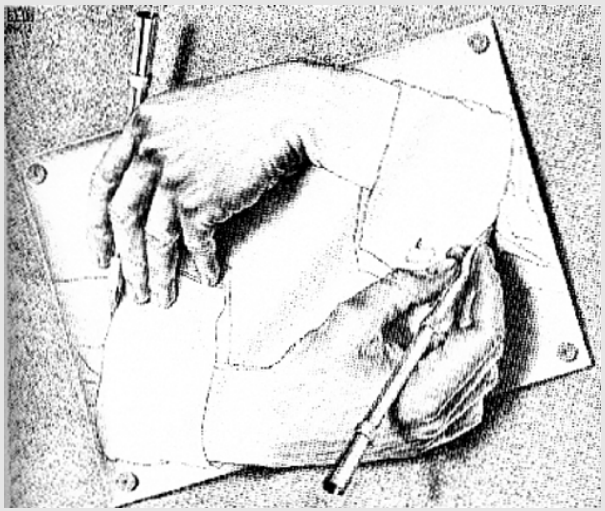
Softwareentwicklung III

Leonie Dreschler-Fischer

Department Informatik der Fakultät für Mathematik, Informatik und Naturwissenschaften, KOGS

WS 2017/2018



Übungsgruppen zu SE-III

Gr. 01 Di. 10-12 F-534 Vinodh 20

Gr. 02 Di. 10-12 G-210 Leonie Dreschler-Fischer 20

Gr. 03 Mi. 8-10 Uhr R-031 Benjamin Seppke 10

Gr. 04 Mi. 8-10 Uhr G-210 Rainer Jürgensen 6

Gr. 05 Mi 08-10 Uhr R-031 Benjamin Seppke 18

Gr. 06 Mi 10-12 Uhr F-009 Rainer Jürgensen 11

Gr. 07 Mi. 10-12 G-210 Rafael Epplée 9

Gr. 08 Mi. 12-14 R-031 Vinodh 20

Gr. 09 Mi. 12-14 G-102 Rafael Epplée 20

Gr. 10 Do. 08-10 R-031 Nicolas Möller 16

Gr. 11 Do. 08-10 F-009 David Mosteller 18

Gr. 12 Do. 10-12 R-031 David Mosteller 20

Gr. 13 Do. 10-12 F-534 Nicolas Möller 18

Klausuren

1. Klausur

Wann und wo: Mo, 19. Feb. 2018 09:30-11:30, ESA A, ESA B , Einlaß ab 9:15 Uhr

Erlaubte Hilfsmittel: Schreibzeug, ggfs. Wörterbuch für die deutsche Sprache

Ausweise: Ein Lichtbildausweis, Studentenausweis

2. Klausur

Tutorium: wird bekanntgegeben

Wann und wo: Di, 20. März 2018 09:30-11:30, ESA B, Einlaß ab 9:15 Uhr

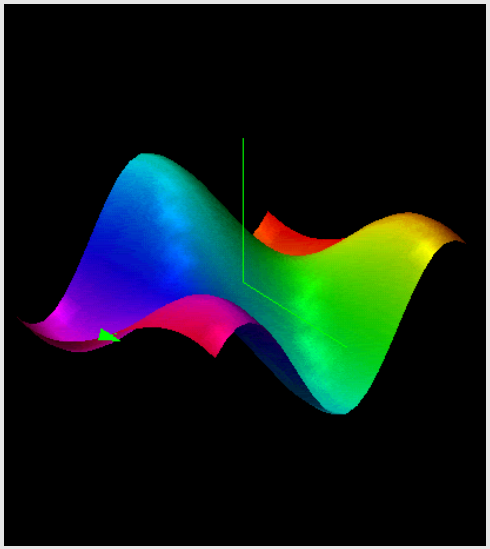
- ☞ Bitte seien Sie spätestens eine Viertelstunde vor Klausurbeginn da.
- ☞ Bitte bald zur Klausur anmelden!

Themen der Vorlesung

1. Einführung ▸ Teil I
2. Funktionale Abstraktion ▸ Teil II
3. Datenabstraktion ▸ Teil III
4. Repräsentation ▸ Teil IV
5. Semantik ▸ Teil IV
6. Der Baukasten ▸ Teil V
7. Closures ▸ Teil VI
8. Algorithmen ▸ Teil VII
9. Fallstudien ▸ Teil VIII
10. CLOS ▸ Teil IX
11. Metaprogrammierung ▸ Teil X
12. Relationale Programmierung ▸ Teil XI
13. Memoization, CLOS Anwendungen ▸ Teil XI
14. Heute: Einführung ▸ Teil I

Teil I

Grundlagen der funktionalen Programmierung



Warum applikative Programmierung?

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 epistomology — the study of the structure of knowledge from an imperative point of view taken by classical mathematical subjects.*

Warum applikative Programmierung?

- **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 Programierstil?

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.

Grundbegriffe der Programmierung

Algorithmus

Definition (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. [**?**]
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. [**?**]

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.

Fachbegriffe zu Algorithmen

Definition

- Ein Algorithmus heißt sequentiell,wenn alle Schritte streng hintereinander ausgeführt werden.
- Ein Algorithmus heißt parallel oder nebenlaeufig, 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.

Verarbeitungsmodell

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

Definition (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.

Grundstile der Programmierung

Den Verarbeitungsmodellen entsprechen Grundstile der Programmierung, kurz Programmierstile genannt.

- Imperative Programmierung
  - von-Neumann Programmierung
  - Objektorientierte Programmierung
- Funktionale oder applikative Programmierung
- Logische Programmierung, auch relationale Programmierung genannt.
- Stromorientierte Programmierung . . .

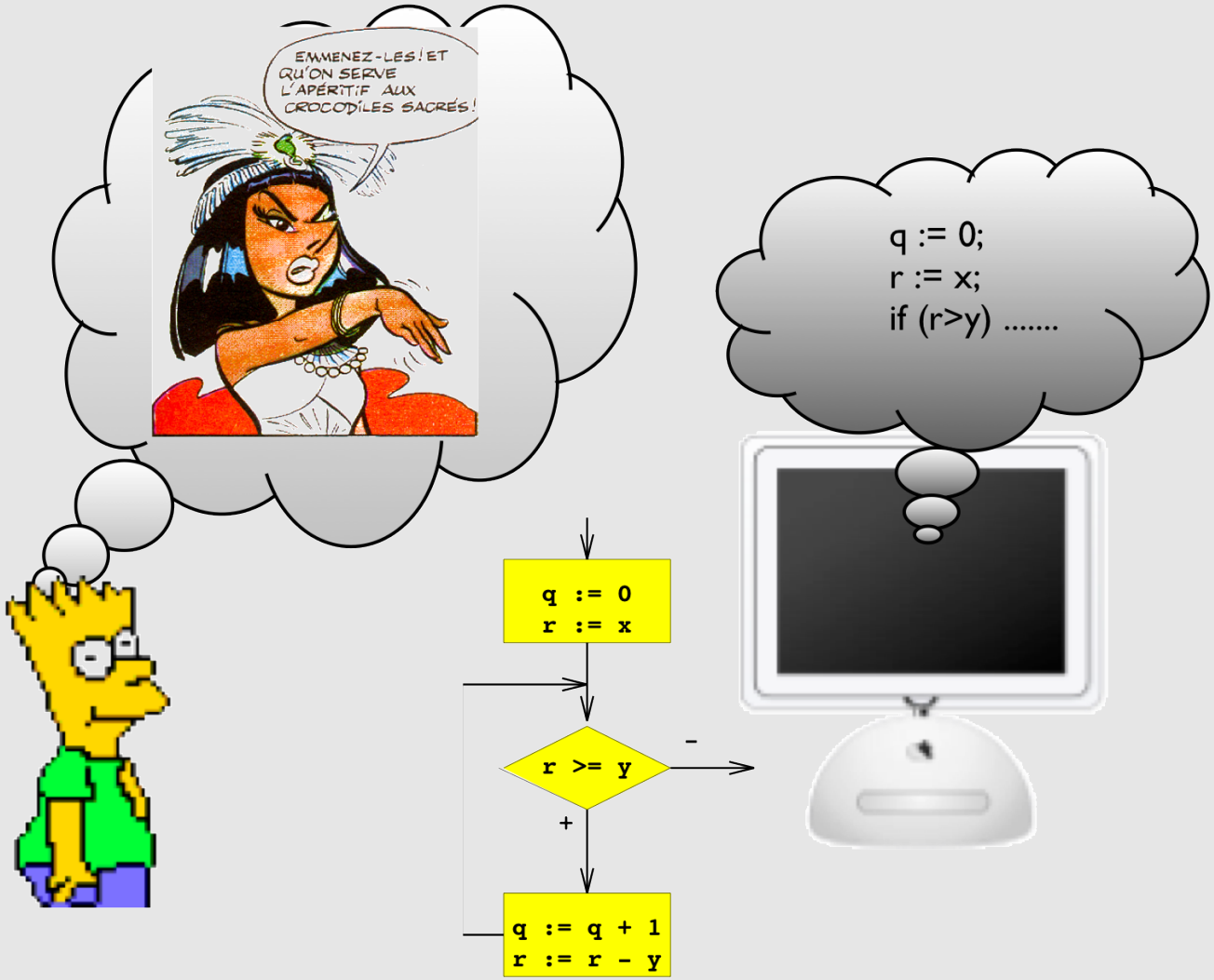
☞ Beachte:

- 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.

Das imperative Verarbeitungsmodell



Das imperative Verarbeitungsmodell

Merkmale

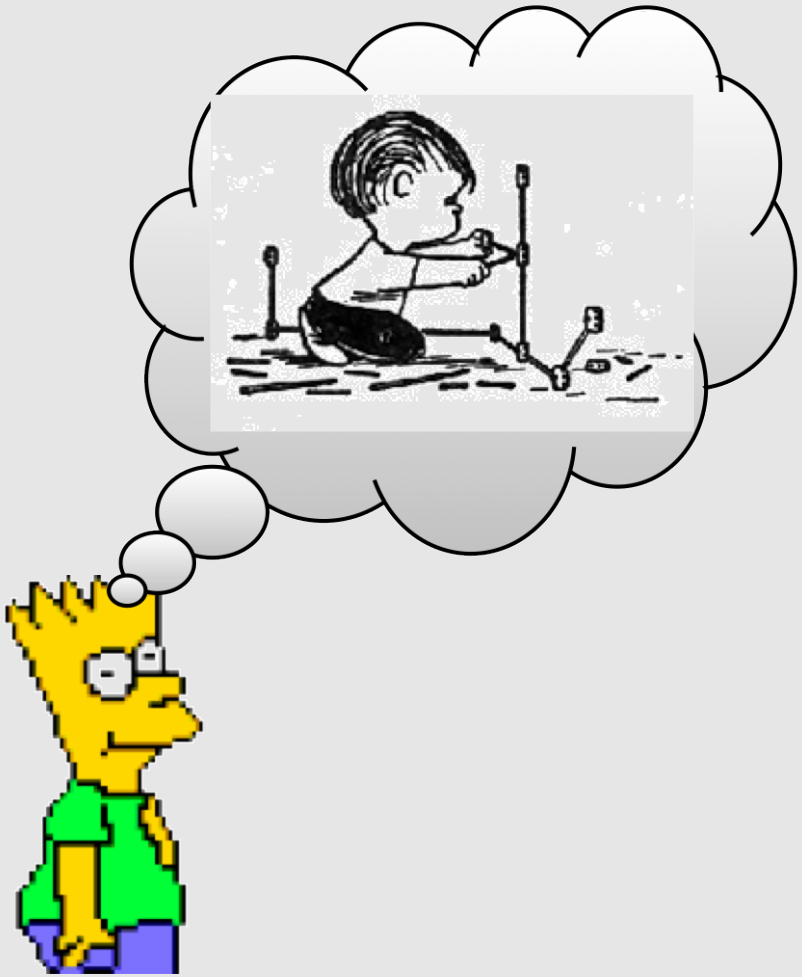
- Der Programmablauf führt zu einer Folge von Zustandsänderungen der Variablen oder Objekte des Programms.
- Die Zustandsänderungen werden durch Anweisungen oder Nachrichten bewirkt.
- Die Reihenfolge der Anweisungen wird durch Kontrollstrukturen (Schleifen, Verzweigungen, früher auch Sprünge) gesteuert.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

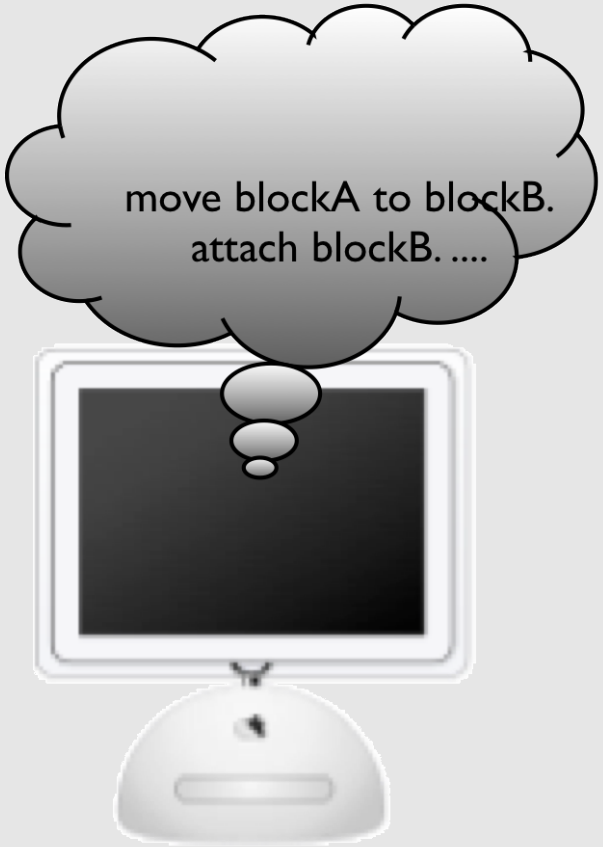
Ein imperativer Multiplikationsalgorithmus

Beispiel

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.

Das objektorientierte Verarbeitungsmodell

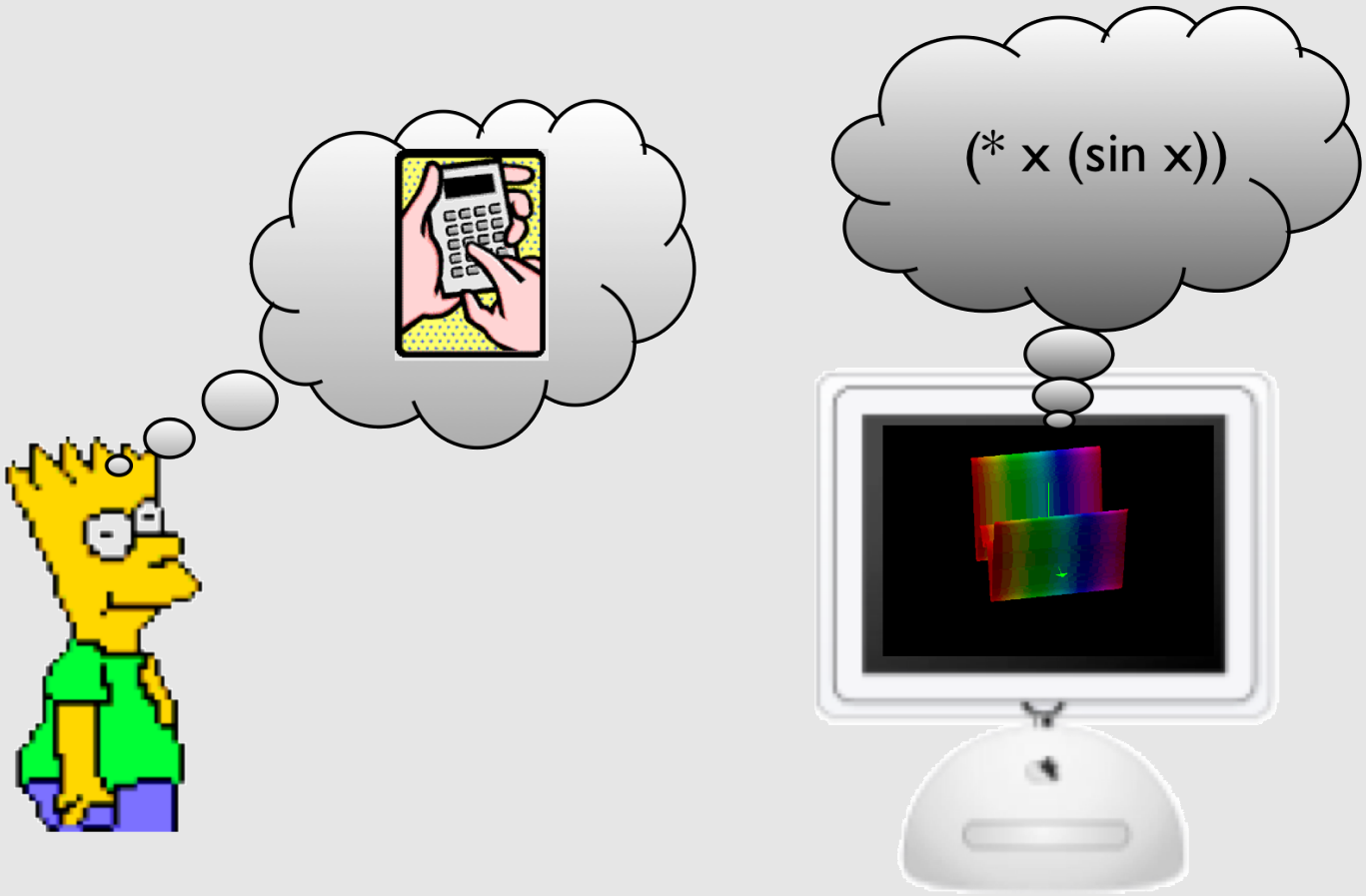




Merkmale des OO-Verarbeitungsmodells

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

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 Vorzug 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.

Ein funktionaler Multiplikationsalgorithmus

Beispiel

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

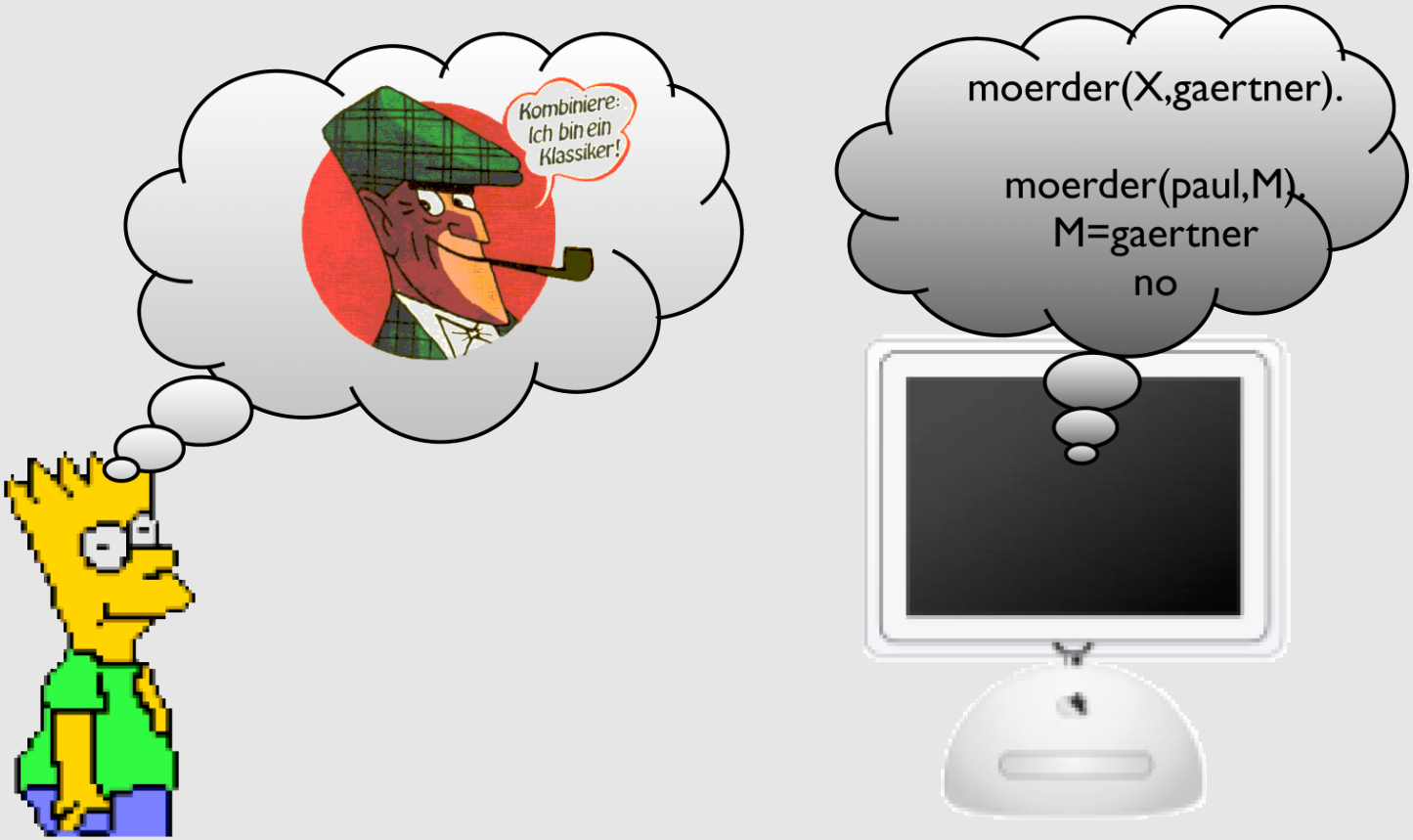
- parallel
- nicht-deterministisch
- terminierend
- determiniert
- funktional, rekursiv

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 $\lambda-$Kalküls [**?**].
- 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.

Das relationale/logische Verarbeitungsmodell



Merkmale des logischen / relationalen Verarbeitungsmodells

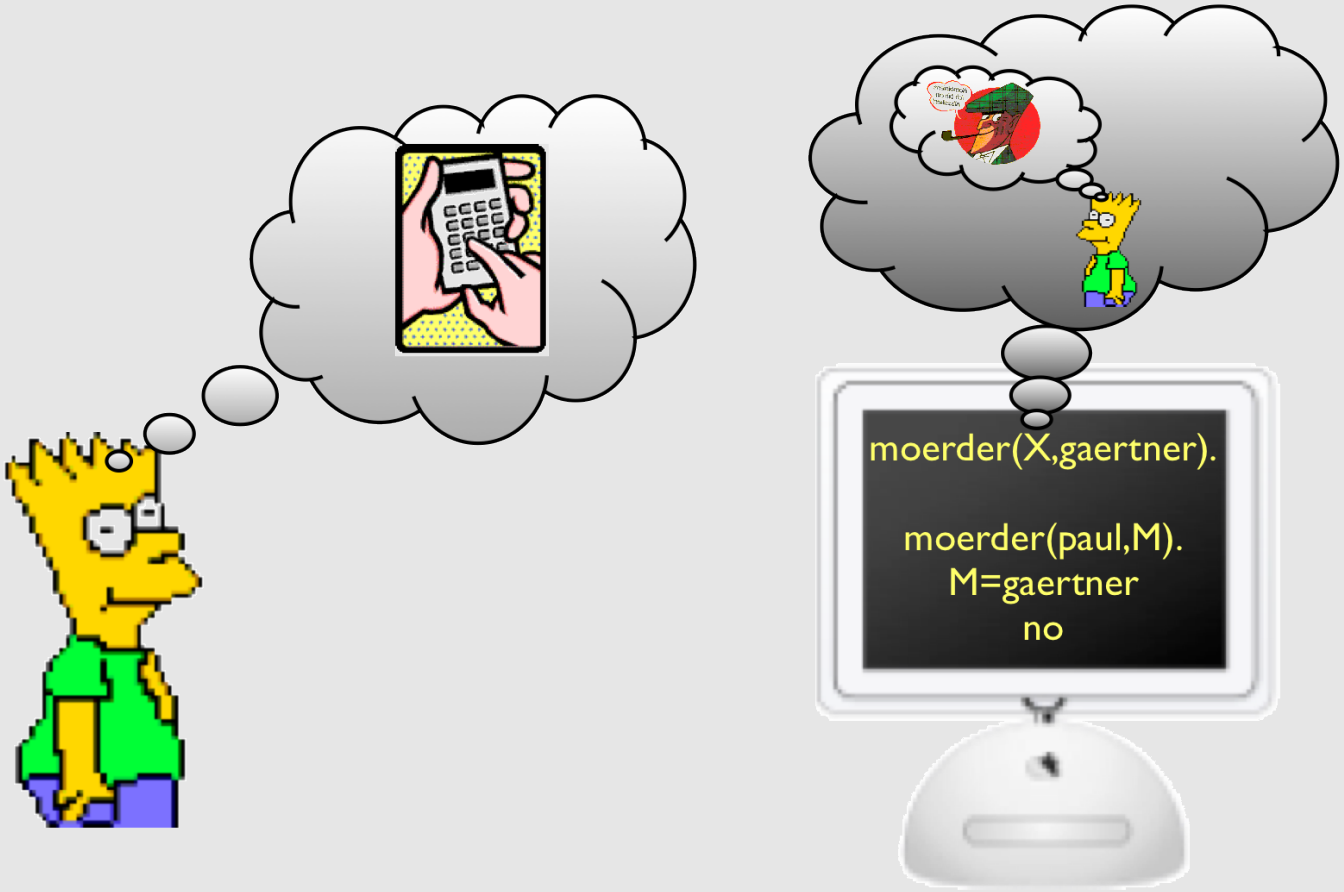
- Explizite Modellierung von Fakten und Beziehungen
- Vordefinierte Such- und Schlußfolgerungsmechanismen
- Einfache Syntax, leicht zu lernen
- Formal definierte Semantik
- Wie der funktionale Programmierstil sehr gut für komplexe Programmieraufgaben geeignet.

Das stromorientierte Verarbeitungsmodell

Merkmale des stromorientierten Verarbeitungsmodells

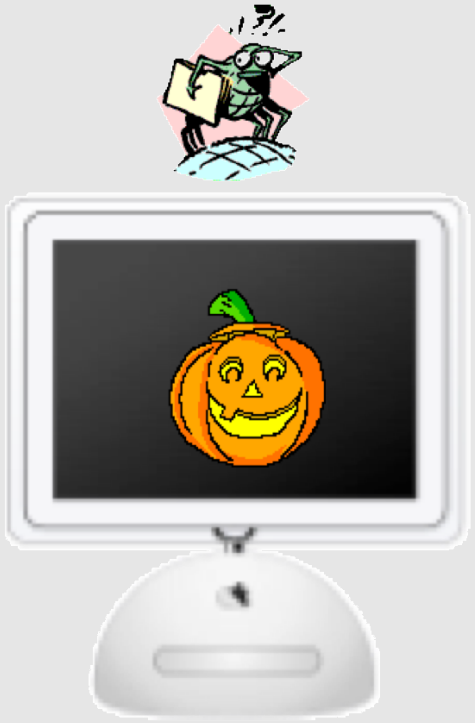
- Das stromorientierte Verarbeitungsmodell ist besonders gut geeignet für die fließbandartige Verarbeitung von (endlosen) Strömen gleichartiger Daten (Objekte).
- Programmkomponenten sind Quellen und Senken von Strömen, sowie Filter, Transformatoren und Kombinatoren.
- Beispiele: Bildverarbeitungssysteme,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

Das Benutzermodell

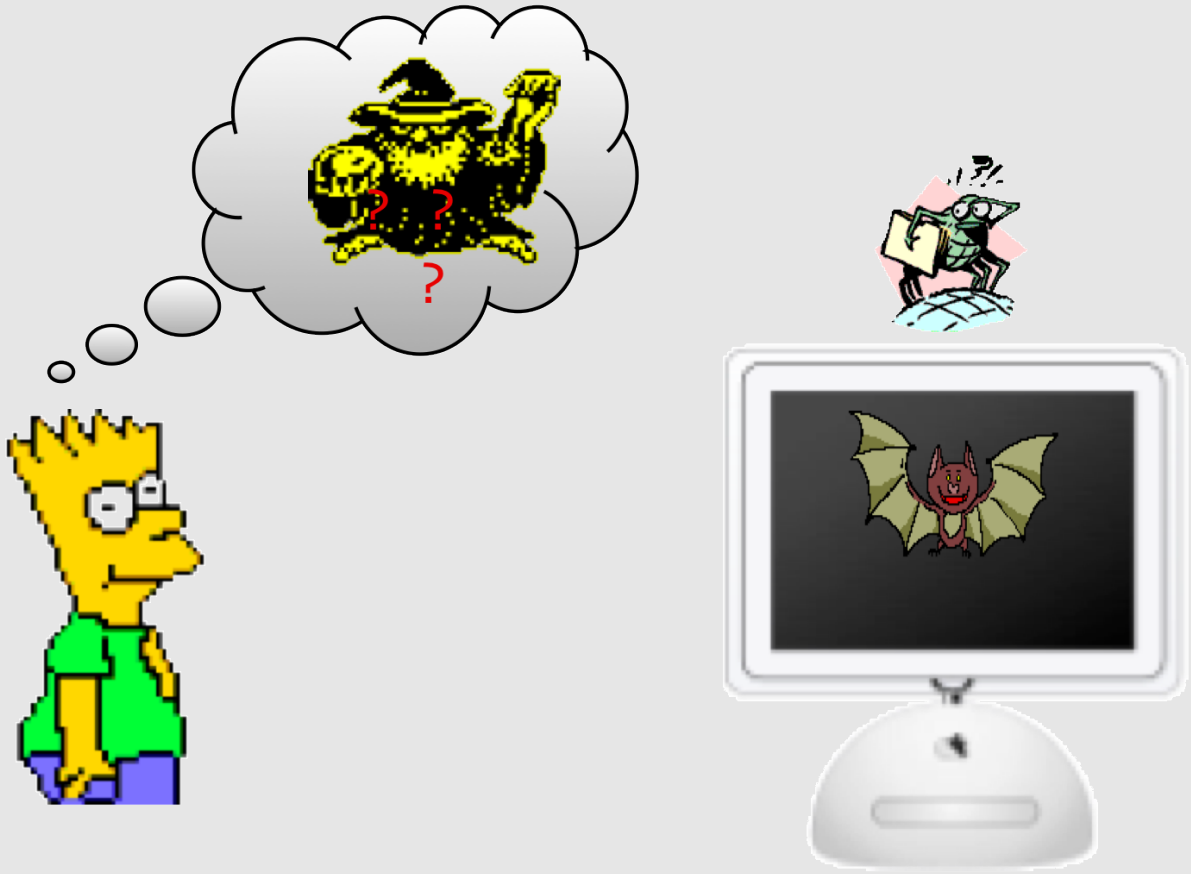


Das mystische Verarbeitungsmodell

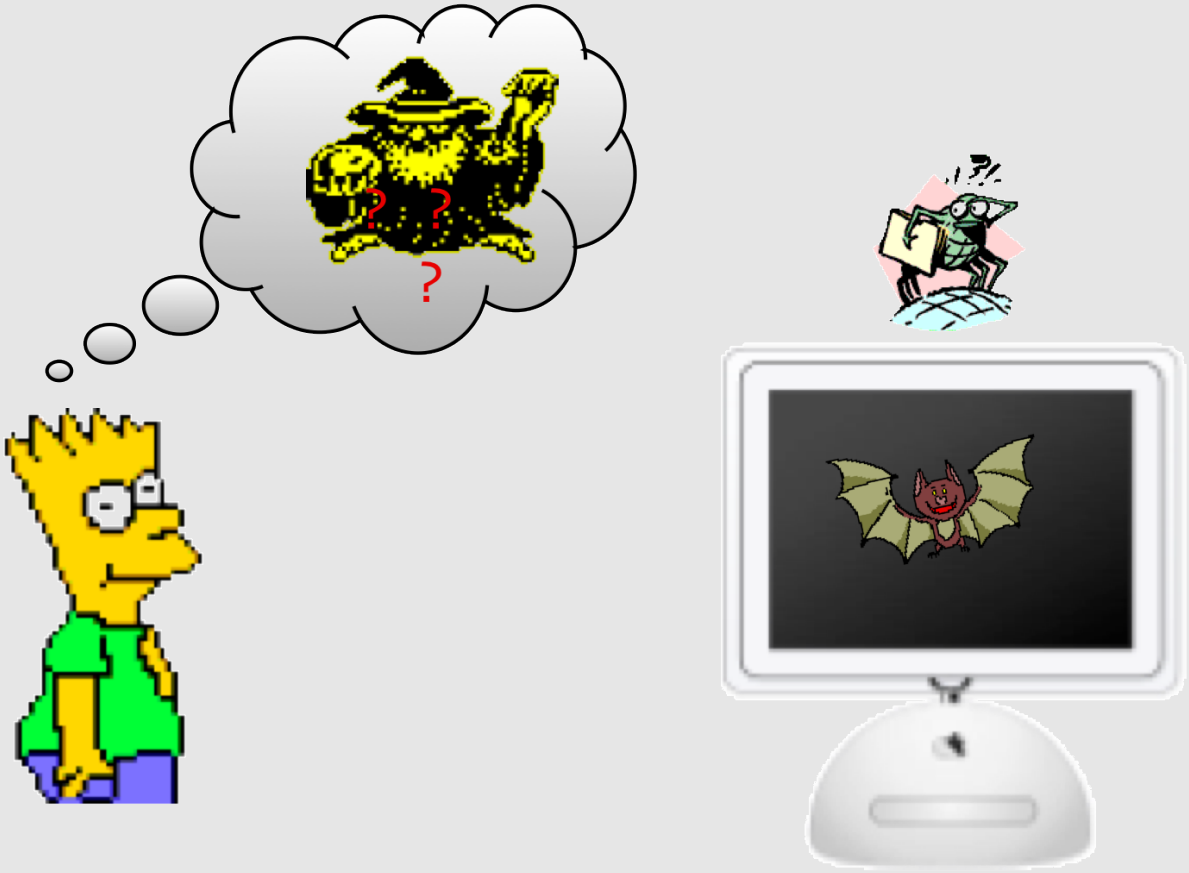
Das mystische Verarbeitungsmodell



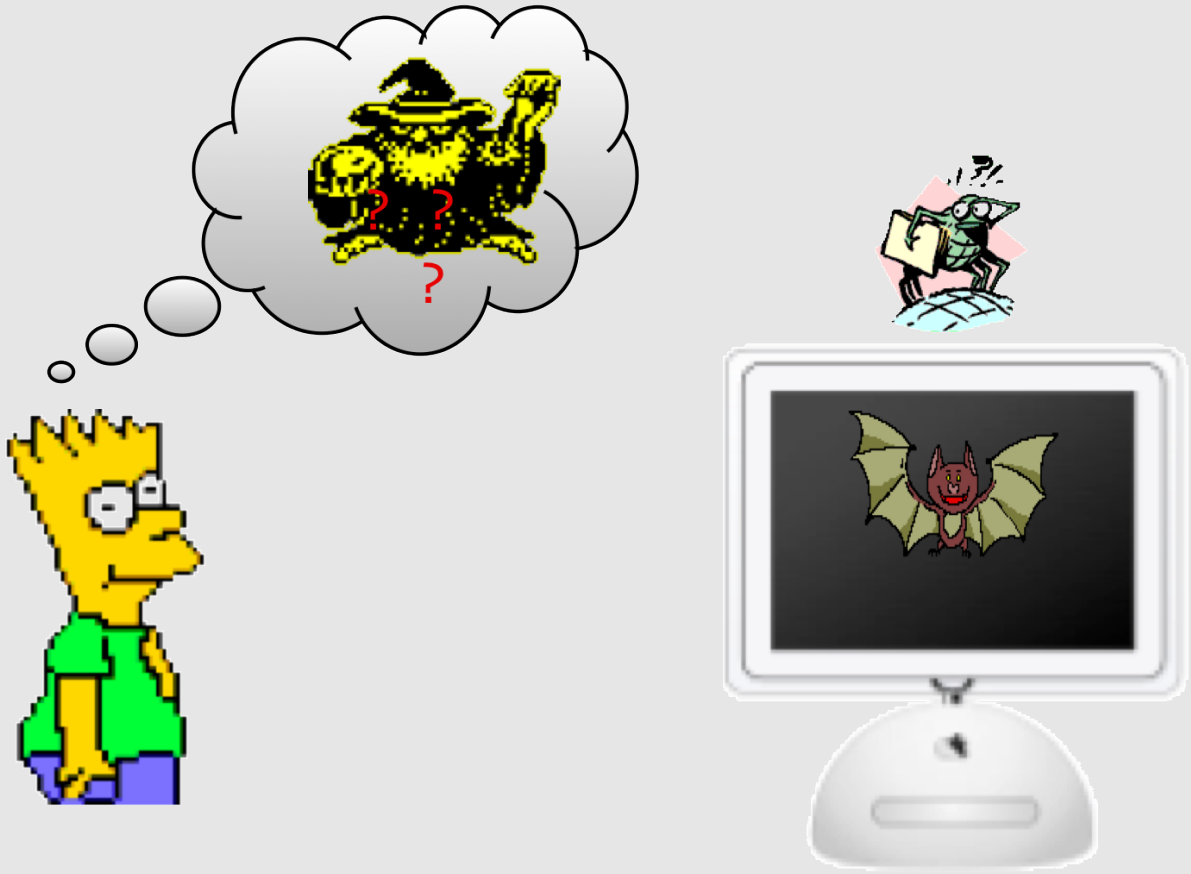
Das mystische Verarbeitungsmodell



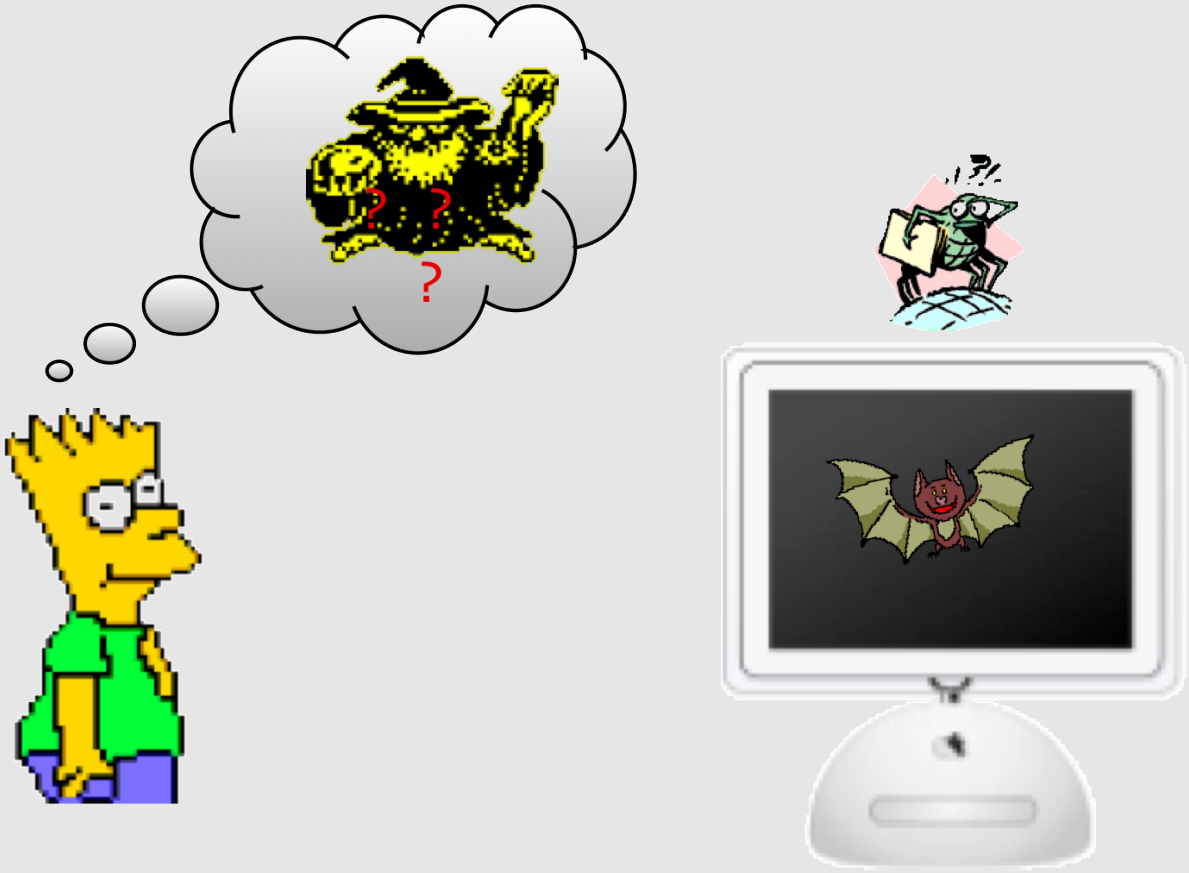
Das mystische Verarbeitungsmodell



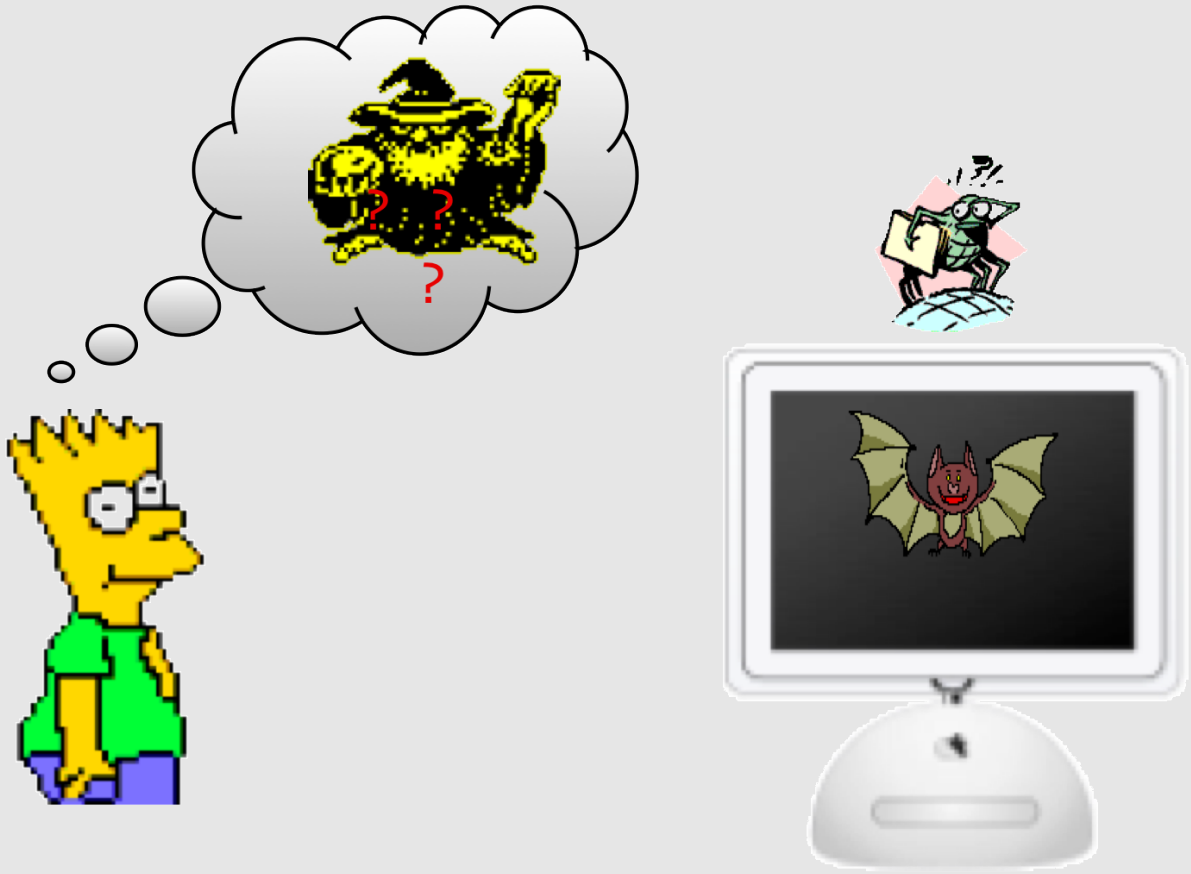
Das mystische Verarbeitungsmodell



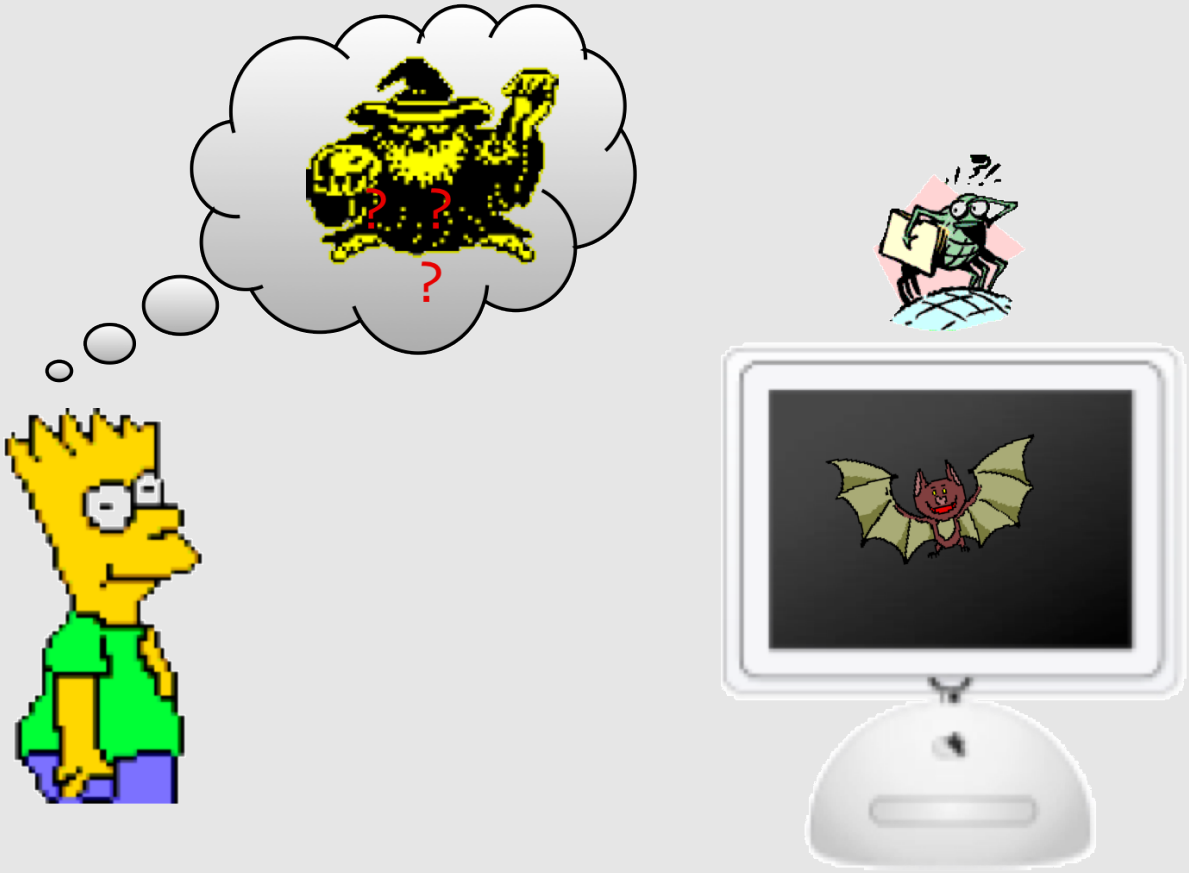
Das mystische Verarbeitungsmodell



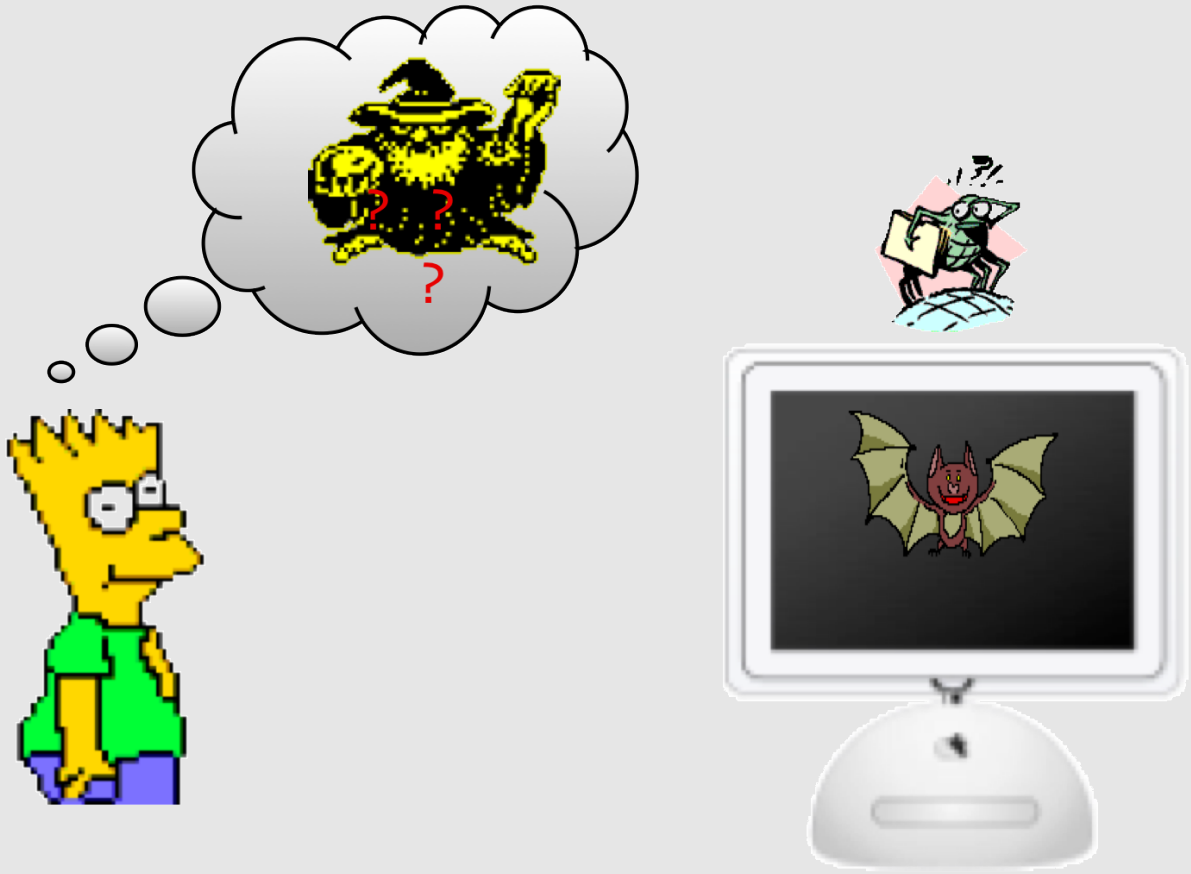
Das mystische Verarbeitungsmodell



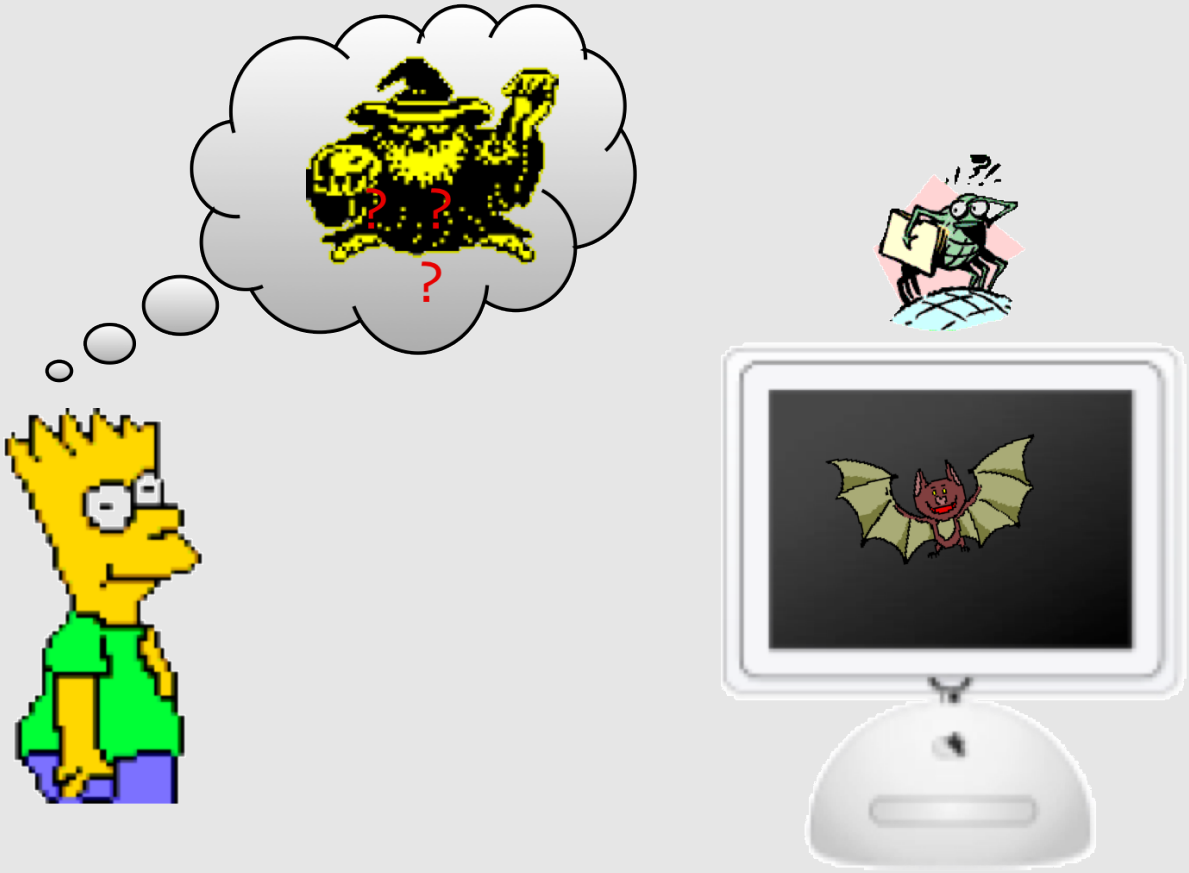
Das mystische Verarbeitungsmodell



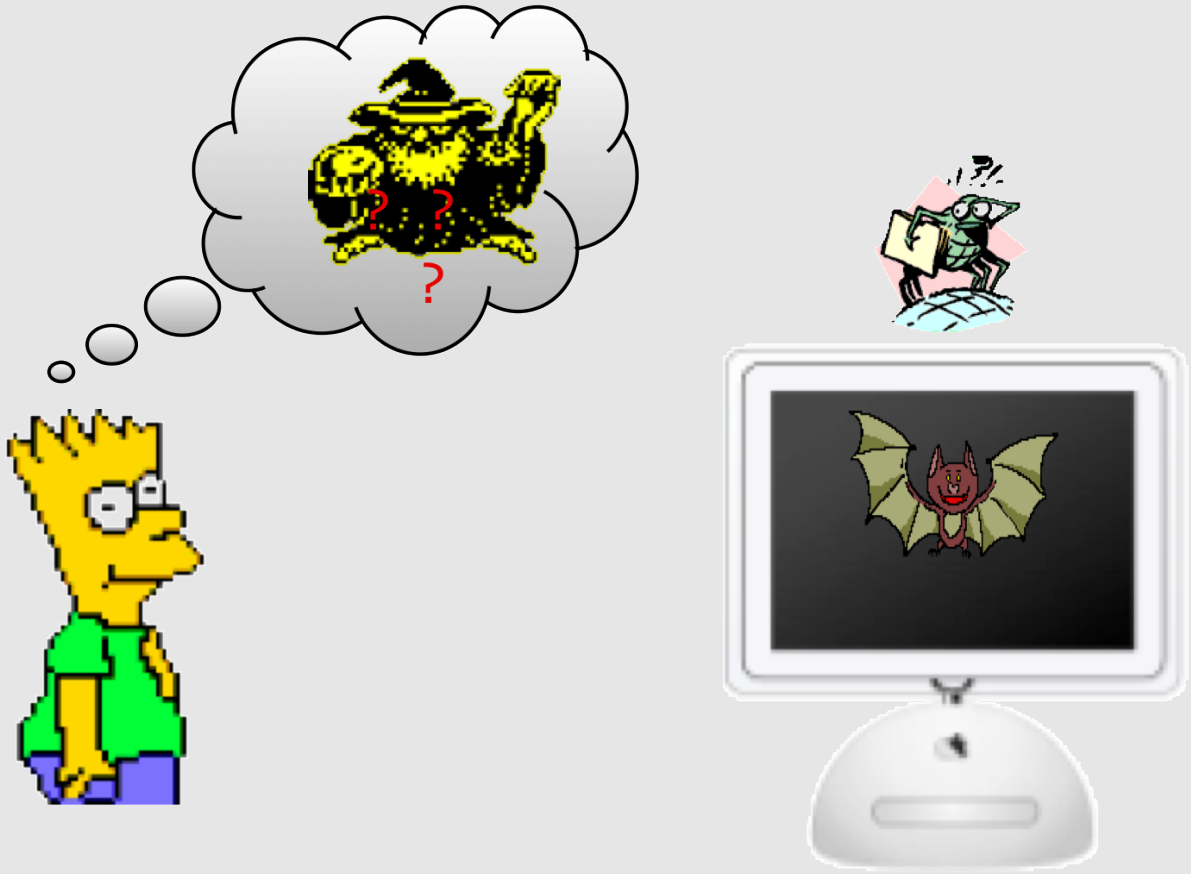
Das mystische Verarbeitungsmodell



Das mystische Verarbeitungsmodell



Das mystische Verarbeitungsmodell



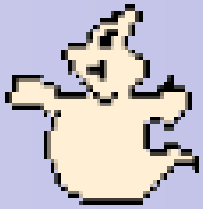
Das mystische Verarbeitungsmodell

Das mystische Verarbeitungsmodell

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.

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!

Weg damit!

- 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 [**?**])

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 (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: über die universelle Turingmaschine

Programmiersprachen für die funktionale Programmierung

Programmiersprachen für die funktionale Programmierung

- Es gibt zwei große Familien von funktionalen Programmiersprachen:

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,

Programmiersprachen für die funktionale Programmierung

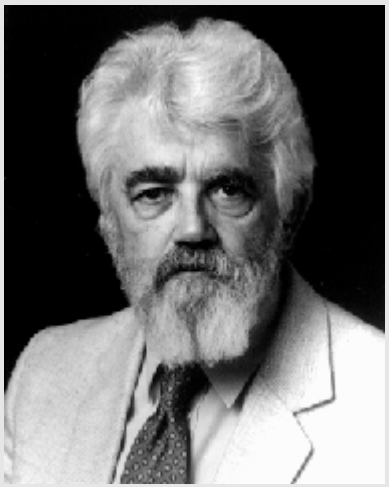
- Es gibt zwei große Familien von funktionalen Programmiersprachen:
  1. Die Lisp-Familie (Symbole, dynamische Typisierung, hybrid), zu der Racket, Scheme und Common Lisp gehören,
  2. sowie die Familie der streng-getypten, referentiell transparenten Sprachen, mit Vertretern wie Haskell und Miranda.

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 $\lambda-$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 **Lis***t* **p***rocessing*.

Der Name„Lisp“

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

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.

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.

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.

Relevanz

- Lisp ist eine weit verbreitete, relevante Programmiersprache, insbesondere auch für KI[1]-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. ***[?]***

[1] Künstliche 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.

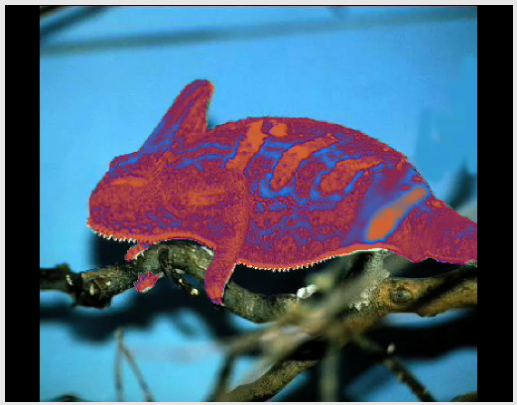
When new styles of programming were invented, other languages died out; Lisp simply incorporated the new styles by defining some new macros. *Peter Norvig*

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.“* *[?]*

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.“* *[?]*

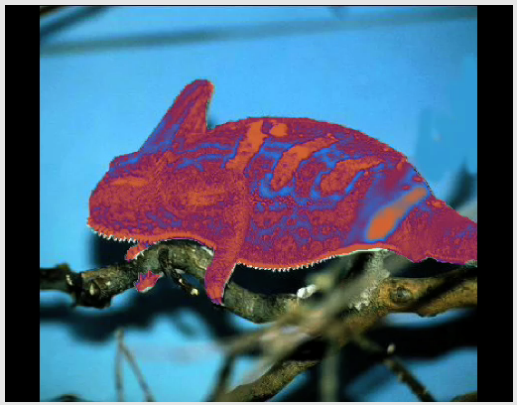


Das Chamäleon Lisp

- *„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.“* *[?]*

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.“* *[?]*



Das Chamäleon Lisp

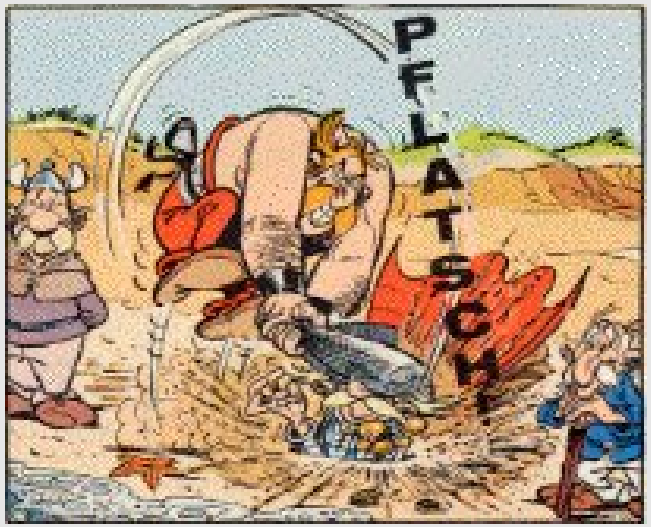
- *„Lisp's flexibility allows it to adapt as programming styles change, but more importantly, Lisp can adapt to your particular programming problem.*
- *In other languages you fit your problem to the language; with Lisp you extend the language to fit your problem.“* *[?]*

Es ist wichtig, mehr als ein Verarbeitungsmodell zu kennen.

Merke:

Für einen Menschen mit einem Hammer sieht alles wie ein Nagel aus.

Es ist wichtig, mehr als ein Verarbeitungsmodell zu kennen.



Merke:

Für einen Menschen, dem ein Hammer fehlt, sieht alles wie ein Hammer aus!

Lisp ist eine effiziente Programmiersprache

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

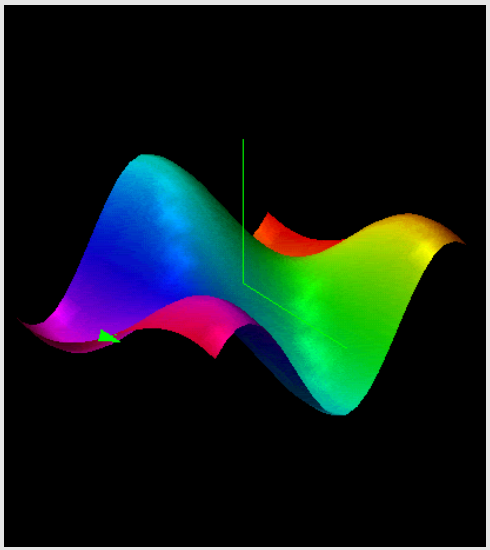
Das Gegenteil ist wahr:

- Lisp-Programme können wegen der sehr guten Programmierumgebungen sehr effizient entwickelt werden.
- Die Compiler optimieren endrekursive Funktionen
- 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.

Lisp ist eine effiziente Programmiersprache

- Es zwar gibt Speicher- und Rechenzeit-aufwendige Lisp-Systeme, aber das liegt an der Komplexität der Probleme (oder an unerfahreren Programmierern) und nicht an Lisp .
- Für besonders schwierige Probleme ist Lisp eben oftmals die beste Sprache, um diese Aufgaben überhaupt bewältigen zu können.

Kommentiertes Literaturverzeichnis



Funktionale Programmierung

- Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurti, S. (2003).
  How to design programs: an introduction to programming.
- 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., 3rd edition.

Funktionale Programmierung: Formale Grundlagen

- 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.

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.

Relationale Programmierung

- 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.

Relationale Spracherweiterungen

- 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.

Handbücher und Spracheinführungen

- Sperber, M., Dybvig, R. K., Flatt, M., and van Straaten, A. (2007).
  The revised^6 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

Organisatorisches

Materialien

Materialien:

- Die Wep-page zur Vorlesung:
  `http://kogs-www.informatik.uni-hamburg.de/~dreschle/teaching/Lehre/Lehre.html`

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	21.10.	–.
1	Notation, Funktionen	25	21.10.	31.10.
2	Namen, Symbole, Zahlen	24	28.10.	7.11.
3	Listen und Strings	30	4.11.	14.11
4	Semantik	30	11.11.	21.11.
5	Rekursion	30	18.11.	28.11.
6	Funkt. höh. Ordnung	30	25.11.	4.12.
7	Closures	35	2.12.	12.12.
8	Baumrekursion	35	9.12.	19.12.
9	Kombinatorik	35	16.12.	2.1.
10	Gener. Funktionen	40	6.1.	16.1.
11	Methodenkombination	40	13.1.	23.1.
12	Prolog	40	20.1.	30.1.
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 2014/15

Zulassungsvoraussetzung: Erfolgreiche Teilnahme an den Übungen zu SE-III

Termine: In der vorlesungsfreien Zeit:

- Mo, 19. Feb. 2018 09:30-11:30, ESA A, ESA B , Einlaß ab 9:15 Uhr
- Wiederholungsklausur: Di, 20. März 2018 09:30-11:30, ESA B, Einlaß ab 9:15 Uhr

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

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`.

Das Racket-System

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

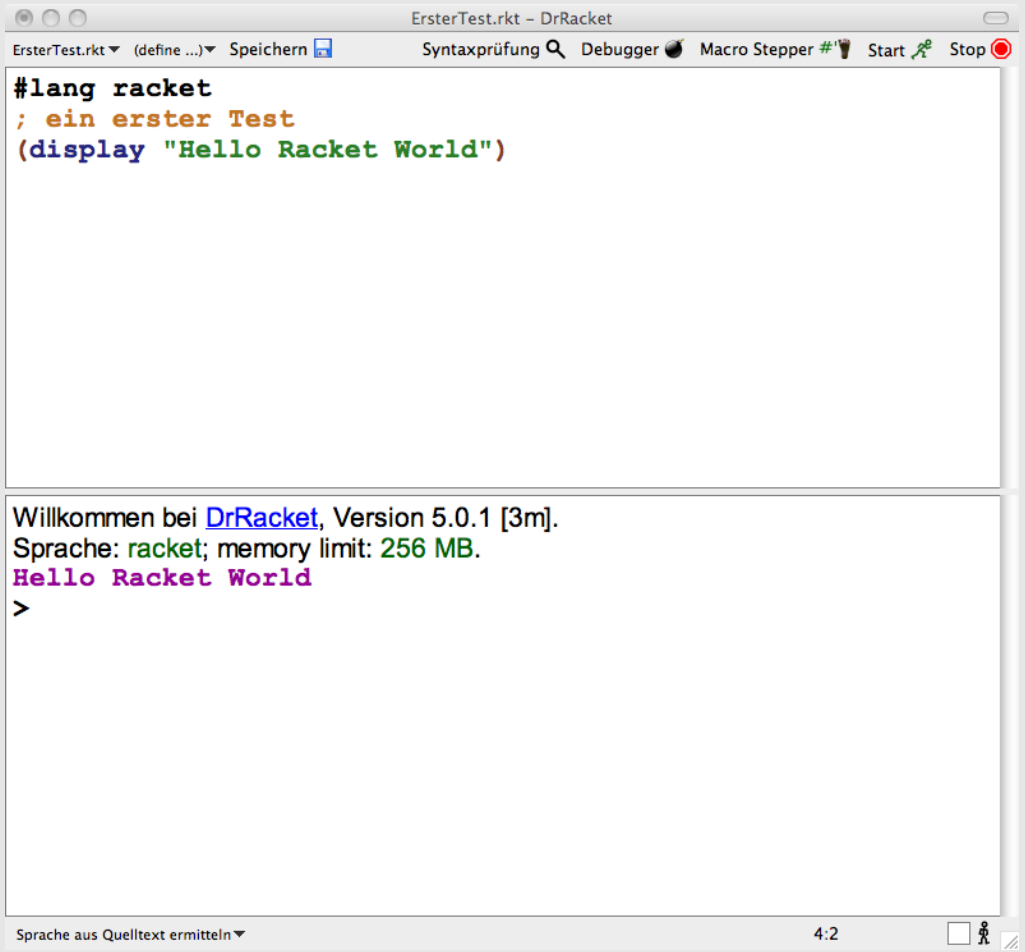
Wir können das Racket-System wie einen Taschenrechner benutzen: Ein Teil des Systems, der *evaluator*, wertet funktionale Ausdrücke, *expressions genannt*, aus und zeigt den Wert an:

```
(+ (* 3 3)
   (* 4 4)) ⟹
```

$\Longrightarrow$ 25

Aufruf von DrRacket

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



Das DrRacket-Hauptfenster

Das Hauptfenster hat zwei Teile:

Ein Editor für Racket-Programme in der oberen Häfte,

Eine Auswertungsumgebung in der untere Hälfte.

In der Auswertungsumgebung läuft der sogenannte toplevel, der

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

Wahl der Sprache

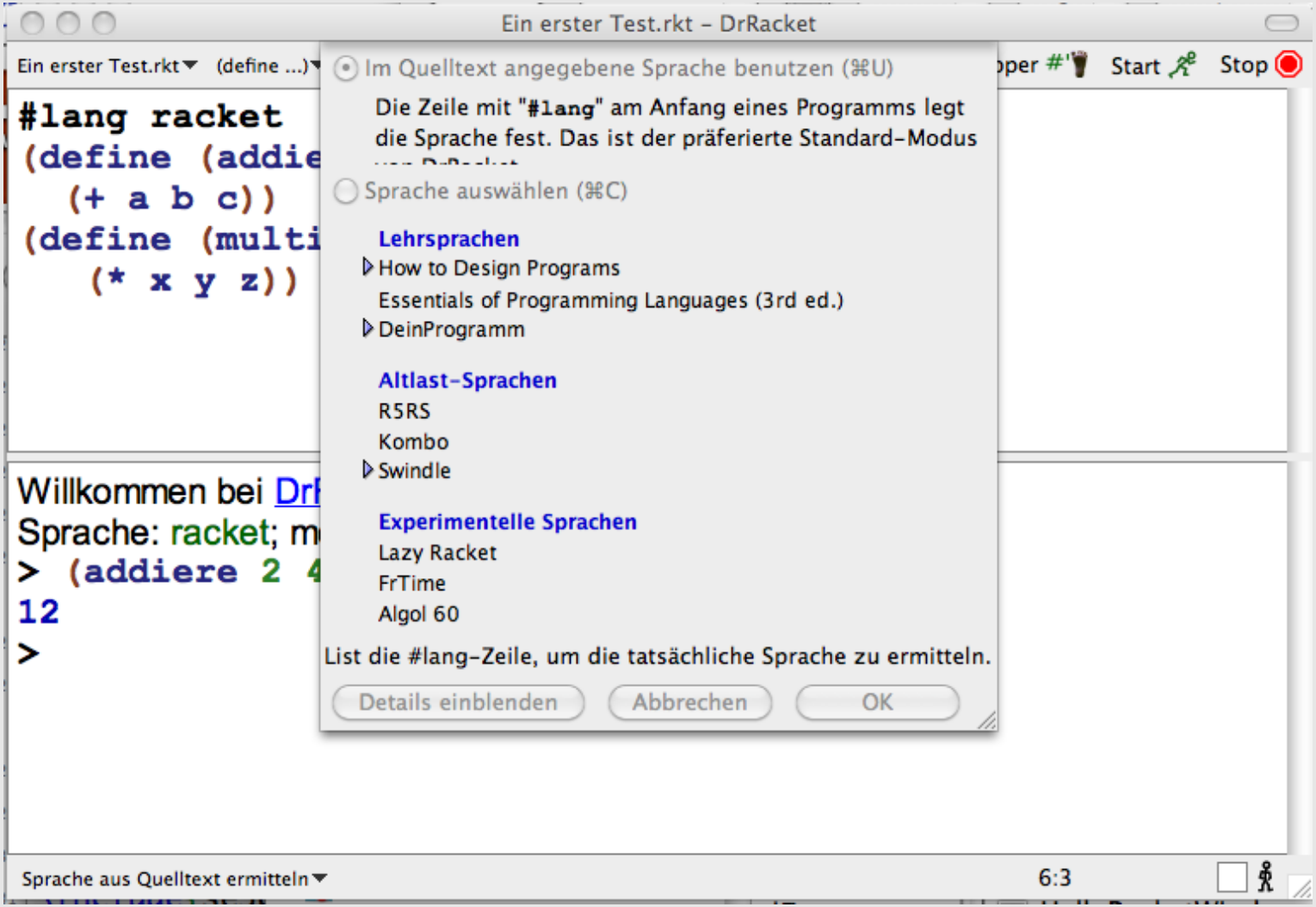
Im DrRacket-System sind mehrere Racket-Dialekte implementiert, die unterschiedliche Teilmengen der Sprache bieten:

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.


Ein erster Test.rkt – DrRacket
Ein erster Test.rkt
(define ...)
Start
Stop
#lang racket
(define (addie
(+ a b c))
(define (multi
(* x y z))
Willkommen bei DrR
Sprache: racket; m
> (addiere 2 4
12
>
Im Quelltext angegebene Sprache benutzen (⌘U)
Die Zeile mit "#lang" am Anfang eines Programms legt die Sprache fest. Das ist der präferierte Standard-Modus
Sprache auswählen (⌘C)
Lehrsprachen
How to Design Programs
Essentials of Programming Languages (3rd ed.)
DeinProgramm
Altlast-Sprachen
R5RS
Kombo
Swindle
Experimentelle Sprachen
Lazy Racket
FrTime
Algol 60
List die #lang-Zeile, um die tatsächliche Sprache zu ermitteln.
Details einblenden
Abbrechen
OK
Sprache aus Quelltext ermitteln
6:3

In der Auswertungsumgebung

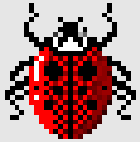
```
Welcome to DrRacket, Version 5.0.1 [3m].
Language: Racket.
> 4
4
> (expt 2 4)
16
> (sin 1.5707963267949)
1.0
> (* (+ 2 2) (- 2 3))
-4
>
```

Fehlermeldungen

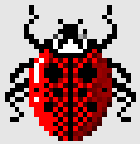
```
> (*2 5)
reference to undefined identifier: *2
> (* 2 5)
10
>
```

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.

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

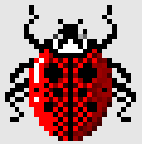
Auf den folgenden Folien wird die Auswertung der Racket-Ausdrücke durch den Evaluator abkürzend dargestellt:

> (+ 1 2) $\longrightarrow$ 3

- Das linke Größerzeichen $>$ ist die Eingabeaufforderung (prompt) von DrRacket.
- Der Pfeil $\longrightarrow$ soll bedeuten: *Evaliert 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
>
```

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)))  ;
```

Gut formatiert:

```
(or     ;    (√5 > √4) ∨ (√6 > √7)
    (> (sqrt 5)
       (sqrt 4))
    (> (sqrt 6)
       (sqrt 7)))  ;
```

Schlecht formatiert:

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

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))))) ;
```

$\cos\alpha = \sqrt{1 - \sin^2\alpha}$

Teil II

Konstrukte der funktionalen Programmierung

Konstrukte der funktionalen Programmierung

sin2cos

Die Elemente der applikativen Programmierung

Das Verarbeitungsmodell basiert auf dem mathematischen Funktionenbegriff.
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)

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 (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.

  (**define** *<Name>* *<Wert>*) *oder*
  (**define** *<Name>* *<s−expression>*)

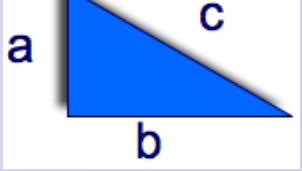
- Die *special form expression* (**define**) definiert eine Variable namens $<$*Name*$>$ und initialisiert sie mit dem Wert des angegebenen Arguments.
- Der Wert eines (**define**)-Ausdrucks ist undefiniert $\perp$.

Verwendung von Namen

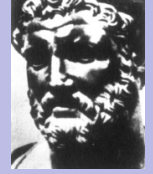
- Namen müssen eindeutig sein. Daher dürfen wir für unsere Definitionen keine Namen verwenden,
  - die schon als Standardfunktionen (wie **sin**, **sqrt**),
  - syntaktische Schlüsselwörter (wie **define**, **begin**, **if**),
  - Konstanten (wie **#t**, **#f**)

  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

Berechne die Hypothenuse eines Dreiecks $c = \sqrt{a^2 + b^2}$:
Im Editor-Fenster von DrRacket schreiben wir:

```
#lang racket
;;; Mein erstes Racket-Programm:
;;; Berechne die Hypothenuse 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
```

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

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

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: Variablennamen 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 abC 4)                    ⟶ ⊥
> (= abc abC)                       ⟶ #f
> abc                               ⟶ 3
> abC                               ⟶ 4
```

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

Radius = 4: Volumen = `4/3 * π * 4 * 4 * 4`

Radius = 5: Volumen = `4/3 * π * 5 * 5 * 5`

Radius = 6: Volumen = `4/3 * π * 6 * 6 * 6`

Radius = ❒: Volumen = `4/3 * π * ❒ * ❒ * ❒`

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)
                              ⟶ 0.356817904804523 9

> (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.
> (λ (r) (* (/ 4 3) pi r r r))
      #<procedure>; eine anonyme Funktion
> ((λ (r) (* (/ 4 3) pi r r r))
        4)
      --> 268.082573106329
```

Lambda-expressions

- Ein *Lambda-Ausdruck* hat die allgemeine Form

$$(\lambda \ (\langle parameters \rangle) \ \langle body \rangle)$$

- 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
```

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)        ⟶ 113.09733552923 26
> (volume 1)        ⟶ 4.188790204786391
```

Eine zweite Notation zur Definition von Funktionen

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

;;; äqivalente 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.

Variablenskopus und closures

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.

Argumente, formale und aktuelle Parameter

Definition

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 Abstraktionsebenen einführen.
- Gleiches E/A-Verhalten von Funktionen als Äquivalenzrelation führt zu Klassen von Funktionen gleichen Typs.
- Wir werden beim Thema „Funktionen höherer Ordnung“ hierauf zurückkommen.

Fallunterscheidungen

Häufig werden in mathematischen Funktionsdefinitionen Fallunterscheidungen gemacht, beispielsweise

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

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

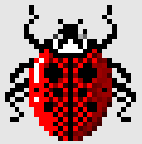
Bedingte Ausdrücke

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.*

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
>
```

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.

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

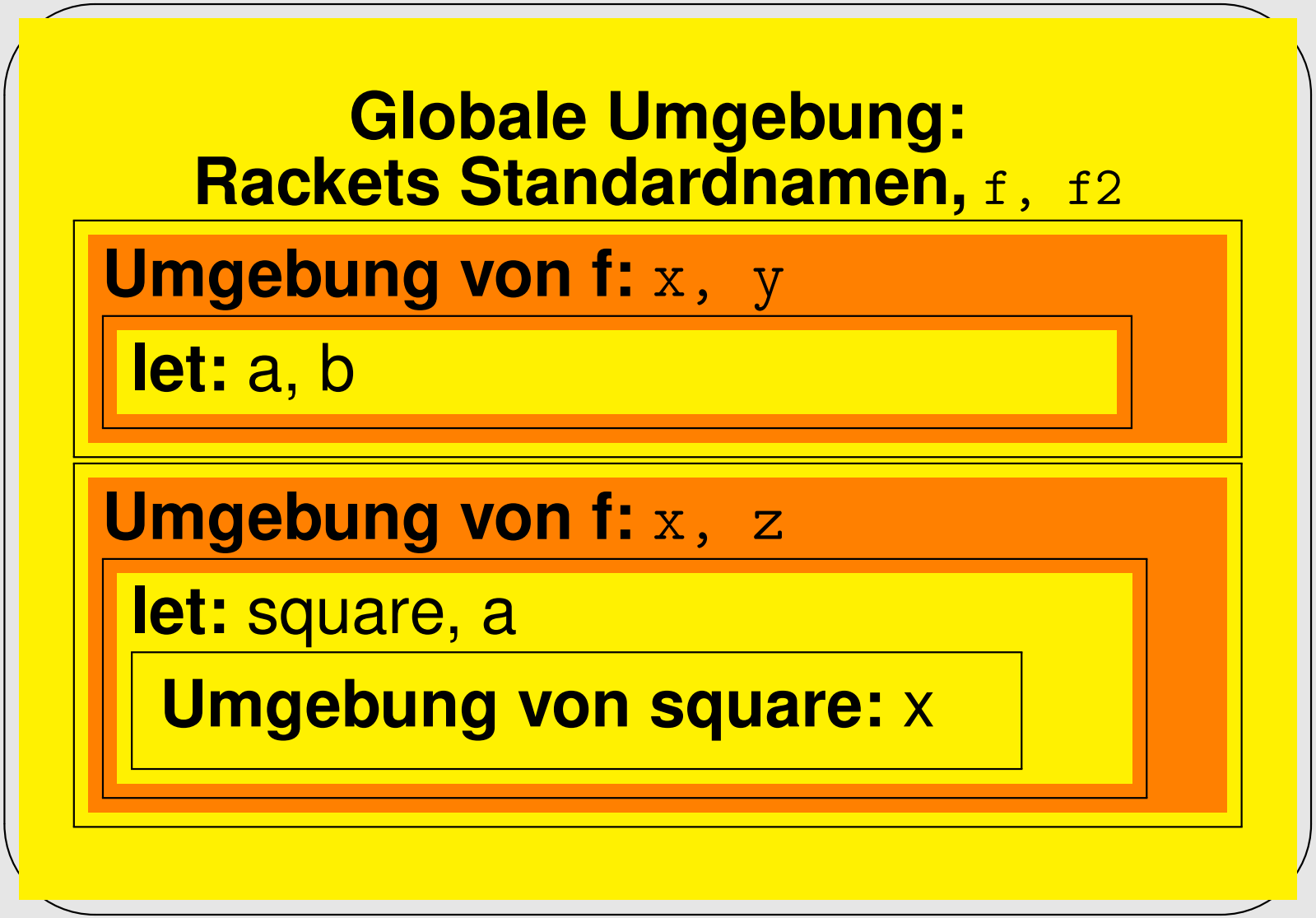
- **let** läßt sich durch **lambda** ersetzen.
- Die **let**-variante ist aber übersichtlicher.

```
> (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)



Definition (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 (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.

NUN EIN PAAR LÄNGST FÄLLIGE

METAPHYSISCHE*)

BETRACHTUNGEN ÜBER

SICHT-BARKEIT

UND

LEBENS-DAUER

VON VARIABLEN.

---

*) NACH DUDEN: PHILOSOPHISCHE LEHRE VON DEN LETZTEN GRÜNDEN & ZUSAMMENHÄNGEN DES SEINS VON VARIABLEN (WOVON DENN SONST?).

Sichtbarkeit einer Variablen

Definition (Sichtbarkeit, engl. scope)

- Die Sichtbarkeit legt den Programmbereich 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 Verschatten genannt.

Lebensdauer

Definition (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.

Symbole

Alice

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

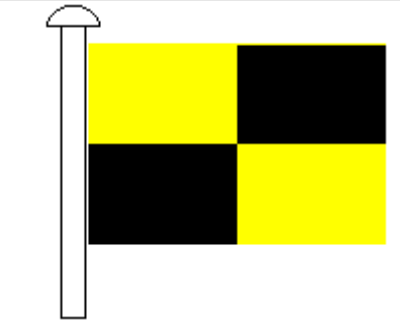
Definition (Symbol)

- 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!“.

Mehrdeutigkeit von Zeichen

- Umgekehrt können viele Symbole dieselbe Bedeutung haben — das Morsezeichen „L“ ($\cdot - \cdot\cdot$), als Schallsignal oder Folge von Lichtblitzen gegeben, bedeutet ebenfalls „Anhalten“.
- Alle diese Zeichen können für die Zahl „vier“ stehen:
  4, IV, ❹, { ✈ ✉ ✿ ✄ }, $3 + 1$

Zur Bedeutung von Zeichen

„I don’t know what you mean by ‘glory’, “ Alice said.
Humpty Dumpty smiled contemptously. „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.“

Symbole in Lisp und Racket

Definition (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 Typprädikat 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
>
```

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

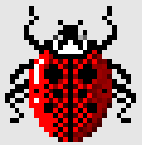
de re: Der Pfeiffer ist ein frecher Lümmel.

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

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

Beispiele

```
Language: Standard (R5RS).
> (define pi 3.14159265358979 3d0) ⟶ ⊥
> pi                ⟶   3.14159265358979 3
> 'pi               ⟶   pi
> ''''pi            ⟶   '''pi
> (+ 2 2)           ⟶   4
> '(+ 2 2)          ⟶   (+ 2 2)
> 'Auto             ⟶   auto
> Auto
```



```
reference to undefined identifier: auto
> (eval 'pi)   ⟶   3.14159265358979 3
> (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
```

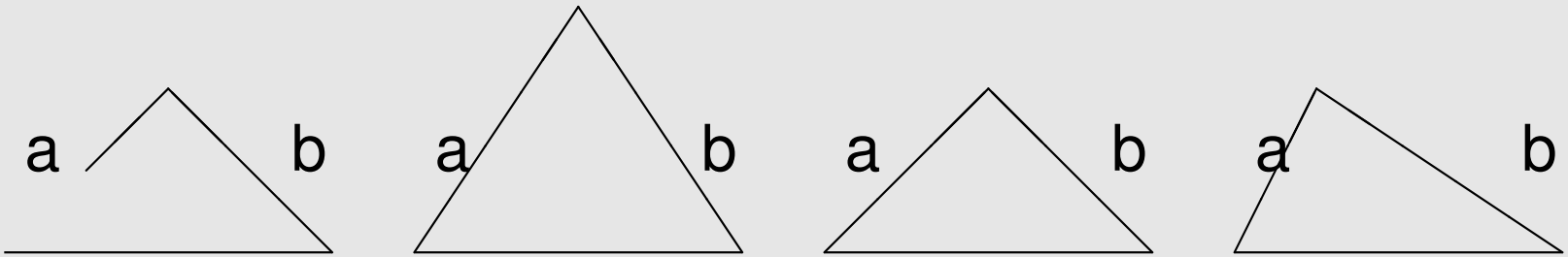
"... The name of the song is called *'Haddocks' Eyes.'* " *"Oh, that's the name of the song, isn't it?"* Alice said, trying to feel interested.
"No, you don't understand," the knight said, looking a little vexed. "That's what the name is *called.* The name really is *'The Aged Aged Man.'* "
"Then I ought to have said 'That's what the *song* is called'?" Alice corrected herself. "No, you oughtn't: that's quite another thing! The *song* ist 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!

Modellierung mittels Racket-Symbolen

Beispiel (The Name of the Song)

```
> (define Ways_and_Means
          'Asitting_on_a_gate)
          ⟶ ⊥
> (define The_Aged_Aged_Man 'Ways_and_Means)
          ⟶ ⊥
> (define Haddocks_Eyes 'The_Aged_Aged_Man)
          ⟶ ⊥
> Haddocks_Eyes
          ⟶ the_aged_aged_man
> (eval Haddocks_Eyes)
          ⟶ ways_and_means
> (eval (eval Haddocks_Eyes))
          ⟶ asitting_on_a_gate
```

Beispiel: Dreiecke



1. kein
2. gleichseitiges
3. gleichschenkliges
4. schiefwinkeliges

Dreieck

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

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

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 "->", z.B. char->integer, list->string.

Teil III

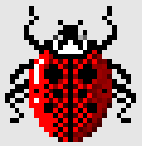
Datenabstraktion in Racket

Datenabstraktion

Typ der Argumente

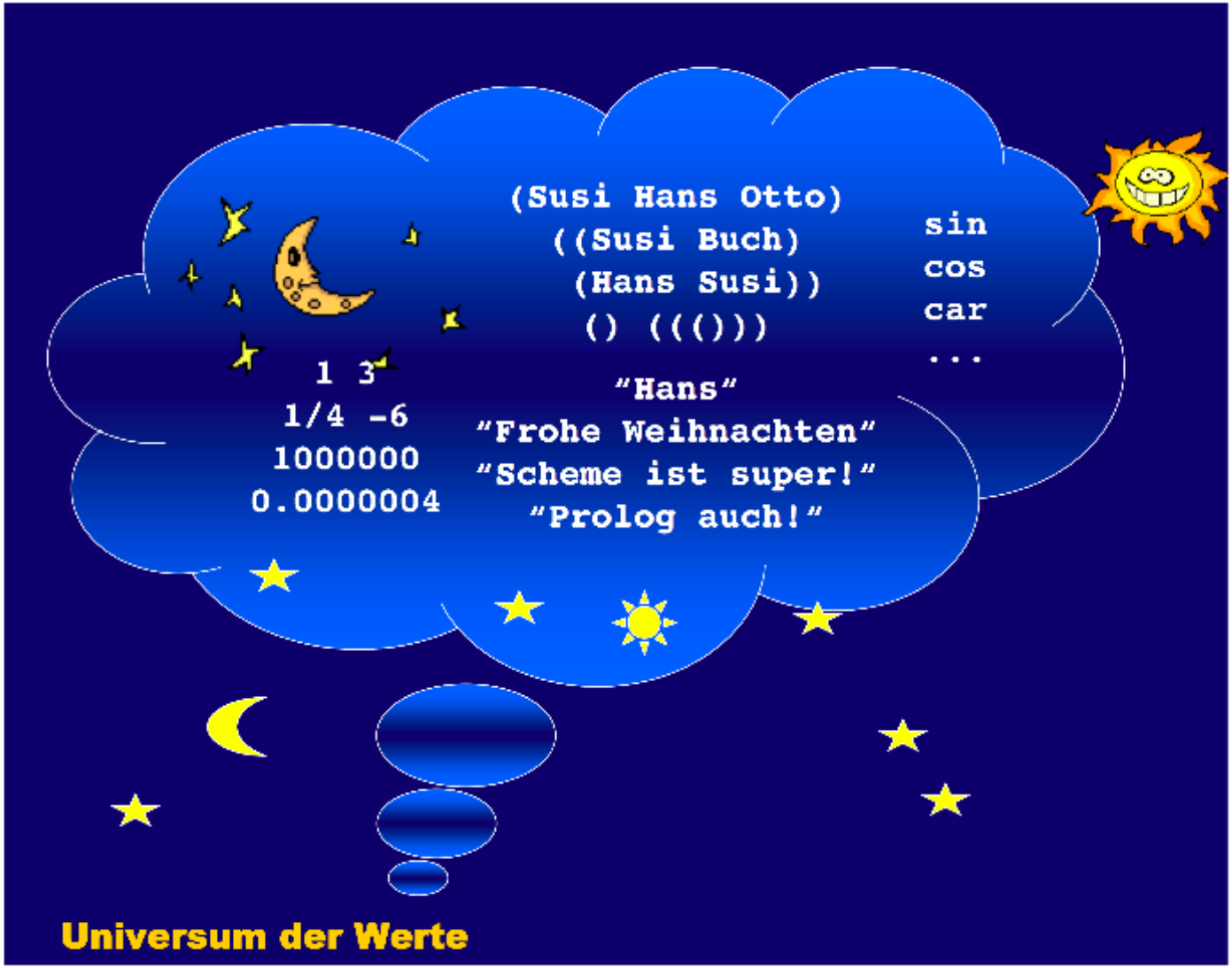

Nicht jede Funktion ist auf jeden Wert anwendbar;
Beispiel:

```
 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)
```

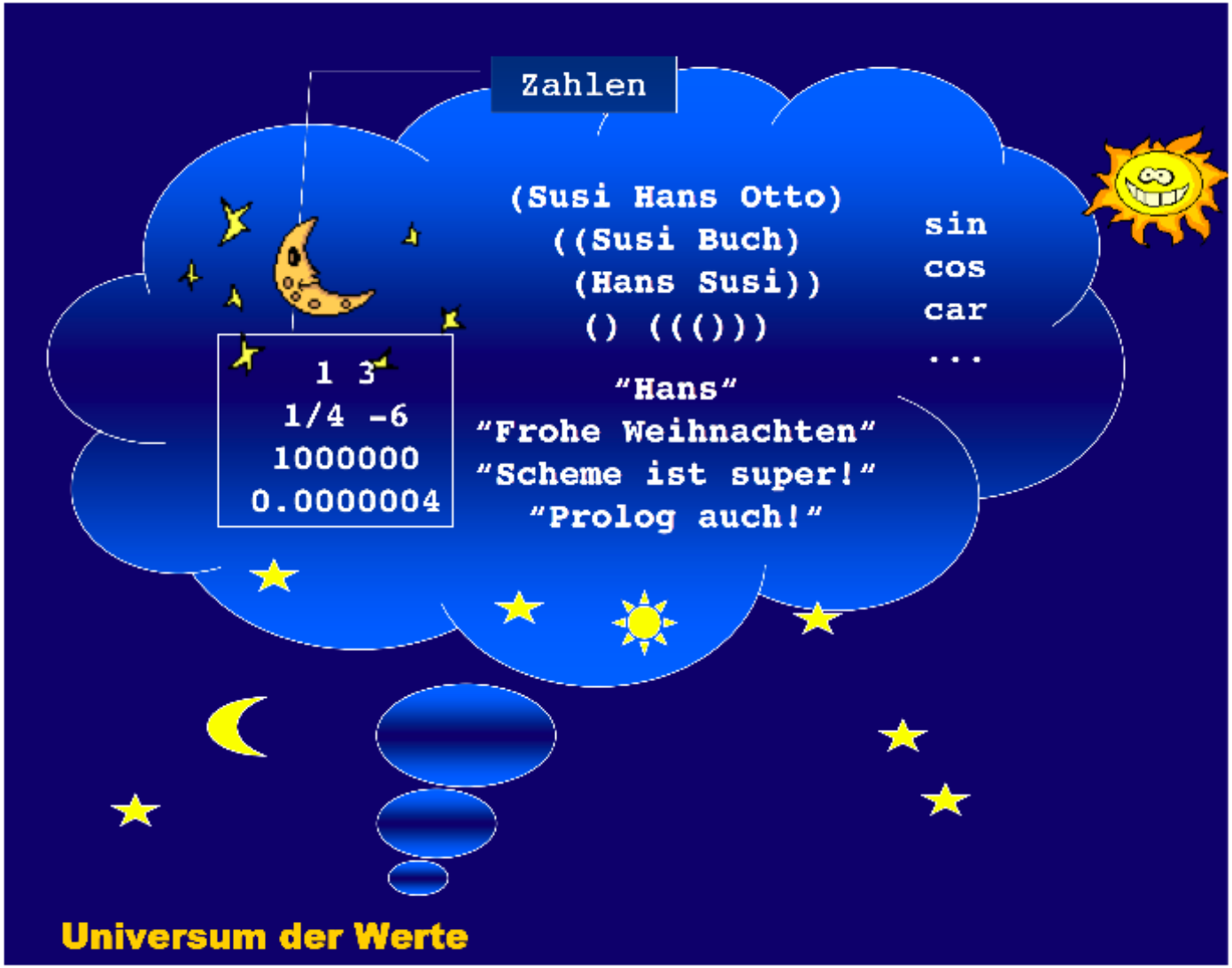


```
+: expects type <number> as 2nd argument,
 given: (1 ,2 ,apfel); other arguments were: 1
>
```

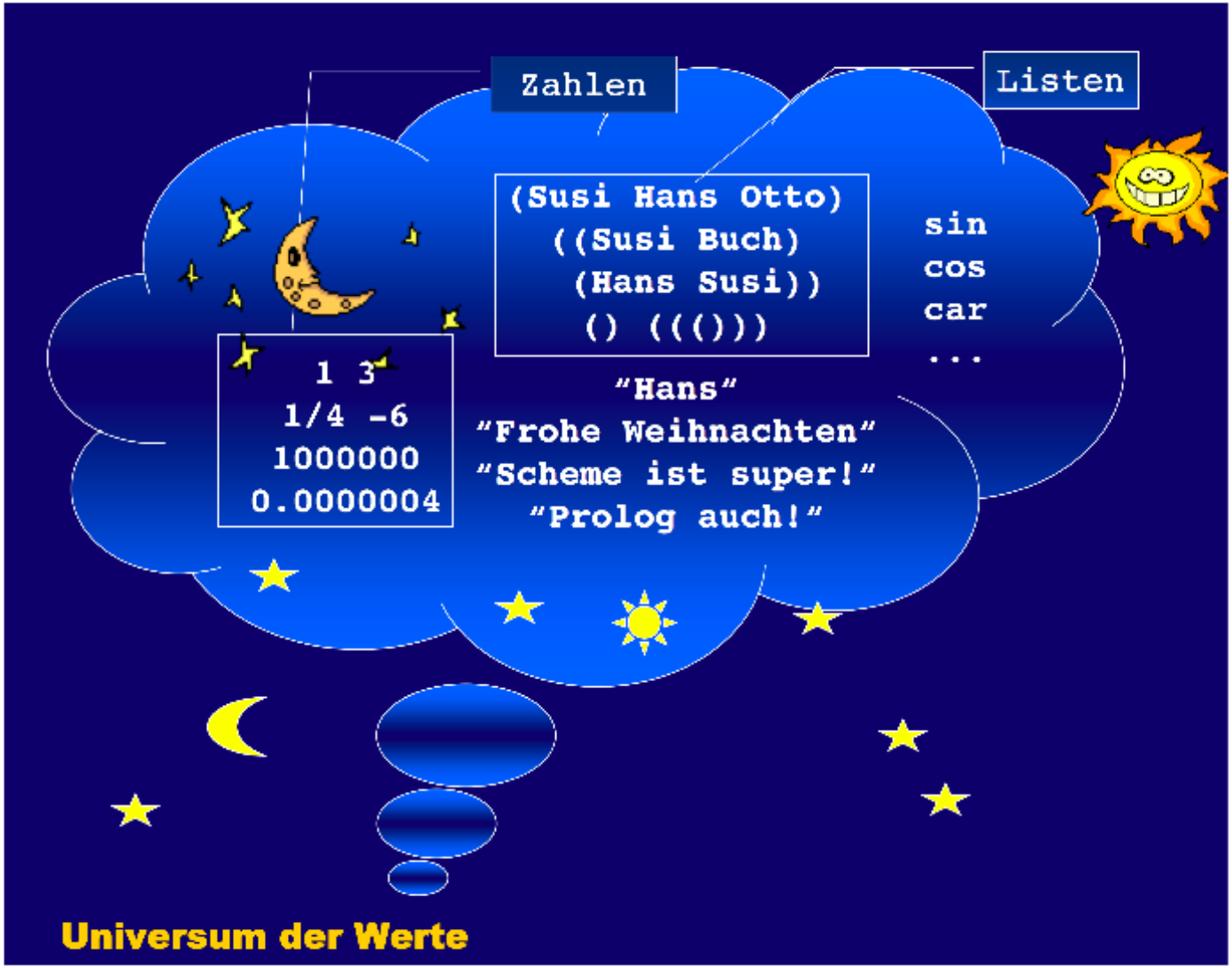
Das Universum der Werte



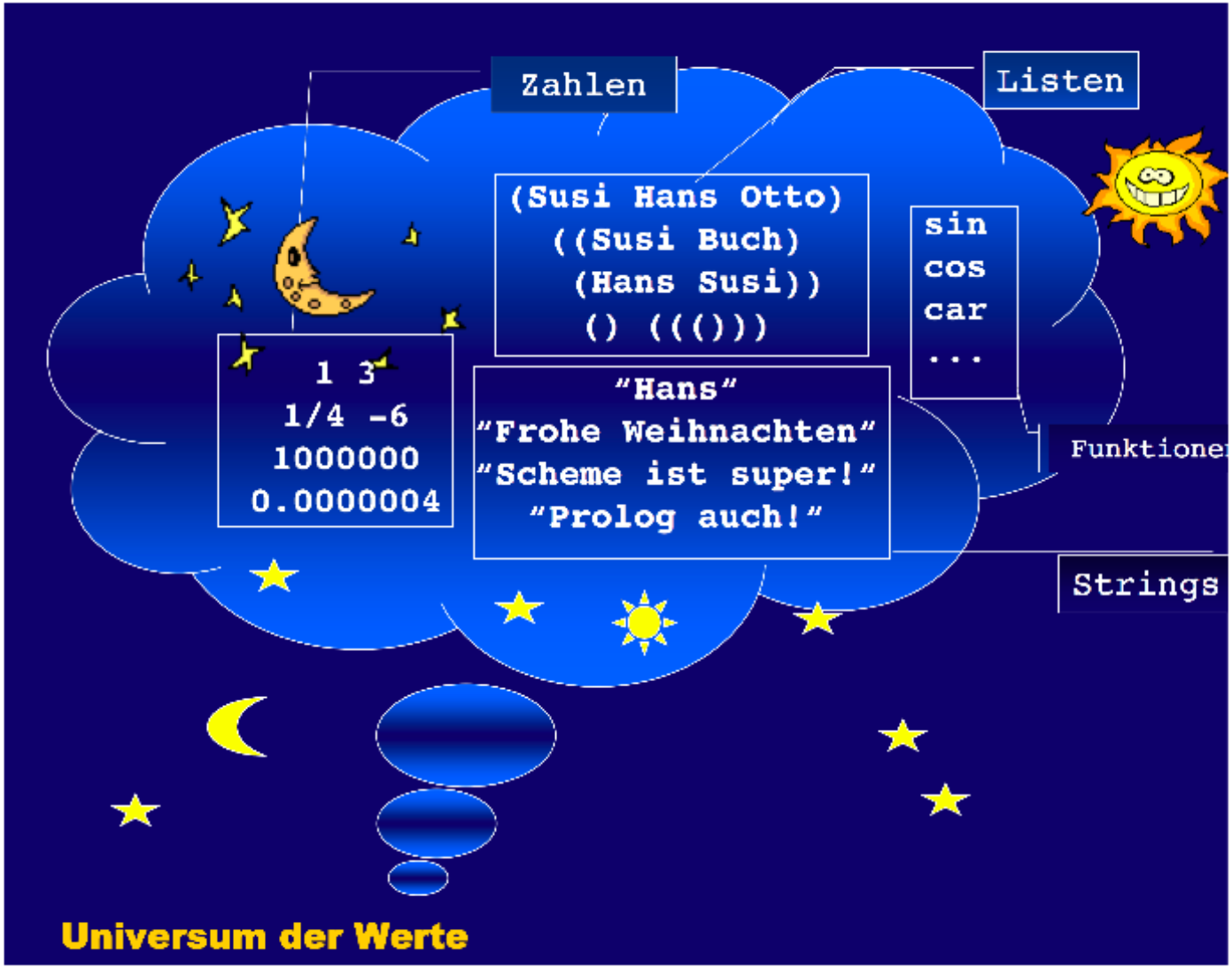
Das Universum der Werte



Das Universum der Werte



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 (Datentyp)

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. [**?**, 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 strukturgleichen 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 (Kardinalität)

- Die Mächtigkeit der Wertemenge eines Datentyps T heißt Kardinalität von T abgekürzt: $\mathrm{card}(T)$.
- 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

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

- Um die Werte eines Typs T mit $\mathrm{card}(T) = 2$ 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* $\rightarrow$ *number* oder
  odd:: *number* $\rightarrow$ *boolean*.

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

Definition (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.

Die Basistypen von Racket sind disjunkt.

Satz (Disjunktheit der Typen)

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

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

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.

$$\begin{aligned} \text{odd} &:: \textit{number} \rightarrow \textit{boolean} \\ \text{id} &:: \alpha \rightarrow \alpha \\ \text{equal?} &:: \alpha, \beta \rightarrow \textit{boolean} \end{aligned}$$

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.

Datenstrukturen

Zahlen (number)

- In Racket gibt es eine Hierarchie von Typen für Zahlen: *number, complex, real, rational, integer*.
- Für jeden Zahlentyp gibt es ein Typprädikat 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)`	$\longrightarrow$	**#t**
`(complex? 3)`	$\longrightarrow$	**#t**
`(real? 3)`	$\longrightarrow$	**#t**
`(real? -2.5+0.0i)`	$\longrightarrow$	**#t** ; *Imaginärteil=0*
`(real? #e1e10)`	$\longrightarrow$	**#t** ; *exakte Zahl* 10^{10}
`(rational? 6/10)`	$\longrightarrow$	**#t**
`(rational? 6/3)`	$\longrightarrow$	**#t**
`(integer? 3+0i)`	$\longrightarrow$	**#t**
`(integer? 3.0)`	$\longrightarrow$	**#t**
`(integer? 8/4)`	$\longrightarrow$	**#t**

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 arithmetrischen Operationen keine Rundungsfehler und Wertebereichsüberläufe auftreten.

Inexakte Repräsentation: Präfix #i

Speicherökonomische Repräsention, bei der Rundungsfehler auftreten können.

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

Beispiel: Fakultät

```
(define (fak n) ;
    (if (= n 1)
        1
        (* n (fak (- n 1) ))))
```

$\prod_{i=1}^{n} i = n \times \prod_{i=1}^{n-1} i, n \in \mathcal{N}$

```
> (fak 200)
```

$\longrightarrow$ 788657867364790503552363213932185062295135977687173263294742533244359449963403342920304284011984623904177212138919638830257642790242637105061926624952829931113462857270763317237396988943922445621451664240254033291864131227428294853277524242407573903240321257405579566860226031904170324062351700858796178922222789623703897374720000000000000000000000000000000000000000000000

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

Zur Genauigkeit

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

> zehnHoch100 ⟶
10000000000000000000000000000000000000
00000000000000000000000000000000000000
00000000000000000000000000000000000000
00000000000000000000000000000000000000

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

Auslöschung

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

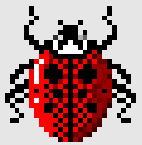
Spezielle Zahlen: ∞

```
Sprache: Module; memory limit: 128 megabytes.
> (/ 1.0 0.0)      ⟶  +inf.0; +∞
> (/ -1.0 0.0)     ⟶  -inf.0; -∞
> (/ 0.0 0.0)      ⟶  +nan.0; not a number
> (/ 1 0)          ⟶  /: 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
>
```

Die Vergleichsoperatoren

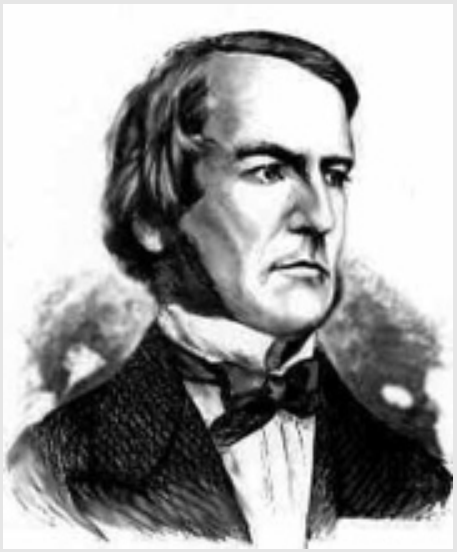
- Die Vergleichsprädikate $>, >=, <=, <$ definieren (partielle) Ordnungsrelationen und eine Äquivalenzrelation $=$ über den Wertemengen der *Zahlen*.
- Auf nicht-numerische Datentypen sind diese Funktionen nicht anwendbar!
  Für andere Datentypen gibt es spezielle Ordnungsprädikate und Vergleichsfunktionen.

```
> (< 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) $\longrightarrow$ fahren

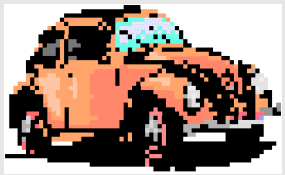
Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

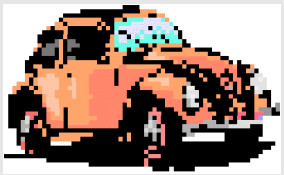
Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

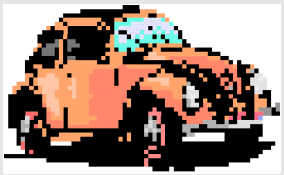
Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 
- > (**if** (**not** 'auto) 'fahren 'laufen) $\longrightarrow$ laufen

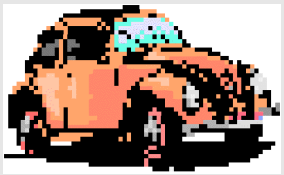
Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

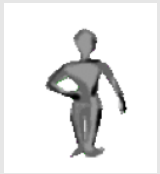
> (**if** (**not** 'auto) 'fahren 'laufen) $\longrightarrow$ laufen

- 

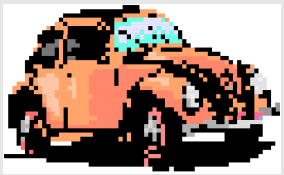
Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

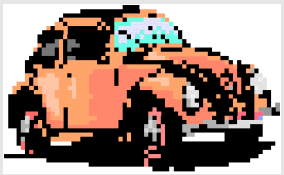
> (**if** (**not** 'auto) 'fahren 'laufen) $\longrightarrow$ laufen

- 

Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 
- > (**if** (**not** 'auto) 'fahren 'laufen) $\longrightarrow$ laufen
- 

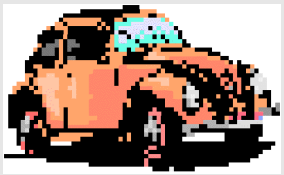
Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

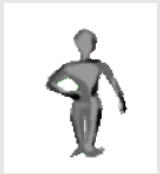
> (**if** (**not** 'auto) 'fahren 'laufen) $\longrightarrow$ laufen

- 

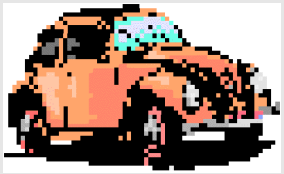
Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

> (**if** (**not** 'auto) 'fahren 'laufen) $\longrightarrow$ laufen

- 

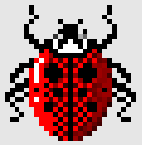
Beispiel

- > (**if** 'auto 'fahren 'laufen) $\longrightarrow$ fahren
- 

> (**if** (**not** 'auto) 'fahren 'laufen) $\longrightarrow$ laufen

- 

```
> (= 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
```

$$
\begin{aligned}
= &:: \mathit{number}, \mathit{number} \rightarrow \mathit{boolean} \\
\text{equal?} &:: \alpha, \beta \rightarrow \mathit{boolean}
\end{aligned}
$$

- 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.*

Verknüpfungstabellen

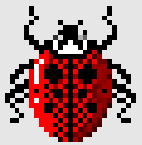
p	q	(and p q)	(or p q)	(not p)
#f	#f	#f	#f	#t
#f	#t	#f	#t	#t
#f	⊥	#f	⊥	#t
#t	#f	#f	#t	#f
#t	#t	#t	#t	#f
#t	⊥	⊥	#t	#f
⊥	*	⊥	⊥	⊥

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 ... an) ist wahr, sobald ein Argument zu **#t** evaluiert, **#f** sonst.
  - Die Konjunktion (**and** a1 ... an) ist falsch, sobald ein Argument zu **#f** evaluiert, **#t** sonst.
  - Die Negation negiert den Wahrheitswert.

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 (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
```

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-Zeichensatzes[2].
- 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.

