20. Februar 1998

Scheme

Revised(5) Report on the Algorithmic Language Scheme

Überarbeiteter^5 Bericht über die Algorithmische Sprache Scheme.

RICHARD **K**ELSEY, **W**ILLIAM **C**LINGER
UND **J**ONATHAN **R**EES (***Herausgeber***)
H. ABELSON
R. K. DYBVIG
C. T. HAYNES
G. J. ROZAS
N. I. ADAMS IV
D. P. FRIEDMAN
E. KOHLBECKER
G. L. STEELE **J**R.
D. H. BARTLEY
R. HALSTEAD
D. OXLEY
G. J. SUSSMAN
G. BROOKS
C. HANSON
K. M. PITMAN
M. WAND

Der Erinnerung an Robert Hieb gewidmet

Zusammenfassung

[Der englische Originaltext dieses Berichts] enthält eine normative Beschreibung der Programmiersprache Scheme. Scheme ist ein Dialekt der Lisp-Programmiersprache mit statischen Geltungsbereichen (auch bekannt als statische Sichtbarkeitsbereiche) und echter Endrekursion, der von Guy Lewis Steele Jr. und Gerald Jay Sussman erfunden wurde. Er wurde mit einem Augenmerk auf besonders klare und einfache Semantik entworfen mit nur wenigen verschiedenen Arten, Ausdrücke zu bilden. Eine große Vielfalt von Programmierparadigmen, unter ihnen imperative, funktionale und nachrichtenweitergebende Stile, finden in Scheme eine gelegene Ausdrucksweise.

Diese Einleitung bietet eine kurze Geschichte der Sprache und des Berichts.

Die ersten drei Kapitel zeigen die grundlegenden Ideen der Sprache und beschreiben die Konventionen der Notation, mit der die Sprache beschrieben und Programme in der Sprache geschrieben werden.

Die Kapitel Kapitel 4 [Ausdrücke], Seite 17, und Kapitel 5 [Programmstruktur], Seite 34, beschreiben Syntax und Semantik von Ausdrücken, Programmen und Definitionen.

Das Kapitel Kapitel 6 [Standardprozeduren], Seite 37, beschreibt die in Scheme eingebauten Prozeduren, wozu alle elementaren Bausteine der Sprache zur Datenmanipulation und zur Ein- und Ausgabe gehören.

Das Kapitel Kapitel 7 [Formale Syntax und Semantik], Seite 82, bietet eine formale Syntax für Scheme, geschrieben in erweiterter BNF, zusammen mit einer formalen denotationellen Semantik. Auf die formale Syntax und Semantik folgt ein Beispiel der Nutzung der Sprache.

Der Bericht endet mit einer Liste von Verweisen und einem alphabetischen Stichwortverzeichnis.

Einleitung

Programmiersprachen sollten nicht entworfen werden, indem man Funktionalitäten über Funktionalitäten schichtet, sondern indem man Schwächen und Einschränkungen entfernt, die es so erscheinen lassen, als wären weitere Funktionalitäten nötig. Scheme demonstriert, dass eine sehr kleine Zahl von Regeln, wie Ausdrücke gebildet werden, ohne Einschränkungen, wie man sie zusammenfügt, genügen, um eine praktische und effiziente Programmiersprache zu bilden, die flexibel genug ist, die meisten größeren Programmierparadigmen, die heute genutzt werden, zu bedienen.

Scheme war eine der ersten Programmiersprachen, die Prozeduren erster Klasse wie im Lambda-Kalkül enthält, und hat so bewiesen, wie nützlich statische Geltungsbereiche und eine Blockstruktur in einer dynamisch typisierten Sprache sind. Scheme war der erste größere Dialekt von Lisp, der Prozeduren von Lambda-Ausdrücken und Symbole unterscheidet, der eine einzelne lexikalische Umgebung für alle Variablen benutzt und der die Operatorposition eines Prozeduraufrufs auf gleiche Weise auswertet wie eine Operandenposition. Indem Scheme Iterationen allein mit Prozeduraufrufen ausdrückt, betonte Scheme, dass endrekursive Prozeduraufrufe im Kern goto-Ausdrücke, die Argumente übergeben, sind. Scheme war die erste weit verbreitete Programmiersprache, die Ausstiegsprozeduren erster Klasse, aus denen alle anderen Programmflussstrukturen synthetisiert werden können, rückhaltlos unterstützte. Eine spätere Fassung von Scheme führte das Konzept exakter und inexakter Zahlen ein, eine Erweiterung der generischen Arithmetik von Common Lisp. Vor weniger langer Zeit wurde Scheme zur ersten Programmiersprache, die hygienische Makros unterstützt, welche Syntaxerweiterungen von in Blöcken strukturierten Sprachen auf konsistente und zuverlässige Weise zulassen.

Hintergrund

Die erste Beschreibung von Scheme wurde im Jahr 1975 [Scheme75] geschrieben. Ein überarbeiteter Bericht („revised report“) [Scheme78] erschien 1978. Er beschrieb die Entwicklung der Sprache, als seine MIT-Implementierung ausgebaut wurde, um einen innovativen Übersetzer (Compiler) [Rabbit]. Drei separate Projekte begannen 1981 und 1982, um Varianten von Scheme für Kurse am MIT, Yale und der Indiana University zu benutzen [Rees82], [MITScheme], [Scheme311]. Ein einführendes Informatiklehrbuch, das Scheme benutzt, wurde 1984 veröffentlicht [SICP]. (Anm. d. Übers.: Besagtes Lehrbuch „Struktur und Interpretation von Computerprogrammen“ ist auch auf Deutsch übersetzt erhältlich.)

Als sich Scheme weiter verbreitete, fingen lokale Dialekte an, auseinanderzugehen, bis Studierende und Forschende Probleme bekamen, den an anderen Orten geschriebenen Code zu verstehen. Fünfzehn Repräsentanten der größeren Implementierungen von Scheme versammelten sich deshalb im Oktober 1984, um an einem besseren und von mehr Leuten anerkannten Standard für Scheme zu arbeiten.

Ihr Bericht [RRRS] wurde am MIT und der Indiana University im Sommer 1985 veröffentlicht. Weitere Revisionen fanden im Frühling 1986 [R3RS] und im Frühling 1988 [R4RS] statt. Der vorliegende Bericht entspricht weiteren Revisionen, auf die man sich in einer Besprechung bei Xerox PARC im Juni 1992 geeinigt hat.

Wir beabsichtigen, dass dieser Bericht der gesamten Scheme-Gemeinschaft gehören soll, und erteilen damit die Erlaubnis, ihn gänzlich oder in Teilen ohne Gebühr zu kopieren. Insbesondere ermutigen wir Implementierer von Scheme, diesen Bericht als Ausgangspunkt für Handbücher und andere Dokumentation zu benutzen und nach Bedarf zu verändern.

Anerkennungen

Wir möchten den folgenden Leuten für ihre Hilfe danken: Alan Bawden, Michael Blair, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Marc Feeley, Andy Freeman, Richard Gabriel, Yekta G"ursel, Ken Haase, Robert Hieb, Paul Hudak, Morry Katz, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Mike Shaff, Jonathan Shapiro, Julie Sussman, Perry Wagle, Daniel Weise, Henry Wu und Ozan Yigit. Wir danken Carol Fessenden, Daniel Friedman und Christopher Haynes für die Erlaubnis, Text aus dem Referenzhandbuch von Scheme 311 Version 4 zu verwenden. Wir danken Texas Instruments, Inc. für die Erlaubnis, Text aus dem *TI Scheme Language Reference Manual*[TImanual85] zu verwenden. Wir erkennen gerne den Einfluss der Handbücher für MIT Scheme[MITScheme], T[Rees84], Scheme 84[Scheme84], Common Lisp[CLtL] und Algol 60[Naur63] an.

Wir danken auch Betty Dexter für die enorme Mühe, mit der sie diesen Bericht in TEX gesetzt hat, und Donald Knuth für den Entwurf des Programms, dass ihr Ärger bereitet hat.

Das Artificial Intelligence Laboratory des Massachusetts Institute of Technology, das Computer Science Department der Indiana University, das Computer and Information Sciences Department der University of Oregon und das NEC Research Institute haben die Vorbereitung dieses Berichts unterstützt. Unterstützung für die Arbeit des MIT kam in Teilen von der Advanced Research Projects Agency des Department of Defense unter dem Vertrag N00014-80-C-0505 des Office of Naval Research. Unterstützung für die Arbeit der Indiana University gab es durch NSF-Förderungen NCS 83-04567 und NCS 83-03325.

1 Übersicht von Scheme

1.1 Semantik

Dieser Abschnitt vermittelt eine Übersicht über die Semantik von Scheme. Eine detaillierte informelle Semantik ist Thema der Kapitel Kapitel 3 [Grundkonzepte], Seite 11, bis Kapitel 6 [Standardprozeduren], Seite 37. Als Referenz bietet der Abschnitt Abschnitt 7.2 [Formale Semantik], Seite 88, eine formale Semantik von Scheme.

Scheme folgt Algol darin, dass es eine Programmiersprache mit statischen Geltungsbereichen ist. Jede Nutzung einer Variablen wird mit einer lexikalisch offenbaren Bindung dieser Variablen assoziiert.

Scheme benutzt latente statt ausdrücklicher Typisierung. Typen werden mit Werten (auch Objekte genannt) assoziiert statt mit Variablen. (Manche Autoren nennen latent typisierte Sprachen auch schwach typisierte oder dynamisch typisierte Sprachen.) Andere Sprachen mit latenten Typen sind APL, Snobol und andere Lisp-Dialekte. Sprachen mit ausdrücklichen Typen (manchmal stark typisierte oder statisch typisierte Sprachen genannt) sind unter Anderem Algol 60, Pascal und C.

Alle Objekte, die im Laufe einer Scheme-Berechnung erzeugt werden, einschließlich Prozeduren und Fortsetzungen, haben einen unbeschränkten Gültigkeitsbereich. Kein Scheme-Objekt wird jemals zerstört. Der Grund dafür, dass Scheme-Implementierungen (meistens!) nicht der Speicher ausgeht, ist, dass sie sich den von einem Objekt belegten Speicher zurückholen dürfen, wenn sie beweisen können, dass das Objekt keinen Einfluss auf jegliche zukünftige Berechnung haben kann. Andere Sprachen, in denen die meisten Objekte einen unbeschränkten Gültigkeitsbereich haben, sind unter Anderem APL und andere Lisp-Dialekte.

Scheme-Implementierungen sind verpflichtet, echt endrekursiv zu sein. Dadurch kann die Ausführung einer iterativen Berechnung mit konstantem Speicher auskommen, selbst wenn die iterative Berechnung syntaktisch durch eine rekursive Prozedur beschrieben wird. Somit kann mit einer echt endrekursiven Implementierung die Iteration mit der herkömmlichen Mechanik von Prozeduraufrufen ausgedrückt werden, so dass besondere Konstrukte für die Iteration nur als syntaktischer Zucker nützlich sind. Siehe den Abschnitt Abschnitt 3.5 [Echte Endrekursion], Seite 13.

Scheme-Prozeduren sind selbst auch Objekte. Prozeduren können dynamisch erstellt werden, in Datenstrukturen gespeichert werden, als Ergebnisse von Prozeduren geliefert werden und so weiter. Andere Sprachen mit diesen Eigenschaften sind unter Anderem Common Lisp und ML.

Eine Scheme auszeichnende Funktionalität ist, dass Fortsetzungen (Continuations), welche in den meisten anderen Sprachen nur im Hintergrund bestehen, auch als Objekte „erster Klasse“ gelten. Fortsetzungen sind nützlich bei der Implementierung einer großen Vielzahl von fortgeschrittenen Programmflussstrukturen, einschließlich nicht-lokaler Sprünge, Rücksetzverfahren (engl. Backtracking) und Ko-Routinen. Siehe den Abschnitt Abschnitt 6.4 [Programmflussfunktionalitäten], Seite 68.

Argumente an Scheme-Prozeduren werden immer als Wertparameter übergeben. Das bedeutet, die eigentlichen Argument-Ausdrücke werden zuerst ausgewertet, bevor die Prozedur ausgeführt wird, egal ob die Prozedur die Ergebnisse der Auswertung überhaupt benutzt.

ML, C und APL sind drei andere Sprachen, die auch Argumente immer als Wertparameter übergeben. Dies unterscheidet sich von Haskells bis zuletzt verzögerter Auswertung („Auswertung nach Vorschrift“) oder den Namensparametern von Algol 60, bei denen ein Argument-Ausdruck erst dann ausgewertet wird, wenn sein Wert von der Prozedur gebraucht wird.

Schemes Modell der Arithmetik wurde so gestaltet, dass es möglichst unabhängig davon ist, welche bestimmten Zahlendarstellungen ein Rechner benutzt. In Scheme ist jede ganze Zahl eine rationale Zahl, jede rationale Zahl ist eine reelle Zahl und jede reelle Zahl ist eine komplexe Zahl. Daher besteht in Scheme kein Unterschied zwischen ganzzahliger und reeller Arithmetik, welcher für andere Sprachen so wichtig ist. Stattdessen unterscheidet Scheme zwischen exakter Arithmetik, welche dem mathematischen Ideal entspricht, und nicht exakter Arithmetik, die Annäherungen macht. Wie in Common Lisp ist exakte Arithmetik nicht auf ganze Zahlen beschränkt.

