map g))
(curry map f-1)
  = (curry map f)-1, falls f-1 existiert.
(map f (append xs ys))
  = (append (map f xs) (map f ys))

```

map ist also *distributiv* hinsichtlich der Funktionskomposition und der Konkatenation.

19.4 Die Dylan-Funktionen

Die Dylan-Funktionen



17 Funktionale Abschlüsse, Verknüpfung von Funktionen

- Curry-Verfahren
- Funktionskomposition
- Funktionale Abschlüsse

18 Callback functions: Closures als Ereignisroutinen

< □ > < ⌂ > < ⌃ > < ⌄ > ⌁ ⌂ ⌃ ⌄

Die Dylan-Funktionen

Konstruktoren für konstante Funktionen

Der Konstruktor `const` konstruiert eine Funktion, die beliebige Argumente stets auf einen festen Wert abbildet.

```
> (const 6)      → #<procedure:const>
> ((const 6) 1 2) → 6
> (define always (const #t))
> (always 'wie 'auch 'immer) → #t
> (define never (const #f))
> (never 'wie 'auch 'immer) → #f
```

`const` ist eine Standardfunktion in Racket, `always` und `never` finden Sie im Modul `tools-module.rkt`.

```
(define disjoin
  ;; return a composite predicate that is true
  ;; if any of the component predicates are true
  ;; all predicates must take the same number of args
  (lambda functions
    (lambda args
      (ormap
        (lambda (f)
          (apply f args))
        functions))))
```



```
(define conjoin
  ;; return a composite predicate that is true
  ;; if all of the component predicates are true
  ;; all predicates must take the same number of args
  (lambda functions
    (lambda args
      (andmap
        (lambda (f)
          (apply f args))
        functions))))
```

Weitere nützliche Verknüpfungen von Funktionen

conjoin: Die Argumente der Funktion sind Prädikate. Das Resultat ist das Prädikat, das wahr ist, wenn *alle* Prädikate auf den Argumenten der Funktion erfüllt sind.

disjoin: Die Argumente der Funktion sind Prädikate. Das Resultat ist das Prädikat, das wahr ist, wenn *mindestens eins* der Prädikate auf den Argumenten der Funktion erfüllt ist.

Beispiele

```
> ((conjoin
    (curryr > 3)
    (curryr < 10)) 6) → #t
> (map (const 1) '(1 2 3 abc a b))
   → (1 1 1 1 1)
> (foldl + 0
    (map (const 1)
      '(1 2 3 abc a b)))
   → 6 ; length
```

Weitere Funktionen in Racket

Komplement: Berechne das inverse Prädikat zu einem Prädikat:

(**negate** <predicate?>) → <procedure>

Suche Element: Suche ein Element in einer Liste, das ein bestimmtes Prädikat erfüllt, #f sonst. (**findf** <predicate?> <list>) → any/c

Suche Restliste: Erfüllt *mindestens ein* Element der Liste das Prädikat? Wenn ja, gib die Restliste zurück.

(**memf** <predicate?> <list>) → (or/c list? #f)

Allquantor: Erfüllen *alle* Elemente der Liste das Prädikat?

(**andmap** <predicate?> <list> ...) → <boolean>

Existenzquantor: Erfüllt *mindestens ein* Element der Liste das Prädikat? (**ormap** proc lst ...+) –? boolean?

☞ **Anmerkung:** memf ist die Common Lisp Funktion **some**, andmap die Common Lisp Funktion **every**.

20 Callback functions: Closures als Ereignisroutinen

20.1 Graphische Bedienoberflächen

Call back functions:
Funktionale Abschlüsse als Ereignisroutinen

17 Funktionale Abschlüsse, Verknüpfung von Funktionen

18 Callback functions: Closures als Ereignisroutinen

- Graphische Bedienoberflächen
- Das teachpack gui.rkt
- Animation und Spiele

Graphische Bedienoberflächen (gui)

- Bei der Programmierung von Bedienoberflächen müssen die Aktionen spezifiziert werden, die durch Interaktionsereignisse ausgelöst werden:
 - Mausbewegungen
 - Knopfdruck
 - Texteingabe usw.
- In Java werden für diesen Zweck Objekte der Klassen „Actionhandler“ oder „Eventhandler“ an die Bedienelemente gebunden.
- In Racket und Lisp werden *callback functions* als Ereignisroutinen an die Bedienelemente gebunden.

Das Teachpack „gui.rkt“: Fenster

```
(require htdp/gui)
(create-window {{{<gui-item>}}})
```

→ <window>

; Erzeugen eines Fensters mit Bedienelementen
; Jede Unterliste ({<gui-item>}) definiert
; eine Zeile von Bedienelementen
(**show-window** <window>) → #t
; zeigt ein Fenster-Objekt an
(**hide-window** <window>) → #t
; Ausblenden des Fensters

Knöpfe (buttons)

(**make-button** <string> <event->boolean>)
→ <gui-item>

- Erzeugt einen Knopf (button) mit der Aufschrift <string>.
- Das Prädikat <event->boolean> dient als *eventhandler* und wird aufgerufen, wenn der Knopf gedrückt wird.

Textanzeigen

(**make-message** <string>) → <gui-item>

- Anzeige einer Nachricht

(**draw-message** <gui-item> <string>) → true

- Anzeige einer Nachricht in einem message-item, löscht die aktuelle Nachricht.

Eingabefelder für Text

(**make-text** <string>) → <gui-item>

- Ein Eingabefeld für Text mit Beschriftung <string>

(**text-contents** <gui-item>) → <string>

- liest den aktuellen Inhalt eines Texteingabefeldes aus.

Auswahlmenü

(**make-choice** '({<string>})) → <gui-item>

- Ein Auswahlmenü; die strings <string> beschreiben Alternativen

(**choice-index** <gui-item>) → <number>

- Auslesen des Index der ausgewählten Alternative, die Zählung beginnt bei „0“.

Beispiel: Ein Fenster mit QUIT-Knopf

The screenshot shows a window titled "Beispiel". Inside the window, there is a code snippet:

```
window w
  (create-window
    (list
      (list
        (make-button
          "QUIT"
          (lambda (e)
            (hide-window w)))))))
```

To the left of the code, there is a small icon of a window with three circles and a "QUIT" button. Below the code, the text "window w" is displayed.

On the right side of the window, there is a green box containing the following text:

Konstruiere ein Fenster mit einem Knopf:

- ▶ Drücken des Knopfes schließt das Fenster;
- ▶ (show–window w) zeigt es wieder an.

At the bottom of the window, there is a status bar with the following information:

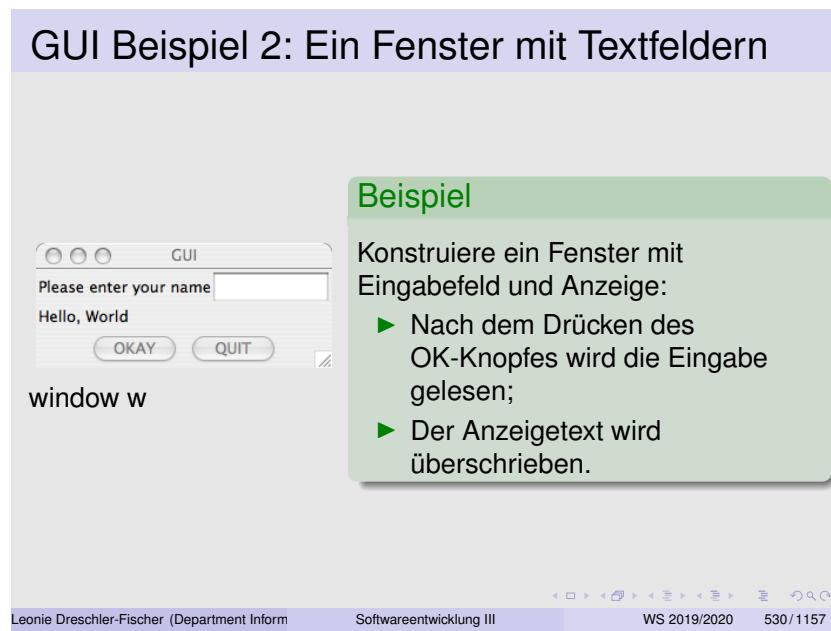
Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 528 / 1157

Ein Fenster mit Quit-Button

```
(require htdp/gui)
(define w
  (create-window
    (list
      (list
        (make-button
          "QUIT"
          (lambda (e)
            (hide-window w)))))))

(show–window w); Anzeige von w
```

GUI Beispiel 2: Ein Fenster mit Textfeldern



Das Programm

```
(define text1
  (make-text "Please_enter_your_name"))
(define msg1
  (make-message
    (string-append "Hello ,_World"
      (make-string 33 #\SPACE))))
```

20.2 Das teachpack gui.rkt

The teachpack gui.rkt provides the following operations:

- * `show-window` : `window` → `true`
to show the window
- * `hide-window` : `window` → `true`
to hide the window
- * `create-window` : (`listof` (`listof` `gui-item`)) → `window`
to add gui-items to a window and to show the window each list of gui-items defines one row of gui items in the window
 - * `make-button` : `str` (`event%` → `boolean`) → `gui-item`
to create a button with label and call-back function
 - * `make-message` : `str` → `gui-item`
to create an item that displays a message
 - * `draw-message` : `gui-item`[`message%`] `stri` → `true`
to display a message in a message item it erases the current message
 - * `make-text` : `str` → `gui-item`
to create an item (with label) that allows users to enter text
 - * `text-contents` : `gui-item`[`text%`] → `str`
to determine the contents of a text field
 - * `make-choice` : (`listof` `str`) → `gui-item`
to create a choice menu that permits users to choose from some string alternatives
 - * `choice-index` : `gui-item`[`choice%`] → `num`
to determine which choice is currently selected in a choice-item the result is the 0-based index in the choice menu

Example 1:

```
> (define w
  (create-window
    (list (list (make-button
      "QUIT"
      (lambda (e)
        (hide-window w)))))))
```

A button appears on the screen. Click on the button and it will disappear.

```
> (show-window w)
```

The frame reappears.

Example 2:

```

; text1 : GUI-ITEM
(define text1
  (make-text "Please_enter_your_name"))

; msg1 : GUI-ITEM
(define msg1
  (make-message
    (string-append "Hello,_World"
                  (make-string 33 #\SPACE)))))

; Event -> true

```

; draws the current contents of text1 into msg1, prepended with "Hello,"

```

(define (respond e)
  (draw-message
    msg1
    (string-append "Hello,_"
                  (text-contents text1))))

```

; set up window with three lines": a text field, a message, and two buttons ; fill in text and click OKAY

```

(define w
  (create-window
    (list
      (list text1)
      (list msg1)
      (list (make-button "OKAY" respond)
            (make-button
              "QUIT"
              (lambda (e) (hide-window w)))))))

```

20.3 Animation und Spiele



The screenshot shows a DrScheme window with a digital clock. The clock has blue numbers from 1 to 12 and blue hands. It is set to approximately 10:10. The window title is "DrScheme". To the right of the window, there is a list of topics:

- 17 Funktionale Abschlüsse, Verknüpfung von Funktionen
- 18 Callback functions: Closures als Ereignisroutinen
 - Graphische Bedienoberflächen
 - Das teachpack gui.rkt
 - Animation und Spiele

At the bottom of the slide, there is footer information: Leonie Dreschler-Fischer (Department Inform), Softwareentwicklung III, WS 2019/2020, 532/1157.

„World“-Programs: universe.rkt

Ein einfacher Rahmen für Animation und Interaktion

Für Spiele oder Animationen benötigen wir:

- Die Möglichkeit, regelmäßig in Abhängigkeit von einer Simulationsuhr eine Aktion auszulösen und den Zustand einer simulierten Welt zu aktualisieren,
- Eine simulierte Welt grafisch darzustellen und bei jeder Zustandsänderung neu zu zeichnen,
- Auf interaktive Eingabe (Maus, Tastatur) zu reagieren.

Das teachpack universe.rkt

Im teachpack universe.rkt werden die Zustandsänderungen über **closures** als Ereignisroutinen (call back functions) modelliert.

Eine gute Einführung finden Sie unter <http://world.cs.brown.edu/1/htdw-v1.pdf>

Zeitabhängige Animation

Für den einfachsten Fall einer Animation müssen wir abhängig von der Uhrzeit zu jedem Zeitpunkt ein Bild der Welt erzeugen und anzeigen.

Die Funktion *animate* nimmt einen Parameter:

create-image: eine call-back Funktion, die den Zustand der Welt, gegeben als natürliche Zahl, auf ein Bild vom Typ `scene?` abbildet.

animate erzeugt einen canvas und startet eine Simulationsuhr, die 28 mal pro Sekunde tickt.

Bei jedem Tick wird

- der Zeitpunkt um eins erhöht,
- die call-back Funktion „`create-image`“ aufgerufen und das nächste Bild berechnet und angezeigt.

Die Simulation läuft, bis das Programm mit der Stop-Taste abgebrochen oder das Fenster geschlossen wird.

Das folgende Beispiel stammt aus ([Flatt and PLT, 2010](#)).



Beispiel: Animation eines UFOs

```
#lang racket
(require 2htdp/image
          2htdp/universe)
(define (create-UFO-scene height)
  (underlay/xy
    (rectangle 100 100 "solid" "white")
    50 height UFO))
(define UFO
  (underlay/align
    "center"
    "center"
    (circle 10 "solid" "green")
    (rectangle 40 4 "solid" "green")))
(animate create-UFO-scene)
```

Beispiel 2: Eine analoge Uhr

Beispiel: 101 (Eine Uhr mit Zeigern:)

Wir schreiben ein Programm, das eine Uhr mit Zeigern zeichnet und jede Sekunde die Uhr mit aktueller Zeit neu zeichnet.

Teilaufgaben:

- Stelle die Uhrzeit fest (`seconds->date (current-seconds)`)
- Zeichne das Zifferblatt und die Zeiger

Die Systemuhr

- Die Funktion `current-seconds` gibt die Systemzeit in Sekunden.
- `seconds->date` decodiert die Systemzeit zu einem Verbund (struct), der das Datum und die Stunden, Minuten und Sekunden der aktuellen Zeit angibt.

```
(define (next-clock-world world )  
  ; next-clock-world: any → date  
  (seconds->date  
   (current-seconds)))
```

Das Zeichnen der Uhr: Das Ziffernblatt

- In DrRacket können Bilder als Werte direkt in den Programmtext eingefügt werden:
special menu → *insert image*.

```
(require 2htdp/image)
```



```
(define *clock-face* )  
(define *clock-canvas-w*  
  (image-width *clock-face*))  
; the width of the canvas  
(define *clock-canvas-h*  
  (image-height *clock-face*))  
; height of canvas
```

Zeichnen einer Szene

- Eine Szene wird gezeichnet, indem mit *empty-scene* eine leere Szene erzeugt wird.
- *place-image* komponiert Bilder, indem es einer Szene ein anderes Bild hinzufügt.
Das Resultat ist das zusammengesetzte Bild.
- Wir zeichnen die Uhr, indem wir dem Bild des Ziffernblattes drei Geraden als Zeiger hinzufügen.

Die Zeiger

```
;place-hand:  
;           frac-of-cycle len colo canvas → Image  
(define (hour-hand hour minute canvas)  
        ; real → Scene  
        (place-hand (/ (+ hour (/ minute 60))  
                     12) 0.6 'blue canvas))  
(define (minute-hand minutes seconds canvas)  
        ; real → Scene  
        (place-hand (/ (+ minutes (/ seconds 60))  
                     60) 1 'blue canvas))  
(define (seconds-hand seconds canvas)  
        ; real → Scene  
        (place-hand (/ seconds 60)  
                     1.0 'red canvas))
```

☞ Anmerkung: Die Funktion *place-hand* berechnet das Bild eines Zeigers als Gerade vom Mittelpunkt des Ziffernblattes aus und fügt dieses der Zeichenfläche *canvas* zu. Die Argumente sind:

- *frac-of-cycle*: Der Bruchteil des vollen Zeigerumlaufs, beginnend bei der 12 Uhr-Position, im Uhrzeigersinn gezählt.
- *len*: Ein Längenfaktor relativ zur Länge des Minutenzeigers.
- *colo*: Die Farbe
- *canvas*: Die Szene, der der Zeiger hinzugefügt werden soll.

Beachten Sie, daß das Koordinatensystem kein kartesisches System ist. Die y-Koordinaten haben ihren Ursprung am oberen Bildrand und wachsen nach unten.

```
(define (place-hand frac-of-cycle len colo canvas)
; draw a hand of the clock
; number: the fraction of the 12 hour cycle
; number: the length relative to the longest hand
; color: the color
(let* ((y (* len 0.4 *clock-canvas-h*))
        (angRad (- (* frac-of-cycle 2 pi))))
        ;x'=x*cos(angle)-y*sin(angle)
        ;y'=y*cos(angle)+x*sin(angle)
        (xn (- (* y (sin angRad)))) ; rotated x at the origin
        (yn (* y (cos angRad))) ; rotated y at the origin
        (xm (/ *clock-canvas-w* 2))
        (ym (/ *clock-canvas-h* 2)))
; y-axis points down, not up!
(add-line canvas xm ym (+ xn xm) (- yn xm) colo)
))
```

animate: Ereignisroutine „draw-clockworld“

```
(define (draw-clockworld world)
; draw-clockworld: date → Image
(let* ([world (seconds->date
                (current-seconds))]
        [h (image-height *clock-face*)]
        [w (image-width *clock-face*)]
        [canvas
         (empty-scene
          *clock-canvas-w*
          *clock-canvas-h*)]
        [hour (date-hour world)]
        [min (date-minute world)]
        [sec (date-second world)])
  (display (list hour ":" min ":" sec
            (zonenkuerzel world) "\n") ....
```

Die Funktion Zonenkuerzel berechnet die die Kennung der Zeitzone, MEZ usw.

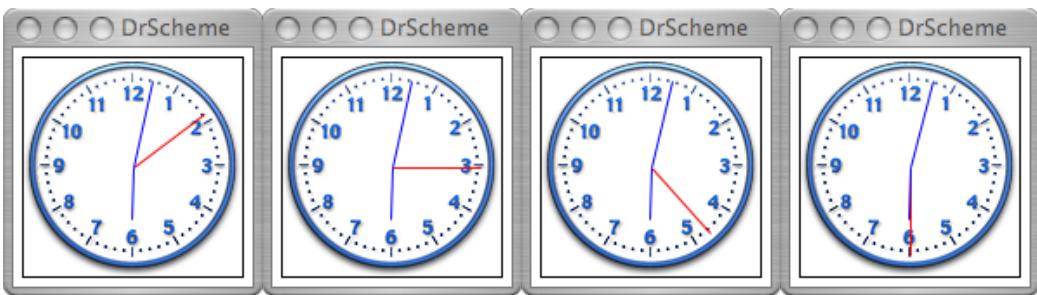
```
(hour-hand
hour min
(minute-hand
min sec
(seconds-hand
```

```

sec
(place-image
 *clock-face*
 (/ *clock-canvas-w* 2)
 (-(/ *clock-canvas-h* 2) 2)
 canvas)))) )
(define (animate-clock)
 (animate draw-clockworld))

```

Bildschirmschnappschüsse: ein Beispielauf



```

(define (place-hand frac-of-cycle len colo canvas)
; drwa a hand of the clock
; number: the fraction of the 12 hour cycle
; number: the length relative to the longest hand
; color: the color
(let* ((y (* len 0.4 *clock-canvas-h*))
      (angRad (- (* frac-of-cycle 2 pi)))
      ;x'=x*cos(angle)-y*sin(angle)
      ;y'=y*cos(angle)+x*sin(angle)
      (xn (- (* y (sin angRad)))) ; rotated x at the origin
      (yn (* y (cos angRad))) ; rotated y at the origin
      (xm (/ *clock-canvas-w* 2))
      (ym (/ *clock-canvas-h* 2)))
; y-axis points down, not up!
(add-line canvas xm ym (+ xn xm) (- ym yn ) colo)
))

(define (hour-hand hour minute canvas)
; real → Scene
(place-hand (/ (+ hour (/ minute 60))
12) 0.6 'blue canvas))

```

```

(define (minute-hand minutes seconds canvas)
; real → Scene
(place-hand (/ (+ minutes (/ seconds 60))
60) 1 'blue canvas))

(define (seconds-hand seconds canvas)
; real → Scene
(place-hand (/ seconds 60)
1.0 'red canvas))

(define *zeitzonensStandard* ; Strings der Standardzeiten
(vector
"GMT" ; 0 , "UTC", "WET" Island, Marokko, Kanaren, Senegal ...
"MEZ" ; 1
"EET" ; 2 Türkei, Israel, Ägypten...
"MSK" ; 3 Moskau
"SAMT" ; 4 Samara
...
"AT" ; -1 Acores
))
(define *zeitzonensSommer*
(vector ; Strings fuer Sommerzeiten
"AST" ; -1 Acores
"WEST" ; 0 , "UTC", "WET" Island, Marokko, Kanaren, Senegal ...
"MESZ" ; 1 Deutschland, Frankreich
"EEST" ; 2 Türkei, Israel, Ägypten...
...
"ARST" ; -3 Argentinien, Brasilien, Uruguay, Franz. Guayana
"FNST" ; -2 Fernando de Noronha
))
(define (zonenkuerzel datum)
(let* ([offset (/ (date-time-zone-offset datum) 3600)]
[sommerzeit? (date-dst? datum)]
[tabpos (if (negative? offset)
(+ offset 24) offset)])
; Zeitzonenumfang auf das Intervall 0..24 normieren
(if (integer? tabpos)
(if sommerzeit?
(vector-ref *zeitzonensSommer* tabpos)

```

```

        (vector-ref *zeitzonenStandard* tabpos))
  (format "GMT+~a" offset))
))

```

Das Programm finden Sie in se3-bib/Beispiele/kapitel6Clock.rkt.

Beispiel 2: Mit Interaktion

big-bang: Der Urknall

Die Funktion *big-bang*

- initialisiert eine simulierte Welt,
- erzeugt ein Fenster, in dem der Zustand der Welt grafisch angezeigt werden kann,
- startet eine Simulationsuhr
- und verwaltet *Ereignisroutinen* für die Interaktion und Simulationsergebnisse:

on-tick: Ereignisroutine für Uhrereignisse

to-draw: Ereignisroutine zum Zeichnen der Welt

stop-when: Prädikat für den Abbruch

on-key: Ereignisroutine für Tastaturereignisse

(big-bang <width> <height> <n> <w>) → #t

- *on-tick*: Dieser Funktion übergeben wir eine Ereignisroutine, die bei jedem Uhr-Ereignis aufgerufen wird.

Die Ereignisroutine erhält das aktuelle Datenobjekt Welt als Argument und errechnet das nächste Weltobjekt.

- *on-key*: Dieser Funktion übergeben wir eine Ereignisroutine, die bei Tastaturereignissen aufgerufen wird.

Als key-event bekommen wir entweder das eingegebene Tastaturzeichen gemeldet oder ein Symbol, wie 'button-down.

- *to-draw*: Diese Funktion erhält eine Ereignisroutine, die die aktuelle Welt grafisch ausgibt.

Eine analoge Uhr mittel Eventhandlern

- Die Ereignisroutine für *on-tick* muß bei jedem Uhr-Ereignis die Uhrzeit abfragen.
- Die Ereignisroutine für *to-draw* muß bei jedem Uhr-Ereignis die Zeiger der Uhr neu zeichnen.
- Die Ereignisroutine für *on-key* werden wir nutzen, um die Uhr anzuhalten.

stop-when beendet die Simulation, wenn die aktuelle Welt das übergebene Prädikat erfüllt.

Der Ablaufrahmen

```
(require 2htdp/universe)
(define (sim-clock tick-rate)
  ; tick: time between clock ticks (seconds)
  (let ((first-clock
         (next-clock-world #t)))
    (display "Type 'q' or escape to stop the clock.\n")
    (big-bang
      first-clock
      (on-tick next-clock-world tick-rate)
      (to-draw draw-clockworld
                *clock-canvas-w*
                *clock-canvas-h*)
      (stop-when clock-stopped?))
    (on-key handle-key)
    (record? #t)
    (name "Clock"))
  )))
```

Abbruch der Simulation Eine spezielle Welt `*last-clock-world*` signalisiert das Ende aller Zeiten.

```
(define *last-clock-world*
  (seconds->date 2000000000))
; return a special date to signal the end of time

(define (clock-stopped? world)
  (equal? world *last-clock-world*))

(define (handle-key world key)
  ; Abbruch der Simulation mit "q"
  (if (or (string=? key "escape")
          (string=? key "q"))
      (begin (display "escape_or_q\n")
             (stop-with *last-clock-world*))
      world))
```

Real Time and Date ³

(current-seconds) returns the current time in seconds. This time is always an exact integer based on a platform-specific starting date (with a platform-specific minimum and maximum value).

The value of (current-seconds) increases as time passes (increasing by 1 for each second that passes). The current time in seconds can be compared with a time returned by file-or-directory-modify-seconds (see section 11.3.3).

(seconds->date secs-n) takes secs-n, a platform-specific time in seconds (an exact integer) returned by current-seconds or file-or-directory-modify-seconds, and returns an instance of the date structure type, as described below. If secs-n is too small or large, the exn:fail exception is raised.

The value returned by current-seconds or file-or-directory-modify-seconds is not portable among platforms. Convert a time in seconds using seconds->date when portability is needed.

The date structure type has the following fields:

- * second : 0 to 61 (60 and 61 are for unusual leap-seconds)
- * minute : 0 to 59
- * hour : 0 to 23
- * day : 1 to 31

³Aus dem Helpdesk: PLT-MzScheme Language Manual

* month : 1 to 12
 * year : e.g., 1996
 * week-day : 0 (Sunday) to 6 (Saturday)
 * year-day : 0 to 365 (364 in non-leap years)
 * dst? : #t (daylight savings time) or #f
 * time-zone-offset : the number of seconds east of GMT for this time zone (e.g., Pacific Standard Time is -28800), an exact integer 52

The date structure type is transparent to all inspectors (see section 4.5).

Theodor Storms Maikatzen

Beispiel: 102 (Bevölkerungswachstum)

Eine Simulation des exponentiellen Wachstums. Wir nehmen an, daß im Mittel jede Katze pro Jahr eine feste Zahl von Nachkommen hat.

- Wir beginnen mit einer Anfangspopulation.
- Bei jedem Tick der Uhr wird die neue Population in Abhängigkeit von der alten Population neu berechnet und gezeichnet.

Maikatzen: Der Simulationsrahmen

```
(define (maikatzen-sim tick)
  ; tick: time between clock ticks (seconds)
  (big-bang
    (make-cat-universe 1 1) ; year 1, one cat
    (on-tick next-generation tick)
    (to-draw show-cat-world
      *cat-canvas-w* *cat-canvas-h*)
    (stop-when last-cat-world?)
    (on-key handle-key)
    (name "Maikatzen")
  ))
```

Die Simulationswelt

```
(define-struct cat-universe
  (year num-cats))

(define kittens-per-year 4)
; four kittens per cat per year

(define (next-generation world)
  ; next-generation: cat-universe -> cat-universe
```

```
(let ([ new-world
      (make-cat-universe
        (add1 (cat-universe-year world))
        (* (add1 kittens-per-year)
            (cat-universe-num-cats world))))]
  new-world))
```

Anzeige der Welt

```
(define (show-cat-world world)
; show-cat-world: cat-universe → Scene
(display (list
  "year:_"
  (cat-universe-year world)
  "number_of_cats:_"
  (cat-universe-num-cats world )
  "\n"))
(draw-cats *katze-winzig*
  (cat-universe-num-cats world ))
)
```

Die Zeichenfläche

```
(define cat-canvas-w 700)
; the width of the canvas
(define cat-canvas-h 500)
; the height of canvas
```



```
(define katze-winzig )
```

```
(define (draw-cats picture num-cats)
; draw-cats: Image natural → Scene
(let ([h (image-height picture)]
[w (image-width picture)])
[initial-canvas
(empty-scene
*cat-canvas-w*
*cat-canvas-h*)])
(letrec
([cats-at ; returns a new canvas
(lambda (r c n canvas) ; row column n-cats
```

```

(cond [(= n 0) canvas]
      [(> r *cat-canvas-h*) ; canvas full
       (display "too_many_cats")
       canvas]
      [(> c *cat-canvas-w*)
       ; row full, start next row
       (cats-at (+ r h) (/ w 2) n canvas)]
      [else
       (cats-at
        r (+ c w) (- n 1)
        (place-image
         picture c r canvas)
        ))])
)
(cats-at
 (/ h 2) (/ w 2) num-cats
 initial-canvas)
) ))

```

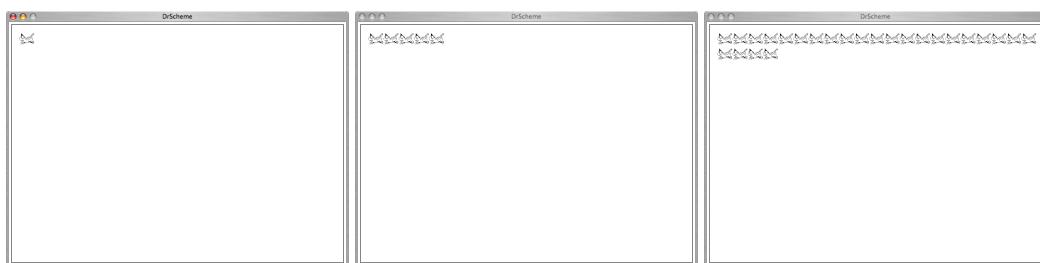
Ein Beispiellauf: Die Textausgabe

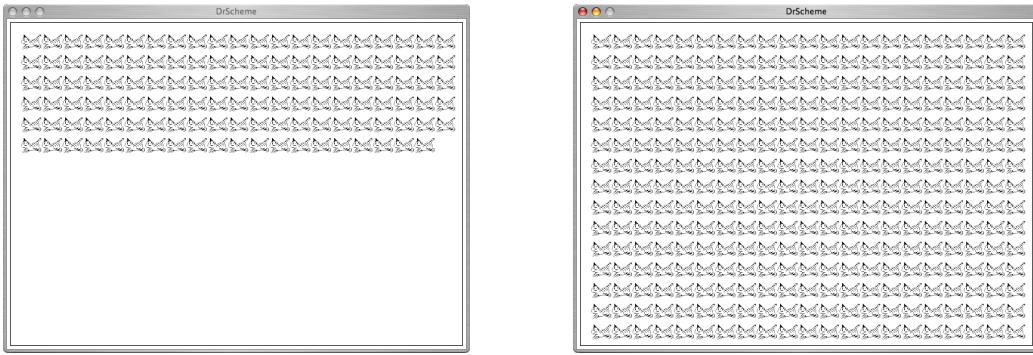
```

> (maikatzen-sim 2) →#t
(year: 1 number of cats: 1)
(year: 2 number of cats: 5)
(year: 3 number of cats: 25)
(year: 4 number of cats: 125)
(year: 5 number of cats: 625)
end of time: too many cats

```

Ein Beispiellauf: Die grafische Anzeige





Zusammenfassung:

Funktionen höherer Ordnung

Funktionen als Argument:

Funktionen höhererer Ordnung zur Kontrollabstraktion, die den Ablauf steuern:

Applikation: apply

Abilden: map

Filtern: filter

Falten: foldl, foldr

Iteration: iterate, (gen-iterate), iter-until

Verträge: flat-contract

Zusammenfassung:

Funktionen höherer Ordnung

Funktionen als Resultat:

Funktionen höherer Ordnung zur Verknüpfung von Funktionen:

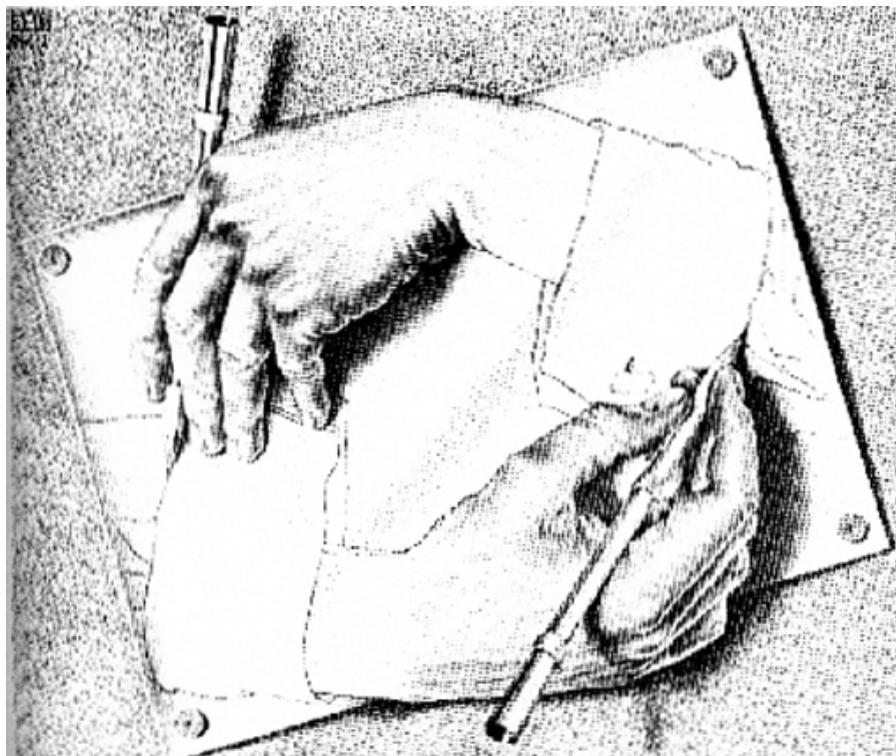
- curry, curryr, compose, conjoin, disjoin
- deriv, newton

Funktionen als call back:

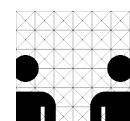
make-button, animate, big-bang

Softwareentwicklung III

Leonie Dreschler-Fischer
Prüfungsunterlagen Band 2:
Algorithmen und Fallstudien



Universität Hamburg
Fachbereich Informatik
WS 2019/2020

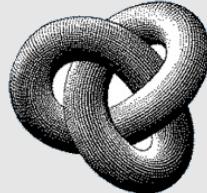


Teil VIII

Ausgewählte funktionale Algorithmen

21 Kombinatorische Probleme

Kombinatorische Probleme



M.C. Escher

- 19 Kombinatorische Probleme
 - Die Liste aller Unterlisten
 - Permutationen
 - Das Rucksackproblem
- 20 Rückzugsverfahren: Backtracking
- 21 Nichtdeterminismus

Navigation icons: back, forward, search, etc.

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 559 / 1157

Kombinatorische Funktionen



Kombinatorische Probleme lassen sich elegant rekursiv lösen:

Unterlisten: Zähle die Unterlisten einer Liste auf.
Permutationen: Zähle die Permutationen einer Liste auf.
Zusammensetzungen: Wieviele Möglichkeiten gibt es,

- ▶ einen Geldbetrag in kleinere Einheiten zu wechseln,
- ▶ einen Rucksack zu füllen?

Navigation icons: back, forward, search, etc.

21.1 Die Liste aller Unterlisten

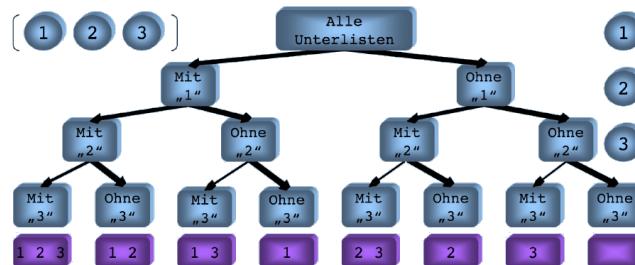
Die Liste aller Unterlisten einer Liste

Die Liste aller Unterlisten entspricht der *Potenzmenge* der Menge der Listenelemente, wenn die Liste als Menge interpretiert wird.

Die Liste aller Unterlisten von xs enthält:

- Alle Unterlisten, die mit *(car xs)* beginnen und deren *cdr* eine Unterliste von *(cdr xs)* ist.
- Alle Unterlisten, die *(car xs)* nicht enthalten und deren *cdr* eine Unterliste von *(cdr xs)* ist.

Die Liste aller Unterlisten (Potenzmenge)



Die Liste aller Unterlisten: *subs*

Beispiel: 103 (subs:)

Eine Baumrekursion über Kopf und Rumpf der Liste.

```
; (length (subs xs)) = (expt 2 (length xs))
(define (subs xs)
; Die Liste aller Unterlisten von xs
(if (null? xs) '()
  (let ([head (car xs)]
        [tail (cdr xs)])
    (append (subs tail)
            (map (curry cons head)
                 (subs tail))))))
```

```
> (subs '(1 2 3)) →  
(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))  
> (subs '()) → ()  
> (subs '(())) → (() ()))
```

Die Funktion **subs** und die weiteren kombinatorischen Funktionen finden Sie in der se3-bib im Modul kombinatorik-module.rkt.

21.2 Permutationen

Verweben (interleave)

Beispiel: 104 (Verweben:)

(*interleave x ys*) berechnet die Liste aller Möglichkeiten, ein Element *x* in die Liste *ys* einzufügen.

- Wenn *ys* nicht leer ist,
 - dann wird erstens *x* an den Anfang von *ys* gesetzt,
 - und es werden weiterhin alle Möglichkeiten berechnet, *y* mit dem Rest von *xs* zu verweben.

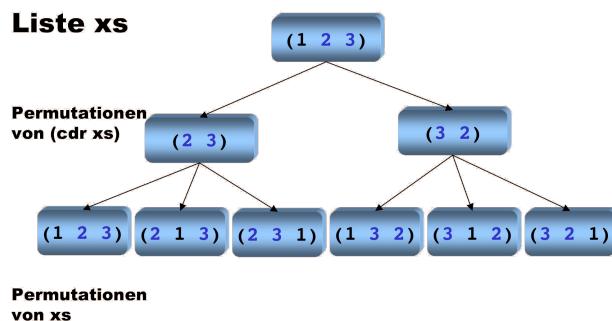
Der Kopf von *y* wird vor jeden verwebten Rest gesetzt.

Verweben (interleave)

```
(define (interleave x ys)
; Die Liste aller Einfuegemoeglichkeiten
; von x in ys
(if (null? ys) (list (list x))
  (let ([head (car ys)]
        [tail (cdr ys)])
    (append (list (cons x ys))
            (map (curry cons head)
                  (interleave x tail))))))
> (interleave 'E '(a b c))
→ ((E a b c) (a E b c)
    (a b E c) (a b c E))
```

Permutationen einer Liste

- Eine Permutation einer Liste enthält alle Elemente der Liste, aber in beliebiger Anordnung.
- Um alle *Permutationen* einer Liste zu errechnen,
 - verwebe den Kopf der Liste
 - mit allen *Permutationen des Restes* der Liste.



Permutationen einer Liste

Beispiel: 105 (Permutationen ohne Wiederholung)

perms xs ist die Liste aller Permutationen der Elemente der Liste *xs* (ohne Wiederholung).

```
(define (perms xs)
  (if (null? xs) '()
      (apply append
             (map (curry interleave (car xs))
                  (perms (cdr xs)))))))
> (perms '(H 2 3)) →
    ((H 2 3) (2 H 3) (2 3 H)
     (H 3 2) (3 H 2) (3 2 H))
```

21.3 Das Rucksackproblem

Das Rucksackproblem



- ▶ Gegeben sei eine Menge von Objekten:
- ▶ wieviele Möglichkeiten gibt es, diese so zu kombinieren, daß eine bestimmte Bedingung erfüllt wird?

Beispielsweise:

- ▶ Einen Rucksack so zu packen, daß ein Grenzgewicht nicht überschritten wird,
- ▶ oder einen Geldschein in Kleingeld zu wechseln?

Alle Möglichkeiten, einen Geldbetrag zu wechseln

Beispiel (Geld wechseln)

Beispiel: Wechsle einen Dollar in:



- ▶ half-dollars (50 c)
- ▶ quarters (25 c),
- ▶ dimes (10 c),
- ▶ nickels (5 c),
- ▶ pennies (1 c).

Rekursive Lösung:

Um einen *Geldbetrag* a zu wechseln, wenn n Münzsorten zur Verfügung stehen, gibt es die folgende Anzahl von Möglichkeiten: (Abelson and Sussman, 1985)

- Die Anzahl aller Möglichkeiten, den *Betrag* a zu wechseln, wobei *alle Münzsorten außer der ersten* verwendet werden, plus
- der Anzahl der Möglichkeiten, den *Betrag* $a-d$ zu wechseln, wobei d der Wert der ersten Münzsorte ist.

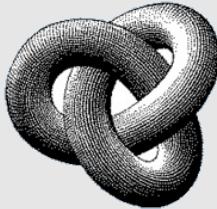
```
(define denominations '((Hdollar 50)
  (Quarter 25) (Dime 10) (Nickel 5) (Cent 1)))
(define (count-change amount)
  (cc amount denominations))

(define (cc a ds)
  (cond [(= a 0) 1]
        [(null? ds) 0]
        [(< a 0) 0]
        [else
         (let ([coin1 (cadr (car ds))]
               [other-coins (cdr ds)])
           (+ (cc (- a coin1) ds)
              (cc a other-coins))))])
> (count-change 100) → 292
```

22 Rückzugsverfahren: Backtracking

22.1 Backtracking-Probleme

Backtracking



M. C. Escher

19 Kombinatorische Probleme

20 Rückzugsverfahren: Backtracking

- Backtracking-Probleme
- Das 8-Damen-Problem
- Allgemeines backtracking-Schema

21 Nichtdeterminismus

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 573 / 1157

Backtracking

Backtracking-Algorithmen

Backtracking-Algorithmen sind *rekursive* Algorithmen, die sich immer dann gut anwenden lassen,

- wenn wir eine Lösung durch systematisches Probieren suchen müssen,
- wenn sich bei jedem Probierschritt mehrere neue Alternativen auftun, die untersucht werden müssen, und der Suchraum exponentiell anwächst,
- wenn jeder Probierschritt dem vorherigen *strukturell ähnlich* ist.

Die Suche in Prolog ist eine wichtige Anwendung von Backtracking.



Typische Backtracking-Probleme

- Suche den Weg aus einem Labyrinth.
- Suche die kürzeste Verbindung zwischen zwei Städten.
- Färbe eine Landkarte mit vier Farben.
- Das „Wolf, Kohlkopf, Ziege“-Rätsel.
- Das 8-Damen-Problem
- Finde magische Quadrate
- Kryptoarithmetik: FOUR+FIVE=NINE

22.2 Das 8-Damen-Problem



Das 8-Damen-Problem

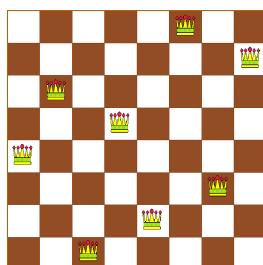
Problem: 106

Gibt es eine Möglichkeit, 8 Damen so auf einem Schachbrett anzurichten, daß sie sich gegenseitig nicht bedrohen?

Eine Dame bedroht eine andere Figur, wenn diese

- in derselben Zeile wie die Dame steht,
- in derselben Spalte wie die Dame steht oder
- auf derselben Diagonalen wie die Dame steht.

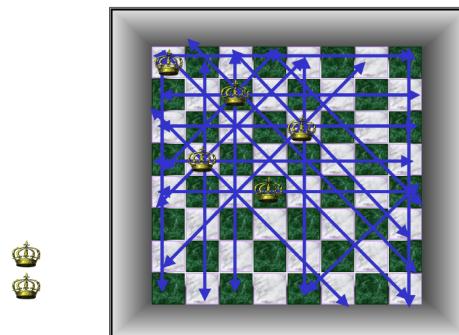
Eine Lösung



- In jeder Zeile, Spalte und Diagonale steht nur eine Dame.

Rekursiver Ansatz:

- Für ein Schachbrett der Größe $n \times n$ ist das Problem gelöst, wenn wir eine Dame in der 1-ten Zeile positioniert haben, und dann eine Lösung fuer die restlichen $n - 1$ Zeilen gefunden haben.
- Wenn es für die weiteren Damen keine Möglichkeit der Plazierung gibt, verschieben wir die Dame in der ersten Zeile (backtracking) und versuchen erneut, den Rest des Brettes zu füllen.
- Wenn wir die Dame erfolglos über die ganze Zeile verschoben haben, dann existiert keine Lösung.
- Dasselbe Schema wenden wir sinngemäß rekursiv auf die weiteren Zeilen an.



Der 1. backtracking-Schritt: Verschieben in Reihe 5

Die Repräsentation des Schachbretts

Das Schachbrett: Da in jeder Zeile und Spalte genau eine Dame stehen muß, führen wir eine Liste p (poss) ein, die für jede Spalte die Position der Dame enthält.

```
(struct queen-coord (row col))  
;definiert die Funktionen:  
  queen-coord-row ; Akzessor  
  queen-coord-col ; Akzessor  
  queen-coord ; Konstruktor
```

Eine print-function für Positionen

Das Schachbrett: Die default print-function zeigt nicht die Felder der Struktur.

```
(struct
  queen-coord
  (row col))

(define (print-queen-coord pos)
  (display (list "row:"_
                 (queen-coord-row pos)
                 " col:"_
                 (queen-coord-col pos))))
```

Die Repäsentation des Resultat

Das Resultat: Als Resultat sollen die Positionen der Damen und die Anzahl der Suchschritte zurückgegeben werden:

```
(struct
  queen-result
  (noMoves ; Zahl der Züge, natural?
   positions) ; die Positionen
   ; (<queen-coord>...)
; constructor: queen-result
; accessors: queen-result-noMoves,
; queen-result-positions

(define (successful? result)
  ; is the result a solution?
  (not (null? (queen-result-positions result))))
```

Die Diagonalen



Zwei Diagonalen

Entlang einer Diagonalen ist entweder

- ▶ die **Summe** der Zeilen- und Spaltenkoordinaten oder
- ▶ die **Differenz** der Zeilen- und Spaltenkoordinaten gleich.

Ist die Dame bedroht?

Seien in den ersten $n - 1$ Zeilen schon Damen aufgestellt, und sei pos die Liste der entsprechenden Zeilen-Spalten-Paare.

```
; Die Dame ist sicher, wenn
(define (safe? positionsSofar newPos)
  ; positionsSofar: list of queen-coord
  ; newPos: queen-coord
  (andmap
    (curry (negate check?) newPos) ; no check
    positionsSofar))
```

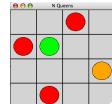
☞ Anmerkung:

- (curry (negate check?) newPos)

ein Prädikat, das prüft, ob eine Dame an newPos die als Argument übergebene nicht Position bedroht: negate erzeugt das inverse Prädikat zu check? und curry bindet dieses inverse Prädikat an newPos .

- (andmap (curry (negate check?) newPos) positionsSofar))

prüft, ob *alle* bisher gesetzten Damen der Liste *positionssofar* die neue Dame nicht bedrohen.



check?

```
; Bedroht eine Dame in Position pos-1
; die Position pos-2?
(define (check? pos-1 pos-2)
  (let ([r1 (queen-coord-row pos-1)]
        [c1 (queen-coord-col pos-1)])
    (or (= c1 pos-2)
        (= (+ r1 c1) (+ r2 pos-2))
        (= (- r1 c1) (- r2 pos-2))))
  ; Diagonale 1
  ; Diagonale 2)

(define (try-queen size positions row column noMoves)
  ; find a single solution
  (let* ((positionToTry
          (make-queen-coord row column))
         (npositions
          (cons positionToTry positions)))
    (cond
      [(> row size)
       (showboard positions size thePause)
       (make-queen-result noMoves positions)]
      ; all queens successfully placed
      [(> column size)
       (make-queen-result noMoves '())
       ; failure, no safe position in this row found
      [(not (safe? positions positionToTry))
       ; move queen to next column
       (begin
         (showboard npositions size thePause)
         (try-queen size positions (+ 1 column)
                    (+ 1 noMoves)))]
      [else ; the queen is safe,
       ; try to place a queen in the next row
       (begin ...)])))
```

```

.....
(else ; the queen is safe,
; try to place next queen
(begin
  (showboard npos size thePause)
  (let ([res1
    (try-queen size npos (+ 1 row)
      1 (+ 1 noMvs) )])
    (if (successful? res1)
      res1 ; return the result
      ; backtrack, if not successful
      (try-queen size
        poss row (+ 1 col)
        (+ (queen-result-noMoves res1)
          noMvs)))))))))))

```

Beispiel

```

(define (queen1 size)
  (try-queen size '() 1 1))

> (queen1 8 0.001) →
(Number of moves: 139201939763888611926)
positions:
  (row: 8 col: 4)(row: 7 col: 2)
  (row: 6 col: 7)(row: 5 col: 3)
  (row: 4 col: 6)(row: 3 col: 8)
  (row: 2 col: 5)(row: 1 col: 1)
> (queen1 4 0.01) →
(Number of moves: 53)
positions:
  (row: 4 col: 3)(row: 3 col: 1)
  (row: 2 col: 4)(row: 1 col: 2)

```

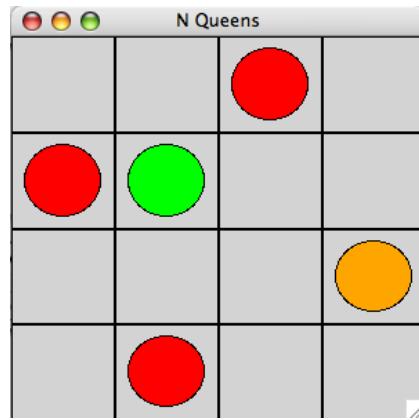
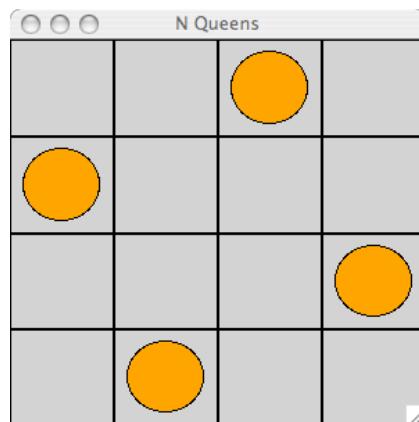
Für die Anzeige: [htdp/show-queen.rkt](#)

- Das teachpack `show-queen` exportiert die Funktion `show-queen`, die ein Schachbrett zeichnet und die Positionen von Damen markiert.
- Das Schachbrett wird als Liste von Listen repräsentiert. Jede Unterliste steht für eine Spalte des Brettes.
- Jede Unterliste enthält für jede Zeile einen Wahrheitswert, der #t ist, wenn die Spalte eine Dame enthält, #f sonst.

```
(require htdp/show-queen)
(show-queen
'((#f #t #f #f)
 (#f #f #f #t)
 (#t #f #f #f)
 (#f #f #t #f)))
```

Zur Übung empfohlen: Implementieren Sie die Funktion show-board, die das Schachbrett und die Position der Damen anzeigt, mittels show-queen.

Das Bord



- Geht man mit der Maus auf ein Feld des Brettes, wird das Feld grün markiert.
- Die bedrohten Damen werden rot markiert.

⌚ Anmerkung: Suche *alle* Lösungen:

Wenn wir alle Lösungen finden wollen, dann lassen wir einfach jeden rekursiven Aufruf von `try-queen` eine Liste von *allen* gefundenen Lösungen zurückgeben und verbinden die Listen mit **append** zu einer Gesamtliste. Da wir systematisch alle Stellungen durchprobieren wollen und nicht aufhören wollen, sobald die erste Lösung gefunden wird, probieren wir für jede Dame alle Felder in der folgenden Reihe aus. Das geht am elegantesten, indem wir eine Liste aller Spaltennummern erzeugen (`nats-1-n size`) und mit **map** `try-all-queens` auf jede dieser Spalten anwenden.

Die Parameter für die nächste Reihe werden wieder mit currying gebunden: Die Reihe ist (+ 1 row) und die Liste der belegten Positionen wird um die zuletzt geprüfte Position verlängert.

Damit wir das “mapping” schon nutzen können, um die Dame systematisch über die erste Reihe zu schieben, müssen wir die Funktion mit `row=0` aufrufen. Diese Koordinate ist aber außerhalb des Schachbretts und darf nicht in die Liste der Positionen aufgenommen werden, deshalb die Abfrage: (= row 0) beim rekursiven Aufruf von `try-all-queens`.

Suche alle Lösungen

Beispiel: 107 (Alle Lösungen des 8-Damen-Problems)

Um alle Lösungen zu finden, ändern wir das Suchverfahren wie folgt:

- Jeder rekursiven Aufruf von `try-queen` gibt eine Liste von allen gefundenen Lösungen zurück.
- Die Listen werden mit **append** zu einer Gesamtliste verbunden.
- Um alle Lösungen zu finden, wird `try-queen` jeweils für alle Felder der Folgezeile aufgerufen (mittels **map**).

Alle Lösungen: Präambel

```
(define
  (try-all-queens size positions row column)
  (let ([positionToTry
         (make-queen-coord row column)])
    (cond
      [(not (safe? positions positionToTry)) '()]
      [(= row size)
       ; all queens successfully placed
      (let ([theSolution
              (list (cons positionToTry
                      positions)))])]
```

```

(showboard (car theSolution) size thePause)
theSolution)] ;return the solution
[else
; the queen is safe ,
; try all positions next row ...

```

Alle Lösungen: Rekursionsschritt

```

(define
  (try-all-queens size poss row column)
  ....
[else ; the queen is safe , try all positions next row
(apply append
  (map (curry try-all-queens
    size
    (if (= row 0) '()
      (cons
        positionToTry
        positions)))
    (+ 1 row))
  (nats-1-n size))))]))))
; try on all columns 1..size

```

```

(define (queens size)
  (let ([solutions
        (try-all-queens
          size '() 0 1)])
    (writeln "\nnumber_of_solutions: "
            (length solutions))
    solutions
  ))
(queens 8 0.0001) →
((#<struct:queen-coord>
  #<struct:queen-coord> ...)... )
number of solutions: 92

```

Drucken der Lösungen

```

> (define sols (queens 8 0.0001))
> (map (lambda (ps)
         (map print-queen-coord ps))
       sols)
(((8 . 4) (7 . 2) (6 . 7) (5 . 3)

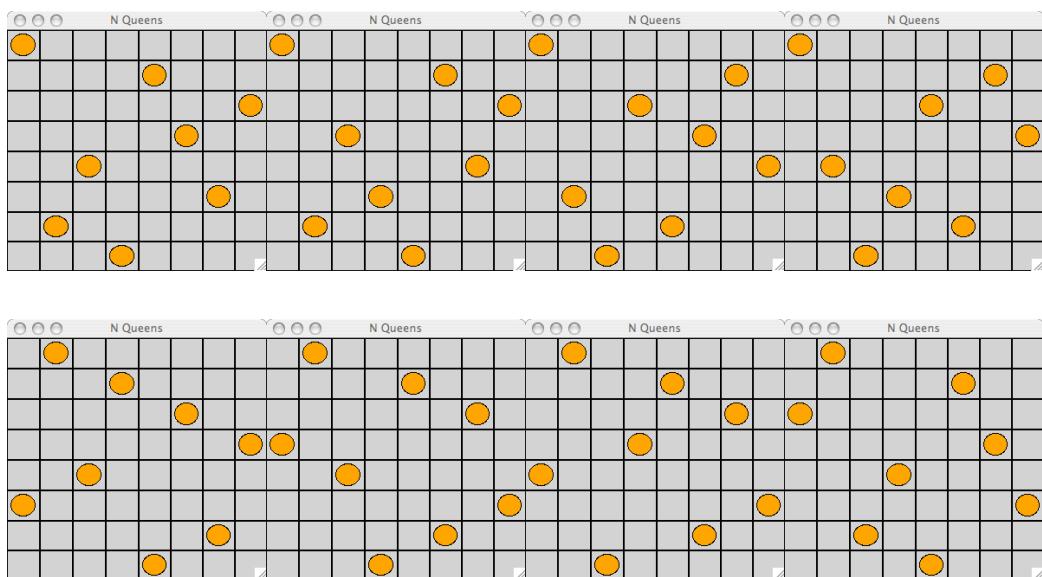
```

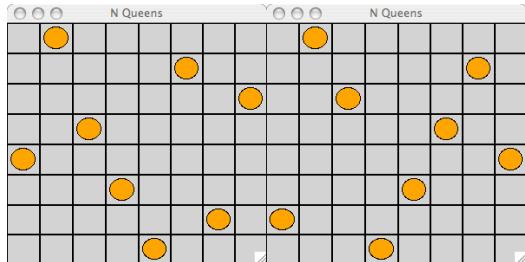
```

(4 . 6) (3 . 8) (2 . 5) (1 . 1))
((8 . 5) (7 . 2) (6 . 4) (5 . 7)
 (4 . 3) (3 . 8) (2 . 6) (1 . 1))
((8 . 3) (7 . 5) (6 . 2) (5 . 8)
 (4 . 6) (3 . 4) (2 . 7) (1 . 1))
((8 . 3) (7 . 6) (6 . 4) (5 . 2)
 (4 . 8) (3 . 5) (2 . 7) (1 . 1))
((8 . 5) (7 . 7) (6 . 1) (5 . 3)
 (4 . 8) (3 . 6) (2 . 4) (1 . 2)) ...

```

Die ersten 10 Lösungen





☞ Anmerkung: Die Funktion `queens` liefert eine Liste *aller* Lösungen.
 Wenn wir nur eine Lösung wünschen, dann rufen wir einfach `(car (queens 8))` auf.

☞ Anmerkung:

Backtracking vs. „list of successes“

- Dem fertigen Programm sieht man nicht mehr an, daß es sich um einen backtracking-Ansatz handelt. Wir beschreiben *konstruktiv und funktional*, wie eine korrekte Lösung aussieht.
- Bei Funktionen in dieser Form spricht man auch vom „*Erfolgslisten-Verfahren*“ (list of successes).
- Der Prozeß, der durch das backtracking ausgelöst wird, ist ebenfalls ein klassisches Beispiel für eine **baumartige Rekursion**.

Backtracking mit Generator

Backtracking bis zur ersten Lösung und Erfolgslistenverfahren haben beide ihre Vor- und Nachteile:

- Wenn wir nur eine Lösung suchen, ist es sinnvoller, auch nur eine Lösung zu berechnen. Das spart Rechenzeit und vermeidet Endlosschleifen, wenn unendlich viele Lösungen existieren.
- Wenn wir Lösungen vergleichen wollen und viele oder alle Lösungen benötigen, ist das Erfolgslistenverfahren besser, da der Algorithmus klarer ist.
- Ein guter Kompromiß ist ein Generator der Lösungen: er verbindet die Vorteile beider Ansätze.

```
; queens with generator
(define
  (generate-all-queens size)
  (generator ()
    (letrec
      ([generate-all-queens
        (lambda (positions row column)
          (let ((positionToTry (make-queen-coord row column)))
            (cond
              [(not (safe? positions positionToTry)) '()]
              [= row size] ; all queens successfully placed
              (let ([theSolution
                    (cons positionToTry positions)])
                (showboard theSolution size thePause)
                (yield theSolution))) ; return the solution
              [else
                ; the queen is safe
                ; try all positions next row
                (for ([col-pos (in-range 1 (+ 1 size))])
                  (generate-all-queens
                    (if (= row 0) '()
                      (cons positionToTry positions)))
                  (+ 1 row) col-pos))))]
              ; try on all columns 1..size
              (generate-all-queens '() 0 1))))
```

```

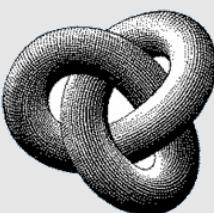
; create the generator
(define next-queen
  (generate-all-queens 8))
; call the generator and print one solution
(map print-queen-coord
  (next-queen)) →
(row: 8 col: 5)(row: 7 col: 2)
(row: 6 col: 4)(row: 5 col: 7)
(row: 4 col: 3) ...

```

☞ Anmerkung: Die Funktion generate-all-queens erzeugt einen Generator, der die Lösungen aufzählt. Bei jedem yield wird eine weitere Lösung zurückgegeben. Der Generator ist zustandsbehaftet: Jeder Aufruf kann unterschiedliche Werte zurückgeben.

22.3 Allgemeines backtracking-Schema

Allgemeines backtracking-Schema



M.C. Escher

19 Kombinatorische Probleme

20 Rückzugsverfahren: Backtracking

- Backtracking-Probleme
- Das 8-Damen-Problem
- Allgemeines backtracking-Schema

21 Nichtdeterminismus

Navigation icons: back, forward, search, etc.

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 600 / 1157

Wie finden wir selbst für ein gegebenes Problem eine backtracking-Lösung?

Wir müssen das Problem auf das allgemeine **backtracking-Schema** abbilden:

Repräsentation des Suchraums als Menge von Zuständen: Beispielsweise Liste der Positionen der Damen, Koordinaten im Labyrinth usw.

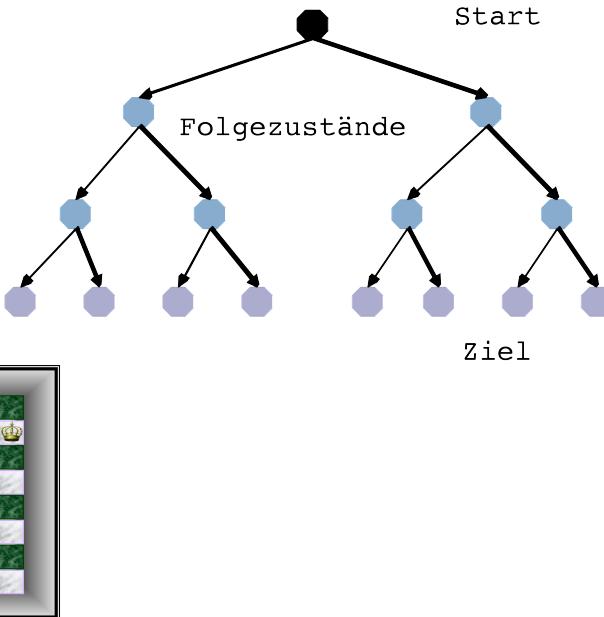
Ausgangszustand: Der Zustand, von dem aus wir systematisch suchen wollen, beispielsweise das leere Schachbrett oder der Eingang des Labyrinths.

Generatorfunktion: Eine Funktion, die einen Zustand auf eine Liste der direkt erreichbaren Folgezustände abbildet.

Test auf Zulässigkeit: Ein Prädikat, das einen Zustand daraufhin überprüft, ob er zulässig ist.

Test auf Erfolg: Ein Prädikat, das feststellt, ob ein Zustand ein Endzustand ist.

Der Suchraum



Dieses Schema können wir als neue Kontrollstruktur verwenden, immer dann, wenn wir ein Suchproblem zu lösen haben. Es ist gleichzeitig ein nützliches Beispiel für die Verwendung von Funktionen höherer Ordnung, denn drei der Parameter sind Funktionen.

Aber Achtung! Das Schema darf nicht in dieser einfachen Form angewendet werden, wenn die Pfade im Zustandsraum Zyklen enthalten. Bei der Labyrinthsuche beispielsweise ist damit zu rechnen, daß Wege im Kreis herum führen. Wenn wir solche Probleme lösen wollen, müssen wir das Schema um einen Test erweitern, der überprüft, ob wir einen Zustand schon einmal untersucht haben. Sonst wird unser schönes rekursives Verfahren eventuell in einer Endlosschleife enden.

Backtracking als Funktion höherer Ordnung

Beispiel: 108 (Allgemeines Backtracking)

Wir abstrahieren den Ablauf des backtracking als Funktion höherer Ordnung *general-backtracking*, mit den vier Parametern:

1. initial-state: state
2. gen-states: state → ({ state })
3. is-legal?: state → boolean
4. is-final-state?: state → boolean

```

;;;
;; Allgemeines Backtracking
;;;
;; General Backtracking
;; initial-state: a data structure describing the state
;;   of an initial tentative solution
;; gen-states: a generator function: "gen-states state"
;;   creates a list of all the possible subsequent states
;; is-legal?: a predicate: "is-legal state" checks
;;   whether a state is admissible
;; is-final-state?: a predicate: "is-final-state state?""
;;   is #t, if the state is a solution to the problem

(define (general-backtracking
          initial-state gen-states
          is-legal? is-final?)
  ;; find all solutions;
  ;; may cause an infinite loop
(letrec
 ([try
  (lambda (state)
    ; (display (list "trying: " state) )
    (cond
      [(not (is-legal? state)) '(fail ,state)]
      [(is-final? state)
       (cons state ; further solutions
             (append-map
               try (gen-states state)))]
      [else
       (append-map
         try (gen-states state)))])))
  (try initial-state)))

```

☞ Anmerkung: Für weitere Formen des allgemeinen backtracking-Schemas siehe: se3-bib/backtracking-module.rkt.

Hier finden Sie eine Variante, die Zyklen vermeidet sowie eine Variante, die nur die erste Lösung sucht.

Ein linearer Suchraum

Beispiel: 109 (Backtrackingsuche in den natürlichen Zahlen)

Ein linearer Zustandsraum; jede Zahl (jeder Knoten) hat nur einen Nachfolger, gesucht n_z

- Ausgangszustand: eine natürliche Zahl n_0
- Folgezustände von n : Die Menge mit dem einen Nachfolger $n+1$: $\{n+1\}$
- Zulässig?: ist n kleiner oder gleich n_z ?
- Erfolg?: Ist die Zahl n gleich der gesuchten Zahl n_z ?

Beispiel: Linearer Suchraum

```
(define (count start goal)
  (general-backtracking
    start ;initial-state
    (lambda (state) (list (+ state 1)))
    ; gen-states
    (lambda (state)
      (<= state goal) ; is-legal?
      (curry equal? goal))) ; is-final?
  (count 1 3) → (3 fail 4)
  (trying: 1)(trying: 2)(trying: 3)(trying: 4))
```

☞ Anmerkung: Ein trace der Funktionsparameter:

```
(define (general-backtrackingTrace
          initial-state
          gen-states
          is-legal?
          is-final?)
  ; find all solutions; careful! may cause infinite loop
  ; define instead of letrec
  (define
    (try state)
    (display (list "trying:" state) )
    (cond
      [(not (is-legal? state)) '(fail ,state)]
      [(is-final? state)
       (cons state ; further solutions
```

```

        (append-map try (gen-states state)))]
[else
  (append-map try (gen-states state))])
  (trace try gen-states is-legal? is-final?)
  (try initial-state))

(bt:trace 1 3)
| (try 1)
| (trying: 1)| (is-legal? 1) → #t
| | (is-final? 1) → #f
| | (gen-states 1) → (2)
| | (try 2)
| (trying: 2)| | (is-legal? 2) → #t
| | | (is-final? 2) → #f
| | | (gen-states 2) → (3)
| | | (try 3)
| (trying: 3)| | (is-legal? 3) → #t
| | | (is-final? 3) → #t
| | | (gen-states 3) → (4)
| | | (try 4)
| (trying: 4)| | | (is-legal? 4) → #f
| | | (fail 4)
| | | (3 fail 4)
| | | (3 fail 4)
|(3 fail 4)
(3 fail 4)

```

Variante 2

```

(define (count-to-n n)
  (general-backtrackingTrace
    1 ; initial-state
    (compose list add1)
    ; gen-states: eine Liste ( state+1)
    (always #t); is-legal? immer wahr
    (curryr = n); is-final? Zustand=n?
  )

```

- Warnung! Endlosschleife, weil Suchraum unbegrenzt, besser: (curry <= n) für is-legal?

Als allgemeines Backtrackingproblem

Beispiel: 110 (8-Damen)

- Der Zustand wird repräsentiert als eine Liste von Koordinatenpaaren. Die Koordinaten der zuletzt aufgestellten Dame stehen am Kopf der Liste. `list-of-next-positions` erzeugt eine Liste von Folgezuständen, für jedes Feld der nächsten Reihe einen.
- Die Funktion `position-is-safe?` prüft jeweils, ob die zuletzt positionierte Dame nicht von anderen Damen bedroht wird.
- Das Brett ist gefüllt, wenn soviel Damen auf dem Brett stehen, wie das Brett Reihen hat, also `(length state)=size` gilt.

8-Damen als allgemeines Backtrackingproblem

```
(define (queens-b size)
  (let
    ([ list-of-next-positions
        (lambda (state) ...]
      [ position-is-safe?
        (lambda (state) ...]
      [ all-rows-filled?
        (compose (curry = size) length))]
      (general-backtracking
        '() ; initial-state , board empty
        list-of-next-positions
        position-is-safe?
        all-rows-filled? )))

(define (queens-b size)
  (let
    ([ list-of-next-positions
        (lambda (st)
          (let* ([next-row (add1 (length st))]
                [columns (nats-1-n size)]
                [next-coords
                  (map (curry make-queen-coord next-row)
                      columns)])
            (map (curryr cons st) next-coords))])
      [ position-is-safe?
        ; check the most recently added position , i.e. (car state)
```

```

; against the queens placed earlier , i.e. (cdr state)
(lambda (st)
  (or (null? st)
      (safe?
        (cdr st) (car st)))))

[ all-rows-filled?
  (compose (curry = size) length)])
(general-backtracking
  '() ; initial-state , the board is empty
  list-of-next-positions ;next states
  position-is-safe? ; is-legal?
  all-rows-filled? ) ) ; is-final?

```



Denksportaufgabe: Missionare und Taschenlampe Vier Missionare stehen bei Nacht an einer tiefen Schlucht mit einer baufälligen Hängebrücke, die man nur bei Licht überqueren kann. Auch sollten nie mehr als zwei Personen gleichzeitig die Brücke betreten, weil diese sonst unweigerlich zusammenbrechen würde. Die Missionare werden von wilden Tieren verfolgt und müssen unbedingt noch in der Nacht die Brücke überqueren, auch wenn das sehr riskant ist. Glücklicherweise haben sie eine Taschenlampe dabei, deren Batterien aber nur noch für eine Stunde reichen werden.

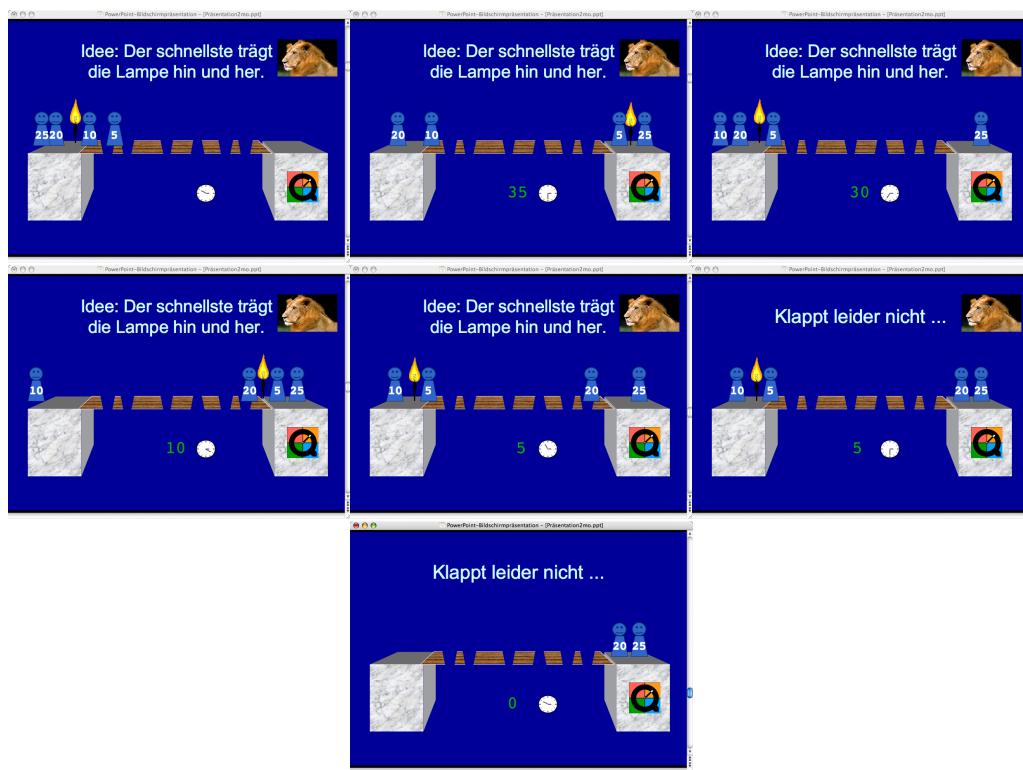
Die Missionare sind unterschiedlich gut in Form: Der Sportlichste schafft es in 5 Minuten, die Brücke zu überqueren, Missionar 2 braucht 10 Minuten, Missionar 3 braucht 20 Minuten und Missionar 4 sogar 25 Minuten. Werden sie es schaffen, die Brücke sicher zu überqueren, und wenn ja, wie???

Eine Denksportaufgabe

Beispiel: 111 (Das Szenario:)

- Die Missionare stehen an einer baufälligen Hängebrücke.
- Diese trägt nur zwei Personen.
- Es ist stockfinstere Nacht.
- Sie haben eine Taschenlampe, die nur noch eine Stunde brennt.
- Sie benötigen unterschiedlich lange, die Brücke zu überqueren: 5, 10, 20, 25 min.
- Sie werden von wilden Tieren verfolgt.

Lösungsidee: Der Schnellste trägt die Taschenlampe hin und her



Suche mit allgemeinem backtracking: Das folgende Programm verwendet das allgemeine backtracking-Schema.

Die Suche geht so vor sich: Die Missionare können entweder auf der einen oder anderen Seite der Brücke sein. Der Zustand drückt sich also in den Mengen {Missionare rechts der Brücke} und {Missionare links der Brücke}, dem Ort der Taschenlampe und der verbrauchten Zeit aus. Wenn alle Missionare rechts angekommen sind und die Lampe noch brennt, ist eine Lösung gefunden.

Zustandsübergänge bestehen darin, daß jeweils ein oder zwei Missionare mit der Taschenlampe die Brücke überqueren. Dabei wird soviel Zeit verbraucht, wie der langsamste der beiden Missionare zum Überqueren benötigt. Die Strategie ist dabei die folgende: Jeweils zwei Missionare gehen von links nach rechts, dann geht einer mit der Taschenlampe zurück, damit dann wieder zwei zusammen nach rechts gehen können, usw. Die Folgezustände sind also unterschiedlich, jenachdem, ob sich die Taschenlampe rechts oder links der Brücke befindet.

Die Funktion, die die Folgezustände erzeugt, führt auch eine Protokoll-Liste, so daß am Ende die Folge der Bewegungen ausgegeben werden kann.

Für die optimale Suche nach einer Lösung ist folgendes wichtig: Der Zustand der Suche ändert sich dadurch, daß jeweils ein oder zwei Missionare sich über die Brücke bewegen. Damit die Suche nicht in Endlos-schleifen hängen bleibt, müssen wir verhindern, daß im Folgezustand wieder genau diejenigen Missionare zurücklaufen, die gerade die Brücke überquert haben. Das läßt sich sehr elegant durch eine unsymmetrische Funktion für die Folgezustände erreichen, abhängig davon, wo sich gerade die Taschenlampe befindet:

Taschenlampe links: Wenn die Taschenlampe links der Brücke ist, dann überqueren zwei Missionare mit der Taschenlampe die Brücke von links nach rechts.

Taschenlampe rechts: Wenn die Taschenlampe rechts der Brücke ist, dann überquert nur ein Missionar mit der Taschenlampe die Brücke von rechts nach links und bringt die Lampe den Zurückgebliebenen.

Auf diese Weise wird die Anzahl der Missionare links der Brücke immer kleiner und es können keine Zyklen auftreten.

Suche mit Backtracking: Repräsentation



Zustände:

- ▶ Listen der Missionare links und rechts
- ▶ Wo ist die Taschenlampe?
- ▶ Restzeit

Übergangsfunktion: Wenn die Taschenlampe links ist,
 → dann gehen **zwei** mit der Taschenlampe nach **rechts**,
 ← sonst geht **einer** mit der Taschenlampe nach **links**.

Zulässig?: Restzeit ist noch positiv.

Fertig? : Keiner steht mehr links.

Protokoll: Liste der Zustände

☞ **Anmerkung:** Die Datenstrukturen und Funktionen dieses Programms werden wir im folgenden Kapitel als Beispiel für das Objektsystem von Racket und Common Lisp besprechen. Hier ist zunächst der funktionale Lösungskern. Das vollständige Programm finden Sie unter se3-bib/missionaries-module.ss.

```
(define (missionaries speeds max-t)
  (map pretty-print-protocol
    (general-backtracking
      (make-m-state speeds '() 'left max-t '())
      gen-moves
      (compose (curryr >= 0)
               time-left) ;;; is-legal:
      (compose null?
               left-group) ;;; nobody left
               ) #t)))
(define (gen-moves state)
  (case (torch state)
    ((left) (all-moves->right state))
    ((right) (all-moves->left state))))
```

Zwei Lösungen: _____

> (missionaries '(5 10 20 25) 60)

```

Time left: 0
(5 10 → right)
(5 → left)
(20 25 → right)
(10 → left)
(5 10 → right)
Time left: 0
(5 10 → right)
(10 → left)
(20 25 → right)
(5 → left)
(5 10 → right) → #t

```

23 Nichtdeterminismus

23.1 Nichtdeterministische Suche: amb

Nichtdeterministische Suche: amb



- 19 Kombinatorische Probleme
- 20 Rückzugsverfahren: Backtracking
- 21 Nichtdeterminismus
 - Nichtdeterministische Suche: amb
 - Färben eines Graphen

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 620/1157

Nondeterminism⁴

⁴Helpdesk: Teach Yourself Scheme in Fixnum Days

McCarthy's nondeterministic operator amb [25, 4, 33] is as old as Lisp itself, although it is present in no Lisp. amb takes zero or more expressions, and makes a nondeterministic (or "ambiguous") choice among them, preferring those choices that cause the program to converge meaningfully. Here we will explore an embedding of amb in Scheme that makes a depth-first selection of the ambiguous choices, and uses Scheme's control operator call/cc to backtrack for alternate choices. The result is an elegant backtracking strategy that can be used for searching problem spaces directly in Scheme without recourse to an extended language. The embedding recalls the continuation strategies used to implement Prolog-style logic programming [16, 7], but is sparser because the operator provided is much like a Scheme boolean operator, does not require special contexts for its use, and does not rely on linguistic infrastructure such as logic variables and unification.

14.1 Description of amb

An accessible description of amb and many example uses are found in the premier Scheme textbook SICP [1]. Informally, amb takes zero or more expressions and nondeterministically returns the value of one of them. Thus,

(amb 1 2)

may evaluate to 1 or 2.

amb called with no expressions has no value to return, and is considered to fail. Thus,

(amb) → **ERROR!!** amb tree exhausted!

(We will examine the wording of the error message later.)

In particular, amb is required to return a value if at least one its subexpressions converges, ie, doesn't fail. Thus,

(amb 1 (amb))

and

(amb (amb) 1)

both return 1.

Clearly, amb cannot simply be equated to its first subexpression, since it has to return a non-failing value, if this is at all possible. However, this is not all: The bias for convergence is more stringent than a merely local choice of amb's subexpressions. amb should furthermore return that convergent value that makes the entire program converge. In denotational parlance, amb is an angelic operator.

For example,

(amb #f #t)

may return either #f or #t, but in the program

```
(if (amb #f #t)
  1
  (amb))
```

the first amb-expression must return #t. If it returned #f, the if's "else" branch would be chosen, which causes the entire program to fail.

An amb macro⁵

This is added just because it is too much fun to miss. To learn about 'amb', look for it in the Help Desk, in the Teach Yourself Scheme in Fixnum Daysön-line manual.

(amb expr ...) [syntax]

Execute forms in a nondeterministic way: each form is tried in sequence, and if one fails then evaluation continues with the next. '(amb)' fails immediately.

(amb-assert cond) [procedure]

Asserts that 'cond' is true, fails otherwise.

(amb-collect expr) [syntax]

Evaluate expr, using amb-fail repeatedly until all options are exhausted and returns the list of all results.

Nicht-deterministische Suche:

McCarthys *amb*-Operator

- Der *amb*-Operator erhält eine endliche Zahl von Argumenten und wählt *nicht-deterministisch* eins davon aus.
- amb* ohne Argumente aufgerufen ergibt das Resultat *fail*.
- Stößt eine *amb*-expression bei der Auswertung auf *fail*, wird automatisch eine Alternative ausprobiert, solange, bis eine Lösung gefunden wurde, oder alle Alternativen erschöpft sind.

amb als engelhafter Operator

amb ist ein engelhafter Operator:

(angelic operator):

Es ist garantiert ist, daß aus der Vielzahl von Möglichkeiten diejenige gewählt wird, die dazu führt, daß der umgebende Programmkontext nicht das Resultat *fail* liefert, sofern eine solche Lösung existiert.

⁵aus der swindle-Dokumentation

fail: Beispiel

Nach jedem `fail` kehrt das Programm zum letzten amb zurück, das noch eine Alternative hatte und wiederholt die Auswertung mit der neuen Alternative.

Willkommen bei DrRacket, Version 5.0.1 [3m].

Sprache: racket; *memory limit: 256 MB*.

```
> (require swindle/extra)
> (amb 1 2 3 4) -> 1
> (amb) -> 2
> (amb) -> 3
> (amb) -> 4
> (amb)
```



amb: tree exhausted

fail: Beispiel

Nach jedem `fail` wird eine neue Alternative gewählt und der Kontext neu ausgewertet.

Willkommen bei DrRacket, Version 5.0.1 [3m].

Sprache: racket; *memory limit: 256 MB*.

```
> (require swindle/extra)
> (define a (amb 1 2 3 4))
> a -> 1
> (amb)
> a -> 2
> (amb)
> a -> 3
>
```

amb-assert

amb-assert:

amb-assert prüft, ob eine Bedingung erfüllt ist.

Wenn nicht, ist das Resultat `fail`.

Sprache: racket; *memory limit: 256 MB*.

```
> (define a (amb 2 4 5 6))
> a -> 2
> (amb-assert (> a 4))
> a -> 4
```

```

> (amb-assert (> a 4))
> a -> 5
> (amb-assert (> a 4))
> a -> 5

```

Sammele alle Lösungen: amb-collect

- Das Macro *amb-collect* verbindet alle positiven Resultate (die nicht zum fail führen) zu einer Liste.
- Wenn es unendlich viele Lösungen gibt, entsteht eine Endlosschleife.

Beispiel: 112 (Prüfen von Summen:)

Suche alle Paare von Werten a und b, deren Summe „sieben“ ergibt.