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

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. Diesen können wir nutzen,

- um eine Ordnung auf den Zeichen zu definieren,

$$(\texttt{char<?}\ \ \#\backslash A\ \ \#\backslash Z) \longrightarrow \textbf{\#t}$$

- 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:

$$(\texttt{char>?}\ \ \#\backslash A\ \ \#\backslash 1) \longrightarrow \textbf{\#t}$$

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

$$\#\backslash 077 \longrightarrow \#\backslash ?$$

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␣\u20AC" ⟶ "€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
```

Teil III

Datenabstraktion in Racket

Listen

Definition (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

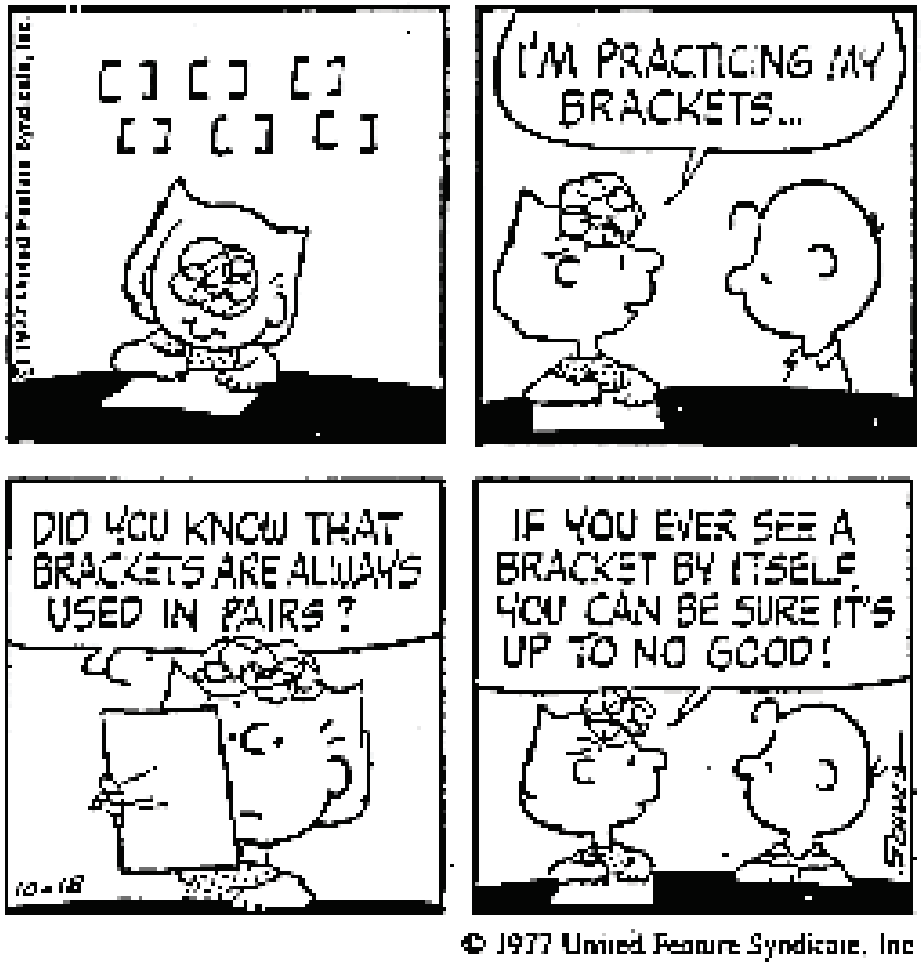
- Die leere Liste '() ist polymorph, denn sie ist leer im Hinblick auf jeden speziellen Typ.
- Die Liste '(()) dagegen ist nicht leer, da sie das Element () enthält; sie ist also eine Liste der Länge 1.
- Wenn Sie Listen direkt durch Aufzählung der Werte notieren, ist das quote-Zeichen wichtig, sonst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

```
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)                    ⟶ 0.141120008059867 2
    ; 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 (Neutrales Element)

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

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

Assoziativität

Satz (Assoziativität von append)

Die Konkatenation ist assoziativ:

```
   (append (append xs ys) zs)
 = (append  xs (append ys zs))
```

```
> (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 m1 und m2 als Mengen M_1 und M_2 interpretieren, dann entspricht die **append**-Operation der Vereinigung zweier Mengen:

$$(\textbf{append}\ \text{m1}\ \text{m2}) \quad \equiv \quad M_1 \cup M_2$$

Mengentheoretische Interpretation

- Wenn wir zwei Listen m1 und m2 als Mengen M_1 und M_2 interpretieren, dann entspricht die **append**-Operation der Vereinigung zweier Mengen:

$$(\textbf{append}\ \text{m1}\ \text{m2}) \quad \equiv \quad M_1 \cup M_2$$

- 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

Mengentheoretische Interpretation

- Wenn wir zwei Listen m1 und m2 als Mengen M_1 und M_2 interpretieren, dann entspricht die **append**-Operation der Vereinigung zweier Mengen:

$$(\textbf{append}\ \text{m1}\ \text{m2}) \quad \equiv \quad M_1 \cup M_2$$

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

$$\text{Länge einer Liste} \equiv \text{Mächtigkeit der Menge}$$

- 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 '())                          ⟶ (1)
> (cons 1
        (cons 1 '()))                   ⟶ (1 1)
> (cons 1
        (cons 1
              (cons 1
                    '())))              ⟶ (1 1 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 '())) ⟶ #t
> (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

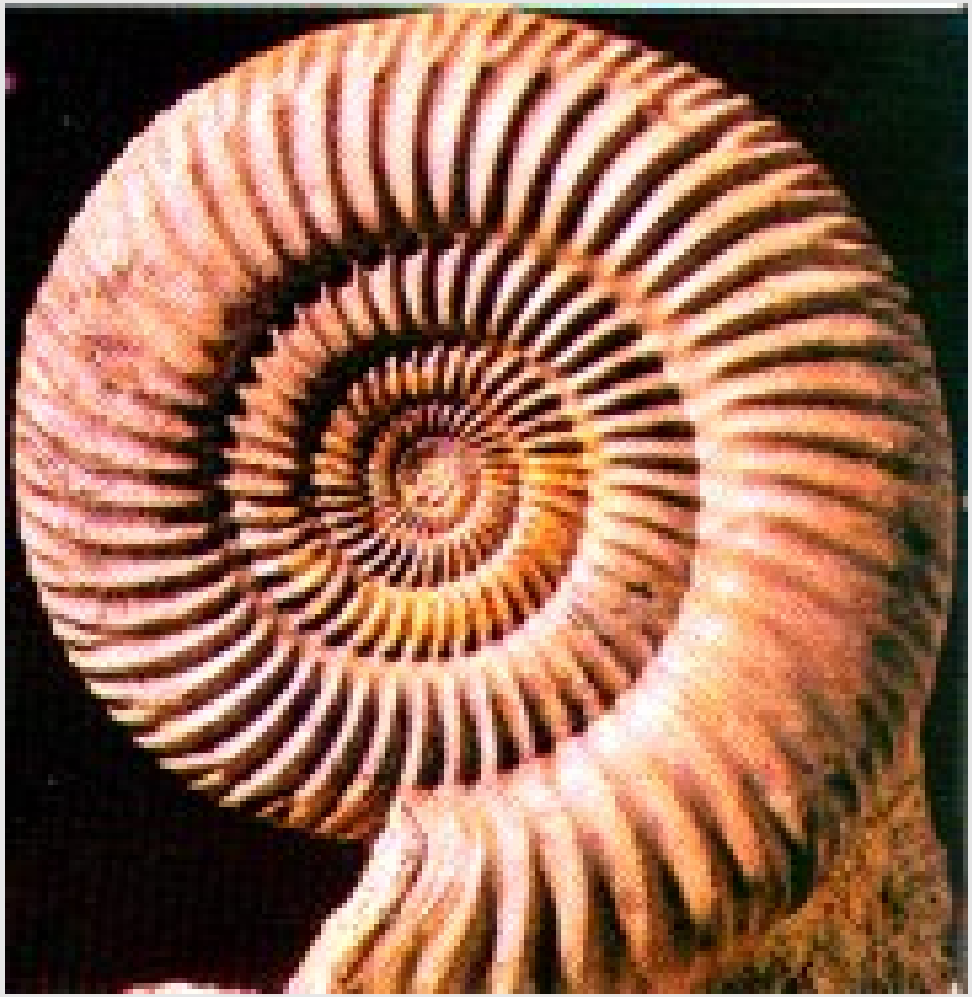
Die Akzessoren car und cdr geben Zugriff auf die Bestandteile einer Liste:

- 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:

```
(car (cons x xs))   =  x
(cdr (cons x xs))   = xs
(car '()) = (cdr '()) = ⊥
xs     = (cons (car xs) (cdr xs))
```

Akzessoren

Kurzform	*steht für*
`(caar xs)`	`(car(car xs))`
`(caaar xs)`	`(car(car(car xs)))`
`(caaaar xs)`	`(car(car(car(car xs))))`
`(cddr xs)`	`(cdr(cdr xs))`
`(cdddr xs)`	`(cdr(cdr(cdr xs)))`
`(cddddr xs)`	`(cdr(cdr(cdr(cdr xs))))`
`(cadr xs)`	`(car(cdr xs))`
`(cdadr xs)`	`(cdr(car(cdr xs)`



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

ammonit

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:

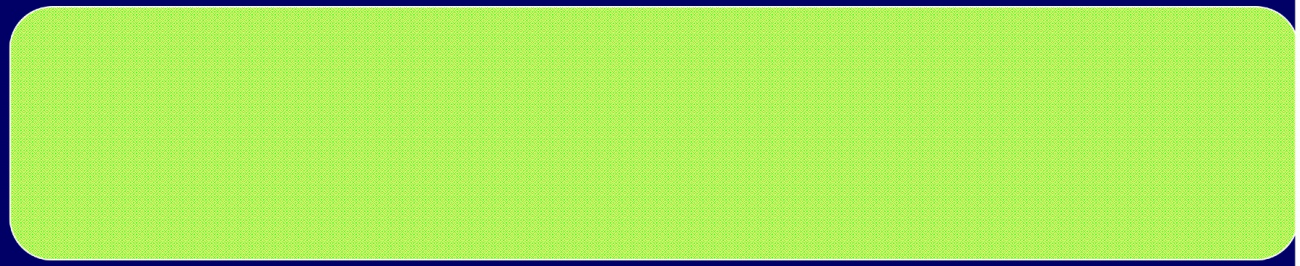
Definition (Liste)

Eine Liste ist

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

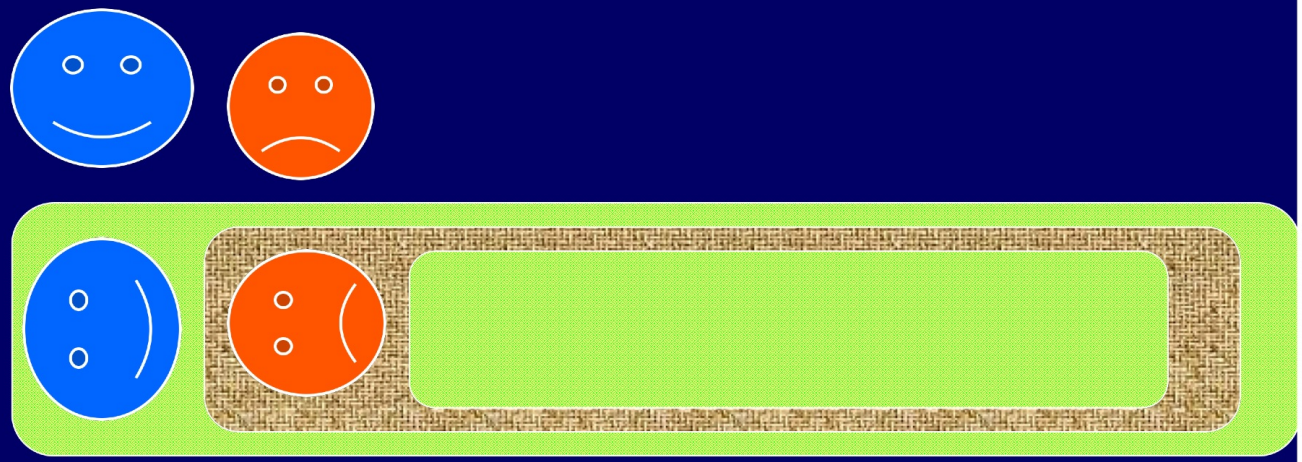
Die rekursive Struktur der Liste spiegelt sich häufig in rekursiven Listen-Funktionen wider.

Rekursive Konstruktion und Selektion

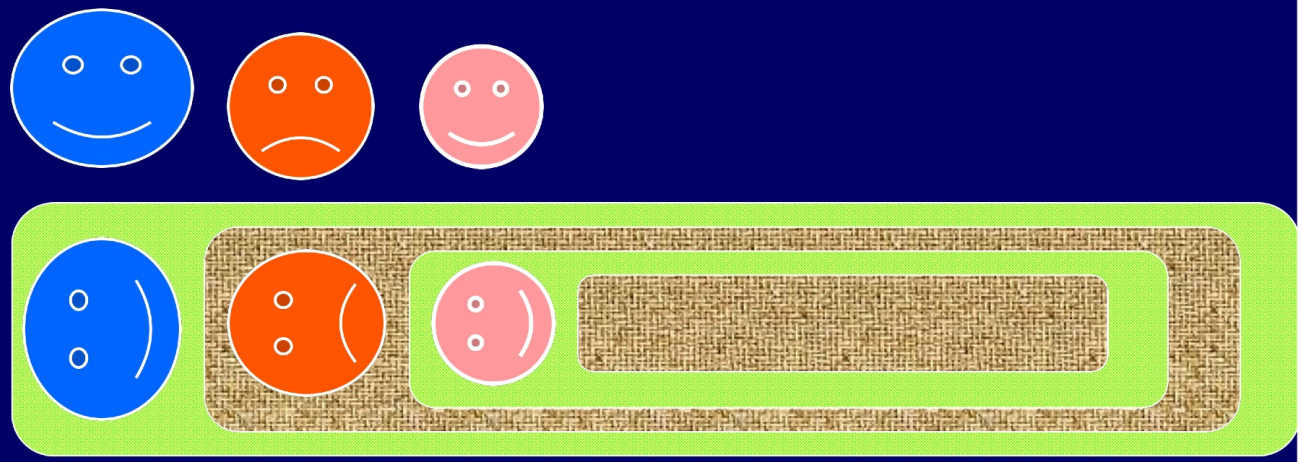


Rekursive Konstruktion und Selektion

Rekursive Konstruktion und Selektion



Rekursive Konstruktion und Selektion



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))
```

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
```

Beispiel (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
```

```
> (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!

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
```

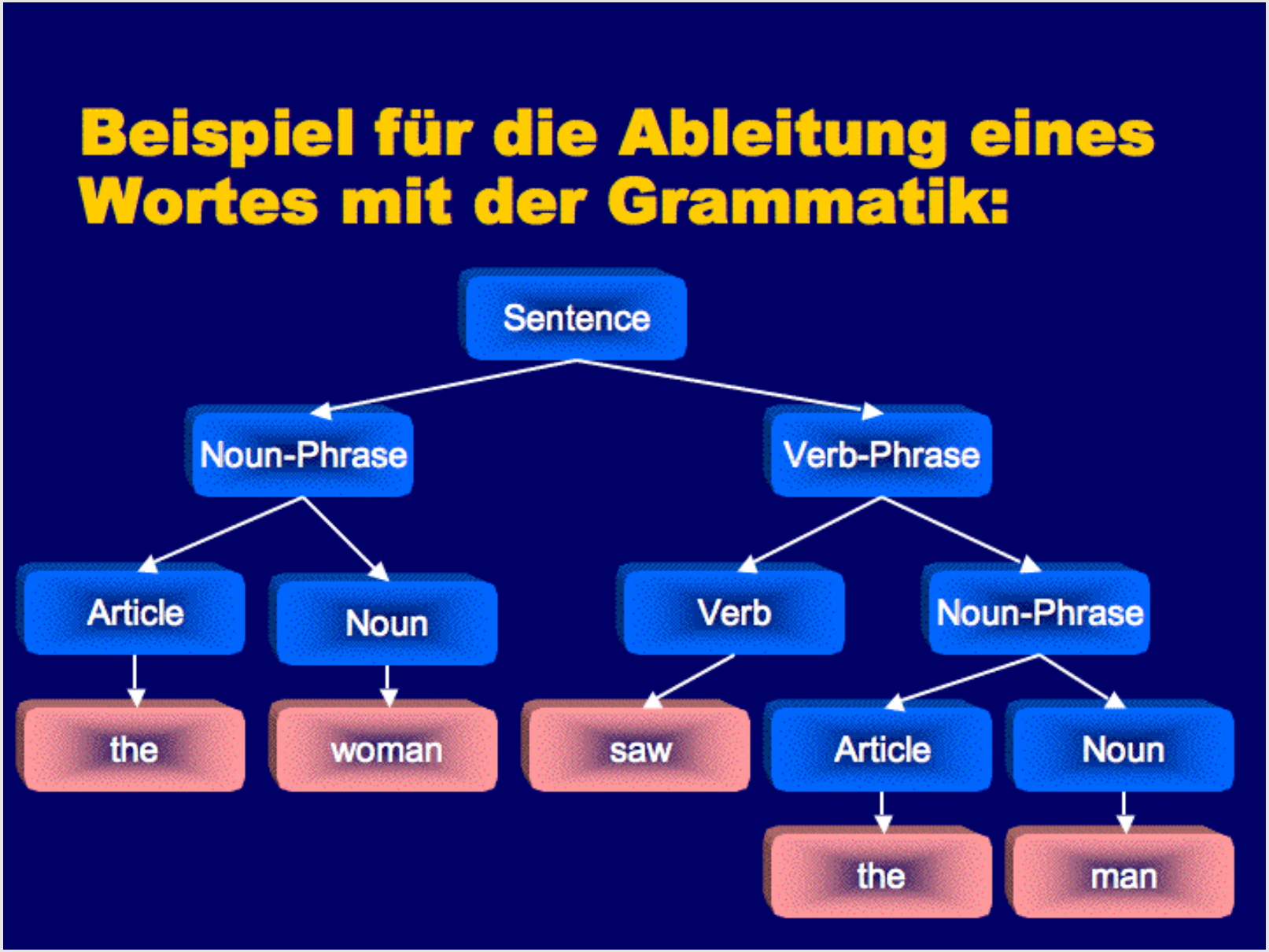
Sprachgenerierung

Beispiel (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 ...
```

▸ als Racket-Funktionen

Der Ableitungsbaum



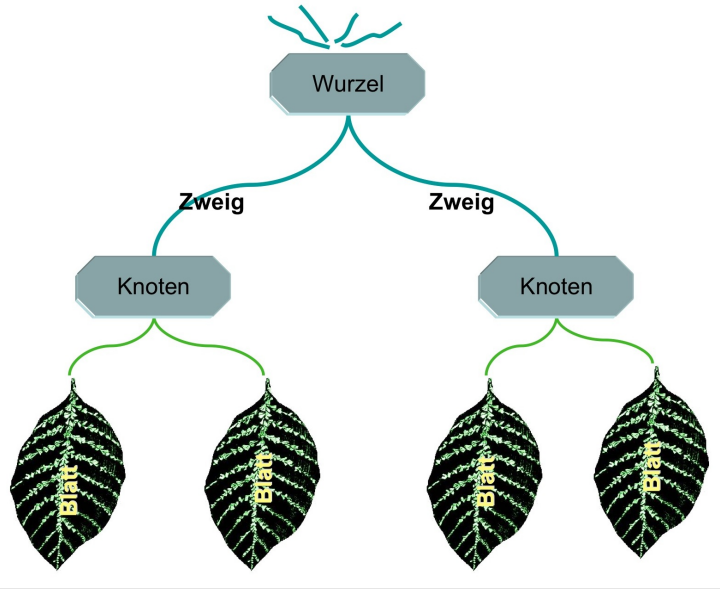
Definition (Ableitungsbaum)

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.

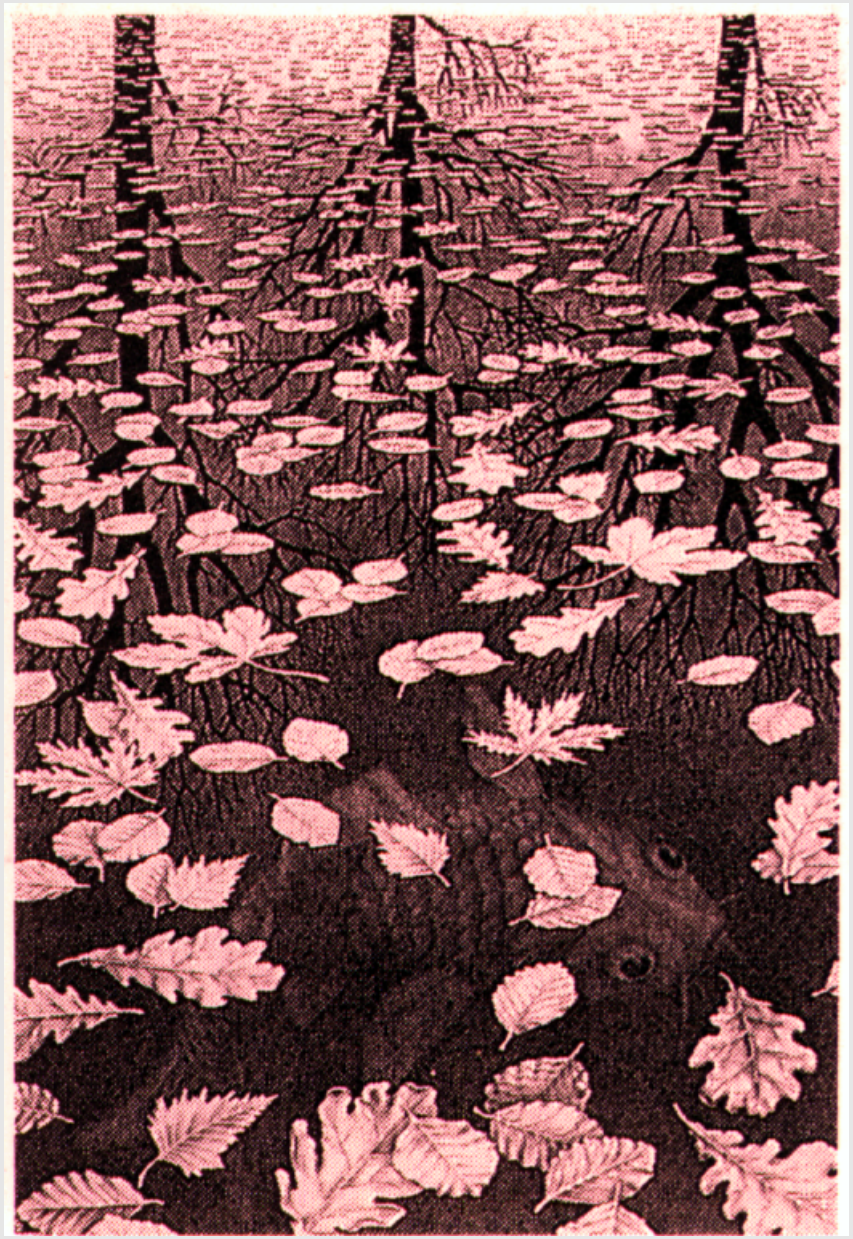
Ableitungsbäume

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 (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.

▸ Grammatik 1

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. ▸ Grammatik 1

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))))
```

- (random *n*) liefert eine zufällige Zahl x, mit $0 \leq x < n$
- (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)
```

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)
```

Erweiterung: Adjektive und Präpositionen

Beispiel (Zusätzliche Regeln: Adjektive und Präpositionen)

$$
\begin{array}{lll}
\langle Sentence \rangle & ::= & \langle Noun\text{-}Phrase \rangle\ \langle Verb\text{-}Phrase \rangle \\
\langle Noun\text{-}Phrase \rangle & ::= & \langle Article \rangle\ \langle Adj^{*} \rangle\ \langle Noun \rangle\ \langle PP^{*} \rangle \\
\langle Adj^{*} \rangle & ::= & \emptyset \mid \langle Adj \rangle\ \langle Adj^{*} \rangle \\
\langle PP^{*} \rangle & ::= & \emptyset \mid \langle PP \rangle\ \langle PP^{*} \rangle \\
\langle PP \rangle & ::= & \langle Prep \rangle\ \langle Noun\text{-}Phrase \rangle \\
\langle Adj \rangle & ::= & big \mid little \mid blue \mid green \mid \\
\langle Prep \rangle & ::= & to \mid in \mid by \mid with \mid \cdots \\
\langle Verb\text{-}Phrase \rangle & ::= & \langle Verb \rangle\ \langle Noun\text{-}Phrase \rangle \\
\langle Article \rangle & ::= & \text{the} \mid \text{a} \mid \cdots \\
\langle Noun \rangle & ::= & \text{man} \mid \text{ball} \mid \text{woman} \mid \text{table} \cdots \\
\langle Verb \rangle & ::= & \text{hit} \mid \text{took} \mid \text{saw} \mid \text{liked} \cdots
\end{array}
$$

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) $\longrightarrow$ (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)

audio

> (sentence) $\longrightarrow$ (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)

audio

Teil IV

Repräsentation der Datenstrukturen

Funktionale Datenabstraktion

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

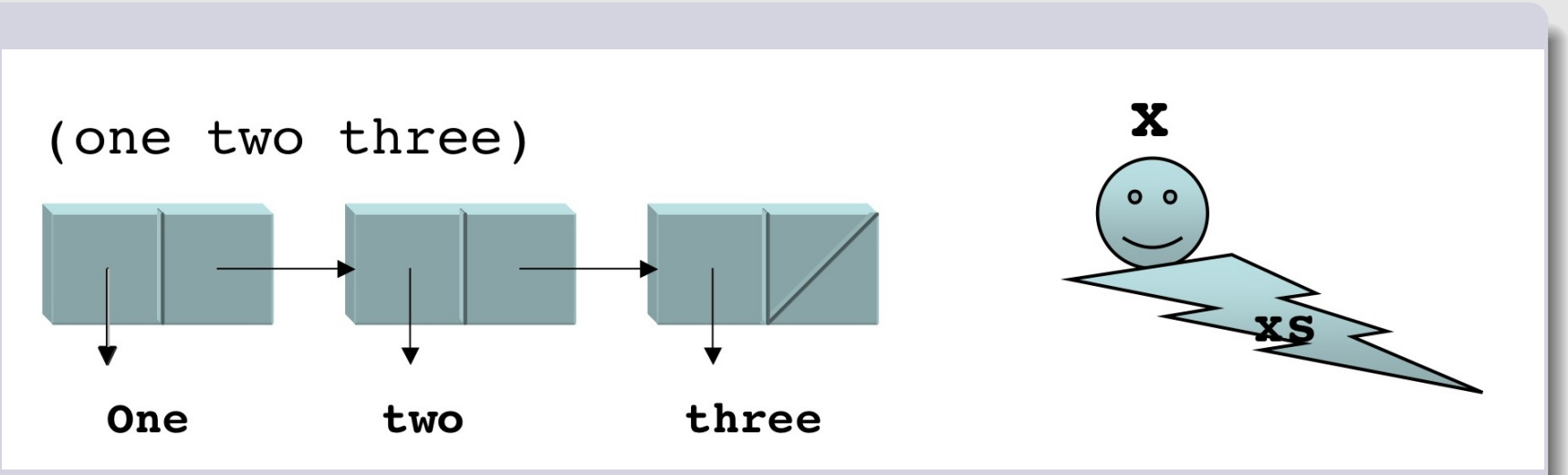
Beispiel (Closure als Liste)

```
> (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
```

Grundlagen der funktionalen Programmierung

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.



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 (Frage:)

- *Sind x und y identisch (eq? x y)?*

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 (Frage:)

- *Sind x und y identisch (eq? x y)?*
- *Sind x und z identisch (eq? x z)?*

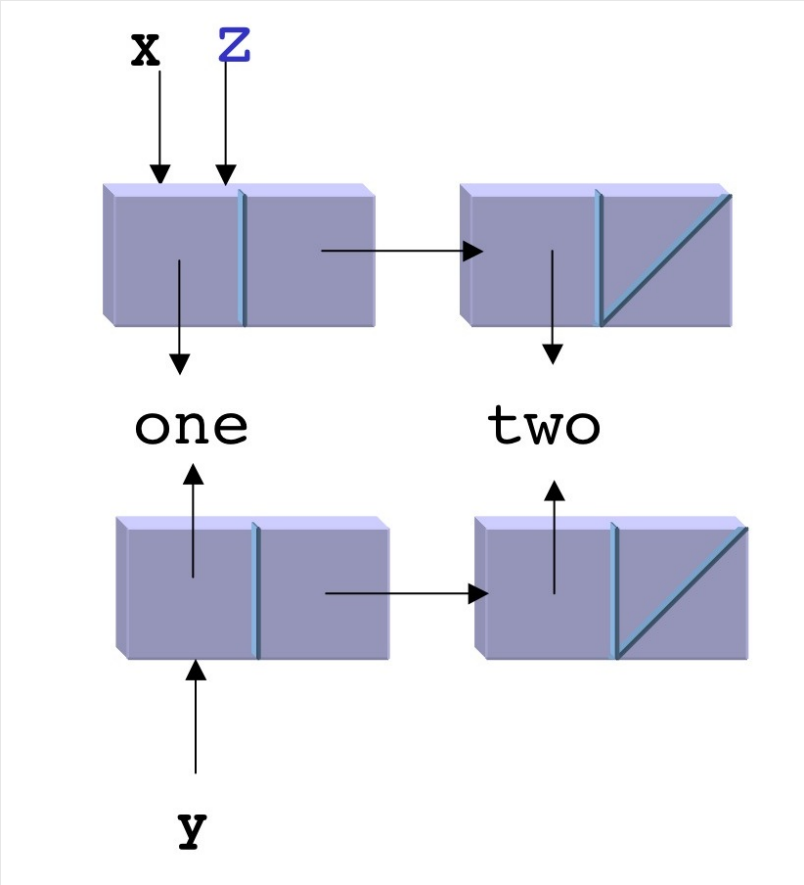
Beispiel:
Listen mit gemeinsamen Elementen

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

Problem (Frage:)

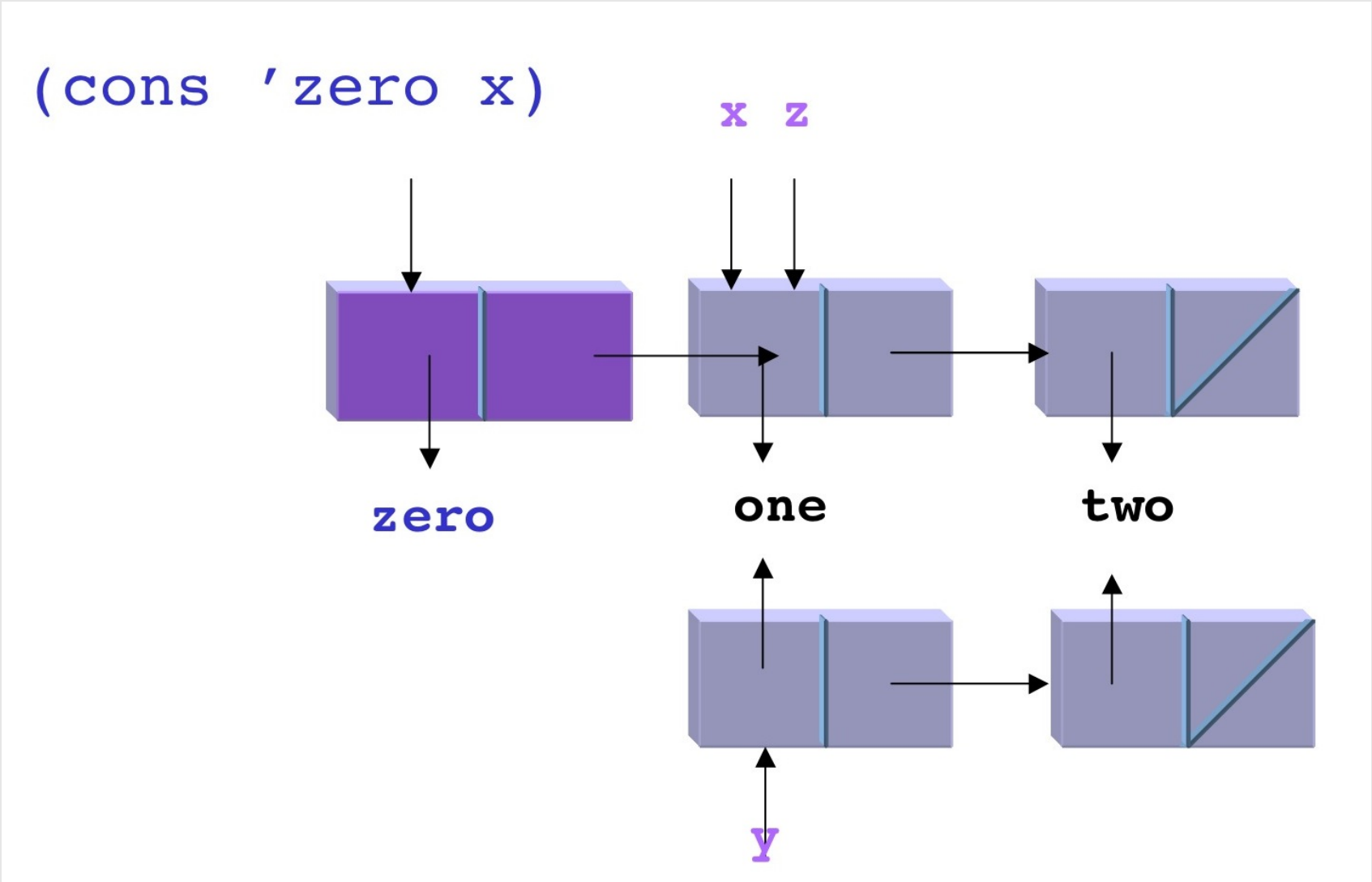
- *Sind x und y identisch (eq? x y)?*
- *Sind x und z identisch (eq? x z)?*
- *Sind x und y gleich (equal? x y)?*

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



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.

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.

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-Famile 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
```

Rekursion über Listen: Länge einer Liste

- Die leere Liste hat die Länge 0.
- Die Länge einer nicht-leeren Liste xs ist:
  1 + Länge des Schwanzes,
  also 1 + (my−length (cdr 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
```

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 (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 (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 $>$ 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))]))
```

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
```

Teil IV

Repräsentation der Datenstrukturen

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 Gänsefüßchen 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.."
```

Textformatierung

Beispiel (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
         (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 (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 $1 \cdots 19$ werden unsystematisch gebildet.
- Die Zahlen $20 \cdots 99$ 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 sechstelliger Zahlen:

`digit`, `convert2`, `convert3`

Kombinatoren: Zusammenfassen von Teilgruppen zu einem String:

`combine2`, `combine3`, `combine6`, `link`

Füllworte: and

```
;;; 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 hundr
                        (convert3 hundreds))])))
> (number->cleartext 1111)

  ⟶ "one␣thousand␣one␣hundred␣and␣eleven"

> (number->cleartext 42042)
  ⟶ "forty-two␣thousand␣and␣forty-two"
```

```
(define (combine6 thousands hundreds)
  (cond [(= 0 thousands)(convert3 hundreds)]
        [(= 0 hundreds)
         (string-append (convert3 thousands)
                        "␣thousand")]
        [else
         (string-append (convert3 thousands)
                        "␣thousand" (link hundr
                        (convert3 hundreds))])))
> (number->cleartext 1111)

  ⟶ "one␣thousand␣one␣hundred␣and␣eleven"

> (number->cleartext 42042)
  ⟶ "forty-two␣thousand␣and␣forty-two"
1111
```

```
(define (combine6 thousands hundreds)
  (cond [(= 0 thousands)(convert3 hundreds)]
        [(= 0 hundreds)
         (string-append (convert3 thousands)
                        "␣thousand")]
        [else
         (string-append (convert3 thousands)
                        "␣thousand" (link hundr
                        (convert3 hundreds))])))
> (number->cleartext 1111)

  ⟶ "one␣thousand␣one␣hundred␣and␣eleven"

> (number->cleartext 42042)
  ⟶ "forty-two␣thousand␣and␣forty-two"
42042
```

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
>
```

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
>
```

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
```

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 Typprädikat: <typname>?,
2. einen Konstruktor: make−<typname>,
3. Selektoren für die Felder: <typname>−<feldname>,
4. Modifikatoren für die Felder: set−<typname>−<feldname>!

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)
```

Teil V

Semantik und Korrektheit

Semantik und Korrektheit

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 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“: Schul*heft* oder *Messer*heft.

  Strukturelle Mehrdeutigkeit liegt vor, wenn die grammatikalische Struktur eines Satzes nicht eindeutig ist.
- unpräzise,
- nicht stabil in der Bedeutung,
- aufwendig zu analysieren.

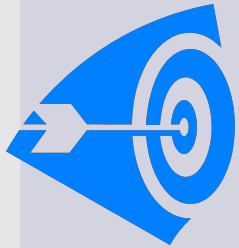
„Time flies like an arrow.“

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 Nomimalphrase (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?

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?

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?

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?

Ein syntaktisch korrekter, aber sinnloser Text

Jabberwocky

'Twas brilling, 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

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.

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 (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
- $f : M \to \Phi$ 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 **??** 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.

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 (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 Programierstile 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.

Wertesemantik und Reduktionsmodelle

13 Semantik
- Programmiersprachen: Grundbegriffe
- Wertesemantik und Reduktionsmodelle
- Die operationale Semantik von Racket
  - Die Auswertung funktionaler Ausdrücke
  - Auswertung von Special Form Expressions

14 Korrektheit und Spezifikation

15 Rekursion und Induktion

Wertesemantik: Der Wert eines Ausdrucks

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

Definition (Bedeutung eines Terms)

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

Wertesemantik: Der Wert eines Ausdrucks

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

Definition (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.

Wertesemantik: Der Wert eines Ausdrucks

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

Definition (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 beeinfußt 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 (Der Wert eines Terms)

Der Wert eines Ausdrucks ist Wert des einfachsten äquivalenten Terms.

Definition (Normalform)

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

Innere Reduktion

- Mit der Definition

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

können wir reduzieren:

```
Variante 1:
(square (+ 3 4))
                ⟶    (square 7)  ;(+)
                ⟶    (* 7 7)     ;(square)
                ⟶    49          ;(*)
```

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

Äußere Reduktion

Variante 2:

```
(square (+ 3 4))
        ⟶  (* (+ 3 4) (+ 3 4))  ;square
        ⟶  (* 7 (+ 3 4))        ;(+)
        ⟶  (* 7 7)              ;(+)
        ⟶  49                   ;(*)
```

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

☞ Beachte:

Satz (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)
```

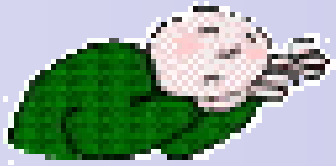
Vorgezogene Auswertung

Definition (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 (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).

S-Exressions: 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.

Reduktionsstrategie für funktionale Ausdrücke

Die Auswertung ist applikativ:

- Zunächst werden die Argumente evaluiert. Die Reihenfolge ist undefiniert.
- 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.

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.

quote

```
(define zahlen (quote (1 2 3))
(define zahlen2 '(1 2 3))
```

- Quote soll ja gerade verhindern, daß das Argument ausgwertet 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

Manche Terme können nicht zu einem wohldefinierten Wert reduziert werden, auch wenn sie syntaktisch wohlgeformt sind, siehe beispielsweise:

> (**cond** [**#f** 1]) $\longrightarrow$ $\bot$

- Damit jeder syntaktisch wohldefinierte Term ausnahmslos einen Wert hat, führen wir in der denotationellen Semantik für Terme mit undefiniertem Wert den speziellen Wert $\bot$ (gesprochen „bottom“) ein.
- Dieses ermöglicht uns, auch *partielle Funktionen* zu untersuchen.

Implementationsabhängigkeiten

Der Wert einiger Ausdrücke ist im Revised Report als undefiniert 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
```

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 (Striktheit einer Funktion)

Eine Funktion f heißt strikt, wenn $f(\perp) = \perp$, d.h. die Funktion hat nur für definierte Argumente einen definierten Wert.

Striktheit einer Funktion

Definition (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 (Strikte Semantik)

Eine Programmiersprache hat eine *strikte Semantik,* wenn alle definierbaren Funktionen strikt sind.
Das Gegenteil heißt *nicht-strikte Semantik.* [**?**]

Strikte Auswertung

Definition (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

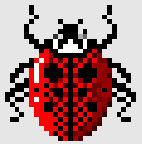
  ```
  (three ⊥) = ⊥.
  ```

Vorteil 2: Kontrollstrukturen

Beispiel

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)) ⟶ ⊥
>
```

Lazy Racket: Unendliche Listen

Beispiel (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
```

Korrektheit

Korrektheit von Programmen

Definition (Korrektheit)

Ein Programm heißt korrekt, wenn es genau die Spezifikation erfüllt, also für alle Argumente des Definitionsbereichs die korrekten Resultate berechnet.[**?**]

Definition (Partielle Korrektheit)

Ein Programm heißt partiell korrekt, wenn es, sofern es terminiert, korrekte Resultate liefert.

Definition (Totale 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)

Debugging is a disgrace!

John McCarthy, 1983

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 Semipraedikat kann als Vertrag verwendet werden.

Beispiel (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 *z*
  (flat-contract odd?) 9)

(define/contract *zz*
  (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

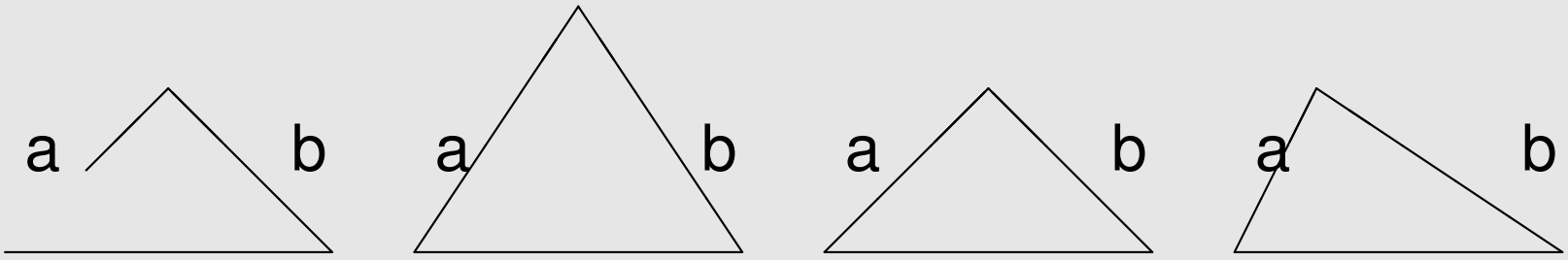
Funktionen,

- die nur ein Resultat zurückgeben,
- die eine feste Zahl von Argumenten haben,
- und die nicht-rekursiv sind,

können mit dem `->`-Operator spezifiziert werden:

```
(define/contract
  nusy      ; Funktionsname
  (-> number? symbol?) ; Kontrakt
  (lambda (n) 'eins)) ; die Funktion
    ;oder in Infix-Notation
(define/contract
  nuboint
  (integer? boolean? . -> . integer?)
  (lambda (n w) 1))
```

Beispiel: Dreiecke



1. kein
2. gleichseitiges
3. gleichschenkliges
4. schiefwinkeliges

Dreieck

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 c))]
       [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 c)))
       (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])))
```

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 (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 expe
 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 (Testfälle für my-length)

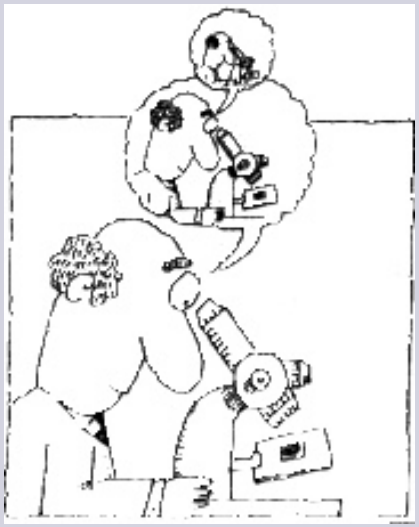
```
(require test-engine/racket-tests)
(define (my-length xs)
  (cond [(not (list? xs))
         (error 'xs "keine Liste")]
        [(null? xs) 0]
        [else
         (+ 1 (my-length (cdr xs)))]))

(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!
```

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 rekursive Definition muß immer einen *trivialen Fall* enthalten, der nicht mehr auf die Definition Bezug nimmt.

Die Menge der natürlichen Zahlen

Definition (Die Menge der natürlichen Zahlen)

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 = $\mathcal{N}$).

Minimalrekursive Funktion

Definition (Minimal-rekursive Funktionsdefinition)

Eine Funktionsdefinition heißt 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.
```

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
```

Lineare Rekursion

Definition (Lineare Rekursion)

Eine Funktionsdefinition, die sich auf der rechten Seite der definierenden Gleichung in jeder Fallunterscheidung selbst nur einmal verwendet, heißt 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))
  ⟶ (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

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:

```
(f (cons x xs)) = (cons (g x) (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 (Indirekte Rekursion)

Eine rekursive 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 (Baumartige Rekursion)

Eine rekursive Definition ist baumartig wenn in der Definition in einer Fallunterscheidung mehrfach auf die Definition Bezug genommen wird.

Beispiel (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
```

Ein Binärbaum

Ein Ternärbaum

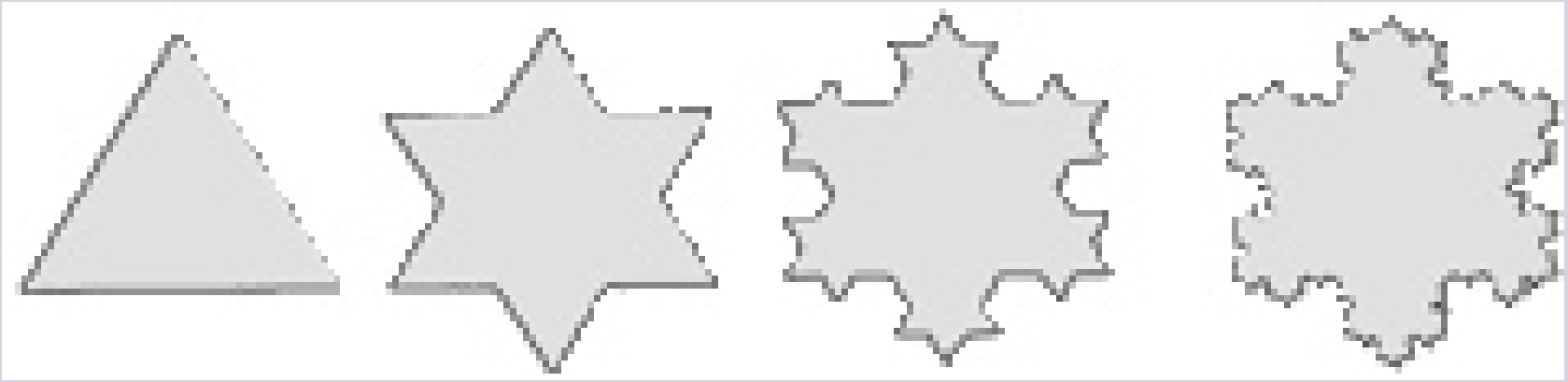
Von Katzen

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. –

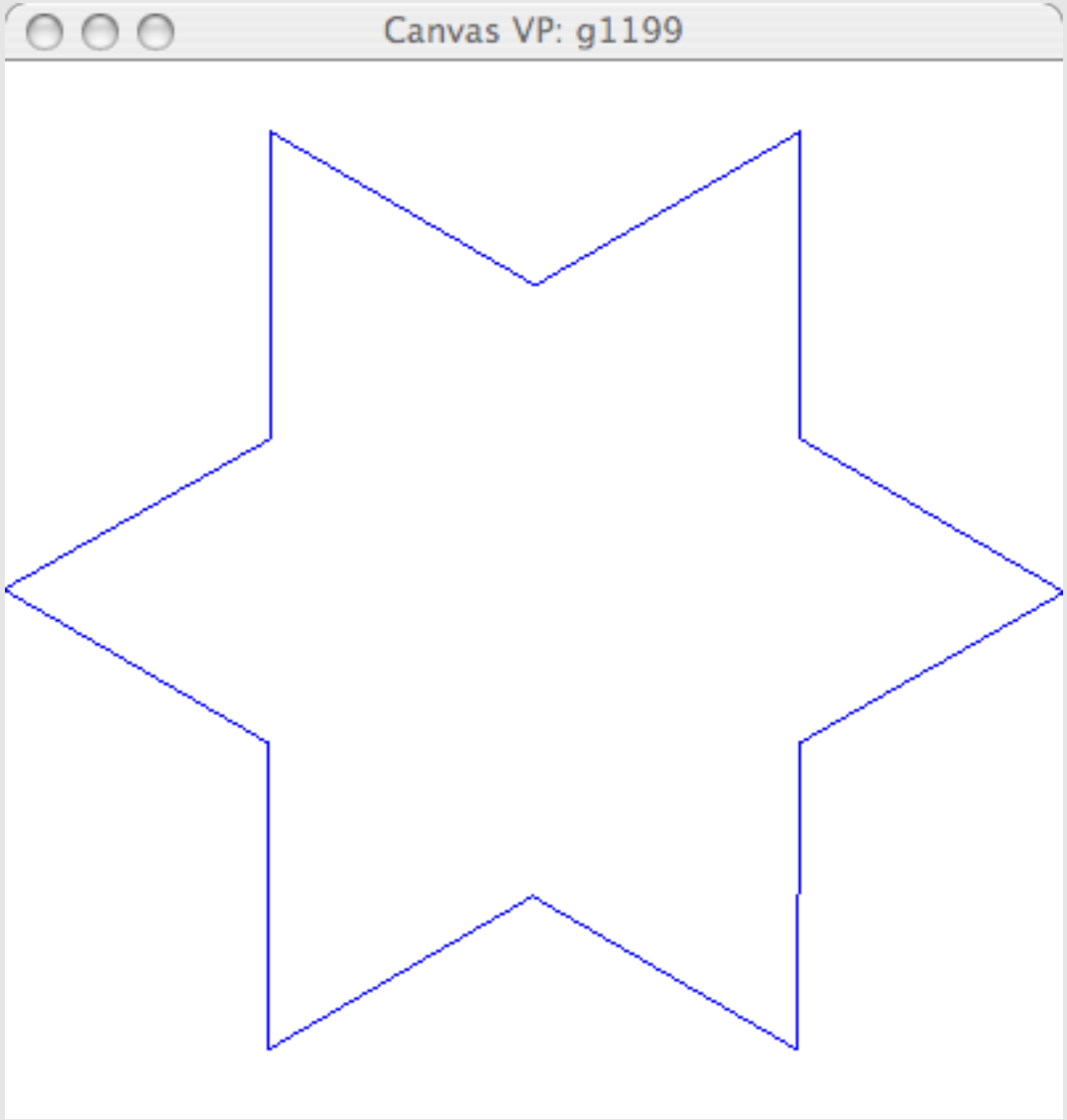
Ein Jahr ging um, und Katzen sind die Kätzchen,
Und Maitag ist's! — Wie soll ich es beschreiben,
Das Schauspiel, das sich jetzt vor mir entfaltet!
Mein ganzes Haus, vom Keller bis zum Giebel,
Ein jeder Winkel ist ein Wochenbettchen!
Hier liegt das eine, dort das andre Kätzchen,
In Schränken, Körben, 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

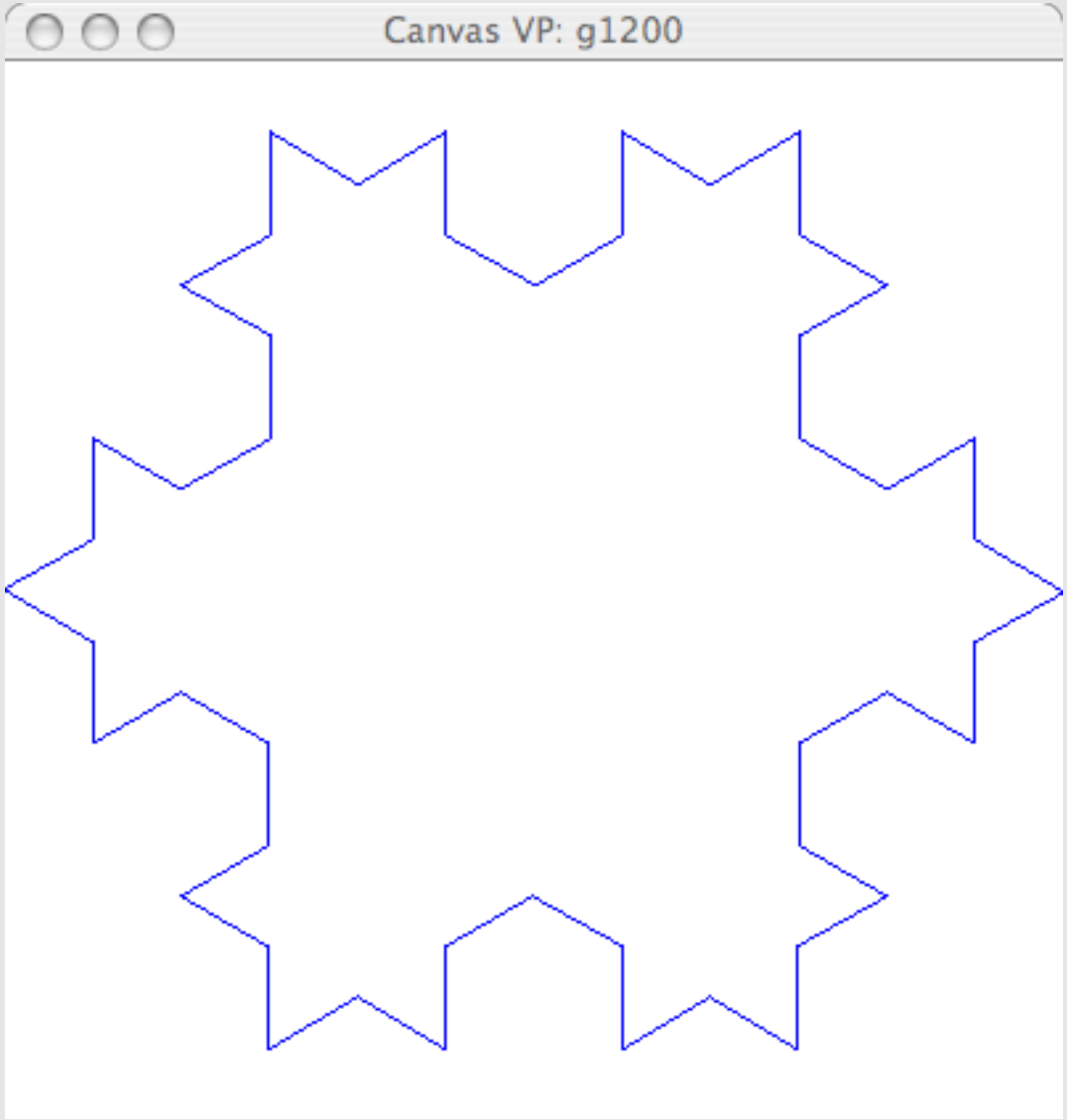
Baumrekursion: Die Koch'sche Schneeflocke



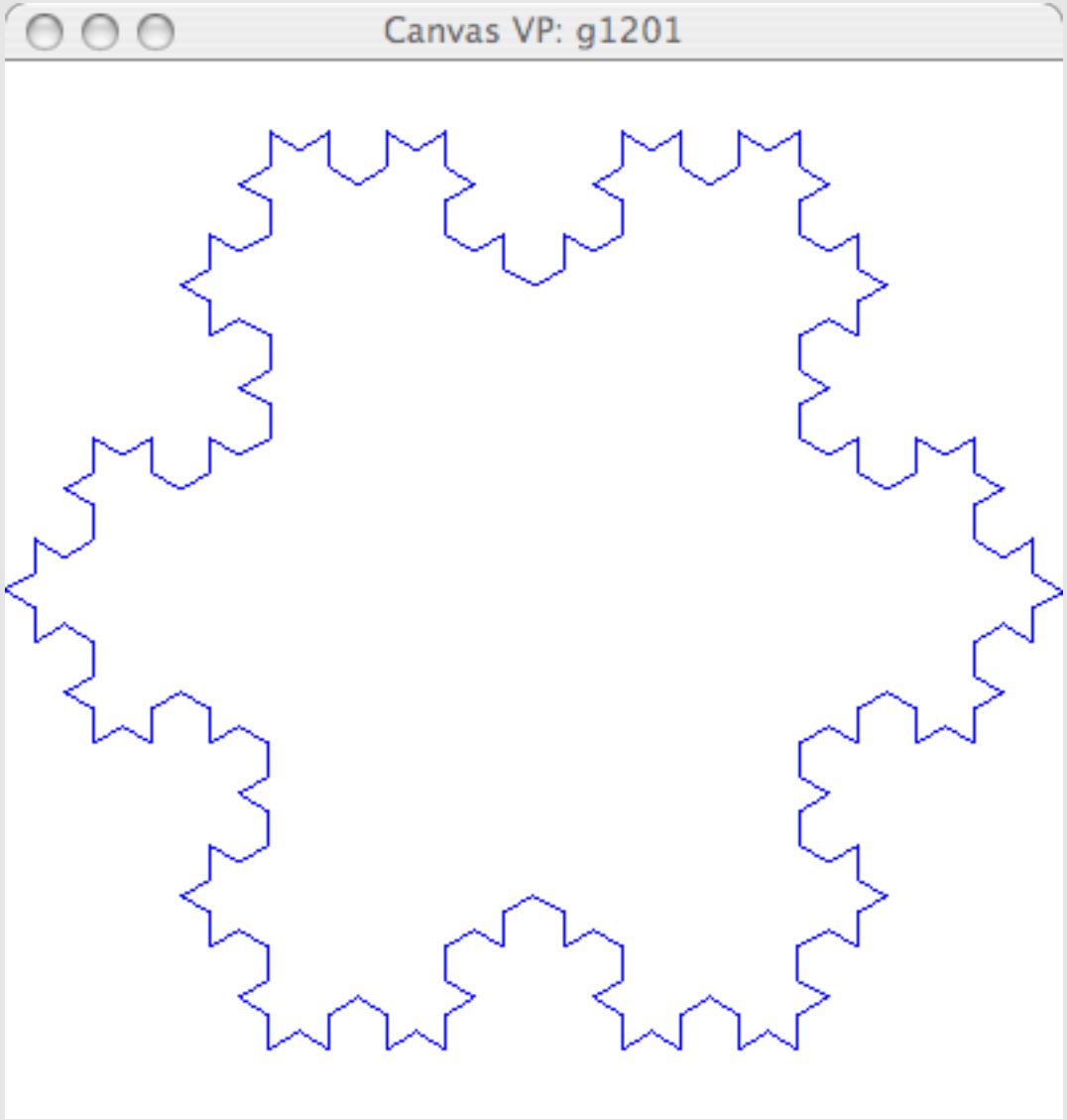
- Ein Bild mit baumartig rekursiver Struktur.
- An jedes Dreieck werden rekursiv drei kleinere Dreiecke angefügt.



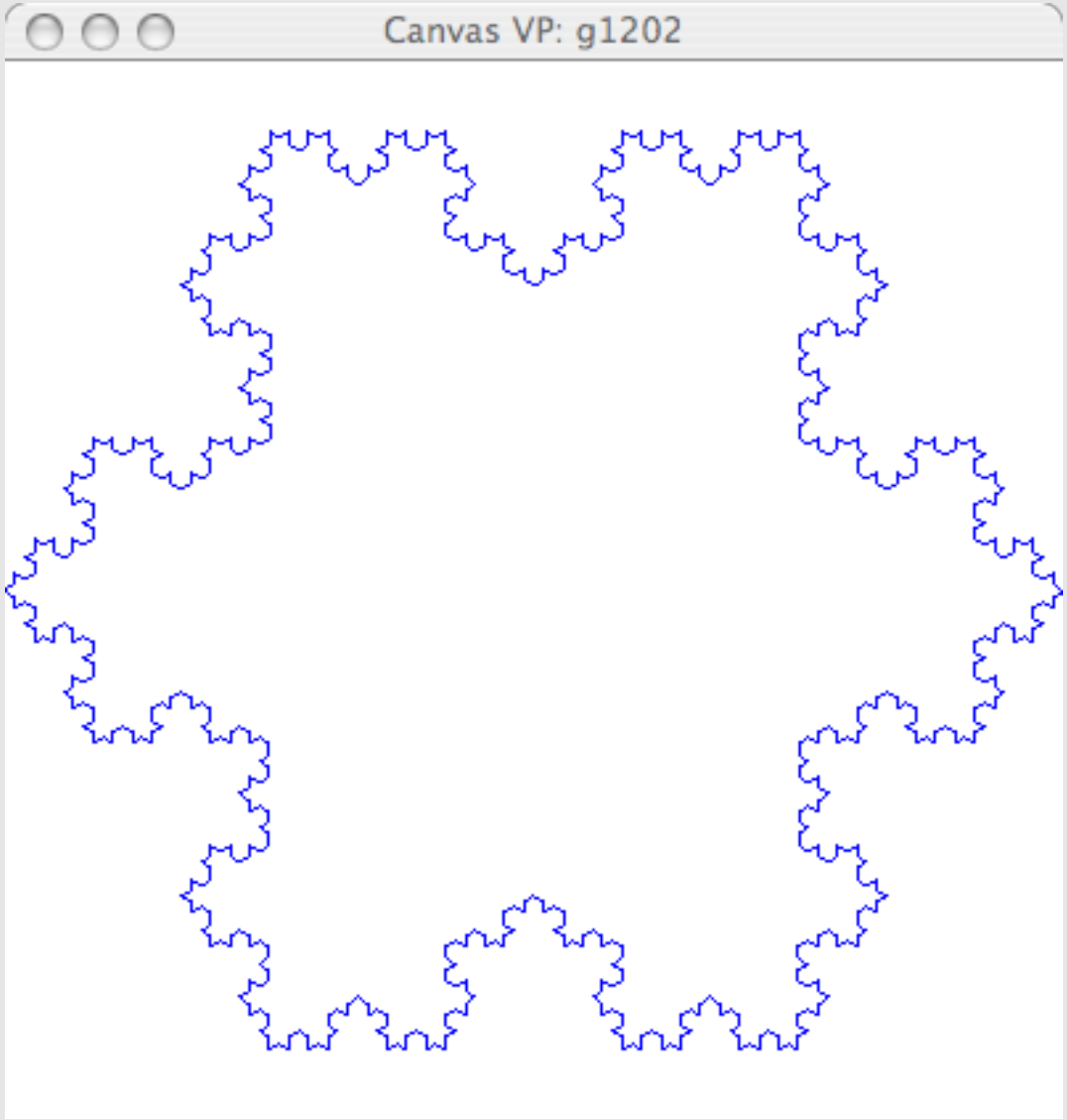
Iteration 1, 12 Linien



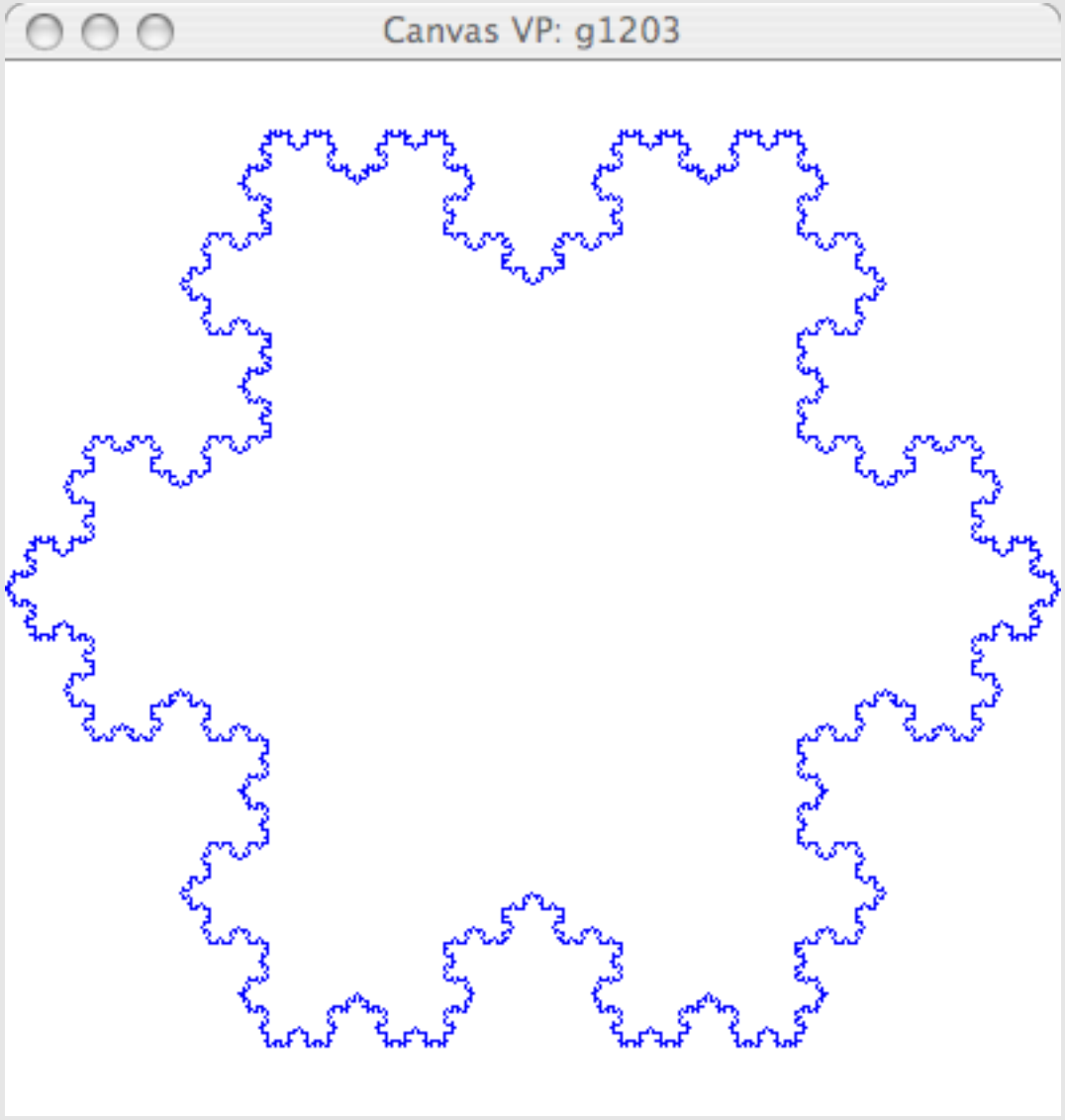
Iteration 2, 48 Linien



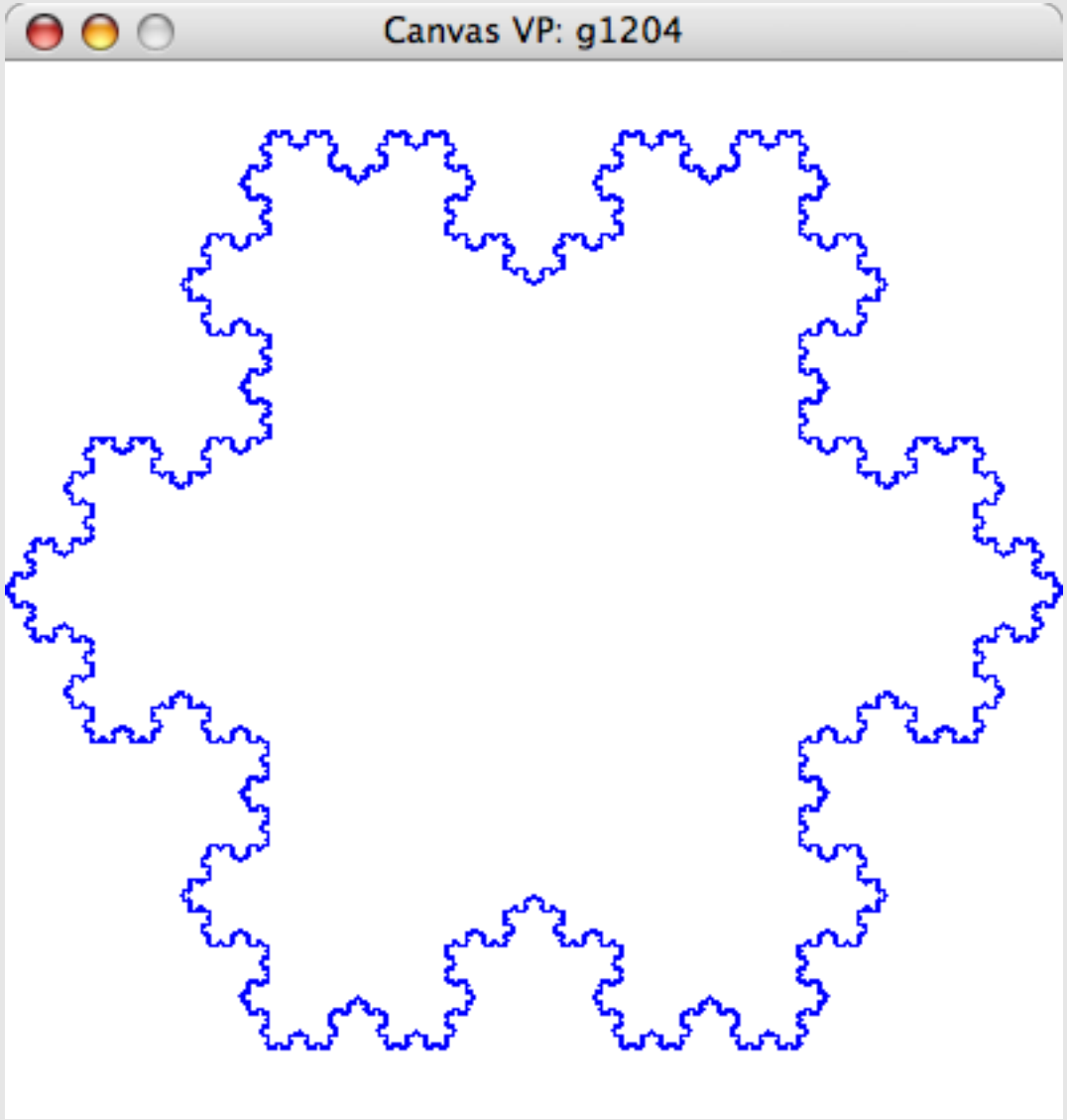
Iteration 3, 192 Linien



Iteration 4, 768 Linien



Iteration 5, 3072 Linien



Iteration 6, 12288 Linien

- 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

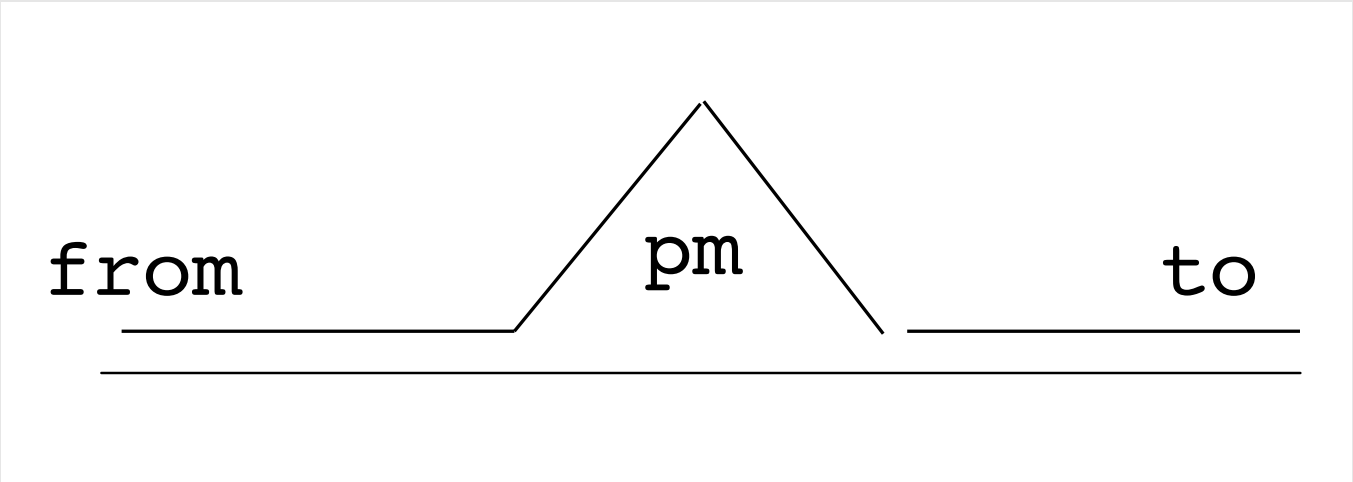
- Der Aufwand wächst exponentiell: In jedem Iterationsschritt vervierfacht sich die Zahl der Linien.

Iteration i	**Zahl der Linien** $= \mathbf{3 \times 4^i}$	**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

- 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

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 triangl
                      (* (distance from to) 1/3 (sqrt 3/
             [tip (translate-pos pm tipOffset)])
        (+ (snowflake-line from 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
```

snowflake

snowflake

Wiederholung: Arten der Rekursion

- Minimalrekursiv

Wiederholung: Arten der Rekursion

- Minimalrekursiv
- Lineare Rekursion

Wiederholung: Arten der Rekursion

- Minimalrekursiv
- Lineare Rekursion
- Indirekte Rekursion

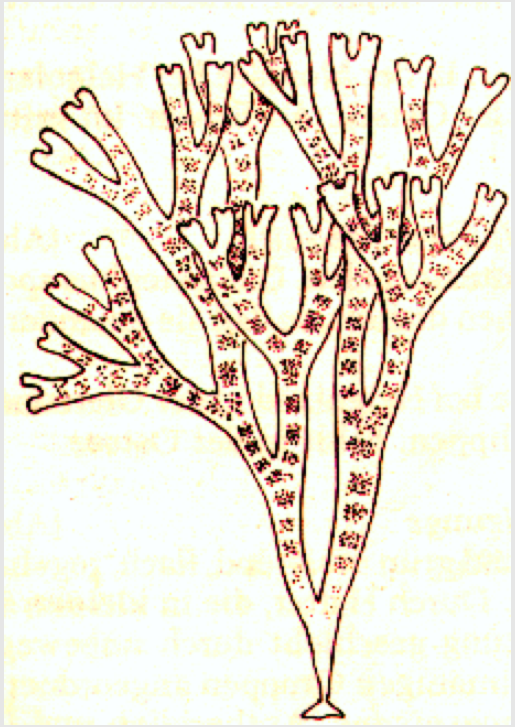
Wiederholung: Arten der Rekursion

- Minimalrekursiv
- Lineare Rekursion
- Indirekte Rekursion
- Baumrekursion

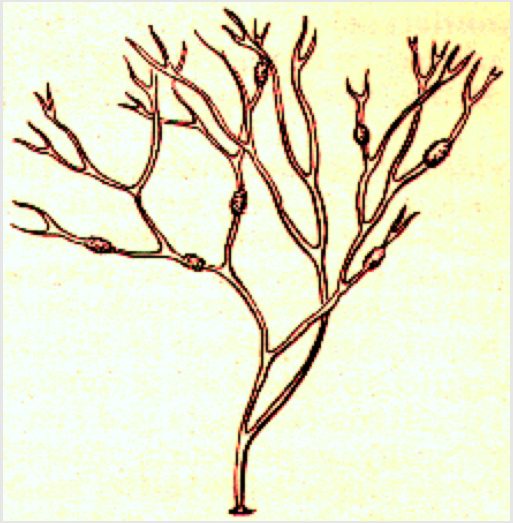
Wiederholung: Arten der Rekursion

- Minimalrekursiv
- Lineare Rekursion
- Indirekte Rekursion
- Baumrekursion
- Geschachtelte Rekursion

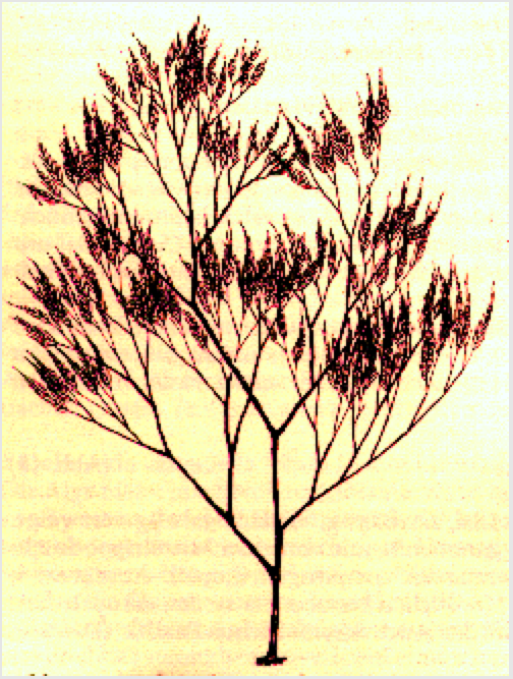
Baumrekursion in der Natur: Algen



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 (Geschachtelte Rekursion)

Eine Rekursion ist geschachtelt, wenn die Funktion in der rekursiven Verwendung selbst als Argument mitgegeben wird.

Beispiel (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 (Ein effizienter Modulo-Algarithmus)

nach [**?**]

$$
\begin{array}{rrcl}
\text{Definition:} & \mathrm{mod}(a,b) & = & a, a < b \\
 & \mathrm{mod}(a,b) & = & \mathrm{mod}(a-b,b), a \geq b \\
\text{Es gilt:} & \mathrm{mod}(a,b) & = & \mathrm{mod}(\mathrm{mod}(a,2*b),b), a \geq 2*b
\end{array}
$$

```
(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
```

Korrektheit rekursiver Funktionen

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 $n \in \mathcal{N}$.

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.

Potenzieren, x^n

Beispiel (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.

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.

Terminieren von power

Beweis.

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 $k+1=n$.

Terminieren von power

Beweis.

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 (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.

Hilfsdefinitionen: k = (**length** xs) , Induktion über k,

Annahme: f terminiert für alle Argumente.

Induktionsverankerung: (my−map f '()) terminiert, da keine weitere Funktionsanwendung erfolgt.
Die Annahme ist für k=0 richtig.

(my−map f '(x)) terminiert für beliebige x, da
(**cons** (f x) (my−map f '()) terminiert.
Die Annahme ist für k=1 richtig.

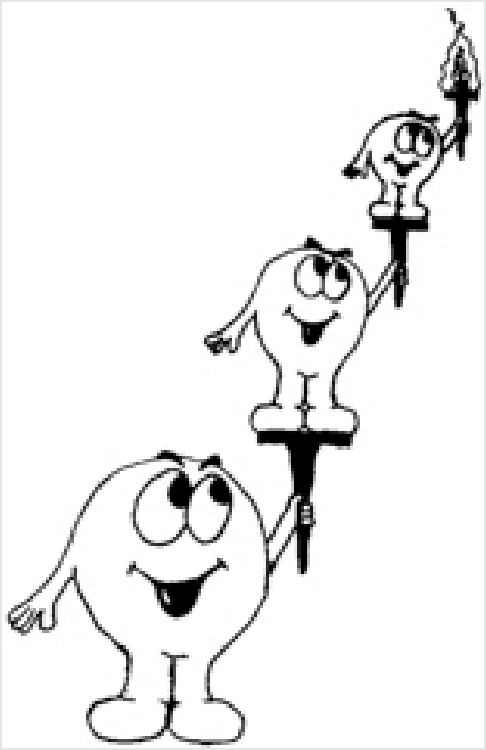
Induktionsschritt: Sei (**cons** x xs) eine Liste der Länge k+1.
Wir postulieren, daß (my−map f xs) terminiert.

Da nach dem Terminieren von (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 $k \in \mathcal{N}$.

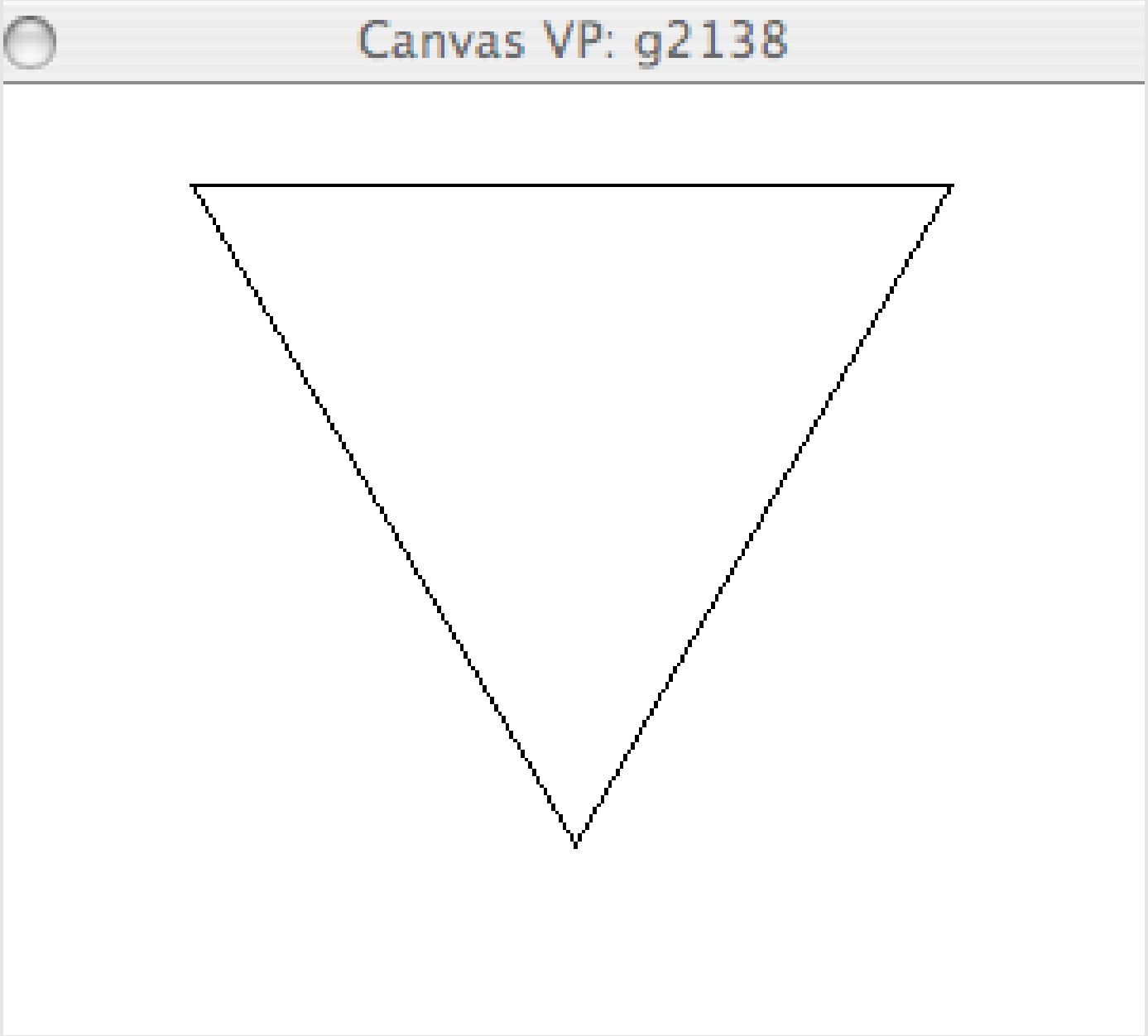
□

Terminieren einer Rekursion: Bedingungen

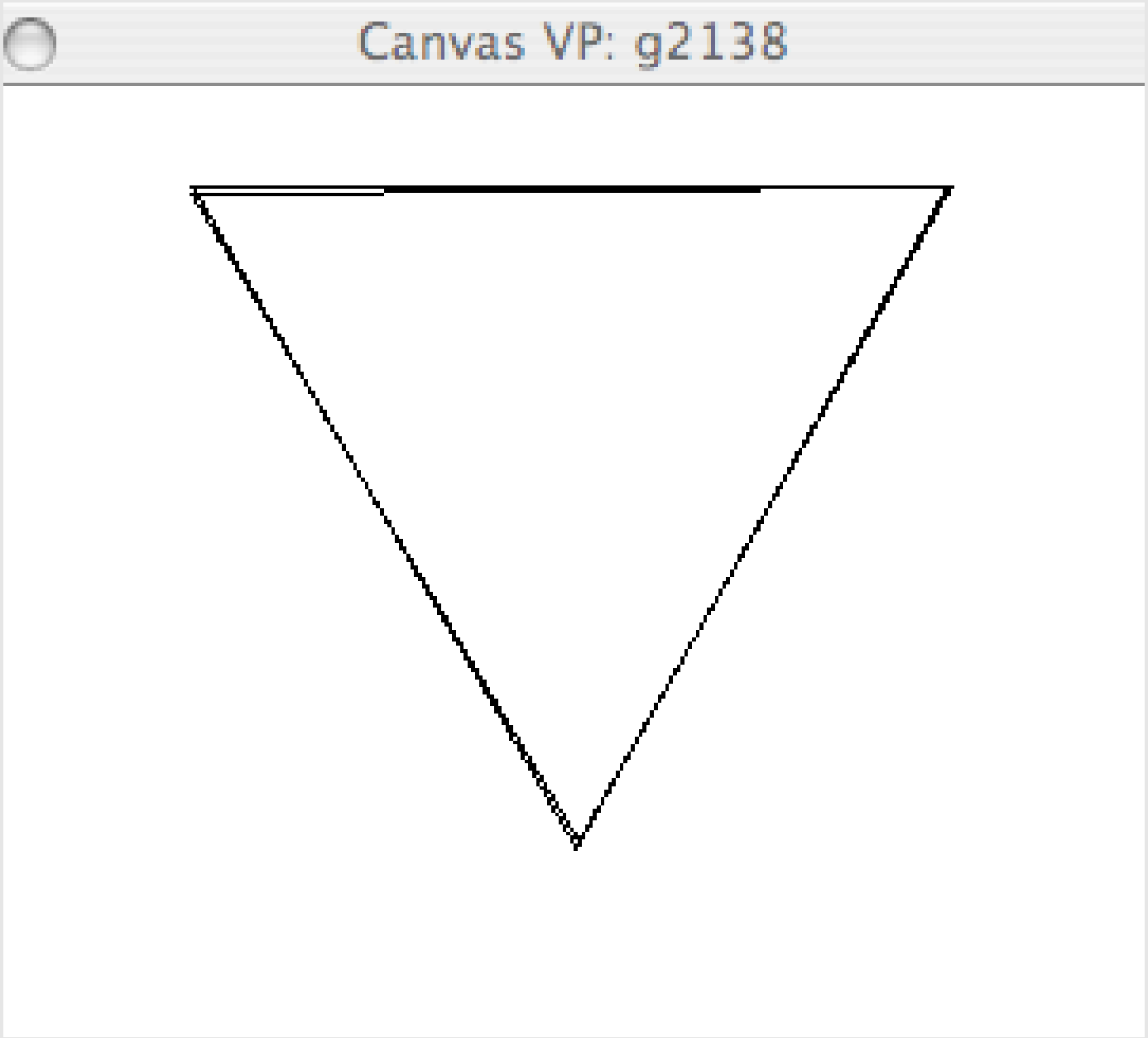


- Jeder Rekursionsschritt sollte die Argumente dichter an die Abbruchbedingung bringen, z.B.
  - ein Zahlenargument inkrementieren
  - oder ein Listenargument verkürzen.
- Die Abbruchbedingung muß so formuliert werden, daß man nicht über das Ziel hinausschießen kann.

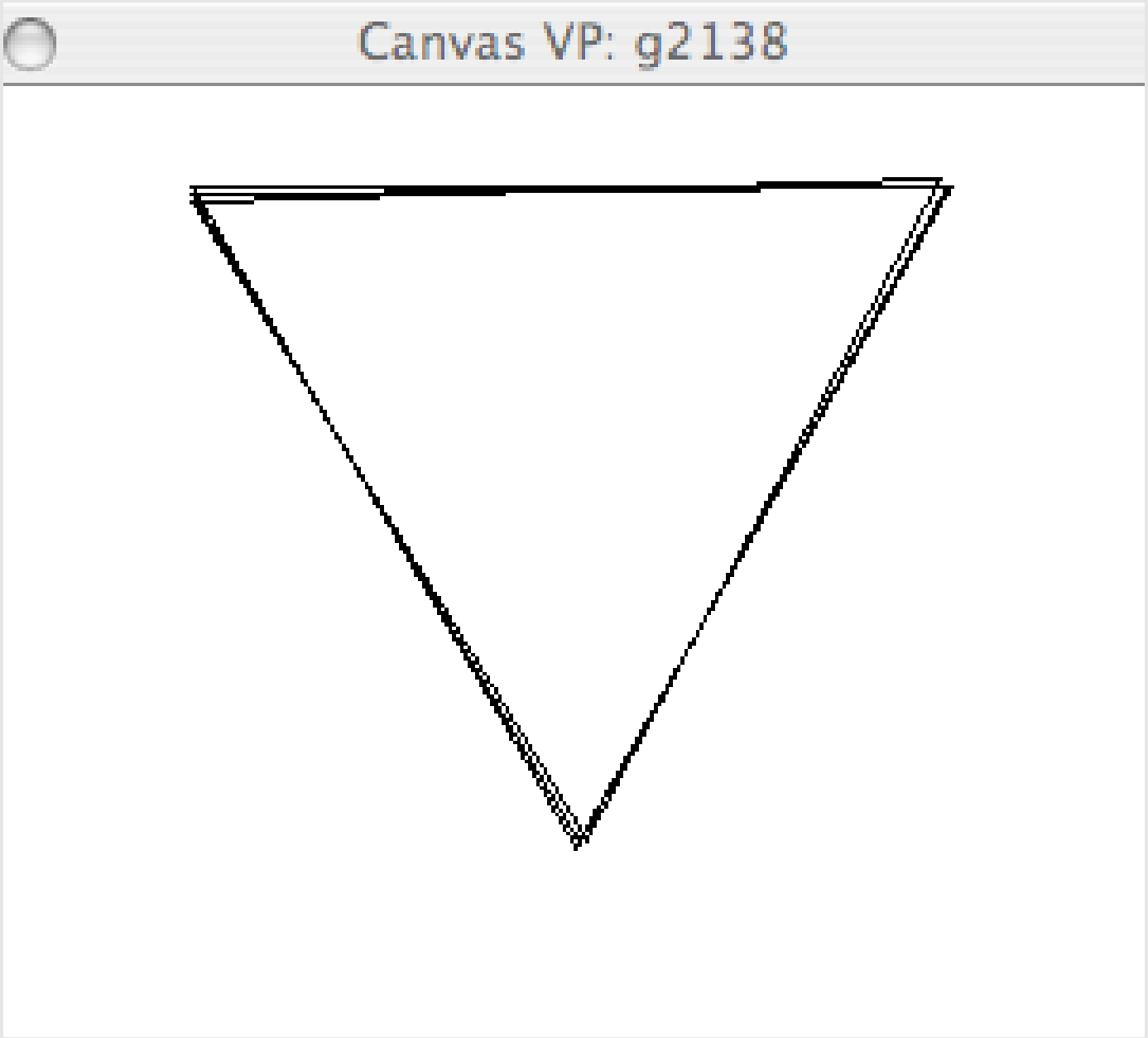
Ein linear-rekursiver Prozeß



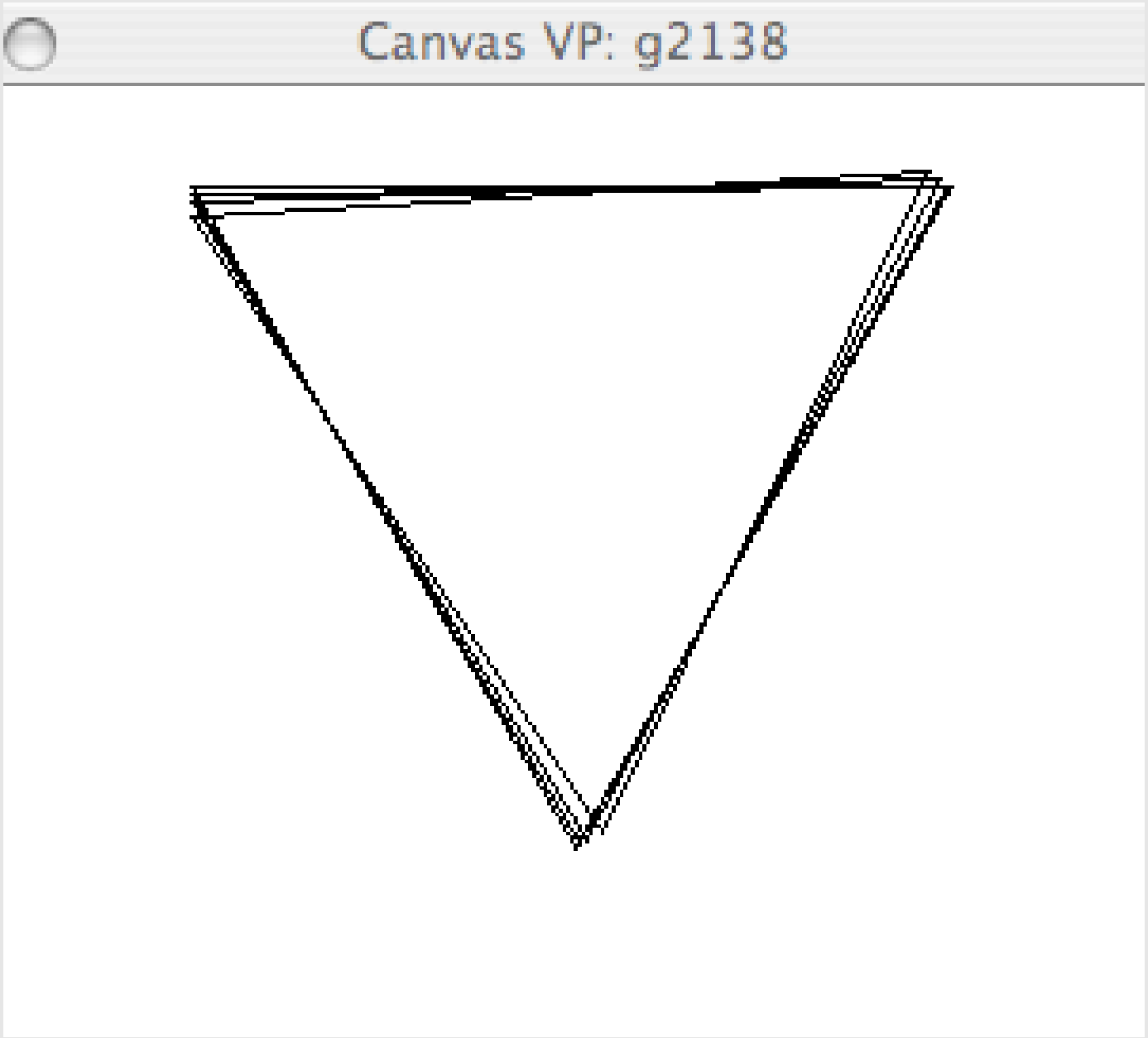
Ein linear-rekursiver Prozeß



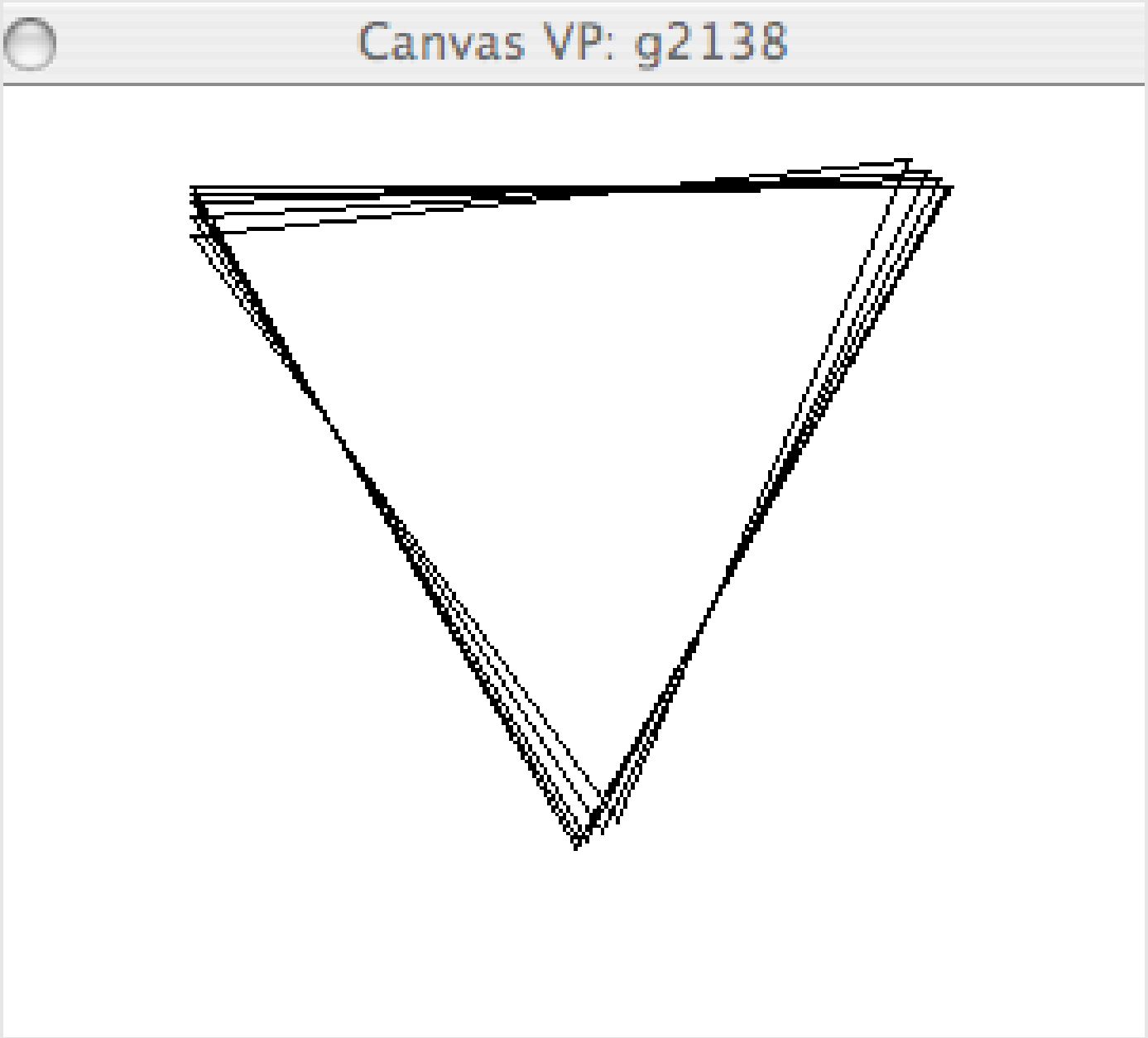
Ein linear-rekursiver Prozeß



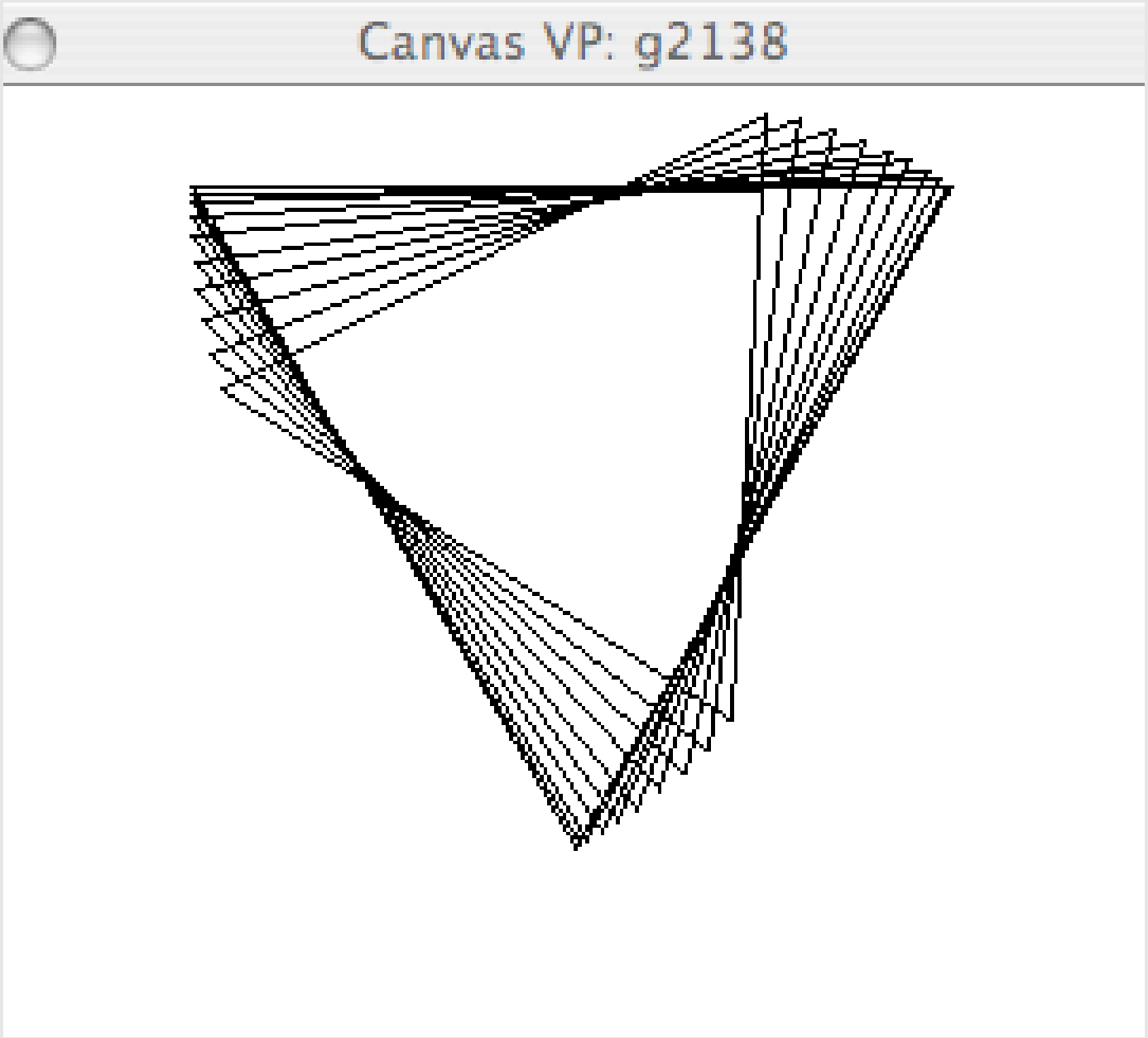
Ein linear-rekursiver Prozeß



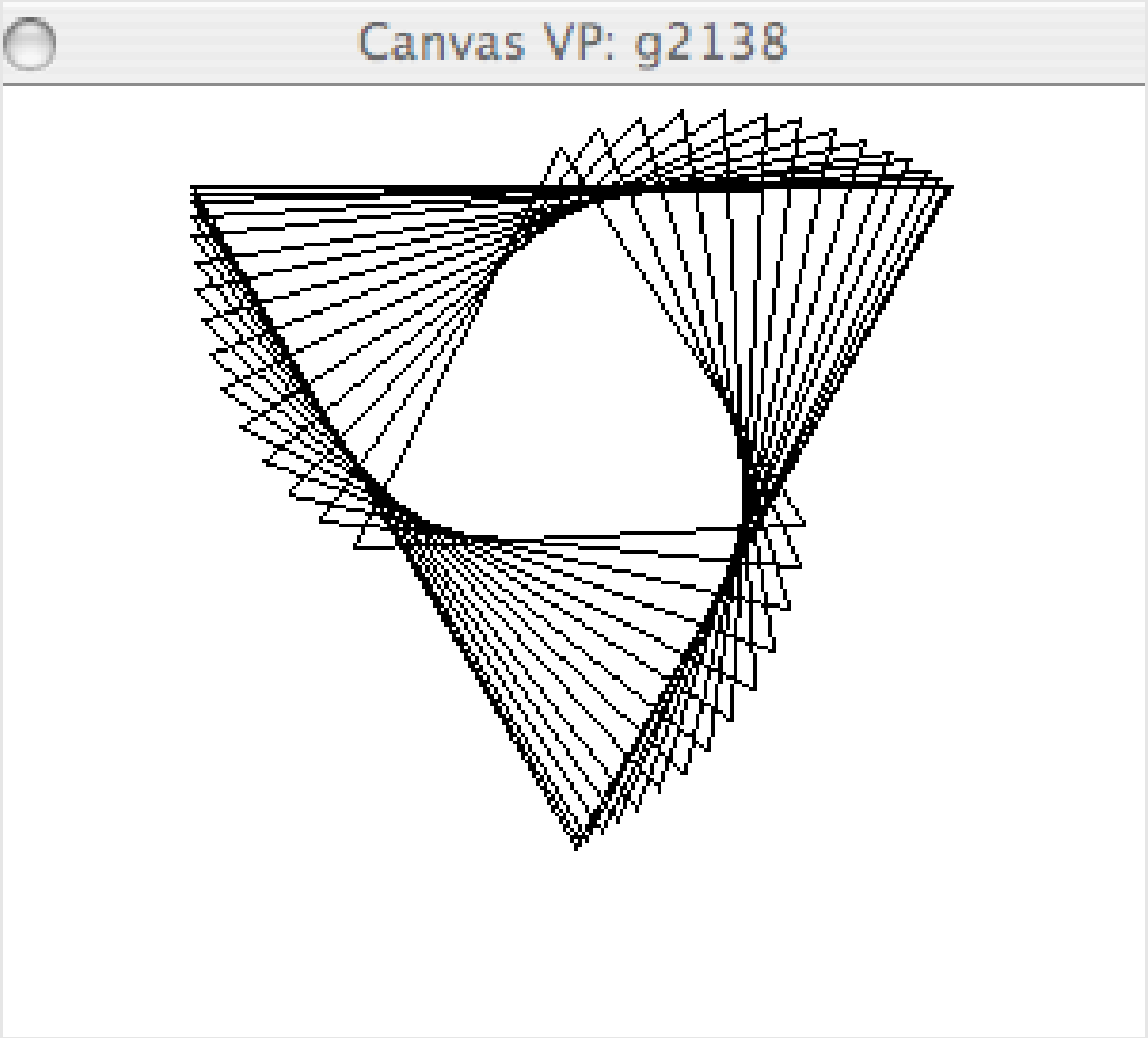
Ein linear-rekursiver Prozeß



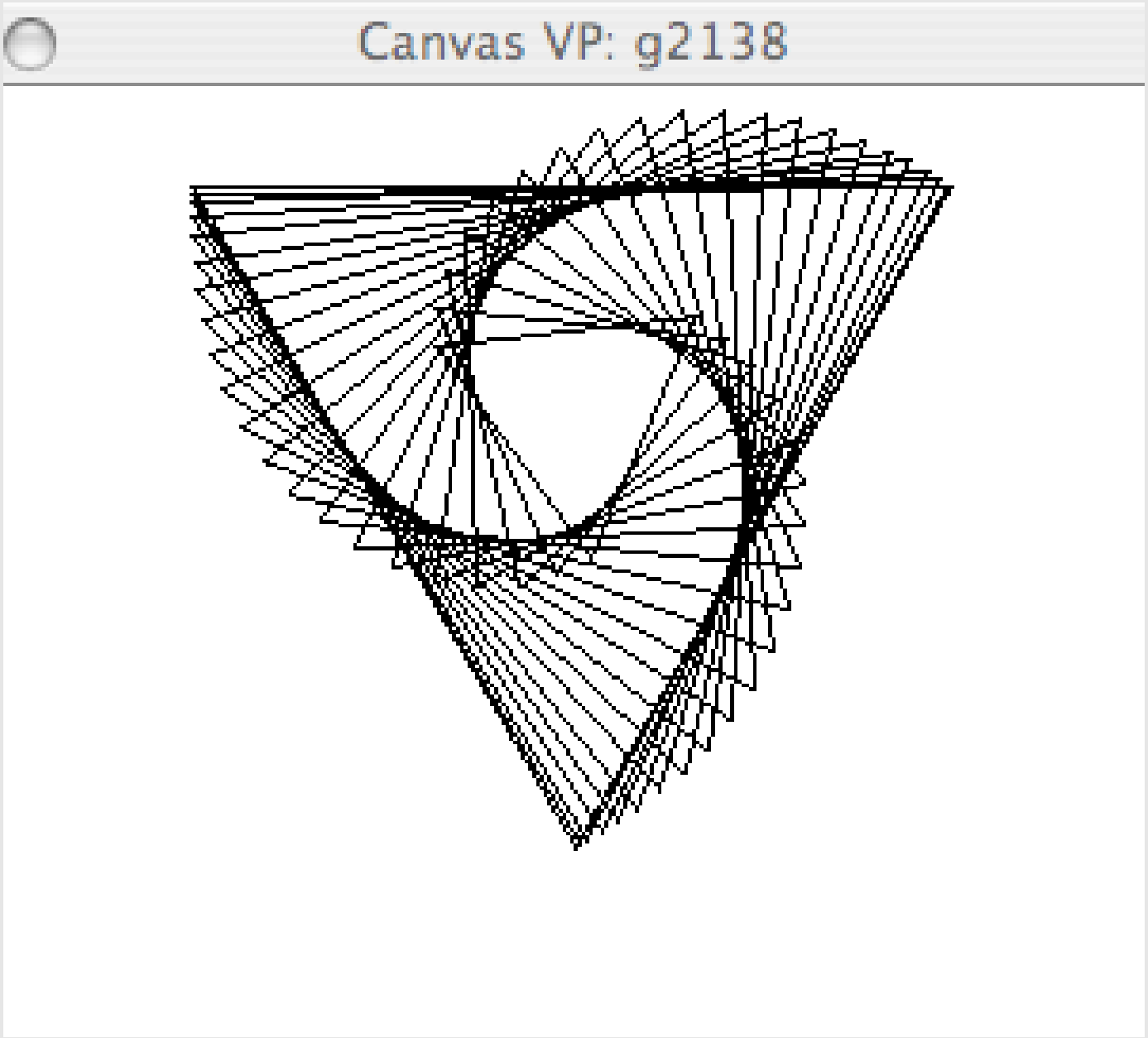
Ein linear-rekursiver Prozeß



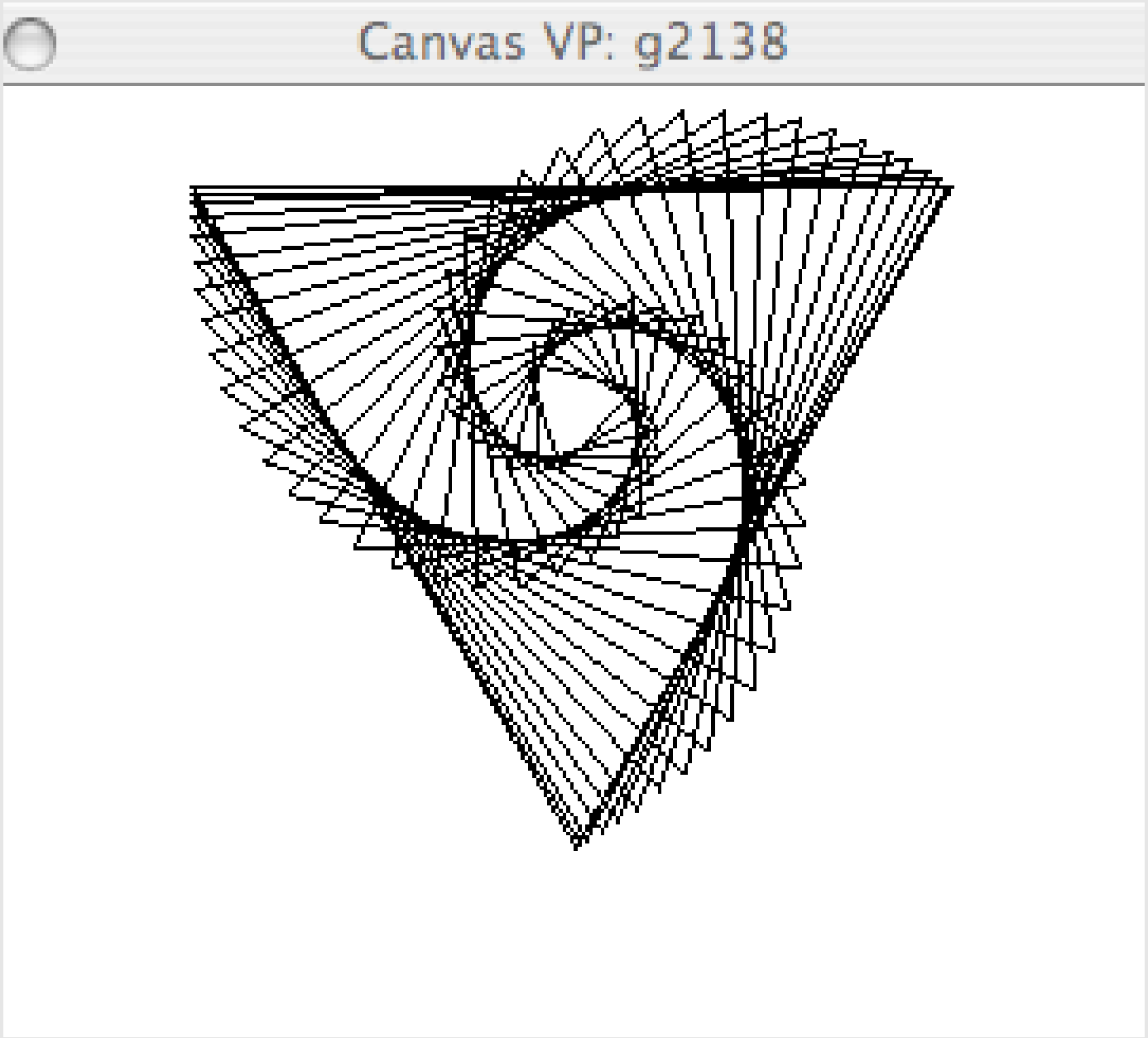
Ein linear-rekursiver Prozeß



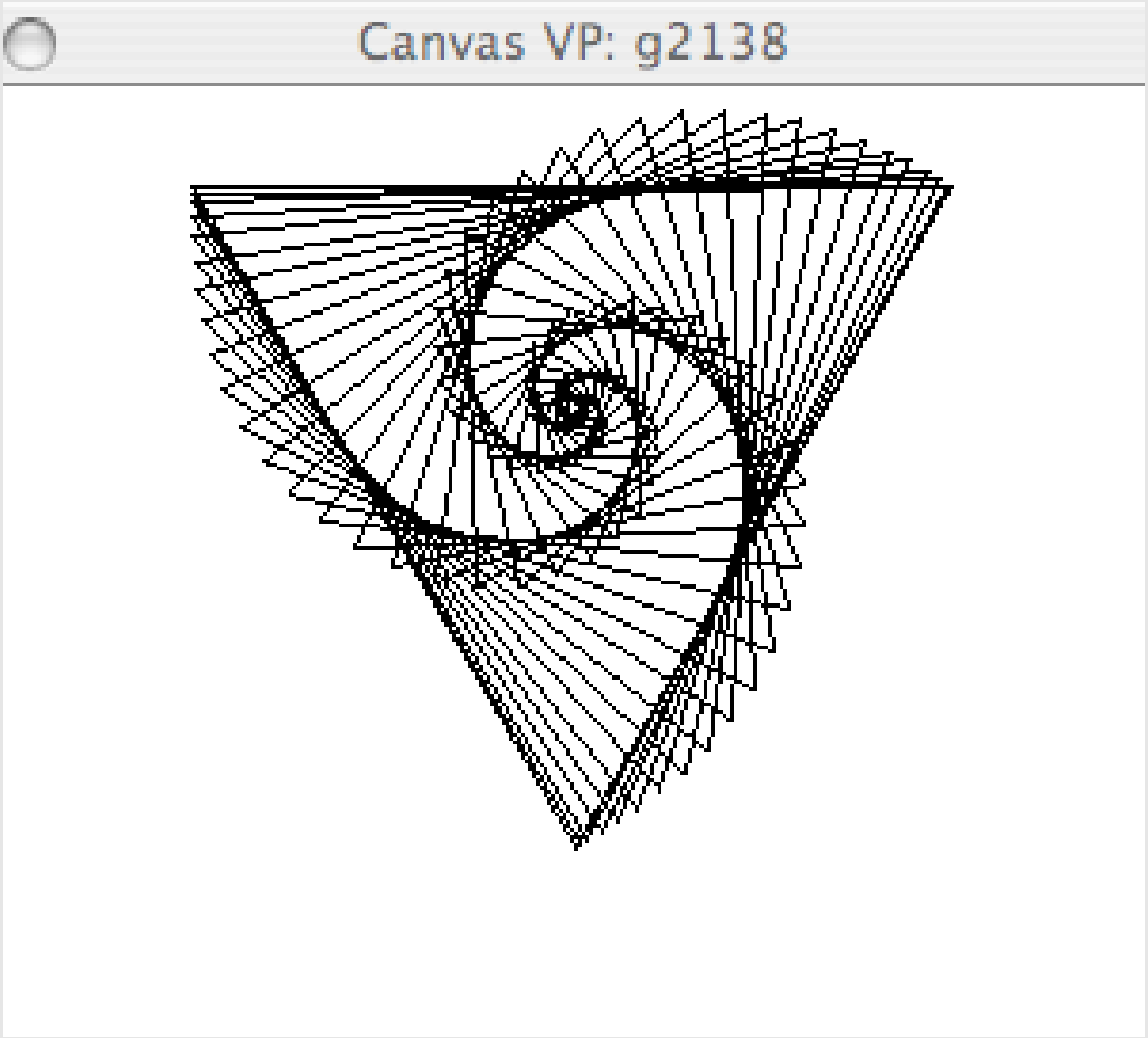
Ein linear-rekursiver Prozeß



Ein linear-rekursiver Prozeß



Ein linear-rekursiver Prozeß



snowflake

Rekursive Prozesse

- Funktionale Ausdrücke definieren nicht nur den Wert, für den sie stehen, sondern auch einen Berechnungsprozeß, der diesen Wert ermittelt.
- Rekursive Ausrücke 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 (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
```

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 (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 (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 (length-acc, ohne Nachklappern)

```
(define (length_acc xs acc)
  (if (null? xs)
      acc
      (length_acc
         (cdr xs)
         (+ acc 1))))

(define (my_length_acc xs)
; verbergen des Akkumulators
  (length_acc xs 0))
```

Trace: Länge einer Liste, endrekursiv

Beispiel (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.