1.2 Syntax

Scheme benutzt wie die meisten Lisp-Dialekte eine vollständig geklammerte Präfix-Notation für Programme und (andere) Daten; die Grammatik von Scheme erzeugt eine Untersprache der für Daten benutzten Sprache. Eine wichtige Folge dieser einfachen, einheitlichen Darstellung ist, dass Scheme-Programme und Daten von anderen Scheme-Programmen oft gleich behandelt werden. Zum Beispiel wertet die ‘`Eval`’-Prozedur ein als Daten ausgedrücktes Scheme-Programm aus.

Die ‘`Read`’-Prozedur führt syntaktische sowie lexikalische Aufgliederung der von ihr gelesenen Daten durch. Die ‘`Read`’-Prozedur analysiert ihre Eingabe grammatikalisch als Daten (Abschnitt siehe Abschnitt 7.1.2 [Externe Darstellung], Seite 84) und nicht als Programm.

Die formale Syntax von Scheme ist im Abschnitt Abschnitt 7.1 [Formale Syntax], Seite 82, beschrieben.

1.3 Notation und Terminologie

1.3.1 Elementare; Bibliotheks- und optionale Funktionalitäten

Jede Implementierung von Scheme muss alle nicht als *optional* markierten Funktionalitäten von Scheme unterstützen. Implementierungen steht es frei, optionale Funktionalitäten von Scheme wegzulassen oder Erweiterungen anzubieten, solange diese Erweiterungen nicht im Konflikt mit der Sprache steht, über die hier berichtet wird. Insbesondere müssen Implementierungen portablen Code unterstützen, indem sie einen syntaktischen Modus anbieten, der keinen Schreibkonventionen dieses Berichts zuvorkommt.

Um das Verständnis und die Implementierung von Scheme zu erleichtern, sind manche Funktionalitäten als *Bibliotheks*funktionalitäten markiert. Diese können leicht unter Benutzung der anderen, elementaren, Funktionalitäten implementiert werden. Streng genommen sind sie überflüssig, aber sie fassen übliche Muster, wie Scheme benutzt wird, und werden daher als bequeme Kurzschreibweisen angeboten.

1.3.2 Fehlersituationen und unbestimmtes Verhalten

Wenn dieser Bericht von einer Fehlersituation spricht, zeigt der Bericht mit „ein Fehler wird signalisiert“ an, dass Implementierungen den Fehler feststellen und melden müssen.

Wenn in der Diskussion eines Fehlers keine solche Formulierung auftaucht, dann müssen Implementierungen den Fehler nicht feststellen oder melden, sind aber dazu angehalten. Eine Fehlersituation, die Implementierungen nicht feststellen müssen, wird in der Regel einfach als „ein Fehler“ bezeichnet.

Zum Beispiel ist es ein Fehler, wenn einer Prozedur ein Argument übergeben wird, für das nicht spezifiziert wurde, dass die Prozedur damit umgehen kann, selbst wenn solche Definitionsbereichsfehler in diesem Bericht nur selten erwähnt werden. Implementierungen dürfen den Definitionsbereich einer Prozedur erweitern, so dass er solche Argumente auch einschließt.

Dieser Bericht verwendet die Formulierung „darf eine Verletzung einer Implementierungseinschränkung melden“, um Umstände anzuzeigen, unter denen eine Implementierung melden darf, dass sie die Ausführung eines korrekten Programms wegen einer von der Implementierung geforderten Einschränkung nicht fortführen kann. Von Implementierungseinschränkungen wird selbstverständlich abgeraten, aber Implementierungen werden ermutigt, Verletzungen solcher Implementierungseinschränkungen zu melden.

Zum Beispiel darf eine Implementierung eine Verletzung einer Implementierungseinschränkung melden, wenn sie nicht genug Speicherkapazität hat, um ein Programm auszuführen.

Wenn der Wert eines Ausdrucks als „unbestimmt“ angegeben wird, dann muss dieser Ausdruck zu irgendeinem Objekt ausgewertet werden, ohne einen Fehler zu signalisieren, aber zu welchem Wert hängt von der Implementierung ab; dieser Bericht schreibt ausdrücklich nicht vor, welcher Wert geliefert werden sollte.

1.3.3 Eintragsformat

Die Kapitel Kapitel 4 [Ausdrücke], Seite 17, und Kapitel 6 [Standardprozeduren], Seite 37, sind in Einträge unterteilt. Jeder Eintrag beschreibt eine Funktionalität der Sprache oder eine Gruppe zusammengehöriger Funktionalitäten, wobei eine Funktionalität entweder ein syntaktisches Konstrukt oder eine eingebaute Prozedur ist. Ein Eintrag beginnt mit einer oder mehreren Kopfzeilen der Form

`Schablone` [*Kategorie*]
für verpflichtende, elementare Funktionalitäten oder

`Schablone` [*Qualifikator Kategorie*]
wobei *Qualifikator* entweder „Bibliotheks“ oder „optionale“ sein kann gemäß der Definition im Abschnitt Abschnitt 1.3.1 [Elementare; Bibliotheks- und optionale Funktionalitäten], Seite 4.

Wenn die *Kategorie* als „Syntax“ angegeben ist, beschreibt der Eintrag einen Ausdruckstyp und die Schablone gibt die Syntax des Ausdruckstyps an. Bestandteile von Ausdrücken werden als syntaktische Variable ausgezeichnet, welche mit spitzen Klammern geschrieben werden, zum Beispiel <Ausdruck> oder <Variable>. Syntaktische Variable sollten verstanden werden als etwas, was Programmtextsegmente bezeichnet; zum Beispiel steht <Ausdruck> für eine beliebige Zeichenkette, die einen syntaktisch gültigen Ausdruck darstellt. Die Notation

<Ding1> ...

zeigt null oder mehr Vorkommen eines <Ding>s an, wogegen

<Ding1> <Ding2> ...

eines oder mehr Vorkommen eines <Ding>s anzeigt.

Wenn *Kategorie* „Prozedur“ ist, dann beschreibt der Eintrag eine Prozedur und die Kopfzeile gibt eine Schablone eines Aufrufs der Prozedur an. Argumentnamen in der Schablone sind *kursiv*. Somit zeigt die Kopfzeile

`vector-ref` *`vektor k`* [Prozedur]

an, dass die eingebaute Prozedur `vector-ref` zwei Argumente nimmt, einen Vektor *Vektor* und eine exakte, nicht negative ganze Zahl *k* (siehe unten). Die Kopfzeilen

`make-vector` *`k`* [Prozedur]
`make-vector` *`k fill`* [Prozedur]

zeigen an, dass die Prozedur `Make-vector` so definiert werden muss, dass sie entweder ein oder zwei Argumente nimmt.

Es ist ein Fehler, wenn einer Operation ein Argument vorgelegt wird, wofür nicht spezifiziert wurde, dass sie damit umgehen kann. Der Kürze wegen folgen wir der Konvention, dass wenn ein Argumentname auch der Name eines im Abschnitt Abschnitt 3.2 [Typfremdheit], Seite 11, aufgeführten Typs ist, dieses Argument vom genannten Typen sein muss. Zum Beispiel schreibt obige Kopfzeile für `Vector-ref` vor, dass das erste Argument an `Vector-ref` ein Vektor sein muss. Die folgenden Namenskonventionen implizieren ebenfalls Typeinschränkungen:

Objekt ein beliebiges Objekt, auch englisch *Object* geschrieben

Liste, *Liste1*, ... *Listej*, ...
Liste (siehe Abschnitt siehe Abschnitt 6.3.2 [Paare und Listen], Seite 54, auch englisch *List* geschrieben)

z, *z1*, ... *zj*, ...
komplexe Zahl

x, *x1*, ... *xj*, ...
reelle Zahl

y, *y1*, ... *yj*, ...
reelle Zahl

q, *q1*, ... *qj*, ...
rationale Zahl

n, *n1*, ... *nj*, ...
ganze Zahl

k, *k1*, ... *kj*, ...
exakte, nicht negative ganze Zahl

1.3.4 Auswertungsbeispiele

Das in Programmbeispielen benutzte Symbol „⇒“ sollte als „wird ausgewertet zu“ gelesen werden. Zum Beispiel bedeutet

```
(* 5 8)                                    ==>  40
```

dass der Ausdruck `(* 5 8)` zum Objekt `40` ausgewertet wird. Oder genauer gesagt: Der Ausdruck, der sich aus der Zeichenfolge „`(* 5 8)`“ ergibt, wird, in der Anfangsumgebung, ausgewertet zu einem Objekt, das durch die Zeichenfolge „`40`“ extern dargestellt werden kann. Siehe den Abschnitt Abschnitt 7.1.2 [Externe Darstellung], Seite 84, für eine Diskussion der externen Darstellungen eines Objekts.

1.3.5 Namenskonventionen

Nach Konvention enden die Namen von Prozeduren, die immer einen booleschen Wert liefern, gewöhnlich auf „`?`“. Solche Prozeduren werden Prädikate genannt.

Nach Konvention enden die Namen von Prozeduren, die Werte in bereits zugeteilte Speicherstellen einspeichern (siehe den Abschnitt siehe Abschnitt 3.4 [Speichermodell], Seite 13), gewöhnlich auf „`!`“. Solche Prozeduren werden Veränderungsprozeduren genannt. Nach Konvention ist der von einer Veränderungsprozedur gelieferte Wert unbestimmt.

Nach Konvention erscheint „`->`“ im Namen von Prozeduren, die ein Objekt einen Typs nehmen und ein entsprechendes Objekt eines anderen Typs liefern. Zum Beispiel nimmt ‘`List->vector`’ eine Liste und liefert einen Vektor, dessen Elemente dieselben sind wie die der Liste.

2 Schreibkonventionen

Dieser Abschnitt berichtet informell über einige der beim Schreiben von Scheme-Programmen benutzten Schreibkonventionen. Eine formale Syntax von Scheme finden Sie im Abschnitt Abschnitt 7.1 [Formale Syntax], Seite 82.

Groß- und kleingeschriebene Formen eines Buchstabens werden nie unterschieden außer innerhalb von Zeichen- und Zeichenketten-Konstanten. Zum Beispiel ist '`Foo`' derselbe Bezeichner wie '`FOO`' und `#x1AB` ist dieselbe Zahl wie `#X1ab`.

2.1 Bezeichner

Die meisten Bezeichner, die in anderen Programmiersprachen zugelassen sind, sind auch in Scheme zulässig. Die genauen Regeln für das Bilden von Bezeichnern unterscheiden sich je nach Scheme-Implementierung, aber in allen Implementierungen ist eine Folge von Zeichen, Ziffern und „erweiterten Buchstabenzeichen“, die mit einem Zeichen beginnt, das keine Zahl beginnen kann, ein Bezeichner. Außerdem sind `+`, `-` und `...` auch Bezeichner. Hier sind einige Beispiele von Bezeichnern:

```
lambda                  q
list->vector            soup
+                       V17a
<=?                     a34kTMNs
Das-Wort-Rekursion-hat-viele-Bedeutungen
```

Erweiterte Buchstabenzeichen dürfen innerhalb von Bezeichnern benutzt werden, als wären sie Buchstaben. Die folgenden Zeichen sind erweiterte Buchstabenzeichen:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

Siehe den Abschnitt Abschnitt 7.1.1 [Lexikalische Struktur], Seite 82, für eine formale Syntax von Bezeichnern.

Bezeichner haben in Scheme-Programmen zwei Zwecke:

- Jeder Bezeichner darf als Variable oder als syntaktisches Schlüsselwort benutzt werden (siehe die Abschnitte siehe Abschnitt 3.1 [Variable; syntaktische Schlüsselwörter; und Regionen], Seite 11, und siehe Abschnitt 4.3 [Makros], Seite 28).
- Wenn ein Bezeichner als Literal oder innerhalb eines Literals auftaucht (siehe den Abschnitt siehe Abschnitt 4.1.2 [Literale Ausdrücke], Seite 17), wird er benutzt, um ein *Symbol* zu bezeichnen (siehe den Abschnitt siehe Abschnitt 6.3.3 [Symbole], Seite 60).

2.2 Leerraum und Kommentare

Leerraum-Zeichen sind Leerzeichen und Zeilenvorschübe. (Implementierungen bieten typischerweise zusätzliche Leerraum-Zeichen wie Tabulatorzeichen oder Seitenvorschübe.) Leerraum wird zur besseren Lesbarkeit benutzt und um Tokens voneinander zu trennen, wobei ein Token eine unteilbare lexikalische Einheit wie zum Beispiel ein Bezeichner oder eine Zahl ist; dort hat der Leerraum sonst keine Auswirkung. Leerraum darf zwischen jeglichen zwei

Tokens auftreten, aber nicht innerhalb eines Tokens. Leerraum darf auch innerhalb einer Zeichenkette auftreten, wo er aber Auswirkungen hat.

Ein Semikolon (`;`) zeigt den Anfang eines Kommentars an. Der Kommentar geht weiter bis zum Ende der Zeile, auf der das Semikolon steht. Kommentare sind gegenüber Scheme unsichtbar, aber das Ende der Zeile ist als Leerraum sichtbar. Dies verhindert, dass ein Kommentar in der Mitte eines Bezeichners oder einer Zahl steht.

```
;;; Die FACT-Prozedur berechnet die Fakultät
;;; einer nicht negativen ganzen Zahl.
(define fact
  (lambda (n)
    (if (= n 0)
        1        ;Rekursionsanfang: liefert 1
        (* n (fact (- n 1))))))
```

2.3 Andere Notationen

Für eine Beschreibung der Notationen, die für Zahlen benutzt werden, siehe den Abschnitt Abschnitt 6.2 [Zahlen], Seite 41.