```

> (require swindle/extr)
> (amb-collect
  (let ((a (amb 1 2 3 4 5 6 7))
        (b (amb 2 4 6 8)))
    (amb-assert (= (+ a b) 7))
    (cons a b)))
'((1 . 6) (3 . 4) (5 . 2))
>

```

☞ **Anmerkung: continuations** Die Implementation des amb-Operators nutzt ein Sprachelement, das typisch für Racket und Scheme ist, aber das wird hier nicht weiter behandelt werden - die continuations.

- Bei jedem amb-Ausdruck wird der Berechnungskontext des amb eingefroren und als continuation gespeichert.
- Bei jedem amb-fail wird die continuation neu aktiviert und die Auswertung mit einem alternativen Wert für amb neu begonnen.
- In ähnlicher Weise können continuations genutzt werden, um Laufzeitfehler abzufangen und nach einer Ausnahme die Berechnung an einem wohldefinierten Aufsetzpunkt neu zu beginnen.

23.2 Färben eines Graphen

Das Vier-Farben-Problem

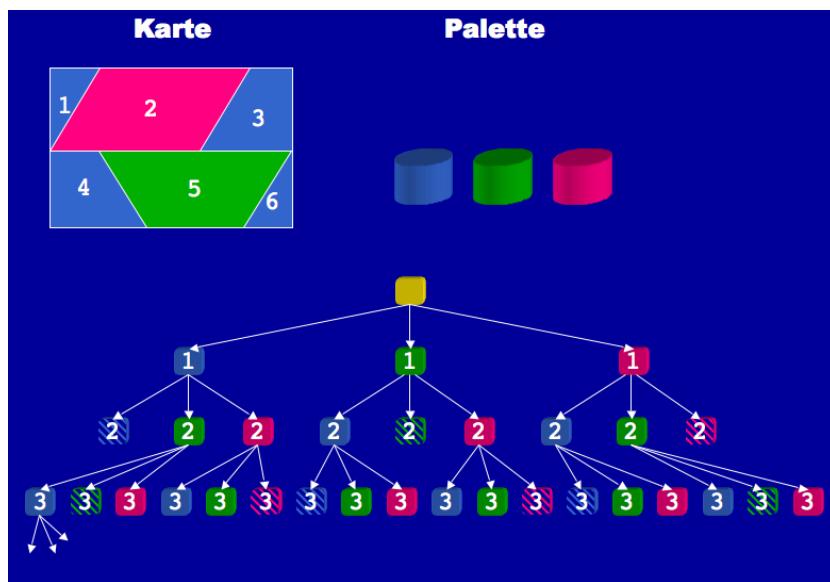
Problem: 113 (Färben einer Landkarte)

Können die Länder auf einer Landkarte mit nur vier Farben so eingefärbt werden, daß benachbarte Länder nie die gleiche Farbe haben?

Lösung: 114

Wir durchsuchen *nicht-deterministisch* den Baum aller Möglichkeiten, die Länder zu färben.

Das Vier-Farben-Problem als Suchproblem



☞ Anmerkung: Das folgende Programm ist fast wörtlich aus den Beispielprogrammen im DrRacket Helpdesk: Teach Yourself Scheme in Fixnum Days .. übernommen. Für bessere Lesbarkeit wurde die Datenstruktur zur Repräsentation von Ländern leicht modifiziert und die structure „country“ eingeführt.

Repräsentation: Die Farben

```
(define choose-color
  (lambda ()
    (amb 'red 'yellow 'blue 'green)))

> (choose-color) → red

(struct country
  (thename thecolor theNeighborColors) )

(define color-europe
  (lambda ()
    ;choose colors for each country
    (let ((p (choose-color)) ;Portugal
          (e (choose-color)) ;Spain
          (f (choose-color)) ;France
          (b (choose-color)) ;Belgium
          (h (choose-color)) ;Holland
          (g (choose-color)) ;Germany
          (l (choose-color)) ;Luxemb
          (i (choose-color)) ;Italy
          (s (choose-color)) ;Switz
          (a (choose-color)) ;Austria
          )....
```

Repräsentation der Länder

```
(let ((portugal
        (country 'portugal p
                  (list e)))
      (spain
        (country 'spain e
                  (list f p)))
      (france
        (country 'france f
                  (list e i s b g l)))
      (belgium
        (country 'belgium b
                  (list f h l g)))
      (holland
        (country 'holland h
                  (list b g)))) ....
```

```

(germany
  (country 'germany g
    (list f a s h b l)))
(luxembourg
  (country 'luxembourg l
    (list f b g)))
(italy
  (country 'italy i
    (list f a s)))
(switzerland
  (country 'switzerland s
    (list f i a g)))
(austria
  (country 'austria a
    (list i s g)))

```

Eine Liste der Länder

```

(let ([countries
      (list portugal spain
            france belgium
            holland germany
            luxembourg
            italy switzerland
            austria)])
(for-each
  (lambda (c)
    (amb-assert
      (not (memq
              (country-theColor c)
              (country-theNeighborColors c))))))
  countries)
  ;output the color assignment
(for-each
  (lambda (c)
    (display (country-theName c))
    (display " ")
    (display (country-theColor c))
    (newline)))
  countries)))))) )

```

Probelauf

```
(require swindle/extra; fuer amb
          se3-bib/backtracking-module);
> (color-europe)
portugal red
spain yellow
france red
belgium yellow
holland red
germany blue
luxembourg green
italy yellow
switzerland green
austria red
```

Weitere Resultate

```
> (amb)
portugal red
spain yellow
france red
belgium yellow
holland red
germany blue
luxembourg green
italy blue
switzerland yellow
austria red
> (amb)
portugal red
spain yellow ....
```

☞ Anmerkung: Die folgende Funktion „general-backtracking-first-solution-only-amb“ sucht nicht-deterministisch nach der ersten Lösung. Sollte keine Lösung existieren, wird der Fehler „amb: tree exhausted“ signalisiert.

Um das Programm auszuprobieren, wird das Modul „swindle/extra.ss“ benötigt (für amb, amb-assert).

amb–car ist eine Funktion, die amb rekursiv auf eine Liste von Argumenten anzuwendet.*amb–car* in tools-module.rkt definiert. Das ist manchmal nötig, da amb eine special form ist, die nicht mit apply auf eine Liste angewendet werden kann.

```
(define (amb–car xs)
;recursively try all elements of xs with amb
  (if (null? xs) (amb)
    (amb (car xs) (amb–car (cdr xs)))))

;; result: solution state
;; find the first solution,
;; non deterministically using "amb"
;; signals an error "amb: tree exhausted",
;; if no solution exists

(define
  (general–backtracking–first–solution–only–amb
   initialState gen–states is–legal? is–final?)
(define (try st)
  (amb–assert (is–legal? st))
  ; fail, if illegal
(cond
  [(is–final? st)
   (display "Solution:_") st]; the solution
  [else
   (let ([nextState
           (amb–car (gen–states st))])
     ; pick non-deterministically a state
   (let ([so (try nextState)])
     (amb–assert so); check the state
     so))]); return the result
  (try initialState))
```


Teil IX

Fallstudien

24 Musterabgleich (pattern matching)

24.1 Eliza: Ein regelbasierter Übersetzer



Fallstudien



Eliza

- 22 Musterabgleich (pattern matching)
- Eliza: Ein regelbasierter Übersetzer
 - Pattern matching
 - Die Regelbasis und Dialogschleife
 - Kontrollabstraktion und Werkzeuge
 - STUDENT: Algebraische Probleme

- 23 Means-Ends-Analyse: GPS

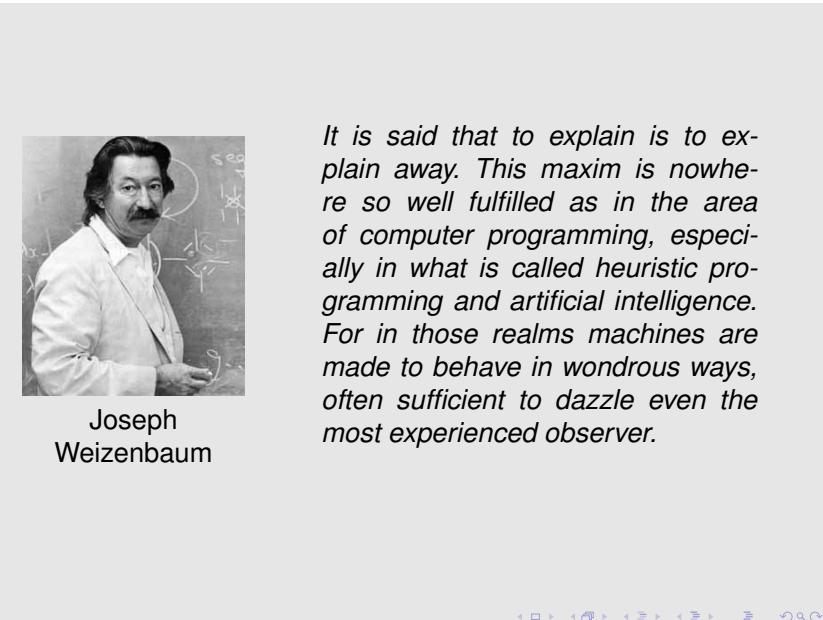
- 24 GPS Anwendungen

Regelbasierte Sprachübersetzung Als erste Anwendung für kombinatorische Funktionen haben wir schon das “backtracking” kennengelernt, eine systematische Suche nach einer Lösung für ein kombinatorisches Problem. Die zweite wichtige Verfahren, das wir hier besprechen wollen, ist der Abgleich von Mustern mit einer Vorlage (pattern matching). Als Anwendungsbeispiel für den Musterabgleich werden wir das ELIZA-Programm von Joseph Weizenbaum kennenlernen, das *pattern matching* auf Sätze der natürlichen Sprache anwendet. Hier in der Vorlesung werden wir Ihnen die Common Lisp-Version von [Norvig, 1992](#) vorstellen, die wir für Sie nach racket übersetzt haben.

ELIZA ist ein Programm, mit dem ein Dialog in englischer Sprache geführt werden kann. Weizenbaum hat es entwickelt, um zu zeigen, daß mit einfachen Mitteln die Illusion von intelligentem Verhalten erzeugt werden kann.

ELIZA war eines der ersten Programme, das sowohl englische Sprache als Eingabe verarbeitet als auch als Ausgabe erzeugt hat. Weizenbaum hat das Programm nach der Helden Eliza aus G.B. Shaws Komödie Pygmalion benannt; Eliza Doolittle, auch bekannt als “My Fair Lady”, hat von Professor Higgins gelernt, perfekt Konversation zu machen. Das Programm ist zuerst 1966 in den *Communications of the ACM* erschienen und zehn Jahre später in Weizenbaums viel diskutiertem Buch *Computer Power and Human Reason* ([Weizenbaum, 1976](#)) nochmals analysiert worden.

Weizenbaum, 1966 selbst schreibt dazu in der Einleitung:



But once a particular program is unmasked, once its inner workings are explained in a language sufficiently plain to induce understanding, its magic crumbles away; it stands revealed as a mere collection of procedures, each quite comprehensible.

The observer says to himself, "I could have written that." With that thought he moves the program in question from the shelf marked "intelligent", to that reserved for curios, fit to be discussed only with people less enlightened than he.

The object of this paper is to cause just such re-evaluation of the program about to be “explained”. Few programs ever needed it more.

Joseph Weizenbaum (zitiert nach (Norvig, 1992))

Die Eliza-Domäne: Gesprächstherapie



An diesem Beispiel sollen Sie mehrere Dinge kennenlernen:

- *Pattern matching* als ein kombinatorisches Problem, das am elegantesten rekursiv zu lösen ist,
- den *modularen Entwurf* eines Programms durch eine geeignete Auswahl von Funktionen für Teilaufgaben,
- die Verwendung von Funktionen höherer Ordnung als *flexible Werkzeuge*,
- die Verwendung von Funktionen höherer Ordnung, um Hilfsfunktionen zur Parametrisierung von Werkzeugen zu erzeugen.

Die Domäne: Gesprächstherapie Die Domäne von ELIZA war die Psychotherapie. Die Aufgabe bestand darin, einen Dialog zu führen, bei dem der Arzt oder die Ärztin im wesentlichen passiv sind und die Patienten ermuntern, möglichst viel über sich zu enthüllen. Wenn die Patienten von sich aus Stichworte zu persönlichen Themen geben, wie etwa auf die Eltern oder Geschwister zu sprechen kommen, wird nachgefragt. Die Antworten werden aber niemals bewertet oder sonst in irgend einer Form kritisiert. Die Initiative geht also immer von den Patienten aus. Auf der nächsten Folie folgt ein Beispieldialog mit dem ursprünglichen ELIZA-Programm. Die Benutzer-Eingaben sind in Groß- und Kleinschreibung wiedergegeben, ELIZAS Antworten in Großbuchstaben. Sie können im XEmacs einen Dialog mit einem weiterentwickelten ELIZA-Programm namens *Doctor* führen, wenn Sie Meta-x doctor eingaben.

Eliza-Original-Dialog

Men are all alike. IN WHAT WAY
They're always bugging us
about something or another.
CAN YOU BE MORE SPECIFIC
Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE
He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED
It's true. I am unhappy.
DO YOU THINK COMING HERE WILL HELP YOU
NOT TO BE UNHAPPY
I need some help, that much is certain.
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
Perhaps I could learn to get along
with my mother.
TELL ME MORE ABOUT YOUR FAMILY

Die Muster

- Der Dialog erweckt den Anschein, als ob ELIZA den Patienten verständne:
I AM SORRY TO HERE YOU ARE DEPRESSED . . .
TELL ME MORE ABOUT YOUR FAMILY . . .
- ELIZA reagiert aber nur schematisch auf bestimmte *Muster* in der Eingabe. Eingabemuster werden zufällig auf passende Ausgabemuster abgebildet.
- Wenn in der Eingabe von Ähnlichkeit gesprochen wird (alike , same), wird eine Frage dazu erzeugt: IN WHAT WAY?
- "I need X" wird auf eine Frage abgebildet, wie „WHAT WOULD IT MEAN TO YOU TO GET X?“ .

Transformation der Eingabe

- ELIZA hängt völlig davon ab, daß die Dialogpartner sinnvolle, interessante Stichworte in das Gespräch einbringen. Sonst werden nur Standardfloskeln generiert:
GO ON, PLEASE **CONTINUE**, VERY INTERESTING usw.
- Die Muster erfassen die Semantik eines Satzes nur teilweise:
Aus "*I need this like a hole in the head.*" wird
WHAT WOULD IT MEAN TO YOU **IF** YOU GOT IT LIKE A HOLE IN THE HEAD
- Auch völlig unsinnige Eingabe wird schematisch transformiert:
I want tybgrjjtghg.
SUPPOSE YOU GOT TYBGRJJTGHG SOON

PARRY - ein Simulator für paranoides Verhalten: Das Programm PARRY von Kenneth Colby, 1975 diente dazu, paranoides Gesprächsverhalten zu simulieren. Norvig zitiert eine Teil des berühmten Dialogs zwischen ELIZA und PARRY. Dieser Dialog zeigt sehr deutlich, daß das pattern matching Verfahren versagt, wenn die Gesprächspartner ebenfalls nur schematisch handeln, keine echte Intelligenz besitzen und keine originären Ideen in das Gespräch einbringen können.

Eliza und Parry im Dialog

I don't understand your motives.
WHY DON'T YOU UNDERSTAND MY MOTIVES?
Shouldn't I be?
WHY DO YOU SAY THAT JUST NOW?
That's just the way it is.
SUPPOSE THAT'S JUST THE WAY IT WERE NOT.
So?
I AM NOT SURE I UNDERSTAND YOU FULLY.
You harp on the same questions too much.
WHAT DOES THAT SIMILARITY SUGGEST TO YOU?
I don't get you.
DO YOU WISH THAT YOU DID GET I?

Spezifikation der Teilaufgaben

Teilaufgaben:

1. Lese die Eingabe.
2. Suche ein passendes Muster.
3. Transformiere die Eingabe zur Ausgabe.
4. Schreibe die Ausgabe.

Die erste und letzte Teilaufgabe sind mit den eingebauten Scheme-Funktionen **read** und **write** trivial zu lösen, wenn die Eingabe als Liste vorliegt.

24.1.1 Pattern matching

Pattern matching im Eliza-Programm

Beispiel: 115 (Musterabgleich)

Durch den Vergleich einer Eingabe mit einem Muster werden Variablen gebunden, die in die Ausgabe eingesetzt werden.

Pattern: (I need a X)
Response: (what would it mean to you
if you got a X?)

Input: (I need a vacation)
Transformation: (what would it mean to you
if you got a vacation ?)

Pattern matching: Rekursionsschema

Wie wird die Eingabe mit den Mustern verglichen?

Am besten rekursiv:

```
(define (pat-match pattern input)
; Does the pattern match the input?
; Any variable can match anything.
(if (variable? pattern) #t
  (if (or (not (pair? pattern))
          (not (pair? input)))
      (eqv? pattern input)
      (and (pat-match (car pattern)
```

```
(car input))
(pat-match (cdr pattern)
  (cdr input)))))
```

Repräsentation der Variablen

Wie werden Variablen im Muster kenntlich gemacht?

Wir repräsentieren die Variablen im Muster durch Symbole, deren Namen mit einem Fragezeichen beginnen: ?X, ?Y usw.

```
(define (variable? x)
; Is x a variable? (a symbol beginning with ?)
(and (symbol? x)
  (char=? (string-ref(symbol->string x) 0) #\?)))

> (pat-match '(I need a ?X)
  '(I need a vacation)) → #t
> (pat-match '(I need a ?X)
  '(I need money)) → #f

(require racket/trace)
```

Ein trace: erfolgreicher Abgleich

```
> (trace pat-match) → (pat-match)
> (pat-match '(I need ?X) '(I need money))
| (pat-match (I need ?X) (I need money))
| (pat-match I I)
| #t
| (pat-match (need ?X) (need money))
| (pat-match need need)
| #t
| (pat-match (?X) (money))
| (pat-match ?X money)
| #t
| (pat-match () ())
| #t → #t
```

Ein trace: nicht erfolgreicher Abgleich

```
> (pat-match '(I need ?X)
  '(I really need money))
| (pat-match (I need ?X) (I really need money))
| (pat-match I I)
| #t
| (pat-match (need ?X) (really need money))
```

```
| (pat-match need really)
| #f
| #f → #f
```

Transformation der Eingabe

- Es wird zwar richtig erkannt, ob das Muster paßt oder nicht, aber wir erfahren nicht, welche Teile der Eingabe den Variablen entsprechen.
- *pat-match* muß so erweitert werden, daß wir eine *Liste der Variablenbindungen* als Resultat erhalten.
- Wenn wir diese Liste als Assoziationsliste zurückgeben, können wir die Transformation der Eingabe sehr bequem mit der **sublis**-Funktion aus dem tools-module vornehmen.

Sublis:

sublis ersetzt in einer Liste die Symbole einer assoc-Liste durch die assoziierten Werte.

```
> (sublis
  '((?X . vacation))
  '(what would it mean to you
    if you got a ?X))
→ (what would it mean to you
  if you got a vacation)
> (sublis
  '((?Y . you) (?X . vacation))
  '(what would it mean to ?Y
    if ?Y got a ?X))
→ (what would it mean to you
  if you got a vacation)

(define (sublis subs xs)
; substitute the elements of a linear list
(let ([replacement
      (lambda (x)
        (let ((found (assoc x subs)))
          (if (not found) x
              (cdr found))))]
  (map replacement xs)))
```

Prädikate und Semiprädicke

pat-match war bisher ein Prädikat. Das Resultat war **#t**, wenn der match erfolgreich war, sonst **#f**.

- Jetzt wollen wir im Falle eines erfolgreichen matches nicht nur **#t** erfahren, sondern eine *Assoziationsliste* der zugeordneten Variablen mit ihren Werten erhalten.
- Auch in dieser Form können wir *pat-match* wie ein Prädikat benutzen, da jeder Wert ungleich **#f** in logischen Ausdrücken **#t** bedeutet.

```
> (or 'auto #f)    → 'auto
> (and #t '((?x . vacation)))
      → '((?x . vacation))
```

Semiprädicke

Definition: 116 (Semiprädicke)

Semiprädicke sind Funktionen, die wie echte Prädikate in logischen Ausdrücken verwendet werden können, aber im True-Fall nicht das Symbol **#t**, sondern eine andere sinnvolle Information verschieden von **#f** zurückgeben.

Beispiele: member, memf

Das Semiprädicat-Problem

- Wenn Semiprädicke so definiert werden, daß sie im Erfolgsfall eine Liste der Erfolge zurückgeben und bei Mißerfolgen eben '(), kann ein Problem auftreten.
- Es kommt gelegentlich vor, daß auch im TRUE-Fall die Erfolgsliste leer ist. Dieses ist dann nicht vom Mißerfolgsfall zu unterscheiden.
- Bei unserem pattern matcher haben wir genau dieses Problem. Ein pattern match kann erfolgreich sein, ohne daß dabei Variablen an Muster gebunden werden, nämlich dann, wenn das Muster gar keine Variablen enthält.

Wir führen Konstanten ein, um die Resultate des pattern matching in den beiden Fällen unterscheiden zu können, wo die Variablenliste leer ist.

```
(define fail #f); Indicates pat-match failure
(define no-bindings '(#t . #t))
```

```
(define nb no-bindings)
; Indicates pat-match success,
; with no variables.
```

Ein erfolgreicher pattern match liefert jetzt immer ein Resultat verschieden von `#f`.

Akzessoren für Variablenbindungen

```
(define (get-binding var bindings)
;Find a (variable . value) pair in a binding list.
(assoc var bindings))
(define (binding-val binding)
;Get the value part of a single binding.
(cdr binding))
(define (lookup var bindings)
;Get the value part (for var) from a binding list.
(binding-val (get-binding var bindings)))
(define (extend-bindings var val bindings)
;Add a (var . value) pair to a binding list.
(cons (cons var val) bindings))
```

Koreferenz:

Eine Variable, die im Muster mehrfach vorkommt, sollte *an allen Positionen* an denselben Wert gebunden werden.

```
>(pat-match '(in ?X um ?X und um ?X herum)
            '(in Ulm um Ulm und um Ulm herum)) → #t
>(pat-match '(in ?X um ?X und um ?X herum)
            '(in Ulm um Rom und um Ulm herum)) → #f
```

Abhilfe: pat-match erhält die Liste der schon festgelegten Variablenbindungen als weiteren Parameter `bindings`.

Wir haben jetzt (fast) alle Zutaten für unseren ersten pattern matcher:

Wir müssen fünf Fälle unterscheiden:

- bindings = fail : Das Resultat ist ebenfalls fail , denn ein vorheriger (Teil)-match ist schon fehlgeschlagen.
- Das Muster ist eine einzelne Variable:
 - Die Variable ist schon gebunden und die Bindung paßt zur Eingabe: Gebe die bindings-Liste zurück.
 - Die Variable ist schon gebunden und die Bindung paßt nicht zur Eingabe: fail .
 - Die Variable ist noch nicht gebunden: Versuche eine Zuordnung zur Eingabe und erweitere gegebenenfalls bindings.
- Das Muster und die Eingabe sind beide Listen: pat-match wird rekursiv auf Kopf und Rumpf der Listen angewendet.

Die fünf Fälle:

```
(define (pat-match pattern input bindings)
  (cond [(eq? bindings fail) fail]
    [(variable? pattern)
      (match-variable
        pattern input bindings)]
    [(eqv? pattern input) bindings]
    [(and (pair? pattern) (pair? input))
      (pat-match
        (cdr pattern) (cdr input)
        (pat-match
          (car pattern) (car input)
          bindings))]
    [else fail]))
```

Abgleich von Variablen

```
(define (match-variable var input bindings)
; Does var match the input?
; Uses (or updates) and returns bindings."
  (let ((binding (get-binding var bindings)))
    (cond [(not binding)
      (extend-bindings
        var input bindings)])
```

```
[( equal? input
    (binding-val binding))
 bindings]
[else fail])))
```

Ein erster Test

```
> (pat-match '(I need a ?X)
  '(I need a vacation) nb)
((?X . vacation) (#t . #t))
```

Die Liste der Bindungen

- Das letzte Listenelement (\#t . \#t) ist lästig.
- Wir ändern extend-bindings, so daß der Wert für no-bindings gelöscht wird, sobald echte Bindungen vorliegen:

```
(define (extend-bindings var val bindings)
;Add a (var . value) pair to a binding list.
(cons (cons var val)
;; Once we add a "real" binding,
;; we can get rid of the dummy no-bindings
  (if (and (eq? bindings no-bindings))
    '()
    bindings)))
```

Test

```
> (pat-match
  '(in ?X um ?X und um ?X herum)
  '(in Ulm um Ulm und um Ulm herum) nb)
  → ((?X . Ulm))
> (pat-match
  '(in ?X um ?X und um ?X herum)
  '(in Ulm um Rom und um Ulm herum) nb)
  → #f
> (pat-match '(?X + ?X) '(1 + 1) nb)
  → ((?X . 1))
> (pat-match '(?X * ?X) '( (3 * 4) * ( 3 * 4)) nb)
  → ((?X 3 * 4))
```

Variablenersetzung

```
> (sublis
  (pat-match '(I need a ?X)
    '(I need a vacation) nb)
    '(what would it mean to you
      if you got a ?X))
  → (what would it mean to you
    if you got a vacation)

> (pat-match '(this is easy)
  '(this is easy) nb)
  → ((#t . #t))
```

Variablenersetzung

```
> (pat-match '(?X is ?X)
  '((2 + 2) is (2 + 2)))
  → '((?X 2 + 2))
;; gleichbedeutend '(?X . (2 + 2))
> (pat-match '(?P need . ?X)
  '(I need a long vacation))
  → '((?X a long vacation) (?P . I))
```

⌚ Anmerkung: Segment pattern matching

Wir haben bisher zwei Fälle von Variablenzuordnungen gesehen:

(?P need . ?X): ?X “matched” den ganzen Rest der Eingabe.

(?P): ?P kann nur einem einzelnen Element der Eingabeliste zugeordnet werden.

Im ELIZA-Programm sollte es aber möglich sein, an beliebiger Stelle des Musters längere Textsegmente einer Variablen zuzuordnen.

Dieses wird *segment matching* genannt. Wir führen eine zweite Art von Variablen ein, die Segmenten zugeordnet werden können. Norvig hat Variable in nicht-atomarer Form gewählt: Listen, die mit der Kennung ?* beginnen. *? soll an den Kleene-Stern erinnern.

Segment pattern matching

Nicht-atomare Variablen der Form (?* ?p) können nicht nur einem einzelnen Element des Eingabetextes zugeordnet werden, sondern einer Folge von Elementen, den Segmenten.

```
> (pat-match '(( ?* ?p) need (?* ?x) nb)
  '(Mr Hulot and I need a vacation))
→ ((?X A VACATION) (?P MR HULOT AND I))
```

Das recognizer Prädikat für Segment- Variable:

```
(define (starts-with list x)
  ; Is x a list whose car element is x?
  (and (pair? list) (eqv? (car list) x)))
```

```
(define (segment-pattern? pattern)
  ; Is this a segment matching pattern:
  ; ((?* var) . pat)
  (and (pair? pattern)
    (starts-with (car pattern) '?*)))
```

Pattern matching: Segmentvariable

```
(define (pat-match pattern input bindings)
  ; Match pattern against input and bindings"
  (cond [(eq? bindings fail) fail]
    [(variable? pattern)
      (match-variable pattern input bindings)]
    [(eqv? pattern input) bindings]
    [(segment-pattern? pattern); ***
```

```

(segment-match
  pattern input bindings 0)]; ***
[(and (pair? pattern) (pair? input))
 (pat-match (cdr pattern)
            (cdr input)
            (pat-match
              (car pattern) (car input)
              bindings))]

[else fail]))

```

Geschachtelte Rekursion

- Beim rekursiven Aufruf erhält pat-match die Variablenbindungen für den Mustervergleich des Kopfes des Musters als Argument.

```

(define (pat-match pattern input bindings)
;Match pattern against input and bindings
(cond [(eq? bindings fail) fail]
      [(variable? pattern) ...]
      [(eqv? pattern input) ...]
      [(segment-pattern? ...) ...]; ***
      [(and (pair? pattern) (pair? input))
       (pat-match (cdr pattern)
                  (cdr input)
                  (pat-match
                    (car pattern) (car input)
                    bindings))]

[else fail]))

```

Segmentgrenzen

```

(pat-match '((?* ?X) is a (?* ?Y))
           '(what he is is a fool) nb)

```

Wie können wir entscheiden, wieviel vom Eingabetext der Segmentvariablen zugeordnet werden soll?

```

(pat-match '((?* ?X) is a (?* ?Y))
           '(what he is is a fool) nb)

```

- Nach dem Segment muß direkt die nächste Textkonstante des Musters folgen, hier „is“.
- Suche die erste Position, an der **is** in der Eingabe auftaucht.

Wenn es nicht vorkommt, dann kann das Muster nicht passen \Rightarrow fail .

- Andernfalls versuche den Rest vom Muster mit dem Rest der Eingabe zu abzugleichen.

Wenn das nicht geht, verlängere das Segment und versuche es wieder.

 **Anmerkung:** Bei dieser Lösung wird angenommen, daß im Muster keine zwei Variablen direkt aufeinanderfolgen.

```
(define (segment-match pattern input bindings start)
  "Match_the_segment_pattern_((?*_var)_._pat)_against_input."
  (let ([var (cadr (car pattern))])
   [pat (cdr pattern)])
  (if (null? pat)
    (match-variable var input bindings)
    ;; We assume that pat starts with a constant
    ;; In other words,
    ;; a pattern can't have 2 consecutive vars
    (let ([pos (position (car pat) input start)])
      (if (not pos)
        fail
        (let ([b2 (pat-match
                    pat (drop input pos)
                    (match-variable var
                      (take input pos)
                      bindings))])
          ;; If this match failed, try another longer one
          (if (eq? b2 fail)
            (segment-match
              pattern input bindings (+ pos 1))
            b2)))))))
  >(pat-match '((?* ?X) is a (?* ?Y))
              '(what he is is a fool) nb)
  → ((?Y fool) (?X what he is))
  >(pat-match '((?* ?X) a b (?* ?X))
              '( 1 2 a b a b 1 2 a b) nb)
  → ((?X 1 2 a b))
```

24.1.2 Die Regelbasis und Dialogschleife

Die Regeln in Eliza

Der nächste Entwurfsschritt besteht darin, Eingabemustern mit Ausgabemustern zu assoziieren.

Format der Regeln:

Im Eliza-Programm ist das in Form einer Liste von Regeln geschehen, wobei die Regeln das Format hatten:

```
(Eingabemuster Antwort-Muster-1 ,  
 ..  
 Antwort-Muster-n)
```

Repräsentation der Regeln

Die Akzessoren für die Regeln:

```
(define (make-rule pattern response)  
  (list pattern response))  
(define (rule-pattern rule) (car rule))  
(define (rule-responses rule) (cdr rule))  
; Ein Beispiel  
(((?* ?x) I want (?* ?y))  
 (What would it mean if you got ?y)  
 (Why do you want ?y)  
 (Suppose you got ?y soon))
```

Organisation der Regeln

- Die Regeln werden in einer Suchliste zusammengefaßt.
- Dabei werden sie so geordnet,
 - daß die spezifischen Regeln an Anfang stehen
 - und allgemeine Regeln, die fast immer zutreffen, am Ende der Liste.
- Die Reihenfolge ersetzt *Prioritäten*, wie sie Weizenbaum in seinem ELIZA-Programm verwendet hat.

Einige Regeln

```
(define *eliza-rules*  
 '(((?* ?x) hello (?* ?y))  
 (How do you do .
```

```

Please state your problem.))
(((?* ?x) I want (?* ?y))
(What would it mean if you got ?y)
(Why do you want ?y)
(Suppose you got ?y soon))
(((?* ?x) if (?* ?y))
(Do you really think its likely that ?y)
(Do you wish that ?y)
(What do you think about ?y)
(Really-- if ?y))

(((?* ?x) no (?* ?y))
(Why not?)
(You are being a bit negative)
(Are you saying "NO"
just to be negative?))
(((?* ?x) I was (?* ?y))
(Were you really?)
(Perhaps I already knew you were ?y)
(Why do you tell me you were ?y now?))
(((?* ?x) I feel (?* ?y))
(Do you often feel ?y ?))
(((?* ?x) I felt (?* ?y))
(What other feelings do you have?)))

```

Die ELIZA-Dialogschleife

In einer Schleife wird

- ein *prompt*-Zeichen gedruckt,
- die Eingabe mit **read** gelesen,
- eine passende Regel gesucht,
- nach den Regeln eine Antwort generiert
- und diese mit **writeln** ausgegeben,
- solange bis Patientin oder Patient den Dialog mit "(Bye)" beenden.

Das Semiprädikat „some“ in Common Lisp sucht das erste Element einer Liste das ein übergebenes Semiprädikat erfüllt und gibt das Resultat der Prädikatsanwendung zurück. Wir verwenden es hier, um bei der Suche mit dem pattern matcher in der Liste der Regeln gleich die Liste der Variablenbindungen zurückzuerhalten und nicht nur die passende Regel. Wir könnten zwar auch mittels der Racket-Funktion „memf“ suchen, aber dann müßten wir diese nochmals anwenden, um dadurch die Variablenbindungen zu erhalten. Norvig, 1992 hat in seinen Fallstudien die Funktion „some“ regelmäßig verwendet, wenn mit einem Semiprädikat in einer Liste gesucht wird und das Ergebnis des Tests weiterverwendet werden soll. Wir haben die Common Lisp Funktion „some“ hier mit einem Präfix „cl:“ aus der Swindle-Library importiert.

```
(require (prefix-in cl: (only-in swindle some)))
```

Transformation der Eingabe

```
(define (use-eliza-rules input)
; Pick some rule to transform the input.
(cl:some
(lambda (rule)
(let ((result (pat-match
(rule-pattern rule)
input
no-bindings)))
(if (not (eqv? result fail))
(sublis (switch-viewpoint result)
(random-elt
(rule-responses rule)))
fail)))
*eliza-rules*))
```

Die Dialogschleife

```
(define (eliza)
; Respond to the user input
; using pattern matching rules.
(let ([dialog
(lambda (x)
(display "eliza>_")
(let ([input (read)])
(writeln
(flatten
(use-eliza-rules input))))
```

```

        input))])
(iter-until
  dialog
  (curry equal? '(bye)) #f))
'ciao!)
```

Die Rolle im Dialog

Wenn Eingabe *persönliche Fürwörter* enthält, *I*, *You* usw., dann werden sie in der Rückfrage so ersetzt, daß die Referenzen richtig bleiben, auch wenn ELIZA den Satz sagt.

```

(define (switch-viewpoint words)
; Change I to you and vice versa, and so on.
  (sublis '((I . you)
            (you . I)
            (me . you)
            (am . are))
         words))
```

Probelauf

```

> (eliza) ; ein Beispieldialog
eliza> (hello there)
(How do you do. Please state your problem.)
eliza> (I want to test this program)
(Why do you want to test this program)
eliza> (I could see if it works)
(Really-- if it works)
eliza> (no not really)
(You are being a bit negative)
eliza> (no)
(Are you saying NO just to be negative?)
eliza> (forget it I was wondering
how general the program is)
(Were you really?)

eliza> (forget it I was wondering
how general the program is)
(Why do you tell me you were wondering
how general the program is now?)
eliza> (I felt like it)
(What other feelings do you have?)
```

```

eliza> (I feel this is enough)
(Do you often feel this is enough ?)
eliza> (I do)
(I am not sure I understand you fully)
eliza> (bye)
(bye)
'ciao!

```

Beispiel: 117

Das vollständige Eliza-Programm: Diese Programm finden Sie in der se3-bib im Modul eliza-module.rkt. Der pattern matcher wurde als eigenes Modul pattern-matching-module.rkt zur Verfügung gestellt. Außerdem wurde für eine einfache Eingabe ein gui angeschlossen und die Ausgabe von Eliza auf die Sprachausgabe des Macintosh geleitet.

```

#lang racket
(provide
  *eliza-rules* *eliza-simple* *eliza-fancy*
  eliza eliza-simple eliza-fancy eliza-tool eliza-with-tool
  use-eliza-rules
  eliza-demo *elizas-voice* *patients-voice*
  )

(require
  se3-bib/tools-module
  se3-bib/demo-gui-module
  se3-bib/macos-module
  se3-bib/string-module
  se3-bib/pattern-matching-module
  (prefix-in cl: (only-in swindle some)))
; die Common Lisp Variante von "some"
; gibt das Resultat der Funktionsanwendung
; für das erste Element der Liste zurück,
; das das übergebene Semipräädikat erfüllt.
)
(define *eliza-simple*
  '((((* ?x) hello (* ?y))
     (How do you do. Please state your problem .))
    ((bye)(bye))
    (((* ?x) I want (* ?y))
     (What would it mean if you got ?y)))

```

```

(Why do you want ?y)(Suppose you got ?y soon))
(((?* ?x) if (?* ?y))
 (Do you really think its likely that ?y)
 (Do you wish that ?y)
 (What do you think about ?y) (Really-- if ?y))
(((?* ?x) no (?* ?y))
 (Why not?) (You are being a bit negative)
 (Are you saying "NO" just to be negative?))
(((?* ?x) I was (?* ?y))
 (Were you really?) (Perhaps I already knew you were ?y)
 (Why do you tell me you were ?y now?))
(((?* ?x) I feel (?* ?y))
 (Do you often feel ?y ?))
(((?* ?x) I felt (?* ?y))
 (What other feelings do you have?)))
;; =====

(define (use-eliza-rules input)
 "Find some rule with which to transform the input."
 (cl:some
  (lambda (rule)
   (let ((result (pat-match
                  (rule-pattern rule)
                  input
                  no-bindings)))
    (if (not (eqv? result fail))
         (sublis (switch-viewpoint result)
                  (random-elt (rule-responses rule)))
         fail)))
   *eliza-rules*)))

(define (switch-viewpoint words)
 "Change I to you and vice versa , and so on."
 (sublis '((I . you) (you . I) (me . you) (am . are))
   words))

;;> (eliza)
;;ELIZA> (hello there)
;;;(HOW DO YOU DO. PLEASE STATE YOUR PROBLEM.)
;;ELIZA> (i want to test this program)
;;;(WHAT WOULD IT MEAN IF YOU GOT TO TEST THIS PROGRAM)

```

```

; ; ; ELIZA> ( i could see if it works)
; ; ;(DO YOU REALLY THINK ITS LIKELY THAT IT WORKS)
; ; ; ELIZA> (no not really)
; ; ;(ARE YOU SAYING "NO" JUST TO BE NEGATIVE?)
; ; ; ELIZA> (no)
; ; ;(ARE YOU SAYING "NO" JUST TO BE NEGATIVE?)
; ; ; ELIZA> (forget it-- i was wondering how general the program is)
; ; ;(WHY DO YOU TELL ME YOU WERE WONDERING
; ; ; HOW GENERAL THE PROGRAM IS NOW?)
; ; ; ELIZA> (i felt like it)
; ; ;(WHAT OTHER FEELINGS DO YOU HAVE?)
; ; ; ELIZA> (i feel this is enough)
; ; ;(DO YOU OFTEN FEEL THIS IS ENOUGH ?)
; ; ; ELIZA> [Abort]

(define *eliza-rules*
'(((bye)(bye))
 (((?* ?x) hello (?* ?y))
  (How do you do. Please state your problem.))
 (((?* ?x) computer (?* ?y))
  (Do computers worry you?) (What do you think about machines?))
  ((Why do you mention computers?))
  ((What do you think machines have to do with your problem?))
 (((?* ?x) computers (?* ?y))
  (Do computers worry you?) (What do you think about machines?))
  ((Why do you mention computers?))
  ((What do you think machines have to do with your problem?))
 (((?* ?x) name (?* ?y))
  (I am not interested in names))
 (((?* ?x) sorry (?* ?y))
  (Please do not apologize) (Apologies are not necessary))
  ((What feelings do you have when you apologize))
 (((?* ?x) I remember (?* ?y))
  (Do you often think of ?y)
  ((Does thinking of ?y bring anything else to mind?))
  ((What else do you remember) (Why do you recall ?y right now?))
  ((What in the present situation reminds you of ?y))
  ((What is the connection between me and ?y)))
 (((?* ?x) do you remember (?* ?y))
  (Did you think I would forget ?y ?)
  ((Why do you think I should recall ?y now))

```

(What about ?y) (You mentioned ?y))
 (((?* ?x) **if** (?* ?y))
 (**Do** you really think its likely that ?y)
 (**Do** you wish that ?y)
 (What **do** you think about ?y) (Really-- **if** ?y))

 (((?* ?x) I dreamt (?* ?y))
 (Really-- ?y)
 (Have you ever fantasized ?y while you were awake?)
 (Have you dreamt ?y before?))
 (((?* ?x) dream about (?* ?y))
 (How **do** you feel about ?y in reality?))
 (((?* ?x) dream (?* ?y))
 (What does this dream suggest to you?)
 (**Do** you dream often?)
 (What persons appear in your dreams?)
 (**Do** you believe that dream has to **do** with your problem?))
 (((?* ?x) my mother (?* ?y))
 (Who **else** in your family ?y)
 (Tell me more about your family))
 (((?* ?x) my father (?* ?y))
 (Your father) (Does he influence you strongly?)
 (What **else** comes to mind **when** you think of your father?))

 (((?* ?x) I want (?* ?y))
 (What would it mean **if** you got ?y)
 (Why **do** you want ?y) (Suppose you got ?y soon))
 (((?* ?x) I need (?* ?y))
 (What would it mean **if** you got ?y)
 (Why **do** you want ?y) (Suppose you got ?y soon))
 (((?* ?x) I am glad (?* ?y))
 (How have I helped you to be ?y)
 (What makes you happy just now)
 (Can you explain why you are suddenly ?y))
 (((?* ?x) I am sad (?* ?y))
 (I am sorry to hear you are depressed)
 (I'm sure its **not** pleasant to be sad))
 (((?* ?x) are like (?* ?y))
 (What resemblance **do** you see between ?x **and** ?y))
 (((?* ?x) is like (?* ?y))
 (In what way is it that ?x is like ?y))

(What resemblance **do** you see?)
(Could there really be **some** connection?) (How?)
(((?* ?x) alike (?* ?y))
(In what way?) (What similarities are there?))
(((?* ?x) same (?* ?y))
(What other connections **do** you see?))

(((?* ?x) I was (?* ?y))
(Were you really?) (Perhaps I already knew you were ?y)
(Why **do** you tell me you were ?y now?))
(((?* ?x) was I (?* ?y))
(What **if** you were ?y ?) (**Do** you think you were ?y)
(What would it mean **if** you were ?y))
(((?* ?x) I am (?* ?y))
(In what way are you ?y) (**Do** you want to be ?y ?))
(((?* ?x) am I (?* ?y))
(**Do** you believe you are ?y) (Would you want to be ?y)
(You wish I would tell you you are ?y)
(What would it mean **if** you were ?y))
(((?* ?x) am (?* ?y))
(Why **do** you say "AM?") (I don't understand that))
(((?* ?x) are you (?* ?y))
(Why are you interested in whether I am ?y **or** not?)
(Would you prefer **if** I weren't ?y)
(Perhaps I am ?y in your fantasies))
(((?* ?x) you are (?* ?y))
(What makes you think I am ?y ?))

(((?* ?x) because (?* ?y))
(Is that the real reason?)
(What other reasons might there be?)
(Does that reason seem to explain anything else?))
(((?* ?x) were you (?* ?y))
(Perhaps I was ?y) (What **do** you think?)
(What **if** I had been ?y))
(((?* ?x) I cannot (?* ?y))
(Maybe you could ?y now)
(What **if** you could ?y ?))
(((?* ?x) I feel (?* ?y))
(**Do** you often feel ?y ?))
(((?* ?x) I felt (?* ?y)))

(What other feelings **do** you have?))
(((?* ?x) I (?* ?y) you (?* ?z))
(Perhaps in your fantasy we ?y each other))
(((?* ?x) why don't you (?* ?y))
(Should you ?y yourself?)
(**Do** you believe I **do not** ?y)
(Perhaps I will ?y in good **time**))
(((?* ?x) yes (?* ?y))
(You seem quite positive) (You are sure) (I understand))
(((?* ?x) no (?* ?y))
(Why not?) (You are being a **bit** negative)
(Are you saying NO just to be negative?))

(((?* ?x) someone (?* ?y))
(Can you be more specific?))
((?* ?x) everyone (?* ?y))
(surely **not** everyone)
(Can you think of anyone in particular?)
(Who for example?)
(You are thinking of a special person))
((?* ?x) always (?* ?y))
(Can you think of a specific example) (When?)
(What incident are you thinking of?)
(Really-- always))
(((?* ?x) what (?* ?y))
(Why **do** you ask?)
(Does that question interest you?)
(What is it you really want to know?)
(What **do** you think?)
(What comes to your mind **when** you ask that?))
(((?* ?x) perhaps (?* ?y))
(You **do not** seem quite certain))
(((?* ?x) are (?* ?y))
(Did you think they might **not** be ?y)
(Possibly they are ?y))
(((?* ?x) bye (?* ?y))
(See you later alligator!))
(((?* ?x)))
(Very interesting) (I am **not** sure I understand you fully)
(What does that suggest to you?)
(Please **continue**) (Go on)

```

        (Do you feel strongly about discussing such things?))

(define *eliza-fancy* *eliza-rules*)

(define(eliza-simple)
  (set! *eliza-rules* *eliza-simple*)
  (eliza)
  )
(define(eliza-fancy)
  (set! *eliza-rules* *eliza-fancy*)
  (eliza))

(define (eliza)
  "Respond_to_user_input_using_pattern_matching_rules."
  (display '(please enter your question as a list of symbols))
  (display "\n")
  (let ([dialog
        (lambda (x)
          (display "eliza>_")
          (let ([input (read)])
            (writeln
              (flatten (use-eliza-rules input)))
            input))])
    (iter-until
      dialog
      (curry equal? '(bye)) #f))
  'ciao!)

(define (get-patient)
  (display "eliza>_")
  (read))

(define (show-answer inp)
  (writeln inp))

(define (eliza-tool
         the-rules
         get-the-patients-remark
         ; a procedure returning a list
         show-elizas-answer

```

```

; a procedure accepting a list
)
"Respond_to_user_input_using_pattern_matching_rules."
(set! *eliza-rules* the-rules)
(let ((dialog
      (lambda (x)
        (let ((input (get-the-patients-remark)))
          (show-elizas-answer
            (flatten (use-eliza-rules input)))
            input))))
  (iter-until dialog (curry equal? '(bye)) #f) 'ciao!))

(define (eliza-with-tool)
  (eliza-tool
    *eliza-fancy*
    get-patient
    ; a procedure returning a list
    show-answer
    ; a procedure accepting a list
  )))

(define *elizas-voice* Victoria)
(define *patients-voice* Fred)

(define (transform-question question-string)
  (let* ([the-question-as-list
         (string->list-of-symbols question-string)]
         [the-answer-as-list
           (flatten (use-eliza-rules the-question-as-list))]
         [the-answer-as-string
           (list-of-symbols->string the-answer-as-list)])
    (*patients-voice* question-string)
    (*elizas-voice* the-answer-as-string)
    the-answer-as-string))

(define (eliza-demo)
  (dialog-gui
    transform-question
    "The_doctor_is_in._Please_come_in."))
)

```

Beispiel: 118

Das pattern-matching-Modul:

```
#lang racket
(provide fail no-bindings nb
         pat-match get-binding extend-bindings
         binding-val binding-var lookup
         rule-pattern rule-responses rule-response
         make-rule variable?      )

(require swindle/extra; fuer amb
         racket/trace
         se3-bib/tools-module)

;; =====
;; Pattern matching
;; =====

(define (variable? x)
  "Is x a variable (a symbol beginning with '?')?"
  (and (symbol? x)
       (char=? (string-ref (symbol->string x) 0) #\?)))

;; =====

(define fail #f); Indicates pat-match failure

(define no-bindings '((#t . #t)))
; Indicates pat-match success, with no variables.
(define nb no-bindings); abbreviation
;; =====

(define (get-binding var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))

(define (binding-val binding)
  "Get the value part of a single binding."
  (cdr binding))

(define (binding-var binding)
```

```

"Get_the_variable_part_of_a_single_binding."
(car binding))

(define (lookup var bindings)
  "Get_the_value_part_(for_var)_from_a_binding_list."
  (binding-val (get-binding var bindings)))

;; =====

(define (match-variable var input bindings)
  "Does_VAR_match_input?_Uses_(or_updates)_and_returns_bindings."
  (let ((binding (get-binding var bindings)))
    (cond [(not binding)
            (extend-bindings var input bindings)]
          [(equal? input
                  (binding-val binding)) bindings]
          [else fail])))

;; =====

(define (extend-bindings var val bindings)
  "Add_a_(var_.value)_pair_to_a_binding_list."
  (cons (cons var val)
        ;; Once we add a "real" binding,
        ;; we can get rid of the dummy no-bindings
        (if (and (eq? bindings no-bindings))
            '()
            bindings)))

;; =====

(define (pat-match pattern input bindings)
  "Match_pattern_against_input_in_the_context_of_the_bindings"
  (cond [(eq? bindings fail) fail]
        [(variable? pattern)
         (match-variable pattern input bindings)]
        [(eqv? pattern input) bindings]
        [(segment-pattern? pattern)           ; ***
         (segment-match
          pattern input bindings 0)]      ; ***
        [(% (and (pair? pattern) (pair? input)))]))

```

```

        (pat-match (cdr pattern)
                    (cdr input)
                    (pat-match
                        (car pattern) (car input)
                        bindings)))
    [else fail])))

(define (starts-with list x)
  "Is_x_a_list_whose_car_element_is_x?"
  (and (pair? list) (equal? (car list) x)))

(define (segment-pattern? pattern)
  "Is_this_a_segment_matching_pattern_((?*_var).._pat)"
  (and (pair? pattern)
        (starts-with (car pattern) '?*)))

(define (segment-match pattern input bindings start)
  "Match_the_segment_pattern_((?*_var).._pat)_against_input."
  (let ([var (cadr (car pattern))])
    [pat (cdr pattern)])
  (if (null? pat)
      (match-variable var input bindings)
      ;; We assume that pat starts with a constant
      ;; In other words, a pattern can't have 2 consecutive va
      (let ([pos (position (car pat) input start)])
        (if (not pos)
            fail
            (let ([b2 (pat-match
                      pat (drop input pos)
                      (match-variable var
                                      ;(take (add1 pos) input) ; bug?
                                      (take input pos) ; bug?
                                      bindings))])
              ;; If this match failed, try another longer one
              (if (eq? b2 fail)
                  (segment-match pattern input bindings (+ pos
b2)))))))
  ;;; =====

(define (rule-pattern rule) (car rule))

```

```

(define (rule-responses rule) (cdr rule))
(define (rule-response rule) (cadr rule))
(define (make-rule pattern response)
  (list pattern response))
;; =====

; test:
;(pat-match '(I need a ?X) '(I need a vacation) nb)

```

> (eliza)

ELIZA> Hello , I am Leonie .

HOW DO YOU DO. PLEASE STATE YOUR PROBLEM.

ELIZA> And who are you?

Would you please introduce yourself?

WHY ARE YOU INTERESTED IN WHETHER I AM OR NOT?

ELIZA> To be, or not to be: that is the question .

I AM NOT SURE I UNDERSTAND YOU FULLY

ELIZA> Please tell me, who you are.

WHAT MAKES YOU THINK I AM ?

ELIZA> I think , therefore I am.

IN WHAT WAY ARE YOU

ELIZA> Whence are we, and why are we?

Of what scene the actors and spectators?

PERHAPS I AM IN YOUR FANTASIES

ELIZA> I have a dream.

DON'T YOU BELIEVE THAT DREAM HAS TO DO WITH YOUR PROBLEM?

ELIZA> The quick Dreams , the passion-winged ministers of Thought .

WHAT PERSONS APPEAR IN YOUR DREAMS?

ELIZA> Martin Luther King: One day this nation will rise up , live out the true meaning of its creed: we hold these truths to be self-evident that all man are created equal .

DID YOU THINK THEY MIGHT NOT BE CREATED EQUAL

ELIZA> All animals are equal but some are more equal than others .

POSSIBLY THEY ARE CREATED EQUAL ALL ANIMALS ARE EQUAL BUT SOME ARE MORE EQUAL THAN OTHERS

ELIZA> From each according to his abilities -

to each according to his needs.

WHAT DOES THAT SUGGEST TO YOU?

ELIZA> The history of all hitherto existing society
is the history of class struggle (Karl Marx)

VERY INTERESTING

ELIZA> A riot is at bottom the language of the unheard. (M.L.K.)

WHAT DOES THAT SUGGEST TO YOU?

ELIZA> I think the necessity of being ready increases.—
look to

it. (A. Lincoln)

PLEASE **CONTINUE**

ELIZA> The ballot is stronger than the bullet (Abraham Lincoln).

GO ON

ELIZA> The philosophers have only interpreted the world
in various ways – the point is to change it.

DO YOU FEEL STRONGLY ABOUT DISCUSSING SUCH THINGS?

ELIZA> You have **not** a single clue what i am talking about
— **do** you?

IN WHAT WAY ARE YOU TALKING ABOUT **DO** I

ELIZA> You can fool all the people **some** of the **time**
and some people all the **time**,

but you cannot fool all the people all of the **time**.

I AM **NOT** SURE I UNDERSTAND YOU FULLY

ELIZA> Good old Eliza , you are the one fixed point
in a changing age.

WHAT MAKES YOU THINK I AM THE ONE FIXED POINT IN A
CHANGING AGE ?

ELIZA> You see but you **do not** observe .

I AM **NOT** SURE I UNDERSTAND YOU FULLY

ELIZA> Computers are useless , Picasso said .

They can only give you answers .

You dont even give answers .

POSSIBLY THEY ARE USELESS PICASSO SAID THEY CAN ONLY
GIVE I ANSWERS I DONT EVEN GIVE ANSWERS

ELIZA> Are you a parrot?

WHY ARE YOU INTERESTED IN WHETHER I AM A PARROT **OR**
NOT?

ELIZA> There is but one **step** from the grotesque
to the horrible (Conan Doyle)

GO ON

ELIZA> **And** now for something completely different:

Let us talk about science.

GO ON

ELIZA> All science is either physics **or** stamp collecting.

PLEASE CONTINUE

ELIZA> You know who critics are?

The men who failed in literature **and** art.

DID YOU THINK THEY MIGHT **NOT** BE THE MEN

WHO FAILED IN LITERATURE AND ART

ELIZA> Silly question.

PLEASE CONTINUE

ELIZA> What **do** think about statistics?

WHY DO YOU ASK?

ELIZA> There are three kinds of lies: lies—

damned lies **and** statistics. (Disraeli)

DID YOU THINK THEY MIGHT **NOT** BE THREE KINDS OF LIES

LIES DAMNED LIES AND STATISTICS

ELIZA> No, **not** really

ARE YOU SAYING NO JUST TO BE NEGATIVE?

ELIZA> I never deny, I never contradict, I sometimes forget.

DO YOU WANT TO BE SAYING ?

ELIZA> Ich bin der Geist, der stets verneint.

VERY INTERESTING

ELIZA> Why don't you pay attention?

I believe you don't understand a single word I am saying.

DO YOU BELIEVE I DON'T UNDERSTAND

ELIZA> Be **not** faithless but believing.

I AM NOT SURE I UNDERSTAND YOU FULLY

ELIZA> Believing where we cannot prove —

DO YOU FEEL STRONGLY ABOUT DISCUSSING SUCH THINGS?

ELIZA> To burn always with this hard, gemlike flame,
to maintain this ecstasy, is success in life (Walter Pater).

CAN YOU THINK OF A SPECIFIC EXAMPLE

ELIZA> I pant, I sink, I tremble, I expire!

REALLY— ALWAYS

ELIZA> Always look at the bright side of life.

WHAT INCIDENT ARE YOU THINKING OF?

ELIZA> I never had a peace of toast

particularly long **and** wide,

but fell upon the sanded **floor**,

and always on the buttered side.

WHEN?

ELIZA> As I said , always .
REALLY-- ALWAYS
ELIZA> I always have a quotation ready ,
it saves original thinking .
CAN YOU THINK OF A SPECIFIC EXAMPLE
ELIZA> Don't think twice , it 's all right . --
Just forget it .
PLEASE **CONTINUE**
ELIZA> **And** may there be no sadness of farewell .
Good night .
ARE YOU SAYING NO JUST TO BE NEGATIVE ?
^C

24.2 Kontrollabstraktion und Werkzeuge

Kontrollabstraktion und Werkzeuge



Eliza

22 Musterabgleich (pattern matching)

- Eliza: Ein regelbasierter Übersetzer
- Kontrollabstraktion und Werkzeuge
- STUDENT: Algebraische Probleme

23 Means-Ends-Analyse: GPS

24 GPS Anwendungen

Navigation icons: back, forward, search, etc.

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 689 / 1157

Kontrollabstraktion und Werkzeuge

- Wir haben schon am Beispiel des backtracking gesehen, wie wir den typischen Ablauf zu einem Werkzeug general–backtracking abstrahieren konnten.
- ELIZA ist ein Prototyp für einen *regelbasierten Übersetzer*.
- Wir werden aus dem ELIZA-Programm zwei weitere Werkzeuge extrahieren:
 1. Ein Werkzeug *rule-based-translator* für die regelbasierte Texttransformation
 2. und ein Werkzeug *dialog-tool* für eine Dialogschleife.
- Beide Werkzeuge werden wieder Funktionen höherer Ordnung sein.

☞ **Anmerkung:** Auch diese Werkzeuge sind aus dem Buch von Norvig, 1992 übernommen und für diese Vorlesung nach Racket portiert worden. Sie finden diese in der se3-bib im Modul translator-module.rkt und

Ein regelbasierter Übersetzer

Der Kern von ELIZA leistet folgendes:

- Suche eine Regel, für die das Muster zur Eingabe paßt.
- Wende die Regel an und berechne die Variablenbindungen.
- Transformiere das Resultat und ersetze die Variable.

Beispiel: 119 (Ein regelbasierter Übersetzer)

Wir werden diesen Kern als Funktion höherer Ordnung *rule-based-translator* realisieren.

Die Parameter

- Unser Werkzeug soll generisch sein; wir wollen den Typ der Regeln nicht festlegen.
- Deshalb wird der Regeltyp als abstrakter Datentyp über die Akzessorfunktionen *rule-if* und *rule-then* definiert.
- Die weiteren Parameter:

input: Die zu transformierend Eingabe

rules: Die Regeln

action: Eine Funktion zur Transformation des Resultats

bindings: Zu berücksichtigende Variablenbindungen

Die Signatur

```
(define (rule-based-translator
    input ; list
    rules ; list of rules
    matcher; Semipredicate:
    ;pattern , (words) , (bindngs)->(bindngs)
    rule-if ; procedure: rule -> list
    rule-then; procedure: rule -> list
    action ; procedure: list list -> list
    bindings
))
```

Die Transformation

```
;Find the first rule in rules that matches
input,;apply the action to that rule.
(cl:some
  (lambda (rule)
    (let ([result (matcher (rule-if rule)
                           input
                           bindings)])
      (if (not (eqv? result fail))
          (let ([r
                (action result
                        (rule-then rule))])
            r)
          #f)))
  rules))
```

Ein Beispiel: Negieren eines Satzes

```
(define *negation-rules*
  '((( no (?* ?y))
     ( yes ?y) )
    (( (?* ?x) do not (?* ?y))
     ( ?x do ?y ) )
    (( (?* ?x) do (?* ?y))
     ( ?x do not ?y ) )
    (( (?* ?x) is not (?* ?y))
     ( ?x is ?y) )
    (((?* ?x) is (?* ?y))
     ( ?x is not ?y) )
    (((?* ?x) often (?* ?y))
     ( ?x seldom ?y) )
    (((?* ?x) never (?* ?y))
     ( ?x always ?y) )
    ....
    (((?* ?x) always (?* ?y))
     ( ?x never ?y) )
    (((?* ?x) love (?* ?y))
     ( ?x hate ?y) )
    (((?* ?x) hate (?* ?y))
     ( ?x love ?y) )
    (((?* ?x) dont (?* ?y)))
```

```
( ?x do ?y )  
((bye) (bye))  
))
```

Der Aufruf

```
(define (the-opposite words)
; negate a sentence using negation rules
; (negate: list-of-symbols
; → list-of-symbols
(rule-based-translator
words; input
*negation-rules*; rules
pat-match; matcher;
car; rule-if
cadr; rule-then
(compose flatten sublis); action
no-bindings
))
```

Ein allgemeines Dialog-Werkzeug

Beispiel: 120 (Abstraktion der Dialogschleife)

Eine Funktion höherer Ordnung, mit Funktionsparametern für die Teilaufgaben.

- Lese die Eingabe
- Transformiere die Eingabe
- Zeige das Resultat an

```
(define (dialog-tool
get-the-input
; a procedure returning a list
transformer
; a procedure accepting a list,
; returning a list
show-the-result
; a procedure accepting a list
)
; Respond to user input using pattern matching.
(let ((dialog
(lambda (x)
```

```
(show-the-result
  (transformer
    (get-the-input))))))
( iter-until dialog (curry equal? '(bye)) #f)
  'ciao!)))
```

Die Eingabeschleife für den Verneiner

```
(define (negation-loop)
  (dialog-tool
    (lambda ()
      (display "negator>_")
      (read))
    the-opposite
    (lambda (r) (writeln r) r)))
)
```

☞ Anmerkung: Die beiden Werkzeuge dialog–tool und rule–based–translator sind im teachpack se3–bib/translator–module.rkt verfügbar, ebenso das Beispiel negation–loop.

Ein Beispiellauf

```
> (require se3–bib/translator–module)
> (negation-loop)
negator> (I do not like ice cream)
→ (I do like ice cream)
negator> (always look at the bright side
of life)
→ (never look on the bright side of life)
negator> (dead men dont ware plaid)
→ (dead men do ware plaid)
negator> (I do love Racket)
→ (I do not love Racket)
negator> (I love Java) → (I hate Java)
negator> (bye) → (bye)
ciao!
```

 **Datengesteuerte Programmierung**

Dank der drei Werzeuge:

1. pat-match,
2. rule-based-translator
3. und dialog-tool

die den Ablauf eines Übersetzungsprogramms definieren, brauchen wir für neue Übersetzungsaufgaben nur noch die Regeln zu definieren, ohne neue Funktionen für den Ablauf schreiben zu müssen.

Trennung von Ablauf und Daten

Wir haben hier die für die *deklarative* Programmierung charakteristische Trennung von Ablauf und Daten:

- Der Ablauf nutzt vordefinierte Strategien.
- Die Programme sind *datengesteuert*.

24.3 STUDENT: Algebraische Probleme

STUDENT: Algebraische Probleme



STUDENT

- 22 Musterabgleich (pattern matching)
 - Eliza: Ein regelbasierter Übersetzer
 - Kontrollabstraktion und Werkzeuge
 - STUDENT: Algebraische Probleme
- 23 Means-Ends-Analyse: GPS
- 24 GPS Anwendungen

Leonie Dreschler-Fischer (Department Inform Softwareentwicklung III WS 2019/2020 704 / 1157

STUDENT: Algebraische Probleme



Das Programm STUDENT [Bobrow, 1968] verwendete Musterabgleich für die Lösung von Textaufgaben zu algebraischen Problemen.

- ▶ Wir werden an einer vereinfachten Fassung sehen,
 - ▶ wie durch pattern matching aus Textaufgaben die Gleichungen extrahiert werden können
 - ▶ und wie die Gleichungssysteme gelöst werden können.
- ▶ Dafür werden wir wieder das Werkzeug „rule-based-translator“ nutzen.

Leonie Dreschler-Fischer (Department Inform Softwareentwicklung III WS 2019/2020 704 / 1157

STUDENT war die Dissertation von Bobrow, 1968.

Ein Beispiel

Problem: 121 (Gegeben sei die Textaufgabe:)

Fran's age divided by Robin's height is one half Kelly's IQ. Kelly's IQ minus 80 is Robin's height. If Robin is 4 feet tall, how old is Fran?

Gesucht:

- Ein Verfahren, um automatisch die definierenden Gleichungen zu finden,
- die Gleichungen zu lösen
- und die Lösung anzuzeigen.

Repräsentation des Aufgabentextes

- Der Text wird als Liste von Symbolen repräsentiert.
- Komma und Punkt dürfen allerdings in Listen nicht verwendet werden, da sie eine besondere Bedeutung haben.
- Wir schließen die Satzzeichen daher zwischen senkrechten Strichen ein:

```
'(Fran 's age divided by Robin 's height  
is one half Kelly 's IQ |.|  
Kelly 's IQ minus 80 is Robin 's height |.|  
If Robin is 4 feet tall |,|  
how old is Fran ?))
```

Die lexikalische Analyse

Um den Musterabgleich zu erleichtern, bereinigen wir den Text um alle Füllworte, die nicht zur Lösung beitragen:

```
(define (noise-word? word)  
; Is this a low-content word  
; which can be safely ignored?  
(member word '(a an the this number of $)))
```

Groß-Klein-Schreibung

- Die Muster wollen wir nur einmal für Kleinschreibung definieren.
- Deshalb transformieren wir alle Schlüsselwörter in kleingeschriebene Symbole.

```
(define *to-lower*
  '((If . if )
    (Then . then)
    (The . the) ....
  )))
(define (translate-upper sentence)
  (sublis *to-lower* sentence))
```

Zahlkonstanten

- Zahlwörter werden durch Ziffern ersetzt.

```
(define *numbers*
  '((zero . 0 )
    (one . 1)
    (two . 2)
    (three . 3)
    (four . 4)
    (five . 5)
    (six . 6)
    (seven . 7)
    (eight . 8)
    (nine . 9)
    (ten . 10)))
(define (translate-numbers sentence)
  (sublis *numbers* sentence))
```

Die lexikalische Analyse

```
(define (lexically-scan-words words)
  (translate-numbers
    (filter (negate noise-word?)
      (translate-upper words))))
> (lexically-scan-words
   '(What do you get
     when you multiply
     the number six by seven ?))
--> (what do you get
     when you multiply 6 by 7 ?)
```

Aufstellen der Gleichungen

- Zur Repräsentation der Gleichungen werden wir die *expressions* von Scheme verwenden.
- Für die Transformation des geschriebenen Textes zu Racket-Ausdrücken wird der *pattern matcher* verwendet:

```
((1 half ?x*)      →      (/ ?x 2))
((twice ?x*)       →      (* 2 ?x))
```

?x* steht für ein Segmentmuster (?* ?x).

Einige Regeln für den Musterabgleich

```
(define *student-rules*
  (map expand-abbrivs
    '(((?x* equals ?y*)          (= ?x ?y))
      ((?x* same as ?y*)        (= ?x ?y))
      ((?x* = ?y*)              (= ?x ?y))
      ((?x* is equal to ?y*)   (= ?x ?y))
      ((?x* is ?y*)             (= ?x ?y))
      ((?x* when you ?y*)     (= ?x ?y))
      ((?x* - ?y*)              (- ?x ?y))
      ((?x* minus ?y*)         (- ?x ?y))
      ((difference between ?x* and ?y*)
       (- ?y ?x))
      ((difference ?x* and ?y*)
       (- ?y ?x)))
      ((?x* + ?y*)              (+ ?x ?y))))
```

Einige Regeln für den Musterabgleich

```
((?x* plus ?y*)           (+ ?x ?y))
((sum ?x* and ?y*)        (+ ?x ?y))
((product ?x* and ?y*)   (* ?x ?y))
((multiply ?x* by ?y*)   (* ?x ?y))
((divide ?x* by ?y*)    (/ ?x ?y))
((add ?x* to ?y*)         (+ ?x ?y))
((subtract ?x* from ?y*) (- ?y ?x))
((?x* * ?y*)              (* ?x ?y))
((?x* times ?y*)          (* ?x ?y))
((?x* / ?y*)              (/ ?x ?y))
((?x* per ?y*)            (/ ?x ?y)))
```

```

(( ?x* divided by ?y*)      (/ ?x ?y))
(( half ?x*)                 (/ ?x 2))
((one half ?x*)              (/ ?x 2))
((1 half ?x*)                (/ ?x 2))
((twice ?x*)                 (* 2 ?x))
.....

```

Die Transformation mit dem regelbasierten Übersetzer Für die Transformation verwenden wir das Werkzeug *rule-based-translator*.

- Da die Ausdrücke geschachtelt sein können, müssen die erzeugten Bindungen rekursiv weiter zerlegt werden (*translate-pair*).
- Dadurch entsteht eine *indirekte Rekursion*.
- Das Ergebnis des Musterabgleichs ist
 - entweder eine Liste der Bindungen
 - oder eine Variable, deren Name aus dem Eingabetext erzeugt wird.

```

(define (translate-to-expression words)
; Translate an English phrase into an equation or expression.
(or (rule-based-translator
      words
      *student-rules*
      pat-match
      rule-pattern ; :rule-if
      rule-response ; :rule-then
      (lambda (bindings response) ; :action
        (let ((nbindings
              (map translate-pair bindings)))
          (sublis nbindings response) )))
      no-bindings
      )
     (make-variable words)))

```

Die indirekte Rekursion

```
(define (make-variable words)
  ;Create a variable name
  ;based on the given list of words
  (first words))

(define (translate-pair pair)
  ;Translate the value part of the pair
  ;into an equation or expression."
  (cons (binding-var pair)
    (translate-to-expression
      (binding-val pair))))
```

Beispiel

```
> (translate-to-expression
  (lexically-scan-words
   '(What do you get
     when you multiply six by seven ?)))
  → (= what (* 6 7))
```

Beispiel

```
> (translate-to-expression
  (lexically-scan-words
   '(Fran's age divided by Robin's height
     is one half Kelly's IQ |.||
     Kelly's IQ minus 80 is Robin's height |.||
     If Robin is 4 feet tall |,|
     how old is Fran ?)))
  → '(((= (/ Fran Robin) (/ Kelly 2))
    ((= (- Kelly 80) Robin)
     ((= Robin 4)
      (= how Fran )))))
```

Bei komplizierteren Gleichungen erhalten wir geschachtelte Ausdrücke.

Zerlegen in eine Liste von Gleichungen

```
(define (create-list-of-equations exp)
  ;Separate out equations embedded
  ;in nested parens."
  (cond ((null? exp) '())
```

```
((atom? (first exp)) (list exp))
(else
  (append
    (create-list-of-equations
      (first exp))
    (create-list-of-equations
      (rest exp))))))
```

Beispiel

```
> (create-list-of-equations
  (translate-to-expression
  (lexically-scan-words
   '(Fran's age divided by Robin's height
     is one half Kelly's IQ |.|.
     Kelly's IQ minus 80
     is Robin's height |.|.
     If Robin is 4 feet tall |,|
     how old is Fran ?))))
→ ((= (/ Fran Robin) (/ Kelly 2))
  (= (- Kelly 80) Robin)
  (= Robin 4)
  (= how Fran))
```

Das Lösen der Gleichungen

1. Suche eine Gleichung, die nur *eine einzige* Unbekannte nur einmal enthält, z.B. $x \times 7 = 3$
2. Isoliere die Unbekannte auf der linken Seite der Gleichung.
3. Berechne die rechte Seite der Gleichung.
4. Setze das Resultat in die anderen Gleichungen ein.
5. Entferne die Gleichung aus der Liste.
6. Wiederhole die Schritte, solange sich noch Variable isolieren lassen.

*;; Solve a system of equations by constraint propagation.
;; Try to solve for one equation, and substitute its value into
;; the others. If that doesn't work, return what is known.*

```
(define (solve equations known)
  (or (cl:some
        (lambda
          (equation)
          (let ([x (one-unknown equation)])
            (if x
                (let ([answer
                      (solve-arithmetic
                        (isolate equation x))])
                  (solve
                    (substit
                      (exp-rhs answer)
                      (exp-lhs answer))
                    (remove
                      equation equations equal?)))
                  (cons answer known))))
                #f)))
      equations)
  known)))
```

Ausrechnen des Wertes

```
(define (solve-arithmetic equation)
  ;Do the arithmetic for the right hand side.
  ;; This assumes that the right hand side
  ;is in the right form.
  (mkexp (exp-lhs equation) '='
         (eval (exp-rhs equation))))
> (solve-arithmetic
  (translate-to-expression
    (lexically-scan-words
      '(What do you get
        when you multiply
        six by seven ?))) → (= what 42)
```

Die Unbekannten

```
(define(unknown? exp)
  ; is exp an unknown to solve for?
  (symbol? exp))

(define (in-exp? x exp)
  ;True if x appears anywhere in exp
```

```

(or (eqv? x exp)
    (and (list? exp)
          (or (in-exp? x (exp-lhs exp))
              (in-exp? x (exp-rhs exp))))))

```



```

(define (no-unknown? exp)
; Returns true if there are no unknowns in exp.
(cond [(unknown? exp) #f]
      [(atom? exp) #t]
      [(no-unknown? (exp-lhs exp))
       (no-unknown? (exp-rhs exp))]
      [else #f])))

(define (one-unknown exp)
; Returns the single unknown in exp,
; if there is exactly one.
(cond [(unknown? exp) exp]
      [(atom? exp) #f]
      [(no-unknown? (exp-lhs exp))
       (one-unknown (exp-rhs exp))]
      [(no-unknown? (exp-rhs exp))
       (one-unknown (exp-lhs exp))]
      [else #f]))

```

Die Operatoren und Umkehrfunktionen

```

(define *operators-and-inverses*
'((+ -) (- +) (* /) (/ *) (= =)))

(define (inverse-op op)
  (cadr (assoc op
                *operators-and-inverses* )))

(define (commutative? op)
; Is operator commutative?
  (member op '(+ * =)))

```

Isolieren einer Variablen

- Sei e eine Gleichung, die die Variable x nur einmal enthält.
- Dann sind fünf Fälle zu unterscheiden:
 - x ist schon auf der linken Seite isoliert.– fertig.
 - Gleichungen der Form $A = f(x)$ werden umgestellt zu $f(x) = A$ und rekursiv gelöst.
 - Aus $f(x) * A = B$ mache $f(x) = B/A$
 - Aus $A * f(x) = B$ mache $f(x) = B/A$
 - Aus $A/f(x) = B$ mache $f(x) = A/B$

```
(define (isolate e x)
  "Isolate_the_lone_x_in_e_on_the_left_hand_side_of_e."
  ;; This assumes there is exactly one x in e,
  ;; and that e is an equation.
  (cond [(eq? (exp-lhs e) x)
         ;; Case I: X = A -> X = n
         e]
        [(in-exp? x (exp-rhs e))
         ;; Case II: A = f(X) -> f(X) = A
         (isolate (mkexp (exp-rhs e) '= (exp-lhs e)) x)]
        [(in-exp? x (exp-lhs (exp-lhs e)))
         ;; Case III: f(X)*A = B -> f(X) = B/A
         (isolate
          (mkexp (exp-lhs (exp-lhs e)) '='
                 (mkexp (exp-rhs e)
                        (inverse-op (exp-op (exp-lhs e)))
                        (exp-rhs (exp-lhs e)))) x))]
        [(commutative? (exp-op (exp-lhs e)))
         ;; Case IV: A*f(X) = B -> f(X) = B/A
         (isolate
          (mkexp (exp-rhs (exp-lhs e)) '='
                 (mkexp (exp-rhs e)
                        (inverse-op (exp-op (exp-lhs e)))
                        (exp-lhs (exp-lhs e)))) x))]
        [else ;; Case V: A/f(X) = B -> f(X) = A/B
         (isolate (mkexp (exp-rhs (exp-lhs e)) '='
                         (mkexp (exp-lhs (exp-lhs e)))))]
```

```
(exp-op (exp-lhs e))
(exp-rhs e))) x))))
```

Ein trace

$$\frac{\text{Fran}}{\text{Robin}} = \frac{\text{Kelly}}{2}$$

```
> (trace isolate) → (isolate)
> (isolate '(= (/ Fran Robin) (/ Kelly 2)))
   , Kelly )
|(isolate
   (= (/ Fran Robin) (/ Kelly 2)) Kelly)
|(isolate
   (= (/ Kelly 2) (/ Fran Robin)) Kelly)
|(isolate
   (= Kelly (* (/ Fran Robin) 2)) Kelly)
| (= Kelly (* (/ Fran Robin) 2))
| (= Kelly (* (/ Fran Robin) 2))

(isolate
  '(= (/ Fran Robin) (/ Kelly 2)) 'Fran)
|(isolate (= (/ Fran Robin) (/ Kelly 2)) Fran)
| (inverse-op /)
| *
|(isolate (= Fran (* (/ Kelly 2) Robin)) Fran)
| (= Fran (* (/ Kelly 2) Robin))
| (= Fran (* (/ Kelly 2) Robin))
```

Löse alle Gleichungen

```
(define (solve-equations equations)
  ; solve-equations → list of equations
  ; Print the equations and their solution"
  (print-equations
    "The_equations_to_be_solved_are:"
    equations)
  (let ([theSolution
         (solve equations '()))
    (if theSolution
      (print-equations
        "The_solution_is:" theSolution)
      (writeln "no_solution"))))
```

Drucken der Gleichungen in Infix-Notation

- Arithmetische Ausrücke sind baumrekursiv.
- Für die Konversion von Prefix-Notation zu Infix- Notation wird der Ausdrucksbaum *rekursiv* (in-order) traversiert.

```
(define (prefix->infix exp)
  ; prefix->infix: exp → list
  ; Translate prefix to infix expressions.
  (if (atom? exp) exp
      (map prefix->infix
           (cond [(binary-exp? exp)
                  (list
                   (exp-lhs exp)
                   (exp-op exp)
                   (exp-rhs exp))])
           [else
            exp]))))
```

Die Druckschleife

```
(define (print-equations header equations)
  ; Print a list of equations.
  (writeln header)
  (map (compose
         writeln
         prefix->infix)
       equations)
#t)
```

$$\begin{aligned} 3 + 4 &= (5 - (2 + \textcolor{red}{x})) * 7 \\ (3 * \textcolor{red}{x}) + \textcolor{blue}{y} &= 12 \end{aligned}$$

```
>(solve-equations
  '((= (+ 3 4) (* (- 5 (+ 2 \textcolor{blue}{x})) 7))
    (= (+ (* 3 \textcolor{blue}{x}) \textcolor{blue}{y}) 12)))
The equations to be solved are:
((3 + 4) = ((5 - (2 + \textcolor{blue}{x})) * 7))
(((3 * \textcolor{blue}{x}) + \textcolor{blue}{y}) = 12)
```

```

| (solve ((= (+ 3 4) (* (- 5 (+ 2 x)) 7))
|         (= (+ (* 3 x) y) 12)) ())
| (isolate (= (+ 3 4) (* (- 5 (+ 2 x)) 7)) x)
| (isolate (= (* (- 5 (+ 2 x)) 7) (+ 3 4)) x)
| (isolate (= (- 5 (+ 2 x)) (/ (+ 3 4) 7)) x)
| (isolate (= (+ 2 x) (- 5 (/ (+ 3 4) 7))) x)
| (isolate (= x (- (- 5 (/ (+ 3 4) 7)) 2)) x)
| (= x (- (- 5 (/ (+ 3 4) 7)) 2))
| (solve ((= (+ (* 3 2) y) 12)) ((= x 2)))

| (solve ((= (+ (* 3 2) y) 12)) ((= x 2)))
| ((isolate (= (+ (* 3 2) y) 12) y)
| ((isolate (= y (- 12 (* 3 2))) y)
| (= y (- 12 (* 3 2))))
| ((solve () ((= y 6) (= x 2)))
| (((= y 6) (= x 2))
| ((= y 6) (= x 2))
| ((= y 6) (= x 2))
The solution is:
(y = 6)
(x = 2) → #t

```

Das Student-Programm

```

(define (student words)
; Solve certain Algebra Word Problems.
(solve-equations
(create-list-of-equations
(translate-to-expression
(lexically-scan-words words)))) ;)

```

Beispiele:

```

> (student
'(What do you get
when you multiply six by seven ?))
The equations to be solved are:
(what = (6 * 7))
The solution is:
(what = 42)
#t

```

```

> (student '(The daily cost of living for a group
  is the overhead cost plus the running cost for each
  person times the number of people in the group |.|)
This cost for one group equals $ 100 |,|
and the number of people in the group is 40 |.|
If the overhead cost is 10 times the running cost |,|
find the overhead and running cost for each person |.|))
The equations to be solved are:
(daily = (overhead + (running * people)))
(cost = 100)
(people = 40)
(overhead = (10 * running))
(to-find-1 = ?x)
(to-find-2 = ?y)
The solution is:
(people = 40)
(cost = 100) → #t

(student
  '(Fran's age divided by Robin's height
    is one half Kelly's IQ |.|)
  Kelly's IQ minus 80 is Robin's height |.|
  If Robin is 4 feet tall |,|
  how old is Fran ?))

The equations to be solved are:
((Fran / Robin) = (Kelly / 2))
((Kelly - 80) = Robin)
(Robin = 4)
(how = Fran)
The solution is:
(how = 168)
(Fran = 168)
(Kelly = 84)
(Robin = 4) → #t

```

Hier kommt das vereinfachte Student-Programm im Zusammenhang. Die Funktionen, die in Peter Norvigs Common Lisp Variante Lisp-property-Listen erfordern, wurden in der Racket-Variante weggelassen, aber die wesentlichen Merkmale des Programms wurden übertragen.

```

#lang racket
; ;;; —* Mode: Lisp ; Syntax: Common-Lisp -*-*
; ;;; Code from Paradigms of AI Programming
; ;;; Copyright (c) 1991 Peter Norvig

; ;;; student.lisp: Chapter 7's STUDENT program to solve algebra word problems
; ;;; Split into packages by Leonie Dreschler-Fischer

(require
  racket/trace
  test-engine/racket-tests
  se3-bib/demo-gui-module
  se3-bib/tools-module
  se3-bib/macos-module
  se3-bib/string-module
  se3-bib/pattern-matching-module
  se3-bib/translator-module
  (only-in swindle/misc maptree)
  (prefix-in cl: (only-in swindle some)))
; die Common Lisp Variante von "some"
; gibt das Resultat der Funktionsanwendung
; für das erste Element der Liste zurück,
; das das übergebene Semiprädikat erfüllt.

(define (set-difference m1 m2)
  ; the difference set of set m1 and m2
  (filter (negate (curryr member m2))
          m1))

(define (set-union m1 m2)
  ; the union set of set m1 and m2
  (append m1 (set-difference m2 m1)))

(define (subset? m1 m2)
  ; is m1 a subset of m2?
  (null? (set-difference m1 m2)))

(define (find-all item sequence test?)
  ; "Find all those elements of sequence that match item,
  ; according to the test test?"
  (filter (curry test? item)

```

```

sequence))

(define (find-all-not item sequence testnot?)
;Find all those elements of sequence that do not match item,
;according to the test testnot?.
  (find-all item sequence (negate testnot?))
)

(define (subst newvalue var exp)
  (maptree
    (lambda (x)
      (if (eqv? x var)
          newvalue ; replace the variable
          x))
    exp))

; (check-expect
;   (subst '(+ 1 2) 'A '(+ A B))
;   '(+ (+ 1 2) B))
;
; (check-expect
;   (subst '(+ 1 2) 'A '(+ (+ A (+ A B) A) ))
;   '(+ (+ (+ 1 2) (+ (+ 1 2) B) (+ 1 2)))))

;(define-struct exp (lhs op rhs)); s.o.
(define (mkexp lhs op rhs)
  (list op lhs rhs)) ; prefix representation

(define (exp? x) (pair? x))
(define (exp-args x)
  (cdr x))

(define (exp-lhs x) (cadr x))
(define (exp-rhs x)
  (caddr x))
(define (exp-op x) (car x))

```

```

; (check-expect (mkexp 1 + 1) (list + 1 1))
; (check-expect
;   (let ((aa (mkexp 1 + 2)))
;
;     (equal? aa
;
;       (mkexp (exp-lhs aa)
;
;             (exp-op aa)
;
;             (exp-rhs aa))))
;
;   #t)
;
;

(define ; abbreviations for pattern variables
*abbrivs*
'(( ?x* . (?* ?x))
  (?y* . (?* ?y)))))

(define (expand-abbrivs rule)
; expand-abbrivs: rule -> rule
; replace the short cuts by expanded variables
; as defined by *abbrivs*
(let* ((old-pattern (rule-pattern rule))
       (response (rule-response rule))
       (new-pattern (sublis *abbrivs* old-pattern)))
  (make-rule new-pattern response)))

(define
*student-rules*
(map expand-abbrivs
'((( ?x* | . |)           ( ?x ))
  (( ?x* | . | ?y*)        (?x ?y))
  (( if ?x* | ,| then ?y*)  (?x ?y))
  (( if ?x* then ?y*)      (?x ?y))
  (( if ?x* | ,| ?y*)      (?x ?y))
  (( ?x* | ,| and ?y*)     (?x ?y))
  (( find ?x* and ?y*)    (= to-find-1 ?x) (= to-find-2 ?y))))
```

```

((find ?x*) (= to-find ?x))
((?x* equals ?y*) (= ?x ?y))
((?x* same as ?y*) (= ?x ?y))
((?x* = ?y*) (= ?x ?y))
((?x* is equal to ?y*) (= ?x ?y))
((?x* is ?y*) (= ?x ?y))
((?x* when you ?y*) (= ?x ?y))
((?x* - ?y*) (- ?x ?y))
((?x* minus ?y*) (- ?x ?y))
((difference between ?x* and ?y*) (- ?y ?x))
((difference ?x* and ?y*) (- ?y ?x))
((?x* + ?y*) (+ ?x ?y))
((?x* plus ?y*) (+ ?x ?y))
((sum ?x* and ?y*) (+ ?x ?y))
((product ?x* and ?y*) (* ?x ?y))
((multiply ?x* by ?y*) (* ?x ?y))
((divide ?x* by ?y*) (/ ?x ?y))
((add ?x* to ?y*) (+ ?x ?y))
((subtract ?x* from ?y*) (- ?y ?x))
((?x* * ?y*) (* ?x ?y))
((?x* times ?y*) (* ?x ?y))
((?x* / ?y*) (/ ?x ?y))
((?x* per ?y*) (/ ?x ?y))
((?x* divided by ?y*) (/ ?x ?y))
((half ?x*) (/ ?x 2))
((one half ?x*) (/ ?x 2))
((1 half ?x*) (/ ?x 2))
((twice ?x*) (* 2 ?x))
((square ?x*) (* ?x ?x))
;; caution! don't use!
((?x* % less than ?y*) (* ?y (/ (- 100 ?x) 100)))
((?x* % more than ?y*) (* ?y (/ (+ 100 ?x) 100)))
((?x* % ?y*) (* (/ ?x 100) ?y))
)))

```

; caution:
the % rules **do not** yet work with this simple translator
(**define** (noise-word? word)
; Is this a low-content word
; which can be safely ignored?
(**member** word '(a an the this number of \$)))

```

(define *numbers*
  '((zero . 0)
    (one . 1)
    (two . 2)
    (three . 3)
    (four . 4)
    (five . 5)
    (six . 6)
    (seven . 7)
    (eight . 8)
    (nine . 9)
    (ten . 10)))

(define *to-lower*
  '((If . if)
    (What . what)
    (Then . then)
    (The . the)
    (This . this)
    (Find . find)
    (Same . same)
    (Equals . equals)
    (Is . is)
    (Difference . difference)
    (Minus . minus)
    (Sum . sum)
    (Product . product)
    (Half . half)))

(define (translate-numbers sentence)
  (sublis *numbers* sentence))

(define (translate-upper sentence)
  (sublis *to-lower* sentence))

(define (lexically-scan-words words)
  (translate-numbers
    (filter (negate noise-word?)
      (translate-upper words)))))


```

```

(define *math-problems*
  '(
    (What do you get when you multiply six by seven ?)