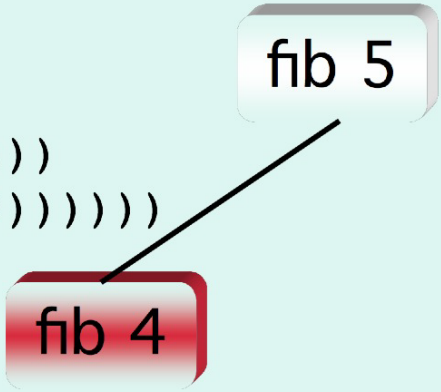
```
(define (fib n)
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [else (+ (fib (- n 1))
                 (fib (- n 2)))]))
```

```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

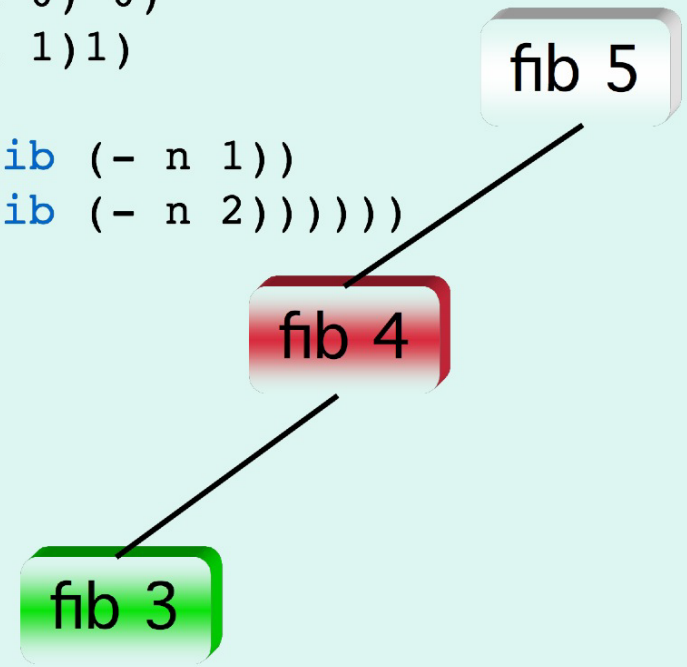
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

fib 5

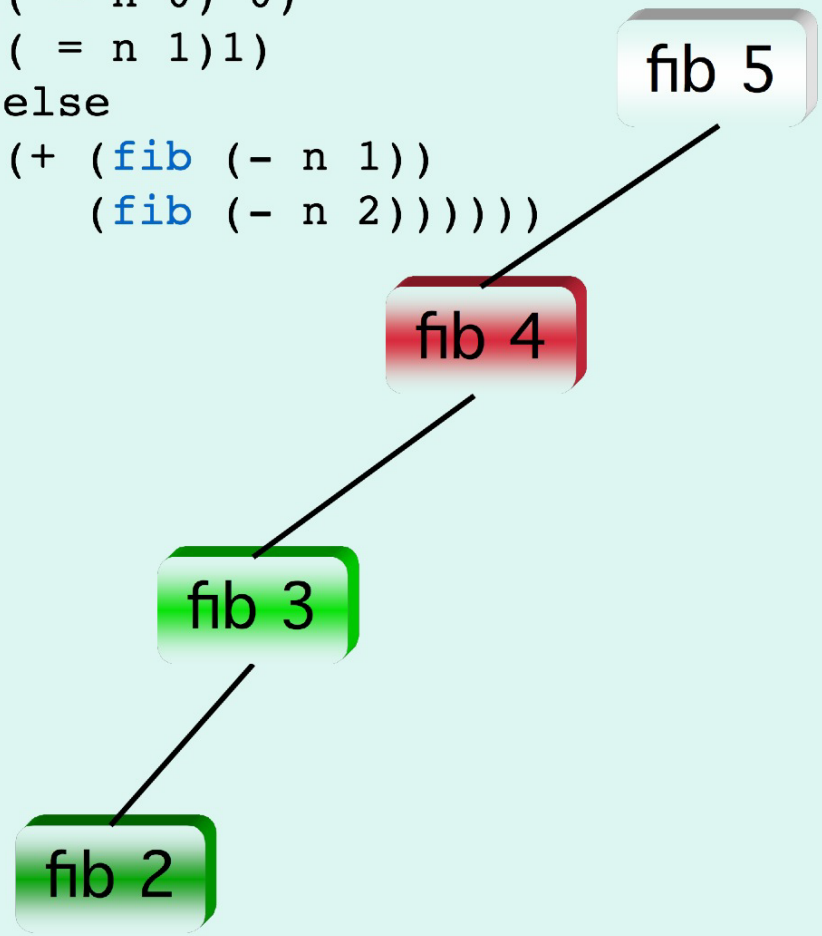
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



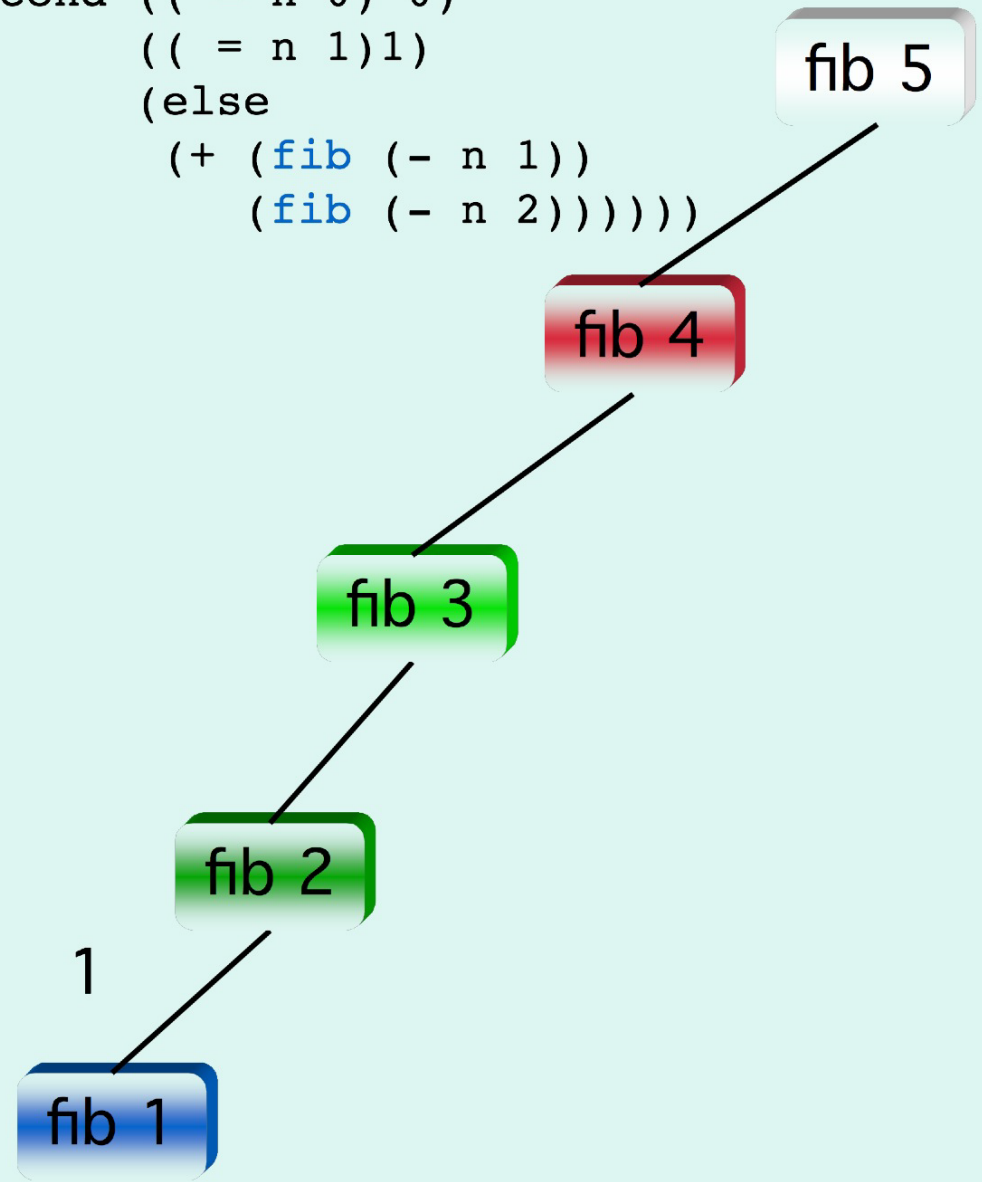
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



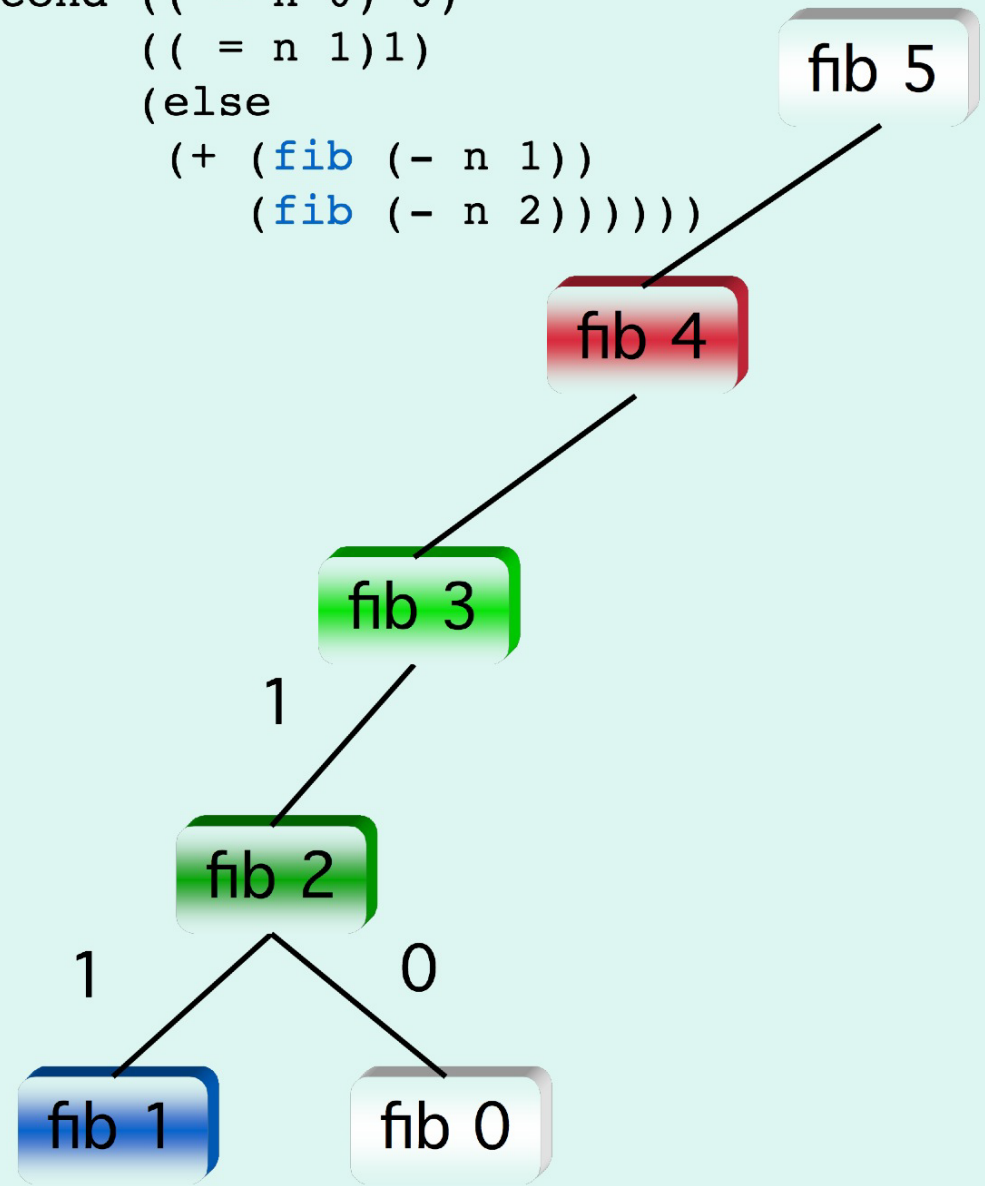
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



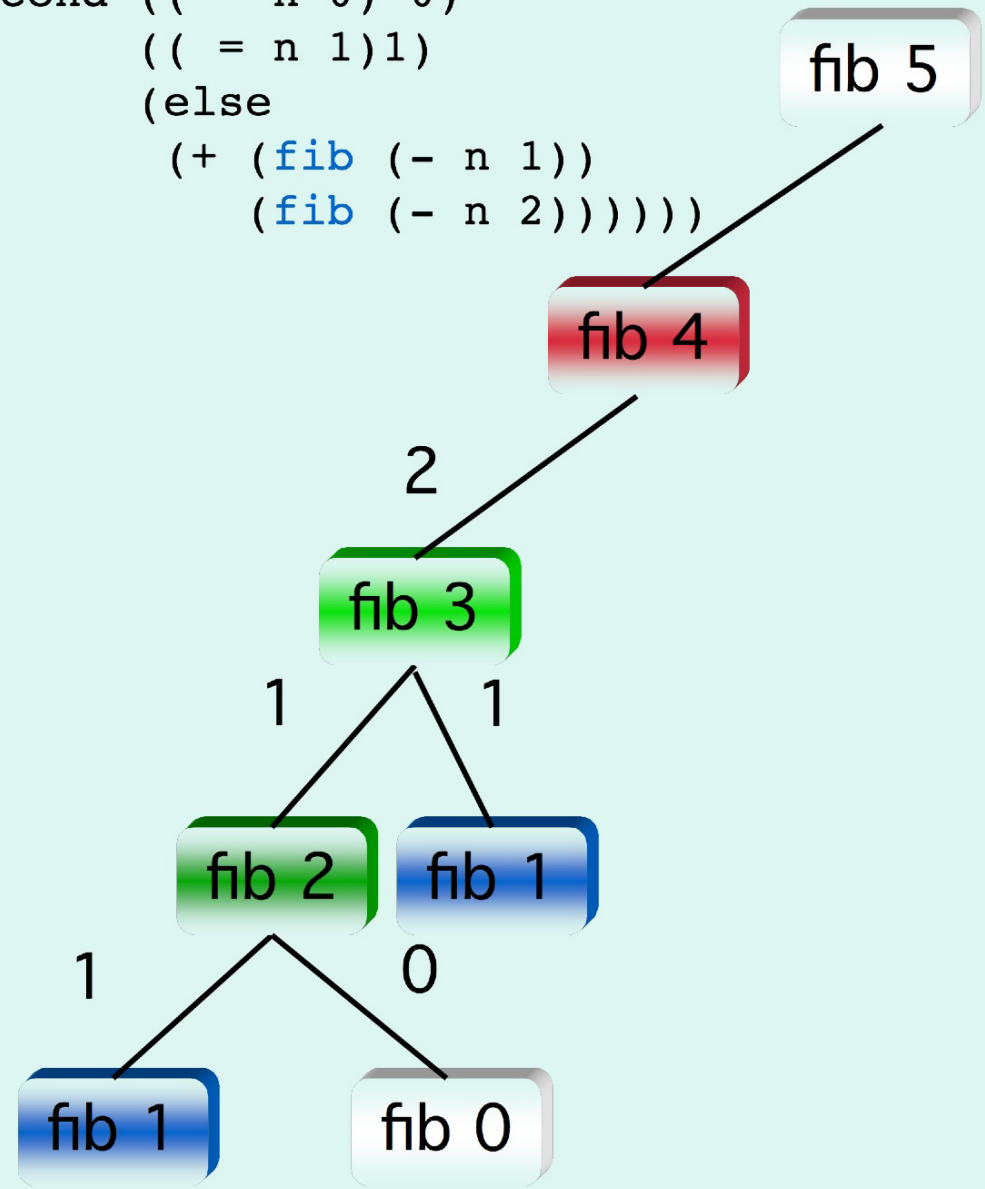
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



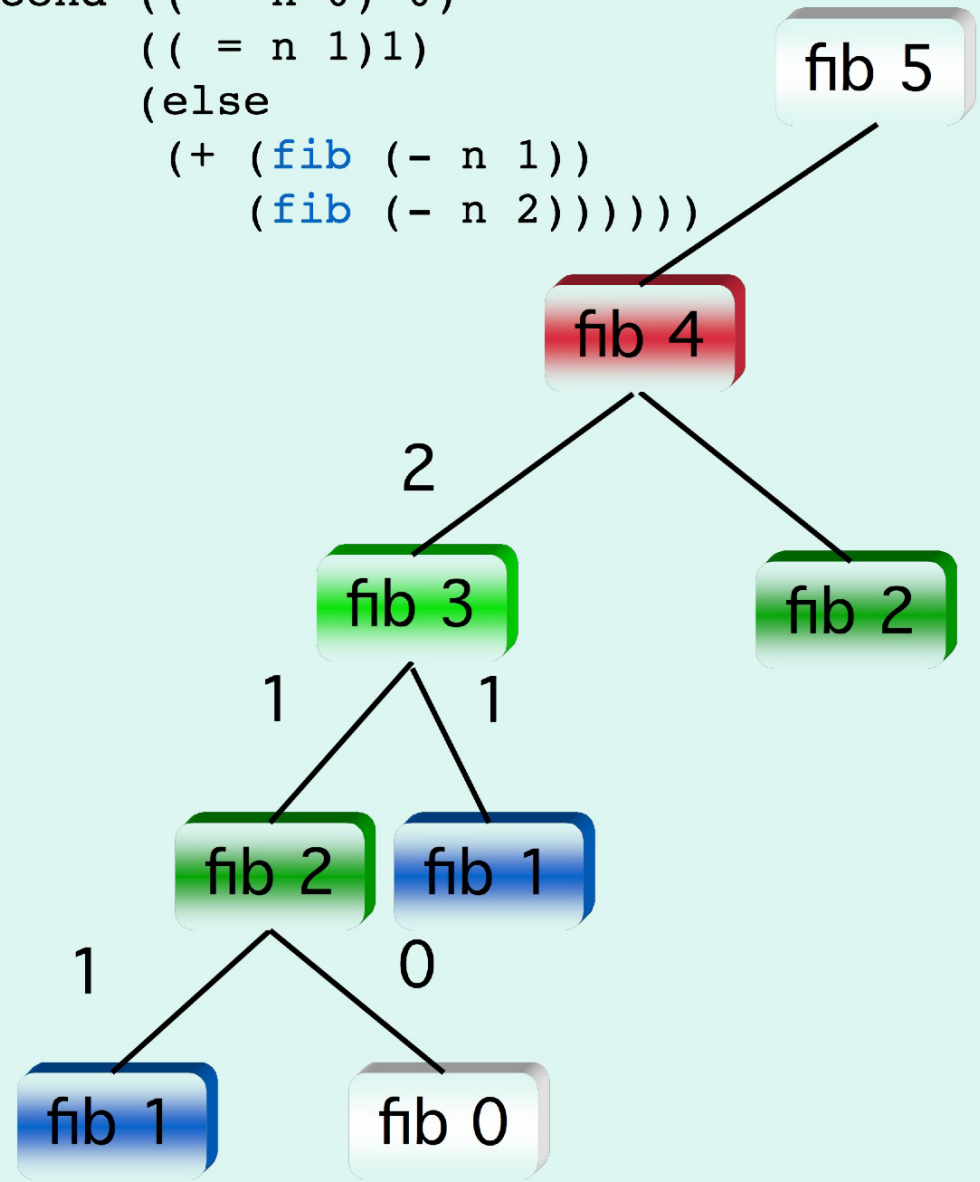
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



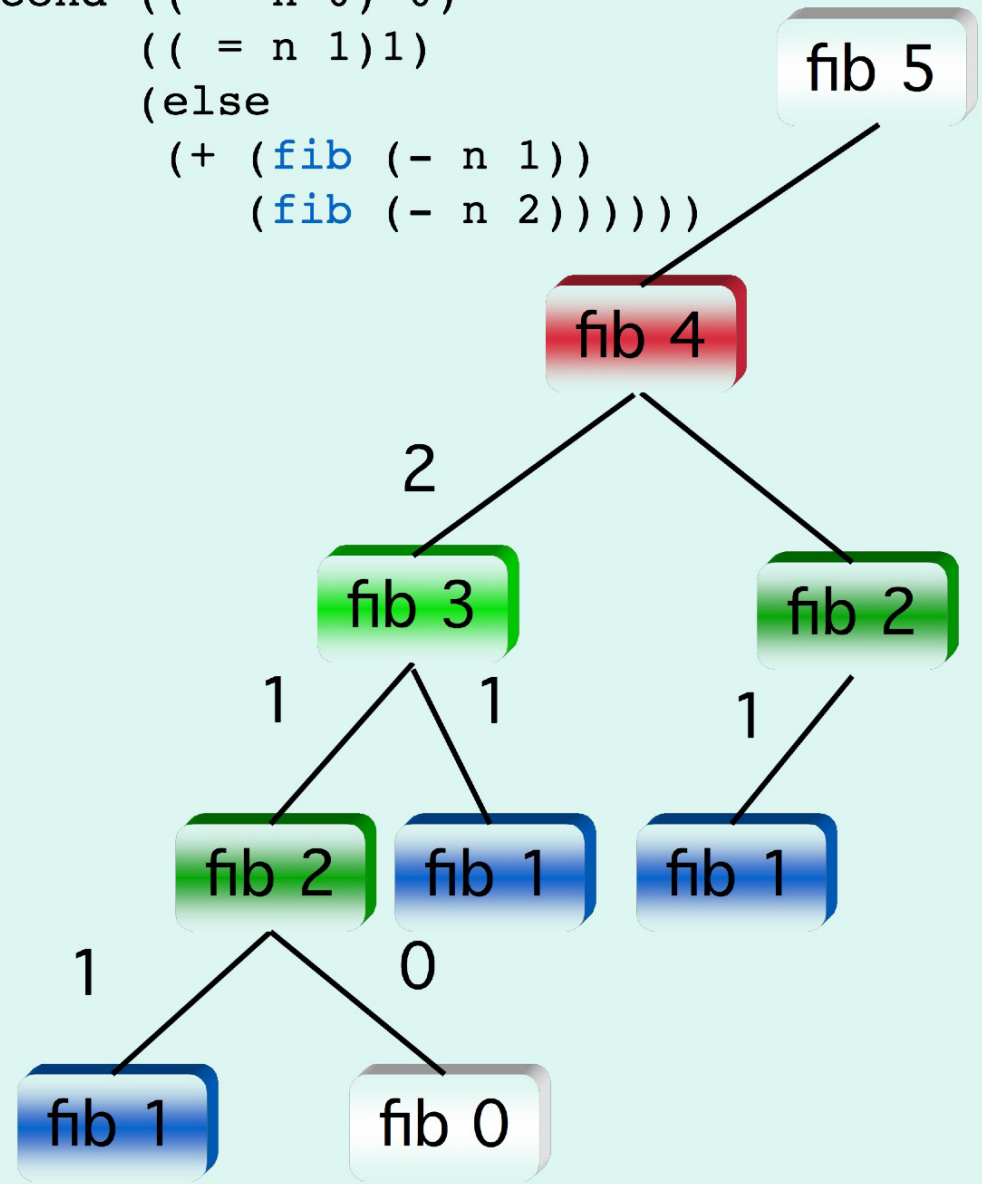
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



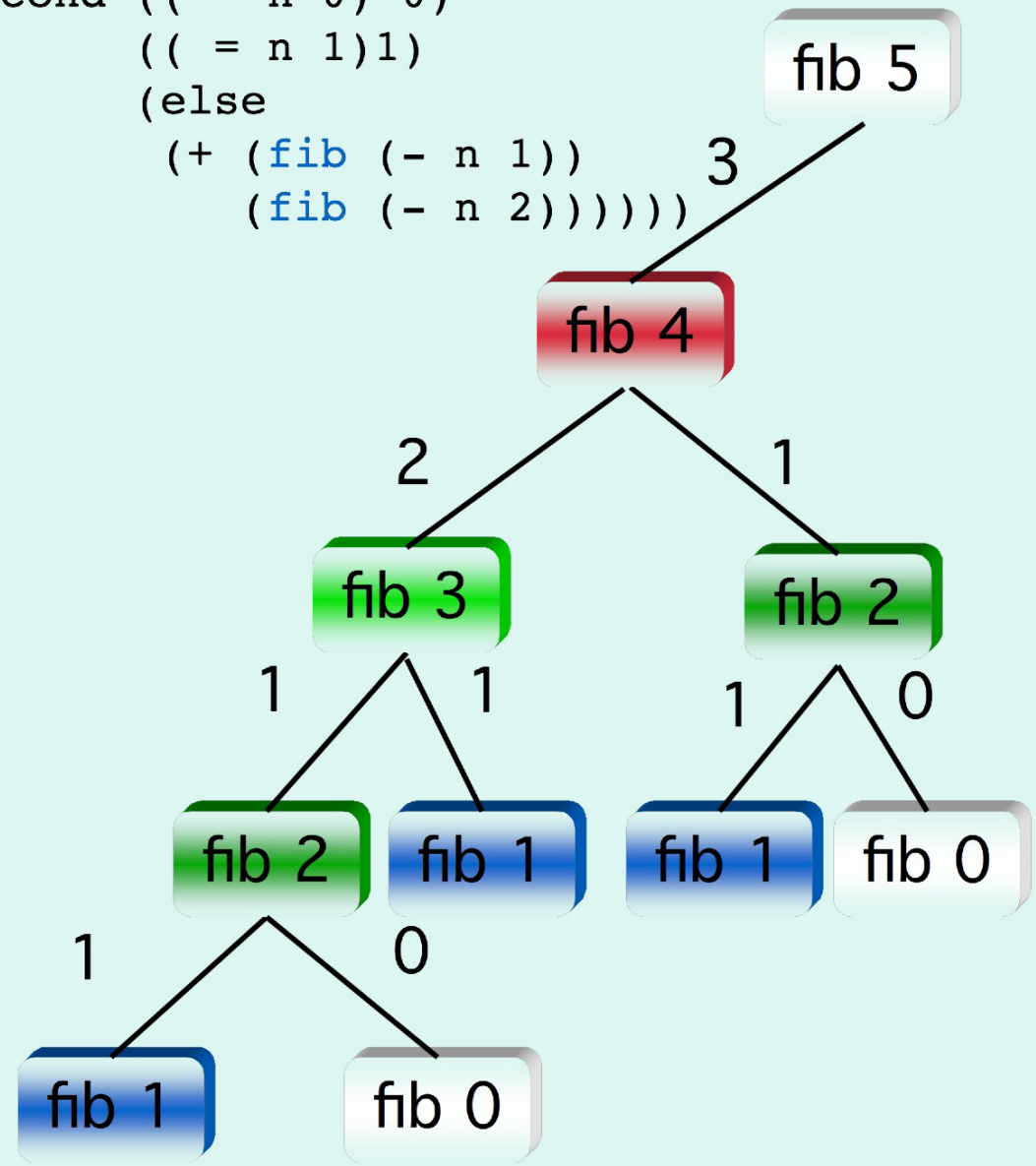
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



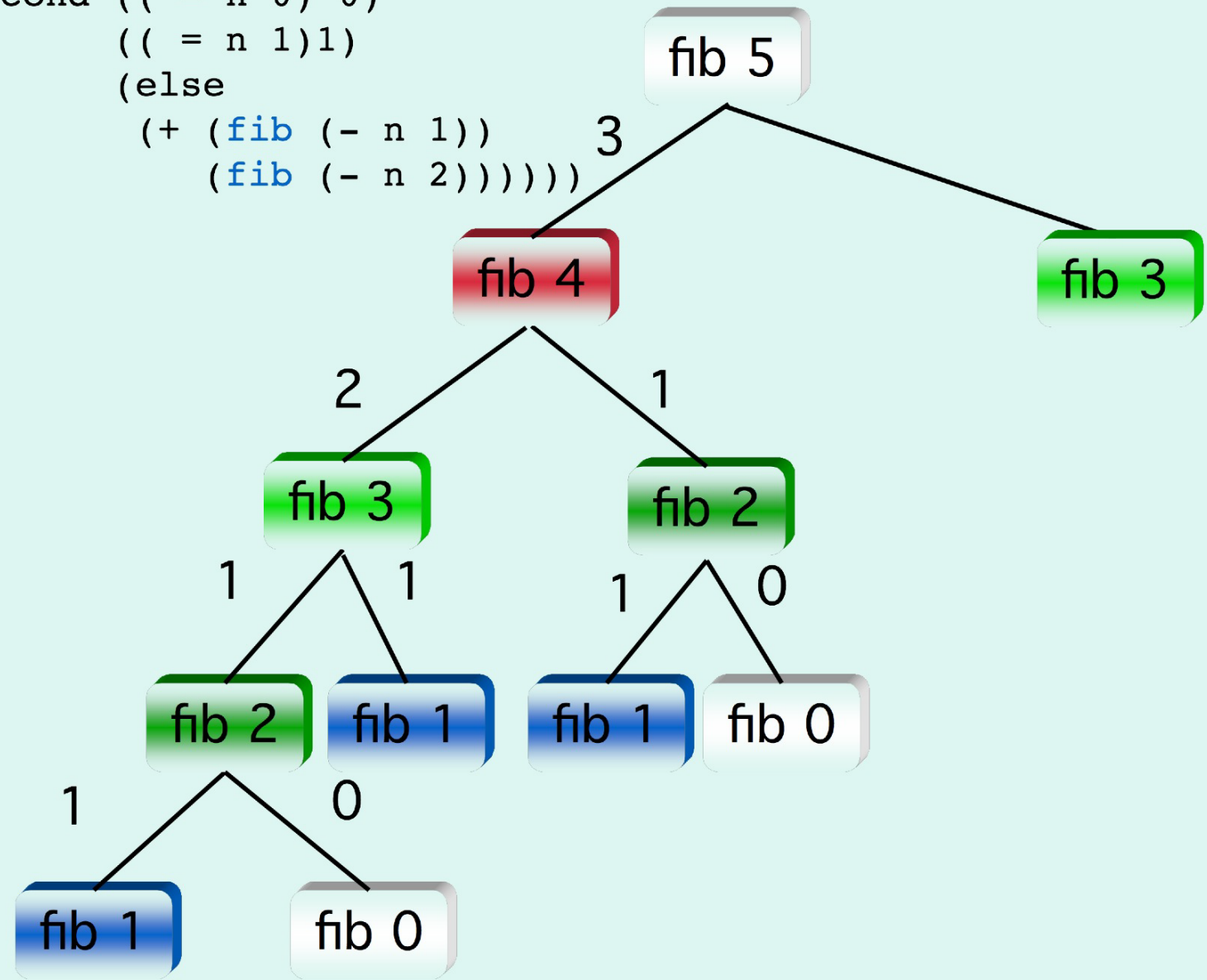
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



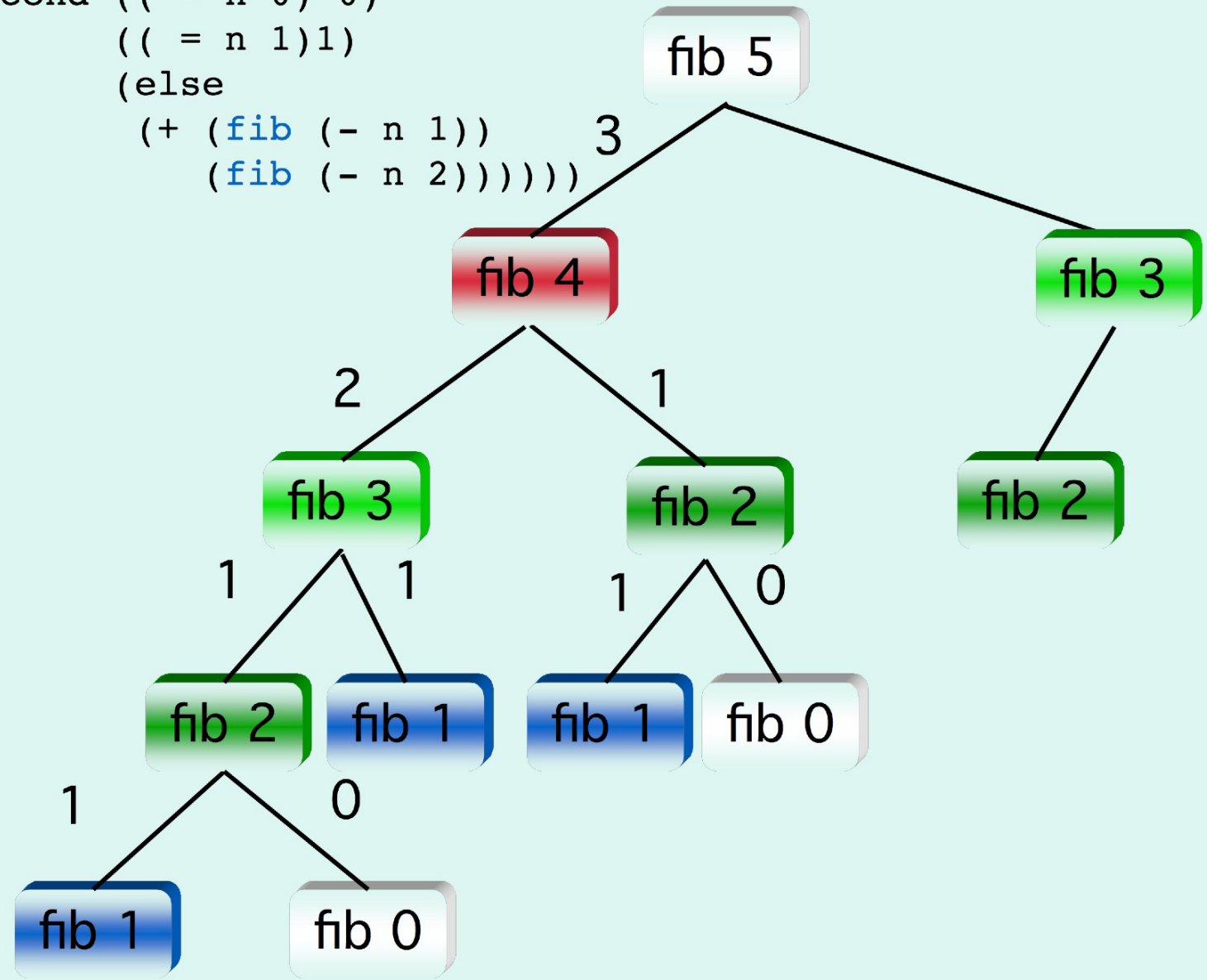
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



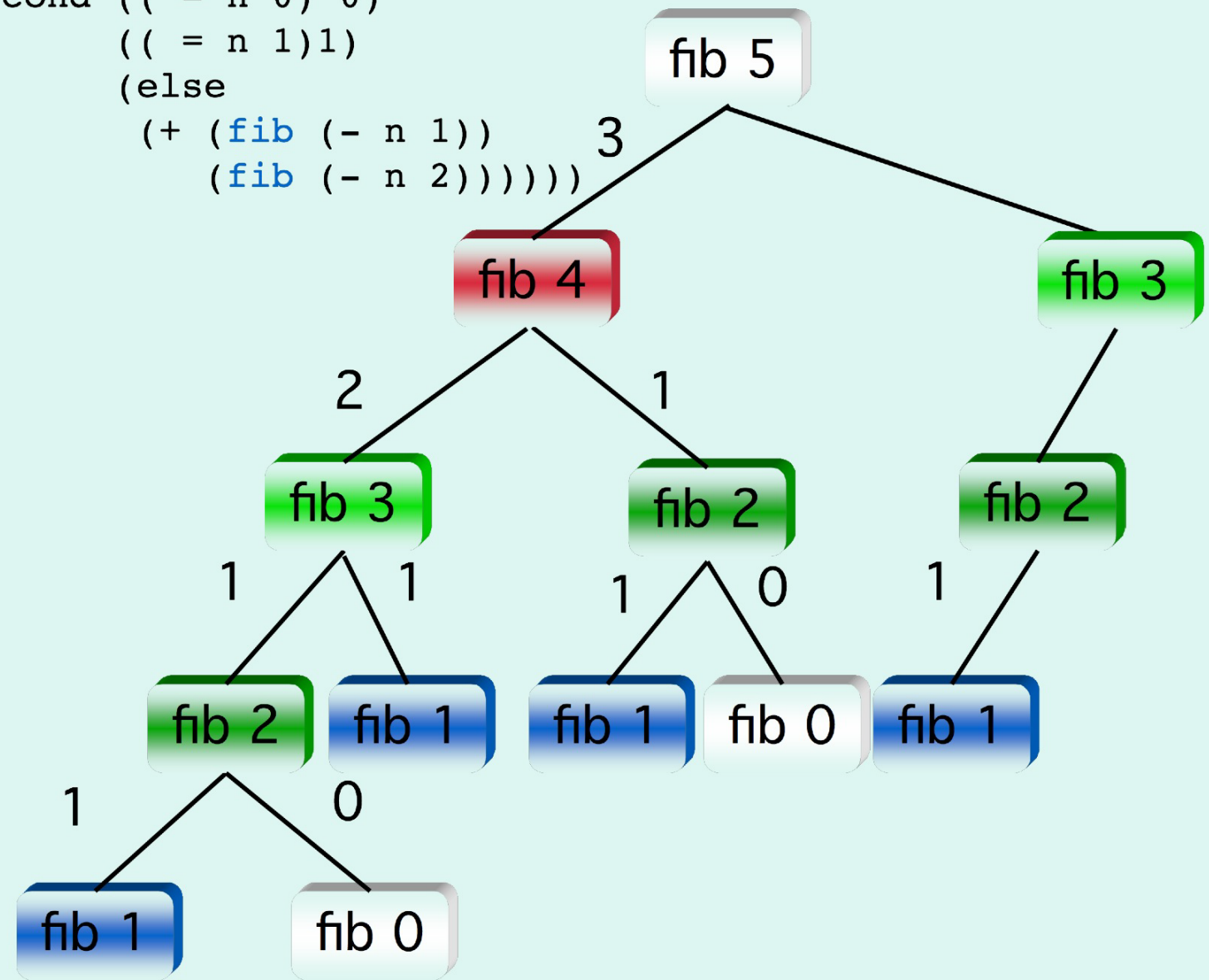
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



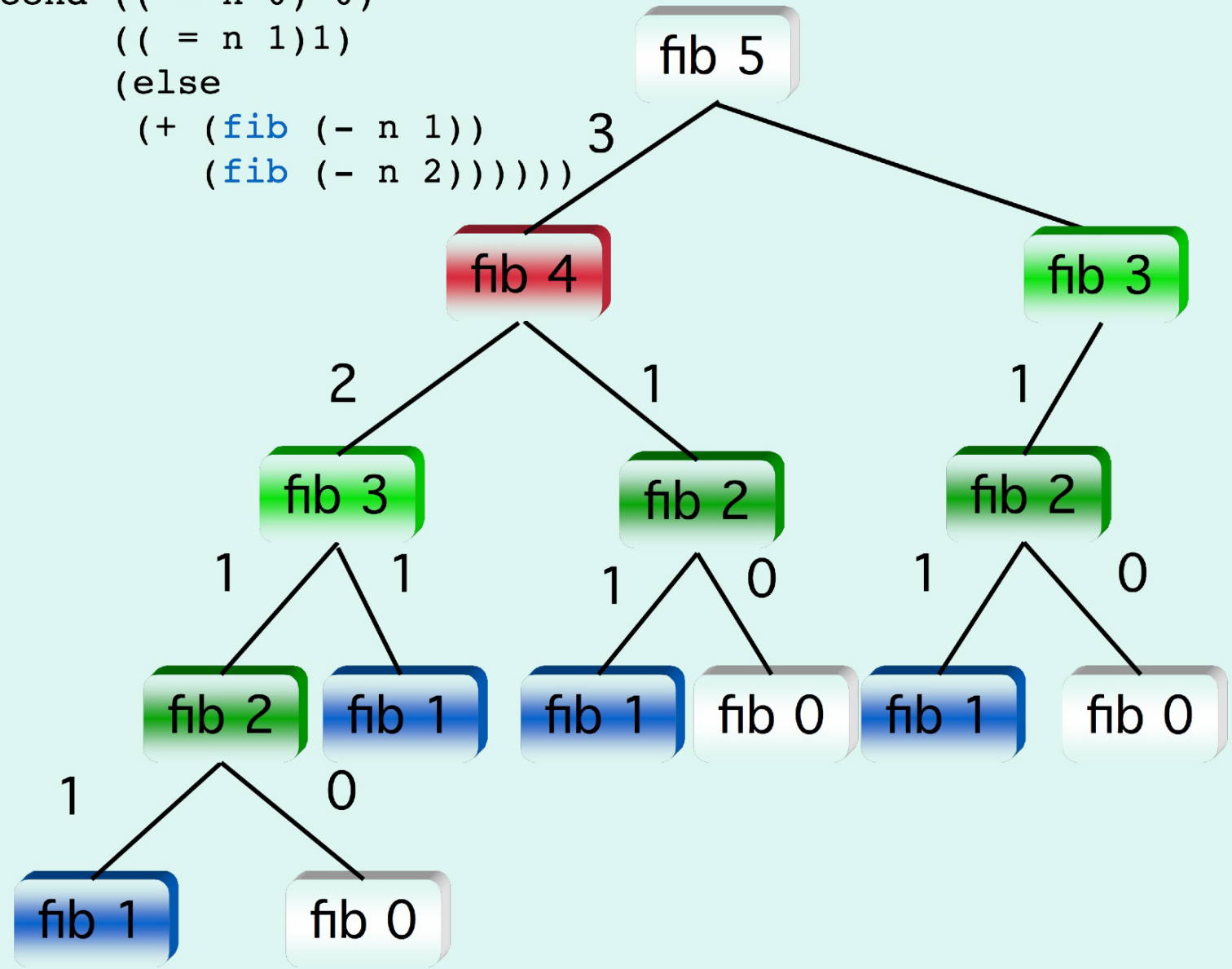
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



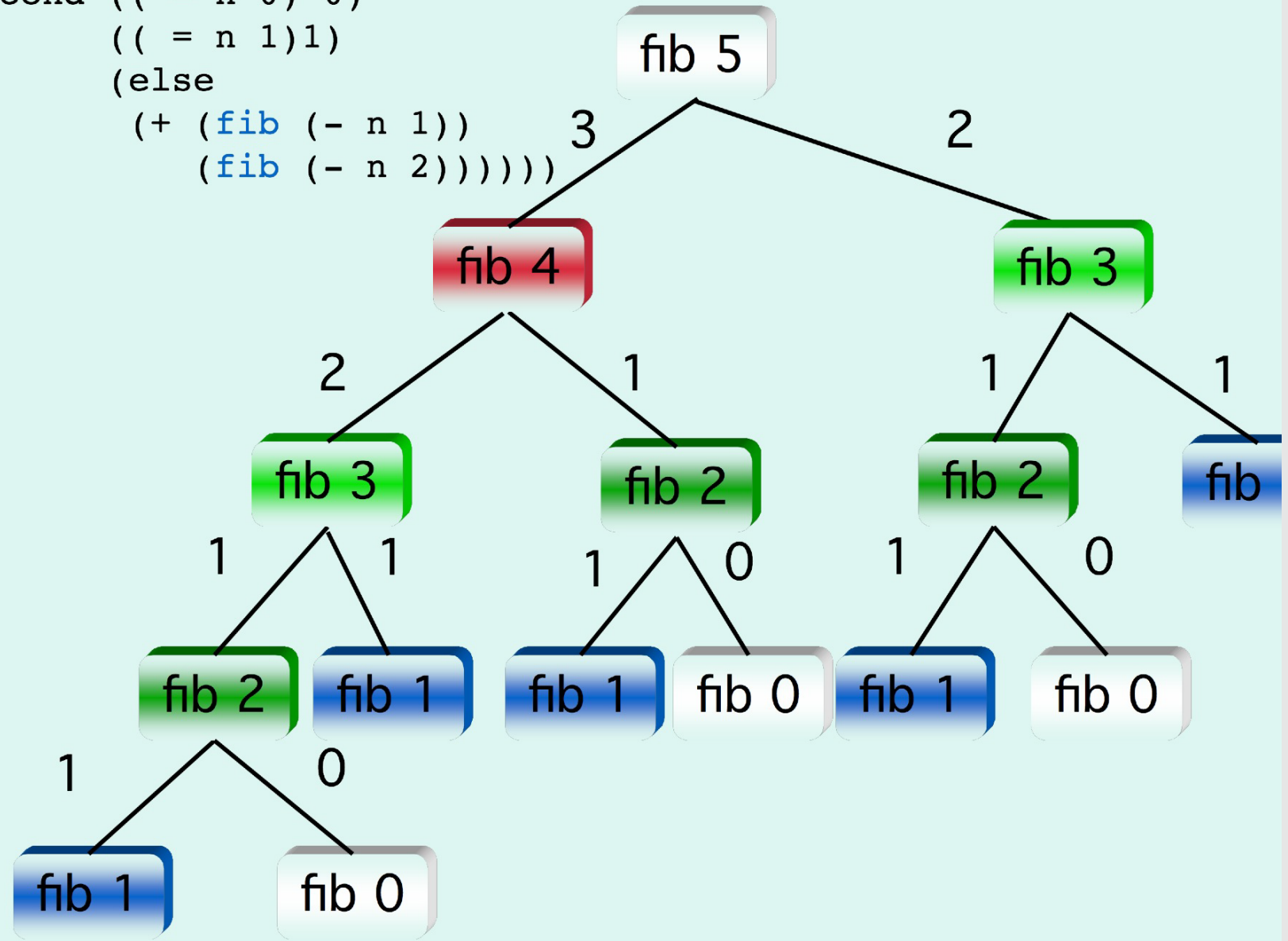
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



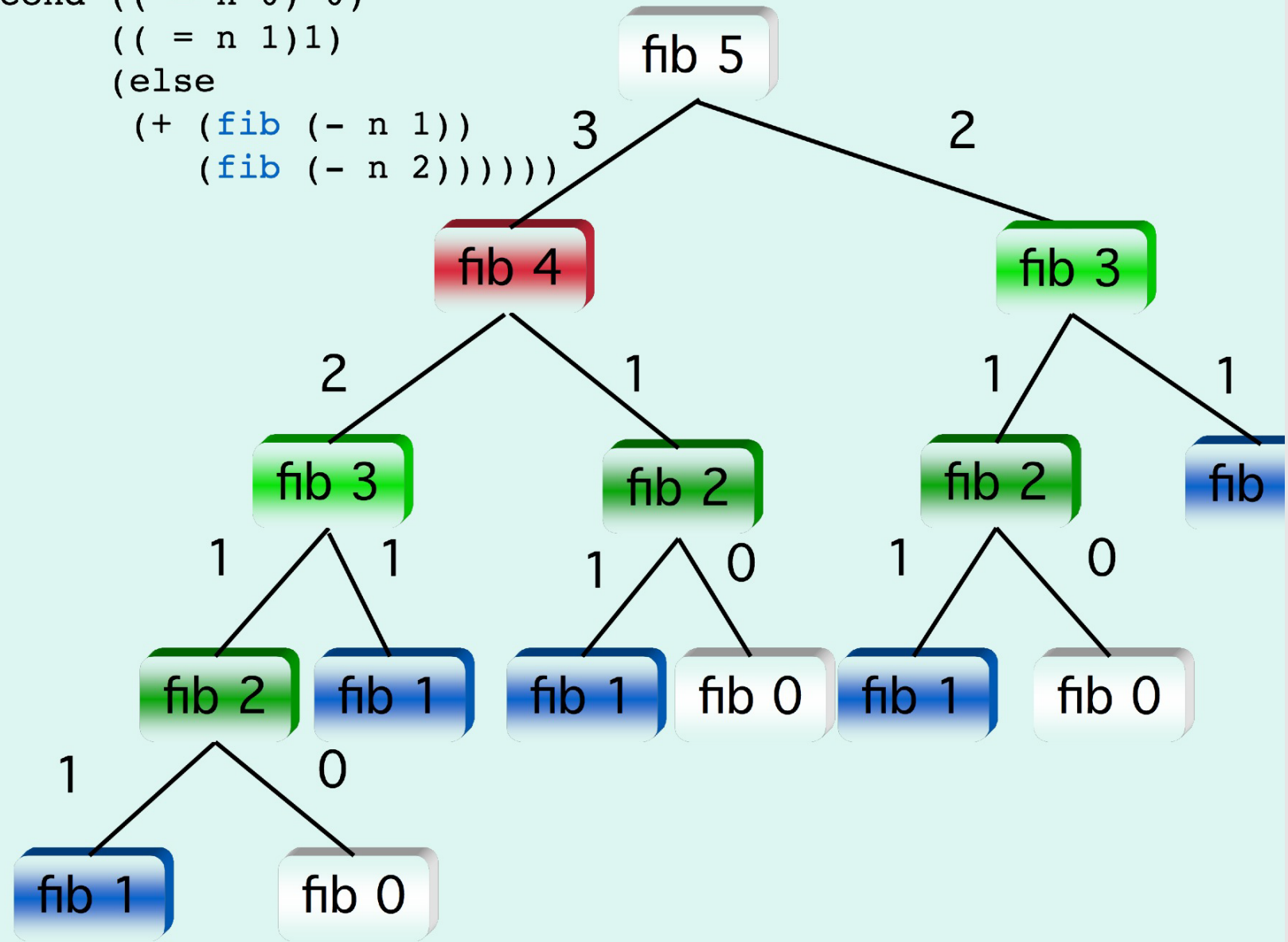
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



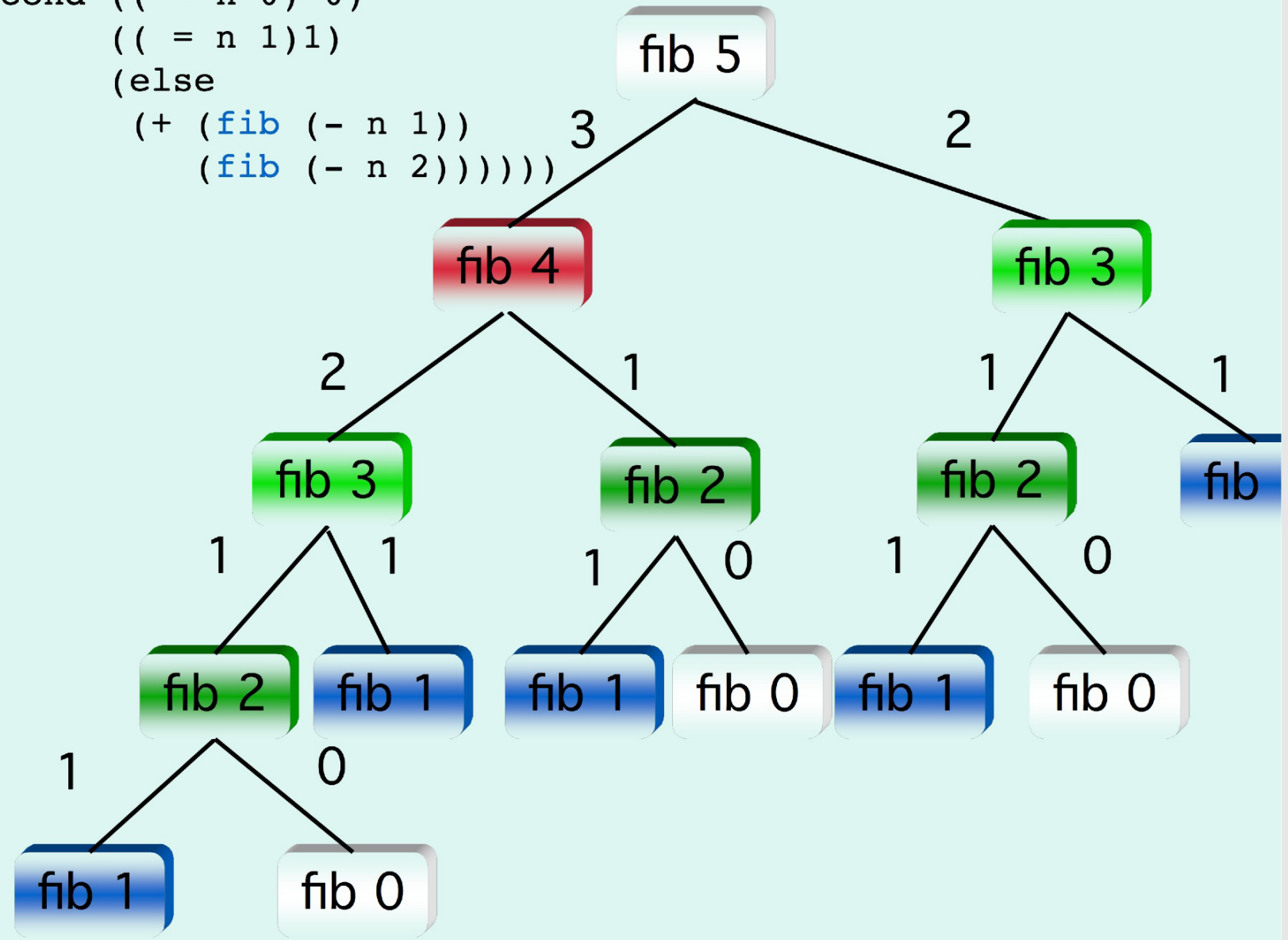
```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

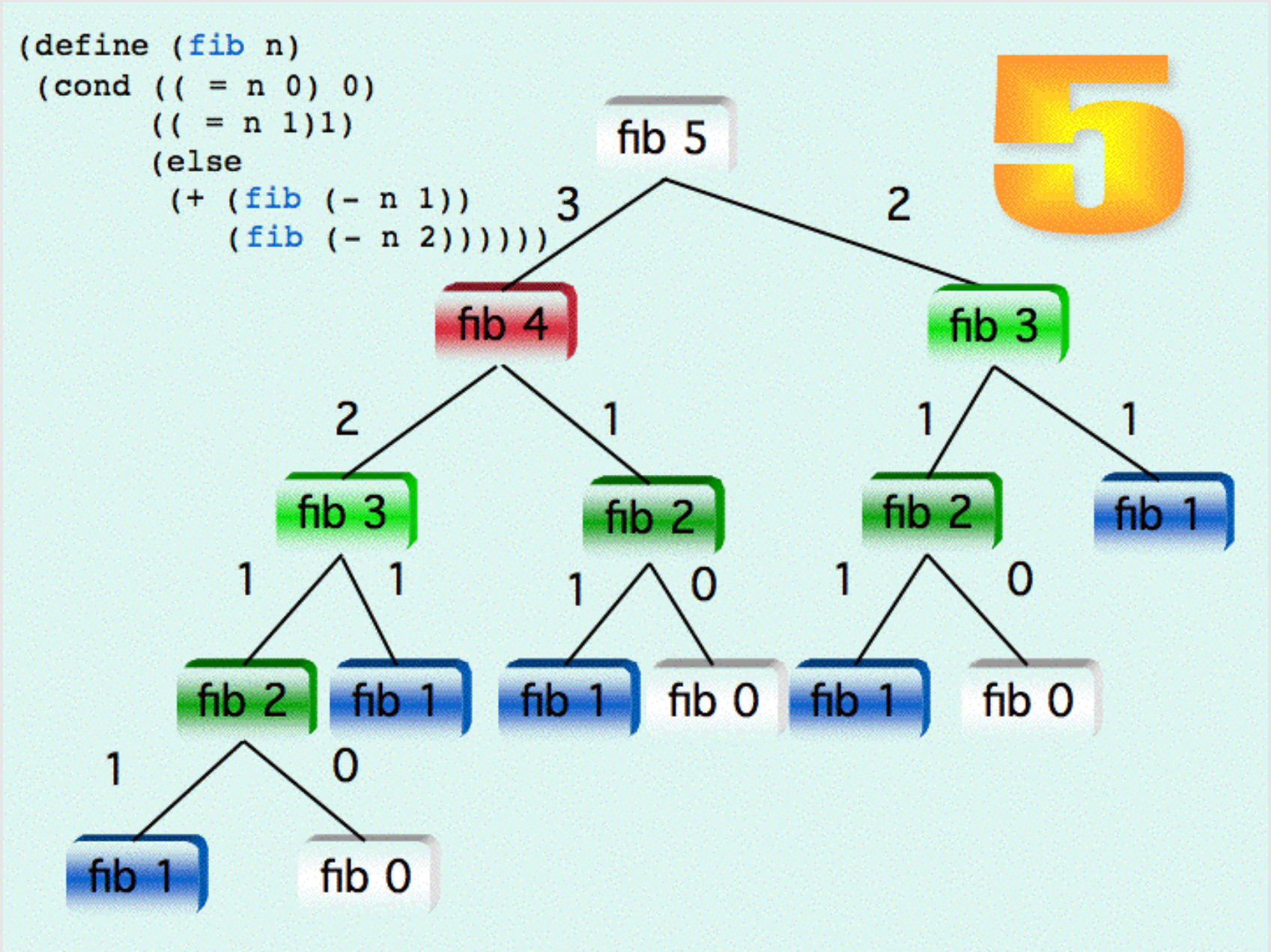


```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```



```
(define (fib n)
 (cond (( = n 0) 0)
       (( = n 1)1)
       (else
        (+ (fib (- n 1))
           (fib (- n 2))))))
```




(define (fib n)
(cond ((= n 0) 0)
((= n 1)1)
(else
(+ (fib (- n 1))
(fib (- n 2))))))
5
fib 5
3
2
fib 4
fib 3
2
1
1
1
fib 3
fib 2
fib 2
fib 1
1
1
1
0
1
0
fib 2
fib 1
fib 1
fib 0
fib 1
fib 0
1
0
fib 1
fib 0

Beobachtung

- Die baumrekursive Variante zur Berechnung der Fibonaccizahlen 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.

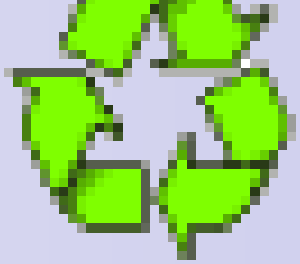
Fibonacci-Zahlen, endrekursiv

Beispiel (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))
```

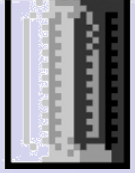
Trace: Fibonacci-Zahlen, endrekursiv

```
> (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
```



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).

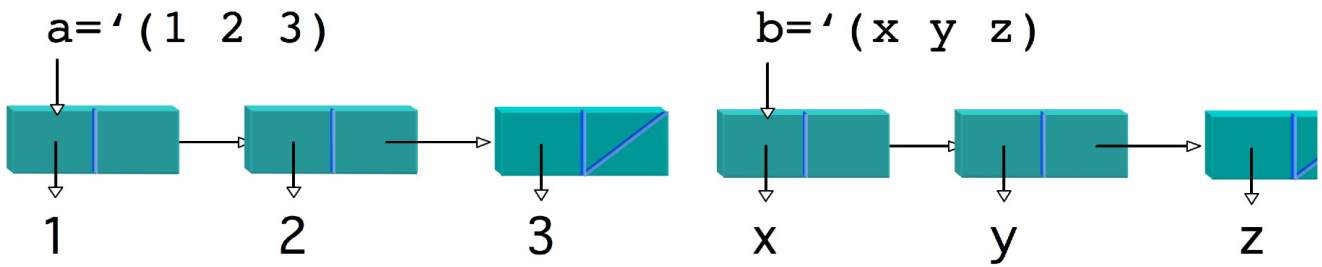


Entstehung von Datenmüll

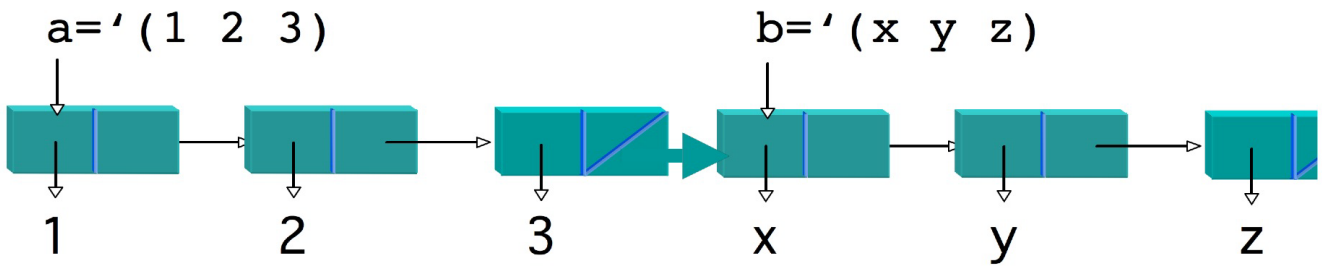
```
> (define garbage1
          "␣Alles␣Müll!")
> (define garbage2
          '(und noch mehr Muell!))
> (set! garbage1 'wegdamit)

> (set! garbage2 'auchweg)
```

Der String „garbage1“ und die Liste „garbage2“ sind jetzt nicht mehr zugreifbar und belasten unnötig den Speicher.

append a b

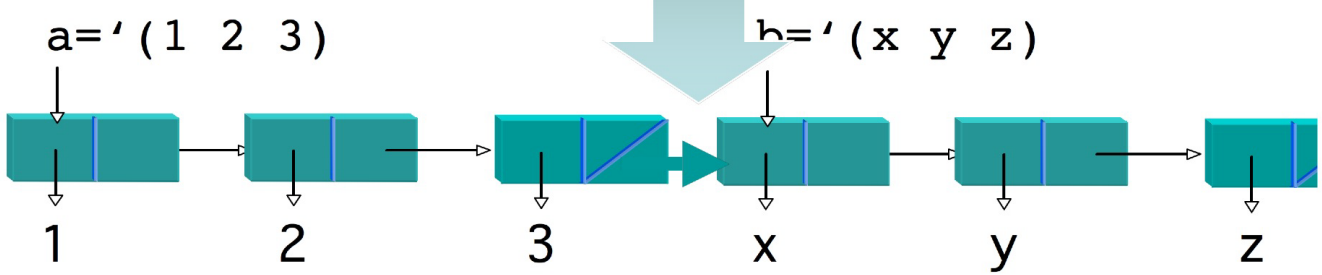


append a b

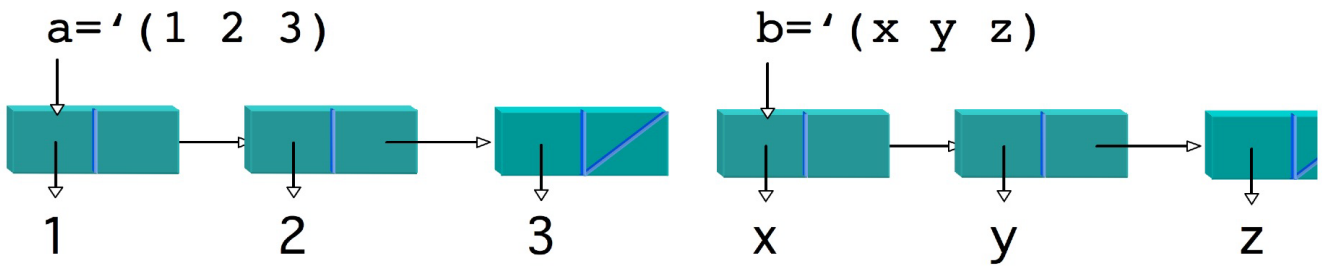


ap b

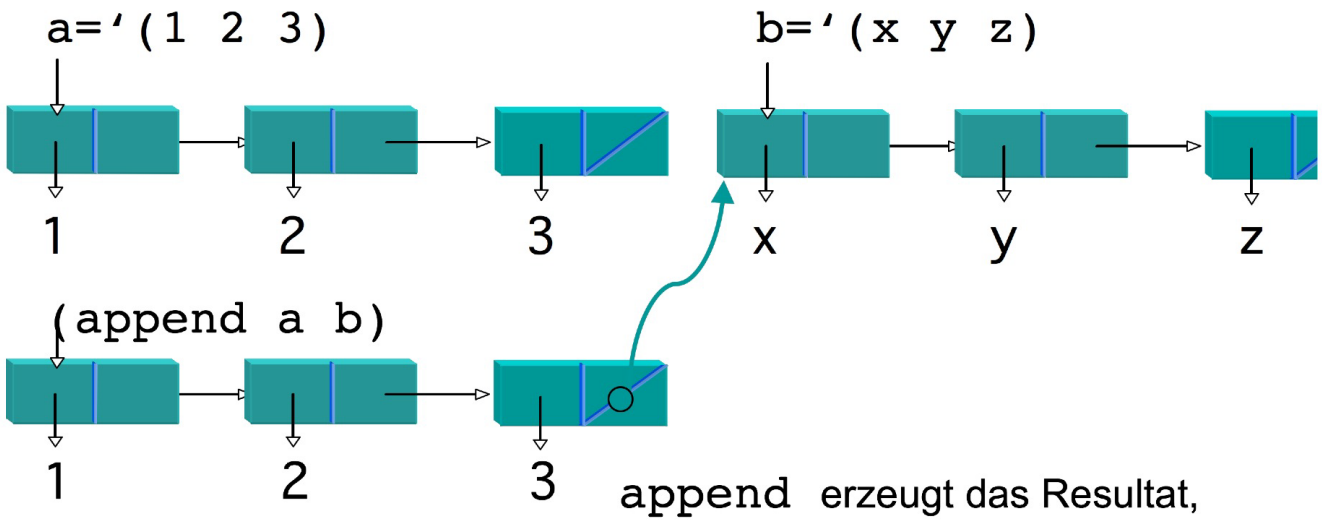
Geht nicht, da Liste a verändert wird!



append a b



append a b



`append` erzeugt das Resultat,
indem das erste Argument kopiert wird
an diese Kopie
das zweite Argument angehängt wird.

Müllintensiv: append

(Kopf | und Schwanz) $\longrightarrow$ (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
```

Aufwand: Spiegeln einer Liste mit „append“

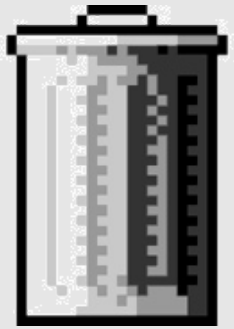
Länge	**100**	**1 000**	**10 000**	**20 000**
CPU (ms)	0	24	7233	23831
gc (ms)	0	0	5249	15935
Garbage Zellen	5 050	500 500	50 005 000	200 010 000

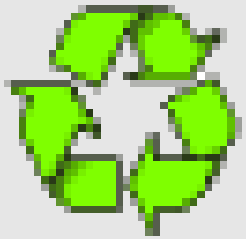
- ☞ Der Rechenaufwand und der Speicheraufwand für cons-Zellen wachsen mit dem Quadrat der Listenlänge.
- ☞ Für die rekursive Konstruktion von Listen sollte daher möglichst cons und nicht append verwendet werden.


HAUT REIN JUNGS,
AM SONNABEND
MUSS DAS STADION
WIEDER KLAR SEIN!


HAUT REIN JUNGS,
AM SONNABEND
MUSS DAS STADION
WIEDER KLAR SEIN!

Umgang mit Datenmüll



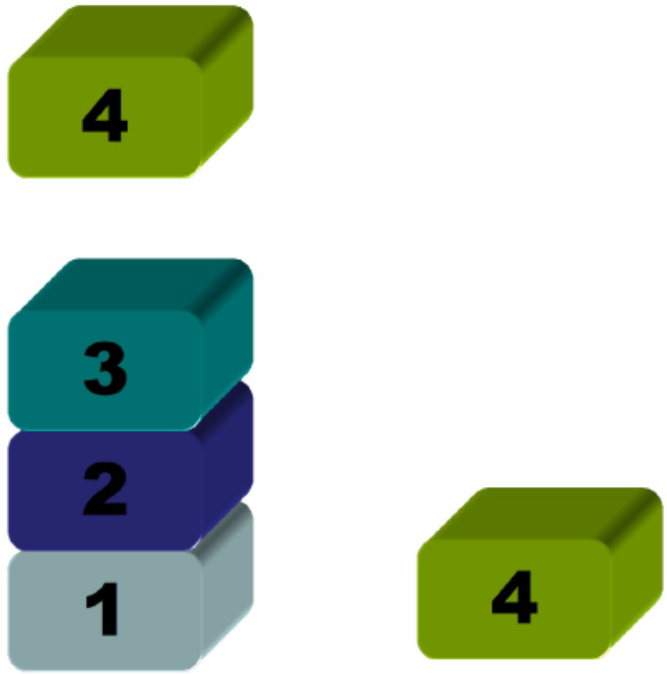


- Konstruierte Datenobjekte sind Müll, wenn es keine Referenzen mehr darauf gibt, denn die Programme können nicht mehr darauf zugreifen.
- Racket sammelt Müll automatisch ein und “recycled“ die Speicherelemente.
  Der Vorgang heißt garbage collection und ist rechenaufwendig.
- Wir sollten beim Entwurf von Funktionen den anfallenden Müll beachten und umweltbewußten Algorithmen den Vorzug geben, die wenig *garbage* erzeugen.

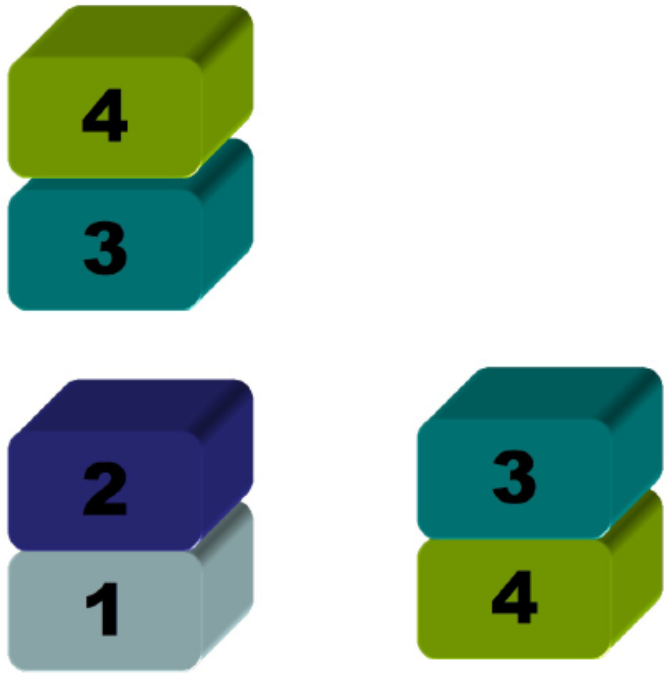
Reverse, effizient mit „cons“
Idee: Umstapeln

Reverse, effizient mit „cons“

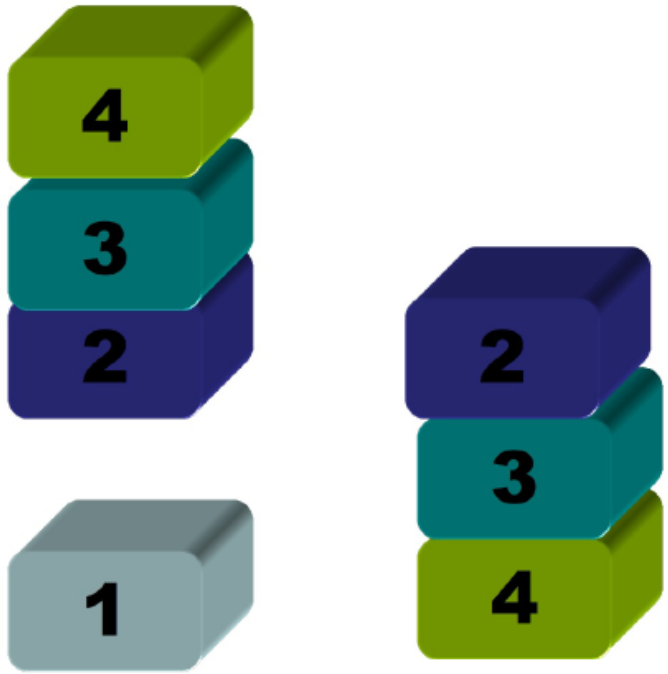
Idee: Umstapeln



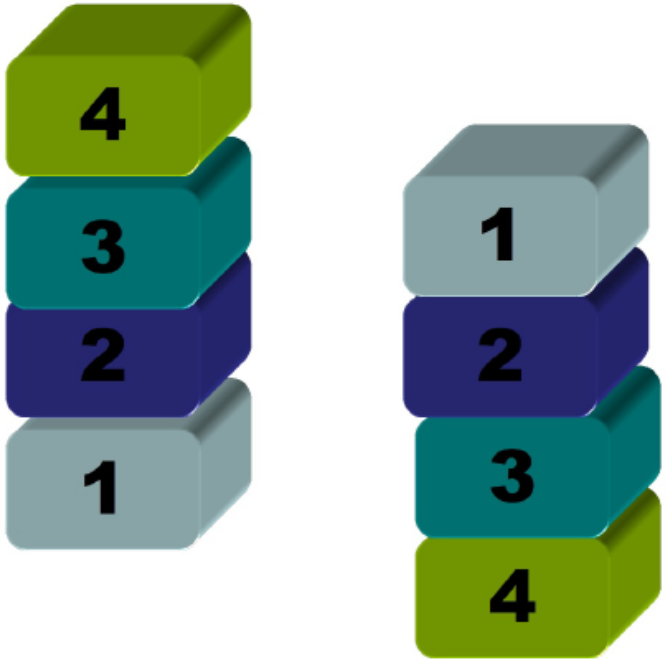
Reverse, effizient mit „cons“
Idee: Umstapeln



Reverse, effizient mit „cons“
Idee: Umstapeln



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	**1000**	**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.

Blütenbaum

Der Entwurf von Funktionen

Teil VI

Der Entwurf von Funktionen

Funktionen als Werte

M.C. Escher

Funktionen höherer Ordnung

Definition (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:

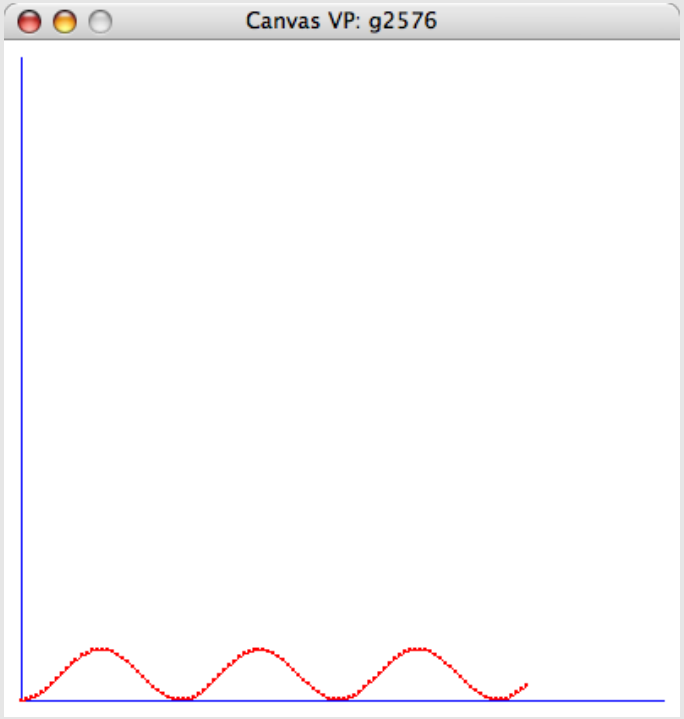
```
(require htdp/graphing)
(graph-fun <procedure> <color>)
```

- graph-fun zeichnet die Kurve der Funktion 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

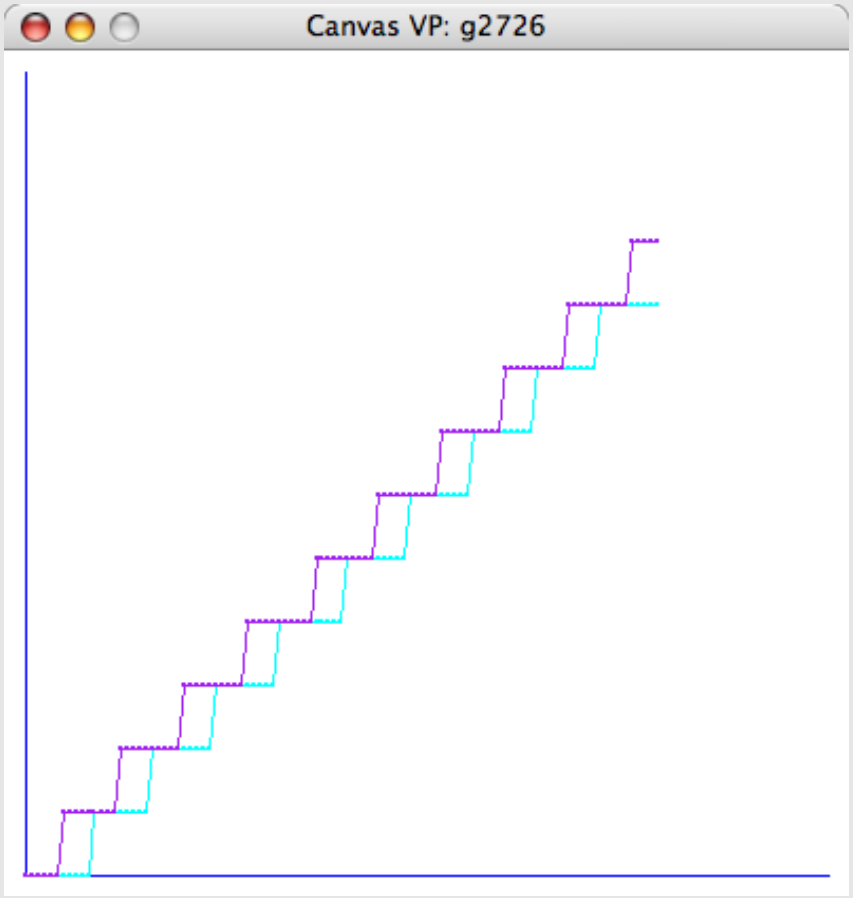

```
> ( graph-fun sqrt 'red)    ⟶    #t
```

Beispiel: Zeichnen einer anonymen Funkion



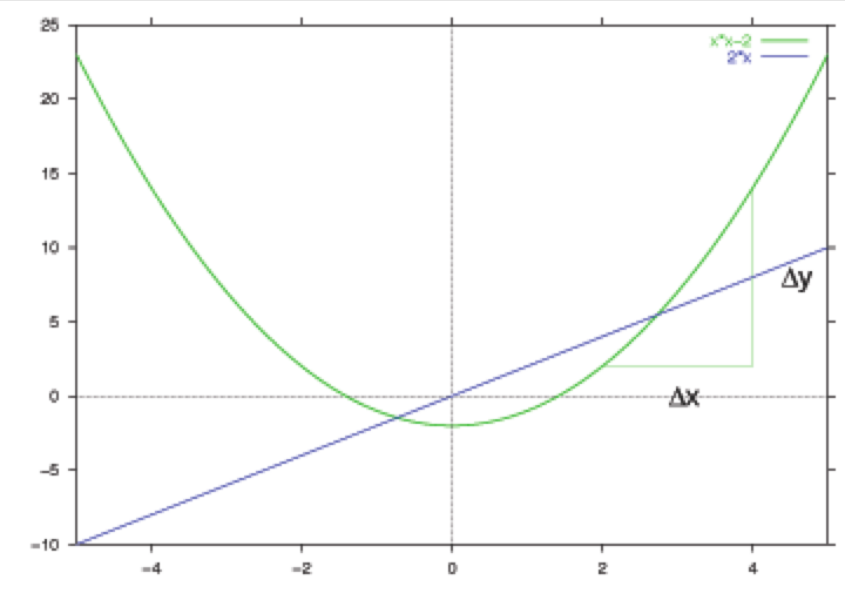

```
> ( graph-fun
    (lambda (x)    ; sin²x
      (* (sin x)
         (sin x)))
    'red)    ⟶    #t
```

Beispiel: round und truncate




```
(require htdp/graphing)
> ( graph-fun truncate 'cyan)   ⟶ #t
> ( graph-fun round 'purple)    ⟶ #t
```

Numerische Ableitung einer Funktion



Beispiel (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)
-1.000000082740371e-05
```

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)                  ⟶ 6.00000999951316
> ((deriv (lambda (x)
            (* -4 x))) 5)
                                      ⟶ -3.9999999998485687
```

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. ((deriv **sin**) 0), berechne die Ableitung der Sinus-Funktion für $x = 0$.

Der Baukasten-Ansatz

Idiome

- Die Idiome gehören zur Pragmatik der Programmiersprache – wie wird die Sprache verwendet?

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.

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 Beherrschen eines Programmierstils und einer Programmiersprache gehört es daher auch, daß man diese Standardlösungen für typische Aufgaben kennenlernt.

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 Beherrschen 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.

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.

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.

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.

Funktionsapplikation (apply)

Problem (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.

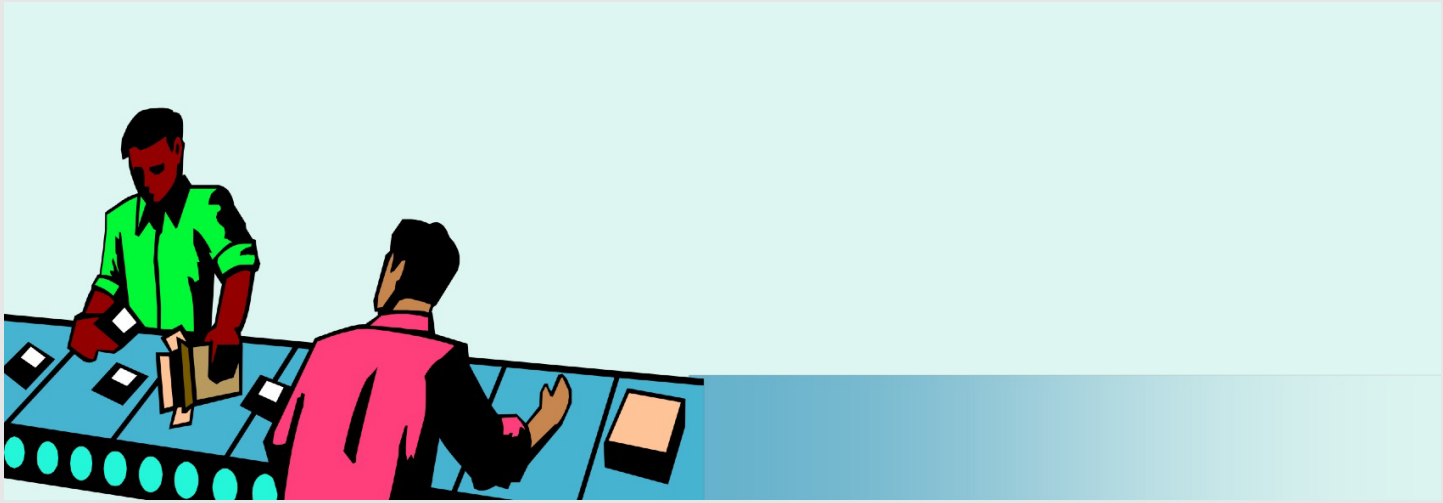
```
(flat-contract number?)

(flat-named-contract type-name predicate?)
```

Implementation von eval

Wenn Ihr Scheme-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
```

Abbilden (mapping)



Abbilden (mapping)

Problem

Eine Folge oder Menge von Werten soll in einheitlicher Weise transformiert werden, so daß die Anzahl und Anordnung der Elemente erhalten bleiben.

Lösung

- *Repräsentation der Folge oder Menge als Liste*
- *und Transformation der Liste mit einer „mapping-function“: map.*

Abbilden (mapping)

Problem

Eine Folge oder Menge von Werten soll in einheitlicher Weise transformiert werden, so daß die Anzahl und Anordnung der Elemente erhalten bleiben.

Lösung

- *Repräsentation der Folge oder Menge als Liste*
- *und Transformation der Liste mit einer „mapping-function“: map.*

$$
\begin{array}{lccccccc}
x: & (& 0 & 1 & 2 & 3 & 4 & 5 & \cdots &) \\
 & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & f(x) = 2^x & \\
2^x & (& 1 & 2 & 4 & 8 & 16 & 32 & \cdots &)
\end{array}
$$

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))
```

Filtern

Problem

Aus Folge oder Menge von Werten sollen diejenigen Werte ausgewählt werden, die eine bestimmte Bedingung erfüllen.

Lösung

- *Repräsentation der Folge oder Menge als Liste*
- *und Filtern der Liste mit einer Filter-Funktion.*

$$
\begin{array}{ccccccccc}
(& 2 & 3 & 4 & 5 & 6 & 7 & \cdots &) \\
 & \downarrow & \downarrow & \perp & \downarrow & \perp & \downarrow & \textit{Primzahl}? & \\
(& 2 & 3 & & 5 & & 7 & \cdots &)
\end{array}
$$

- 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) '()] ; fertig
        [(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
```

Algebraische Eigenschaften von filter

Für Terme mit filter gelten die folgenden Zusicherungen (ohne Beweis):

Satz (Eigenschaften von filter)

```
(filter p (append xs ys))
  = (append (filter p xs)(filter p ys))
(filter p (filter q xs))
  = (filter q (filter p xs))
```

- *filter ist also distributiv bezüglich der Konkatenation von Listen.*
- *Mehrfaches Filtern ist unabhängig von der Reihenfolge, solange unsere Funktionen referentiell transparent sind!*

Falten (folding)

Problem

Die Elemente einer Folge von Werten sollen

- *paarweise mit demselben Operator verknüpft*
- *und zu einem Wert reduziert werden.*

Lösung

- *Repräsentation der Folge als Liste*
- *und Reduzieren der Liste mit einer Faltungsfunktion* *foldl* *oder* *foldr* *.*

$$
\begin{array}{cccccccccc}
(& 0 & & 1 & & 2 & & 3 & \cdots &) \\
 & \curvearrowright & & \curvearrowright & & \curvearrowright & & \curvearrowright & & \\
(\in & +0 & & +1 & & +2 & & +3 & \cdots &)
\end{array}
$$

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.

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.

Allgemeinrekursive Reduktion

```
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.

Aufzählen einer Folge

Problem

Aufzählen und Sammeln: Gegeben sei eine Funktion f der natürlichen Zahlen.
Gesucht: Die Folge der n Bilder $\{f(0), f(1) \ldots f(n-1)\}$.

Lösung

Die Funktion (build−list n f) berechnet die Liste der n Funktionsresultate f (0), f (1), . . . f (n−1).

```
> (build-list 5 (lambda (x) (* x x)))
  ⟶ '(0 1 4 9 16); Aufzählen der Quadrate
```

Wiederholung (Iteration)

Problem

Iterative Anwendung einer Funktion auf das Resultat der letzten Berechnung, beispielsweise zur Berechnung von Folgen und Reihen.

Lösung

Verwendung des Bausteins ***iterate*** *oder Rekursion.*

$$\text{powers\_of\_two} = (1, \overset{(2*)}{\curvearrowright} 2, \overset{(2*)}{\curvearrowright} 4, \overset{(2*)}{\curvearrowright} 8, \overset{(2*)}{\curvearrowright} 16, \ldots)$$

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))
```

✎ 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.

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 höhererer Ordnung zur Kontrollabstraktion, die den Ablauf steuern:

Applikation: apply

Abbilden: 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 (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)
```

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))
```

Wiederholung II

Problem

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

Verwendung des Bausteins iter−until oder Rekursion.

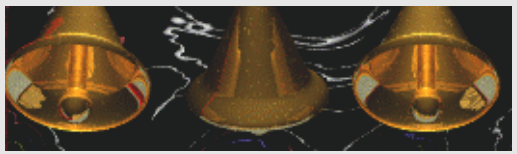
$$\text{powers\_of\_two} = (1, \overset{(2*)}{\curvearrowright} 2, \overset{(2*)}{\curvearrowright} 4, \overset{(2*)}{\curvearrowright} 8, \overset{(2*)}{\curvearrowright} 16, \ldots)$$

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)  ⟶
```

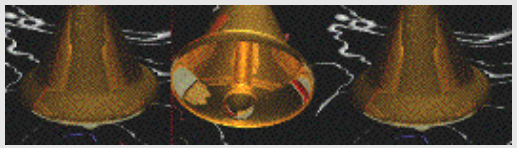
Rekursive Implementation von iter-until

```
(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)  ⟶
```



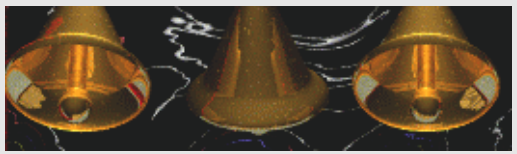
Rekursive Implementation von iter-until

```
(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)  ⟶
```



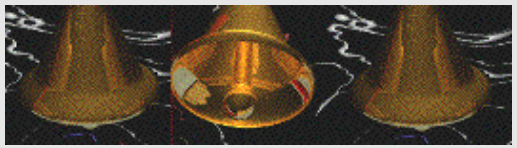
Rekursive Implementation von iter-until

```
(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)  ⟶
```



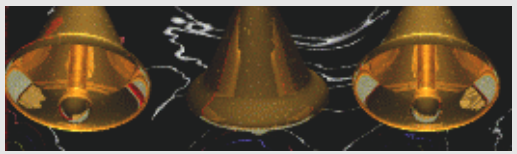
Rekursive Implementation von iter-until

```
(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)  ⟶
```



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 höhererer Ordnung zur Kontrollabstraktion, die den Ablauf steuern:

- **Applikation:** apply
- **Abbilden:** 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.

Modularer Entwurf

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,
- 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.

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> ...)
```

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)
```

Formale Spezifikation und modularer Entwurf

Spezifikation: Quadratwurzel einer Zahl

Beispiel

Wir programmieren eine Funktion (msqrt x), die die Quadratwurzel von x berechnet:
Die mathematische Spezifikation für msqrt:

$$\text{msqrt}(x) \geq 0 \wedge \text{msqrt}(x)^2 = x \tag{1}$$

oder schwächer:

$$\text{msqrt}(x) \geq 0 \wedge |\text{msqrt}(x)^2 - x| < \epsilon, \epsilon > 0 \tag{2}$$

Spezifikation und Implementation

Definition (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.

Wir fordern:

$$\begin{aligned} (\text{msqrt } x) &\geq 0 \\ |(\text{msqrt } x)^2 - x| &< \text{eps} \\ \text{eps} &> 0 \end{aligned}$$

Beispielrechnung: eps = 0.0001, x = 2

```
1.4141 * 1.4141 = 1.99967881
1.4142 * 1.4142 = 1.99996164
1.4143 * 1.4143 = 2.00024449

(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:

$$\begin{array}{lcll} y_0 & & & = 2 \\ y_1 & = & (2 + 2/2)/2 & = 1.5 \\ y_2 & = & (1.5 + 2/1.5)/2 & = 1.4167 \\ y_3 & = & (1.4167 + 2/1.4167)/2 & = 1.4142157 \\ & & \vdots & \vdots \end{array}$$

Teilaufgaben bei der Wurzelberechnung:

Iterationsregel: $(\text{improve } y) \rightarrow (y + \frac{x}{y})/2$

Abbruchbedingung:

$(\text{good-enough? } y) \rightarrow \text{ abs } (y^2 - x) < \text{eps};$
$\text{eps} = 0.0003$

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 −>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:

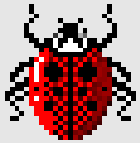
```
(define (good? x wurzelx)
  (let ([eps 0.00001])
    (< (abs (- (sqr wurzelx) x))
       eps)))
```

Der Vertrag

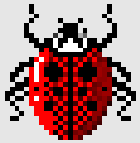
```
(define/contract ; der Vertrag
  msqrt3
  (->d ([x (>=/c 0)])
       (r (and/c
           number?
           (lambda (wu) (good? wu x)) )))
  (lambda(x)
    (let ([good-enough?
           (lambda (wurzelx) ; x an good? binden
             (good? x wurzelx))]
          [improve
           (lambda (wurzelx)
                   (/ (+ wurzelx (/ x wurzelx))
                      2))])
      (iter-until
       improve good-enough? x))))
> (msqrt3 2)  ⟶ 1 169/408
> (exact->inexact (msqrt3 2))  ⟶ 1.4142156862745
```

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 Teilfunktionen zusammengesetzt:

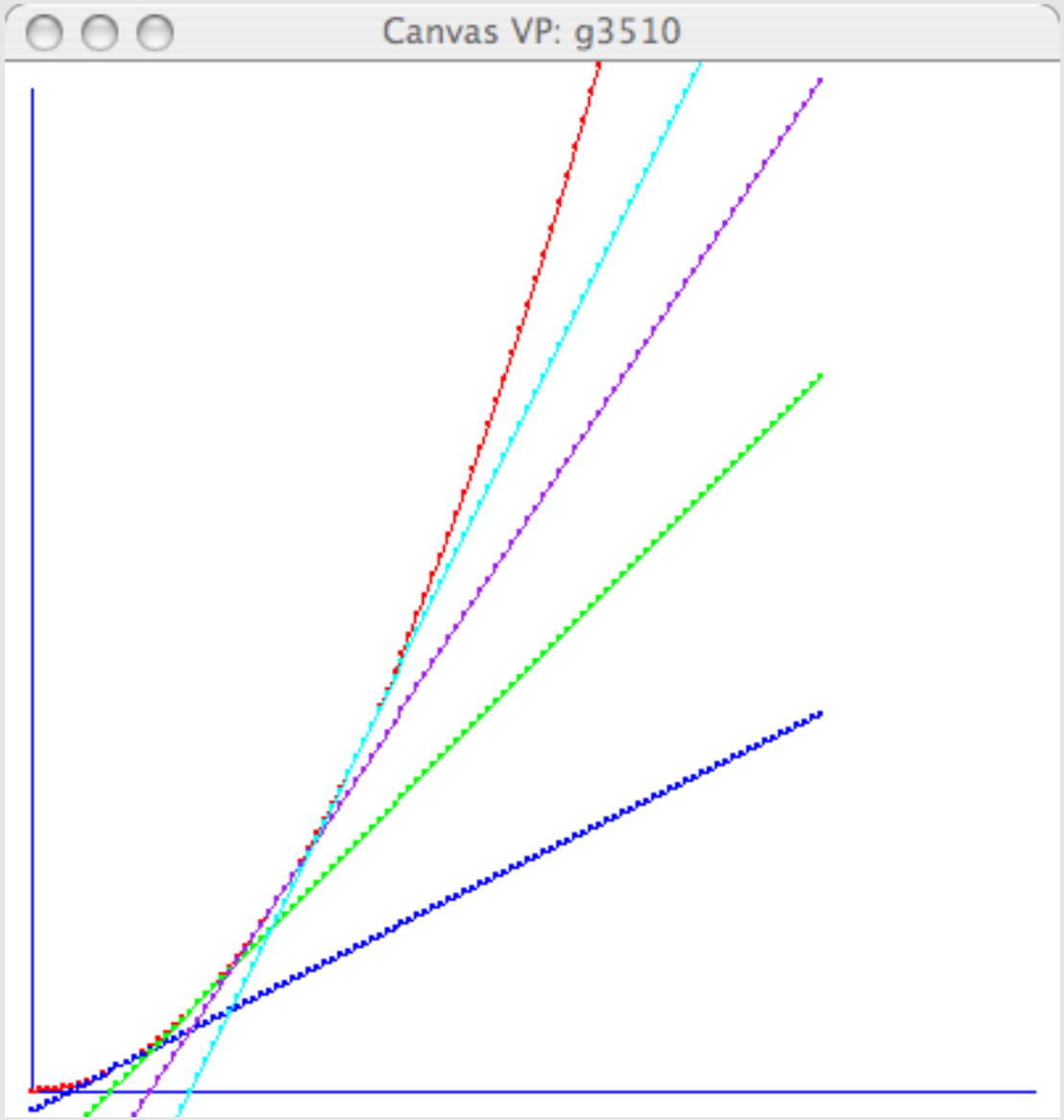
- good−enough? und improve werden nur lokal benötigt.
  Wir haben diese Hilfsfunktionen daher lokal definiert.
- Die lokalen definierten Funktionen können auf den Parameter x zugreifen, so daß sie nur einen Parameter benötigen.
- Dieser Programmierstil heißt *modular*, denn wir konstruieren komplexere Einheiten aus einfacheren Moduln.
- Wenn die Moduln gut gewählt wurden, sind solche Programme leicht zu verstehen und zu verändern.
- Eine Stärke dieser Vorgehensweise ist es auch, daß Moduln leicht wiederverwendet werden können.

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, f(x)), 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 ($f'(x_n) = 0$).

$f(x) = 0.25 \times x^2$ und einige Tangenten



Allgemeines Newton-Verfahren:
Wurzeln einer Funktion

Beispiel

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.

$$\text{Mit} \quad f(x) = x^2 - a \quad \text{und} \quad f'(x) = 2x$$

$$y_n - \frac{f(y_n)}{f'(y_n)} = y_n - \frac{y_n^2 - a}{2y_n} = (y_n + a/y_n)/2$$

```
(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))))
```

Beispiele

```
(define (n-sqrt x) ;  √x
  ((newton    ; solving: y*y-x = 0.
     (lambda (y)
       (- (square y) x)))
     x))
(define (cubrt x) ;     ∛x
  ((newton    ; solving y*y*y-x=0.
     (lambda (y) (- (expt y 3) x)))
     x))
(define (root x n);     ⁿ√x
  ;;; the nth root of x
  ((newton
     (lambda (y) (- (expt y n) x)))
     x))
```

Testfälle

```
> (n-sqrt 4)                  ⟶  2.0000000944921563
> (cubrt 27)                  ⟶  3.000000020413817
> (cubrt
     (expt 1.5 3))            ⟶  1.5000006487422155
```

Closures: Funktionale Abschlüsse

Teil VII

Funktionale Abschluesse

Grundlagen der funktionalen Programmierung

Das Curry-Verfahren

Definition (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 [**?**] 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
```

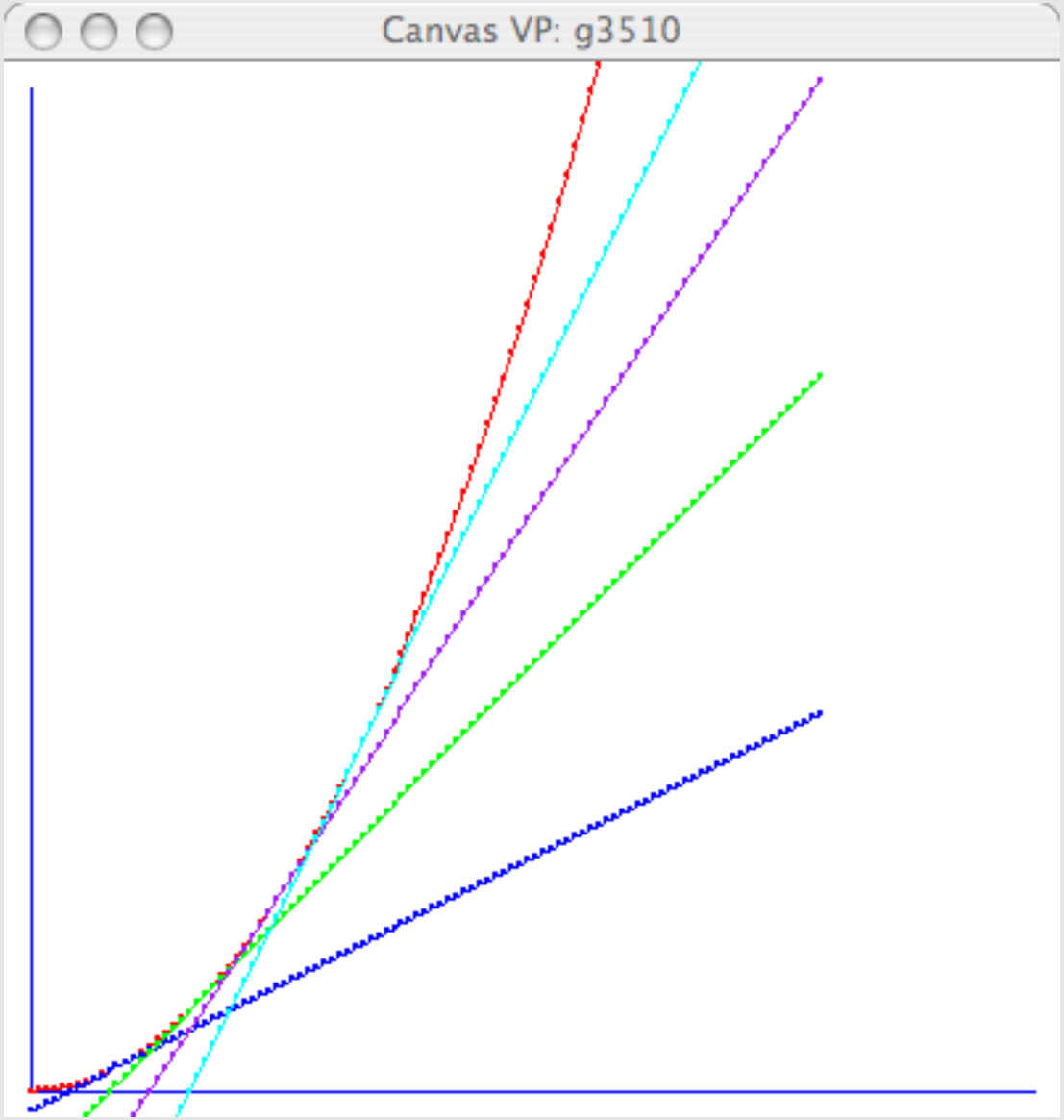
Wir abstrahieren wieder strukturgleiche Terme mit der curry-Funktion und bilden die Funktion (curry **max** 3)

```
(define (curry f x)
  ;; curry left arg to the function f
        (lambda (y) (f x y)))
> ((curry max 3) 10)  ⟶ 10
```

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)
```

$f(x) = 0.25 \times x^2$ und einige Tangenten



Zeichnen einer Liste von Kurven

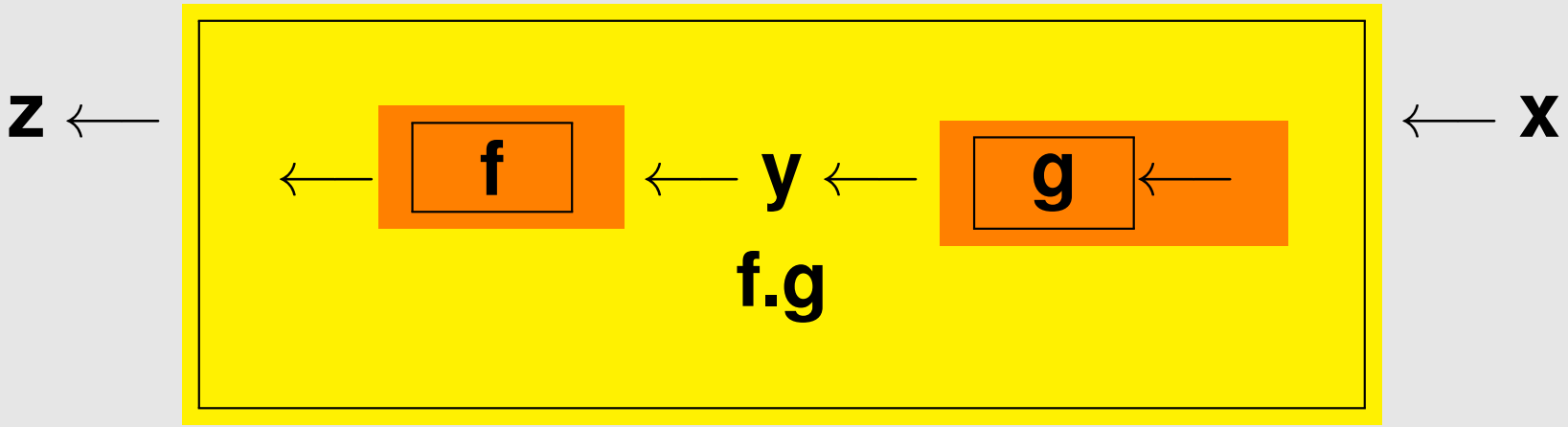
```
(define (vieleTangenten2 f)
 ;zeichne f und mehrere Tangenten
 (graph-fun f 'red)
 (map (curry zeicheTangente f)
      '(1 2 3 4)
      '(blue green purple cyan)) )
```

Funktionskomposition

Definition (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 (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 f1, f2, f3
  (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))
```

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), siehe Definition **??**.
- 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 (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.

Die Dylan-Funktionen

19 Funktionale Abschlüsse, Verknüpfung von Funktionen
- Curry-Verfahren
- Funktionskomposition
- Funktionale Abschlüsse

20 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
```

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 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?>) $\longrightarrow$ <procedure>

Suche Element: Suche ein Element in einer Liste, das ein bestimmtes Prädikat erfüllt, #f sonst.