`. + -` Diese werden in Zahlen benutzt und können auch sonst irgendwo in einem Bezeichner auftauchen, außer als deren erstes Zeichen. Ein abgegrenztes Plus- oder Minuszeichen für sich alleine ist auch ein Bezeichner. Ein abgegrenzter Punkt (der nicht innerhalb einer Zahl oder eines Bezeichners auftritt) wird in der Notation für Paare benutzt (Abschnitt siehe Abschnitt 6.3.2 [Paare und Listen], Seite 54) und um den Rest-Parameter in einer formalen Parameterliste anzuzeigen (Abschnitt siehe Abschnitt 4.1.4 [Prozeduren], Seite 19). Eine abgegrenzte Folge von drei Punkten nacheinander ist auch ein Bezeichner.

`( )` Runde Klammern werden zur Gruppierung und zum Kennzeichnen von Listen (siehe den Abschnitt siehe Abschnitt 6.3.2 [Paare und Listen], Seite 54).

`'` Ein halbes Anführungszeichen wird benutzt, um literale Daten anzuzeigen (Abschnitt siehe Abschnitt 4.1.2 [Literale Ausdrücke], Seite 17).

`` ` `` Das umgekehrte Hochkomma (oft englisch „Backquote“ genannt) wird benutzt, um fast-konstante Daten anzuzeigen (Abschnitt siehe Abschnitt 4.2.6 [Quasimaskierung], Seite 27).

`, ,@` Das Kommazeichen und die Folge aus Komma und At-Zeichen werden zusammen mit dem umgekehrten Hochkomma (Backquote) benutzt (Abschnitt siehe Abschnitt 4.2.6 [Quasimaskierung], Seite 27).

`"` Das doppelte Anführungszeichen wird zum Abgrenzen von Zeichenketten benutzt (Abschnitt siehe Abschnitt 6.3.5 [Zeichenketten], Seite 64).

`\` Der Backslash wird in der Syntax für Zeichen-Konstante benutzt (Abschnitt siehe Abschnitt 6.3.4 [Zeichen], Seite 62) und als ein Maskierungszeichen innerhalb von Zeichenketten-Konstanten (Abschnitt siehe Abschnitt 6.3.5 [Zeichenketten], Seite 64).

`[ ] { } |`
: Linke und rechte eckige und geschweifte Klammern sowie der vertikale Strich sind reserviert für mögliche spätere Erweiterungen der Sprache.

`#`
: Das Rautezeichen wird für eine Vielfalt von Zwecken benutzt, je nachdem, welches Zeichen direkt darauf folgt:

`#t #f`
: Dies sind die booleschen Konstanten (Abschnitt siehe Abschnitt 6.3.1 [Boolesche Werte], Seite 53).

`#\`
: Dies leitet eine Zeichen-Konstante ein (Abschnitt siehe Abschnitt 6.3.4 [Zeichen], Seite 62).

`#(`
: Dies leitet eine Vektor-Konstante ein (Abschnitt siehe Abschnitt 6.3.6 [Vektoren], Seite 66). Vektor-Konstante werden beendet mir `)` .

`#e #i #b #o #d #x`
: Diese werden zur Notation von Zahlen benutzt (Abschnitt siehe Abschnitt 6.2.4 [Syntax numerischer Konstanter], Seite 44).

3 Grundkonzepte

3.1 Variable; syntaktische Schlüsselwörter; und Regionen

Ein Bezeichner darf einen Syntaxtyp benennen oder er darf eine Stelle benennen, an der ein Wert gespeichert werden kann. Ein Bezeichner, der einen Syntaxtyp benennt, heißt ein *syntaktisches Schlüsselwort* und wird als an diese Syntax *gebunden* bezeichnet. Ein Bezeichner, der eine Stelle benennt, heißt eine *Variable* und wird als an diese Stelle *gebunden* bezeichnet. Die Menge aller sichtbaren Bindungen, die an einem Punkt eines Programms gelten, sind bekannt als die an diesem Punkt geltende *Umgebung*. Der Wert, der an der Stelle, an die die Variable gebunden ist, gespeichert ist, heißt der Wert der Variablen. Manchmal wird, die Terminologie missbrauchend, gesagt, die Variable benenne den Wert oder sei an den Wert gebunden. Das ist nicht ganz zutreffend, aber diese Praxis führt selten zu Verwirrung.

Bestimmte Ausdruckstypen werden benutzt, um neue Arten von Syntax zu erzeugen und um syntaktische Schlüsselwörter an diese neuen Syntaxen zu binden, während andere Ausdruckstypen neue Stellen erzeugen und Variable an diese Stellen binden. Diese Ausdruckstypen werden *Bindungskonstrukte* genannt.

Jene, die syntaktische Schlüsselwörter binden, werden im Abschnitt Abschnitt 4.3 [Makros], Seite 28, aufgeführt. Das grundlegendste Variablenbindungskonstrukt ist der '`Lambda`'-Ausdruck, weil alle anderen Variablenbindungskonstrukte als '`Lambda`'-Ausdrücke beschrieben werden können. Die anderen Variablenbindungskonstrukte sind die Ausdrücke '`Let`', '`Let*`', '`Letrec`' und '`Do`' (siehe die Abschnitte siehe Abschnitt 4.1.4 [Prozeduren], Seite 19, siehe Abschnitt 4.2.2 [Bindungskonstrukte], Seite 23, und siehe Abschnitt 4.2.4 [Iteration], Seite 26).

Wie Algol und Pascal und anders als die meisten anderen Dialekte von Lisp außer Common Lisp ist Scheme eine Sprache mit statischen Geltungsbereichen und einer Block-Struktur. Für jeden Ort, wo ein Bezeichner in einem Programm gebunden wird, gibt es eine zugehörige *Region* des Programmtexts, in der die Bindung sichtbar ist. Die Region wird durch die jeweiligen Bindungskonstrukte bestimmt, die die Bindung herstellen; wenn die Bindung zum Beispiel durch einen '`Lambda`'-Ausdruck hergestellt wird, ist ihre Region der gesamte '`Lambda`'-Ausdruck. Jede Erwähnung eines Bezeichners bezieht sich auf diejenige Bindung dieses Bezeichners, die die am weitesten innen liegende diese Erwähnung enthaltende Region hergestellt hat. Wenn keine der Regionen, in denen diese Benutzung des Bezeichners vorkommt, eine Bindung für diesen Bezeichner enthält, dann bezieht sich diese Benutzung auf die Bindung der Variablen in der Umgebung auf oberster Ebene, falls vorhanden (siehe die Kapitel siehe Kapitel 4 [Ausdrücke], Seite 17, und siehe Kapitel 6 [Standardprozeduren], Seite 37); ist keine solche Bindung des Bezeichners vorhanden, nennt man ihn *ungebunden*.

3.2 Typfremdheit

Kein Objekt erfüllt mehr als eines der folgenden Prädikate:

```
boolean?        pair?
symbol?         number?
```

```
char?             string?
vector?           port?
procedure?
```

Diese Prädikate definieren die Typen *Boolean* (boolescher Typ), *Pair* (Paar), *Symbol* (Symbol), *Number* (Zahl), *Char* (oder *Character*, deutsch Zeichen), *String* (Zeichenkette), *Vector* (Vektor), *Port* (Port) und *Procedure* (Prozedur). Die leere Liste ist ein besonderes Objekt und hat ihren eigenen Typ; sie erfüllt keines der obigen Prädikate.

Obwohl es einen gesonderten booleschen Typ gibt, kann jeder Scheme-Wert als ein boolescher Wert zum Überprüfen einer Bedingung benutzt werden. Wie im Abschnitt Abschnitt 6.3.1 [Boolesche Werte], Seite 53, erklärt wird, gelten alle Werte beim Überprüfen als wahr außer `#f`. Dieser Bericht benutzt das Wort „wahr“ für einen beliebigen Scheme-Wert außer `#f` und der Bericht benutzt das Wort „falsch“ für `#f`.

3.3 Externe Darstellungen

Ein wichtiges Konzept in Scheme (und Lisp) ist das der *externen Darstellung* eines Objekts als eine Folge von Zeichen. Zum Beispiel ist eine externe Darstellung der ganzen Zahl 28 die Zeichenfolge „`28`“ und eine externe Darstellung einer Liste bestehend aus den ganzen Zahlen 8 und 13 ist die Zeichenfolge „`(8 13)`“.

Die externe Darstellung eines Objekts muss nicht eindeutig sein. Die ganze Zahl 28 hat auch die Darstellungen „`#e28.000`“ und „`#x1c`“, die Liste im vorherigen Absatz hat auch die Darstellungen „`( 08 13 )`“ und „`(8 . (13 . ()))`“ (siehe den Abschnitt siehe Abschnitt 6.3.2 [Paare und Listen], Seite 54).

Viele Objekte haben Standarddarstellungen als externe Darstellung, aber manche, wie Prozeduren, haben keine Standarddarstellungen (auch wenn bestimmte Implementierungen für sie Darstellungen festlegen dürfen).

Eine externe Darstellung darf in ein Programm geschrieben werden, um das zugehörige Objekt zu erhalten (siehe ‘`Quote`’, Abschnitt siehe Abschnitt 4.1.2 [Literale Ausdrücke], Seite 17).

Externe Darstellungen können auch zur Eingabe und Ausgabe benutzt werden. Die Prozedur ‘`Read`’ (Abschnitt siehe Abschnitt 6.6.2 [Eingabe], Seite 78) analysiert externe Darstellungen grammatikalisch und die Prozedur ‘`Write`’ (Abschnitt siehe Abschnitt 6.6.3 [Ausgabe], Seite 80) erzeugt solche. Zusammen bieten sie ein elegantes und mächtiges Mittel zur Ein- und Ausgabe.

Beachten Sie, dass die Zeichenfolge „`(+ 2 6)`“ *keine* externe Darstellung für die ganze Zahl 8 ist, wenn sie auch ein Ausdruck *ist*, der zu der ganzen Zahl 8 ausgewertet wird; stattdessen ist sie eine externe Darstellung einer dreielementigen Liste, deren Elemente das Symbol `+` und die ganzen Zahlen 2 und 6 sind. Die Syntax von Scheme hat die Eigenschaft, dass eine beliebige Zeichenfolge, die ein Ausdruck ist, auch die externe Darstellung irgendeines Objekts ist. Dies kann zu Verwirrung führen, da es ohne Kontext nicht offensichtlich sein kann, ob eine bestimmte Zeichenfolge Daten oder Programm bezeichnen soll, es ist aber auch eine Quelle der Macht, da so das Schreiben von Programmen wie Interpretierern und Übersetzern erleichtert wird, die Programme als Daten behandeln (oder umgekehrt).

Die Syntax externer Darstellungen verschiedener Arten von Objekten begleitet die Beschreibungen der elementaren Operationen, um die Objekte zu manipulieren, welche in den

jeweiligen Abschnitten des Kapitels Kapitel 6 [Standardprozeduren], Seite 37, zu finden sind.

3.4 Speichermodell

Variable und Objekte wie Paare, Vektoren und Zeichenketten bezeichnen implizit Speicherstellen oder Folgen von Speicherstellen. Zum Beispiel bezeichnet eine Zeichenkette so viele Stellen, wie es Zeichen in der Zeichenkette gibt. (Diese Stellen müssen keinem ganzen Maschinenwort entsprechen.) Ein neuer Wert darf mit der Prozedur `String-set!` in einer dieser Stellen gespeichert werden, aber die Zeichenkette bezeichnet dann noch immer dieselben Stellen wie vorher.

Ein von einer Stelle geholtes Objekt, durch Variablenreferenz oder durch eine Prozedur wie '`Car`', '`Vector-ref`' oder '`String-ref`', ist im Sinne von `Eqv?` gleichwertig (Abschnitt siehe Abschnitt 6.1 [Äquivalenzprädikate], Seite 37) zum Objekt, was vor dem Holen zuletzt an der Stelle gespeichert war.

Jede Stelle wird markiert, um anzuzeigen, ob sie in Benutzung ist. Keine Variable und kein Objekt bezieht sich je auf eine Stelle, die nicht in Benutzung ist. Wann immer dieser Bericht davon spricht, dass Speicher für eine Variable oder ein Objekt zugeteilt wird, ist damit gemeint, dass eine entsprechende Anzahl von Stellen aus der Menge unbenutzer Stellen gewählt wird und die ausgewählten Stellen markiert werden, um anzuzeigen, dass sie nun benutzt sind, bevor die Variable oder das Objekt dazu gebracht wird, diese Stellen zu bezeichnen.

In vielen Systemen ist es wünschenswert, dass sich Konstante (d.h. die Werte von literalen Ausdrücken) in Nur-Lese-Speicher befinden. Um dies auszudrücken, ist es gelegen sich vorzustellen, dass jedes Objekt, welches Stellen bezeichnet, mit einer Markierung assoziiert ist, die angibt, ob das Objekt veränderlich oder unveränderlich ist. In solchen Systemen sind literale Konstante und die Zeichenketten, die `symbol->string` liefert, unveränderliche Objekte, während alle Objekte, die durch andere in diesem Bericht angegebene Prozeduren erzeugt wurden, veränderlich sind. Es ist ein Fehler zu versuchen, einen neuen Wert an eine Stelle zu speichern, die von einem unveränderlichen Objekt bezeichnet wird.