    (Fran's age divided by Robin's height is one half Kelly's IQ |.|  

     Kelly's IQ minus 80 is Robin's height |.|  

     If Robin is 4 feet tall |,| how old is Fran ?)

    (The daily cost of living for a group is the overhead cost plus  

     the running cost for each person times the number  

     of people in the group |.|  

     This cost for one group equals $ 100 |,|  

     and the number of people in the group is 40 |.|  

     If the overhead cost is 10 times the running cost |,|  

     find the overhead and running cost for each person |.|)

    (Fran's age divided by Robin's height is one half Kelly's IQ |.|  

     Kelly's IQ minus 80 is Robin's height |.|  

     If Robin is 4 feet tall |,| how old is Fran ?)))

; (check-expect
;   (lexically-scan-words
;     (car *math-problems*))
;   '(What do you get when you
;     ;
;     multiply 6 by 7 ?))
;
;
```

```

(define (make-variable words)
  ;Create a variable name
  ;based on the given list of words
  (first words))

(define (translate-pair pair)
  ;Translate the value part of the pair
  ;into an equation or expression."
  (cons (binding-var pair)
```

```

(translate-to-expression
  (binding-val pair)))
;
; (define (replace-nested-vars expr)
;   ; (substit newvalue var exp)
;   (map (lambda (newval var)
;             (substit newval var expr))
;         (map binding-val *glob-bindings*)
;         (map binding-var *glob-bindings*)))
;

(define *glob-bindings* '())
; for returning the bindings in addition to the phrase

(define (translate-to-expression words)
  "Translate_an_English_phrase_into_an_equation_or_expression."
  (or (rule-based-translator
        words
        *student-rules*
        pat-match
        rule-pattern ; :rule-if
        rule-response ; :rule-then
        (lambda (bindings response) ; :action
          (let* ((nbindings (map translate-pair bindings))
                 ;(resp (replace-nested-vars nbindings response)))
                 (resp (sublis nbindings response)))
            (set! *glob-bindings* nbindings)
            resp ))
        no-bindings
        )
      (make-variable words)))
)

(define (create-list-of-equations exp)
  ;Separate out equations embedded in nested parens.
  (cond ((null? exp) '())
        ((atom? (first exp)) (list exp))
        (else
          (append
            (create-list-of-equations (first exp))
            (create-list-of-equations (rest exp)))))))
;
; (check-expect

```

```

; (translate-to-expression
;
;   '(if z is 3 | ,|
;
;     what is twice z))
; '((= z 3) (= what (* 2 z))))
;
; (check-expect
;   (translate-to-expression
;
;     (lexically-scan-words
;
;       '(What do you get when you multiply six by seven ?)))
;     '(= What (* 6 7))
;   )
;
; (check-expect
;   (translate-to-expression
;
;     (lexically-scan-words
;
;       '(Fran's age divided by Robin's height
;
;         is one half Kelly's IQ | .|
;
;         Kelly's IQ minus 80
;
;         is Robin's height | .|
;
;         If Robin is 4 feet tall | ,|
;
;           how old is Fran ?)))
;     '((= (/ Fran Robin) (/ Kelly 2))
;
;     ((= (- Kelly 80) Robin)
;
;      (= If (= 4 Fran)))))
;
;
;
; (check-expect

```

```

;  (create-list-of-equations
;
;  (translate-to-expression
;
;  (lexically-scan-words
;
;  '(Fran's age divided by Robin's height
;
;      is one half Kelly's IQ |.|)
;
;      Kelly's IQ minus 80
;
;      is Robin's height |.|)
;
;      If Robin is 4 feet tall |,|)
;
;      how old is Fran ?))))
; '((= (/ Fran Robin) (/ Kelly 2))
;
;      (= (- Kelly 80) Robin)
;
;      (= If (= 4 Fran)))))
;
;
```

(**define** (**binary-exp?** **x**)
 (**and** (**exp?** **x**)
 (**=** (**length** (**exp-args** **x**)) 2)))

(**define** (**prefix->infix** **exp**)
 ; *prefix->infix*: *exp* → *list*
 ; Translate prefix to infix expressions.
 (**if** (**atom?** **exp**) **exp**
 (**map** **prefix->infix**
 (**cond** [(**binary-exp?** **exp**)
 (**list**
 (**exp-lhs** **exp**)
 (**exp-op** **exp**)
 (**exp-rhs** **exp**))]
 [**else**
 exp]))))

```

(define (print-equations header equations)
  ; Print a list of equations.
  (writeln header)
  (map (compose
         writeln
         prefix->infix)
        equations)
#t)

;; solving the equations

(define *operators-and-inverses*
  '((+ -) (- +) (* /) (/ *) (= =)))

(define (inverse-op op)
  (cadr (assoc op *operators-and-inverses*)))

(define (commutative? op)
  ; Is operator commutative?
  (member op '(+ * =)))
;

; (check-expect (not (commutative? '+)) #f)
; (check-expect (commutative? '-) #f)
; (check-expect (inverse-op '*) '/')
; (check-expect (inverse-op
;   (inverse-op '*))) '*)
;

(define (unknown? exp)
  ; is exp an unknown to solve for?
  (symbol? exp))

(define (in-exp? x exp)
  ; True if x appears anywhere in exp
  (or (eqv? x exp)
      (and (list? exp)
           (or (in-exp? x (exp-lhs exp))
               (in-exp? x (exp-rhs exp))))))

(define (no-unknown? exp)

```

```

; Returns true if there are no unknowns in exp.
(cond ((unknown? exp) #f)
      ((atom? exp) #t)
      ((no-unknown? (exp-lhs exp)))
      ((no-unknown? (exp-rhs exp))))
      (else #f)))

(define (one-unknown exp)
; Returns the single unknown in exp,
; if there is exactly one.
(cond ((unknown? exp) exp)
      ((atom? exp) #f)
      ((no-unknown? (exp-lhs exp)) (one-unknown (exp-rhs exp)))
      ((no-unknown? (exp-rhs exp)) (one-unknown (exp-lhs exp)))
      (else #f)))

(define (solve-arithmetic equation)
; Do the arithmetic for the right hand side.
;; This assumes that the right hand side
; is in the right form.
(mkexp (exp-lhs equation) '=
       (eval (exp-rhs equation)))  

;  

; (check-expect (solve-arithmetic
;   (translate-to-expression
;     (lexically-scan-words
;  

;     ; '(What do you get
;     ;       when you multiply
;     ;         six by seven ?))))  

;     ' (= What 42))
;  

;  

;

(define (isolate e x)
"Isolate the lone x in e on the left hand side of e."
;; This assumes there is exactly one x in e,
;; and that e is an equation.
(cond [(eq? (exp-lhs e) x)

```

```

;; Case I: X = A -> X = n
e]
[(in-exp? x (exp-rhs e))
;; Case II: A = f(X) -> f(X) = A
(isolate (mkexp (exp-rhs e) '= (exp-lhs e)) x)]
[(in-exp? x (exp-lhs (exp-lhs e)))]
;; Case III: f(X)*A = B -> f(X) = B/A
(isolate
  (mkexp (exp-lhs (exp-lhs e)) =
    (mkexp (exp-rhs e)
      (inverse-op (exp-op (exp-lhs e)))
      (exp-rhs (exp-lhs e)))) x)]
[(commutative? (exp-op (exp-lhs e)))]
;; Case IV: A*f(X) = B -> f(X) = B/A
(isolate
  (mkexp (exp-rhs (exp-lhs e)) =
    (mkexp (exp-rhs e)
      (inverse-op (exp-op (exp-lhs e)))
      (exp-lhs (exp-lhs e)))) x)]
[else ;; Case V: A/f(X) = B -> f(X) = A/B
(isolate (mkexp (exp-rhs (exp-lhs e)) =
  (mkexp (exp-lhs (exp-lhs e))
    (exp-op (exp-lhs e))
    (exp-rhs e)))) x)])
;

; (check-expect (prefix->infix
;
;   (isolate '(= (/ Fran Robin)
;
;               (/ Kelly 2))
;
;               'Kelly))
;             '(Kelly = ((Fran / Robin) * 2)))
; (check-expect (isolate '(= If (= 4 Fran))
;
;                     'Fran ) '(= Fran (= If 4)))
;

(define (solve equations known)
;Solve a system of equations by constraint propagation.
;; Try to solve for one equation, and substitute its value into

```

```

;; the others. If that doesn't work, return what is known.
(or (cl:some
  (lambda
    (equation)
    (let ([x (one-unknown equation)])
      (if x
        (let ([answer
              (solve-arithmetic
                (isolate equation x))])
          (solve
            (substit
              (exp-rhs answer)
              (exp-lhs answer)
              (remove equation equations equal?)))
            (cons answer known)))
        #f))) equations)
known))

(define (solve-equations equations)
; solve-equations → list of equations
; Print the equations and their solution"
(print-equations
  "The_equations_to_be_solved_are:"
  equations)
(let ([theSolution
      (solve equations '())])
  (if theSolution
    (print-equations
      "The_solution_is:" theSolution)
    (writeln "no_solution"))))

(define (student words)
; Solve certain Algebra Word Problems.
(solve-equations
  (create-list-of-equations
    (translate-to-expression
      (lexically-scan-words words)))))

(define aProblem
  '(Peter's account is six $|.|

```

Susie 's account is twice of Peter 's account |.|
How much is Susie 's account ?))

; Zum Testen eingeben:

25 Means-Ends-Analyse: GPS

25.1 Der „General Problem Solver“: GPS

Fallstudien: Means-Ends-Analyse (GPS)



Nick Knatterton

22 Musterabgleich (pattern matching)

23 Means-Ends-Analyse: GPS

- Der „General Problem Solver“: GPS
- GPS Version 2

24 GPS Anwendungen

Navigation icons: back, forward, search, etc.

Bisher

Unsere bisherige Vorgehensweise beim Programmieren war, daß wir zunächst

- einen Algorithmus gesucht haben, um das Problem zu lösen, und dann
- den Algorithmus in ein Programm umgesetzt haben.

Der schwierigste Schritt dabei war das Finden des Algorithmus, und dafür haben wir den Rechner nicht eingesetzt.

Wir haben ihn als dummen Rechenknecht benutzt, dem genau gesagt werden muß, was er zu tun hat, ganz im Sinne des Mythos vom dummen Computer, der nicht selbständig denken kann.

Das Problemlöse-Paradigma, GPS 1

- Das Problemlöse-Paradigma geht davon aus, daß auch die *Suche nach dem Algorithmus* ein Problem ist, für das wir einen Algorithmus formulieren können.
- Programmieren besteht dann darin, möglichst genau unser Problem zu beschreiben, und der Rechner sucht systematisch nach Lösungsmöglichkeiten.
- Als diese Idee aufkam, war man sehr euphorisch und hat die Möglichkeiten weit überschätzt (und die Probleme unterschätzt).

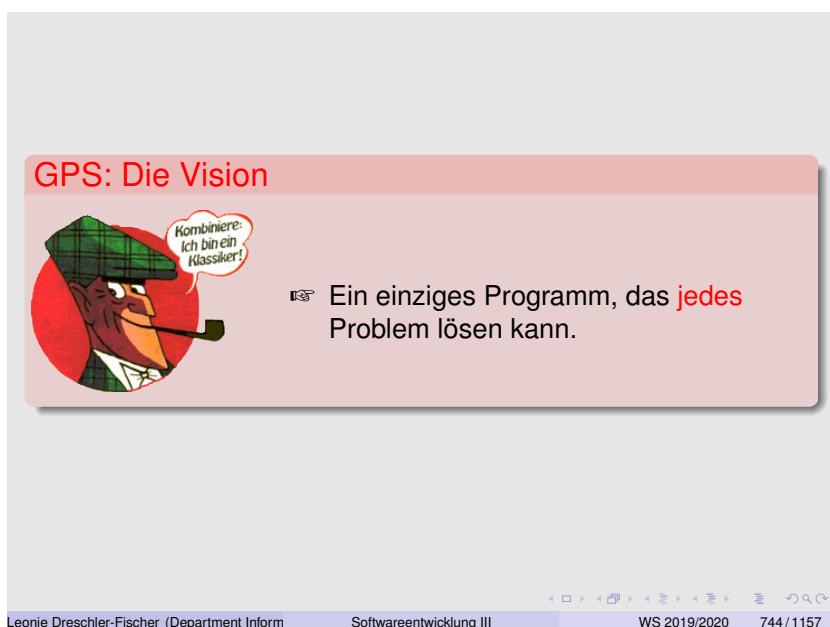
Alan Newell

It is not my aim to surprise or to shock you. . . .

But the simplest way I can summarize is to say that there are now in the world machines that think, that learn and create.

Moreover, their ability to do these things is going to increase rapidly until — in a visible future —the range of problems they can handle will be coextensive with the range to which the human mind has been applied.

(zitiert nach (Norvig, 1992))



Ein Meilenstein

GPS hat die hohen Erwartungen nie erfüllt: Damals konnte man die Schwierigkeiten noch nicht abschätzen, die sich auftun würden. Aber dennoch war GPS ein sehr wichtiger Meilenstein der Informatik:

Leistungen von GPS:

- ☞ GPS war das erste Programm, in dem das *Wissen* über einen Problembereich von der *Problemlösestrategie* getrennt wurde und das damit für eine Vielzahl von Problemen anwendbar war.
- ☞ GPS hat ein neues Forschungsgebiet — *Wissensrepräsentation* — erschlossen.

☞ **Anmerkung:** Warum beschäftigen wir uns mit GPS? Wir wollen am Beispiel von GPS nicht nur einen Klassiker der Informatik kennenlernen, sondern auch die Phasen des Programmierens — von der ersten vagen Idee bis zum fertigen Produkt aus der Sicht des funktionalen Programmierstils durchgehen und dabei Funktionen als Werkzeuge nutzen.

Phasen der Programmentwicklung

1. Problemanalyse, informelle Beschreibung
2. Problemspezifikation, algorithmische Beschreibung
3. Implementation in einer Programmiersprache
4. Test anhand von repräsentativen Daten
5. Fehlerkorrektur (debugging), Bewertung und Analyse des Programms
6. Wiederholung des Prozesses, bis alle Kriterien für das Programm erfüllt sind.

The main methods of GPS jointly embody the heuristic of means-ends analysis. Means-ends analysis is typified by the following kind of common-sense argument:

I want to take my son to nursery school. What's the difference between what I have and what I want? One of distance. What changes distance? My Automobile. My automobile won't work. What is needed to make it work? A new battery. What has new batteries? An auto repair shop. I want the repair shop to put in a new battery; but the shop does not know I need one. What is the difficulty? One of communication. What allows communication? A telephone... and so on.

The kind of analysis — classifying things in terms of the functions they serve and oscillating among ends, functions required, and means that perform them — forms the basic system of heuristic of GPS.

Newell and Simon, Human Problem Solving, 1972, zitiert nach ([Norvig, 1992](#))

Die Mittel-zum-Zweck-Analyse

Ein Beispiel

Problem: 122 (Wir wollen das Kind zum Kindergarten fahren.)

- Was ist der Unterschied zwischen dem, was wir erreichen wollen und was wir erreicht haben?

Die Entfernung.

- Was verändert die Entfernung?

Das Auto. Das Auto ist aber defekt.

- Was brauche ich, um es zu reparieren?

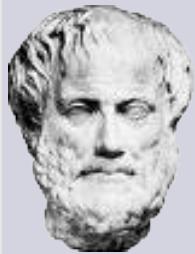
Eine neue Batterie.

- Wie komme ich zu einer neuen Batterie?

usw.

Die Mittel-zum-Zweck-Analyse ist nicht neu.

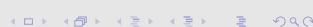
- ☞ Die Theorie der means-ends-Analyse wurde schon von Aristoteles vor rund 2300 Jahren entwickelt, wie Norvig-1992 anführt:



"Of course, this kind of analysis is not exactly new. The theory of means-ends analysis was laid down quite elegantly by Aristotle 2300 years earlier in the chapter entitled "The nature of deliberation and its objects" of the Nichomachean Ethics" (Book III. 3, 1112b).

Aristoteles

Aristoteles lebte 384–322 v. Chr.



We deliberate not about ends, but about means. For a doctor does not deliberate whether he shall heal, nor an orator whether he shall persuade, nor a statesman whether he shall produce law and order, nor does anyone else deliberate about his end. They assume the end and consider how and by what means it is attained; and if it seems to be produced by several means they consider by which it is most easily and best produced, while if it is achieved by one only they consider how it will be achieved by this and by what means this will be achieved, till they come to the first cause, which in the order of discovery is last . . . and what is last in the order of analysis seems to be first in the order of becoming. And if we come on an impossibility, we give up search, e.g., if we need money and this cannot be got; but if a thing appears possible we try to do it.

Und hier eine klassische deutsche Übersetzung derselben Textstelle:

„Unsere Überlegung betrifft nicht das Ziel, sondern die Mittel es zu erreichen. Der Arzt überlegt nicht, ob er heilen, der Redner nicht, ob er überzeugen, der Staatsmann nicht, ob er dem Gemeinwesen eine gute Verfassung geben, und überhaupt niemand, ob er sein Ziel verfolgen soll, sondern nachdem man sich sein Ziel gestellt hat, sieht man sich um, wie und durch welche Mittel es zu erreichen ist; wenn es durch verschiedene Mittel möglich scheint, sieht man zu, durch welches es am leichtesten und besten erreicht wird; und wenn es durch eines regelrecht verwirklicht wird, fragt man wieder, wie es durch dasselbe verwirklicht wird, und wodurch wiederum jenes, bis man zu der ersten Ursache gelangt, die als letzte gefunden wird. Auf diese so beschriebene Weise verfährt man bei der Überlegung suchend und analysierend, d. h. zergliedernd, wie wenn es sich um die Konstruktion einer geometrischen Figur handelte. Doch ist nicht jedes Suchen eine Überlegung, z.B. das Suchen des Geometers nicht: dagegen ist jede Überlegung ein Suchen, und das, was bei der Zergliederung als Letztes herauskommt, ist bei der Verwirklichung durch die Handlung das Erste. Stößt man auf eine Unmöglichkeit, so steht man von der Sache ab, z. B. wenn sich Geldmittel erforderlich zeigen, die man nicht aufbringen kann. Erscheint die Sache aber möglich, so nimmt man sie zur Hand.“ (Aristoteles, 1995, III.5.)

Phase 1: Beschreibung

☞ **Die Idee:** Suche ein allgemeines Verfahren, das den Unterschied beseitigt,

- zwischen dem, *was ich habe*,
- und dem, *was ich wünsche*.
- Das Resultat sollte eine Liste von Aktionen sein, die auszuführen sind.
- Manche Aktionen sind erst durchführbar, wenn *Vorbedingungen* erfüllt sind:
 - Ein Auto braucht eine funktionsfähige Batterie, um fahren zu können.
 - Andere Aktionen sind eventuell direkt durchführbar.

☞ Ein Problem kann als gelöst betrachtet werden,

- wenn es eine direkt anwendbare Operation gibt, die das Ziel erreicht,
- oder wenn andere Aktionen die Vorbedingungen sicherstellen, damit eine solche Aktion anwendbar wird.

☞ Wir benötigen Beschreibungen der *anwendbaren Aktionen*, der

Vorbedingungen: Was muß gegeben sein, damit sie anwendbar sind? und

Konsequenzen: Wie verändern sie den Zustand der Welt?

Phase 2: Spezifikation

Wir repräsentieren den Ausgangszustand und den Zielzustand als eine Liste von Bedingungen.

Beispiel:

Ausgangszustand: (arm, unbekannt, unglücklich)

Zielzustand: (reich, berühmt, glücklich)

GPS ist dann eine Funktion von drei Parametern:

(**GPS** '(unknown poor happy)
'(rich famous happy) list-of-ops)

Stützfunktion: find-all

```
(require se3-bib/tools-module)

(define (find-all item sequence test?)
  ; "Find all those elements of sequence
  ; that match item according to test?
  (filter (curry test? item)
          sequence))

(define (find-all-not item sequence testnot?)
  ; Find all those elements of sequence
  ; that do not match item, acc. to testnot?.
  (find-all item sequence (negate testnot?)))
```

Die Repräsentation des Problems

```
(define op; An operation
  (action
    preconds
    add-list
    del-list) )

(define (gps theState goals ops )
  ; General problem solver:
  ; achieve all goals using ops.
  ; theState: a list of conditions that hold
  ; ops: the operators available
  ...
)
```

☞ Anmerkung: Die Repräsentation des Problems

theState: Eine Liste von Symbolen, die den Zustand der Welt beschreiben. Jedes Symbol repräsentiert eine Bedingung, die zur Zeit erfüllt ist.

'(Habe-Geld Habe-Haus Bin-ungluecklich)

ops: Eine Liste von anwendbaren Operatoren, die den Zustand der Welt verändern können. Jeder Operator hat eine Menge von Vorbedingungen, die erfüllt sein müssen, damit man ihn anwenden kann, sowie zwei Listen, die die Wirkung beschreiben:

- Eine add-Liste, die angibt, welche neuen Bedingungen durch die Anwendung des Operators hinzukommen,
- sowie eine del-Liste, die angibt, welche Bedingungen danach nicht mehr gelten.

op: Der Datentyp zur Repräsentation von Operationen.

```
( struct op "An_operation"
  (action nil) ; Der Name des Operators
  (preconds nil) ; Die Vorbedingungen für die Anwendung
  (add-list nil) ; Bedingungen, die hinzukommen
  (del-list nil)) ; Bedingungen, die danach nicht mehr
                   ; erfüllt sein werden.
```

Ein Problem ist gelöst, wenn eine Folge von Operationen gefunden wird, die den Ausgangszustand so verändern, dass ein gewünschter Zielzustand erreicht wird.

GPS wendet einen Operator an, sofern die Bedingungen in seiner add-list zur Lösung beitragen. Wenn seine Vorbedingungen nicht erfüllt sind, sucht GPS rekursiv nach Operatoren, die diese Vorbedingungen erfüllen können.

Die Operationen

```
( struct op; An operation
  (action
  preconds
  add-list
  del-list) )
```

Durch die Definition erzeugte Akzessorfunktionen:

```
op, op-action, op-preconds,
op-add-list, op-del-list,
op?
```

Die Ablaufkontrolle

```
(defun gps (*state* goals *ops*)
; General problem solver:
; achieve all goals using *ops*.

(define (achieve goal)
  ....
)
(if (every achieve goals)
  'solved
  'not-solved))
```

- Das Problem ist gelöst, wenn die Funktion achieve jedes (**every**) der Ziele erreicht hat.
- Andernfalls ist das Resultat 'not-solved.

achieve

Beispiel: 123 (Erreiche ein Ziel:)

- Suche einen anwendbaren Operator und wende ihn an.
- Falls das Ziel schon erreicht ist, tue nichts.

```
(define (achieve goal)
  ; A goal is achieved if it already holds,
  ; or if there is an appropriate op for it
  ; that is applicable.
  (writeln (list "trying:" goal))
  (or (member goal theState)
    (cl:some apply-op
      (find-all goal ops appropriate?))))
```

appropriate?

Trägt der Operator zum Ziel bei?

```
(define (appropriate? goal op)
  ; An op is appropriate to a goal
  ; if it is in its add list.
  (member goal (op-add-list op)))
```

☞ Anmerkung: An dieser Stelle wird ein wichtiger Unterschied zwischen Rückzugsverfahren (backtracking) und Mittel-Zweck-Analyse (means-ends-analysis) deutlich: Beim backtracking haben wir nach Operatoren gesucht, die direkt im aktuellen Zustand anwendbar sind, hier suchen wir vom Ziel aus nach einem Operator, der uns dem Ziel näher bringt.

Anwendung des Operators

```
(define (apply-op op)
  ; Print a message and update *state*
  ; if op is applicable.
  (if (every achieve (op-preconds op))
    (begin
      (writeln (list 'executing (op-action op)))
      (set! *state*
        (set-difference
          *state* (op-del-list op)))
      (set! *state*
        (set-union
          *state* (op-add-list op))) #t)
    #f))
```

Achtung: Modifikatoren!

☞ Anmerkung: Wir werden später eine funktionale Lösung zeigen; aber zunächst sollen die Probleme gezeigt werden, die sich aus der Verwendung der Modifikatoren ergeben.

Indirekte Rekursion: *apply-op* und *achieve*:

achieve versucht ein Ziel zu erreichen, indem es solange Operatoren mit *apply-op* anzuwenden versucht, bis *mindestens ein* Aufruf erfolgreich war (*some*).

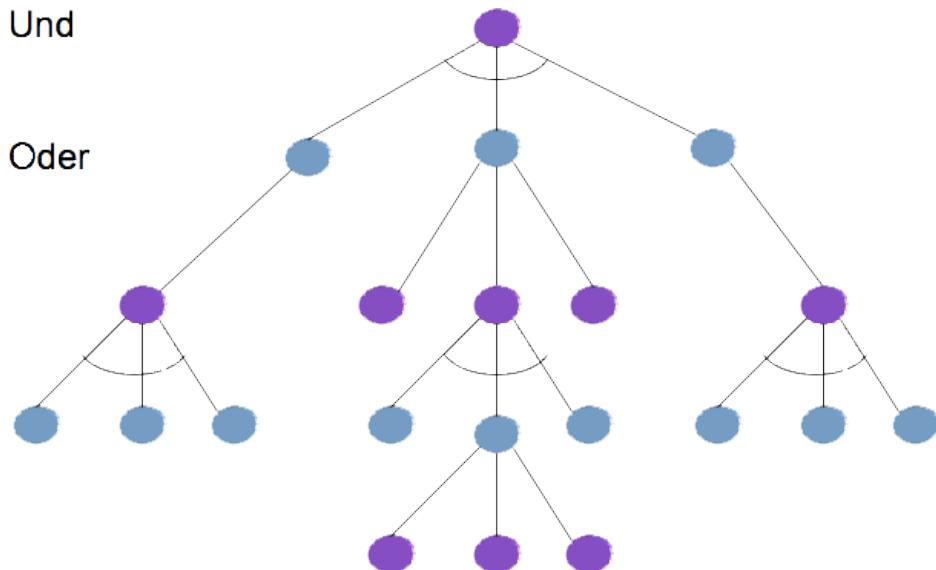
apply-op versucht den Operator anzuwenden, indem es mit *achieve alle Vorbedingungen (every)* herzustellen versucht.

- Wir haben eine *indirekte* Baumrekursion vorliegen, bei der abwechselnd alle oder nur ein Knoten erfolgreich abgearbeitet werden müssen.
- Solche Bäume heißen und/oder-Bäume (and/or trees).

Und/oder-Bäume

Und

Oder



```
#lang racket
; ;;; Das vollstaendige GPS-Programm
; ;;; Version 1 – sehr problematisch
; Übersetzt nach Racket von Leonie Dreschler-Fischer,
; nach einer CommonLisp Vorlage von Peter Norvig

(require se3-bib/tools-module
          test-engine/scheme-tests
          (prefix-in cl: (only-in swindle some)))
```

```

; (define (complement pred?)
;   ; the complement of the predicate pred?
;   ;defined in swindle as negate
;   (compose not pred?))

(define (set-difference m1 m2)
  ; the difference set of set m1 and m2
  (filter (negate (curryr member m2))
          m1))
(define (set-union m1 m2)
  ; the union set of set m1 and m2
  (append m1 (set-difference m2 m1)))

(define (find-all item sequence test?)
  ;"Find all those elements of sequence that match item,
  ; according to the test test?
  (filter (curry test? item)
          sequence))

(define (find-all-not item sequence testnot?)
  ;Find all those elements of sequence that do not match item ,
  ;according to the test testnot?.
  (find-all item sequence (negate testnot?)))
  )

(struct op; An operation
  (action
   preconds
   add-list
   del-list) )

(define(appropiate? goal op)
  ;An op is appropiate to a goal if it is in its add list.
  (member goal (op-add-list op)))

(define *school-ops*
  (list
    (op 'drive-son-to-school ; action
        '(son-at-home car-works) ;preconds

```

```

        '(son-at-school) ; add-list
        '(son-at-home)) ; del-list
(op 'shop-installs-battery ;action
    '(car-needs-battery
      shop-knows-problem shop-has-money)
    '(car-works)
    '(car-needs-battery))
(op 'tell-shop-problem
    '(in-communication-with-shop)
    '(shop-knows-problem)
    '())
(op 'telephone-shop
    '(know-phone-number)
    '(in-communication-with-shop)
    '())
(op 'look-up-number
    '(have-phone-book)
    '(know-phone-number)
    '())
(op 'give-shop-money
    '(have-money)
    '(shop-has-money)
    '(have-money))))
(define (gps theState goals ops )
;General problem solver: achieve all goals using *ops*.
; theState: a list of conditions that hold
; ops: the operators available
; (trace achieve apply-op)

(define (achieve goal)
; A goal is achieved if it already holds,
; or if there is an appropriate op for it
; that is applicable.
(writeln (list "trying:" goal))
(or (member goal theState)
  (cl:some apply-op
    (find-all goal ops appropiate?)))))

(define (apply-op op)
; Print a message and update theState if op is applicable.

```

```

(if (every achieve (op-preconds op))
  (begin
    (writeln (list 'executing (op-action op)))
    (set! theState (set-difference theState (op-del-list op)))
    (set! theState (set-union theState (op-add-list op)))
    #t)
  #f))

(if (every achieve goals)
  'solved 'not-solved))

(check-expect
  (gps '(son-at-home
         car-needs-battery
         have-money
         have-phone-book)
        '(son-at-school)
        *school-ops*) 'solved)

(check-expect
  (gps '(son-at-home
         car-needs-battery
         have-money )
        '(son-at-school)
        *school-ops*)
  'not-solved)
  (check-expect
    (gps '(son-at-home
           car-works)
      '(son-at-school)
      *school-ops*)
    'solved)

```

Beispiel: Kind zur Schule fahren

Wir definieren die Domänen-spezifischen Operatoren: Bedingungen werden durch Symbole mit aussagekräftigen Namen repräsentiert.

```
(op 'drive-son-to-school; action
  '(son-at-home car-works); preconds
  '(son-at-school); add-list
  '(son-at-home)); del-list
```

- Die Operation `drive-son-to-school` ist also anwendbar, wenn der Junge zuhause ist und das Auto betriebsbereit.
- Als Konsequenz ist der Junge anschließend in der Schule und nicht mehr zuhause.

```
(op 'drive-son-to-school ; action
  '(son-at-home car-works) ; preconds
  '(son-at-school) ; add-list
  '(son-at-home)) ; del-list
(op 'shop-installs-battery ; action
  '(car-needs-battery
    shop-knows-problem shop-has-money)
  '(car-works)
  '(car-needs-battery))
(op 'tell-shop-problem
  '(in-communication-with-shop)
  '(shop-knows-problem)
  '())
(op 'telephone-shop
  '(know-phone-number)
  '(in-communication-with-shop)
  '())
(op 'look-up-number
  '(have-phone-book)
  '(know-phone-number)
  '())
(op 'give-shop-money
  '(have-money)
  '(shop-has-money)
  '(have-money))))
```

☞ **Anmerkung: Drei Testläufe** In allen drei folgenden Beispielen besteht die Aufgabe darin, den Sohn zur Schule zu bringen. Der einzige Operator, der son-at-school in seiner add-list hat, ist drive-son-to-school. Deshalb wird dieser Operator von GPS als anwendbar ausgewählt. Allerdings müssen die Vorbedingungen hergestellt werden.

- Im ersten Beispiel ist die Autobatterie defekt. Durch eine Kette von Operatoren, die jeweils die Vorbedingungen für einen anderen Operator herstellen, kann das Auto repariert und anschließend das Kind zur Schule gebracht werden.
- Im zweiten Beispiel ist kein Telefonbuch verfügbar, so daß die Telefonnummer der Werkstatt nicht gefunden wird und GPS meldet ')(), um anzugeben, daß das Problem nicht lösbar war.
- Im dritten Beispiel ist das Auto von Anfang an einsatzbereit, so daß direkt die Operation drive-son-to-school ausgeführt werden kann.

Beispiel 1: Wir haben ein Telefonbuch

```
> (gps '(son-at-home car-needs-battery  
          have-money have-phone-book)  
        '(son-at-school)  
        *school-ops*)  
(executing look-up-number)  
(executing telephone-shop)  
(executing tell-shop-problem)  
(executing give-shop-money)  
(executing shop-installs-battery)  
(executing drive-son-to-school)  
solved
```

Beispiel 2: Wir haben kein Telefonbuch

```
> (gps '(son-at-home car-needs-battery  
        have-money )  
      '(son-at-school) *school-ops*)  
(trying: son-at-school)  
(trying: son-at-home)  
(trying: car-works)  
(trying: car-needs-battery)  
(trying: shop-knows-problem)  
(trying: in-communication-with-shop)  
(trying: know-phone-number)  
(trying: have-phone-book)  
→ not-solved
```

Beispiel 3: Das Auto ist heil

```
> (gps '(son-at-home car-works)  
      '(son-at-school) *school-ops*)  
(trying: son-at-school)  
(trying: son-at-home)  
(trying: car-works)  
(executing drive-son-to-school)  
solved
```

Analyse und Probleme:

Wie allgemein ist GPS?

- Wir werden im nächsten Abschnitt auf vier harte Beschränkungen des Verfahrens eingehen, die man teilweise auch als *bugs* sehen könnte.
 - Anschließend werden wir eine zweite Version von GPS präsentieren, die allgemeiner einsetzbar ist.
- ☞ Dieses ist ein Beispiel für *exploratives Programmieren und rapid prototyping*.

☞ **Anmerkung: Exploratives Programmieren** Bei der klassischen Art zu programmieren wird ein Problem zunächst vollständig spezifiziert und dann ein System zur Lösung implementiert. Das ist dann anwendbar, wenn wir ein Problem von Anfang an vollständig überblicken können. Gelegentlich aber müssen wir uns mit dem Problem erst einmal soweit vertraut machen, daß wir unsere Anforderungen spezifizieren können.

Auch beim Analysieren des Problems kann der Rechner helfen, wenn wir schnell einen arbeitsfähigen Prototypen des System schaffen können. An diesem können wir viele Aspekte unseres Problems erkunden, um dann unsere Anforderungen präziser fassen zu können. Das Schaffen eines Prototypen noch in der Entwurfsphase nennt sich *rapid prototyping* und der Einsatz des Rechners zur Problemanalyse nennt sich *exploratives Programmieren*. Wie Sie sehen, sind die Lisp-Sprachen für diese Herangehensweise besonders gut geeignet.

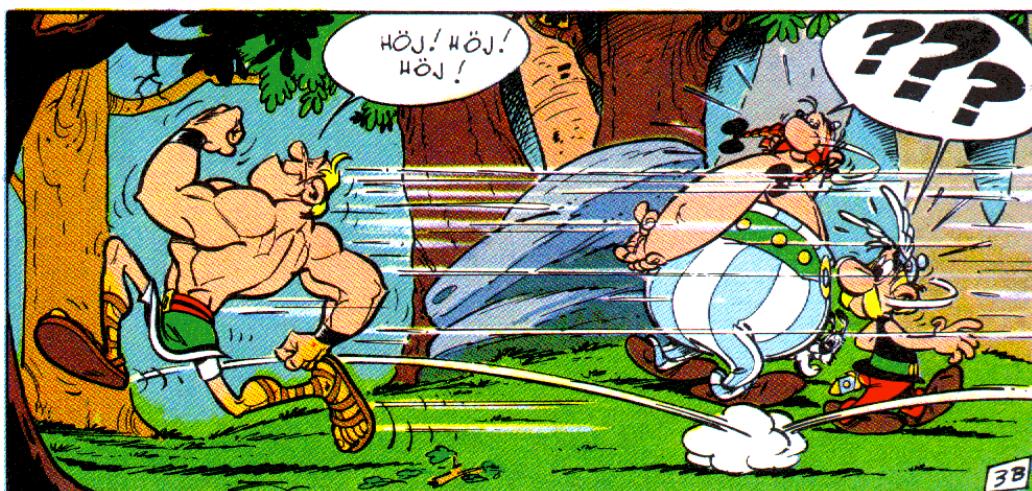
The Running Around the Block Problem

Schauen wir uns nochmals den Operator 'drive-son-to-school an: Seine Wirkung wird dadurch beschrieben, daß der Sohn vorher zuhause ist und anschließend in der Schule.

```
(make-op 'drive-son-to-school ; action
        '(son-at-home car-works) ; preconds
        '(son-at-school) ; add-list
        '(son-at-home)) ; del-list
```

Wenn wir auf dieselbe Weise einen Operator beschreiben wollen, um einmal um die Alster zu joggen, haben wir ein Problem. Wir sind vorher und nachher am selben Ort. D.h., die add-list und delete-list des Operators wären leer. GPS würde einen solchen Operator nie anwenden. Wir brauchen also noch andere Wirkungen, die wir in der addlist beschreiben können, wie got-some-exercise.

„Running around the Block“ Problem



Zustände

GPS kann keine Operatoren anwenden, die *nicht* den Zustand der Welt verändern.

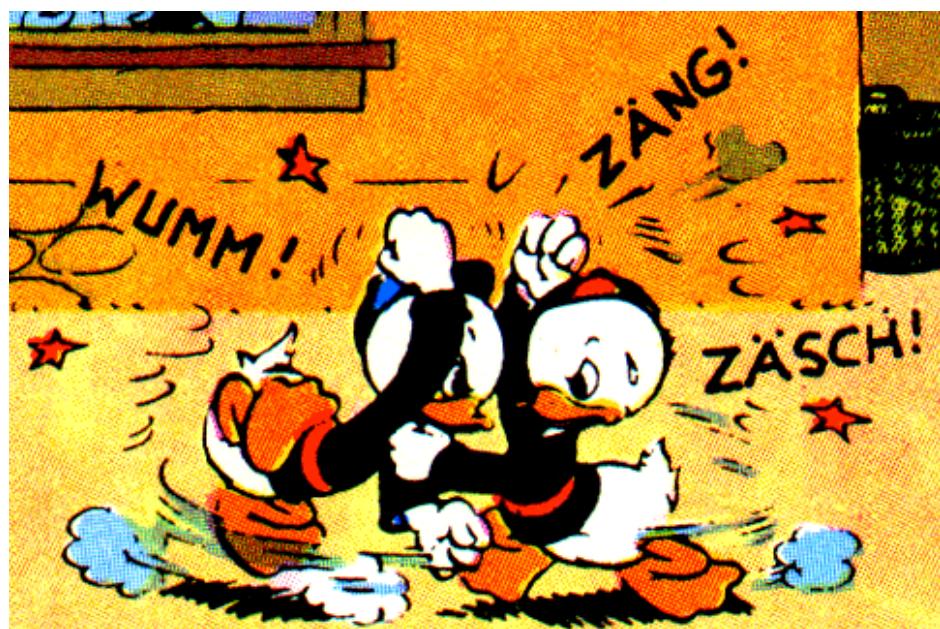
Beispiel: 124 (Um die Alster laufen:)

Wenn wir einmal um die Alster joggen, sind wir hinterher wieder am selben Ort — die „add-list“ und „del-list“ des Operators sind leer.

```
(op 'jogging-around-the-Alster
  '(have-running-shoes
    need-some-exercise) ; :preconds
  '() ; :add-list
  '()) ; :del-list
```

GPS wird den Operator nie anwenden, da die add-list leer ist.

Keilerei verwandter Ziele



The „Clobbered Sibling Goal Problem“

Was passiert, wenn wir mehrere Ziele haben?

Beispiel: 125 (Mehrere Ziele gleichzeitig)

Wir wollen das Kind zur Schule bringen und hinterher noch Geld übrig haben.

```
> (gps '(son-at-home have-money car-works)
      '(have-money son-at-school))
```

```

*school-ops*)
(executing drive-son-to-school)
solved

```

Dieses Beispiel geht gut, aber mit anderen Vorbedingungen ist die Lösung von GPS1 falsch.

```

> (gps '(son-at-home
         car-needs-battery
         have-money have-phone-book)
        '(have-money son-at-school)
        *school-ops*)
(executing look-up-number)
(executing telephone-shop)
(executing tell-shop-problem)
(executing give-shop-money)
(executing shop-installs-battery)
(executing drive-son-to-school)
→ solved ; ; Stimmt nicht!
> *state*
(have-phone-book know-phone-number
in-communication-with-shop shop-knows-problem
shop-has-money car-works son-at-school)

```

☞ Anmerkung: The „Clobbered Sibling Goal Problem“

GPS1 meldet, daß alle Ziele erreicht sind, obwohl have-money gar nicht mehr gilt; das Geld wurde ja alles für die Autoreparatur ausgegeben. Der Fehler ist, daß GPS1 die Ziele nacheinander zu erreichen versucht und nicht berücksichtigt, daß dabei die Wirkungen vorheriger Operatoren wieder zunichte gemacht werden können. achieve have-money meldet Erfolg, weil ja anfangs Geld da ist, und betrachtet das Teilproblem als gelöst, und anschließend gibt give-shop-money das ganze Geld aus. Gerald Sussman ([Sussman, 1973](#)) hat dieses Problem *prerequisite clobbers brother goal* genannt, und Peter Norvig nennt es, politisch korrekt, *the clobbered sibling goal problem*.

Modifikation von „achieve“

Wir können achieve so ändern, daß es prüft, ob am Ende wirklich *alle* Teilziele erreicht sind:

```

(define (achieve-all goals)
  ; Try to achieve each goal,
  ; then make sure they still hold.

```

```
(and (every achieve goals)
  (subset? goals *state* )))
```

Mit Überprüfung aller Teilziele

```
> (gps2 '(son-at-home  
         car-needs-battery  
         have-money have-phone-book)  
      '(have-money son-at-school)  
      *school-ops*)  
Goal: have-money  
Goal: son-at-school  
Consider: drive-son-to-school  
Consider: shop-installs-battery  
Consider: tell-shop-problem  
Consider: telephone-shop  
Consider: look-up-number  
Consider: give-shop-money  
→ ()
```

The Leaping Before You Look Problem

Wir drehen die Reihenfolge der Ziele um:

```
> (gps '(son-at-home car-needs-battery  
        have-money have-phone-book)  
      '(son-at-school have-money)  
      *school-ops*)  
(executing look-up-number)  
(executing telephone-shop)  
(executing tell-shop-problem)  
(executing give-shop-money)  
(executing shop-installs-battery)  
(executing drive-son-to-school)  
(trying: have-money)  
→ ()
```

Das Problem:

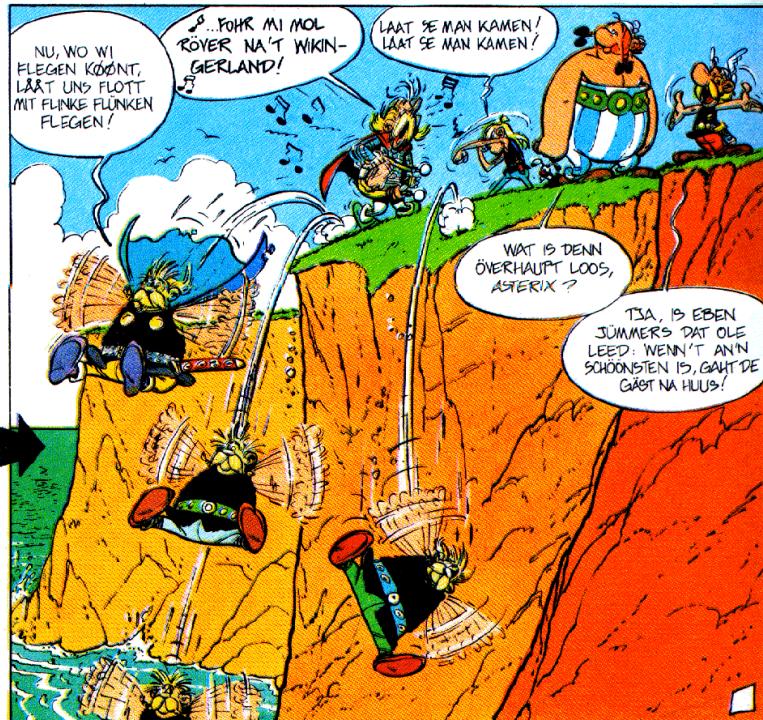
- GPS hat alle Operationen ausgeführt, um das erste Ziel zu erreichen,
- nur um dann festzustellen, daß das zweite Ziel nicht erreicht werden kann. Das kann fatal sein.

Beispiel: 126 (Seien die Ziele:)

'(jump-off-cliff land-safely)

GPS würde uns gegebenenfalls von den Klippen springen lassen und erst dann prüfen, ob eine sichere Landung möglich ist, eben „leaping before you look“.

The „Leaping Before You Look“ Problem



The „Recursive Subgoal“ Problem:

- Wir erweitern unsere Liste der Operatoren:
- Um eine Telefonnummer herauszubekommen,
 - kann man nicht nur im Telefonbuch nachschlagen,
 - man könnte auch jemanden anrufen.

```
> (set! *school-ops*
  (cons
    (op
      'ask-phone-number
      '(in-communication-with-shop)
      '(know-phone-number)  '())
    *school-ops*))
```

```
> (gps
  '(son-at-home
    car-needs-battery
    have-money )
  '(son-at-school)
  *school-ops*)
```



Segmentation fault

```
(gps  '(son-at-home car-needs-battery
           have-money )  '(son-at-school)
           *school-ops*)
trying: son-at-school
trying: son-at-home
trying: car-works
trying: car-needs-battery
trying: shop-knows-problem
trying: in-communication-with-shop
trying: know-phone-number
trying: in-communication-with-shop
trying: know-phone-number
trying: in-communication-with-shop
trying: know-phone-number
trying: in-communication-with-shop
trying: know-phone-number
....
```

Endlosschleife

Der trace zeigt uns klar, was das Problem ist:

- GPS ist in einer Endlosschleife hängengeblieben, in der es immer wieder versucht, als Voraussetzung für die Lösung des Problems das Problem selbst zu lösen.

Um die Werkstatt anrufen zu können, will das System sie anrufen, um die Telephonnummer herauszubekommen.

- Newell und Simon nennen dieses: “oszillating among ends, functions required, and means that perform them.”
- Eine Möglichkeit, solche Endlosschleifen zu vermeiden, besteht darin, zu überprüfen, ob ein Ziel schon erfolglos zu erreichen versucht wurde.

Eine typische means-ends-Analyse, leider erfolglos, weil zyklisch (in der Fassung des Medium-Terzetts).

1. Ein Loch ist im Eimer, Karl-Otto, Karl-Otto,
Ein Loch ist im Eimer, Karl-Otto, ein Loch.
2. Verstopf' es, o Henry, o Henry
Verstopf' es, o Henry, mach's dicht!
3. Womit denn, Karl-Otto, Karl-Otto, Karl-Otto, womit?
4. Mit Stroh, o Henry, o Henry, o Henry, mit Stroh!
5. Das Stroh ist zu lang, Karl-Otto, Karl-Otto, Karl-Otto, zu lang.
6. Dann kürz' es, o Henry, o Henry, o Henry, hack's ab!
7. Womit denn, Karl-Otto, Karl-Otto, Karl-Otto, womit?
8. Mit 'ner Axt, o Henry, o Henry, o Henry, mit 'ner Axt!
9. Die Axt ist stumpf, Karl-Otto, Karl-Otto, Karl-Otto, ist stumpf!
10. Dann schärf' sie, o Henry, o Henry, o Henry, mach sie scharf!
11. Womit denn, Karl-Otto, Karl-Otto, Karl-Otto, womit?
12. Mit 'nem Stein, o Henry, o Henry, o Henry, mit 'nem Stein.
13. Der Stein ist zu trocken, Karl-Otto, Karl-Otto, Karl-Otto, zu trocken.
14. Hol' Wasser, o Henry, o Henry, o Henry, hol' Wasser!
15. Worin, denn, Karl-Otto, Karl-Otto, Karl-Otto, worin?
16. Im Eimer, o Henry, o Henry, o Henry, im Eimer.
17. Ein Loch ist im Eimer, Karl-Otto, Karl-Otto, Karl-Otto, ein Loch!



Der Mangel an Information

Wenn GPS keine Lösung findet, gibt uns GPS keinen Hinweis darauf, woran das liegen könnte. Wir erfahren nur not-solved.

Wir werden deshalb ein debug-Werkzeug entwickeln, mit dem wir bei Bedarf mehr erfahren können.

- Das Programm wird mit Annotationen versehen, an den Stellen, wo wir bestimmte Informationen brauchen.

```
(dbg :gps "The_current_goal_is:_~a" goal)
```

- Mit einer Funktion my-debug können wir die debug-Ausgaben aktivieren und mit undebug wieder abschalten.

```
> (my-debug :gps)    -> (:gps)
> (undebug :gps)    -> NIL
```

☞ Anmerkung: Die drei debug-Funktionen my-debug, undebug, dbg drucken wir hier nicht ab, sondern überlassen Ihnen die Implementation zur Übung.

25.2 GPS Version 2

Um die o.g. Schwächen von GPS zu beheben, werden folgende Erweiterungen/Modifikationen vorgenommen:

GPS Version 2

- Die Operatoren ändern nicht den globalen Zustand, sondern führen eine eigene *Zustandsvariable* (*leaping before you look problem*).
- Es gibt eine Liste der aktuell verfolgten Ziele *goal-stack*. Ein neues Ziel wird nur dann verfolgt, wenn es nicht Element dieser Liste ist (*recursive subgoal problem*).
- Auch das *Ausführen* einer Operation ist eine Zustandsänderung. Es gibt neue Bedingungen der Form (executing xy), (move x to z). Das behebt das *running around the block problem* und vermindert den Schreibaufwand.
- Es werden Annotationen zum *debugging* eingeführt, um bei Bedarf Informationen über die verfolgten Ziele und geplanten Operationen zu erhalten (*lack of information problem*)

⌚ Anmerkung: Clobbered sibling goal problem

In der Liste der gelösten Probleme fehlt das *clobbered sibling goal problem*. Dieses Problem besteht leider auch in der verbesserten Fassung von GPS weiter. Es ist auch mit einfachen Mitteln nicht so leicht zu beheben, denn es erfordert, daß wir nicht einfach aufgeben, wenn wir feststellen, daß unsere Lösung für Problem x die Lösung für Problem y zunichte gemacht hat; wir müssen einen Plan entwickeln, wie beide Ziele gleichzeitig erreicht werden können und am besten die Abhängigkeiten zwischen den Zielen erkennen. *Planen* ist ein aktuelles Thema der KI-Forschung. Im Abschnitt über die „Klötzchenwelt“ ([Norvig, 1992](#), Kapitel 4.14) finden sie einige Hinweise auf die Komplexität dieses Problems, aber wir werden hier nicht weiter darauf eingehen.

Beispiel: 127

Da die GPS-Version 2 nicht mehr auf die Folien paßt, stelle ich hier das Programm zur Lektüre vor. Wir werden in der Vorlesung nur die Anwendung auf neue Domänen besprechen. Das wird dann auch die Frage beantworten: „Wie allgemein ist der General Problem Solver eigentlich?“

```

; ;;; Das vollstaendige GPS–Programm
; ;;; Version 2 – kein running around the block problem

#lang racket
; ;;; Das vollstaendige GPS–Programm
; ;;; Version 2 – kein running around the block problem

; Übersetzt nach Racket von Leonie Dreschler–Fischer,
; nach einer CommonLisp Vorlage von Peter Norvig

(require swindle/extra; fuer amb, amb–collect
         racket/trace
         se3–bib/tools–module
         test–engine/scheme–tests
         (prefix-in cl: (only-in swindle some)))
)

(define (set–difference m1 m2)
  ; the difference set of set m1 and m2
  (filter (negate (curryr member m2))
          m1))

(define (set–union m1 m2)
  ; the union set of set m1 and m2
  (append m1 (set–difference m2 m1)))

(define (subset? m1 m2)
  ; is m1 a subset of m2?
  (null? (set–difference m1 m2)))

(define (find–all item sequence test?)
  ;"Find all those elements of sequence that match item,
  ; according to the test test?
  (filter (curry test? item)
          sequence))

(define (find–all–not item sequence testnot?)
  ;Find all those elements of sequence that do not match item,
  ;according to the test testnot?.
  (find–all item sequence (negate testnot?)))

```

```

)
(define (ensure-list state)
  (if (list? state) state '()))

; testing
(define (state-check state)
  (unless (list? state)
    (error "not_a_list")))

(define (pretty depth what opOrGoal)
  (writeln
    (space depth)
    what "_"
    opOrGoal))

(define (protocolOp depth what op)
  (pretty depth what
    (op-action op)))

(define (protocolGoal depth what goal)
  (pretty depth what goal))

; the representation of operations
(struct op; An operation
  (action
   preconds
   add-list
   del-list) )

(define (print-op theOp)
  (writeln "Action:_" (op-action theOp))
  (writeln "Preconds:_" (op-preconds theOp))
  (writeln "Add_list:_" (op-add-list theOp))
  (writeln "Del-list:_" (op-del-list theOp)))

(define (make-ex-op
  action

```

```

    preconds
    add-list
    del-list)
;Make a new constructor that obeys the (EXECUTING op) convention.
; the add-list starts with (EXECUTING op) .
(op
  action
  preconds
  (cons (list 'executing action)
        add-list)
  del-list))

;; Operations

(define (starts-with? xs x)
;Is this a list whose first element is x?
(and (pair? xs) (eqv? (car xs) x)))

(define (executing? x)
; Is x of the form: (executing ...) ?
(starts-with? x 'executing))

(define (action? x)
; Is x something that is (start) or (executing ...) ?
(or (equal? x '(start)) (executing? x)))

(define (appropriate? goal op)
;An op is appropriate to a goal if it is in its add list.
(member goal (op-add-list op)))

(define (apply-op op state goal goal-stack ops)
;Return a new, transformed state if op is applicable.
(state-check state)

(protocolOp (length goal-stack) "Consider:" op)

(let ([state2 (ensure-list
              (achieve-all
               state
               (op-preconds op))

```

```

                (cons goal goal-stack) ops))])
(if (not (null? state2))
;; Return an updated state
(begin
  (protocolOp (length goal-stack)
              "Action: " op)

  (set-difference
    (set-union state2
               (op-add-list op))
    (op-del-list op)))
  '())))

(define (achieve state goal goal-stack ops)
;A goal is achieved if it already holds,
; or if there is an appropriate op for it that is applicable.
; (state-check state)
; (display "Goal state: ")(display state)
(protocolGoal (length goal-stack)
              "Goal: " goal)

(cond [(member goal (ensure-list state)) state]
      [(member goal goal-stack) '()]
      [else (cl:some
              (curryr apply-op
                     state goal goal-stack ops)
              (find-all goal ops appropriate?))])
)

(define (achieve-all
          state goals goal-stack ops)
(state-check state)
;Achieve each goal, and make sure
;they still hold at the end.
(let ((current-state state))
  (if (and (every
            (lambda (g)
              (set! current-state
                    (achieve
                      current-state
                      g

```

```

            goal-stack ops))
  (not (null? current-state)))
  goals)
  (list? current-state)
  (subset? goals current-state))
current-state '()
)))

(define (gps2 state goals ops)
;General problem solver: achieve all goals using *ops*.
; ;state*: a list of conditions that hold
; ;ops: the operators available
; ;(trace achieve apply-op)
(state-check state)
(filter action?
  (achieve-all
    (cons '(start) state)
    goals '() ops)))

;; Operations

(define *school-ops*
(list
  (make-ex-op 'drive-son-to-school ; action
    '(son-at-home car-works) ;preconds
    '(son-at-school); add-list
    '(son-at-home)); del-list
  (make-ex-op 'shop-installs-battery
    '(car-needs-battery
      shop-knows-problem shop-has-money)
    '(car-works)
    '(car-needs-battery))
  (make-ex-op 'tell-shop-problem
    '(in-communication-with-shop)
    '(shop-knows-problem)
    '())
  (make-ex-op 'telephone-shop
    '(know-phone-number)
    '(in-communication-with-shop)
    '())
  (make-ex-op 'look-up-number
    '(in-communication-with-shop)
    '())))

```

```

        '(have-phone-book)
        '(know-phone-number)
        '())
(make-ex-op 'give-shop-money
            '(have-money)
            '(shop-has-money)
            '(have-money)))))

(define *banana-ops*
  (list
    (make-ex-op 'climb-on-chair
                '(chair-at-middle-room
                  at-middle-room on-floor)
                '(at-bananas on-chair)
                '(at-middle-room on-floor)))
    (make-ex-op 'push-chair-from-door-to-middle-room
                '(chair-at-door at-door)
                '(chair-at-middle-room at-middle-room)
                '(chair-at-door at-door))
    (make-ex-op 'walk-from-door-to-middle-room
                '(at-door on-floor)
                '(at-middle-room)
                '(at-door))
    (make-ex-op 'grasp-bananas
                '(at-bananas empty-handed)
                '(has-bananas)
                '(empty-handed)))
    (make-ex-op 'drop-ball
                '(has-ball)
                '(empty-handed)
                '(has-ball)))
    (make-ex-op 'eat-bananas
                '(has-bananas)
                '(empty-handed not-hungry)
                '(has-bananas hungry)))))

;;;; A hole in the bucket, Ein Loch ist im Eimer
=====

(define *eimer-ops*
  (list

```

```

(make-ex-op 'Loch-mit-Stroh-verstopfen
            '(Habe-Eimer Eimer-hat-ein-Loch Habe-Stroh Stroh-ist-kur
              '(Eimer-ist-heil)
              '(Eimer-hat-ein-Loch Habe-Stroh))
(make-ex-op 'Stroh-abhacken
            '(Habe-Axt Habe-Stroh Axt-ist-scharf Stroh-ist-zu-lang)
              '(Stroh-ist-kurz)
              '(Stroh-ist-zu-lang))
(make-ex-op 'Schaerfe-Axt
            '(Axt-ist-stumpf Habe-Axt Habe-Stein Stein-ist-nass)
              '(Axt-ist-scharf)
              '(Axt-ist-stumpf))
(make-ex-op 'Benetze-Stein
            '(Habe-Stein Habe-Wasser Stein-ist-trocken)
              '(Stein-ist-nass)
              '(Stein-ist-trocken))
(make-ex-op 'Hole-Wasser
            '(Habe-Eimer Eimer-ist-heil)
              '(Habe-Wasser)
              '())
))

(check-expect
(gps2
  '(son-at-home
    car-needs-battery
    have-money
    have-phone-book)
  '(son-at-school)
  *school-ops*)
  '((start)
    (executing look-up-number)
    (executing telephone-shop)
    (executing tell-shop-problem)
    (executing give-shop-money)
    (executing shop-installs-battery)
    (executing drive-son-to-school)))
)

(check-expect
(gps2
  '(at-door

```

```

on-floor
has-ball
hungry
chair-at-door)
'(not-hungry)
*banana-ops*)
'((start)
  (executing
    push-chair-from-door-to-middle-room)
  (executing climb-on-chair)
  (executing drop-ball)
  (executing grasp-bananas)
  (executing eat-bananas)))

(check-expect
(gps2
  '(Habe-Eimer
    Eimer-hat-ein-Loch
    Habe-Stroh
    Stroh-ist-zu-lang
    Habe-Stein
    Stein-ist-trocken
    Habe-Axt
    Axt-ist-stumpf
    Habe-Wasser)
  '(Eimer-ist-heil)
  *eimer-ops*)
'((start)))
(check-expect
(gps2
  '(Habe-Eimer
    Eimer-hat-ein-Loch
    Habe-Stroh
    Stroh-ist-zu-lang
    Habe-Stein
    Stein-ist-trocken
    Habe-Axt
    Axt-ist-stumpf
  )
  '(Eimer-ist-heil)
  *eimer-ops*)

```

```

'())

;; labyrinth

(define (make-maze-op here there)
;Make an operator to move between two places
; make-maze-op : integer integer —> op
(make-ex-op
  '(move from ,here to ,there); action
  '((at ,here)); preconds
  '((at ,there)); add-list
  '((at ,here))); del-list
)
(define (make-maze-ops connection)
;Make maze ops in both directions
;make-maze-ops: pair of numbers -> op
(list (make-maze-op (first connection)
                     (second connection))
      (make-maze-op (second connection)
                     (first connection)))))

(define *maze-ops*
(append-map ;mappend
make-maze-ops
'((1 2)(2 3)(3 4)(4 9)(9 14)(9 8)
  (8 7)(7 12)(12 13)(12 11)(11 6)
  (11 16)(16 17)(17 22)(21 22)(22 23)
  (23 18)(23 24)(24 19)(19 20)(20 15)
  (15 10)(10 5)(20 25)))))

(define (getDest executing)
;Find the Y in
;(executing (move from X to Y))

(if (eqv? (car executing) 'executing)
  (last (second executing))
  executing))

(define (find-path start end)
;Search a maze for a path from start to end."
(let ((results
```

```

(gps2 '((at ,start)) '((at ,end)) *maze-ops*)))
(unless (null? results)
  (cons start (cdr
    ;remove start marker, add start number
    (map getDest results)))))

(check-expect
  (find-path 16 25)
  '(16 17 22 23 24 19 20 25))

;; the blocks world

(define (move-ons a b c)
; all add-ons for moving 'a from 'b to 'c
(if (eqv? b 'table)
  '(( ,a on ,c))
  '(( ,a on ,c)
    (space on ,b)))))

(define (move-op a b c)
; Make an operator to move A from B to C.
(make-ex-op
  '(move ,a from ,b to ,c); action
  '((space on ,a)
    (space on ,c)
    ( ,a on ,b)); :preconds
  (move-ons a b c); :add-list
  (move-ons a c b)); :del-list

(define (make-block-ops blocks)
  (flatten
    (amb-collect
      (let ((a (amb-car blocks))
            (b (amb-car blocks)))
        ; you can't move a block onto itself
        (amb-assert (not (eqv? a b)))
        (let* ((c (amb-car blocks))
              (from-to-table
                (list (move-op a 'table b)
                      (move-op a b 'table))))
          (from-block-to-block
            (amb-collect
              (let ((d (amb-car blocks))
                    (e (amb-car blocks)))
                ; you can't move a block onto itself
                (amb-assert (not (eqv? d e)))
                (let* ((f (amb-car blocks))
                      (to-table
                        (list (move-op d 'table e)
                              (move-op d e 'table)))))))
```

```

        (move-op a b c) ))
(if (or (eqv? c a) (eqv? c b))
    from-to-table
    (cons from-block-to-block from-to-table)))
)))
(check-expect
  (gps2 '((a on table)
          (b on table)
          (space on a)
          (space on b)
          (space on table))
         '((a on b)
           (b on table))
         (make-block-ops '(a b)))
  '((start)
    (executing
      (move a from table to b))))
  )

(define blocks '(red-block green-block
                      blue-block))

(define *block-ops* (make-block-ops '(a b c)))

```

☞ Anmerkung: Anpassung von Daten an neue Systemversionen

;; Conversion of old data
`(map convert-op *school-ops*)`

Die Datenbasis der Operatoren für das drive-on-to-school-Beispiel ist mit dem neuen GSP nicht mehr verwendbar, da die Operatoren keine executing--Formen enthalten. Sie wurde daher mittels einer sehr einfachen Funktion konvertiert, um den neuen Konventionen zu genügen. Dieses ist bei Lisp-Daten meistens einfach, wie man an diesem Beispiel gut sehen kann.

Wenn irgend möglich sollten wir immer die Anpassung von Datenbasen an neue Konventionen automatisieren, da so keine Gefahr besteht, daß bei der Übertragung Fehler eingebaut werden, wie es bei der Konversion von Hand leicht passieren könnte.

Testlauf mit alten Daten

> (gps2
 '(son-at-home car-needs-battery

```

have-money have-phone-book)
  '(son-at-school)
  *school-ops*)
→ (( start)
    (executing look-up-number)
    (executing telephone-shop)
    (executing tell-shop-problem)
    (executing give-shop-money)
    (executing shop-installs-battery)
    (executing drive-son-to-school))

```

Das Resultat ist jetzt eine *Liste der Aktionen* — sehr praktisch für die Weitergabe der Resultate an andere Funktionen.

Ein Testlauf mit debugger

```

> (gps2 '(son-at-home car-needs-battery
           have-money have-phone-book)
           '(son-at-school) *school-ops*)
Goal: son-at-school
Consider: drive-son-to-school
Goal: son-at-home
Goal: car-works
Consider: shop-installs-battery
Goal: car-needs-battery
Goal: shop-knows-problem
Consider: tell-shop-problem
Goal: in-communication-with-shop
Consider: telephone-shop
Goal: know-phone-number
Consider: look-up-number

Goal: have-phone-book
Action: look-up-number
Action: telephone-shop
Action: tell-shop-problem
Goal: shop-has-money
Consider: give-shop-money
Goal: have-money
Action: give-shop-money
Action: shop-installs-battery
Action: drive-son-to-school
(( start) (executing look-up-number))

```

```
(executing telephone-shop)
(executing tell-shop-problem)
(executing give-shop-money)
(executing shop-installs-battery)
(executing drive-son-to-school))
```

Test: Sind die bugs behoben?

```
> (gps2 '(son-at-home car-needs-battery
           have-money have-phone-book)
           '(have-money son-at-school)
           *school-ops*)
→ '() ; die Ausgabe ist jetzt korrekt
> (gps2 '(son-at-home car-needs-battery
           have-money have-phone-book)
           '(son-at-school have-money))
→ '() ; kein "leaping before you look" mehr
> (gps2 '(son-at-home) '(son-at-home))
→ ((START))
```

Übertragung auf neue Domänen

Um zu zeigen, daß GPS allgemein einsetzbar ist, wenden wir das Verfahren auf neue Domänen an:

A Hole in the Bucket: Ein Loch ist im Eimer ...

Monkey and Banana: Wie erreicht ein hungriges Äffchen seine Bananen?

Maze searching: Finde den Weg aus einem Labyrinth.

The Blocks World Domain: Löse Planungsaufgaben beim Arrangieren von Klötzchen.

Ein Loch ist im Eimer

```
(define *eimer-ops* (list
  (make-ex-op 'Loch-mit-Stroh-verstopfen
    '(Habe-Eimer Eimer-hat-ein-Loch
      Habe-Stroh Stroh-ist-kurz)
    '(Eimer-ist-heil)
    '(Eimer-hat-ein-Loch Habe-Stroh))
  (make-ex-op 'Stroh-abhacken
    '(Habe-Axt Habe-Stroh
```

```

        Axt-ist-scharf Stroh-ist-zu-lang)
'(Stroh-ist-kurz)
'(Stroh-ist-zu-lang))

(define *eimer-ops* (list
  (make-ex-op 'Schaerfe-Axt
    '(Axt-ist-stumpf Habe-Axt
      Habe-Stein Stein-ist-nass)
    '(Axt-ist-scharf)
    '(Axt-ist-stumpf))
  (make-ex-op 'Benetze-Stein
    '(Habe-Stein Habe-Wasser
      Stein-ist-trocken)
    '(Stein-ist-nass)
    '(Stein-ist-trocken))
  (make-ex-op 'Hole-Wasser
    '(Habe-Eimer Eimer-ist-heil)
    '(Habe-Wasser)
    '()))
  )

(gps2
  '(Habe-Eimer
    Eimer-hat-ein-Loch
    Habe-Stroh
    Stroh-ist-zu-lang
    Habe-Stein
    Stein-ist-trocken
    Habe-Axt
    Axt-ist-stumpf
    Habe-Wasser)
  '(Eimer-ist-heil)
  *eimer-ops*) → '()

Goal: Eimer-ist-heil
Consider: Loch-mit-Stroh-verstopfen
Goal: Habe-Eimer
Goal: Eimer-hat-ein-Loch
Goal: Habe-Stroh
Goal: Stroh-ist-kurz
Consider: Stroh-abhacken
Goal: Habe-Axt
Goal: Habe-Stroh
Goal: Axt-ist-scharf

```

Consider: **Schaerfe–Axt**
Goal: Axt–ist–stumpf
Goal: Habe–Axt
Goal: Habe–Stein
Goal: Stein–ist–nass
Consider: **Benetze–Stein**
Goal: Habe–Stein
Goal: Habe–Wasser
Consider: **Hole–Wasser**
Goal: Habe–Eimer
Goal: **Eimer–ist–heil** → ()

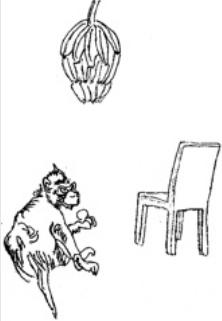
Die folgenden Beispiele „Affe und Banane“, „Labyrinthsuche“ und „Klötzchenwelt“ stammen ebenfalls aus dem Buch von [Norvig, 1992](#). An einigen wenigen Stellen habe ich den Code nicht nur direkt von CommonLisp nach Racket/Swindle übertragen, sondern erheblich modifiziert,

- entweder wenn Sprachelemente von Lisp benutzt wurden, für die es keine direkten Entsprechungen in Scheme gibt, wie beispielsweise Property-Listen,
- oder wenn Norvigs Lösungen eher imperativ war und es eine elegante Möglichkeit gab, den imperativen Algorithmus funktional zu formulieren, so daß es besser in den Rahmen dieser Vorlesung paßte.

26 GPS Anwendungen

26.1 Beispiel: Monkey and Banana

Fallstudien: Means-Ends-Analyse (GPS)



22 Musterabgleich (pattern matching)

23 Means-Ends-Analyse: GPS

24 GPS Anwendungen

- Beispiel: Monkey and Banana
- Beispiel: Pfade im Labyrinth
- Beispiel: Klötzchenwelt

Monkey and Banana: Ein klassisches KI-Problem

Beispiel: 128 (Affe und Banane)

Gegeben sei das folgende Szenario ([Amarel, 1968](#)):

Ein hungriger Affe steht bei der Tür eines Raumes.

Bananen hängen in der Mitte des Raumes von der Decke, so hoch, daß der Affe sie gerade nicht erreichen kann.

Ein Stuhl steht bei der Tür, der leicht genug ist, daß der Affe ihn verschieben kann.

Ein Ball: Der Affe hält einen Ball in der Hand, kann aber nicht mehr als einen Gegenstand zur Zeit halten.

```
(define *banana-ops* (list
  (make-ex-op
    'climb-on-chair
    '(chair-at-middle-room
      at-middle-room on-floor)
    '(at-bananas on-chair) ; +
    '(at-middle-room on-floor)) ; -
  (make-ex-op
    'push-chair-from-door-to-middle-room
    '(chair-at-door at-door)
    '(chair-at-middle-room at-middle-room)
    '(chair-at-door at-door))
  (make-ex-op
    'walk-from-door-to-middle-room
    '(at-door on-floor)
    '(at-middle-room)
    '(at-door))
  (make-ex-op 'grasp-bananas
    '(at-bananas empty-handed)
    '(has-bananas)
    '(empty-handed))
  (make-ex-op 'drop-ball
    '(has-ball)
    '(empty-handed)
    '(has-ball)))
```

```

(make-ex-op 'eat-bananas
  '(has-bananas)
  '(empty-handed not-hungry)
  '(has-bananas hungry )))

> (gps2
  '(at-door chair-at-door on-floor
    has-ball hungry chair-at-door)
  '(not-hungry)
  *banana-ops* )

→
((start)
 (executing
   push-chair-from-door-to-middle-room)
 (executing climb-on-chair)
 (executing drop-ball)
 (executing grasp-bananas)
 (executing eat-bananas))
>

(gps2 '(at-door chair-at-door
on-floor has-ball hungry chair-at-door)
'(not-hungry)*banana-ops* )

Goal: not-hungry
Consider: eat-bananas
Goal: has-bananas
Consider: grasp-bananas
Goal: at-bananas
Consider: climb-on-chair
Goal: chair-at-middle-room
Consider: push-chair-from-door-to-middle-room
Goal: chair-at-door
Goal: at-door
Action: push-chair-from-door-to-middle-room
Goal: at-middle-room
Goal: on-floor
Action: climb-on-chair
Goal: empty-handed
Consider: drop-ball
Goal: has-ball

```

```
Action: drop-ball
Action: grasp-bananas
Action: eat-bananas
(( start)
  (executing push-chair-from-door-to-middle-room)
  (executing climb-on-chair)
  (executing drop-ball)
  (executing grasp-bananas)
  (executing eat-bananas))
>
```

☞ **Anmerkung:** Wir haben hier wieder ein schönes Beispiel dafür, wie vorteilhaft die datengesteuerte Programmierung ist. Wir mußten nur die Daten ändern, die die Domäne beschreiben, aber nicht das Programm selbst.

Ebenso werden wir noch zwei weitere Domänen untersuchen. Bei diesen Domänen sind die Operatoren im Gegensatz zu den Operatoren der bisherigen Domänen systematisch erzeugbar. Wie werden uns daher Funktionen definieren, die uns die Operatoren-Datenbasis automatisch erzeugen.

26.2 Beispiel: Pfade im Labyrinth

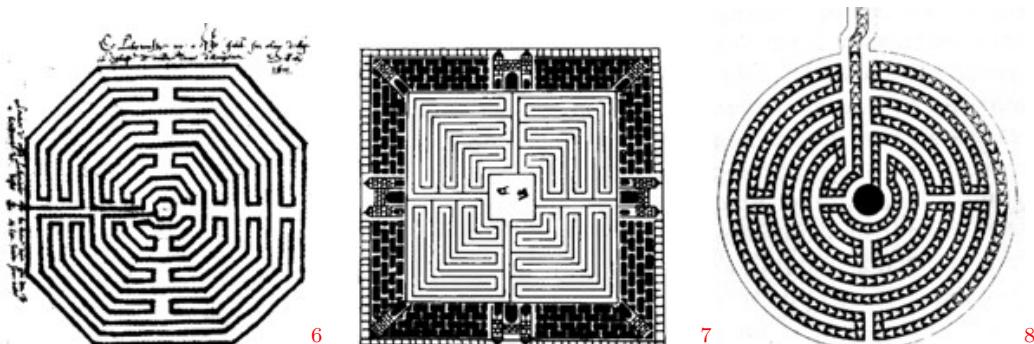
☞ Anmerkung: Labyrinthe Ursprünglich waren Labyrinthe zweidimensionale, mehr oder weniger symmetrische Muster, bei denen es, im Unterschied zu Meandern, einen eindeutigen Pfad vom Rand zum Ausgang gab. Da auch das Gefängnis des Minotauros von Ovid „Labyrinth“ genannt wurde, wird das Wort heute vor allem im Sinne von Irrgarten verwendet.

Die Abbildungen der Labyrinthe stammen aus (Becker, 1994)

Suche einen Weg aus dem Irrgarten



Labyrinth-Graffiti in Pompeii; der Schriftzug lautet: „LABYRINTHUS HIC HABITAT MINOTAURUS“ (Labyrinth, hier wohnt Minotaurus).



Beispiel: 129

- Repräsentation: Ein Raster von verbundenen, numerierten Orten.
- Die Verbindungen werden durch eine Nachbarschaftsrelation beschrieben.

⁶Achteckiges Labyrinth aus der Kathedrale von Amiens, Seitenlänge 5.2m

⁷Römisches Fußbodenmosaik aus Orbe (Schweiz), ca. 200 v.u.Z.

⁸Fußbodenmosaik aus dem 16. Jahrhundert (San Vitale, Ravenna)

Die Irrgarten-Operatoren

Wir betrachten nur eine Art von Operation:

- Gehe von einem Feld x zu einem erreichbaren, benachbarten Feld y:

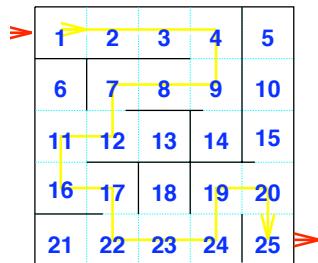
(move from x to y)

Die möglichen Zustände unserer Suche werden durch die Nummer des Feldes beschrieben, auf dem wir sind:

(at x)

Die add-list eines Operators (move from x to y) enthält daher (at y), die del-list eines Operators enthält (at x).

Die Nachbarschaftsrelation



```
(define *maze-ops*
  (mappend
   make-maze-ops
   '(((1 2)(2 3)(3 4)(4 9)(9 14)(9 8)
      (8 7)(7 12)(12 13)(12 11)(11 6)
      (11 16)(16 17)(17 22)(21 22)(22 23)
      (23 18)(23 24)(24 19)(19 20)(20 15)
      (15 10)(10 5)(20 25))))
```

Quasiquote oder backquote

- *quasiquote* ' ist eine Spezialform, die wie *quote* die Auswertung blockiert.
- Allerdings kann für Teile der quotierten Liste die Auswertung erzwungen werden, indem davor ein Komma gesetzt wird.
- *quasiquote* ist nützlich, wenn eine Form in weiten Teilen konstant ist und nur wenige Teile variabel sind.

```

> '(list ,(+ 1 2) 4)      → (list 3 4)
> (let ((name 'a))
  '(list ,name ',name))
  → (list a (quote a))

```

Listen werden im backquote ohne die umschließenden Klammern eingesetzt, wenn wir vor den Namen einen „Klammeraffen“ setzen. Beispiel:

```

> (define a '( 1 2 3)) → (1 2 3)
> `(' 1 ,@a 2 3) → (1 1 2 3 2 3)

```

Nachbarschaftsrelation: Operatoren

```

(define (make-maze-op here there)
;Make an operator to move between two places
; make-maze-op : integer integer → op
(make-ex-op
  '(move from ,here to ,there); action
  '((at ,here)); preconds
  '((at ,there)); add-list
  '((at ,here))); del-list
(define (make-maze-ops connection)
;Make maze ops in both directions
;make-maze-ops: pair of numbers → op
(list (make-maze-op (first connection)
                     (second connection))
      (make-maze-op (second connection)
                     (first connection)))

```



```

> (gps2 `((at 1)) `((at 25)) *maze-ops*)
→ ((start)
  (executing (move from 1 to 2))
  (executing (move from 2 to 3))
  (executing (move from 3 to 4))
  (executing (move from 4 to 9))
  (executing (move from 9 to 8))
  (executing (move from 8 to 7))
  (executing (move from 7 to 12))
  (executing (move from 12 to 11))
  (executing (move from 11 to 16))
  (executing (move from 16 to 17)))

```

```

(executing (move from 17 to 22))
(executing (move from 22 to 23))
(executing (move from 23 to 24))
(executing (move from 24 to 19))
(executing (move from 19 to 20))
(executing (move from 20 to 25)))

(gps2 '((at 1)) '((at 25))*maze-ops* )
  Goal: (at 25)
  Consider: (move from 20 to 25)
    Goal: (at 20)
    Consider: (move from 19 to 20)
      Goal: (at 19)
      Consider: (move from 24 to 19)
        Goal: (at 24)
        Consider: (move from 23 to 24)
          Goal: (at 23)
          Consider: (move from 22 to 23)
            Goal: (at 22)
            Consider: (move from 17 to 22)
              Goal: (at 17)
              Consider: (move from 16 to 17)
                Goal: (at 16)
                Consider: (move from 11 to 16)
                  Goal: (at 11)
                  Consider: (move from 12 to 11)
                    Goal: (at 12)
                    Consider: (move from 7 to 12)
                      Goal: (at 7)
                      Consider: (move from 8 to 7)
                        Goal: (at 8)
                        Consider: (move from 9 to 8)
                          Goal: (at 9)
                          Consider: (move from 4 to 9)
                            Goal: (at 4)
                            Consider: (move from 3 to 4)
                              Goal: (at 3)
                              Consider: (move from 2 to 3)
                                Goal: (at 2)
                                Consider: (move from 1 to 2)
                                  Goal: (at 1)

```

```

        Action: (move from 1 to 2)
        Action: (move from 2 to 3)
        Action: (move from 3 to 4)
        Action: (move from 4 to 9)
        Action: (move from 9 to 8)
        Action: (move from 8 to 7)
        Action: (move from 7 to 12)
        Action: (move from 12 to 11)
        Action: (move from 11 to 16)
        Action: (move from 16 to 17)
        Action: (move from 17 to 22)
        Action: (move from 22 to 23)
        Action: (move from 23 to 24)
        Action: (move from 24 to 19)
        Action: (move from 19 to 20)
        Action: (move from 20 to 25)

```

Ausgabe des Pfades von x nach y

```

(define (getDest exec)
;Find the Y in (exe... (move from X to Y))
  (if (eqv? (car exec) 'executing)
      (last (second exec))
      executing))

(define (find-path start end)
;Search a maze for a path: start->end."
  (let ((results
        (gps2 '((at ,start)) '((at ,end))
              *maze-ops*)))
    (unless (null? results)
      (cons start
            (cdr (map getDest results))))))

> (find-path 16 25)
→ (16 17 22 23 24 19 20 25)

```

 **Anmerkung:**

Ein Vorteil der neuen GPS-Version:

- Da die Aktionen, die zur Lösung führen, jetzt nicht ausgedruckt, sondern als Liste von Werten zurückgegeben werden, können andere Programme GPS als Funktion zum Lösen von Teilproblemen aufrufen.
- Das Resultat, das GPS liefert, kann leicht weiterverarbeitet werden.

26.3 Beispiel: Klötzchenwelt

Die Klötzchenwelt (blocks world)



- 22 Musterabgleich (pattern matching)
- 23 Means-Ends-Analyse: GPS
- 24 GPS Anwendungen
 - Beispiel: Monkey and Banana
 - Beispiel: Pfade im Labyrinth
 - Beispiel: Klötzchenwelt

Die Klötzchenwelt (blocks world)



Die Blocks World Domäne:

Ein Planungsproblem.

Beispiel (Klötzchenwelt)

Gegeben sei das folgende Szenario: Auf einem Tisch sind eine Reihe von Bauklötzen angeordnet.



- Sie sollen zu einer neuen Konfiguration, Turm, Brücke usw. arrangiert werden.
 - Ein Klotz kann nur bewegt werden, wenn kein anderer Klotz auf ihm drauf liegt.
 - Er kann nur auf einen freien Platz gelegt werden, wo nicht schon ein anderer Klotz liegt.

Blocks World-Repräsentation

- Die *Blöcke* werden durch Namen bezeichnet: A, B, Roter-Block usw.
 - Die Orte werden durch das Objekt angegeben, auf dem der Block ruht: A, B, table usw.
 - Wir nehmen an, daß ein Block auf seiner Oberfläche nur einen einzigen anderen Block liegen hat und jeder Block nur auf einem anderen Block (oder dem Tisch) liegt, aber die Türme können beliebig hoch werden.

Blocks World-Operatoren

In dieser Domäne gibt es nur die Operationen:

Beweise Block A von B nach C.