(findf <predicate?> <list>) $\longrightarrow$ any/c

Suche Restliste: Erfüllt mindestens ein Element der Liste das Prädikat? Wenn ja, gib die Restliste zurück.

(memf <predicate?> <list>) $\longrightarrow$ (or/c list? **#f**)

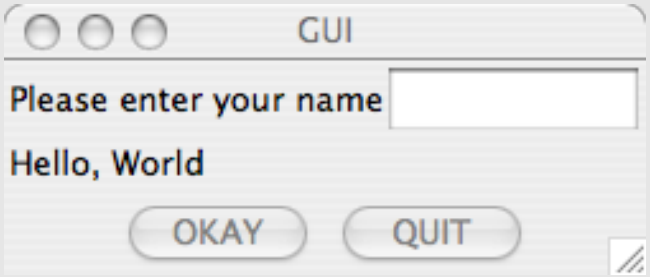
Allquantor: Erfüllen alle Elemente der Liste das Prädikat?

(andmap <predicate?> <list> ...) $\longrightarrow$ <boolean>

Existenzquantor: Erfüllt mindestens ein Element der Liste das Prädikat?

(ormap proc lst ...+) $-?$ boolean?

Call back functions:
Funktionale Abschlüsse als Ereignisroutinen



19 Funktionale Abschlüsse, Verknüpfung von Funktionen

20 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 aufgefufen, 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

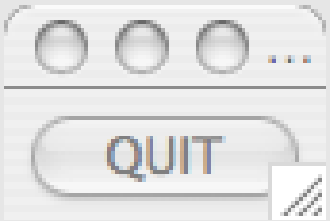
(make-choice '({<string>})) ⟶ <gui-item>

- 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



window w

Beispiel

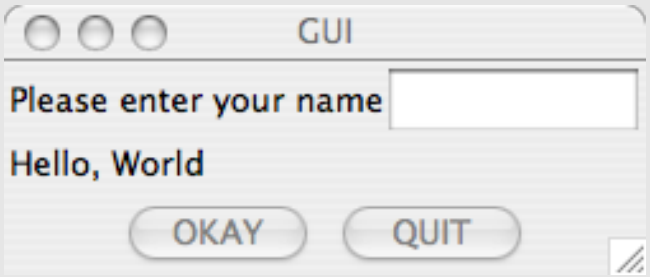
Konstruiere ein Fenster mit einem Knopf:

- Drücken des Knopfes schließt das Fenster;
- (show−window w) zeigt es wieder an.

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



window w

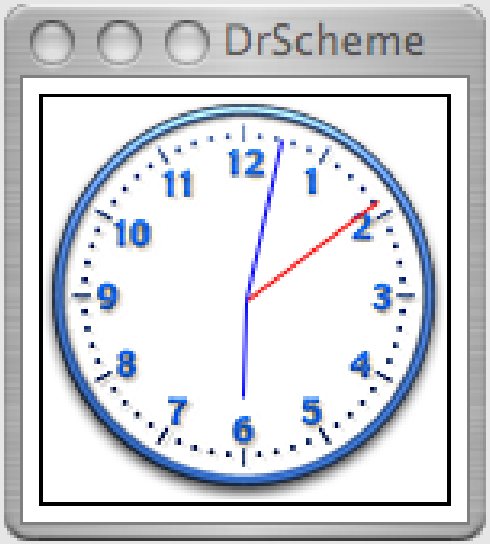
Beispiel

Konstruiere ein Fenster mit Eingabefeld und Anzeige:

- Nach dem Drücken des OK-Knopfes wird die Eingabe gelesen;
- Der Anzeigetext wird überschrieben.

Das Programm

```
(define text1
  (make-text "Please␣enter␣your␣name"))
(define msg1
  (make-message
   (string-append  "Hello,␣World"
    (make-string 33 #\SPACE))))
```



„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,

„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,

„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.

„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.

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.

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: Ein UFO erscheint ...

Beispiel 2: Eine analoge Uhr

Beispiel (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 menue $\longrightarrow$ *insert image*.

```
(require 2htdp/image)

(define *clock-face*  )
(define *clock-canvas-w*
  (image-width *clock-face*))
; the with 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))
```

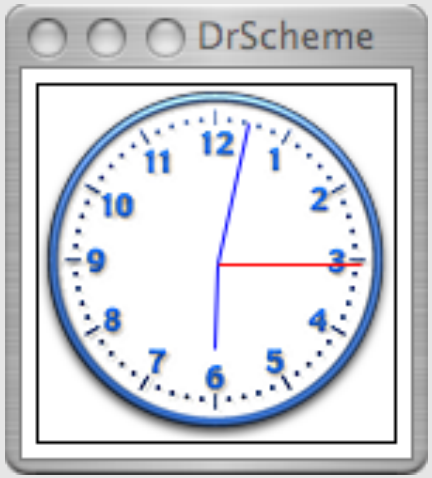
animate: Ereignisroutine „draw-clockworld“

```
(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" )) ...
```

```
      (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 Beispiellauf

Bildschirmschnappschüsse: ein Beispiellauf



Bildschirmschnappschüsse: ein Beispiellauf

Bildschirmschnappschüsse: ein Beispiellauf

Beispiel 2: Mit Interaktion

big-bang: Der Urknall

Die Funktion big-bang

- initialisiert eine simulierte Welt,

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,

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

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 Simulationsereignisse:

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 Simulationsereignisse:

  on-tick: Ereignisroutine für Uhrereignisse

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 Simulationsereignisse:

  on-tick: Ereignisroutine für Uhrereignisse

  to-draw: Ereignisroutine zum Zeichnen der Welt

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 Simulationsereignisse:

on-tick:	Ereignisroutine für Uhrereignisse
to-draw:	Ereignisroutine zum Zeichnen der Welt
stop-when:	Prädikat für den Abbruch

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 Simulationsereignisse:

  on-tick: Ereignisroutine für Uhrereignisse
  to-draw: Ereignisroutine zum Zeichnen der Welt
  stop-when: Prädikat für den Abbruch
  on-key: Ereignisroutine für Tastaturereignisse

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 Simulationsereignisse:

on-tick:	Ereignisroutine für Uhrereignisse
to-draw:	Ereignisroutine zum Zeichnen der Welt
stop-when:	Prädikat für den Abbruch
on-key:	Ereignisroutine für Tastaturereignisse

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 Simulationsereignisse:

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**

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 cloc
    (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 ti

(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))
```

Theodor Storms Maikatzen

Beispiel (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

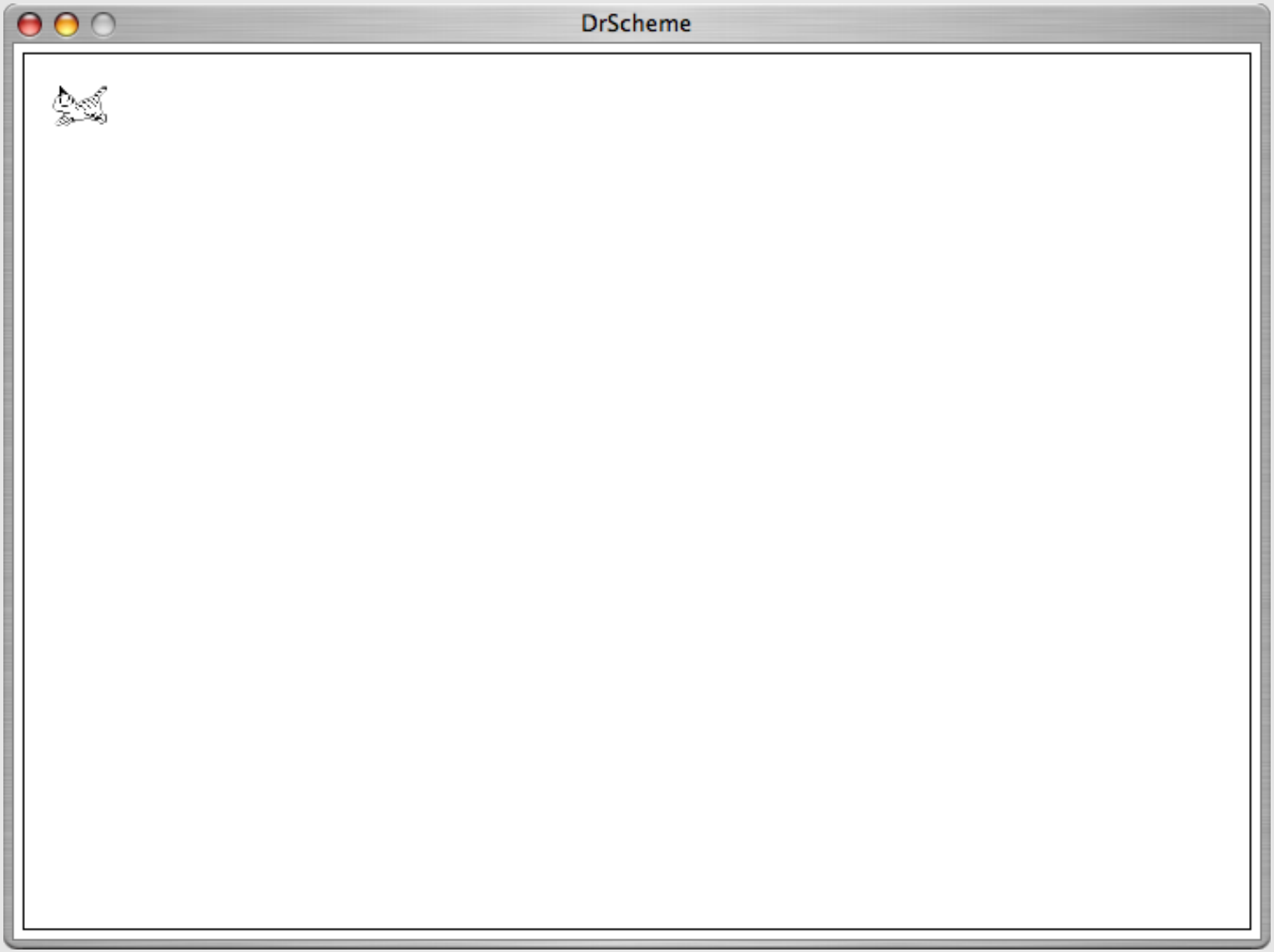
```
(define cat-canvas-w 700)
; the width of the canvas
(define cat-canvas-h 500)
; the height of canvas
```

```
(define katze-winzig  )
```

Ein Beispiellauf: Die Textausgabe

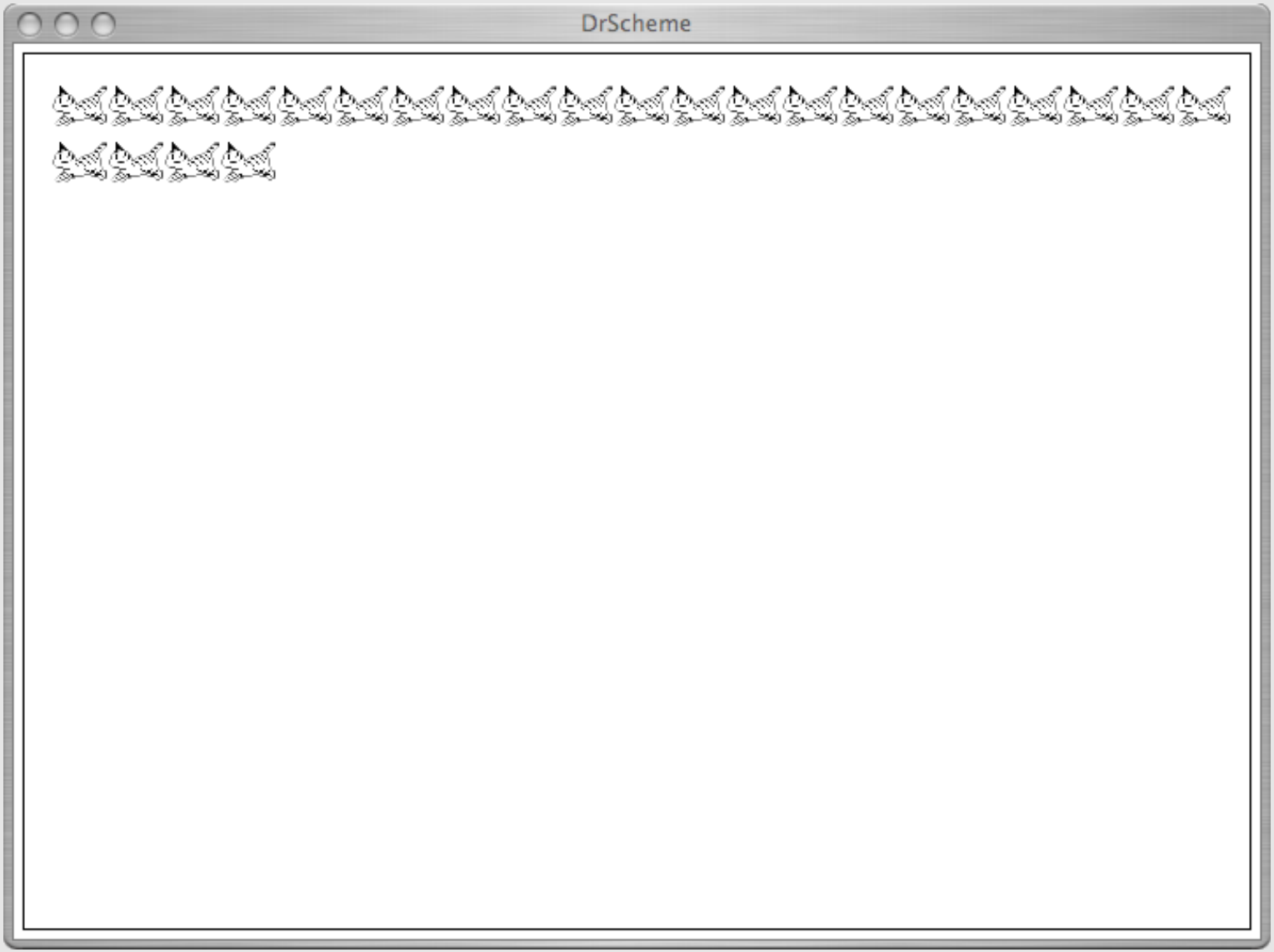
```
> (maikatzen-sim 2) ⟶ #t
(year:  1 number of cats:  1)
(year:  2 number of cats:  5)
(year:  3 number of cats:  25)
(year:  4 number of cats:  125)
(year:  5 number of cats:  625)
end of time: too many cats
```

Ein Beispiellauf: Die grafische Anzeige

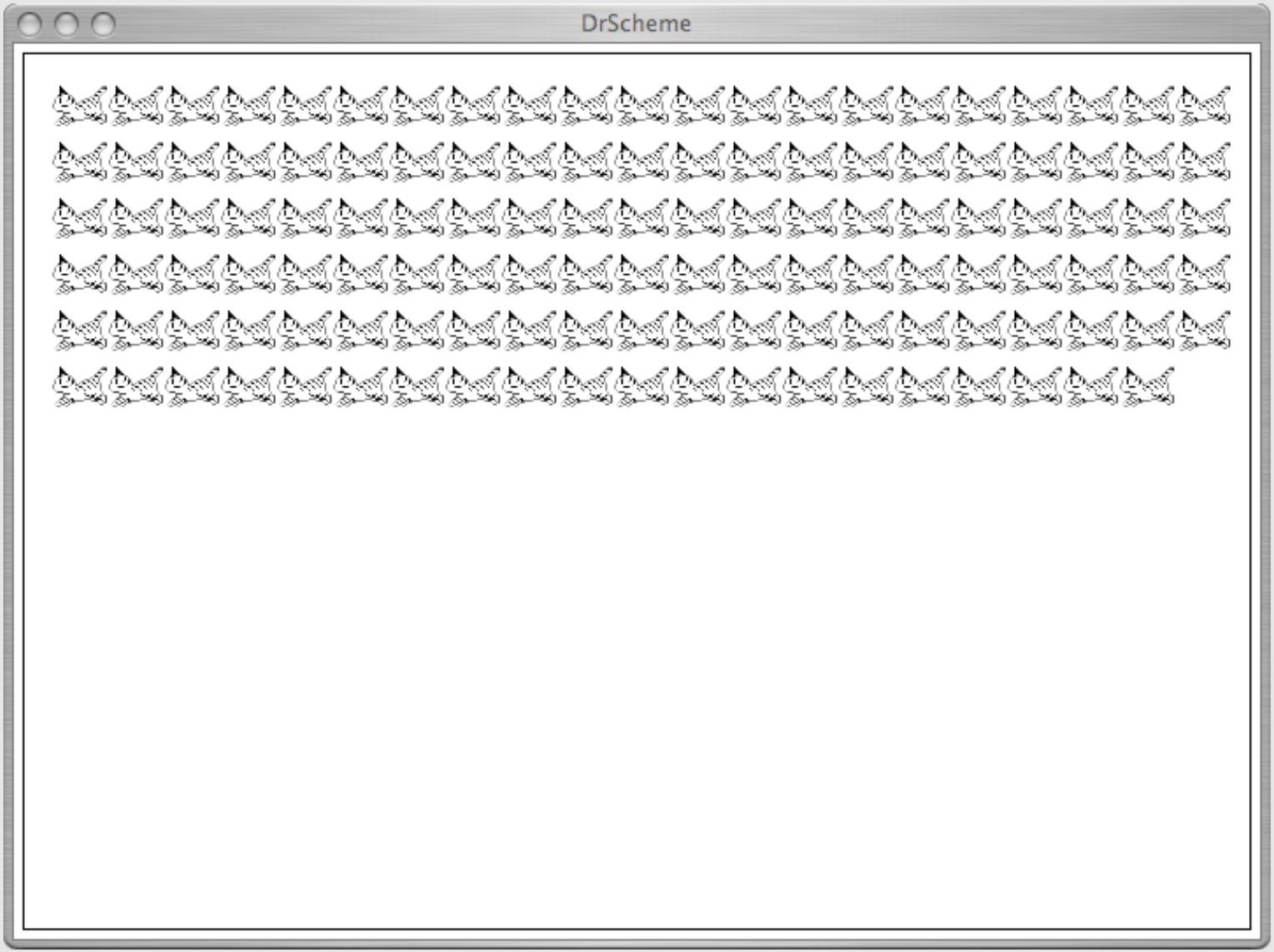


Ein Beispiellauf: Die grafische Anzeige

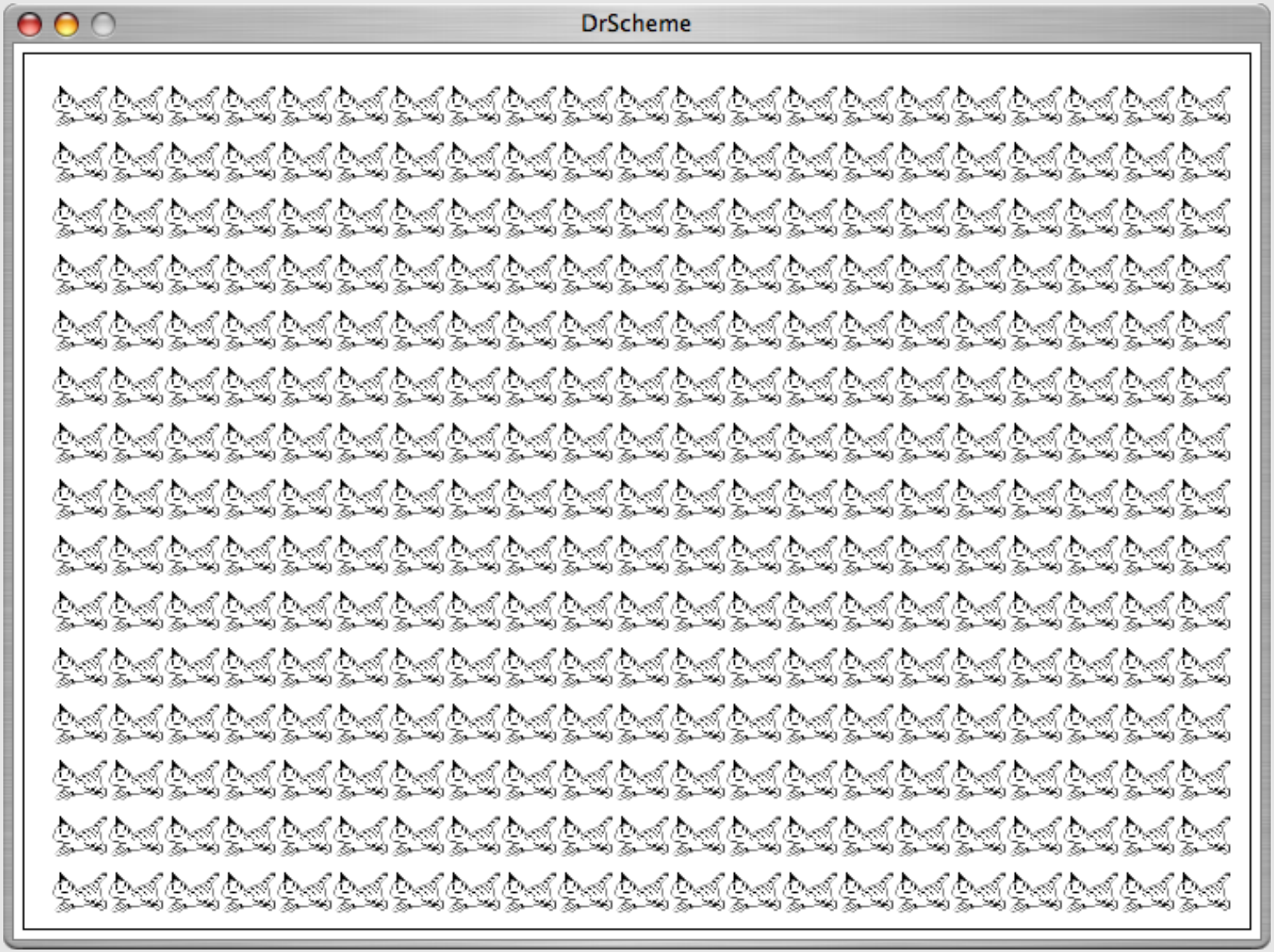
Ein Beispiellauf: Die grafische Anzeige



Ein Beispiellauf: Die grafische Anzeige



Ein Beispiellauf: Die grafische Anzeige



Zusammenfassung:
Funktionen höherer Ordnung

Funktionen als Argument:

Funktionen höhererer Ordnung zur Kontrollabstraktion, die den Ablauf steuern:

Applikation: apply

Abbilden: 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

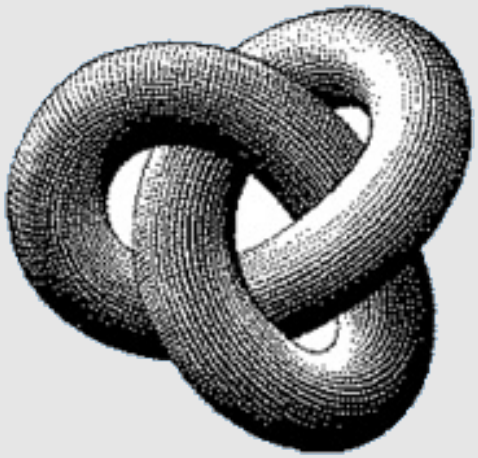
Funktionen als call back:

make-button, animate, big-bang

Teil VIII

Ausgewählte funktionale Algorithmen

Kombinatorische Probleme



M.C. Escher

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?

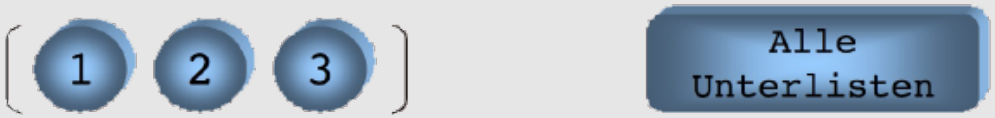
Die Liste aller Unterlisten einer Liste

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:

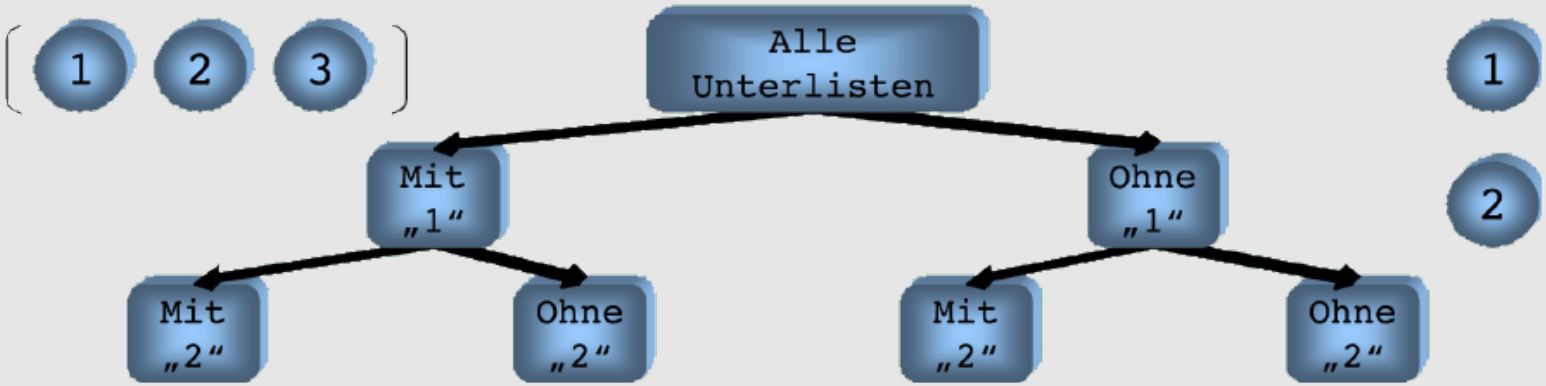
- Alle Unterlisten, die mit (car xs) beginnen und deren cdr eine Unterliste von (cdr xs) ist.
- Alle Unterlisten, die (car xs) nicht enthalten und deren cdr eine Unterliste von (cdr xs) ist.

Die Liste aller Unterlisten (Potenzmenge)

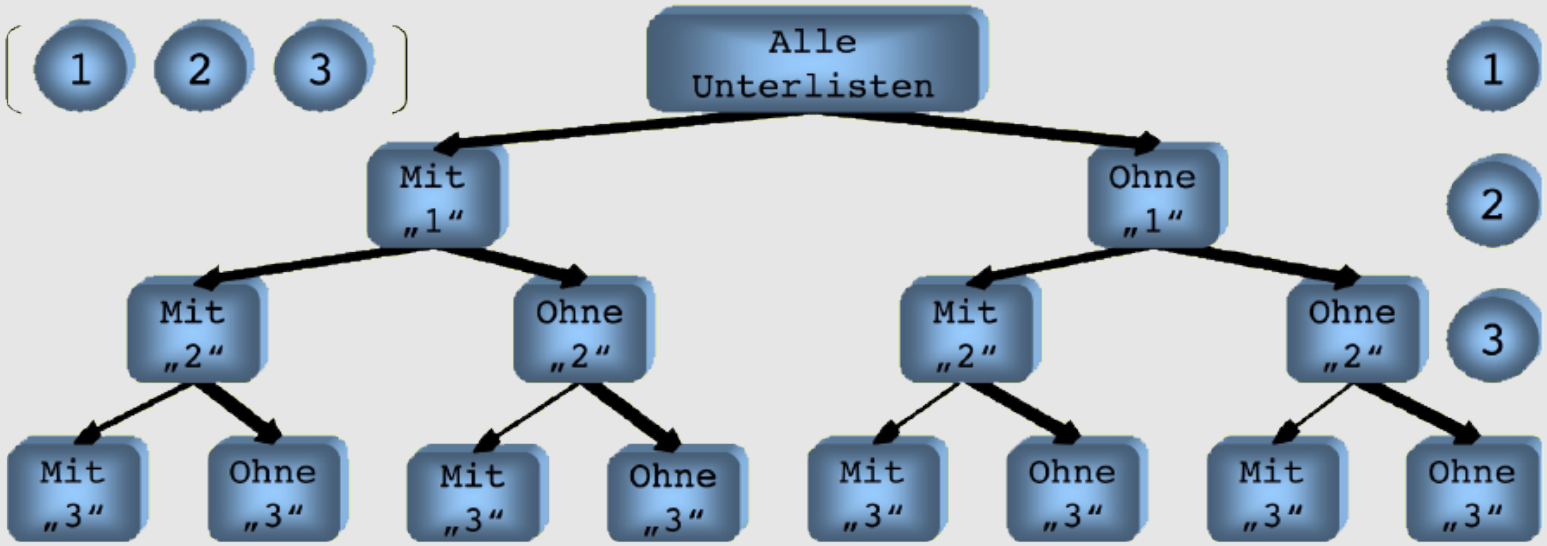


Die Liste aller Unterlisten (Potenzmenge)

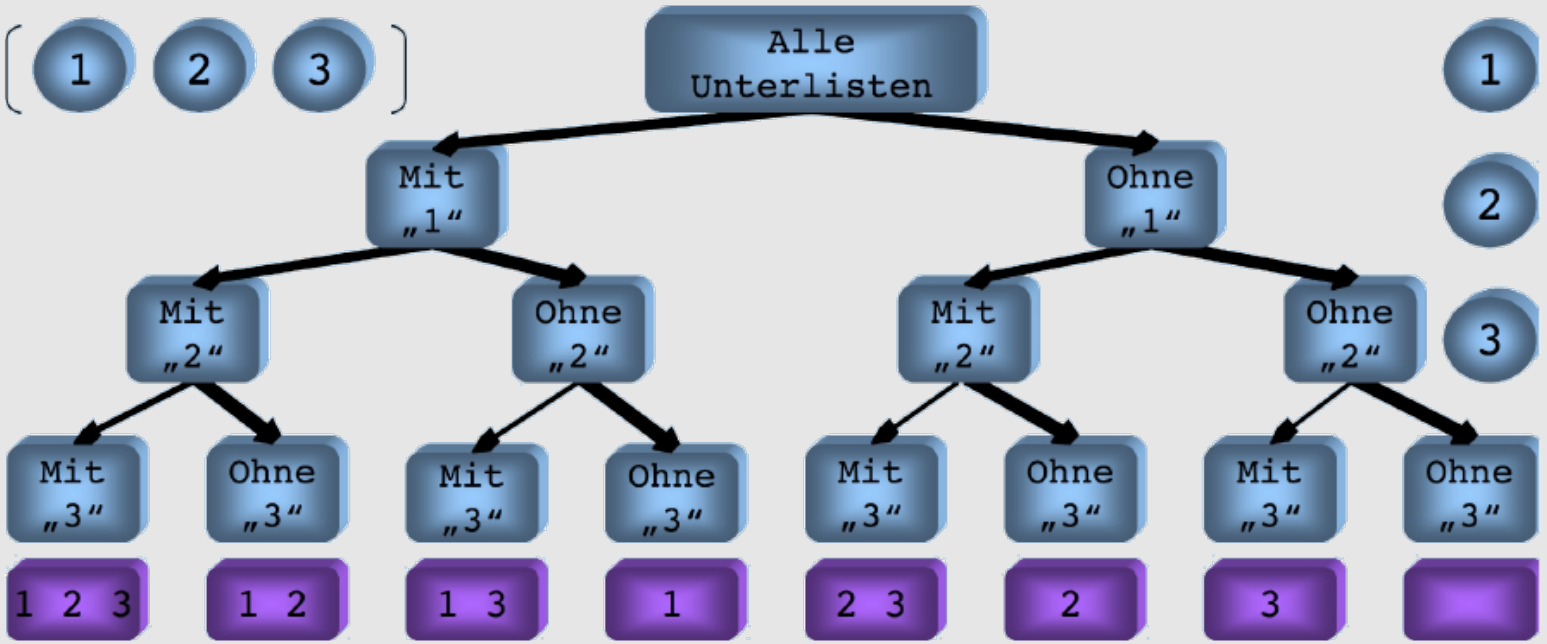
Die Liste aller Unterlisten (Potenzmenge)



Die Liste aller Unterlisten (Potenzmenge)



Die Liste aller Unterlisten (Potenzmenge)



Die Liste aller Unterlisten: subs

Beispiel (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 '(()))  ⟶  (() (()))
```

Verweben (interleave)

Beispiel (Verweben:)

(interleave x ys) berechnet die Liste aller Möglichkeiten, ein Element x in die Liste ys einzufügen.

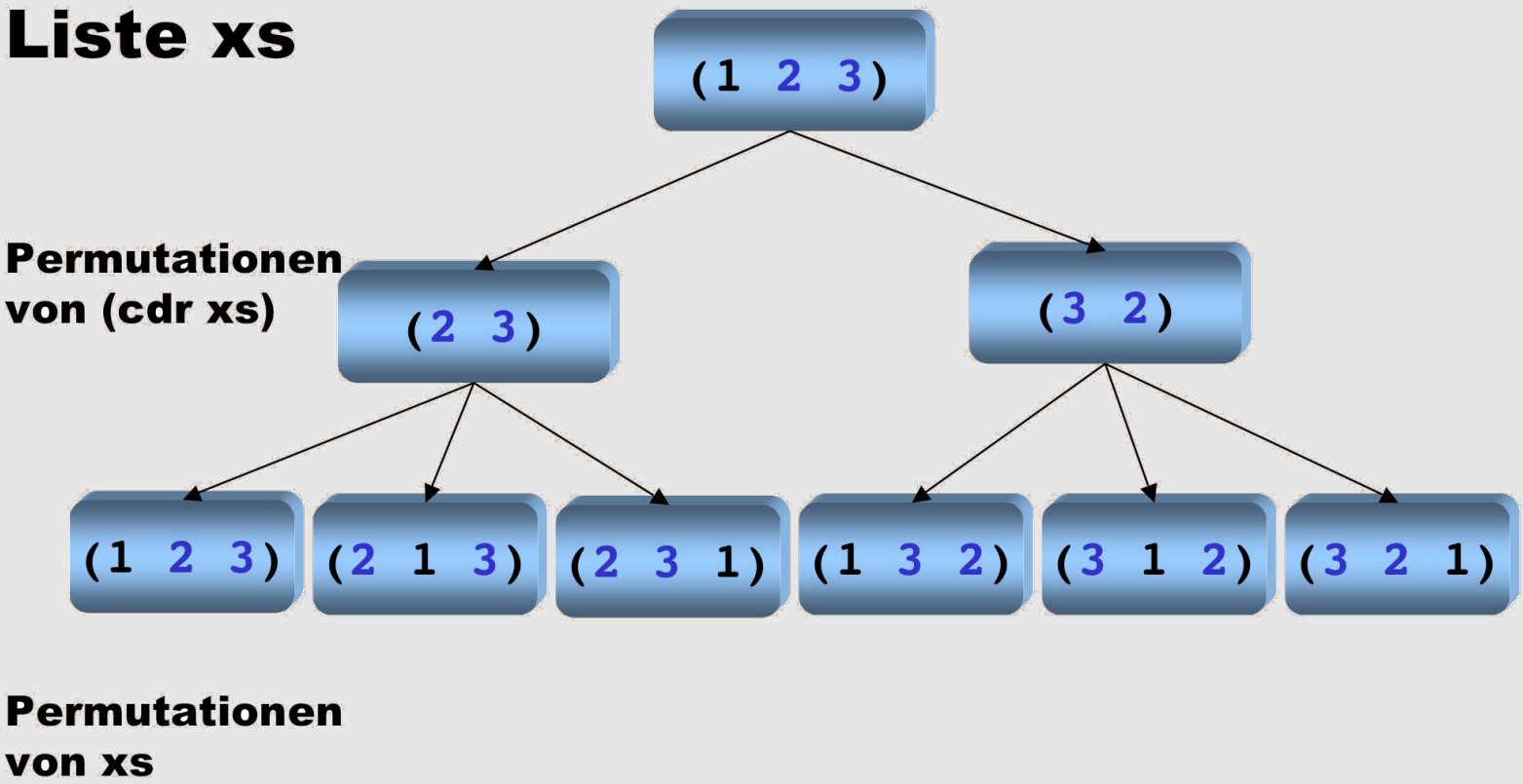
- Wenn ys nicht leer ist,
  - dann wird erstens x an den Anfang von ys gesetzt,
  - und es werden weiterhin alle Möglichkeiten berechnet, y mit dem Rest von xs zu verweben. Der Kopf von y wird vor jeden verwebten Rest gesetzt.

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.


Liste xs
(1 2 3)
Permutationen von (cdr xs)
(2 3)
(3 2)
(1 2 3)
(2 1 3)
(2 3 1)
(1 3 2)
(3 1 2)
(3 2 1)
Permutationen von xs

Permutationen einer Liste

Beispiel (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))
```

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: Wechsele einen Dollar in:

- half-dollars (50 c)
- quarters (25 c),
- dimes (10 c),
- nickels (5 c),
- pennies (1 c).

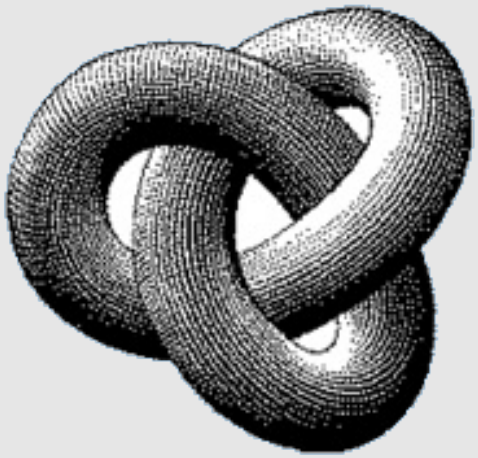
Rekursive Lösung:

Um einen Geldbetrag a zu wechseln, wenn n Münzsorten zur Verfügung stehen, gibt es die folgende Anzahl von Möglichkeiten: [**?**]

- 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
```

Backtracking



M. C. Escher

Backtracking

Backtracking-Algorithmen

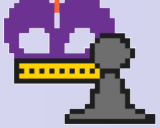
Backtracking-Algorithmen sind rekursive Algorithmen, die sich immer dann gut anwenden lassen,

- 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



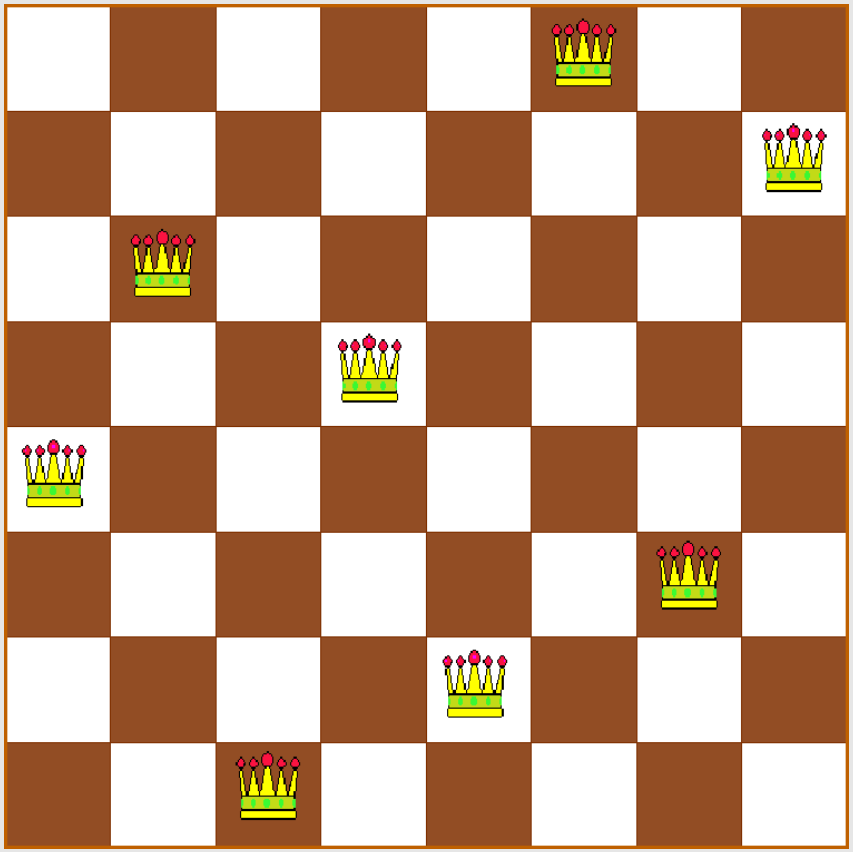
Das 8-Damen-Problem

Problem

Gibt es eine Möglichkeit, 8 Damen so auf einem Schachbrett anzuordnen, 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.*

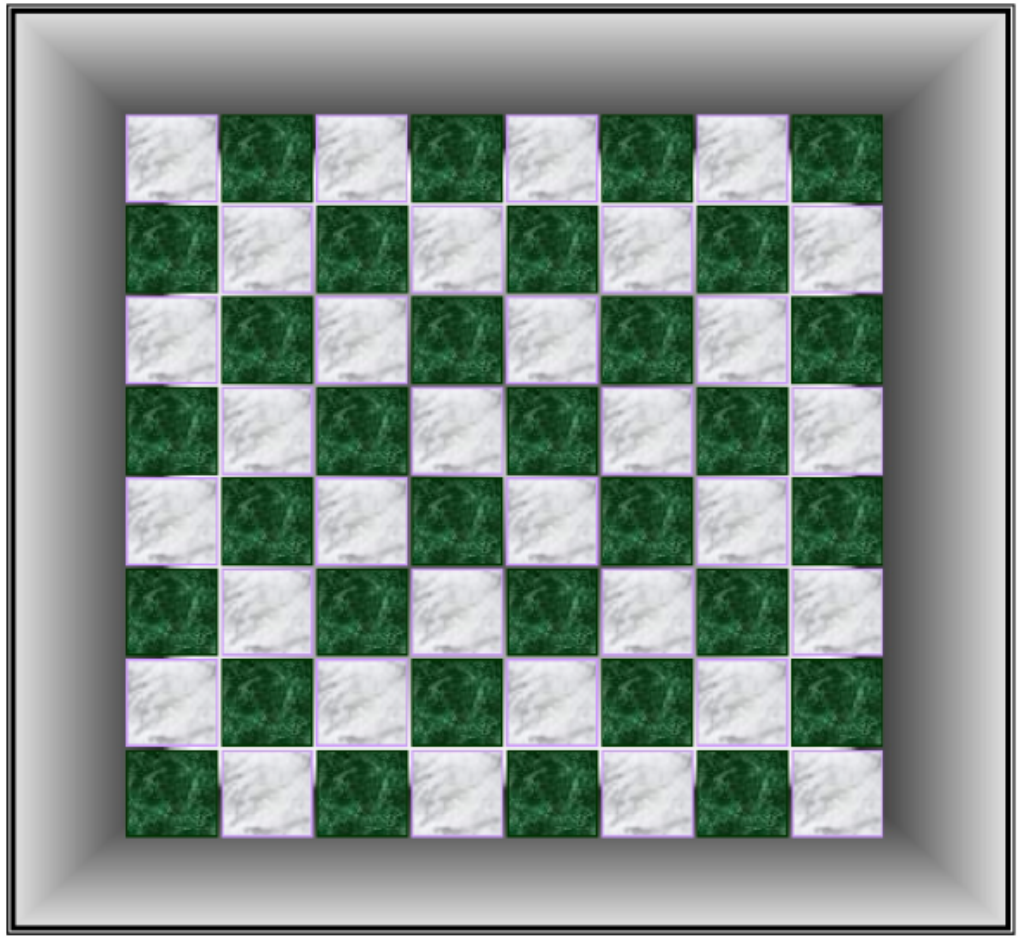
Eine Lösung



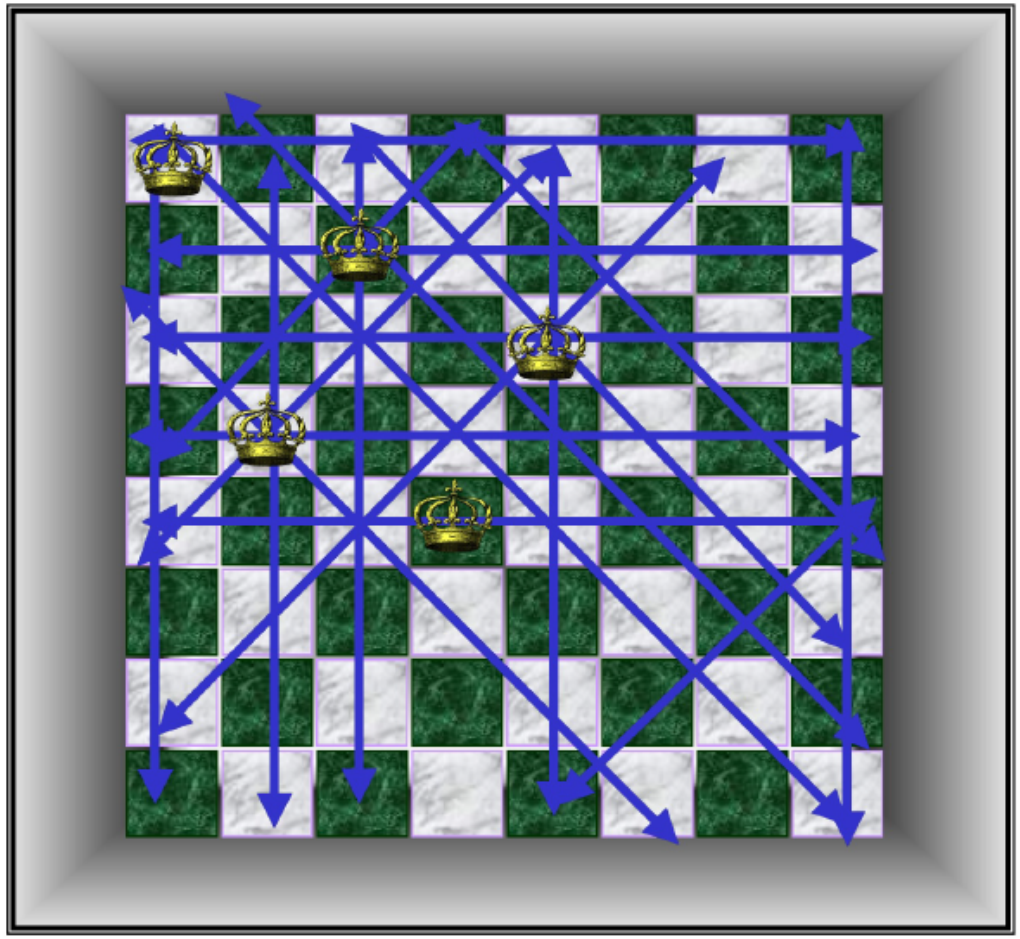
- In jeder Zeile, Spalte und Diagonale steht nur eine Dame.

Rekursiver Ansatz:

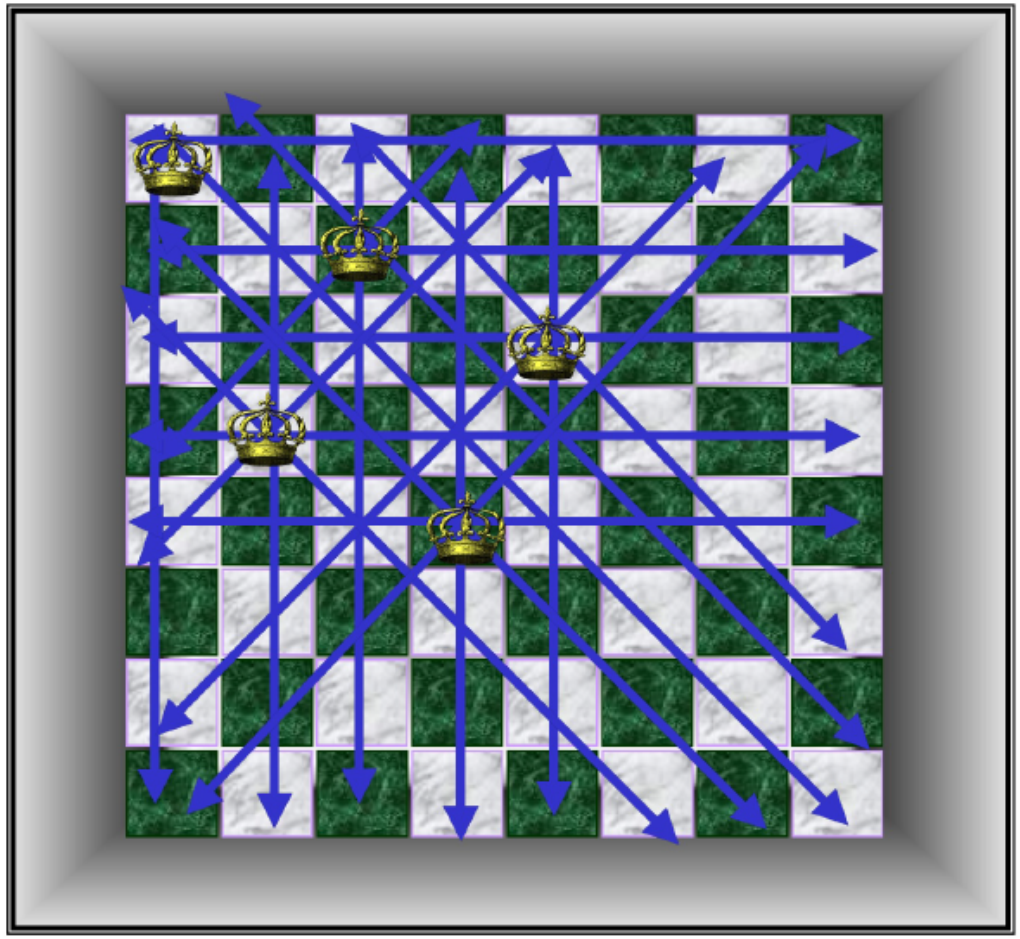
- Für ein Schachbrett der Größe $n \times n$ ist das Problem gelöst, wenn wir eine Dame in der 1-ten Zeile positioniert haben, und dann eine Lösung fuer die restlichen $n - 1$ Zeilen gefunden haben.
- Wenn es für die weiteren Damen keine Möglichkeit der Plazierung gibt, verschieben wir die Dame in der 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.



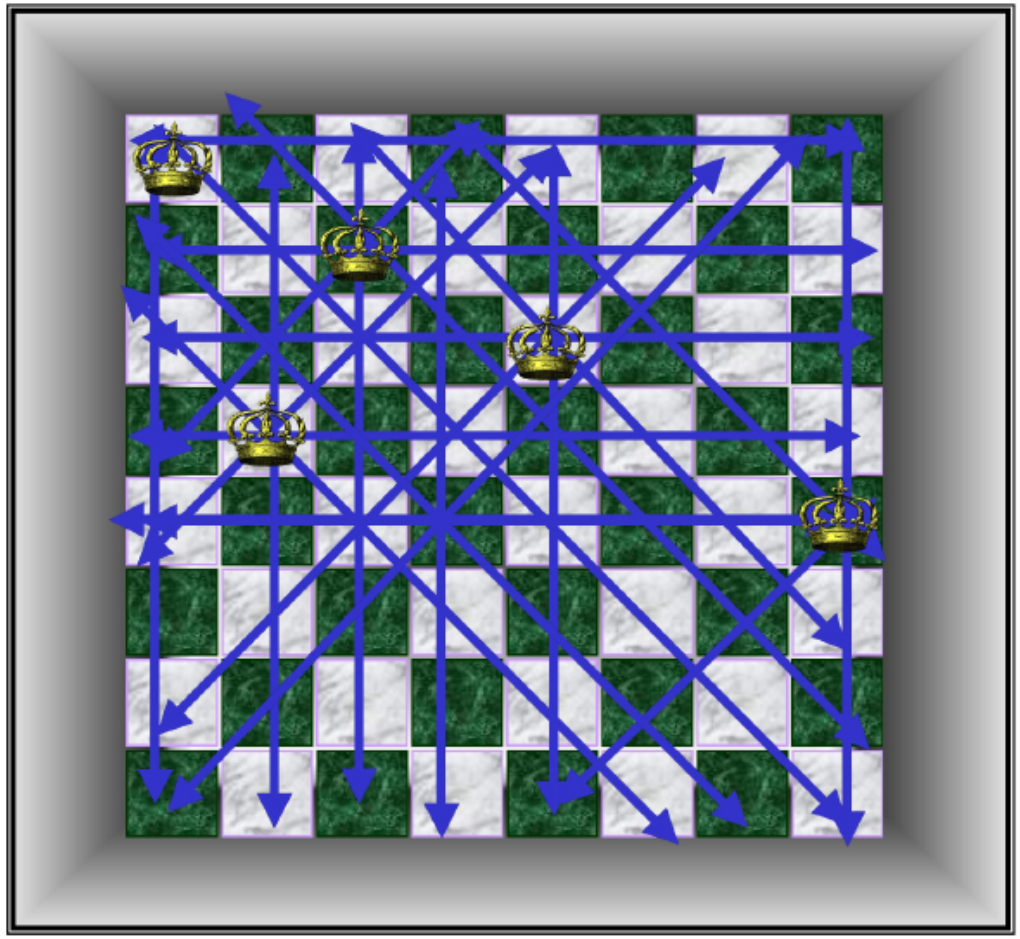
In jeder Zeile, Spalte und Diagonale steht nur eine Dame.



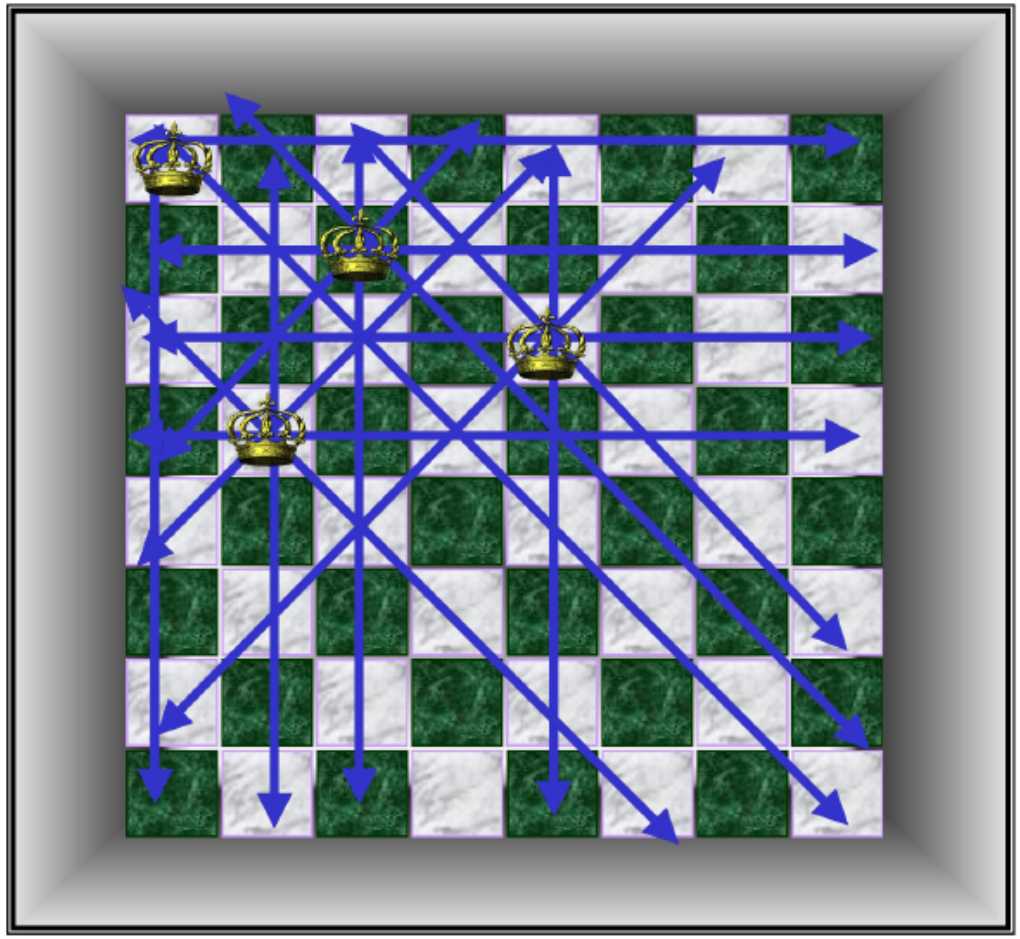
Der 1. backtracking-Schritt: Verschieben in Reihe 5



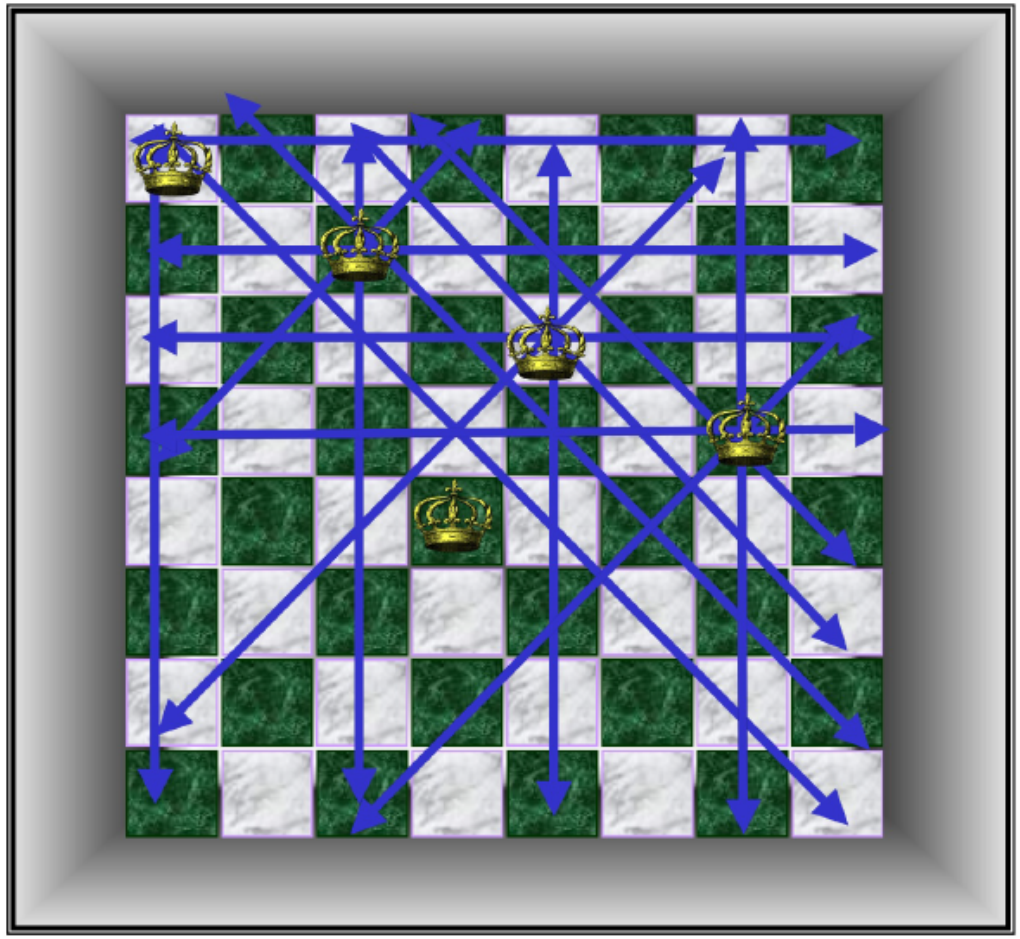
In jeder Zeile, Spalte und Diagonale steht nur eine Dame.



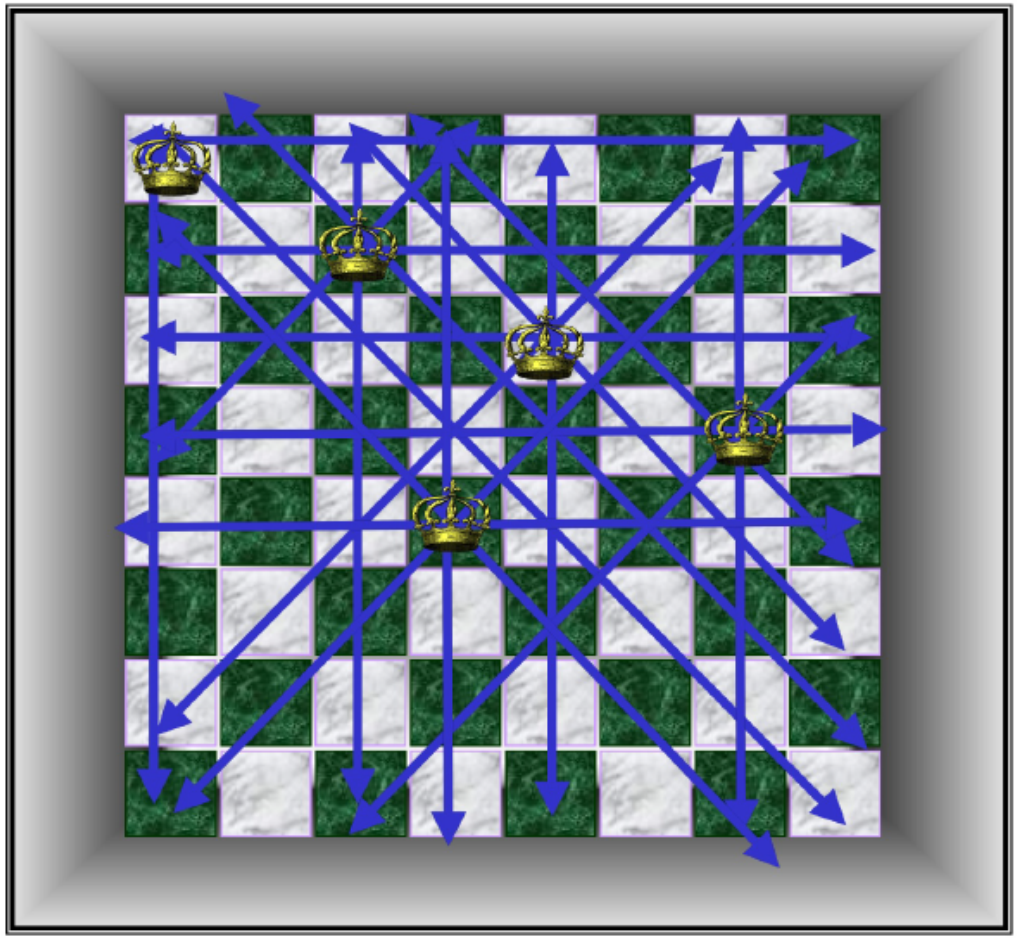
Der 1. backtracking-Schritt: Verschieben in Reihe 5



Der 2. backtracking-Schritt: Verschieben in Reihe 4



Der 2. backtracking-Schritt: Verschieben in Reihe 4



Der 2. backtracking-Schritt: Verschieben in Reihe 4

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.

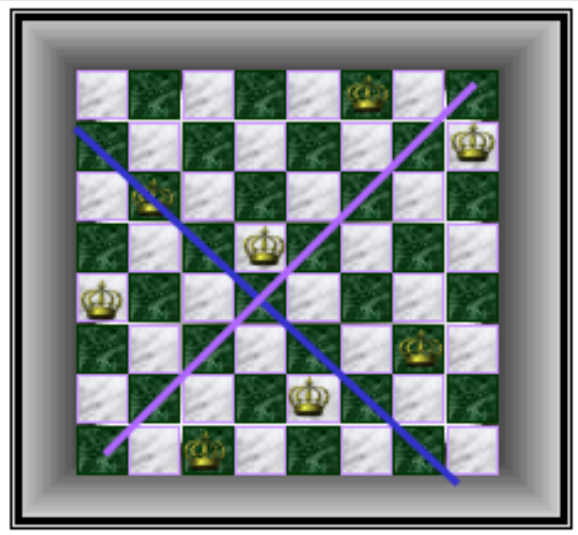
```
(struct
  queen-coord
  (row col))

(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

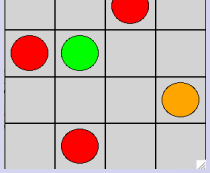
Entlang einer Diagonalen ist entweder

- 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 poss 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))
```



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)]
        [r2 (queen-coord-row pos-2)]
        [c2 (queen-coord-col pos-2)])
    (or (= c1 c2) ; gleiche Spalte
        (= (+ r1 c1) (+ r2 c2))
        ; Diagonale 1
        (= (- r1 c1) (- r2 c2)))))
        ; 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 row (+ 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

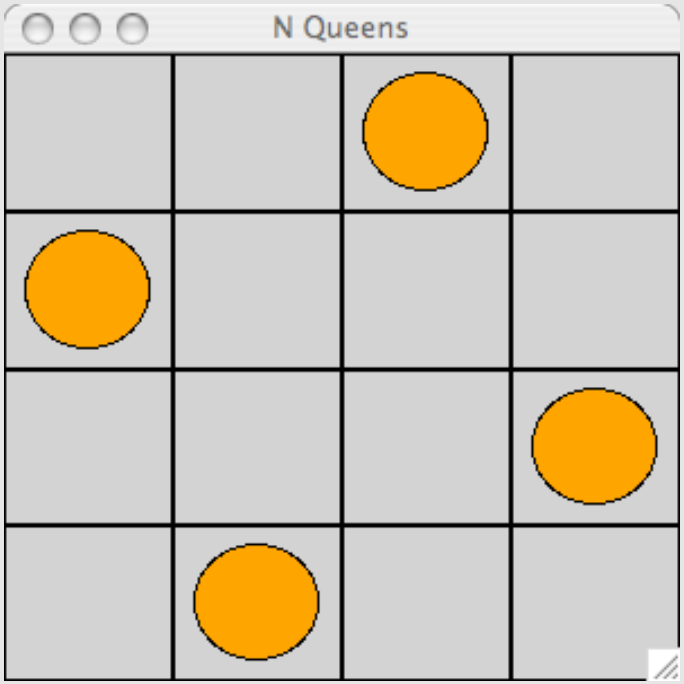
```
(define (queen1 size)
  (try-queen size '() 1 1))

> (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)
```

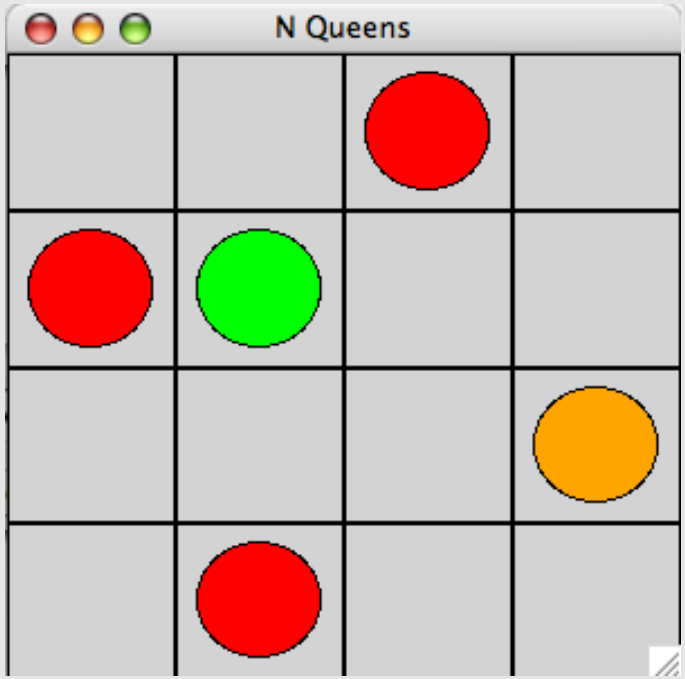
Für die Anzeige: htdp/show-queen.rkt

- 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.

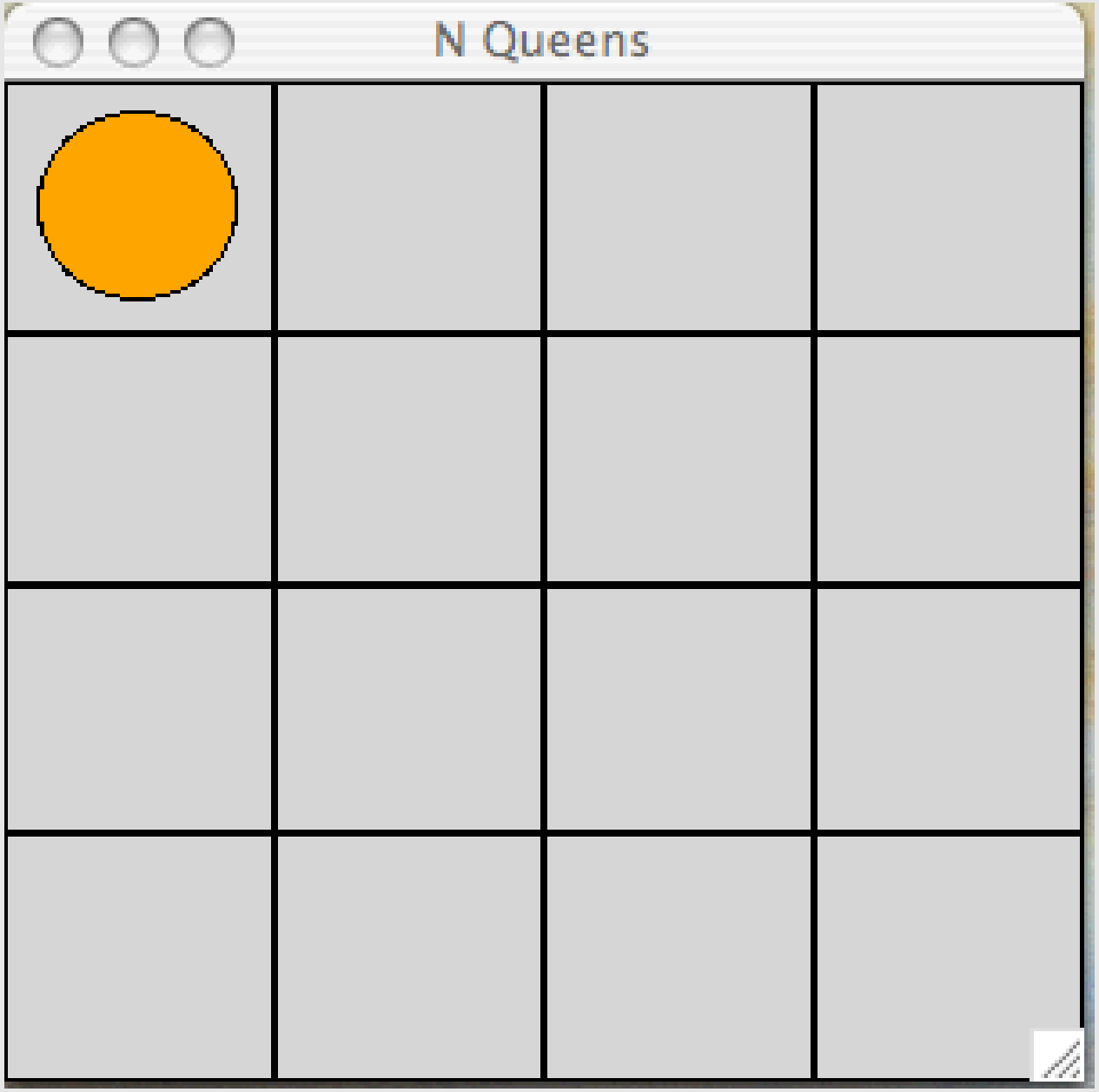
```
(require htdp/show-queen)
(show-queen
  '((#f #t #f #f)
    (#f #f #f #t)
    (#t #f #f #f)
    (#f #f #t #f)))
```

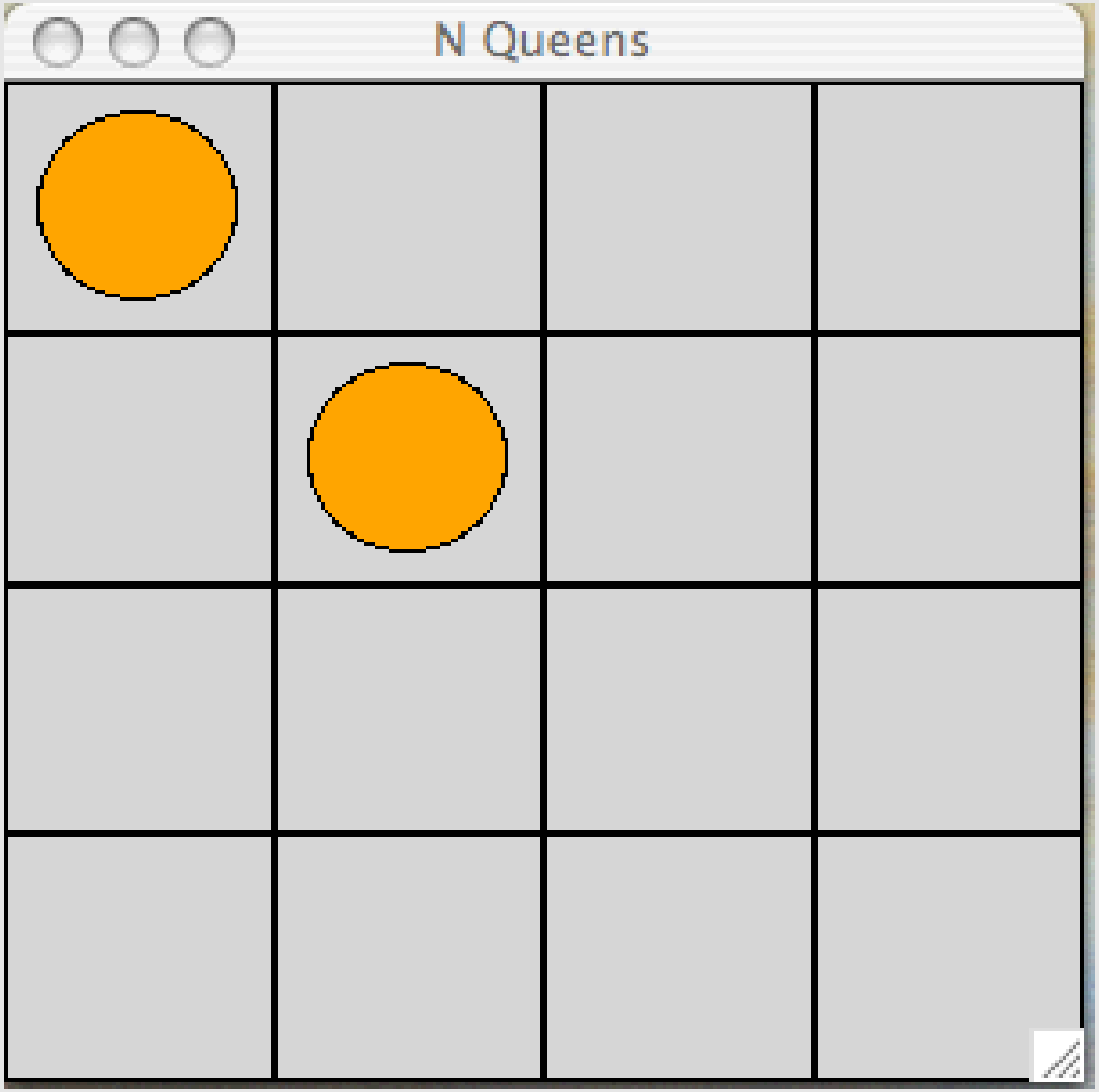
Das Bord

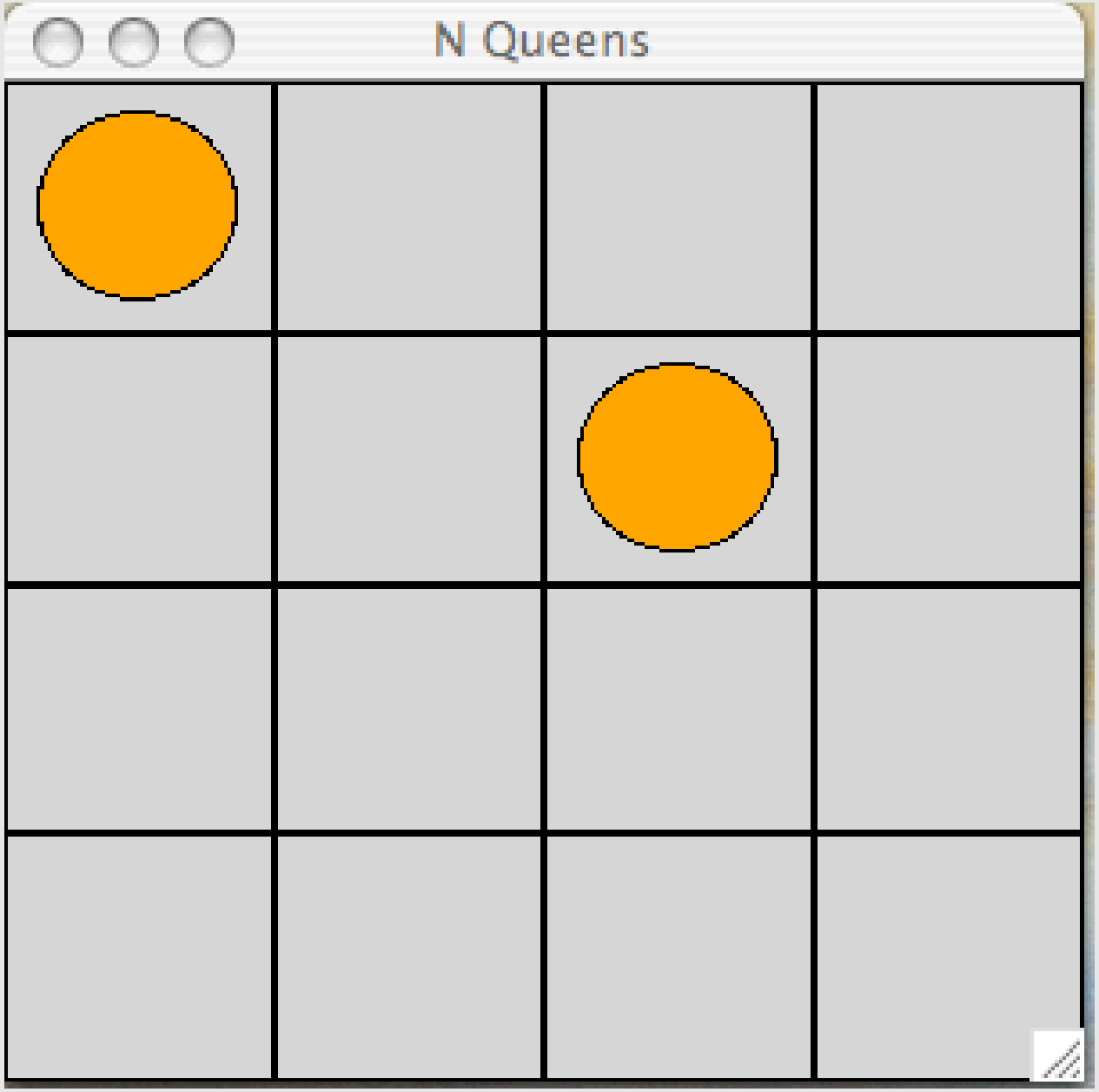


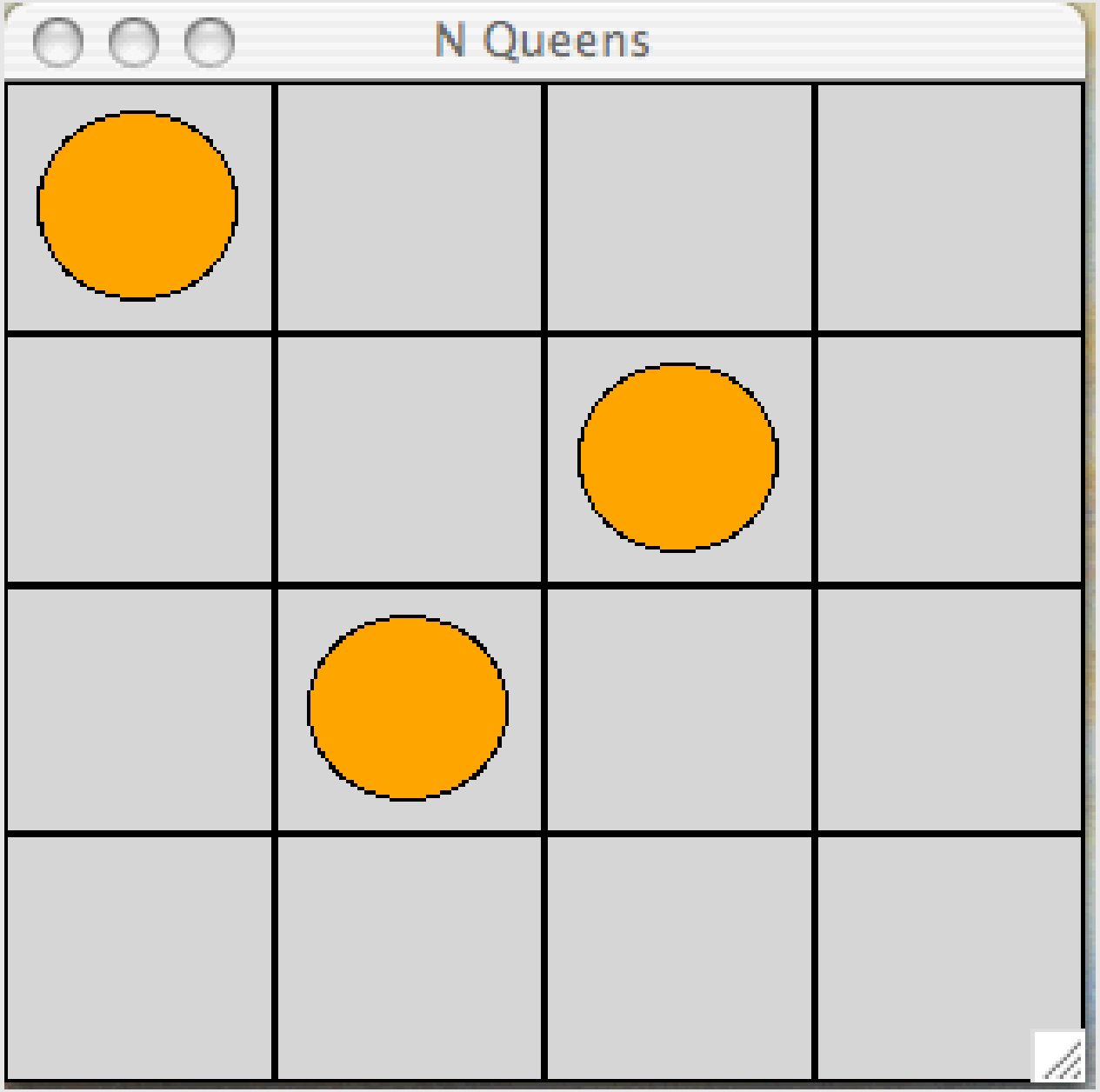


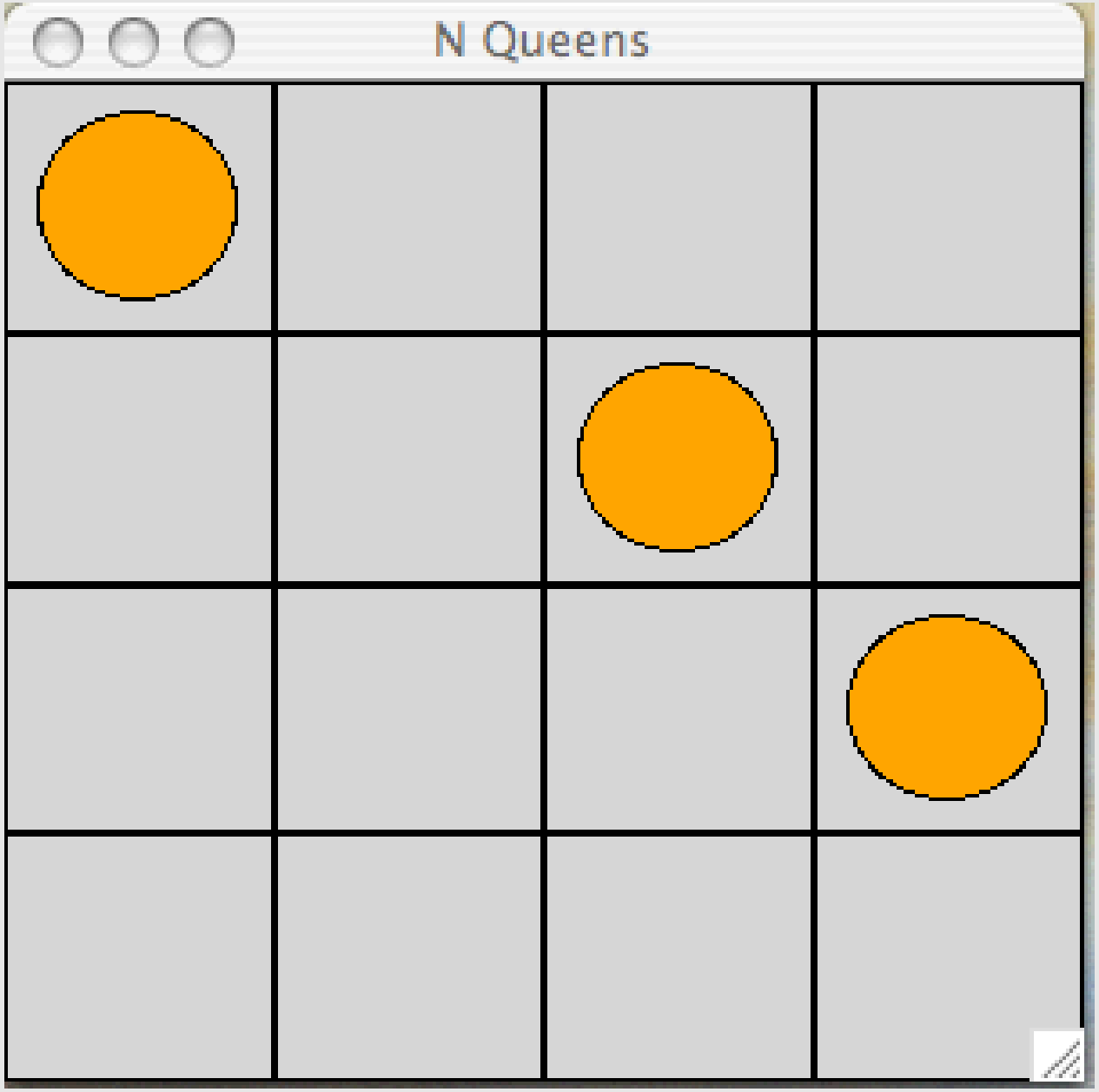
- Geht man mit der Maus auf ein Feld des Brettes, wird das Feld grün markiert.
- Die bedrohten Damen werden rot markiert.

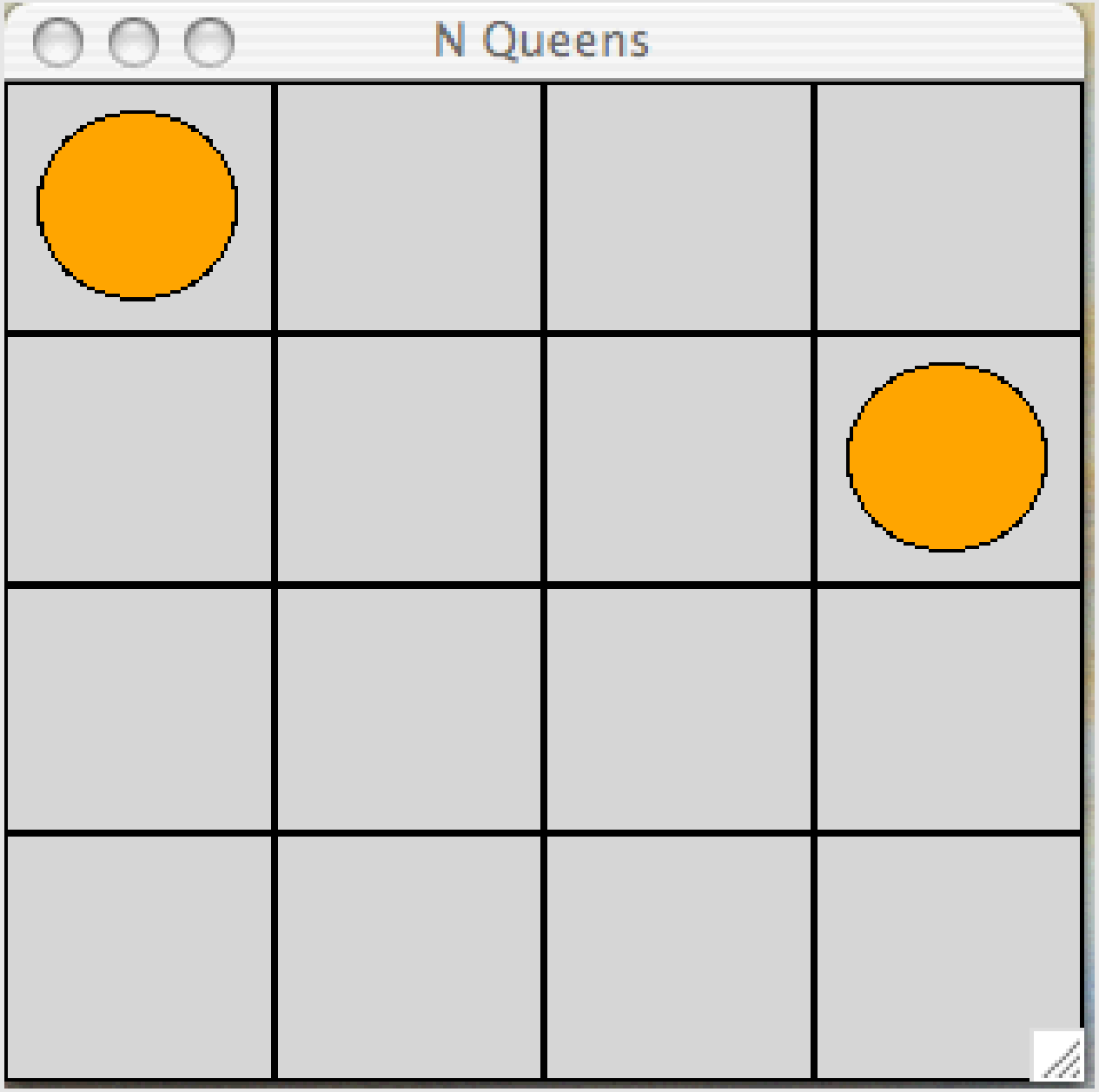

N Queens

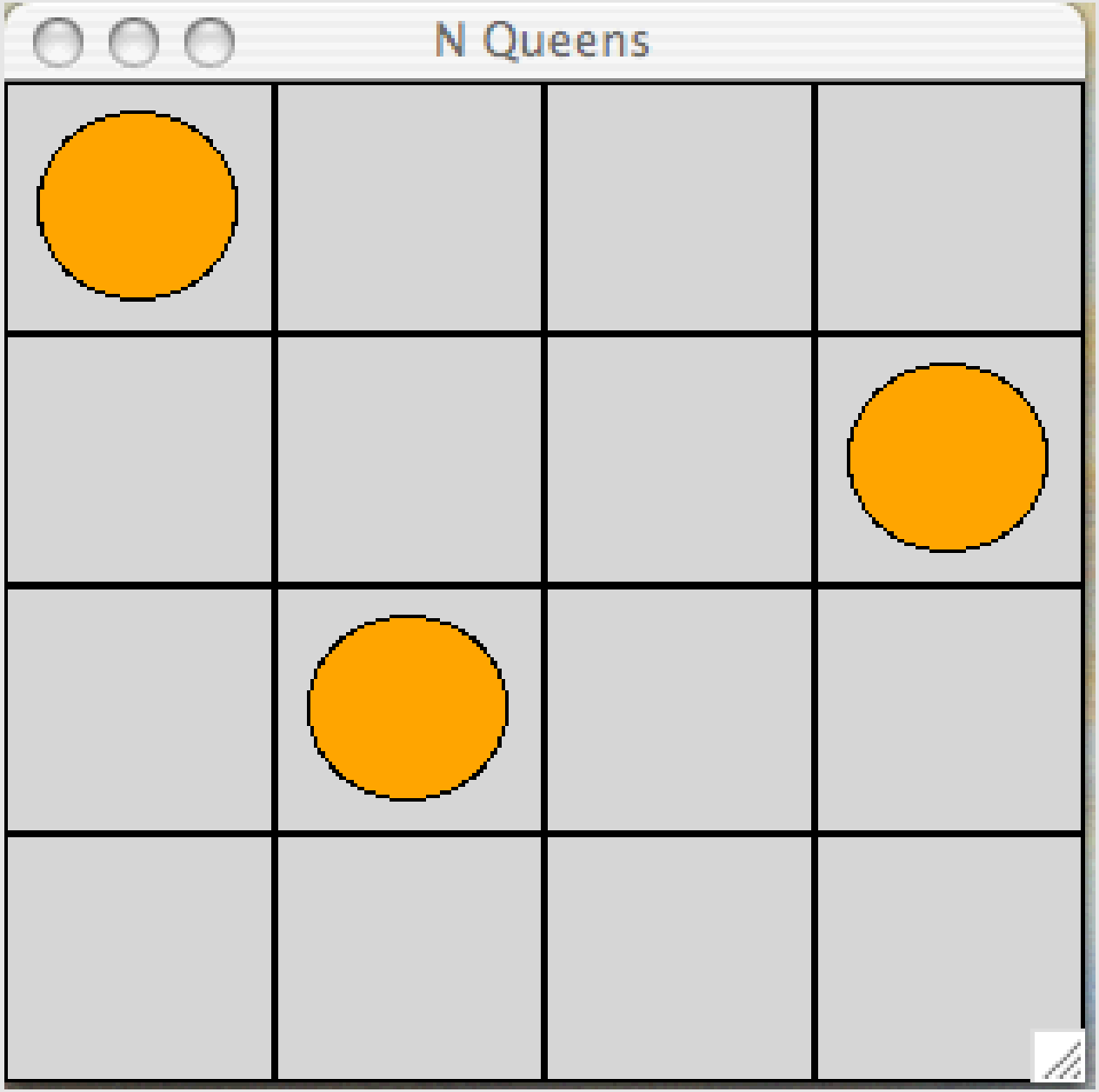

N Queens

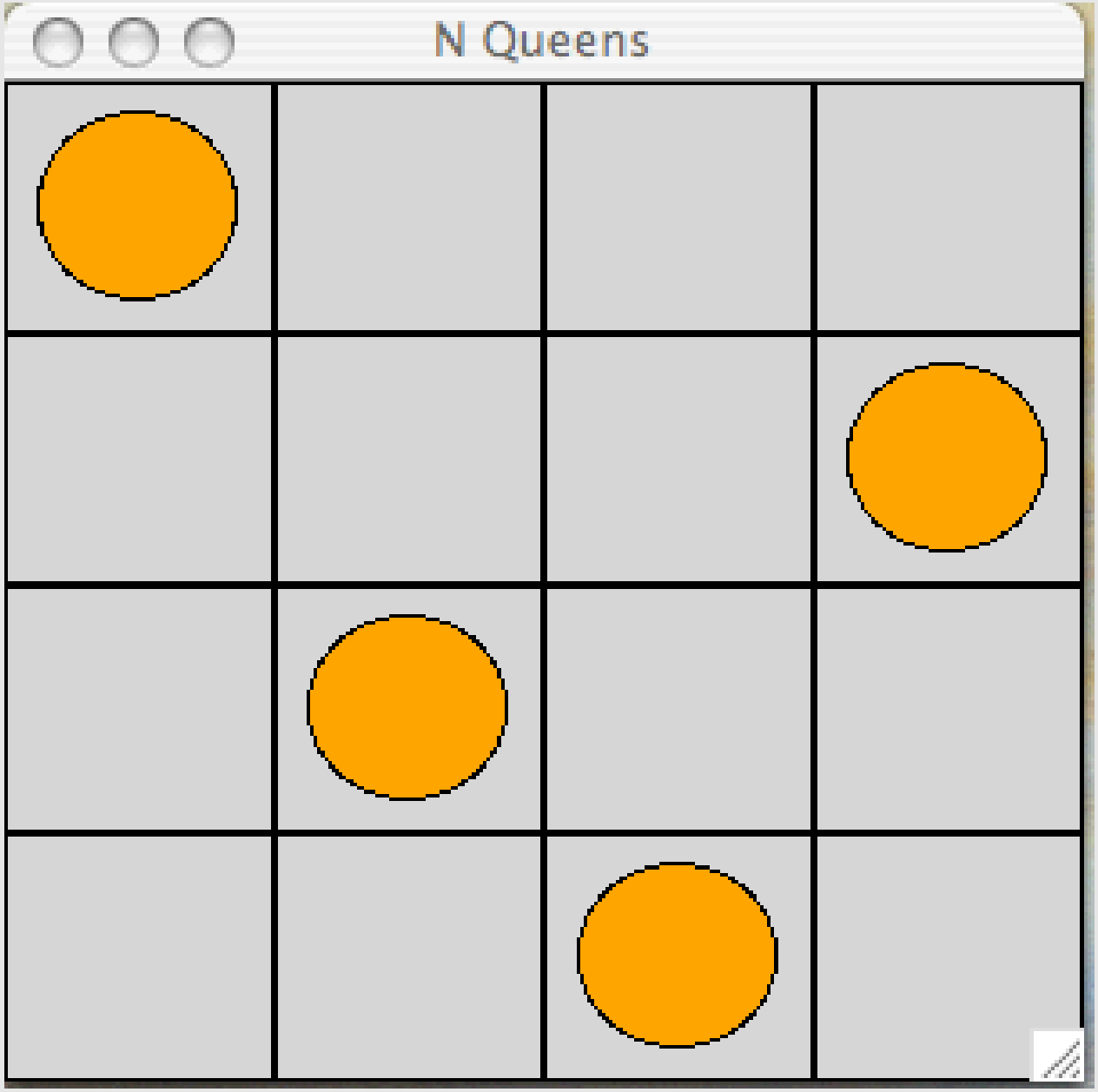

N Queens

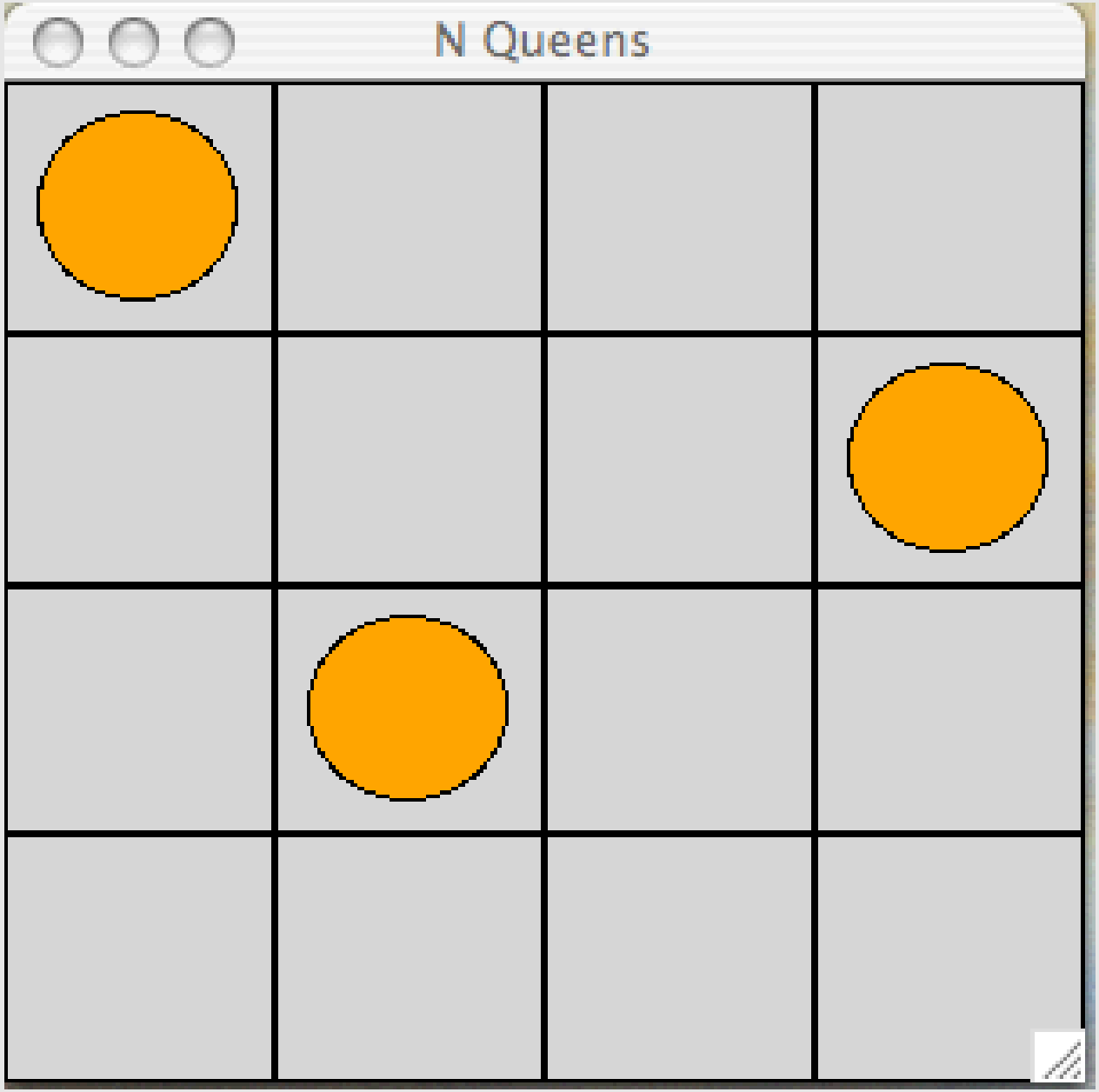

N Queens

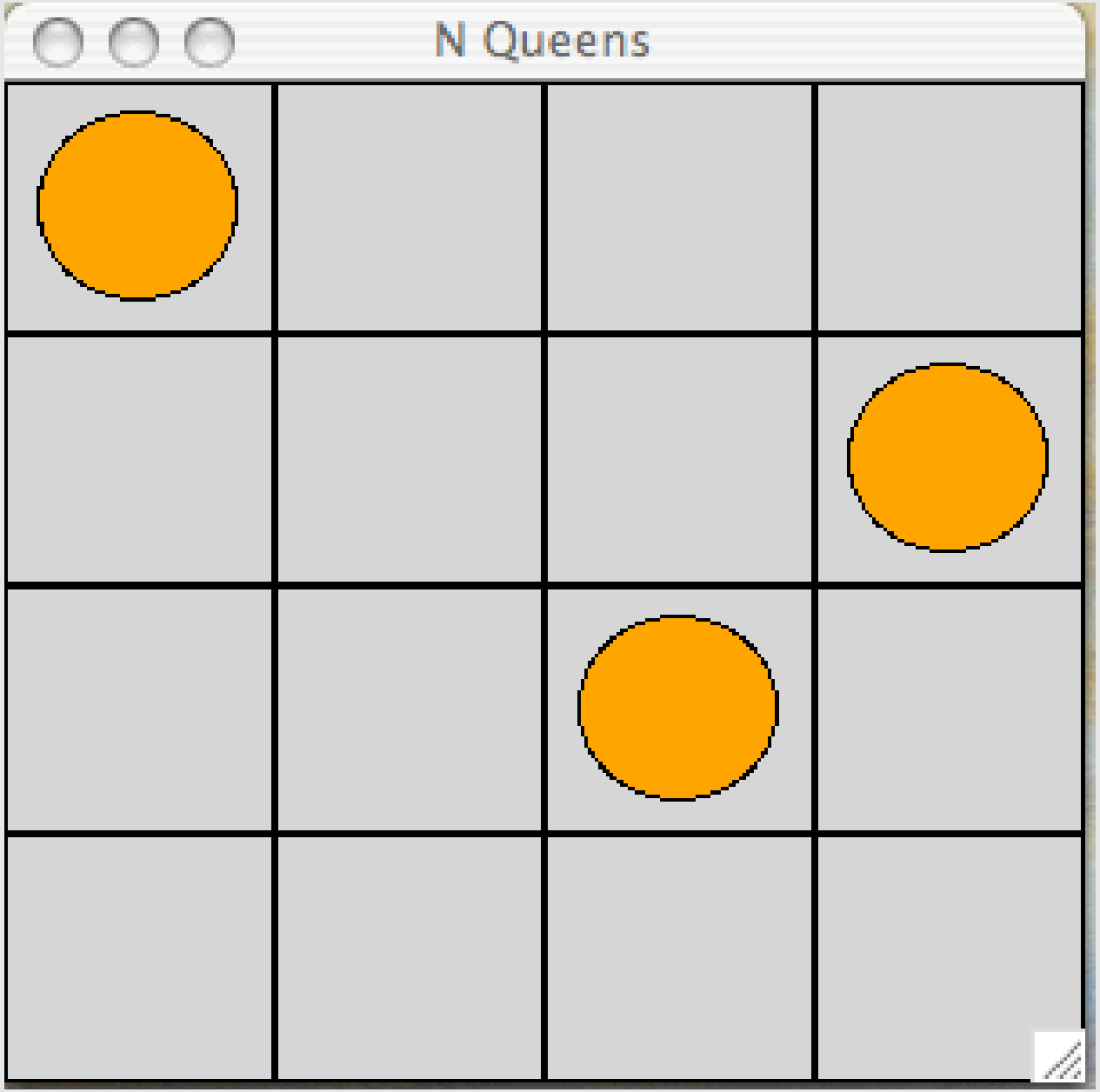

N Queens

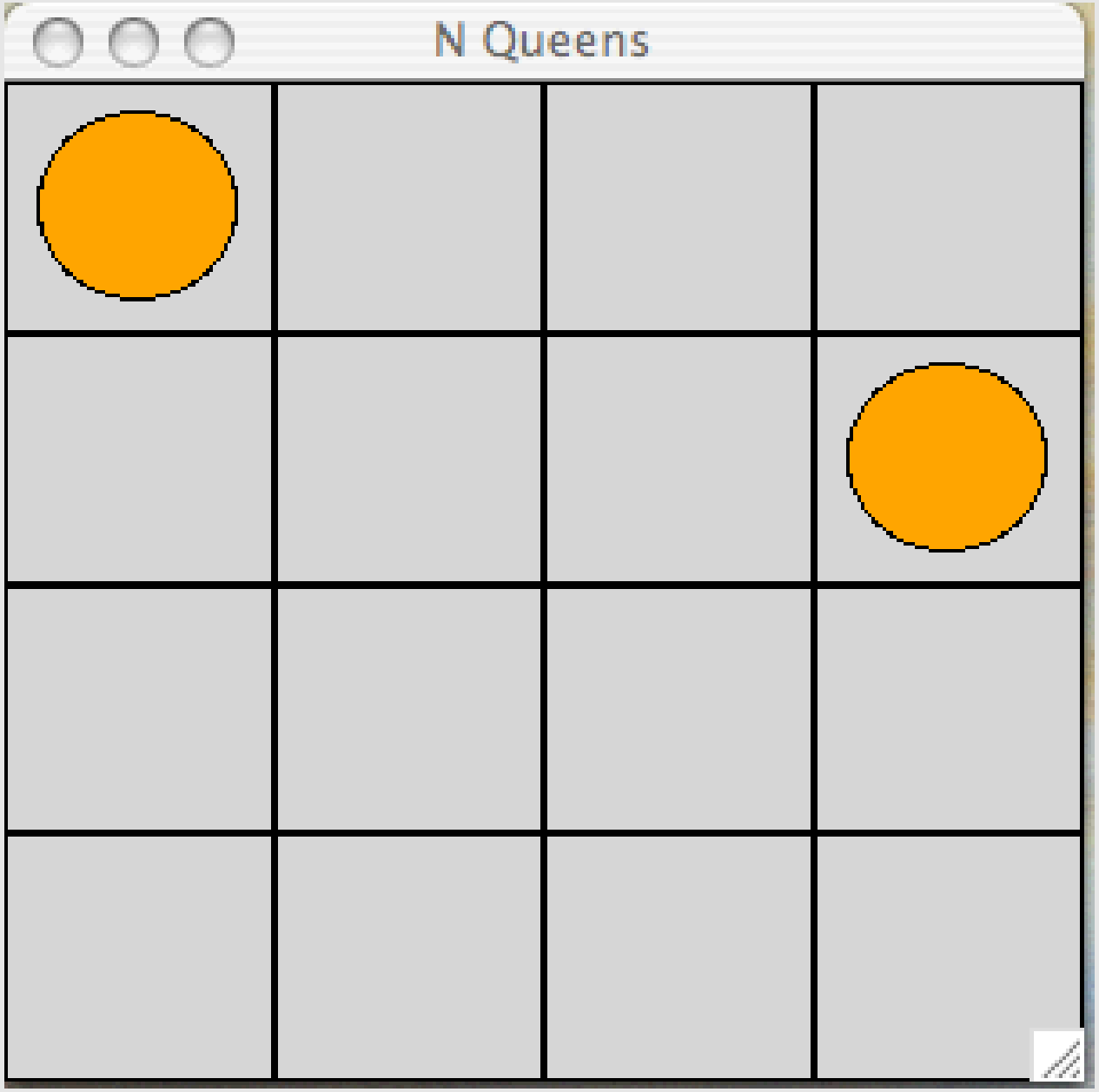

N Queens

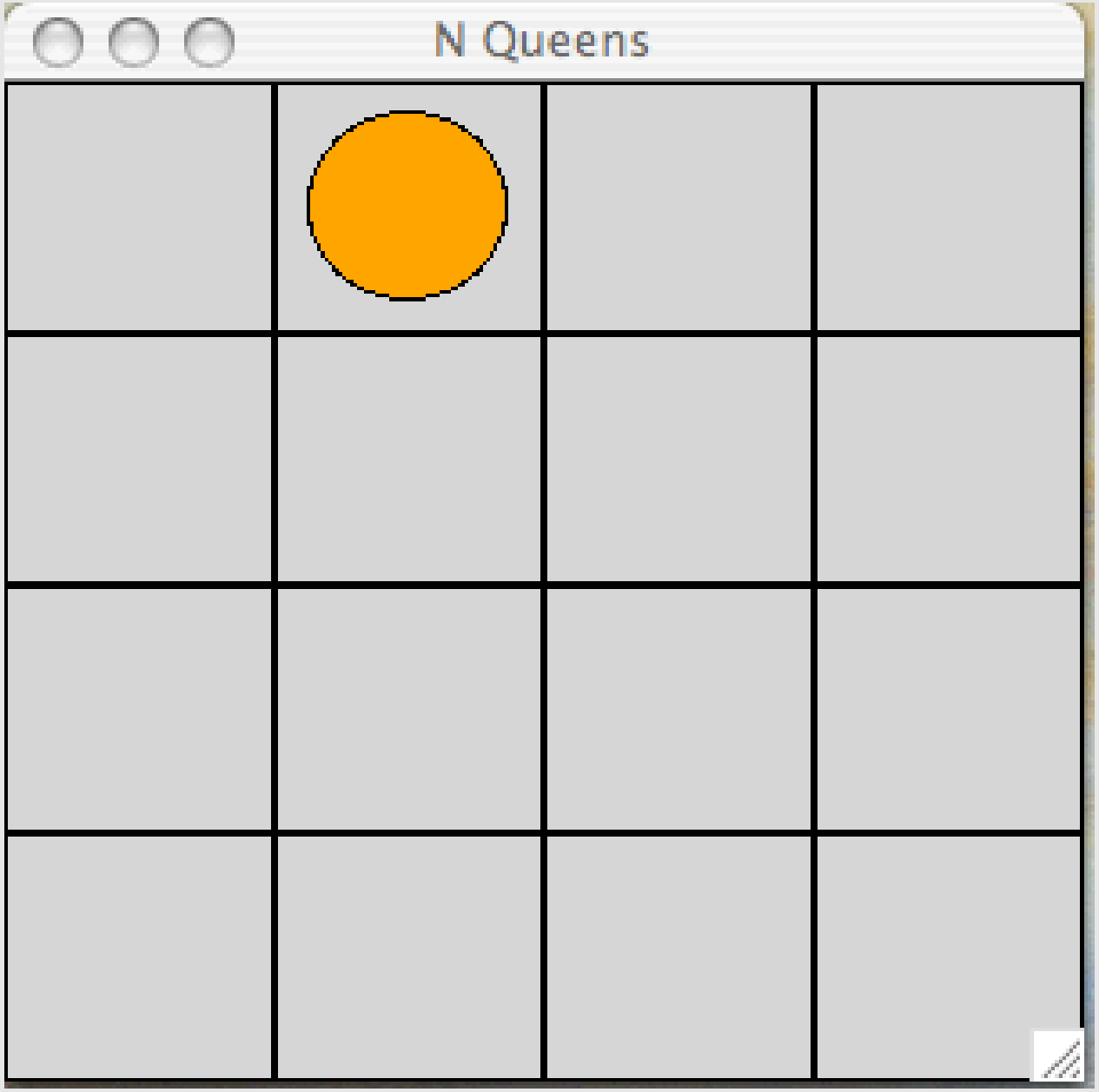

N Queens

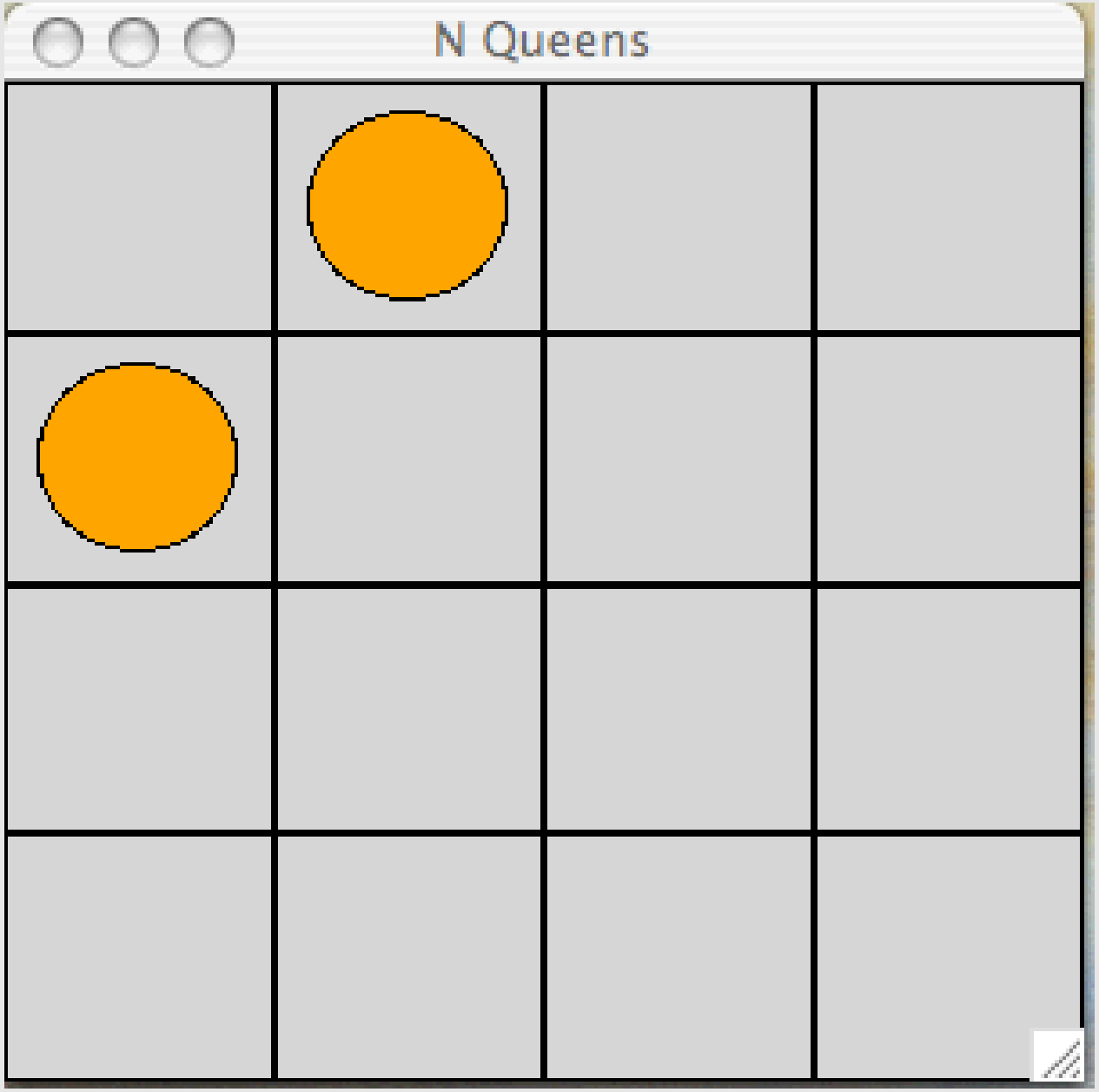

N Queens

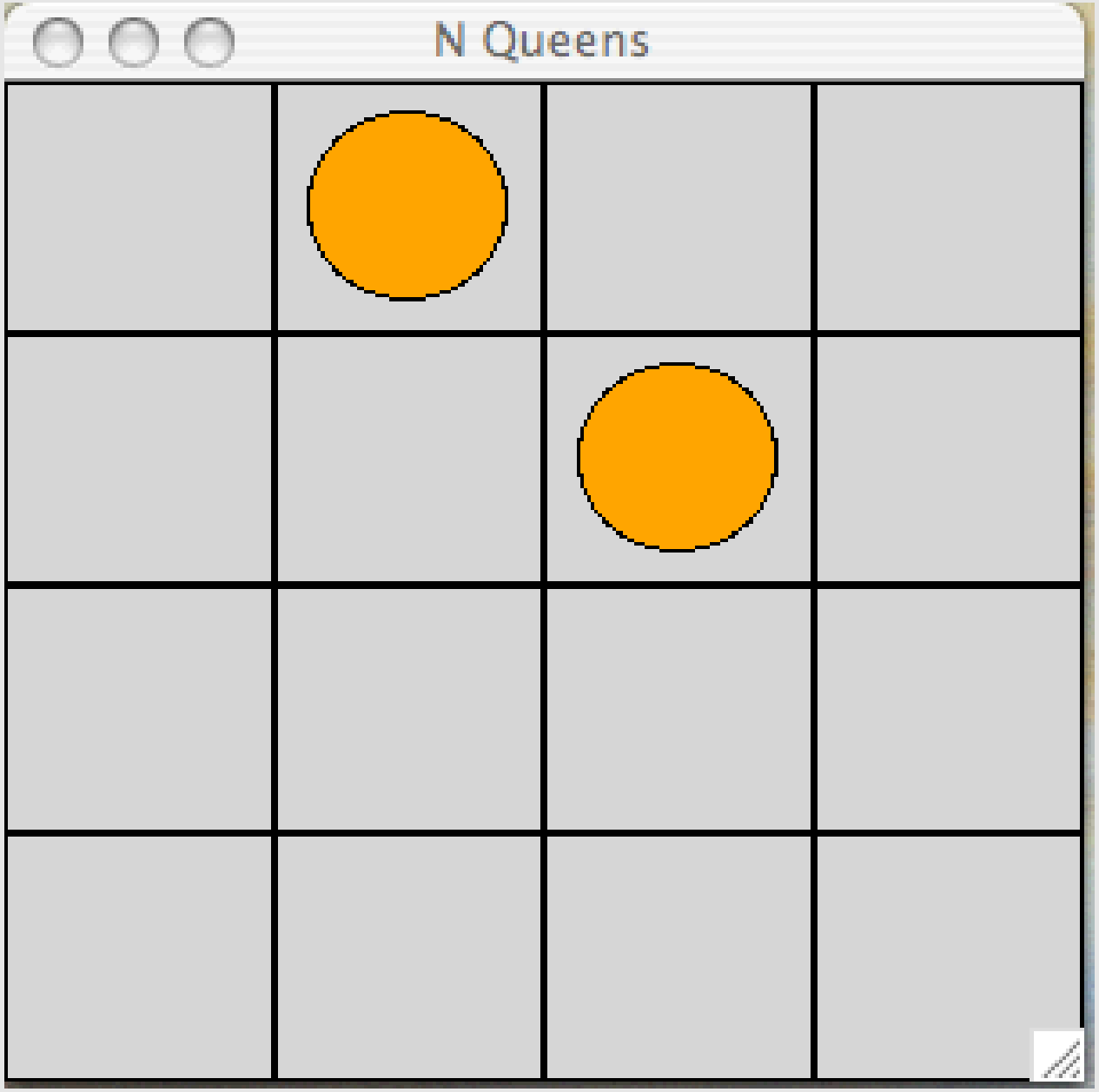

N Queens

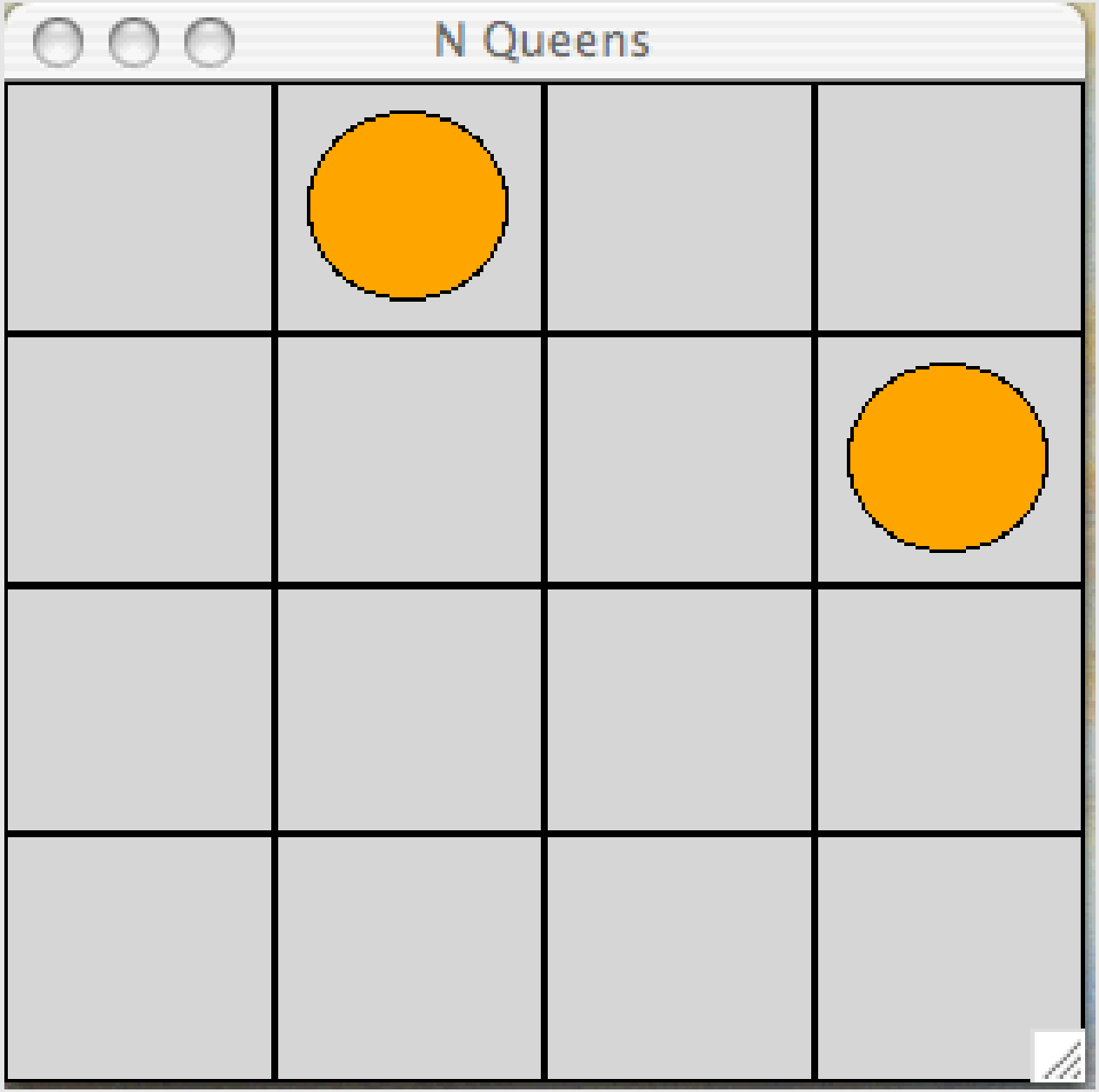

N Queens

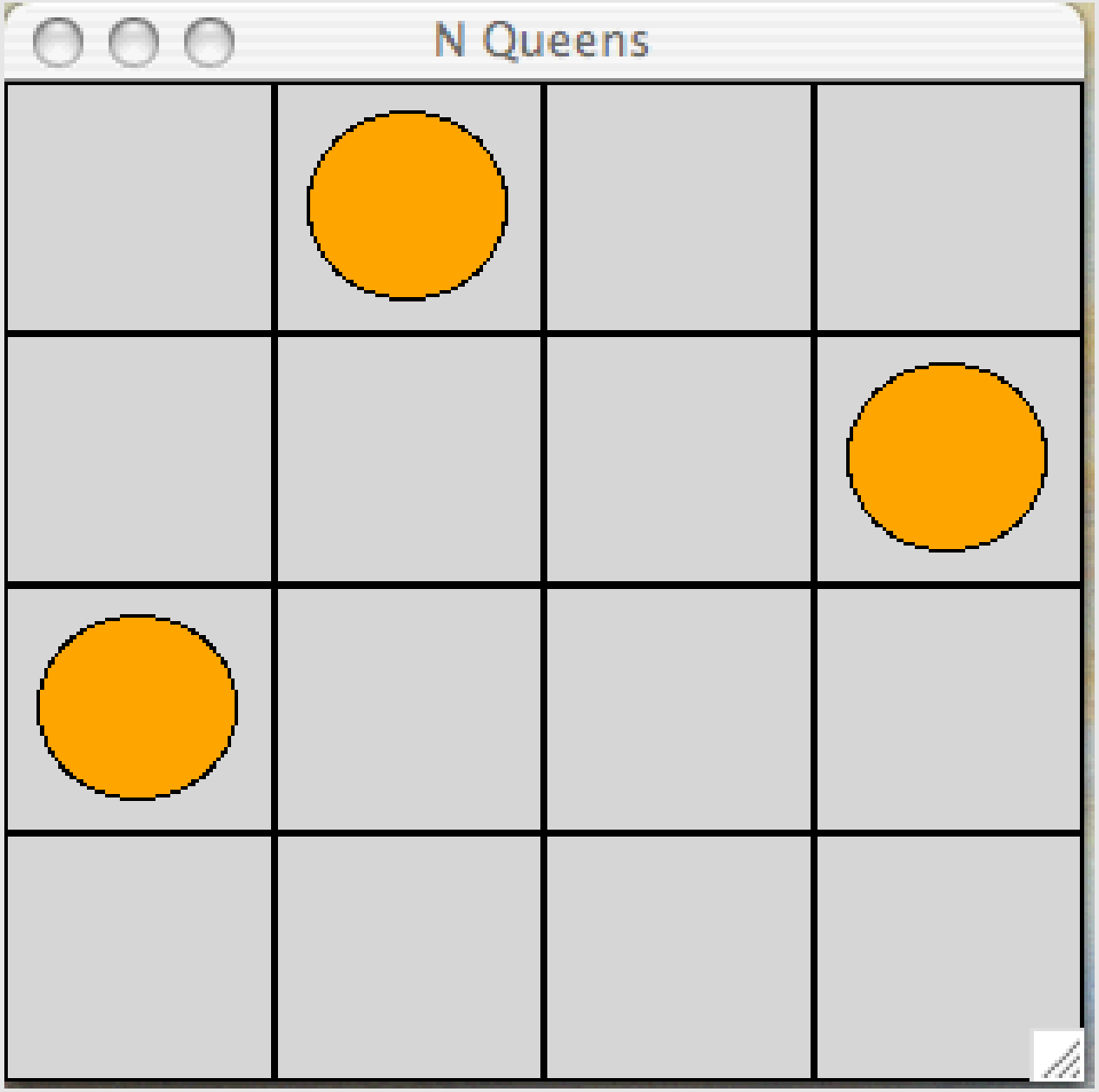

N Queens

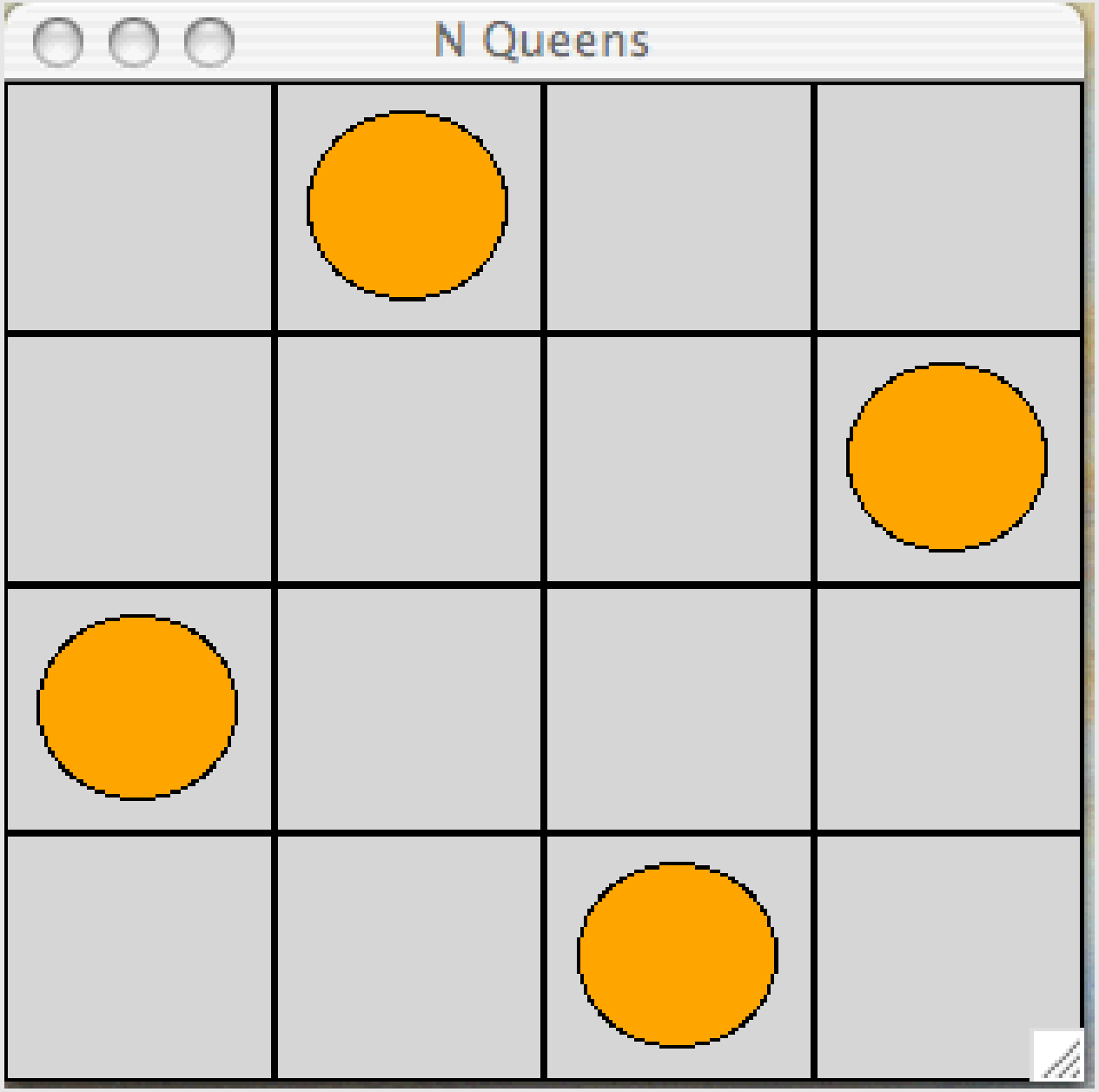

N Queens


N Queens


N Queens


N Queens


N Queens


N Queens

Suche alle Lösungen

Beispiel (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

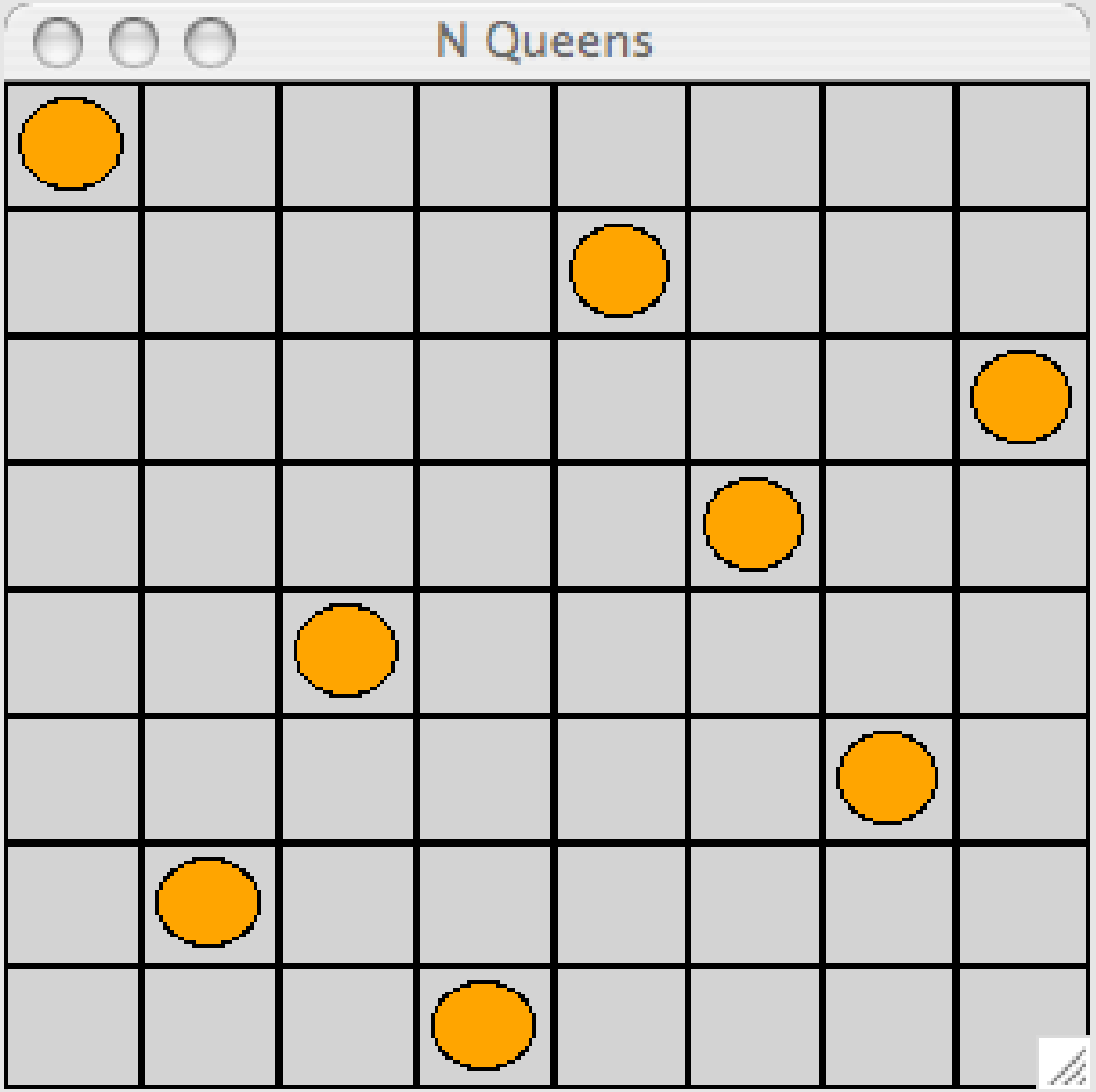
```
(define
   (try-all-queens size poss row column)
         .....
 [else ; the queen is safe, try all positions
  (apply append
         (map (curry try-all-queens
                     size
                     (if (= row 0) '()
                         (cons
                          positionToTry
                          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
```

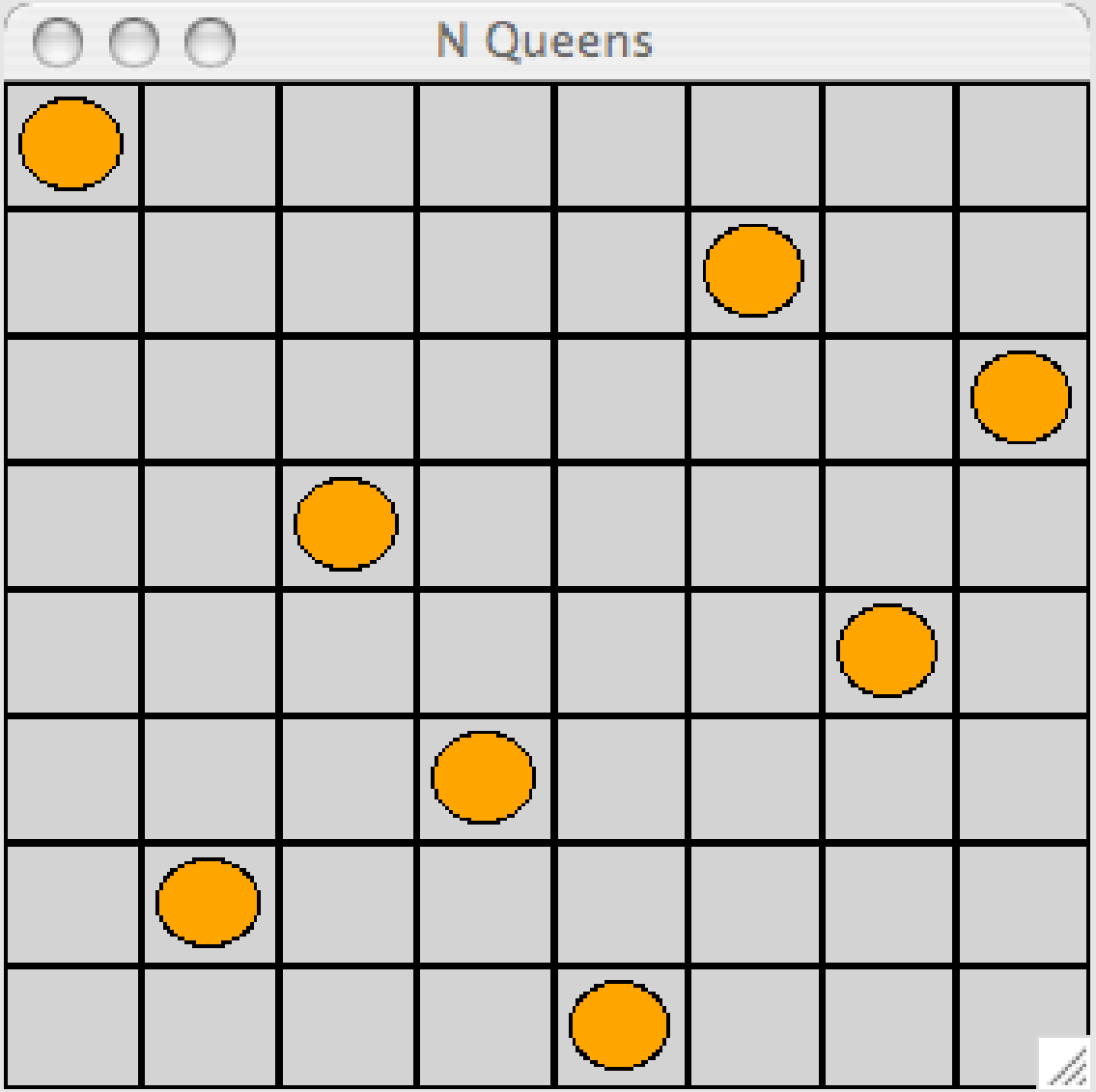
Drucken der Lösungen