3.5 Echte Endrekursion

Implementierungen von Scheme müssen *echt endrekursiv.* sein. Prozeduraufrufe, die in den unten beschriebenen bestimmten syntaktischen Kontexten vorkommen, sind ‚endständige Aufrufe'. Eine Scheme-Implementierung ist echt endrekursiv, wenn sie eine unbeschränkte Anzahl von aktiven endständigen Aufrufen unterstützt. Ein Aufruf ist *aktiv*, wenn die aufgerufene Prozedur noch immer einen Wert liefern kann. Beachten Sie, dass dies alle Aufrufe einschließt, die entweder durch die aktuelle Fortsetzung oder durch zuvor mit '`Call-with-current-continuation`' gefangene Fortsetzungen, wenn sie später aufgerufen werden, noch einen Wert liefern können. Ohne gefangene Fortsetzungen könnten Aufrufe höchstens einen Wert liefern und aktive Aufrufe wären diejenigen, die noch keinen Wert geliefert haben. Eine formale Definition von echter Endrekursion ist in [propertailrecursion] zu finden.

Begründung:

Intuitiv wird kein Speicher für einen aktiven endständigen Aufruf benötigt, weil die Fortsetzung, die im endständigen Aufruf benutzt wird, dieselbe Semantik

> hat wie die Fortsetzung, die der den Aufruf enthaltenden Prozedur übergeben wurde. Obwohl eine nicht richtige Implementierung in diesem Aufruf eine neue Fortsetzung benutzen könnte, würde ein Liefern des Wertes des Aufrufs an diese neue Fortsetzung ein sofortiges Liefern des Wertes an die der Prozedur übergebene Fortsetzung nach sich ziehen. Eine echt endrekursive Implementierung liefert den Wert direkt an diese übergebene Fortsetzung.
>
> Echte Endrekursion war eine der zentralen Ideen in Steeles und Sussmans ursprünglicher Fassung von Scheme. Ihr erster Scheme-Interpretierer implementierte sowohl Funktionen als auch Akteure. Programmfluss wurde über Akteure ausgedrückt, welche sich darin von Funktionen unterschieden, dass sie ihre Ergebnisse einem anderen Akteur übergaben statt an den Aufrufer. In der Terminologie dieses Abschnitts endete jeder Akteur mit einem endständigen Aufruf an einen anderen Akteur.
>
> Steele und Sussman beobachteten später, dass der Code in ihrem Interpretierer, der mit Akteuren umging, identisch war mit dem für Funktionen und es daher keinen Grund dafür gab, beide in die Sprache aufzunehmen.

Ein *endständiger Aufruf* ist ein Prozeduraufruf, der *endständig* auftritt. Endständigkeit ist induktiv definiert. Beachten Sie, dass Endständigkeit immer in Bezug auf einen bestimmten Lambda-Ausdruck bestimmt wird.

- Der letzte Ausdruck im Rumpf eines Lambda-Ausdrucks, unten dargestellt als <endständiger Ausdruck>, tritt endständig auf.

```
(lambda <Formale>
  <Definition>* <Ausdruck>* <endständiger Ausdruck>)
```

- Wenn einer der folgenden Ausdrücke endständig ist, sind die als <endständiger Ausdruck> dargestellten Unterausdrücke endständig. Diese wurden über die Regeln der im Kapitel Kapitel 7 [Formale Syntax und Semantik], Seite 82, angegebenen Grammatik abgeleitet, durch Ersetzung der Vorkommen von <Ausdruck> durch <endständiger Ausdruck>. Hier sind nur die Regeln dargestellt, die Endständigkeit enthalten.

```
(if <Ausdruck> <endständiger Ausdruck> <endständiger Ausdruck>)
(if <Ausdruck> <endständiger Ausdruck>)

(cond <cond-Klausel>+)
(cond <cond-Klausel>* (else <endständige Folge>))

(case <Ausdruck>
  <case-Klausel>+)
(case <Ausdruck>
  <case-Klausel>*
  (else <endständige Folge>))

(and <Ausdruck>* <endständiger Ausdruck>)
(or <Ausdruck>* <endständiger Ausdruck>)
```

```
(let (<Bindungsspezifikation>*) <endständiger Rumpf>)
(let <Variable> (<Bindungsspezifikation>*) <endständiger Rumpf>)
(let* (<Bindungsspezifikation>*) <endständiger Rumpf>)
(letrec (<Bindungsspezifikation>*) <endständiger Rumpf>)

(let-syntax (<Syntaxspezifikation>*) <endständiger Rumpf>)
(letrec-syntax (<Syntaxspezifikation>*) <endständiger Rumpf>)

(begin <endständige Folge>)

(do (<Iterationsspezifikation>*)
    (<Test> <endständige Folge>)
  <Ausdruck>*)
```

where

```
<cond-Klausel> --> (<Test> <endständige Folge>)
<case-Klausel> --> ((<Datenelement>*) <endständige Folge>)

<endständiger Rumpf> --> <Definition>* <endständige Folge>
<endständige Folge> --> <Ausdruck>* <endständiger Ausdruck>
```

- Wenn ein '`Cond`'-Ausdruck endständig auftritt und eine Klausel der Form '`(`<Ausdruck1> `=>` <Ausdruck2>`)`' enthält, dann ist der (implizite) Aufruf der Prozedur, die sich aus der Auswertung von <Ausdruck2> ergibt, endständig. <Ausdruck2> selbst ist nicht endständig.

Bestimmte eingebaute Prozeduren müssen auch endrekursive Aufrufe durchführen. Das erste an `apply` und an `call-with-current-continuation` übergebene Argument sowie das zweite an `call-with-values` übergebene Argument muss endrekursiv aufgerufen werden. Ebenso muss `eval` sein Argument so auswerten, als wäre es innerhalb der `eval`-Prozedur endständig.

Im folgenden Beispiel ist der einzige endrekursive Aufruf der Aufruf von '`f`'. Keiner der Aufrufe von '`g`' oder '`h`' sind endrekursive Aufrufe. Die Referenz auf '`x`' ist endständig, aber sie ist kein Aufruf und daher nicht endrekursiv.

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (and (g) (f))))
```

Anmerkung: Implementierungen dürfen, sie müssen aber nicht, erkennen, dass einige der nicht endrekursiven Aufrufe, wie oben der Aufruf von '`h`', ausgewer-

tet werden können, als wären sie endrekursiv. Im obigen Beispiel könnte der ‘`Let`’-Ausdruck zu einem endrekursiven Aufruf von ‘`h`’ übersetzt werden. (Die Möglichkeit, dass ‘`h`’ eine unerwartete Anzahl von Werten liefert, kann ignoriert werden, weil dann die Wirkung des ‘`Let`’ ausdrücklich unbestimmt und implementierungsabhängig ist.)

4 Ausdrücke

Ausdruckstypen werden eingeteilt in *elementare* oder in *abgeleitete* Ausdruckstypen. Elementare Ausdruckstypen schließen Variable und Prozeduraufrufe ein. Abgeleitete Ausdrückstypen sind nicht semantisch elementar, sondern können stattdessen als Makros definiert werden. Mit Ausnahme von '`Quasiquote`' zur Quasimaskierung (auch bekannt als „Quasiquotierung"), dessen Makrodefinition komplex ist, werden die abgeleiteten Ausdrücke zu den Bibliotheksfunktionalitäten gezählt. Angemessene Definitionen werden im Abschnitt Abschnitt 4.2 [Abgeleitete Ausdruckstypen], Seite 21, vorgestellt.

4.1 Elementare Ausdruckstypen

4.1.1 Variablenreferenzen

<Variable> [Syntax]

Ein Ausdruck, der aus einer Variablen (Abschnitt siehe Abschnitt 3.1 [Variable; syntaktische Schlüsselwörter; und Regionen], Seite 11) besteht, ist eine Variablenreferenz. Der Wert der Variablenreferenz ist der an der Stelle, an die die Variable gebunden ist, gespeicherte Wert. Es ist ein Fehler, eine ungebundene Variable zu referenzieren.

```
(define x 28)
x                                       ==>  28
```

4.1.2 Literale Ausdrücke

`quote` <Datenelement> [Syntax]
'<Datenelement> [Syntax]
<Konstante> [Syntax]

'`(quote` <Datenelement>`)`' wird ausgewertet zum <Datenelement>. Durch '`Quote`' wird das <Datenelement> *maskiert* (man sagt auch, es wird „quotiert"). <Datenelement> darf eine beliebige externe Darstellung eines Scheme-Objekts sein (siehe den Abschnitt siehe Abschnitt 3.3 [Externe Darstellungen], Seite 12). Diese Notation wird zum Einfügen literaler „wortwörtlicher" Konstanter in Scheme-Code benutzt.

```
(quote a)                               ==>  a
(quote #(a b c))                        ==>  #(a b c)
(quote (+ 1 2))                         ==>  (+ 1 2)
```

'`(quote` <Datenelement>`)`' darf abgekürzt werden als '<Datenelement>. Die beiden Notationen sind in allen Bezügen äquivalent.

```
'a                                      ==>  a
'#(a b c)                               ==>  #(a b c)
'()                                     ==>  ()
'(+ 1 2)                                ==>  (+ 1 2)
```

```
'(quote a)                              ==>  (quote a)
''a                                     ==>  (quote a)
```

Numerische Konstante, Zeichenkettenkonstante, Zeichenkonstante und boolesche Konstante werden „zu sich selbst“ ausgewertet; sie müssen nicht mit ‘`Quote`’ maskiert werden.

```
'"abc"                                  ==>  "abc"
"abc"                                   ==>  "abc"
'145932                                 ==>  145932
145932                                  ==>  145932
'#t                                     ==>  #t
#t                                      ==>  #t
```

Wie im Abschnitt Abschnitt 3.4 [Speichermodell], Seite 13, angemerkt ist es ein Fehler, eine Konstante (d.h. den Wert eines literalen Ausdrucks) mit einer Veränderungsprozedur wie ‘`Set-car!`’ oder ‘`String-set!`’ abzuändern.

4.1.3 Prozeduraufrufe

<Operator> <Operand1> . . ., [Syntax]

Ein Prozeduraufruf wird geschrieben, indem man einfach die Ausdrücke für die aufzurufende Prozedur und die ihr zu übergebenden Argumente mit runden Klammern umschließt. Die Operator- und Operand-Ausdrücke werden (in unbestimmter Reihenfolge) ausgewertet und der daraus hervorgehenden Prozedur werden die sich ergebenden Argumente übergeben.

```
(+ 3 4)                                 ==>  7
((if #f + *) 3 4)                       ==>  12
```

Eine Menge von Prozeduren sind schon als die Werte von Variablen in der Anfangsumgebung verfügbar; zum Beispiel sind die Prozeduren für die Addition und Multiplikation in obigen Beispielen die Werte der Variablen ‘`+`’ und ‘`*`’. Neue Prozeduren entstehen durch das Auswerten von Lambda-Ausdrücken (siehe den Abschnitt siehe Abschnitt 4.1.4 [Prozeduren], Seite 19).

Prozeduraufrufe dürfen eine beliebige Anzahl von Werten liefern (siehe `values` im Abschnitt siehe Abschnitt 6.4 [Programmflussfunktionalitäten], Seite 68). Mit Ausnahme von ‘`Values`’ liefern die in der Anfangsumgebung verfügbaren Prozeduren einen Wert oder, bei Prozeduren wie ‘`Apply`’, genau die Werte, die ein Aufruf eines ihrer Argumente liefert. .

Prozeduraufrufe werden auch *Kombinationen* genannt.

> *Anmerkung:* Anders als bei anderen Dialekten von Lisp ist die Auswertungsreihenfolge unbestimmt und die Ausdrücke für den Operator und

die Operanden werden immer nach denselben Auswertungsregeln ausgewertet.

Anmerkung: Obwohl die Auswertungsreihenfolge ansonsten unbestimmt ist, ist die Wirkung einer beliebigen nebenläufigen Auswertung von Operator- und Operandausdrücken so weit eingeschränkt, dass sie mit der Wirkung bei einer beliebigen sequentiellen Auswertungsreihenfolge übereinstimmen muss. Die Auswertungsreihenfolge darf für jeden Prozeduraufruf unterschiedlich gewählt werden.

Anmerkung: In vielen Dialekten von Lisp ist die leere Kombination `()` ein zulässiger Ausdruck. In Scheme müssen Kombinationen mindestens einen Unterausdruck haben, so dass `()` kein syntaktisch gültiger Ausdruck ist.

4.1.4 Prozeduren

`lambda` <Formale> <Rumpf> [Syntax]

Syntax: <Formale> sollte eine Liste formaler Argumente sein, wie sie unten beschrieben ist, und <Rumpf> sollte eine Folge von einem oder mehr Ausdrücken sein.

Semantik: Ein Lambda-Ausdruck wird immer zu einer Prozedur ausgewertet. Die Umgebung, die bestand, während der Lambda-Ausdruck ausgewertet wurde, wird als Teil der Prozedur gespeichert. Wird die Prozedur später mit tatsächlichen Argumenten aufgerufen, wird die Umgebung, in der der Lambda-Ausdruck ausgewertet wurde, erweitert, indem die Variablen in der Liste formaler Argumente an neue Stellen gebunden werden, dann werden die entssprechenden tatsächlichen Argumentwerte an diese Stellen gespeichert und die Ausdrücke im Rumpf des Lambda-Ausdrucks werden der Reihe nach in der erweiterten Umgebung ausgewertet. Das oder die Ergebnisse des letzten Ausdrucks im Rumpf wird bzw. werden als das oder die Ergebnisse des Prozeduraufrufs geliefert.