```
(define (move-op a b c)
;Make an operator to move A from B to c."
(make-ex-op
  '(move ,a from ,b to ,c) ; action
  '((space on ,a)
    (space on ,c)
    (,a on ,b)) ; :preconds
  (move-ons a b c) ; :add-list
  (move-ons a c b))) ; :del-list
```

```
(define (move-ons a b c)
  ; all add-ons for moving 'a from 'b to 'c
  (if (eqv? b 'table)
    '(( ,a on ,c))
    '(( ,a on ,c)(space on ,b))))
```

Die Menge aller Operatoren

Problem: 130

Gegeben sei eine Menge von Blöcken a, b, \dots auf dem Tisch oder übereinander gestapelt:

Welche Bewegungen sind möglich?

Lösung: 131 (Ein kombinatorischer Ansatz:)

Betrachte für jeden Block die Bewegungen relativ zu den anderen:

- Jeder Block kann von jedem anderen Block zu jedem anderen bewegt werden.
- Zusätzlich sind Bewegungen vom Tisch oder zum Tisch möglich.

Beobachtung

- Die Menge der Operatoren ist eine Funktion der Menge der Klötze.
- Schon bei nur wenigen Blöcken sind so sehr viele Operatoren nötig.
- Daher werden diese nicht von Hand definiert sondern automatisch konstruiert.

☞ Anmerkung: Norvigs Common Lisp Lösung für dieses Problem war elegant, aber imperativ:

- Die dolist-Anweisungen ergeben Iterationsschleifen über die Listen der Blöcke.
- Die Schleifen wirken nur durch Seiteneffekte: In die Liste der Klötze (globale Variable) wird mit push jeweils ein weiteres Klötzchen eingefügt.

; Language: Common Lisp

```
(defun make-block-ops (blocks)
  (let ((ops nil))
    (dolist (a blocks)
```

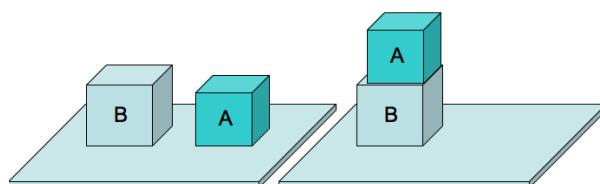
```
(dolist (b blocks)
  (unless (equal a b)
    (dolist (c blocks)
      (unless (or (equal c a) (equal c b))
        (push (move-op a b c) ops)))
      (push (move-op a 'table b) ops)
      (push (move-op a b 'table) ops))))
  ops))
```

Für unsere Racket-Variante wurde daher eine funktionale Variante gewählt: Die Kombinationen werden nicht-deterministisch mit amb erzeugt.

Die Liste der Operatoren

```
(define (make-block-ops blocks)
  (flatten
    (amb-collect
      (let ((a (amb-car blocks))
             (b (amb-car blocks)))
        ; you can't move a block onto itself
        (amb-assert (not (eqv? a b)))
        (let* ([c (amb-car blocks)])
          [from-to-table
            (list (move-op a 'table b)
                   (move-op a b 'table))]
          [from-block-to-block
            (move-op a b c)])
        (if (or (eqv? c a) (eqv? c b))
              from-to-table
              (cons from-block-to-block from-to-table)))
        )))))
```

Block vom Tisch nehmen und stapeln



```
> (gps2 '((a on table)
  (b on table)
  (space on a))
```

```

    (space on b)
    (space on table))
'((a on b)
  (b on table))
  (make-block-ops '(a b)))
→ ((start)
  (executing (move a from table to b)))

```

Beispiel 2: Block auf den Tisch legen

```

> (gps2 '((a on table)(b on a)(space on b)
           (space on table))
          '(((a on table) (b on table))
            (make-block-ops '(a b))))
Goal: (a on table)
Goal: (b on table)
Consider: (move b from a to table)
Goal: (space on b)
Goal: (space on table)
Goal: (b on a)
Action: (move b from a to table)
→ ((start)
  (executing (move b from a to table)))

```

Beispiel 3: Einen Turm bauen

```

>
(gps2
'((red-block on blue-block)
  (blue-block on green-block)
  (green-block on table)
  (space on red-block)
  (space on table))
'((green-block on blue-block)
  (blue-block on red-block))
  (make-block-ops
    '(red-block green-block
      blue-block)))
→ () ; keine Lösung

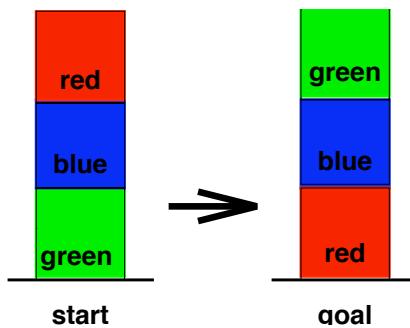
```

Keilerei der Ziele

Das überraschende Ergebnis:

- In der Klötzenwelt ist das *clobbered-sibling-problem* unvermeidbar, da mehrere Ziele gleichzeitig erreicht werden müssen.
- Es hängt von der Anordnung der Operatoren und der Reihenfolge der Ziele ab, ob eine Lösung gefunden werden kann.
- In Norvig-92 finden Sie einen Algorithmus, der für die Klötzenwelt die Ziele sortiert, so daß mehr Probleme gelöst werden können.
- Für manche Probleme gibt allerdings es überhaupt keine Anordnung der Ziele, die zu einer Lösung führt (Sussman-Anomalie).

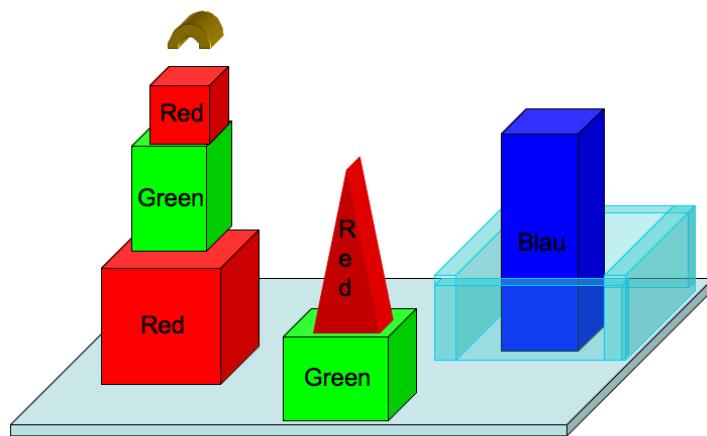
Mit Sortierung der Ziele



Language Common Lisp

```
((START)
 (EXECUTING
  (MOVE RED-BLOCK FROM BLUE-BLOCK TO TABLE))
 (EXECUTING
  (MOVE BLUE-BLOCK FROM GREEN-BLOCK TO RED-BLOCK))
 (EXECUTING
  (MOVE GREEN-BLOCK FROM TABLE TO BLUE-BLOCK)))
```

☞ Anmerkung: Die Blocksworld ist nicht nur für Planungs- und Problemlöseprobleme untersucht worden, sondern ist auch ein wichtiges Test-Szenario für Bild- und Sprachverstehen. Das folgende Bild wurde in Anlehnung an (Raphael, 1976) neu gezeichnet und zeigt einen Testlauf von Terry Winograds (Winograd, 1972) simuliertem Robotersystem, das in natürlicher Sprache gestellte Planungsprobleme lösen konnte.



“Find a block which is taller than the one you are holding and put it into the box.”

“Will you please stack up both of the red blocks and either a green cube or a pyramid?”

Anstelle einer Zusammenfassung: Norvig-92

“The following quote from Drew McDermott’s article “*Artificial Intelligence Meets Natural Stupidity*” sums up the current feeling about GPS. Keep it in mind the next time you have to name a program:

Remember GPS? By now, “GPS” is a colorless term denoting a particularly stupid program to solve puzzles. But it originally meant “General Problem Solver,” which caused everybody a lot of needless excitement and distraction. It should have been called *LFGNS* — “Local Feature-Guided Network Searcher.”

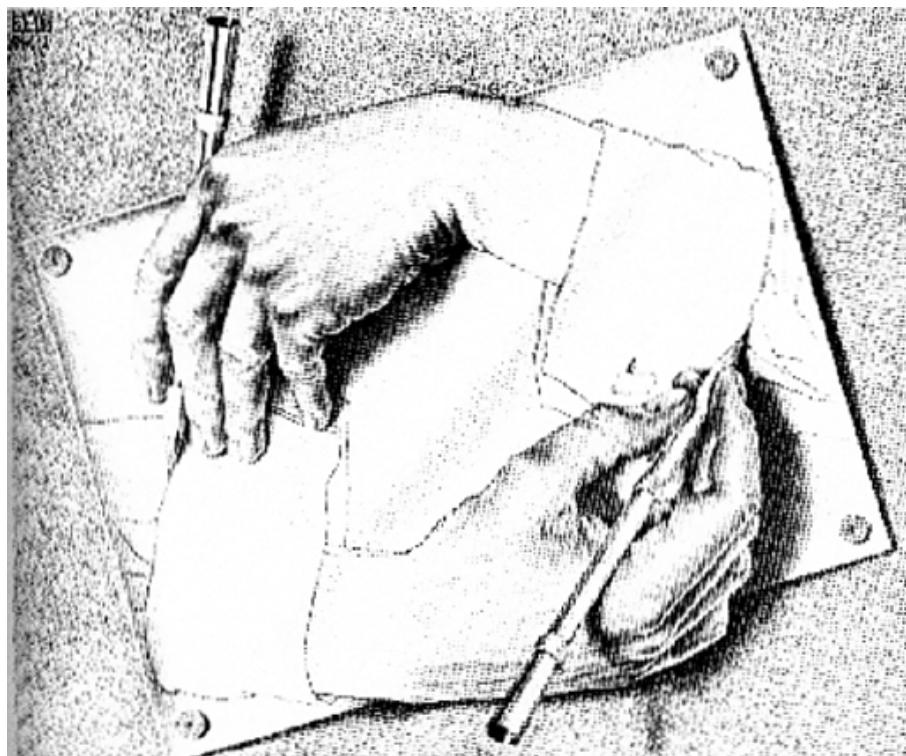
Norvig-92:

- „Nonetheless, GPS has been a useful vehicle for exploring programming in general, and AI programming in particular.
- More importantly, it has been a useful vehicle for exploring *the nature of deliberation*. Surely we’ll admit that Aristotle was a smarter person than you or me, yet with the aid of the computational model of mind as a guiding metaphor, and the further aid of a working computer program to help explore the metaphor, we have been led to a more thorough appreciation of means-ends-analysis — at least within the computer model.
- We must resist the temptation to believe that all thinking follows this model.“
- The appeal of AI can be seen as a split between means and ends. The end of a successful AI project can be a program that accomplishes some useful task better, faster, or cheaper than it could be done before.
- By that measure, GPS is mostly a failure, as it doesn’t solve many problems particularly well.
- But the means toward that end involved an investigation and formalization of the problem solving process. By that measure, our reconstruction of GPS is a success to the degree in which it leads the reader to a better understanding of the issues.“

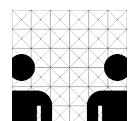
(Norvig, 1992, Seite 147)

Softwareentwicklung III

Leonie Dreschler-Fischer
Prüfungsunterlagen Teil 1:
Objekte und generische Funktionen



Universität Hamburg
Fachbereich Informatik
WS 2019/2020



Teil X

Objekte und generische Funktionen

27 CLOS: Objekte und generische Funktionen

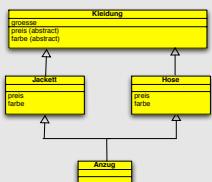
27.1 CLOS: Klassen, generische Funktionen

CLOS: Common Lisp Object System

Objektorientierte Programmierung in Swindle

25 CLOS: Objekte und generische Funktionen

- CLOS: Klassen, generische Funktionen
- Mehrfachvererbung und Methodenkombination
- Ergänzungsmethoden



26 Entwurf eines ereignisorientierten Simulationssystems

27 Objektorientierte Verarbeitungsmodelle





OO-funktionale Programmierung

 Keene, S. (1989).
Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.
Addison-Wesley.

 Graham, P. (1996).
Ansi Common Lisp.
Prentice-Hall, Englewood Cliffs, New Jersey, London.

Leonie Dreschler-Fischer (Department Inform) Softwareentwicklung III WS 2019/2020 825 / 1157

Von Racket zu Common Lisp (Swindle)

- In DrRacket ist als Spracherweiterung das Common Lisp Object System (CLOS) verfügbar.
- Um es benutzen zu können, benötigen Sie:
 - Die Sprache: Full Swindle
 - Die Bibliotheken: **setf.ss**, **misc.ss**.

```
#lang swindle
(require swindle/setf
         swindle/misc)
```

Objektorientierte Sprachelemente in CLOS

- **defclass**: Definition von Klassen und Attributen (*slots*), Objektinitialisierung, Akzessorfunktionen
- **defgeneric**: Definition von generischen Funktionen als Interface für Operationen auf Objekten.
- **defmethod**: Definition von Methoden als Implementation der generischen Funktionen.
- Imperative Sprachelemente: Modifikatoren, Schleifen
- Syntax: Schlüsselwort-Parameter

☞ Anmerkung: Eine Klasse von Schlangen

Beispiel: 132

Als erstes Beispiel werden wir in *CLOS* eine Klasse von Schlangen definieren, die später zu Unterklassen, wie *LIFO*-, *FIFO*- oder geordnete Schlangen (*RANKED-Queue*) spezialisiert werden soll. (Reiser and Wirth, 1994).

Die Operationen auf einer allgemeinen Schlange sollen sein:

`enqueue!`: Einreihen in die Schlange.

`dequeue!`: Entfernen aus der Schlange.

`head-queue`: Erstes Element der Schlange.

`empty-queue?`: Ist die Schlange leer?

`reset-queue!`: Lösche alle Einträge.

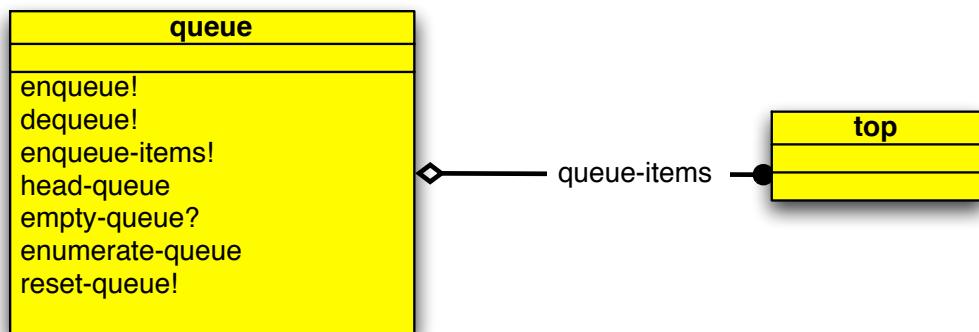
`enumerate-queue`: Erzeuge eine Liste der Einträge.

Diese Operationen spezifizieren die Schnittstelle für alle Anwendungen, die Schlangen verwenden oder die Klasse erweitern wollen. `enqueue!` und `dequeue!` sind Operationen, die von der speziellen Unterklasse der Schlange abhängen und sind daher abstrakt in der Oberklasse. Sie werden nur für die Unterklassen durch Methoden implementiert. Die Methoden für die anderen Operationen dagegen können für die Oberklasse `queues` implementiert von den spezielleren Klassen geerbt werden.

Die Schlange ist ein Aggregat aus Elementen einer beliebigen Klasse. Die Multiplizität der Aggregat-Assoziation ist „0 oder beliebig viele“. Die allgemeinste Klasse in Swindle-CLOS ist die Klasse „<top>“. Deshalb wurde diese Klasse für die Einträge gewählt.

Eine Klasse von Schlängen

Beispiel: Schlangen



Eine abstrakte Klasse „queues“

Klassen werden in CLOS mit *defclass* definiert:

```
#lang swindle .

(defclass* queue ()
  (items
   :reader queue-items
   :writer set-queue-items!
   :initvalue '()
   :type <list>
   :documentation "The_items_in_the_queue"
   )
  :autopred #t ; auto generate pred. queue?
  :printer #t
  :documentation
  "the_top_class_of_all_queues"
  )
```

Klassenattribute

Objektattribute, die für alle Instanzen der Klasse denselben Wert haben sollen, können als *Attribute der Klasse* betrachtet werden.

- Klassenattribute sollten nur einmal für die ganze Klasse repräsentiert werden, und nicht für jede Instanz einzeln.
- Mit *:allocation :class* werden Attribute statisch als Klassenattribute zentral für die Klasse alloziert.
- Mit *:allocation :instance* (das ist der default) werden Attribute dynamisch für jedes Objekt alloziert.
- Klassenattribute ersetzen globale Variable, die so den passenden Klassen zugeordnet werden können.

Die Syntax von defclass

```
(defclass <Name der Klasse> ({<Oberklassen>})
  {(<Slot> {<Slot-keys>})}
  {<Class-keys>}
)
```

☞ Anmerkung: Der Stern bei defgeneric* und defclass* bewirkt ein „auto provide“.

- So definierte Klassen und Funktionen werden automatisch von den definierenden Modulen exportiert.
- Die Akzessorfunktionen der „slots“ werden beim „auto provide“ allerdings nicht automatisch mit exportiert.

Ein Auszug aus der Dokumentation für swindle/closs.ss:

```
(defclass
  name (super ...)
  slot ...
  class-initarg ...) [syntax]
```

Available slot options are:

- *:initarg keyword*:

Use 'keyword' in 'make' to provide a value for this slot.

- *:initializer func*

Use the given function to initialize the slot – either a thunk or a function that will be applied on the initargs given to ‘make’.

- `:initvalue value`

Use ‘value’ as the default for this slot.

- `:reader name`

Define ‘name’ (an unquoted symbol) as a reader method for this slot.

- `:writer name`

Define ‘name’ (an unquoted symbol) as a writer method for this slot.

- `:accessor name`

Define ‘name’ (an unquoted symbol) as an accessor method for this slot – this means that two methods are defined: ‘name’ and ‘set-name!’.

- `:type type`

Restrict this slot value to objects of the given ‘type’.

- `:lock { #t | #f | value }`

If specified and non-‘#f’, then this slot is locked. ‘#t’ locks it permanently, but a different value works as a key: they allow setting the slot by using cons of the key and the value to set.

- `:allocation { :class | :instance }`

Specify that this slot is a normal one (‘:instance’, the default), or allocated per class (‘:class’). The specific way of creating helper methods (for readers, writers, and accessors) is determined by ‘-defclass-accessor-mode-’ (see above).

Available class options (in addition to normal ones that initialize the class slots like ‘:name’, ‘:direct-slots’, ‘:direct-supers’) are:

- `:metaclass class`

create a class object which is an instance of the ‘class’ meta-class (this means that an instance of the given meta-class should be used for creating the new class).

- `:autoinitargs { #t | #f }`

if set to '#t', make the class definition automatically generate initarg keywords from the slot names.

- `:autoaccessors { #f | #t | :class-slot | :slot }`

if set to non-'#f', generate accessor methods automatically – either using the classname Blotname convention (':class-slot') or just the slotname (':slot'). If it is '#t' (or turned on by ':auto') then the default naming style is taken from the '-defclass-autoaccessors-naming-' syntax parameter. Note that for this, and other external object definitions (':automaker' and ':autopred'), the class name is stripped of a surrounding <>B if any.

- `:automaker { #f | #t }`

automatically creates a 'maker' function using the make-classname naming convention. The maker function is applied on arguments and keyword-values – if there are n slots, then arguments after the first n are passed to 'make' to create the instance, then the first n are 'slot-set!'ed into the n slots. This means that it can get any number of arguments, and usually there is no point in additional keyword values (since if they initialize slots, their values will get overridden anyway). It also means that the order of the arguments depend on the *complete* list of the class's slots (as given by 'class-slots'), so use caution when doing multiple inheritance (actually, in that case it is probably better to avoid these makers anyway).

- `:autopred { #f | #t }`

automatically create a predicate function using the 'classname "?' naming convention.

- `:default-slot-options { #f | '(keyword ...)`

if specified as a quoted list, then slot descriptions are modified so the first arguments are taken as values to the specified keywords. For example, if it is "(:type :initvalue)" then a slot description can have a single argument for ':type' after the slot name, a second argument for ':initvalue', and the rest can be more standard keyword-values. This is best set with '-defclass-auto-initargs-'

- `:auto { #f | #t }`

if specified as '#t', then all automatic behavior available above is turned on.

- `:printer { #f | #t | procedure }`

if given, install a printer function. '#t' means install the 'print-object-with-slots' function from clos.ss", otherwise, it is expected to be a function that gets an object, an escape boolean flag an an optional port (i.e, 2 or more arguments), and prints the object on the class using the escape flag to select 'display'-style ('#f') or 'write'-style (#t).

Spezifikation von Operationen

Operationen auf Objekten werden in CLOS durch *generische Funktionen* definiert.

Die Signatur einer generischen Funktion wird mit *defgeneric* spezifiziert.

Klassenspezifische Methoden werden mit *defmethod* implementiert.

Im Fall einer *abstrakten Operation* definieren die generischen Funktionen durch ihre Signaturen ein *Protokoll*, das von den Methoden der Unterklassen eingehalten werden muß.

Das Protokoll der enqueue!-Operation:

Die generische Funktion ist spezialisiert für Argumente der Klasse queue und hat genau zwei Argumente.

Es wird (noch) keine Methode definiert.

```
(defgeneric* enqueue! ((qu queue) item)
  ; enqueue!: queue any -> queue
  ; Arg. item: keine Spezialisierung
  ; abstract
  :documentation
    "Push_an_item_onto_the_queue." )
```

Das Protokoll der weiteren Operationen:

```
(defgeneric* dequeue! ((qu queue))
  ; dequeue!: queue -> any
  )
(defgeneric* enqueue-items!
  ((qu queue) (items <list>))
  )
(defgeneric* head-queue ((qu queue)))
  )
(defgeneric* empty-queue? ((qu queue)))
  )
(defgeneric* enumerate-queue ((qu queue)))
```

```

)
(defgeneric* reset-queue! ((qu queue))
)

```

☞ Anmerkung:

- Im folgenden nehmen wir an, daß die Aggregation von *items* durch eine Liste implementiert wird, bei der stets das erste Element den Kopf der Schlange repräsentiert.
- Damit können alle weiteren Operationen einheitlich durch Methoden implementiert werden.
- Nur das Einfügen hängt dann von der Klasse der Schlange ab.

Implementation der Methoden

```

(defmethod head-queue ((qu queue))
  "Return_the_item_at_the_front_of_the_queue."
  (first (queue-items qu)))

(defmethod empty-queue? ((qu queue))
  "Is_the_queue_empty?"
  (null? (queue-items qu)))

(defmethod reset-queue! ((qu queue))
  "Reset_the_queue,_remove_all_items"
  (set-queue-items! qu '()))

(defmethod enumerate-queue ((qu queue))
  "Return_a_list_of_the_queue-items"
  (queue-items qu))

(defmethod dequeue! ((qu queue))
  "Remove_an_item_from_the_front
  _of_the_queue._Return_the_item."
  (pop! (queue-items qu)))

(defmethod enqueue-items!
  ((qu queue) (items <list>))
  "Push_all_items_of_a_list_onto_the_queue."
  (map (curry enqueue! qu) items)
  qu)

```

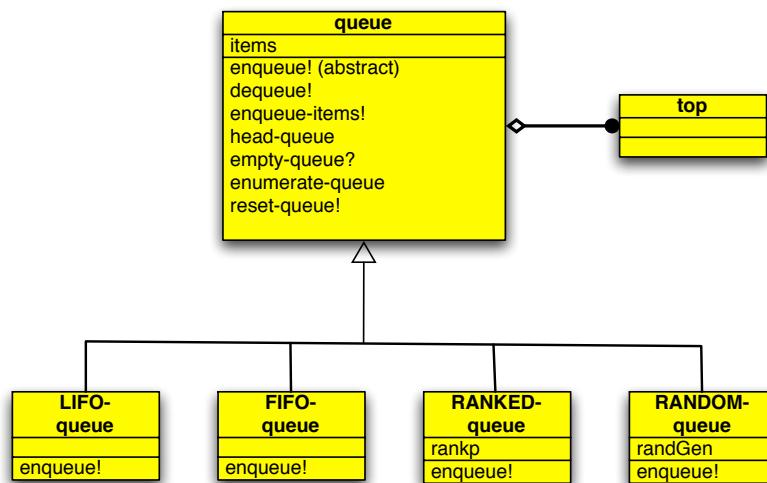
Unterklassen und Vererbung:

FIFO-Queue: Warteschlange: First in, first out,

LIFO-Queue: Stack, Stapel: Last in, first out,

RANKED-Queue: geordnete Schlangen,

RANDOM-queue: Zufällige Einreihung.



LIFO-Queue: Last in, first out

Ein Stapel (LIFO-Queue) als Unterklasse von queue:

```
(defclass* LIFO-queue (queue)
  ; all slots are inherited
  :autopred #t
  :printer #t
  :documentation
  "Queues_with_a_last-in-first-out_strategy"
  )
(defmethod enqueue! ((qu LIFO-queue) item)
  "Push_an_item_onto_the_queue."
  (push! item (queue-items qu))
  qu)
```

FIFO-queue: Warteschlange, first in, first out

```
(defclass* FIFO-queue (queue)
  (tail :reader FIFO-queue-tail
```

```

:writer set-FIFO-queue-tail!
:initvalue '()
:type <list>
:documentation
  "The last cons of the queue")
:documentation
"Queues with a last-in-first-out strategy"
:autopred #t
:printer #t
)

```

enqueue! in der FIFO-queue

```

(defmethod enqueue! ((qu FIFO-queue) item)
  "Insert an item at the tail of the queue."
  (let ((new-cons (list item)))
    (if (not (null? (queue-items qu)))
        (set-tail!
         (FIFO-queue-tail qu)
         new-cons)
        (set-queue-items! qu new-cons))
    (set-FIFO-queue-tail! qu new-cons) qu)))

```

Operationen auf Klassen und generischen Funktionen

- Definierte Klassen sind Instanzen der Klasse <class>. Die Attribute der Klassenobjekte beschreiben die direkten Oberklassen, die *slots* und die Initialisierung.
- Definierte generische Funktionen sind Instanzen der Klasse <generic>, einer Unterklasse von <function>. Die Attribute der generic-Objekte beschreiben die Stelligkeit, die hinzugefügten Methoden und die Methodenkombination.
- CLOS bietet viele *Meta-Methoden*, um Informationen über die definierten Klassen und Methoden zu erhalten.

Operationen auf Klassen

```

> (print (class-of (make RANDOM-queue)))
→ #<class:RANDOM-queue>
> (print (subclass? RANDOM-queue queue) )
→ (#<class:queue>

```

```

#<class:object>
#<class:top>
> (print (more-specific? RANDOM-queue <top>
                           (make RANDOM-queue)))
→ (#<class:top>)

```

Abfrage der definierten Attribute

```

> (print (class-slots FIFO-queue)) →
((items :reader queue-items
  :writer set-queue-items!
  :initvalue ()
  :type #<primitive-class:list>
  :documentation
  "The_items_in_the_queue")
(tail :reader FIFO-queue-tail
  :writer set-FIFO-queue-tail!
  :initvalue ()
  :type #<primitive-class:list>
  :documentation
  "The_last_cons_of_the_queue"))

```

Abfrage der Methoden einer generischen Funktion

```

> (generic-methods enqueue!) →
  (#<method:enqueue!:RANDOM-RANKED-queue ,
   <top>>
   #<method:enqueue!:RANDOM-queue , <top>>
   #<method:enqueue!:RANKED-queue , <top>>
   #<method:enqueue!:FIFO-queue , <top>>
   #<method:enqueue!:LIFO-queue , <top>>)

> (generic-arity enqueue!)
  → 2 ; die Stelligkeit

```

Definition von Methoden

- Methoden können auch direkt definiert werden, ohne vorher mit *def-generic* die Signatur zu spezifizieren.

In diesem Fall erzeugt CLOS automatisch eine entsprechende generische Funktion.

- Es ist aber guter Stil, zumindest bei abstrakten Klassen, die generischen Funktionen explizit zu definieren, da so die Schnittstelle dokumentiert wird.

Erzeugen von Objekten als Instanz einer Klasse

Objekte einer Klasse werden mit *make* erzeugt.

- *make* erzeugt die Instanz mittels der generischen Funktion *allocate-instance*.
- Anschließend wird die neue Instanz mittels der generischen Funktion *initialize* initialisiert.

```
(define (make-LIFO-queue)
  (make LIFO-queue))
```

Voreinstellungen, Schlüsselwort-Argumente

- Bei der Definition einer Klasse können *defaults* für die Attribute und Schlüsselwörter für die Initialisierung angegeben werden.

```
(defclass* eisbecher ()
  (sorte1 :initvalue 'Erdbeer
   :reader s1
   :initarg :so1)
  (sorte2 :initvalue 'Schoko
   :reader s2
   :initarg :so2)
  :printer #t)
> (make eisbecher)
#<eisbecher: sorte1=Erdbeer sorte2=Schoko>
> (make eisbecher :so2 'Zitrone)
#<eisbecher: sorte1=Erdbeer sorte2=Zitrone>
```

- Anstelle von Defaultwerten für die Initialisierung können auch parameterlose Funktionen (thunks) angegeben werden:

:initializer

```
(define (randomKugel)
  (random-elt
   '(Vanille Erdbeer Schoko Pistazie Walnuss))

(defclass* eisbecher2 ()
  (sorte1 :initializer randomKugel
          :reader s1
          :initarg :so1)
  (sorte2 :initializer randomKugel
          :reader s2)
  :printer #t)
> (make eisbecher2)
#<eisbecher2: sorte1=Walnuss sorte2=Erdbeer>
> (make eisbecher2)
#<eisbecher2: sorte1=Pistazie sorte2=Vanille>
```

RANKED-Queues: Geordnete Schlangen

Um die Elemente einer Schlange zu ordnen, müssen wir den Elementen einen Rang zuordnen können.

- Wenn diese Zuordnung für jedes Schlangenobjekt unterschiedlich sein soll, müssen wir jeder Schlange ihre eigene Rang-Methode geben können.

```
(defclass* RANKED-queue (queue)
  (rankp :reader theRankp
   :initarg :rankP
   :type <function>
   :documentation
   "compute_the_rank_of_an_item"))
```

Implementation der RANKED-queue

```
(defmethod rLess? ((qu queue))
  ;rLess?: queue -> (<top><top> -> boolean)
  "Generate_a_less?-predicate_using_rankp"
  (lambda (item1 item2)
    (< ((theRankp qu) item1)
      ((theRankp qu) item2))))
```



```
(defmethod enqueue! ((qu RANKED-queue) item)
  "Insert_an_item_according_to_the_rank."
```

```

(set-queue-items!
 qu
 (sort
  (cons item (queue-items qu))
  (rLess? qu)))
 qu)

(defclass* RANDOM-queue (queue)
  (randGen :reader theRandomGen
            :initvalue (lambda (x) (random)))
  :initarg :randGen
  :type <function>
  :documentation
  "a_procedure_to_generate_a_random_position
   in_the_queue"
  ; <top> -> real >= 0
  )
 :documentation "Queues_that_are_ordered_randomly"
 :autopred #t
 :printer #t
)
(defmethod randomLess? ((qu queue))
 ;randomLess?: queue -> (<top><top> -> boolean)
 "Generate_a_less?-predicate_according_to_randGen"
 (lambda (item1 item2)
  (< ((theRandomGen qu) item1)
    ((theRandomGen qu) item2))))
 
(defmethod enqueue! ((qu RANDOM-queue) item)
 "Insert_an_item_at_a_random_Position."
 (set-queue-items!
 qu
 (sort
  (cons item (queue-items qu))
  (randomLess? qu)))
 qu)

```

Ein Beispiellauf

```
> (let ((ranked
          (make RANKED-queue :rankP id)))
    (begin (enqueue! ranked 3)
           (enqueue! ranked 1)
           (enqueue! ranked 2)
           (enqueue! ranked 4)
           (list (dequeue! ranked)
                 (dequeue! ranked)
                 (dequeue! ranked)
                 (dequeue! ranked))))
→ (1 2 3 4)
```

☞ Anmerkung: Vorteil der Vererbung:

Am Beispiel der Warteschlangen ist gut zu sehen, wie durch Vererbung ein Programmpaket konsistent erweitert werden kann. Neue Klassen können so eingeführt werden, dass bestehende Methoden genutzt werden und eine Duplizierung und Redundanz von Code vermieden wird. Dieses gelingt aber nur, wenn beim Entwurf der Klassen sorgfältig geplant wird, welche Gemeinsamkeiten sich zur Mehrfachnutzung anbieten. Objekt-orientierte Programmierung allein schützt nicht davor, schlecht entworfene, unverständliche Programme zu produzieren. Die Methode allein erspart uns nicht das Nachdenken; wir müssen sie mit Verstand anwenden. Der Spaghetticode der objektorientierten Programmierung sind schlecht geplante, unmotivierte Vererbungshierarchien und Aggregationen, die umso eindrucksvoller erscheinen, je konfuser sie sind.

27.2 Mehrfachvererbung und Methodenkombination

Mehrfachvererbung

- Klassen können von *mehreren* Oberklassen erben.
- Das parallele Erben von mehreren direkten Oberklassen heißt *Mehrfachvererbung*.
- Mehrfachvererbung wird in Lisp (CLOS) häufig angewendet, um
 - Aggregate zu modellieren,
 - um durch „mixin“-Klassen eine bestimmte Funktionalität zur Verfügung zu stellen,
 - um Objekte zu modellieren, die in sich die Eigenschaften mehrerer Klassen vereinigen.

Was wird vererbt?

Eine Klasse erbt von den Oberklassen

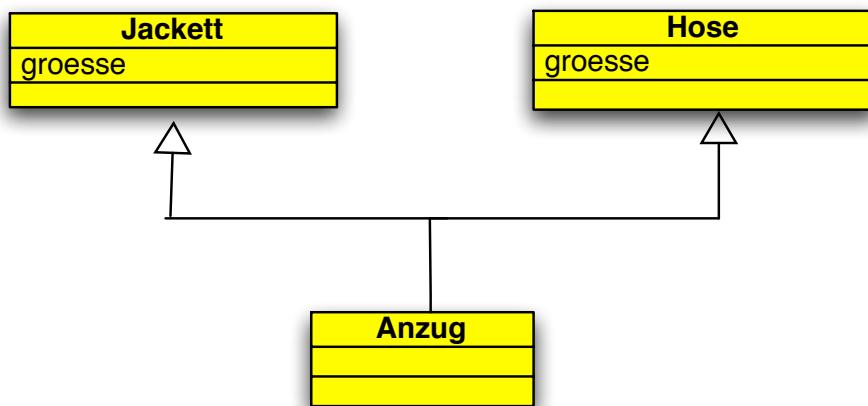
- alle anwendbaren Methoden
- sowie die Vereinigungsmenge der Attribute (*slots*).
- Wenn die Oberklassen Attribute (*slots*) mit gleichem Namen haben, werden diese Attribute nur einmal vererbt.

☞ **Anmerkung:** Nicht alle Programmiersprachen lassen Mehrfachvererbung zu, beispielweise Oberon oder Java verbieten die Mehrfachvererbung, da so Operationen auf unterschiedlichen Wegen mehrfach geerbt werden können und aus dem Vererbungsbaum ein Netzwerk werden kann. In Java gibt es als Alternative zur Mehrfachvererbung die Interfaces, mit denen sich eine ähnliche Modellierung erreichen lässt, allerdings mit erhöhtem Aufwand, da das Interface jedesmal neu implementiert werden muß.

Komplizierte Erbschaftsverhältnisse sind allerdings nur dann gefährlich, wenn wir sie bei Programmieren nicht durchschauen und dann Fehler machen. In CLOS ist die Vererbung aber klar geregelt, auch bei netzartiger Vererbung. Im Zweifelsfall können Sie im Interpreter jede Klasse nach den geerbten *slots* und Methoden und deren Präzedenzen fragen.

Mehrfachvererbung kann sehr sinnvoll sein, wenn wir Objekte modellieren, die in natürlicher Weise die Eigenschaften zweier Klassen in sich vereinigen, oder wenn wir bestimmte Funktionalitäten zu einer Klasse hinzufügen wollen (mixin-Klassen). Wenn wir dieses über Interfaces realisieren wollen, und das Interface mehrfach benötigen und implementieren, entsteht unnötige und gefährliche Redundanz, da wir eventuell mehrere widersprüchliche Implementationen desselben Interfaces erstellen. Die Standardlösung ist dann, eine Implementationsklasse für das Interface zu erstellen, auf die alle Neuimplementierungen zurückgreifen, aber dieser Weg ist auch nicht sehr befriedigend, da er den Code noch mehr aufbläht.

Beispiel für die Vererbung der Attribute



Definition der Klasse Anzug in CLOS

```
(defclass Jackett ()
  (groesse :initvalue 40 :accessor gr)
  :printer #t )
(defclass Hose ()
  (groesse :initvalue 44 :accessor grHose)
  :printer #t )
(defclass Anzug (Jackett Hose)
  :printer #t )
```

```
> (make Anzug) → #<Anzug: groesse=40>
```

Inspektion der Klasse

```
> (class-slots Anzug) →
((groesse :initvalue 40 :accessor gr
  :initvalue 44 :accessor grHose))
```

```

> ( subclass? Anzug Hose) →
  (#<class:Hose> #<class:object> #<class:top> )
> (instance-of? (make Anzug) Jackett) →
  (#<class:Jackett>
   #<class:Hose>
   #<class:object>
   #<class:top> )

```

- Beobachtung:
- Der slot „groesse“ wurde in der Klasse „Anzug“ nur einmal angelegt.
 - Beide Akzessorfunktionen „gr“ und „grAnzug“ wurden vererbt.

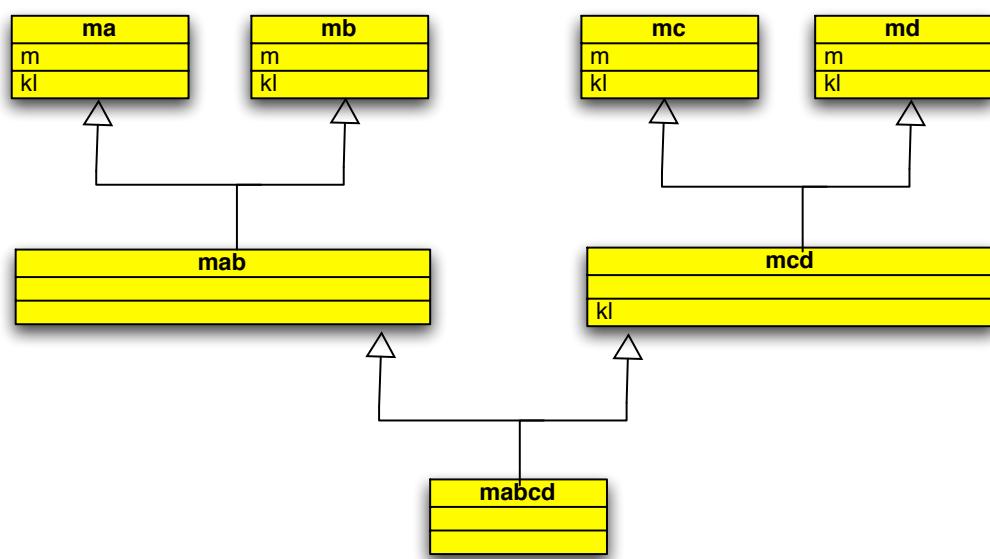
Regelung von Erbschaftskonflikten

- CLOS legt eine *Präzedenzliste* für die Klassen an, die nach folgenden Regeln geordnet ist:
 - Jede Klasse hat Vorrang vor ihren Oberklassen.
 - Jede Klasse legt die Präzedenz der direkten Oberklassen fest (bei Mehrfachvererbung).
- Es muß eine strikte Ordnung auf den Klassen existieren, die diese beiden Regeln erfüllt, sonst signalisiert CLOS einen Fehler.
- Methoden mit höherer Präzedenz überladen und verschatten Methoden von geringerer Präzedenz.

☞ **Anmerkung:** Wenn sogar mehrere strikte Ordnungsrelationen den Präzedenzgraph der Klassen erfüllen, dann ist unbestimmt, welche Ordnung CLOS berücksichtigen wird. Normalerweise sollte das kein Problem sein, denn es bedeutet, daß bei der Definition der Klassen Präzedenzbeziehungen offen geblieben sind, weil sie für das Programm nicht relevant sind. Sollte sich aber herausstellen, daß diese Präzedenzen doch wichtig sind, dann kann man immer nachhelfen, indem die Ordnung der betreffenden Klassen in einer Liste von Oberklassen explizit gemacht wird.

Die Präzedenzliste (class precedence list) der Oberklassen einer Klasse kann mithilfe der generischen Funktion class-cpl abgefragt werden.

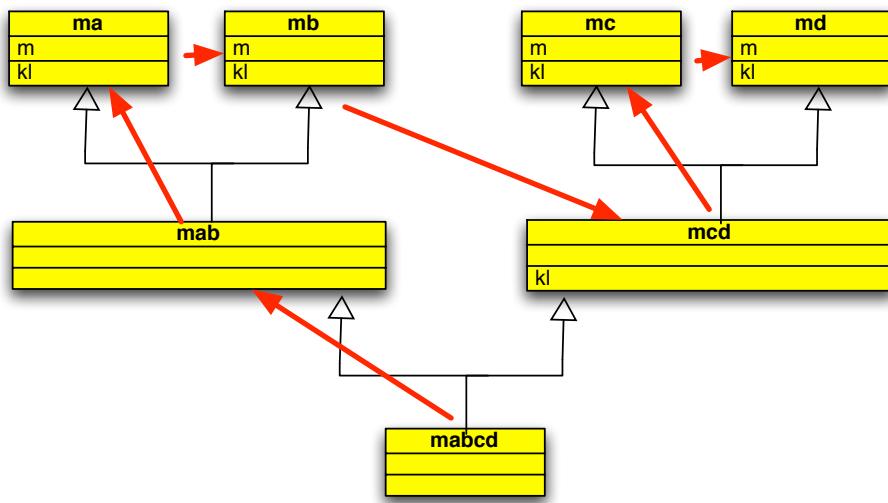
Beispiel



Baumartige Mehrfachvererbung

```
(defclass ma ()  
  (m :initvalue 'ma) :printer #t)  
(defclass mb ()  
  (m :initvalue 'mb) :printer #t)  
(defclass mc ()  
  (m :initvalue 'mc) :printer #t)  
(defclass md ()  
  (m :initvalue 'md) :printer #t)  
(defclass mab (ma mb) :printer #t)  
(defclass mcd (mc md) :printer #t)  
(defclass mabcd (mab mcd) :printer #t)
```

Klassenpräzedenzgraph



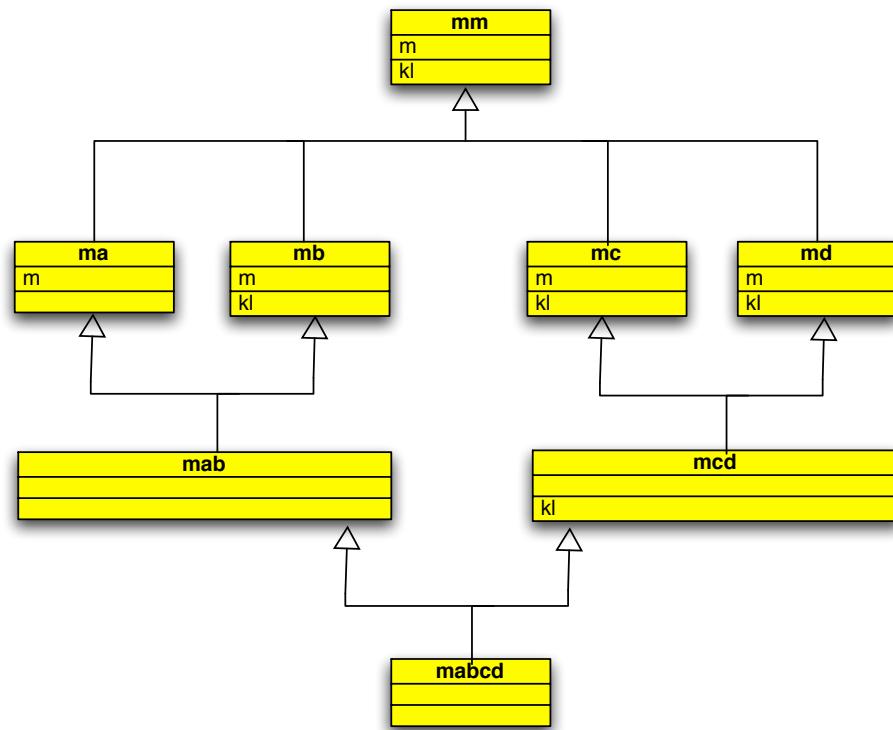
```
> (class-cpl mabcd) → ; Präzedenzliste
(#<class:mabcd> #<class:mab> #<class:ma>
 #<class:mb> #<class:mcd> #<class:mc>
 #<class:md> #<class:object> #<class:top> )
```

Vererbte Methoden

```
(defmethod kl ((k ma)) (display 'ma))
(defmethod kl ((k mb)) (display 'mb))
(defmethod kl ((k mc)) (display 'mc))
(defmethod kl ((k md)) (display 'md))
(defmethod kl ((k mcd)) (display 'mcd))

> (kl (make mabcd)) → ma
```

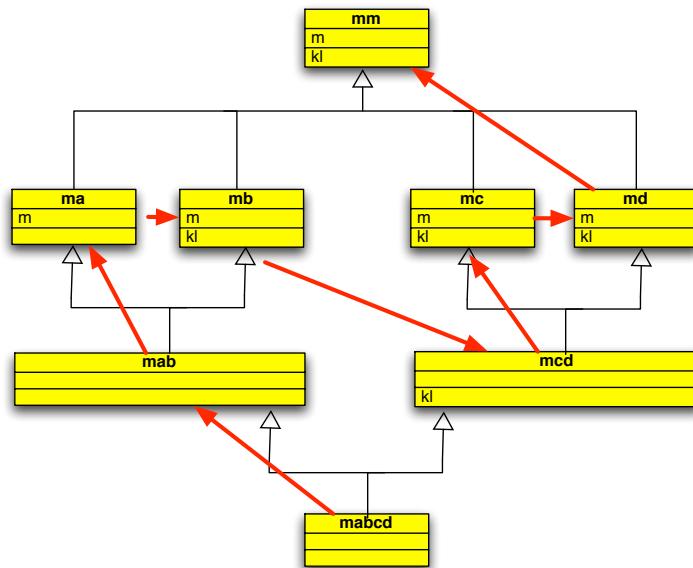
Beispiel für Vererbung auf mehreren Wegen



```

(defclass* mm () (m :initvalue 'ma) :printer #t)
(defmethod kl ((k mm)) (display 'mm))
(defclass* ma (mm) (m :initvalue 'ma) :printer #t)
(defclass* mb (mm) (m :initvalue 'mb) :printer #t)
(defclass* mc (mm) (m :initvalue 'mc) :printer #t)
(defclass* md (mm) (m :initvalue 'md) :printer #t)
(defclass* mab (ma mb) :printer #t)
(defclass* mcd (mc md) :printer #t)
(defclass* mabcd (mab mcd) :printer #t)
(defmethod kl ((k mb)) (display 'mb))
(defmethod kl ((k mc)) (display 'mc))
(defmethod kl ((k md)) (display 'md))
(defmethod kl ((k mcd)) (display 'mcd))
  
```

Die Klassenpräzedenzliste



```
> ( kl (make mabcd)) →mb
```

```
> (class-cpl mabcd) →
(#<class:mabcd> #<class:mab> #<class:ma> #<class:mb>
#<class:mcd> #<class:mc> #<class:md> #<class:mm>
#<class:object> #<class:top> )
> ( kl (make mabcd)) →mb
```

Methodenkombination

- Wenn für ein Objekt mehrere Methoden anwendbar sind, wird im Standardfall die spezifischste Primärmethode (entsprechend der Klassenpräzedenzliste) ausgeführt.
- Gelegentlich kann es aber sinnvoll sein, *alle* anwendbaren Methoden auszuführen und die Ergebnisse zusammenzufassen.
- Für jede generische Funktion kann eine Methodenkombination spezifiziert werden, die beschreibt, wie die anwendbaren Methoden zu kombinieren sind.
- Alle Methoden, die eine bestimmte generische Funktion implementieren, müssen aber dieselbe Methodenkombination verwenden.

Preis und Farbe für einen Anzug

Beispiel: 133 (Methodenkombination)

Für ein Aggregat Anzug:

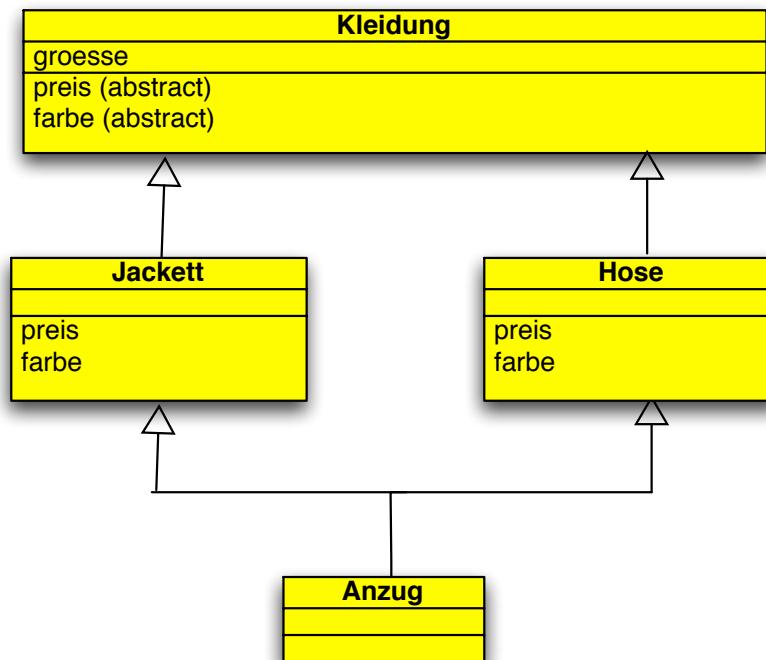
Der Preis für einen Anzug sollte

- die Summe der Preise von Jackett und Hose sein,
- nicht nur der Preis für Jackett oder Hose.

Die Farbe für einen Anzug sollte

- eine Liste der Farben aller Einzelteile sein,
- nicht nur die Farbe von Jackett oder Hose.

Beispiel für Methodenkombination



Methodenkombinationen: +, append

```
(defclass Kleidung ()  
  (groesse :initvalue 52  
            :accessor gr)  
  :printer #t )  
  
(defgeneric preis ((kleidung))  
  :combination generic+-combination)  
  
(defgeneric farbe ((kleidung))  
  :combination generic-append-combination)  
  
(defclass Jackett (Kleidung))  
  
(defclass Hose (Kleidung))  
  
(defclass Anzug (Jackett Hose))
```

Implementation der generischen Funktionen

```
(defmethod preis ((j Jackett))  
  20)  
(defmethod preis ((h Hose))  
  40 )  
  
(defmethod farbe ((j Jackett))  
  '(rot) )  
(defmethod farbe ((h Hose))  
  '(blau) )
```

☞ **Anmerkung:** Operatormethodenkombination: Unter der Methodenkombination generic+-combination werden alle anwendbaren Methoden einer generischen Funktion angewendet und die Resultate addiert.

Unter der Methodenkombination generic-append-combination werden alle anwendbaren Methoden einer generischen Funktion angewendet und die Resultate mit append zusammengefaßt.

Unter der Methodenkombination generic-begin-combination werden alle anwendbaren Methoden einer generischen Funktion angewendet und das Resultat der letzten Anwendung zurückgegeben.

Die Methodenkombination generic-and-combination führt die anwendbaren Methoden einer generischen Funktion nacheinander aus, bis ein Resultat #f ergibt.

Die Methodenkombination generic-or-combination führt die anwendbaren Methoden einer generischen Funktion nacheinander aus, bis ein Resultat #t ergibt. (siehe das Online-Manual im DrRacket-Helpdesk für tiny-closs.ss)

Das Anzug-Beispiel wurde von [Graham, 1996](#) übernommen.

Beispieldlauf

```
> (preis (make Jackett)) → 20
> (preis (make Anzug)) → 60
> (farbe (make Jackett)) → (rot)
> (farbe (make Hose)) → (blau)
> (farbe (make Anzug)) → (rot blau)
```

Weitere Methodenkombinationen

generic-+-combination
generic-list-combination
generic-min-combination
generic-max-combination
generic-append-combination
generic-append!-combination
generic-begin-combination
generic-and-combination
generic-or-combination

Methodenkombination

Akzessorfunktionen

Auch die Akzessorfunktionen für die *slots* sind generische Funktionen, die frei kombiniert werden können.

- ☞ Wir können die Methodenkombination also auch auf die Zugriffsfunktionen für Attribute anwenden.

Beispiel: 134 (Methodenkombination für den slot „Preis“:)

- Wir speichern die Preise für die Einzelteile eines Anzugs in den „preisJ“ für das Jackett, „PreisH“ für die Hose.
- Für beide slots heißt der Akzessor „preis“ und verwendet die generic-+-combination.

```

(defgeneric preis ((<top>))
  :combination generic---combination)

(defclass Jackett ()
  (preisJ :initarg :preisJ :accessor preis)
  :printer #t)
(defclass Hose ()
  (preisH :initarg :preisH :accessor preis)
  :printer #t)
(defclass Anzug (Jackett Hose)
  :printer #t)

(define a
  (make Anzug :preisH 100 :preisJ 300))
> a → #<Anzug: preisH=100 preisJ=300>
> (preis a) → 400

```

27.3 Ergänzungsmethoden

Ergänzung einer Methode

Wenn wir Klassen durch Unterklassen spezialisieren, dann ist es oftmals so, daß wir die Methoden der Oberklasse fast vollständig übernehmen könnten.

Sie müssen nur durch vorbereitende oder nachbereitende Aktionen ergänzt werden.

Zwei Ansätze:

Es gibt zwei Wege, um in dieser Situation nicht die Methoden völlig neu schreiben zu müssen, sondern die übergeordneten Methoden mitzubenutzen:

1. Den „super call“: bekannt aus Java
2. Ergänzungsmethoden: Spezielle Hilfsmethoden (nur in CLOS)

Ergänzung einer Methode

„super call“: Wir rufen mit *call-next-method* direkt die Methode der Oberklasse auf, und führen vorher oder nachher die notwendigen Zusatzoperationen aus.

Ergänzungsmethoden: Wir ergänzen die primäre Methode der Oberklasse durch Hilfsmethoden, die vorher oder nachher (oder beides) zusätzlich auszuführen sind.

Es gibt

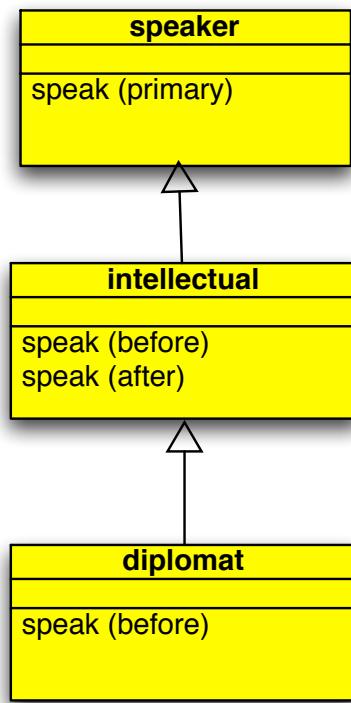
- Vormethoden,
- Nachmethoden
- oder einhüllende Methoden,

kenntlich durch die Schlüsselwörter **:before**, **:around**, **:after**.

☞ Anmerkung: Ergänzungsmethoden erfordern die Standard-Methodenkombination: Zuerst alle Vormethoden, dann die speziellste einhüllende Methode oder Primärmethode, dann alle Nachmethoden.

Wenn Ergänzungsmethoden verwendet werden, sind andere Methodenkombinationen unzulässig!

Beispiel: Vorsichtige Rede



Ein sehr direkter Redner

Ein schlichter Redner, der alles einfach so sagt, wie es ist:

```
(defclass speaker ())  
  
(defmethod speak ((s speaker) text)  
  (display text)); primary method  
  
> (define s (make speaker))  
> (speak s "I_am_hungry")  
I am hungry
```

Spezialisierte vorsichtige Redner

Redner, die ihre Aussagen vorsichtig einschränken:

```
(defclass intellectual (speaker))  
  
(defmethod speak :before
```

```

((s intellectual) text)
(display "I\_think\_"))

(defmethod speak :after
  ((s intellectual) text)
  (display "\_in\_some\_sense"))

> (define i (make intellectual))
> (speak i "I_am_hungry")
I think I am hungry in some sense

```

Noch vorsichtiger: Ein Diplomat

```

(defclass diplomat (intellectual))

(defmethod speak :before
  ((d diplomat) text)
  (display "Perhaps"))

> (define d (make diplomat))
> (speak d "the_world_is_round")

Perhaps I think the world is round
in some sense

```

Ausführung der Methoden

- Zunächst werden *alle* anwendbaren *Vormethoden* ausgeführt, beginnend bei der *spezifischsten*.
- Dann wird die spezifischste *Primärmethode* ausgeführt.
- Danach werden *alle* anwendbaren *Nachmethoden* ausgeführt, beginnend bei der *allgemeinsten*.

Beispiel

```

(defclass* klasseA ())
(defclass* klasseB (klasseA))
(defclass* klasseC (klasseB))

(defmethod teste ((a klasseA))
  (writeln "Klasse_A"))

```

```

(defmethod teste :before ((b klasseB))
  (writeln "vor_b:"))
(defmethod teste :after ((b klasseB))
  (writeln "nach_b:"))
(defmethod teste :before ((c klasseC))
  (writeln "vor_c:"))
(defmethod teste :after ((c klasseC))
  (writeln "nach_c:"))

> (define c (make klasseC))
> (teste c)
vor c:
vor b:
Klasse A
nach b:
nach c:
>

```

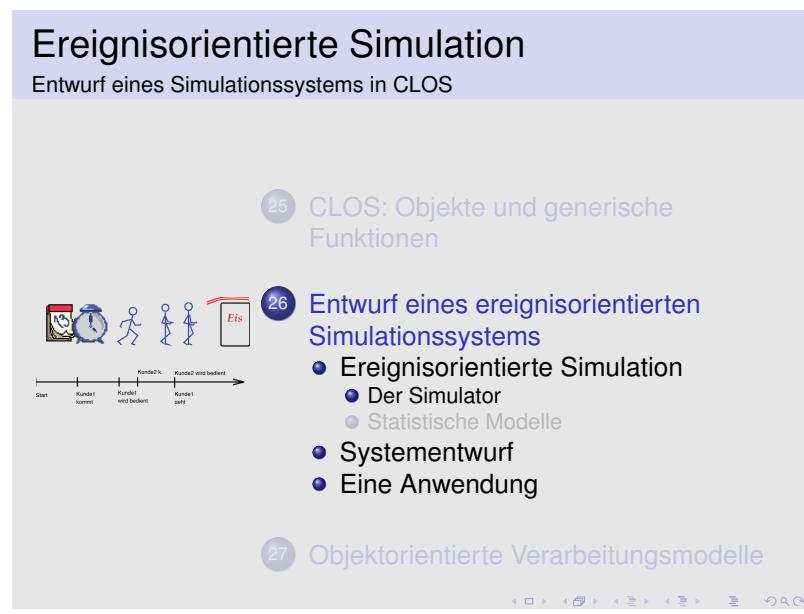
Vorteil der Ergänzungsmethoden

- Im Gegensatz zum super call ist bei Ergänzungsmethoden sichergestellt, daß alle Ergänzungsmethoden ausgeführt werden.
So können keine Initialisierungen vergessen oder unterdrückt werden, die in den Oberklassen definiert wurden.
- Die geerbten Methoden brauchen nicht durch Modifikationen überladen zu werden, sondern werden nur ergänzt.

28 Entwurf eines ereignisorientierten Simulationssystems

28.1 Ereignisorientierte Simulation

28.1.1 Der Simulator



Simulation

- Bei der *Simulation* entwerfen wir ein *Modell* der Anwendungsdomäne,
 - indem wir Experimente durchführen können,
 - deren Ergebnisse wir auf die reale Welt übertragen.
- Simulation kann sinnvoll eingesetzt werden,
 - wenn der Versuch in der realen Welt technisch nicht möglich,
 - zu teuer
 - oder zu gefährlich ist.

☞ **Anmerkung:** Ein Simulationsmodell ist gut, wenn es von allem Unwesentlichen abstrahiert, aber alle für unser Experiment wichtigen Eigenschaften korrekt wiedergibt und explizit macht. Wir müssen durch einen *Validierungsprozeß* sicherstellen, daß diese Bedingung auch erfüllt ist, sonst sind die Ergebnisse der Simulation wertlos.

Modelle können konkrete physikalische Nachbildungen sein oder virtuell durch ein Simulationsprogramm erzeugt werden. Ein rechnerbasierter Simulationsmodell hat gegenüber physikalischen Modellen den Vorteil, daß es wirklich abstrakt ist. Materielle Objektmodelle können unbeabsichtigte Eigenschaften haben, die den Versuch ungeplant beeinflussen. Versuche mit einem Simulationsprogramm sind beliebig oft reproduzierbar, ohne daß unsere Modelle verschleißen. Versuche können schneller durchgeführt werden, als wenn Materie bewegt werden müßte. Das Modell ist leichter zu modifizieren.

Simulation ist die klassische Anwendungsdomäne für die objektorientierte Programmierung, weil wir das Modell direkt durch Programmobjekte beschreiben können, die sich bijektiv auf Domänenobjekte abbilden lassen.

Simulationsmodelle

An einem Modell sind Versuche möglich, die wir in der Wirklichkeit nicht durchführen dürfen oder können.

Der Simulationszyklus:

Entwurf eines Modells der Anwendungsdomäne durch Abstraktion.

Validierung des Modells.

Simulieren der Anwendungsprozesse im Modell.

Rückübertragung der Ergebnisse in die Anwendungsdomäne.

Vorteil virtueller Modelle

Materielle Modelle sind nicht wirklich abstrakt.

Ihre konkreten Eigenschaften können das Experiment verfälschen.

- Virtuelle Modelle dagegen sind verschleißfrei,
- beliebig oft zu reproduzieren,
- leicht zu parametrisieren,
- schneller zu modifizieren als materielle Objekte
- und sie können die Experimente selbst protokollieren.

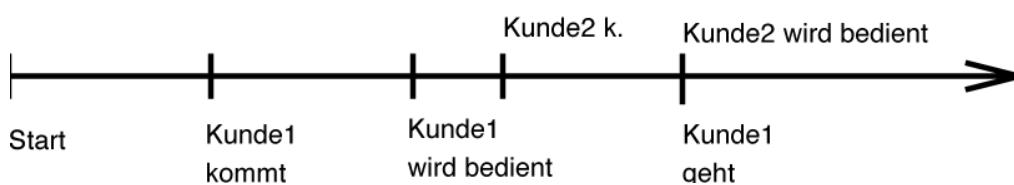
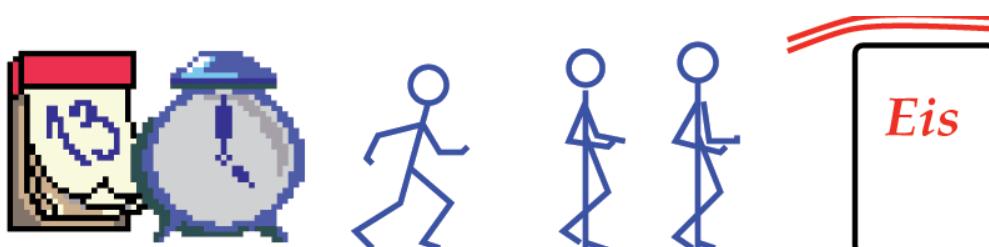
⌚ Anmerkung: Diskrete, ereignisgesteuerte Simulation

Bei der ereignisorientierten Simulation werden Abläufe nachgebildet, deren Effekte sich auf diskrete Ereignisse zurückführen lassen. Dieses Verfahren ist zwar ungeeignet für Fragestellungen, bei denen ein Prozeß kontinuierlich beobachtet werden muß, aber für eine ganze Reihe von interessanten Anwendungen geeignet, beispielsweise Bediensysteme oder Lagerhaltungssysteme.

Bei dieser Art von Simulation wird der Ablauf der Zeit nur an diskreten Ereignissen sichtbar. Zwischen den Ereignissen vergeht die Zeit in Sprüngen. Ereignisorientierte Simulation ist nur eine Variante von Simulationsverfahren. Weitere Verfahren finden Sie bei [Page, 1991](#). Wir werden hier ein Simulationssystem vorstellen, das in Anlehnung an das von [Reiser and Wirth, 1994](#) in Oberon entwickelte System neu in CLOS entworfen wurde.

Das Kernstück eines ereignigesteuerten Simulationssystems ist ein Kalender von Ereignissen, zusammen mit einer Uhr, die die Zeit in der virtuellen Welt angibt. Jeweils das nächste anstehende Ereignis wird vom Kalender abgelesen, die Uhr wird vorgestellt, das Ereignis tritt ein und löst weitere Ereignisse aus, deren Eintreten im Kalender vermerkt wird. Bei einem Bediensystem sind solche Ereignisse beispielsweise: Das Eintreffen eines Kunden, Start der Bedienung, Ende der Bedienung usw. Damit das Simulationssystem „lebendig“ bleibt, muß jedes Ereignis ein Folgeereignis auslösen, das im Kalender vermerkt wird. Wenn wir beispielsweise damit beginnen, einen Kunden zu bedienen, dann planen wir gleich das Ende der Bedienung. Sollte der Kalender leer werden, steht die Zeit still und wir sprechen von einer Verklemmung (deadlock).

Zeitdiskrete Simulation



Schema eines Bediensystems:

REPEAT *Lese nächstes anstehendes Ereignis im Kalender.*

CASE Ereignis

- Ankunft eines Kunden:
Plane nächstes Ankunftsereignis.
Schlange leer? Bediene den Kunden,
ansonsten Einreihen in die Schlange.
- Ende einer Bedienung:
Kunde verläßt die Warteschlange.
Bediene den nächsten Kunden,
falls die Schlange nicht leer ist.

UNTIL Simulationszeit abgelaufen.

28.1.2 Statistische Modelle

☞ Anmerkung: Statistische Modelle

Viele Fragen, die durch Simulation beantwortet werden sollen, betreffen Vorhersagen über das Verhalten einer sehr großen Menge von gleichartigen Individuen. In diesen Fällen sind analytische Lösungen praktisch nicht möglich, und wir entwerfen ein statistisches Modell, in dem wir das Auftreten von Ereignissen durch geeignete Verteilungen beschreiben.

Ein wichtiger Schritt bei der Validierung unserer Modelle besteht darin, daß wir prüfen, ob die angenommenen Verteilungen wirklich das reale Verhalten der simulierten Objekte beschreiben. Da Sie die notwendigen mathematischen Grundlagen hierzu erst in späteren Vorlesungen kennenlernen werden, wollen wir hier nur einige typische Verteilungen ansprechen, die bei der Simulation immer wieder benötigt werden, und die Frage der Validierung unserer Modelle zurückstellen.

Statistische Modelle

- Die Simulation von Vorgängen, die eine große Menge gleichartiger Objekte betreffen, wird statistisch modelliert.
- Die Simulationsereignisse werden als *Zufallsgrößen* modelliert, und die Wahrscheinlichkeiten für das Auftreten bestimmter Werte durch *Verteilungen* beschrieben.

Typische Verteilungen

Gleichverteilung

Die **Gleichverteilung** dient zur Beschreibung von Ereignissen, die zu jedem Zeitpunkt gleichwahrscheinlich auftreten können, beispielsweise

- das Eintreffen eines Kunden,
- der Zerfall eines Atomkerns usw.

Dichte: $p(x) = c$

Exponentialverteilung:

Wartezeit zwischen gleichverteilten Ereignissen

- Dies Exponentialverteilung beschreibt die Verteilung der *zeitlichen Abstände* t zwischen zwei gleich verteilten Ereignissen.
- Die Wahrscheinlichkeit von langen Wartezeiten t ist geringer als für kurze Wartezeiten und geht für unendlich lange Wartezeiten gegen Null.

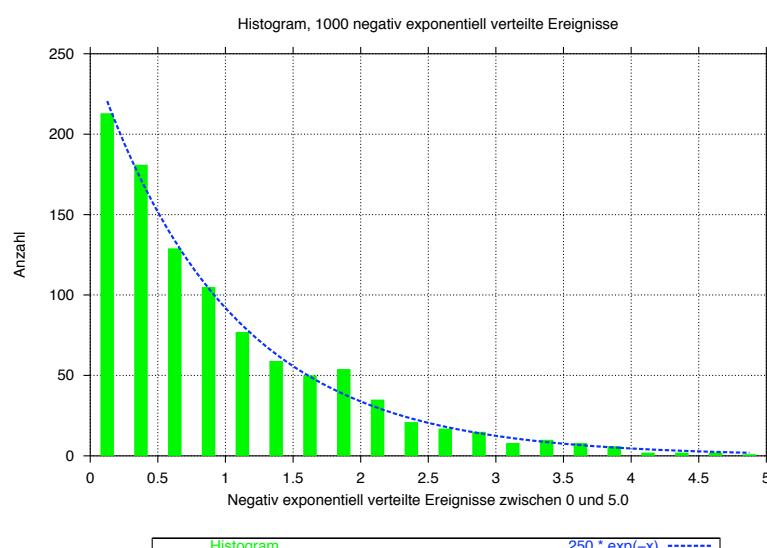
Dichte: $p(t) = \begin{cases} 0 & \text{für } t = 0 \\ \mu \cdot e^{-\mu t} & t > 0 \quad (\mu > 0) \end{cases}$

Typisch für diese Verteilung ist die *Halbwertszeit*:

- In jeweils gleichen Abständen sinkt die Wahrscheinlichkeit auf die Hälfte des vorherigen Wertes.

1000 negativ exponential verteilte Ereignisse

Histogramm



Normalverteilung:

Zur Beschreibung von Ereignissen, deren Werte mit mehr oder minder großen Abweichungen um einen Mittelwert schwanken.

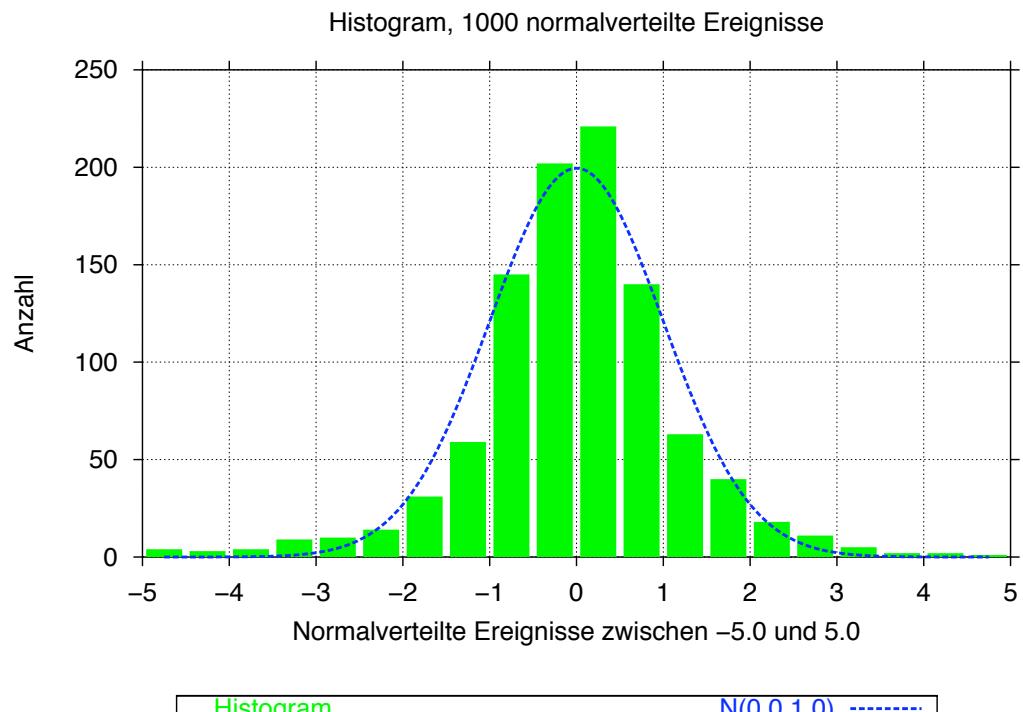
$$p(t) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(t-\mu)^2}{2\sigma^2}}, \quad \sigma > 0$$

Die Größe μ heißt der *Mittelwert* der Normalverteilung und die Größe σ^2 die *Varianz*. Je größer σ , desto stärker streut die Verteilung um den Mittelwert.

Bei einer sehr großen Stichprobe, die normalverteilt ist, liegen ungefähr 95% der Werte im Intervall $|t - \mu| \leq 2\sigma$. Abkürzende Schreibweise: $N(\mu, \sigma)$

1000 normalverteilte Ereignisse

Histogramm



☞ **Anmerkung:** Erzeugung zufälliger Ereignisse mit dem Rechner

Die Güte einer Simulation hängt nicht nur entscheidend davon ab, daß unser statistisches Modell stimmt, sondern auch davon, wie gut wir dieses statistische Modell in unserer simulierten Welt implementieren können. Wir müssen zufällige Ereignisse erzeugen, die denselben Verteilungen unterliegen, wie wir sie für die reale Welt postuliert haben. Eine sehr lesenswerte und vollständige Übersicht hierzu finden Sie in Knuths Kompendium ([Knuth, 1973](#), Band 2), deutlich knapper, aber auch lesenswert: ([Press et al., 1988](#)).

Da unsere Rechenanlagen deterministische Maschinen sind, können wir nicht wirklich zufällige Ereignisse auslösen, wir sprechen von *quasi-zufälligen Ereignissen*. Mit einem Zufallszahlengenerator erzeugen wir *reproduzierbare, aber zufällig erscheinende* Folgen von Ereignissen. Eine anwendungsbezogene Definition, was Zufälligkeit bei einem Programm bedeutet, finden Sie bei [Press et al., 1988](#).

In diesem Buch finden Sie auch viele Hinweise, wie Sie einen Zufallszahlengenerator testen können und wie Sie die Generatoren, die Sie in den die Programmbibliotheken finden, notfalls verbessern können.

A working, though imprecise, definition of randomness in the context of computer generated sequences, is to say that the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that *uses* its output. In other words, any two different random number generators ought to produce statistically the same results when coupled to your particular application program. If they don't, then at least one of them is not (from your point of view) a good generator.

The above definition may seem circular, comparing, as it does, one generator to another. However, there exists a body of random number generators which mutually do satisfy the definition over a very, very broad class of application programs. And it is also found empirically that statistically identical results are obtained from random numbers produced by physical processes. ([Press et al., 1988](#)).

Erzeugung gleichverteilter Zahlen

Die Kongruenzmethode:

Die Kongruenzmethode erzeugt eine Folge von ganzen Zahlen $I_1, I_2, I_3 \dots$ aus dem Intervall $[0 \dots m]$ nach der Rekursionsformel

$$I_{j+1} = (a \cdot I_j + c) \bmod m.$$

- Auf diese Weise können maximal m verschiedene Zahlen erzeugt werden, bevor die Folge sich wiederholt.
- Die Güte des Verfahrens hängt davon ab, wie gut die Parameter a, c, m gewählt wurden.

☞ Anmerkung: Man kann die Folge von Zahlen, die mit der Kongruenzmethode erzeugt wird, an einem *zufälligen Punkt* beginnen lassen, wenn das Startelement I_0 zufällig gewählt wird, beispielsweise aus der Uhrzeit abgeleitet wird. So können Sie bei interaktiven Spielprogrammen vermeiden, daß die Programme sich vorhersagbar verhalten.

Lesen Sie bei Knuth oder Press nach, warum es so wichtig ist, daß ein Zufallszahlengenerator unkorrelierte Zahlenfolgen liefert, und wie Sie sich vergewissern können, daß Ihr Zufallszahlengenerator diesen Anspruch erfüllt. Besonders Besitzer von Spielautomaten sollten sich sehr genau vergewissern, daß ihre Zufallszahlen wirklich zufällig und nicht vorhersagbar sind.

Die Programmabibliotheken enthalten meistens nur Zufallszahlengeneratoren für gleichverteilte Zufallszahlen. Wie können wir normalverteilte oder negativ exponentiell verteilte Zahlen erzeugen?

Die Lösung: Wir erzeugen gleichverteilte Zufallszahlen und leiten daraus beliebig verteilte Zufallszahlen ab. Das ist möglich, wenn wir die Umkehrfunktion des Integrals der gewünschten Dichtefunktion kennen.

Beliebig verteilte Zufallszahlen

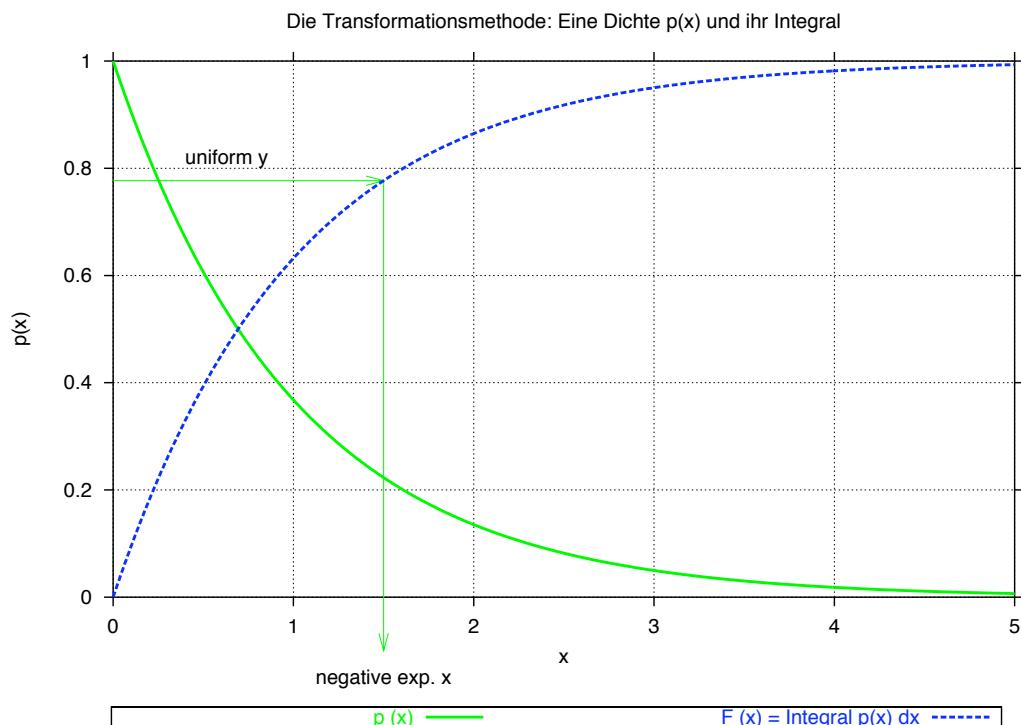
Die Transformationsmethode

Wenn die von uns gewünschte Dichtefunktion $p(x)$

- integrierbar ist
- und für das unbestimmte Integral die *Umkehrfunktion* existiert,

erhalten wir Zufallszahlen verteilt mit der Dichte $p(x)$, wenn wir *gleichverteilte* Zufallszahlen mit der Umkehrfunktion des unbestimten Integrals der Dichte transformieren.

Transformation mittels des Integrals



```
(define (random-uni &key (range 1.0)
                           (granularity 2147483647))
  "A_random_real_number,_uniformly_distributed
  between_0_and_range."
  (* (/ (random granularity)
        granularity)
     range))
```

```

(define (random-sig &key (p-positive 0.5))
  "A_random_signum,_1_or_-1"
  (if (> (- (random-uni) p-positive) 0.0)
      -1 1))

(define (random-exp &key (1/mu 1.0))
  "Return_a_random_number,
   with_a_negative_exponential_distribution ,
   mean_value_1_/_'1/mu',_e.g._arrival_rate
   '1/mu'in_a_simulation."
  (/ (- (log (random-uni)))
     1/mu))

(define (random-normal &key (mu 0.0) (sigma 1.0))
  "Return_a_random_number,_with_a_normal_distribution ,
   mean_value_mu,_variance_sigma^2."
  (let* ((v (1- (* 2 (random-uni)))))
    (r (abs v))
    (fact (sqrt (/ (* -2 (log r) r))))))
  (+ mu (* fact v sigma)))

(define (random-mixture
           randoms-1 frac-1 randoms-2)
  "Random_numbers_obeying_a_mixture_distribution"
  (if (> (random-uni) frac-1)
      (randoms-1)
      (randoms-2)))

(define (random-normal-mixture
           &key (mu-1 1.0) (sigma-1 1.0)
           (mu-2 2.0) (sigma-2 1.0)
           (frac-1 0.5))
  "Random_numbers_obeying_a_mixture_distribution"
  (random-mixture
    (curry random-normal :mu mu-1 :sigma sigma-1)
    frac-1
    (curry random-normal :mu mu-2 :sigma sigma-2)))

```

28.2 Systementwurf

28.2.1 Das Basissystem

Komponenten eines Simulationssystems

Grobentwurf

Beispiel: 135 (Ein Basissystem zur Simulation)

Akteure (actors): Repräsentationen der Domänenobjekte

Ereignisquellen (event sources): Objekte, die Ereignisse auslösen.

Ereignisse (events): Diskrete Marken auf dem Zeitstrahl, die den Zustand der Simulation ändern.

Kalender und Uhr

Protokollfunktionen, Interaktion

Dienstleistungen: Statistische Funktionen, Warteschlangen.

☞ **Anmerkung:** Wenn wir ein Szenarium für einen Simulationslauf aufbauen, sind eine Reihe von Operationen für alle Komponenten des Szenariums nötig, unabhängig von der speziellen Anwendung.

- Alle Komponenten müssen für den ersten Simulationslauf in einen Grundzustand versetzt und initialisiert werden können.
- Sie müssen für den Simulationslauf gestartet werden können.
- Alle Komponenten müssen für eine Wiederholung des Simulationslauf reinitialisiert werden können.
- Zu geeigneten Zeitpunkten müssen Zustandsabfragen möglich sein.

Es lohnt sich daher, eine abstrakte Klasse `sim-actor` zu definieren, die das Protokoll für diese Grundoperationen für alle Komponenten festlegt.

Die abstrakte Klasse `sim-actor-view` definiert das Protokoll für die Darstellung der Simulationsobjekte (actor-picture).

Die Klasse „sim-actor“:

Das Protokoll für Simulationsobjekte

Wenn wir ein Simulationsszenario aufbauen, sind die folgenden Operationen typisch für *alle* Klassen von Objekten:

- Initialisieren aller Objekte für den ersten Simulationslauf.
- Rücksetzen der Objekte für weitere Experimente.
- Starten des Prozesses.
- Zustandsabfrage.

Im Simulationsszenario muß es eine Datenstruktur geben, in der alle Komponenten der Szene inventarisiert sind, so daß es möglich ist, Initialisierungsoperationen und andere Operationen auf allen Komponenten auszuführen.

Die Klasse sim-actor

sim-actor	sim-actor-view
actorName	actorPicture
actorNum	actor-picture
sim-init!	...
sim-start	
sim-info	
broadcast	
...	

Methodenkombinationen

- Viele unserer Simulationsobjekte werden *Aggregate* sein, die wir durch Mehrfachvererbung erzeugen. Beispielsweise ist der Simulationskalender ein Aggregat aus einer Uhr und einer Tabelle mit Daten.
- Jede Komponente eines solchen Aggregats wird eine klassenspezifische Initialisierungs- oder Rücksetzmethode haben.
- Wenn wir das Aggregat initialisieren, wollen wir, daß die Initialisierungsmethoden für *alle* Komponenten angewendet werden, nicht nur diejenige Methode mit der höchsten Präzedenz, wie es die Voreinstellung wäre, denn dann würde ja nur ein Teil initialisiert.

Methodenkombination für die Initialisierung

```
(defgeneric* sim-init! ((obj sim-actor))
  :documentation
  "further_initializations_after_all_actors
   have_been_created"
  :combination generic-begin-combination)

(defgeneric* sim-start ((obj sim-actor))
  :documentation
  "Trigger_the_start-up_actions ,
   get_the_system_going"
  :combination generic-begin-combination)

(defgeneric* sim-info ((obj sim-actor))
  :documentation
  "request_state_information."
  :combination generic-begin-combination)
```

☞ Anmerkung: Ergänzung einer Initialisierungsmethode

Wenn wir mit `make-instance` ein Objekt erzeugen, so wird eine klasse-spezifische Methode `initialize` auf unser neues Objekt angewendet, die alle Attribute so initialisiert, wie wir bei der Definition der Klasse angegeben haben. Diese Methode brauchen brauchen wir nicht zu definieren, da CLOS sie automatisch für uns erzeugt.

Manchmal aber möchten wir zusätzliche Initialisierungen vornehmen, die sich nicht in der Klassendefinition angeben lassen. Dieses können wir über Ergänzungsmethoden erreichen.

Im folgenden Beispiel definieren wir eine Nachmethode zu `initialize`, die jedes neu erzeugte Objekt vom Typ `sim-actor` in eine Liste aller Simulationsobjekte einhängt. So bekommen wir leicht eine Liste aller erzeugten Objekte, und können diese gebündelt initialisieren, reinitialisieren usw. Diese Nachmethode wird automatisch im Anschluß an die primäre Methode für `initialize` ausgeführt. Wir werden noch viele andere Beispiele für Ergänzungsmethoden kennenlernen.

Ein Verzeichnis aller Akteure

```
(defclass* setOfActors ()
  (theActors
   :accessor theActors
   :writer set-theActors!)
```

```

:initvalue '()
:type <list> )
:autopred #t
:printer #t )

(define *all-actors* (make setOfActors))

(defgeneric* add-actor!
  ((a sim-actor) (s setOfActors))
  :documentation
  "add_an_actor_to_the_set_of_all_actors")

```

Ergänzung einer Initialisierungsmethode

Beispiel: 136 (Nachmethode für initialize)

Vermerke beim Erzeugen eines Simulationsobjektes eine Referenz auf das Objekt in einer Liste:

```

(defmethod initialize :after
  ; maintaining the list of actors;
  ; applied after the main initialization
  ((obj sim-actor) args)
  (add-actor! obj *all-actors*))

```

- Das zweite Argument (`args`) faßt alle Initialisierungsargumente von `make` zusammen.

☞ Anmerkung: Wir haben hier `initialize` durch eine Nachmethode ergänzt. Wenn die Primärmethode `initialize` spezialisiert wird, muß anschließend unbedingt ein super call erfolgen, damit die eigentliche Initialisierung durchgeführt wird.