```
> (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)) ...
```

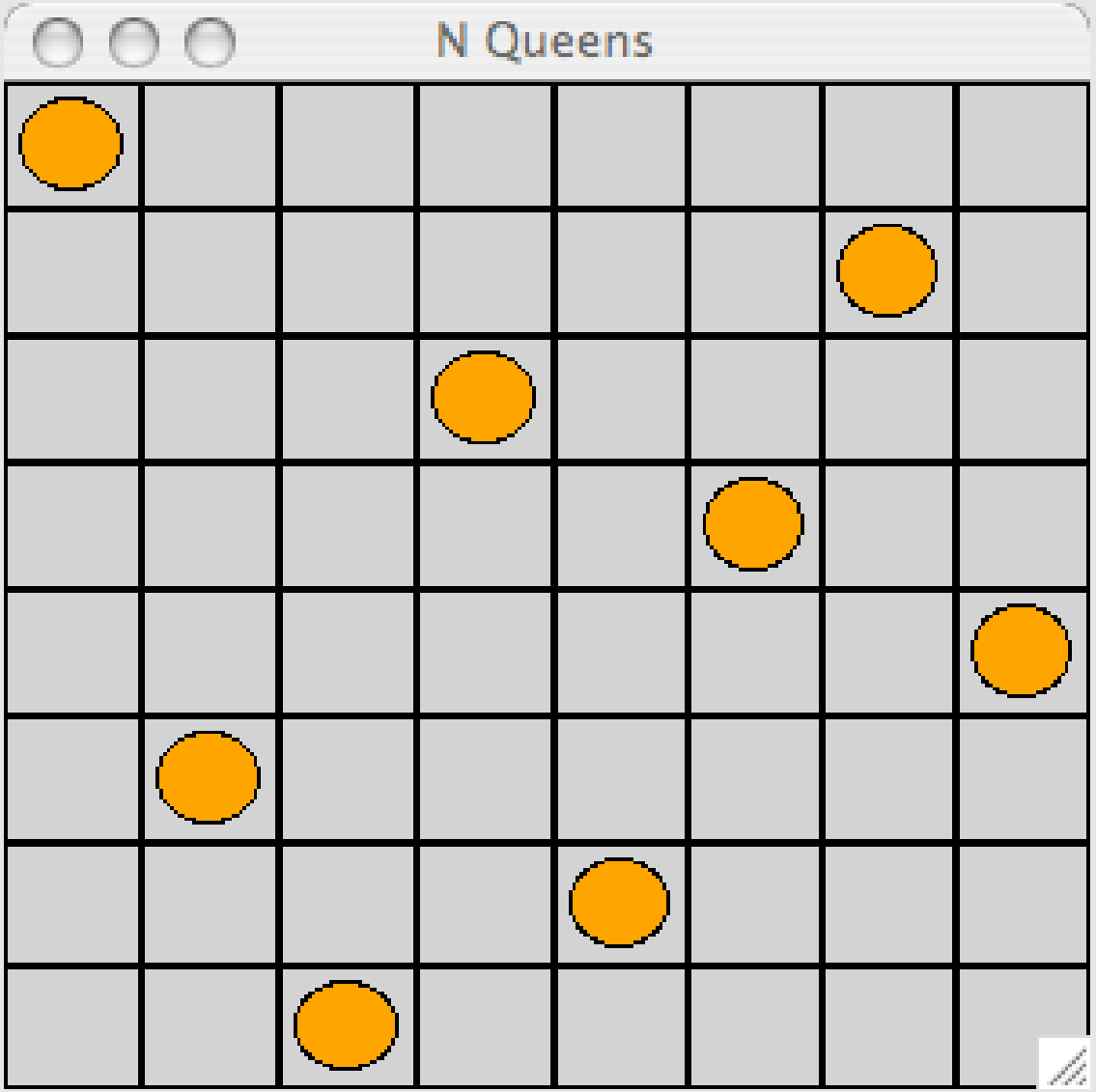
Die ersten 10 Lösungen



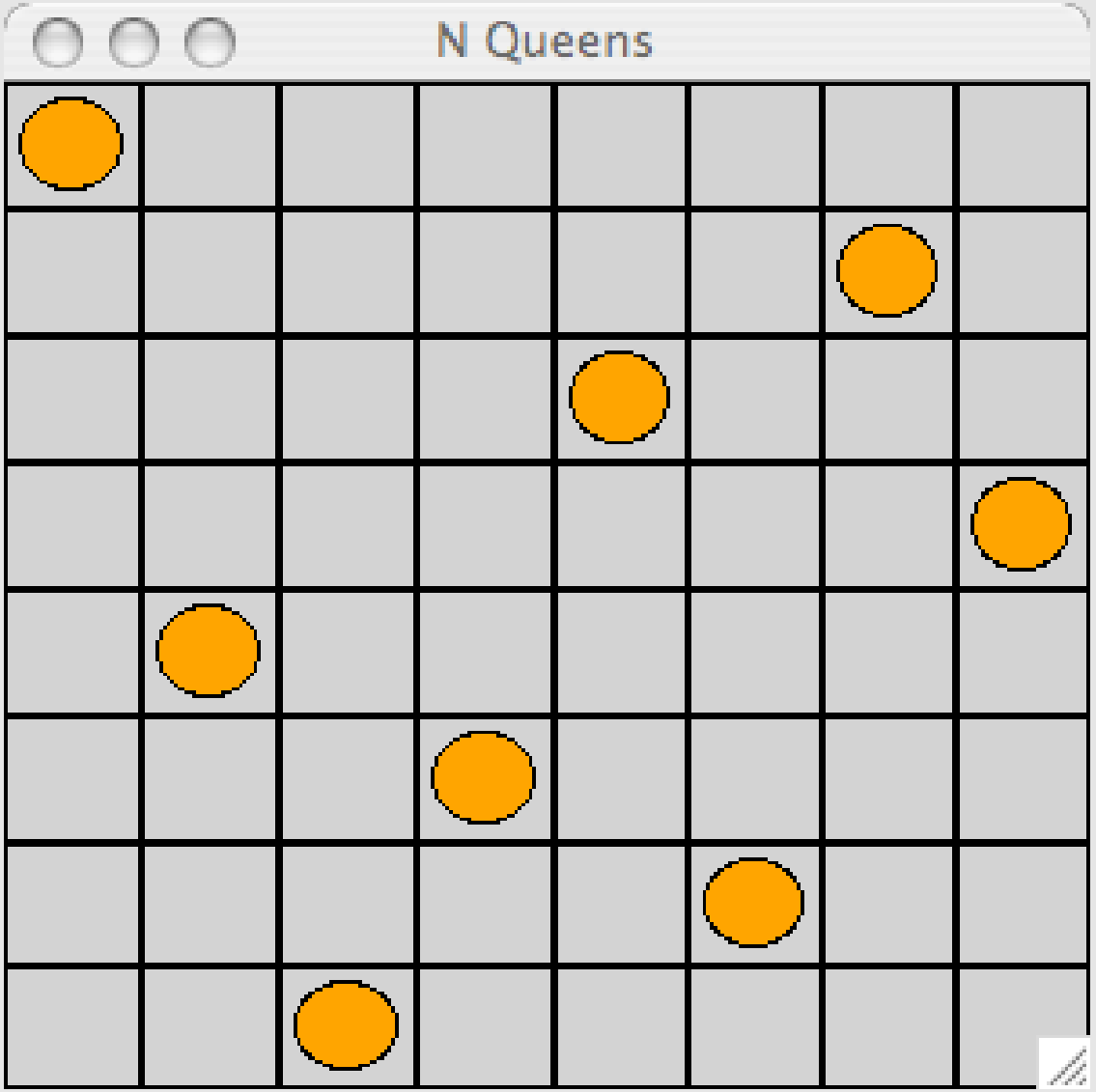
Die ersten 10 Lösungen



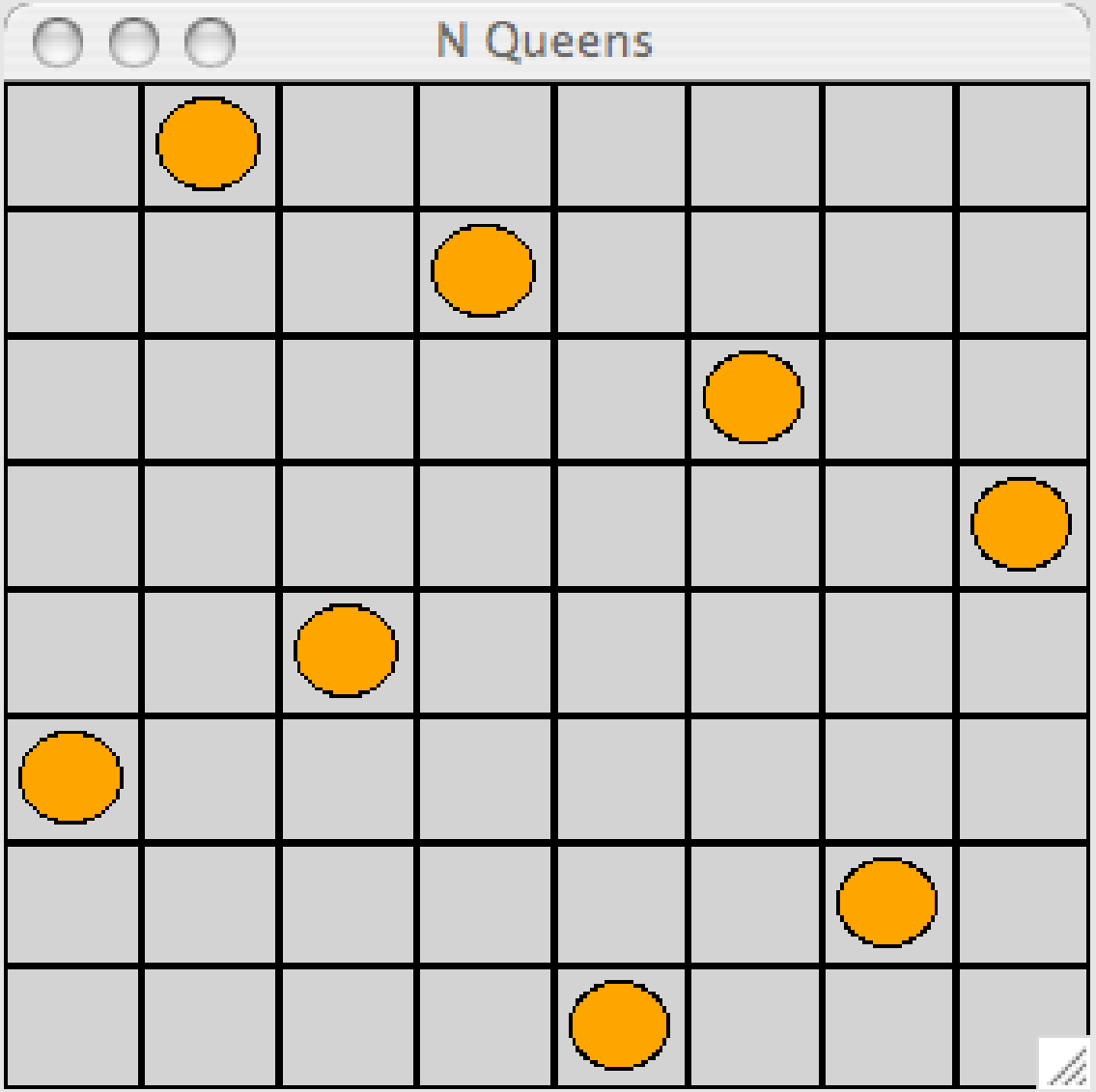
Die ersten 10 Lösungen



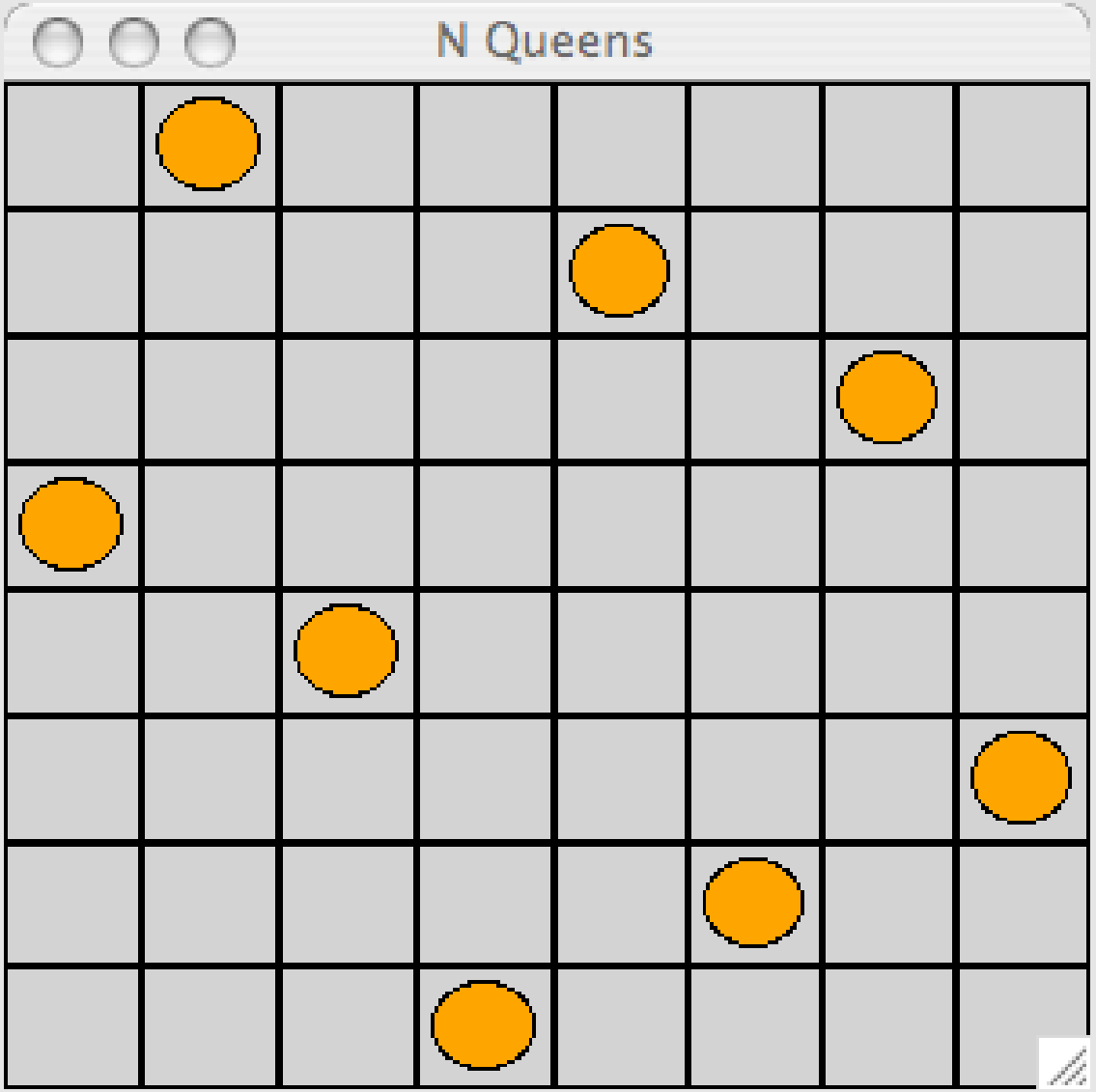
Die ersten 10 Lösungen



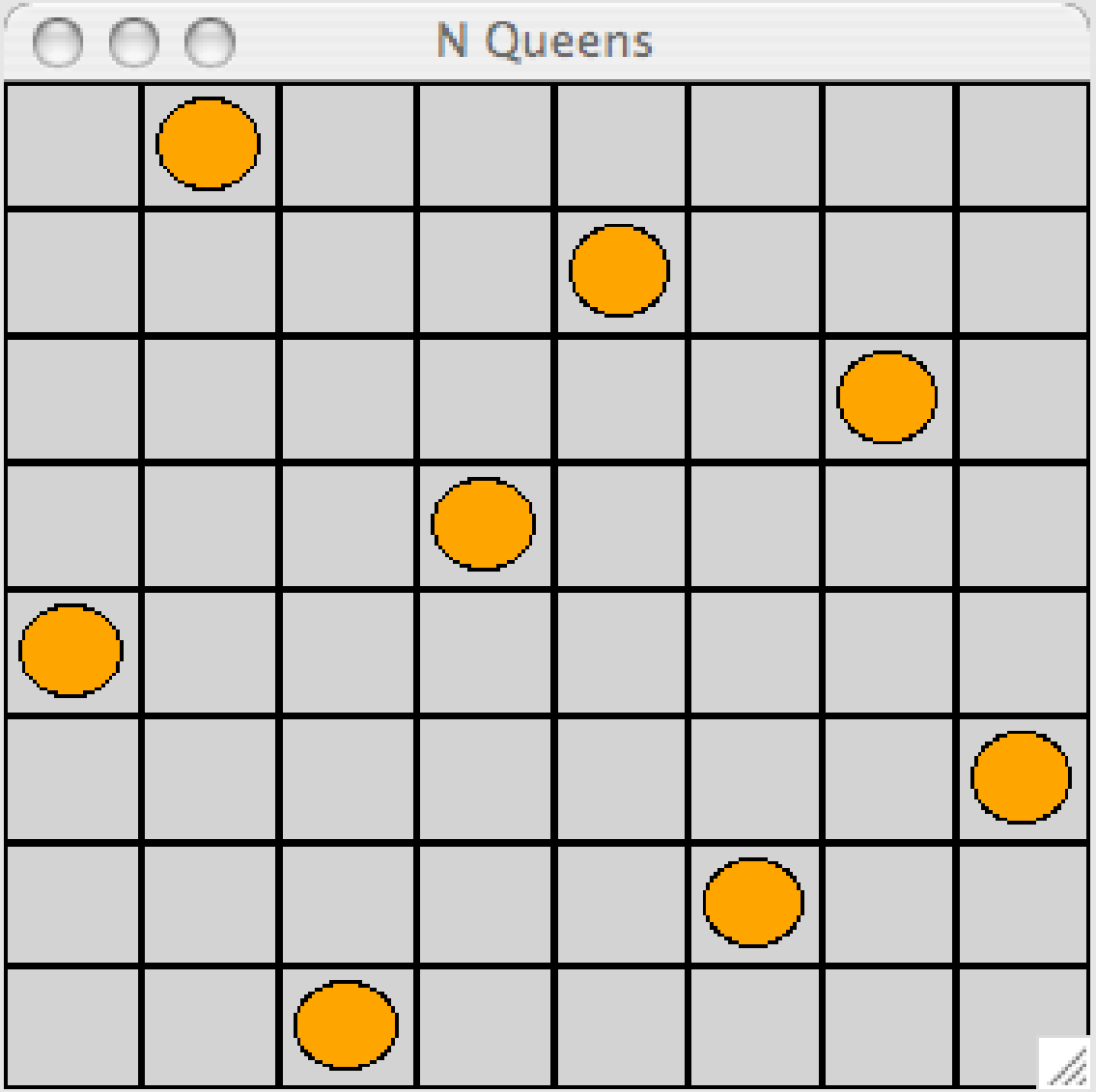
Die ersten 10 Lösungen



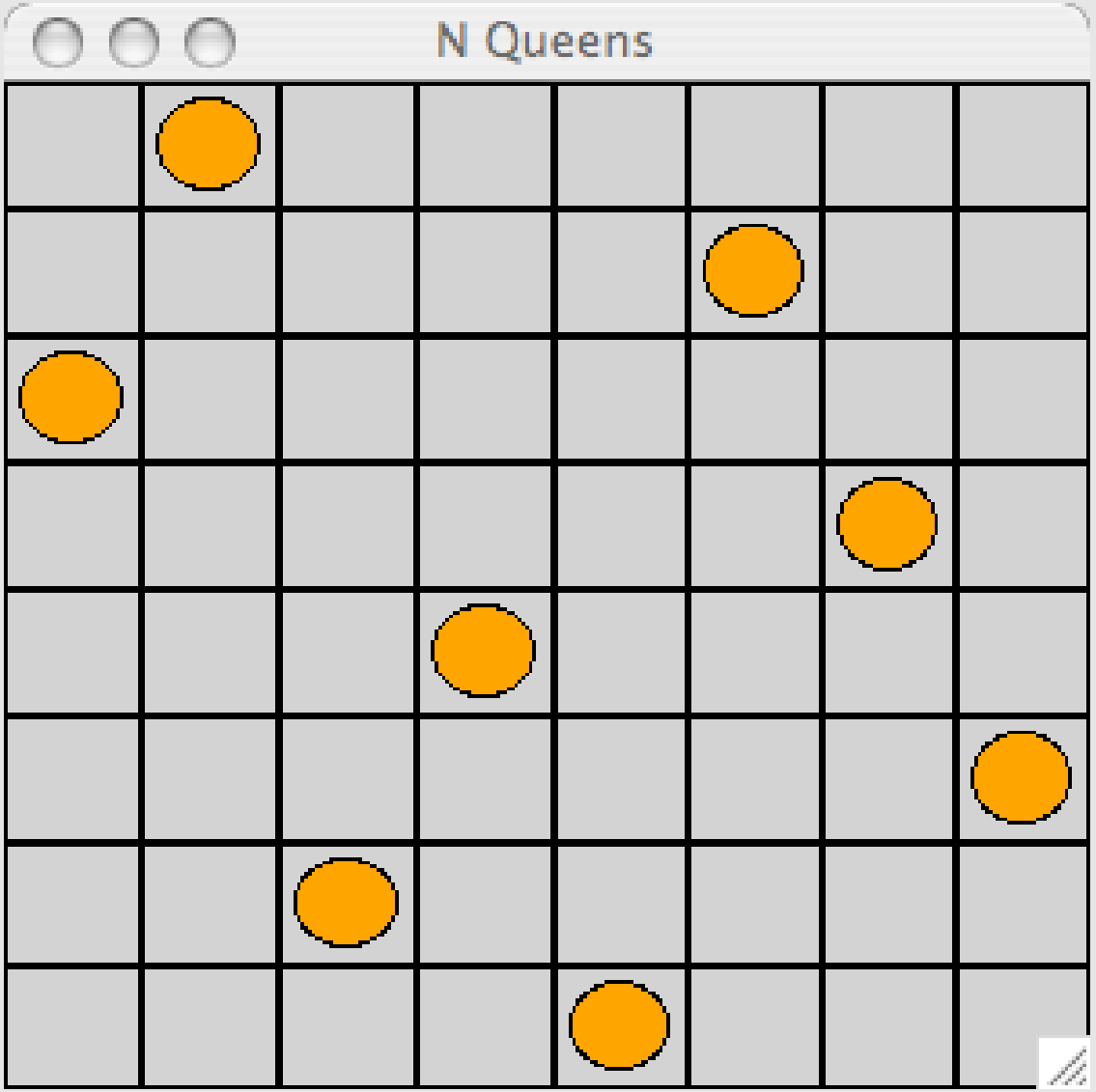
Die ersten 10 Lösungen



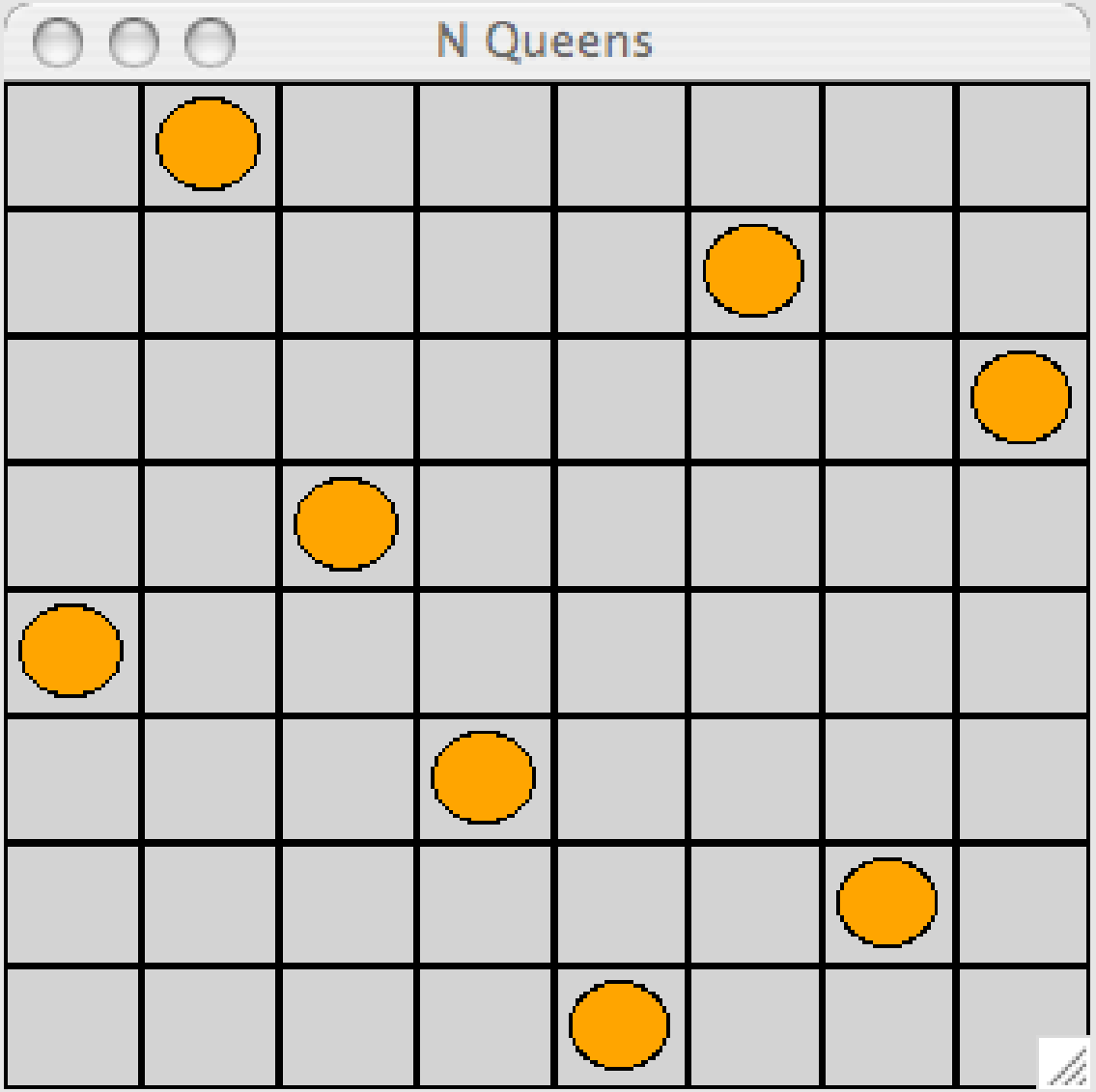
Die ersten 10 Lösungen



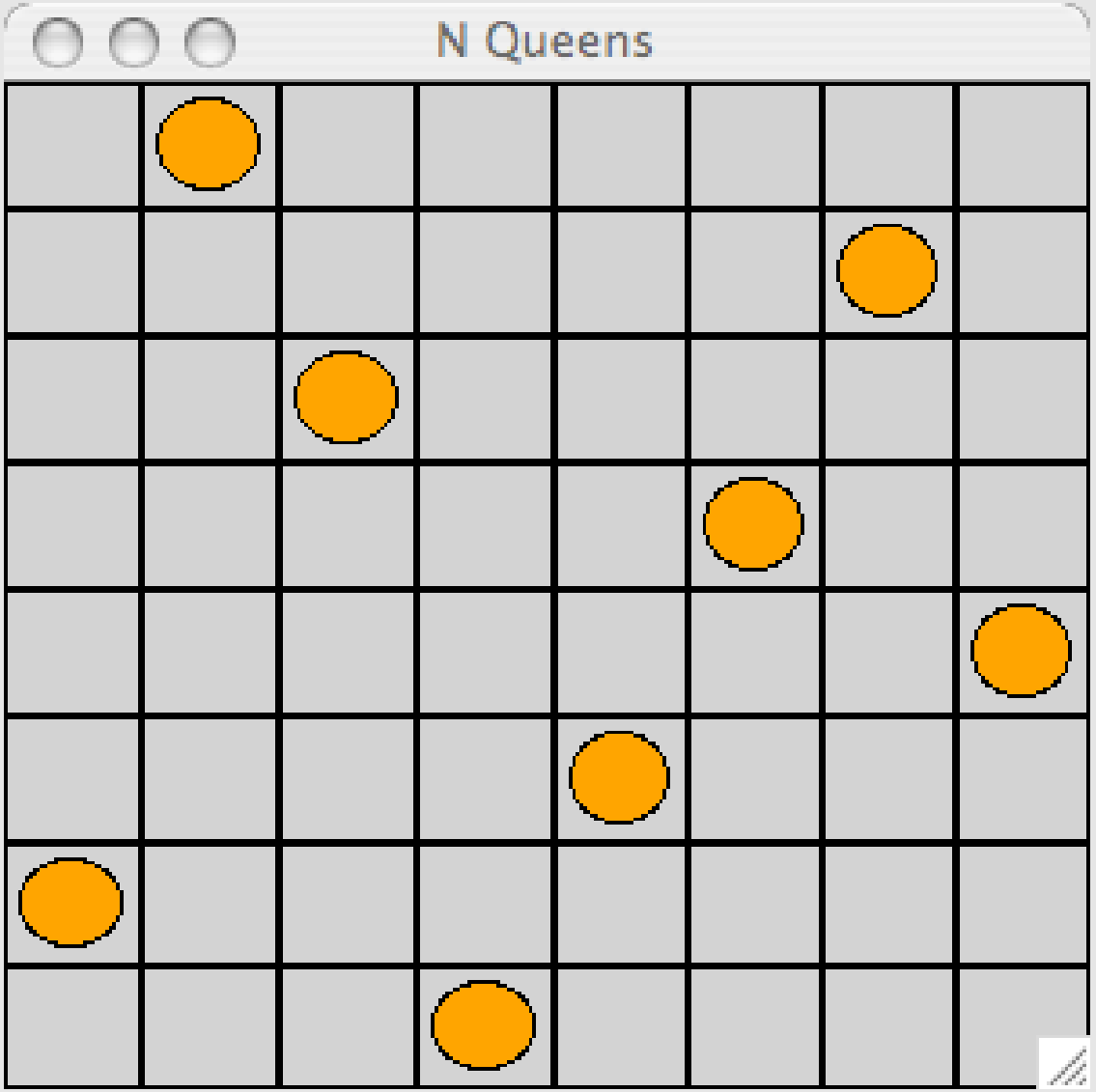
Die ersten 10 Lösungen



Die ersten 10 Lösungen



Die ersten 10 Lösungen



✎ Anmerkung:

Backtracking vs. „list of successes“

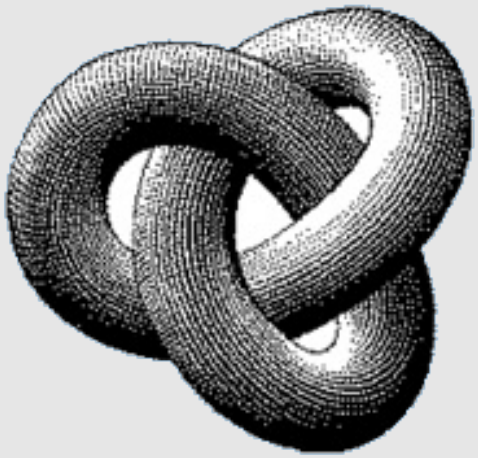
- Dem fertigen Programm sieht man nicht mehr an, daß es sich um einen backtracking-Ansatz handelt. Wir beschreiben konstruktiv und funktional, wie eine korrekte Lösung aussieht.
- Bei Funktionen in dieser Form spricht man auch vom „Erfolgslisten-Verfahren“ (list of successes).
- Der Prozeß, der durch 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:

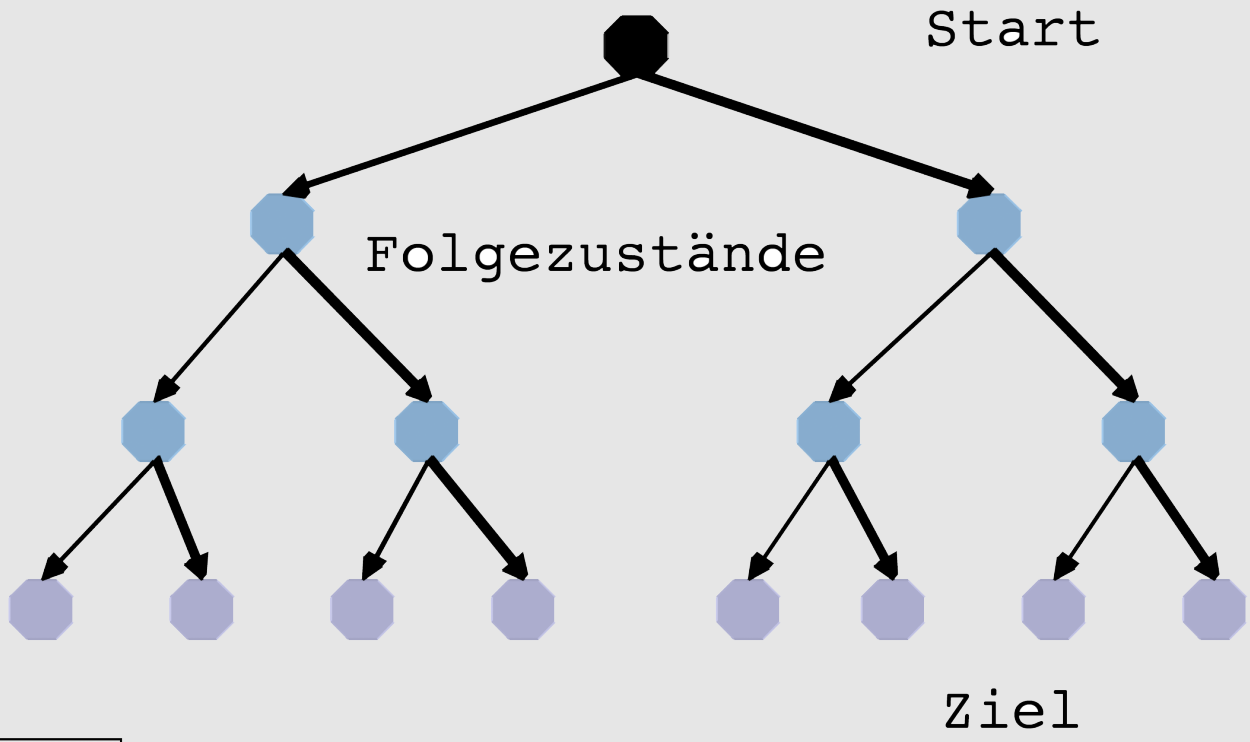
- Wenn wir nur eine Lösung suchen, ist es sinnvoller, auch nur eine Lösung zu berechnen. Das spart Rechenzeit und vermeidet Endlosschleifen, wenn unendlich viele Lösungen existieren.
- Wenn wir Lösungen vergleichen wollen und viele oder alle Lösungen benötigen, ist das Erfolgslistenverfahren besser, da der Algorithmus klarer ist.
- Ein guter Kompromiß ist ein Generator der Lösungen: er verbindet die Vorteile beider Ansätze.

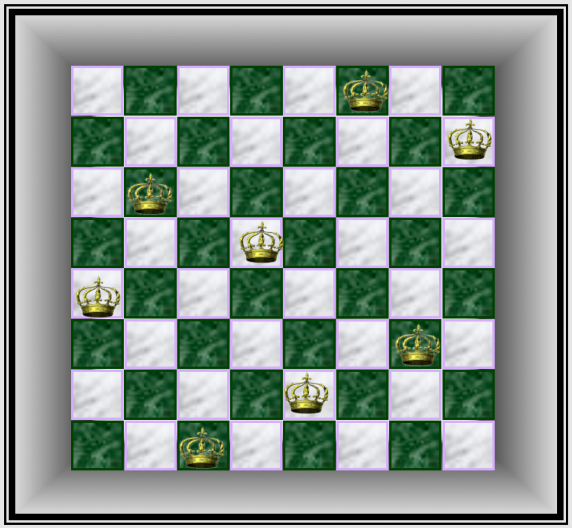
Allgemeines backtracking-Schema



M.C. Escher

Der Suchraum





Allgemeines backtracking-Schema:

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 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.

Backtracking als Funktion höherer Ordnung

Beispiel (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 $\longrightarrow$ ({ state })
3. is−legal?: state $\longrightarrow$ boolean
4. is−final−state?: state $\longrightarrow$ boolean

```
(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)))
```

Ein linearer Suchraum

Beispiel (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)
```

```
(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?
  )
```

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

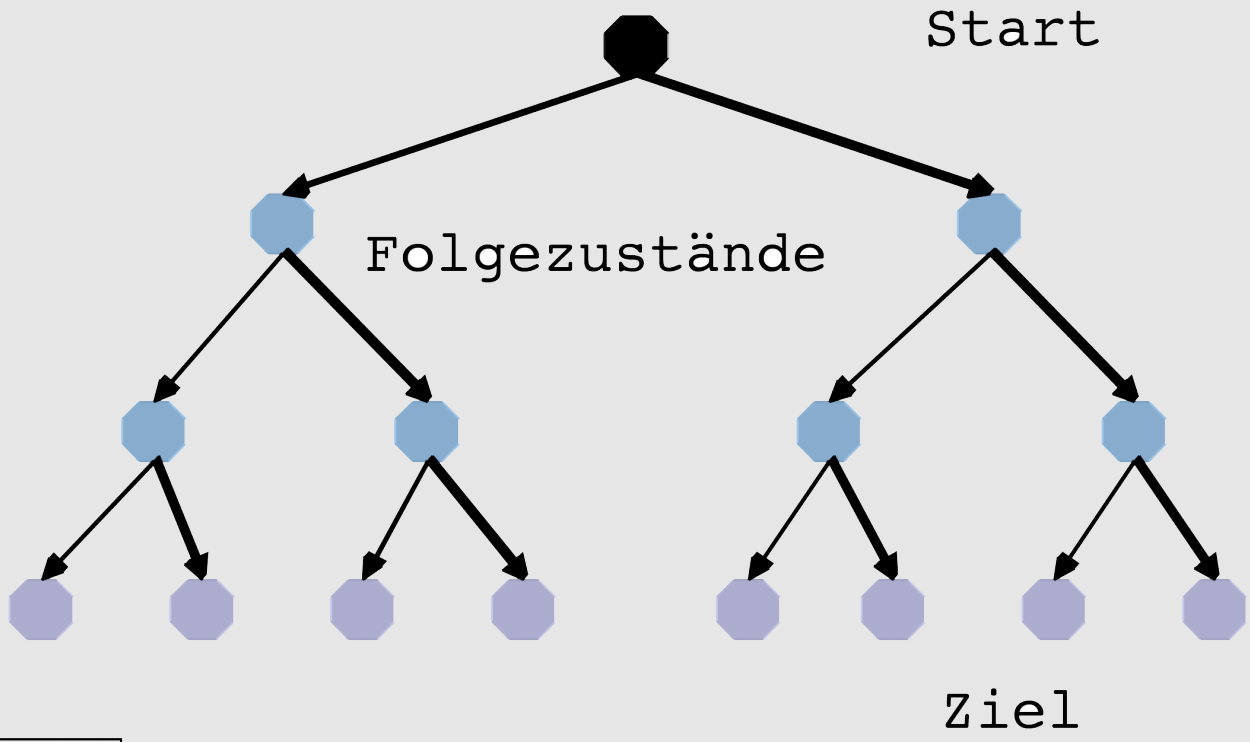
Beispiel (8-Damen)

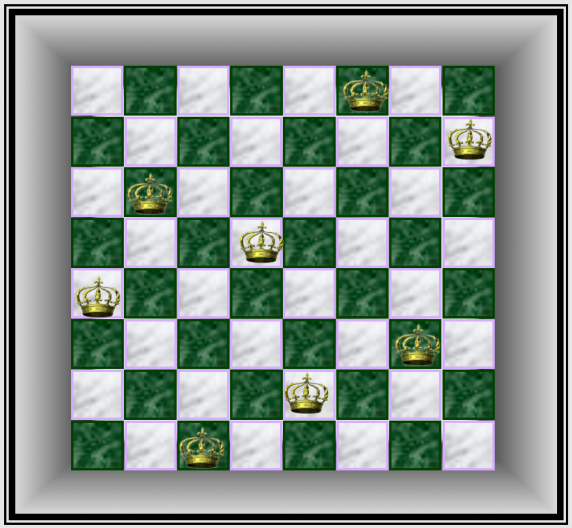
- Der Zustand wird repräsentiert als eine Liste von Koordinatenpaaren. Die Koordinaten der zuletzt aufgestellten Dame stehen am Kopf der Liste. list−of−next−positions erzeugt eine Liste von Folgezuständen, für jedes Feld der nächsten Reihe einen.
- Die Funktion position−is−safe? prüft jeweils, ob die 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?)))
```

Der Suchraum





Allgemeines backtracking-Schema:

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 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.

Missionare

Beispiel:
Missionare auf der Flucht

Beispiel:
Missionare auf der Flucht

–

Beispiel:
Missionare auf der Flucht

Beispiel:
Missionare auf der Flucht

–

Beispiel:
Missionare auf der Flucht

Beispiel:
Missionare auf der Flucht

–

Beispiel:
Missionare auf der Flucht

Beispiel:
Missionare auf der Flucht

–

Eine Denksportaufgabe

Beispiel (Das Szenario:)

- Die Missionare stehen an einer baufälligen Hängebrücke.

Eine Denksportaufgabe

Beispiel (Das Szenario:)

- Die Missionare stehen an einer baufälligen Hängebrücke.
- Diese trägt nur zwei Personen.

Eine Denksportaufgabe

Beispiel (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.

Eine Denksportaufgabe

Beispiel (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.

Eine Denksportaufgabe

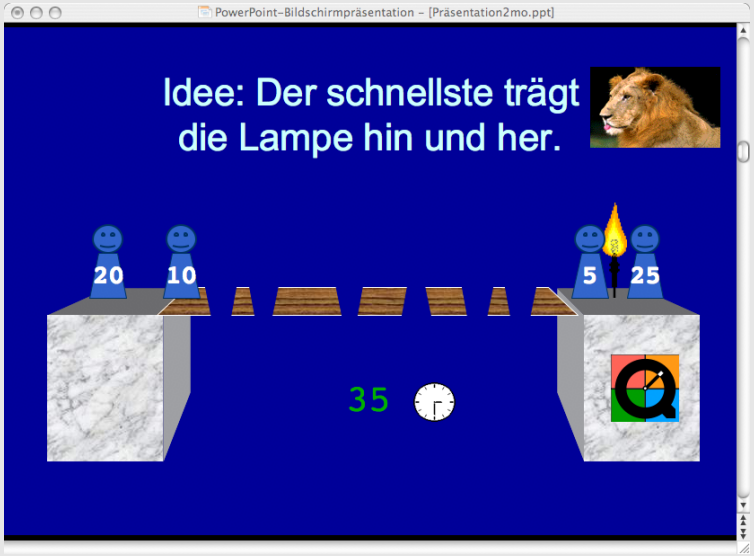
Beispiel (Das Szenario:)

- 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.

Löwe

Missionare

Suche mit Backtracking: Repräsentation



Zustände:

- Listen der Missionare links und rechts
- Wo ist die Taschenlampe?
- Restzeit

Übergangsfunktion: Wenn die Taschenlampe links ist,

$\rightarrow$ dann gehen zwei mit der Taschenlampe nach rechts,

$\leftarrow$ sonst geht einer mit der Taschenlampe nach links.

Zulässig?: Restzeit ist noch positiv.

Fertig? : Keiner steht mehr links.

Protokoll: Liste der Zustände

```
(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
```

Nichtdeterministische Suche: amb

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:

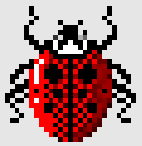
(angelic operator):

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.

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

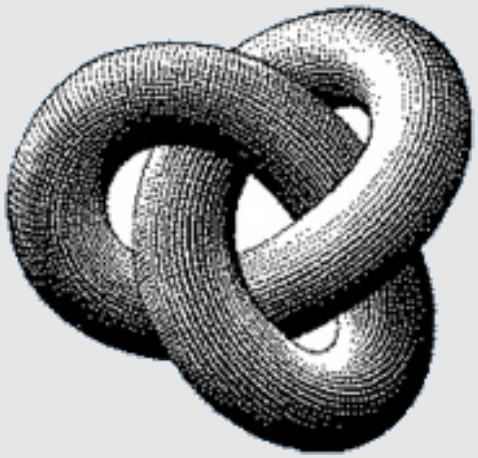
- Das Macro amb-collect verbindet alle positiven Resultate (die nicht zum fail führten) zu einer Liste.
- Wenn es unendlich viele Lösungen gibt, entsteht eine Endlosschleife.

Beispiel (Prüfen von Summen:)

Suche alle Paare von Werten a und b, deren Summe „sieben“ ergibt.

```
>  (require swindle/extra)
>  (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))
```

Färben eines Graphen



M.C. Escher

Das Vier-Farben-Problem

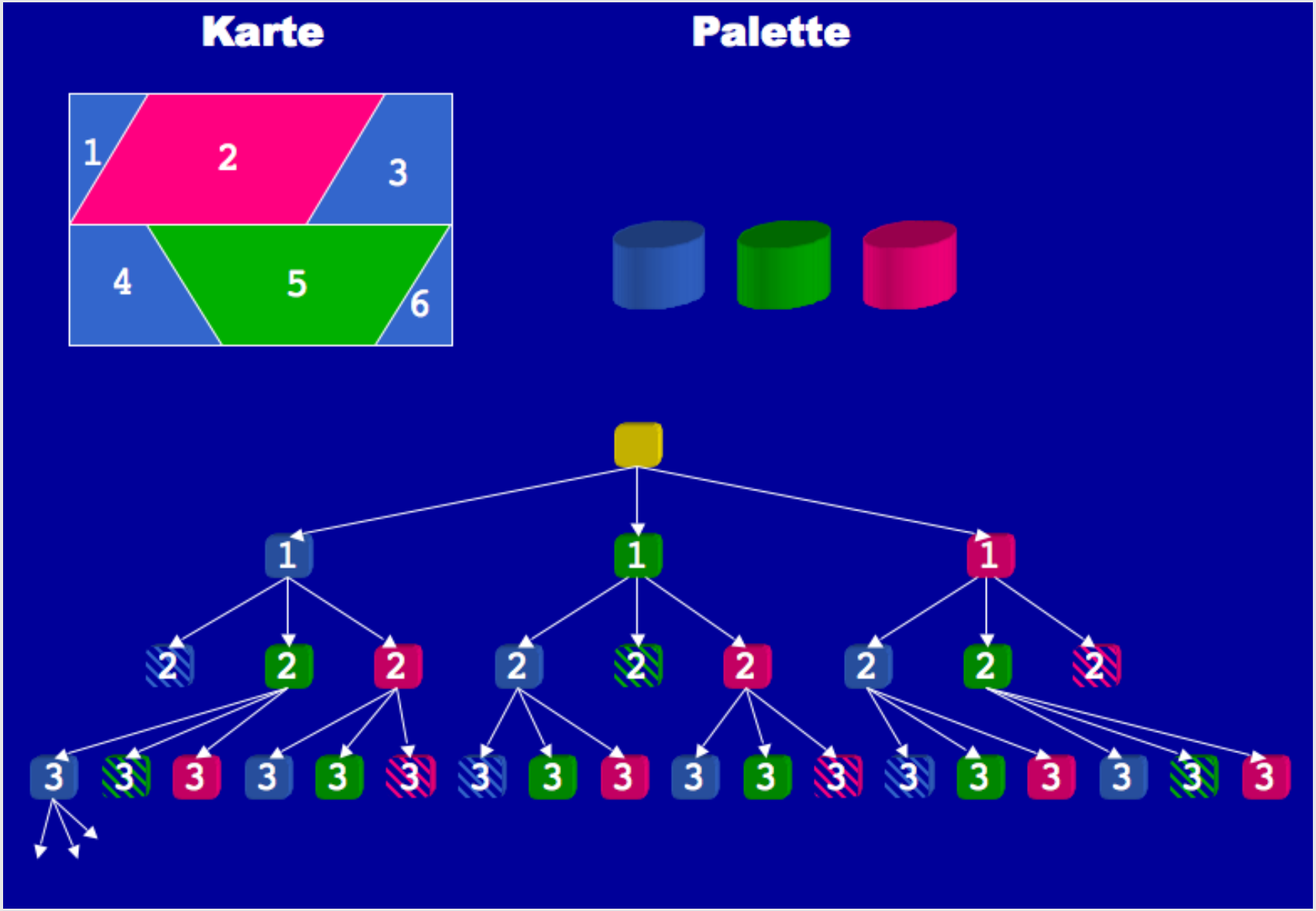
Problem (Färben einer Landkarte)

Können die Länder auf einer Landkarte mit nur vier Farben so eingefärbt werden, daß benachbarte Länder nie die gleiche Farbe haben?

Lösung

Wir durchsuchen nicht-deterministisch den Baum aller Möglichkeiten, die Länder zu färben.

Das Vier-Farben-Problem als Suchproblem



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 ....
```

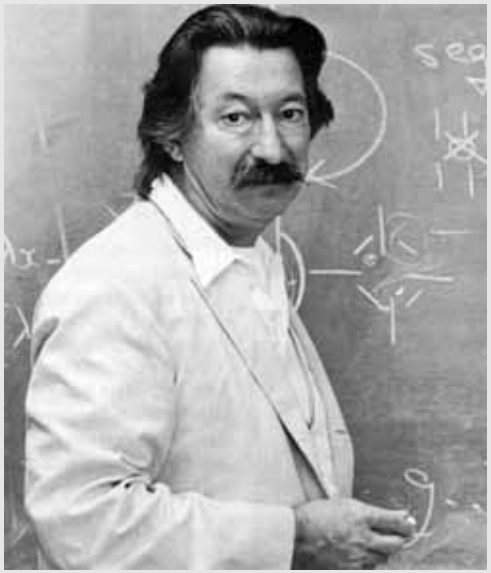
```
(define
 (general-backtracking-first-solution-only-amb
  initialState gen-states is-legal? is-final?)
(define (try st)
  (amb-assert (is-legal? st))
    ; fail, if illegal
  (cond
   [(is-final? st)
    (display "Solution: ") st]; the solution
   [else
     (let ([nextState
            (amb-car (gen-states 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

Fallstudien

Eliza



Joseph Weizenbaum

It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the area of computer programming, especially in what is called heuristic programming and artificial intelligence. For in those realms machines are made to behave in wondrous ways, often sufficient to dazzle even the most experienced observer.

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 [**?**])*

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.

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
I'ts 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

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
I'ts 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

Originaldialog

Die Muster

- Der Dialog erweckt den Anschein, als ob ELIZA den Patienten verstünde:

  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

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?

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?

Originaldialog

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.

Pattern matching im Eliza-Programm

Beispiel (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
```

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)
```

Prädikate und Semiprädikate

pat−match war bisher ein *Prädikat*. Das Resultat war **#t**, wenn der match erfolgreich war, sonst **#f**.

- Jetzt wollen wir im Falle eines erfogreichen 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))
```

Semiprädikate

Definition (Semiprädikate)

Semiprädikate 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 Semiprädikat-Problem

- Wenn Semiprädikate 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
  (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
  (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 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))
  ⟶ ((?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))
```

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 $\Longrightarrow$ 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.

```
>(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))
```

Die Regeln in Eliza

Der nächste Entwurfsschritt besteht darin, Eingabemustern mit Ausgabemustern zu assozieren.

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.

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!
```

Kontrollabstraktion und Werkzeuge

Eliza

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.

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 (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 (Abstraktion der Dialogschleife)

Eine Funktion höherer Ordung, 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))
          'ciao!))
```

Die Eingabeschleife für den Verneiner

```
(define (negation-loop)
  (dialog-tool
   (lambda ()
     (display "negator> ")
     (read))
   the-opposite
   (lambda (r) (writeln r) r)))
)
```

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.

Teil IX

Fallstudien

STUDENT: Algebraische Probleme

STUDENT

STUDENT: Algebraische Probleme

Das Programm STUDENT [**?**] 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.

Ein Beispiel

Problem (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 c
  (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.

```
(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 $f(x) = A/B$

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))
```

$$\frac{\text{Fran}}{\text{Robin}} = \frac{\text{Kelly}}{2}$$

```
(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)
```

$$3+4 = (5-(2+x))*7$$
$$(3*x)+y = 12$$

```
>(solve-equations
      '((= (+ 3 4) (* (- 5 (+ 2 x)) 7))
        (= (+ (* 3 x) y) 12)))
The equations to be solved are:
((3 + 4) = ((5 - (2 + x)) * 7))
(((3 * x) + 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)))
```

$$3 + 4 = (5 - (2 + x)) * 7$$
$$(3 * x) + y = 12$$

```
| (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
    '(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
```

Ausgewählte funktionale Algorithmen

Teil IX

Fallstudien

Fallstudien: Means-Ends-Analyse (GPS)

Nick Knatterton

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, möglicht 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 summerize 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 [**?**])*

GPS: Die Vision

☞ Ein einziges Programm, das jedes Problem lösen kann.

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.

Die Mittel-zum-Zweck-Analyse

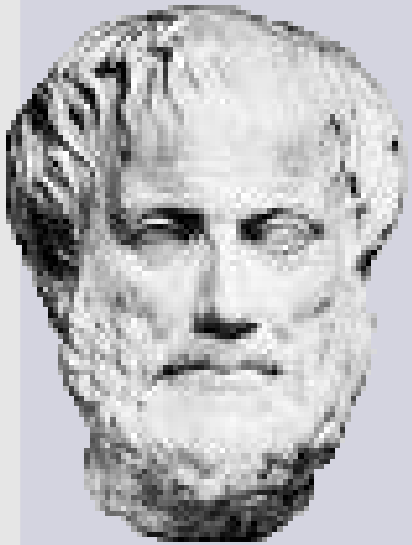
Ein Beispiel

Problem (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.

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
....
)
```

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 (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 appropiate?))))
```

appropiate?

Trägt der Operator zum Ziel bei?

```
(define (appropiate? goal op)
  ; An op is appropiate to a goal
  ; if it is in its add list.
  (member goal (op-add-list op)))
```

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))
```

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!

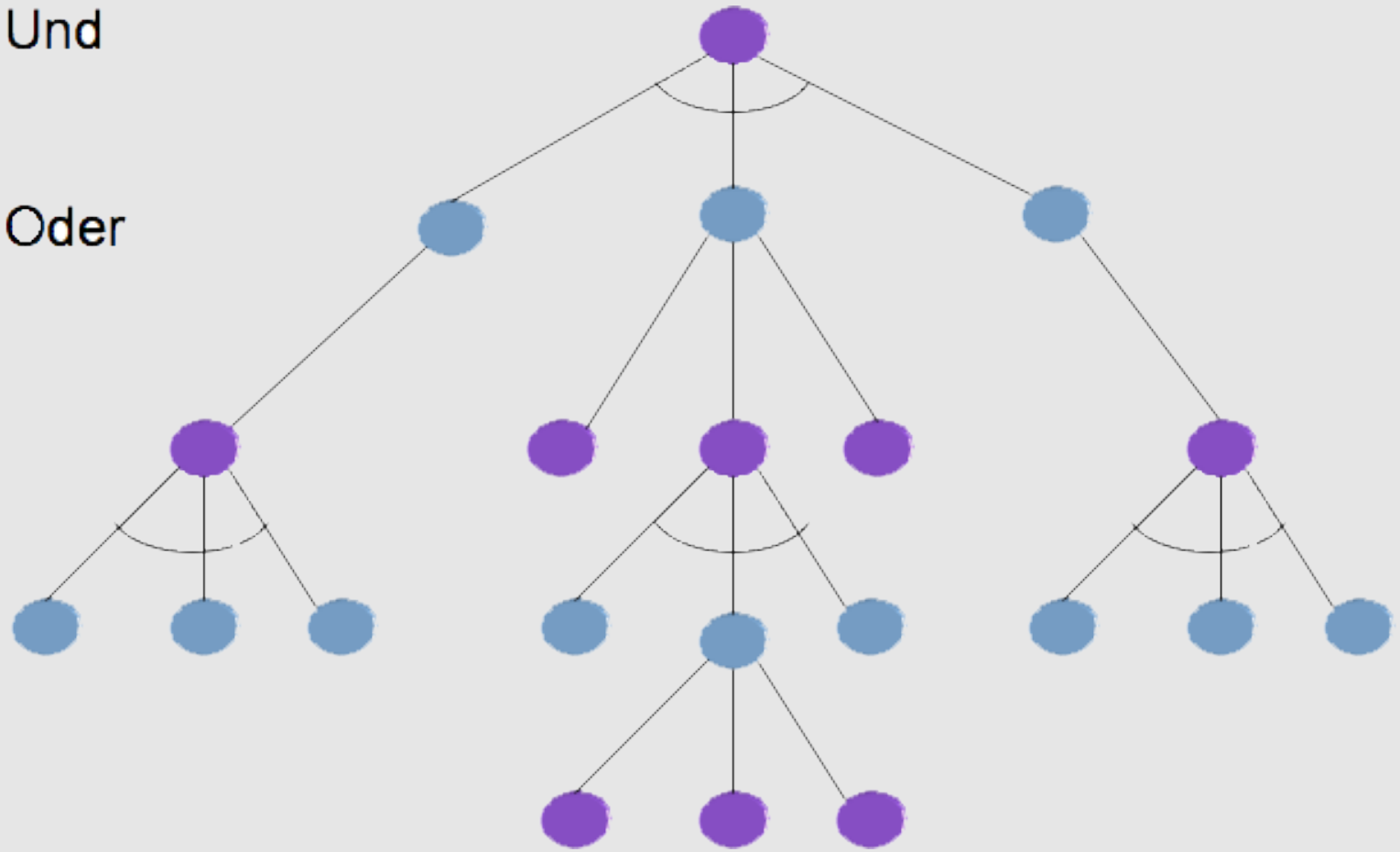
Indirekte Rekursion: apply-op und achieve:

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



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)))
```

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

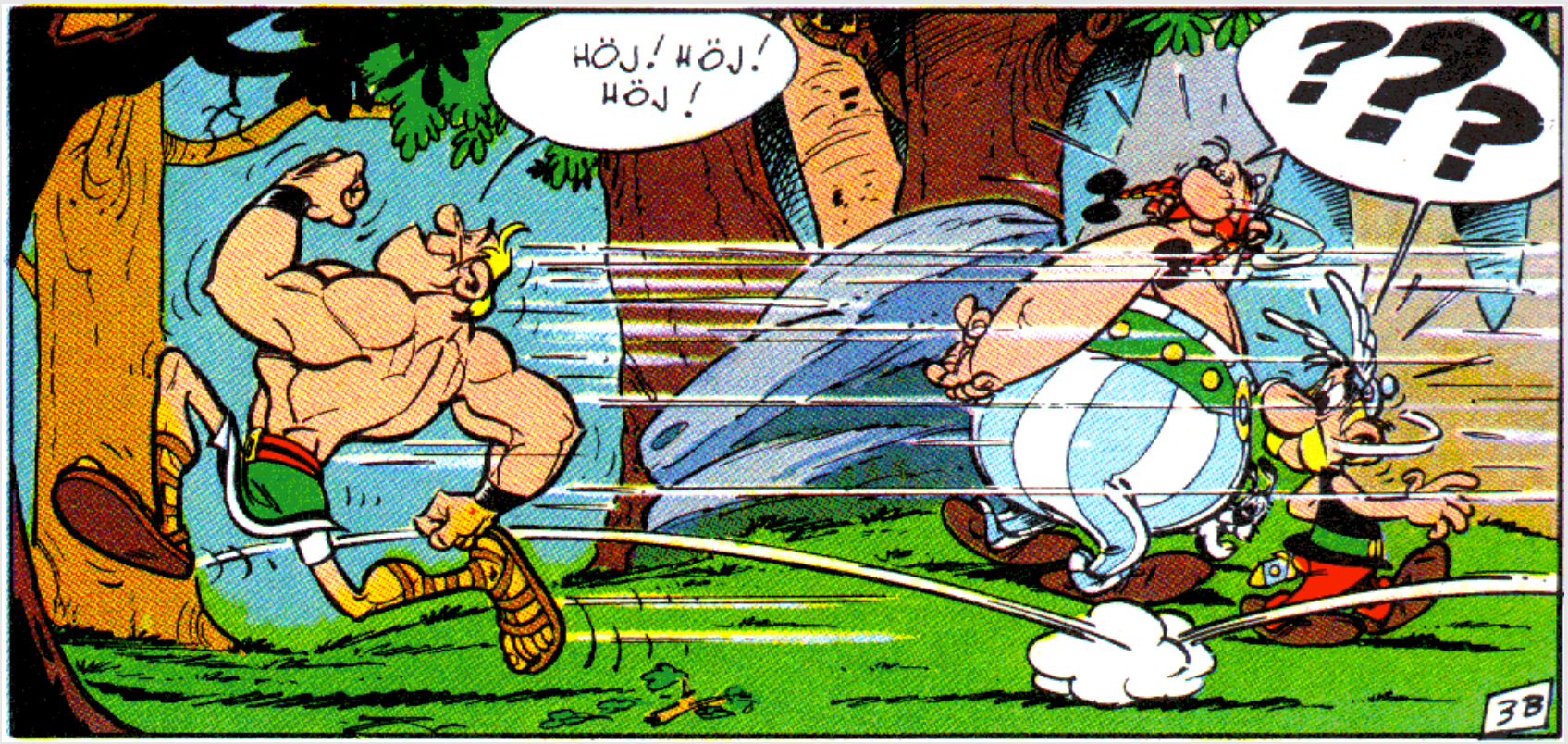
```
> (gps '(son-at-home car-works)
       '(son-at-school) *school-ops*)
(trying: son-at-school)
(trying: son-at-home)
(trying: car-works)
(executing drive-son-to-school)
solved
```

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.*

„Running around the Block“ Problem



Zustände

GPS kann keine Operatoren anwenden, die nicht den Zustand der Welt verändern.

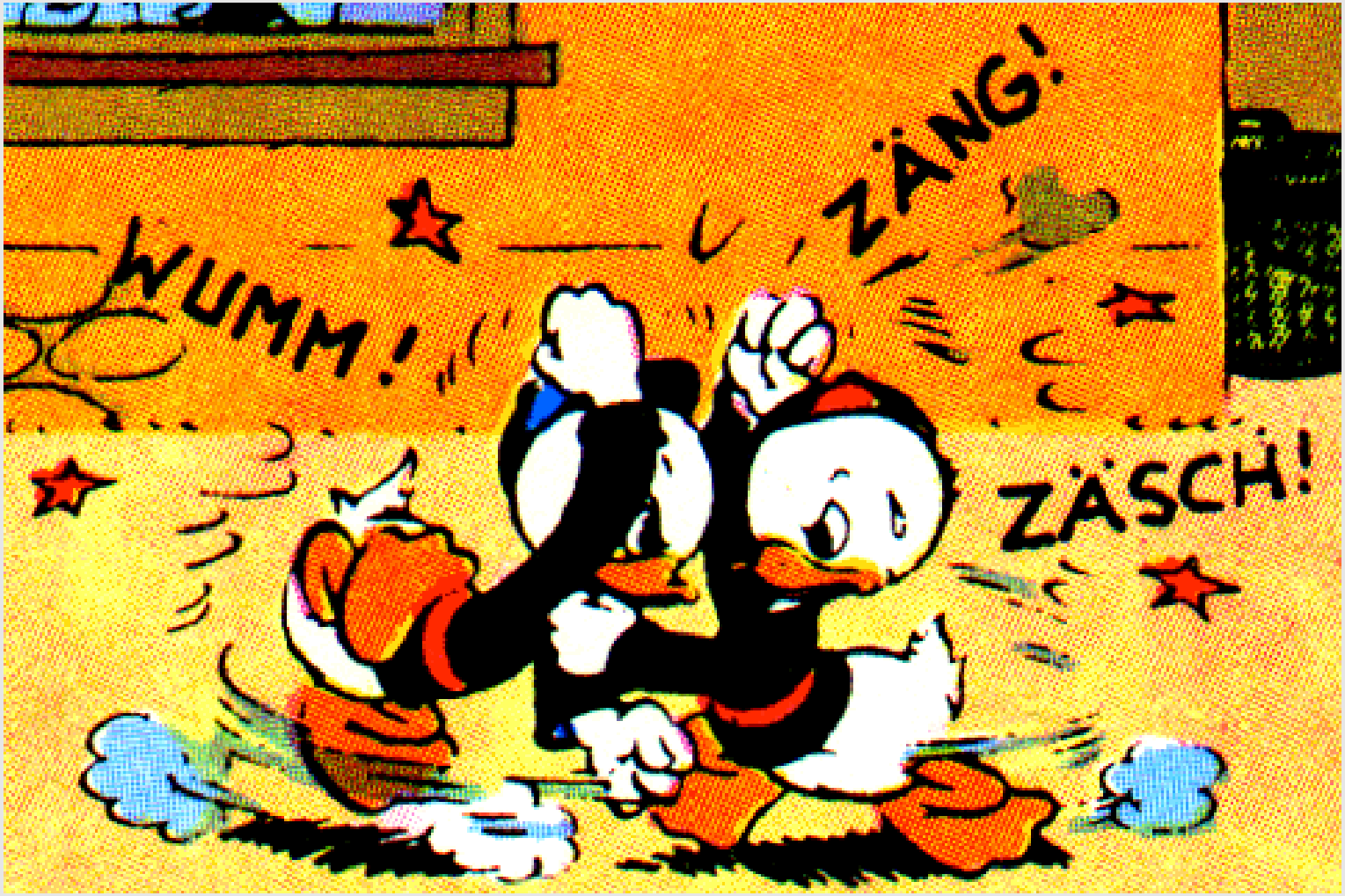
Beispiel (Um die Alster laufen:)

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 (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)
```

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:

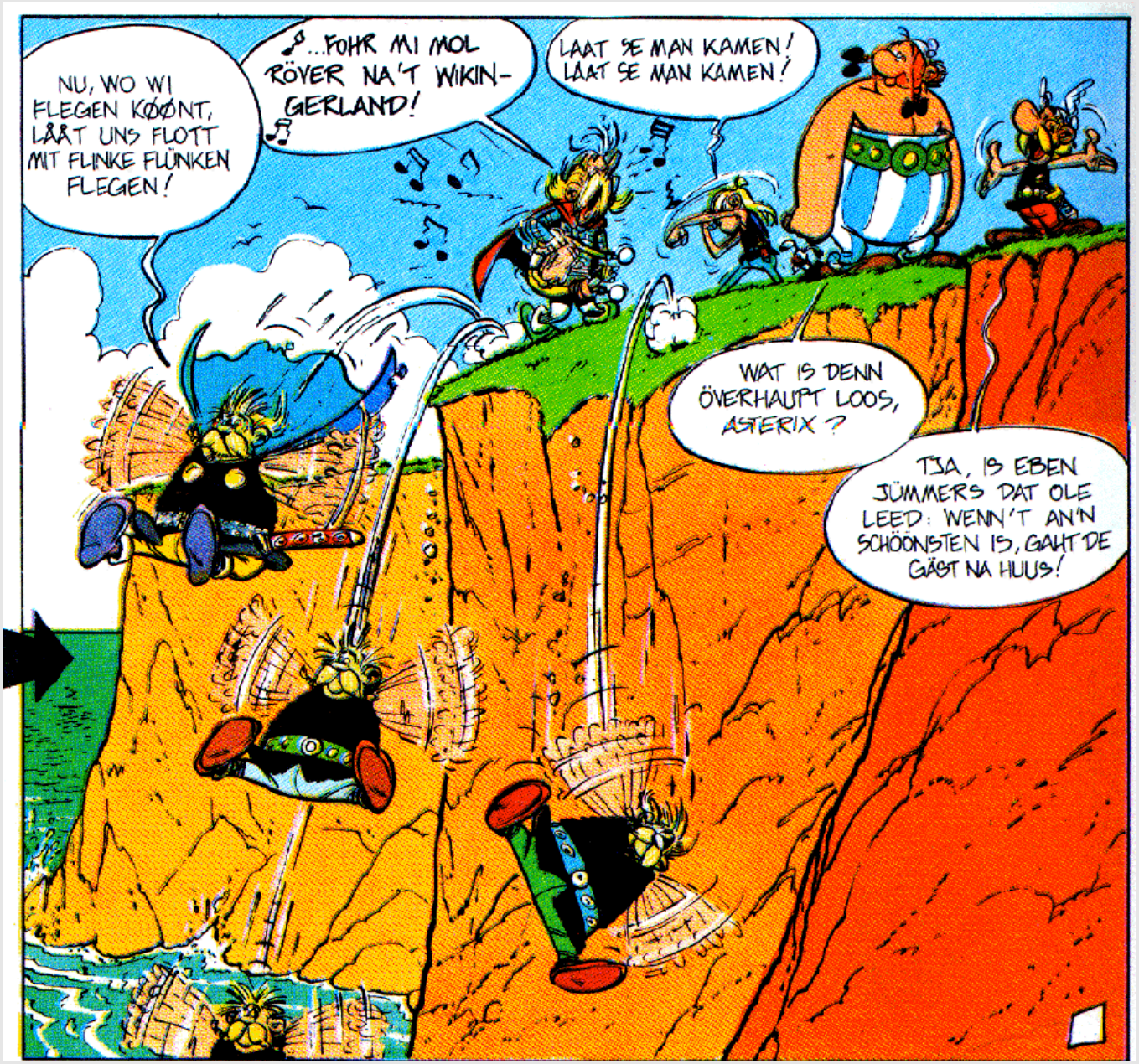
- GPS hat alle Operationen ausgeführt, um das erste Ziel zu erreichen,
- nur um dann festzustellen, daß das zweite Ziel nicht erreicht werden kann. Das kann fatal sein.

Beispiel (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

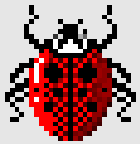


The „Recursive Subgoal“ Problem:

- 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.

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
```

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*)

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 Planungaufgaben 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))
```

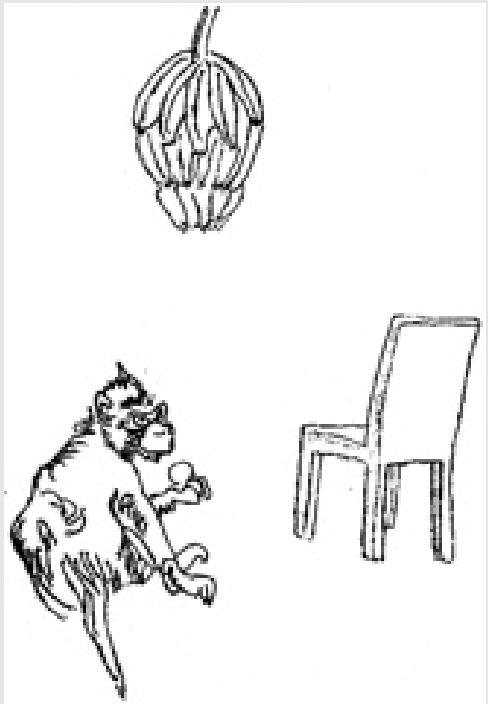
```
(define *eimer-ops* (list
   (make-ex-op 'Schaerfe-Axt
         '(Axt-ist-stumpf Habe-Axt
           Habe-Stein Stein-ist-nass)
         '(Axt-ist-scharf)
         '(Axt-ist-stumpf))
   (make-ex-op 'Benetze-Stein
         '(Habe-Stein Habe-Wasser
           Stein-ist-trocken)
         '(Stein-ist-nass)
         '(Stein-ist-trocken))
   (make-ex-op 'Hole-Wasser
         '(Habe-Eimer Eimer-ist-heil)
         '(Habe-Wasser)
          '()) ))
```

Ein Loch ist

```
(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 $\longrightarrow$ ()

Fallstudien: Means-Ends-Analyse (GPS)



Monkey and Banana: Ein klassisches KI-Problem

Beispiel (Affe und Banane)

Gegeben sei das folgende Szenario [**?**]:

Ein hungriger Affe steht bei der Tür eine 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))
```

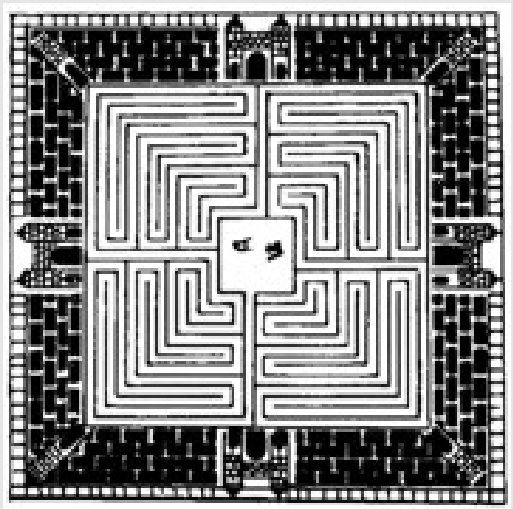
```
(make-ex-op 'grasp-bananas
    '(at-bananas empty-handed)
    '(has-bananas)
    '(empty-handed))
(make-ex-op 'drop-ball
    '(has-ball)
    '(empty-handed)
    '(has-ball))
(make-ex-op 'eat-bananas
    '(has-bananas)
    '(empty-handed not-hungry)
    '(has-bananas hungry))))
```

```
> (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))
>
```

Suche einen Weg aus dem Irrgarten

Labyrinth-Graffiti in Pompeii; der Schriftzug lautet:
„LABYRINTHUS HIC HABITAT MINOTAURUS“
(Labyrinth, hier wohnt Minotaurus).

 [3]

 [4]

 [5]

Beispiel

- Repräsentation: Ein Raster von verbundenen, numerierten Orten.
- Die Verbindungen werden durch eine Nachbarschaftsrelation beschrieben.

[3] Achteckiges Labyrinth aus der Kathedrale von Amiens, Seitenlänge 5.2m

[4] Römisches Fußbodenmosaik aus Orbe (Schweiz), ca. 200 v.u.Z.