```
(lambda (x) (+ x x))                    ==>  eine Prozedur
((lambda (x) (+ x x)) 4)                ==>  8

(define reverse-subtract ; rückwärts subtrahieren
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)                 ==>  3

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                                ==>  10
```

<Formale> sollte eine der folgenden Formen haben:

- `(`<Variable1> `...,)`: Die Prozedur nimmt eine feste Anzahl von Argumenten; wenn die Prozedur aufgerufen wird, werden die Argumente in den Bindungen der entsprechenden Variablen gespeichert.
- <Variable>: Die Prozedur nimmt eine beliebige Anzahl von Argumenten entgegen; wird die Prozedur aufgerufen, wird die Folge aus den tatsächlichen Argumenten

in eine neu angeforderte Liste umgewandelt und diese Liste in der Bindung von <Variable> gespeichert.

- (<Variable1> ..., <Variable_n> . <Variable_n+1>): Wenn ein durch ein Leerzeichen abgetrennter Punkt vor der letzten Variablen steht, dann nimmt die Prozedur n oder mehr Argumente, wobei n die Anzahl formaler Argumente vor dem Punkt ist (es muss mindestens eines geben). Der Wert, der in der Bindung der letzten Variablen gespeichert wird, wird der einer neu angeforderten Liste derjenigen tatsächlichen Argumente sein, die nach Verteilung aller anderen tatsächlichen Argumente auf die anderen formalen Argumente übrig bleiben.

Es ist ein Fehler, wenn eine <Variable> mehr als einmal in <Formale> vorkommt.

```
((lambda x x) 3 4 5 6)                  ==>  (3 4 5 6)
((lambda (x y . z) z)
 3 4 5 6)                               ==>  (5 6)
```

Jede Prozedur, die durch das Auswerten eines Lambda-Ausdrucks entsteht, wird (konzeptionell) mit einer Speicherstelle beschriftet, damit `eqv?` und `eq?` mit Prozeduren umgehen können (siehe den Abschnitt siehe Abschnitt 6.1 [Äquivalenzprädikate], Seite 37).

4.1.5 Bedingungen

`if` <Test> <Folgerung> <Alternative> [Syntax]

`if` <Test> <Folgerung> [Syntax]

Syntax: <Test>, <Folgerung> und <Alternative> dürfen beliebige Ausdrücke sein.

Semantik: Ein '`If`'-Ausdruck wird wie folgt ausgewertet: Zuerst wird <Test> ausgewertet. Wenn er einen wahren Wert liefert (siehe den Abschnitt siehe Abschnitt 6.3.1 [Boolesche Werte], Seite 53), dann wird <Folgerung> ausgewertet und ihr(e) Wert(e) geliefert. Ansonsten wird <Alternative> ausgewertet und ihr(e) Wert(e) geliefert. Wenn <Test> einen falschen Wert liefert und keine <Alternative> angegeben wurde, ist das Ergebnis des Ausdrucks unbestimmt.

```
(if (> 3 2) 'ja 'nein)                  ==>  ja
(if (> 2 3) 'ja 'nein)                  ==>  nein
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                            ==>  1
```

4.1.6 Zuweisungen

`set!` <Variable> <Ausdruck> [Syntax]

<Ausdruck> wird ausgewertet und der sich ergebende Wert an der Stelle gespeichert, an die <Variable> gebunden ist. <Variable> muss entweder in einer Region gebunden sein, die den '`Set!`'-Ausdruck einschließt, oder auf oberster Ebene gebunden sein. Das Ergebnis des '`Set!`'-Ausdrucks ist unbestimmt.

```
(define x 2)
```

```
(+ x 1)                                 ==>  3
(set! x 4)                              ==>  unbestimmt
(+ x 1)                                 ==>  5
```

4.2 Abgeleitete Ausdruckstypen

Die Konstrukte in diesem Abschnitt sind hygienisch, wie im Abschnitt Abschnitt 4.3 [Makros], Seite 28, diskuttiert. Als Referenz sind im Abschnitt Abschnitt 4.2 [Abgeleitete Ausdruckstypen], Seite 21, Makrodefinitionen zu finden, die die meisten hier beschriebenen Konstrukte in elementare Konstrukte, die im vorherigen Abschnitt beschrieben wurden, umwandeln.

4.2.1 Bedingungen

`cond` <*Klausel1*> <*Klausel2*> ..., [Bibliothekssyntax]

Syntax: Jede <Klausel> sollte entweder folgende Form haben:
`(<Test> <Ausdruck1> ...,)`

wobei <Test> ein beliebiger Ausdruck ist, oder die <Klausel> hat alternativ diese Form:
`(<Test> => <Ausdruck>)`

Die letzte <Klausel> darf auch eine „else-Klausel“ sein, welche die folgende Form hat: `(else <Ausdruck1> <Ausdruck2> ...,)`.

Semantik: Ein ‘`Cond`’-Ausdruck wird ausgewertet, indem man der Reihe nach die <Test>-Ausdrücke der aufeinanderfolgenden <Klausel>n auswertet, bis eine von ihnen zu einem wahren Wert ausgewertet wird (siehe den Abschnitt siehe Abschnitt 6.3.1 [Boolesche Werte], Seite 53). Sobald ein <Test> zu einem wahren Wert ausgewertet wird, werden die verbleibenden in seiner <Klausel> stehenden Vorkommen eines <Ausdruck>s der Reihe nach ausgewertet und das Ergebnis bzw. die Ergebnisse des letzten <Ausdruck>s in der <Klausel> wird(werden) als Ergebnis(se) des gesamten ‘`Cond`’-Ausdrucks geliefert. Wenn die gewählte <Klausel> nur den <Test> und keine Vorkommen von <Ausdruck> enthält, dann wird der Wert vom <Test> als Ergebnis geliefert. Wenn die gewählte <Klausel> die alternative Form mit `=>` benutzt, dann wird der <Ausdruck> ausgewertet. Sein Wert muss eine Prozedur sein, die ein Argument annimmt; diese Prozedur wird dann mit dem Wert vom <Test> aufgerufen und der von dieser Prozedur gelieferte Wert bzw. die von ihr gelieferten Werte werden dann auch vom ‘`Cond`’-Ausdruck geliefert. Wenn jeder <Test> zu falschen Werten ausgewertet wird und es keine else-Klausel gibt, dann ist das Ergebnis des bedingten Ausdrucks unbestimmt; gibt es eine else-Klausel, dann werden seine <Ausdruck>-Vorkommen ausgewertet und der Wert bzw. die Werte des letzten Ausdrucks geliefert.

Das folgende Beispiel nimmt an, dass die benutzte Scheme-Implementierung die Umlaute und das scharfe S (ß) auch als Buchstabenzeichen ansieht, als Erweiterung zur in diesem Bericht spezifizierten Syntax:

```
(cond ((> 3 2) 'größer)
      ((< 3 2) 'kleiner))              ==>  größer

(cond ((> 3 3) 'größer)
      ((< 3 3) 'kleiner)
      (else 'gleich))                  ==>  gleich

(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))                       ==>  2
```

`case` <Schlüssel> <*Klausel1*> <*Klausel2*> . . . , [Bibliothekssyntax]

Syntax: <Schlüssel> darf ein beliebiger Ausdruck sein. Jede <Klausel> sollte die folgende Form haben:

((<Datenelement1> ...,) <Ausdruck1> <Ausdruck2> ...,),

wobei jedes <Datenelement> eine externe Darstellung irgendeines Objekts ist. Alle <Datenelement>e müssen verschieden sein. Die letzte <Klausel> darf eine „else-Klausel“ sein, welche die folgende Form hat:

(else <Ausdruck1> <Ausdruck2> ...,).

Semantik: Ein ‘`Case`’-Ausdruck wird wie folgt ausgewertet. <Schlüssel> wird ausgewertet und sein Ergebnis mit jedem <Datenelement> verglichen. Wenn das Ergebnis der Auswertung von <Schlüssel> gleichwertig ist (im Sinne von ‘`Eqv?`’; siehe den Abschnitt siehe Abschnitt 6.1 [Äquivalenzprädikate], Seite 37) mit einem <Datenelement>, dann werden die Ausdrücke in der entsprechenden <Klausel> von links nach rechts ausgewertet und das Ergebnis bzw. die Ergebnisse des letzten Ausdrucks in der <Klausel> werden als Ergebnis(se) des ‘`Case`’-Ausdrucks geliefert. Wenn die Auswertung von <Schlüssel> zu einem anderen Ergebnis als jedes <Datenelement> führt, dann werden, wenn es eine else-Klausel gibt, ihre Ausdrücke ausgewertet und das Ergebnis bzw. die Ergebnisse des letzten davon ist das Ergebnis bzw. sind die Ergebnisse des ‘`Case`’-Ausdrucks; andernfalls ist das Ergebnis des ‘`Case`’-Ausdrucks unbestimmt.

```
(case (* 2 3)
  ((2 3 5 7) 'prim)
  ((1 4 6 8 9) 'zusammengesetzt))      ==>  zusammengesetzt
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                            ==>  unbestimmt
(case (car '(c d))
  ((a e i o u) 'vokal)
```

```
((j) 'halbvokal)
(else 'konsonant))                    ==>  konsonant
```

`and` <*Test1*> ..., [Bibliothekssyntax]

Die <Test>-Ausdrücke werden von links nach rechts ausgewertet und der Wert des ersten Ausdrucks, der zu einem falschen Wert ausgewertet wird (siehe den Abschnitt siehe Abschnitt 6.3.1 [Boolesche Werte], Seite 53), wird geliefert. Jegliche verbleibenden Ausdrücke werden nicht ausgewertet. Wenn alle Ausdrücke zu wahren Werten ausgewertet werden, wird der Wert des letzten Ausdrucks geliefert. Gibt es keine Ausdrücke, so wird `#t` geliefert.

```
(and (= 2 2) (> 2 1))                 ==>  #t
(and (= 2 2) (< 2 1))                 ==>  #f
(and 1 2 'c '(f g))                   ==>  (f g)
(and)                                 ==>  #t
```

`or` <*Test1*> ..., [Bibliothekssyntax]

Die <Test>-Ausdrücke werden von links nach rechts ausgewertet und der Wert des ersten Ausdrucks, der zu einem wahren Wert ausgewertet wird (siehe den Abschnitt siehe Abschnitt 6.3.1 [Boolesche Werte], Seite 53), wird geliefert. Jegliche verbleibenden Ausdrücke werden nicht ausgewertet. Wenn alle Ausdrücke zu falschen Werten ausgewertet werden, wird der Wert des letzten Ausdrucks geliefert. Gibt es keine Ausdrücke, so wird `#f` geliefert.

```
(or (= 2 2) (> 2 1))                  ==>  #t
(or (= 2 2) (< 2 1))                  ==>  #t
(or #f #f #f)                         ==>  #f
(or (memq 'b '(a b c))
    (/ 3 0))                          ==>  (b c)
```

4.2.2 Bindungskonstrukte

Die drei Bindungskonstrukte '`Let`', '`Let*`' und '`Letrec`' verleihen Scheme eine Block-Struktur wie bei Algol 60. [Auf Deutsch heißt „Let ...“ soviel wie „Sei ...“.] Die Syntax der drei Konstrukte ist identisch, aber sie unterscheiden sich in den Regionen, die sie für ihre Variablenbindungen herstellen In einem '`Let`'-Ausdruck werden die Anfangswerte berechnet, bevor ihre Variablen gebunden werden; in einem '`Let*`'-Ausdruck werden die Bindungen und Auswertungen eine nach der anderen der Reihe nach durchgeführt, während in einem '`Letrec`'-Ausdruck alle Bindungen bereits gelten, während ihre Anfangswerte berechnet werden, wodurch wechselseitig rekursive Definitionen ermöglicht werden.

`let` <Bindungen> <Rumpf> [Bibliothekssyntax]

Syntax: <Bindungen> sollte die folgende Form haben:

((<Variable1> <Anfang1>) ...,),

wobei jeder <Anfang> ein Ausdruck ist und der <Rumpf> eine Folge von einem oder mehr Ausdrücken sein sollte. Es ist ein Fehler, wenn eine <Variable> mehr als einmal in der Liste der zu bindenden Variablen vorkommt.

Semantik: Jeder <Anfang> wird in der aktuellen Umgebung ausgewertet (in unbestimmter Reihenfolge), dann werden die <Variable>n an neue Stellen gebunden, die die Ergebnisse enthalten, woraufhin der <Rumpf> in der so erweiterten Umgebung ausgewertet wird und der Wert bzw. die Werte des letzten Ausdrucks im <Rumpf> geliefert wird(werden). Jede Bindung einer <Variable>n hat den <Rumpf> als ihre Region.

```
(let ((x 2) (y 3))
  (* x y))                              ==>  6

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))                           ==>  35
```

Siehe auch das '`Let`' mit Namen, Abschnitt Abschnitt 4.2.4 [Iteration], Seite 26.

`let*` <Bindungen> <Rumpf> [Bibliothekssyntax]

Syntax: <Bindungen> sollte die folgende Form haben:

`((`<Variable1> <Anfang1>`) ...,)`,

und der <Rumpf> sollte eine Folge von einem oder mehr Ausdrücken sein.

Semantik: '`Let*`' ist ähnlich wie '`Let`', aber die Bindungen werden der Reihe nach von links nach rechts durchgeführt und die Region einer Bindung, die durch '`(`<Variable> <Anfang>`)`' angezeigt wird, ist der Teil des '`Let*`'-Ausdrucks rechts von der Bindung. Dadurch wird die zweite Bindung in einer Umgebung ausgeführt, in der die erste Bindung bereits sichtbar ist, und so weiter.

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))                           ==>  70
```

`letrec` <Bindungen> <Rumpf> [Bibliothekssyntax]

Syntax: <Bindungen> sollte folgende Form haben:

`((`<Variable1> <Anfang1>`) ...,)`,

und <Rumpf> sollte eine Folge von einem oder mehreren Ausdrücken sein. Es ist ein Fehler, wenn eine <Variable> mehr als einmal in der Liste der zu bindenden Variablen vorkommt.