```

(if (next-method?)
  (call-next-method obj args))

```

Wir hätten dieselbe Initialisierung auch erreichen können, indem wir einen eigenen Konstruktor für die Klasse `sim-actor` definiert hätten, aber da hätte den Nachteil, daß der Konstruktor eine Funktion und keine Methode ist. Er würde nicht vererbt werden.

Der Vorteil einer Nachmethode für initialize: Wir können sicher sein, daß unsere zusätzliche Initialisierung auch für alle spezialisierten Klassen durchgeführt wird.

Ablaufschema der Simulation:

- Ein Terminkalender führt Buch über die anstehenden Ereignisse.

dispatch: Bestimme das nächste anstehende Ereignis.

Setze die Uhrzeit auf den entsprechenden Termin.

handle: Simuliere das Ereignis.

schedule: Trage die durch ein Ereignis ausgelösten Folgeereignisse im Terminkalender ein.

- Wiederhole, bis die Simulationszeit abgelaufen ist.

Objekte zur Ablaufkontrolle

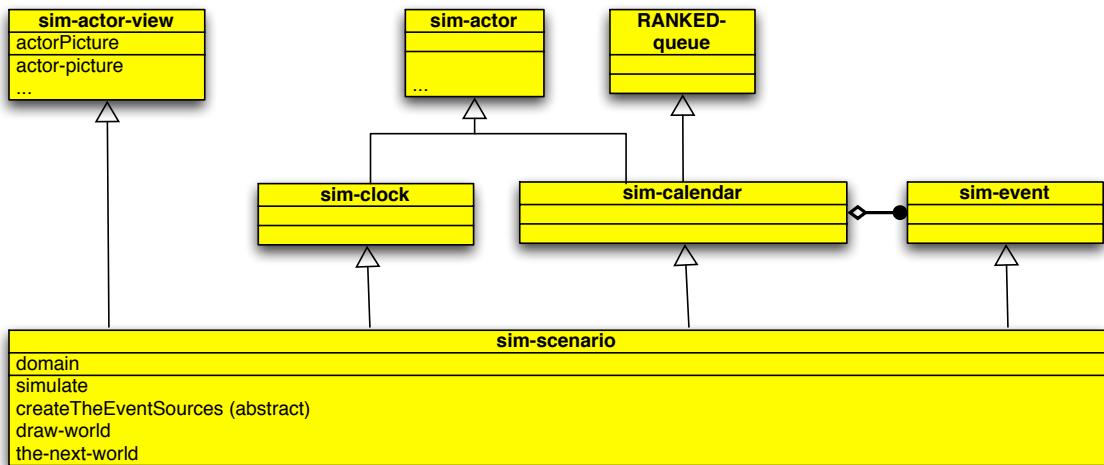
Terminkalender (sim–calendar): Enthält eine nach Uhrzeit geordnete Schlange (RANKED–queue) von anstehenden Ereignissen (events) sowie eine Uhr (sim–clock).

Uhr (sim–clock): Setzen und Ablesen der Uhrzeit.

Ereignisse (sim–event): Ereignisse bestehen aus einem Zeitpunkt und einer Beschreibung für die Art des Ereignisses. Sie werden vorausgeplant (schedule) und abgearbeitet (handle).

Szenarium (sim–scenario): Anstoßen des Simulationslaufs, Inspektion der Komponenten, Schnappschüsse usw.

Das Simulationsszenario: Ein Aggregat



 **Anmerkung:** Dieser Entwurf macht keine Annahmen darüber, welcher Art die Ereignisse sind. Im Anwendungspaket muß für jede Klasse von Ereignissen eine Methode handle definiert werden, die die entsprechenden Simulationsschritte ausführt.

Die Simulationsuhr (sim-clock) ist eine Unterklasse von sim-actor, so daß die entsprechenden Operationen sim-info usw. geerbt werden.

Der Kalender wurde als Aggregat modelliert und erbt durch Mehrfachvererbung sowohl von sim-actor als auch von ranked-queue.

Alle Komponenten eines speziellen Simulationslaufs bilden zusammen als Aggregat ein Simulationsuniversum (sim-scenario). Aggregation gibt es leider nicht in CLOS. Deshalb wurde hier der Weg gewählt, das Universum durch Mehrfachvererbung sowohl von sim-clock als auch von sim-calendar erben zu lassen. Das Universum erbt auch von der Klasse sim-event, da es selbst als „big-bang“ das start-Ereignis der Simulation darstellt.

Die Variable *current-universe* zeigt auf das aktuelle Universum. In diesem Objekt erfolgt die Buchführung über die Parameter für den speziellen Simulationslauf. Prinzipiell sollte es möglich sein, mehrere Paralleluniversen zu betreiben, aber in dieser speziellen Implementation darf es jeweils nur ein Universum geben. Das liegt daran, daß die Parameter der Simulation als Klassenattribute definiert sind und so für alle Universen nur einmal existieren.

Die Simulationsuhr

```
(defclass* sim-clock (sim-actor)
  (sim-time
    :reader read-sim-clock
    :writer set-sim-clock!
    :initvalue 0
    :type <number>
    :documentation
      "The_simulation_time")
  :autopred #t
  :printer #t
  :documentation "The_simulation_clock"
)
```

Der Kalender

Der Kalender

Ereignisse werden im Kalender nach Uhrzeit geordnet eingetragen.

- Der Kalender hat die Klasse *ranked-queue* und benötigt eine Methode *rankp* für Objekte der Klasse *sim-event*.
- Wir überladen die entsprechende Initialisierung der Klasse *ranked-queue*.

```
(defclass* sim-calendar
  (RANKED-QUEUE sim-actor)
  (rankp
    :initvalue time-due
    :documentation
      "the_rank_of_an_event
       is_its_time_to_occur.))
```

Die Ereignisse: sim-event

Eine mixin-Klasse für Akteure, die vom Kalender geweckt werden wollen.

```
(defclass* sim-event ()
  (scheduled-time
    :reader time-due
    :writer set-time-due!
    :initvalue 0
    :initarg :time-due
```

```
:type <number>) )

(defgeneric* handle ((event sim-event))
  :documentation
  "A_handler_for_simulation_events .")
```

Spezialisierte Ereignisse zur Ablaufkontrolle

```
(defclass* sim-deadlock (sim-event)
  :documentation
  "The_calendar_hasemptied_out_prematurely:
  Simulation_stalled"
  )
(defclass* sim-Quit (sim-event)
  :documentation
  "Stop_the_simulation"
  )
(defclass* sim-Snapshot (sim-event)
  :documentation
  "Display_the_state_of_the_simulation"
  )
```

Implementation des Kalenders: Scheduler

Der scheduler:

Trage ein Ereignis (Weckaufrag) im Kalender ein.

```
(defmethod schedule
  ((event sim-event)
   time-due)
  (set-time-due! event time-due)
  (enqueue! *current-calendar* event)
  *current-calendar*)
```

Implementation des Kalenders: Dispatcher

Implementation des Kalenders: Der Dispatcher

Lasse das nächste anstehende Ereignis eintreten:

- Entferne den Kopf der Ereignisschlange aus dem Kalender.
- Setze die Uhrzeit auf die Zeit des aktuellen Ereignisses.
- Führe das Ereignis aus.

```
(defmethod dispatch ((calendar sim-calendar))
  (if (empty-queue? calendar)
      (schedule (make sim-deadlock) (now))
      (let ((nextEvent (dequeue! calendar)))
        (set-clock! (time-due nextEvent))
        (handle nextEvent)
        calendar)))
```

Initialisierung des Kalenders

```
(define *current-calendar* #f)

(defmethod
  initialize :after
  ((newCalendar sim-calendar) initargs)
  (setf! *current-calendar*
    newCalendar) )
```

☞ Anmerkung: setf!: setf! (set form) ist ein imperatives swindle-Sprach-element, das von Common Lisp übernommen wurde. Es ist ein universeller Modifikator. Mit setf! lassen sich die Werte in Datenstrukturen (places) ändern. Wir können können Ausdrücken, die den Wert einer Teilstruktur liefern, einen neuen Wert zuweisen und ändern so den Wert der Struktur an dieser Stelle. Beispiel:

```
(require swindle/setf)
> (define xs '(1 2 3))
> (setf! (car xs) 4)
> xs → (4 2 3)
> (defclass Jackett ()
  (groesse :initvalue 40 :accessor gr)
  :printer #t )
> (define j (make Jackett))
> j → #<Jackett: groesse=40>
> (setf! (gr j) 50)
> j → #<Jackett: groesse=50>
```

☞ Anmerkung: Der Simulationszyklus: simulate

Die wichtigste Operation auf Objekten der Klasse `sim-scenario` ist die `simulate`-Operation. Die Simulation wird entweder für dt Zeiteinheiten durchgeführt oder bis ein `sim-break`-Ereignis auftritt. Während der Simulation wird mit der `dispatch`-Operation jeweils das nächste anstehende Ereignis im Simulationskalender `universe` gelesen und die Operationen werden von `handle` durchgeführt, um es eintreten zu lassen.

Die Schnittstelle für Anwendungimplemen-tationen ist die abstrakte Funktion `createtheEventSources`. Diese sollte anwendungsspezifische `sim-event-sources` erzeugen.

Das Universum

- Das Universum ist der *controller* der Simulation.
- Es ist ein Aggregat und dient sowohl als Kalender als auch als Uhr und als Startereignis und erbt von `sim-actor`.

```
(defclass* sim-scenario
  (sim-clock sim-calendar sim-event)
  (actorPicture
    :initvalue 
  )
  (domain
    :reader scene-description
    :initvalue "unknown-universe"
    :initarg :scene-description
    :allocation :class
    :type <string>))

```

Aufruf des Simulators

```
(defgeneric* simulate
  ((universe sim-scenario)
   &key (canvas-w 500) (canvas-h 500)
   (tick 1) (dt 100))
  :documentation
  "Simulate_the_scenario_for_an_interval
  of_dt_time_units"
  ; tick: clock ticks in seconds,
  ; dt: simulation time in seconds
  )
```

Weitere Universum-Operationen

Für den Simulationsablauf wird der Rahmen aus dem DrRacket-Modul `world.ss` verwendet.

```

(defgeneric* createTheEventSources
  ((universe sim-scenario))) ; abstrakt

(defgeneric* drawWorld
  ((universe sim-scenario)))
; grafische Darstellung der Welt

(defgeneric* theNextworld
  ((universe sim-scenario)))
; der nächste Simulationsschritt

(defgeneric* snap-shot
  ((universe sim-scenario)))
; Momentaufnahme

```

Initialisierung des Szenarios

- Für den Start der Simulation müssen in world.ss die action handler angemeldet werden.
 - (on-key-event handle-key)
 - (on-tick-event theNextworld)
 - (on-redraw drawWorld)
 - (stop-when last-world?)
- Die Methode „theNextworld“ ruft den *dispatcher* auf und gibt das geänderte Universum als neue Welt zurück.

```

(defmethod sim-start ((universe sim-scenario))
  (on-key-event handle-key)
  (on-tick-event theNextworld)
  (on-redraw drawWorld)
  (stop-when last-world?)
  #t )

```

Ende der Simulation?

```

(defmethod last-world?
  ((universe sim-scenario))
  (if (sim-Quit?
    (peak-next-event *current-universe*)
    #t #f)))

```

Die event handler

```
(defmethod theNextworld
  ((universe sim-scenario))
  (dispatch universe)
  universe)

(defmethod drawWorld ((universe sim-scenario))
  "draw_the_world_onto_the_canvas,
  ↵ return_a_scene_object"
  (sim-info universe)
  (put-pinhole (actor-picture universe) 0 0)
  )

(defmethod handle ((event sim-Quit))
  (set-event-message! event
    "end_of_time,_game_over")
  (display "end_of_time,_game_over"))

(defmethod simulate
  ((universe sim-scenario)
   &key (canvas-w 500) (canvas-h 500)
   (tick 1) (dt 100))
  "Simulate_the_scenario_for_dt_time_units"
  ; tick: clock ticks in seconds,
  ; dt: simulation time in seconds
  (createTheEventSources ;abstract
   *current-universe*)
  (sim-init! *current-universe*)
  (schedule universe (now))
  (schedule (make sim-Quit) dt))

(defmethod big-bang
  (canvas-w canvas-h
   tick
   *current-universe*)
  (sim-start *current-universe*))
```

Ereignisquellen

Einige Komponenten lösen regelmäßig Ereignisse aus. Hier wollen wir angeben können,

- wie häufig die Ereignisse sind (Ereignisrate)
- und zählen, wieviele Ereignisse aufgetreten sind.

Die wichtigste Operation für solche Ereignisquellen ist das Planen des nächsten Ereignisses entsprechend den statistischen Eigenschaften der Quelle:

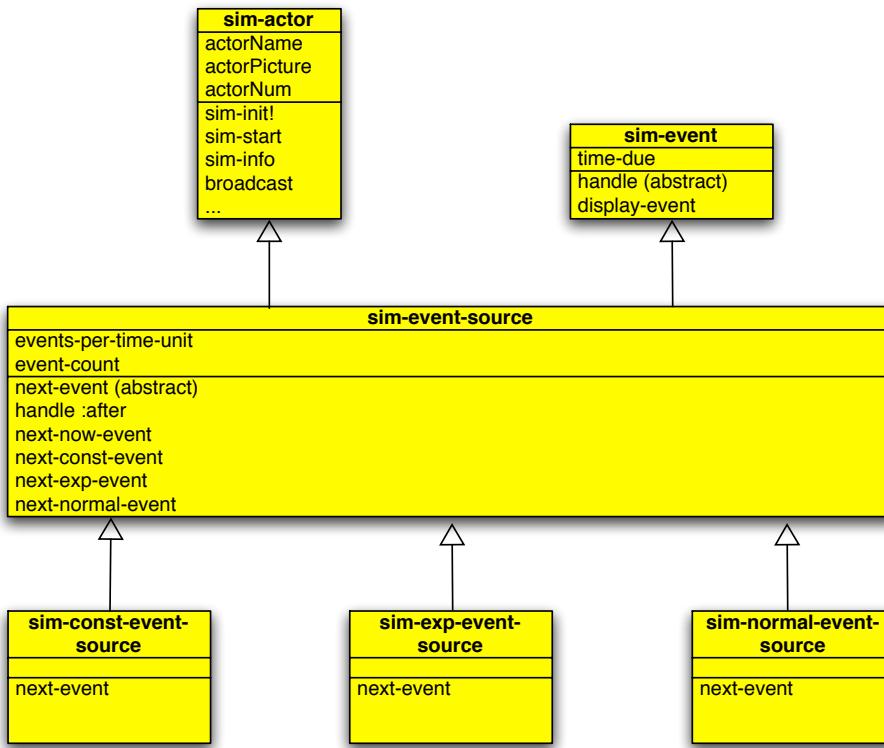
next-now-event: Das nächste Ereignis findet sofort statt.

next-exp-event: Das nächste Ereignis findet nach einer exponential verteilten Wartezeit statt.

next-normal-event: Das nächste Ereignis findet nach einer normalverteilten Wartezeit statt.

next-const-event: Das nächste Ereignis findet nach einer konstanten Wartezeit statt.

Die Klasse sim-event-source



Akteure als Ereignisse: Weckaufträge

Die Wecker-Metapher:

Wie beschreiben wir die Ereignisse am einfachsten? Die von einem Ereignis betroffenen Akteure wissen am besten, wie auf das Ereignis zu reagieren ist. Deshalb werden sie selbst zu Ereignissen gemacht.

- Die Ereignisquellen sind Unterklassen sowohl von `sim-actor` als auch `sim-event`.
- Ereignisquellen können damit direkt als Ereignis im Kalender vermerkt werden.
- Der Kalender wird als Liste von *Weckaufträgen* interpretiert.
- Jede Ereignisquelle hat ihre klassenspezifische `handle`-Methode, die beschreibt, was beim Wecken zu geschehen hat.

⌚ Anmerkung: Repräsentation von Ereignissen

Damit eine einzige Operation handle die Operationen im Simulationsuniversum ausführen kann, die ein Ereignis ausmachen, muß jeder Typ von Ereignis eine eigene Klasse haben. So werden automatisch die richtigen handle-Methoden ausgewählt.

Die handle-Methode benötigt aber noch weitere Informationen:

- Welche Akteure sind betroffen?
- Welche weiteren Parameter sind zu berücksichtigen?

Je nach Klasse der Akteure ergibt sich vermutlich eine andere Signatur für die handle-Methode. Wenn wir davon ausgehen können, daß

- jedes Ereignis primär einen Akteur betrifft
- und dieses Akteur-Objekt (sim-actor) alle Parameter zur Simulation des Ereignisses umfaßt,

dann ist es am einfachsten, die Akteure selbst zum Ereignis zu machen.

Die Lösung: Die Klasse sim-event-source wird zur Unterklasse von sim-actor und sim-event. Wir können dann die Ereignisquellen direkt in den Simulationskalender eintragen, da sie ja von der Klasse sim-event sind. Dieses entspricht der folgenden Metapher:

- Der Kalender ist eine Liste von Weckaufträgen.
- Für jeden Akteur wird im Kalender vermerkt, wann er etwas zu tun hat oder wann etwas mit ihm geschehen wird.
- Wenn dieser Zeitpunkt gekommen ist, dann wird der Akteur geweckt.

Die spezifische handle-Methode für die Klasse des Akteurs beschreibt, was beim Wecken zu geschehen hat.

```
(defclass* sim-event-source
  (sim-actor sim-event)
  (events-per-time-unit
    :reader event-rate
    :type <number>
    :initvalue 1
    :initarg :rate
    :documentation
    "The_mean_value_of_events_per_time_unit")
  (event-count
    :reader event-count
```

```
:writer set-event-count!
:initvalue 0
:documentation
"The_number_of_events_so_far")
```

Spezialisierte Ereignisquellen

```
; Ereignisse im konstanten Abstand
(defclass* sim-const-event-source
  (sim-event-source))

; Gleichverteilte Ereignisse
(defclass* sim-exp-event-source
  (sim-event-source))

; Normalverteilte Ereignisse
(defclass* sim-normal-event-source
  (sim-event-source)
  (variance
   :reader event-sig
   :type <number>
   :initvalue 0.3
   :initarg :sigma))
```

Das Protokoll der Ereignisquellen

```
(defgeneric* next-event
  ((actor sim-event-source)))
(defgeneric* next-now-event
  ((actor sim-event-source)))
(defgeneric* next-exp-event
  ((actor sim-event-source)) )
(defgeneric* next-normal-event
  ((actor sim-event-source)
   &key (sigma 1)))
(defgeneric* next-const-event
  ((actor sim-event-source)))
)
```

Implementation der Ereignisquellen

- Rufe den klassenspezifischen event handler (*handle*) auf.
- Bestimme den Zeitpunkt des nächsten Ereignisses.
- Trage es in den Simulationskalender ein (*schedule*).
- Das Planen des nächsten Ereignisses (*next-event*) geschieht in einer *Nachmethode* zu „*handle*“.

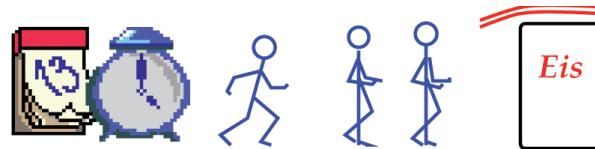
So kann die Primärmethode „*handle*“ anwendungsspezifisch spezialisiert werden, und es ist sichergestellt, daß *next-event* auf jeden Fall aufgerufen wird.

```
(defmethod next-normal-event
  ((actor sim-event-source) &key (sigma 1))
  ;Schedule the actor for the next event.
  (inc! (event-count actor))
  (schedule actor
    (add-time (now)
      (abs (random-normal
        :mu (/ 1 (event-rate actor))
        :sigma sigma)))))

(defmethod handle :after
  ((actor sim-event-source))
  ;After handling an event source
  ;create the next event"
  (next-event actor))
```

28.2.2 Das Bediensystem

Anwendungssystem: Eine Bedienstation



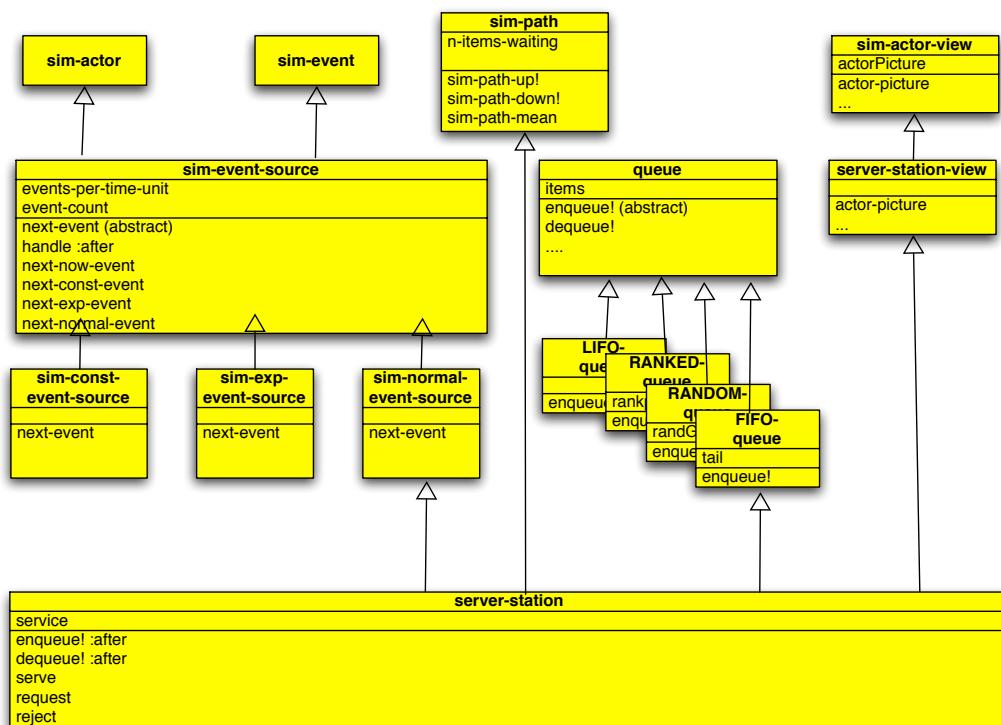
Benötigt werden:

Eine Bedieneinheit: Löst Bedienereignisse aus (sim-event-source).

Warteschlange von Kunden: FIFO-queues.

Protokoll: (sim-path:) Mittlere Wartezeit, Schlangenlänge usw.

Die Klasse server-station



☞ Anmerkung: Erweiterung von Objektbibliotheken durch Vererbung und Ergänzungsmethoden Unsere Klassen FIFO-Queues und sim-event-source erfüllen schon fast alle Anforderungen, die wir an eine Bedienstation haben. Es fehlen nur noch die Protokollierung von Ereignissen sowie eine einfache Statistik über die Länge der Warteschlange usw.

Der Vererbungsmechanismus erlaubt es uns, auf den bestehenden Klassen aufzubauen, ohne daß wir diese ändern müssen. Wir definieren eine Klasse sim-path, die die zusätzlichen Operationen und Attribute für die Buchführung bereitstellt (Reiser and Wirth, 1994) und machen durch *Mehrfachvererbung* unsere neue Klasse server-station zur Unterklasse von allen drei Klassen.

Ein Problem dabei ist, daß wir die Buchführung genau dann machen müssen, wenn ein Element in die Warteschlange eingehängt oder entfernt wird. Wir wollen aber die enqueue-, dequeue-Operationen nicht modifizieren und auch nicht neu schreiben müssen. Hier sind die Ergänzungsmethoden von CLOS sehr nützlich. Wir nutzen die primären enqueue-, dequeue-Methoden der Klasse FIFO-queues, aber erweitern sie durch *Nachmethoden*, die bei jedem enqueue-, dequeue die richtige Buchführung machen.

Eine Klasse für Bedienstationen

```
(defclass* server-station
  (sim-normal-event-source
   FIFO-queue
   sim-path
   server-station-view)
  (service
   :reader station-service
   :initvalue "Super_Market"
   :initarg :station-service
   :type <string>))
```

Protokoll der Bedienstationen

```
(defgeneric* serve ; bediene den Kunden
  ((s server-station)(c customer)))

(defgeneric* request; Anstellen am Schalter
  ((s server-station)(c customer)))

(defgeneric* reject; Verweigern der Bedienung
  ((server-station)(c customer)))
```

Statistik über Warteschlangen

```
(defclass* sim-path ()  
  (n-items-waiting :accessor items-w  
    :initvalue 0 :type <integer>)  
  (accumulated-waiting-time  
    :accessor wt :initvalue 0 :type <number>  
    :documentation  
    "Accumulated_waiting_time_since")  
  (time-of-start  
    :accessor ts :initvalue 0  
    :type <number>)  
  (time-of-last-update :type <number>  
    :accessor tlu :initvalue 0))
```

Sim-path Operationen

```
(defgeneric* sim-path-up! ((sp sim-path))  
  :documentation  
  "Update_the_statistics_for_a_new_arrival"  
  :combination generic-begin-combination)  
  
(defgeneric* sim-path-down! ((sp sim-path))  
  :documentation  
  "Update_the_statistics_for_a_departure"  
  :combination generic-begin-combination)  
  
(defgeneric* path-info ((sp sim-path))  
  :documentation "request_state_inform.")  
  
(defgeneric* sim-path-mean ((sp sim-path))  
  :documentation "The_average_waiting_time")
```

Vormethoden für enqueue! und dequeue!

```
(defmethod enqueue! :before  
  ((qu server-station) item)  
  ; Update the path statistics  
  ; before entering the queue  
  (sim-path-up! qu))  
  
(defmethod dequeue! :before  
  ((qu server-station))  
  ; Update the path statistics
```

```
; before leaving the queue  
(sim-path-down! qu))
```

Bedieneignisse der Serverstation

Ein Ereignis startet,

1. wenn ein Kunde an einen leeren Schalter tritt,
2. oder wenn die Bedienung des vorherigen Kunden abgeschlossen ist.

Implementation der Klasse server-station

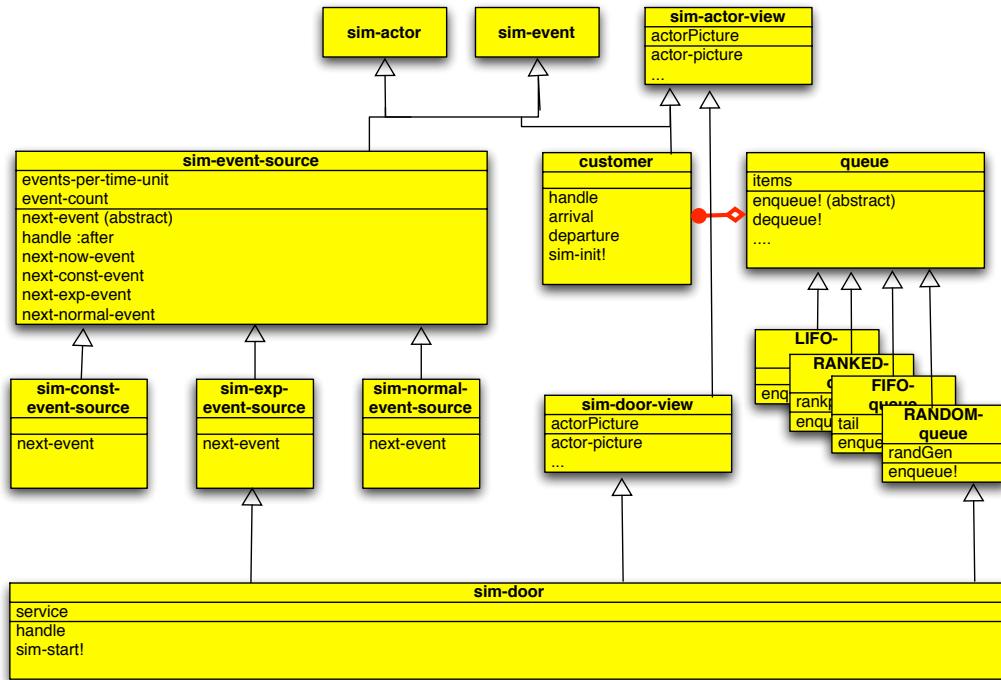
Ein Kunde stellt sich in der Warteschlange an:

```
(defmethod request  
  ((s server-station)(c customer))  
  (let* ((idle (empty-queue? s)))  
    (enqueue! s c)  
    (when idle  
      (serve s c)  
      ; serve immediately,  
      ; if the queue is empty  
      (next-event s))  
      ; schedule the departure event  
  ))
```

Die Kunden

- Die Kunden (customer) betreten das Szenario zu zufälligen, gleichverteilten Zeitpunkten und gehen zu einer zufälligen Bedienstation.
- Der event handler für ein Kunden-Ereignis führt den request an der Bedienstation durch.
- Eine Ereignisquelle - sim-door - sendet ständig neue Kunden auf die Bühne.
- Die Kunden, die gerade untätig sind, warten hinter den Kulissen in einer zufällig geordneten Warteschlange auf ihren Auftritt.

Kunden und Simulationstür



☞ Anmerkung: Das Basissystem bietet bisher zur Erzeugung der aktiven Simulationsobjekte nur die generische Funktion „createTheEventSources“. Die Art der Objekte war gleichgültig.

Wir werden diese Methode durch drei anwendungsspezifische generische Methoden ersetzen:

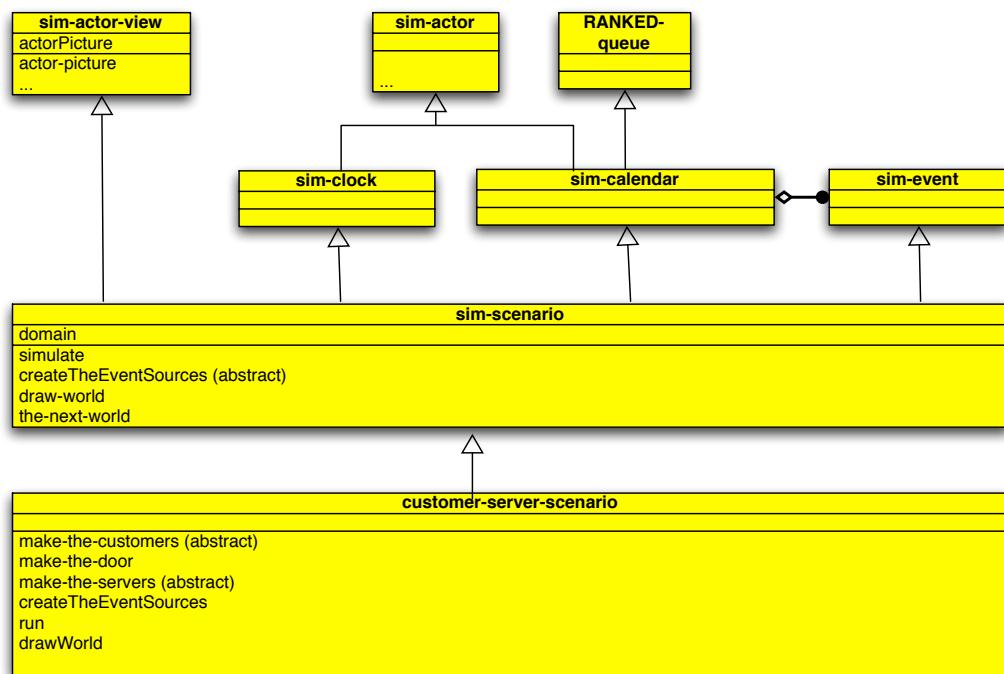
`make-the-customers`, `make-the-door`, `make-the-servers`.

Diese generischen Funktionen sind jeweils für konkrete Anwendungsszenarien zu implementieren.

Für `make-door` gibt es eine default-Implementation, die gleichverteilte Kundenereignisse erzeugt, und zwar mit einer doppelt so hohen Ereignisrate, wie die schnellste Bedienstation, so daß die Warteschlangen sich schnell füllen.

Außerdem führen wir noch eine einhüllende Methode ein, um `simulate` aufzurufen: `run`.

Die Klasse `customer-server-scenario`



Controller: Das Customer-Server-Szenario

Die Spezialisierung des controllers für Bediensysteme:

```
(defclass* customer-server-scenario
  (sim-scenario))
```

```
(defgeneric* run
  ((world customer-server-scenario)
   &key (tick 1) (dt 100))
  :documentation "a_default_simulation_run"
  )
```

Protokoll für das Erzeugen der Objekte

Diese generischen Funktionen implementieren die abstrakte Funktion „createTheEventSources“.

```
; Erzeuge die Kunden-Objekte
(defgeneric* make-the-customers
  ((world customer-server-scenario)))

; Erzeuge die Eingangstür: Quelle für Kunden
(defgeneric* make-the-door
  ((world customer-server-scenario)))

; Erzeuge die Bedienstationen
(defgeneric* make-the-servers
  ((world customer-server-scenario)))
```

Initialisierung und Start der Simulation

- *sim-init!* für einen *Kunden* reiht diesen Kunden in der Warteschlange ein.
- *sim-init!* für die *Simulationstür* führt für alle Kundenobjekte *sim-init!* durch (*broadcast*).
- *sim-start!* für das *Simulationsuniversum* trägt ein Türereignis als erstes Ereignis in den Simulationskalender ein.

☞ Anmerkung: Das Dynamische Modell Wie lebendig ist unser Simulationssystem? An dieser Stelle müßten wir jetzt die Zustände des Systems und die Zustandsübergänge modellieren und tief in die imperativen, zustandsorientierte Programmierung einsteigen. Das ist aber nicht das Thema dieser Vorlesung. Wir wollen uns in dieser Vorlesung auf die funktionale Programmierung konzentrieren, und in diesem Kapitel speziell auf den Unterschied zwischen generischen Funktionen und der Nachrichtenmetapher .

Informell können wir festhalten, daß unser Simulationssystem lebendig bleibt, solange es mindestens einen Kunden und eine Bedienstation mit endlicher Bedienzeit gibt.

- Die Simulationstür erzeugt unbedingt zu jedem Türereignis ein Folgeereignis.
- Die Bedienstationen sind lebendig, solange Kunden davor warten und können jederzeit durch request-Ereignisse geweckt werden.
- Die Bedienstationen füllen in endlicher Zeit den Vorrat an Akteuren, die für einen Auftritt zur Verfügung stehen, wieder auf, auch wenn zwischenzeitlich alle verfügbaren Kunden vor den Bedienstationen warten sollten.

28.3 Eine Anwendung

28.3.1 Mensa-Szenario

Anwendung 1: Ein Mensa-Szenario

Beispiel: 138 (Mensa-Szenario 1:)

Als erstes Anwendungsbeispiel werden wir die Essensausgabe in einer Mensa simulieren:

Die Akteure:

Bedienstationen: Essensausgaben (Essen 1, Essen 2 usw.).

Wir abstrahieren zunächst davon, daß es auch Kassen geben sollte.

Kunden: Studentinnen und Studenten mit Namen

Eingangstür: Ereignisquelle für gleichverteilte Ereignisse. Wir abstrahieren davon, daß Studenten nur zwischen den Vorlesungen in der Mensa sitzen sollten.

Um die konkreten Anwendungsobjekte zu erzeugen, spezialisieren wir

- die Klasse (*customer-server-scenario*)
- sowie die generischen Funktionen
 - *make-the-servers* und
 - *make-the-customers*

Die Essensausgabe

```
(defclass* mensa-scenario
  (customer-server-scenario))
(define (makeStation nam rte serv)
  (make server-station
    :actorName nam
    :actorPic 
    :rate rte
    :station-service serv))
(defmethod make-the-servers
  ((mensa mensa-scenario))
  (makeStation "Essen-1" 2 "Eintopf")
  (makeStation "Essen-2" 2.3 "Pizza")
  (makeStation "Essen-3" 2.3 "Currywurst")
  (makeStation "Essen-4" 0.5 "Corn-Dogs"))
```

Die Studentinnen und Studenten

```
(defmethod make-the-customers
  ((mensa mensa-scenario))
  (let ((someStudents
    '("Harry" "Susie" "Sally" "Paula" "Anja"
      "Ernie" "Bert" "Hermione" "Arnold"
      "Paris" "Maja" "Heidi" "Siegfried"
      "Kunigunde" "Erwin" "Otto" "Hilde"
      "Christoph" "Wolfgang" "Christiane"
      "Ronald" "Ingbert" "Karin" "Helmut"
      "Sascha" "Dominique" "Donald" "Daisy"
      "Prudence" "Ansgar" "Ottolie" "Anton")))
    (map (lambda (c)
      (make customer :actorName c))
      someStudents)))
```

Starten der Simulation

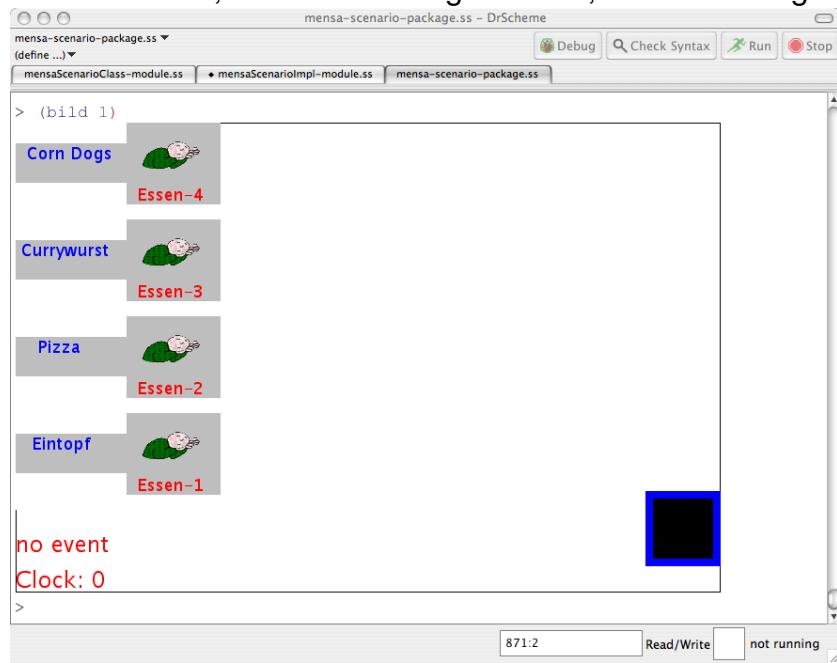
```
(define (mensaDemo1)
  (run (make mensa-scenario
    :actorName "StEllingen" )))
```

Laden des Simulationspaketes

```
(require se3-bib/sim/mensa-scenario-package)
```

(mensaDemo1)

Start der Simulation, alle Essensausgaben frei, die Bedienung schl鋑t.



Nach 10 Simulationsschritten: Zwei G鋘ste werden bedient, zwei G鋘ste warten.



28.3.2 Spezialisierung von erzeugten Objekten

Mensa-Szenario 2:

Spezialisierte Essensausgaben

Beispiel: 139 (Mensa-Szenario 2: Spezialisierte Essensausgaben und Gäste)

- Wir führen eine neue Kategorie von Mensagästen ein: Vegetarier.
- Nur eine Essensausgabe hat vegetarisches Essen, alle anderen Essensausgaben weisen die Vegetarier zurück.
- Um das neue Verhalten zu implementieren, muß nur die generische Funktion „serve“ spezialisiert werden.

Spezialisierte Bedienstationen

; Hier gibt es Essen mit Fleisch
(**defclass*** general–food–server
(server–station))

; Hier gibt es vegetarisches Essen
(**defclass*** vegetarian–food–server
(server–station))

; Mensagast, der kein Fleisch isst
(**defclass*** vegetarian (**customer**))

Spezialisierung der Serve-Operation

(**defmethod** rejecting?
 ((s general–food–server)(c vegetarian))
 #t); the default, serve all customers

(**defmethod** serve
 ((s general–food–server)(c vegetarian))
 (reject s c))

☞ Anmerkung: Spezialisierung von Methoden in Abhängigkeit von mehreren Parametern

Normalerweise verhält sich die `serve`-Operation für alle Mensagäste an allen Essensausgaben gleich. Es wird nur das Ende der Bedienzeit im Terminkalender vermerkt (`next-normal-event`).

Wenn aber von der Essensausgabe nur Fleischgerichte serviert werden und der Gast ein vegetarisches Essen verlangt, dann wird die Bedienung verweigert. Die Methode `serve` wurde in Abhängigkeit von allen beiden Parametern spezialisiert.

In CLOS können generische Funktionen im Hinblick auf beliebig viele Parameter spezialisiert und dynamisch zur Laufzeit ausgewählt werden. Nicht alle objektorientierten Sprachen lassen die Spezialisierung von mehr als einem Parameter zu.

☞ **Anmerkung:** Für das Mensa-Szenario 1 haben wir Klassen spezialisiert und dann die Objekte mit den passenden Klassen erzeugt. Bei unserem neuen Szenario ist die Situation grundsätzlich anders. Das Paket, von dem wir geerbt haben, hat leider die Objekte schon erzeugt, und die Klassen sind zum Teil nicht speziell genug.

Die Studentinnen und Studenten: Die Nichtvegetarier werden schon durch die Methode `make-the-customers` des Mensa-Szenario 1 mit der richtigen Klasse `customer` erzeugt. Daher reicht eine *Nachmethode*, um noch ein paar Vegetarier hinzuzufügen.

Die Essensausgabe: Die Essensausgaben haben alle die falsche Klasse. Daher verändern wir die Klasse der Essensausgaben in einer Nachmethode zu `make-the-servers` mittels `change-class!`.

`change-class!` verändert die Klasse eines Objektes, und bewahrt die Identität des Objektes. Das Verfahren:

- Zunächst wird eine Kopie des Objektes mit neuer Klasse erzeugt.
- Alle passenden *slots* werden kopiert.
- Dann wird erneut `initialize` mit den neuen Initialisierungsparametern durchgeführt.
- Das neue Objekt wird unter der Referenz des bisherigen Objektes in alle bisherigen Datenstrukturen eingebunden.

In unserem Beispiel wird daher die Nachmethode zu `initialize` durch das `change-class!` zweimal ausgeführt, und die Essensausgaben werden doppelt in die Listen aller Akteure eingetragen – deshalb der lästige Aufruf von `remove-duplicates!`

`change-class!` ist immer dann nützlich und sinnvoll, wenn wir eine bestehende Objektbibliothek durch Spezialisierung der Klassen erweitern. Wenn die Objekte im zu erweiternden Paket zunächst als Instanzen der allgemeineren Oberklasse erzeugt werden, können wir sie anschließend in unserer Erweiterung mit `change-class!` spezialisieren. Wir brauchen nicht den Code im erzeugenden Paket zu ändern.

Auch in die Gegenrichtung können wir `change-class!` verwenden und die Klasse eines Objekts zur Oberklasse generalisieren. Das ist besonders bei Mehrfachvererbung nützlich, wenn wir ein Objekt nur noch in einer Variante benötigen, beispielsweise ein Amphibienfahrzeug nur noch als Schiff und nicht mehr als Landfahrzeug.

Erzeugen der Studentinnen und Studenten

- Die von `mensa-scenario` geerbte Methode `make-the-customers` erzeugt die Fleischliebhaber.
- Eine *Nachmethode* fügt Vegetarier hinzu:

```
(defmethod make-the-customers :after
  ((mensa mensa-scenario2))
  (let ((someVegetarians '("Demeter"
  "Flora" "WurzelSepp" "Waldfee"
  "Rapunzel" "Rosemarie" "Tinkerbell"
  "Winnie" "Kräuterhexe" "Persephone"
  "MrClou" "Fleur" "Artemis" "Diana" "Hera"
  "Sonja" "Adonis" "Ganymed" "Donald"))))
```

```
(map (lambda (c)
  (make vegetarian :actorName c
    :actorPic  ))
  someVegetarians )))
```

Erzeugen der Essensausgaben

- Die Klasse der geerbten Essensausgabe-Objekte ist nicht speziell genug: *change-class!*

```
(defmethod veggie ((s server-station))
  (if (member
    (station-service s)
    '("Corn_Dogs" "Pizza"))
    (change-class!
      s vegetarian-food-server)))

(defmethod meat ((s server-station))
  (if (member
    (station-service s)
    '("Currywurst" "Eintopf"))
    (change-class!
      s general-food-server)))
```

Ändern der Klasse aller server-stations

Ein Rundruf an alle Objekte der Klasse *server-station*:

```
(defmethod make-the-servers :after
  ((mensa mensa-scenario2))
  (broadcast mensa veggie
    :sim-class server-station)
  (broadcast mensa meat
    :sim-class server-station)
  (remove-duplicates! *all-servers*))
```

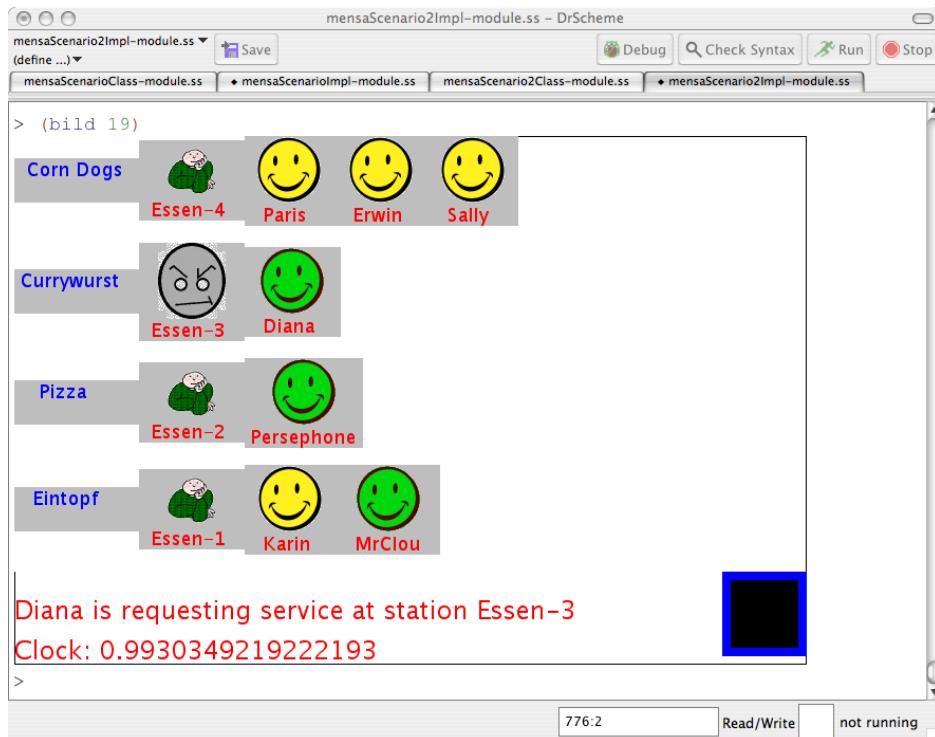
Ein Beispieldlauf

Language: **Swindle**.

> (**require** se3-bib/sim/mensa-scenario2-package)

> (**mensa2Demo**)

Nach 19 Simulationsschritten: Die Vegetarierin Persophene wird am Schalter mit vegetarischer Pizza bedient, aber Diana wird am Currywurstschalter zurückgewiesen.



☞ Anmerkung: Initialisierung von Objekten Wir haben am Beispiel dieses Simulationssystems eine Reihe von unterschiedlichen Verfahren zur Initialisierung der Objekte kennengelernt. Welches Verfahren jeweils anwendbar ist, hängt davon ab, wann alle benötigten Informationen zur Verfügung stehen. Für die Initialisierungsmethoden von CLOS (:initializer, initialize :after) gilt, daß sie vererbt werden und ohne einen super call zum übergeordneten Konstruktor durchgeführt werden.

Verfahren zur Initialisierung von Objekten

:initvalue, :initializer : Anwendbar, wenn alle Informationen schon bei der Definition der Klasse bekannt sind.

:initarg : Anwendbar, wenn alle Informationen zum Zeitpunkt der Erzeugung des Objektes bekannt sind.

initialize :after : Anwendbar, wenn bei der Initialisierung eine Referenz auf das Objekt existieren muß.

change-class! Notwendig, wenn bei der Erzeugung des Objektes die genaue Klasse noch nicht bekannt ist.

init-sim! : Selbstdefinierte Initialisierungsfunktionen: notwendig, wenn die Initialisierung anderer Objekte abgewartet werden muß.

29 Objektorientierte Verarbeitungsmodelle

29.1 Nachrichten vs. Generics

Objektorientierte Verarbeitungsmodelle



- 25 CLOS: Objekte und generische Funktionen
- 26 Entwurf eines ereignisorientierten Simulationssystems
- 27 Objektorientierte Verarbeitungsmodelle
 - Nachrichten vs. Generics
 - Delegation
 - Backtracking mittels Delegation

In den ersten objektorientierten Programmiersprachen (SIMULA, Smalltalk) wurde objektorientierte Programmierung gleichgesetzt mit dem Verarbeitungsmodell des *Nachrichtenausstauschs*. Die Komponenten des Programms waren aktive Objekte, von denen der Kontrollfluß des Programms ausging. Methoden wurden nicht auf Objekte angewendet, sondern die Objekte bekamen Aufforderungen — Nachrichten über gewünschte Operationen — und haben selbstständig die passenden Methoden Ausführung der Operationen ausgewählt (Nachrichtenaustausch, message passing). Die Syntax der Sprachen hat diese Sicht widergespiegelt. Auch in Java oder Oberon finden wir noch die Syntax, die zuerst das Objekt nennt und dann erst die gewünschte Operation.

Die Nachrichtenmetapher ist ein angemessenes Verarbeitungsmodell für Simulationsanwendungen, da so die Objekte wirklich als aktive Komponenten sichtbar werden. Es ist auch leicht, Operationen auf Gruppen von Objekten anzuwenden, per Rundruf oder *broadcast*. Diese Metapher hat aber einen entscheidenden Nachteil: Operationen, deren Methodenauswahl von mehreren Objekten abhängt, lassen sich so nicht formulieren, da nur ein Objekt als Nachrichtenempfänger anhand seiner Klasse über die Auswahl der Methode entscheidet. Eine Methode wie *serve*, die sowohl von der Klasse des Kunden als auch von der Klasse der Bedienstation abhängt, ließe sich so nicht realisieren.

**Die Nachrichtenmetapher: message passing Beispieleweise
in Java oder Smalltalk**

- Methoden gehören zu Objekten und sind der *einen Klasse* des Objektes zugeordnet.
- Methoden werden aktiviert, wenn ein Objekt eine Aufforderung zur Ausführung einer Operation erhält.
- Methoden werden wie Attribute vererbt.
- Die Syntax betont das Objekt, das die Nachricht erhält:

```
tell object message  
object.message(params)
```

Generische Funktionen versus Nachrichtenmetapher

Generische Funktionen

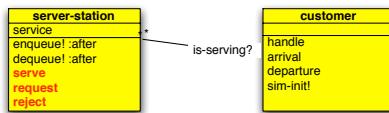
Generische Funktionen sind eine Generalisierung des Nachrichtenmodells:

- Im *Nachrichtenmodell* sind Methoden das Eigentum der Objekte.
- Im Modell der *generischen Funktionen* werden die Funktionen für Objekte spezialisiert.

Die generischen Funktionen sind aber autonom und nicht einer bestimmten Klasse zugeordnet.

- Die dynamische Auswahl einer Methode kann von *mehreren* Objekten abhängen.

Generische Funktionen und UML



- UML wurde für die Nachrichtenmetapher entworfen: Die grafische Repräsentation von Operationen ordnet diese zwangsweise einer *einzigsten Klasse* zu.
- UML ist daher zur Repräsentation generischer Funktionen nur *bedingt geeignet*:
Die Operationen serve, request, reject usw. im Simulationssystem können von UML nicht korrekt wiedergegeben werden.

Klassen und Modularisierung

Nachrichtenmodell:

Für das Nachrichtenmodell ist typisch, daß eine Klasse auch gleichzeitig zur Modularisierung dient:

Eine Klasse ist gleichzeitig

- eine Übersetzungseinheit,
- ein gekapselter Namensraum für definierte Namen,
- eine Datenkapsel für die Attribute der Objekte.

Generische Funktionen:

Beim Modell der generischen Funktionen werden diese Aufgaben klar getrennt:

- Eine Klasse dient allein der Vererbung und ist eine Datenkapsel für Attribute der Objekte.
- Modularisierung wird über Module oder Pakete erreicht.

⌚ Anmerkung: Nachrichtenaustausch in CLOS

CLOS verwendet das Modell der generischen Funktionen, wie wir gesehen haben; es ist aber leicht, in CLOS das Nachrichtenmodell zu implementieren. An den folgenden Beispielen werden Sie sehen, daß sich das Nachrichtenmodell auf „syntaktischen Zucker“ reduzieren läßt. Wir werden eine Operation `tell` implementieren, die einem Objekt die Nachricht gibt, eine Operation auszuführen, sowie eine Funktion `broadcast`, die per Rundruf eine Nachricht an alle Objekte einer bestimmten Klasse verbreitet.

Diese Operationen verwenden wir, um einige Operationen zu implementieren, die von allen Simulationsobjekten ausgeführt werden müssen, wie Initialisieren oder die Suche nach Freiwilligen.

— Nachrichten: `tell` — Beauftrage einen Akteur, die Operation „message“ auszuführen:

```
(defmethod tell
  ((a sim-actor) (message <function>))
  (message a))

> (tell (make sim-actor) class-of)
→ #<class:sim-actor>
```

Rundruf an alle: `broadcast`

Beispiel: 140 (Nachricht an alle Objekte: `broadcast`)

```
(defmethod broadcast
  ((actor sim-actor)
   (message <function>)
   &key (sim-class sim-actor))
  (dolist
    (actr (reverse (theActors *all-actors*)))
    (when (instance-of? actr sim-class)
      (tell actr message))))
> (make sim-actor) → #<sim-actor: .... >
> (broadcast (make sim-actor) sim-info )
1: anonymous #<class:sim-actor>
2: anonymous #<class:sim-actor>
>
```

Suche ein Objekt: `broadcast-some`

Suche ein Objekt, das ein bestimmtes Prädikat erfüllt.

(defmethod broadcast-some

```
((actor sim-actor) (message <function>)
 &optional (sim-class sim-actor))
(some
 (lambda (actr)
  (if (and
    (equal? (class-of actr) sim-class)
    (tell actr message))
   actr #f))
 (theActors *all-actors)))
>(broadcast-some (make sim-actor)
  (compose (curry = 3)actor-num))
#<sim-actor: actorName="anonymous" actorNum=3>
>
```

29.2 Delegation

Funktionale Kontrollabstraktion

Zur funktionalen Kontrollabstraktion wird ein Ablaufschema definiert, das mittels Funktionen parametrisiert werden kann.

Funktionen höherer Ordnung: In *funktionalen Programmiersprachen* können wir *funktionale Abschlüsse* an Funktionen höherer Ordnung übergeben (map, general-backtracking, ...)

Delegation: In *objektorientierten Sprachen* parametrisieren wir den Ablauf durch Objekte, an die wir die Auswahl der Operationen delegieren.

- Sie können die meisten Funktionen höherer Ordnung in Java mittels Delegation realisieren – am besten in generischen Klassen.

29.3 Backtracking mittels Delegation

Backtracking mittels Delegation

Beispiel: 141 (Backtracking mittels Delegation)

Wir definieren den Zustandsraum, in dem wir nach einer Lösung suchen wollen, als Klasse „state“.

- Die zustandsabhängigen Operationen
 - Folgezustände errechnen,
 - Test auf Zulässigkeit,
 - Test auf Erfolg

werden als generische Funktionen definiert.

- Die generische Funktion *general-backtracking* erhält als einzigen Parameter, als „delegate“, den Ausgangszustand der Suche.
- Konkrete Backtracking-Probleme spezialisieren die Klasse „state“ und implementieren das Protokoll der generischen Funktionen.

Die Klasse „state“

```
(defclass* state ()  
  ; a state of the state space search  
  (parentState  
   ; the parent state of this state  
   :accessor theParentState  
   :initarg :state-parent  
   :initvalue ???)  
  :autopred #t  
  :printer #t  
  :documentation  
  "the_top_class_of_states"  
)
```

Die generischen Funktionen

```
(defgeneric* general-backtracking-oo
  ((initialState state)))
; Erzeugen der Nachfolgerzustaende
(defgeneric* gen-states ((st state)))
; Zustand zulaessig?
(defgeneric* is-legal? ((st state)))
; Ziel gefunden?
(defgeneric* is-final? ((st state)))
```

Das Backtracking-Schema mit Delegation

```
(defmethod general-backtracking-oo
  ((initialState state))
  ;; find all solutions; may cause inf. loops
  (letrec
    ((try
      (lambda (st)
        (cond
          ((not (is-legal? state)) '())
          ((is-final? st)
           (cons st ; further solutions
                 (apply append
                       (map try (gen-states st))))))
          (else
            (apply append
                   (map try (gen-states st)))))))
     (try initialState)))
```

☞ Anmerkung: Vorteil der generischen Funktionen:

- Die Rahmenfunktion „general-backtracking-oo“ benötigt nur noch einen einzigen Parameter, nicht eine Vielzahl von Funktionsargumenten.
- Default-Methoden können geerbt werden.
- Das Backtracking-Schema kann durch Vererbung erweitert werden, ohne daß wir den Rahmen ändern müssen.

Das vollständige Programm finden Sie im teachpack „backtracking-oo-module.ss“. Anwendungen des objektorientierten Backtracking-Schemas finden Sie in den teachpacks „missionaries-module.ss“ (vier Missionare auf der Flucht), „queens-oo-module.ss“ (8-Damen-Problem).

Anforderungen an eine Programmiersprache zur objektorientierten Programmierung

- Zustände
- Hybride Datenstrukturen, Typpräädikate
- Polymorphie
- Dynamisches Binden von Funktionen (Methoden)
- Datenkapselung
- Um Zustände zu realisieren, müssen wir funktionale Programmiersprachen um imperitative Sprachelemente (Modifikatoren) erweitern,
- aber alle anderen Anforderungen lassen sich sehr gut mit einem reinen funktionalen Programmierstil verbinden, wie wir am Beispiel von CLOS gesehen haben.

☞ Anmerkung: Einbettung der objektorientierten Programmierung

Jede Programmiersprache ist daraufhin entworfen, einen bestimmten Programmierstil besonders gut zu unterstützen, aber es ist meistens möglich, auch andere Programmierstile zu verwenden, indem man sich in der Programmiersprache eigene Sprachelemente definiert.

Um objektorientiert programmieren zu können, muß eine Programmiersprache bestimmte Grundfunktionen bieten:

Zustände: Um Objekte zu implementieren, muß es Datenstrukturen geben, die ihren Zustand verändern können.

Dynamisches Erzeugen von Datenstrukturen: Datenstrukturen müssen zur Laufzeit erzeugt und gelöscht werden können.

Hybride Datenstrukturen, Typprädikate: Es muß möglich sein, strukturierte Daten aus Objekten unterschiedlichen Typs aufzubauen und die Typen der Komponenten dynamisch abzufragen.

Polymorphie: Zur Implementation von Generalisierung und Spezialisierung, Vererbung

Dynamisches Binden von Funktionen (Methoden): Erst zur Laufzeit wird anhand der Klasse eines Objekts die Methode zur Durchführung einer Operation ausgewählt.

Datenkapselung: Objekte bilden eine Datenkapsel um ihre Attribute.

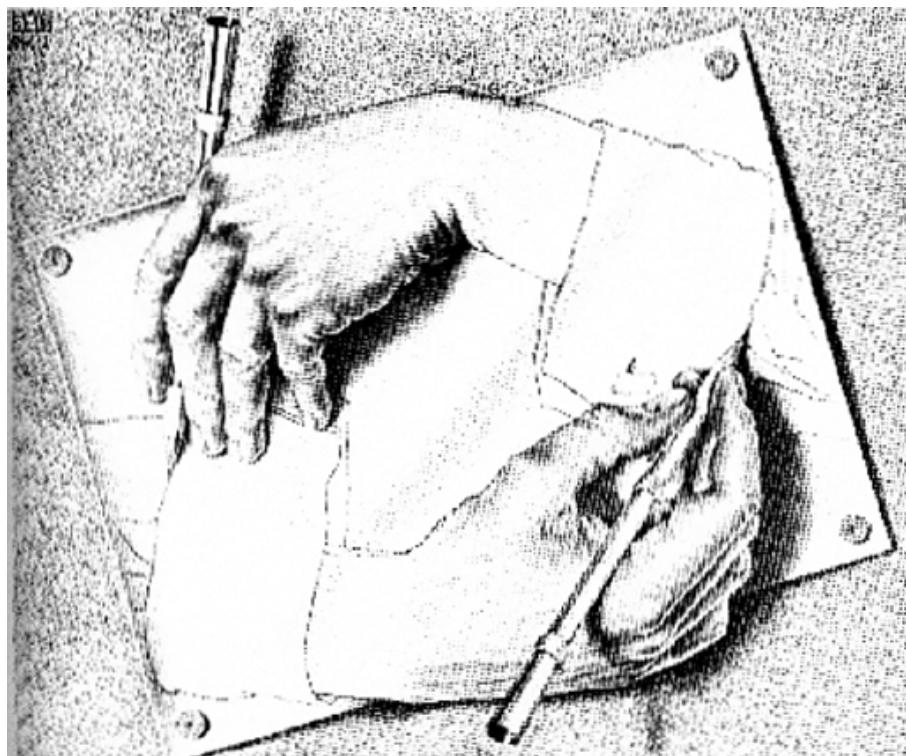
In den meisten imperativen Sprachen lassen sich viele Aspekte des objektorientierten Programmierstils nachbilden, aber der Vererbungsmechanismus läßt sich nur nutzen, wenn die Programmiersprache polymorphe Typen kennt.

Swindle und CLOS sind vollständig in Scheme programmiert. Sie können sich die Implementation im Unterverzeichnis „collects“ des DrScheme-Systems anschauen. Swindle und CLOS sind eine konsistente Erweiterung von Scheme, die sich harmonisch dem funktionalen Programmierstil anpaßt. Die Klassen und die Vererbung nutzen die ohnehin schon vorhandene Typhierarchie; der Zugriff auf Attribute und die Spezialisierung von Operationen sind vollständig funktional programmiert. Auch die zusätzlichen Modifikatoren sind kein Stilbruch, da die Lisp-Sprachen ohnehin schon imperative Sprachelemente enthalten.

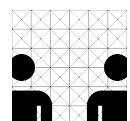
Im nächsten Kapitel werden wir selbst eine Erweiterung der Sprache Scheme definieren, indem wir Sprachelemente für die stromorientierte Programmierung definieren. Dazu werden wir eigene *special forms* einführen.

Softwareentwicklung III

Leonie Dreschler-Fischer
Prüfungsunterlagen Teil 2:
Objekte und generische Funktionen



Universität Hamburg
Fachbereich Informatik
WS 2019/2020



30 Das Simulationssystem: Der Sourcocode

30.1 Modulstruktur

☞ Anmerkung: Das Simulationssystem ist nach dem Model-View-Control-Entwurfsschema strukturiert.

Modellklassen modellieren die aktiven Objekte des Simulationssystems.

View-Klassen modellieren die externe Darstellung der Simulation, aufbauend auf `world.ss`.

Control-Klassen modellieren die Interaktion und die Ablaufsteuerung, aufbauend auf `world.ss`.

Sofern notwendig, wurden die Klassen auf mehrere Module aufgeteilt:

- Ein Modul für die Definition der Klasse und das Interface der generischen Funktionen,
- ein Modul für die Implementation der Methoden,
- Module für die View-Klassen.

Alle Module des Simulationssystems finden Sie im Verzeichnis `se3-bib/sim`.

Architektur des Simulationssystems

Module	sim-utility-base-package
setf	Swindle utilities
misc	Swindle utilities
world	Simulationsrahmen
tools-module	Funktionen höherer Ordnung

Module	sim-base-package	
queuesClass	queuesImpl	model
simActorClass	simActorImpl	model
simActorViewClass	simActorViewImpl	view
simClockClass	simClockImpl	model
simEventClass	simEventImpl	model
simEventSourceClass	simEventSourceImpl	model
simCalendarClass	simCalendarImpl	model
simScenarioClass	simScenarioImpl	model
simPathClass	simPathImpl	control

Module	sim-application-package	
simDoorClass	simDoorImpl simDoorView	model view
simServerStationsClass	simServerStationsImpl	model
simServerStationsViewClass	simServerStationsViewImpl	view
simCustomerClass	simCustomerImpl simCustomerView	model view
customerServerScenarioClass	customerServerScenarioCont customerServerScenarioView	control
mensa-scenario-package mensaScenarioClass	mensaScenarioImpl	model
mensa-scenario2-package mensaScenario2Class	mensaScenario2Impl	model
aremorica-scenario-package aremoricaScenarioClass	aremoricaScenarioImpl	model
aremorica-scenario2-package aremoricaScenario2Class	aremoricaScenario2Impl	model

☞ Anmerkung: Das Aremoricas-Szenario Das Aremoricas-Szenario werden wir in der Vorlesung nicht besprechen. Es ist eine weitere Anwendung des Bediensystems mit spezialisierten Kunden.

Wir sehen ein kleines Dorf im alten Gallien (Provinz Aremorica) zur Zeit der gallischen Kriege. Die tapferen Gallier haben unterschiedliche Besorgungen zu erledigen. Der Druide hat einen Stand, an dem er Zaubertrank ausschenkt, der Häuptling gibt Audienzen, der Schmied verkauft Waffen, der Fischhändler frisch aus Lutetia importierte Fische. Auch auswärtige Besucher kommen zum Einkaufsbummel.

Die Kundenklasse und serve-Methode wurde hier ebenfalls spezialisiert: Nicht alle Kunden erhalten Zaubertrank (nur die erwachsenen Gallier, die nicht als kleine Kinder in den Zaubertrank gefallen sind) und Waffen werden auch nur an Erwachsene Gallier und befreundete Ausländer verkauft.

Am Aremorica-Szenario können Sie sich anschauen, wie große Mengen von erzeugten Objekten mittels einer Hashtabelle der Zielklassen automatisch zur Zielklasse konvertiert werden können.

30.2 Das Basissystem

File: sim-utility-base-package.ss

Dieser File bündelt Grundfunktionen von Swindle und das tools-module zu einem Paket, das alle die exportierten Namen zusammenfaßt.

```
(module sim-utility-base-package swindle

  (print "loading:_sim-utility-base-package_")
  (require
    swindle/setf
    swindle/misc
    teachpack/htdp/world
    (all-except se3-bib/tools-module
      mappend every some last concat)
    se3-bib/sim/simBase/sentinel-lists-module
  )
  (provide
    (all-from swindle/setf)
    (all-from swindle/misc)
    (all-from teachpack/htdp/world)
    (all-from se3-bib/tools-module)
    (all-from se3-bib/sim/simBase/sentinel-lists-module)
  ) )
```

File: sim-base-package.ss

Dieser File bündelt die Basismodule zu einem Paket, das alle die exportierten Namen zusammenfaßt.

```
#|
  Author: Leonie Dreschler-Fischer
  Version: 0002.0 last changed December 25, 2006 |#  
  
(module sim-base-package swindle  
  
  (print "loading:_sim-base-package_")  
  
  (require  
    se3-bib/sim/simBase/sim-utility-base-package  
    se3-bib/sim/simBase/queuesClass-module  
    se3-bib/sim/simBase/queuesImpl-module  
    se3-bib/sim/simBase/simActorClass-module  
    se3-bib/sim/simBase/simActorImpl-module  
    se3-bib/sim/simBase/simActorViewClass-module  
    se3-bib/sim/simBase/simActorViewImpl-module  
    se3-bib/sim/simBase/simClockClass-module  
    se3-bib/sim/simBase/simClockImpl-module  
    se3-bib/sim/simBase/simEventClass-module  
    se3-bib/sim/simBase/simEventImpl-module  
    se3-bib/sim/simBase/simEventSourceClass-module  
    se3-bib/sim/simBase/simEventSourceImpl-module  
    se3-bib/sim/simBase/simCalendarClass-module  
    se3-bib/sim/simBase/simCalendarImpl-module  
    se3-bib/sim/simBase/simScenarioClass-module  
    se3-bib/sim/simBase/simScenarioImpl-module  
    se3-bib/sim/simBase/simPathClass-module  
    se3-bib/sim/simBase/simPathImpl-module)  
  
  (provide  
    sim-base-test  
    (all-from se3-bib/sim/simBase/sim-utility-base-package)  
    (all-from se3-bib/sim/simBase/queuesClass-module)  
    (all-from se3-bib/sim/simBase/queuesImpl-module)  
    (all-from se3-bib/sim/simBase/simActorClass-module)  
    (all-from se3-bib/sim/simBase/simActorImpl-module)  
    (all-from se3-bib/sim/simBase/simClockClass-module))
```

```
(all-from se3-bib/sim/simBase/simClockImpl-module)
(all-from se3-bib/sim/simBase/simEventClass-module)
(all-from se3-bib/sim/simBase/simEventImpl-module)
(all-from se3-bib/sim/simBase/simEventSourceClass-module)
(all-from se3-bib/sim/simBase/simEventSourceImpl-module)
(all-from se3-bib/sim/simBase/simCalendarClass-module)
(all-from se3-bib/sim/simBase/simCalendarImpl-module)
(all-from se3-bib/sim/simBase/simScenarioClass-module)
(all-from se3-bib/sim/simBase/simScenarioImpl-module)
(all-from se3-bib/sim/simBase/simActorViewClass-module)
(all-from se3-bib/sim/simBase/simActorViewImpl-module)
(all-from se3-bib/sim/simBase/simPathClass-module)
(all-from se3-bib/sim/simBase/simPathImpl-module)
)
```

File: simActorClass-module.ss

```
(module simActorClass-module swindle
  (provide
    actor-num actor-name sim-actor?
    theActors set-theActors! theLastNum
    *theNumberGenerator* *all-actors*
  )
  (print "loading:_simActorClass-module_")
  (require
    se3-bib/sim/simBase/sim-utility-base-package)

  (defclass* actorNumbers ()
    (lastnum :initvalue 0
              :accessor theLastNum
              :type <integer>
              :allocation :class)
    :autopred #t ; auto generate predicate sim-actor?
    :printer #t
    :documentation "a_generator_for_unique_actor_numbers" )

  (define *theNumberGenerator* (make actorNumbers))

  (defgeneric* next-actor-number ((ng actorNumbers))
    :documentation "generate_a_unique_actor_number"
```

```

:method (method
          ((ng actorNumbers))
          (inc! (theLastNum ng))
          (theLastNum ng)))

(defmethod next-actor-number
  ((ng actorNumbers))
  (inc! (theLastNum ng))
  (theLastNum ng))

(defclass* sim-actor ()
  (actorName
    :reader actor-name
    :initvalue "anonymous"
    :initarg :actorName
    :type <string>
    :documentation "The_name_of_the_actor")
  (actorNum
    :reader actor-num
    :initializer
    (lambda ()
      (next-actor-number
        *theNumberGenerator*))
    :type <integer>
    :documentation
    "A_picture_of_the_current_state_of_the_actor" )
  :autopred #t ; auto generate predicate sim-actor?
  :printer #t )

(defclass* setOfActors ()
  (theActors
    :accessor theActors
    :writer set-theActors!
    :initvalue '()
    :type <list>
    :documentation
    "A_set_of_actors")
  :autopred #t ; auto generate predicate set-of-actors?
  :printer #t )

(define *all-actors* (make setOfActors))

```

```

(defgeneric* add-actor! ((a sim-actor) (s setOfActors))
  :documentation
  "add_an_actor_to_the_set_of_all_actors")

(defgeneric* remove-duplicates! ((s setOfActors))
  :documentation
  "remove_duplicate_entries")

(defgeneric* sim-init! ((obj sim-actor))
  :documentation
  "further_initializations_after_all_actors_have_been_created"
  :combination generic-begin-combination)

(defgeneric* sim-start ((obj sim-actor))
  :documentation
  "Trigger_the_start-up_actions,_get_the_system_going"
  :combination generic-begin-combination)

(defgeneric* sim-info ((obj sim-actor))
  :documentation
  "request_state_information."
  :combination generic-begin-combination)

;;; Message passing

(defgeneric* tell ((obj sim-actor) (message <function>))
  :documentation
  "Pass_message_to_actor.
  The_message_must_be_the_name
  of_an_applicable_method_or_procedure.")

(defgeneric* broadcast
  ((actor sim-actor)
   (message <function>)
   &key (sim-class sim-actor))
  :documentation
  "Broadcast_a_message_to_all_actors_of_type_sim-class")

(defgeneric* broadcast-some
  ((actor sim-actor))

```

```

(message <function>)
&optional (sim-class sim-actor))
:documentation
"Send_message_to_actors_of_type_sim-class ,
and_find_some_actor_who_confirms_the_message
by_answerignot_#f .
) )

```

File: simActorImpl-module.ss

Die Implementation der Klasse simActor.

```

(module simActorImpl-module swindle

  (print "loading:simActorImpl-module")
  (require
    se3-bib/sim/simBase/sim-utility-base-package
    se3-bib/sim/simBase/simActorClass-module)

  (defmethod equals?
    ((a sim-actor) (b sim-actor))
    (= (actor-num a)(actor-num b)))

  (defmethod add-actor! ((a sim-actor) (s setOfActors))
    ; add an actor to the list of all actors
    (push! a (theActors s)))

  (define (remove-dupl! xs)
    (if (null? xs) '()
        (if (some (curry equals? (car xs)) (cdr xs))
            (remove-dupl! (cdr xs))
            (cons (car xs) (remove-dupl! (cdr xs))))))

  (defmethod remove-duplicates! ((s setOfActors))
    "remove_duplicate_entries"
    ; necessary , if change class is used on an actor
    ; then initialize will be called again
    (set-theActors! s (remove-dupl! (theActors s))) )

  (defmethod initialize
    ; maintaining the list of actors;

```

```

; applied after the main initialization
:after
((obj sim-actor) args)
(add-actor! obj *all-actors*))

;; Message passing

(defmethod sim-info ((obj sim-actor))
  (writeln (actor-num obj) ":"_
            (actor-name obj) " " (class-of obj)))

(defmethod sim-start ((obj sim-actor))
  "Trigger_the_start-up_actions ,_get_the_system_going"
  #t)

(defmethod sim-init! ((obj sim-actor))
  "further_initializations_after_all_actors_have_been_created"
  #t)

(defmethod tell
  ((a sim-actor) (message <function>))
  ; tell: sim-actor, (sim-actor -> any) -> any
  "Pass_message_to_actor.
  The_message_must_be_the_name
  of_an_applicable_method_or_procedure."
  (message a))

(defmethod broadcast
  ((actor sim-actor)
   (message <function>)
   &key (sim-class sim-actor))
  "Broadcast_message_to_all_actors_of_type_sim-class"
  (dolist (actr (reverse (theActors *all-actors*)))
    (when (instance-of? actr sim-class)
      (tell actr message)))
  #t)

(defmethod broadcast-some
  ((actor sim-actor)
   (message <function>)
   &optional (sim-class sim-actor)))

```

```

"Send_message_to_actors_of_type_sim-class,
and_find_some_actor_who_confirms_the_message_by_answering'_not_#f"
(some
  (lambda
    (actr)
    (if (and
      (equal? (class-of actr) sim-class)
      (tell actr message))
      actr #f))
    (theActors *all-actors*))) )

```

File simActorViewClass-module.ss

Die view-Klasse für sim-actor-Objekte. Die generische Funktion actor-picture liefert ein image-Objekt des sim-actor-Objektes.

```

(module simActorViewClass-module swindle

  (print "loading:_simActorViewClass-module_")
  (require
    se3-bib/sim/simBase/sim-utility-base-package)

  (provide
    set-actor-pic! actor-pic
    *defaultpic* *defaultRCpic* )

  (define *defaultpic*  ); a default picture for actors
  (define *defaultRCpic*  ); a deflt. pict. for special actors

  (defclass* sim-actor-view ()
    (actorPicture
      :reader actor-pic
      :writer set-actor-pic!
      :initarg :actorPic
      :initvalue *defaultpic*
      :documentation
        "A_picture_of_the_current_state_of_the_actor" )
      :autopred #t ; auto generate predicate sim-actor-view?
    :printer #t

```

```

:documentation "A_mixin_class_for_simulation_objects ,
a_framework_for_creating_pictures_of_all_actors"
)

(defgeneric* has-picture? ((obj sim-actor-view))
:documentation
"Does_the_actor_have_an_unique_picture
or_only_a_default_picture?")

(defgeneric* actor-picture ((a sim-actor-view))
:documentation
"Abstract:_return_a_picture_of_the_actor's_state" )

(defgeneric* pictureWithName ((actor sim-actor) )
; a picture of the actor with the name below
; on a gray background
) )

```

File: simActorViewImpl-module

Die Implementation von simActorView.

(module simActorViewImpl-module swindle

```

(provide
set-actor-pic! actor-pic
*defaultpic*
string->image
cjustify
*canv-width* *canv-height*
*bg-screen* )

(print "loading:_simActorViewImpl-module_")
(require
se3-bib/sim/simBase/sim-utility-base-package
se3-bib/sim/simBase/simActorViewClass-module
se3-bib/sim/simBase/simActorClass-module
se3-bib/sim/simBase/simActorImpl-module)
(provide sim-info)

; the canvas of the application window

```

```

(define *canv-width* 750)
(define *canv-height* 500)
(define *bg-screen*
  (empty-scene *canv-width* *canv-height*))

(defmethod actor-picture ((a sim-actor-view))
  "Default: return a picture of the actor's state"
  (writeln "actor-picture:sim-actor-view")
  (actor-pic a))

(defmethod has-picture? ((obj sim-actor-view))
  "does the actor have an unique picture?"
  (not (eq? (actor-pic obj) *defaultpic*)))

(defmethod sim-info ((obj sim-actor-view))
  (writeln (actor-pic obj)))

(define (string->image theString)
  ; make an image from a string
  (text theString
        24 'red))

(define (cjustify n s)
  (let* ((len (string-length s))
         (left (quotient (- n len) 2))
         (right (- n left len)))
    (if (>= n len)
        (string-append
         (space left) s (space right))
        (substring s 0 n)))

(defmethod pictureWithName ((actor sim-actor))
  ; a picture of the actor with the name below
  ; on a gray background
  (let* ((longest (text "graftvornix_" 14 'red))
         (aPic (actor-pic actor))
         (namePic
          (text
           (cjustify 12 (actor-name actor)) 18 'red)))
    (h (+ (image-height aPic)
          (image-height namePic))))

```

```

(w (max (image-width aPic)
         (image-width namePic)
         (image-width longest )))
(bg (rectangle w h 'solid 'gray))
(picOnBG
  (overlay/xy bg 0
    (- (floor (/ (image-height namePic) 2)))
      aPic))
(picOnBGwithName
  (overlay/xy
    picOnBG
    (- (pinhole-x namePic)
        (pinhole-x picOnBG)))
    (+ (image-height aPic)
        (- (pinhole-y namePic)
            (pinhole-y picOnBG)
            )))
    namePic))
picOnBGwithName)))

```

File: simCalendarClass-module.ss

Der Simulationskalender: Eine Warteschlange von Weckaufträgen, geordnet nach Weckzeit.