[5] Fußbodenmosaik 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:

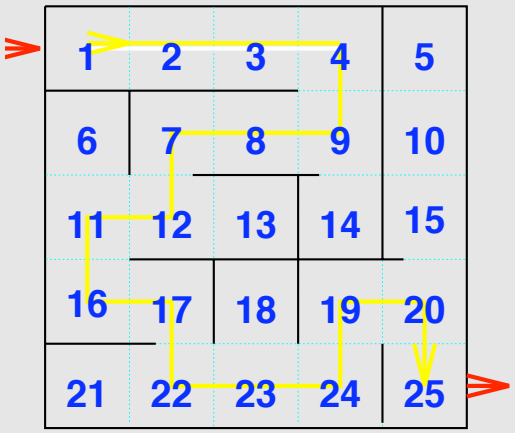
```
(move from x to y)
```

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))
```

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)))
```

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)
```

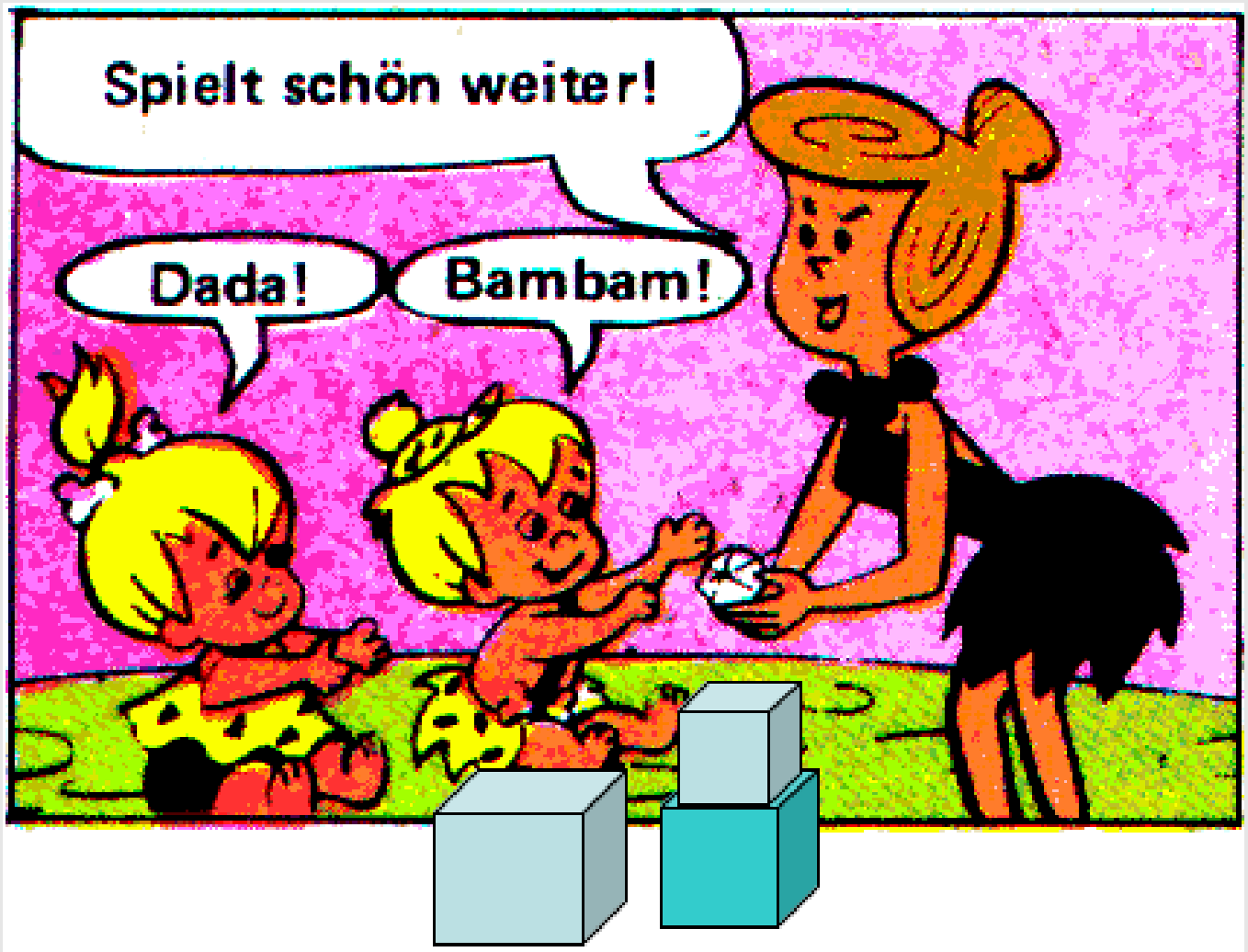
✎ Anmerkung:

EinVorteil 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.

Die Klötzchenwelt (blocks world)

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:
Bewege 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

Gegeben sei eine Menge von Blöcken $a, b, \cdots$ auf dem Tisch oder übereinander gestapelt:
Welche Bewegungen sind möglich?

Lösung (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.*

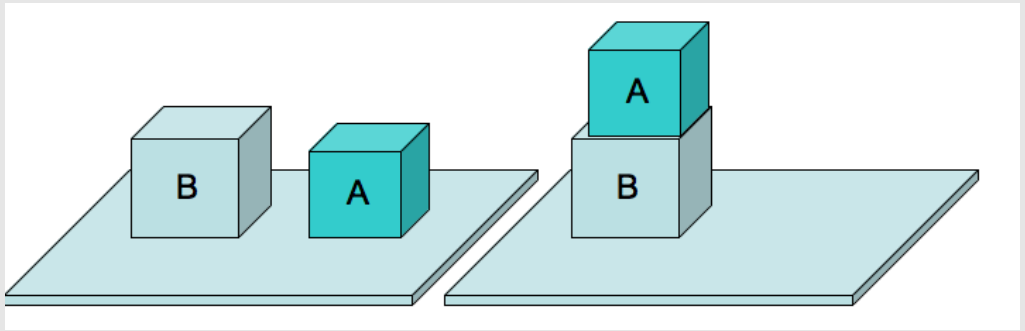
Beobachtung

- 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.

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-
    ))))
```

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

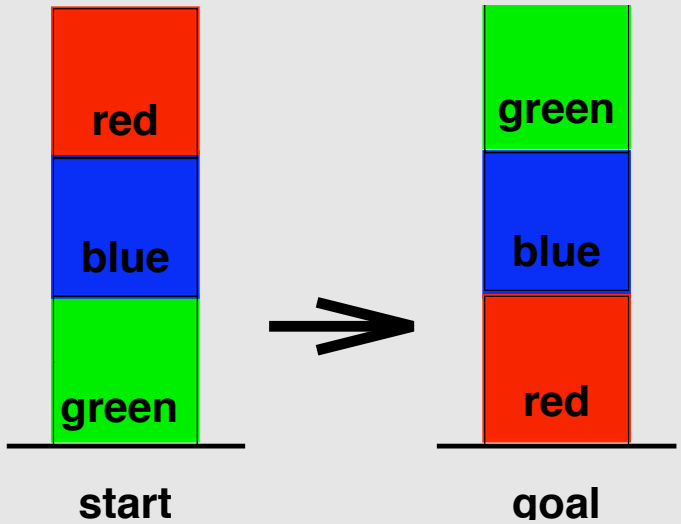
```
>
(gps2
  '((red-block on blue-block)
    (blue-block on green-block)
    (green-block on table)
    (space on red-block)
    (space on table))
  '((green-block on blue-block)
    (blue-block on red-block))
   (make-block-ops
         '(red-block green-block
           blue-block)))
-> (); keine Lösung
```

Keilerei der Ziele

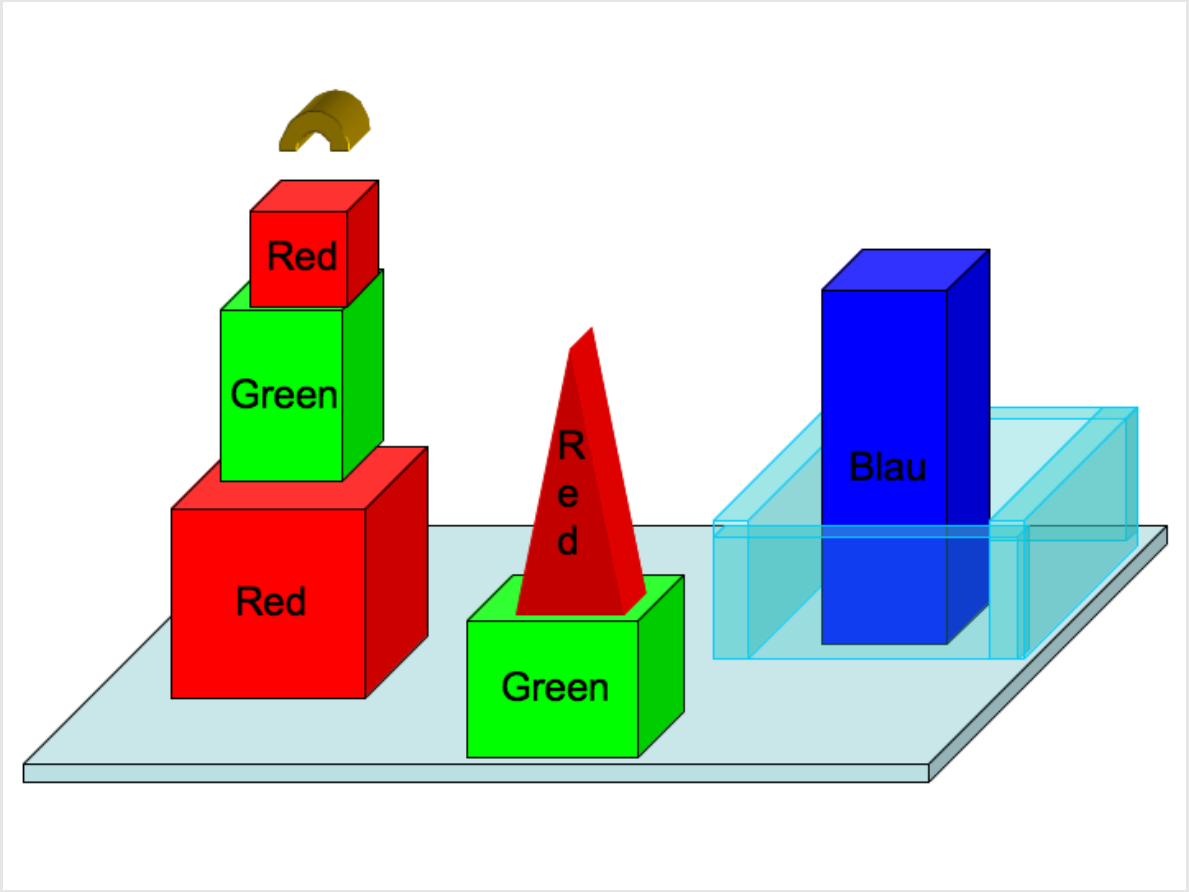
Das überraschende Ergebnis:

- In der Klötzchenwelt 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 Klötzchenwelt 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)))
```



“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 ands ends. The end of a succesful 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 it leads the reader to a better understanding of the issues."

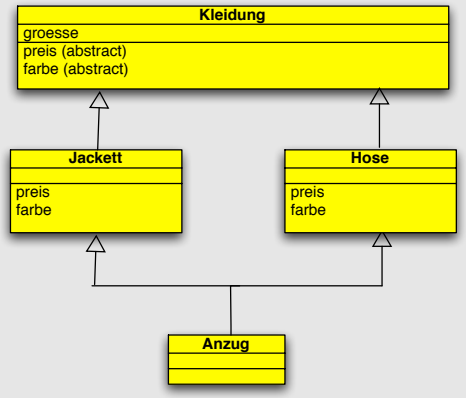
[**?**, Seite 147]

Teil X

Objekte und generische Funktionen

CLOS: Common Lisp Object System

Objektorientierte Programmierung in Swindle



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.

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.

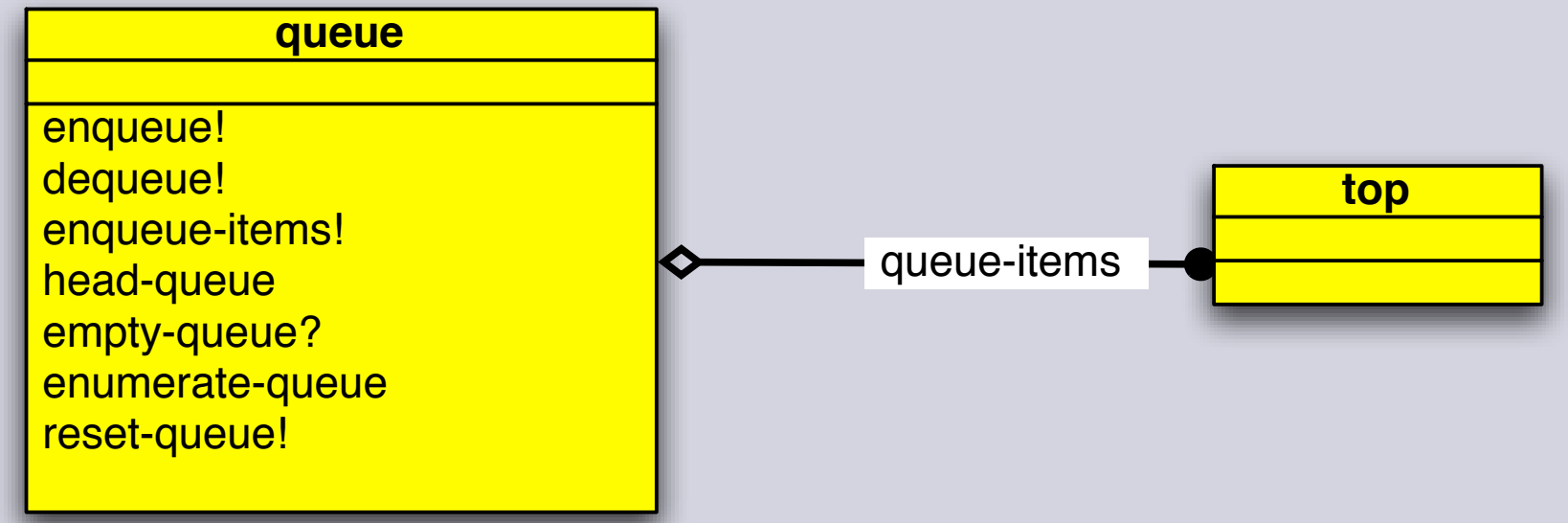
```
#lang swindle
  (require swindle/setf
   swindle/misc)
```

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

Eine Klasse von Schlangen

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.

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.

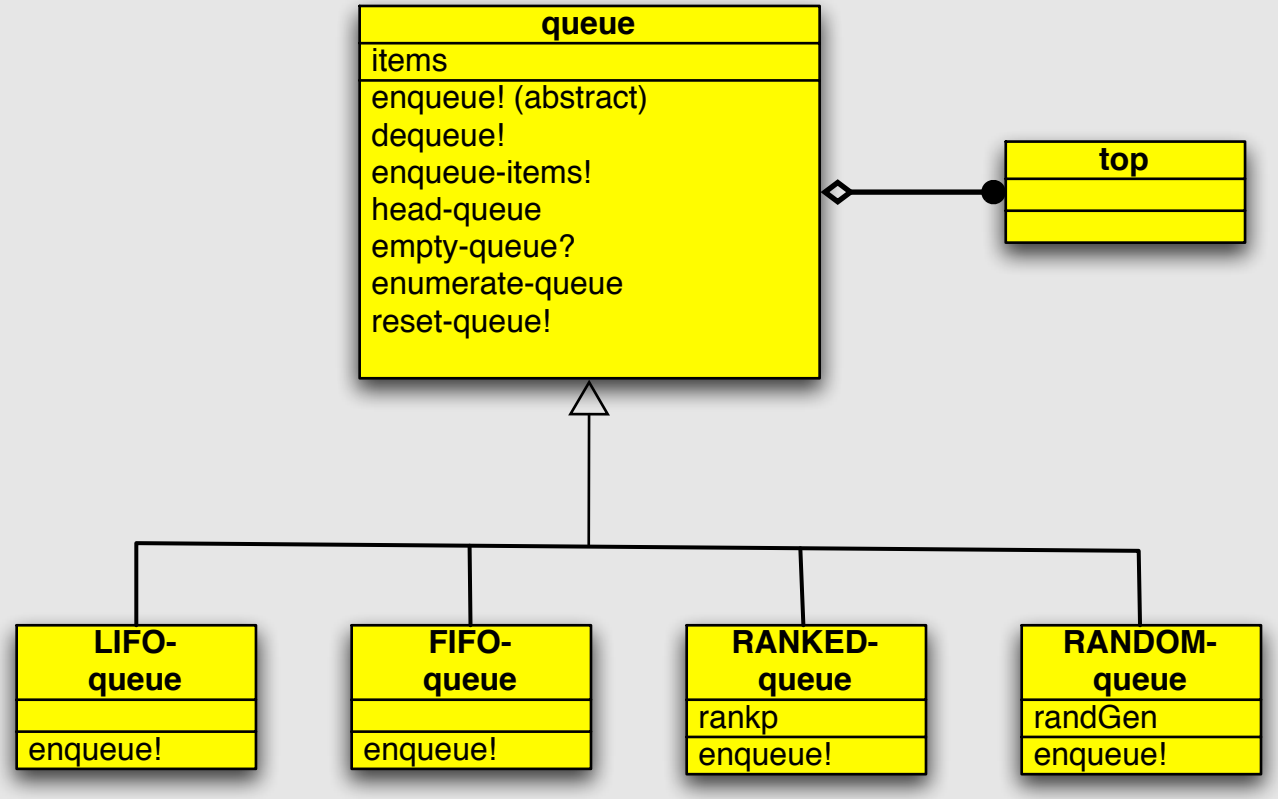
Implementation der Methoden

```
(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 defgeneric die Signatur zu spezifieren.
  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 wir 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>
```

: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)
```

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)
```

Mehrfachvererbung

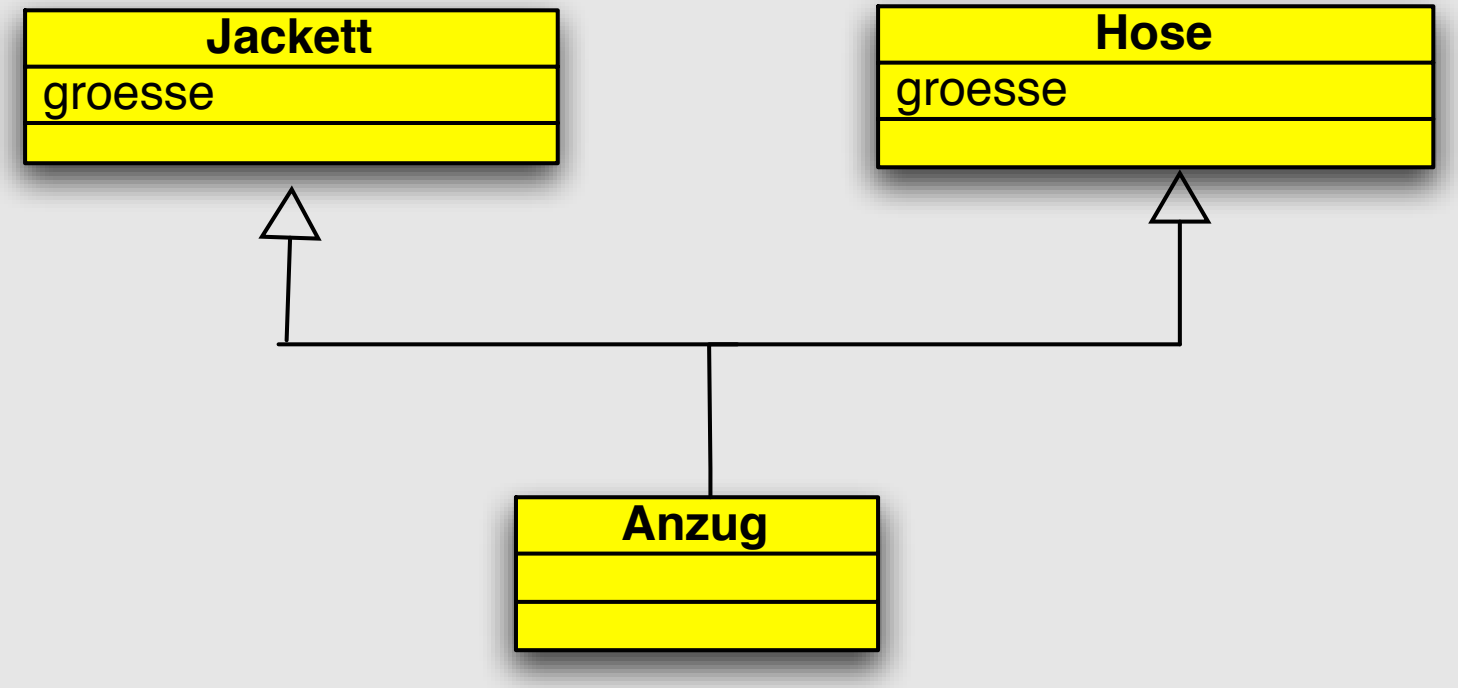
- Klassen können von mehreren Oberklassen erben.
- Das parallele Erben von mehreren direkten Oberklassen heißt Mehrfachvererbung.
- Mehrfachvererbung wird in Lisp (CLOS) häufig angewendet, um
  - Aggregate zu modellieren,
  - um durch „mixin“-Klassen eine bestimmte Funktionalität zur Verfügung zu stellen,
  - 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.

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>)
```

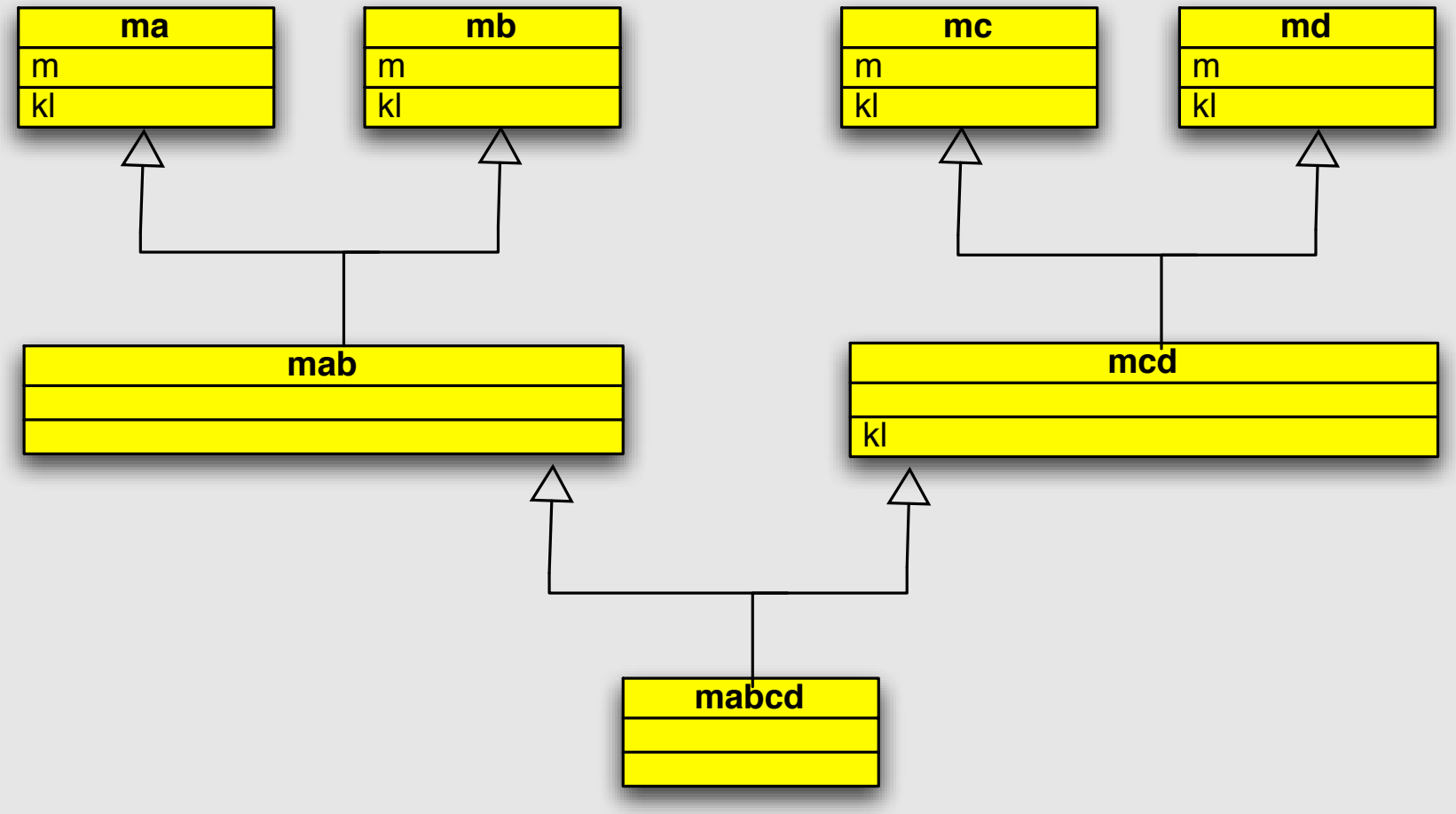
Beobachtung:

- Der *slot* „groesse“ wurde in der Klasse „Anzug “ nur einmal angelegt.
- Beide Akzessorfunktionen „gr“ und „ grAnzug“ wurden vererbt.

Regelung von Erbschaftskonfliken

- 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.

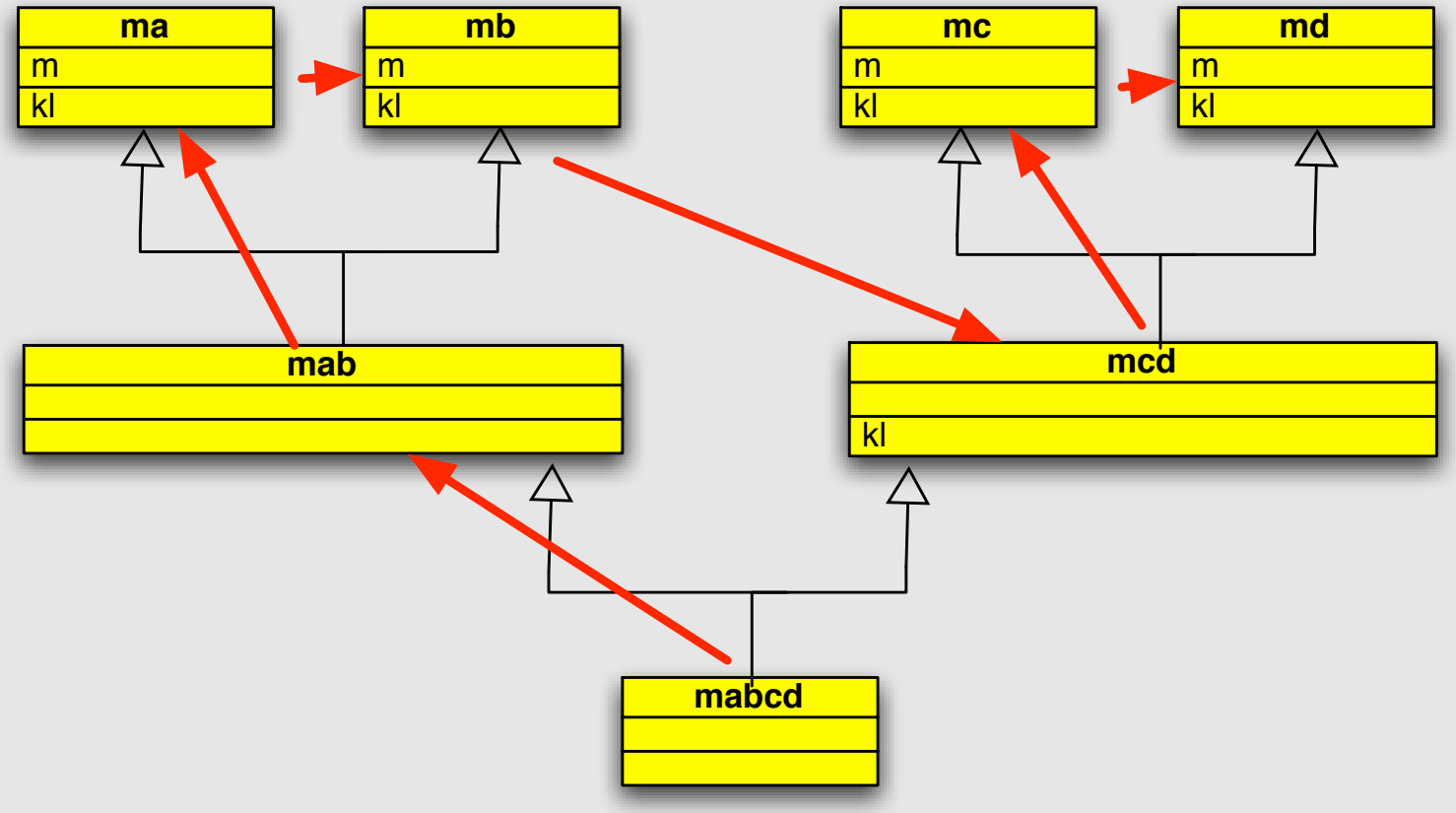
Beispiel



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

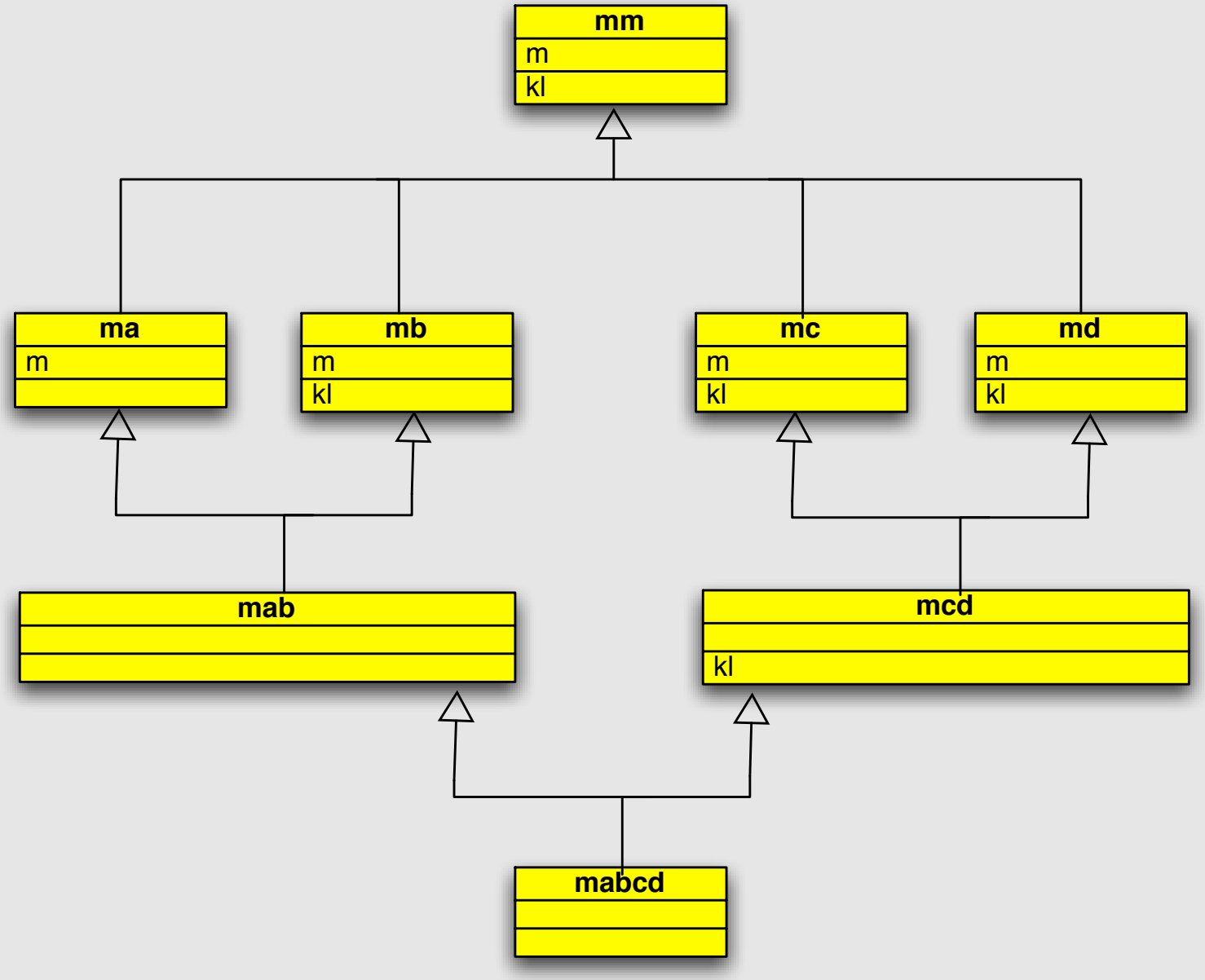



```
> (class-cpl mabcd) ⟶ ; Präzedenzliste
(#<class:mabcd> #<class:mab> #<class:ma>
 #<class:mb> #<class:mcd> #<class:mc>
 #<class:md> #<class:object> #<class:top>)
```

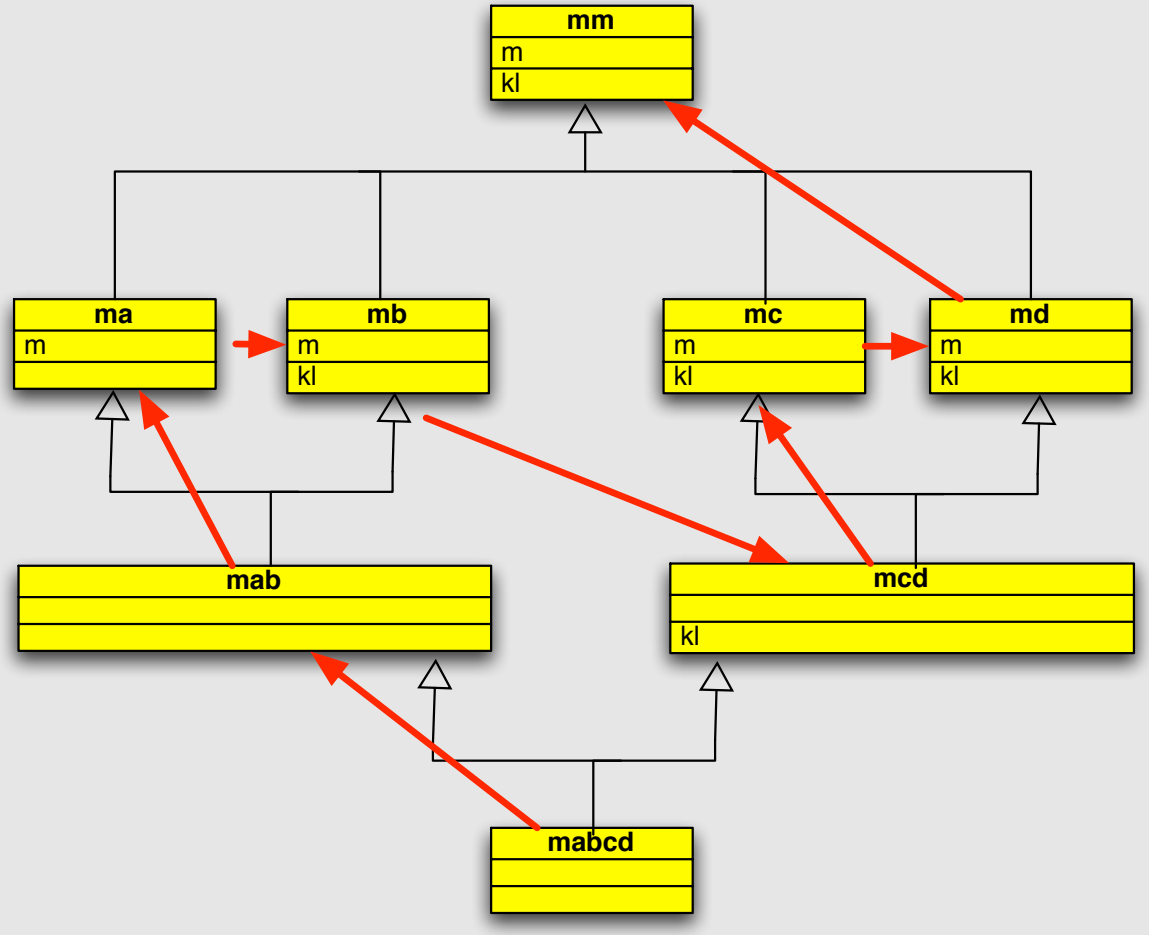
Vererbte Methoden

```
(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



Die Klassenpräzedenzliste



> (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 (Methodenkombination)

Für ein Aggregat Anzug:

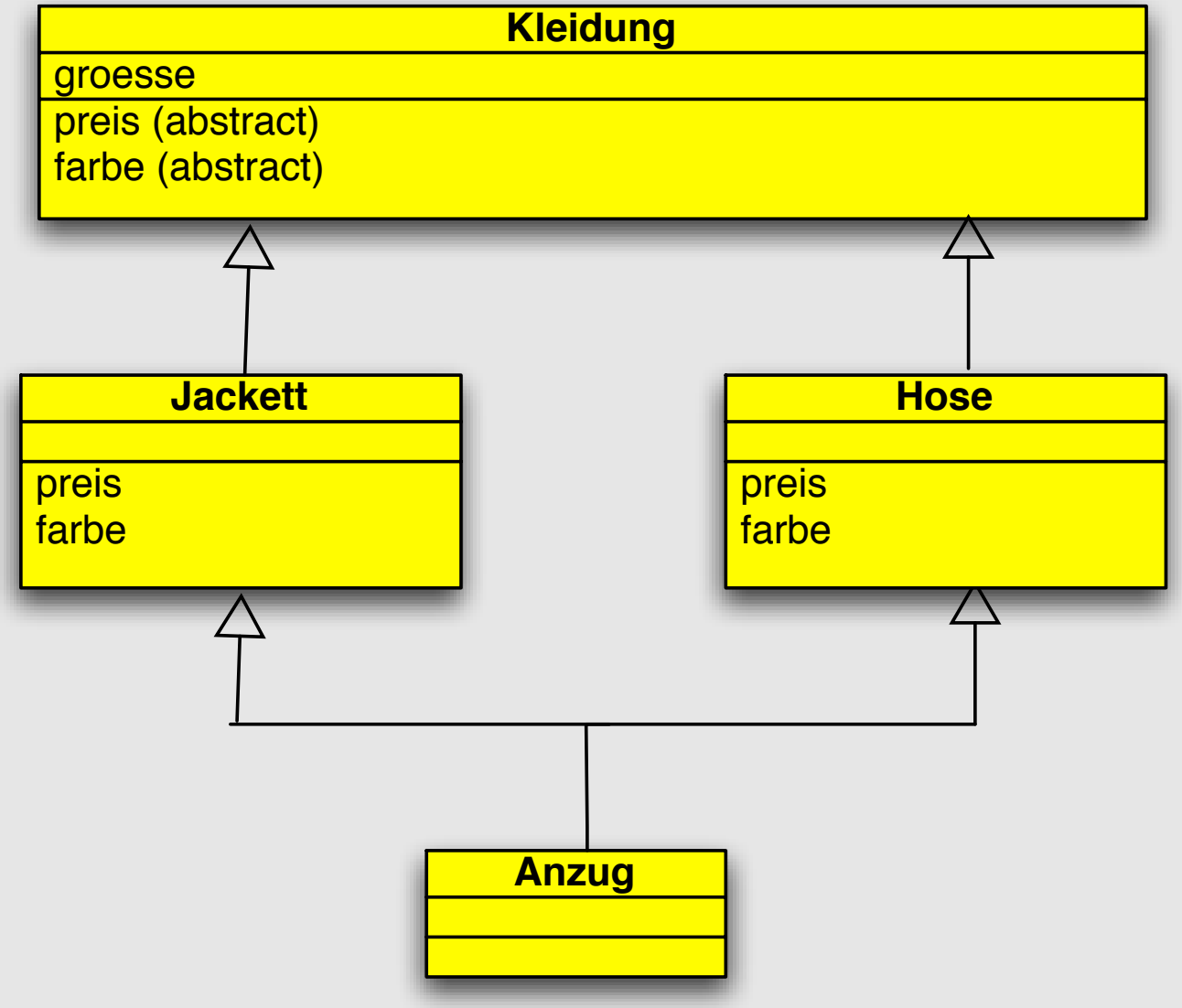
Der Preis für einen Anzug sollte

- 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) )
```

Beispiellauf

```
> (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 (Methodenkombination für den slot „Preis“:)

- Wir speichern die Preise für die Einzelteile eines Anzugs in den „preisJ“ für das Jackett, „PreisH“ für die Hose.

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 (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
```

Teil X

Objekte und generische Funktionen

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

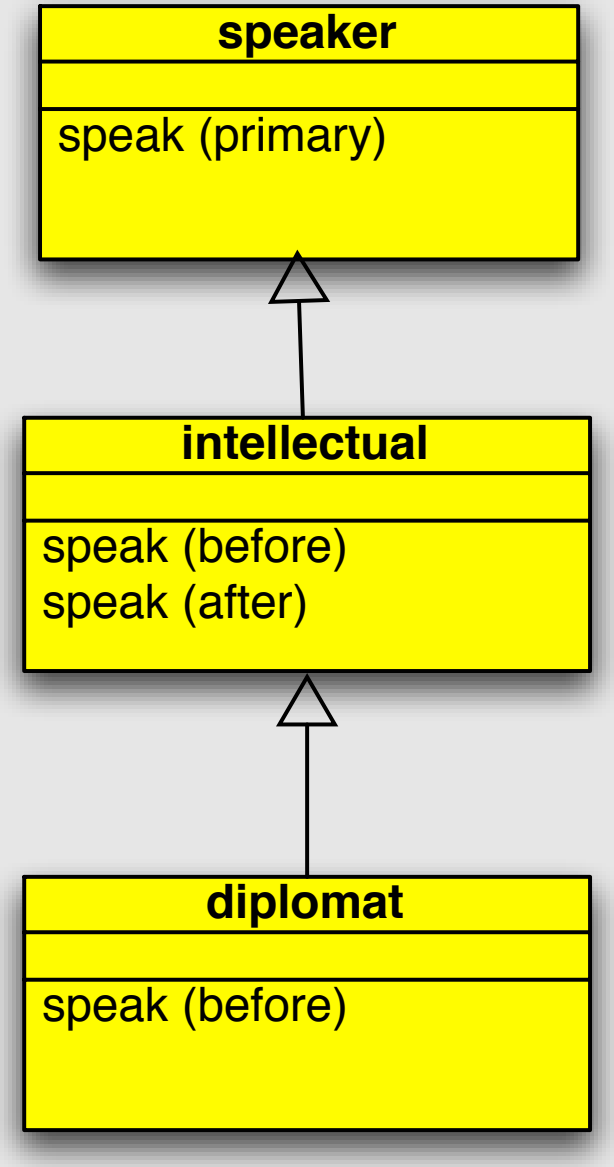
„super call“: Wir rufen mit call-next-method direkt die Methode der Oberklasse auf, und führen vorher oder nachher die notwendigen Zusatzoperationen aus.

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**.

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: "))
```

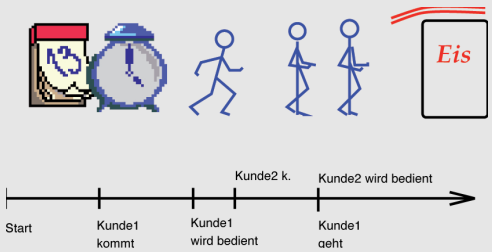
```
> (define c (make klasseC))
> (teste c)
vor c:
vor b:
Klasse A
nach b:
nach c:
>
```

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.

Teil X

Objekte und generische Funktionen



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.

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.

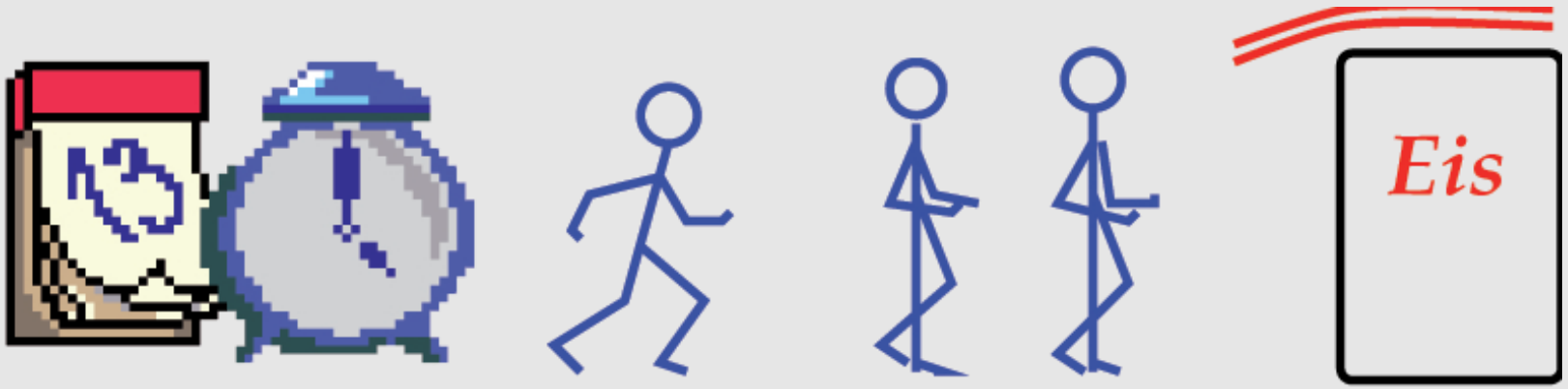
Rückübertragung der Ergebnisse in die Anwendungsdomäne.

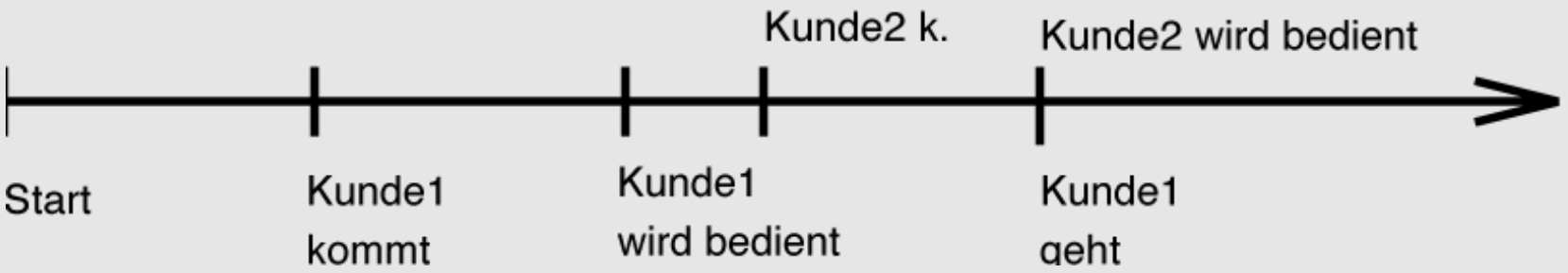
Vorteil virtueller Modelle

Materielle Modelle sind nicht wirklich abstrakt.
Ihre konkreten Eigenschaften können das Expriment 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.

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.

Ein ereignisorientiertes Simulationssysstem

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,

Dichte: $p(x) = c$

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.

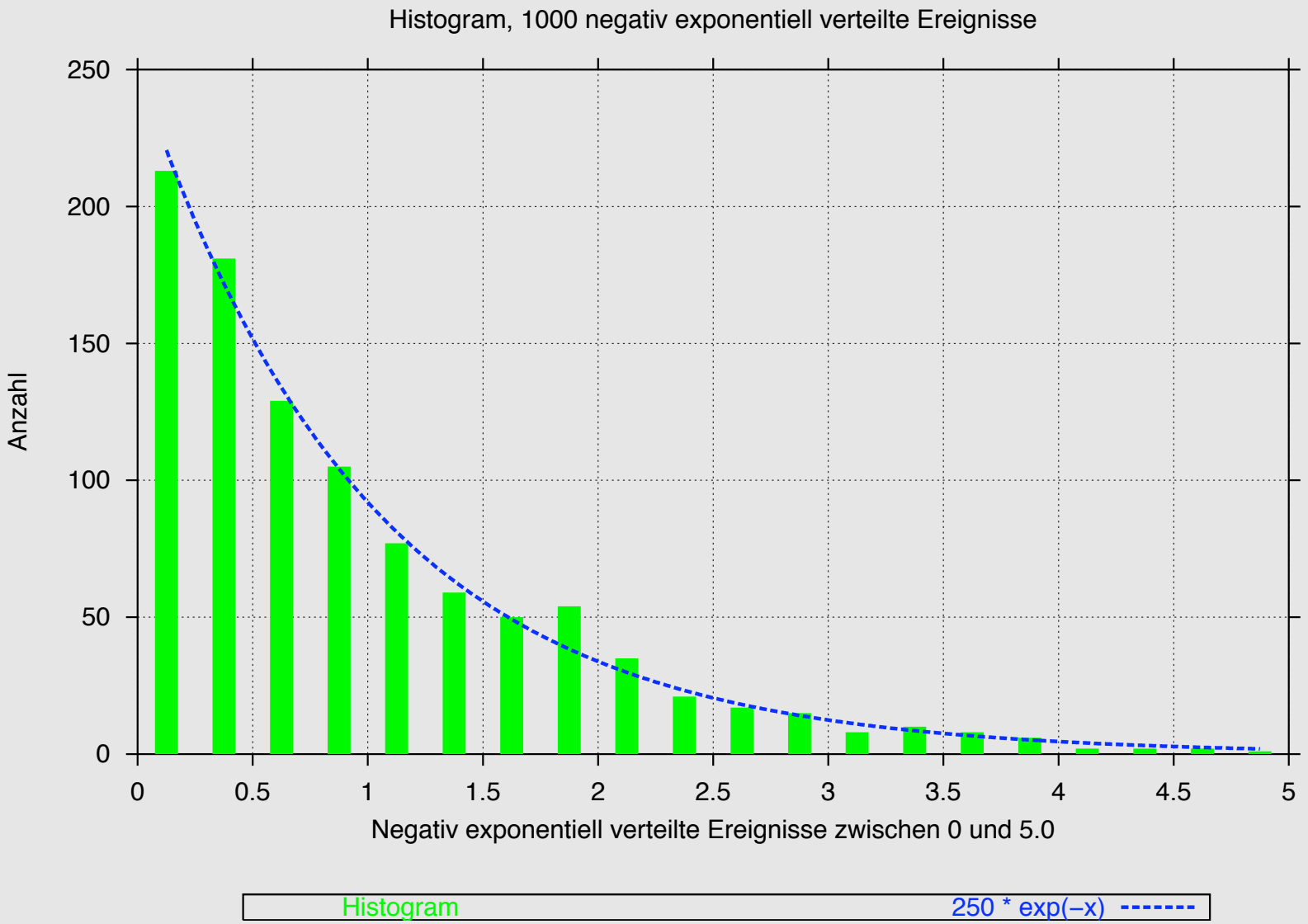
$$\text{Dichte: } p(t) = \begin{cases} 0 & \text{für } t = 0 \\ \mu \cdot e^{-\mu t} & t > 0 \qquad (\mu > 0) \end{cases}$$

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:

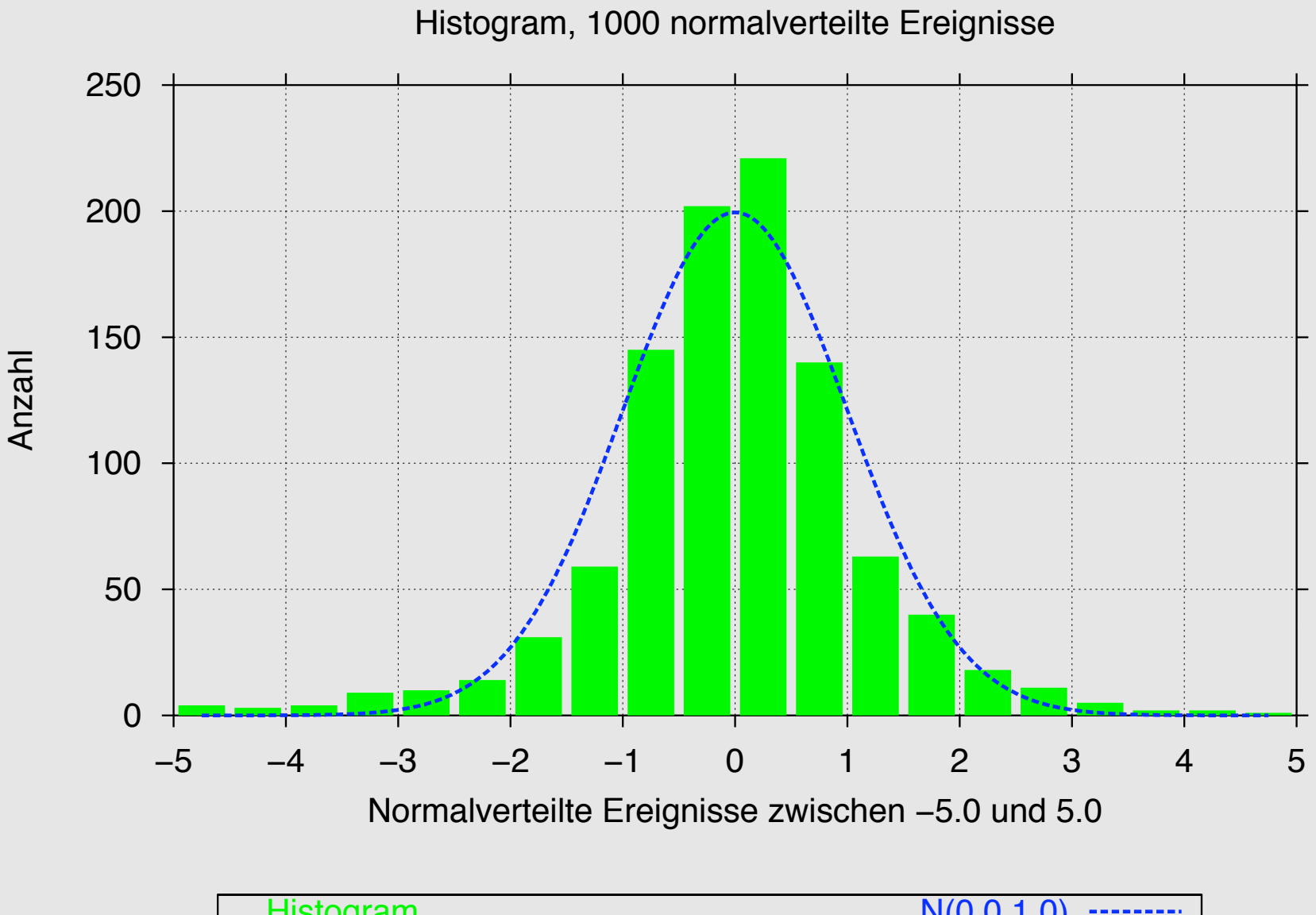
Zur Beschreibung von Ereignissen, deren Werte mit mehr oder minder großen Abweichungen um einen Mittelwert schwanken.

$$p(t) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{\left(-\frac{(t-\mu)^2}{2\sigma^2}\right)}, \quad \sigma > 0$$

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 Interval $|t - \mu| \leq 2\sigma$.
Abkürzende Schreibweise: $N(\mu, \sigma)$

1000 normalverteilte Ereignisse

Histogramm



Erzeugung gleichverteilter Zahlen

Die Kongruenzmethode:

Die Kongruenzmethode erzeugt eine Folge von ganzen Zahlen $I_1, I_2, I_3 \ldots$ aus dem Interval $[0 \ldots m]$ 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.

Erzeugung gleichverteilter Zahlen

Die Kongruenzmethode:

Die Kongruenzmethode erzeugt eine Folge von ganzen Zahlen $I_1, I_2, I_3 \ldots$ aus dem Interval $[0 \ldots m]$ 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.

Beliebig verteilte Zufallszahlen

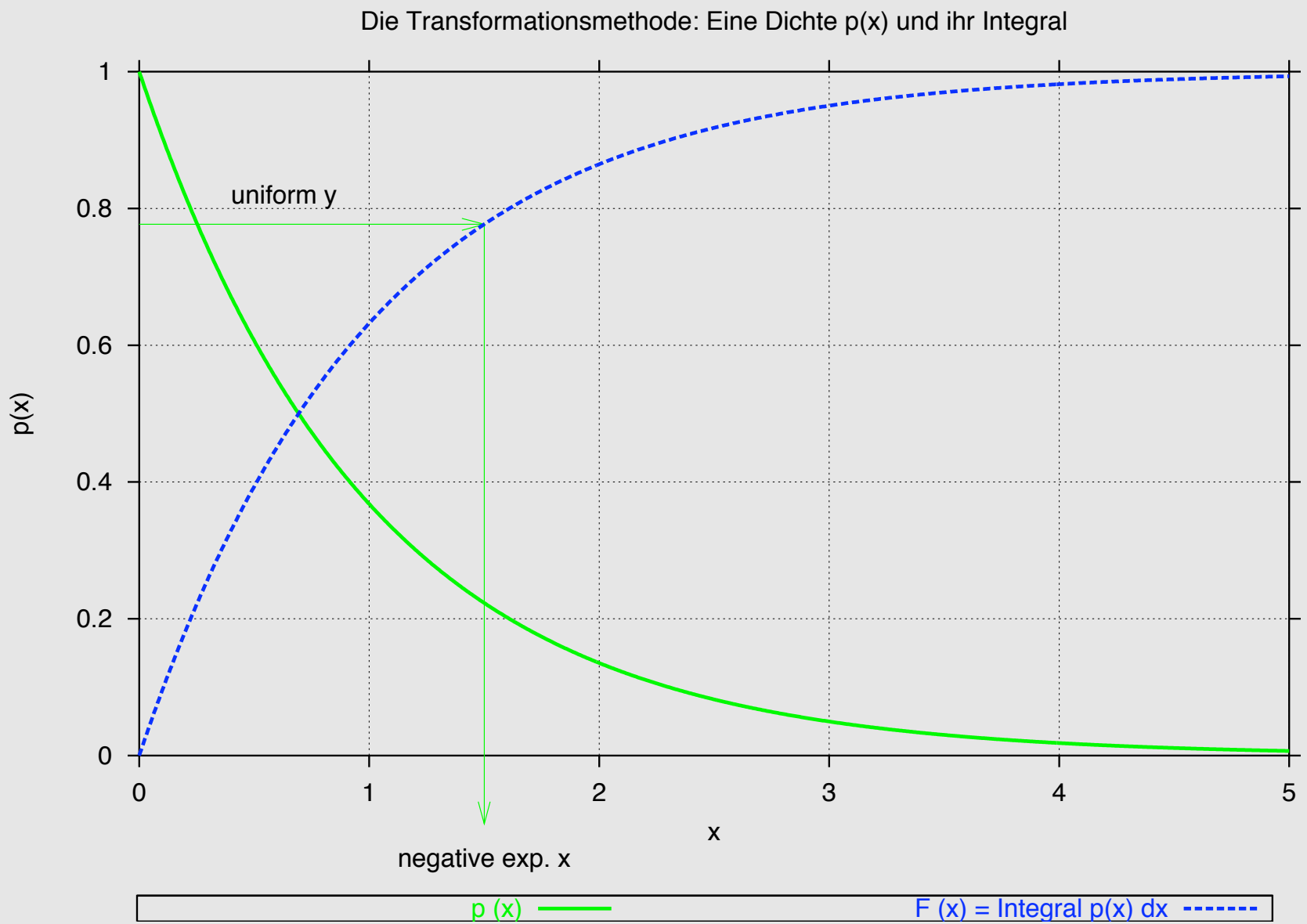
Die Transformationsmethode

Wenn die von uns gewünschte Dichtefunktion p(x)

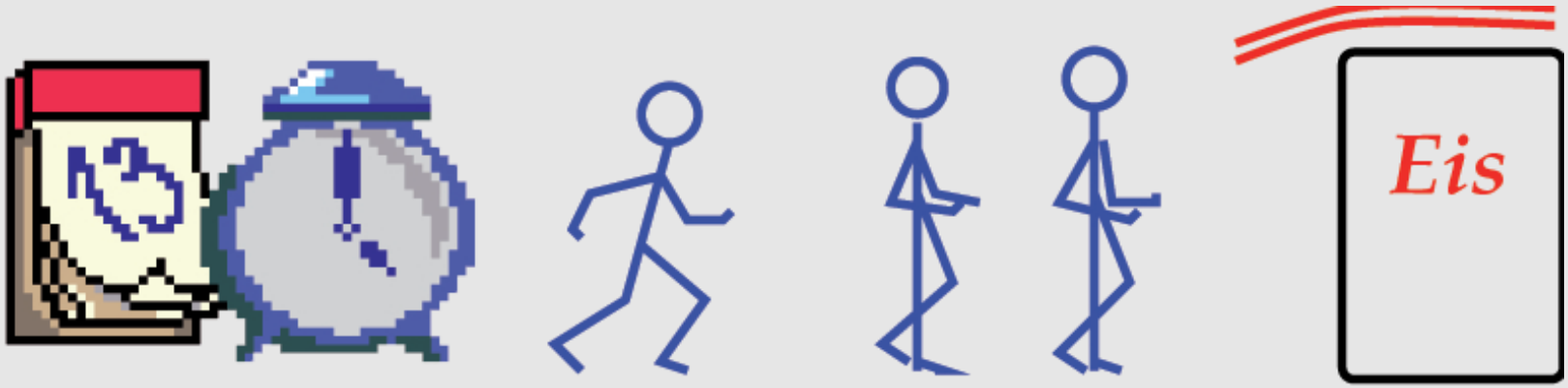
- integrierbar ist
- und für das unbestimmte Integral die Umkehrfunktion existiert,

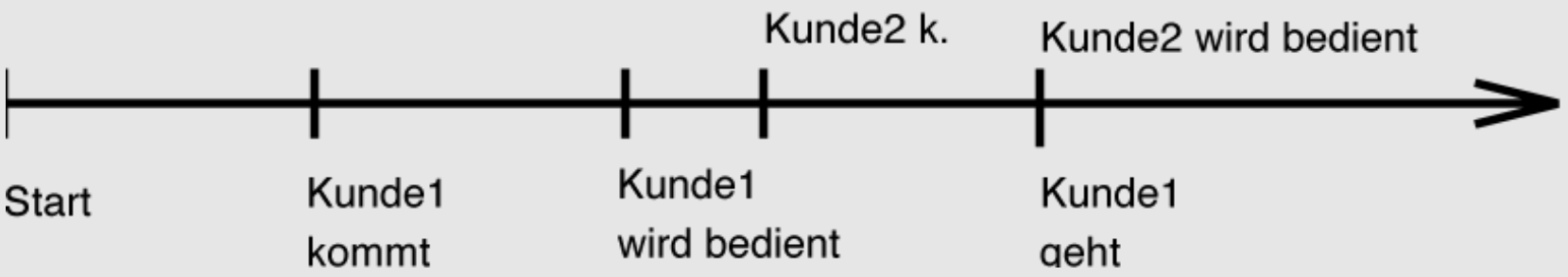
erhalten wir Zufallszahlen verteilt mit der Dichte $p(x)$, wenn wir *gleichverteilte* Zufallszahlen mit der Umkehrfunktion des unbestimten Integrals der Dichte transformieren.

Transformation mittels des Integrals



Ein Basissystem zur Simulation





Komponenten eines Simulationssystems

Grobentwurf

Beispiel (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.

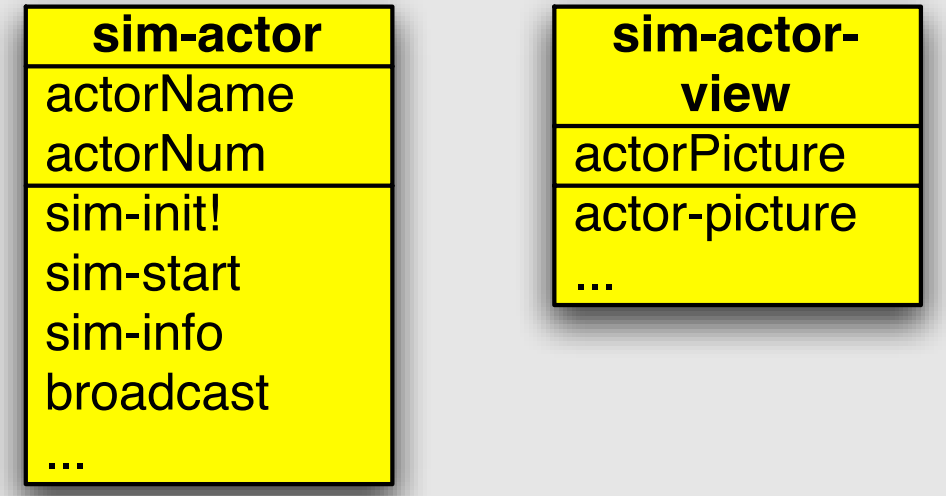
Die Klasse „sim-actor“:
Das Protokoll für Simulationsobjekte

Wenn wir ein Simulationsszenarium 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 Simulationsszenarium 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



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)
```

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

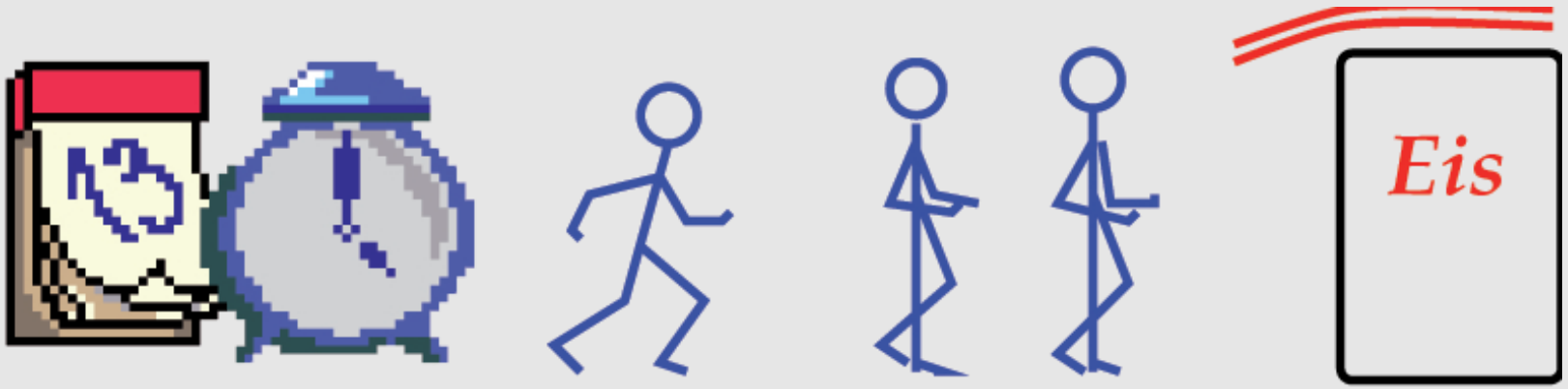
Beispiel (Nachmethode für initialize)

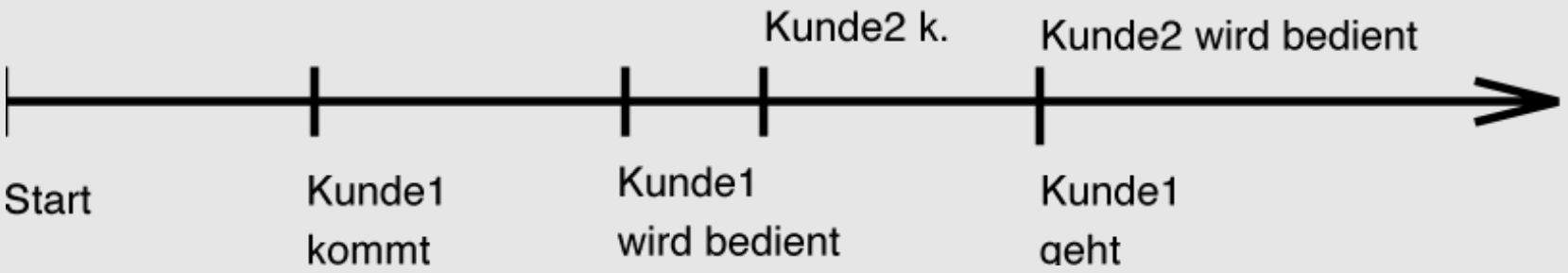
Vermerke beim Erzeugen eines Simulationsobjektes eine Referenz auf das Objekt in einer Liste:

```
(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.

Ein Basissystem zur Simulation





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

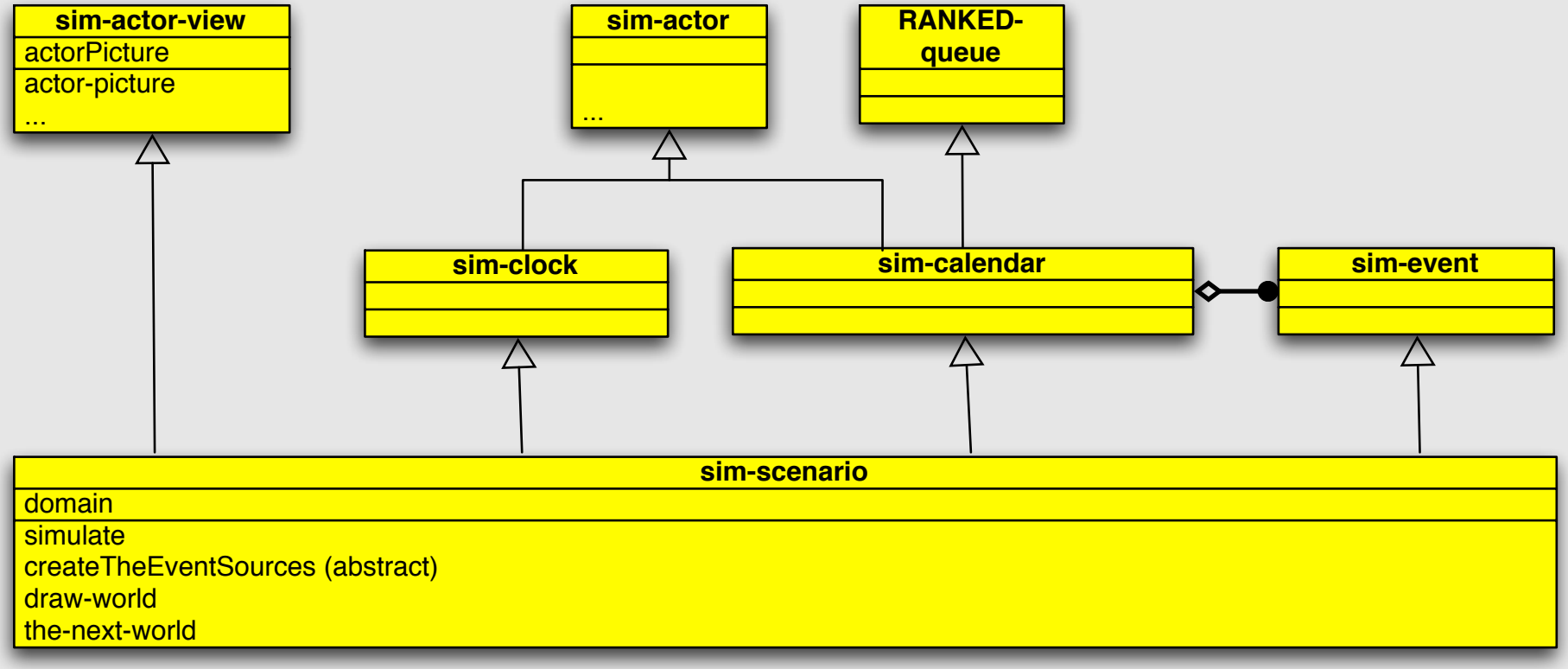
Terminkalender (sim−calendar): Enthält eine nach Uhrzeit geordnete Schlange (RANKED−queue) von anstehenden Ereignissen (events) sowie eine Uhr (sim−clock).

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



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

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 has emptied 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 (Weckauftrag) 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: 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) )
```

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
```

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)

  (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:

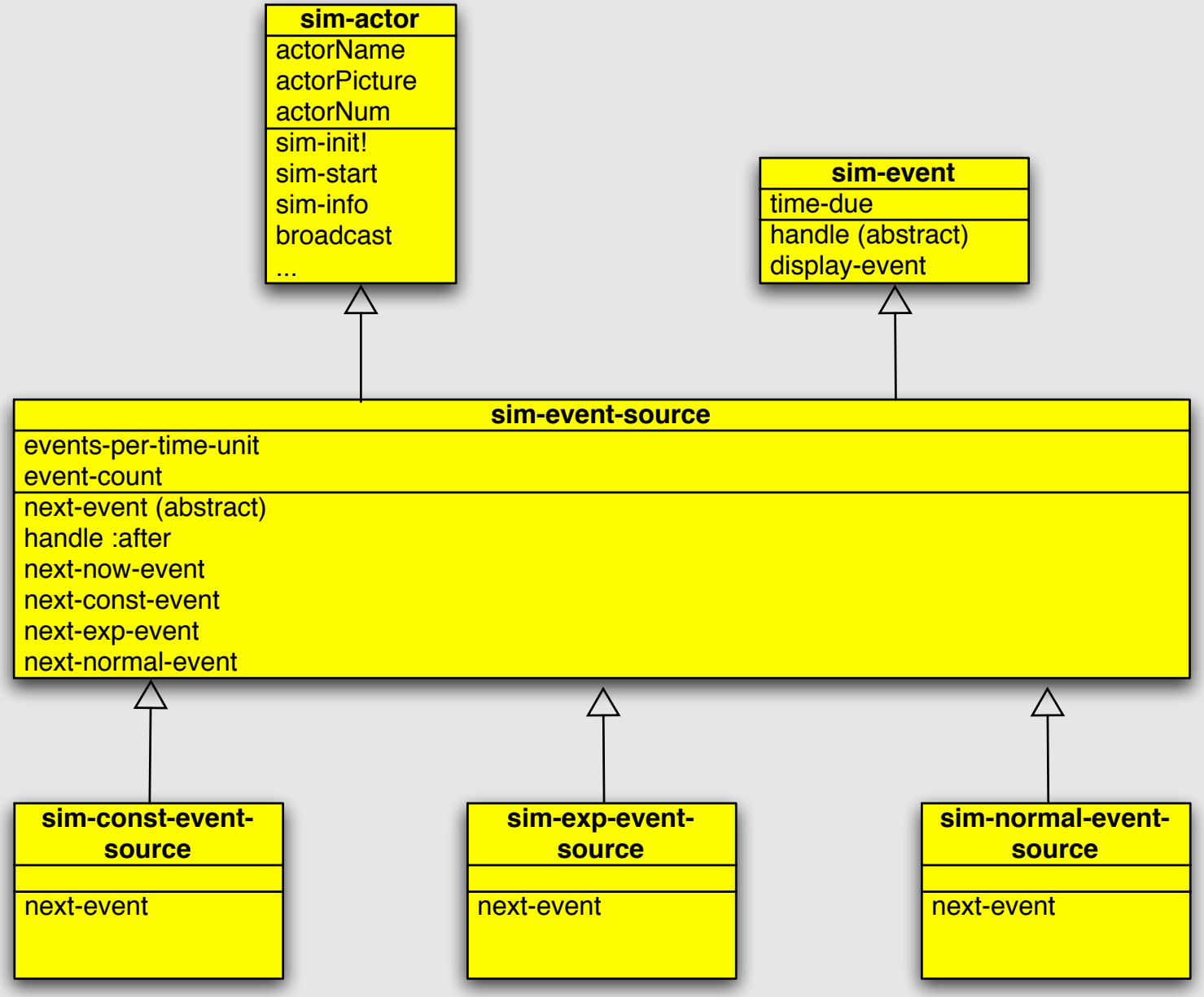
next-now-event: Das nächste Ereignis findet sofort statt.

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.

```
(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

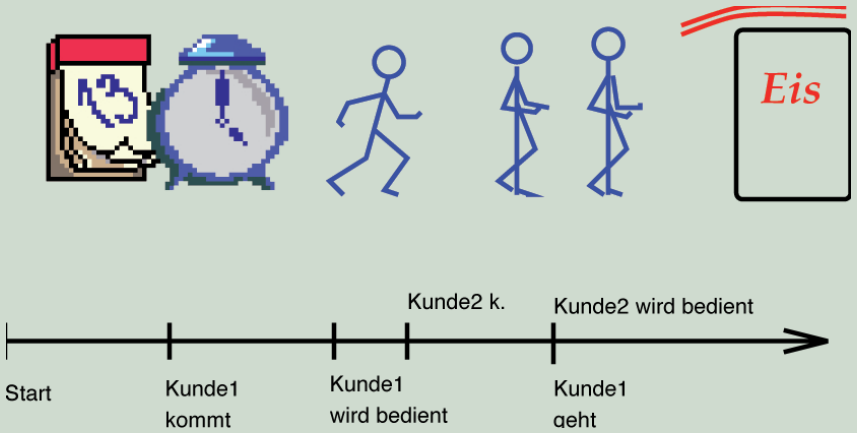
- Rufe den klassenspezifischen event handler (handle) auf.
- Bestimme den Zeitpunkt des nächsten Ereignisses.
- Trage es in den Simulationskalender ein (schedule).
- Das Planen des nächsten Ereignisses (next-event) geschieht in einer Nachmethode zu „handle“.
  So kann die Primärmethode „handle“ anwendungsspezifisch 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))
```

Ein ereignisorientiertes Simulationssysstem

Anwendungssystem: Eine Bedienstation

Beispiel (Kunden-Bediensystem)



Benötigt werden:

Eine Bedieneinheit:

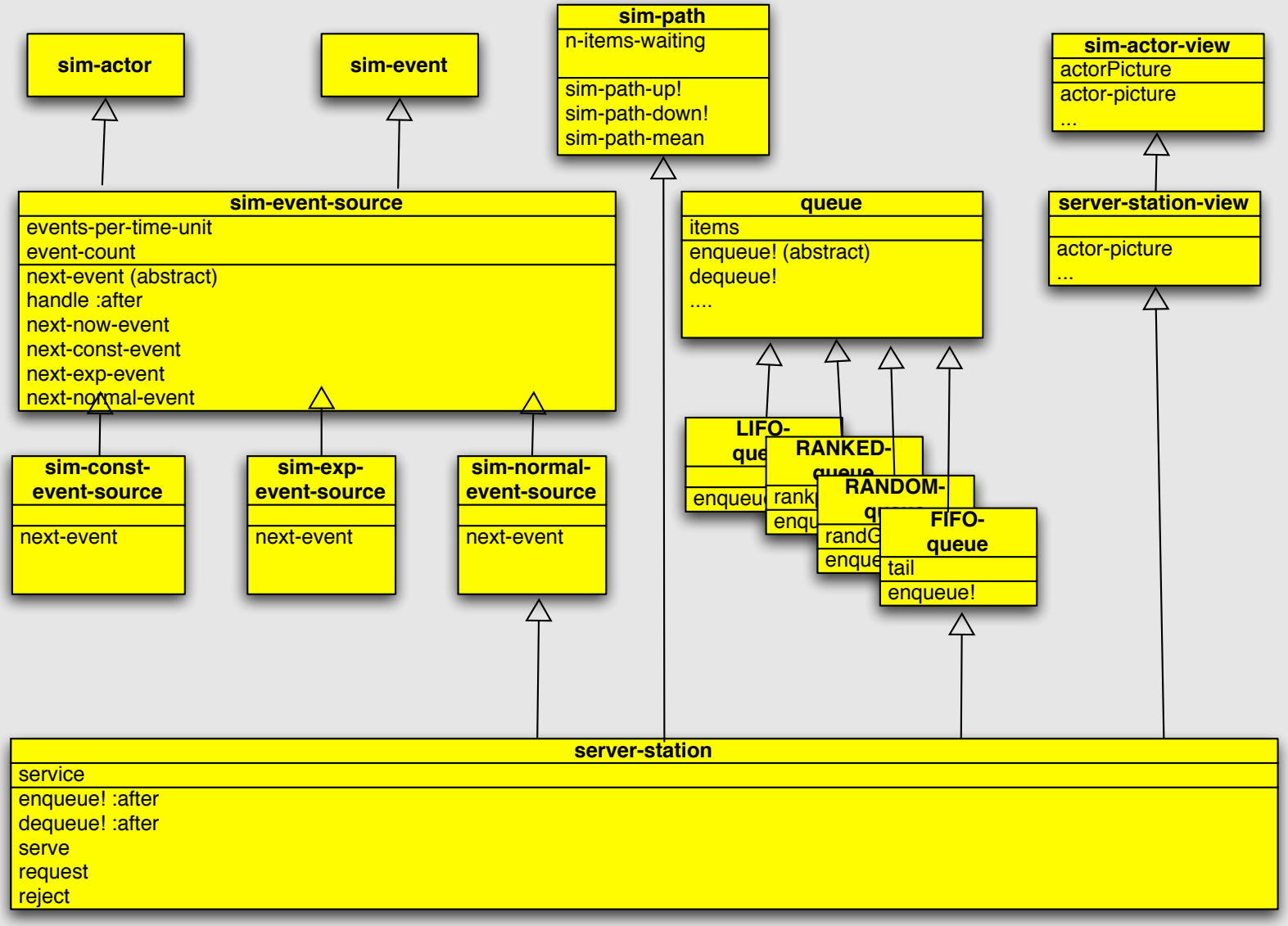
Löst Bedienereignisse aus (sim−event−source).

Warteschlange von Kunden: FIFO−queues.

Protokoll: (sim−path:) Mittlere Wartezeit, Schlangenlänge usw.

Die Klasse „server-station“

Die Klasse server-station



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))
```

Bedienereignisse der Serverstation

Ein Ereignis startet,

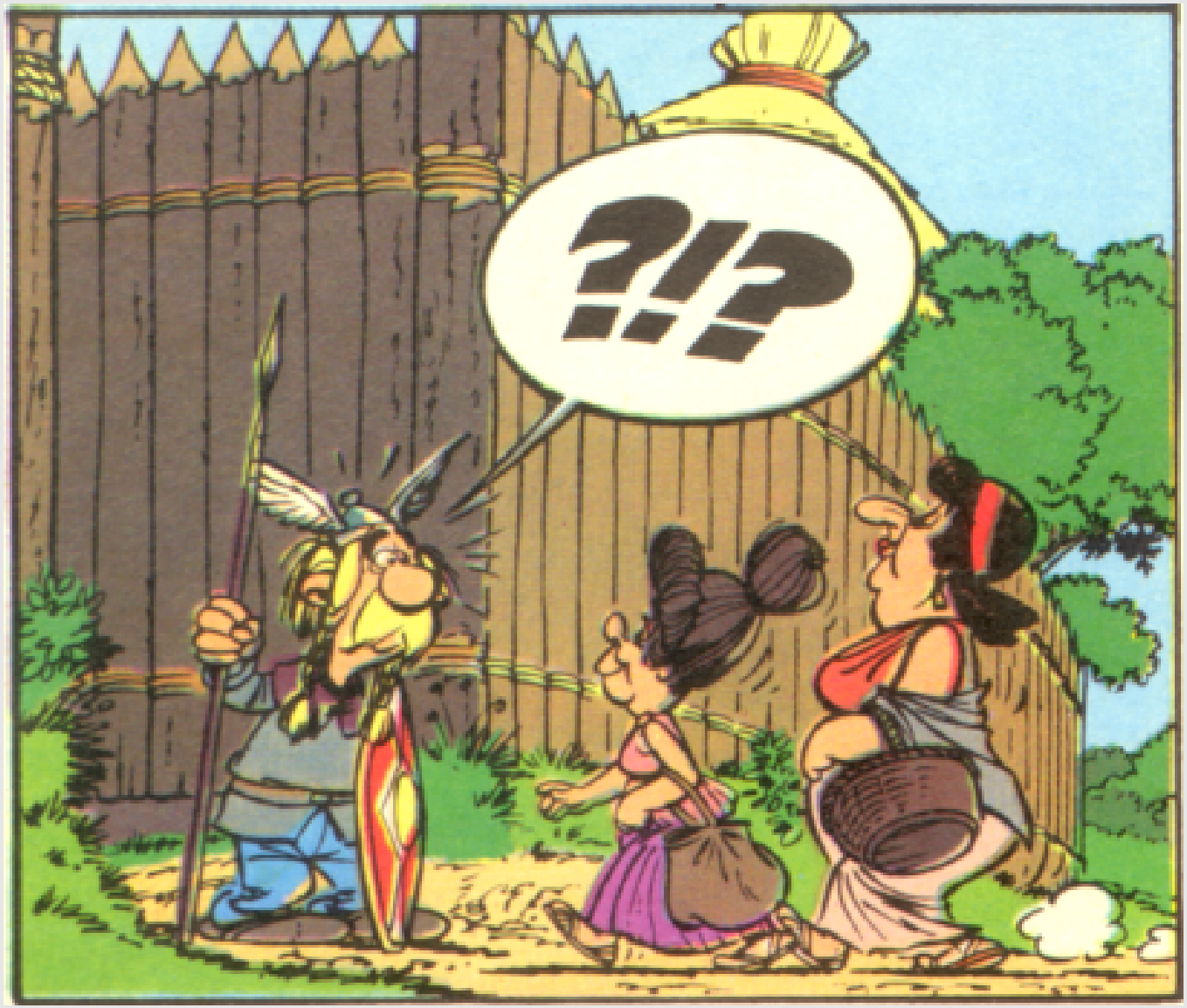
1. wenn ein Kunde an einen leeren Schalter tritt,
2. oder wenn die Bedienung des vorherigen Kunden abgeschlossen ist.

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
))
```

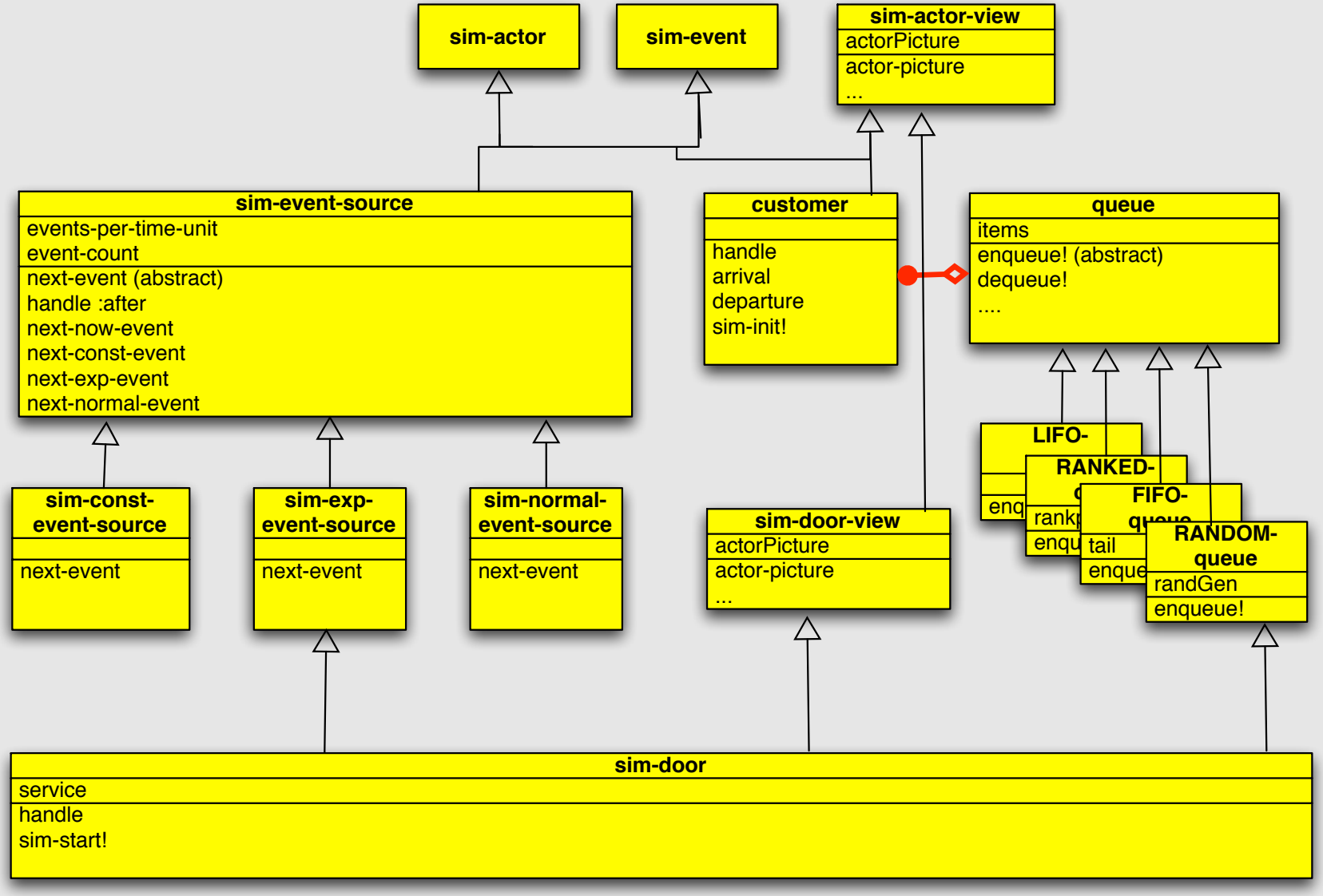
Kunden und Simulationstür



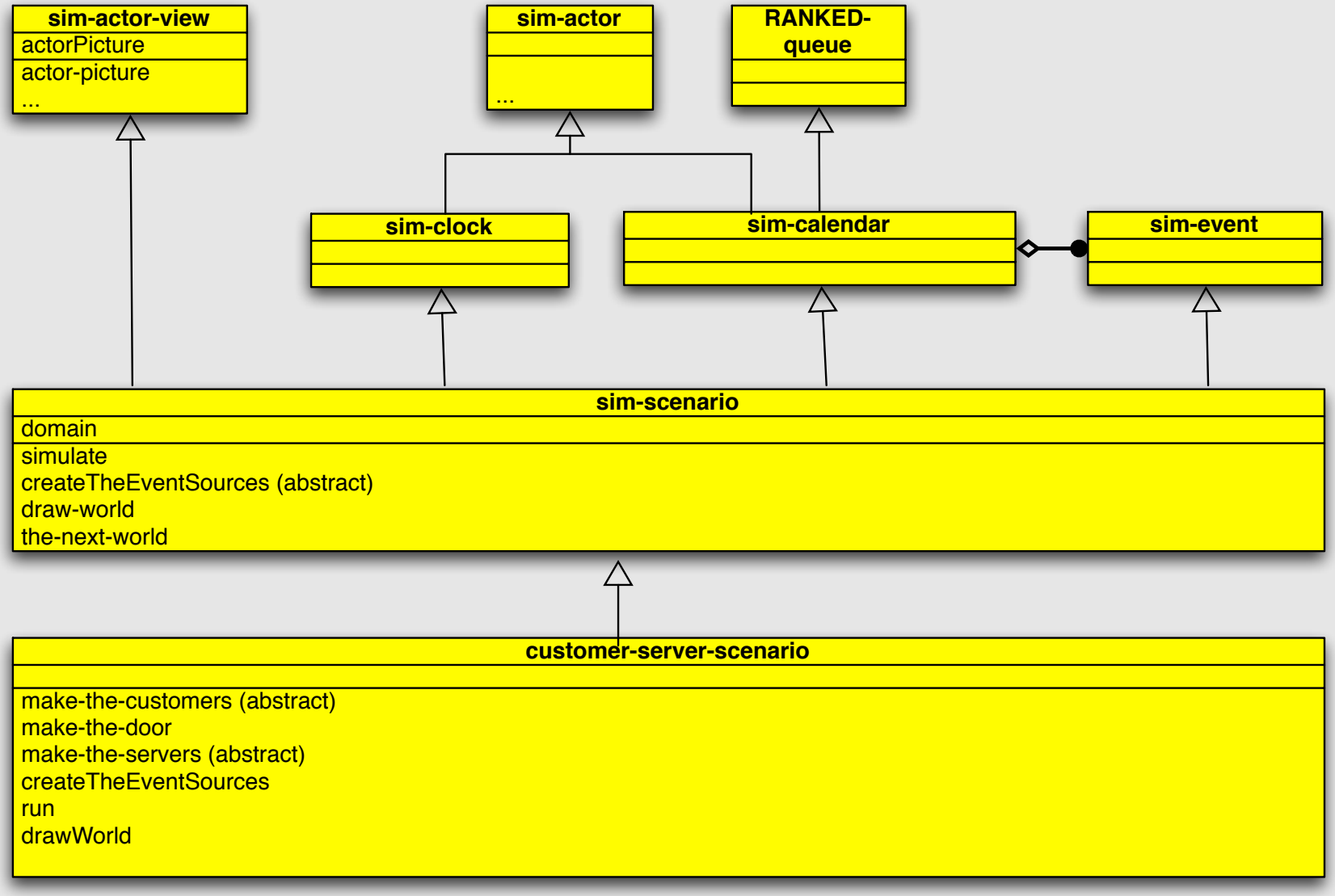
Die Kunden

- 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



Die Klasse customer-server-scenario



Controller: Das Customer-Server-Szerario

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.

Ein ereignisorientiertes Simulationssysstem

Anwendung 1: Ein Mensa-Szenario

Beispiel (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" "Ottilie" "Anton")))
      (map (lambda (c)
          (make customer :actorName c))
        someStudents)))
```

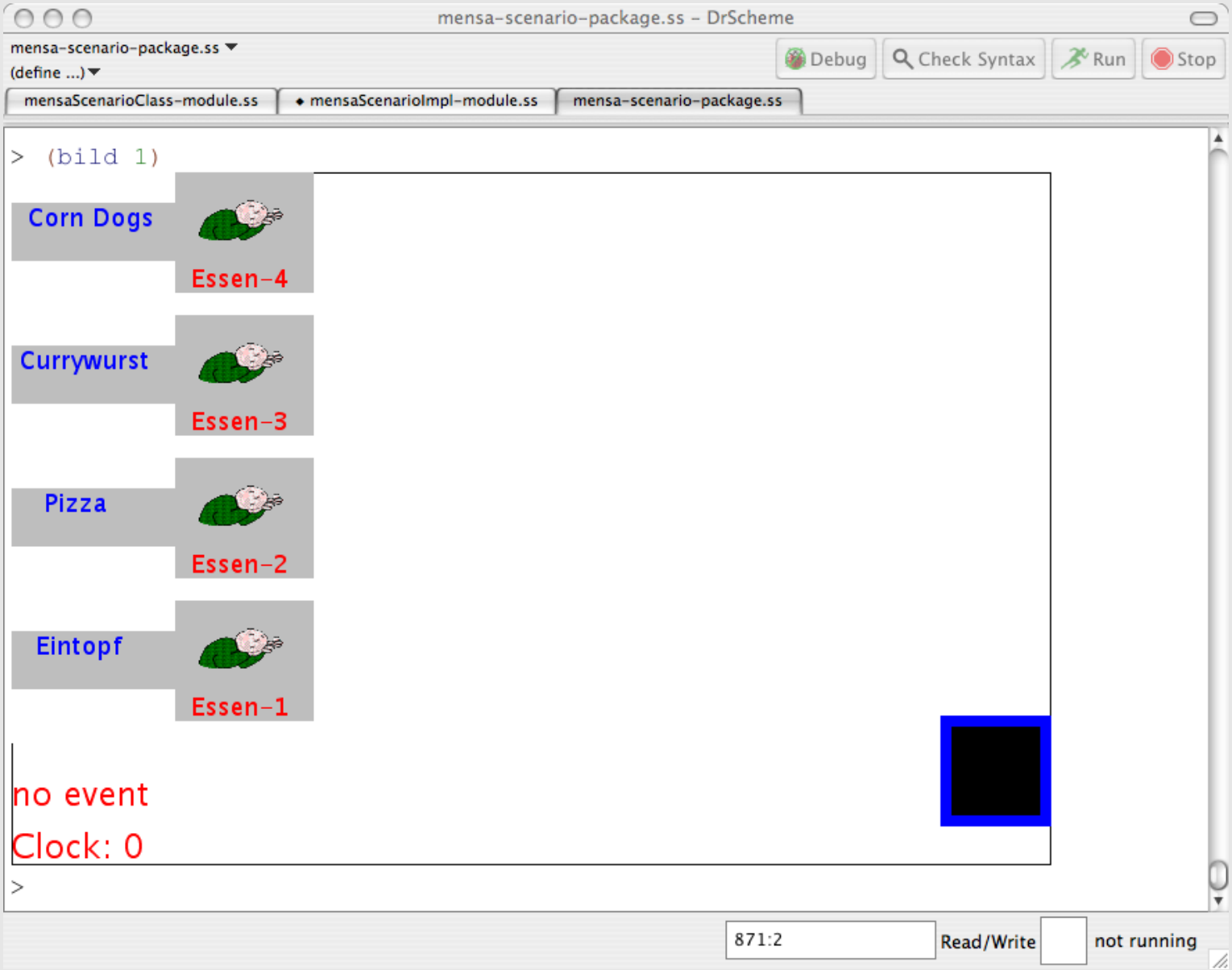
Starten der Simulation

```
(define (mensaDemo1)
   (run (make mensa-scenario
              :actorName "StEllingen" )))
```

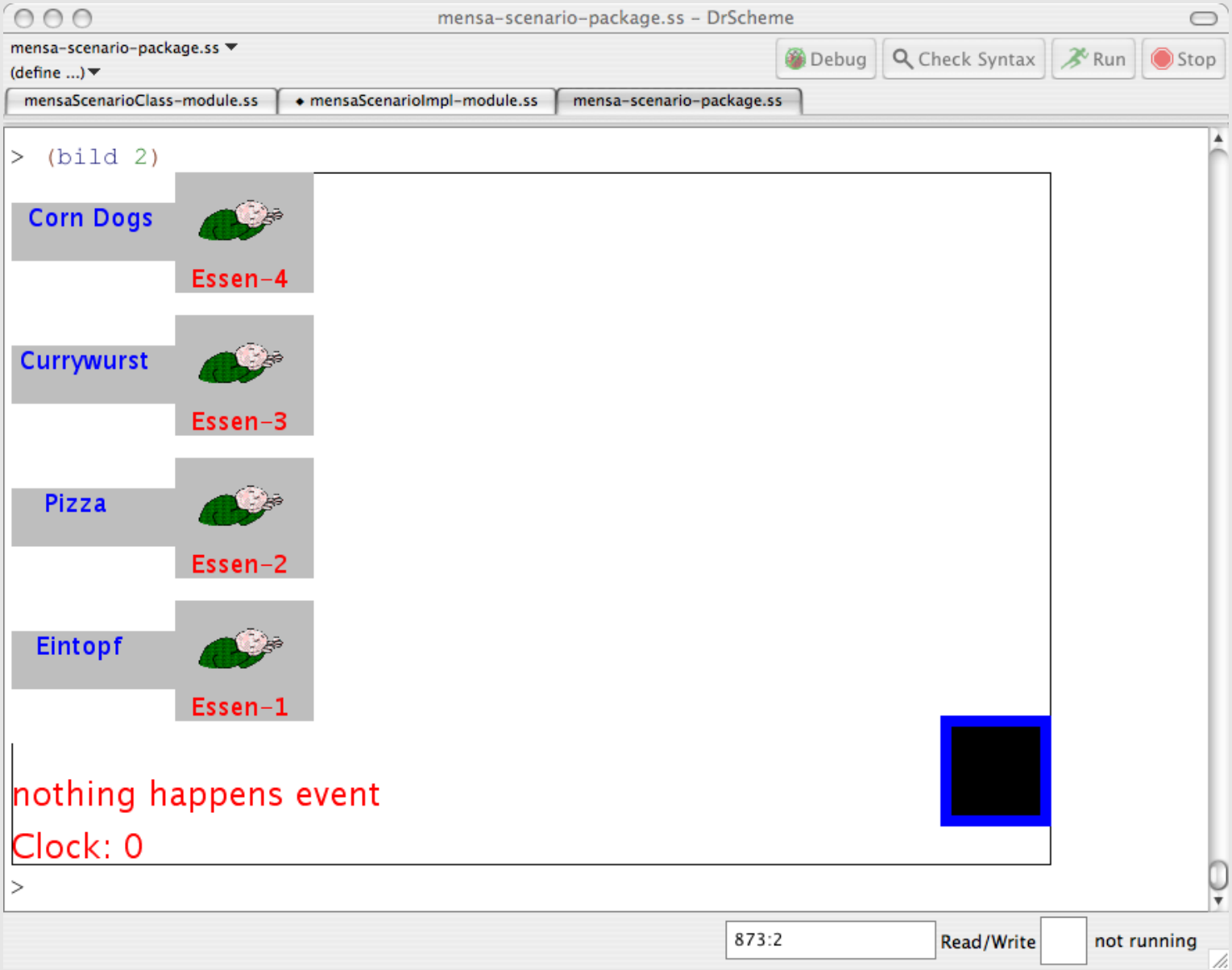
Laden des Simulationspaketes

```
(require se3-bib/sim/mensa-scenario-package)

(mensaDemo1)
```