Semantik: Die <Variable>n werden an neue Stellen gebunden, die noch unfertige Werte enthalten, jeder <Anfang> wird (in unbestimmter Reihenfolge) ausgewertet, jeder Variablen das Ergebnis des zugehörigen <Anfang>s zugewiesen und der <Rumpf> in der daraus hervorgehenden Umgebung ausgewertet und der Wert bzw. die Werte des letzten Ausdrucks im <Rumpf> geliefert. Jede Bindung einer <Variable>n hat den gesamten '`Letrec`'-Ausdruck als ihre Region, was es ermöglicht, wechselseitig rekursive Prozeduren zu definieren.

```
(letrec ((even? ; gerade?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd? ; ungerade?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
                                        ==>  #t
```

Eine Einschränkung von '`Letrec`' ist sehr wichtig: es muss möglich sein, jeden <Anfang> auszuwerten, ohne dass ein Wert irgendeiner <Variable>n zugewiesen oder referenziert wird. Wird diese Einschränkung verletzt, ist es ein Fehler. Die Einschränkung ist notwendig, weil Scheme Argumente als Wertparameter und nicht als Namensparameter übergibt. In den häufigsten Nutzungen von '`Letrec`' ist jeder <Anfang> ein Lambda-Ausdruck und die Einschränkung gilt automatisch.

4.2.3 Sequenzierung

`begin` <*Ausdruck1*> <*Ausdruck2*> *. . .*, [Bibliothekssyntax]

Jeder <Ausdruck> wird der Reihe nach von links nach rechts ausgewertet und der Wert bzw. die Werte des letzten <Ausdruck>s geliefert. Dieser Ausdruckstyp wird für Nebenwirkungen wie Ein- und Ausgabe benutzt.

```
(define x 0)

(begin (set! x 5)
       (+ x 1))                         ==>  6

(begin (display "4 plus 1 ist gleich ")
       (display (+ 4 1)))               ==>  unbestimmt
          und gibt aus  4 plus 1 ist gleich 5
```

4.2.4 Iteration

`do` ((<Variable1> <Anfang1> <Schritt1>) . . .) (<Test> <Ausdruck> . . .) <Befehl> . . . [Bibliothekssyntax]

'`Do`' (deutsch „tu“) ist ein Iterationskonstrukt. Es gibt eine Menge von Variablen an, die zu binden sind, welchen Wert sie anfänglich haben sollen und wie sie in jeder Iteration aktualisiert werden sollen. Wird eine Terminierungsbedingung erfüllt, dann wird die Schleife nach der Auswertung jedes <Ausdruck>s verlassen.

'`Do`'-Ausdrücke werden wie folgt ausgewertet: Die <Anfang>sausdrücke werden ausgewertet (in einer unbestimmten Reihenfolge), die <Variable>n werden an neue Stellen gebunden, die Ergebnisse der <Anfang>sausdrücke in den Bindungen der <Variable>n gespeichert und dann beginnt die Iterationsphase.

Jede Iteration beginnt mit dem Auswerten vom <Test>; wenn das Ergebnis falsch ist (siehe den Abschnitt siehe Abschnitt 6.3.1 [Boolesche Werte], Seite 53), dann werden der Reihe nach die Ausdrücke des <Befehl>s für ihre Wirkung ausgewertet, die <Schritt>-Ausdrücke in einer unbestimmten Reihenfolge ausgewertet, die <Variable>n an neue Stellen gebunden, die Ergebnisse der <Schritt>e in den Bindungen der <Variable>n gespeichert und die nächste Iteration fängt an.

Wenn der <Test> zu einem wahren Wert ausgewertet wird, dann wird von rechts nach links jeder <Ausdruck> ausgewertet und der Wert bzw. die Werte des letzten <Ausdruck>s geliefert. Gibt es keinen <Ausdruck>, ist der Wert des '`Do`'-Ausdrucks unbestimmt.

Die Region der Bindung einer <Variable>n besteht aus dem gesamten '`Do`'-Ausdruck außer jedem <Anfang>. Es ist ein Fehler, wenn eine <Variable> mehr als einmal in der Liste der '`Do`'-Variablen vorkommt.

Ein <Schritt> darf weggelassen werden. In diesem Fall ist die Wirkung dieselbe, wie wenn '`(<Variable> <Anfang> <Variable>)`' statt '`(<Variable> <Anfang>)`' geschrieben worden wäre.

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))               ==>  #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (summe 0 (+ summe (car x))))
      ((null? x) summe)))              ==>  25
```

`let` <Variable> <Bindungen> <Rumpf> [Bibliothekssyntax]

„'`Let`' mit Namen“ ist eine Variante der Syntax von `Let`, die ein allgemeineres Schleifenkonstrukt als '`Do`' bietet und auch zum Ausdrücken von Rekursion benutzt werden kann. Es hat dieselbe Syntax und Semantik wie das herkömmliche '`Let`', außer dass <Variable> im <Rumpf> an eine Prozedur gebunden wird, deren formale Argumente die gebundenen Variablen sind und deren Rumpf der <Rumpf> ist. Daher kann

durch einen Aufruf der durch die <Variable> benannten Prozedur die Ausführung des <Rumpf>s wiederholt werden.

```
(let loop ((zahlen '(3 -2 1 6 -5))
           (nichtneg '())
           (neg '()))
  (cond ((null? zahlen) (list nichtneg neg))
        ((>= (car zahlen) 0)
         (loop (cdr zahlen)
               (cons (car zahlen) nichtneg)
               neg))
        ((< (car zahlen) 0)
         (loop (cdr zahlen)
               nichtneg
               (cons (car zahlen) neg)))))
          ==>  ((6 1 3) (-5 -2))
```

4.2.5 Verzögerte Auswertung

`delay` <Ausdruck> [Bibliothekssyntax]

Das '`Delay`'-Konstrukt (deutsch „verzögern“) wird zusammen mit der Prozedur `Force` (deutsch „erzwingen“) benutzt, um *verzögerte Auswertung* (Lazy evaluation) oder *Bedarfsparameter* (Call by need) zu implementieren. `(delay` <Ausdruck>`)` liefert ein Objekt, was *Versprechen* (Promise) genannt wird, welches zu einem zukünftigen Zeitpunkt (durch die '`Force`'-Prozedur) um die Auswertung des <Ausdruck>s gebeten werden kann und dann den sich daraus ergebenden Wert liefert. Wenn der <Ausdruck> mehrere Werte liefert, ist die Wirkung davon unbestimmt.

Siehe die Beschreibung von '`Force`' (im Abschnitt siehe Abschnitt 6.4 [Programmflussfunktionalitäten], Seite 68) für eine vollständigere Beschreibung von '`Delay`'.

4.2.6 Quasimaskierung

`quasiquote` <qq-Schablone> [Syntax]

`‘`<qq-Schablone> [Syntax]

„Backquote“- oder „Quasiquote“-Ausdrücke zur Quasimaskierung (auch bekannt als „Quasiquotierung“) nützen beim Erstellen einer Liste oder Vektorstruktur, wenn die meisten aber nicht alle Bestandteile der gewünschten Struktur im Voraus bekannt sind. (Ein Backquote-Zeichen wird auf deutsch auch als umgekehrtes Hochkomma bezeichnet.) Wenn kein Komma innerhalb der <qq-Schablone> vorkommt, ist das Ergebnis der Auswertung von `‘`<qq-Schablone> äquivalent zum Ergebnis der Auswertung von `’`<qq-Schablone>. Kommt jedoch ein Komma innerhalb der <qq-Schablone> vor, so wird der auf das Komma folgende Ausdruck ausgewertet („demaskiert“) und sein Ergebnis in die Struktur an Stelle des Kommas und des Ausdrucks eingefügt. Kommt ein Komma direkt gefolgt von einem At-Zeichen (@) vor, muss der folgende Ausdruck zu einer Liste ausgewertet werden; die öffnenden und schließenden runden Klammern der Liste werden dann „abgetrennt“ und die Elemente der Liste an Stelle

der Komma-At-Zeichen-Ausdrucksfolge eingefügt. Ein Komma-At-Zeichen sollte nur innerhalb einer Listen- oder Vektor-<qq-Schablone> vorkommen.

```
‘(list ,(+ 1 2) 4)                    ==>  (list 3 4)
(let ((name ’a)) ‘(list ,name ’,name))
          ==>  (list a (quote a))
‘(a ,(+ 1 2) ,@(map abs ’(4 -5 6)) b)
          ==>  (a 3 4 5 6 b)
‘((‘foo’ ,(- 10 3)) ,@(cdr ’(c)) . ,(car ’(cons)))
          ==>  ((foo 7) . cons)
‘#(10 5 ,(sqrt 4) ,@(map sqrt ’(16 9)) 8)
          ==>  #(10 5 2 4 3 8)
```

Quasimaskierungsformen dürfen verschachtelt werden. Ersetzungen werden nur für demaskierte Bestandteile durchgeführt, die auf derselben Verschachtelungsstufe wie das äußerste umgekehrte Hochkomma (Backquote-Zeichen) stehen. Die Verschachtelungsstufe erhöht sich um eins für jede folgende Quasimaskierung und sinkt um eins in jeder Demaskierung.

```
‘(a ‘(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
          ==>  (a ‘(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 ’x)
      (name2 ’y))
  ‘(a ‘(b ,,name1 ,’,name2 d) e))
          ==>  (a ‘(b ,x ,’y d) e)
```

Die beiden Notationen ‘<qq-Schablone> und (quasiquote <qq-Schablone>) sind in allen Bezügen identisch. ‘,<Ausdruck>’ ist identisch mit ‘(unquote <Ausdruck>)’, und ‘,@<Ausdruck>’ ist identisch mit ‘(unquote-splicing <Ausdruck>)’. Welche externe Syntax Write für zweielementige Listen, deren Car eines dieser Symbole ist, erzeugt, darf je nach Implementierung abweichen.

```
(quasiquote (list (unquote (+ 1 2)) 4))
          ==>  (list 3 4)
’(quasiquote (list (unquote (+ 1 2)) 4))
          ==>  ‘(list ,(+ 1 2) 4)
     d.h. (quasiquote (list (unquote (+ 1 2)) 4))
```

Unvorhersehbares Verhalten darf auftreten, wenn eines der Symbole Quasiquote, Unquote oder Unquote-splicing in einer Positions innerhalb einer <qq-Schablone> auf andere Weise als oben beschrieben vorkommt.

4.3 Makros

Scheme-Programme können neue abgeleitete Ausdruckstypen definieren und benutzen. Diese nennt man *Makros*. Vom Programm definierte Ausdruckstypen haben die Syntax

`(<Schlüsselwort> <Datenelement> ...)`

wobei <Schlüsselwort> ein Bezeichner ist, der den Ausdruckstyp eindeutig festlegt. Dieser Bezeichner wird das *syntaktische Schlüsselwort* oder einfach *Schlüsselwort* des Makros genannt. Die Anzahl der <Datenelement>e und ihre Syntax hängt vom Ausdruckstyp ab.

Jede Instanz eines Makros wird als eine *Benutzung* des Makros bezeichnet. Die Menge an Regeln, die festlegen, wie eine Makro-Benutzung zu einem grundlegenderen Ausdruck umgeschrieben wird, heißt der *Umwandler* des Makros.

Die Mittel zur Makro-Definition setzen sich aus zwei Teilen zusammen:

- Eine Menge von Ausdrücken, um bestimmte Bezeichner als Makroausdrücke zu kennzeichnen, sie mit Makroumwandlern zu verknüpfen und den Geltungsbereich, in dem ein Makro definiert ist, zu steuern, sowie
- eine Mustersprache zum Angeben von Makroumwandlern.

Das syntaktische Schlüsselwort eines Makros darf Variablenbindungen überdecken und lokale Variablenbindungen dürfen Schlüsselwortbindungen überdecken. Alle Makros, die mit der Mustersprache definiert wurden, sind „hygienisch“ und „referenziell transparent“, also bleibt Schemes lexikalische Bindung intakt [Kohlbecker86], [hygienic], [Bawden88], [macrosthatwork], [syntacticabstraction]:

- Wenn ein Makroumwandler eine Bindung für einen Bezeichner (Variable oder Schlüsselwort) einfügt, verhält sich diese, als würde der Bezeichner in seinem gesamten Geltungssbereich umbenannt, um Konflikte mit anderen gleichnamigen Bezeichnern zu vermeiden. Beachten Sie, dass ein `define` auf oberster Ebene eine Bindung einführen darf, aber nicht muss, siehe den Abschnitt Abschnitt 5.2 [Definitionen], Seite 34.
- Wenn ein Makroumwandler eine freie Referenz auf einen Bezeichner einfügt, bezieht sich die Referenz auf die Bindung, die sichtbar war, als der Makroumwandler festgelegt wurde, ohne Berücksichtigung jeglicher lokaler Bindungen, die die Makro-Benutzung umgeben könnten.

4.3.1 Bindungskonstrukte für syntaktische Schlüsselwörter

‘`Let-syntax`’ und ‘`Letrec-syntax`’ sind analog zu ‘`Let`’ und ‘`Letrec`’, aber sie binden syntaktische Schlüsselwörter an Makroumwandler statt Variable an Stellen zu binden, die Werte enthalten. Syntaktische Schlüsselwörter dürfen auch auf oberster Ebene gebunden werden; siehe den Abschnitt Abschnitt 5.3 [Syntaxdefinitionen], Seite 36.

`let-syntax` <Bindingungen> <Rumpf> [Syntax]

Syntax: <Bindungen> sollte folgende Form haben:

`((<Schlüsselwort> <Umwandlerspezifikation>) ...,)`

Jedes <Schlüsselwort> ist ein Bezeichner, jede <Umwandlerspezifikation> ist eine Instanz von ‘`Syntax-rules`’ und <Rumpf> sollte eine Folge von einem oder mehreren Ausdrücken sein. Es ist ein Fehler, wenn ein <Schlüsselwort> mehr als einmal in der Liste der Schlüsselwörter auftaucht, die gebunden werden.

Semantik: Der <Rumpf> wird in der syntaktischen Umgebung umgeschrieben, die entsteht, wenn die syntaktische Umgebung des '`Let-syntax`'-Ausdrucks um Makros, deren Schlüsselwörter die aus den <Schlüsselwort>-Vorkommen sind, gebunden an die angegebenen Umwandler, erweitert wird. Jede Bindung eines <Schlüsselwort>s hat den <Rumpf> als ihre Region.

Das folgende Beispiel nimmt an, dass die benutzte Scheme-Implementierung das scharfe S (ß) auch als Buchstabenzeichen ansieht, als Erweiterung zur in diesem Bericht spezifizierten Syntax:

```
(let-syntax ((when (syntax-rules ()
                     ((when test anweisung1 anweisung2 ...)
                      (if test
                          (begin anweisung1
                                 anweisung2 ...))))))
  (let ((if #t))
    (when if (set! if 'jetzt))
    if))                                ==>  jetzt

(let ((x 'außen))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'innen))
      (m))))                            ==>  außen
```

`letrec-syntax` <Bindungen> <Rumpf> [Syntax]

Syntax: Dieselbe wie bei '`Let-syntax`'.

Semantik: Der <Rumpf> wird in der syntaktischen Umgebung umgeschrieben, die entsteht, wenn die syntaktische Umgebung des '`Letrec-syntax`'-Ausdrucks um die Makros, deren Schlüsselwörter die aus den <Schlüsselwort>-Vorkommen sind, gebunden an die angegebenen Umwandler, erweitert wird. Jede Bindung eines <Schlüsselwort>s hat die <Bindungen> sowie den <Rumpf> in ihrer Region, so dass die Umwandler Ausdrücke zu Benutzungen der Makros umschreiben können, die durch den '`Letrec-syntax`'-Ausdruck eingeführt wurden.

```
(letrec-syntax
  ((mein-or (syntax-rules ()
              ((mein-or) #f)
              ((mein-or e) e)
              ((mein-or e1 e2 ...)
               (let ((temp e1))
                 (if temp
                     temp
                     (mein-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
```

```
        (if even?))
    (mein-or x
           (let temp)
           (if y)
           y)))                          ==>  7
```

4.3.2 Mustersprache

Eine <Umwandlerspezifikation> hat folgende Form:

`syntax-rules` <Literale> <Syntaxregel> ...,

Syntax: <Literale> sind eine Liste von Bezeichnern, und jede <Syntaxregel> sollte die folgende Form haben:

(<Muster> <Schablone>)

Das <Muster> in einer <Syntaxregel> ist ein listenförmiges <Muster>, dessen Listenform mit dem Schlüsselwort für das Makro beginnt.

Allgemein ist ein <Muster> entweder ein Bezeichner, eine Konstante oder eines der Folgenden

```
(<Muster> ...)
(<Muster> <Muster> ... . <Muster>)
(<Muster> ... <Muster> <Auslassungspunkte>)
#(<Muster> ...)
#(<Muster> ... <Muster> <Auslassungspunkte>)
```

und eine Schablone ist ein Bezeichner, eine Konstante oder eines der Folgenden

```
(<Element> ...)
(<Element> <Element> ... . <Schablone>)
#(<Element> ...)
```

wobei ein <Element> eine <Schablone> ist, auf die optional <Auslassungspunkte> folgen, wobei <Auslassungspunkte> der Bezeichner „'...'“ ist (welcher nicht als Bezeichner in einer Schablone oder einem Muster benutzt werden darf).

Semantik: Eine Instanz von '`Syntax-rules`' erstellt einen neuen Makroumwandler, indem eine Folge hygienischer Umschreiberegeln angegeben wird. Eine Benutzung eines Makros, deren Schlüsselwort mit einem über '`Syntax-rules`' festgelegten Umwandler verknüpft ist, wird mit den Mustern abgeglichen, die unter den <Syntaxregel>n vorkommen, angefangen mit der am weisten links stehenden <Syntaxregel>. Wenn ein passendes Muster gefunden wurde, wird die Makrobenutzung hygienisch entsprechend der Schablone umgeschrieben.

Ein Bezeichner, der im Muster einer <Syntaxregel>n steht, ist eine *Mustervariable*, außer wenn sie das Schlüsselwort ist, das das Muster beginnt, unter den <Literale>n vorkommt oder der Bezeichner „‘...’“ ist. Mustervariable werden beliebigen Eingabeelementen zugeordnet und benutzt, um Elemente der Eingabe in der Schablone zu bezeichnen. Es ist ein Fehler, wenn dieselbe Mustervariable mehr als einmal in einem <Muster> auftaucht.

Das Schlüsselwort am Anfang des Musters in einer <Syntaxregel> ist nicht an dem Abgleich beteiligt und wird nicht als eine Mustervariable oder als Literalbezeichner angesehen.

> *Begründung:* Der Geltungsbereich des Schlüsselworts wird anhand des Ausdrucks oder der Syntaxdefinition festgelegt, die das Schlüsselwort an den verknüpften Makroumwandler bindet. Wenn das Schlüsselwort eine Mustervariable oder ein Literalbezeichner wäre, dann wäre die auf das Muster folgende Schablone selbst in seinem Geltungsbereich, unabhängig davon, ob das Schlüsselwort durch ‘`Let-syntax`’ oder ‘`Letrec-syntax`’ gebunden wurde.

Bezeichner, die unter den <Literale>n vorkommen, werden als Literalbezeichner ausgelegt, die mit den entsprechenden Unterformen der Eingabe abgeglichen werden. Eine Unterform in der Eingabe passt zu einem Literalbezeichner genau dann, wenn sie ein Bezeichner ist und entweder sowohl ihr Vorkommen im Makroausdruck als auch ihr Vorkommen in der Makrodefinition dieselbe lexikalische Bindung haben oder die beiden Bezeichner gleich sind und beide keine lexikalische Bindung haben.

Ein Untermuster gefolgt von ‘`...`’ kann zu null oder mehr Elementen der Eingabe passen. Es ist ein Fehler, wenn ‘`...`’ unter den <Literale>n vorkommt. Innerhalb eines Musters muss der Bezeichner ‘`...`’ auf das letzte Element einer nichtleeren Folge von Untermustern folgen.

Formaler ausgedrückt passt eine Eingabeform F zu einem Muster M genau dann, wenn:

- M ein nichtliteraler Bezeichner ist, oder
- M ein literaler Bezeichner ist und F ein Bezeichner mit derselben Bindung ist, oder
- M eine Liste ‘`(M_1 ... M_n)`’ und F eine Liste aus n Formen ist, welche jeweils zu M_1 bis M_n passen, oder
- M eine unechte Liste ‘`(M_1 M_2 ... M_n . M_n+1)`’ ist und F eine Liste oder unechte Liste von n oder mehr Formen ist, die jeweils zu M_1 bis M_n, passen und deren n-ter „Cdr“ zu M_n+1 passt, oder
- M von der Form ‘`(M_1 ... M_n M_n+1 <Auslassungspunkte>)`’ ist, wobei <Auslassungspunkte> der Bezeichner ‘`...`’ ist und F eine echte Liste aus mindestens n Formen ist, deren erste n Formen jeweils zu M_1 bis M_n passen, und jedes verbleibende Element von F zu M_n+1 passt, oder
- M ein Vektor von der Form ‘`#(M_1 ... M_n)`’ und F ein Vektor aus n Formen ist, die zu M_1 bis M_n passen, oder
- M von der Form ‘`#(M_1 ... M_n M_n+1 <Auslassungspunkte>)`’ ist, wobei <Auslassungspunkte> der Bezeichner ‘`...`’ und F ein Vektor aus n oder mehr Formen

ist, deren erste n jeweils zu M_1 bis M_n passen und wobei jedes verbleibende Element von F zu M_n+1 passt, oder

- M ein Datenelement und F gleich M ist, in dem Sinne der '`Equal?`'-Prozedur.

Es ist ein Fehler, ein Makroschlüsselwort im Geltungsbereich seiner Bindung in einem Ausdruck zu benutzen, der zu keinem seiner Muster passt.

Wenn eine Makro-Benutzung anhand der Schablone der passenden <Syntaxregel> umgeschrieben wird, werden Mustervariable, die in der Schablone vorkommen, durch die Subformen ersetzt, zu denen sie in der Eingabe passen. Mustervariable, die in Untermustern gefolgt von einer oder mehr Instanzen des Bezeichners '`...`' vorkommen, dürfen nur in Unterschablonen vorkommen, auf die ebenso viele Instanzen von '`...`' folgen. Sie werden in der Ausgabe durch all die Subformen ersetzt, die zur Eingabe passen, so verteilt, wie sie vorkommen. Es ist ein Fehler, wenn die Ausgabe nicht wie festgelegt aufgebaut werden kann.

Bezeichner, die in der Schablone vorkommen, aber keine Mustervariablen oder der Bezeichner '`...`' sind, werden wörtlich als literale Bezeichner eingefügt. Wenn ein literaler Bezeichner als freier Bezeichner eingefügt wird, referenziert er diejenige Bindung dieses Bezeichners, in dessen Geltungsbereich die Instanz von '`Syntax-rules`' vorkommt. Wenn ein literaler Bezeichner als gebundener Bezeichner eingefügt wird, dann verhält er sich, als würde er umbenannt, so wird unerwünschtes Fangen freier Bezeichner verhindert.

Wenn zum Beispiel `Let` und `Cond` wie im Abschnitt Abschnitt 4.2 [Abgeleitete Ausdruckstypen], Seite 21, definiert sind, dann sind sie hygienisch (wie vorgeschrieben) und das Folgende ist kein Fehler.

```
(let ((=> #f))
  (cond (#t => 'ok)))                    ==> ok
```

Der Makroumwandler für '`Cond`' erkennt '`=>`' als eine lokale Variable und daher als Ausdruck und nicht als den Bezeichner auf oberster Ebene '`=>`', den der Makroumwandler als ein syntaktisches Schlüsselwort behandelt. Daher wird das Beispiel umgeschrieben zu

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

statt zu

```
(let ((=> #f))
  (let ((temp #t))
    (if temp ('ok temp))))
```

was zu einem ungültigen Prozeduraufruf führen würde.

5 Programmstruktur

5.1 Programme

Ein Scheme-Programm besteht aus einer Folge von Ausdrücken, Definitionen und Syntaxdefinitionen. Ausdrücke werden im Kapitel Kapitel 4 [Ausdrücke], Seite 17, beschrieben, Definitionen und Syntaxdefinitionen sind Thema im Rest des momentanen Kapitels.

Programme werden typischerweise in Dateien gespeichert oder interaktiv in ein laufendes Scheme-System eingegeben, wobei auch andere Paradigmen möglich sind; Fragen der Benutzerschnittstelle gehören dabei nicht zum in diesem Bericht behandelten Stoff. (Tatsächlich könnte Scheme auch als eine Notation zum Ausdrücken von Berechnungsmethoden von Nutzen sein, selbst ohne jegliche mechanische Implementierung.)

Definitionen und Syntaxdefinitionen, die auf der obersten Ebene eines Programms stehen, können deklarativ interpretiert werden. Sie führen dazu, dass Bindungen in der Umgebung auf oberster Ebene erzeugt werden oder sie ändern den Wert einer bestehenden Bindung auf oberster Ebene. Ausdrücke, die auf der obersten Ebene eines Programms vorkommen, werden imperativ interpretiert; sie werden der Reihe nach ausgeführt, wenn das Programm aufgerufen oder geladen wird und führen typischerweise irgendeine Art von Initialisierung des Programms durch.

Auf der obersten Ebene eines Programms ist ein `(begin <Form1> ...,)` äquivalent zur Folge von Ausdrücken, Definitionen und Syntaxdefinitionen, die den Rumpf des `Begin` bilden.

5.2 Definitionen

Definitionen sind in manchen, aber nicht in jedem Kontext gültig, in dem Ausdrücke stehen dürfen. Sie sind nur auf der obersten Ebene eines <Programm>s gültig und am Anfang eines <Rumpf>s.

Eine Definition sollte eine der folgenden Formen haben:

- `(define` <Variable> <Ausdruck>`)`
- `(define (`<Variable> <Formale>`)` <Rumpf>`)`

  <Formale> sollten entweder eine Folge von null oder mehr Variablen sein oder eine Folge von einer oder mehr Variablen, gefolgt von einem durch Leerzeichen begrenzten Punkt und einer weiteren Variablen (wie in einem Lambda-Ausdruck). Diese Form ist äquivalent zu

  ```
  (define <Variable>
    (lambda (<Formale>) <Rumpf>)).
  ```

- `(define (`<Variable> `.` <Formale>`)` <Rumpf>`)`

  <Formale> sollte eine einzelne Variable sein. Diese Form ist äquivalent zu

  ```
  (define <Variable>
    (lambda <Formale> <Rumpf>)).
  ```

5.2.1 Definitionen auf oberster Ebene

Auf der obersten Ebene eines Programms hat eine Definition

```
(define <Variable> <Ausdruck>)
```

grundlegend dieselbe Wirkung wie ein Zuweisungsausdruck

```
(set! <Variable> <Ausdruck>)
```

wenn <Variable> gebunden ist. Wenn <Variable> jedoch nicht gebunden ist, dann wird die Definition die <Variable> an eine neue Stelle binden, bevor sie die Zuweisung durchführt, wohingegen es ein Fehler wäre, '`Set!`' auf einer ungebundenen Variablen durchzuführen.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                                ==>  6
(define first car)
(first '(1 2))                          ==>  1
```

Manche Implementierungen von Scheme benutzen eine Anfangsumgebung, in der alle möglichen Variablen schon vorab an Stellen gebunden sind, von denen die meisten undefinierte Werte enthalten. Definitionen auf oberster Ebene sind in einer solchen Implementierung wirklich gleichbedeutend mit Zuweisungen.