```

(module simCalendarClass-module swindle

  (print "loading:_simCalendarClass-module")
  (require
    se3-bib/sim/simBase/sim-utility-base-package
    se3-bib/sim/simBase/simActorClass-module
    se3-bib/sim/simBase/queuesClass-module
    se3-bib/sim/simBase/simClockClass-module
    se3-bib/sim/simBase/simEventClass-module)

  (defclass* sim-calendar
    (RANKED-queue sim-actor)
    (rankp
      :initvalue time-due
      :documentation

```

```

        "the_rank_of_an_event_is_its_time_to_occur")
:autopred #t ; auto generate predicate sim-calendar?
:printer #t
:documentation "The_calendar_of_scheduled_events"
)

(defgeneric* peak-next-event ((calendar sim-calendar))
:documentation
"The_next_event_in_the_event_queue")

(defgeneric* dispatch ((calendar sim-calendar))
:documentation
"Dispatch_a_calendar_event_to_happen_right_now.
Advance_the_clock_to_the_time_of_the_event.
returns_the_calendar")

(defgeneric* schedule
((event sim-event)
 time-due )
:documentation
"enter_an_event_in_the_calendar"
:combination generic-begin-combination)

(defgeneric* calendar-info ((calendar sim-calendar));
:documentation
"request_state_information_from_the_calendar.")

;; special simulation events and handlers
(defclass* sim-deadlock (sim-event)
:autopred #t ; auto generate predicate sim-deadlock?
:printer #t
:documentation
"The_calendar_hasemptied_out_prematurely:
Simulation_stalled" )

(defclass* sim-Quit (sim-event)
:autopred #t ; auto generate predicate sim-deadlock?
:printer #t
:documentation
"stop_the_simulation"
) )

```

File: simCalendarImpl-module

Implementation des Simulationskalenders, scheduling, dispatching.

```
(module simCalendarImpl-module swindle

(print "loading:_simCalendarImpl-module_")

(require
  se3-bib/sim/simBase/sim-utility-base-package
  se3-bib/sim/simBase/simActorClass-module
  se3-bib/sim/simBase/queuesClass-module
  se3-bib/sim/simBase/queuesClass-module
  se3-bib/sim/simBase/queuesImpl-module
  se3-bib/sim/simBase/simClockClass-module
  se3-bib/sim/simBase/simClockImpl-module
  se3-bib/sim/simBase/simEventClass-module
  se3-bib/sim/simBase/simEventImpl-module
  se3-bib/sim/simBase/simCalendarClass-module)

(define *current-calendar* #f); class sim-calendar
; "The current simulation scenario, hidden,
; will be set by 'initialize'"

(defmethod
  initialize :after
  ((newCalendar sim-calendar) initargs)
  (setf! *current-calendar*
    newCalendar))

(defmethod calendar-info ((calendar sim-calendar));
  "request_state_information_from_the_calendar."
  (writeln "___Calendar_events:")
  (map (rcurry display-event)
    (enumerate-queue calendar))
  #t)

(defmethod sim-info ((calendar sim-calendar));
  (calendar-info calendar))

(defmethod schedule
```

```

((event sim-event)
 time-due)
"enter_an_event_in_the_calendar"
(set-time-due! event time-due)
(enqueue! *current-calendar* event)
*current-calendar*)

(defmethod peak-next-event ((calendar sim-calendar))
  (head-queue calendar))

(defmethod dispatch ((calendar sim-calendar))
  "Dispatch_a_calendar_event_to_happen_right_now.
  Advance_the_clock_to_the_time_of_the_event."
  (if (empty-queue? calendar)
    (schedule
      (make sim-deadlock)
      (now)))
    (let ((nextEvent (dequeue! calendar)))
      (set-clock! (time-due nextEvent))
      (writeln "Clock:" (now));-----
      (when (scene-protocol nextEvent)
        (display-event nextEvent))
      (handle nextEvent)
      calendar)))

(defmethod handle ((event sim-deadlock))
  (print "****_Warning:_Simulation_deadlock_occured!")
  (schedule (make sim-Quit) (now))
  #f)
)

```

File: simClockClass-module.ss

Die Definition der Klasse simClock.

```

(module simClockClass-module
  swindle

  (provide
    read-sim-clock set-sim-clock!    )

```

```

(print "loading:_simClockClass-module_")
(require
  se3-bib/sim/simBase/sim-utility-base-package
  se3-bib/sim/simBase/simActorClass-module)

(defclass* sim-clock (sim-actor)
  (sim-time
    :reader read-sim-clock
    :writer set-sim-clock!
    :initvalue 0
    :type <number>
    :documentation
      "The_simulation_time")
  :autopred #t ; auto generate predicate sim-clock?
  :printer #t
  :documentation "The_simulation_clock"
  )

(defgeneric* clock-info ((clock sim-clock)))
)

```

File: simClockImpl-module

Die Implementation der Simulationsuhr

```

(module simClockImpl-module swindle

(provide
  now set-clock! advance-clock! add-time
  *current-clock* )

(print "loading:_simClockImpl-module")
(require
  se3-bib/sim/simBase/sim-utility-base-package
  se3-bib/sim/simBase/simActorClass-module
  se3-bib/sim/simBase/simActorImpl-module
  se3-bib/sim/simBase/simClockClass-module
  )

(define *current-clock* #f)
; the current simulation clock, hidden,

```

```

; will be set by initialize , whenever a new clock is created

(defmethod initialize :after
  ((newClock sim-clock) initargs)
  (setf! *current-clock*
        newClock)
  (if (next-method?)
      (call-next-method obj args)))

(defmethod clock-info ((clock sim-clock))
  (writeln "Clock:" (read-sim-clock clock)))

(defmethod sim-info ((clock sim-clock))
  (clock-info clock))

(define (set-clock! sim-time)
  "Set_the_clock_to_a_specific_time_t ."
  (set-sim-clock! *current-clock* sim-time))

(define (now)
  "Read_the_simulation_clock ."
  (read-sim-clock *current-clock*))

(define (add-time time incr-time-units)
  "Add_incr_to_time"
  ; might be extended for more units of time , e.g. h m s d
  (+ time incr-time-units))

(define (advance-clock! incr-time-units)
  "Advance_the_simulation_time_by_increment_time_units"
  (set-clock! (add-time (now) incr-time-units)))
)

```

File: simEventClass-module.ss

Die mixin-Klasse sim-event für Simulationsereignisse, der event-handler handle.

```
(module simEventClass-module swindle
```

```

(provide
  set-time-due! time-due
  scene-protocol set-scene-protocol!
  set-event-message! eventMes )

(print "loading:_simEventClass-module")
(require
  se3-bib/sim/simBase/sim-utility-base-package
  se3-bib/sim/simBase/simClockClass-module
  se3-bib/sim/simBase/simClockImpl-module
)

(defclass* sim-event ()
  (scheduled-time
    :reader time-due
    :writer set-time-due!
    :initvalue 0
    :initarg :time-due
    :type <number>
    :allocation :instance
    :documentation
      "The_scheduled_time_for_the_event_to_take_place")
  (eventMessage
    :reader eventMes
    :writer set-event-message!
    :initvalue "no_event"
    :initarg :eventMes
    :allocation :class
    :type <string>
    :documentation
      "A_message_text_describing_the_current_event")
  (protocol
    :reader scene-protocol
    :writer set-scene-protocol!
    :initvalue #t
    :initarg :scene-protocol
    :allocation :class
    :documentation
      "When_true,_each_event_will_be_traced")
  :autopred #t ; auto generate predicate sim-event?
  :printer #t)

```

```

:documentation
"Simulation_events_to_be_entered_in_a_simulation_calendar"
)

(defclass* sim-nothingHappens (sim-event)
 :autopred #t
 :printer #t
 :documentation
 "a_do_nothing_event" )

(defgeneric* handle ((event sim-event))
 :documentation
 "A_handler_for_simulation_events.
 For_each_application_specific_subclass_of_events
 you_should_provide_a_specific_handler." )

(defgeneric* display-event ((event sim-event) )
 :documentation
 "Show_the_event"
 :combination generic-begin-combination)
)

```

File: simEventImpl-module.ss

```

(module simEventImpl-module
 swindle

(print "loading:simEventImpl-module")
(require
 se3-bib/sim/simBase/sim-utility-base-package
 se3-bib/sim/simBase/simClockClass-module
 se3-bib/sim/simBase/simClockImpl-module
 se3-bib/sim/simBase/queuesClass-module
 se3-bib/sim/simBase/queuesImpl-module
 se3-bib/sim/simBase/simEventClass-module)

(defmethod display-event ((event sim-event) )
  (writeln (class-of event)
   ",_time_due:_" (time-due event)))

```

```

event)

; generic defined in queuesClass-module
(defmethod rank ((item sim-event))
  ; a rank in a queue, item → number
  (time-due item))

; generic defined in queuesClass-module
(defmethod less? ((event-1 sim-event) (event-2 sim-event))
  "The_ranking_of_events_in_a_ranked_queue."
  ; an earlier event has precedence before a later event
  (< (time-due event-1)
    (time-due event-2)))

(defmethod handle ((event sim-event))
  "A_default_handler_for_simulation_events.
  For_each_application_specific_subclass_of_events
  you_should_provide_a_specific_handler."
  (set-event-message!
    event
    (symbol->string (class-name (class-of event)))))

(defmethod handle ((event sim-nothingHappens))
  (set-event-message! event "nothing_happens_event")
#t)
)

```

File: simEventSourceClass-module.ss

Die Klasse der Ereignisquellen: sim-event-source, generische Funktionen für die Erzeugung der Ereignisse next-event, next-exp-event usw.

```

(module simEventSourceClass-module
  swindle

  (provide
    event-count set-event-count!
    event-rate event-sig )

  (print "loading:_simEventSourceClass-module_")
  (require

```

```

se3-bib/sim/simBase/sim-utility-base-package
se3-bib/sim/simBase/simActorClass-module
se3-bib/sim/simBase/queuesClass-module
se3-bib/sim/simBase/simClockClass-module
se3-bib/sim/simBase/simEventClass-module
se3-bib/sim/simBase/simCalendarClass-module)

(defclass* sim-event-source (sim-actor sim-event)
  (events-per-time-unit
   :reader event-rate
   :type <number>
   :initvalue 1
   :initarg :rate
   :documentation
   "The_mean_value_of_events_per_time_unit")
  (event-count
   :reader event-count
   :writer set-event-count!
   :initvalue 0
   :documentation "The_number_of_events_so_far"
   :documentation "A_source_of_random_events"
   :autopred #t ; auto generate predicate sim-event?
   :printer #t))

(defclass* sim-const-event-source (sim-event-source)
  :documentation "A_source_of_events_at_constant_intervals"
  :autopred #t
  :printer #t)

(defclass* sim-exp-event-source (sim-event-source)
  :documentation "A_source_of_uniformally_distributed_events"
  :autopred #t
  :printer #t)

(defclass* sim-normal-event-source (sim-event-source)
  (variance
   :reader event-sig
   :type <number>
   :initvalue 0.3
   :initarg :sigma
   :documentation

```

```

    "The_variance_of_the_time_between_events")
:documentation "A_source_of_normally_distributed_events"
:autopred #t
:printer #t )

(defgeneric* next-event ((actor sim-event-source))
  :documentation
  "Schedule_the_actor_for_the_next_event_to_occur." )

(defgeneric* next-now-event ((actor sim-event-source))
  :documentation
  "Schedule_the_actor_for_an_event_to_take_place_right_now."
  )
(defgeneric* next-exp-event ((actor sim-event-source))
  :documentation
  "Schedule_the_actor_for_the_next_exponentially
  distributed_event.")

(defgeneric* next-normal-event ((actor sim-event-source)
                                &key (sigma 1))
  :documentation
  "Schedule_the_actor_for_the_next_normally_distributed_event." )

(defgeneric* next-const-event ((actor sim-event-source))
  :documentation
  "Schedule_the_actor_for_the_next_event
  with_constant_intervals_between_events."
  )

```

File: simEventSourceImpl-module.ss

```

(module simEventSourceImpl-module swindle

(provide
  random-sig random-uni random-normal
  random-exp )

(print "loading:_simEventSourceImpl-module_")
(require

```

```

se3-bib/sim/simBase/sim-utility-base-package
se3-bib/sim/simBase/simActorClass-module
se3-bib/sim/simBase/simActorImpl-module
se3-bib/sim/simBase/queuesClass-module
se3-bib/sim/simBase/queuesImpl-module
se3-bib/sim/simBase/simClockClass-module
se3-bib/sim/simBase/simClockImpl-module
se3-bib/sim/simBase/simEventClass-module
se3-bib/sim/simBase/simEventImpl-module
se3-bib/sim/simBase/simCalendarClass-module
se3-bib/sim/simBase/simCalendarImpl-module
se3-bib/sim/simBase/simEventSourceClass-module
)

(define (random-uni &key (range 1.0)
                           (granularity 2147483647))
  "A_random_real_number,
  uniformly_distributed_between_0_and_range .
  (* (/ (random granularity)
         granularity)
     range))

(define (random-sig &key (p-positive 0.5))
  "A_random_signum,_1_or_-1"
  (if (> (- (random-uni) p-positive) 0.0)
      -1 1))

(define (random-exp &key (1/mu 1.0))
  "Return_a_random_number,
  with_a_negative_exponential_distribution ,
  arrival_rate_1/mu_in_a_simulation."
  (/ (- (log (random-uni)))
    1/mu))

(define (random-normal &key (mu 0.0) (sigma 1.0))
  "Return_a_random_number,_with_a_normal_distribution ,
  mean_value_mu,_variance_sigma^2."
  (let* ((v (1- (* 2 (random-uni)))))
    (r (abs v))
    (fact (sqrt (/ (* -2 (log r) r))))))
  (+ mu (* fact v sigma))))
```

```

(define (random-mixture
            randoms-1 frac-1 randoms-2)
  "Random_numbers_obeying_a_mixture_distribution"
  (if (> (random-uni) frac-1)
      (randoms-1)
      (randoms-2)))

(define (random-normal-mixture
            &key (mu-1 1.0) (sigma-1 1.0)
                  (mu-2 2.0) (sigma-2 1.0)
                  (frac-1 0.5))
  "Random_numbers_obeying_a_mixture_distribution"
  (random-mixture
    (curry random-normal :mu mu-1 :sigma sigma-1)
    frac-1
    (curry random-normal :mu mu-2 :sigma sigma-2)))

(defmethod next-event ((actor sim-event-source))
  "Default:_Schedule_the_actor_for_the_next
  exp._distributed_event."
  (next-exp-event actor))

(defmethod next-event ((actor sim-const-event-source))
  "Schedule_the_actor_for_the_next_event
  with_constant_intervals_between_events."
  (next-const-event actor))

(defmethod next-event ((actor sim-exp-event-source))
  "Schedule_the_actor_for_the_next
  exponentially_distributed_event."
  (next-exp-event actor))

(defmethod next-event ((actor sim-normal-event-source))
  "Schedule_the_actor_for_the_next
  exponentially_distributed_event."
  (next-normal-event actor :sigma (event-sig actor)))

(defmethod handle :after ((actor sim-event-source))
  "After_handling_an_event_source_create_the_next_event"
  (writeln "handle:_after_" (actor-name actor)))

```

```

(next-event actor))

(defmethod next-now-event ((actor sim-event-source))
  "Schedule the actor for an event to take place right now."
  (inc! (event-count actor))
  (schedule actor (now)))

(defmethod next-exp-event ((actor sim-event-source))
  "Schedule the actor for the next exponentially distributed event."
  (inc! (event-count actor))
  (schedule actor
            (add-time (now)
                      (random-exp :1/mu
                                  (event-rate actor)))))

(defmethod next-normal-event
  ((actor sim-event-source) &key (sigma 1))
  "Schedule the actor for the next normally distr. event."
  (inc! (event-count actor))
  (schedule actor
            (add-time (now)
                      (abs (random-normal
                            :mu (/ 1 (event-rate actor))
                            :sigma sigma)))))

(defmethod next-const-event ((actor sim-event-source))
  "Schedule the actor for the next event with constant intervals between events."
  (inc! (event-count actor))
  (schedule actor
            (add-time (now)
                      (/ 1 (event-rate actor)))))

(defmethod sim-init! ((source sim-event-source))
  "Initialize for the first simulation run"
  (set-event-count! source 0))

(defmethod sim-info ((source sim-event-source))
  "request state information"
  (writeln "Event_source:" (actor-name source))

```

```

        " _Number_of_events:_ " (event-count source))
))

```

File: sentinel-list-module.ss

Destruktive Operationen auf Listen, für eine effiziente Implementation der queue-Operationen. Diese Operationen nutzen die modifizierbaren Listen (mutable list), um Elemente in Listen einzufügen oder entfernen, ohne Kopien der Listen erstellen zu müssen. Die Listen sind mit Wächtern (sentinels, spezialisierte Elemente am Anfang und am Ende) versehen, so daß die Liste nie leer wird.

```

(module sentinel-lists-module scheme
  (provide sm-push!
            sm-pop!
            sm-makelist;
            sm-first-element
            sm-insert-at-pos!
            sm-insert-at-back!
            sm-insert-Ranked!
            sm-empty?
            sm-length
            sm-first-element ; the first element that is not a sentinel
            sm->list)

  (require scheme/mpair)
  (all-except scheme/mpair mappend)) ; use mutable lists

  (define (mcadr mxs)
    (mcar (mcdr mxs)))

  (define (mcddr mxs)
    (mcdr (mcdr mxs)))

  (define (sm-first-element mxs)
    (mcadr mxs)); skip sentinel

  (define-struct sentinel
    (firstCons lastCons theLength)
    #:mutable)

```

```

(define (sm-makelist)
  ; create sentinels at front and end,
  ; initialize the reference to the last cons-cell:
  ; the sentinel points to the last cons
  ; and keeps track of the length of the mlist.
  (let* ((the-sentinel (make-sentinel '() '() 0))
         (new-list (mlist the-sentinel the-sentinel)))
    (set-sentinel-firstCons! the-sentinel new-list)
    (set-sentinel-lastCons! the-sentinel new-list)
    new-list
  ))

(define (theSentinel mxs)
  (mcar mxs))

(define (sm-length mxs)
  (sentinel-theLength (theSentinel mxs)))

(define (sm-empty? mxs)
  (zero? (sentinel-theLength (theSentinel mxs)))))

(define (sm->list mxs)
  ; copy a sentinel-list to a proper list and remove the sentinel
  (filter (lambda (x)
             (not (sentinel? x)))
          (mlist->list mxs)))

(define (insert-item-behind-cons! mxs item senti)
  ; inserts item as second element into mxs.
  ; mxs must have at least one element.
  ; updates the sentinel senti.
  ; Returns the modified list.
  (when (null? mxs)
    (error "mxs_must_not_be_empty")) ; illegal argument
  (let ((newcons (mcons item
                        (mcdr mxs))))
    (set-mcdr! mxs newcons)
    (set-sentinel-theLength!
      senti (+ 1 (sentinel-theLength senti)))
    (when (eq? (mcadr newcons) senti) ; neues letztes Element
      (set-sentinel-lastCons! senti newcons)))

```

```

        mxs)))
(define (sm-push! mxs item)
; adds a new cons-cell as second element.
; mxs must have at least one element.
; Returns the modified list.
(insert-item-behind-cons! mxs item (mcdr mxs))
mxs
)

(define (sm-pop! mxs)
; removes the mcons-cell behind the sentinel and returns its value.
; the mlist must have at least three Elements:
; the two sentinels and one element to pop.
(when (or
        (null? mxs)
        (null? (mcdr mxs)))
        (null? (mcdr (mcdr mxs))))
(error "mxs_must_have_at_least_three_elements"))

(let ((result (mcadr mxs)))
  (set-mcdr! mxs (mcdr (mcdr mxs))) ; remove the second cons
  (set-sentinel-theLength!
   (theSentinel mxs) (- (sm-length mxs) 1))
  (when (sm-empty? mxs)
    (set-sentinel-lastCons! (theSentinel mxs) mxs)) ; empty list
  result; return the element popped
))

(define (sm-insert-at-back! mxs item)
(insert-item-behind-cons!
 (sentinel-lastCons (theSentinel mxs))
 item (theSentinel mxs))
)

;(insert-item-behind-cons! mxs item senti)
(define (sm-insert-at-pos-senti! mxs item senti pos)
; insert the item into the sm-list mxs
; at the position pos, zero based
(cond ((or (null? (mcddr mxs));mxs is empty
           (zero? pos)))

```

```

        (insert-item-behind-cons! mxs item senti))
(else
  (sm-insert-at-pos-senti! (mcdr mxs) item senti (- pos -))

(define (sm-insert-at-pos! mxs item pos)
  (sm-insert-at-pos-senti! mxs item (theSentinel mxs) pos)
  mxs)

(define (sm-insert-Ranked-senti! mxs item senti lessp?)
  ;mxs mutable list
  (if (or (null? (mcddr mxs)) ;mxs is empty (mlist item))
    (lessp? item (mcadr mxs))); insertion location found
    (insert-item-behind-cons! mxs item senti)
    (sm-insert-Ranked-senti! (mcdr mxs) item senti lessp?)))

(define (sm-insert-Ranked! mxs item lessp?)
  ;mxs mutable list with sentinels
  (sm-insert-Ranked-senti! mxs item (theSentinel mxs) lessp?))
  mxs

(define testml
  (sm-push! (sm-push! (sm-push! (sm-makelist) 1) 2) 3))
)

```

File: queuesClass-module.ss

Eine Klasse von Schlangen und spezialisierte Klassen mit unterschiedlichen Strategien für das Einreihen (enqueue!).

(module queuesClass-module swindle

```

#|
A class of queues that can hold a sequence of items,
subclasses with different enqueue strategies
|#

```

```

(provide
; slots
queue-items set-queue-items!
theRankp theRandomGen

```

```

#|
auto provide:
classes: LIFO-queue FIFO-queue
  RANKED-queue RANDOM-queue
generics: queue enqueue! dequeue!
  head-queue empty-queue?
  enumerate-queue
  reset-queue! rLess?
|#)
  )
(print "loading _queuesClass-module_")
(require
  se3-bib/sim/simBase/sim-utility-base-package
  (all-except scheme/mpair mappend) )

(defclass* queue ()
  (items :reader queue-items
         :writer set-queue-items!
         :initvalue (sm-makelist); mutable list with sentinels
         ;:type mlist?
         :documentation "The_items_in_the_queue"
         )
  :autopred #t ; auto generate predicate queue?
  :printer #t
  :documentation
  "the_top_class_of_the_hierarchy_of_queues"
  )

(defclass* LIFO-queue (queue)
; all slots are inherited
  :autopred #t
  :automaker #t
  :printer #t
  :documentation
  "Queues_with_a_last-in-first-out_strategy"
  )

(defclass* FIFO-queue (queue)
  :documentation
  "Queues_with_a_last-in-first-out_strategy"
  :autopred #t

```

```

:automaker #t
:printer #t
)

(defclass* RANKED-queue (queue)
  (rankp :reader theRankp
         :initarg :rankP
         :type <function>
         :documentation
           "a_procedure_to_compute_the_rank_of_an_item")
  :documentation "Queues that are ordered
according_to_a_rank_of_the_items"
  :autopred #t
  :automaker #t
  :printer #t
)

(defclass* RANDOM-queue (queue)
  (randGen :reader theRandomGen
            :initvalue (lambda (x) (random x))
            :initarg :randGen
            :type <function>
            :documentation
              "a_procedure_to_generate_a_random_position_in_the_queue
; <top>-> real >= 0
")
  :documentation "Queues that are ordered randomly"
  :autopred #t
  :automaker #t
  :printer #t
)

(defgeneric* enqueue! ((qu queue) item)
  ; enqueue!: queue any -> queue
  ; Arg. item: keine Spezialisierung
  ; abstract, should be provided by the subclasses
  :documentation "Push_an_item_onto_the_queue."
)

(defgeneric* dequeue! ((qu queue))
  ; dequeue!: queue -> any
)

```

```

:documentation
"Remove_an_item_from_the_front_of_the_queue.
Return_the_item."
)

(defgeneric* enqueue-items! ((qu queue) (items <list>))
; enqueue!: queue <list> -> queue
:documentation "Push_all_items_of_a_list_onto_the_queue."
)

(defgeneric* head-queue ((qu queue)))
:documentation
"Return_the_item_at_the_front_of_the_queue."
)

(defgeneric* empty-queue? ((qu queue)))
:documentation "Is_the_queue_empty?"
)

(defgeneric* enumerate-queue ((qu queue)))
:documentation
"Return_a_list_of_the_queue-items"
)

(defgeneric* reset-queue! ((qu queue)))
:documentation
"Reset_the_queue,_remove_all_items"
)

(defgeneric* rLess? ((qu queue))
;rLess?: queue -> (<top><top> -> boolean)
:documentation
"Generate_a_less?-predicate_according_to_rankp")
)

```

File: queuesImpl-module.ss

```

(module queuesImpl-module swindle ;
(require

```

```

se3-bib/sim/simBase/sim-utility-base-package
se3-bib/sim/simBase/queuesClass-module
(all-except scheme/mpair mappend) ; use mutable lists
(lib "scheme-tests.ss" "test-engine")
)

(defmethod head-queue ((qu queue))
  "Return_the_item_at_the_front_of_the_queue."
  (sm-first-element (queue-items qu)))

(defmethod empty-queue? ((qu queue))
  "Is_the_queue_empty?"
  (sm-empty? (queue-items qu)))

(defmethod reset-queue! ((qu queue))
  "Reset_the_queue,_remove_all_items"
  (set-queue-items! qu (sm-makelist)))

(defmethod enumerate-queue ((qu queue))
  "Return_a_list_of_the_queue-items"
  (sm->list (queue-items qu)))

(defmethod dequeue! ((qu queue))
  "Remove_an_item_from_the_front
  _of_the_queue._Return_the_item."
  (sm-pop! (queue-items qu)))

(defmethod enqueue-items! ((qu queue) (items <list>))
  "Push_all_items_of_a_list_onto_the_queue."
  (map (curry enqueue! qu) items)
  qu)

(defmethod enqueue! ((qu LIFO-queue) item)
  "Push_an_item_onto_the_queue."
  (sm-push!(queue-items qu) item)
  qu)

(defmethod enqueue! ((qu FIFO-queue) item)
  "Insert_an_item_at_the_tail_of_the_queue."
  (sm-insert-at-back! (queue-items qu) item)
  qu)

```

```

(defmethod rLess? ((qu queue))
  ;rLess?: queue -> (<top><top> -> boolean)
  "Generate_a_less?-predicate_according_to_rankp"
  (lambda (item1 item2)
    (< ((theRankp qu) item1)
      ((theRankp qu) item2)))))

(defmethod enqueue! ((qu RANKED-queue) item)
  "Insert_an_item_according_to_the_rank."
  (sm-insert-Ranked!
    (queue-items qu) item (rLess? qu))
  qu)

(defmethod enqueue! ((qu RANDOM-queue) item)
  "Insert_an_item_at_a_random_Position."
  (sm-insert-at-pos!
    (queue-items qu)
    item
    (random ; to do: (theRandomGen qu)
      (max 1 (sm-length (queue-items qu))))))
  qu)
)

```

File: simPathClass-module.ss

Eine mixin-Klasse für Warteschlangen, Statistik über die mittlere Länge und mittlere Wartezeit. Für jedes Einreihen ist sim-path-up! auszuführen, für jedes Entfernen sim-path-down!.

```

(module simPathClass-module swindle

  (provide
    ;slots
    items-w set-items-w!
    set-wt! wt
    tlu set-tlu!
    ts
    #/ auto provide
    sim-path
    sim-path-up sim-path-down sim-path-mean
  )

```

```

)
#/
=====
mixin class for server stations,
provides statistics on the waiting time in a queue
=====
/#  

(print "loading:_simPathClass-module")  

(require  

  se3-bib/sim/simBase/sim-utility-base-package  

)  

(defclass* sim-path ()  

  (n-items-waiting  

   :accessor items-w  

   :initvalue 0  

   :type <integer>  

   :documentation  

   "The_number_of_items_in_the_associated_queue")  

  (accumulated-waiting-time  

   :accessor wt  

   :initvalue 0  

   :type <number>  

   :documentation  

   "Accumulated_waiting_time_since_the_creation_of_the_queue")  

  (time-of-start  

   :accessor ts  

   :initvalue 0  

   :type <number>  

   :documentation  

   "Creation_time_of_the_queue")  

  (time-of-last-update  

   :type <number>  

   :accessor tlu  

   :initvalue 0)  

  :autopred #t ; auto generate predicate sim-path?  

  :printer #t  

  :documentation  

  "Statistics_on_the_ups_and_downs_of_the_queue_length")

```

```

(defgeneric* sim-path-up! ((sim-p sim-path))
  :documentation
  "Update_the_statistics_for_a_new_arrival_in_the_queue"
  :combination generic-begin-combination)

(defgeneric* sim-path-down! ((sim-p sim-path))
  :documentation
  "Update_the_statistics_for_a_departure_from_the_queue"
  :combination generic-begin-combination)

(defgeneric* path-info ((sim-p sim-path))
  :documentation
  "request_state_information_from_path_of_a_queue."
  )

(defgeneric* sim-path-mean ((sim-p sim-path))
  :documentation
  "The_average_waiting_time_of_the_associated_queue"
  )

```

File: simPathImpl-module.ss

```

(module simPathImpl-module swindle

  (print "loading:_simPathImpl-module")
  (require
    se3-bib/sim/simBase/sim-utility-base-package
    se3-bib/sim/simBase/simClockClass-module
    se3-bib/sim/simBase/simClockImpl-module
    se3-bib/sim/simBase/simPathClass-module)

  (defmethod sim-path-up! ((sim-p sim-path))
    "Update_the_statistics_for_a_new_arrival_in_the_queue"
    (inc! (wt sim-p) ;update accumulated waiting time
          (* (items-w sim-p)
             (- (now) (tlu sim-p))))
    (inc! (items-w sim-p))
    (setf! (tlu sim-p) (now)) )

```

```

(defmethod sim-path-down! ((sim-p sim-path))
  "Update_the_statistics_for_a_departure_from_the_queue"
  (inc! (wt sim-p) ;update accumulated waiting time
        (* (items-w sim-p)
           (- (now) (tlu sim-p))))
  (dec! (items-w sim-p))
  (setf! (tlu sim-p) (now)) )

(defmethod sim-path-mean ((sim-p sim-path))
  "The_average_waiting_time_of_the_associated_queue"
  (let ((delta-t (- (now) (ts sim-p))))
    (if (> delta-t 0)
      (begin
        (inc! (wt sim-p) ;update accumulated waiting time
              (* (items-w sim-p)
                 (- (now) (tlu sim-p))))
        (setf! (tlu sim-p) (now))
        (/ (wt sim-p) (- (now) (ts sim-p)))
        0)))
    0))

(defmethod path-info ((sim-p sim-path)); progn
  "request_state_information_from_path_of_a_queue."
  (writeln "_The_average_waiting_time:_"
          (sim-path-mean sim-p))
  (writeln "Number_of_clients_still_waiting:_"
          (items-w sim-p)))
)

```

File: simScenarioClass-module.ss

The controller class, interface to world.ss, to use implement: actor-picture, createTheEventSources and call simulate.

```

(module simScenarioClass-module swindle

  (provide scene-description)

  (print "loading:_simScenarioClass-module_")

```

```

(require
  se3-bib/sim/simBase/sim-utility-base-package
  se3-bib/sim/simBase/simActorClass-module
  se3-bib/sim/simBase/simActorViewClass-module
  se3-bib/sim/simBase/simClockClass-module
  se3-bib/sim/simBase/simEventClass-module
  se3-bib/sim/simBase/simCalendarClass-module
)

(defclass* sim-scenario
  (sim-clock sim-calendar sim-event sim-actor-view)
  (actorPicture
    :initvalue  )
  (domain
    :reader scene-description
    :initvalue "unknown_universe"
    :initarg :scene-description
    :documentation "a_string_that_describes_the_domain"
    :allocation :class
    :type <string>)
  ; in this implementation there is only one universe
  :documentation
  "A_simulation_universe_with_a_clock_and_a_calendar_of_events"
  :autopred #t ; auto generate predicate sim-scenario?
  :printer #t )
  ; The sim-scenario doubles as the start-up event.
  ; Every application must provide a handler for this event.
  ; This handler must schedule an event that kicks off
  ; the simulation process

  ;(define *current-universe* #f)

(defgeneric* simulate
  ((universe sim-scenario)
   &key (canvas-w 500) (canvas-h 500)
   (tick 1) (dt 100))
  :documentation
  "Simulate_the_scenario_for_an_interval_of_dt_time_units"
  ; tick: clock ticks in seconds,
  ; dt: simulation time in seconds

```

```

)
(defgeneric* drawWorld
  ((universe sim-scenario) )
  :documentation
  "draw_the_world_onto_the_canvas")

(defgeneric* createTheEventSources ((universe sim-scenario))
  :documentation
  "create_all_the_active_scenario_components")
; abstract, must be specialized by the application

(defgeneric* snap-shot((universe sim-scenario))
  :documentation
  "Show_the_similation_time,_the_calendar_of_events,
and_the_state_of_all_actors.")

(defgeneric* theNextworld ((universe sim-scenario))
  :documentation
  "The_changed_world_after_the_current_event")

(defclass* sim-Snapshot (sim-event)
  :autopred #t
  :printer #t
  :documentation
  "display_the_state_of_the_simulation"
  )  )

```

File: simScenarioImpl-module.ss

```

(module simScenarioImpl-module
  swindle

(provide
  verbose silent stepping unstep *stepping*
  *current-universe*)

(print "loading:_simScenarioImpl-module_")

(require

```

```

se3-bib/sim/simBase/sim-utility-base-package
se3-bib/sim/simBase/simActorClass-module
se3-bib/sim/simBase/simActorViewClass-module
se3-bib/sim/simBase/simActorViewImpl-module
se3-bib/sim/simBase/simActorImpl-module
se3-bib/sim/simBase/simClockClass-module
se3-bib/sim/simBase/simClockImpl-module
se3-bib/sim/simBase/simEventClass-module
se3-bib/sim/simBase/simEventSourceClass-module
se3-bib/sim/simBase/simEventImpl-module
se3-bib/sim/simBase/simCalendarClass-module
se3-bib/sim/simBase/simCalendarImpl-module
se3-bib/sim/simBase/simScenarioClass-module)

(define *current-universe* #f)

; the constructor
(defmethod initialize :after
  ((universe sim-scenario) initargs)
  (setf! *current-universe* universe))

(defmethod snap-shot((universe sim-scenario))
  "Show_the_similation_time,_the_calendar_of_events,
and_the_state_of_all_actors."
  (let ((oldProto (scene-protocol *current-universe*)))
    (set-scene-protocol! universe #t)
    (broadcast universe sim-info)
    (set-scene-protocol! universe oldProto)
    #t))

(defmethod sim-init! ((universe sim-scenario)) ;progn
  "Reset_the_scenario_for_a_new_simulation_run"
  (broadcast *current-universe*
             sim-init!
             :sim-class sim-event-source) #t )

(defmethod last-world? ((universe sim-scenario))
  (if (sim-Quit?
        (peak-next-event *current-universe*))
      #t #f))

```

```

(defmethod sim-start ((universe sim-scenario))
  "Trigger the start-up actions, get the system going"
  ; initialize all component
  (on-key-event handle-key)
  (on-tick-event theNextworld)
  (on-redraw drawWorld)
  (stop-when last-world?)
  #t )

(defmethod sim-info ((universe sim-scenario))
  ; (clock-info universe)
  ; (calendar-info universe)
  (writeln "Simulation_scenario:"
           (scene-description universe))
  #t )

(defmethod theNextworld ((universe sim-scenario))
  (when *stepping*
    (writeln "break:_return_for_continue" )
    (read))
  (dispatch universe)
  universe)

(define *stepping* #f)

(define (stepping) (set! *stepping* #t))

(define (unstep) (set! *stepping* #f))

(define (handle-key world key)
  (case key
    ((#\s) (snap-shot))
    ((#\q) (schedule (make sim-Quit) (now)))
    (else (display "."))
    )
  world)

(defmethod handle ((event sim-Quit))
  (set-event-message! event "end_of_time,_game_over")
  (end-of-time "end_of_time,_game_over"))

```

```

(defmethod handle ((event sim-Snapshot))
  (set-event-message! event "cheese:_snapshot")
  (snapshot *current-universe*))

(define (verbose)
  "Print_out_each_event"
  (set-scene-protocol! *current-universe* #t))

(define (silent)
  "Turn_off_printing_of_events"
  (set-scene-protocol! *current-universe* #f))

(defmethod handle ((event sim-scenario))
  "activate_the_other_even_sources"
  (broadcast *current-universe*
             sim-start
             :sim-class sim-event-source)
  ; schedule the start up events
  )

(defmethod createTheEventSources
  ((universe sim-scenario))
  #t )

(defmethod simulate
  ((universe sim-scenario)
   &key (canvas-w 500) (canvas-h 500)
   (tick 1) (dt 100))
  "Simulate_the_scenario_for_an_interval_of_dt_time_units"
  ; tick: clock ticks in seconds,
  ; dt: simulation time in seconds
  (createTheEventSources
   *current-universe*)
  (sim-init! *current-universe*)
  (schedule (make sim-nothingHappens) (now))
  (schedule universe (now))
  (schedule (make sim-Quit) dt)

  (big-bang
   canvas-w canvas-h
   tick

```

```

    *current-universe*)
(sim-start *current-universe*)
)

(defmethod drawWorld
((universe sim-scenario) )
"draw_the_world_onto_the_canvas"
(sim-info universe)
(actor-picture universe)
))

```

30.3 Das Anwendungssystem: Bediensystem

Dieses Modul bündelt die Module des Anwendungssystems (Bedienstationen, Kunden, Tür) und des Basissystems zu einem Paket, das alle definierten Namen exportiert.

File: sim-application-package.ss

```

(module sim-application-package
swindle

(print "loading:_sim-application-package_")
(require
se3-bib/sim/simBase/sim-base-package

se3-bib/sim/simAppl/simDoorClass-module
se3-bib/sim/simAppl/simDoorImpl-module
se3-bib/sim/simAppl/simDoorView-module

se3-bib/sim/simAppl/simServerStationsClass-module
se3-bib/sim/simAppl/simServerStationsImpl-module
se3-bib/sim/simAppl/simServerStationsViewClass-module
se3-bib/sim/simAppl/simServerStationsViewImpl-module

se3-bib/sim/simAppl/simCustomerClass-module
se3-bib/sim/simAppl/simCustomerImpl-module
se3-bib/sim/simAppl/simCustomerView-module

se3-bib/sim/simAppl/customerServerScenarioClass-module
se3-bib/sim/simAppl/customerServerScenarioCont-module

```

```

se3-bib/sim/simAppl/customerServerScenarioView-module)

(provide
  (all-from se3-bib/sim/simBase/sim-base-package)

  (all-from se3-bib/sim/simAppl/simDoorClass-module)
  (all-from se3-bib/sim/simAppl/simDoorImpl-module)
  (all-from se3-bib/sim/simAppl/simDoorView-module)

  (all-from se3-bib/sim/simAppl/simServerStationsClass-module)
  (all-from se3-bib/sim/simAppl/simServerStationsImpl-module)
  (all-from se3-bib/sim/simAppl/simServerStationsViewClass-module)
  (all-from se3-bib/sim/simAppl/simServerStationsViewImpl-module)

  (all-from se3-bib/sim/simAppl/simCustomerClass-module)
  (all-from se3-bib/sim/simAppl/simCustomerImpl-module)
  (all-from se3-bib/sim/simAppl/simCustomerView-module)

  (all-from se3-bib/sim/simAppl/customerServerScenarioClass-module)
  (all-from se3-bib/sim/simAppl/customerServerScenarioView-module)
  (all-from se3-bib/sim/simAppl/customerServerScenarioCont-module)
)

```

File: simCustomerClass-module.ss

Die Klasse der Kunden

```

(module simCustomerClass-module swindle

(require
  se3-bib/sim/simBase/sim-base-package)

(defclass* customer (sim-actor sim-event sim-actor-view)
  :autopred #t ; auto generate predicate queue?
  :printer #t
  :documentation
  "the_customers"
)

```

File: simCustomerImpl-module.ss

```

(module simCustomerImpl-module swindle

(print "loading:_simCustomerImpl-module_")
(require
  se3-bib/sim/simBase/sim-base-package

  se3-bib/sim/simAppl/simDoorClass-module
  se3-bib/sim/simAppl/simDoorImpl-module
  se3-bib/sim/simAppl/simServerStationsClass-module
  se3-bib/sim/simAppl/simServerStationsImpl-module
  se3-bib/sim/simAppl/simCustomerClass-module)

(defmethod handle ((c customer))
  "handle_a_customer_event:_pick_a_service_station ,
  enqueue_the_customer_at_the_station , _close_the_door."
  ;; sofar the customers visit only one station
  ;; no dispatch to next station is necessary

  (let ((station (pickARandomStation)))
    (if (not station)
        (begin
          (writeln "no_station_available")
          (send-backstage c))
        (request station c)))
    (close-the-door *the-sim-door)))

(defmethod sim-init! ((c customer))
  "go_to_the_waiting_line_behind_the_door
  and_wait_for_an_appearance"
  (send-backstage c)
  )
)

```

File: simCustomerView-module.ss

```

(module simCustomerView-module swindle

(require
  se3-bib/sim/simBase/sim-base-package
  se3-bib/sim/simAppl/simCustomerClass-module
  se3-bib/sim/simAppl/simCustomerImpl-module)

```

```
(defmethod actor-picture
  ((c customer)) ;
  (pictureWithName c))
)
```

File: simDoorClass-module.ss

Die Klasse der Simulationstür: Eine Ereignisquelle, die zufällig Kunden auswählt und auf die Bühne schickt. Die Simulation wird gestartet, indem ein simDoor-Objekt im Simulationskalender eingetragen wird.

```
(module simDoorClass-module swindle

  (provide theActorInTheDoor
            set-ActorInTheDoor!
            )
  (print "loading:_simDoorClass-module")
  (require
    se3-bib/sim/simBase/sim-base-package
    se3-bib/sim/simAppl/simCustomerClass-module
  )

  (defclass* sim-door
    (sim-exp-event-source RANDOM-queue sim-actor-view)
    (actorInTheDoor
      :reader theActorInTheDoor
      :writer set-ActorInTheDoor!
      :initvalue ???
      ; :type sim-actor
      :documentation "The_name_of_the_actor"
    )
    :autopred #t ; auto generate predicate queue?
    :printer #t
    :documentation
    "An_event_source_of_customers_picked_randomly_from_a_queue" )

  (defgeneric* send-backstage ((c customer))
    :documentation "make_the_customer_c_idle")

  (defgeneric* arrival ((a customer))
```

```

:documentation
"an_actor_is_entering_the_stage .")

(defgeneric* departure ((a customer))
:documentation
"an_actor_is_leaving_and_idle_again .")

(defgeneric* close-the-door ((d sim-door))
:documentation
"an_actor_is_closing_the_door_after_entering_the_scene .")

(defgeneric* door-empty? ((d sim-door))
:documentation
"is_there_no_actor_entering_just_now? ")
)

```

File: simDoorImpl-module

```

(module simDoorImpl-module swindle
(provide *the-sim-door*)

(print "loading:_simDoorImpl-module")
(require
  se3-bib/sim/simBase/sim-base-package
  se3-bib/sim/simAppl/simDoorClass-module
  se3-bib/sim/simAppl/simCustomerClass-module
)

(define *the-sim-door* #f)

(defmethod initialize :after
  ((door sim-door) initargs)
  (setf! *the-sim-door* door))

(defmethod arrival ((a customer))
  (let ((theMessage (string-append
                     "Enter:_" (actor-name a))))
    (set-ActorInTheDoor! *the-sim-door* a)
    (set-event-message! *the-sim-door* theMessage)))

```

```

(defmethod close-the-door ((d sim-door))
  "an_actor_is_closing_the_door
   after_entering_the_scene."
  (set-ActorInTheDoor! d ???))

(defmethod send-backstage ((a customer))
  "make_the_actor_a_idle"
  ; recycle actor for the next appearance
  (enqueue! *the-sim-door* a))

(defmethod departure ((a customer))
  "an_actor_is_leaving_and_idle_again"
  (let ((theMessage (string-append
                      "Departure:" (actor-name a))))
    (set-event-message! *the-sim-door* theMessage)
    (send-backstage a)))

(defmethod handle ((door sim-door))
  "handle_a_door_event,_schedule_a_customer_to_appear"

  (unless (empty-queue? door)
    ; dispatch the next customer from the queue
    (let ((nextCustomer (dequeue! door)))
      (arrival nextCustomer)
      (schedule nextCustomer (now)))))

(defmethod sim-init! ((door sim-door))
  "tell_the_customers_that_the_door_is_ready,
   call_the_customers_backstage"
  (broadcast door
             sim-init!
             :sim-class customer))

(defmethod sim-start ((door sim-door))
  "the_customers_are_ready_to_go,
   schedule_the_first_door_event"
  (next-event door))

(defmethod door-empty? ((d sim-door))
  "is_there_no_actor_entering_just_now?"
  (not (slot-bound? d 'actorInTheDoor)))

```

)

File: simDoorView-module.ss

```
(module simDoorView-module swindle

  (provide show-empty-door show-person-in-door)

  (require
    se3-bib/sim/simBase/sim-base-package

    se3-bib/sim/simAppl/simDoorClass-module
    se3-bib/sim/simAppl/simDoorImpl-module

    se3-bib/sim/simAppl/simCustomerClass-module
    se3-bib/sim/simAppl/simCustomerImpl-module
    se3-bib/sim/simAppl/simCustomerView-module )

  (define *doorWidth* 80)
  (define *doorHeight* 80)

  (defmethod show-empty-door ((d sim-door))
    (let ((thelm
           (overlay
             (rectangle
               *doorWidth*
               *doorHeight*
               'solid
               'blue)
             (rectangle
               (- *doorWidth* 16)
               (- *doorHeight* 16)
               'solid
               'black))))
      (set-actor-pic! d thelm)
      thelm))

  (defmethod show-person-in-door ((d sim-door))
    (let ((thelm
           (if (door-empty? d)
               (show-empty-door d)
```

```

(overlay
  (show-empty-door d)
  (actor-picture (theActorInTheDoor d))))))
(set-actor-pic! d thelm)
thelm))

(defmethod actor-picture
  ((d sim-door))
  (show-person-in-door d))
)

```

File: simServerStationsClass-module.ss

Die Klasse der Bedienstationen: Verkaufsstände mit Warteschlangen und Statistik:

```

(module simServerStationsClass-module
  swindle

  (provide
    station-service
    *all-servers*  )

  (require
    se3-bib/sim/simBase/sim-base-package
    se3-bib/sim/simAppl/simServerStationsViewClass-module
    se3-bib/sim/simAppl/simCustomerClass-module
  )

  (define *all-servers* (make setOfActors))
  ; A set of all server stations, set by initialize

  (defclass* server-station
    (sim-normal-event-source FIFO-queue sim-path
      server-station-view)
    (service
      :reader station-service
      :initvalue "Super_Market"
      :initarg :station-service
      :type <string>)

```

```

:autopred #t ; auto generate predicate queue?
:printer #t
:documentation
"A_FIFO-queue_with_statistics
on_the_ups_and_downs_of_the_queue_length")

(defgeneric* serve ((s server-station)(c customer))
:documentation "serve_customer_c_at_station_s" )

(defgeneric* request ((s server-station)(c customer))
:documentation "customer_c_is_requesting_service
at_station_s" )

(defgeneric* reject
((server-station)(c customer))
:documentation "do_not_serve_customer_c_at_station_s")

(defgeneric* is-idle? ((s server-station))
:documentation "is_the_station_idle?" )

(defgeneric* is-serving? ((s server-station))
:documentation "is_the_station_serving_somebody_right_now?" )

(defgeneric* is-rejecting? ((s server-station))
:documentation "is_the_station_rejecting_somebody_right_now?" )

(defgeneric* rejecting? ((rs server-station) (c customer))
:documentation "will_customer_c_be_served_at_station_s?")
)

```

File: simServerStationsImpl-module.ss

```

(module simServerStationsImpl-module swindle

(provide pickARandomStation quickest-rate)

(require
se3-bib/sim/simBase/sim-base-package

```

```

se3-bib/sim/simAppl/simDoorClass-module
se3-bib/sim/simAppl/simDoorImpl-module
se3-bib/sim/simAppl/simServerStationsClass-module
se3-bib/sim/simAppl/simServerStationsViewClass-module
se3-bib/sim/simAppl/simCustomerClass-module
)

(define (pickARandomStation)
; customers pick a random service station
(if (null? (theActors *all-servers*))
#f
(random-elt
(theActors *all-servers*)))

(defmethod serve
((s server-station) (c customer))
"Perform_service_for_a_customer
at_a_server_station"
; nothing to do but to schedule the end of service event
(let* ((sName (actor-name s))
(cName (actor-name c))
(theM (string-append
sName "_is_serving:_" cName)))
(set-event-message! s theM))

(defmethod is-idle? ((s server-station))
"is_the_station_idle?"
(empty-queue? s))

(defmethod rejecting? ((rs server-station) (c customer))
#f ; the default, serve all customers

(defmethod is-rejecting? ((s server-station))
"is_the_station_serving_somebody_right_now?"
(and (not (is-idle? s))
(rejecting? s (head-queue s)))))

(defmethod is-serving? ((s server-station))
"is_the_station_serving_somebody_right_now?"
(and (not (is-idle? s)))

```

```

(not (is-rejecting? s)))))

(defmethod next-event ((rStation server-station))
  "Schedule the end-of-service-event"
  (writeln "serverStation:_next_event");-----
  (unless (empty-queue? rStation)
    (if (rejecting? rStation (head-queue rStation))
        (next-now-event rStation) ; send away immediately
        (next-normal-event rStation ; take time to serve
                           :sigma (event-sig rStation)))))

(defmethod request ((s server-station)(c customer))
  "customer_c_is_requesting_service_at_station_s"
  (let* ((sName (actor-name s))
         (cName (actor-name c))
         (theM (string-append
                 cName
                 "_is_requesting_service_at_station_"
                 sName)))
    (idle (is-idle? s)))
  (enqueue! s c)
  (when idle
    (writeln "first_request"); -----
    (serve s c); serve immediately, if queue is empty
    (next-event s)); schedule the departure event
  (set-event-message! s theM)))

(defmethod reject
  ((rs server-station) (rc customer))
  "Reject_a_customer_not_entitled_to_the_service"
  ; nothing to do but to schedule the end of service event
  (let* ((sName (actor-name rs))
         (cName (actor-name rc))
         (theM (string-append
                 (string-append sName
                               "_is_rejecting:_"
                               cName)))))

  (set-event-message! rs theM)
))

(defmethod handle ((station server-station))

```

```

"handle_a_free-station_event:_customer_finished ,
  send_the_customer_backstage ,
  start_to_serve_the_next_customer"
    ; sofar the customers visit only one station
    ; no dispatch to next station is necessary
  (writeln "handle:_station"); ---
  (if (empty-queue? station)
      (writeln "warning:_nobody_waiting_at_"
              (actor-name station)))
  (begin
    (departure (dequeue! station))
    ; recycle actor for the next appearance
    (if (not (empty-queue? station))
        ;serve the next customer, if somebody is waiting
        (serve station (head-queue station)))))

(defmethod enqueue! :before ((qu server-station) item)
  "Update_the_path_statistics_before_entering_the_queue"
  (sim-path-up! qu))

(defmethod dequeue! :before ((qu server-station))
  "Update_the_path_statistics_before_leaving_the_queue"
  (sim-path-down! qu))

(defmethod sim-init! ((station server-station))
  "Initialize_the_server_station_for_the_first_simulation_run"
  (reset-queue! station)
  )

(defmethod sim-info ((station server-station))
  "request_state_information_from_the_station."
  (writeln "Station:_" (station-service station))
  (path-info station))

(defmethod initialize :after
  ((station server-station) args)
  ; maintaining the list of server stations;
  ; applied after the main initialization
  (add-actor! station *all-servers*))

(define (quickest-rate)

```

```

; the event rate of the quickest server
(let ((theRates
      (map event-rate
            (theActors *all-servers*))))
(if (null? theRates) 1.0
    (apply max theRates)
    )))
)
```

File: simServerStationsViewClass-module.ss

Create images of the server stations: A picture of the clerk, a billboard announcing the goods, a picture of the waiting line

```

(module simServerStationsViewClass-module
  swindle

  (provide
   *defaultServingpic* *defaultIdlePic* *defaultRejectPic*
   idle-pic serving-pic reject-pic
   set-idle-pic! set-serving-pic! set-reject-pic! )

  (require se3-bib/sim/simBase/sim-base-package)

  (define *defaultServingpic*  )
  (define *defaultIdlePic*  )
  (define *defaultRejectPic*  )

  (defclass* server-station-view
    (sim-actor-view)
    (actorPicture
     :initvalue *defaultIdlePic*)
    (idlePicture
     :reader idle-pic
     :writer set-idle-pic!
     :initarg :idlePic
     :initvalue *defaultIdlePic*
     :documentation
     "A_picture_of_an_idle_server")
    (servingPicture
```

```

:reader serving-pic
:writer set-serving-pic!
:initarg :servingPic
:initvalue *defaultServingpic*
:documentation
"A_picture_of_the_current_state_of_the_actor" )
(rejectPicture
:reader reject-pic
:writer set-reject-pic!
:initarg :rejectPic
:initvalue *defaultRejectPic*
:documentation
"A_picture_of_the_current_state_of_the_actor")
:autopred #t ; auto generate predicate queue?
:printer #t
:documentation
"The_View_of_a_server_station")

(defgeneric* picture-of-clerk ((s server-station))
:documentation "a_picture_of_the_state_of_the_clerk")
(defgeneric* billboard ((s server-station))
:documentation
"an_advertisement_of_the_station_service")
)

```

File: simServerStationsViewImpl-module.ss

```

(module simServerStationsViewImpl-module
swindle
  (provide allStationsPicture)

(require
  se3-bib/sim/simBase/sim-base-package
  se3-bib/sim/simAppl/simServerStationsClass-module
  se3-bib/sim/simAppl/simServerStationsImpl-module
  se3-bib/sim/simAppl/simServerStationsViewClass-module
  se3-bib/sim/simAppl/simCustomerClass-module
  se3-bib/sim/simAppl/simCustomerImpl-module
  se3-bib/sim/simAppl/simCustomerView-module
)

```

```

(defmethod picture-of-clerk ((s server-station))
  :documentation "a_picture_of_the_state_of_the_clerk"
  (let ((clerk-pic
         (cond ((is-idle? s)
                (idle-pic s))
               ((is-serving? s)
                (serving-pic s))
               ((is-rejecting? s)
                (reject-pic s))
               (else *defaultServingpic))))
    (set-actor-pic! s clerk-pic)
    clerk-pic))

(defmethod sim-init! ((sv server-station-view))
  (picture-of-clerk sv))

(provide sim-init!)

(defmethod billboard ((station server-station))
  ; an advertisement of the station service:
  ; a board with the service written on it
  (let* ((longest (text "Magic_Potion_" 18 'blue))
         (servicePic
          (text
           (cjustify 13 (station-service station)) 18 'blue))
         (h (image-height servicePic))
         (w (max (image-width servicePic)
                  (image-width longest)))
         (bg (rectangle w (* 2 h) 'solid 'gray))
         (picOnBG
          (overlay/xy
           bg
           (- (pinhole-x servicePic)
              (pinhole-x bg))
           (- (pinhole-y servicePic)
              (pinhole-y bg))
           servicePic)))
         picOnBG))

(define verticalGap 16) ; 10 Pixel space between the lines

```

```

(define (ovLines linePics bg posY)
; a utility used by allStationsPicture
; linePics: a list of the pictures of all waiting lines
; bg: a background image
; posY: the line number of the top image
(cond ((null? linePics) bg)
(else
(let ((thisIm (car linePics)))
(ovLines
(cdr linePics)
(overlay/xy
bg
(- (pinhole-x thisIm) (pinhole-x bg))
(+ posY (- (pinhole-y thisIm) (pinhole-y bg)))
thisIm)
(+ posY (image-height thisIm) verticalGap))))))
; construct a picture of all waiting lines:
; left: the Server, left to right: the waiting line

(define (allStationsPicture)
(let* ((picsOfTheLines
(map actor-picture
(theActors *all-servers*)))
(maxWidth
(apply max (map image-width picsOfTheLines)))
(totalHeight
(+ (apply + (map image-height picsOfTheLines))
(* (length (theActors *all-servers*))
verticalGap)))
(bg (rectangle maxWidth totalHeight 'solid 'white)))
(ovLines picsOfTheLines bg 0)
))

(define (ovlImages actorPics bg posX)
; ovlImages: list of actors, Image, int -> Image
; the items waiting, a background image, a x-displacement
(cond ((null? actorPics) bg)
(else
(let ((thisIm (car actorPics)))

```

```

(ovlImages
  (cdr actorPics)
  (overlay/xy
    bg
    (+ posX (- (pinhole-x thisIm) (pinhole-x bg)))
    0
    thisIm)
  (+ posX (image-width thisIm))))))
)

(defmethod
  actor-picture
  ; an image of the queue
  ((station server-station))
  (let* ((clerk (picture-of-clerk station))
         (sPic (pictureWithName station)); the server
         (label (billboard station))
         (thePictures
           (cons label
                 (cons sPic
                       (map actor-picture
                            (enumerate-queue station))))))
         (totalWdth
           (apply + (map image-width thePictures)))
         (maxHeight
           (apply max (map image-height thePictures)))
         (bg (rectangle totalWdth maxHeight 'solid 'white)))
         )
    (ovlImages thePictures bg 0)))
)

```

File: customerServerScenarioClass-module.ss

Definiert ein Szenario mit Bedienstationen und Kunden, sowie generische Funktionen zur Implementation der Stationen und Kunden.

```

(module customerServerScenarioClass-module
  swindle

(require
  se3-bib/sim/simBase/sim-base-package
  se3-bib/sim/simAppl/simDoorClass-module
  se3-bib/sim/simAppl/simServerStationsClass-module

```

```

se3–bib/sim/simAppl/simCustomerClass–module)

(defclass* customer-server-scenario
  (sim-scenario)
  :documentation
  "A_world_with_server_stations_and_customers"
  :autopred #t ; auto generate predicate sim-scenario?
  :printer #t )

(defgeneric* run ((world customer-server-scenario)
                  &key (tick 1) (dt 100))
  :documentation "a_default_simulation_run" )

(defgeneric* make-the-customers
  ((world customer-server-scenario))
  :documentation "construct_the_eager_customers"
  ) ; abstract

(defgeneric* make-the-door
  ((world customer-server-scenario))
  :documentation "construct_the_source_of_customers"
  ) ; abstract

(defgeneric* make-the-servers
  ((world customer-server-scenario))
  :documentation "construct_the_friendly_attendants"
  ) ; abstract

)

```

File: customerServerScenarioCont-module.ss

Implementation des Controlers customer-server-scenario, implementiert die abstrakte Operation create-the-event-sources aus sim-scenario.

```

(module customerServerScenarioCont–module
  swindle

(require
  se3–bib/sim/simBase/sim–base–package
  se3–bib/sim/simAppl/simDoorClass–module

```

```

se3-bib/sim/simAppl/simServerStationsClass-module
se3-bib/sim/simAppl/simCustomerClass-module
se3-bib/sim/simAppl/customerServerScenarioClass-module
se3-bib/sim/simAppl/simDoorImpl-module
se3-bib/sim/simAppl/simServerStationsImpl-module
)

(defmethod make-the-servers
  ((world customer-server-scenario))
  (writeln "make-the-servers_abstract"))

(defmethod make-the-door
  ((world customer-server-scenario))
  "construct_the_source_of_customers"
  (make sim-door
    :actorName "Main_Entrance"
    :rate (* 3.0 (quickest-rate)))
  );produce events faster than the quickest server

(defmethod make-the-customers
  ((world customer-server-scenario))
  (writeln "make-the-customers_abstract"))

(defmethod
  createTheEventSources
  ((world customer-server-scenario))
  "create_the_server_stations_and_customers"

  (make-the-servers world)

  (make-the-door world)
;produce events faster than the quickest server

  (make-the-customers world))

(defmethod run ((world customer-server-scenario)
               &key (tick 1) )
  (simulate
    world
    :canvas-w *canv-width* :canvas-h *canv-height*
    :tick 1 :dt 100))

```

)

30.4 Das Mensa-Szenario

File: mensa-scenario-package

Dieses Modul bündelt alle Module des Mensa-Szenarios zu einen Paket und exportiert die definierten Namen.

```
(module mensa-scenario-package swindle

(require
  se3-bib/sim/simAppl/sim-application-package
  se3-bib/sim/simMensa/mensaScenarioClass-module
  se3-bib/sim/simMensa/mensaScenarioImpl-module)

(provide
  (all-from se3-bib/sim/simAppl/sim-application-package)
  (all-from se3-bib/sim/simMensa/mensaScenarioClass-module)
  (all-from se3-bib/sim/simMensa/mensaScenarioImpl-module)
))
```

File: mensaScenarioClass-module

```
(module mensaScenarioClass-module
  swindle

(require
  se3-bib/sim/simAppl/sim-application-package)

(defclass* mensa-scenario
  (customer-server-scenario)
  :documentation
  "Eine_Mensa_mit_Essenausgabe"
  :autopred #t ; auto generate predicate sim-scenario?
  :printer #t
) )
```

File: mensaScenarioImpl-module

```

(module mensaScenarioImpl-module swindle
  (provide mensaDemo1))

(require
  se3-bib/sim/simAppl/sim-application-package
  se3-bib/sim/simMensa/mensaScenarioClass-module)

(define (makeStation nam rte serv)
  (make server-station
    :actorName nam
    :actorPic :actorPic 
    :rate rte
    :station-service serv))

(defmethod make-the-servers ((mensa mensa-scenario))
  (makeStation "Essen-1" 2 "Eintopf")
  (makeStation "Essen-2" 2.3 "Pizza")
  (makeStation "Essen-3" 2.3 "Currywurst")
  (makeStation "Essen-4" 0.5 "Corn_Dogs") )

(defmethod make-the-customers ((mensa mensa-scenario))
  (let ((someStudents
        '("Harry" "Susie" "Sally" "Paula" "Anja"
          "Ernie" "Bert" "Hermione" "Arnold"
          "Paris" "Maja" "Heidi" "Anton"
          "Kunigunde" "Erwin" "Otto" "Hilde"
          "Christoph" "Wolfgang" "Christiane"
          "Ronald" "Ingbert" "Karin" "Helmut"
          "Sascha" "Dominique" "Donald" "Daisy"
          "Prudence" "Ansgar" "Ottolie" "Siegfried")))
    (map (lambda (c)
           (make customer
             :actorName c))
      someStudents )))

(define (mensaDemo1)
  (run (make mensa-scenario
    :actorName "StEllingen" ))) )

```

File: mensa-scenario2-package

Ein Simulationspaket für eine Simulation mit spezialisierten Essensausgaben für Vegetarier und Nicht-Vegetarier.

```
(module mensa-scenario2-package swindle

  (require
    se3-bib/sim/simMensa/mensa-scenario-package

    se3-bib/sim/simMensa/mensaScenario2Class-module

    se3-bib/sim/simMensa/mensaScenario2Impl-module
  )
  (provide
    (all-from se3-bib/sim/simMensa/mensa-scenario-package)
    (all-from se3-bib/sim/simMensa/mensaScenario2Class-module)
    (all-from se3-bib/sim/simMensa/mensaScenario2Impl-module))
)
```

File: mensaScenario2Class-module

Spezialisierte Simulationskomponenten: Vegetarier, Essensausgaben mit vegetarischem Essen.

```
(module mensaScenario2Class-module swindle

  (provide reject-pic set-reject-pic! )
  (require
    se3-bib/sim/simAppl/sim-application-package
    se3-bib/sim/simMensa/mensaScenarioClass-module )

  (defclass* mensa-scenario2
    (mensa-scenario)
    :documentation
    "A_mensa_with_unfriendly_servers"
    :autopred #t ; auto generate predicate sim-scenario?
    :printer #t )

  (defclass* general-food-server (server-station)
    :documentation
    "A_station_with_restricted_access:
```

```

    „serving_menus_with_meat”)

(defclass* vegetarian-food-server (server-station)
 :documentation
 "A_station_with_restricted_acces:
  „serving_menus_without_meat”)

(defclass* vegetarian (customer)
 :documentation
 "Guests_who_don't_like_meat")
)

```

File: mensaScenario2Impl-module.ss

```

(module menaScenario2Impl-module swindle
 (provide mensa2Demo)

(require
 se3-bib/sim/simMensa/mensa-scenario-package
 se3-bib/sim/simMensa/mensaScenario2Class-module)

(defmethod has-picture? ((obj vegetarian))
 "does_the_actor_have_an_unique_picture?"
 (not (member (actor-pic obj)
 (list *defaultpic* *defaultRCpic* ))))

(defmethod veggie ((s server-station))
 (if (member
 (station-service s)
 '("Corn_Dogs" "Pizza"))
 (change-class!
 s vegetarian-food-server)))

(defmethod meat ((s server-station))
 (if (member
 (station-service s)
 '("Currywurst" "Eintopf"))
 (change-class!
 s general-food-server)))

```

```

(defmethod make-the-servers :after
  ((mensa mensa-scenario2))
  (broadcast mensa veggie
             :sim-class server-station)
  (broadcast mensa meat
             :sim-class server-station)
  (remove-duplicates! *all-servers*))

(defmethod rejecting? ((s general-food-server)
                      (c vegetarian))
#t); the default, serve all customers

(defmethod serve
  ((s general-food-server)
   (c vegetarian))
  (reject s c))

(defmethod make-the-customers :after
  ((mensa mensa-scenario2))

  (let ((someVegetarians
         '("Demeter" "Flora" "WurzelSepp" "Waldfee" "Donald"
           "Rapunzel" "Rosemarie" "Tinkerbell" "Winnie"
           "Kräuterhexe" "Persephone" "MrClou" "Fleur"
           "Artemis" "Diana" "Hera" "Sonja"
           "Adonis" "Ganymed")))
    (map (lambda (c)
            (make vegetarian
                  :actorName c
                  :actorPic *defaultRCpic* ;.
                  )))
      someVegetarians )))

(define (mensa2Demo)
  (let ((dieMensa
         (make mensa-scenario2 :actorName "StEllingen")))
    (run dieMensa)))
)

```

Teil XI

Metaprogrammierung und Programmierstile

31 Metaprogrammierung und Spracherweiterung

31.1 Programmieren mit Macros

31.1.1 Spracherweiterung durch Macros

Metaprogrammierung und Programmierstile



- 28 Metaprogrammierung und Spracherweiterung
 - Programmieren mit Macros
 - Spracherweiterung durch Macros
 - Imperative Kontrollstrukturen in Racket
 - Macros zur Effizienzsteigerung
 - Fallstricke der Macroprogrammierung
- 29 Stromorientierte Programmierung
- 30 Programmieren mit Zuständen

☞ Anmerkung: Racket ist eine berechnungsuniverselle Programmiersprache, auch wenn wir nur funktionale Sprachelemente benutzen und referentiell transparent programmieren.

In bestimmten Situationen ist es allerdings effizienter, imperativ zu programmieren und Modifikatoren zu benutzen: Beispielsweise kann es günstiger sein, Datenstrukturen zu modifizieren, statt modifizierte Kopien zu erzeugen, falls die Datenstrukturen sehr groß sind. Wir werden in diesem Kapitel einige Beispiele sehen, bei denen aus Effizienzgründen imperative Sprachelemente benutzt werden, siehe Memo-Funktionen.[Unterabschnitt 33.1](#)

Ein weiterer Grund für Spracherweiterungen ist der Wunsch, ein Programm möglichst problemadäquat formulieren zu können. Wir werden am Beispiel der stromorientierten Programmierung ein neues Verarbeitungsmodell in Racket integrieren, so daß wir Datenflüsse und unendlich große Listen in Racket modellieren können.

In DrRacket gibt es zwei Ansätze, um Spracherweiterungen zu definieren: `define-syntax` und `define-macro`. Beide Ansätze definieren neue syntaktische Schlüsselwörter und führen neue *special form expressions* ein, für die ein Syntax-Transformator definiert, wie diese auszuwerten sind. Wir werden hier nur Macros besprechen, da wir für `define-syntax` die sehr aufwendige Mustersprache von racket einführen müssen, auch wenn `define-syntax` der sicherere Weg ist, um Macros zu definieren. Spracherweiterungen sind in den Sprachen der Lisp-Familie sehr leicht zu definieren, da uns der „Reader“ die Syntaxanalyse abnimmt und wir Symbole als syntaktische Schlüsselworte nutzen können. In anderen Programmiersprachen ist dieses sehr viel aufwendiger, siehe beispielsweise ([Parr, 2010](#)).

Macros

Definition: 142 (Macro)

Macros definieren Schablonen, in die die Argumente textuell, unausgewertet, durch einen Macro-Transformer eingesetzt werden.

Erst wenn alle Ersetzungen vorgenommen wurden, wird der resultierende Ausdruck evaluiert (äußere Reduktion).

- Durch Macros können neue Sprachelemente eingeführt werden, die sich mit Funktionen (in applikativer Auswertung) nicht definieren lassen.
- Durch Macros können Berechnungen für konstante Ausdrücke schon zur Übersetzungszeit ausgeführt werden.

Entwurf von Macros

Da Macros Spracherweiterungen darstellen, sollten sie nur wohlüberlegt verwendet werden.

Schritte beim Entwurf eines Macros:

1. Überlegen, ob wir es wirklich brauchen.
2. Die Syntax entwerfen.
3. Den expandierten Code festlegen.
4. Mit **`defmacro`** definieren.

Achtung: Eine Warnung

The first step in writing a macro is to recognize that every time you write one, you are defining a new language that is just like Lisp except for your new macro.

The programmer who thinks that way will rightfully be extremely frugal in defining macros. (Besides, when someone asks, “What did you do today?” it sounds more impressive to say “I defined a new language and wrote a compiler for it” than to say “I just hacked up a couple of macros.”)

Norvig-92

Introducing a macro puts much more memory strain on the reader of your program than does introducing a function, variable or data type, so it should not be taken lightly.

Introduce macros only when there is a clear need, and when the macro fits in well with your existing system.

As C.A.R. Hoare put it, “One thing the language designer should not do is to include untried ideas of his own.”

Norvig-1992

(Norvig, 1992)

Ein einfaches Beispiel: myif

Beispiel: 143

myif soll die folgende Syntax haben:

(*myif* <bedingung> <wenn> <dann>)

und expandieren zu

(**cond** (<bedingung> <wenn>)
 (**else** <dann>))

Die Implementation:

```
(require swindle/setf swindle/misc)
```

```
(defmacro (myif1 bed wenn sonst)  
  (list 'cond (list bed wenn)  
           (list 'else sonst)))
```

Variante 2: Mit quasiquote

Die vielen **list**-Aufrufe machen das Macro schwer lesbar:

```
(defmacro (myif1 bed wenn sonst)
  (list 'cond (list bed wenn)
        (list 'else sonst)))
```

Übersichtlicher wird es mit **quasiquote**:

```
(defmacro (myif2 bed wenn sonst)
  `(#(cond (,bed ,wenn)
            (#(else ,sonst))))
```

Weitere Beispiele

Eine bedingte Anweisungsfolge:

```
(defmacro (my-when test &rest body)
  `(#(if ,test
       (#(begin ,@body)
          #f)))
```

```
(defmacro (my-unless test &rest body)
  `(#(if (#(not ,test))
       (#(begin ,@body)
          #f)))
```

31.1.2 Imperative Kontrollstrukturen in Racket

Eine while-Schleife

while soll die folgende Syntax haben:

```
(while <test> <expr1> ... <exprn>)
```

```
(letrec
  ((loop
    (lambda ()
      (when <test> <expr1> ... <exprn>
        (loop)))))

(defmacro (my-while test &rest body)
  "Repeat_body_while_test_is_true."
  '(letrec
    ((loop
      (lambda ()
        (when ,test ,@body (loop)))))

    (loop))))
```

☞ Anmerkung: Eine While-Schleife testet iterativ eine Bedingung und führt eine Folge von Anweisungen aus, solange die Bedingung erfüllt ist.

Die Form „@body“ ist ein sogenannter „splice“: Das Argument „body“ faßt die restlichen Parameter des Macros zu einer Liste zusammen - der splice fügt die Elemente der Liste ohne die Klammern in den expandierten Macrotext ein.

Imperative Macros in swindle/misc.ss

```
(while condition body ...)

(until condition body ...)

(dotimes (i n) body ...)

(dolist (x list) body ...)

(no-errors body ...)

> (define i 7)
> (while (< i 10)
```

```

(inc! i)(writeln i))
8
9
10

```

31.1.3 Macros zur Effizienzsteigerung

Berechnung zur Übersetzungszeit

- Mit Macros können wir die Berechnung von Ausdrücken schon zur Übersetzungszeit ausführen und so an kritischen Stellen das Programm optimieren.
- Konstante Ausdrücke können zur Übersetzungszeit berechnet werden, wie (**length** '(1 2 3)) oder (* pi 2)
- Wenn wir eine Funktion als Macro definieren, wird der Aufwand für einen Funktionsaufruf eingespart, z.B. die Bindung von lokalen Variablen usw.

Rechenzeitsparnis durch Macros

Beispiel: 144 (Mittelwert von mehreren Zahlen)

Auch wenn die aktuellen Werte zur Übersetzungszeit noch nicht bekannt sind, steht die Anzahl schon fest und kann vorausberechnet werden:

```

(defmacro (avgM &rest args)
  ;The average of the arguments: Macro
  '(/ (+ ,@args) ,(length args)))
> ((avgM 1 2 3) → 2

```

Ein Laufzeitvergleich

```

(define (avgF &rest args)
  ;The average of the arguments: Function
  (/ (apply + args) (length args)))
> (time (dotimes (i 1000000)
    (avgM 1 2 3 4 5 6 7 8 9 10)))
cpu time: 472 real time: 472 gc time: 0
> (time (dotimes (i 1000000)
    (avgF 1 2 3 4 5 6 7 8 9 10)))
cpu time: 2967 real time: 3010 gc time: 1663

```

Beobachtung:

Das Macro spart nicht nur Rechenzeit sondern auch Aufwand für die garbage collection.

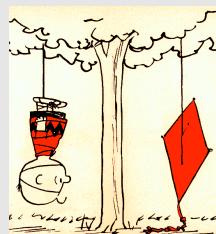
31.2 Fallstricke der Macroprogrammierung

Es gibt ein paar böse Fallstricke, die beim Schreiben von Macros zu beachten sind: Diese betreffen ungewollte Mehrfachauswertung und den Kontext von lokalen Namen. Im günstigsten Fall wird das Programm nur ineffizient, wenn wir von diesen Fallstricken betroffen sind, im schlimmsten Fall aber kann das Programm etwas ganz anderes tun, als wir beabsichtigt haben.

Wir müssen beim Entwurf von Macros bedenken, daß

- jede Auswertung eines Ausdrucks Seiteneffekte haben kann und eventuell mehr tut, als nur einen Wert zu ermitteln (sofern wir imperativ programmieren). Es ist also wichtig, wie oft und in welcher Reihenfolge Ausdrücke evaluiert werden.
- daß es für die Bedeutung einer Variablen wichtig ist, in welchem lexikalischen Kontext sie ausgewertet wird. Derselbe Name kann an unterschiedlichen Stellen des Programms an unterschiedliche Werte gebunden sein.

Fallstricke beim Macroentwurf



- ☞ Unbeabsichtigte Mehrfachauswertung
(Rechenaufwand, Seiteneffekte)
- ☞ Unbeabsichtigte falsche Referenzen

31.2.1 Mehrfachauswertung

Mehrfachauswertung und Effizienzverlust

```
> (defmacro (power8M x)
  ;compute the 8th power of x
  '(* ,x ,x ,x ,x ,x ,x ,x ,x))
> (power8M (sin 1.0)) →
0.25136983699568205
```

(**sin** 1.0) wird in diesem Beispiel *achtma*l berechnet, obwohl wir den Ausdruck nur einmal geschrieben haben.

Ein Vergleich

```
(define (power8F x)
  (* x x x x x x x x))

> (time (dotimes (i 1000000)
           (power8M (sin 1.0))))
cpu time: 3559 real time: 3594 gc time: 1668
> (time (dotimes (i 1000000)
           (power8F (sin 1.0))))
cpu time: 1703 real time: 1721 gc time: 997
```

Das folgende Macro incf! soll zwei Dinge tun:

1. Die Variable inkrementieren, die als Argument übergeben wird,
2. und den neuen Wert der Variablen als Ergebnis zurückgeben.

In Swindle gibt es zwar schon die Funktion inc!, aber diese gibt kein Resultat zurück, da Funktionen, die Seiteneffekte verursachen, in Racket nicht in Ausdrücken verwendet werden sollen.

31.2.2 Seiteneffekte

Mehrfachauswertung und Seiteneffekte

```
> (defmacro (incf! x)
  '(begin (inc! ,x) ,x))
> (define *x* 1)
> (incf! *x*) → 2
> (defmacro (power4M x)
  ;compute the 4th power of x
  '(* ,x ,x ,x ,x))
```

```

> (power4M (incf! *x*)) → 360
> *x* → 6
; *x* wird viermal erhöht, nicht nur einmal!

```

Achtung!

Auch `(incf *x*)!` wird viermal berechnet, obwohl wir den Ausdruck nur einmal geschrieben haben.

```

(define (show x)
  ; drucke und gebe Wert zurück
  (writeln x) x)
> (define *x* 1)
> (power4M (show (incf! *x*)))
2
3
4
5
120
> (power4M (show (random 100)))
76
52
78
66
20344896

```

☞ Anmerkung: Wie können wir Mehrfachberechnungen verhindern? Indem wir die Argumente des Macros vor der ersten Verwendung an lokale Variable binden. Im folgenden Beispiel wird x nur einmal ausgewertet und wir sparen sogar noch eine Multiplikation.

```

(defmacro (power-4L x)
  "compute_the_4th_power_of_x"
  '(let* ((y ,x)
          (y2 (* y y)))
    (* y2 y2)))
> (define *x* 1)
> (power-4L (show (incf! *x*)))
2
16

```

Aber lokale Variable haben ebenfalls ihre Tücken, und wenn wir nicht aufpassen, treiben wir den Teufel mit dem Beelzebub aus, wie wir gleich sehen werden.

31.2.3 Lexikalischer Kontext

Macros und lokale Variable

```
> (defmacro (ntimes n &rest body) ;repeat body n times
  '(dotimes (i ,n) ,@body))

> (ntimes 5 (push! 5 *xs*))
> *xs* → (5 5 5 5 5)

> (define *xs* '())
> (define i 10)
> (ntimes i (push! i *xs*))
> *xs* → ()
> i → 10

> (ntimes 5 (show(incf! i)))
1
3
5
```

Macros und lokale Variable

Was ist passiert?

Die Referenz zur *globalen Variable „i“* wird im Kontext einer *lokalen Variablen „i“* ausgewertet.

```
> (ntimes 5 (show(incf! i)))
1
3
5
; expandiert zu
(dotimes (i 5) (show(incf! i)))
```

☞ Anmerkung: Wir dürfen in einem Macro nur Namen verwenden, die nicht schon als Symbole eingeführt sind. Nur, wie finden wir Namen, die garantiert nie jemand verwenden wird? Wir sind ja keine Hellseher. Jeder Name, der uns einfällt, und sei er auch noch so abwegig, kann auch einem anderen Anwender unseres Macros einfallen, denn was unwahrscheinlich ist, ist noch lange nicht unmöglich. Dafür sorgen schon Murphy's laws.

Es bleibt nur eins: Wenn wir einen neuen Namen brauchen, dann durchsuchen wir die Symboltabellen von Lisp und wählen dann einen Namen, der zur Zeit nicht benutzt wird. Diese Arbeit nimmt uns die Standardfunktion **gensym** ab. Wann immer wir **gensym** aufrufen, erhalten wir einen eindeutigen und taufrischen Namen zurück, der bisher noch nicht in den Symboltabellen vorkommt. **gensym** steht für *generate symbol*.

☞ Anmerkung: Wenn Sie neue syntaktische Strukturen mit `define-syntax` definieren, anstelle von **defmacro**, dann vermeiden Sie diese unangenehmen lexikalischen Probrem. `define-syntax` definiert „hygienische“ Macros. Für anspruchsvolle Programmierprojekte lohnt es sich daher sehr, die etwas aufwendige *pattern language* von Racket zu lernen.

Generierung eindeutiger Namen:**gensym**

gensym liefert neue, ungebundene Namen für Symbole.

```
> (gensym) → g681
> (gensym) → g682

(defmacro (ntimesG n &rest body)
  "repeat_body_n_times"
  (let ((index (gensym))
         (times (gensym)))
    '(%let ((, times ,n))
      (dotimes (index ,n) ,@body))))
> (define i 1)
> (ntimesG 3 (show(incf! i)))
2
3
4
```

Relationale Programmierung [Weiter mit Prolog](#)

32 Stromorientierte Programmierung

32.1 Verzögerte Auswertung

The screenshot shows a presentation slide with a light purple header bar containing the title 'Verzögerte Auswertung'. The main content area has a light gray background. On the left side, there is a small green icon of a leaf with a brain inside it. To the right of the icon, three numbered items are listed in circles: 28 Metaprogrammierung und Spracherweiterung, 29 Stromorientierte Programmierung (with two sub-points: Verzögerte Auswertung and Ströme), and 30 Programmieren mit Zuständen. At the bottom right of the slide, there is a set of small blue navigation icons.

Vorteile:

Verzögerte Auswertung kann nützlich sein:

- Zur Definition nicht-strikter Funktionen.
- Zur Vermeidung von unnötigen Berechnungen.
- Zur Verarbeitung potentiell unendlich großer Datenstrukturen.
- Zur Implementierung des strom-orientierten Verarbeitungsmodells (siehe streams in Miranda, pipes unter Unix).

32.2 Ströme

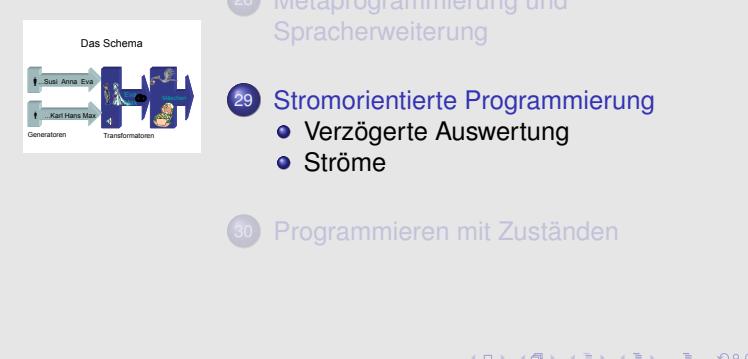
☞ Anmerkung: Wir wollen Kontrollstrukturen zur Verzögerung der Auswertung einführen (`delay` und `force`) und einen abstrakten Datentyp `stream` definieren, der den streams von Miranda(Holyer, 1991) entspricht. In Miranda beispielsweise ist jede Liste gleichzeitig ein Strom.

Der Ansatz: Eine Datenstruktur für unendliche Mengen:

- Die Menge beschreibt, wie die Werte zu berechnen sind.
- Wenn ein Element der Menge schon berechnet wurde, wird es gespeichert und kann später wieder direkt ohne Neuberechnung abgerufen werden.
- Wenn der Wert noch nicht berechnet wurde, dann wird er bei Bedarf berechnet.

Dieses Vorgehen hat große Ähnlichkeit mit dem Verfahren, welches wir für die `memo`-Funktion verwendet haben. Auch hier werden wir *closures* nutzen, um eine Kapsel herzustellen, die die Funktion zur Berechnung der noch nicht bekannten Werte an die Umgebung zu binden, in der die Funktion definiert wurde.

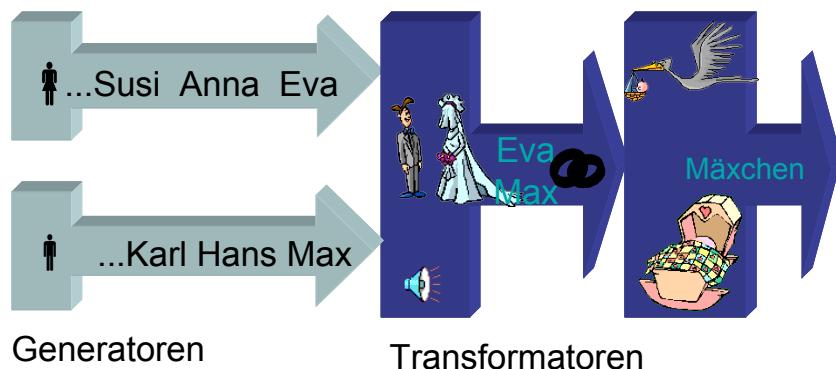
Stromorientierten Programmierung



Stromorientierten Programmierung

Bei der stromorientierten Programmierung liegt der Akzent auf dem Fluß der Daten: Wir modellieren die Programme als ein System von Generatoren, die Ströme von Werten produzieren, und Transformatoren, die die Werte des Eingabestroms elementweise transformieren und einen neuen Strom von Ausgaben erzeugen (Filter, Abbildungsfunktionen).
Kombinatoren, die mehrere Ströme zu einem kombinieren.
Selektoren und Konstruktoren

Das Schema



Ströme als rekursive Datenstruktur

- Ströme haben wie Listen eine rekursive Struktur:
 - Sie bestehen aus einem (endlichen) *Kopf*
 - und einem (potentiell unendlichen) *Reststrom*
 - oder sind leer.
- Ströme können unendlich lang definiert sein.

Da immer nur ein Element zur Zeit (der Kopf) dem Strom entnommen werden kann, ist stets nur ein endlicher Teil des Stroms ausgewertet.

- Der *Rest* ist ein Versprechen, auf Anforderung beliebig viele Elemente sequentiell zu produzieren.

Strompunktion und Kopfform

Definition: 145 (*Stromfunktion*)

Eine Stromfunktion ist eine Funktion, die einen Eingabestrom auf einen Ausgabestrom abbildet. Dabei werden die Eingabewerte elementweise verarbeitet und direkt wieder in Ausgabewerte abgebildet, ohne daß der Rest des Eingabestroms ausgewertet sein muß.

Damit eine Funktion eine Stromfunktion ist, muß sie in Kopfform definiert sein:

Definition: 146 (*Kopfform*)

Ein Funktion ist in Kopfform definiert, wenn das Kopfstück des Ausgabestroms nur vom Kopfstück des Eingabestroms abhängt.

Map als Stromfunktion

Eine Funktion mit Akkumulator ist nicht in Kopfform, da sie erst Werte zurückgibt, wenn die Rekursion beendet ist.