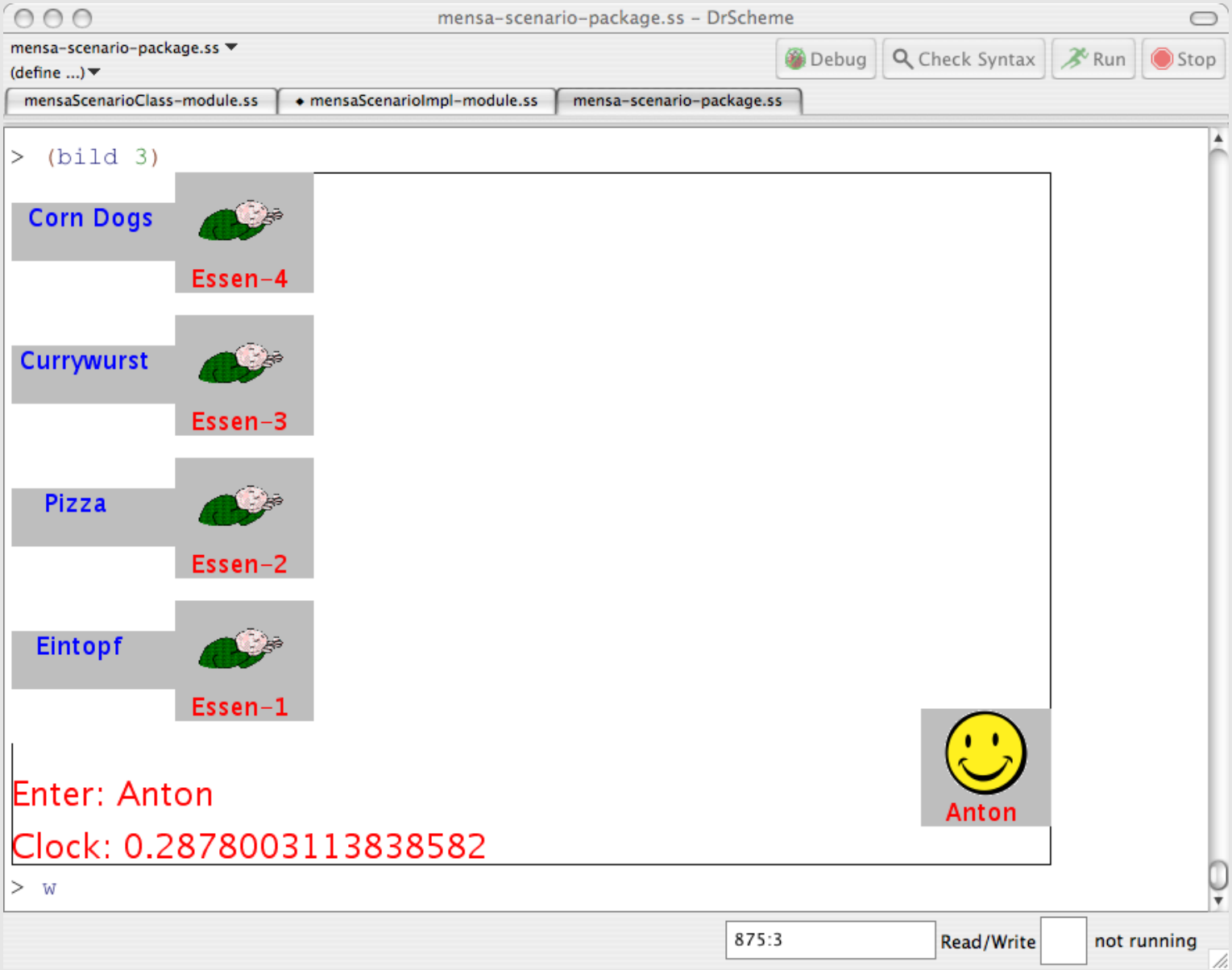
Ein Beispiellauf: Mensa-Szenario 1



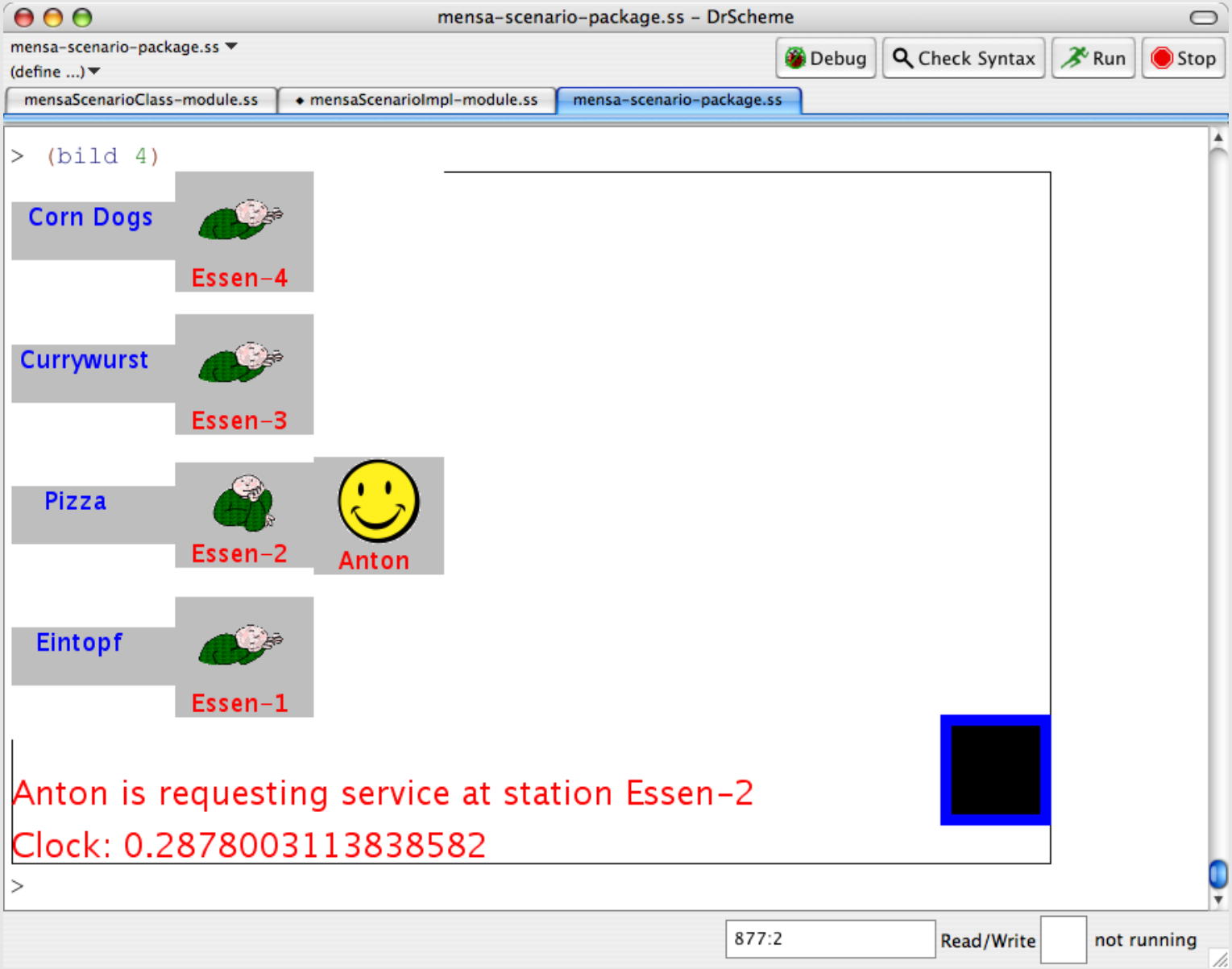
Ein Beispiellauf: Mensa-Szenario 1



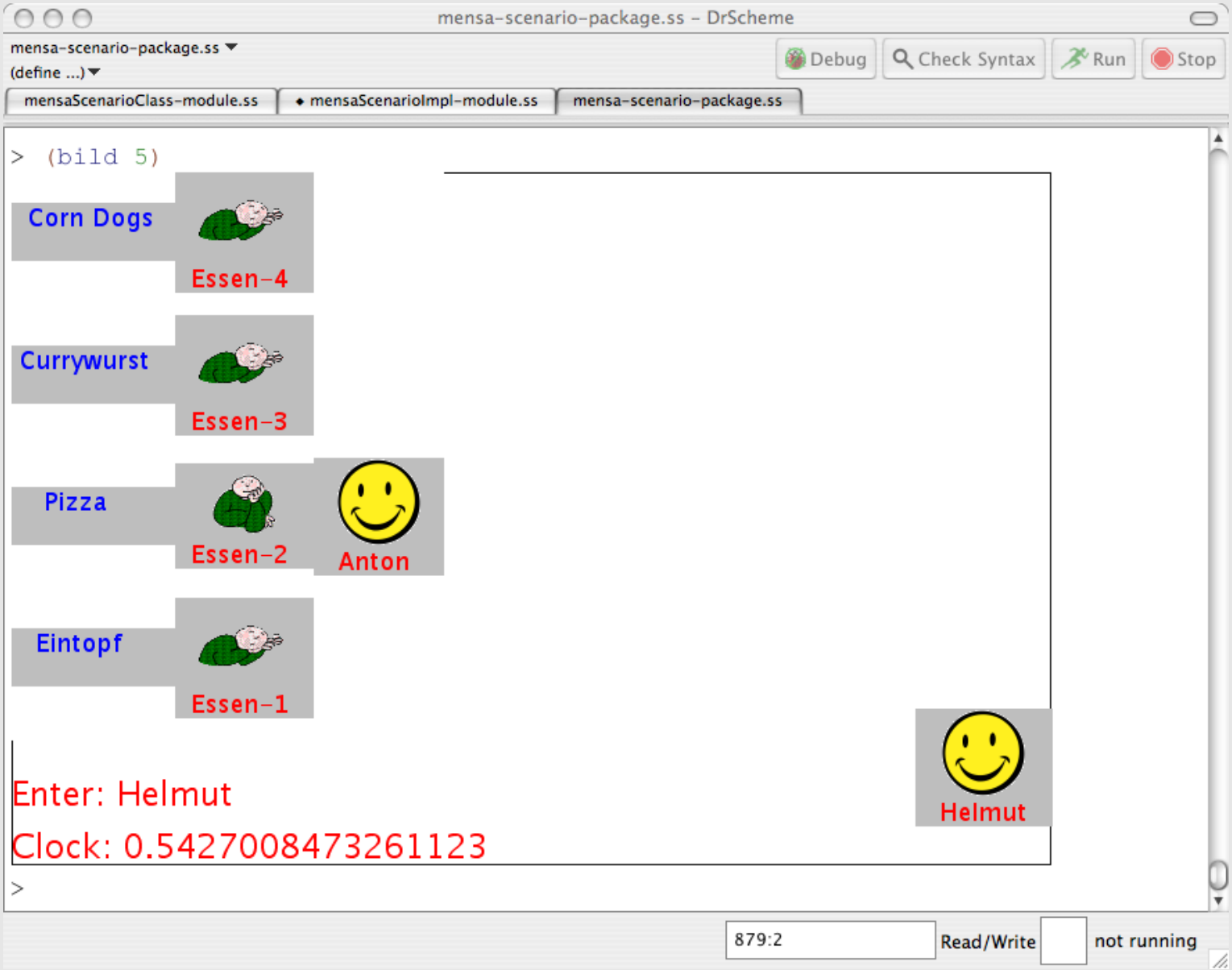
Ein Beispiellauf: Mensa-Szenario 1



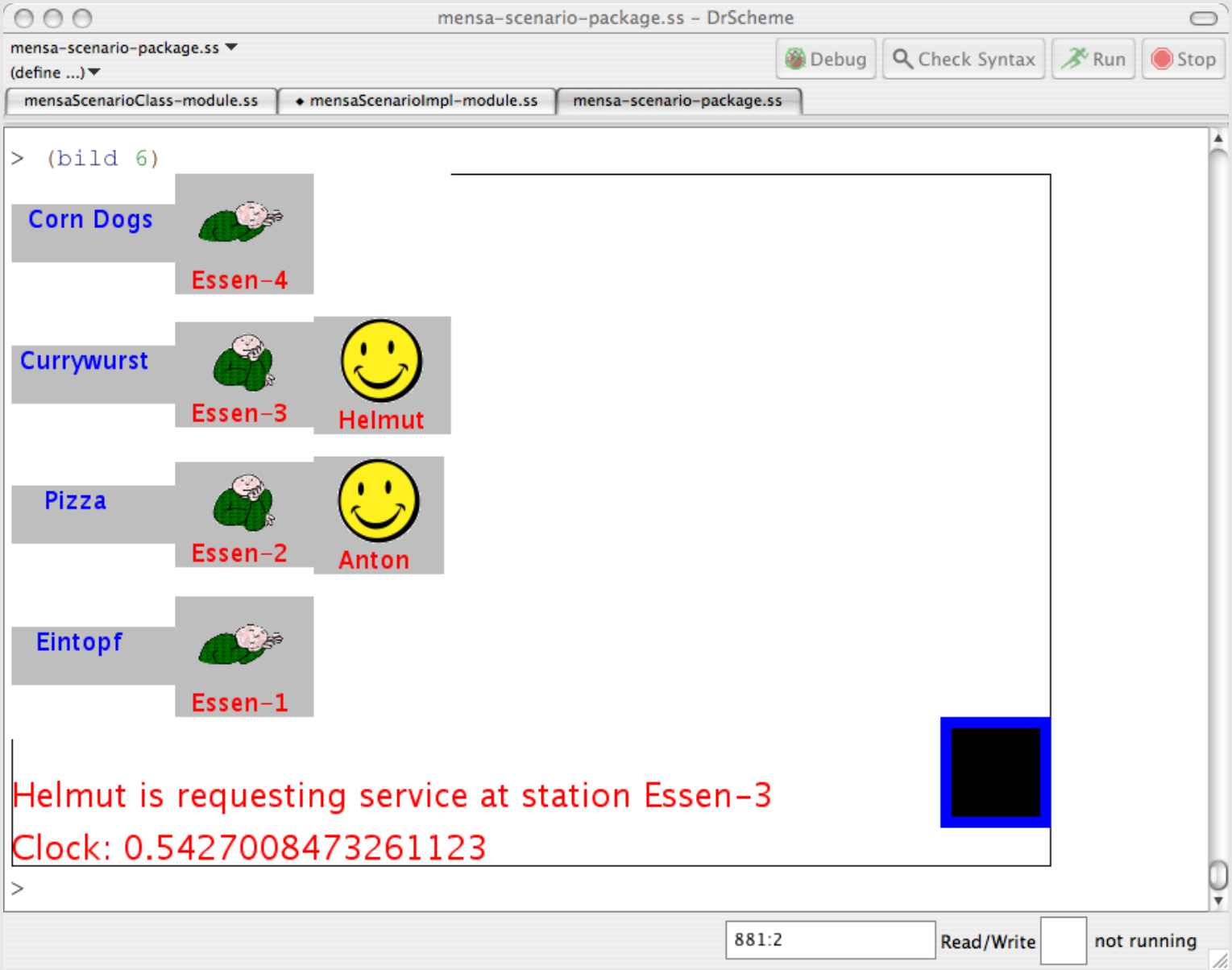
Ein Beispiellauf: Mensa-Szenario 1



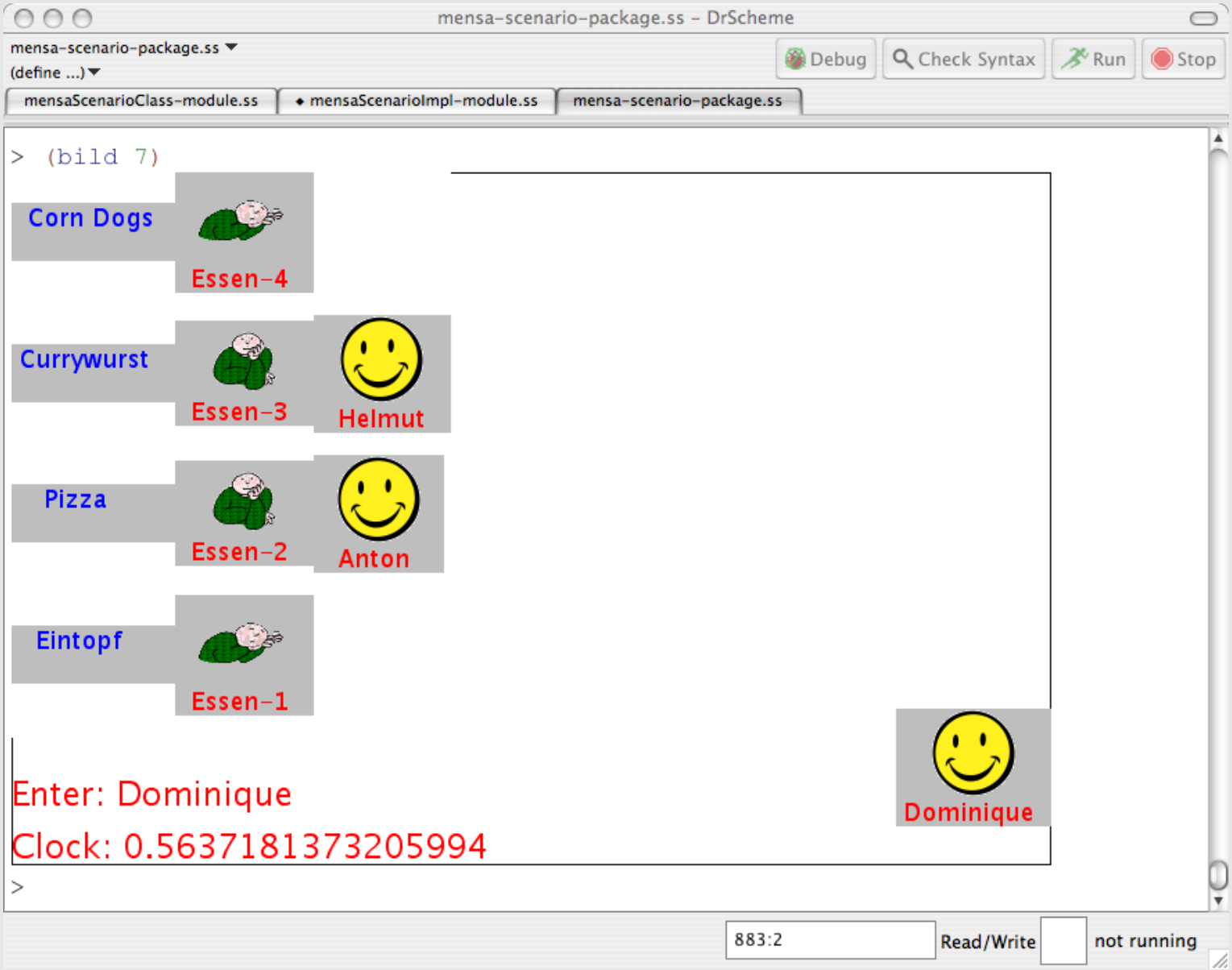
Ein Beispiellauf: Mensa-Szenario 1



Ein Beispiellauf: Mensa-Szenario 1

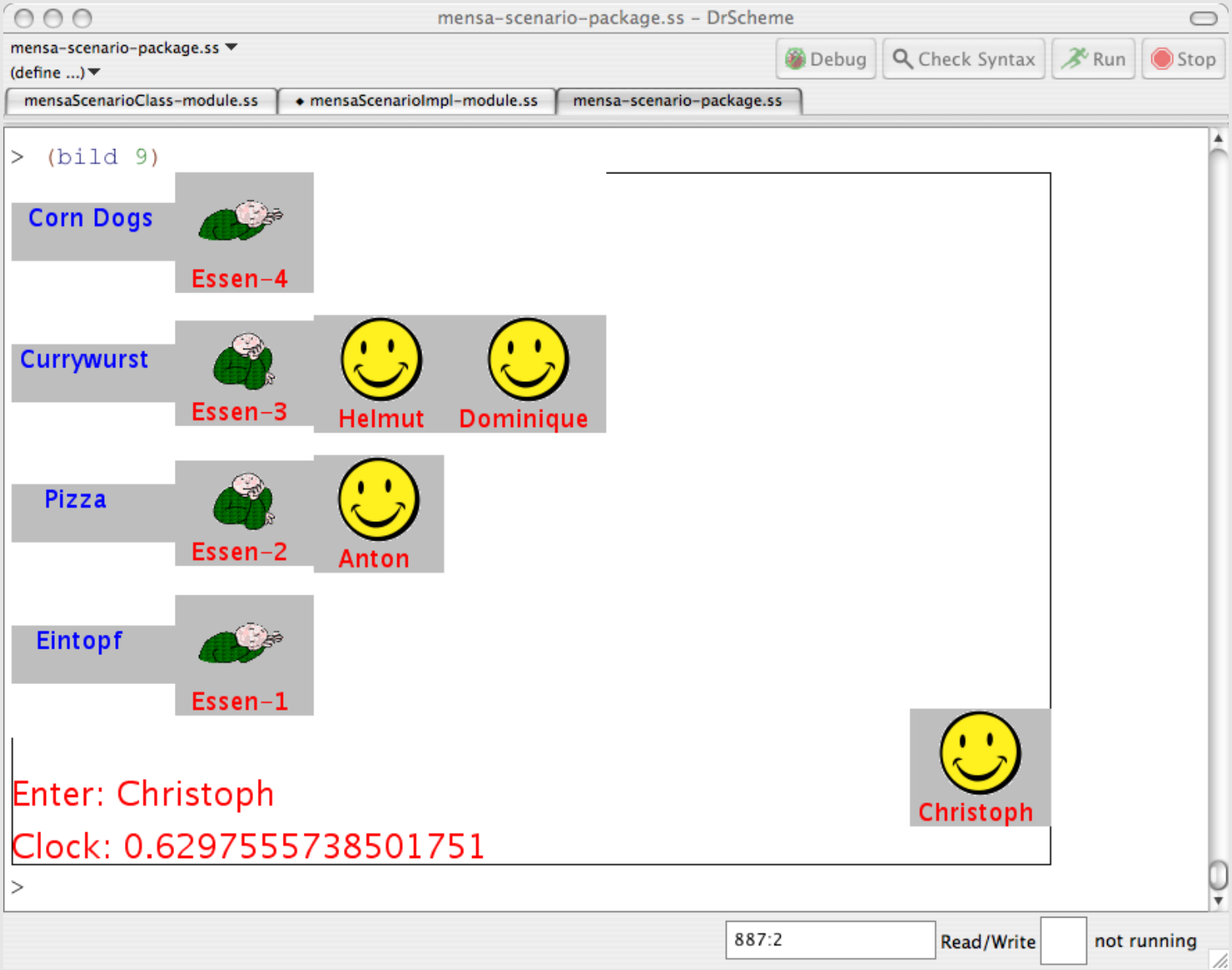


Ein Beispiellauf: Mensa-Szenario 1

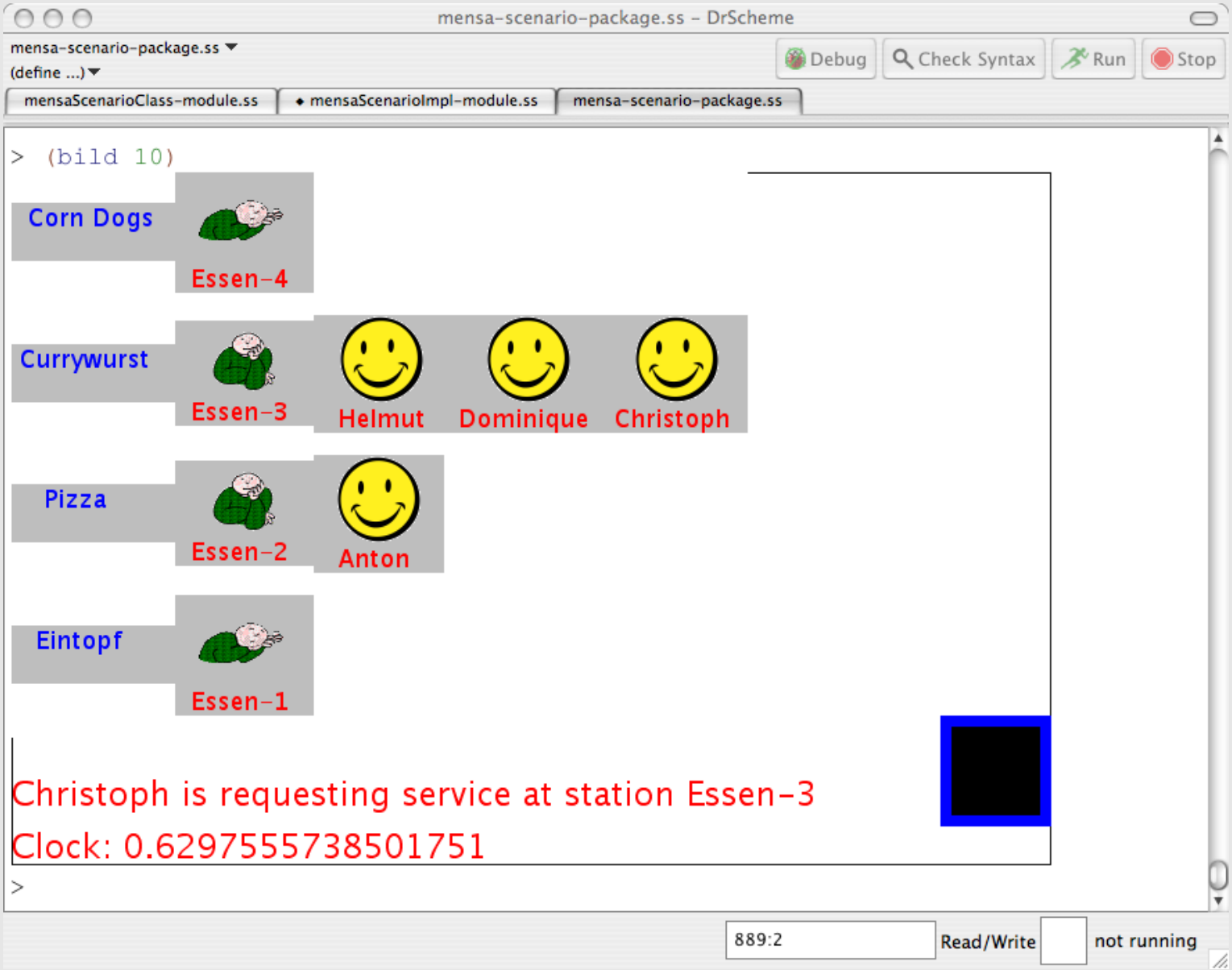


Ein Beispiellauf: Mensa-Szenario 1

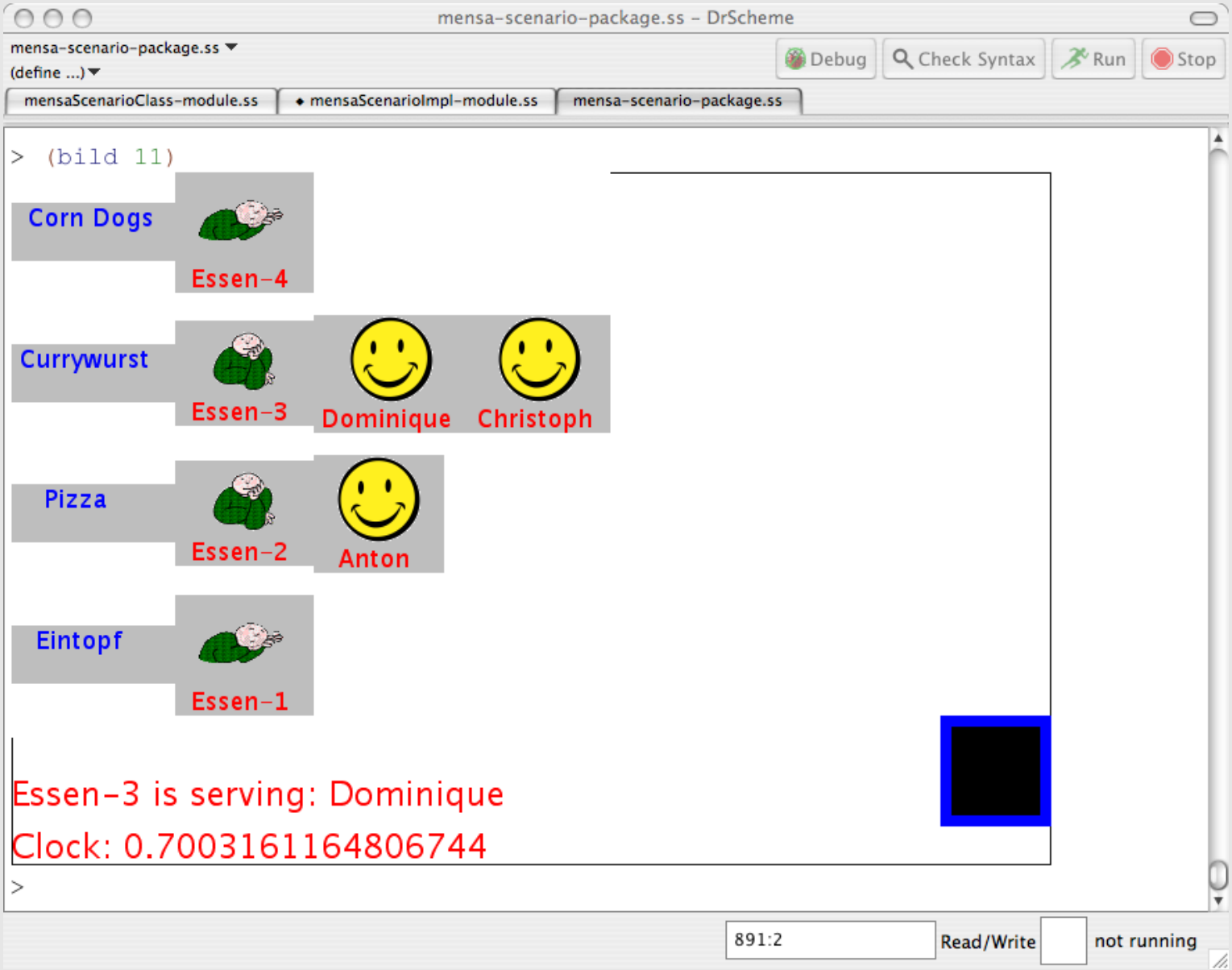
Ein Beispiellauf: Mensa-Szenario 1



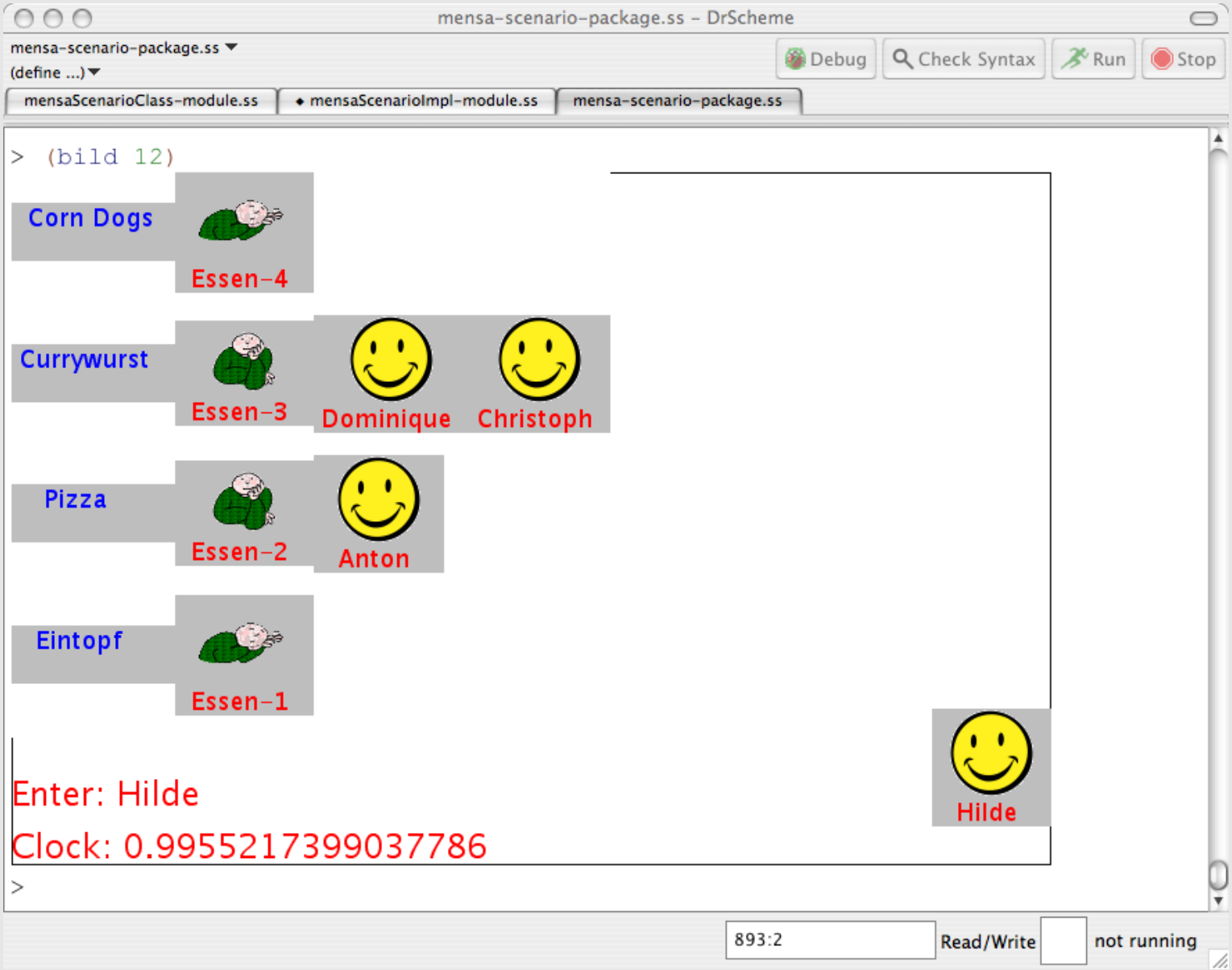
Ein Beispiellauf: Mensa-Szenario 1



Ein Beispiellauf: Mensa-Szenario 1

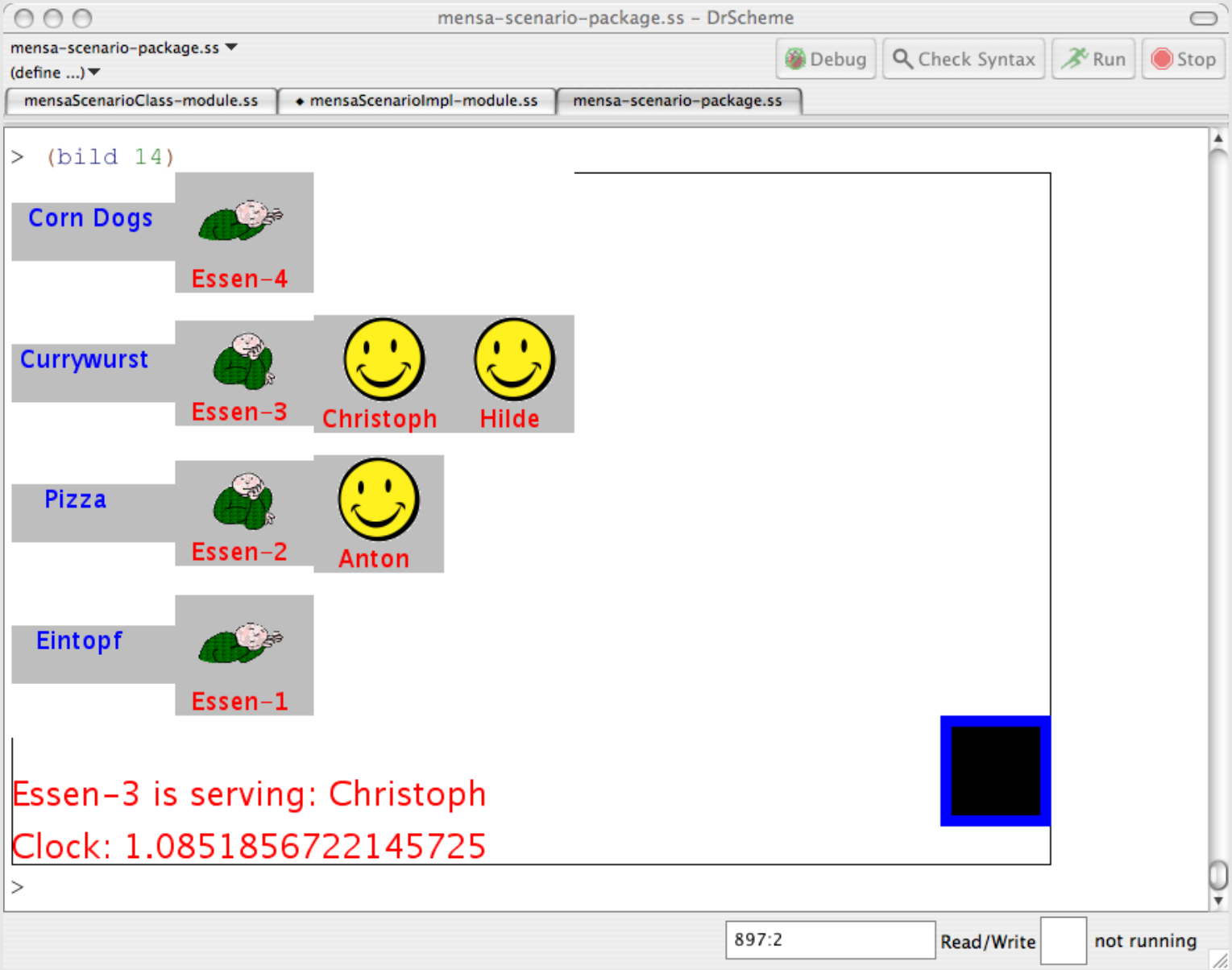


Ein Beispiellauf: Mensa-Szenario 1



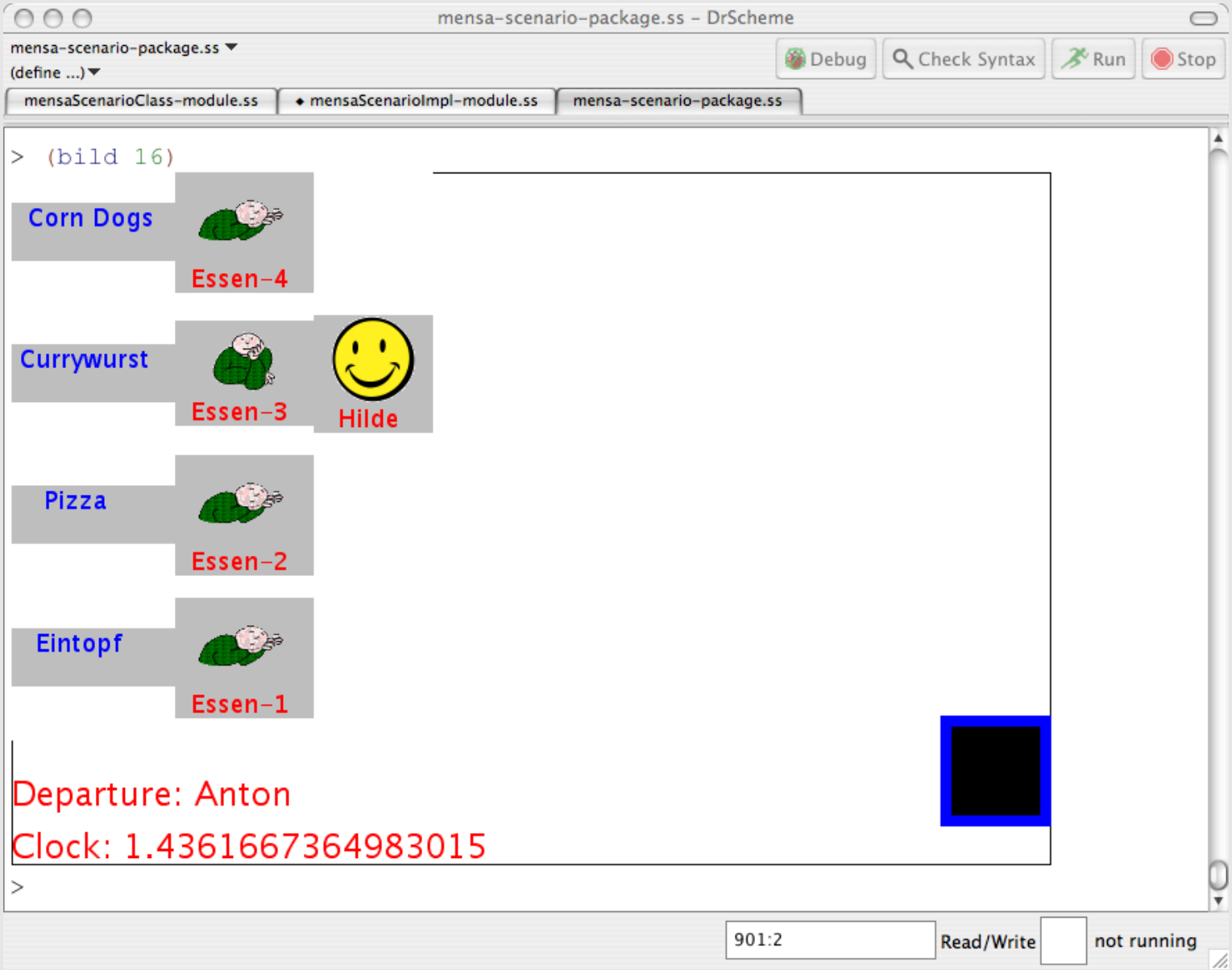
Ein Beispiellauf: Mensa-Szenario 1

Ein Beispiellauf: Mensa-Szenario 1

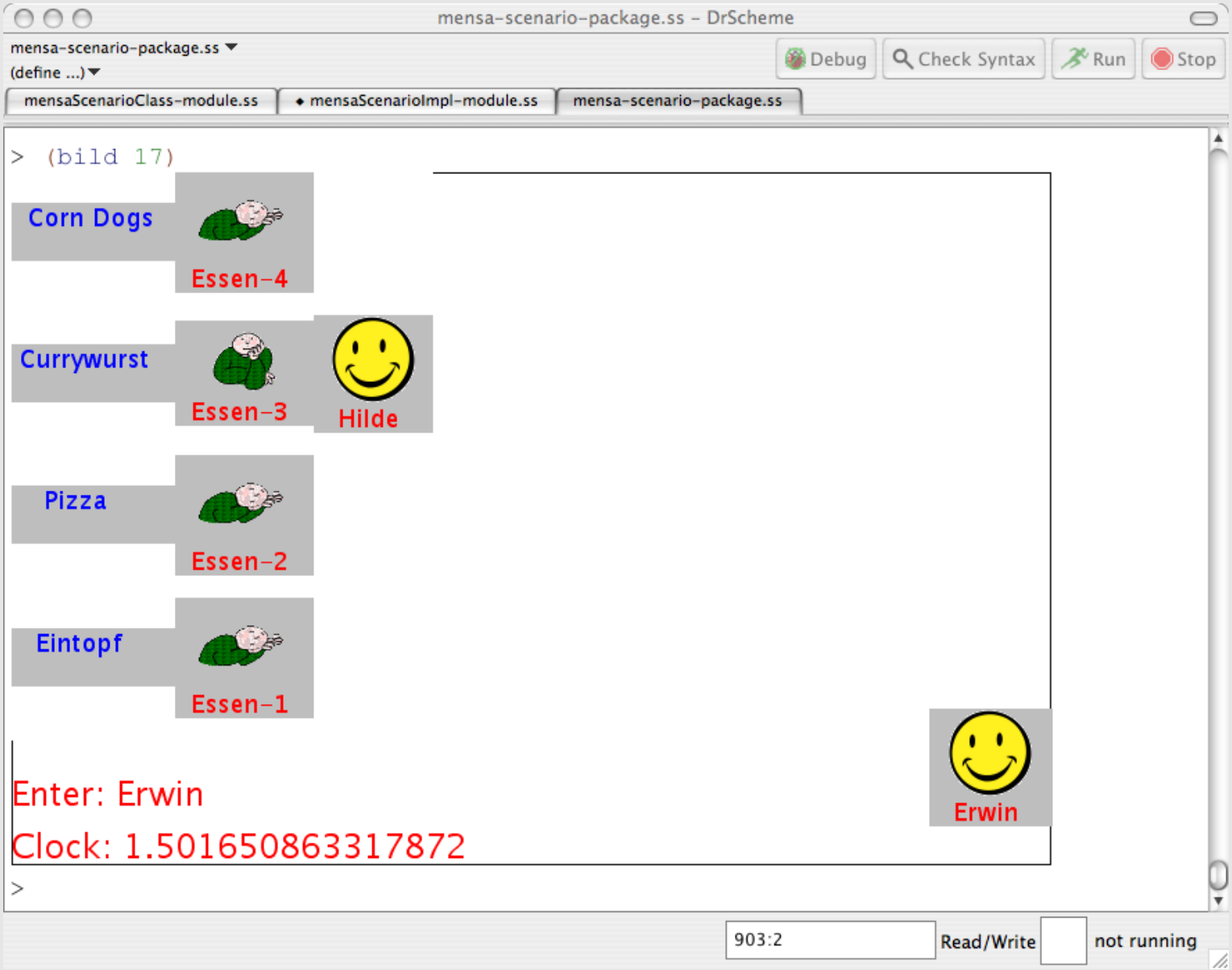


Ein Beispiellauf: Mensa-Szenario 1

Ein Beispiellauf: Mensa-Szenario 1

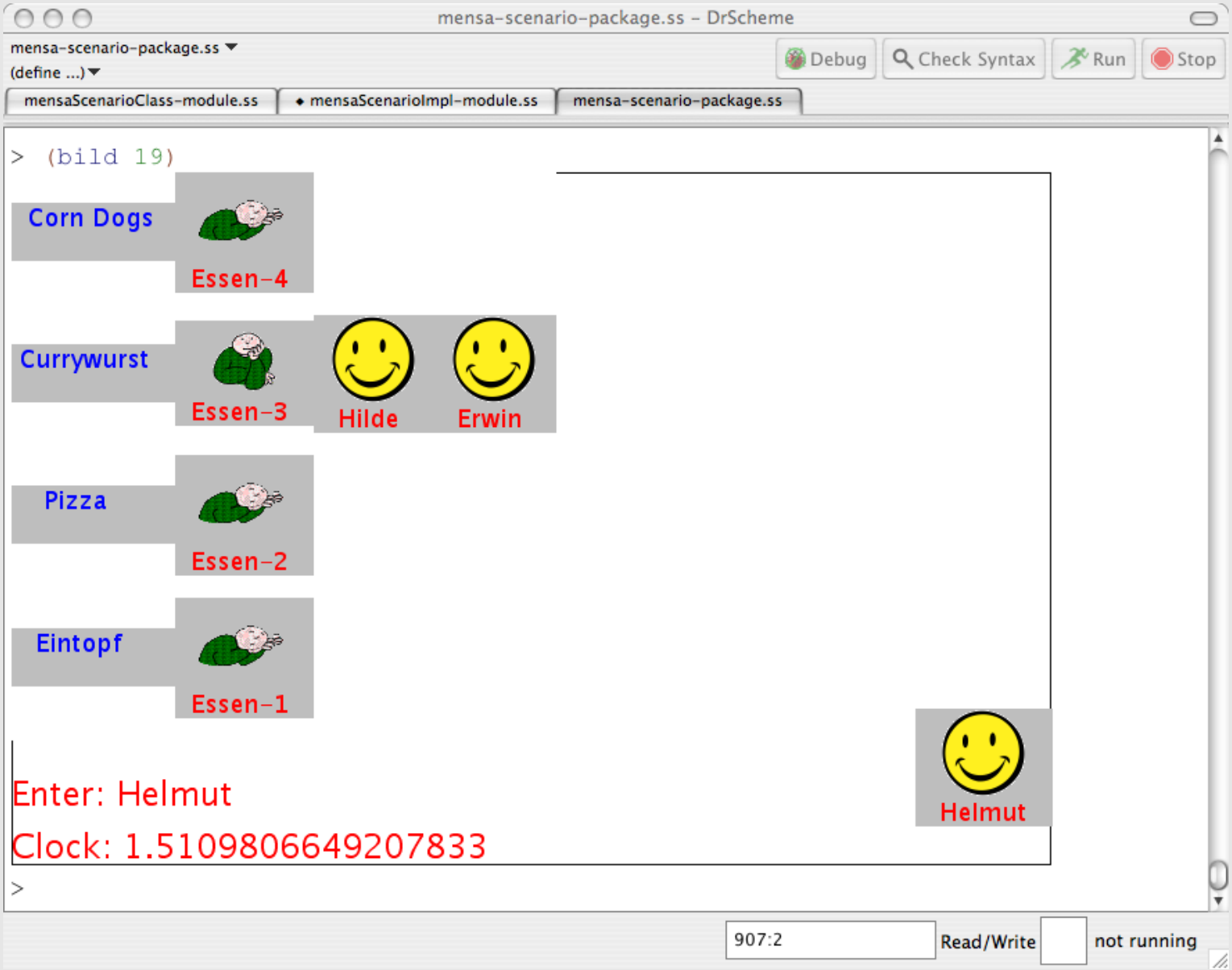


Ein Beispiellauf: Mensa-Szenario 1

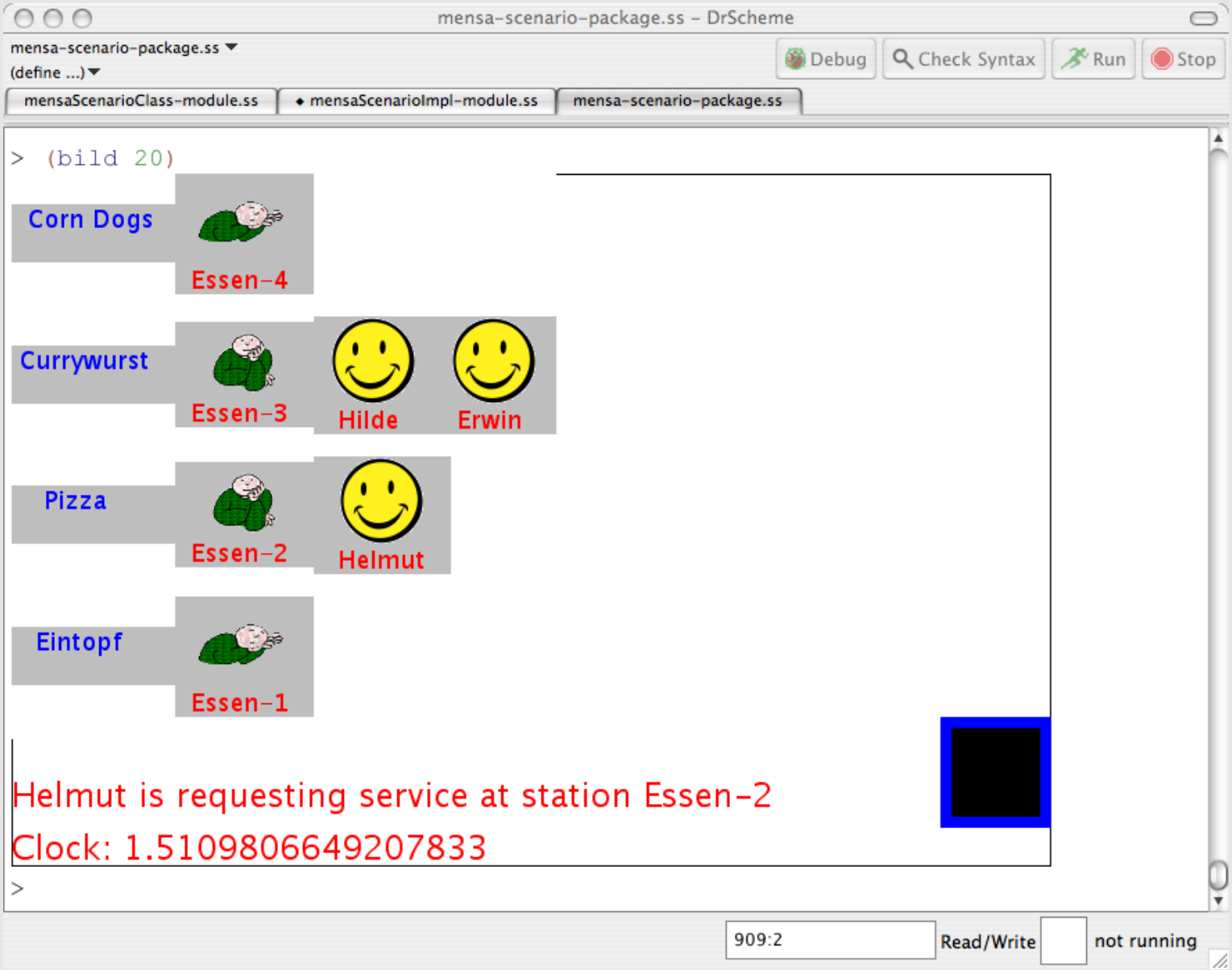


Ein Beispiellauf: Mensa-Szenario 1

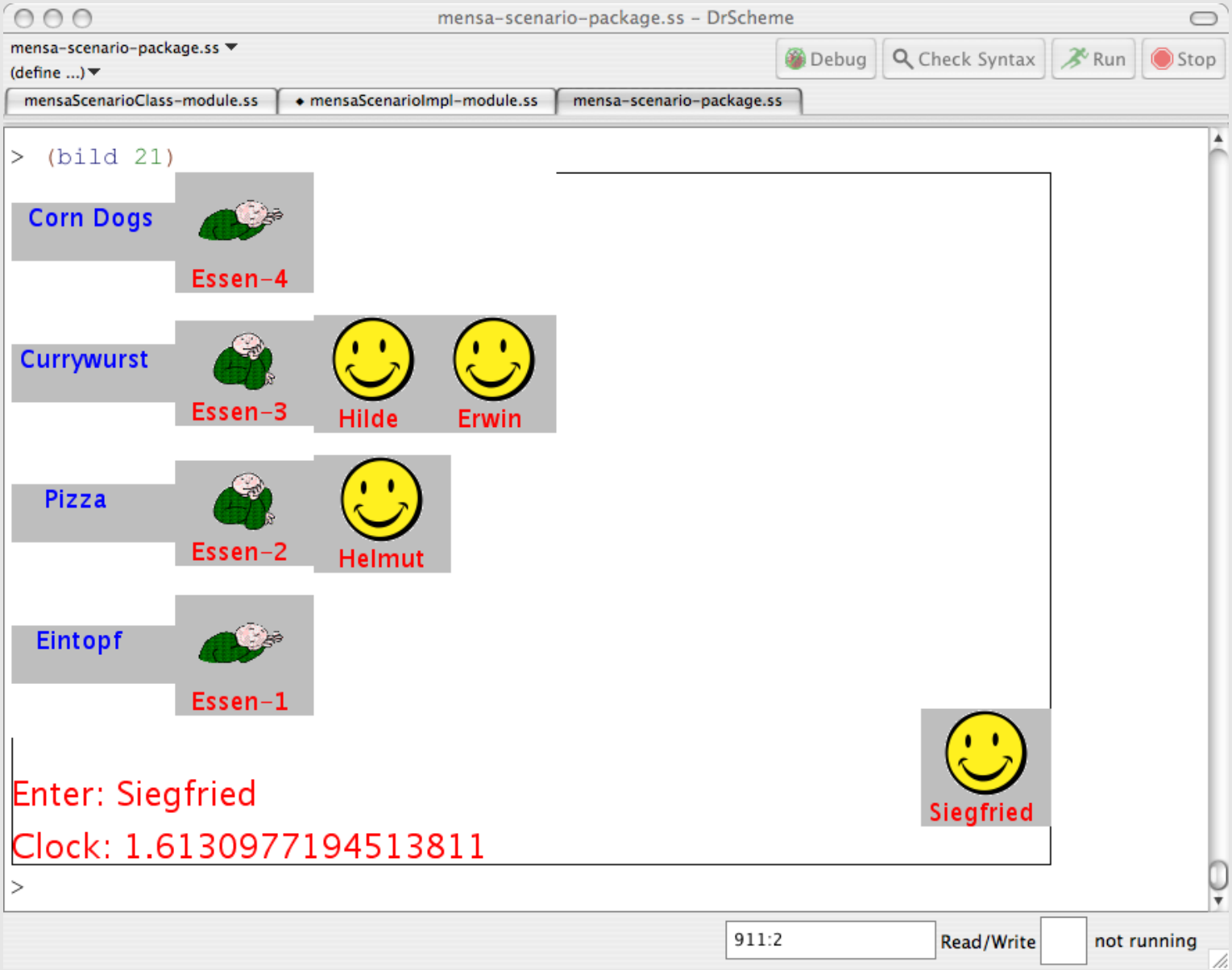
Ein Beispiellauf: Mensa-Szenario 1



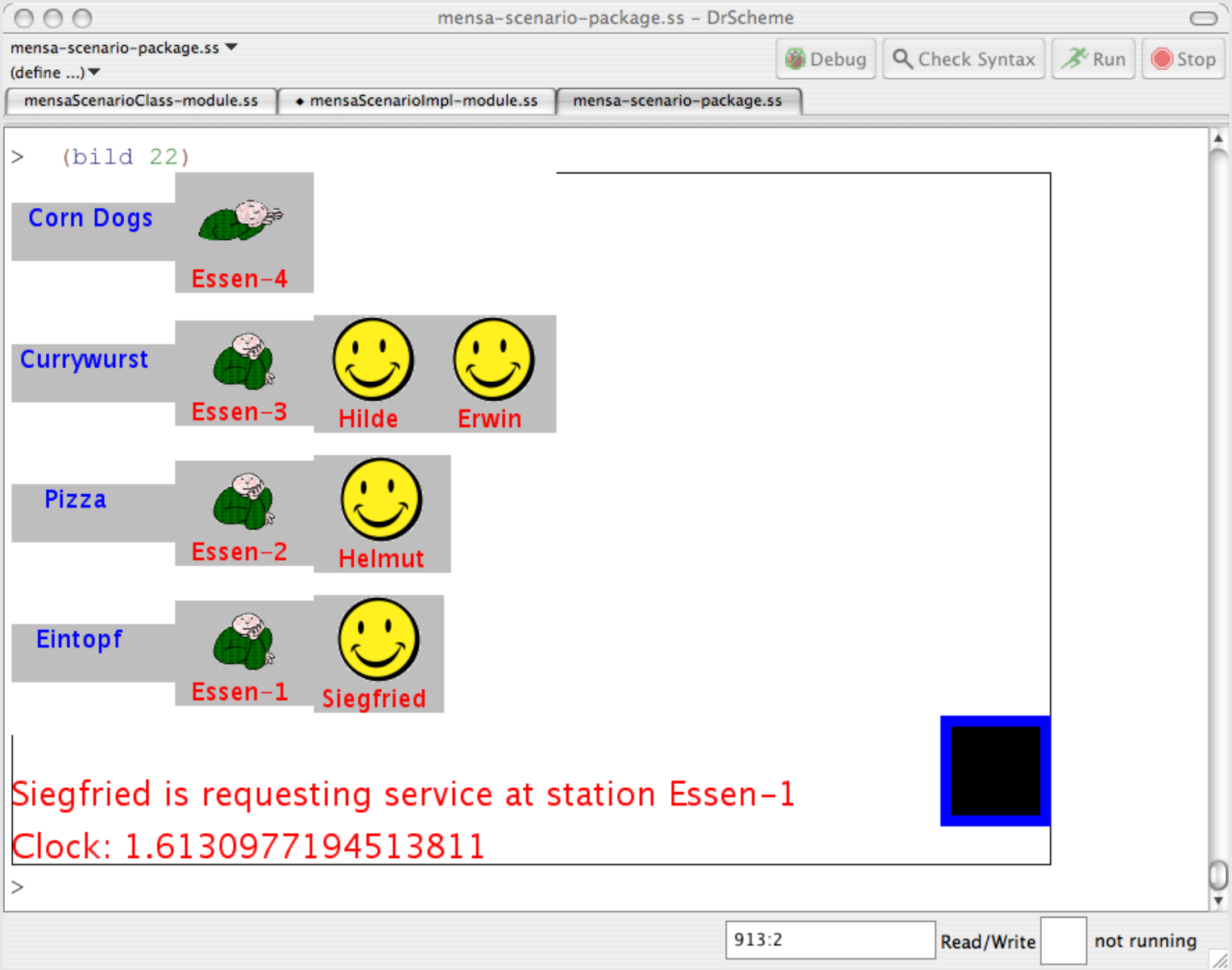
Ein Beispiellauf: Mensa-Szenario 1



Ein Beispiellauf: Mensa-Szenario 1

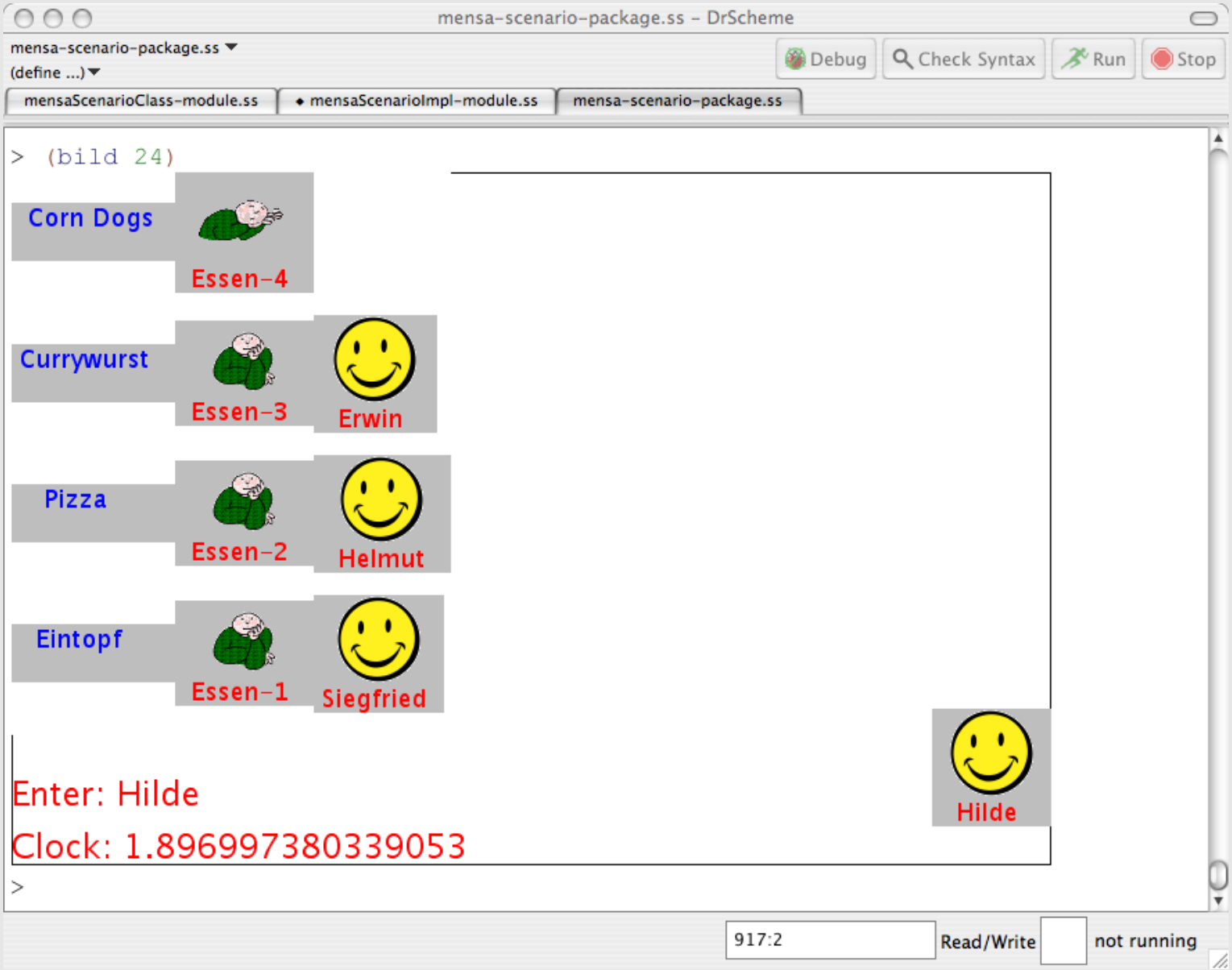


Ein Beispiellauf: Mensa-Szenario 1



Ein Beispiellauf: Mensa-Szenario 1

Ein Beispiellauf: Mensa-Szenario 1

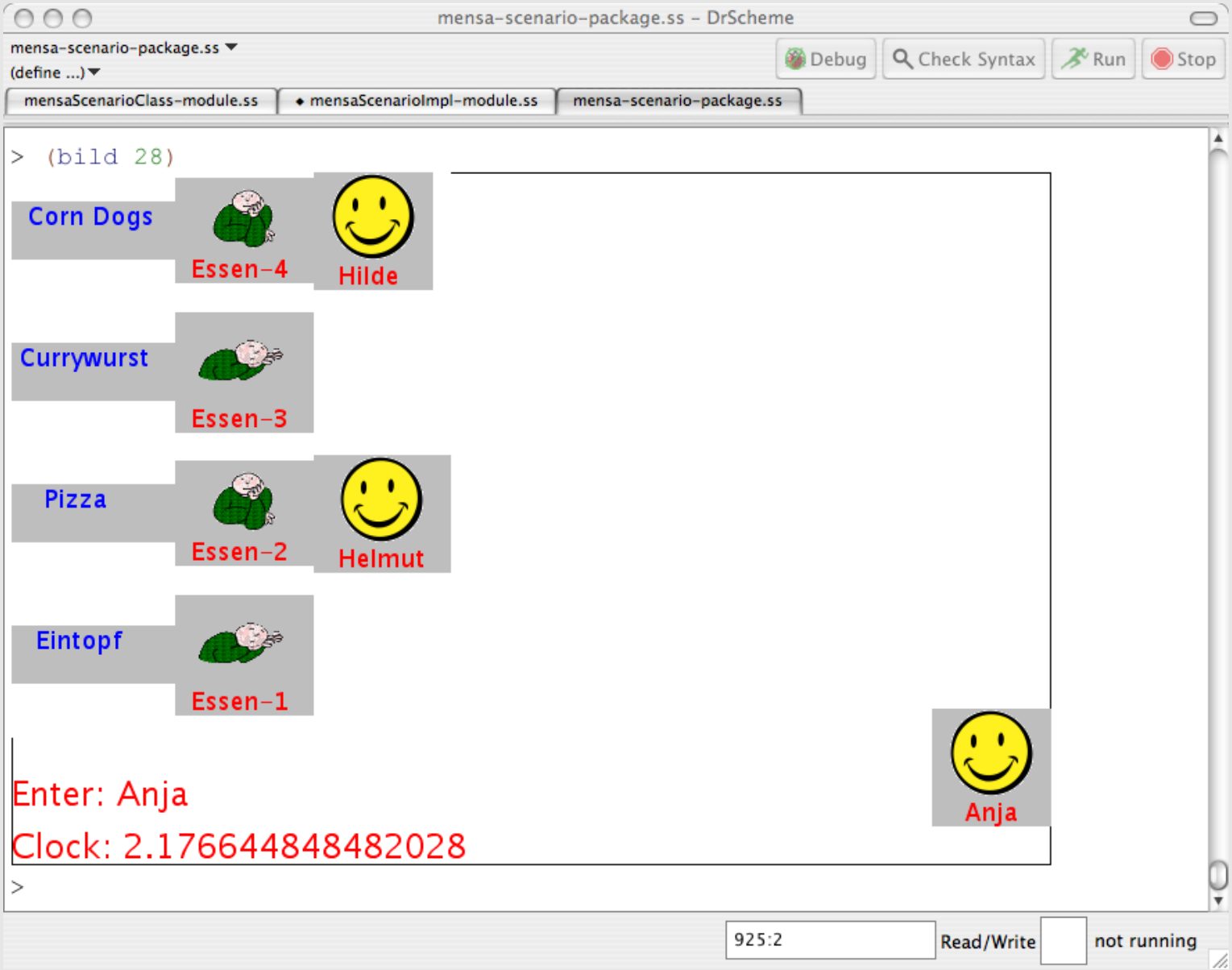


Ein Beispiellauf: Mensa-Szenario 1

Ein Beispiellauf: Mensa-Szenario 1

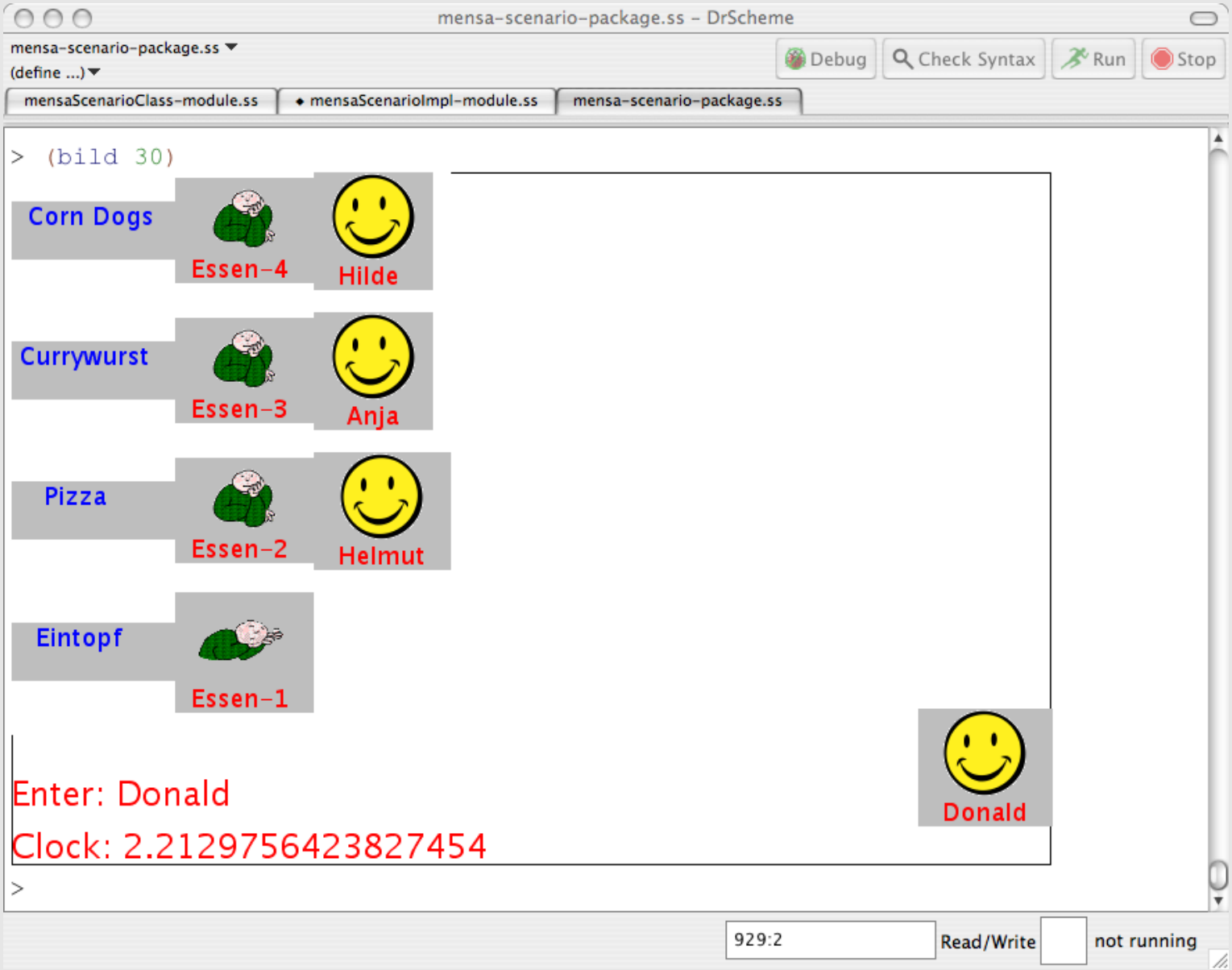
Ein Beispiellauf: Mensa-Szenario 1

Ein Beispiellauf: Mensa-Szenario 1

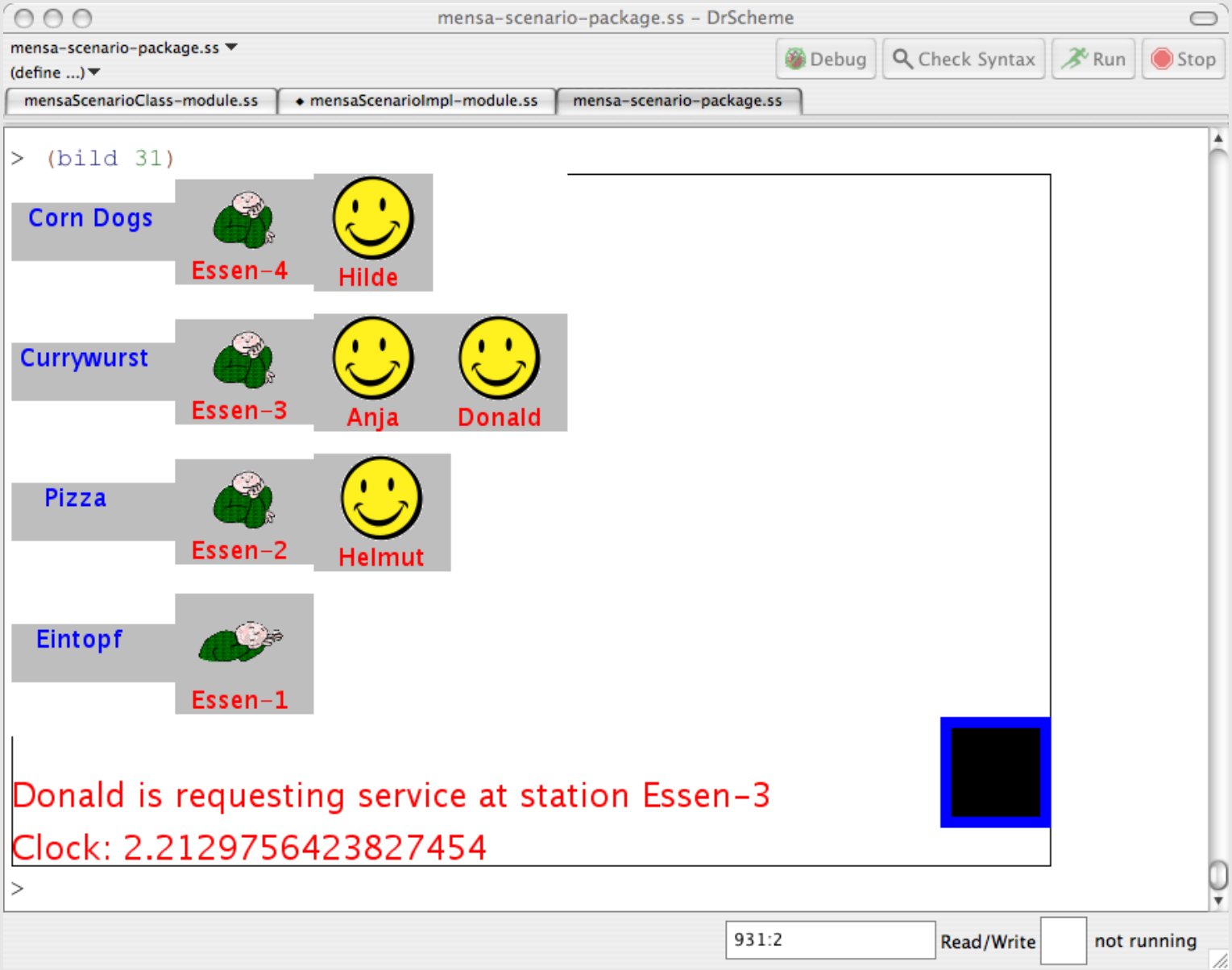


Ein Beispiellauf: Mensa-Szenario 1

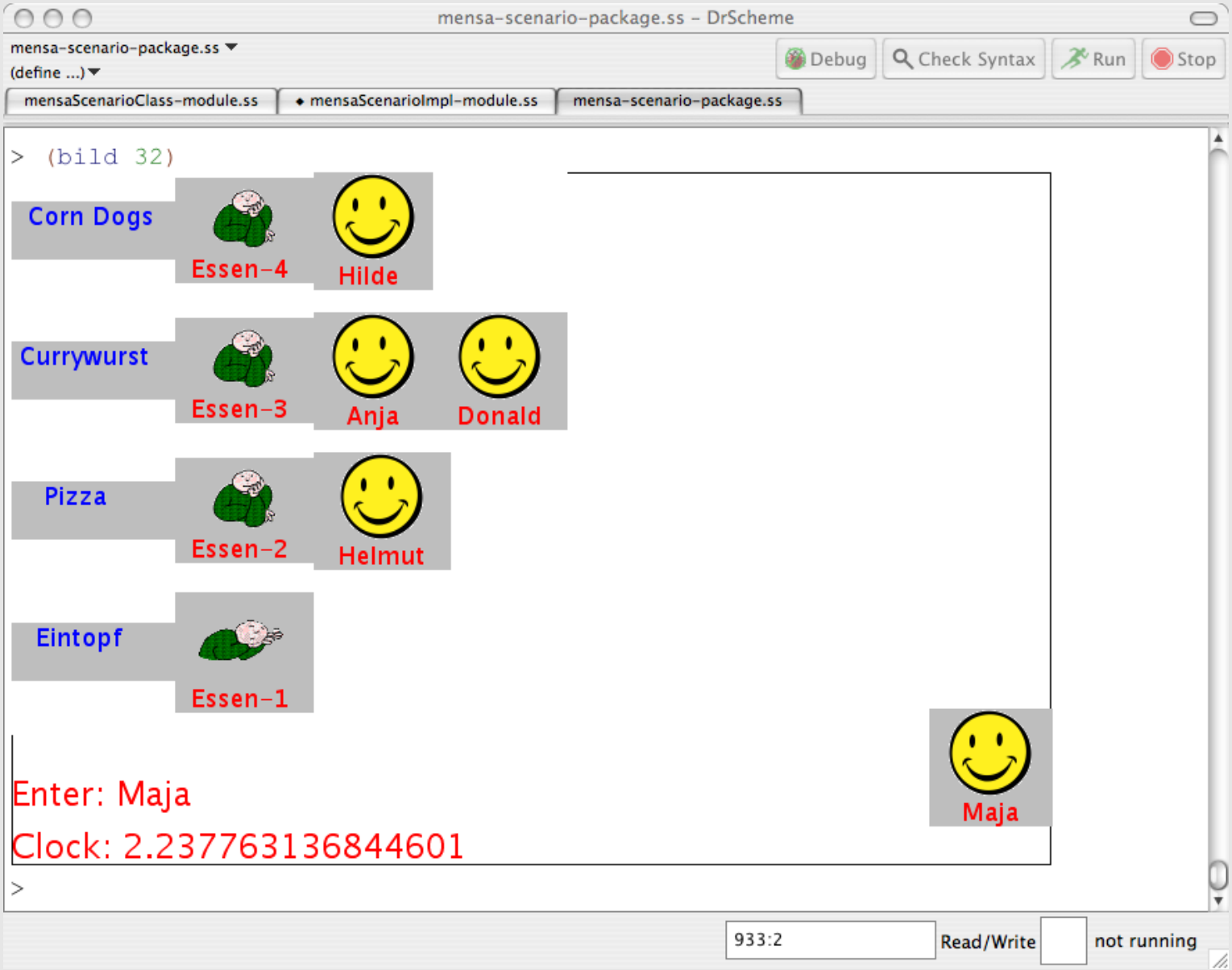
Ein Beispiellauf: Mensa-Szenario 1



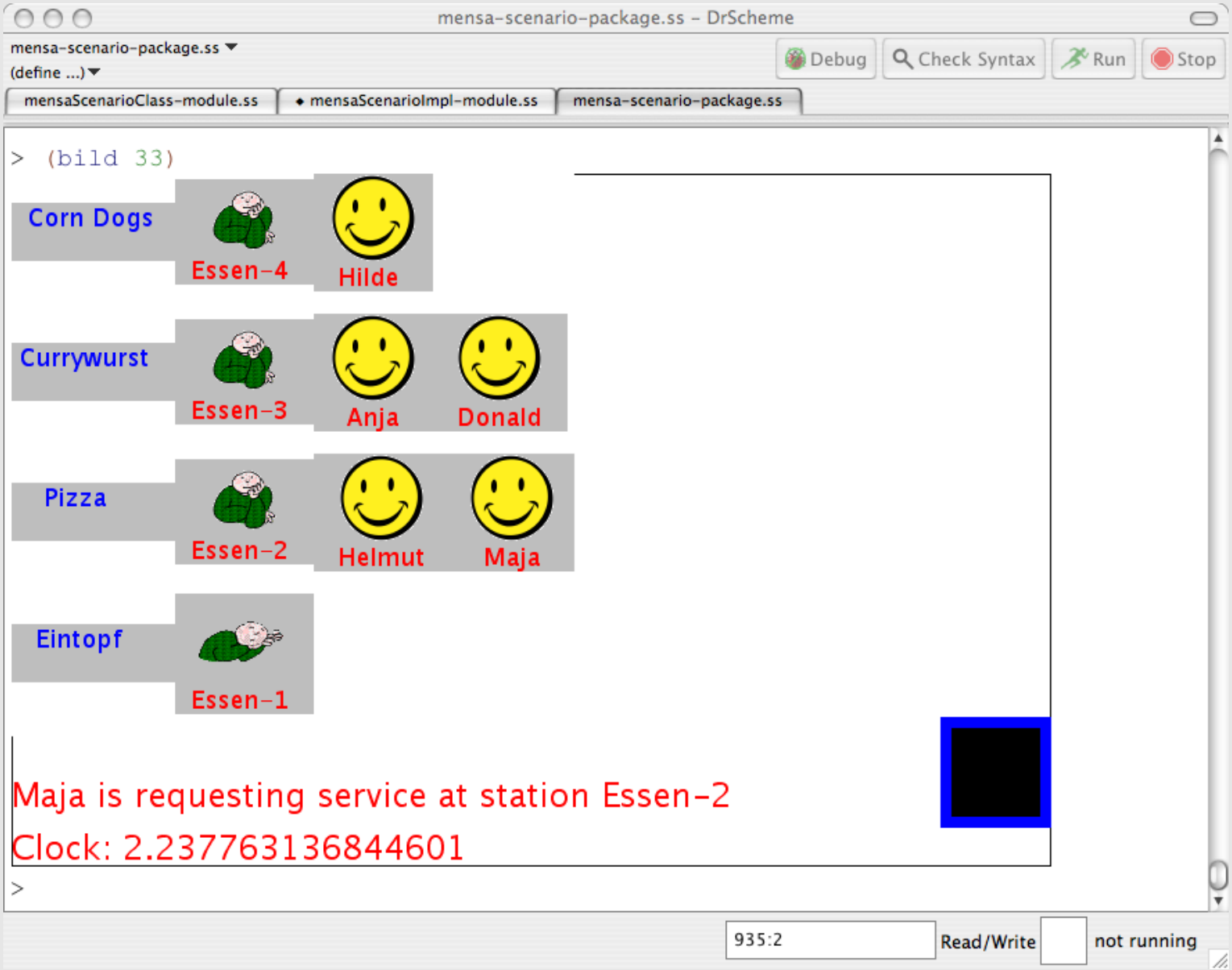
Ein Beispiellauf: Mensa-Szenario 1



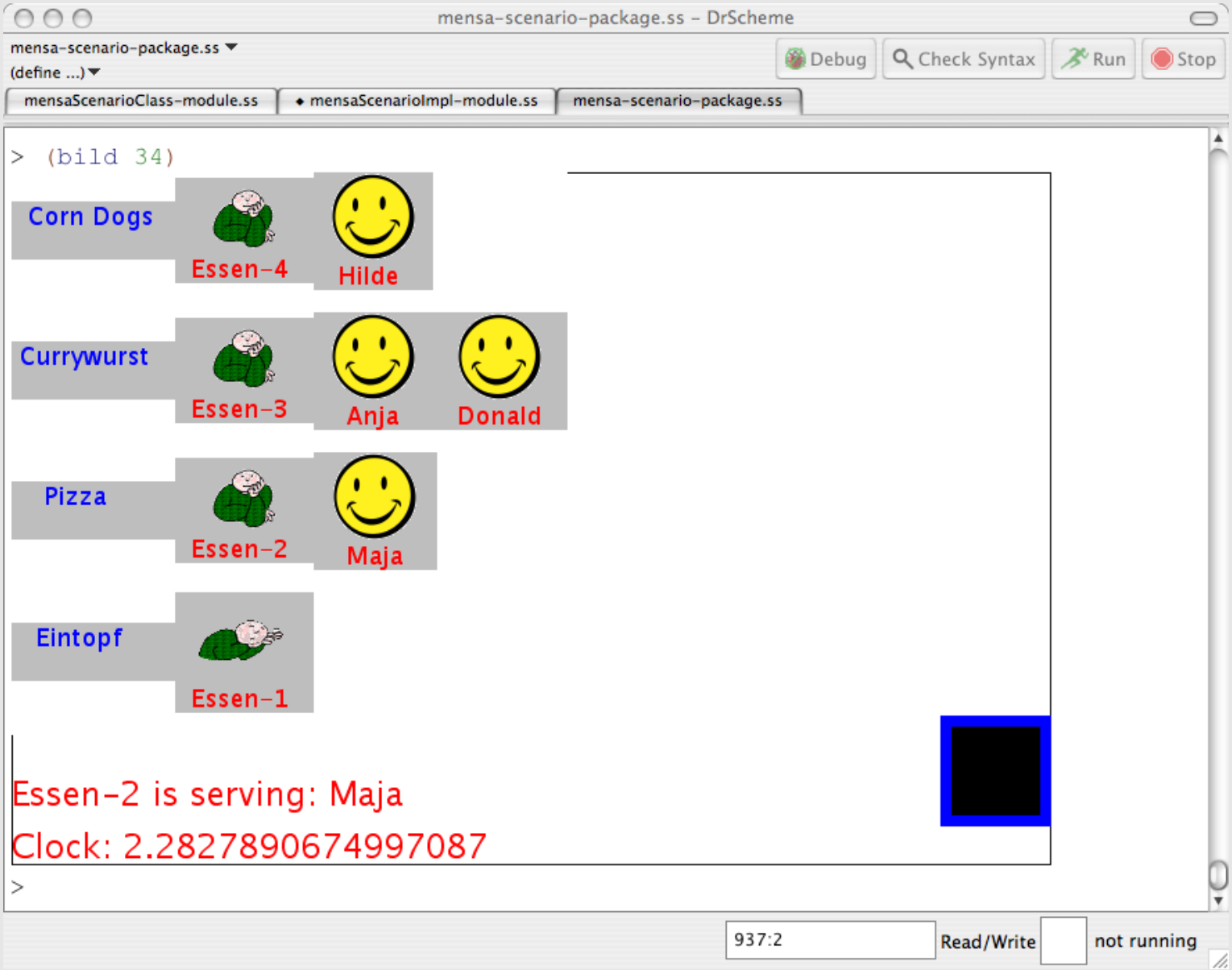
Ein Beispiellauf: Mensa-Szenario 1



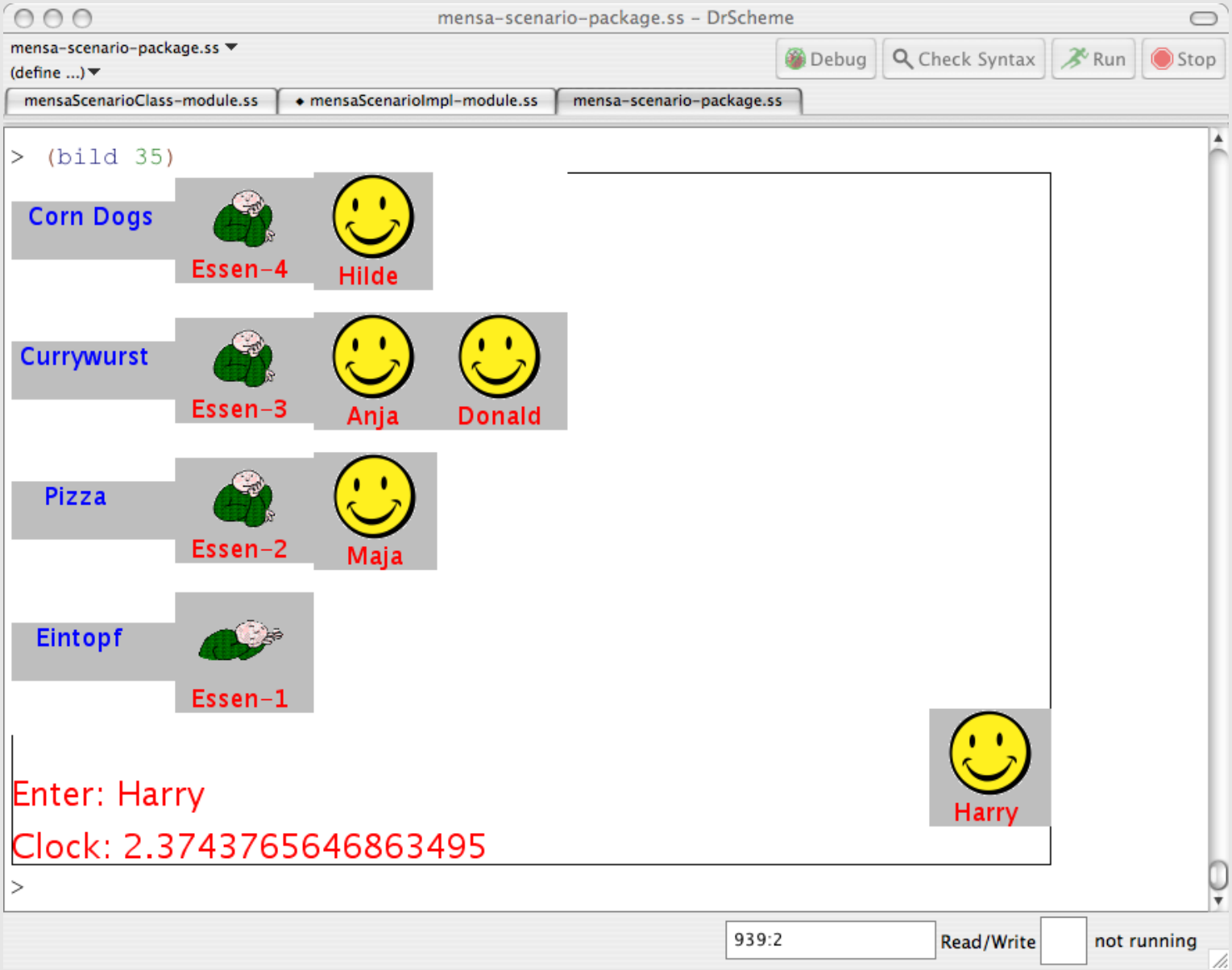
Ein Beispiellauf: Mensa-Szenario 1



Ein Beispiellauf: Mensa-Szenario 1



Ein Beispiellauf: Mensa-Szenario 1



Ein Beispiellauf: Mensa-Szenario 1

Ein Beispiellauf: Mensa-Szenario 1

Ein ereignisorientiertes Simulationssysstem

Mensa-Szenario 2:
Spezialisierte Essensausgaben

Beispiel (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))
```

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

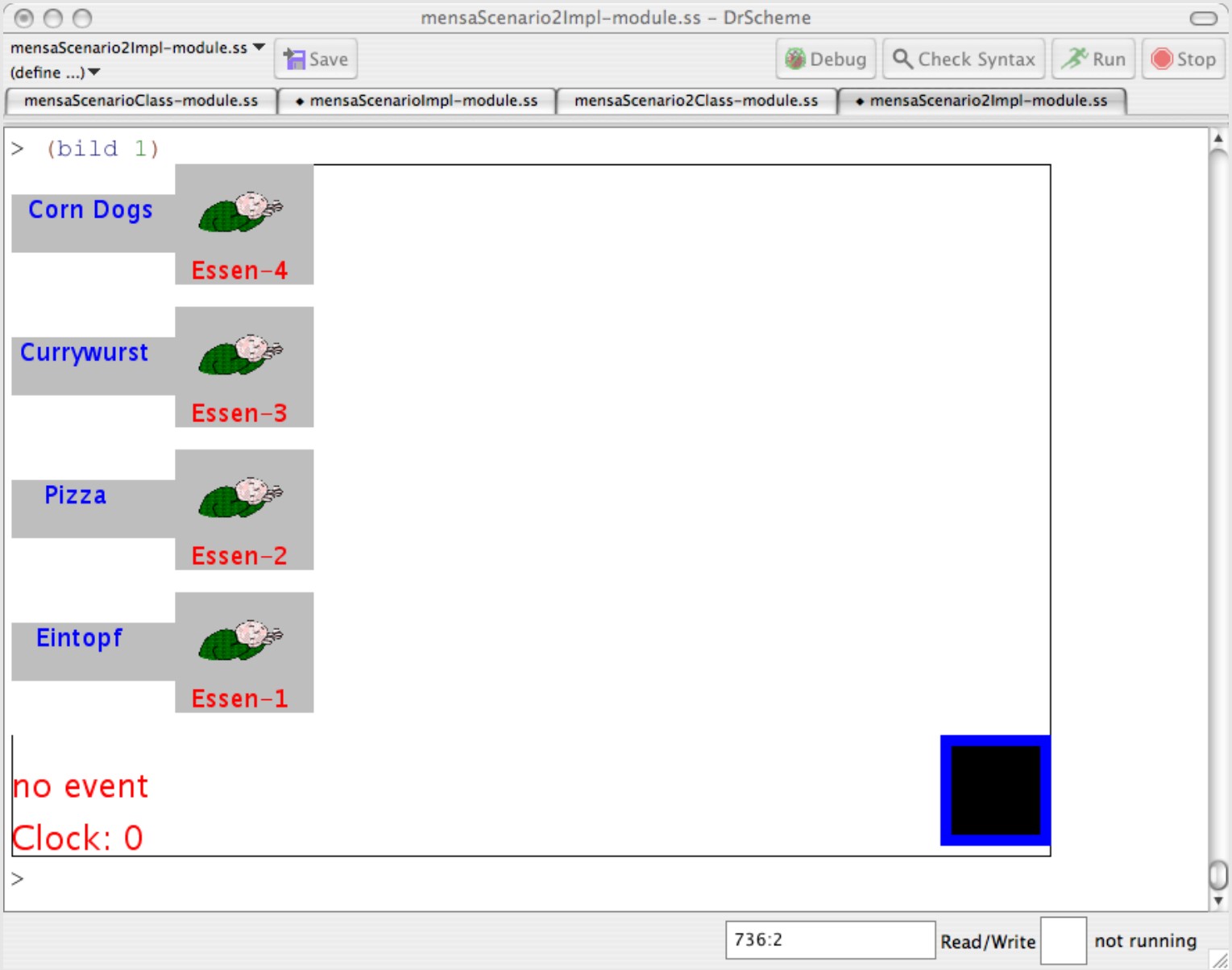
Ein Rundruf an alle Objekte der Klasse server-station:

```
(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*))
```

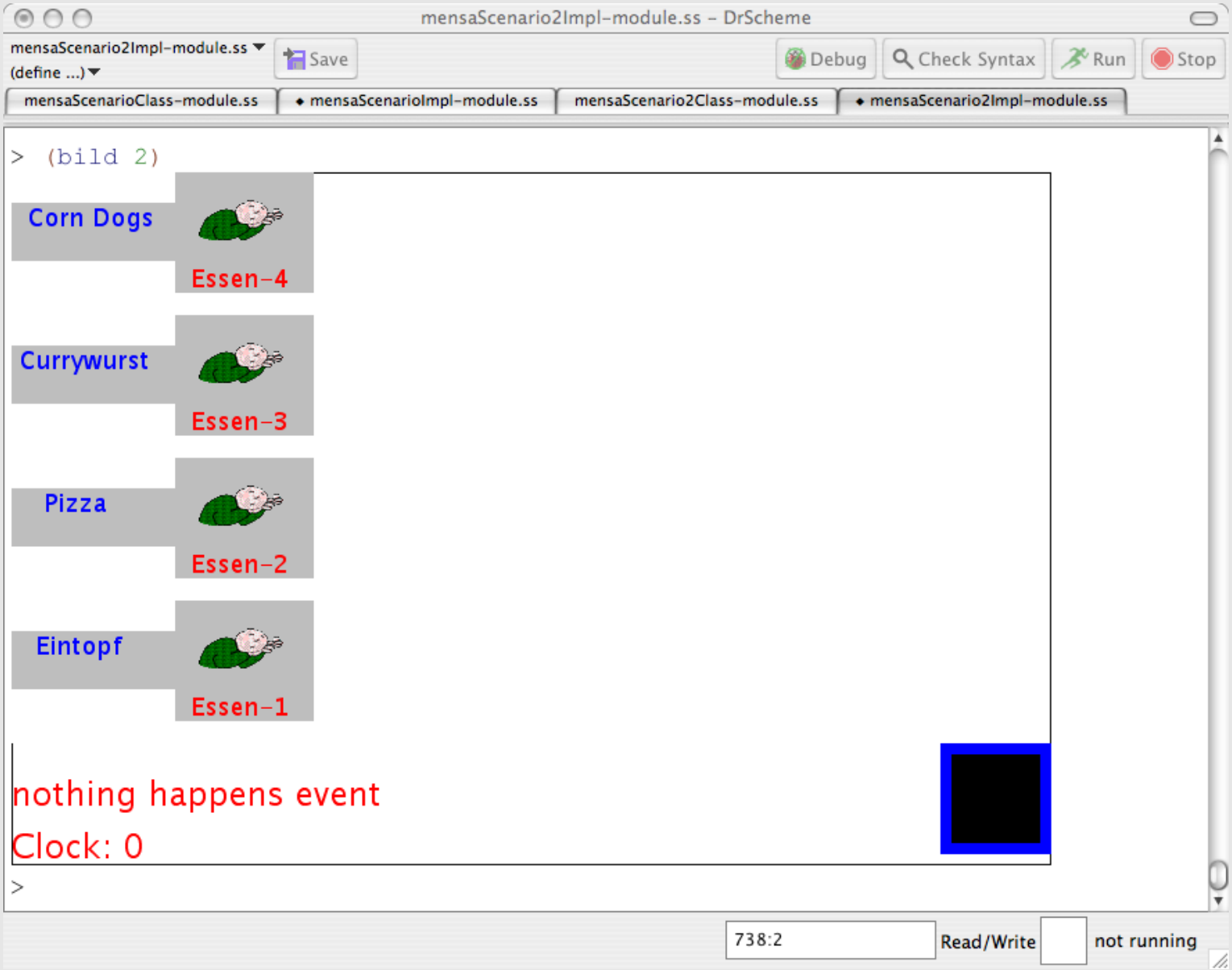
Ein Beispiellauf

```
Language: Swindle.
> (require se3-bib/sim/mensa-scenario2-package)