```
(define (map-no-k f acc xs) ; keine Kopfform
  (cond ((null? xs) (reverse acc))
         (else (map-no-k
                  (cons (f (car xs)) acc) (cdr xs)))))
 
(define (map-k f xs) ;Kopfform
  (cond ((null? xs) '())
         (else
          (cons (f (car xs)) (map-k f (cdr xs))))))
```

☞ Anmerkung: Wenn wir Stromfunktionen programmieren wollen, dann sind die endrekursiven Formulierungen mit Akkumulator also ungünstig. Unendliche Listen lassen sich gar nicht mit Akkumulator programmieren.

In Programmiersprachen mit vorgezogener Auswertung, wie in Racket oder Common Lisp, müssen wir die verzögerte Auswertung erzwingen, was aber relativ einfach ist, wie wir gleich sehen werden.

Die Grundoperationen für verzögerte Auswertung

In Racket gibt es die special form operators *delay* und *force*:

delay: Legt einen zu berechnenden Ausdruck in einer Datenstruktur ab, so daß er später bei Bedarf evaluiert werden kann.

force: Werte eine Datenstruktur aus, die eine verzögerte Berechnung beschreibt.

Speichere das Resultat der Evaluation, so daß der Ausdruck kein zweites Mal evaluiert werden muß.

Ströme als ADT in Racket

- Ein Strom ist wie eine Liste eine *linear rekursive Datenstruktur*, die
 - leer sein kann, oder
 - aus einem *Kopf*
 - und einem *Schwanz* vom Typ Strom besteht.
- Der Schwanz ist zu jedem Zeitpunkt nur endlich weit ausgewertet, kann aber unbegrenzt viele Elemente enthalten. **Das Sieb des Erathostenes**

Die Schnittstelle zum ADT “Strom”

the-empty-stream: Eine Konstante, der leere Strom..

(cons-stream head tail): Kombiniere head und tail zu einem Strom.

(head stream): Gebe das Kopf-Element des Stroms zurück.

(tail stream): Gebe den Schwanz zurück. Erzwinge die Auswertung, sofern noch nicht geschehen.

(stream-ref stream i): Gebe das Element an der Position i im Strom zurück.

Beispiel: 147 (Implementation der Ströme)

Da die Struktur eines Stroms einer Liste ähnlich ist, implementieren wir Ströme als Listen.

```

(defmacro (cons-stream2 a b)
  '(cons ,a (delay ,b)))
(define (head-stream stream)
  (car stream))
(define (tail-stream stream)
  (cond [(null? stream) '()]
        [((null? (cdr stream)) '())
         [((pair? (cdr stream)) (cdr stream))]
         [else (force (cdr stream))]]])
(define (empty-stream? stream)
  (null? stream))
(define the-empty-stream '())

```

Der Strom der natürlichen Zahlen

Der Strom der natürlichen Zahlen ab der Zahl n:

```

(define (integers-from-n n)
  (cons
    n
    (delay
      (integers-from-n (+ 1 n)))))

> (define ab-3 (integers-from-n 3))
> ab-3 → (3 . #<struct:promise>)

```

- Die Rekursion hat keine Abbruchbedingung.
- Der Strom ist potentiell unendlich lang.
- Die Rekursion geht über den Nachfolger, nicht den Vorgänger.

 **Anmerkung:** Das Rekursionsschema unterscheidet sich wesentlich von unserem bisherigen Schema: Bisher haben wir in jedem Rekursionsschritt das Argument beim rekursiven Aufruf verkleinert und die Rekursion terminierte, wenn der Basisfall (0 oder '()) erreicht wurde. Die Definition für integers-from-n enthält keinen Basisfall und das Argument wird bei jedem Aufruf größer und nicht kleiner. Das Resultat der Funktion ist eine potentiell unendliche Datenstruktur, aber sie wird nur soweit berechnet, wie es durch force erzwungen wird. Es ist daher wichtig, daß der rekursive Aufruf direkt als Argument eines delays erscheint, sonst würde wegen der vorgezogenen Auswertung ein unendlicher Prozeß entstehen.

Die Implementation der weiterer Stromfunktionen finden Sie im Anhang, siehe C, Seite 741.

Das Sieb des Eratosthenes für Primzahlen

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Primzahlen: $\{2, 3, 5, 7, \dots\}$

Stromorientierte Lösung

Beispiel: 148 (Das Sieb des Eratosthenes)

Siebe die unendliche Menge der natürlichen Zahlen nach folgendem Schema:

- Die Folge der natürlichen Zahlen außer der 1 bildet einen Strom von Kandidaten für Primzahlen.
- Der Kopf des aktuellen Kandidatenstroms ist eine Primzahl.
- Nehme diese und siebe den Reststrom, indem alle Vielfachen der Primzahl im Reststrom gelöscht werden.
- Nimm den gesiebten Reststrom als neuen Kandidatenstrom und wende des Verfahren iterativ an.
- Es entsteht ein Strom von Primzahlen aus den Köpfen der Kandidatenströme.

Schema

- Primzahl n ist der Kopf von Kandidatenstrom n .
- Der Kandidatenstrom $n + 1$ wird aus Kandidatenstrom n gebildet, indem Strom n mit Primzahl n gesiebt wird.

$$\begin{array}{ll} () & (2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ \dots) \\ (2) & (3 \ 5 \ 7 \ 9 \ 11 \ 13 \ 15 \ 17 \ 19 \ 23 \ 25 \ 27 \ \dots) \\ (2 \ 3) & (5 \ 7 \ 11 \ 13 \ 17 \ 19 \ 23 \ 25 \ 29 \ \dots) \\ (2 \ 3 \ 5) & (7 \ 11 \ 13 \ 17 \ 19 \ 23 \ 29 \ \dots) \\ \dots & \dots \end{array}$$

```

(define (not-divisible? x y)
  (not (= 0 (remainder x y)))))

(define (sieve stream)
  (cons
    (head-stream stream)
    (delay
      (filter-stream
        (rcurry
          not-divisible?
          (head-stream stream))
        (sieve (tail-stream stream))))))

> (define *primes*
  (sieve (integers-from-n 2)))
> (take-stream 10 *primes*)
(2 3 5 7 11 13 17 19 23 29)

> (take-stream 1000 *primes*)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283
293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509
521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631
641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751
757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877
881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997
1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091
1093 1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193
1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291
1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423
1427 1429 1433 1439 1447 1451 1453 1459 1471 1481 1483 1487 1489 1493
1499 1511 1523 1531 1543 1549 1553 1559 1567 1571 1579 1583 1597 1601
1607 1609 1613 1619 1621 1627 1637 1657 1663 1667 1669 1693 1697 1699
1709 1721 1723 1733 1741 1747 1753 1759 1777 1^C<<... interrupt>>
^C<<interrupt>>

```

☞ Anmerkung: Eratosthenes Eratosthenes war ein griechischer Gelehrter, geboren in Afrika in Kyrene um 295/280 v. Chr. gestorben um 200 v. Chr. Gegen 240 wurde er von Ptolemäus III. Euergetes als Leiter der Bibliothek und Lehrer von Ptolemäus IV nach Alexandria berufen. Heute noch überliefert sind seine mathematischen und geographischen Arbeiten, insbesondere sein Verfahren zur Bestimmung der Primzahlen. Weit über tausend Jahre, bevor Galileo Galileo vor der Inquisition dem heliozentrischen Weltbild abschwören mußte, hat Eratosthenes schon ziemlich genau den Durchmesser der Erdkugel und die Schiefe der Ekliptik (die Neigung der Erdachse gegenüber der Bahnebene der Erde um die Sonne) gemessen.

Lazy Scheme

In der experimentellen DrRacket-Sprache „Lazy Racket“ sind – wie in Miranda und Haskell – alle Listen automatisch *Ströme*. Die Funktionen höherer Ordnung **map**, **filter**, **reduce** usw. sind *Stromfunktionen*.

```
#lang lazy
```

```
(define (natsAbN n)
  (cons n (natsAbN (+ n 1))))  
  
(!! (take 10 (natsAbN 1))) ; force promises  
→ (1 2 3 4 5 6 7 8 9 10)
```

Das Sieb des Erathostenes in Lazy Racket

```
#lang lazy
(define (sieve stream)
  (cons
    (car stream)
    (filter
      (lambda (num)
        (not-divisible? num
          (car stream)))
      (sieve (cdr stream)))))

(define primes (sieve (natsAbN 2)))

> (!! (take 10 primes ))
→ (2 3 5 7 11 13 17 19 23 29)
```

☞ Anmerkung: Erzeugung von Sätzen einer Sprache:

Erinnern Sie sich an unser erstes Racket-Programm, die regelbasierte Erzeugung von zufälligen Sätzen einer Sprache (siehe 9.3.3, Seite 131)?

Bisher haben wir einzelne, zufällige Sätze der Sprache generiert. Als nächste Anwendung werden wir *alle* Sätze einer Sprache aufzählen. Mit einer Stromfunktion geht das, auch wenn die Sprache unendlich ist und die Länge der Sätze unbeschränkt ist. Dazu definieren wir eine Datenstruktur, die die Regeln der Sprache beschreibt.

Weiterhin definieren wir eine Funktion `generate-all`, die einen Strom von Sätzen der Sprache erzeugt, wobei jeder Satz, der ja beliebig lang sein kann, ebenfalls ein Strom ist. Es werden noch einige kombinatorische Stützfunktionen benötigt, um das Kreuzprodukt von Strömen aufzählen zu können. `append-streams`, usw. Im `tools-module` finden Sie das Programm als Anwendungsbeispiel für unsere selbstdefinierten Stromfunktionen.

```
(define (generate-all phrase)
  (cond
    ((null? phrase) (list '()))
    ((pair? phrase)
     (combine-all-streams
      (generate-all (car phrase))
      (generate-all (cdr phrase))))
    (else
     (let ((choices (assoc phrase *grammar*)))
       (if choices
           (mappend-stream generate-all
            (rule-rhs choices))
           (list (list phrase)))))))
```

33 Programmieren mit Zuständen

33.1 Caching und Memoization

Standardverfahren zur Effizienzsteigerung von Algorithmen



28 Metaprogrammierung und Spracherweiterung

29 Stromorientierte Programmierung

30 Programmieren mit Zuständen

- Caching und Memoization
- Empfehlungen
- Satire: Man mordet nicht nach Sprungbefehl

Navigation icons: back, forward, search, etc.

☞ Anmerkung: Optimierung auf der algorithmischen Ebene In den Modulen zu den formalen Grundlagen der Informatik haben Sie ja schon kennengelernt, daß für die Effizienz eines Algorithmus wichtig ist, von welcher Ordnung das Verfahren ist — also wie der Aufwand der Berechnung in Abhängigkeit von der Größe der Eingabe wächst.

Was tun, wenn wir feststellen, daß der einzige Algorithmus, den wir zur Lösung unseres Problems gefunden haben, von exponentieller Ordnung und damit, abgesehen von trivialen Fällen, praktisch unbrauchbar? Baumrekursion beispielsweise führt typischerweise zu exponentiellem Aufwand.

In diesem Kapitel wollen wir zwei Standardverfahren der funktionalen Programmierung ansprechen, mit denen wir die Ordnung eines Verfahrens reduzieren können: Caching und Memoization.

Caching und Memoization

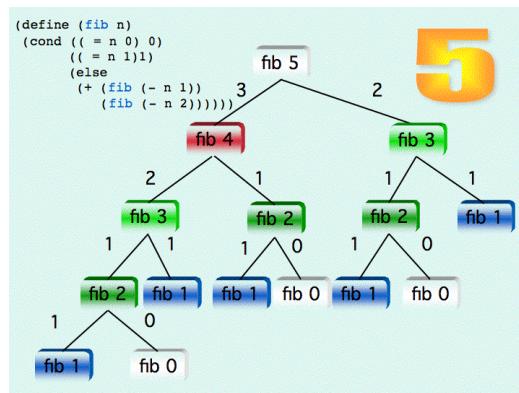
Caching ist eine Technik, bei der Werte, die aufwendig zu berechnen sind, gespeichert werden. Wenn ein solcher Wert ein zweites Mal benötigt wird, dann wird er einfach aus dem Speicher ausgelesen.

Memoization ist die Technik, Funktionsdefinitionen so umzuformen, daß Mehrfachberechnungen durch *caching* vermieden werden.

Nochmal: Die Fibonacci-Zahlen

Der baumartig-rekursive Prozeß zur Berechnung der Fibonacci-Zahlen ist von exponentieller Ordnung: $\text{fib}(n) = \mathcal{O}(1.7^n)$

```
(define (fib n)
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [else (+ (fib (- n 1))
                  (fib (- n 2))))]))
```



☞ Anmerkung: Wir wollen die vielen Mehrfachberechnungen vermeiden, indem wir uns in einer Tabelle merken, welche Funktionswerte schon berechnet wurden. Ehe eine Funktion mit der Berechnung für ein Argument beginnt, prüft sie, ob das Resultat schon bekannt ist.

Memoization

Beispiel: 149 (Memoization)

Wandlung einer Funktion in eine Memo-Funktion:

- Lege eine Tabelle für schon berechnete Funktionswerte an.
- Wenn die Funktion aufgerufen wird, prüfe, ob für das Argument schon als ein Funktionswert verfügbar ist.
 - Wenn ja, gebe diesen Funktionswert als Resultat zurück.
 - Wenn nein, berechne den Funktionswert, trage ihn in der Tabelle ein und gebe ihn als Resultat zurück.
- Binde den Namen der Funktion an die neue Memo-Funktion.

Memoization: Das Schema

```
"Return_a_memo-function_of_fn . "
(letrec
  ([table (make-table)]
   [store (lambda (arg val) ...)]
   [retrieve (lambda (arg) ...)]
   [make-table (lambda () ...)])
  [ensure-val
   (lambda (x)
     (let ([stored-val (retrieve x)])
       (if stored-val stored-val
           (store x (fn x))))))
  ensure-val))
```

☞ Anmerkung: Und nochmals closures `memo` ist eine Funktion höherer Ordnung, die aus der Funktion `fn` eine neue Funktion erzeugt. Man beachte, daß diese neue Funktion innerhalb eines `lets` definiert wird, in dem als lokale Variable die Tabelle erzeugt wird, in der die memo-Funktion die schon berechneten Funktionswerte speichern soll. Da diese Tabelle eine lokale Variable ist, kann sie von der Funktion `memo` nicht mehr zugegriffen werden, wenn der Aufruf von `memo` beendet ist. Nur für die von `memo` erzeugte Funktion ist die Tabelle noch zugreifbar, denn die Variable `table` gehört zum lexikalischen Kontext, in dem diese Funktion definiert wurde.

Der Wert, den `memo` zurückgibt, ist die Funktion `ensure-val` — eine *closure (Abschluß)*, ein Paar aus dem Code der neuen Funktion und einer Umgebung, in der die Variablen der Umgebung, in der diese Funktion definiert wurde, eingefroren sind. In dieser eingefrorenen Umgebung sind der Aktualparameter `fn` und die Tabelle `table` zusammen mit dem Funktionscode gekapselt. Wenn wir mehrere Funktionen mit `memo` transformieren, wird jede dieser so erzeugten *closures* ihre eigene Tabelle besitzen.

Closures lassen sich also hervorragend verwenden, um Datenkapseln herzustellen, die Datenstrukturen mit Funktionen verbinden. Wir werden diese Methode noch mehrfach nutzen, vor allem um verzögerte Auswertung in Scheme einzuführen. Closures eignen sich auch als Basis für die objektorientierte Programmierung. (Abelson and Sussman, 1985).

Implementation der Tabelle:

Wir realisieren die Tabelle als Assoziationsliste:

```
(define (memo fn)
  (letrec
```

```
([table '()]
 [store
  (lambda (arg val)
    (set! table (cons (cons arg val) table)))
    val))
 [retrieve
  (lambda (arg)
    (let ((val-pair (assoc arg table)))
      (if val-pair (cdr val-pair) #f))))]
 ...
)
```

☞ Anmerkung: Die Repräsentation der Tabelle als Assoziationsliste ist nicht besonders effizient. Die Liste wird für jeden Zugriff linear durchsucht und der Zugriff dauert im Mittel umso länger, je mehr Einträge die Tabelle enthält. Wenn Sie memo-Funktionen anwenden, dann sollten Sie unbedingt die Tabellen so implementieren, daß die Zugriffszeit auf einen Eintrag (nahezu) unabhängig von der Größe der Tabelle ist. Geeignete Datenstrukturen hierfür wären beispielsweise Arrays oder – noch besser, weil dynamisch erweiterbar – Hash-Tabellen. Hash-Tabellen können in Scheme einfach mit (make-hash-table) erzeugt werden.

Beispiel: fib als Memo-Funktion

```
> (define memo-fib (memo fib))
> (memo-fib 3)
3
```

Ein Trace mit memo: Der erste Aufruf

```
> (trace memo-fib fib)      → (memo-fib fib)
> (memo-fib 3)
|(memo-fib 3)
| (fib 3)
| |(fib 2)
| | |(fib 1)
| | | |1
| | | |(fib 0)
| | | | |1
| | | |2
| | | | |(fib 1)
| | | | | |1
| | | | |3
|3 → 3
```

Beim zweiten Aufruf

```
> (memo-fib 3)
|(memo-fib 3)
|3
→ 3
```

Beim zweiten Aufruf von memo-fib wird nur der Tabellenwert zurückgegeben, ohne neu zu rechnen. Dummerweise werden die rekursiven Aufrufe von fib aber immer noch von der ursprünglichen Funktion fib ausgeführt, so daß sich am exponentiellen Aufwand einer Neuberechnung nichts geändert hat. Es ist aber leicht, diese Aufrufe so umzulenken, daß auch die rekursiven Aufrufe durch memo-fib ausgeführt werden. In Lisp können wir ja Funktionsdefinitionen zur Laufzeit des Programms durch neue Definitionen ersetzen. Wir brauchen nur den Namen fib mit set! an die neue Funktion memo-fib zu binden.

Ändern der Referenz auf fib

```
> (set! fib (memo fib))
>> (fib 4)
|(fib 4)
| (fib 3)
| |(fib 2)
| | |(fib 1)
| | |1
| | |(fib 0)
| | |1
| |2
| |3
|5 → 5
```

Auch die rekursiven Aufrufe gehen jetzt an die Memo-Funktion.

```
> (fib 8)
|(fib 8)
| (fib 7)
| |(fib 6)
| | |(fib 5)
| | |8
| |13
| |21
|34
-- > 34
```

Laufzeitmessung

CPU-Zeit in ms, Sparc 4

n	fib(n)	ohne memo.	mit memo.
5	8	0	0
10	89	0	0
20	10946	10	10
30	1346269	730	10
40	165580141	89890	10
100	573147844013817084101	—	20

fib (1000)

```
fib (1000)= 703303677 1142281582 1835254877 1835497701 8126983635  
8732742604 9050871545 3711819693 3579742249 4945626117 3348775044  
9241765991 0881863632 6545022364 7106012053 3741212738 6733911119  
8139373125 5987676900 9190224524 5323403501 ≈ 7.0e208, CPU 210  
ms
```

Imperative Operationen in memo

Für die Programmierung von memo haben wir an zwei Stellen imperativer Sprachelemente eingesetzt:

- Das Speichern in der Tabelle: *store*
- Die Modifikation der Referenz auf *fib*.
- Zusammen haben diese beiden Maßnahmen eine Funktion von exponentieller Ordnung in eine Funktion von linearer Ordnung gewandelt.

☞ Anmerkung: In der angegebenen Form sind die memoization-Funktionen noch nicht optimal:

- Es können nur Funktionen mit nur einem Argument gewandelt werden.

Bei (Norvig, 1992) können Sie nachlesen, wie diese Einschränkung beseitigt werden kann. Wir können hier aus Zeitgründen nicht näher darauf eingehen.

Bei einigen Funktionen versagt allerdings auch diese Technik. Schon mit dem Argument (4,4) läßt sich die Ackermann-Funktion so nicht berechnen, weil die Tabelle so groß wird, daß ein „segmentation fault“ ausgelöst wird, weil der Speicher nicht reicht.

☞ Anmerkung: In `swindle-misc.ss` finden Sie effiziente vordefinierte Funktion `memoize` und `memoize!` und auch mehrdimensionale Hash-Tabellen.

33.2 Empfehlungen

- Programme sind besser verständlich, wenn Variable nur einmal an einen Wert gebunden werden und diesen Wert behalten.
- Für jede Variable sollte klar sein, wo sie ihren Wert erhält und wo dieser Wert möglicherweise geändert wird.
- Nutzen sie die Initialisierungsmöglichkeiten der Programmiersprachen, so daß Variable möglichst gleich bei der Definition gebunden werden.
- Definieren Sie Variable, die ihren Wert nicht ändern werden, gleich als Konstante, wenn die Programmiersprache das zuläßt.
- Reduzieren Sie Modifikatoren auf wenige, gut dokumentierte Stellen, so daß bei Programmänderungen klar nachzuvollziehen ist, welche Objekte sich wie ändern können.
- Auch in imperativen Programmiersprachen können Sie viele Entwurfsprinzipien des funktionalen Programmierens einsetzen, um Ihre Programme lesbarer und wartbarer zu machen.

In der Anfangszeit der imperativen Programmierung waren nicht die Zustandsänderungen die Hauptsorge sondern der Spaghetticode der unstrukturierten Programme, wie die folgende Satire zeigt. In einigen älteren Lisp-Dialekten gibt es den Sprungbefehl `goto` noch immer, aber in Scheme wurde er glücklicherweise gar nicht erst eingeführt.

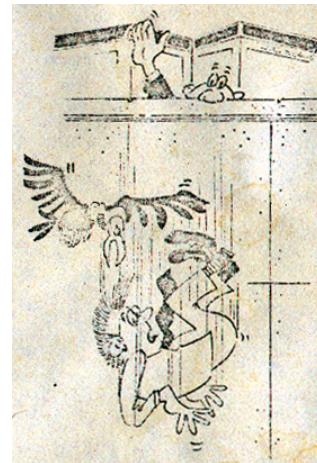
33.3 Satire: Man mordet nicht nach Sprungfehl

Also komme ich als Gutachter in diesem einzigartigen Fall von Software-Kriminalität zu dem Schluß, daß der Angeklagte Theo McSnell, Chefprogrammierer bei Generators Unlimited, für den angeblichen Mordversuch an Marius Schmidt-Schleicher vielleicht ein Motiv hatte, jedoch trotz seiner provozierten und darum verständlichen Wut mit dem Täter (falls es überhaupt einen Täter gibt) niemals identisch sein kann. Ich werde Ihnen das beweisen.

Lassen Sie mich zuvor rekapitulieren, was an jenem denkwürdigen Maiabend auf der Trelement-Dachebene der Computerhalle in Hannover geschah, als McSnell und Schmidt-Schleicher über die Vorzüge und Nachteile ihrer jeweiligen Software-Innovationen in Streit gerieten.

Nach übereinstimmenden Berichten zahlreicher Zeugen begann der Disput während eines Cocktail-Empfangs in der sogenannten Junior-Galerie. Dort hatte Schmidt-Schleicher — und das ist verbürgt — eine abstrakte Collage des Dadaisten Pawel Kahos mit den Struktogrammen aus den Standard-Utilities der Fa. Generators Unlimited verglichen. Offenbar animiert von der berufstypischen Fachsimpelie der EDV-Journalisten — und weitgehend enthemmt nach dem Genuß von fünf bis acht Gläsern Campari — erklärte Schmidt-Schleicher lautstark (wobei er auf das gegenstandslose Liniengewirr des Kunstwerks deutet): „Das ist bestimmt so eine miese Matrix modular vermantelter Multiprogramme aus dem NUTS-Angebot!“

Alle Zeugen stimmten darin überein, daß Schmidt-Schleicher den anwesenden Entwickler von NUTS, den amerikanischen Spezialisten Theo McSnell, mit dieser Bemerkung kränken wollte. Zur Information: Das Programmpaket NUTS (Native Users Transaction Software) steht in scharfem Wettbewerb zu PACMAS (Productive And Carefree Mainframe Application Software), das die Firma des besagten Schmidt-Schleicher vertreibt und bei verschiedenen Pilot-Installationen einsetzt. Was Wunder, das der tarantelgestochene NUTS-Erfinder McSnell heftig reagierte und Schmidt-Schleichers unqualifizierte Invektive zornig zurückwies!



Bei dem nachfolgenden Duett der Schmähreden fielen Ausdrücke wie „Spaghetti-Codierer“, Schleifenmuffel, „Assemblist“, „dezentrale Intelligenz“, „interaktiver Idiot“ und „strukturierter Quatschgenerator“. Es ist, nebenbei erwähnt, bezeichnend für die professionelle Psychoverkrüppelung der Widersacher, daß sie sich weniger über den beleidigenden Inhalt der wechselseitigen Verbalinjurien aufregten als über die Trivialfrage, ob es (wenn schon) „strukturierter“ oder „strukturierender“ Quatschgenerator heißen müsse.

In der entscheidenden Phase der Auseinandersetzung befanden sich die beiden Experten, deren Erzeugnisse übrigens im letzten Osiris-Report gleichermaßen das Prädikat „anwendungsnahe, implementierungsunabhängige Qualitätssoftware“ erhielten, plötzlich hinter dem Kunstgalerietrelement und außer Sicht der anderen Partygäste.

Sekunden später stürzte Schmidt-Schleicher vom Dach der CeBIT-Halle in die Tiefe. Glücklicherweise blieb er trotz des Höhenunterschieds von über 18 Meter fast unverletzt, weil er auf die Ladefläche eines geparkten Lkws mit hochgehäuften Lochkarten fiel. Wie erinnerlich, hatte die Messleitung am gleichen Abend diese Lochkarten wegen Erregung öffentlichen Ärgernisses beschlagnahmen lassen — angeblich aufgrund einer Beschwerde der Hersteller moderner Datenerfassungsgeräte.

Schmidt-Schleicher kam mit dem Schrecken und einer leichten Nabelprellung davon. Er sagt nun aus, Theo McSnell habe ihn über die Brüstung gestoßen. Mit Mordabsicht. Mr. McSnell bestreitet dies und erklärt seinerseits, der vom Unfallopfer verinnerlichte Campari sei wohl Schuld gewesen an diesem fernwirkungsfreien Top-Down-Approach des besagten Schmidt-Schleichers.

Ich bin überzeugt, daß McSnell die Wahrheit spricht. Denn: der Abgestürzte beharrt auf seiner Version, er sei von hinten geschubst worden, und der amerikanische Softwarekonstrukteur habe dabei ausgerufen: „Go to hell!“

Das, hohes Gericht, ist eine im Wortsinn unglaubliche Behauptung! Theo McSnell, das weiß jeder Branchenkollege, gehört seit vielen Jahren zum inneren Zirkel der Professor Dijkstra-Bruderschaft. Und wie Ihnen bekannt sein dürfte, sind solche Menschen völlig außerstande, ein „go to“ auch nur zu denken, geschweige denn auszusprechen. Allenfalls ist ihnen der orgastische Aufschrei „Heureka!“ zuzutrauen. Doch kein eingefleischter Strukturprogrammierer würde jemals einen Absprung mit „go to“ einleiten. Er duldet nämlich überhaupt keine widernatürlichen Hüpfen — weder in den Programmfspuren noch auf dem Dach der CeBIT-Halle.

Sogar bei der Ausführung eines Kapitalverbrechens könnte ein Dijkstra-Jacopini-Schüler nicht aus seiner Haut heraus. Er mordet nicht nach Sprungbefehl. Der Angeklagte Theo McSnell ist daher unschuldig und selbst das Opfer böswilliger Verleumdung.

Computerwoche, 11. Juni 1976:

Heinz-Günther Klaus

Teil XII

Relationale Programmierung

34 Relationale Programmierung

34.1 Der relationale Programmierstil

The screenshot shows a presentation slide with a light blue header bar containing the title 'Relationale Programmierung'. Below the header is a large, faint watermark-like silhouette of Sherlock Holmes smoking a pipe. To the right of the silhouette, there is a list of three items, each preceded by a small circular icon:

- 31 Relationale Programmierung
 - Der relationale Programmierstil
 - Die Datenbasis
 - Unifikation und Suche
- 32 Prolog als Datenbanksprache
- 33 Prolog als Inferenzmaschine

At the bottom right of the slide area, there is a set of small, standard presentation navigation icons.

Mit Racket haben wir einen Vertreter derjenigen Programmiersprachen kennengelernt, deren Semantik vollständig mathematisch formal spezifiziert wurde, und zwar auf der Basis von Funktionen.

Jetzt werden wir eine Programmiersprache kennenlernen, die für das relationale Verarbeitungsmodell entworfen wurde (Prolog) und deren Semantik ebenfalls formal definiert ist, nämlich über Logik und der Relationen. Mit Prolog haben wir neben einer mächtigen berechnungsuniversellen Programmiersprache auch noch gleichzeitig eine relationale Datenbank, eine Inferenzmaschine und eine deduktive Datenbank zur Verfügung.

Lisp (Racket) und Prolog sind insofern sehr verwandt, daß beide Sprachen interpretierte Sprachen sind, Symbolverarbeitung und Metaprogrammierung unterstützen und schwach getypt sind. Es ist daher relativ wenig aufwendig, Lisp um Prolog-Sprachelemente zu erweitern und umgekehrt.

Diese Kapitel basiert sehr stark auf dem Kapitel 11 aus Norvig, 1992. Wir werden hier Norvigs Lisp-Implementation eines Prolog-Interpreters vorstellen, die wir nach Racket portiert haben. Als weiterführende Literatur zu Logikprogrammierung sollten Sie auf die Prüfungsunterlagen zu SE-3, Teil B: Logikprogrammierung (Wolfgang Menzel) zurückgreifen, die im Informatikum ausliegen. Das Buch von Clocksin and Mellish, 2003 ist eine sehr empfehlenswerte Einführung in die Grundlagen der Prolog-Programmierung, das Buch von Sterling and Shapiro, 1994 beschreibt fortgeschrittene Methoden der Logikprogrammierung.

How, in the name of good fortune, did you know all that, Mr. Holmes?

Our visitor bore every mark of being an average commonplace British tradesman, obese, pompous, and slow. He wore rather baggy gray shepherd's check trousers, a not overclean black frock coat, unbuttoned in the front, and a drab waistcoat with a heavy brassy Albert chain, and a square pierced bit of metal dangling down as an ornament. A frayed top hat and a faded brown overcoat with a wrinkled velvet collar lay upon the a chair beside him. Altogether, look as I would, there was nothing remarkable about the man, save his blazing red head, and the expression of extreme chagrin and discontent upon his features.

Sherlock Holmes's quick eye took in my occupation, and he shook his head with a smile as he noticed my questioning glances. "Beyond the obvious facts that he has at some time done manual labor, that he takes snuff, that he has been in China, and that he has done a considerable amount of writing lately, I can deduce nothing else."

Mr. Jabez Wilson started up in his chair, with his forefinger upon the paper, but his eye on my companion.

„How, in the name of good fortune, did you know all that, Mr. Holmes?“ he asked. „How did you know, for example, that I did manual labor? It's true as gospel, for I began as a ship's carpenter.“

„Your hands, my dear Sir. Your right hand is quite a size larger than your left. You have worked with it, and the muscles are more developed.“

„Well, the snuff, then?“

„I won't insult your intelligence by telling how you how I read that.“

„Ah. But the writing?“

„What else can be indicated by that right cuff so very shiny for five inches, and the left one with the smooth patch near the elbow where you rest it upon the desk?“

„Well, but China?“

„The fish that you have tattooed immediately upon your right wrist could only have been done in China. I have made a small study of tattoo marks and have even contributed to the literature. That trick of staining the fishes' scales of a delicate pink is quite peculiar to China. When, in addition, I see a Chinese coin hanging from your watch chain, the matter becomes even more simple.“

Mr. Jabez Wilson laughed heavily. „Well, I never!“ said he. „I thought at first that you had done something clever, but I see that there was nothing in it, after all.“

„I begin to think, Watson,“ said Holmes, „that I make a mistake in explaining. '*On me ignotum pro magnifico*', you know, and my poor little reputation, such as it is, will suffer shipwreck if I am so candid.“

A. Conan Doyle, *The Red-Headed League* zitiert nach Raphael, 1976

Problemlösen durch Deduktion: Das Textfragment von Conan Doyle ist ein Lehrstück für Problemlösen durch Deduktion: Deduktion bedeutet, daß wir ein Problem lösen, indem wir ein systematisches Schlußfolgerungsverfahren anwenden. Und obwohl die Resultate, die so erzielt werden, sehr eindrucksvoll und verblüffend sein können, kann doch jeder Schritt entlang der Schlußfolgerungskette erklärt werden. Sobald der Beweis vollständig dargelegt ist, erscheinen die Resultate klar und eventuell sogar trivial. Deduktive Verfahren beruhen auf der mathematischen Logik, die in den Modulen zu den formalen Grundlagen der Informatik ausführlicher behandeln werden wird.

Um so schlau wie Sherlock Holmes zu sein, müssen wir zwei Dinge können:

1. Die für unser Problem relevanten Fakten erkennen, beobachten und erfassen.
2. Aus diesen Daten die passenden Schlußfolgerungen ziehen.

Der erste Schritt ist noch schwer auf Maschinen zu übertragen,⁹ aber der zweite Schritt läßt sich auf formale Schlußfolgerungssysteme übertragen, und deren Anwendung kann systematisch als Algorithmus formuliert werden.

Das logische Verarbeitungsmodell

Programmieren mit Relationen

Das logische Verarbeitungsmodell unterstützt besonders gut

- die Modellierung von *Beziehungen* zwischen Objekten,
- die Deklaration von *Fakten*,
- das automatische Beweisen von Aussagen über Objekte und deren Beziehungen durch *Deduktion*.

Die *Semantik* des Verarbeitungsmodells wird über Relationen und Logik spezifiziert.

Das Verarbeitungsmodell von Prolog

Für Prolog sind folgende Ansätze charakteristisch:

- Eine einheitliche, uniforme *Datenbasis*, die alle Fakten und Regeln als Klauseln relational (und sehr effizient) repräsentiert.
- *Logische Variable*: Prolog ist referentiell transparent, und Variable werden nur einmal durch Unifikation gebunden.
- *Automatisches Backtracking*: Die Auswertung der Beziehungen und die Deduktion von Fakten erfolgt automatisch.

Richtungsunabhängigkeit

- Ein *einzelnes* Prolog-Programm kann *viele* unterschiedliche Programmabläufe beschreiben, abhängig von der Art der Anfragen.
- Die Relationen wirken sowohl als *Selektoren* für Datenstrukturen als auch als *Konstruktoren*.

⁹Hierin war auch der arme Dr. Watson gar nicht gut, wie das Beispiel zeigt.

Das Ziel in diesem Kapitel:

- Wesentliche Merkmale des relationalen/logischen Verarbeitungsmodells kennenlernen,
- typische Programmierprobleme kennenlernen, für die das relationale Verarbeitungsmodell besonders gut geeignet ist,
- eine Implementation der Grundideen von Prolog kennen zu lernen, so daß diese einzeln oder zusammen in anderen Programmiersprachen bei Bedarf genutzt werden können.

34.2 Die Datenbasis

Die Prolog-Datenbasis: Klauseln

Die Datebasis repräsentiert das Wissen über die Anwendungsdomäne in Form von Zusicherungen (Klauseln).

Es gibt zwei Arten von Klauseln:

Fakten: Fakten beschreiben Beziehungen zwischen Objekten.

Regeln: Regeln beschreiben, wie Fakten aus anderen Fakten gefolgert werden können.

Beispiel: Fakten über Städte

Bevölkerung: Die Relation *population* stellt eine Beziehung zwischen einer Stadt und der Anzahl ihrer Einwohner her.

Hauptstadt: Die Relation *capital* stellt eine Beziehung zwischen einem Land und seiner Hauptstadt her.

(**population** SanFrancisco 750000)

(**capital** Sacramento California)

Zur Notation

- Wir verwenden hier Racket-Syntax, da wir Prolog in Racket einbetten wollen.
- Die Originalsyntax in Prolog wäre:

population (sanFrancisco ,750000).

capital (sacramento ,california).

Wer mag wen?

Eine Relation „likes“:

- Kim mag Robin.
- Sandy mag Lee und Kim.
- Robin mag Katzen.

```
( likes Kim Robin)  
( likes Sandy Lee)  
( likes Sandy Kim)  
( likes Robin cats )
```

Über Fakten können wir *endliche* Relationen durch *Aufzählung* definieren.

Kennzeichnen von Prologfakten

Der special-form-Operator `<–` soll Fakten als Prolog-Syntax kennzeichnen und in die Datenbasis eintragen.

Wir werden diese neue Sprachelement `<–` als Macro implementieren.

```
(<– ( likes Kim Robin ))  
(<– ( likes Sandy Lee ))  
(<– ( likes Sandy Kim ))  
(<– ( likes Robin cats ))
```

Relationen versus Funktionen

In Racket würden wir die „likes“-Relation durch Funktionen repräsentieren:

- Eine Funktion „likes“, die uns die Menge aller Dinge berechnet, die eine Person mag, z.B.

```
( likes 'Sandy ) → ( Lee Kim )
```

- Eine Funktion „likers-of“, die die Menge aller Personen berechnet, die etwas mögen, z.B.

```
( likers-of 'Lee ) → ( Sandy )
```

Die Relation „likes“

Die Relation „likes“ ist im Gegensatz zu den Funktionen *richtungsunabhängig* und kann in Anfragen an die Datenbasis für beide Richtungen benutzt werden:

```
( likes Sandy ?whom ) → ?whom=Lee , ?whom=Kim  
( likes ?who Lee ) → ?who=Sandy
```

Instanziierungsvarianten

Die unterschiedlichen Formen, beim Aufruf die Argumente an Werte zu binden oder ungebunden zu lassen, heißen *Instanziierungsvarianten* eines Prädikats.

Regeln

Der zweite Typ von Klauseln sind die Regeln:

Sie beschreiben Abhängigkeiten zwischen Fakten: Z.B. die Regel:

Sandy mag jeden, der Katzen liebt.

(\leftarrow (**likes** Sandy ?x) :– (**likes** ?x cats))

Mit Regeln können wir — im Gegensatz zu Fakten — auch *unendliche* Relationen definieren und berechnen.

Semantik der Regeln

Die Regel hat zwei Interpretationen:

(\leftarrow (**likes** Sandy ?x) :– (**likes** ?x cats))

Deklarativ: Als logische Zusicherung bedeutet die Regel:

Für alle x gilt: Sandy mag x , wenn x Katzen mag.

Operational (oder prozedural): Als Teil eines Prolog-Programms bedeutet die Regel:

Wenn Du beweisen willst, daß Sandy jemanden mag, versuche zu zeigen, daß dieser Katzen mag.

Rückwärtsverkettung

- Die operationale Interpretation heißt *Rückwärtsverkettung* oder *backward chaining*, da wir ausgehend vom Ziel rückwärts die Voraussetzungen zu beweisen versuchen.
- Die Relationen lassen prinzipiell auch die Gegenrichtung zu (forward chaining), aber Prolog verwendet nur *backward chaining*.
- Die \leftarrow Notation betont diese Semantik:

Der Klausel-Operator \leftarrow symbolisiert sowohl die logische Implikation \leftarrow als auch die Richtung des *backward chaining*.

Die Komponenten einer Klausel

(`<- (likes Sandy ?x) :- (likes ?x cats)`)

- Der linke Teilausdruck heißt der *Kopf* der Klausel.
- Die Teilausdrücke rechts vom `:-`-Operator sind der *Körper* der Klausel.
- Fakten können als spezielle Klauseln ohne Körper betrachtet werden, die ohne Vorbedingungen immer wahr sind.

Konjunktion von Zusicherungen

Eine Klausel sichert zu, daß der Kopf der Klausel wahr ist, wenn *alle* Vorbedingungen im Klauselkörper wahr sind. Beispiel:

(`<- (likes Kim ?x)`
 `:- (likes ?x Lee)`
 `(likes ?x Kim))`)

Für alle `?x`, schließe daß Kim `?x` mag, wenn bewiesen werden kann,

- daß `?x` Lee mag *und*
- daß `?x` Kim mag.

☞ Anmerkung: Eine Implementation der Datenbasis finden Sie in der SE3-Bibliothek im Unterverzeichnis prolog, file: prologDB.ss., siehe <http://kogs-www.informatik.uni-hamburg.de/~dreschle/informatik/Skripte/se3-bib.zip> Abweichen von Norvigs Implementation wurde die Speicherung der Regeln über Hash-Tabelle realisiert, da sein Ansatz mit Property-Listen in Racket leider nicht direkt umsetzbar war.

34.3 Unifikation und Suche

Unifikation von logischen Variablen

- Logische Variablen werden über *Unifikation* gebunden.
- *Unifikation* ist eine Verallgemeinerung des pattern matching:
 - Pattern matching ist unsymmetrisch:
 - * Variable dürfen nur im Muster vorkommen, nicht im zu analysierenden Ausdruck.
 - Die *Unifikation* ist symmetrisch:
 - * Beide Ausdrücke dürfen Variable enthalten;
 - * Variable können sogar mit anderen Variablen unifizieren, so daß die Identität festgestellt werden kann.

Beispiel

```
> (pat-match '(?x + ?y)
              '(2 + 1) no-bindings)
   → ((?y . 1) (?x . 2))
> (unify '(?x + 1)
          '(2 + ?y) no-bindings)
   → ((?y . 1) (?x . 2))
```

Bedingungen für erfolgreiche Unifikation

Zwei Ausdrücke unifizieren,

- wenn die *Stelligkeit* stimmt,
- wenn die *Namen* der Prädikate stimmen,
- wenn gleiche Variablen gleiche Werte bezeichnen (*Koreferenz*).

Unifikation von Variablen

Der folgende Ausdruck stellt die Gleichheit zweier Variablen fest:

```
> (unify '(f ?x)
          '(f ?y) no-bindings)
   → ((?x . ?y))
```

- Variablen können auch unifizieren, wenn sie noch ungebunden sind.
- Sollte später eine der Variablen durch Unifikation an einen Wert gebunden werden, haben automatisch alle Variablen diesen Wert, mit denen sie unifizieren.

Variablenersetzung: unifier

Die Funktion *unifier* ersetzt die Variablen entsprechend ihren Bindungen.

```
> (unify '(?a + ?a = 0)
          '(?x + ?y = ?y) no-bindings)
   → ((?y . 0) (?x . ?y) (?a . ?x))
> (unifier '(?a + ?a = 0)
            '(?x + ?y = ?y))
   → (0 + 0 = 0)
```

Die Ersetzung ist rein lexikalisch – es erfolgt keine Auswertung der Terme.

```
> (unify '(?a + ?a = 2) '(?x + ?y = ?y) ....)
   → ((?y . 2) (?x . ?y) (?a . ?x))
> (unifier '(?a + ?a = 2) '(?x + ?y = ?y))
   → (2 + 2 = 2)
```

Unifikation in der Logikprogrammierung

Die Aufgaben der Unifikation

Die *Unifikation* dient in der Logikprogrammierung

- zum *Binden* von logischen Variablen an Werte,
- als *Selektor* für Teile einer Struktur,
- zur *Konstruktion* von Strukturen,
- zum *Test* auf Gleichheit.

 **Anmerkung:** Die Funktion „unify“ ist in der Struktur der pat-mach-Funktion sehr ähnlich, ebenfalls eine geschachtelte Baumrekursion. Das Schema mußte nur bezüglich des Musters und des abzugleichenden Ausdrucks symmetrisch gemacht werden und die Unifikation von zwei Variablen zugelassen werden.

Den Code finden Sie im Abschnitt 37.1 sowie im file „unify.ss“ in der se3-bib.

Programmieren in Prolog

Wir programmieren Prolog-Programme

- indem wir relevante Relationen als Klauseln in der Datenbasis zuschern (mit dem assert-Macro `<–`)
- und dann im Interpreter *Anfragen* an die Datenbasis stellen.
- Die Suche wird automatisch mit *Unifikation* und *backtracking* nach geeigneten Fakten und Regeln suchen, um die Anfrage zu beantworten.
- Die Suche merkt sich an allen Entscheidungspunkten die offenen Alternativen, so daß wir nach alternativen Lösungen fragen können (inkrementelles backtracking).
- Das Resultat einer Anfrage sind die Variablenbindungen, die die Anfrage wahr machen.
- Die Teilklauseln einer Anfrage heißen *Ziele*.

Der Interpreter

Um Anfragen in Racket als Prolog-Sprachelement kenntlich zu machen, definieren wir ein weiteres Macro: „`?-`“, das den Prolog-Interpreter aufruft.

```
(defmacro* (?- &rest goals)
  '(top-level-prove
    (replace-?-vars (quote ,goals))))
```

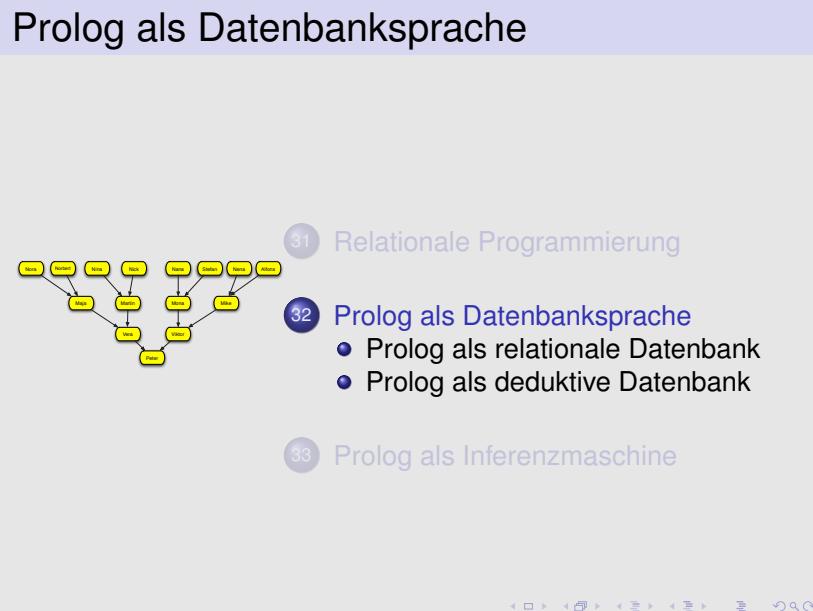
- Ersetze alle Variablen durch neue, eindeutige Name.
- Versuche die Anfragen (goals) als wahr zu beweisen, indem
 - die Anfrage mit den Datenbankeinträgen unifiziert wird
 - und rekursiv die Vorbedingungen für Regeln verifiziert werden.

Anzeige des Resultats

- Eine Anfrage gilt als *wahr*, wenn die Zusicherungen in der Datenbank dieses unterstützen, sonst als *falsch* (closed world assumption).
- Falls die Anfrage erfolgreich mit der Datenbank unifiziert werden konnte,
 - Drucke „Yes“.
 - Drucke die Variablenbindungen.
 - Frage die Benutzer, ob sie weitere Ergebnisse sehen möchten.
- Wenn der Interpreter nicht die Korrektheit der Anfrage anhand der Datenbank beweisen kann, wird „No“ geantwortet.

35 Prolog als Datenbanksprache

35.1 Prolog als relationale Datenbank

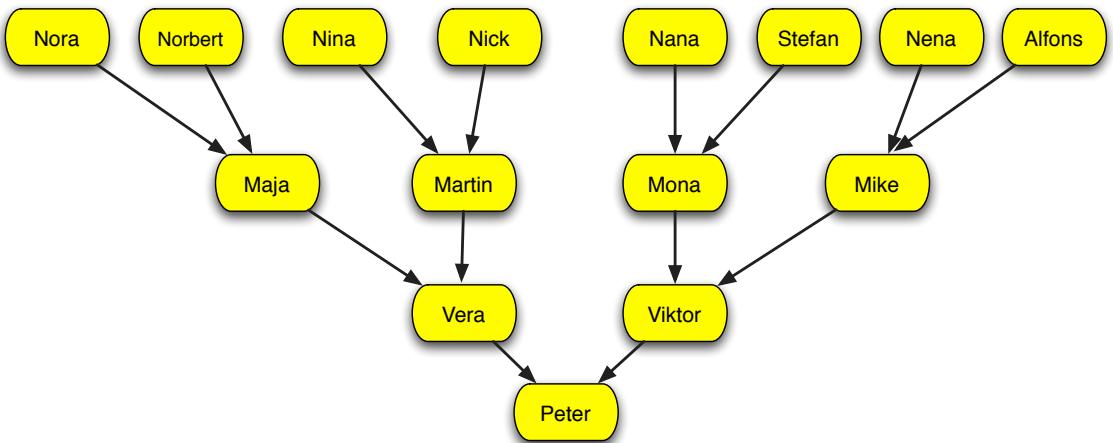


Beispiel: Familienverhältnisse

Beispiel: 150 (Relationen zu Familienverhältnissen)

- Dieses Beispiel zeigt, wie wir Prolog als relationale Datenbank nutzen können:
- Die Relation (`parents <mother> <father> <child>`) beschreibt, welches Kind von welchen Eltern abstammt.
- Mit dieser Relation können wir fragen, wer welche Kinder, Eltern, Enkel usw. hat.

Eine Ahnentafel



Die Ahnentafel als Fakten

```
; (parents mother father child)
; Die Eltern von Peter
(<- (parents Vera Viktor Peter))
; Die Großeltern von Peter
(<- (parents Maja Martin Vera))
(<- (parents Mona Mike Viktor))
; Die Urgroßeltern von Peter
(<- (parents Nora Norbert Maja))
(<- (parents Nina Nick Martin))
(<- (parents Nana Stefan Mona))
(<- (parents Nena Alfons Mike))

> (?- (parents ?mother ?father Peter))
?mother = Vera
?father = Viktor
; No.
```

Kombinierte Anfragen (join): Die Großeltern

Beim „join“ werden Prädikate über gemeinsame Variable in Beziehung gesetzt.

```
> (?- (parents ?mother ?father Peter)
  (parents ?grandma ?grandpa ?mother)
  (parents ?oma ?opa ?father))
```

```
?grandma = Maja
?grandpa = Martin
?mother = Vera
?oma = Mona
?opa = Mike
?father = Viktor
; No.
```

Aufzählen einer Relation

```
> (?- (parents ?mother ?father ?child))
?mother = Vera ?father = Viktor ?child = Peter ;
?mother = Maja ?father = Martin ?child = Vera ;
?mother = Mona ?father = Mike ?child = Viktor ;
?mother = Nora ?father = Norbert ?child = Maja ;
?mother = Nina ?father = Nick ?child = Martin ;
?mother = Nana ?father = Stefan ?child = Mona ;
?mother = Nena ?father = Alfons ?child = Mike ;
No.
```

Anonyme Variable

- Wenn eine Variable unwichtig ist,
- nicht zur Unifikation und Koreferenz benötigt wird
- und wir an ihrem Wert nicht interessiert sind,

können wir sie anonymisieren und ihr den Namen „?“ geben.

Anonyme Variable werden bei der Ausgabe unterdrückt.

```
> (?- (parents ?mother ? Walter))
?mother = Petra
; → No.
```

Projektionen

- *Projektionen* verringern die Dimension einer Relation.
- Wir können anonyme Variable oder Regeln nutzen, um Teile des Definitionsbereichs zu unterdrücken.

```
(<- (mother ?mother ?child)
  :- (parents ?mother ? ?child))
(<- (father ?father ?child)
  :- (parents ? ?father ?child))
```

35.2 Prolog als deduktive Datenbank

Ein Prolog-Sytem ist nicht nur eine programmierbare relationale Datenbank, es ist auch eine mächtige deduktive Datenbank. Mehr zu deduktiven Datenbanken aus der Sicht der logischen Programmierung finden Sie in ([Cremers et al., 1994](#)).

Beispiel: Vorfahren

Dieses Beispiel zeigt, wie wir Prolog über berechnete Relationen als *deduktive Datenbank* nutzen können:

Die Eltern sind entweder Vater oder Mutter.

Die Großeltern sind die Eltern der Eltern.

Vorfahren sind entweder Eltern oder Eltern von Vorfahren.

Die Regeln: Vorfahren

```
(<- (parent ?mother ?child)
     :- (parents ?mother ? ?child))
(<- (parent ?father ?child)
     :- (parents ? ?father ?child))

(<- (granny ?gran ?grandchild)
     :- (parent ?gran ?parent)
         (parent ?parent ?grandchild))

(<- (predecessor ?predc ?person)
     :- (parent ?predc ?person))

(<- (predecessor ?predc ?person)
     :- (parent ?predc ?parent)
         (predecessor ?parent ?person))
```

☞ Anmerkung: Mit berechneten Relationen können auch unendliche Relationen beschrieben werden, wenn wir Rekursion nutzen.

Anfragen

```
> (?- (granny ?gran Peter))
?gran = Maja
;
?gran = Mona
```

```
;  
?gran = Martin  
;  
?gran = Mike  
;  
No.  
>
```

Vorfahren und Nachkommen

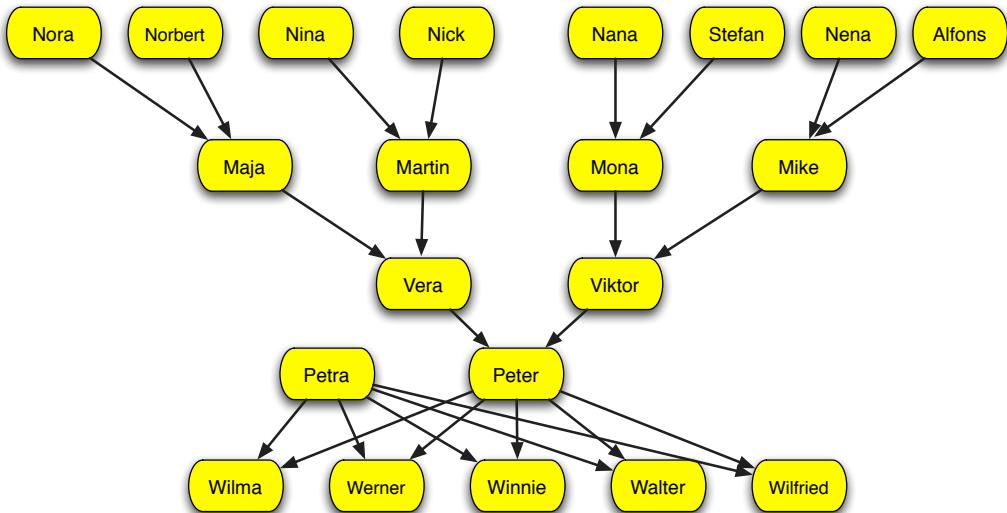
Die Vorfahren von Mike:

```
> (?- (predecessor ?pred Mike))  
?pred = Nena  
;  
?pred = Alfons  
;  
No.
```

Die Nachkommen von Maja:

```
> (?- (predecessor Maja ?desc))  
?desc = Vera  
;  
?desc = Peter  
;  
No.  
>
```

Die Nachkommen von Petra und Peter



Viele Geschwister

```
; Petras und Peters Kinder
(<- (parents Petra Peter Wilma))
(<- (parents Petra Peter Werner))
(<- (parents Petra Peter Winnie))
(<- (parents Petra Peter Walter))
(<- (parents Petra Peter Wilfried))

(<- (siblingA ?x ?y) :-
  (parents ?m ?f ?x)
  (parents ?m ?f ?y))
```

Anfrage. Die Geschwister von Walter

```
> (?- (siblingA ?x Walter))
?x = Wilma;
?x = Werner;
?x = Winnie;
?x = Walter;
?x = Wilfried;
No.
```

```
>
```

Unschön, *Walter* wird als sein eigener Bruder gemeldet.

Wir müssen ausdrücken können, daß der Bruder einer Person ungleich der Person ist.

Gleichheit und Ungleichheit

- Zwei Strukturen sind gleich, wenn sie unifizieren.
- Wir können eine Gleichheitsrelation „=“ daher so definieren:

```
(<- (= ?x ?x)) ; do the clauses unify?
```

- Zwei Strukturen sind ungleich, wenn sie nicht unifizieren.
- Wir können eine Ungleichheitsrelation „!=“ mittels des eingebauten „not“-Operators definieren.

```
(<- (!= ?x ?y) :-  
      (not = ?x ?y) )
```

Geschwister, Version 2

```
(<- ( sibling ?x ?y) :-  
      (parents ?m ?f ?x)  
      (parents ?m ?f ?y)  
      (!= ?x ?y) )
```

```
> (?- ( sibling ?x Walter ))  
?x = Wilma ;  
?x = Werner ;  
?x = Winnie ;  
?x = Wilfried ;  
No.
```

Walter ist jetzt nicht mehr sein eigener Bruder.

Logische Operatoren in Prolog(-in-Racket)

- ∧: Die *Konjunktion* wird implizit in der Liste aller Prämissen einer Regel ausgedrückt.
- ∨: Die *Disjunktion* wird implizit in alternativen Regeln für dasselbe Prädikat ausgedrückt.
- ¬: Die *Negation* wird durch den not-Operator ausgedrückt: **not** <Klausel> ist erfolgreich, wenn die Klausel nicht erfolgreich mit der Datenbasis unifiziert.

Jedes „fail“ (Nichterfolg der Unifikation mit der Datenbasis) ist implizit eine Negation.

Gültigkeitsbereich von Variablen

- Der Gültigkeitsbereich einer Variablen ist die Klausel, in der die Variable eingeführt wurde.
- Es gibt keine globalen Variablen und keine geschachtelte Blockstruktur.
- Eine Variable kann anfänglich undefiniert sein.
- Wenn eine Variable durch Unifikation gebunden wurde, behält sie ihren Wert und kann nicht wieder geändert werden.

Beispiel: member, Rekursion über Listen

Beispiel: 151 (Die member-Relation)

Ein Element `?item` steht in der Relation `member` zu einer Liste `xs`, wenn `?item` ein Element der Liste `xs` ist.

- Ein Element der Liste ist entweder das erste Element,
- oder es ist ein Element der Restliste.

```
(<- (member ?item (?item . ?rest)))
(<- (member ?item (?x . ?rest))
     :- (member ?item ?rest))
```

Beispielanfragen

```
> (?- (member 1 (2 1 3)))
Yes
No.
> (?- (member 1 (2 1 3 1)))
Yes
;
Yes
;
No.
>
```

Konstruktion einer Liste

```
> (?- (member 1 (?a ?b ?c)))
?a = 1
?b = ?b
?c = ?c
;
?a = ?x4412
?b = 1
?c = ?c
;
?a = ?x4412
?b = ?x4417
?c = 1
;
No.
>
```

Prädikate zweiter Ordnung

- Prädikaten 2. Ordnung erhalten Prädikatsaufrufe (Ziele) als Eingabewerte.
- Mit Prädikaten 2. Ordnung können wir aggregierende Aussagen über die Fakten in der Datenbank machen, beispielsweise alle Strukturen, die ein Prädikat erfüllen, in einer Liste zusammenfassen.
- In dieser einfachen Prolog-in-Racket Version ist nur das Prädikat *findall* als Prädikat 2. Ordnung implementiert.

findall

(**findall** <Term> <Ziel> <Liste>)

findall sammelt alle Resultate des Prädikatsaufrufs <Ziel> in einer Liste <Liste> als instanzierte Varianten des Ausdrucks <Term>.

Der Term und das Ziel sind über gemeinsame uninstantiierte Variable verknüpft.

Beispiel: Vorfahren und Geschwister

```
(<- (allAncestors ?person ?ancs) :-  
  (findall ?predc  
   (predecessor ?predc ?person) ?ancs))  
  
> (?- (allAncestors Walter ?ancs))  
?ancs = (Alfons Stefan Nick Norbert Mike  
Martin Viktor Nena Nana Nina Nora Mona  
Maja Vera Peter Petra)  
; No.  
  
> (?- (findall ?sibl  
        (sibling ?sibl Walter) ?siblings))  
?sibl = ?sibl  
?siblings = (Wilfried Winnie Werner Wilma)  
;  
No.
```

36 Prolog als Inferenzmaschine

An den beiden folgenden Beispielen können Sie sehen, wie wir Prolog zum Schlußfolgern und Lösen von Logikrätseln einsetzen können.

Außerdem sind sie nochmals ein schönes Beispiel für die Richtungsunabhängigkeit von Prolog. Beachten Sie besonders die Verwendung des member-Prädikats: Vorrangig dient dieses Prädikat ja zum Prüfen der Mitgliedschaft in einer Liste, aber hier ist es in erster Linie ein Konstrukt, der bestimmte Elemente in die Listen einfügt, so daß diese anschließend das member-Prädikat erfüllen.

36.1 Das Zebra-Rätsel

Prolog als Inferenzmaschine



- 31 Relationale Programmierung
- 32 Prolog als Datenbanksprache
- 33 Prolog als Inferenzmaschine
 - Das Zebra-Rätsel
 - Severus Snapes Rätsel
 - Funktionale Sprachelemente in Prolog

Beim Zebra-Rätsel besteht die Aufgabe darin, fünf Häusern ihre Bewohner und Haustiere zuzuordnen. Jeder Bewohner hat eine Nationalität und bestimmte Trink- und Rauchgewohnheiten. Die Häuser haben alle unterschiedliche Farben. Es sind Randbedingungen an die Lösung vorgegeben, die wir hier als Regeln formulieren werden.

Wir geben hier die Originallösung von Peter Norvig wieder.

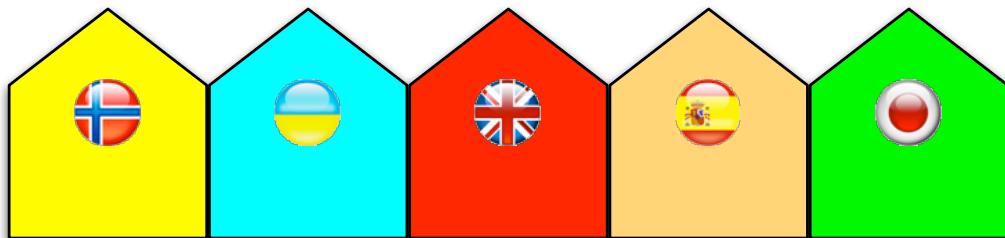
Das Zebra-Rätsel

Beispiel: 152

Es gibt fünf Häuser in einer Reihe und bestimmte Regeln, die die Bewohner, ihre Nationalität, ihre Haustiere und ihre Rauch-und Trinkgewohnheiten einschränken.

Die Frage ist:

- Wo wohnt der Wassertrinker?
- Wer hat ein Zebra als Haustier?



Die Regeln

Regel 1: Jedes Haus hat eine andere Farbe.

Jeder Besitzer der fünf Häuser hat

- eine andere Nationalität,
- ein anderes Lieblingsgetränk,
- ein anderes Haustier,
- eine eigene Zigarettenmarke.

Regel 2: Der Engländer lebt im roten Haus.

Regel 3: Der Spanier hat einen Hund.

Regel 4: Im grünen Haus trinkt man Kaffee.

Regel 5: Der Ukrainer trinkt Tee.

Regel 6: Das grüne Haus ist direkt rechts neben dem elfenbeinfarbenen Haus.

Regel 7: Der Winston-Raucher hält Schnecken.

Regel 8: Im gelben Haus wird Cools geraucht.

Regel 9: Im mittleren Haus wird Milch getrunken.

Regel 10: Der Norweger lebt im ersten Haus von links.

Regel 11: Der Chesterfield-Raucher ist ein direkter Nachbar vom Mann mit dem Fuchs.

Regel 12: Der Cools-Raucher ist ein direkter Nachbar vom Pferdebesitzer.

Regel 13: Der Lucky-Strike-Raucher trinkt Orangensaft.

Regel 14: Der Japaner raucht Parliaments.

Regel 15: Der Norweger wohnt neben dem blauen Haus.

; ; ; ; —*— Mode: Lisp ; Syntax: Common—Lisp —*—
; ; ; ; Code from Paradigms of AI Programming
; ; ; ; Copyright (c) 1991 Peter Norvig

Die Datenstruktur

Die Häuserzeile wird als Liste mit fünf Elementen repräsentiert.

Die Häuser sind Strukturen (Listen), die mit dem Wort „house“ beginnen und als weitere Elemente die Eigenschaften aufzählen :

```
(house
  nationality
  pet
  cigarette
  drink
  house-color)
```

Prädikate: next-to (benachbart), irth (rechts-von)

Stützprädikate

```
(<- (nextto ?x ?y ?list)
      :- (iright ?x ?y ?list))
(<- (nextto ?x ?y ?list)
      :- (iright ?y ?x ?list))

(<- (iright ?left ?right
      (?left ?right . ?rest)))
(<- (iright ?left ?right (?x . ?rest))
      :- (iright ?left ?right ?rest))
(<- (= ?x ?x))
```

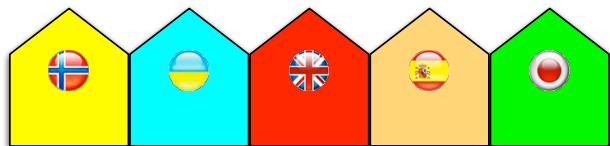
Einschränkende Relationen

```
(<- (zebra ?h ?w ?z) :-
; (house nation pet cig drink house-color)
(= ?h ((house norwegian ? ? ? ?) ; 1,10
?
(house ? ? ? milk ?) ; 9
?
?))
(member (house englishman ? ? ? red) ?h); 2
(member (house spaniard dog ? ? ?) ?h); 3
(member (house ? ? ? coffee green) ?h); 4
(member (house ukrainian ? ? tea ?) ?h); 5
(iright (house ? ? ? ? ivory) ; 6
(house ? ? ? green) ?h)

(member (house ? snails winston ? ?) ?h) ; 7
(member (house ? ? kools ? yellow) ?h) ; 8
(nextto (house ? ? chesterfield ? ?) ; 11
(house ? fox ? ? ?) ?h)
(nextto (house ? ? kools ? ?) ; 12
(house ? horse ? ? ?) ?h)
(member (house ? ? luckyStrike oJuice ?)
?h) ; 13
(member (house japanese ? parliaments ? ?)
?h) ; 14
(nextto (house norwegian ? ? ? ?) ; 15
(house ? ? ? ? blue) ?h)
(member (house ?w ? ? water ?) ?h) ; Q1
(member (house ?z zebra ? ? ?) ?h)) ; Q2
```

Die Anfrage

```
(?- (zebra
    ?houses ?water-drinker ?zebra-owner))
?houses = (
(house norwegian fox kools water yellow)
(house ukrainian horse chesterfield tea blue)
(house englishman snails winston milk red)
(house spaniard dog luckystrike oJuice ivory)
(house japanese zebra parliaments coffee green))
?water-drinker = norwegian
?zebra-owner = japanese
; No.
```



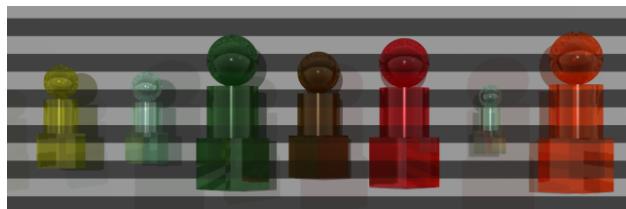
36.2 Severus Snapes Rätsel

Das folgende Rätsel stammt aus „Harry Potter und der Stein der Weisen“([Rowling, 1997](#)):

Beispiel: 153

Harry und Hermione sind in einen Raum gefangen; hinter ihnen brennt ein tödliches schwarzes Feuer, vor ihnen ein rotes. Im Raum stehen sieben Flaschen mit geheimnisvollen Flüssigkeiten aufgereiht, und sie finden ein Rätsel vor: Sie müssen das Rätsel lösen, um die beiden Zaubertränke, die ihnen durch die Flammen helfen können, sicher vom Gift zu unterscheiden.

Severus Snapes Rätsel



Danger lies before you, while safety lies behind.
Two of us will help you, whichever you would find.
One among us seven will let you move ahead,
Another will transport the drinker back instead,
Two among our numbers hold only nettle wine,
Three of us are killers, waiting hidden in line.
Choose, unless you wish to stay here forevermore,

To help you in your choice, we give you these clues four:
First, however slyly the poison tries to hide
You will always find some on nettle wine's left side.
Second, different are those who stand at either end,
But if you would move onward, neither is your friend.
Third, as you see clearly, all are different size,
Neither dwarf nor giant holds death in their insides;
Fourth, the second left and the second on the right
Are twins once you taste them, though different at first sight.

aus: „Harry Potter und der Stein der Weisen“

Die Datenstrukturen

- Datenstruktur für eine Flasche:
eine Liste der Form: (?name ?inhalt)
- Die Reihe der sieben Flaschen:
eine Liste mit sieben Elementen
- Die Flaschenliste wird wieder mittels *member* konstruiert.

Konstruktion der Liste der Flaschen:

Anzahl und Art

```
; genau drei Giftflaschen
(<- (giftflaschen ?flaschen) :-
  (member (Gift1 Gift) ?flaschen)
  (member (Gift2 Gift) ?flaschen)
  (member (Gift3 Gift) ?flaschen))
; genau zwei Flaschen mit Nesselwein
(<- (nesselwein ?flaschen) :-
  (member (Wein1 Wein) ?flaschen)
  (member (Wein2 Wein) ?flaschen))
(<- (traenke ?flaschen) :-
  (member (TrankVoraus TrankVoraus)
    ?flaschen)
  (member (TrankZurueck TrankZurueck)
    ?flaschen))
```

Beziehungen zwischen den Flaschen

```
(<- (riese ( ? ?inhalt)) :- (!= ?inhalt Gift))
(<- (zwerg ( ? ?inhalt)) :- (!= ?inhalt Gift))

(<- (verschieden ( ? ?inhalt1)( ? ?inhalt2))
  :- (!= ?inhalt1 ?inhalt2))
(<- (zwilling ( ? ?inhalt)( ? ?inhalt)))
(<- (keinFreundVoraus ( ? ?inhalt)) :- (!= ?inhalt TrankVoraus))
```

Das Rätsel: Variante 1

```
(<- (raetsel ?flaschen) :-  
; Anzahl und Art der Flaschen  
 (= ?flaschen (?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7 ))  
 (giftflaschen ?flaschen)  
 (nesselwein ?flaschen)  
 (traenke ?flaschen)  
 ; 1. links vom Nesselwein steht Gift  
 (iright (? Gift) (Wein1 Wein) ?flaschen)  
 (iright (? Gift) (Wein2 Wein) ?flaschen)  
 ; 2a. verschieden an den Enden  
 (verschieden ?f1 ?f7)  
 ; 2b. kein Freund, wenn man weiter will  
 (keinFreundVoraus ?f1)  
 (keinFreundVoraus ?f7)  
 ; 4. zwei links, zwei rechts Zwilling  
 (zwilling ?f3 ?f5) )
```

Ein Probelauf: Variante 1

Lösung 1:

```
> (?- (raetsel (?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7 )))  
?f1 = (Gift1 Gift)  
?f2 = (Gift2 Gift)  
?f3 = (Wein1 Wein)  
?f4 = (Gift3 Gift)  
?f5 = (Wein2 Wein)  
?f6 = (TrankVoraus TrankVoraus)  
?f7 = (TrankZurueck TrankZurueck)  
;
```

Lösung 2:

```
;  
?f1 = (Gift1 Gift)  
?f2 = (Gift2 Gift)  
?f3 = (Wein2 Wein)  
?f4 = (Gift3 Gift)  
?f5 = (Wein1 Wein)  
?f6 = (TrankVoraus TrankVoraus)  
?f7 = (TrankZurueck TrankZurueck)  
;
```

Lösung 4

```
;  
?f1 = (Gift1 Gift)  
?f2 = (Wein1 Wein)  
?f3 = (Gift2 Gift)  
?f4 = (TrankVoraus TrankVoraus)  
?f5 = (Gift3 Gift)  
?f6 = (Wein2 Wein)  
?f7 = (TrankZurueck TrankZurueck)
```

Lösung 20

```
?f1 = (TrankZurueck TrankZurueck)  
?f2 = (Gift1 Gift)  
?f3 = (Wein1 Wein)  
?f4 = (Gift2 Gift)  
?f5 = (Wein2 Wein)  
?f6 = (TrankVoraus TrankVoraus)  
?f7 = (Gift3 Gift)
```

Die Mehrdeutigkeiten sind leider lebensgefährlich.

Wir müssen noch die *Größen* der Flaschen berücksichtigen.

☞ Anmerkung: J. K. Rowling war mit dem Rätsel sehr unfair zu uns Leserinnen und Lesern, da wir die Flaschen ja nicht sehen können und nicht wissen, welches die größte und die kleinste Flasche sind. Erst aus der Auflösung des Rätsels erfahren wir, daß die kleinste Flasche irgendwo zwischen den anderen Flaschen steht.

Mit diesem Wissen können wir experimentieren und Annahmen über den Ort der größten Flasche und kleinsten Flasche machen, und wir stellen fest, daß das Rätsel dann eine eindeutige Lösung hat, wenn wir annehmen,

- daß eine große Flasche ganz rechts steht
- und eine kleine Flasche auf Position 3.

Wir bekommen zwar immer noch mehrere Lösungen, aber bei allen Lösungen stehen die gesuchten Flaschen an einem festen Platz.

Variante 2: Groessen der Flaschen bekannt

```
(<- (raetsel ?flaschen) :-  
    (= ?flaschen (?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7))
```

```

(giftflaschen ?flaschen)
(nesselwein ?flaschen)
(traenke ?flaschen)
(iright (? Gift) (Wein1 Wein) ?flaschen)
(iright (? Gift) (Wein2 Wein) ?flaschen)
(verschieden ?f1 ?f7)
(keinFreundVoraus ?f1)
(keinFreundVoraus ?f7)
; 3. Zwerge und Riese ungiftig
(riese ?f7)
(zwerg ?f3);
(zwilling ?f3 ?f5))

```

Probelauf: Variante 2

```

> (?- (raetsel (?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7)))
?f1 = (Gift1 Gift)
?f2 = (Gift2 Gift)
?f3 = (Wein1 Wein)
?f4 = (Gift3 Gift)
?f5 = (Wein2 Wein)
?f6 = (TrankVoraus TrankVoraus)
?f7 = (TrankZurueck TrankZurueck)

```

36.3 Funktionale Sprachelemente in Prolog

Arithmetik

- Wenn wir in Prolog Arithmetik betreiben wollen, benötigen wir Relationen über Zahlen und Zahlobjekte.
- Beides könnte rekursiv über die Peano-Axiome definiert werden, beispielsweise über Strichlisten.
- Dieser Ansatz ist aber rechenaufwendig, da dann der Aufwand für die Verknüpfung von Zahlen proportional zu deren Größe wäre.

Die arithmetische Auswertungsumgebung

Prolog bietet die Möglichkeit, arithmetische Operationen funktional durchzuführen:

- Auch in unserem Prolog in Racket haben wir ein Schlupfloch in die funktionale Programmierung eingebaut:
- Der *is-Operator* wertet arithmetische Ausdrücke funktional aus und bindet das Resultat an eine Variable.
- Der *test-Operator* wertet boolsche Ausdrücke funktional aus. Die Unitifikation ist erfolgreich, sofern der Ausdruck „wahr“ ergibt.
- Alle Variablen des Ausdrucks müssen *instanziert* sein.
- Die *Richtungsunabhängigkeit* geht verloren!!!

Beispiel für funktionale Klauseln

```
> (?- (is ?x (+ 2 4)))
?x = 6
; No.
> (?- (is ?x 3)(is ?y 5)(test (> ?y ?x)))
?y = 5
?x = 3
; No.
```

Verbindung von Racket und Prolog

```
> (?- (is ?xs (map add1 '(1 2 3))))
?xs = (2 3 4)
; No.
> (?- (is ?xs (map add1 '(1 2 3)))
      (member ?x ?xs))
?x = 2
?xs = (2 3 4)
;
?x = 3
?xs = (2 3 4)
;
?x = 4
?xs = (2 3 4)
; No.
```

☞ Anmerkung: Mittels des is-Operators und des test-Operators können wir beliebige Werte funktional in Racket berechnen, diese Werte an Prolog-Variable binden und dann mit Prädikaten weiterverarbeiten. Aber der Preis dafür ist hoch: Funktionale Prädikate können nur gerichtet benutzt werden, da alle Variablen der funktionalen Teilziele vorher gebunden werden müssen.

Eine length-Relation

```
; (length list length-of-x)
(<- (length () 0))
(<- (length (?x . ?rest) ?len) :-
    (length ?rest ?lenR)
    (is ?len (+ 1 ?lenR)))

> (?- (length (a b c) ?len))
```

```
?len = 3  
;  
No.
```

Prolog in Racket

Das „Prolog in Racket“-System besteht aus den Modulen

- prologDB.ss: Die Datenbasis
- unify.ss: Die Unifikation
- prolog.ss: Interaktion und Backtracking
- prologInScheme.ss: ein Paket, das alle Module bündelt.

Die files liegen in der se3-bib im Verzeichnis se3-bib/prolog.

37 Ein Prologinterpret in Racket

Beispielauf

```
Welcome to DrScheme, version 4.2.3 [3m].  
Language: Module; memory limit: 256 megabytes.  
> (require se3–bib/prolog/prologInScheme)  
> (<– (parents Vera Viktor Peter)); Fakten  
> (?– (parents ?m ?f Peter)); Anfrage  
  
?m = Vera  
?f = Viktor  
; No.  
>
```

37.1 Die Unifikation: unify.ss

siehe (Norvig, 1992)

```
(module unify (lib "swindle.ss" "swindle")  
; ;;; Peter Norvigs Common Lisp "unify"  
; ;;; Code from Paradigms of AI Programming  
; ;;; Copyright (c) 1991 Peter Norvig  
; ;;; ported to Scheme by Leonie Dreschler-Fischer  
  
(provide *occurs–check*  
    unify unifier  
    unify–variable subst–bindings  
)  
  
(require  
    swindle/setf swindle/misc  
    (all–except se3–bib/tools–module  
        mappend every some last concat )  
    se3–bib/pattern–matching–module)  
  
(define *occurs–check* #t); "Should we do the occurs check?"  
  
(define (unify x y &optional (bindings no–bindings))  
    "See if x and y match with given bindings."  
    (cond ((eqv? bindings fail) fail)
```

```

((equal? x y) bindings) ;eq/
((variable? x) (unify-variable x y bindings))
((variable? y) (unify-variable y x bindings))
((and (pair? x) (pair? y)); consp
; (writeln "trying pair: " x y bindings)
  (unify (cdr x) (cdr y)
         (unify (car x) (car y) bindings)))
(else fail)))

(define (unify-variable var x bindings)
  "Unify_var_with_x,_using_(and_maybe_extending)_bindings."
  (cond ((get-binding var bindings)
          (unify (lookup var bindings) x bindings))
        ((and (variable? x) (get-binding x bindings))
         (unify var (lookup x bindings) bindings))
        ((and *occurs-check* (occurs-check var x bindings))
         fail)
        (else (extend-bindings var x bindings)))))

(define (occurs-check var x bindings)
  "Does_var_occur_anywhere_inside_x?"
  (cond ((eq? var x) #t)
        ((and (variable? x) (get-binding x bindings))
         (occurs-check var (lookup x bindings) bindings))
        ((pair? x) (or (occurs-check var (first x) bindings)
                        (occurs-check var (rest x) bindings)))
        (#t #f)))

;; =====

(define (subst-bindings bindings x)
  "Substitute_the_value_of_variables_in_bindings_into_x,
  taking_recursively_bound_variables_into_account."
  (cond ((eq? bindings fail) fail)
        ((eq? bindings no-bindings) x)
        ((and (variable? x) (get-binding x bindings))
         (subst-bindings bindings (lookup x bindings)))
        ((atom? x) x)
        (else (cons (subst-bindings bindings (car x))
                    (subst-bindings bindings (cdr x)))))))

```

```

(define (unifier x y)
  "Return something that unifies with both x and y (or fail)."
  (subst-bindings (unify x y) x))

)

```

37.2 Die Datenbasis: prologDB.ss

Dieser Teil konnte nicht von Norvig, 1992 übernommen werden, da Norvigs Datenstruktur sehr stark auf den Properties von Lisp-Symbolen aufbaut.