> (mensa2Demo)
```

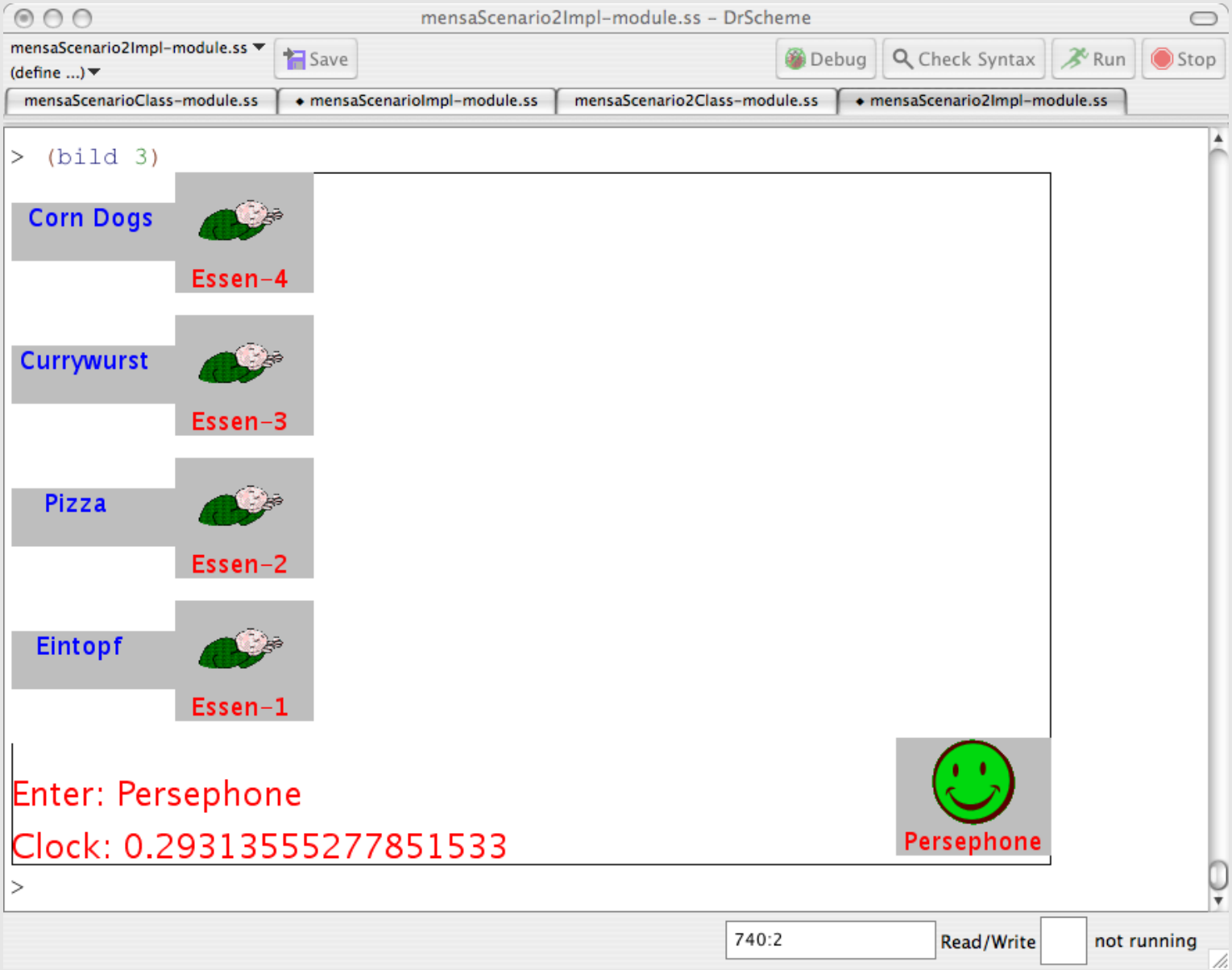
Ein Beispiellauf: Mensa-Szenario 2



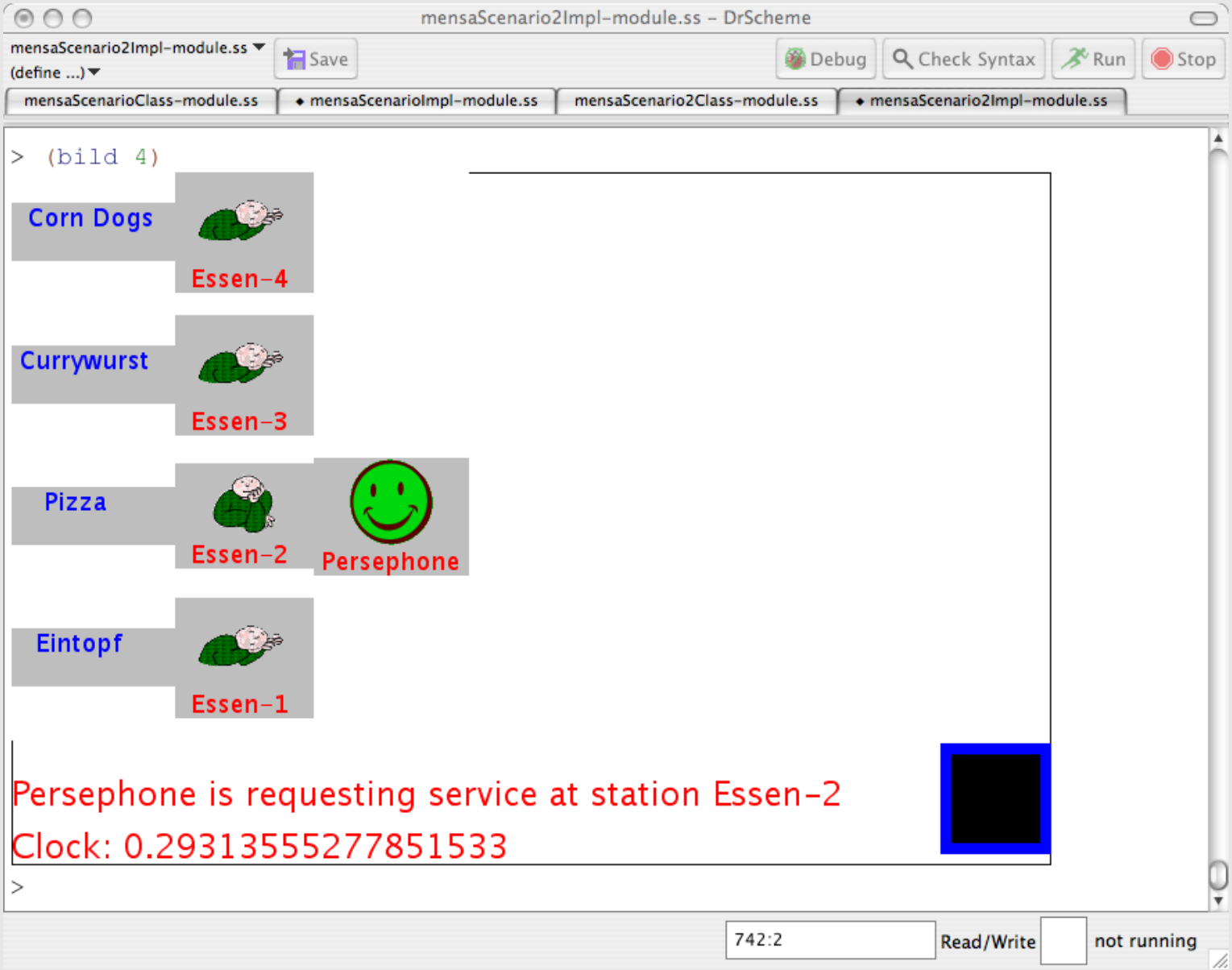
Ein Beispiellauf: Mensa-Szenario 2



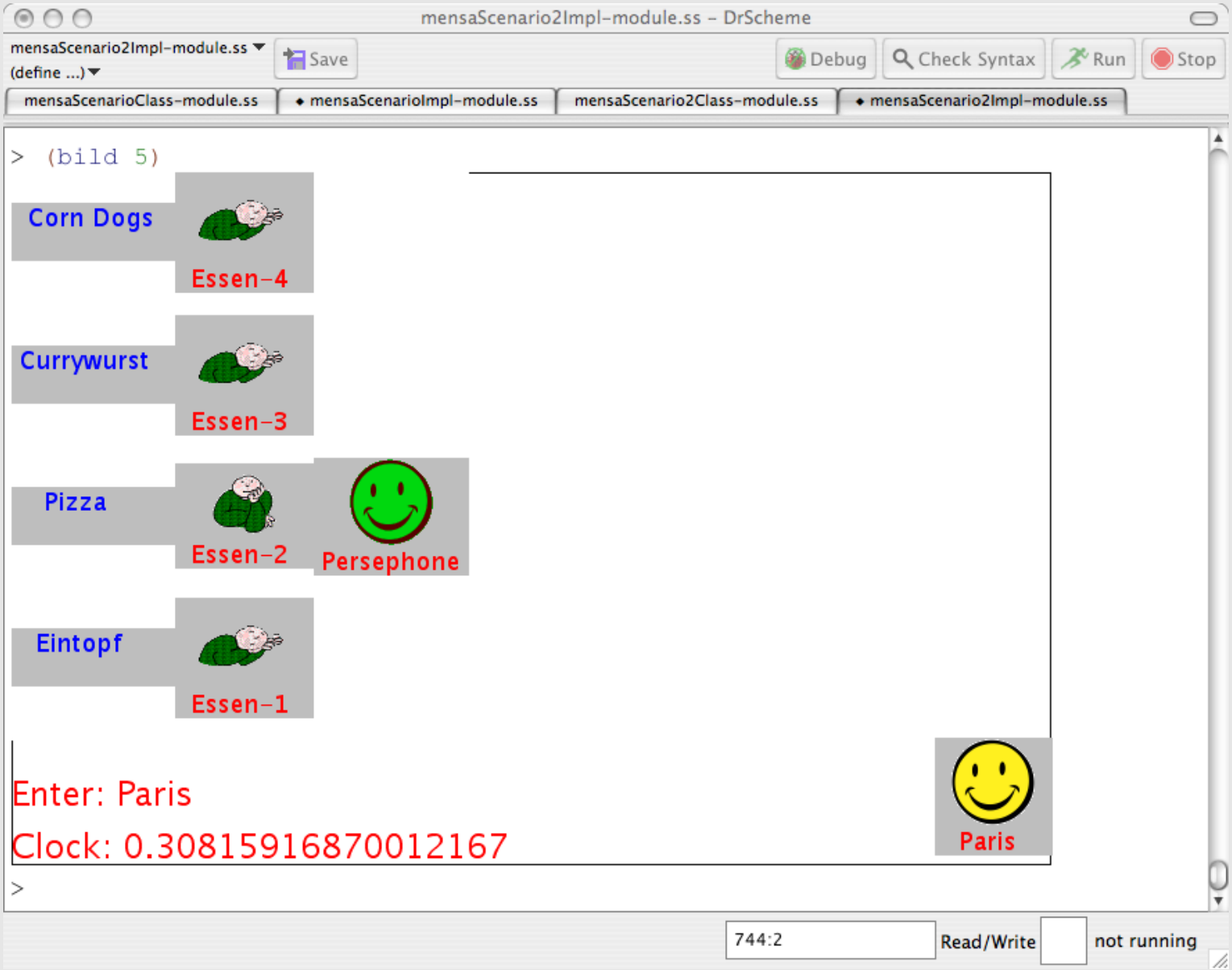
Ein Beispiellauf: Mensa-Szenario 2



Ein Beispiellauf: Mensa-Szenario 2

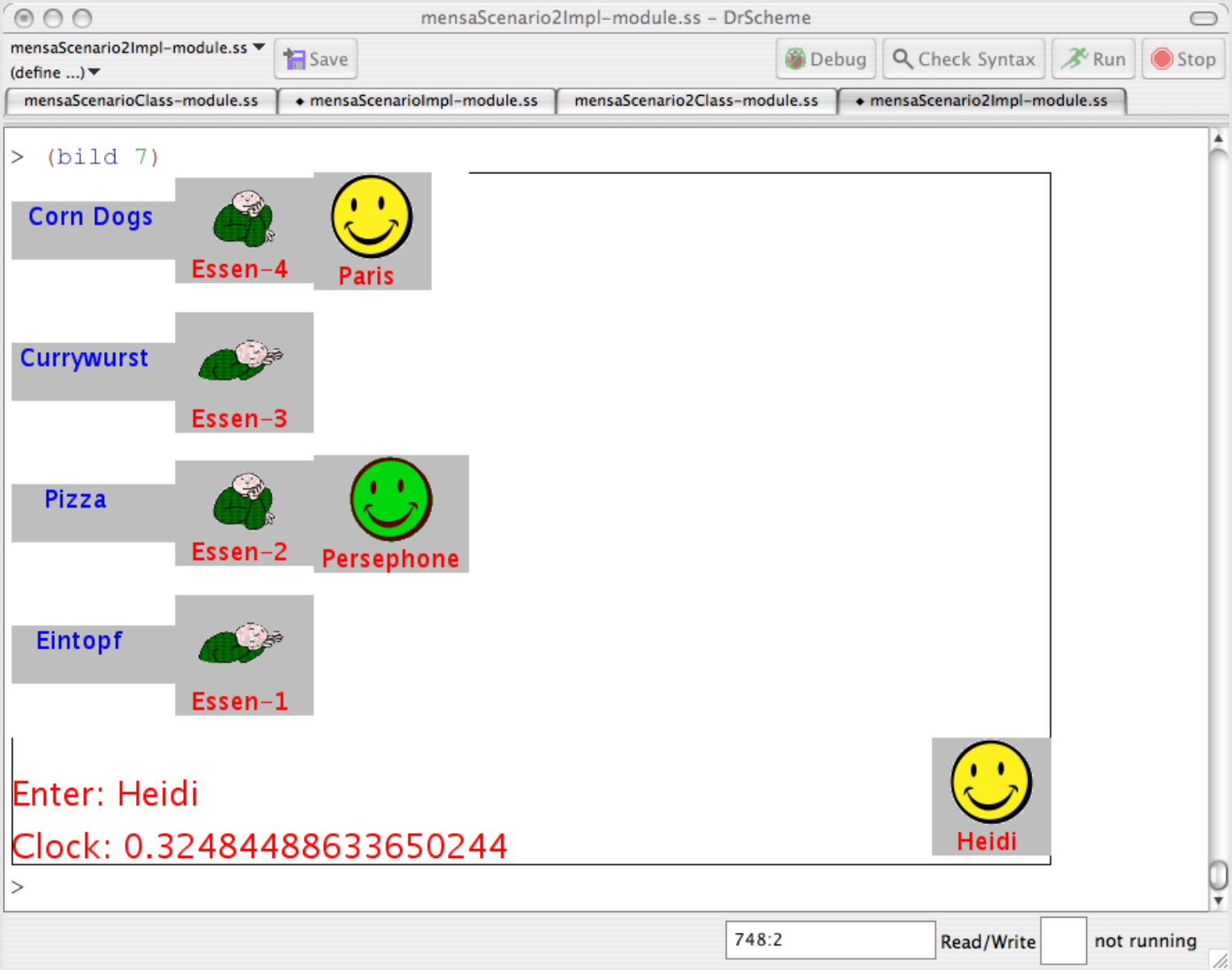


Ein Beispiellauf: Mensa-Szenario 2



Ein Beispiellauf: Mensa-Szenario 2

Ein Beispiellauf: Mensa-Szenario 2

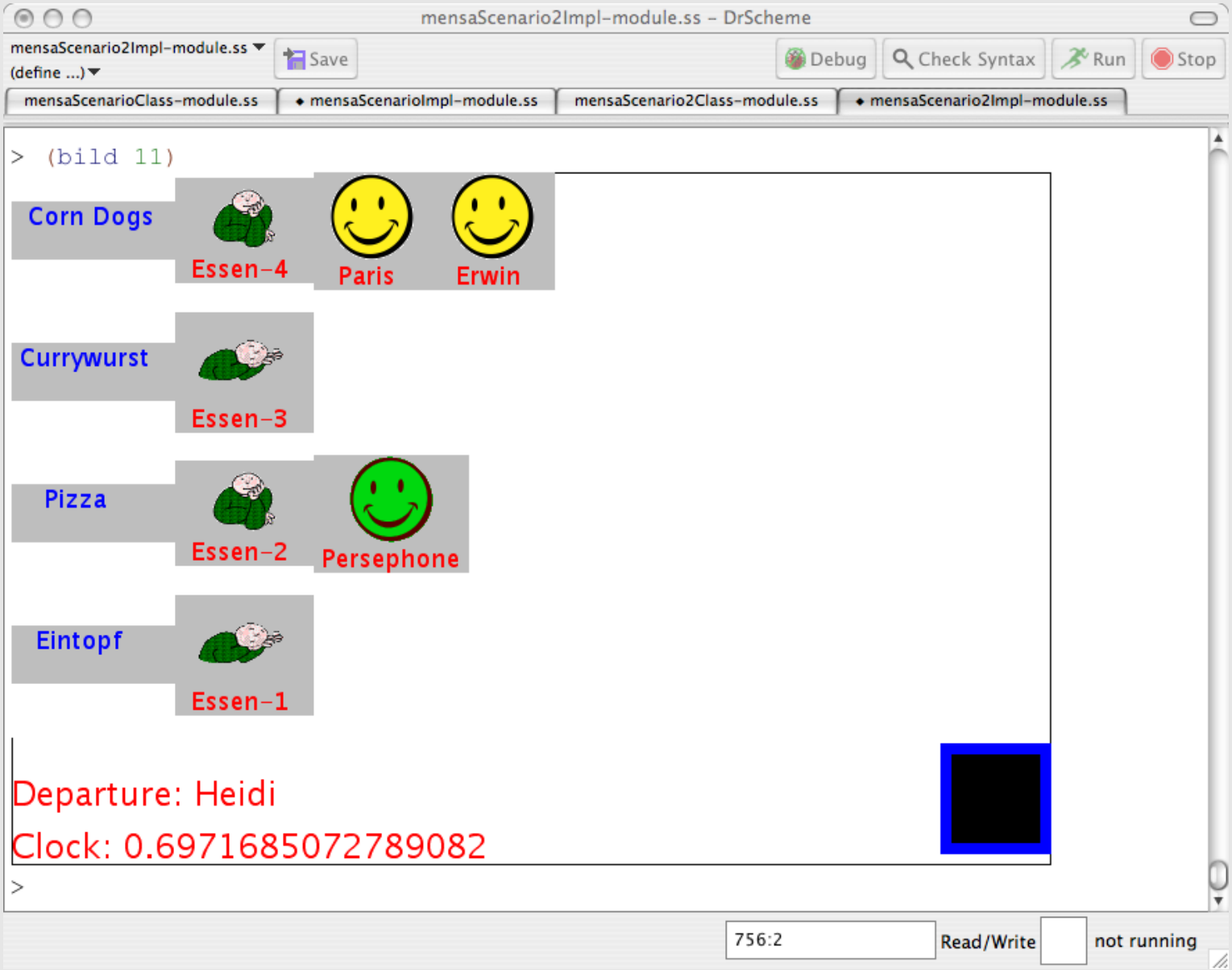


Ein Beispiellauf: Mensa-Szenario 2

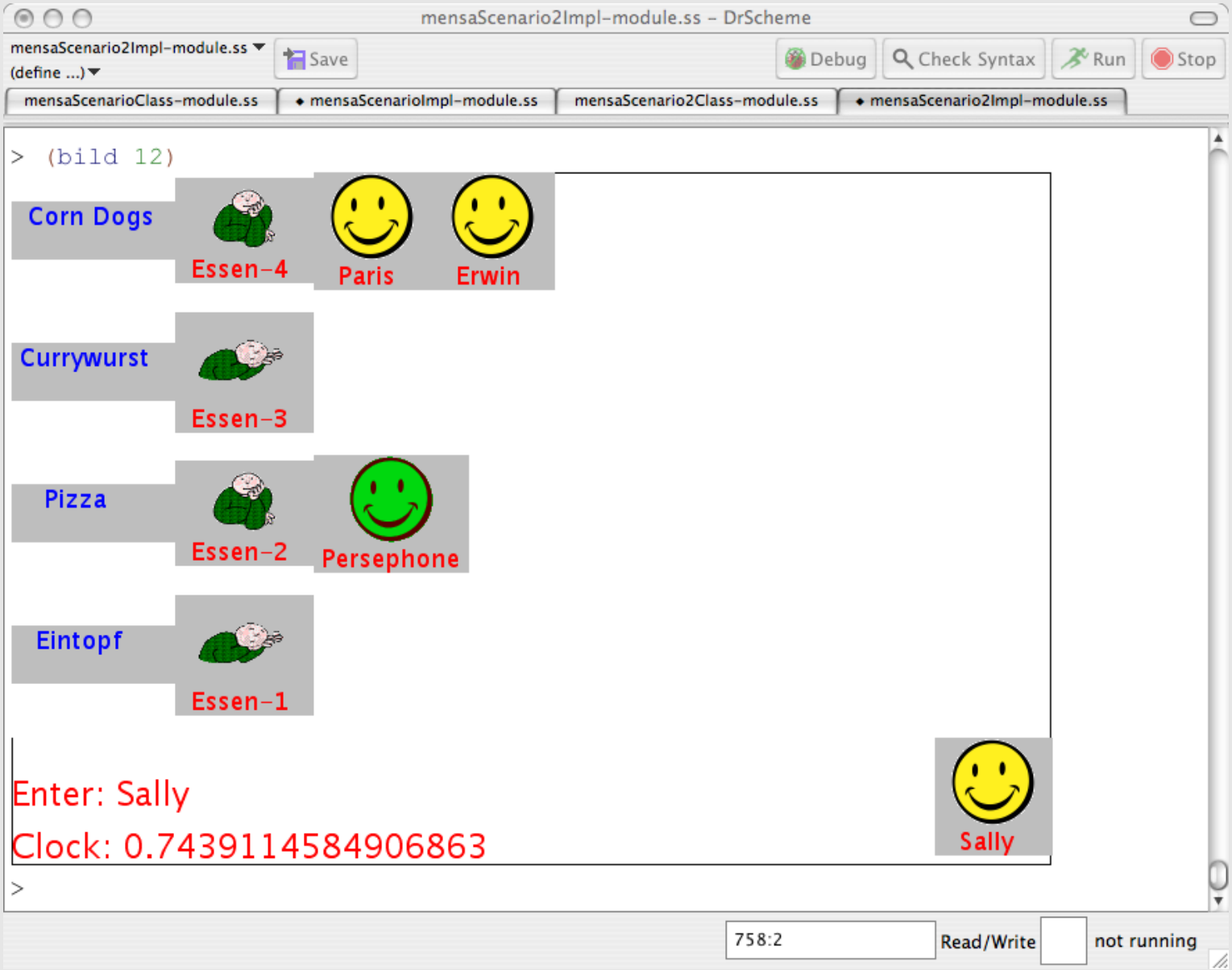
Ein Beispiellauf: Mensa-Szenario 2

Ein Beispiellauf: Mensa-Szenario 2

Ein Beispiellauf: Mensa-Szenario 2

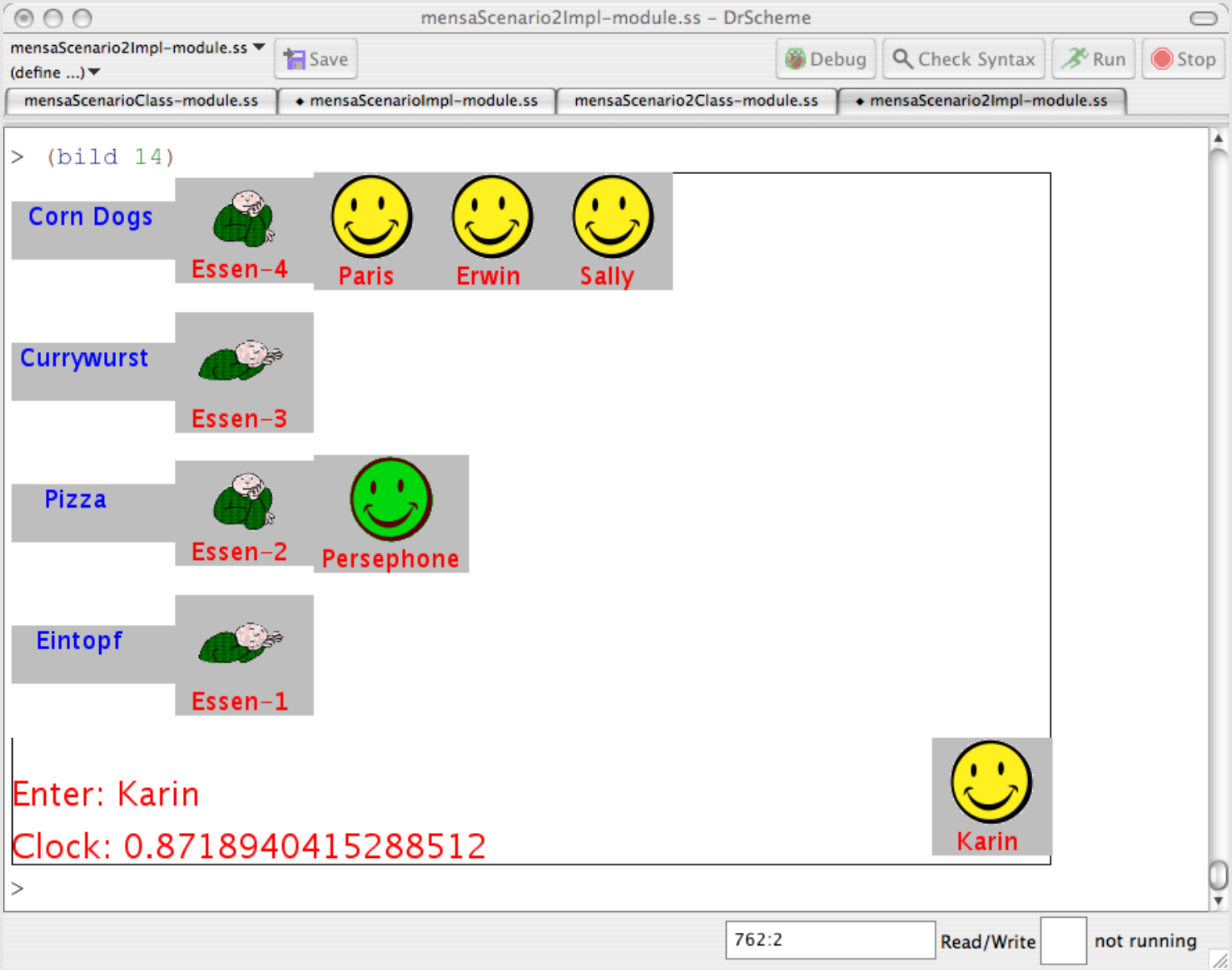


Ein Beispiellauf: Mensa-Szenario 2



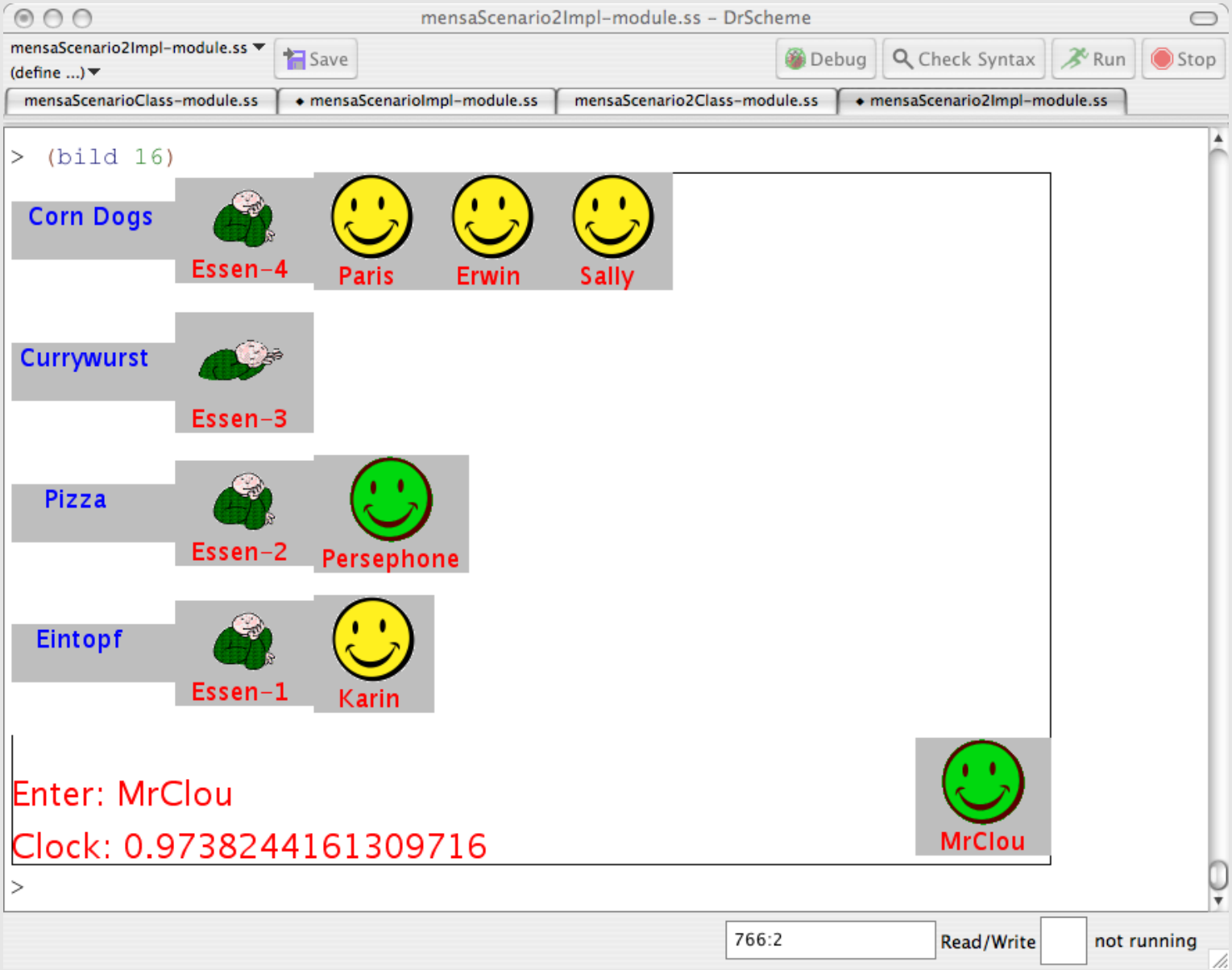
Ein Beispiellauf: Mensa-Szenario 2

Ein Beispiellauf: Mensa-Szenario 2

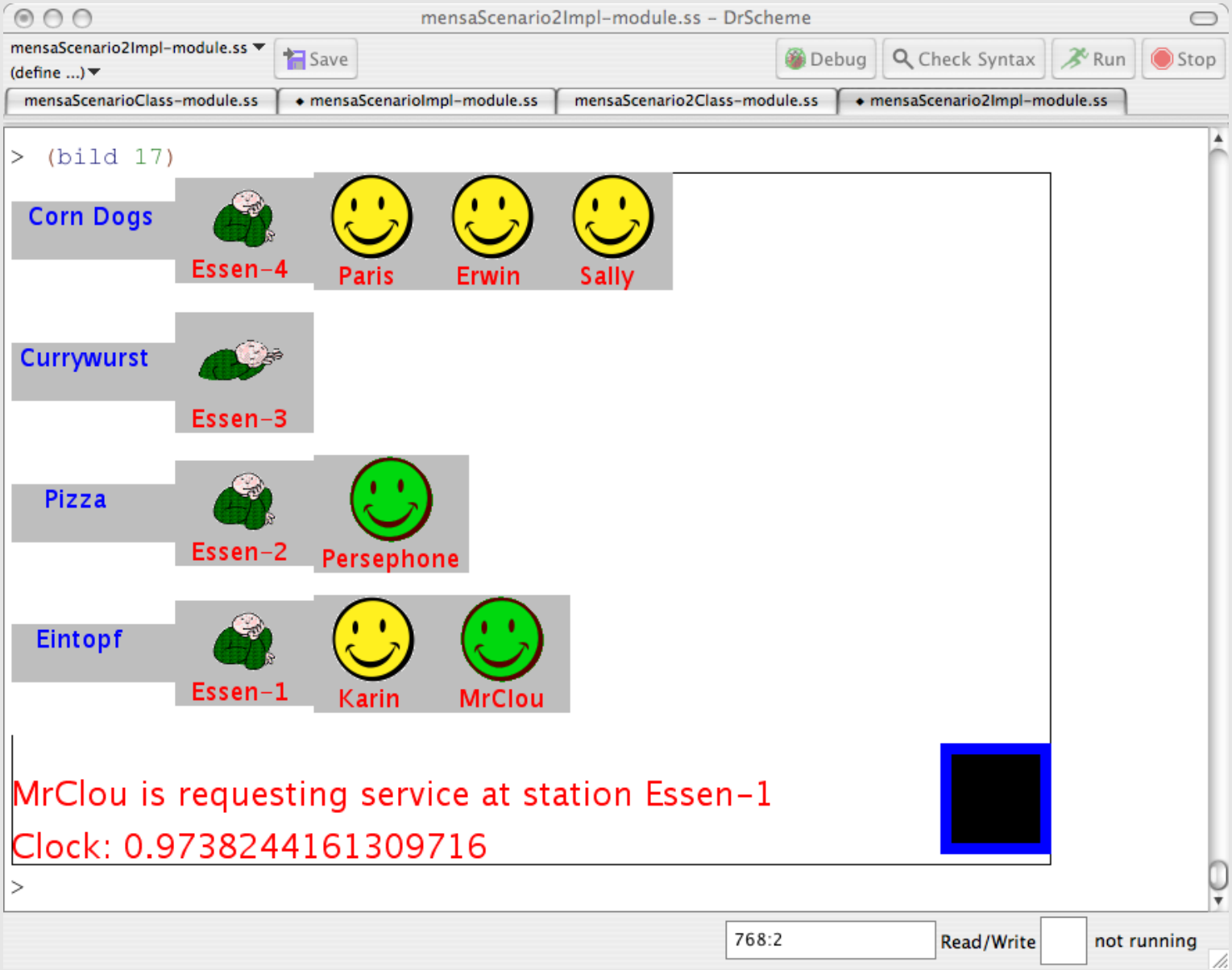


Ein Beispiellauf: Mensa-Szenario 2

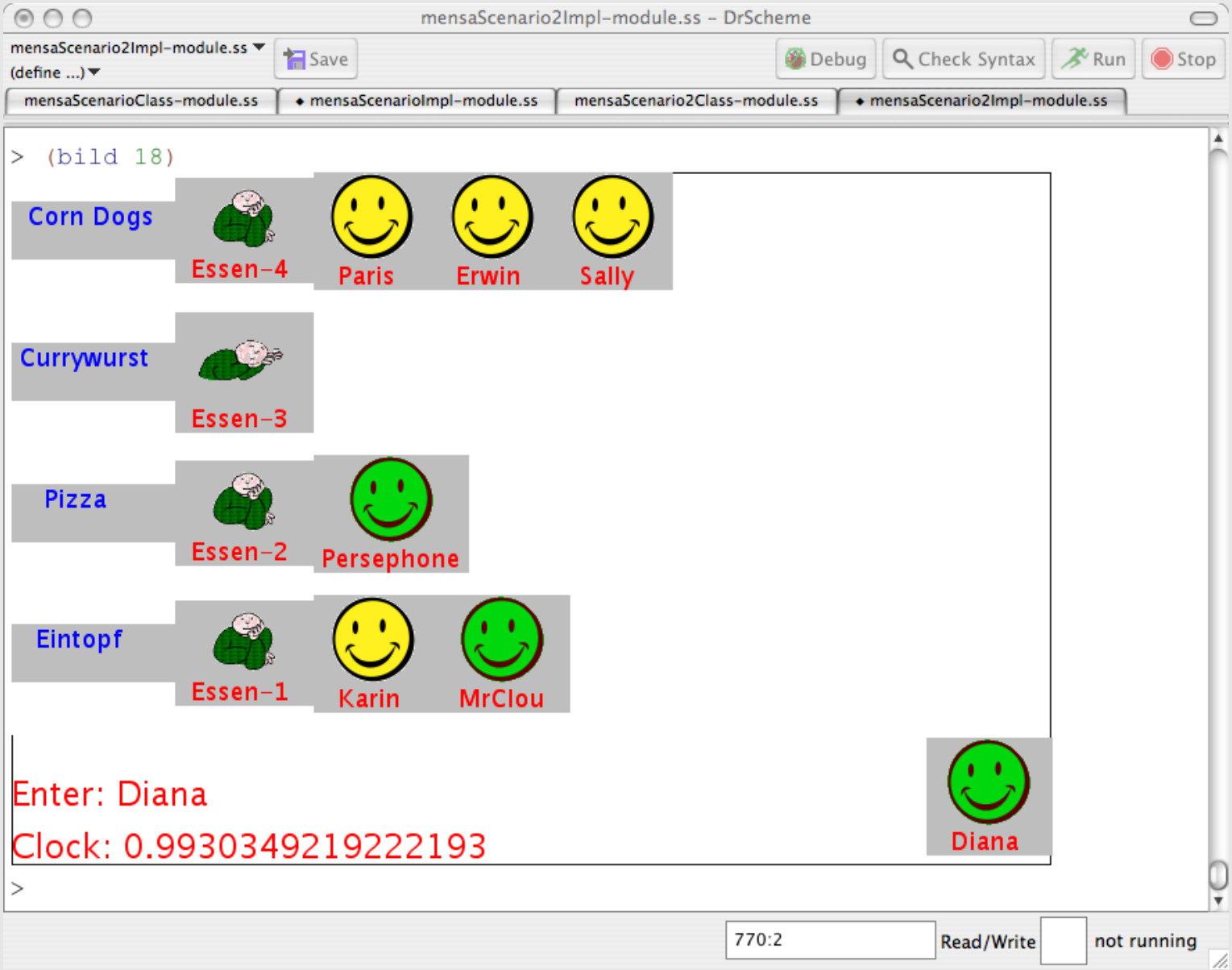
Ein Beispiellauf: Mensa-Szenario 2



Ein Beispiellauf: Mensa-Szenario 2

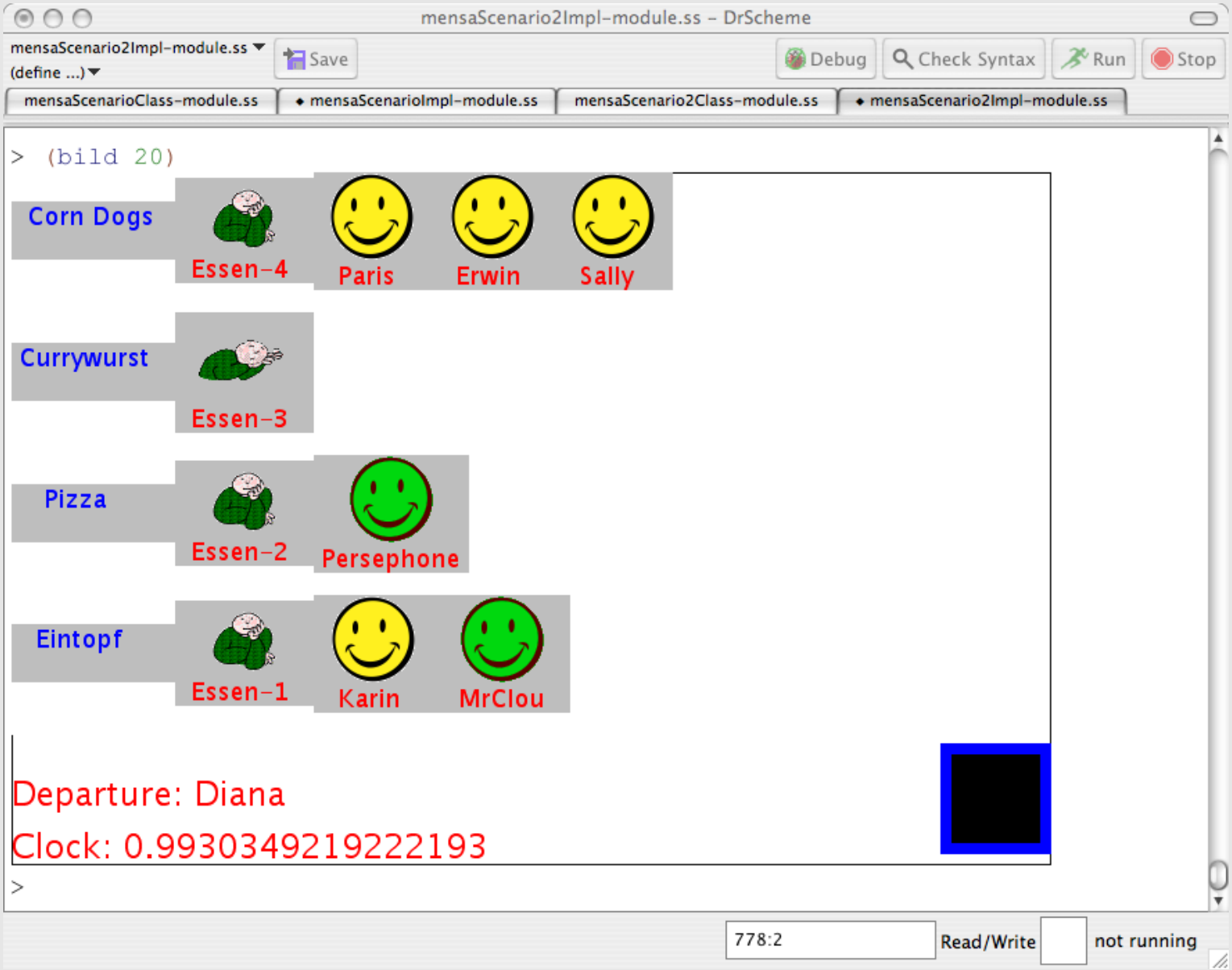


Ein Beispiellauf: Mensa-Szenario 2

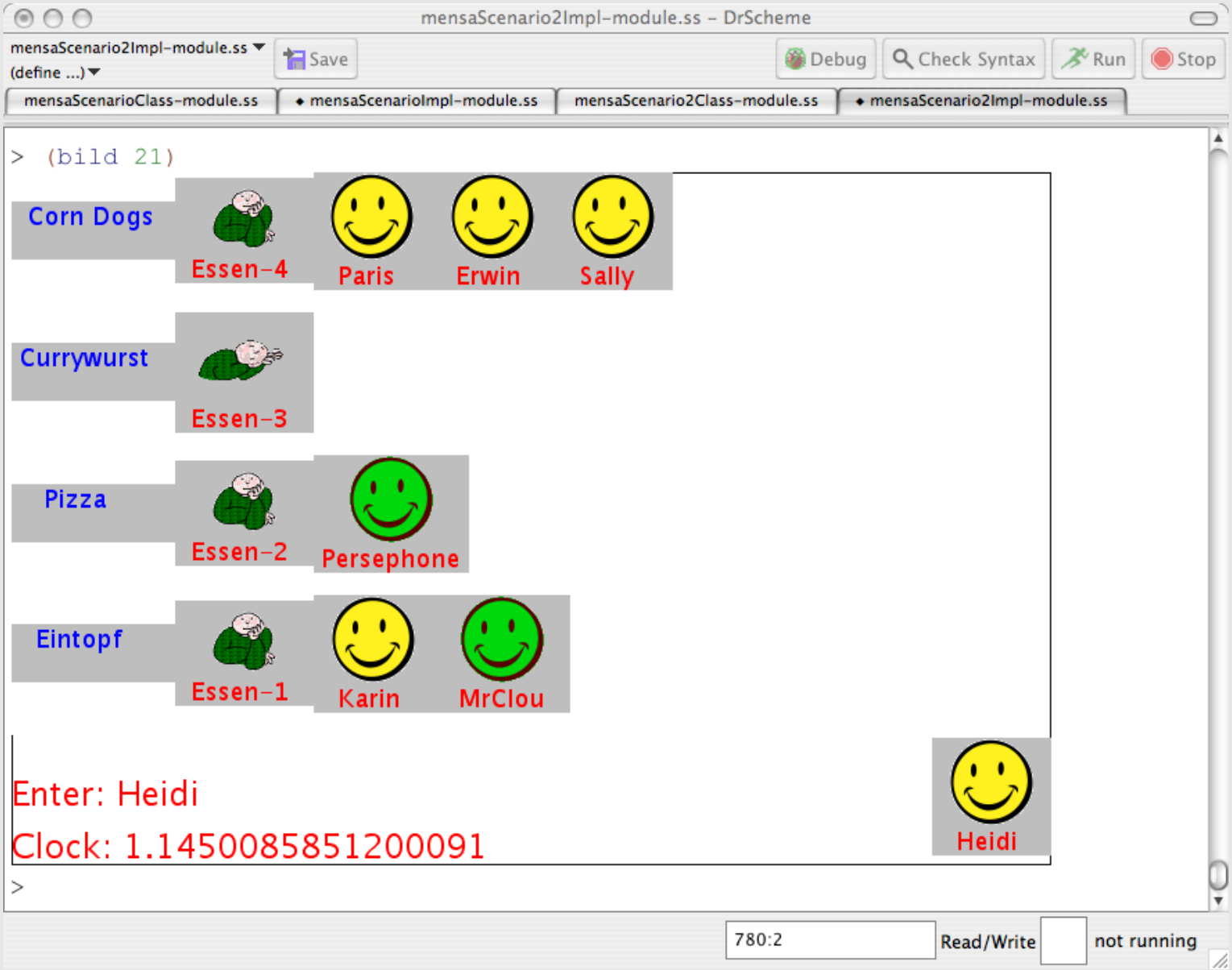


Ein Beispiellauf: Mensa-Szenario 2

Ein Beispiellauf: Mensa-Szenario 2

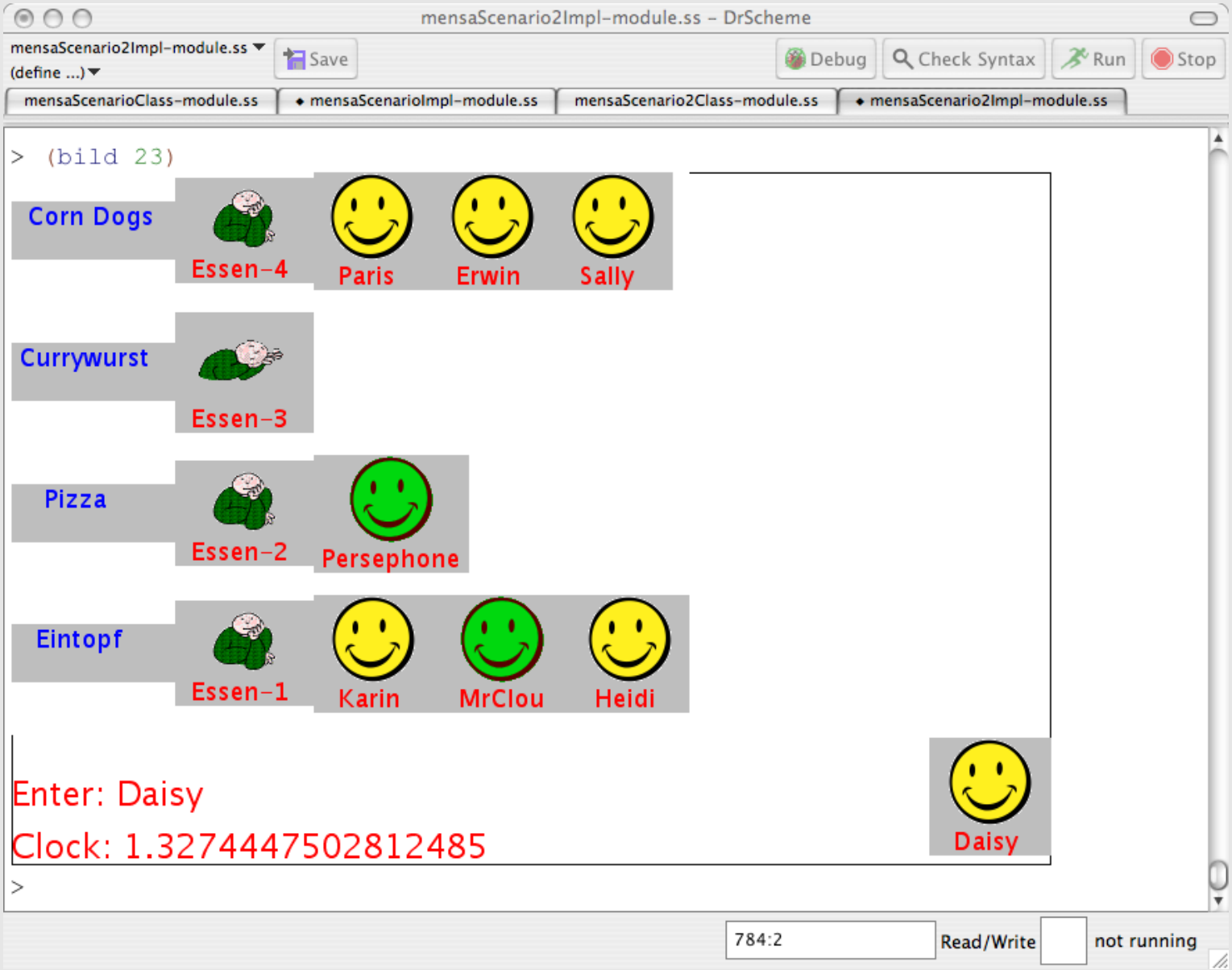


Ein Beispiellauf: Mensa-Szenario 2



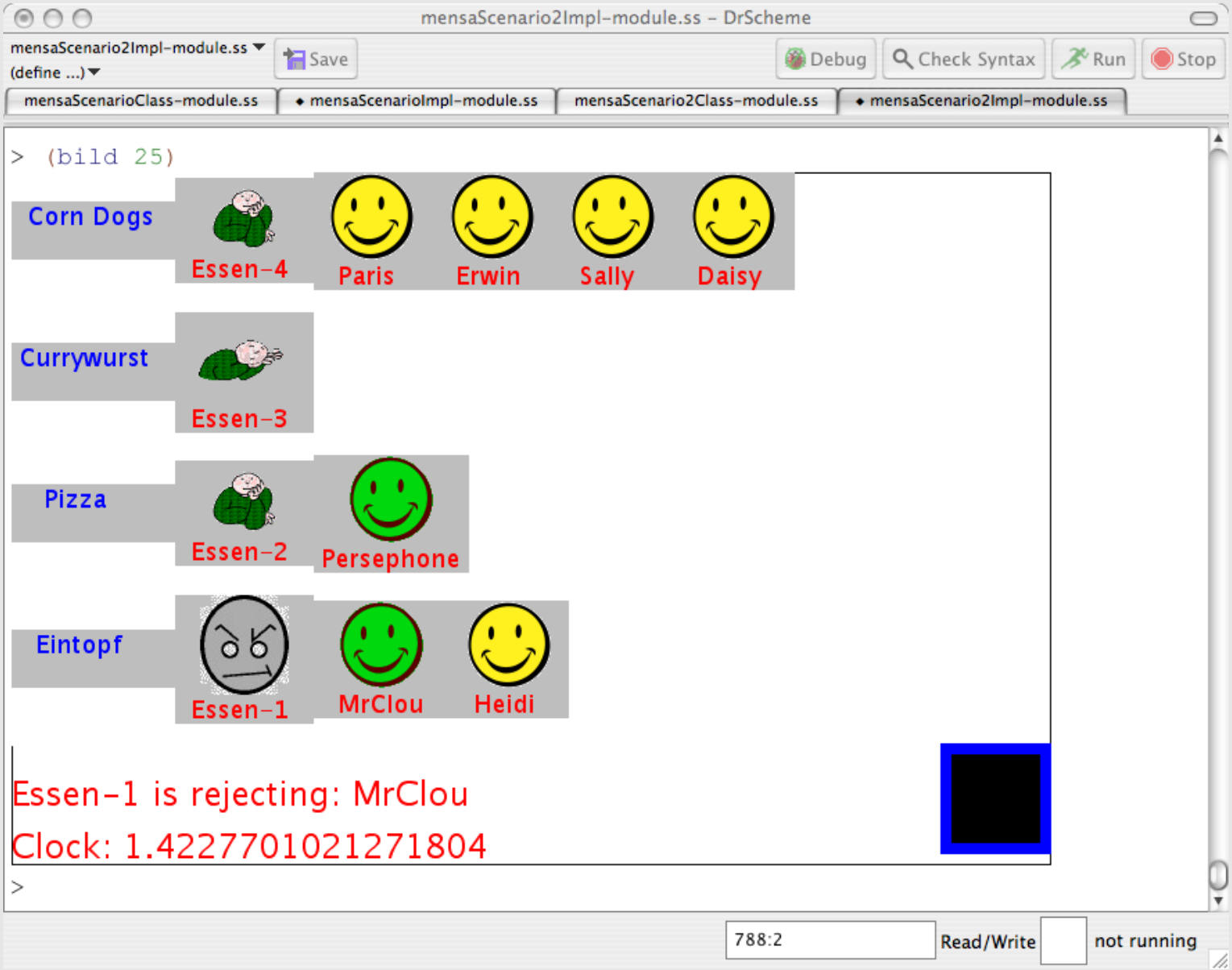
Ein Beispiellauf: Mensa-Szenario 2

Ein Beispiellauf: Mensa-Szenario 2

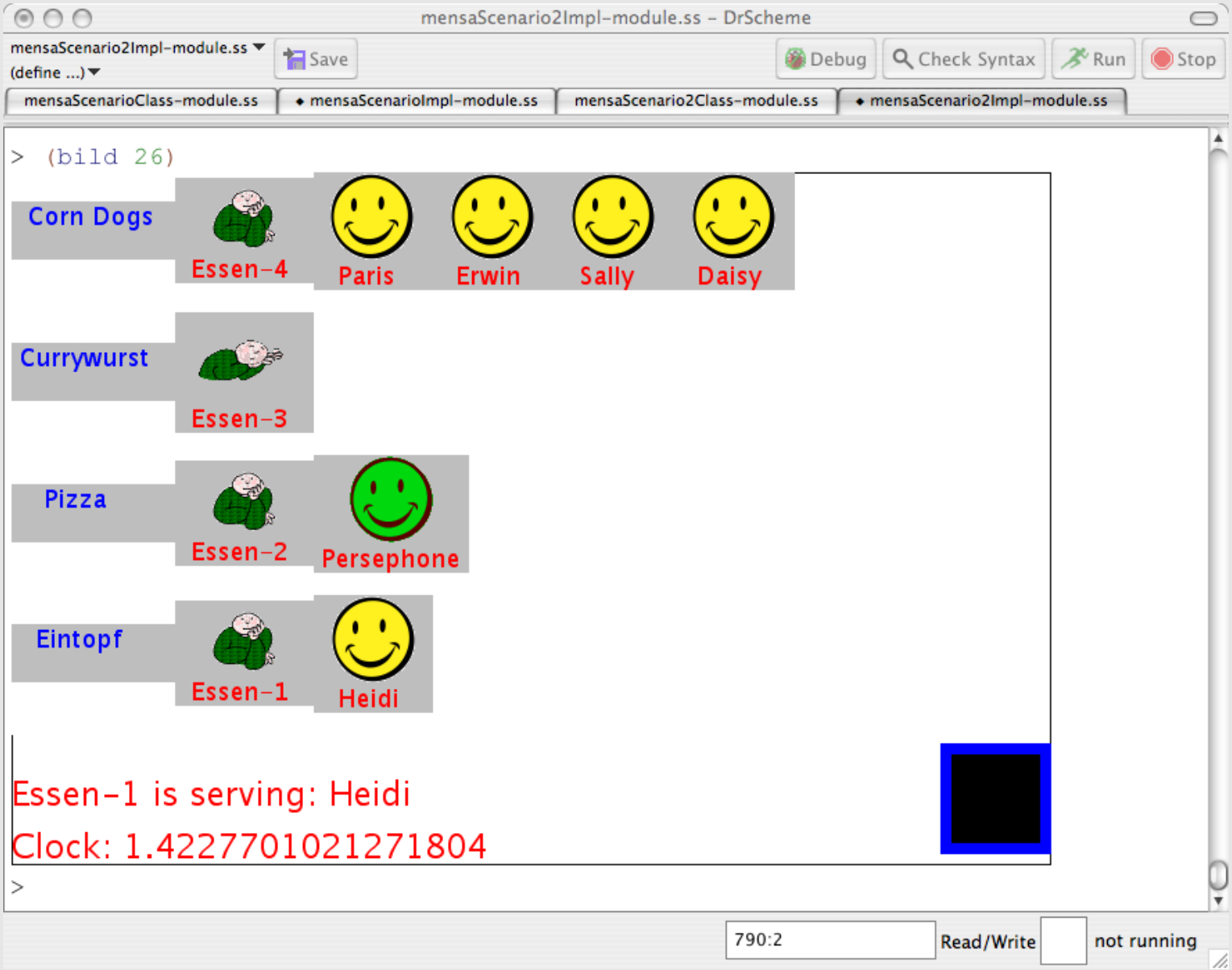


Ein Beispiellauf: Mensa-Szenario 2

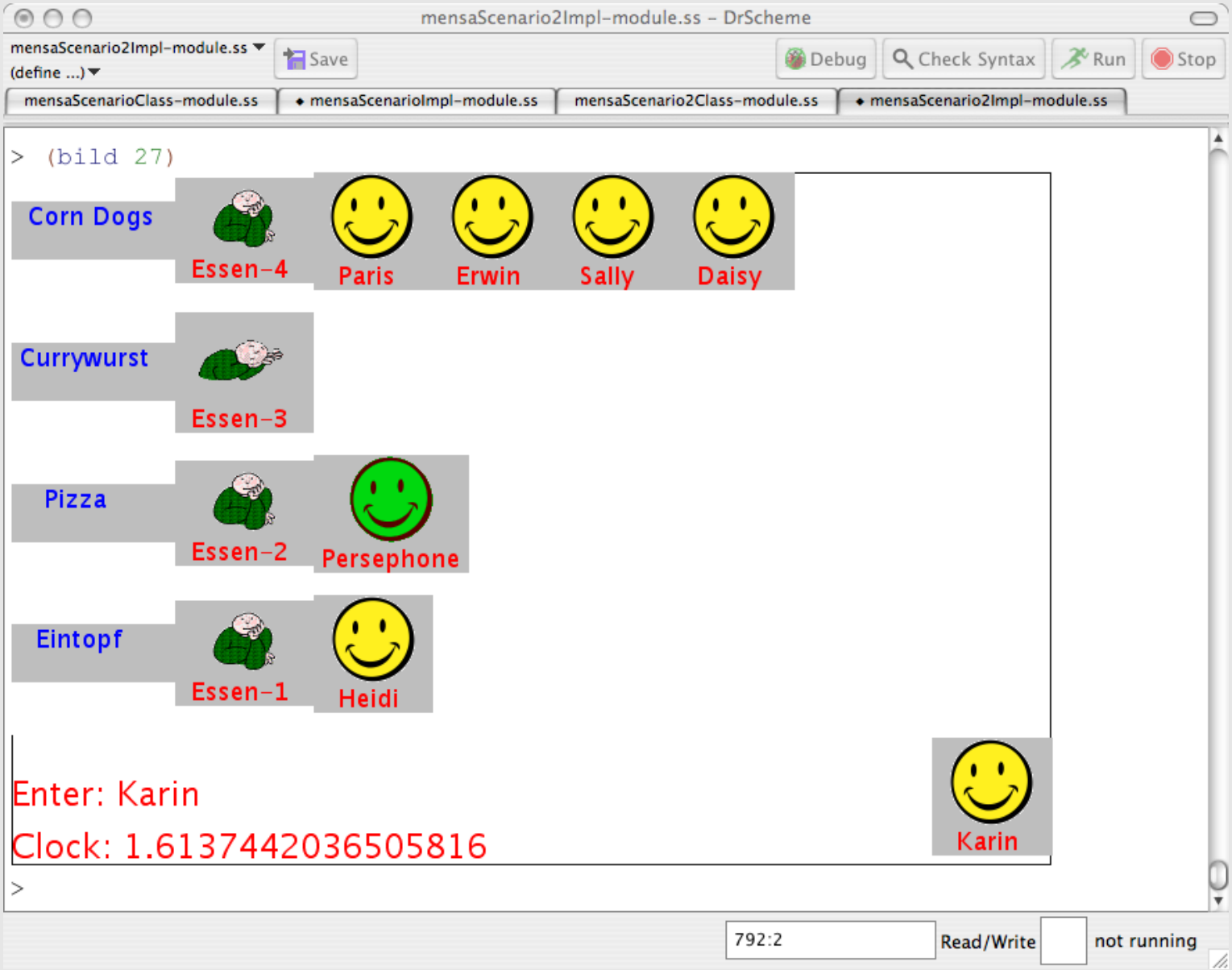
Ein Beispiellauf: Mensa-Szenario 2



Ein Beispiellauf: Mensa-Szenario 2

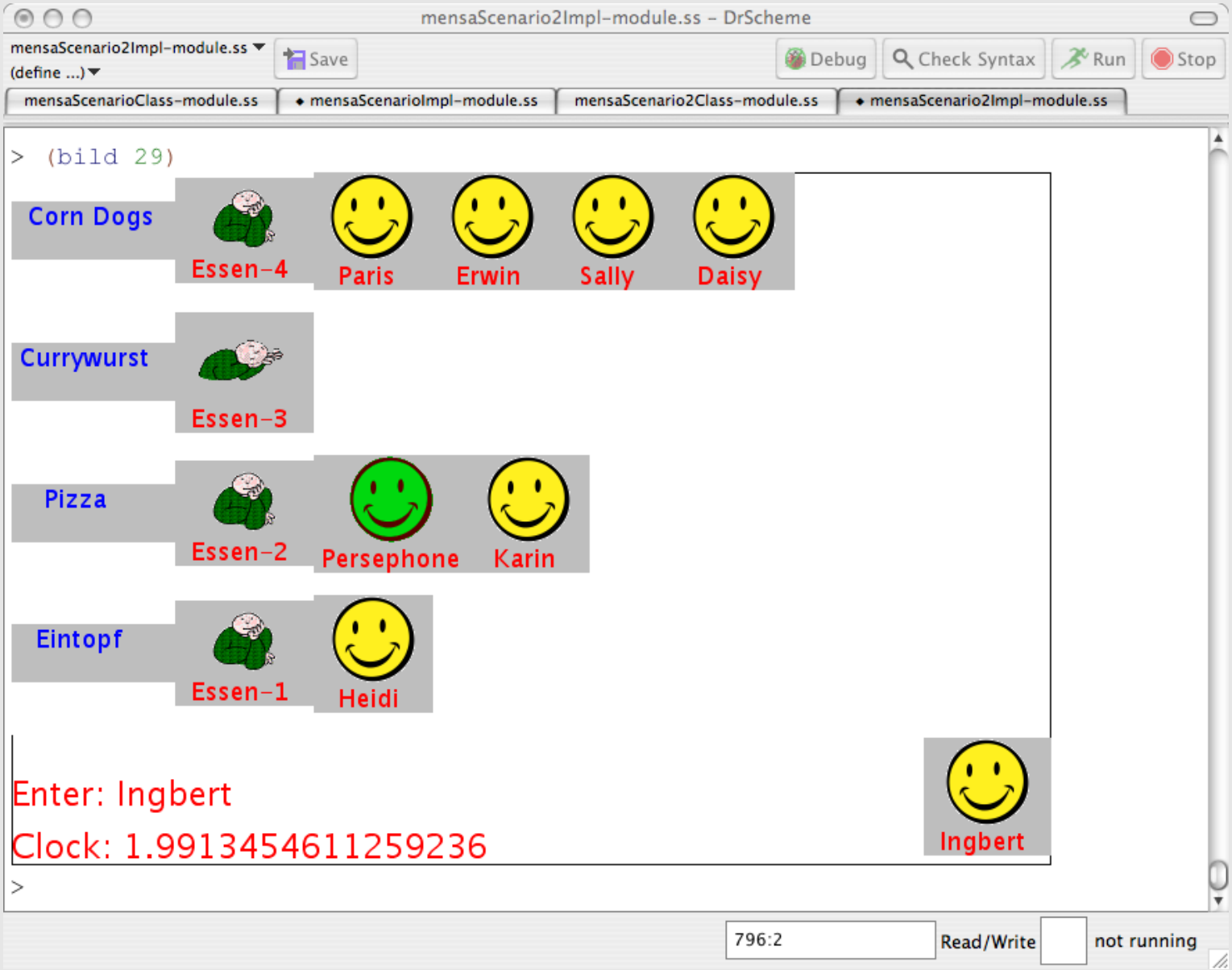


Ein Beispiellauf: Mensa-Szenario 2

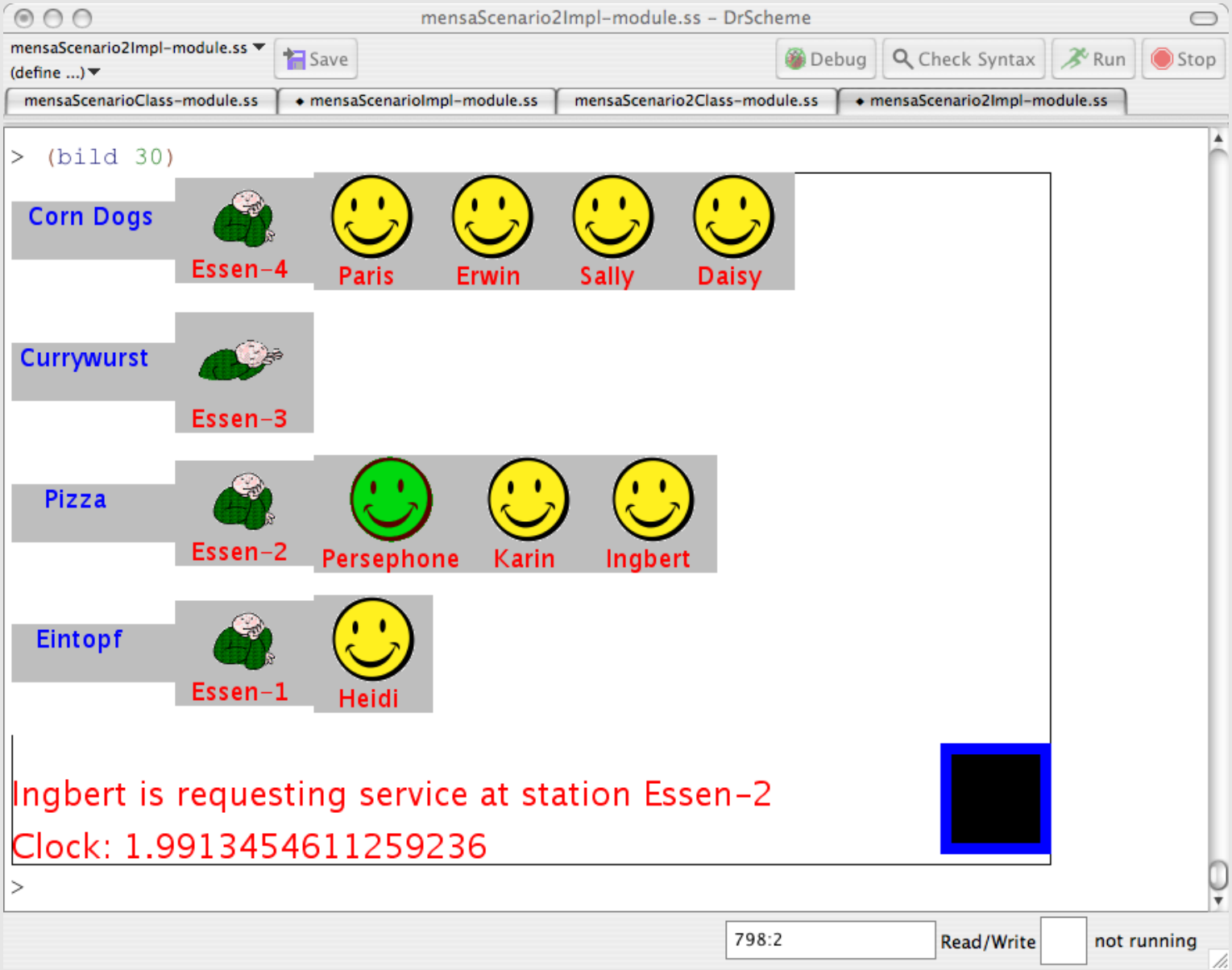


Ein Beispiellauf: Mensa-Szenario 2

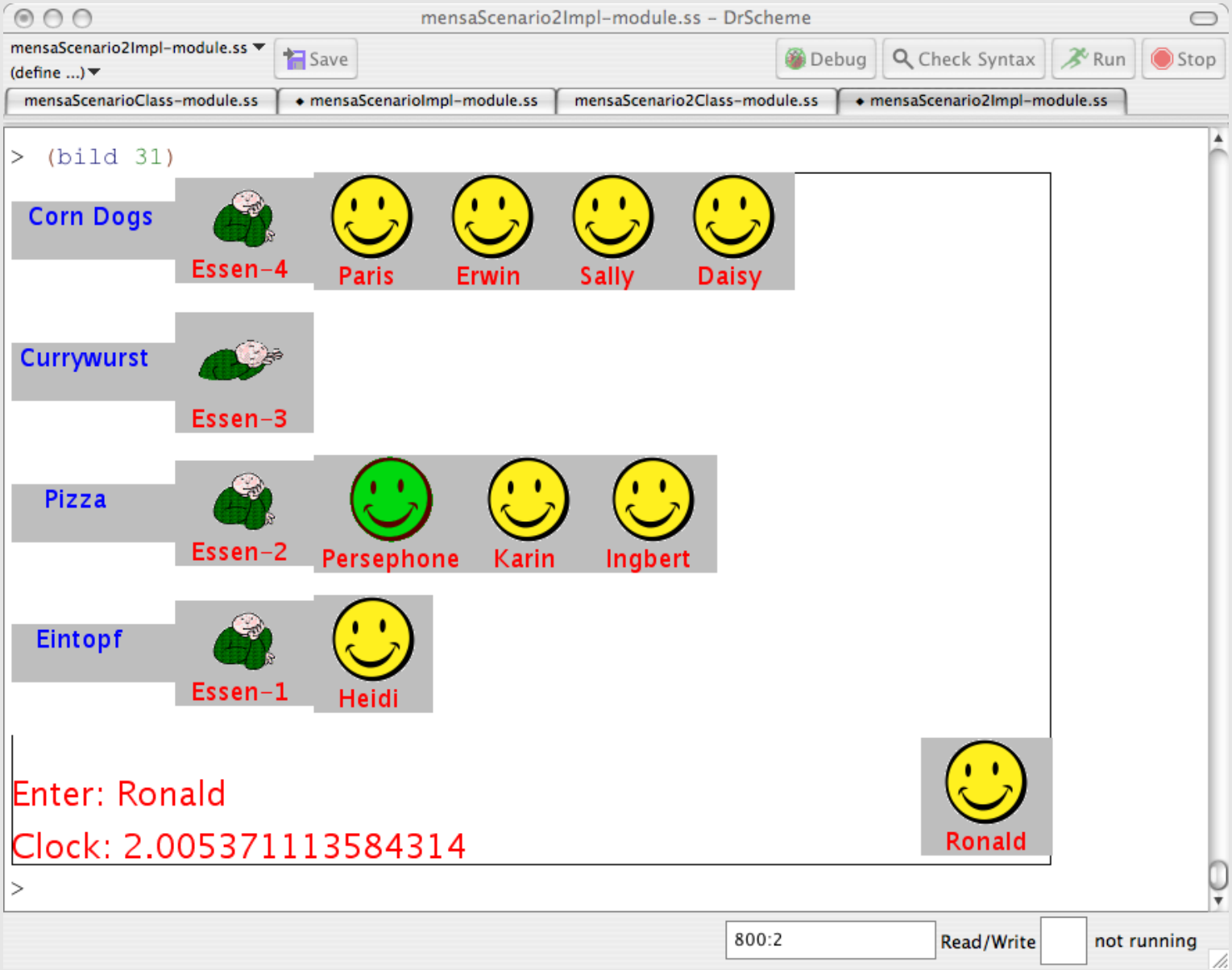
Ein Beispiellauf: Mensa-Szenario 2



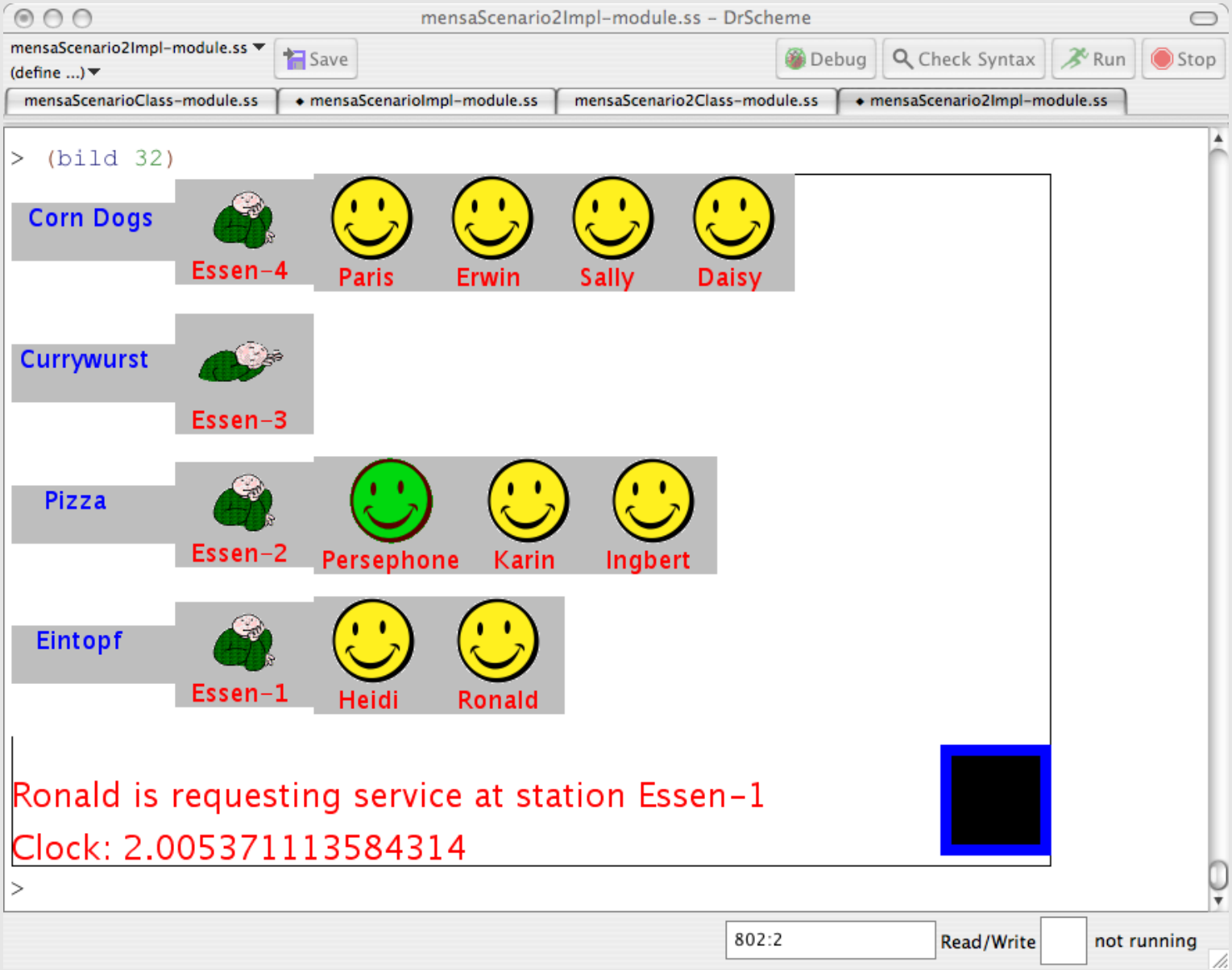
Ein Beispiellauf: Mensa-Szenario 2



Ein Beispiellauf: Mensa-Szenario 2

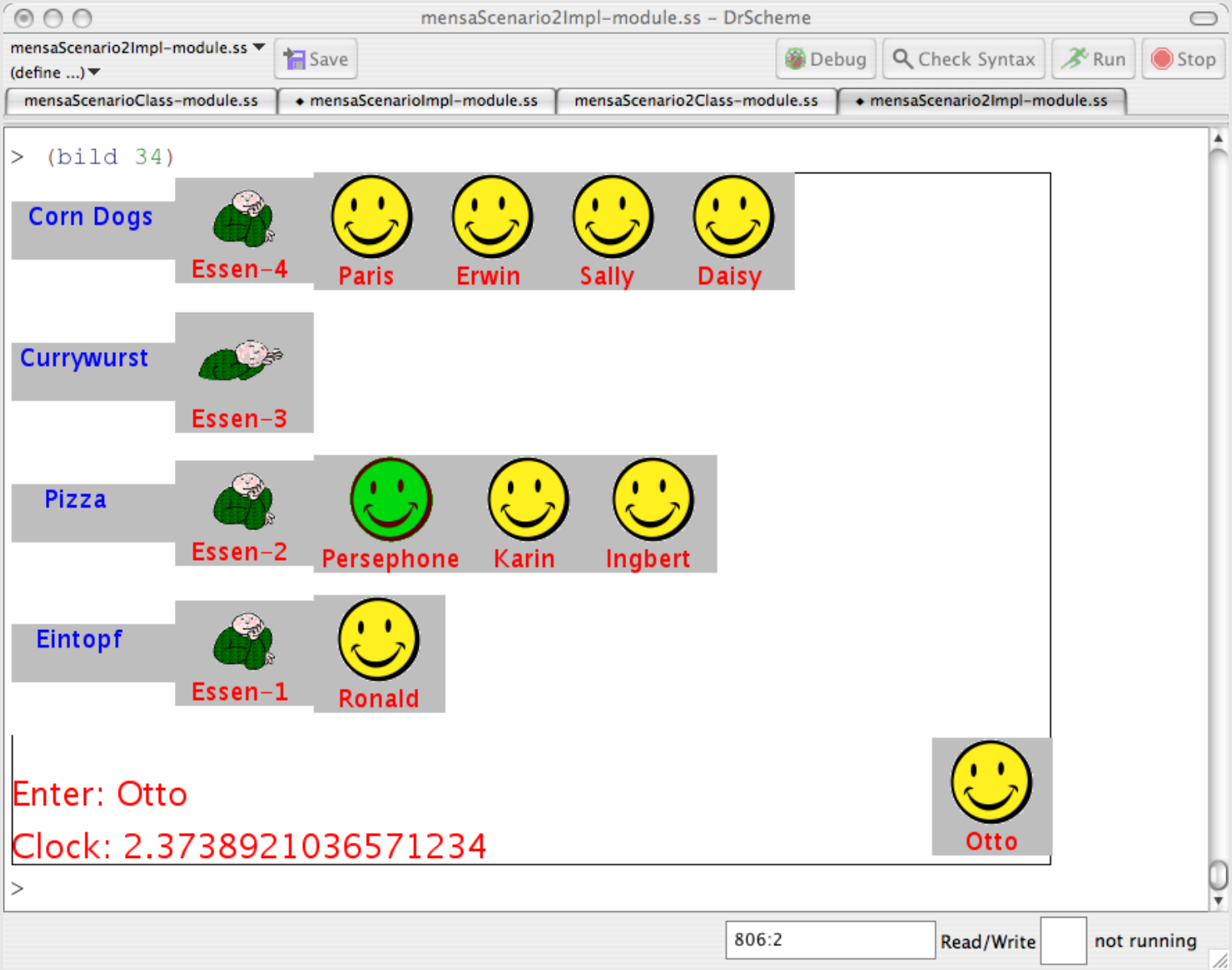


Ein Beispiellauf: Mensa-Szenario 2



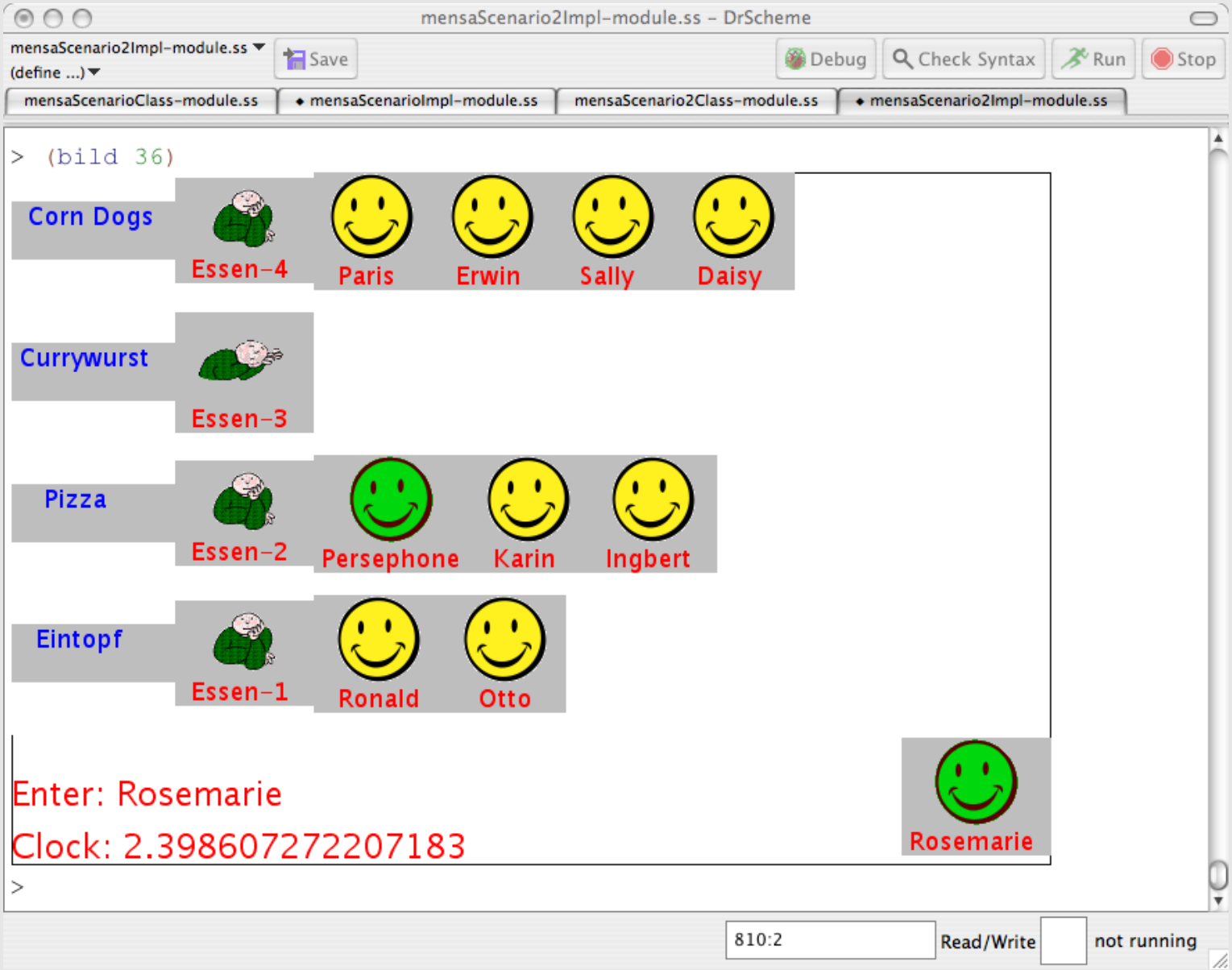
Ein Beispiellauf: Mensa-Szenario 2

Ein Beispiellauf: Mensa-Szenario 2



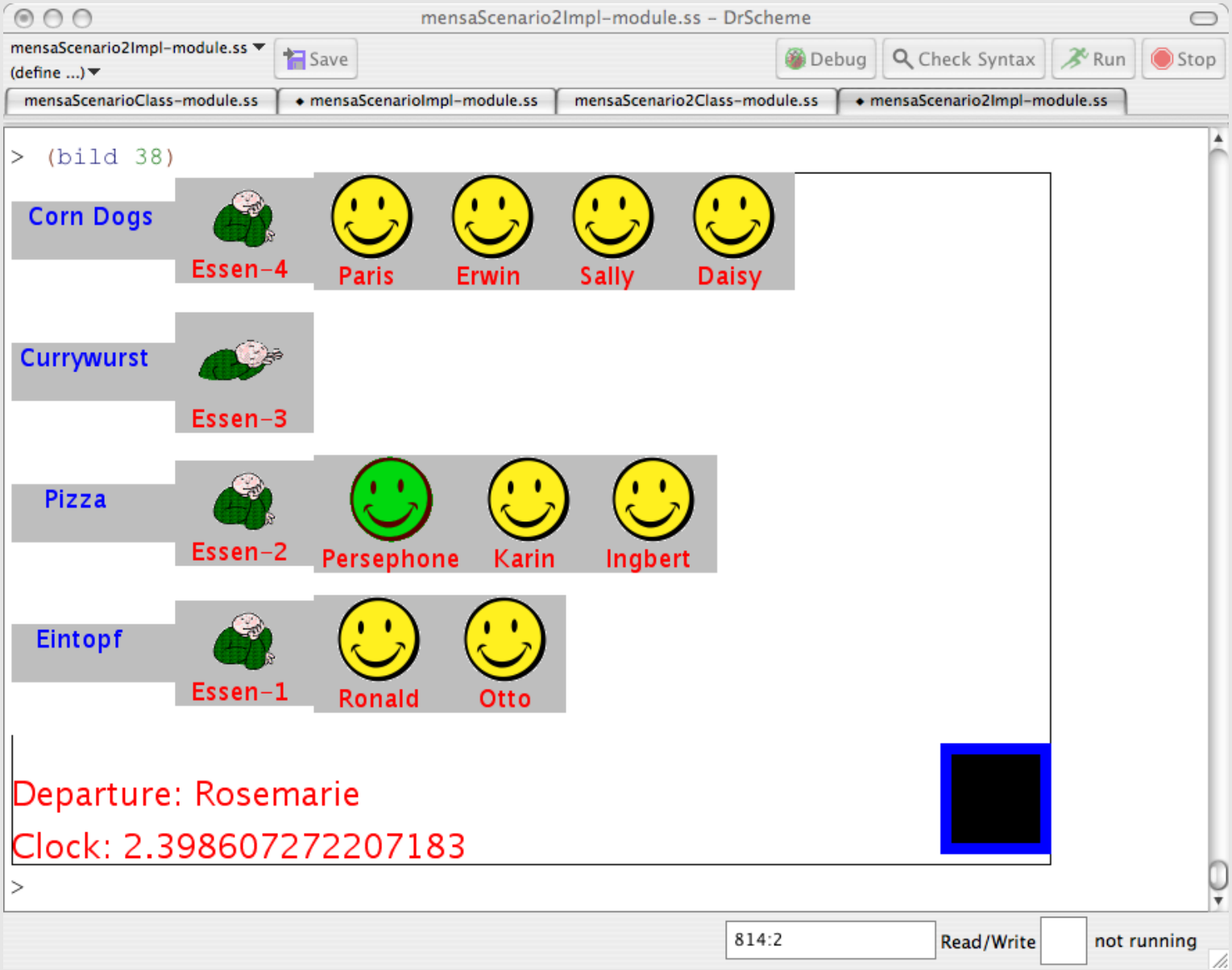
Ein Beispiellauf: Mensa-Szenario 2

Ein Beispiellauf: Mensa-Szenario 2

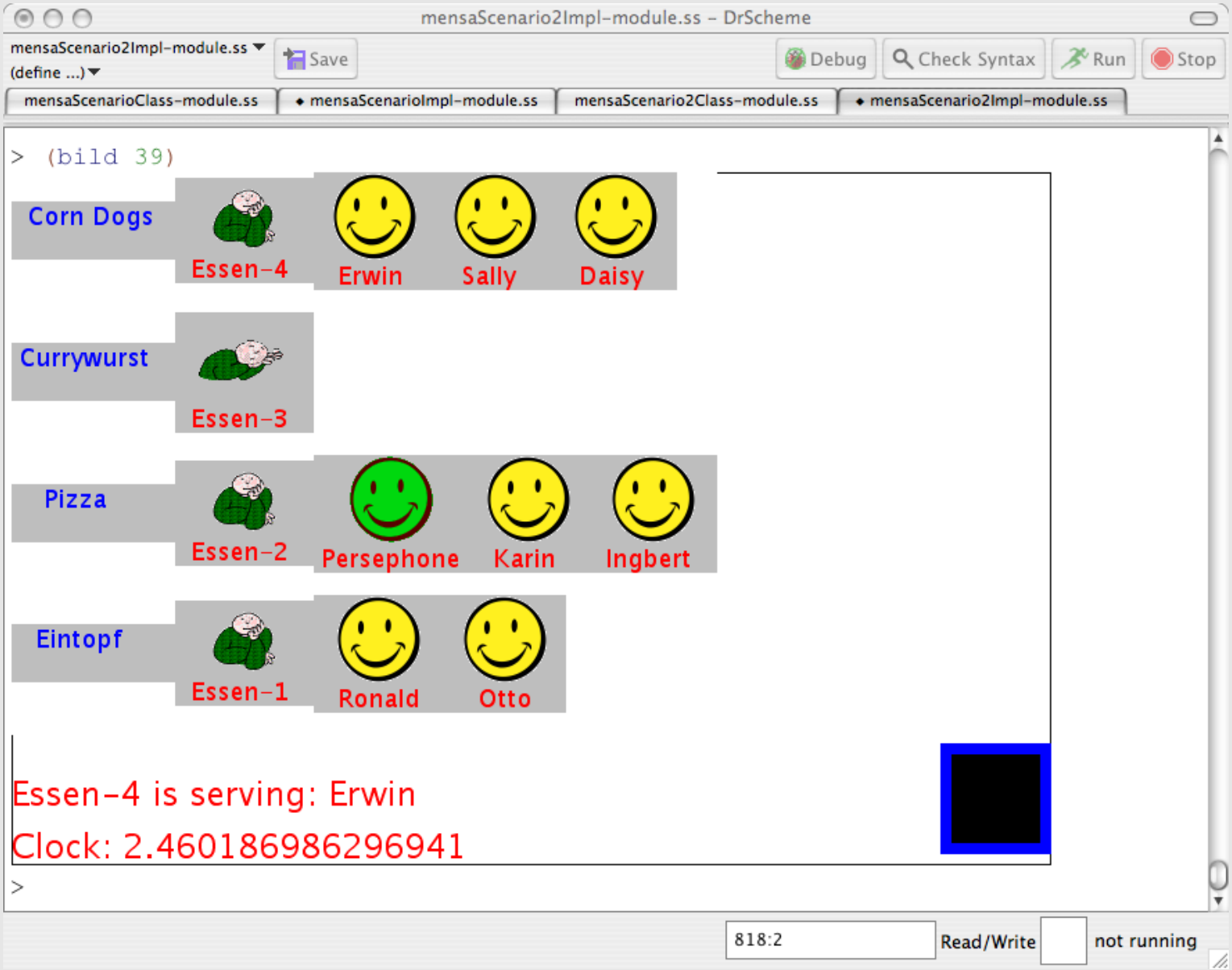


Ein Beispiellauf: Mensa-Szenario 2

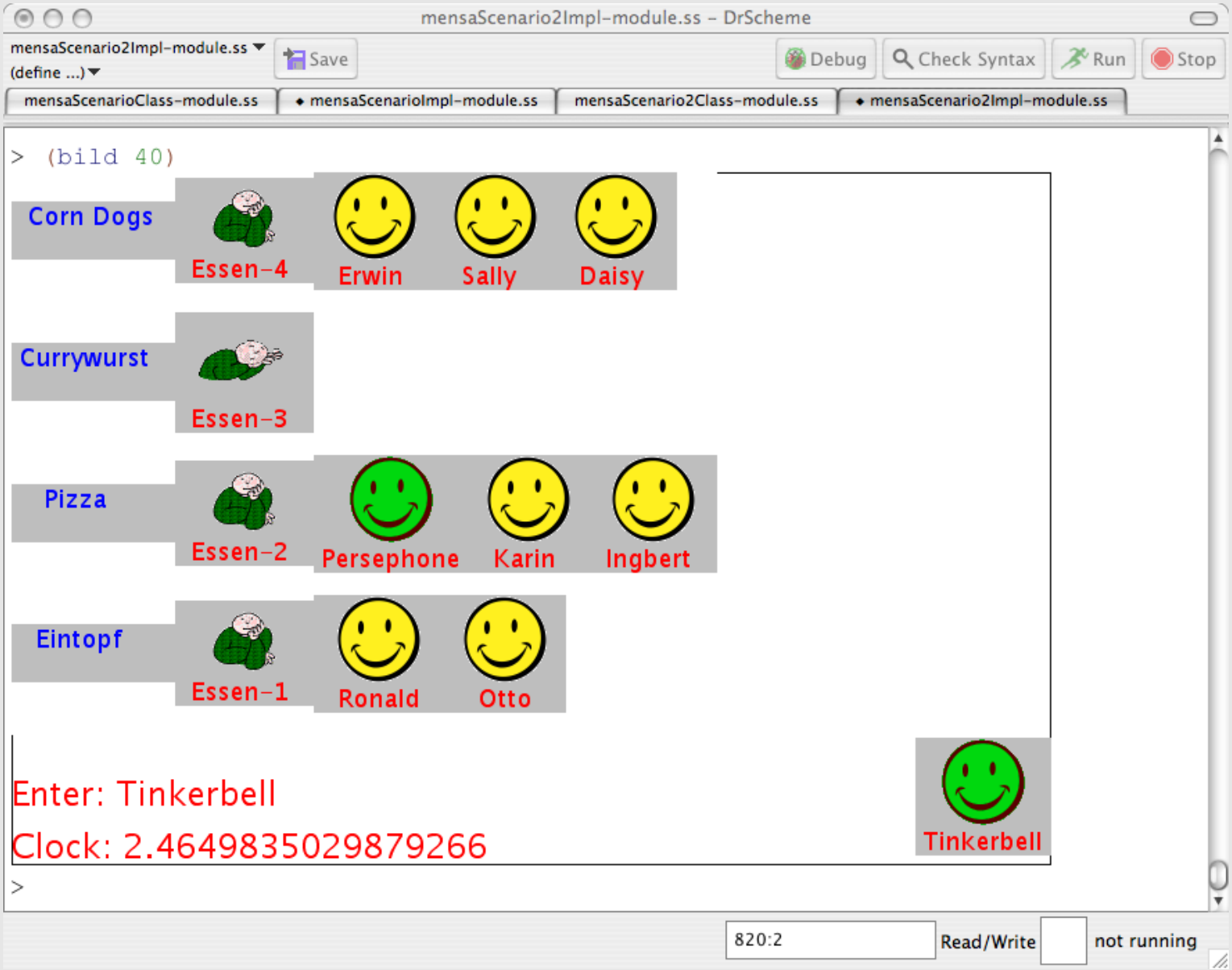
Ein Beispiellauf: Mensa-Szenario 2



Ein Beispiellauf: Mensa-Szenario 2



Ein Beispiellauf: Mensa-Szenario 2



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.

initalize :after : Anwendbar, wenn bei der Initalisierung 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.

Objektorientierte Verarbeitungsmodelle

Objektorientierte Verarbeitungsmodelle

Beispielsweise 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)
```

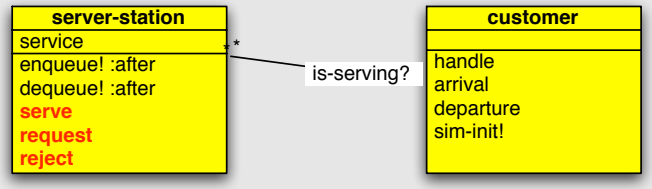
Generische Funktionen versus Nachrichtenmetapher

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 einzigen 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.

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 (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>
>
```

Objektorientierte Verarbeitungsmodelle

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.

Backtracking mittels Delegation

Beispiel (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)))
```

Anforderungen an eine Programmiersprache zur objektorientierten Programmierung

- Zustände
- Hybride Datenstrukturen, Typprädikate
- Polymorphie
- Dynamisches Binden von Funktionen (Methoden)
- Datenkapselung

- Um Zustände zu realisieren, müssen wir funktionale Programmiersprachen um imperative 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.

Teil XI

Metaprogrammierung und Programmierstile

Teil XI

Metaprogrammierung und Programmierstile

Metaprogrammierung und Programmierstile

Macros

Definition (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

Ein einfaches Beispiel: myif

Beispiel

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))
```

Eine while-Schleife

while soll die folgende Syntax haben:

```
(while <test> <expr1> ... <exprn>)
```

```
(letrec
      ((loop
        (lambda ()
          (when <test> <expr1> ... <exprn>
            (loop)))))
      (loop)))
```

```
(defmacro (my-while test &rest body)
  "Repeat body while test is true."
  `(letrec
      ((loop
        (lambda ()
          (when ,test ,@body (loop)))))
      (loop)))
```

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
```

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.

Rechenzeitersparnis durch Macros

Beispiel (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
```

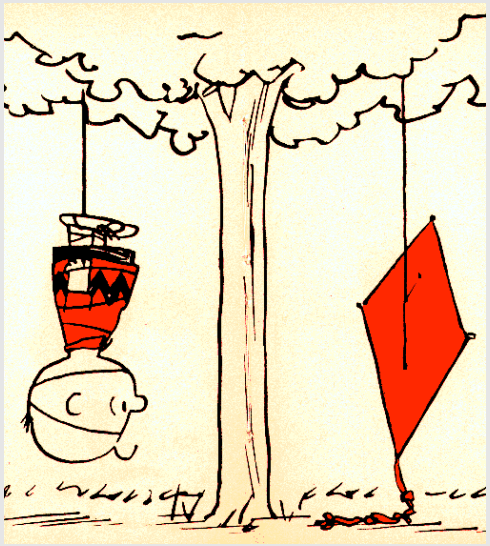
Ein Laufzeitvergleich

```
(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.

Fallstricke beim Macroentwurf



- Unbeabsichtigte Mehrfachauswertung (Rechenaufwand, Seiteneffekte)
- Unbeabsichtigte falsche Referenzen

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 achtmal 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
```

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
```

Macros und lokale Variable

```
>(defmacro(ntimes n &rest body) ;repeat body n
     `(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)))
```

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
```

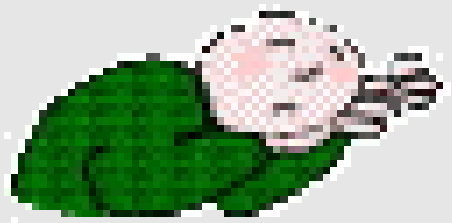
Teil XI

Metaprogrammierung und Programmierstile

Relationale Programmierung

▸ Weiter mit Prolog

Verzögerte Auswertung

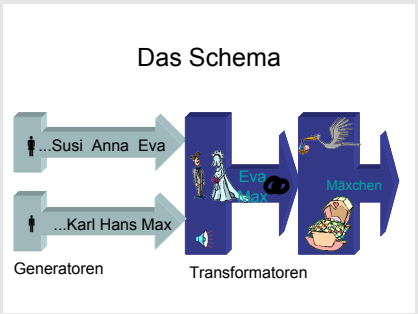


Vorteile:

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).

Stromorientierten Programmierung



Stromorientierten Programmierung

Bei der stromorientierten Programmierung liegt der Akzent auf dem Fluß der Daten: Wir modellieren die Programme als ein System von

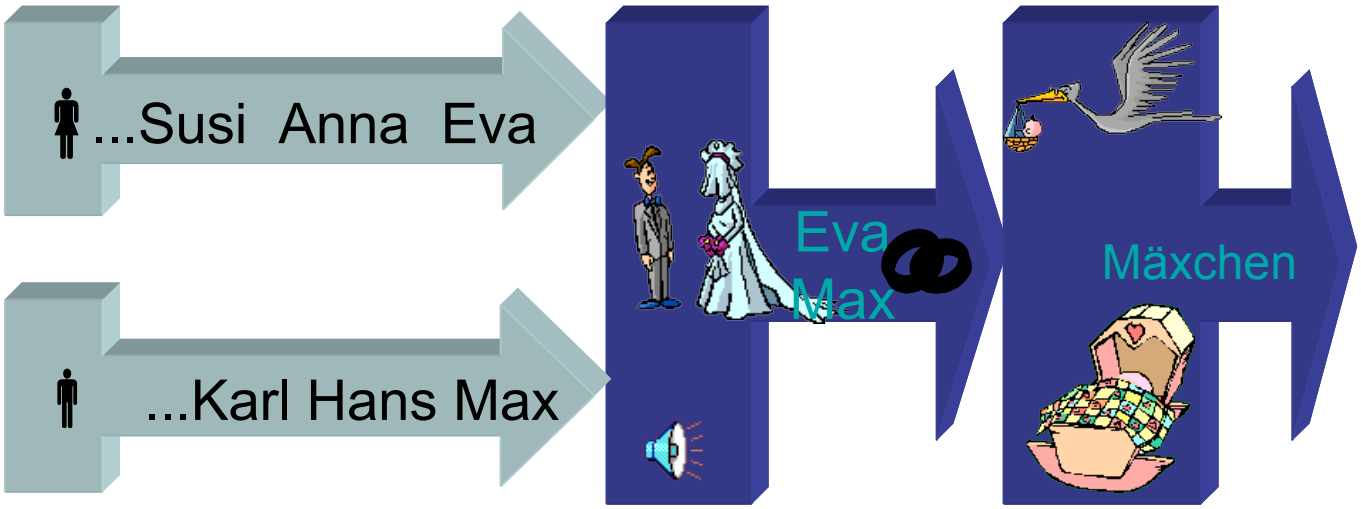
Generatoren, die Ströme von Werten produzieren, und

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


...Susi Anna Eva
...Karl Hans Max
Eva
Max
Mäxchen
Generatoren
Transformatoren

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 (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 (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 f
                (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))))))
```

Die Grundoperationen für verzögerte Auswertung

In Racket gibt es die special form operators delay und force:

delay: Lege 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 (Implemetation 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.

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: $\{2, 3, 5, 7, \ldots\}$

Stromorientierte Lösung

Beispiel (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.

```
()              (2  3  4  5  6  7  8  9  10  11  12  13  14  15  .
(2)             (3  5  7  9  11  13  15  17  19  23  25  27  ...
(2  3)          (5  7  11  13  17  19  23  25  29  ...)
(2  3  5)       (7  11  13  17  19  23  29  ..)
....                    ....
```

```
(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)
```

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)
```

```
(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)))))))
```

Standardverfahren zur Effizienzsteigerung von Algorithmen

30 Metaprogrammierung und Spracherweiterung

31 Stromorientierte Programmierung

32 Programmieren mit Zuständen

- Caching und Memoization
- Empfehlungen
- Satire: Man mordet nicht nach Sprungbefehl

Caching und Memoization

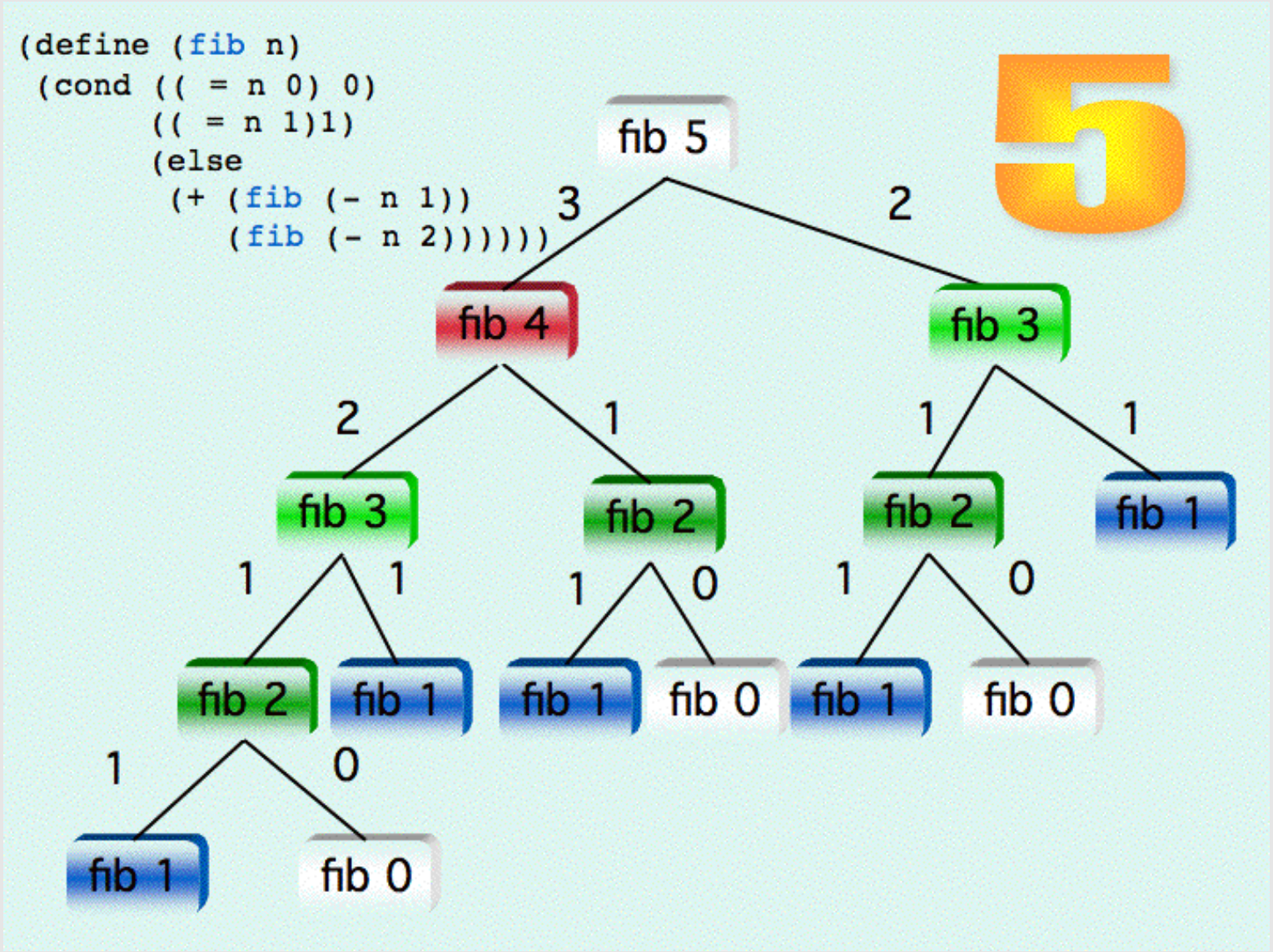
Caching ist eine Technik, bei der Werte, die aufwendig zu berechnen sind, gespeichert werden. Wenn ein solcher Wert ein zweites Mal benötigt wird, dann wird er einfach aus dem Speicher ausgelesen.

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:
fib (n) = $\mathcal{O}(1.7^n)$

```
(define (fib n)
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [else (+ (fib (- n 1))
                 (fib (- n 2)))]))
```


(define (fib n)
(cond ((= n 0) 0)
((= n 1)1)
(else
(+ (fib (- n 1))
(fib (- n 2))))))
fib 5
3
2
fib 4
fib 3
2
1
1
1
fib 3
fib 2
fib 2
fib 1
1
1
1
0
1
0
fib 2
fib 1
fib 1
fib 0
fib 1
fib 0
1
0
fib 1
fib 0
5

Memoization

Beispiel (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))
```

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)))]
....
```

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
```

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 imperative Sprachelemente eingesetzt:

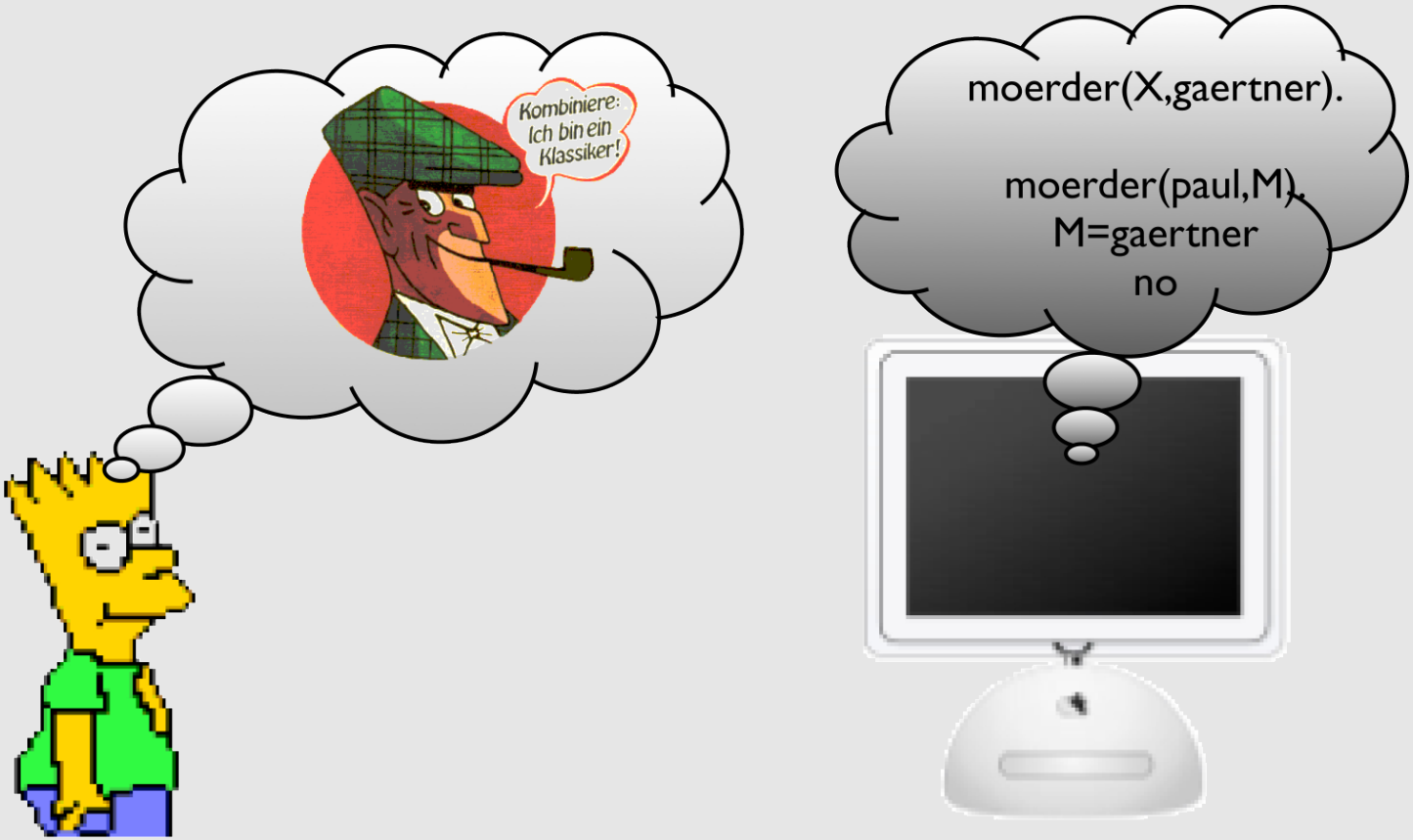
- Das Speichern in der Tabelle: store
- Die Modifikation der Referenz auf fib.
- Zusammen haben diese beiden Maßnahmen eine Funktion von exponentieller Ordnung in eine Funktion von linearer Ordnung gewandelt.

Teil XII

Relationale Programmierung

Relationale Programmierung

Das relationale/logische Verarbeitungsmodell



Das logische Verarbeitungsmodel

Programmieren mit Relationen

Das logische Verarbeitungsmodel 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 Verabeitungsmodells 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.

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.

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)` $\longrightarrow$ `(Lee Kim)`

- Eine Funktion „likers-of“, die die Menge aller Personen berechnet, die etwas mögen, z.B.

  `(likers-of 'Lee)` $\longrightarrow$ `(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) $\longrightarrow$?whom=Lee , ?whom=Kim
(likes ?who Lee) $\longrightarrow$?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.*

```
(<- (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:

```
(<- (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 <− Notation betont diese Semantik:
  Der Klausel-Operator <− symbolisiert sowohl die logische Implikation ← 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.*

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.

Programmieren in Prolog

Wir programmieren Prolog-Programme

- indem wir relevante Relationen als Klauseln in der Datenbasis zusichern (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.

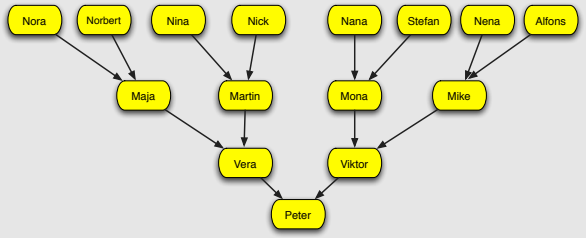
```
(defmacro* (?- &rest goals)
  `(top-level-prove
    (replace-?-vars (quote ,goals))))
```

- 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.

Prolog als Datenbanksprache

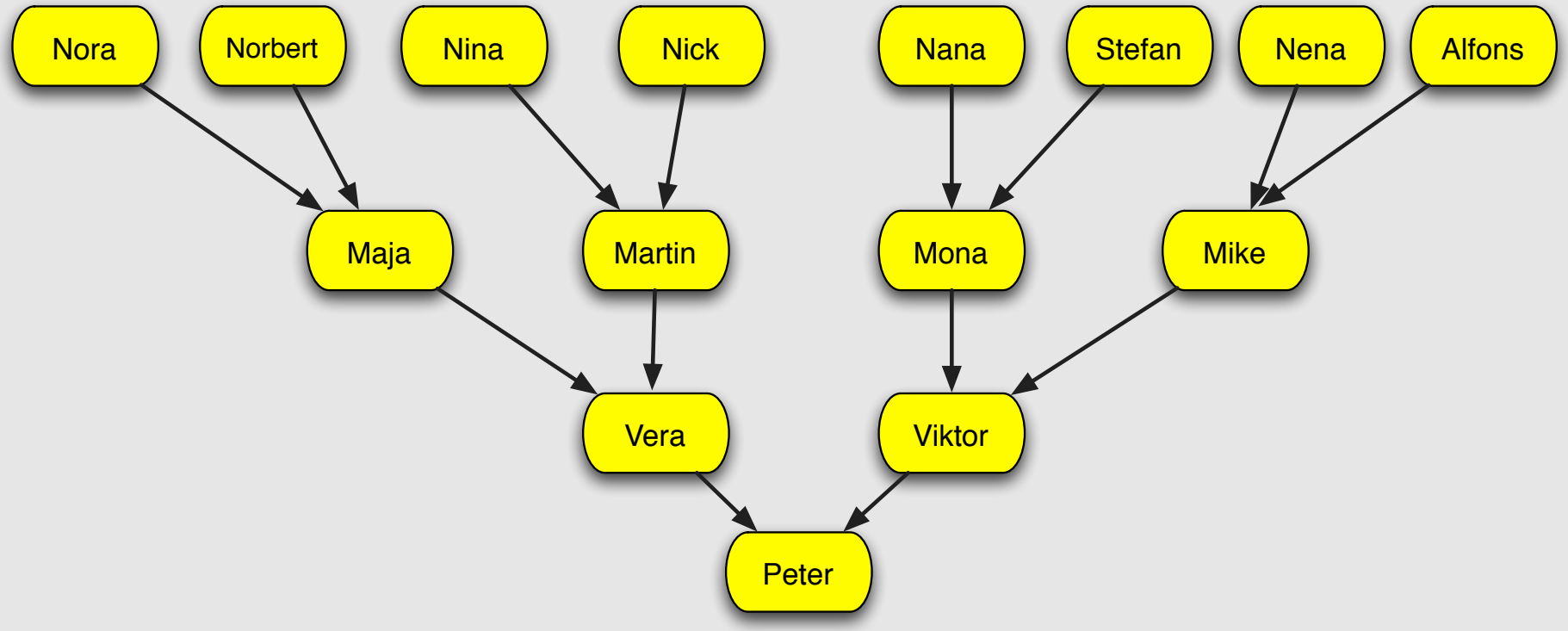


Beispiel: Familienverhältnisse

Beispiel (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))
```

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))
```

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:

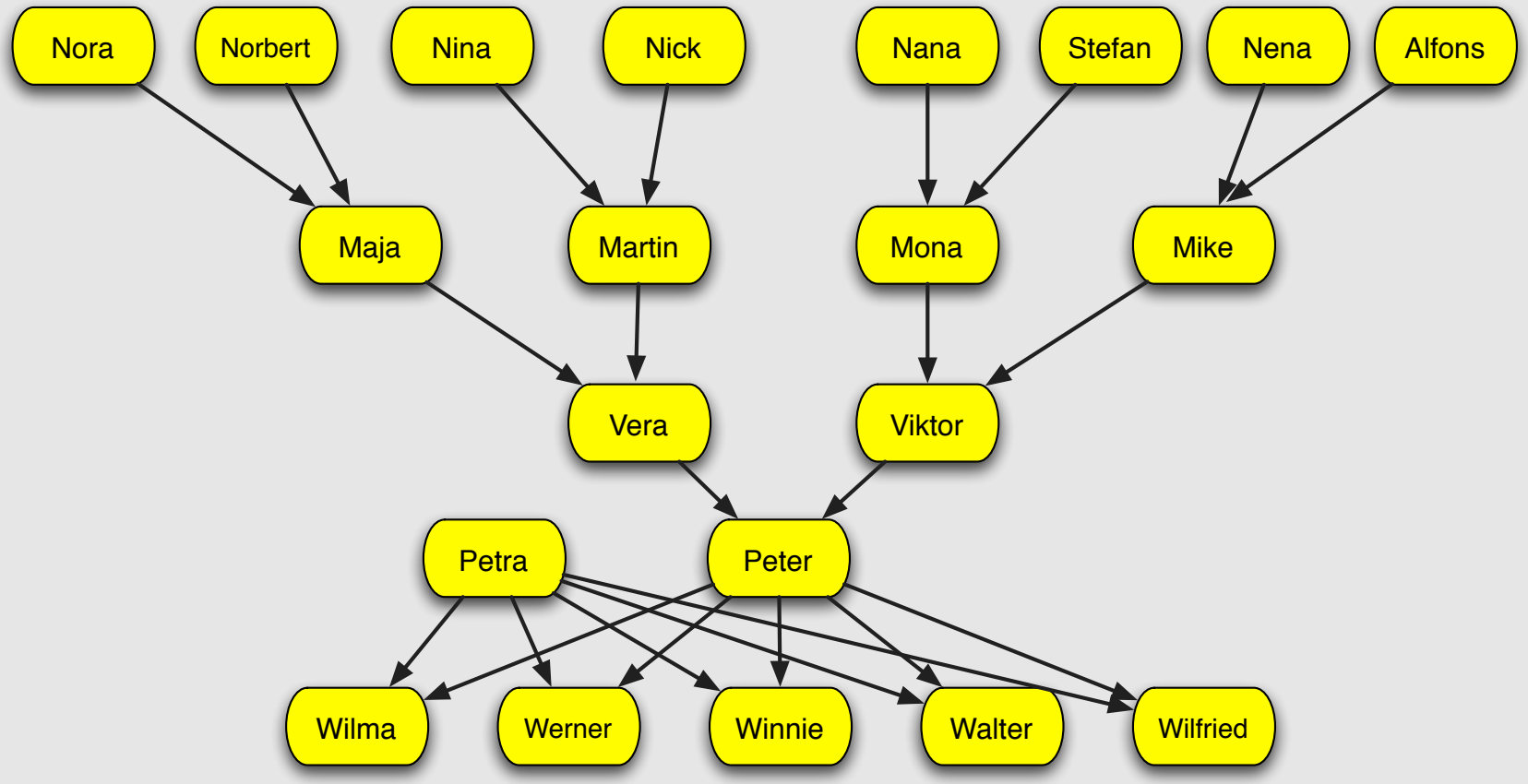
```
> (?- (predecessor Maja ?desc))
?desc = Vera
;
?desc = Peter
;
 No.
>
```

Teil XII

Relationale Programmierung

Prolog in Racket 2: Deduktive Datenbanken

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 Ungleichheitheit

- 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)

- $\wedge$: Die Konjunktion wird implizit in der Liste aller Prämissen einer Regel ausgedrückt.
- $\vee$: Die Disjunktion wird implizit in alternativen Regeln für dasselbe Prädikat ausgedrückt.
- $\neg$: 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 (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 instanziierte Varianten des Ausdrucks <Term> .
Der Term und das Ziel sind über gemeinsame uninstanziierte 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.
```

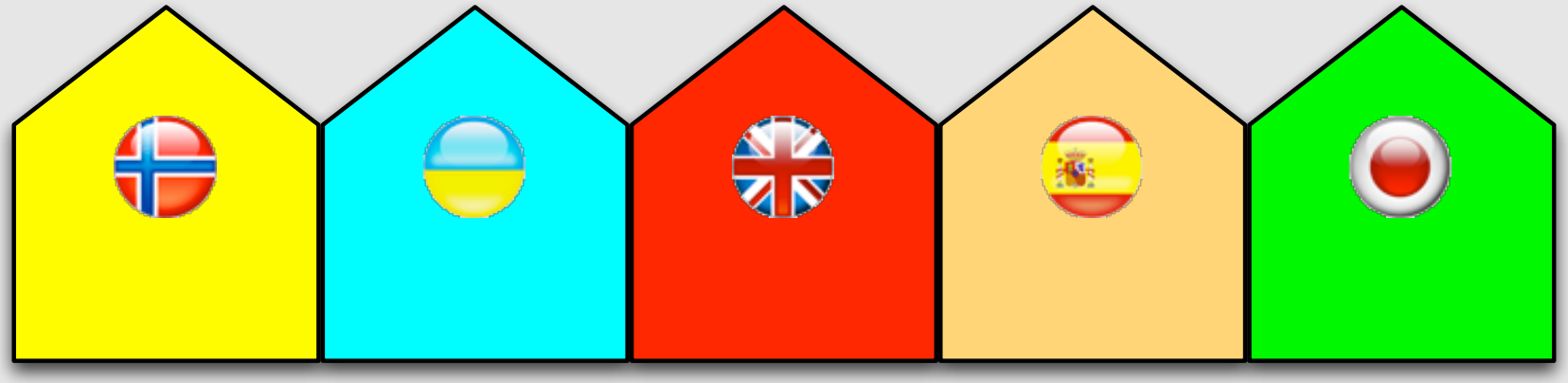
Prolog als Inferenzmaschine

Das Zebra-Rätsel

Beispiel

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.

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), iright (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))
```

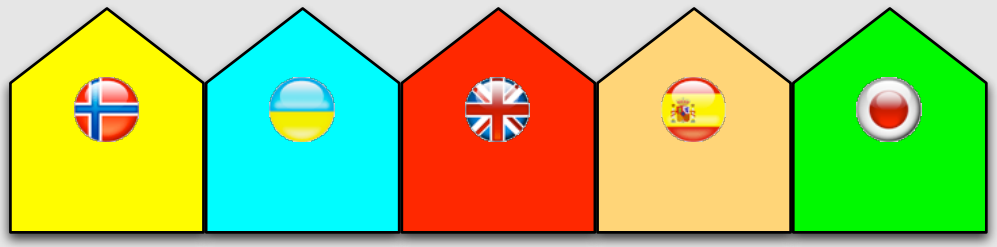
Einschränkende Relationen

```
(<- (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)
```

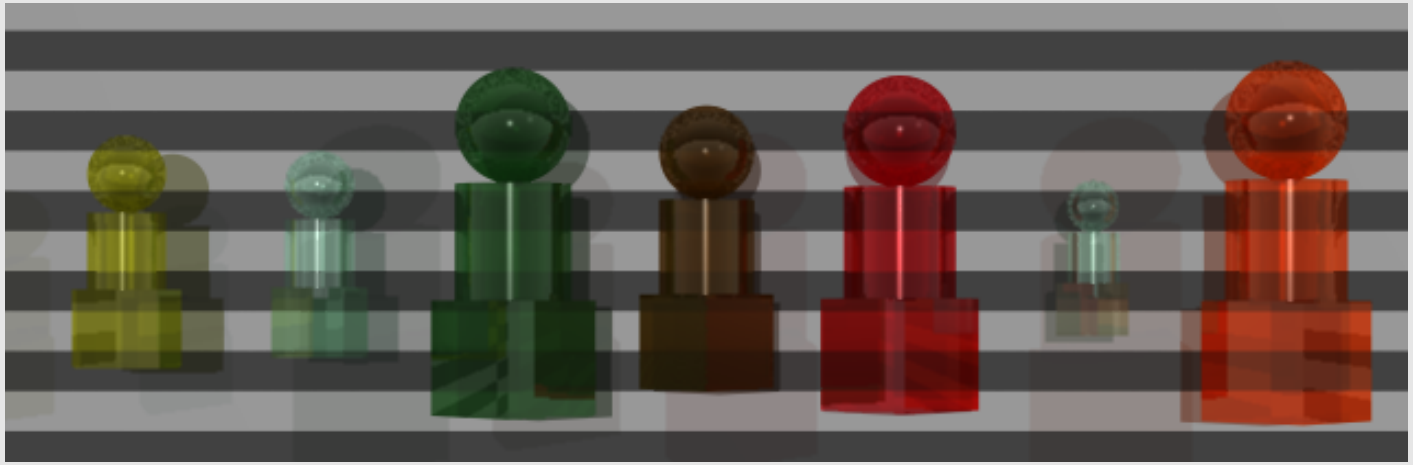
```
(member (house ? snails winston ? ?) ?h) ; 7
(member (house ? ? kools ? yellow) ?h) ; 8
(nextto (house ? ? chesterfield ? ?) ;11
        (house ? fox ? ? ?) ?h)
(nextto (house ? ? kools ? ?) ;12
        (house ? horse ? ? ?) ?h)
(member (house ? ? luckystrike oJuice ?)
        ?h) ;13
(member (house 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.
```



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 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.

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. Zwerg 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)
```

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 Unifikation ist erfolgreich, sofern der Ausdruck „wahr“ ergibt.
- Alle Variablen des Ausdrucks müssen instanziiert 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.
```

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.
```

- Memoization

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.

Teil XIII

Zusammenfassung

Zusammenfassung und Ausblick

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

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

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

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

Zusammenfassung: Eigenschaften

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		

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

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 treffendsten 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

Klausuren

1. Klausur

Wann und wo: Mo, 19. Feb. 2018 09:30-11:30, ESA A, ESA B , Einlaß ab 9:15 Uhr

Erlaubte Hilfsmittel: Schreibzeug, ggfs. Wörterbuch für die deutsche Sprache

Ausweise: Ein Lichtbildausweis, Studentenausweis

2. Klausur

Tutorium: wird bekanntgegeben

Wann und wo: Di, 20. März 2018 09:30-11:30, ESA B, Einlaß ab 9:15 Uhr

- ☞ Bitte seien Sie spätestens eine Viertelstunde vor Klausurbeginn da.
- ☞ Bitte bald zur Klausur anmelden!

ENDE

Viel Erfolg für das weitere Studium!

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

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
```

Abbilden eines Stroms

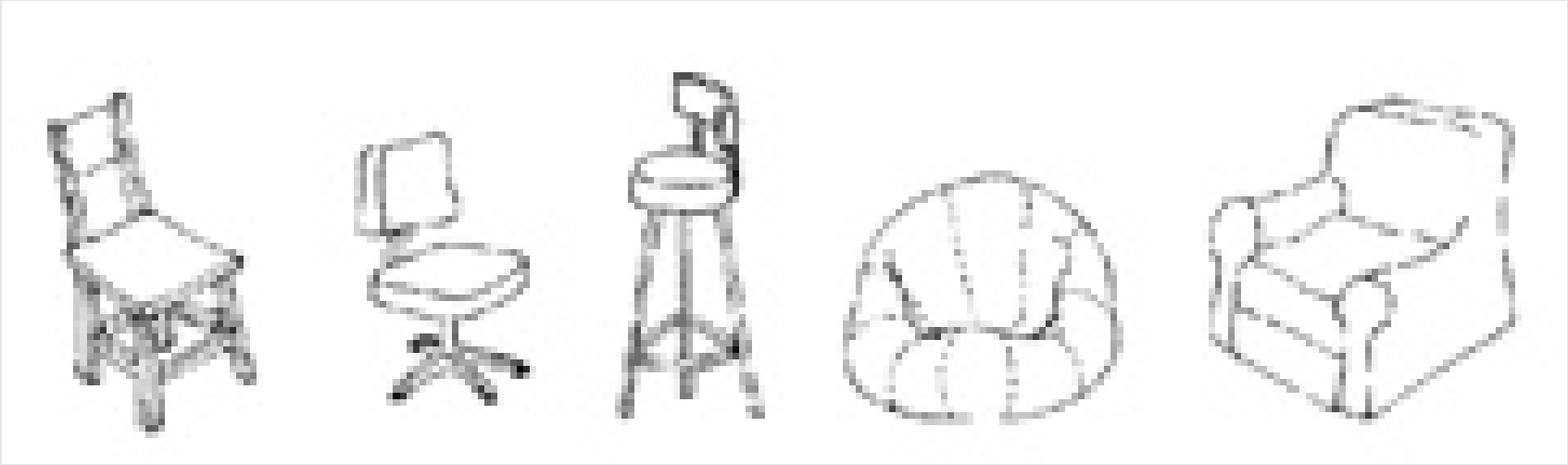
```
;; map each element of a stream
(define (map-stream proc stream)
    (if (empty-stream? stream)
        the-empty-stream
        (cons
         (proc (head-stream stream))
         (delay
           (map-stream
            proc (tail-stream stream))))))
> (map-stream add1 ab-3)
⟶ (4 . #<struct:promise>)
```

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))))))
```

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.



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.

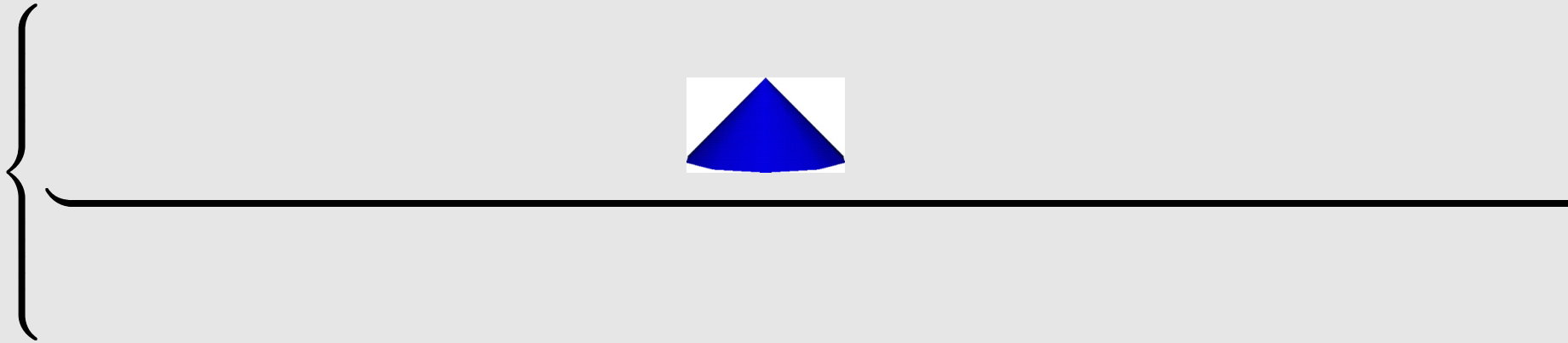
Formal wird die Abstraktionsmethode mithilfe des Abstraktionsschemas dargestellt:

- 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. [**?**]

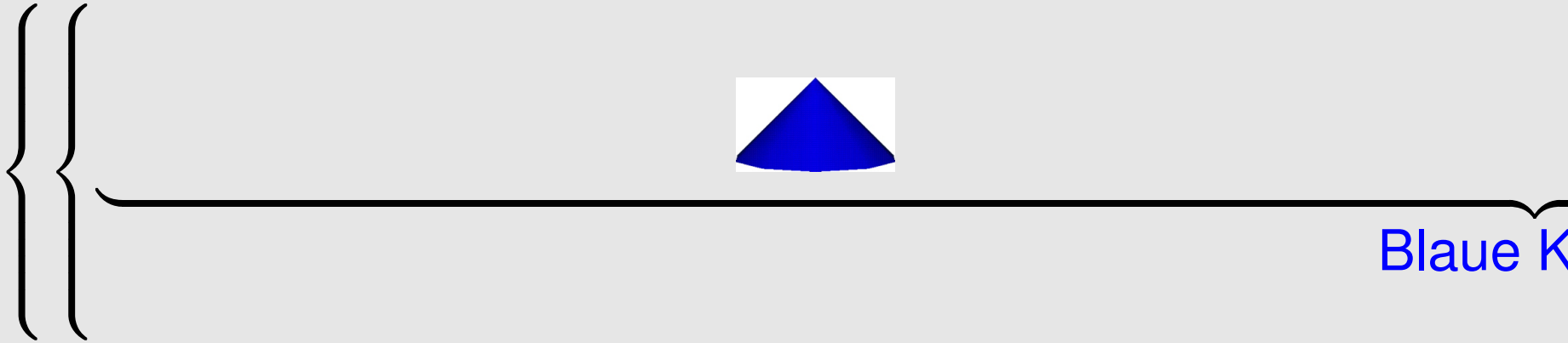
Generalisierung

- 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.*

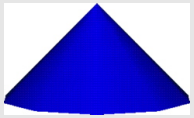
Äquivalenzklassen von Objekten



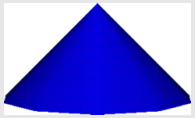
Äquivalenzklassen für $R_{\text{gleicheFarbe}}$



Blaue K



Äquivalenzklassen für $R_{\text{gleicheForm}}$



△Keg

Abstraktoren und invariante Aussagen

Nach der Definition von [**?**]:

- 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 (Abstraktion)

- Abstraktion ist der Übergang von Aussagen

$$A(a) \quad \text{über Objekte } a, b$$

mit einer zwischen ihnen erklärten

Äquivalenzrelation $\sim$

zu Aussagen $A(\tilde{\alpha})$

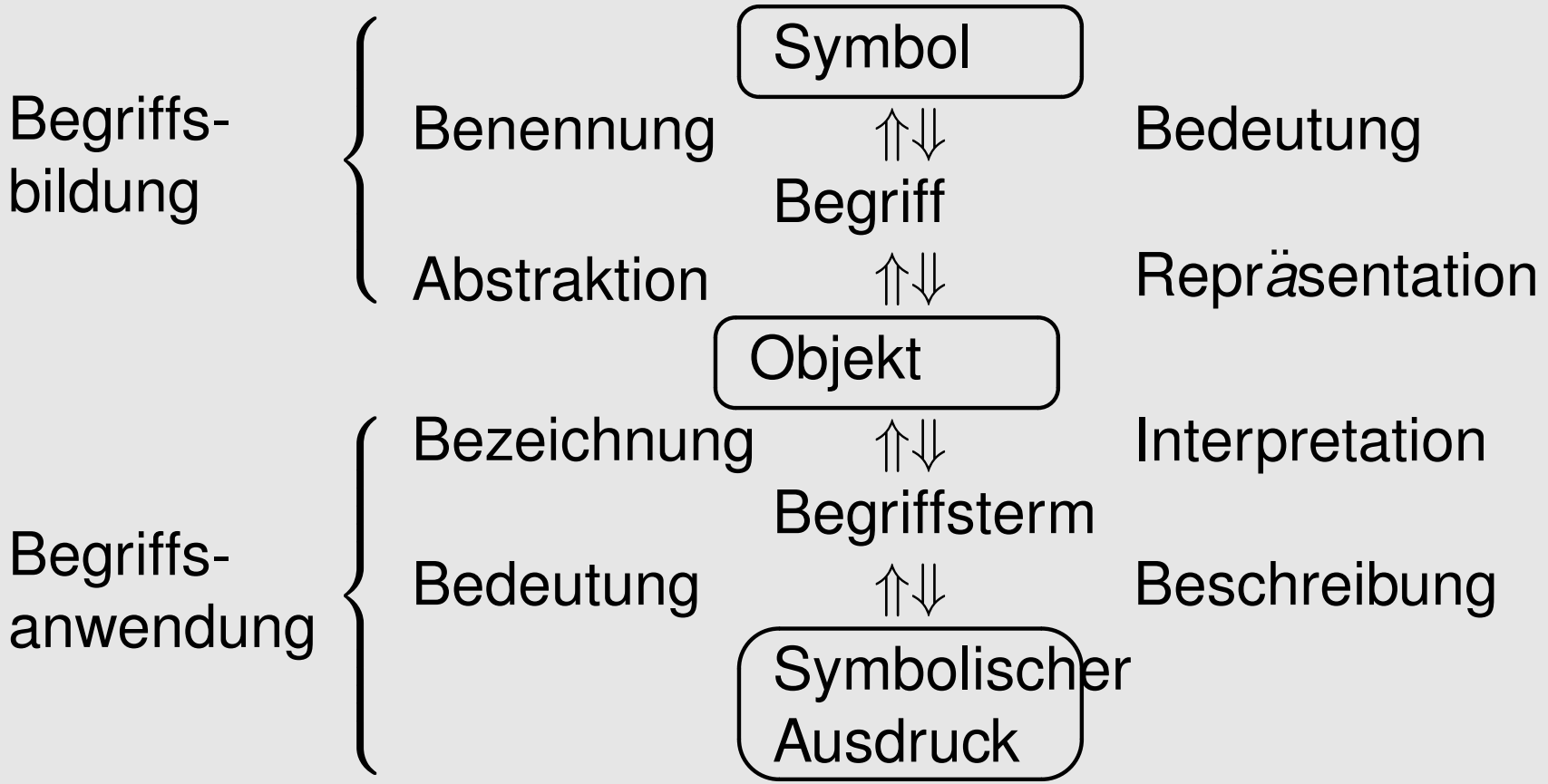
- Mittels des *Abstraktionsschemas*

$$A(\tilde{x}) \rightleftharpoons \forall_y (x \sim y \rightarrow A(y))$$

wird die Rede über die *abstrakten Objekte* eingeführt: $\tilde{a}, \tilde{b}, \cdots$

- 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*




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).