```

(module prologDB (lib "swindle.ss" "swindle")
  ;; a database for predicates and clauses
  ;; Leonie Dreschler-Fischer

  (require
    swindle/setf swindle/misc
    (all-except se3-bib/tools-module
      mappend every some last concat )
    se3-bib/prolog/unify
    se3-bib/pattern-matching-module)

  (provide
    *database* <Database>
    predic-clauses <Clause>
    set-predic-clauses!
    replace-?-vars)

  ; primitive classes
  (define <Database> <hash-table>)
  (define <Clause> <list>)
  (define <Relation> <list>)

  (define *database* (make-hash-table))

  ; a hashtable of all predicates stored in the database."
  ; key: the predicate symbol,
  ; value: a predic-structure representing the clauses

```

```

(defclass* predic ()
  (predSym :reader theSymbol
            :initarg :thesym
            :type <symbol>
            :documentation
            "The_symbol_of_the_predicate")
  (clauses :reader predic-clauses
            :writer set-predic-clauses!
            :initarg :theClaus
            :initvalue '()
            :type <Clause>; <list>
            :documentation
            "The_clauses_of_the_predicate")
  :autopred #t ;
  :printer #t
  :documentation
  "a_predicate_and_its_clauses"
)

(defclass* toplevel-predic (predic)
  (clauses :initvalue (thunk (writeln "Scheme-Prolog"))
            :type <function>
            :documentation
            "A_simple_function")
  :autopred #t ; auto generate predicate queue?
  :printer #t
  :documentation
  "a_predicate_and_its_clauses"
)
;; a data structure for predicates

(defgeneric* clear-db ((db <Database>)))

(defgeneric* store-predicate!
  ((newPredi <symbol>)(db <Database>)))

(defgeneric* store-toplevel-predicate!
  ((newPredi <symbol>)
   (fu <function>)
   (db <Database>)))

```

```

(defgeneric* retrieve-predicate!
  ((pred <symbol>) (db <Database>)))

(defgeneric* clear-predicate ((pred <symbol>)))

(defgeneric* clause-head ((clause <Clause>)))

(defgeneric* clause-body ((clause <Clause>)))

(defgeneric* get-clauses ((pred <top>)))

(defgeneric* store-clause!
  ((clause <Clause>) (db <Database>)))

(defgeneric* add-clause!
  ((pred <symbol>) (clause <Clause>)))

(defgeneric* predicate
  ((relation <Relation>)))

(defgeneric* args ((x <Relation>)))

(defmethod store-predicate!
  ((newPredi <symbol>) (db <Database>))
  ; create an empty predicate place in the database
  ; for the symbol of the predicate
  (unless (retrieve-predicate! newPredi db)
    (let ((predi(make predic
                      :thesym newPredi ; the name of the pred.
                      )))
      (hash-table-put! db newPredi predi)
      newPredi)))

(defmethod store-toplevel-predicate!
  ((newPredi <symbol>) (fu <function>) (db <Database>))
  ; create a predicate place in the database
  ; for the symbol of the predicate, store the function
  (unless (retrieve-predicate! newPredi db)
    (let ((predi
           (make toplevel-predic
                 :thesym newPredi ; the name of the predicate
                 )))))

```

```

        :theClauses fu ; the function
      )))
(hash-table-put! db newPredi predi)
newPredi)))

(defmethod retrieve-predicate!
  ((pred <symbol>) (db <Database>))
  ; retrieve the predicate from the database
  ; using the symbol of the predicate as key
  (hash-table-get db pred #f))

#| implement the clauses |#
;; clauses are represented as (head . body) cons cells
(defmethod clause-head ((clause <Clause>))
  (first clause))

(defmethod clause-body ((clause <Clause>))
  (rest clause))

;; clauses are stored on the predicate object
(defmethod get-clauses ((pred ???))
  ; return nil if the predicate is undefined
  '())

(defmethod get-clauses ((predS <symbol>))
  ; retrieve the clauses from the symbol
  (let ((pred (retrieve-predicate! predS
                                    *database*)))
    (when (not pred)
      (error "undefined_predicate:_" predS))
    (predic-clauses pred)))

(defmethod get-clauses ((predO predc))
  ; retrieve the clauses from the object
  (predic-clauses predO))

(defmethod add-clause!
  ((predS <symbol>) (clause <Clause>))
  (let ((predO
         (retrieve-predicate!
          predS

```

```

        *database*)))
(set-predic-clauses!
predO
(append
; preserve the order of the clauses
(predic-clauses predO)
(list clause)))))

#| implement the relations |#
;; relations
(defmethod predicate ((relation <Relation>))
(first relation))

(defmethod args ((x <Relation>))
"The arguments of a relation"
(rest x))

(define (replace-?-vars exp)
"Replace any ? within exp with a var of the form ?123."
(cond ((eq? exp '?) (gensym "?"))
((atom? exp) exp)
(else (cons (replace-?-vars (first exp))
(replace-?-vars (rest exp)))))))

(defmacro* (<- &rest clause)
;add a clause to the data base.
'(store-clause!
(replace-?-vars (quote ,clause))
*database*))

(defmethod store-clause!
((clause <symbol>) (db <Database>))
#t)

(define (remove-syntactical-sugar clause)
; remove infix operator :-
(if (pair? clause)
(filter (lambda (c)
(not (eq? ':- c)))
clause)
clause)))

```

```

(defmethod store-clause!
 ((clause <Clause>) (db <Database>))
;add a clause to the data base,
;indexed by head's predicate."
;; the predicate must be a non-variable symbol.
(let ((pred (predicate (clause-head clause))))
  (amb-assert
   (and (symbol? pred)
          (not (variable? pred))))
   (store-predicate! pred db)
   (add-clause!
    pred
    (remove-syntactical-sugar clause)))
;pred))

(defmethod clear-predicate ((predS <symbol>))
 "remove_the_clauses_for_a_single_predicate."
(let ((predO
        (retrieve-predicate!
         predS
         *database*)))
  (set-predic-clauses! predO '())))

(defmethod clear-db ((db <Database>))
 "remove_all_clauses_(for_all_predicates)_from_the_data_base."
(hash-table-for-each
 (singleton-value db)
 clear-predicate))
); end module

```

37.3 Backtracking und Interaktion: prolog.ss

```

(module prolog (lib "swindle.ss" "swindle")
 ;;; Peter Norvigs Common Lisp "prolog"
 ;;; Code from Paradigms of AI Programming
 ;;; Copyright (c) 1991 Peter Norvig
 ;;; ported to Scheme by Leonie Dreschler-Fischer
 ;;; is, not, test by Leonie Dreschler-Fischer

```

```

(provide
  *occurs–check*
  rename–variables
  adjoin
  unique–find–anywhere–if
  find–anywhere–if
  prove–all
  prove
  prove–not
  top–level–prove
  show–prolog–vars
  toplevel–predic
  continue?
  variables–in
  non–anon–variable?
  )

(require
  swindle/setf swindle/misc
  (all–except se3–bib/tools–module
   mappend every some last concat )
  se3–bib/prolog/unify
  se3–bib/prolog/prologDB
  se3–bib/pattern–matching–module)

(define (rename–variables x)
  ;replace all variables in x with new ones.
  (let ((newBindings (map
    (lambda (var)
      (cons var
        (gensym (symbol–>string var))))
      (variables–in x))))
    (subtree newBindings x))) ;sublis

(define (adjoin item xs)
  (if (not (member item xs)))
    (cons item xs)
    xs))

(define (unique–find–anywhere–if
  predicate tree

```

```

    &optional (found-so-far '()))
; return a list of leaves of tree satisfying predicate,
; with duplicates removed.

(if (atom? tree)
  (if (predicate tree)
      (adjoin tree found-so-far)
      found-so-far)
  (unique-find-anywhere-if
   predicate
   (first tree)
   (unique-find-anywhere-if predicate (rest tree)
                             found-so-far)))))

(define (find-anywhere-if
         predicate tree)
; does predicate apply to any atom in the tree?
(if (atom? tree)
  (predicate tree)
  (or (find-anywhere-if predicate (first tree))
      (find-anywhere-if predicate (rest tree)))))

(define (prove-all goals bindings)
; Find a solution to the conjunction of goals.
(cond ((equal? bindings fail) fail)
      ((null? goals) bindings)
      (else (prove (car goals) bindings (cdr goals))))))

(define (prove goal bindings other-goals)
; Return a list of possible solutions to goal.
(let ((clauses (get-clauses (predicate goal)))))

  (if (list? clauses)
      (some
       (lambda (clause)
         (let ((new-clause (rename-variables clause)))
           (prove-all
            (append (clause-body new-clause) other-goals)
            (unify goal (clause-head new-clause) bindings))))
       clauses)
      (list)))
  )
)
```

```

;; The predicate's "clauses" can be an atom:
;; a primitive function to call
(clauses (cdr goal) bindings
          other-goals)))

(define (prove-not goal bindings other-goals)
; a built-in "not", author L. D.-F.
(if (prove goal bindings '())
#f
(prove-all other-goals bindings)))

(define (is goal bindings other-goals)
; Builtin predicate is: (is <var> <expr>)
; evaluate a functional expression <expr>
; and bind the result to a variable <var>.
; author L. D.-F.
(let* ((clause
        (subst-bindings bindings goal))
       (theVar (car clause))
       (theExpr (cadr clause))
       (result
        (begin
         (writeln "is:_bindings:_" bindings)
         (writeln "is:_goal_" goal)
         (writeln "is:_Var_" theVar)
         (writeln "is:_Expr_" theExpr)

         (eval theExpr)))
       (newB (extend-bindings theVar result bindings)))
       (writeln "is:_goal_" goal)
       (writeln "is:_Var_" theVar)
       (writeln "is:_Expr_" theExpr)
       (writeln "is:_result_" result)
       (prove-all other-goals newB)))

(define (test goal bindings other-goals)
; Builtin predicate is: (test <expr>)
; succeeds if the functional expression <expr>
; evaluates to #t
; author L. D.-F.
(let* ((clause

```

```

        (subst-bindings bindings goal))
(theExpr (car clause))
(result
(begin
  (writeln "is:_bindings:_" bindings)
  (writeln "is:_goal_" goal)
  (writeln "is:_Expr_" theExpr)

  (eval theExpr)))
(writeln "is:_goal_" goal)
(writeln "is:_Expr_" theExpr)
(writeln "is:_result_" result)
(if (not result) #f
  (prove-all other-goals bindings)))))

(define (top-level-prove goals)
  (prove-all '(,@goals (show-prolog-vars ,@(variables-in goals)))
             no-bindings)
  (writeln "_No._")
  (values))

(define (show-prolog-vars vars bindings other-goals)
  "Print_each_variable_with_its_binding.
  Then_ask_the_user_if_more_solutions_are_desired."
  (if (null? vars)
      (writeln "Yes")
      (dolist
        (var vars)
        (writeln
          var "=_"
          (subst-bindings bindings var))))
  (if (continue?)
      fail
      (prove-all other-goals bindings)))

(define (continue?)
  "Ask_user_if_we_should_continue_looking_for_solutions."
  (case (read-char)
    ((#\;) #t)
    ((#\.) #f)))

```

```

((#\newline) (continue?))
(else
  (writeln "Type ; to see more or . to stop")
  (continue?)))

(define (variables-in exp)
  "Return_a_list_of_all_the_variables_in_EXP."
  (unique-find-anywhere-if non-anon-variable? exp))

(define (non-anon-variable? x)
  (and (variable? x) (not (eqv? x '?)))))

(defmacro* (?- &rest goals)
  '(top-level-prove
    (replace-?-vars (quote ,goals)))))

(store-toplevel-predicate!
 'show-prolog-vars show-prolog-vars *database*)

; added by L. D.-F.
(store-toplevel-predicate!
 'not prove-not *database*)

(store-toplevel-predicate!
 'is is *database*)

(store-toplevel-predicate!
 'test test *database*)

(trace
 ; prove
 ; prove-all
 ; prove-not
 )
); end module

```

37.4 Das Prolog-in-Scheme-Paket: prologInScheme.ss

Dieses Modul bündelt und exportiert alle Module, die Sie benötigen, um Prolog-in-Scheme zu benutzen. Außerdem definiert es die Hilfsprädikate member, length, = und !=.

```
(module prologInScheme (lib "swindle.ss" "swindle")
; ;;; some predefined prolog predicates by L. Dreschler-Fischer,
; ;;; exports all functions from the prolog library
(require
  swindle/setf
  swindle/misc
  (all-except
    se3-bib/tools-module
    mappend  every some last concat )
  se3-bib/pattern-matching-module
  se3-bib/prolog/unify
  se3-bib/prolog/prologDB
  se3-bib/prolog/prolog
)
(provide
  (all-from swindle/setf)
  (all-from swindle/misc)
  (all-from se3-bib/tools-module)
  (all-from se3-bib/prolog/unify)
  (all-from se3-bib/prolog/prologDB)
  (all-from se3-bib/prolog/prolog)
  (all-from se3-bib/pattern-matching-module)
)
; some predefined predicates

; equality
(<- (= ?x ?x)) ; do the clauses unify?
(<- (!= ?x ?y) :-
  (not = ?x ?y)) ; do the clauses not unify?

; list processing
```

```

(<- (member ?item (?item . ?rest)))
(<- (member ?item (?x . ?rest))
     :- (member ?item ?rest))

; functional predicates , require scheme functions

; (length list length-of-x)
(<- (length () 0))
(<- (length (?x . ?rest) ?len) :-
   (length ?rest ?lenR)
   (is ?len (+ 1 ?lenR)))

; numbers

; (between: is x larger than a and smaller than b?
(<- (between ?a ?x ?b) :-
   (test (< ?a ?x ?b)))
)

```

Teil XIII

Zusammenfassung

38 Zusammenfassung und Ausblick

38.1 Zusammenfassung

Zusammenfassung und Ausblick



34 Zusammenfassung und Ausblick

- Zusammenfassung
- Vorbereitung auf die Klausur

Navigation icons: back, forward, search, etc.

Wichtige Themen der funktionalen Programmierung

- Funktionale Abstraktion, Definitionen und Ausdrücke
- Semantik: Substitutionsmodell, Reduktionsstrategien
- Funktionen höherer Ordnung, Idiome, Funktionale Abschlüsse
- Rekursion: Definitionen, Prozesse, Algorithmen
- Kontrollabstraktion: Backtracking, pattern matching
- Objekte und generische Funktionen
- Memo-Funktionen, (Strom-orientierte Programmierung)
- Relationale Programmierung

Zusammenfassung

Verarbeitungsmodell und Programmierstil

Programmierstil

Scheme	Prolog	Java
Funktional objektorientiert	Relational -	Imperativ objektorientiert

Verarbeitungsmodell

Scheme	Prolog	Java
Anwendung von von Funktionen auf Werte	Relationale Anfragen an eine Datenbasis	Zustände von Objekten

Zusammenfassung: Semantik

Denotational und operational

Denotationale Semantik

Scheme	Prolog	Java
λ -Kalkül, Ausdrücke, Wertesemantik	Logik und Relationen, Anfragen	Anweisungen, Vor- und Nach- bedingungen

Sematik: operational

Scheme	Prolog	Java
Reduktionsstrategien: Vorgezogene Auswertung, eval	Unifikation und Suche	Virtuelle Maschine

Zusammenfassung: Eigenschaften

Bezugstransparenz und Richtungsunabhängigkeit

Bezugstransparenz

Scheme	Prolog	Java
ja, aber nur ohne Modifikatoren	ja	nein

Richtungsunabhängigkeit

Scheme	Prolog	Java
nein	ja, aber nur ohne funktionale Auswerteumgebung	nein

Schlupflöcher, eingebettete Programmierstile

Scheme	Prolog	Java
Imperativ	funktional	-

Zusammenfassung

Typsystem und Typprüfung

Typsystem

Scheme	Prolog	Java
Latentes	Latentes	Manifestes Typsystem

Typprüfung

Scheme	Prolog	Java
Laufzeit	Laufzeit	Übersetzungszeit

Zusammenfassung

Vor- und Nachteile

Latente versus statische Typisierung

Scheme	Prolog	Java
Kurze Programme, Symbolverarbeitung, einfache Metaprogrammierung Codeerzeugung zur Laufzeit		frühzeitige Fehlererkennung, wohldefinierte Schnittstellen

Programmierumgebung

Scheme	Prolog	Java
Interpreter Compiler exploratives Programmieren	Interpreter	Compiler

Wann wählen wir welchen Programmierstil?

Wir haben die Stärken von zwei unterschiedlichen Verarbeitungsmodellen besprochen.

Beachte:

- ☞ Die Verarbeitungsmodelle sind nicht disjunkt.
Die meisten Abstraktionen aus einem Verarbeitungsmodell haben Entsprechungen in anderen Modellen.
- ☞ Es ist Ihre Aufgabe, bei einem gegebenen Modell jeweils die tiefsten Abstraktionen zu finden.
- ☞ Es ist Ihre Aufgabe, bei einem gegebenen Problem jeweils ein adäquates Verarbeitungsmodell zu wählen.

Sechs Maximen für guten Programmierstil

- Be specific.
- Use abstractions.
- Be concise.
- Use the provided tools.
- Don't be obscure.
- Be consistent.

(nach Norvig 92)

Ausblick: Vertiefungsmöglichkeiten im Bachelorstudium

- 64.145 SoSe Grundstudiumspraktikum (GPRAK) „Bildverarbeitungspraktikum“
- Modul: Wissensverarbeitung
- Modul: Interactive Visual Computing
- Module: Bildverarbeitung 1 + 2

ENDE

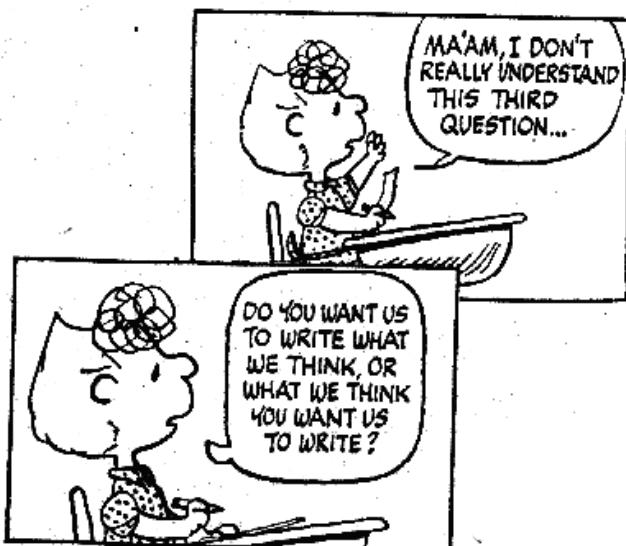
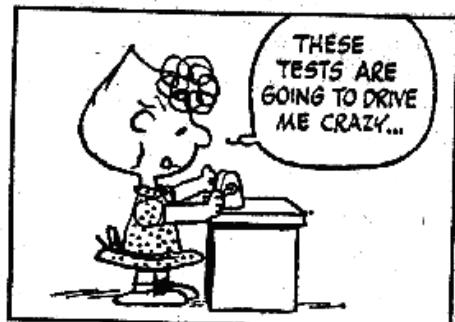
Viel Erfolg für das weitere Studium!

Aus der Einleitung von [Norvig, 1992](#):

The premise of this book is that you can only write something useful and interesting when you both understand what makes good writing and have something interesting to say. This holds for writing programs as well as for writing prose. As Kerningham and Plauger put it on the cover of *Software Tools in Pascal*:

Good programming is not learned from generalities, but by seeing how significant programs can be made clean, easy to read, easy to maintain and modify, human-engineered, efficient, and reliable, by the application of common sense and good programming practices. Careful study and imitations of good programs leads to better writing.

Kerningham and Plauger



38.2 Vorbereitung auf die Klausur

Zur Vorbereitung auf die Klausur prüfen Sie sich selbst und versuchen Sie, ob Sie die folgenden Fragen beantworten können:

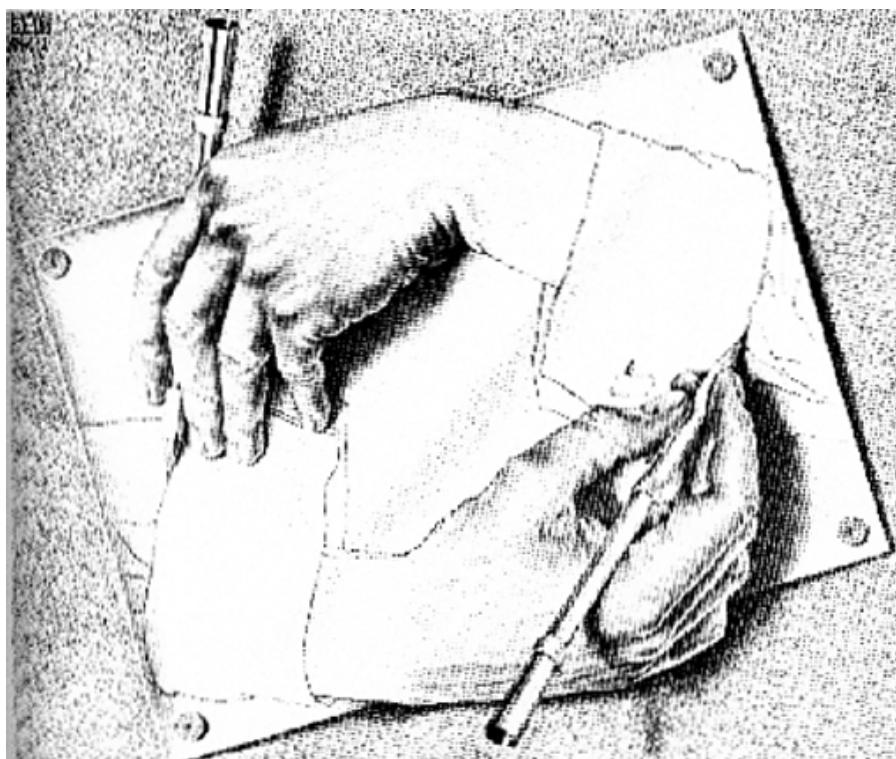
- Auf welche Grundoperationen beziehen wir uns bei der funktionalen Programmierung? Erklären Sie den special form operator `lambda`.
- Was ist die Bedeutung eines funktionalen Ausdrucks? denotationell, operational? Wann sind beide Semantikdefinitionen äquivalent, wann nicht?
- Was ist das Substitutionsmodell der Auswertung? Welche Reduktionsstrategien gibt es? Erklären Sie die Begriffe „Striktheit“, „vorgezogene Auswertung“, „verzögerte Auswertung“. Welche Rolle spielt die Umgebung bei der Auswertung?
- Was sind Funktionen höherer Ordnung? Welche Funktionen bilden den Werkzeugkasten für den Entwurf von Funktionen?
- Warum sind funktionale Abschlüsse so wichtig? Was ginge nicht so einfach, wenn es in Scheme keine “closures” gäbe?
- Wann ist eine Definition rekursiv? Welche Prozesse werden durch rekursive Funktionen ausgelöst? Was ist Endrekursion? Wie überführen Sie eine allgemeine lineare Rekursion in eine Endrekursion? Können Sie eine endrekursive Funktion definieren, die die Länge einer Liste errechnet, oder die Funktion “map” endrekursiv definieren? Nennen Sie ein Beispiel für eine Baumrekursion.
- Nennen Sie ein Problem, das durch backtracking gelöst werden kann. Nach welchem Schema geht man beim backtracking vor? Welche Probleme lassen sich mit dem in der Vorlesung behandelten backtracking-Schema nicht lösen? Warum nicht?
- Läßt sich die Unifikation als pattern matching Problem formulieren?
- Was ist nötig, den Strom-orientierten Programmierstil in Scheme einzubetten, d.h. welche Sprachelemente müssen wir nachbilden? Welche Vorteile bringt verzögerte Auswertung?
- Was ist ein Datentyp? Welches Typsystem verwendet Scheme?

Was ist ein *rekursiver* Datentyp? Nennen Sie ein Beispiel.

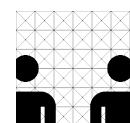
- Was ist eine Spezifikation? Was ist eine Zusicherung (assertion)? Wann ist ein Programm korrekt? Wie stellen Sie sicher (und beweisen Sie), daß eine rekursive Funktion terminiert?
- Für welche Algorithmen haben wir Modifikatoren (`set!`) verwendet? Warum war das nötig? Welche Konsequenzen hat der Einsatz von Modifikatoren für die Semantik eines Programms?
- Wie unterscheiden sich generische Funktionen in CLOS von der Nachrichtenmetapher in Java?
- Was sind Ergänzungsmethoden? Was ist Methodenkombination?
- Was sind Symbole (in Scheme)?
- Nennen Sie je ein Problem, für das Sie den Algorithmus zur Lösung lieber in Prolog oder lieber in Racket programmieren würden und begründen Sie Ihren Vorschlag.

Softwareentwicklung III

Leonie Dreschler-Fischer
Prüfungsunterlagen Teil 3:
Anhang



Universität Hamburg
Fachbereich Informatik
WS 2019/2020



Teil XIV

Anhang

A Das Modul SE III: (aus dem Akkreditierungsantrag)

Das Modul Softwareentwicklung III (SE III), engl.: Software Development III ist ein Pflichtmodul in den Studiengängen BSc Informatik und BSc Wirtschaftsinformatik.

A.1 Motivation und Stellung im Gesamtprogramm

Aufbauend auf den in den Modulen Softwareentwicklung I und II vermittelten Grundlagen der zustands- und objektorientierten Programmierung, behandelt das Modul Softwareentwicklung III mit der funktionalen und der logik-basierten Programmierung zwei alternative Paradigmen, deren Stärken vor allem beim Prototyping für komplexe Symbolverarbeitungsaufgaben und das explorative Programmieren in Anwendungsbereichen wie Compilerbau, Formelmanipulation, Hardwaredesign, biochemische Strukturanalyse, Bild- und Sprachverarbeitung, technische Diagnose und Wissensverarbeitung zum Tragen kommen. Ausschlaggebend hierfür ist die Kombination leistungsfähiger Programmiersprachenkonzepte mit einer flexiblen Erweiter- und Modifizierbarkeit von Syntax und Semantik der Sprache. Damit bieten diese Paradigmen hervorragende Möglichkeiten zur Definition und experimentellen Validation neuer Abstraktionsebenen. Aufgrund ihrer soliden theoretischen Fundierung in den Berechnungsmodellen des Lambda- bzw. Prädikatenkalküls bieten Sie in vielen Fällen alternative Sichten auf die Konzeptualisierung von Informationsverarbeitungsproblemen an und stellen somit zusätzliche Denkmodelle bereit, die eine wichtige Quelle der Inspiration für neuartige Lösungsansätze sein können.

Um eine vertiefte Befassung mit den zentralen Konzepten eines Programmierparadigmen zu ermöglichen, wird in dem Modul wahlweise eine Einführung in das funktionale bzw. logik- basierte Paradigma angeboten. Dadurch kann erreicht werden, dass zumindest eine der alternativen Sichten auf die Programmierung über das bloße Kennenlernen hinaus auch produktiv umgesetzt werden kann.

A.2 Lernziele und Kompetenzen

Bezug zum Leitbild:

- Vermittlung von Wissen über Programmiersprachenkonzepte und ihre Umsetzung in Programmiersprachen
- Kennenlernen alternative Programmierparadigmen und der ihnen zugrunde liegenden Verarbeitungs- bzw. Denkmodelle
- Entwicklung von Problemlösungskompetenz in einem alternativen Programmierparadigma
- Exemplarisches Veranschaulichen der wissenschaftlichen Methodik der Informatik anhand des Zusammenwirkens von formal-theoretischem Grundlagenwissen und programmiersprachlicher Umsetzung

Grundlagen-/Faktenwissen:

- Kenntnis der relevanten Begriffe funktionaler und relationaler SW-Entwicklung
- Kenntnis fortgeschritten objektorientierter SW-Konstruktionen
- Kenntnis der denotationellen semantik einer Programmiersprache, Korrektheit von Programmen
- Grundlegende Techniken der Symbolverarbeitung und der Metaprogrammierung

Methodenwissen:

- Programmiersprachenkonzepte
- Grundprinzipien und Idiome der funktionalen bzw. logik-basierten Programmierung
- Metaprogrammierung und Definition neuer Verarbeitungsmodelle

Transferkompetenz:

- Aufgaben- und problemorientiertes Denken
- Erkennen von Analogien zwischen Programmiersprachen und -paradigmen
- Fähigkeit zum Übertragen bekannter Konzepte in neue programmiersprachliche Kontexte
- Anwendung von Konzepten und Erkenntnissen aus der Theoretischen Informatik im Kontext der Programmiersprachen

Normativ-bewertende Kompetenz:

- Fähigkeit zum Bewerten von Programmierwerkzeugen und -umgebungen hinsichtlich ihrer Eignung für unterschiedliche Programmierszenarien und Aufgabenklassen
- Fähigkeit zum Beurteilen von Programmierkonzepten- und paradigm im Hinblick auf ihre Eignung zur Behandlung unterschiedlicher Problemklassen bzw. zur Anpassung an diese.

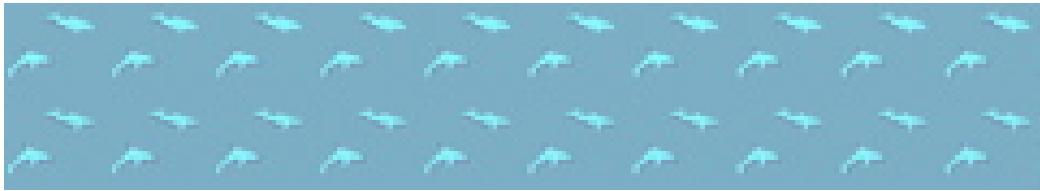
ABK/BOK/Schlüsselqualifikationen

- Kooperations- und Teamfähigkeit in den Präsenzübungen Strategien des Wissenserwerbs: Kombination aus Vorlesung, Vor- und Nachbereitung am Vorlesungsmaterial, Präsenzübungen mit betreuter Gruppenarbeit und eigenständiges Lösen von Übungsaufgaben
- Kreatives Problemlösen anhand von Beispielen aus verschiedenen Bereichen der Informationsverarbeitung Umgang mit Softwareentwicklungswerkzeugen auch außerhalb einer integrierten Entwicklungs-umgebung

Das Modul baut auf Voraussetzungen auf, die hauptsächlich in den Pflichtmodulen Softwareentwicklung I und II, sowie Formale Grundlagen der Informatik gelegt werden. Es ergänzt die Lehrinhalte dieser Module um alternative Sichten auf die Programmierung, bzw. zeigt Anwendungen der formalen Grundlagen auf. Das Modul legt Grundlagen für die Wahlpflichtmodule Wissensverarbeitung, sowie Logik, Formale Sprachen und Semantik, sowie verschiedene Vertiefungen insbesondere im Bereich der Intelligenzen Systeme. Der Modulbestandteil Logikprogrammierung vermittelt darüberhinaus auch einen guten Einblick in die grundlegenden Methoden und Werkzeuge des Integrierten Anwendungsfaches Verarbeitung natürlicher Sprache. Der Modul kann durch ein Praktikum im Bereich der funktionalen bzw. Logikprogrammierung ergänzt werden.

B Anmerkungen zu Abstraktion und Begriffsbildung

B.1 Begriffsbildung



Die Lehre von der Bildung der Begriffe und deren Beziehung zu komplexeren Wissenseinheiten (Schlußfolgerungen, Theorien usw.) ist eine grundlegende Problemstellung der Erkenntnistheorie und Philosophie. Wir beschränken uns hier auf einen Teil der *mathematisch-naturwissenschaftlichen* Aspekte.

Seit Aristoteles basiert die Begriffstheorie auf der Lehre der Begriffsbildung durch *Abstraktion*.

Abstraktion ist das Verfahren der Hervorhebung gemeinsamer Merkmale von Gegenständen, Sachverhalten oder Ereignissen zu einer Einheit. Erst dadurch ergibt sich der allgemeine Begriff, der nach antiker Lehre zugleich auch das *Wesen* eines individuellen Dinges oder Sachverhaltes ausmacht.

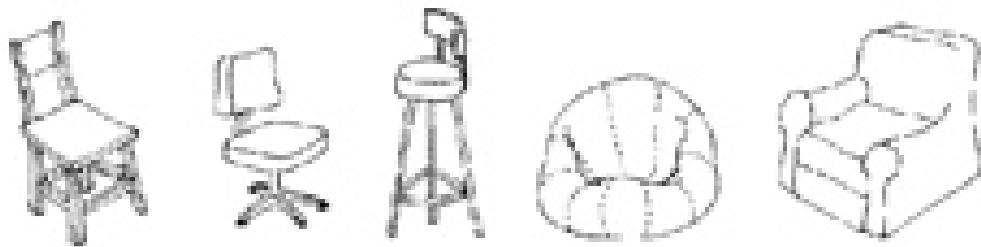
Ein Begriff wird definiert durch die Angabe der übergeordneten *Gattung* und des *spezifischen oder wesentlichen Merkmals*. Abstraktion und Definition sind daher nach der Aristotelischen Begriffslehre eng verbunden: Durch Abstraktion werden Begriffe gebildet und durch Definitionen festgelegt und voneinander abgegrenzt. ([Strube, 1996](#))

Mit dem Aufkommen der empirischen Naturwissenschaften wurde die Aristotelische *Wesensabstraktion* in eine methodische *Klassenabstraktion* umgewandelt. Unter Begriff versteht man nicht mehr das eigentliche Wesen eines Gegenstandes im metaphysischen Sinn, sondern lediglich die Zusammenfassung jener Merkmale, welche die charakteristischen oder relevanten Eigenschaften einer Klasse oder *Kategorie von Objekten* beschreiben.

Welche Eigenschaften eines Objektes als charakteristisch oder relevant angesehen werden, wird von den methodischen Zielsetzungen eines Wissensgebiets bestimmt. *Begriffe sind immer kontextabhängig!*

Ein Beispiel

Alle diese Objekte verbindet die Eigenschaft, daß man auf ihnen mehr oder weniger bequem sitzen kann. Wir können daher den Begriff „Sitzgelegenheit“ als abstrakte Kategorie einführen.



B.2 Abstraktionsverfahren

Abstraktionsschema

Abstraktion ist ein zielgerichtetes logisches Verfahren. Die auf Frege zurückgehende Abstraktionsmethode der modernen Logik ist konstruktiv:

- Durch Gemeinsamkeiten von Gegenständen wird ein positiv bestimmtes und explizit benanntes Merkmal angegeben und so verfahren, als ob man über neue Gegenstände — abstrakte Objekte — redete, obwohl man nur in einer *neuen Weise* über die konkreten Objekte spricht.

Formal wird die Abstraktionsmethode mithilfe des Abstraktionsschemas dargestellt:

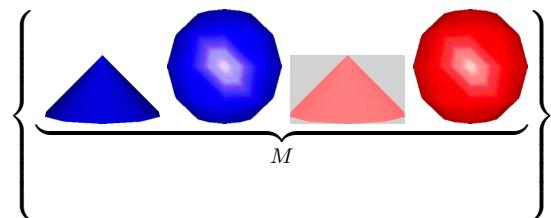
- Alle Gegenstände, die eine gleiche Eigenschaftsausprägung haben, werden einer Klasse zugeordnet, wobei eine *Äquivalenzrelation* definiert wird.
- Über die Elemente einer solchen Klasse lassen sich gemeinsame (invariante) Aussagen machen. (**Strube, 1996**)

Generalisierung

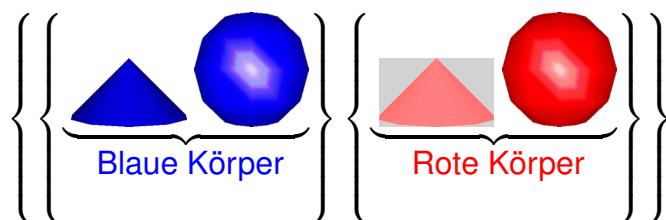
- Abstraktion ist eine Form der *Generalisierung*, bei der von einer Menge von Objekten und ihrer unmittelbar kennzeichnenden Merkmale abgehoben wird und zu einer allgemeinen sprachlich-begrifflichen Charakterisierung der betreffenden Objektmenge übergegangen wird.
- Die besondere Bedeutung der Abstraktion für das Programmieren besteht in der *Datenabstraktion* und der *Kontrollabstraktion*, hier der *funktionalen Abstraktion*.

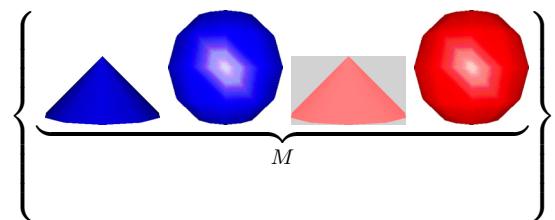
Zur funktionale Abstraktion siehe Abschnitt 4.1 und zur Datenabstraktion siehe Abschnitt 148.

Äquivalenzklassen von Objekten

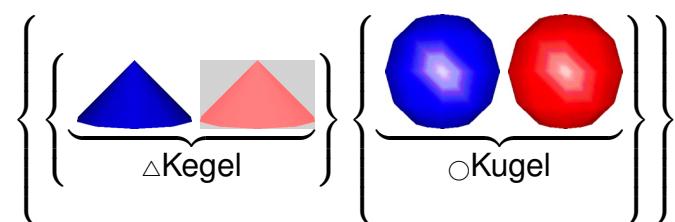


Äquivalenzklassen für $R_{\text{gleicheFarbe}}$





Äquivalenzklassen für $R_{\text{gleicheForm}}$



Abstraktoren und invariante Aussagen

Nach der Definition von (Görz, 1989):

- Um auszudrücken, daß Aussagen bzgl. einer Äquivalenzrelation invariant sind, verwenden wir einen *hypothetischen abstrakten Gegenstand*, der nur die gemeinsamen Eigenschaften hat.
- Invariante Aussagen A werden durch Einführung eines *Abstraktors* α neu ausgedrückt.

$$A(\alpha x)$$

- mit dem *abstrakten* Gegenstand

$$\tilde{x} = \alpha x$$

Beispiel: Natürliche Zahlen

Grundklasse: Zählzeichen x in verschiedenen Zahlendarstellungen (Strichlisten, römische und arabische Zahlen)

Abstraktor: *Zahl*, als abstrakte Zählzeichen

Äquivalenzrelation: Besteht aus gleich vielen Zählzeichen

Wir können jetzt präziser definieren, was Abstraktion ist:

Definition: 154 (Abstraktion)

- *Abstraktion ist der Übergang von Aussagen*

$$A(a) \quad \text{über Objekte } a, b$$

mit einer zwischen ihnen erklärten

Äquivalenzrelation \sim

zu Aussagen $A(\tilde{\alpha})$

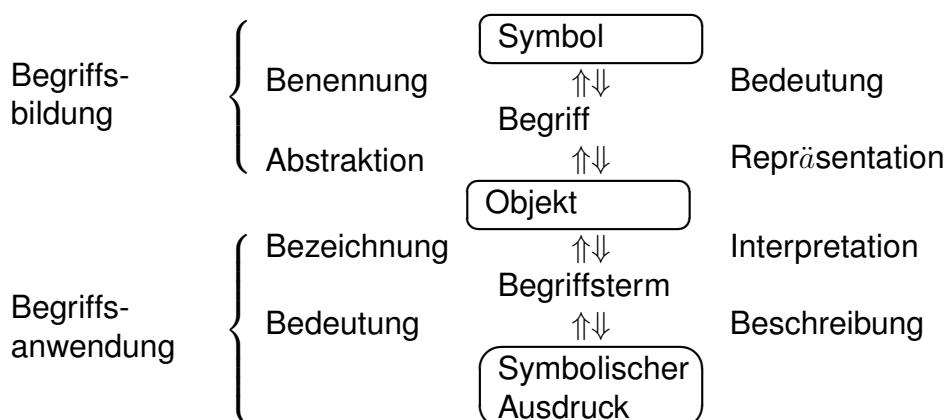
- *Mittels des Abstraktionsschemas*

$$A(\tilde{x}) \Leftrightarrow \forall_y(x \sim y \rightarrow A(y))$$

wird die Rede über die abstrakten Objekte eingeführt: $\tilde{a}, \tilde{b}, \dots$

- *Stehen zwei Objekte a, b in der Beziehung $a \sim b$, so sagt man, daß a und b dasselbe abstrakte Objekt \tilde{a} (oder gleichwertig \tilde{b}) darstellen.*

Schema der Begriffsbildung nach Schefe-1985



Eine auch für Anfängerinnen und Anfänger gut verständliche Lektüre zum Thema Begriffsbildung ist ([Lakoff, 1987](#)) . Der Titel des Buches, “Women, fire, and dangerous things” bezieht sich darauf, daß es eine Sprache gibt, in der die Menschen in den drei Dingen “Frauen, Feuer und anderen gefährlichen Sachen” soviel Gemeinsames gesehen haben, daß sie dafür eine eigene Kategorie als Oberbegriff geschaffen haben, für mich völlig unverständlich ;-).

C Implementation der Ströme in Racket

Beispiele

```
> (define ab-3 (integers-from-n 3))
> ab-3 → (3 . #<struct:promise>)

> (head-stream
  (tail-stream ab-3)) → 4
> (head-stream
  (tail-stream
    (tail-stream ab-3))) → 5
```

Indizierung eines Stroms:

```
(define (stream-ref stream i)
  "The _i_-th _element_ of _a_ stream , _0-based"
  (cond ((null? stream) #f)
        ((= 0 i) (head-stream stream))
        (else (stream-ref
                  (tail-stream stream) (- i 1))))))

> (stream-ref ab-3 20) → 23
```

Abilden eines Stroms

```
; ; map each element of a stream
(define (map-stream proc stream)
  (if (empty-stream? stream)
      the-empty-stream
      (cons
        (proc (head-stream stream))
        (delay
          (map-stream
            proc (tail-stream stream))))))

> (map-stream add1 ab-3)
→ (4 . #<struct:promise>)
```

Weitere Stromfunktionen: Filtern

```
(define (filter-stream pred? stream)
  (cond ((empty-stream? stream)
         the-empty-stream)
        ((pred? (head-stream stream))
         (cons
```

```

(head-stream stream)
(delay
  (filter-stream
    pred? (tail-stream stream))))))
(else (filter-stream
  pred? (tail-stream stream)))))

> (filter-stream even? ab-3)
(4 . #<struct:promise>)

```

Seiteneffekte: for-each:

Achtung: gibt bei unendlichen Strömen eine Endlosschleife!

```

(define (for-each-stream proc stream)
  (cond ((empty-stream? stream)
         'done)
        (else (proc (head-stream stream))
              (for-each-stream
                proc (tail-stream stream))))))

```

Teilstücke eines Stroms

```

(define (take-stream n stream)
  ; the sublist of the first n elements
  ; of xs
  (cond
    ((empty-stream? stream) '())
    ((= 0 n) '())
    (else
      (cons (head-stream stream)
            (take-stream
              (- n 1)
              (tail-stream stream))))))

(define (drop-stream n stream)
  ; drop the first n elements of xs
  ; and return the rest
  (cond
    ((empty-stream? stream) '())
    ((= 0 n) stream)
    (else (drop-stream
              (- n 1)
              (tail-stream stream))))))

```

Erzwinge die Auswertung der ersten n Elemente:

```
(define (force-stream-to-depth n depth stream)
  ; force the first n elements of the stream
  ; to depth "depth"
  (cond
    ((not (is-stream? stream)) stream)
    ((empty-stream? stream) '())
    ((= 0 n) stream)
    (else
      (cons
        (force-stream-to-depth
          depth depth (head-stream stream))
        (force-stream-to-depth
          (- n 1) depth (tail-stream stream)))))))
```



```
(define (force-stream n stream)
  ; force the first n elements of the stream
  (force-stream-to-depth n n stream))
```

Verbinden von Strömen:

```
(define (append-streams x y)
  "Return_a_stream_that_appends_the_elements_of_x_and_y."
  (if (empty-stream? x)
    y
    (cons (head-stream x)
      (delay (append-streams (tail-stream x) y)))))

(define (mappend-stream fn stream)
  "Lazily_map_fn_over_stream,_appending_results."
  (if (empty-stream? stream)
    the-empty-stream
    (let ((x (fn (head-stream stream))))
      (cons
        (head-stream x)
        (delay (append-streams
          (tail-stream x)
          (mappend-stream
            fn (tail-stream stream))))))))
```



```
(define (combine-all-streams xstream ystream)
  ; Return a stream of streams
```

*; formed by appending an x to an y.
; in other words: form the cartesian product.*

```
(mappend-stream
  (lambda (y)
    (map-stream
      (lambda (x)
        (append-streams x y))
      xstream))
  ystream))

(define (write-with-space x)
  (write x) (display " "))

(define (print-stream stream)
  (for-each-stream write-with-space stream)
  (writeln))
```

D Kurzreferenz: Racket

D.1 Funktionslexikon

Verknüpfung von Funktionen

(**curry** f arg1 ... argn) : Partielle Anwendung von f auf arg1 ... argn von links nach rechts

(**curryr** f argm ... argn) : Partielle Anwendung von f, Bindung der Argumente von rechts nach links

(**compose** f1 ... fn) : Funktionskomposition, Hintereinanderausführung
(f1 (... (fn x)))

(**negate** p?) : erzeuge das inverse Prädikate zu p?

(**conjoin** p1? ... pn?) : Konjunktion von Prädikaten

(**disjoin** p1? ... pn?) : Disjunktion von Prädikaten

(**always** args) : unabhängig von den Argumenten immer #t

(**never** args) : unabhängig von den Argumenten immer #f

(**const** x) : Erzeugt eine konstante Funktion, deren Wert unabhängig von den Argumenten x ist.

Idiome der funktionalen Programmierung

(**apply** f xs) : Anwendung einer Funktion f auf eine Liste von Argumenten xs

(**map** f xs1 ... xsn) : Abbilden einer oder mehrerer Listen auf die Liste der Bilder der Elemente. Die Stelligkeit der Funktion f muß mit der Anzahl der angegebenen Listen xs1...xsn usw. übereinstimmen.

(**filter** p? xs) : Berechne die Liste der Elemente von xs, die p? erfüllen

(**filter-not** p? xs) : Berechne die Liste der Elemente von xs, die p? nicht erfüllen

(**foldl** f seed xs) : Paarweise Verknüpfung der Elemente von xs mit f, Startwert seed, (f xn (f xn-1 (... (f x1 seed)))), **end-rekursiv**, falten mit **foldr** und **append** oder **cons** erzeugt eine invertierte Liste.

(**foldr** f seed xs) : Paarweise Verknüpfung der Elemente von xs mit f, Startwert seed, (f x1 (f x2 (... (f xn seed)))), **linear rekursiv**.

(**iterate** f end? start) : Die Liste der Funktionsanwendungen, bis end? erfüllt ist, (start (f start) (f (f start)) ...)

(**iter-until** f end? start) : Der erste Wert der Folge start, (f start), (f (f start)), ... der end? erfüllt

(**memf** p? xs) : Finde das erste Element von xs, das das Prädikat p? erfüllt und gebe die Teilliste ab dem Element zurück, ansonsten gebe #f zurück.

(**ormap** p? xs) : wahr, wenn mindestens ein Element von xs das Prädikat p? erfüllen.

(**andmap** p? xs) : wahr, wenn alle Elemente von xs das Prädikat p? erfüllen.

(**sort** xs rel<) : sortiert die Liste xs nach der Ordnungsrelation rel<,

(**assoc** key alist) : Suche in der Assoziationsliste alist das erste Paar, daß das als *Kopf* den Schlüssel key enthält.

(**rassoc** key alist) : Suche in der Assoziationsliste alist das erste Paar, daß das als *Rest* den Schlüssel key enthält.

D.2 CLOS

Klassen

```
(defclass <Name der Klasse> ({{<Oberklassen>}}  
  {(<Slot> {<Slot-keys>})}  
  {<Class-keys>}  
)
```

Optionen für die Attribute (slots):

- :initarg *keyword*: <key> *Schlüsselwort für den Konstruktor*
- :initializer <func> *Initialisierungsfunktion*
- :initvalue <value> *Defaultwert*
- :reader <name> *Akzessorfunktion zum Lesen des Wertes*
- :writer <name> *Akzessorfunktion zum Schreiben des Wertes*
- :accessor <name> *Akzessorfunktion zum Lesen und Schreiben des Wertes*
- :accessor <name> *Akzessorfunktion zum Lesen und Schreiben des Wertes*
- :type <type> *Typdefinition für die Klasse des slots*

```
(defgeneric <name> ({{(<arg> <class>)}} {<arg>})  
  {:combination <combination>}  
  {:documentation <string>} )
```

```
(defmethod <name> [<qualifier>] ({{(<arg> <class>)}} {<arg>})  
  {:documentation <string>} )
```

Ergänzungsmethoden

<qualifier> ::= :after | :before | :around

Methodenkombinationen

generic-+-combination
generic-list-combination
generic-min-combination
generic-max-combination
generic-append-combination

generic–append!–combination
generic–begin–combination
generic–and–combination
generic–or–combination

D.3 Prolog-in-Scheme-Lexikon

Variable: ?X : Die Namen von Variablen beginnen mit einem Fragezeichen.

Anonyme Variable: ? : Ein Fragezeichen bezeichnet eine anonyme Variable, die in Ausgaben unterdrückt wird.

Regeln: Klauseln mit Prämissen (Zielen) :

<Klausel–Kopf> :- <Ziel 1>...<Ziel n>

Negation: not : Der not-Operator negiert eine Klausel : (**not** <Klausel>)

Ungleichheit: != : Das !=-Prädikat ist wahr, wenn zwei Strukturen oder Variablen nicht unifizieren :

(!= <struktur1> <struktur2>)

Das assert-Macro: ← : Trage eine Klausel in die Datenbasis ein.

(← <Klausel>)

Das query-Macro: ?- : Anfrage, durch welche Variablenbindungen die Konjunktion der Ziele in der Anfrage erfüllt werden kann.

(?- <Ziel 1> ... <Ziel n>)

findall: **findall** sammelt alle Resultate des Prädikatsaufrufs <Ziel> in einer Liste <Liste> als instanzierte Varianten des Ausdrucks <Term>

.

(findall <Term> <Ziel> <Liste>)

count: **count** zählt die Resultate des Prädikatsaufrufs <Ziel> und bindet die Anzahl an die Variable <Var>.

(**count** <Var> <Ziel>)

Funktionale Auswertung: is: Das is-Prädikat bindet den Wert eines funktionalen Ausdrucks an eine Variable. Alle Variablen des Ausdrucks müssen vorher an Werte gebunden sein.

(is <Var> <Ausdruck>)

Funktionale Auswertung: test: Das test-Prädikat evaluiert einen funktionalen Ausdruck. Das Prädikat ist erfüllt, wenn der Ausdruck wahr ist. Alle Variablen des Ausdrucks müssen vorher an Werte gebunden sein.

(test <Ausdruck>)

D.4 Unterschiede zwischen den Schemadialekten, DrScheme, Version 4.1.1

Zahl der Operanden

Lehresprachen: Fortgeschritten

Für `+`, `-`, `and`, `or` sind mindestens zwei Operanden notwendig.

Sprache: Fortgeschritten ;
> `(+ 1)`
procedure `+`: expects at least 2 arguments, given 1: 1
> `(and #t)`
`and`: expected at least two expressions after '`and`' ...

Racket, Kombo, R5RS, Module (#lang scheme):

Für `+`, `-`, `and`, `or` ist die Operandenzahl beliebig.

Sprache: Fortgeschritten ;
> `(+)` → 0
> `(+ 1)` → 1
> `(+ 1 1 2)` → 4
> `(and)` → `#t`
> `(and #t)` → `#t`

eval

Lehresprachen: Fortgeschritten

`eval` ist undefiniert.

Sprache: Fortgeschritten ;
> `eval`
reference to an identifier before its definition: `eval`
> `(eval 1)`
reference to an identifier before its definition: `eval`

Kombo, R5RS, Module (#lang scheme):

`eval` ist eine Standardfunktion.

Sprache: Fortgeschritten ;
Sprache: Kombo ;
> `(eval '(cons 1 2))` → `(1 . 2)`
> `eval` → #<procedure:eval>

Ausdrücke als Wahrheitswerte

Lehresprachen: Fortgeschritten

Die logischen Operatoren akzeptieren nur **#t** und **#f** Ausdrücke als Operanden.

Sprache: Fortgeschritten ;
> (**define** auto 'ich_habe_eins)
> (**if** auto 'fahren 'laufen)
if: question result is **not** true **or** false: 'ich_habe_eins

Kombo, R5RS, Module (#lang scheme):

Jeder Ausdruck kann als Wahrheitswert wirken. **#f** wirkt als **#f**, jeder andere Wert als **#t**.

Sprache: Kombo ;
> (**if** auto 'fahren 'laufen) —> fahren
> (**not** auto) —> #f

Implizites „begin“ im Let

Lehresprachen: Fortgeschritten

Kein implizites „begin“.

Nur *ein* Ausdruck im Rumpf eines **Let** erlaubt.

Sprache: Fortgeschritten ;
> (**let** ((x 1))
 (display x)
 x)
intermediate-let: expected only one expression after
the name-defining sequence, but found one extra part
> (**let** ((x 1))
 (**begin** (display x) x) —> 11

Kombo, R5RS, Module (#lang scheme):

Das „begin“ ist implizit.

Sprache: Kombo ;
> (**let** ((x 1))
 (display x)
 x) —> 11

Implizites „begin“ in Funktionen

Lehresprachen: Fortgeschritten

Nur *ein* Ausdruck im Rumpf einer Funktion erlaubt.

Sprache: Fortgeschritten ;

```
> (define (doppel a)
  (display a)
  (+ a a))
```

```
define: expected only one expression
for the function body, but found one extra part
```

Kombo, R5RS, Module (#lang scheme):

lambda impliziert ein „begin“.

Sprache: Kombo ;

```
> (define (doppel a)
  (display a)
  (+ a a))
> (doppel 3) → 36
```

Prädikate und Semiprädicat

Lehresprachen: Fortgeschritten

member ist ein Prädikat; **memq** ein Semiprädicat.

Sprache: Fortgeschritten ;

```
> (member 1 '(1 2 3)) → #t
> (memq 2 '(1 2 3)) → (2 3)
```

Kombo, R5RS, Module (#lang scheme):

Sprache: Kombo ;

```
> (member 1 '(1 2 3)) → (1 2 3)
```

Das cond-Konditional

Lehresprachen: Fortgeschritten

Im **cond**-Konditional muß für mindestens einen Wächter die Bedingung erfüllt sein:

Willkommen bei DrScheme, Version 4.1.1 [3m].

Sprache: Fortgeschritten ;

```
> (cond (#f 1))
cond: all question results were false
(cond (#f 1)
  (else (void))) →
```

Kombo, R5RS, Module (#lang scheme):

Das Resultat darf undefiniert sein.

Sprache: Kombo ;

```
(cond (#f 1)) →
(void? (cond (#f 1))) → #t
```

Bedingte Ausdrücke mit if

Lehresprachen: Fortgeschritten

if erwartet genau drei Argumente.

Sprache: Fortgeschritten ;

```
> (if #t 'wahr)
if: expected one question expression and
two answer expressions, but found 2 expressions
> (if #t 'wahr (void)) → wahr
> (when #t 'wahr) → wahr
> (unless #f 'wahr) → wahr
```

Kombo, R5RS, Module (#lang scheme):

if kann als einseitige Verzweigung verwendet werden.

Sprache: Kombo ;

```
> (if #t 'wahr) → wahr
> (when #t 'wahr) → wahr
```

Listen und „Paare“

Lehresprachen: Fortgeschritten

„cons“ ist kein allgemeiner Konstruktor für Paare, sondern des zweite Element eines Paars muß eine echte Liste sein.

Sprache: Fortgeschritten ;
> (**cons** 'a 1)
cons: **second** argument must be of type <list
or cyclic list>, given a **and** 1
> (**cons** 'a '(1)) → (a 1)

Kombo, R5RS, Module (#lang scheme):

Sprache: Kombo ;
> (**define** tupel (**cons** 'a 1))
> (**car** tupel) → a
> (**cdr** tupel) → 1

Der Infix-Operator „.“

Lehresprachen: Fortgeschritten

Der „dot-Operator“ als Kurzform für „cons“ ist unzulässig.
Die Funktion assoc ist undefined.

Sprache: Fortgeschritten ;
> (**define** dottedPair '(a . 1))
read: illegal use of ".."
> (**assoc** 'a pairs)
reference to an identifier before its definition: **assoc**

Kombo, R5RS, Module (#lang scheme):

Paare können mit dem Punkt-Operator in Infix-Notation geschrieben werden.

Sprache: **Module**; *memory limit: 128 megabytes*.
> (**define** dottedPair '(a . 1))
> dottedPair → (a . 1)
>

Nur in den Lehresprachen verfügbar:

Ein fehlgeschlagener Testfall

Beispiel: 155 (Testfall: nicht erfolgreich)

Im Definitionsfenster:

```
(define a 1)
(check-expect (* 3 a) 4)
```

Im Test-Results-Fenster:

```
Ran 1 check.
0 checks passed.
Actual value 3 differs from expected value 4 ...
```

Prüfe die Genauigkeit: check-within

```
(check-within expr1 expr2 expr3)
```

Prüfe, ob der numerische Ausdruck *expr1* zu *expr2* evaluiert.

expr3 ist ein Toleranzwert für die Genauigkeit.

Im Definitionsfenster:

```
(check-expect (sin pi) 0.0)
(check-within (sin pi) 0.0 0.1e-14)
```

Im Interaktionsfenster:

```
Ran 2 checks.
1 of the 2 checks failed.
Actual value #i1.2246467991473532e-16 differs
from 0, the expected value.
In testcases.scm at line 16 column 0
```

Special Form Operators für Test Cases

Prüfe die Fehlermeldung: check-error

```
(check-error expr expr)
```

Prüfe, ob der Ausdruck *expr1* die erwartete Fehlermeldung *expr2* signalisiert.

expr2 muß zu einer Zeichenkette evaluieren.

Im Definitionsfenster:

```
(check-error (/ 1 0)
            "/:_division_by_zero")
```


E Glossar und Index

Glossar

Ableitungsbaum

Rekursive Definition: Ein Ableitungsbaum ist entweder

- ein einzelner *Knoten*, der mit einem *Terminalzeichen* markiert ist (ein Blatt des Baumes)
- oder ein Knoten, markiert mit einer *metalinguistischen Variablen* und gefolgt von und verbunden mit den Elementen einer endlichen, geordneten Menge von (Teil-)Bäumen.
- Ein Knoten heißt *Wurzel*, wenn er keine übergeordneten Knoten hat.

. 129

Akkumulator

Ein zusätzliches Argument einer rekursiven Funktion, um **Rekursion: Endrekursion** zu ermöglichen.. 257, 258

Algorithmus

Ein grundlegender Begriff der Informatik, benannt nach dem persischen Mathematiker *Masa al Khowarizmi*:

Definition 1: Ein Algorithmus ist ein eindeutig bestimmtes Verfahren unter Verwendung von Grundoperationen über *primitiven* gegebene Objekten (nach Scheife-1985).

Definition 2: Ein Algorithmus ist eine präzise, d.h. in einer festgelegten Sprache abgefaßte, endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer Verarbeitungsschritte (nach Bauer-Goos-1982).

Wir wollen den Begriff des Algorithmus von seiner textuellen Beschreibung als **Programm** unterscheiden.. 15, 16, 271

Argument

siehe **Parameter**. 70

Auswertung

Die Auswertung funktionaler Ausdrücke erfolgt in einer Programmiersprache mit **Wertesemantik** nach dem **Substitutionsmodell**. Wir unterscheiden bzgl. des Reduktionsaufwandes zwischen **vorgezogener** und **verzögerter Auswertung**, siehe auch **Striktheit**.⁷⁶³

Wir unterscheiden bzgl. der Reduktionsreihenfolge zwischen **innerer Reduktion** und **äußerer Reduktion**.. 174

Auswertung, verzögert

Man spricht von verzögerter Auswertung (engl. lazy evaluation) oder Normalform der Auswertung (engl. normal order evaluation), wenn die Auswertung aller Argumente verzögert wird, bis sich zeigt, daß sie für die Reduktion des Ausdrucks erforderlich sind (Miranda, Haskell, Lazy Racket).. 174

Auswertung, vorgezogen

Man spricht von vorgezogener **Auswertung** (engl. eager evaluation) oder applikativer Auswertung (engl. applicative order evaluation), wenn alle Argumente einer Funktion ausgewertet werden, bevor die Funktion darauf angewendet wird (Racket, Common Lisp).. 174

Backtracking

Backtrackingalgorithmen sind **rekursive** Algorithmen, die sich immer dann gut anwenden lassen, wenn wir eine Lösung durch systematisches Probieren suchen müssen, wenn sich bei jedem Probierschritt mehrere neue Alternativen auftun, die untersucht werden müssen, und der Suchraum exponentiell anwächst, wenn jeder Probierschritt dem vorherigen strukturell ähnlich ist. Die Suche in Prolog ist eine wichtige Anwendung von Backtracking.. 322

Backtracking: allgemeines Schema

Um einen backtracking-Algorithmus zu entwerfen, benötigen wir die folgenden Zutaten:

Repräsentation des Suchraums als Menge von Zuständen: Z.B.
Liste der Positionen der Damen, Koordinaten im Labyrinth usw.

Ausgangszustand: Der Zustand, von dem aus wir systematisch suchen wollen, beispielsweise das leere Schachbrett oder der Eingang des Labyrinths.

Generatorfunktion: Eine Funktion, die einen Zustand auf eine Liste der direkt erreichbaren Folgezustände abbildet.

Test auf Zulässigkeit: Ein Prädikat, das einen Zustand daraufhin überprüft, ob er zulässig ist.

Test auf Erfolg: Ein Prädikat, das feststellt, ob ein Zustand ein Endzustand ist.

. 336

berechnungsuniversell

Eine **Programmiersprache** ist *berechnungsuniversell*, wenn in ihr jeder intuitive **Algorithmus** formuliert werden kann. General Purpose Programmiersprachen, wie Racket, Prolog, Java, C usw. sind berechnungsuniverselle Programmiersprachen, aber es gibt viele Spezialsprachen, wie Datenbankanfragesprachen und Modellierungssprachen. Diese sind in der Regel nicht berechnungsuniversell.. 30

Bezugstransparenz

Eine Sprache, bei der die denotationelle und die operationale **Semantik von Programmiersprachen** zusammenfallen, nennt man *referentiell transparent* bzw. *deklarativ*.

- Referentielle Transparenz bedeutet insbesondere, daß das Berechnungsresultat vom Zustand der (abstrakten) Maschine unabhängig ist.
- Die "Bezugstransparenz" oder "Referentielle Transparenz" wird auch *Konfluenz* oder nach den Entdeckern "Church-Rosser-Eigenschaft" genannt.

. 169, 171, 173, 176, 184, 189, 256

Closure, funktionaler Abschluß

Ein funktionaler Abschluß ist eine Funktion zusammen mit den Namensbindungen der lokalen Umgebung, in der die Funktion definiert wurde.. 78, 289, 299

Curry-Verfahren

Currying ist ein Mittel zur funktionalen Abstraktion. Das Curry-Verfahren basiert auf der partiellen Anwendung von Funktionen. Aus einer Funktion mit mehr als einem Argument können wir eine Menge von neuen Funktionen erzeugen, indem wir das erste Argument (oder die ersten Argumente) an einen festen Wert binden. Dieses leistet die Funktion *curry*.. 285

Datentyp

Ein Datentyp ist eine Äquivalenzklasse von Werten, auf die dieselben Funktionen und Operationen anwendbar sind. Ein Datentyp (kurz Typ) bestimmt

- die *Menge der Werte*, der eine Konstante angehört oder die von einer Variablen oder einem Term angenommen werden können, sowie die
- *anwendbaren Operationen* oder Funktionen auf diesen Werten (nach Wirth-1986).

Es gibt konkrete Datentypen, die von den Programmiersprachen direkt bereitgestellt werden (elementare Datentypen und strukturierte Datentypen), sowie abstrakte Datentypen, die wir selbst entwerfen, indem wir Konstruktoren und Akzessoren definieren.. **94**

determiniert

Ein **Algorithmus** ist *determiniert*, wenn er ein eindeutig bestimmtes Ergebnis liefert.. **16**

deterministisch

Ein **Algorithmus** heißt *deterministisch*, wenn alle Schritte vollständig geregelt sind, andernfalls heißt er *nicht-deterministisch*.. **16**

Funktionen höherer Ordnung

Funktionen höherer Ordnung (high order functions) sind Funktionen,

- die Funktionen als Argumente erhalten
- oder als Wert zurückgeben.

. **244, 252**

Funktionskomposition

Die Funktionskomposition $f \cdot g$ ist die Verknüpfung der beiden Funktionen f und g zu einer neuen Funktion h , die beide Funktionen nacheinander ausführt:

$$f \cdot g(x) = f(g(x))$$

. **288**

Implementation

Ein **Algorithmus**, der eine gegebene Spezifikation erfüllt, wird Implementation der Spezifikation genannt. Der Vorgang des Implementierens heißt *Implementierung*. Generell ist stets formal zu beweisen, daß eine Implementation ihre Spezifikation erfüllt.. [271](#)

Kardinalität

Die Mächtigkeit der Wertemenge eines Datentyps T heißt *Kardinalität von T* abgekürzt: $\text{card}(T)$.

Bei strukturierten Typen ist die Kardinalität das Produkt der Kardinalitäten der Typen der Komponenten.

Die Kardinalität ist ein Maß für die Menge an Speicherplatz, die für die Repräsentation der Werte dieses Typs nötig ist:. [95](#)

korrekt

Korrektheit: Ein Programm heißt *korrekt*, wenn es genau die Spezifikation erfüllt, also für alle Argumente des Definitionsbereichs die korrekten Resultate berechnet.

Partielle Korrektheit: Ein Programm heißt *partiell korrekt*, wenn es, sofern es terminiert, korrekte Resultate liefert.

Totale Korrektheit: Ein Programm heißt *total korrekt*, wenn es für *alle* Eingaben mit korrekten Resultaten terminiert.

. [187](#)

nebenläufig

Ein **Algorithmus** heißt *parallel* oder *nebenläufig*, wenn einige seiner Schritte gleichzeitig, z.B. durch mehrere Ausführende, bearbeitet werden können.. [16](#)

Parameter

An eine Funktionsdefinition gebundene Variable heißen *Parameter* oder *Argumente* einer Funktion.

Formale Parameter: In der *Funktionsdefinition* werden die gebundenen Variablen *formale Parameter (Formalparameter)* genannt.

Aktuelle Parameter: Bei der *Funktionsanwendung* werden die Formalparameter an feste Werte gebunden. Diese heißen die *aktuellen Parameter (Aktualparameter)*.

. 70, 244

Peanosche Axiome

Die Menge der natürlichen Zahlen \mathcal{N} ist rekursiv definiert durch das *Peanosche Axiomensystem*.

1. 1 ist eine natürliche Zahl.
2. Jede Zahl $a \in \mathcal{N}$ hat einen bestimmten Nachfolger $a' \in \mathcal{N}$.
3. Es gibt keine Zahl mit dem Nachfolger 1.
4. Aus $a' = b'$ folgt $a = b$.
5. Jede Menge M von natürlichen Zahlen, welche die Zahl 1 und zu jeder Zahl a auch den Nachfolger a' enthält, enthält alle natürlichen Zahlen ($M = \mathcal{N}$).

. 215, 219

Pragmatik

Die Pragmatik regelt die Verwendung einer **Programmiersprache**.

- Im Gegensatz zu Syntax und Semantik wird die Pragmatik nicht formal spezifiziert, sondern in Handbüchern informell beschrieben.
- Wichtige Aspekte der Pragmatik sind
 - das Verarbeitungsmodell und der Programmierstil,
 - Entwurfsmuster und Idiome für typische Lösungen,
 - Konventionen zur Dokumentation.

. 163, 249

Programm

Ein Programm ist die textuelle Beschreibung eines **Algorithmus** und mußpräzise, eindeutig und endlich sein.. 15

Programmiersprache

Eine Programmiersprache oder algorithmische Sprache ist ein Notationssystem für **Algorithmen**. Eine *Programmiersprache* über einem Alphabet Σ besteht aus

- einer *formalen Sprache* $L \subseteq \Sigma^*$, der Menge der syntaktisch richtigen Programme,
- einer Vorschrift, die syntaktisch richtigen Programmen eine Bedeutung zuordnet, der **Semantik von Programmiersprachen**,
- und Vereinbarungen zur richtigen Verwendung der Sprache, der **Pragmatik**.

. 17, 96, 167

Programmierstil

Der Fachbegriff „Programmierstil“ bezieht sich auf den *Modellierungsansatz* beim Programmentwurf und wird vom **Verarbeitungsmodell** bestimmt. Er meint nicht auf die saubere Arbeitsmethodik, wie Namenswahl, Kommentare, sorgfältiges Testen usw. **Programmiersprachen** sind jeweils für einen (oder mehrere) Programmierstile entworfen, enthalten aber in der Regel Schlupflöcher zu anderen Programmierstilen.

. 17

Prädikat

Ein Prädikat ist eine Funktion oder Prozedur, die ihre Argumente auf Wahrheitswerte abbildet.. 145

Reduktion: innere Reduktion

Bei der inneren Reduktion werden die Terme von innen nach außen reduziert.. 173

Reduktion: äußerer Reduktion

Bei der äußeren Reduktion werden die Terme von außen nach innen reduziert.. 173

Rekursion

Eine Definition ist rekursiv, wenn sie auf den gerade zu definierenden Begriff Bezug nimmt. Eine rekursive Definition muß immer einen trivialen Fall enthalten, der nicht mehr auf die Definition Bezug nimmt. Wir unterscheiden zwischen rekursiven *Definitionen* und rekursiven *Prozessen*.. 213, 214, 219

Rekursion, Verwendung

- Rekursive Funktionen werden verwendet, wenn ein schwieriges Problem durch eine Folge von Vereinfachungen auf ähnliche, aber leichtere Probleme zurückgeführt werden kann, bis ein trivialer Fall erreicht wird.
- Rekursion wird verwendet, wenn unendliche Prozesse oder Datenstrukturen in endlicher Form beschrieben werden sollen.
- Rekursive Funktionen werden zur Verarbeitung rekursiver Datenstrukturen verwendet.

. 214

Rekursion: allgemeine Rekursion

Die allgemeinen Rekursionen umfassen die linearen und die nicht-linearen Rekursionen. Wichtige allgemeine Rekursionen sind: **indirekte Rekursion, baumartige Rekursion, geschachtelte Rekursion**.

. 219

Rekursion: baumartige Rekursion

Eine **rekursive** Definition ist *baumartig*, wenn in der Definition in *einer* Fallunterscheidung *mehrfach* auf die Definition Bezug genommen wird.. 219, 220, 333

Rekursion: Endrekursion

Rekursive Funktionen, bei denen das Ergebnis der Rekursion nicht mehr mit anderen Termen verknüpft werden muß, heißen endrekursiv (engl. tail-recursion). Die zugehörigen Prozesse heißen iterative Prozesse. **Minimalrekursive** Funktionen sind endrekursiv. Um eine Funktion in eine endrekursive Form zu überführen, wird (meist) ein zusätzlicher Parameter, ein Akkumulator, benötigt, der die bisherigen Zwischenergebnisse sammelt (akkumuliert). Mit Erreichen der Abbruchbedingung steht dann das Ergebnis fest. Um den Akkumulator zu verbergen und korrekt zu initialisieren, sollte man eine einhüllende Funktion definieren.. 235, 257, 748

Rekursion: geschachtelte Rekursion

Eine **Rekursion** ist geschachtelt, wenn die Funktion in der rekursiven Verwendung selbst *als Argument* mitgegeben wird.. 219

Rekursion: indirekte Rekursion

Eine rekursive Definiton heißt *indirekt* oder *verschränkt*, wenn zwei oder mehrere Definitionen sich *wechselseitig* **rekursiv** verwenden.. 219

Rekursion: linear-rekursiv

Eine Funktionsdefinition, die sich auf der rechten Seite der definierenden Gleichung in *jeder* Fallunterscheidung selbst nur einmal verwendet, heißt *linear-rekursiv*.. 217, 258, 748

Rekursion: Minimal rekursiv

Eine Funktionsdefinition heißt **Rekursion: Minimal rekursiv**, wenn sie auf der rechten Seite der definierenden Gleichung

- nur aus einem *elementaren Fall*
- und einer *rekursiven* Verwendung der Funktion besteht.
- Der *elementare Fall* beschreibt die *Abbruchbedingung* und das Funktionsergebnis.

. 216, 235

Semantik von Programmiersprachen

Die Semantik eines Programms ist seine Bedeutung. Wir unterscheiden bei einer **Programmiersprache** zwischen **denotationeller** und **operationaler Semantik**. Die denotationelle und operationale Semantik einer Programmiersprache sind nicht notwendigerweise identisch:

- Auch wenn die denotationelle Semantik eindeutig ist, kann es sein, daß der Algorithmus der operationalen Semantik die Bedeutung nicht berechnen kann.
- Die denotationelle Semantik läßt es zu, prinzipiell unentscheidbare Probleme zu formulieren.

. 163, 182

Semantik, denotationell

Eine deklarative Beschreibung der Semantik, bei der ein Programm als eine statische Beschreibung von Algorithmen und Werten gesehen wird. Die Spezifikation ist völlig unabhängig von speziellen Implementationen.. 168

Semantik, operational

Die Bedeutung der primitiven Grundoperationen oder das Verfahren zur Berechnung der Semantik, und damit die Ausführung, wird spezifiziert. So wird über die Operationen das *Verhalten* des Programms, der erzeugte Prozeß, eindeutig festgelegt.. 168

Semiprädikat

Semiprädikate sind **Prädikate**, deren Resultat wie ein Wahrheitswert verwendet werden kann, aber im True-Fall nicht das Symbol #t, sondern eine andere sinnvolle Information verschieden von #f zurückgeben.. 119, 145, 191, 366

sequentiell

Ein **Algorithmus** heißt sequentiell, wenn alle Schritte streng hintereinander ausgeführt werden.. 16

strikt

Eine Funktion f heißt strikt, wenn $f(\perp) = \perp$, d.h. die Funktion hat nur für definierte Argumente einen definierten Wert.

- Eine **Programmiersprache** hat eine *strikte Semantik von Programmiersprachen*, wenn alle definierbaren Funktionen strikt sind. Das Gegenteil heißt *nicht-strikte Semantik*.

- Eine Programmiersprache verwendet eine *strikte Auswertung*, wenn alle Argumente einer Funktion vor der Anwendung der Funktion ausgewertet werden, wie beispielsweise in Racket und Common Lisp.
- Strikte Auswertung ist die Kombination von *vorgezogener Auswertung (eager evaluation)* und *innerer Reduktion*.

. 175, 182

Substitutionsmodell

Um die Bedeutung eines Ausdrucks nach dem Substitutionsmodell zu ermitteln, werden Terme durch andere Terme mit gleichem Wert ersetzt. Der Wert eines Ausdrucks ist Wert des *einfachsten äquivalenten Terms*. Ein Term ist *kanonisch* oder in *Normalform*, wenn er nicht weiter reduziert werden kann. Wenn die Bezugstransparenz gegeben ist, dann ist der ermittelte Wert unabhängig von der Reihenfolge der Ersetzungen.. 171, 175

terminiert

Ein **Algorithmus** *terminiert*, wenn er nach endlich vielen Schritten abbricht.. 16

Typsystem

Es gibt manifeste und latente Typsysteme.

- Eine Programmiersprache hat ein *manifestes Typsystem*, wenn für die Variablen bei der Deklaration Typen spezifiziert werden müssen. Manifeste Typsysteme sind typisch für die imperativen Programmiersprachen (Pascal, C, Java usw.)
- Eine Programmiersprache hat ein *latentes Typsystem*, wenn die Typen der Variablen aus den Typen der an diese Variablen gebundenen Werte abgeleitet werden: entweder
 - dynamisch zur Laufzeit des Programms wie in Racket, Scheme und Common Lisp oder
 - statisch zur Übersetzungszeit wie in Haskell und Miranda.

. 96

Verarbeitungsmodell

Wenn wir einen **Algorithmus** formulieren, beziehen wir uns auf ein **Modell** desjenigen Automaten, der den Algorithmus ausführen wird, das **Verarbeitungsmodell**. Ein Verarbeitungsmodell *abstrahiert* von der konkreten, verarbeitenden Maschine. Das Verarbeitungsmodell spezifiziert

1. die *Grundoperationen*, die ein Automat versteht und ausführen kann,
2. sowie das *Verhalten* des Automaten als Reaktion auf die Eingaben.

. 16, 17, 55, 163

Vollständige Induktion

Das 5. Peanosche Axiom heißt auch das Prinzip der **vollständigen Induktion** Eine häufig verwendete Fassung des Prinzips lautet:

Induktionsverankerung: Eine Aussage $A(n)$ sei richtig für die Zahl 1.

Induktionsschritt: Zeige, daß aus der Induktionsannahme $A(k)$ für eine natürliche Zahl k stets die Richtigkeit von $A(k')$ für den *Nachfolger* k' folgt.

Induktionsschluß: Die Aussage $A(n)$ gilt dann für *alle* natürlichen Zahlen $n \in \mathbb{N}$.

. 229

Wertesemantik

- Ein funktionaler Ausdruck oder Term wird einzig und allein dazu verwendet, einen *Wert* zu bezeichnen.
- Die Bedeutung eines funktionalen Ausdrucks ist der Wert, den er bezeichnet (denotiert).
- Ein Ausdruck hat *keine andere Wirkung*, als daß er einen Berechnungsprozeß auslöst, der diesen Ausdruck auf seinen Wert reduziert.

- Es gibt keine versteckten Wirkungen (Seiteneffekte), d.h. konkret, daß durch die Auswertung eines Terms die Umgebung des Terms nicht beeinflußt wird.

. 171, 244

Literatur

- Abelson, H., Sussman, G. J., and Sussman, J. (1996). *Structure and interpretation of computer programs*. The mit electrical engineering and computer science series. Mit Press Cambridge, Ma, 2 edition.
- Abelson, H., Sussman, G. J., and Sussman, J. (1998). *Struktur und Interpretation von Computerprogrammen*. The mit electrical engineering and computer science series. Springer, Berlin – Heidelberg - New York u.a., 3 edition.
- Abelson, H. and Sussman, J. (1985). *Structure and interpretation of computer programs*. The mit electrical engineering and computer science series. Mit Press Cambridge, Ma.
- Adams, D. (1981). *The Hitchhiker's Guide to the Galaxy*. Pocket Books, New York.
- Amarel, S. (1968). On representation of problems of reasoning about actors. In Michie, D., editor, *Machine Intelligence*, volume 3. Edinburgh University Press.
- Aristoteles (1995). *Nikomachische Ethik*, volume 3 of *Philosophische Schriften*. Felix Meiner Verlag, Hamburg. übersetzt von Eugen Rolffes, bearbeitet von Günther Bien.
- Bauer, F. L. and Goos, G. (1982). *Informatik: Eine einführende Übersicht, 2 Bände*. Heidelberger Taschenbücher. Springer-Verlag, Berlin – Heidelberg – New York.
- Becker, U. (1994). *The Element Encyclopedia of Symbols*. Element, Shaftesbury — Brisbane.
- Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Prentice-Hall, Englewood Cliffs, New Jersey.
- Bobrow, D. G. (1968). Natural language input for a computer problem-solving system. In Minsky, M., editor, *Semantic Information Processing*, pages 133–215. MIT Press, MIT Press.
- Carroll, L. (1951). *Through the Looking-Glass*. Pocket Books New York.
- Church, A. (1941). The calculi of lambda-conversion. *Annals of Mathematical Studies*, 6.
- Clocksin, W. F. and Mellish, C. (2003). *Programming in Prolog*. Springer, Berlin.
- Colby, K. (1975). *Artificial Paranoia*. Pergamon.

- Cremers, A. B., Griefahn, U., and Hinze, R. (1994). *Deduktive Datenbanken : eine Einführung aus der Sicht der logischen Programmierung*. Künstliche Intelligenz. Vieweg, Braunschweig.
- Dijkstra, E. W. (1968). Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148.
- Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurti, S. (2003). How to design programs: an introduction to programming.
- Flatt, M. (2006). Plt: Mzscheme: Language manual. im DrScheme-Helpdesk.
- Flatt, M., Findler, B., and PLT (2010). Guide: Racket. <http://pre.plt-scheme.org/docs/pdf/guide.pdf>.
- Flatt, M. and PLT (2010). Reference: Racket. <http://pre.plt-scheme.org/docs/pdf/reference.pdf>.
- Friedman, D. P., Byrd, W. E., and Kiselyov, O. (2005). *The Reasoned Schemer*. The MIT Press, Cambridge, MA.
- Friedman, D. P. and Felleisen, M. (1996). *The Little Schemer*. Mit Press Cambridge, MA, 4th edition.
- Görz, G. (1989). *Grundzüge der Informatik A2*. Prüfungsunterlagen.
- Graham, P. (1996). *Ansi Common Lisp*. Prentice-Hall, Englewood Cliffs, New Jersey, London.
- Hofstadter, D. R. (1980). *Gödel, Escher, Bach: an Eternal Golden Braid*. Vintage Books, New York.
- Hoyer, I. (1991). *Functional Programming with Miranda*. Pitman Publishing.
- Keene, S. (1989). *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley.
- Knuth, D. (1973). *The Art of Computer Programming*, volume III, Sorting and Searching. Addison-Wesley, Reading, Ma.
- Lakoff, G. (1987). *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. University of Chicago Press.
- Lippe, W.-M. (2009). *Funktionale und Applikative Programmierung*. Springer-Verlag, Berlin – Heidelberg.

- McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195.
- Meschkowski, H. (1971). *Mathematisches Begriffswörterbuch*. Number 99/a/b in B-I -Hochschultaschenbücher. Bibliographisches Institut, Mannheim – Wien – Zürich.
- Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers San Mateo, CA 1992, San Mateo, Calif.
- Page, B. (1991). *Diskrete Simulation: eine Einführung mit Modula-2*. Springer-Lehrbuch. Springer-Verlag, Berlin – Heidelberg – New York.
- Parr, T. (2010). *Language implementation patterns : create your own domain-specific and general programming languages*. The pragmatic programmers. Raleigh, NC : Pragmatic Bookshelf.
- Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (1988). *Numerical Receipes in C — The Art of Scientific Computing*. Cambridge University Press.
- Raphael, B. (1976). *The Thinking Computer: Mind Inside Matter*. Freeman & Co., San Francisco.
- Reiser, M. and Wirth, N. (1994). *Programming in Oberon — Steps Beyond Pascal and Modula*. ACM Press New York.
- Rowling, J. K. (1997). *Harry Potter and the Sorcerer's Stone*. Scholastic.
- Schefe, P. (1985). *Informatik — Eine konstruktive Einführung*. BI-Wissenschaftsverlag, Zürich.
- Schönfinkel, M. (1977). On the building blocks of mathematical logic. In van Heijenoort, J., editor, *From Frege to Gödel, A source book on mathematical logic, 1879–1931*. Harvard University Press, Cambridge, MA.
- Sperber, M., Dybvig, R. K., Flatt, M., and van Straaten, A. (2007). The revised⁶ report on the algorithmic language scheme. Im DrScheme-Helpdesk.
- Springer, G. and Friedman, D. (1989). *Scheme and the Art of Programming*. Addison-Wesley, Reading, Mass.
- Sterling, E. and Shapiro, E. (1994). *The Art of Prolog : Advanced Programming Techniques*. MIT Press, Cambridge, Ma.
- Strube, G., editor (1996). *Wörterbuch der Kognitionswissenschaft*. Klett-Cotta Stuttgart.

- Sussman, G. (1973). *A Computer Model of Skill Acquisition*. Elsevier.
- Weigert, A. and Zimmermann, H. (1971). *ABC Astronomie*. Verlag Werner Dausin, Hanau/Main.
- Weizenbaum, J. (1966). Eliza — a computer program for the study of natural language communication between men and machines. *Communications of the ACM*, 9:36–45.
- Weizenbaum, J. (1976). *Computer Power and Human Reason*. Free-man.
- Weizenbaum, J. (1978). *Die Macht der Computer und die Ohnmacht der Vernunft*, volume 274 of *Suhrkamp Taschenbuch Wissenschaft*. Suhrkamp-Verlag, Frankfurt am Main.
- Winograd, T. (1972). Understanding natural language. *Cognitive Psychology*, 3:1–191.
- Winskel, G. (1993). *The Formal Semantics of Programming Languages*. Mit Press Cambridge, Ma.
- Wirth, N. (1986). *Algorithmen und Datenstrukturen in Modula-2*. B.G. Teubner Stuttgart.
- Züllighoven, H. (1995). *Grundzüge der Informatik A, Prüfungsunterlagen*. Fachbereich Informatik der Universität Hamburg.
- Zwittlinger, H. (1981). *Comic Pascal*. R. Oldenbourg Verlag, München — Wien.