5.2.2 Interne Definitionen

Definitionen dürfen am Anfang eines <Rumpf>s vorkommen (das heißt, eines Rumpfs eines `Lambda`-, `Let`-, `Let*`-, `Letrec`-, `Let-syntax`- oder `Letrec-syntax`-Ausdrucks oder dem einer Definition in angemessener Form). Solche Definitionen sind bekannt als *interne Definitionen* im Gegensatz zur Umgebung auf oberster Ebene, was oben beschrieben ist. Die durch eine interne Definition definierte Variable gilt lokal für den <Rumpf>. Das heißt, die <Variable> wird gebunden statt zugewiesen und die Region der Bindung ist der gesamte <Rumpf>. Zum Beispiel

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))                        ==>  45
```

Ein <Rumpf>, der interne Definitionen enthält, kann immer in einen völlig gleichbedeutenden '`Letrec`'-Ausdruck umgewandelt werden. Zum Beispiel ist der '`Let`'-Ausdruck im obigen Beispiel gleichbedeutend mit

```
(let ((x 5))
  (letrec ((foo (lambda (y) (bar x y)))
```

```
        (bar (lambda (a b) (+ (* a b) a))))
  (foo (+ x 3))))
```

Genau wie bei dem gleichbedeutenden '`Letrec`'-Ausdruck muss es möglich sein, jeden <Ausdruck> jeder internen Definition in einem <Rumpf> auszuwerten, ohne den Wert irgendeiner der <Variable>n, die definiert werden, zuzuweisen oder sich darauf zu beziehen.

Wann immer eine interne Definition vorkommen darf, ist `(begin` <Definition1> `...,)` gleichbedeutend mit der Folge von Definitionen, die den Rumpf des `Begin` bilden.

5.3 Syntaxdefinitionen

Syntaxdefinitionen sind nur auf der obersten Ebene eines <Programm>s gültig.

Sie haben die folgende Form:

`(define-syntax` <Schlüsselwort> <Umwandlerspezifikation>`)`

<Schlüsselwort> ist ein Bezeichner und die <Umwandlerspezifikation> sollte eine Instanz von `Syntax-rules` sein. Die syntaktische Umgebung auf oberster Ebene wird erweitert durch Bindung des <Schlüsselwort>s an den angegebenen Umwandler.

Es gibt keine Entsprechung von '`Define-syntax`' für interne Definitionen.

Obwohl Makros in jedem Kontext, der sie zulässt, zu Definitionen und Syntaxdefinitionen umgeschrieben werden können, ist es ein Fehler, wenn eine Definition oder eine Syntaxdefinition ein syntaktisches Schlüsselwort überschattet, dessen Bedeutung nötig ist, um zu bestimmen, ob eine Form aus der Gruppe von Formen, die die überschattende Definition enthält, tatsächlich eine Definition ist oder, für interne Definitionen, nötig ist, um die Grenze zwischen der Gruppe und den auf die Gruppe folgenden Ausdrücken zu bestimmen. Zum Beispiel sind die folgenden Code-Stücke Fehler:

```
(define define 3)

(begin (define begin list))

(let-syntax
  ((foo (syntax-rules ()
          ((foo (proz args ...) rumpf ...)
           (define proz
             (lambda (args ...)
               rumpf ...))))))
  (let ((x 3))
    (foo (plus x y) (+ x y))
    (define foo x)
    (plus foo x)))
```

6 Standardprozeduren

Dieses Kapitel beschreibt die eingebauten Prozeduren von Scheme. Die anfängliche Scheme-Umgebung (bzw. die Scheme-Umgebung auf „oberster Ebene“) enthält von Anfang an eine Reihe von Variablen, die an Stellen mit nützlichen Werten gebunden sind, von denen die meisten elementare Prozeduren sind, die Daten manipulieren. Zum Beispiel ist die Variable ‘`Abs`’ gebunden an eine (Stelle mit einer) Prozedur mit einem einzelnen Argument, die den Absolutbetrag einer Zahl berechnet, und die Variable ‘`+`’ ist an eine Prozedur gebunden, die Summen berechnet. Eingebaute Prozeduren, die leicht über andere eingebaute Prozeduren ausgedrückt werden könnten, werden als „Bibliotheksprozeduren“ ausgewiesen.

Ein Programm darf eine Definition auf oberster Ebene benutzen, um eine beliebige Variable zu binden. Es darf daraufhin eine jede solche Bindung durch eine Zuweisung (siehe siehe Abschnitt 4.1.6 [Zuweisungen], Seite 20) wieder ändern. Diese Operationen verändern das Verhalten der in Scheme eingebauten Prozeduren nicht. Das Ändern einer beliebigen Bindung auf oberster Ebene, die nicht durch eine Definition eingeführt wurde, hat allerdings eine unbestimmte Wirkung auf das Verhalten der eingebauten Prozeduren.

6.1 Äquivalenzprädikate

Ein *Prädikat* ist eine Prozedur, die immer einen booleschen Wert (`#t` oder `#f`) liefert. Ein *Äquivalenzprädikat* ist die berechnende Entsprechung der mathematischen Äquivalenzrelation (es ist symmetrisch, reflexiv und transitiv). Von den in diesem Abschnitt beschriebenen Äquivalenzprädikaten ist ‘`Eq?`’ das feinste oder am stärksten unterscheidende und ‘`Equal?`’ ist das gröbste. ‘`Eqv?`’ unterscheidet etwas weniger als ‘`Eq?`’.

`eqv?` *obj1 obj2* [Prozedur]

Die Prozedur ‘`Eqv?`’ definiert eine nützliche Äquivalenzrelation auf Objekten. Kurz gesagt liefert sie `#t`, wenn *Obj1* und *Obj2* unter normalen Umständen als dasselbe Objekt angesehen werden sollte. Die Bedeutung der Relation lässt ein wenig Interpretationsspielraum, aber die folgende teilweise Spezifikation von ‘`Eqv?`’ gilt für alle Implementierungen von Scheme.

Die ‘`eqv?`’-Prozedur liefert `#t`, wenn:

- *Obj1* und *Obj2* beide `#t` oder beide `#f` sind.
- *Obj1* und *Obj2* beide Symbole sind und

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                                    ==>  #t
```

  Anmerkung: Die Annahme ist, dass weder *Obj1* noch *Obj2* ein „nicht interniertes Symbol“ ist, wie im Abschnitt Abschnitt 6.3.3 [Symbole], Seite 60, angedeutet. Dieser Bericht versucht nicht, das Verhalten von ‘`Eqv?`’ für implementierungsabhängige Erweiterungen festzulegen.

- *Obj1* und *Obj2* beide Zahlen und numerisch gleich sind (siehe ‘`=`’ im Abschnitt siehe Abschnitt 6.2 [Zahlen], Seite 41) und außerdem entweder beide exakt oder beide inexakt sind.

- *Obj1* und *Obj2* beide Zeichen sind und laut der Prozedur '`Char=?`' dasselbe Zeichen sind (siehe den Abschnitt siehe Abschnitt 6.3.4 [Zeichen], Seite 62).
- *Obj1* und *Obj2* beide die leere Liste sind.
- *Obj1* und *Obj2* Paare, Vektoren oder Zeichenketten sind, die dieselben Stellen im Speicher bezeichnen (siehe den Abschnitt siehe Abschnitt 3.4 [Speichermodell], Seite 13).
- *Obj1* und *Obj2* Prozeduren sind, deren Stellenbeschriftungen gleich sind (siehe den Abschnitt siehe Abschnitt 4.1.4 [Prozeduren], Seite 19).

Die Prozedur '`Eqv?`' liefert `#f`, wenn:

- *Obj1* und *Obj2* verschiedene Typen sind (Abschnitt siehe Abschnitt 3.2 [Typfremdheit], Seite 11).
- eines der *Obj1* und *Obj2* ein `#t` ist, aber das andere ein `#f` ist.
- *Obj1* und *Obj2* Symbole sind, aber

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                                    ==>  #f
```

- eines von *Obj1* und *Obj2* eine exakte Zahl ist, aber das andere eine inexakte Zahl ist.
- *Obj1* und *Obj2* Zahlen sind, für die die Prozedur '`=`' den Wert `#f` liefert.
- *Obj1* und *Obj2* Zeichen sind, für die die Prozedur '`Char=?`' den Wert `#f` liefert.
- eines von *Obj1* und *Obj2* die leere Liste ist, aber das andere nicht.
- *Obj1* und *Obj2* Paare, Vektoren oder Zeichenketten sind, die unterschiedliche Stellen bezeichnen.
- *Obj1* und *Obj2* Prozeduren sind, die sich für manche Argumente unterschiedlich verhalten würden (verschiedene Werte liefern oder verschiedene Nebenwirkungen haben würden).

```
(eqv? 'a 'a)                            ==>  #t
(eqv? 'a 'b)                            ==>  #f
(eqv? 2 2)                              ==>  #t
(eqv? '() '())                          ==>  #t
(eqv? 100000000 100000000)              ==>  #t
(eqv? (cons 1 2) (cons 1 2))            ==>  #f
(eqv? (lambda () 1)
      (lambda () 2))                    ==>  #f
(eqv? #f 'nil)                          ==>  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                           ==>  #t
```

Die folgenden Beispiele veranschaulichen Fälle, für die die obigen Regeln das Verhalten von '`Eqv?`' nicht vollständig angeben. Alles, was sich über solche Fälle sagen lässt, ist, dass der von '`Eqv?`' gelieferte Wert ein boolescher Wert sein muss.

```
(eqv? "" "")                          ==>  unbestimmt
(eqv? '#() '#())                      ==>  unbestimmt
(eqv? (lambda (x) x)
      (lambda (x) x))                 ==>  unbestimmt
(eqv? (lambda (x) x)
      (lambda (y) y))                 ==>  unbestimmt
```

Der nächste Satz von Beispielen zeigt die Nutzung von '`Eqv?`' mit Prozeduren, die lokalen Zustand haben. Die Beispiele nehmen an, dass die benutzte Scheme-Implementierung die Umlaute auch als Buchstabenzeichen ansieht, als Erweiterung zur in diesem Bericht spezifizierten Syntax. '`Gen-Zähler`' muss jedes Mal eine andere Prozedur liefern, denn jede Prozedur hat ihren eigenen internen Zähler. '`Gen-Verlierer`' liefert jedoch jedes Mal gleichwertige Prozeduren, denn der lokale Zustand beeinflusst die Werte oder Nebenwirkungen der Prozeduren nicht.

```
(define gen-zähler
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-zähler)))
  (eqv? g g))                         ==>  #t
(eqv? (gen-zähler) (gen-zähler))
                                      ==>  #f
(define gen-verlierer
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-verlierer)))
  (eqv? g g))                         ==>  #t
(eqv? (gen-verlierer) (gen-verlierer))
                                      ==>  unbestimmt

(letrec ((f (lambda () (if (eqv? f g) 'beide 'f)))
         (g (lambda () (if (eqv? f g) 'beide 'g))))
  (eqv? f g))
                                      ==>  unbestimmt

(letrec ((f (lambda () (if (eqv? f g) 'f 'beide)))
         (g (lambda () (if (eqv? f g) 'g 'beide))))
  (eqv? f g))
                                      ==>  #f
```

Weil es ein Fehler ist, konstante Objekte zu verändern (die, die von literalen Ausdrücken geliefert werden), dürfen Implementierungen die Struktur zwischen Konstan-

ten teilen, müssen aber nicht. Daher ist der Wert von ‘`Eqv?`’ für Konstante manchmal implementierungsabhängig.

```
(eqv? '(a) '(a))                    ==>  unbestimmt
(eqv? "a" "a")                      ==>  unbestimmt
(eqv? '(b) (cdr '(a b)))            ==>  unbestimmt
(let ((x '(a)))
  (eqv? x x))                       ==>  #t
```

> *Begründung:* Obige Definition von ‘`Eqv?`’ gestattet Implementierungen Freiraum bei der Behandlung von Prozeduren und Literalen: Implementierungen steht es frei, entweder die Gleichheit zweier Prozeduren oder zweier Literale zu bestätigen oder nicht bestätigen zu können, ob sie gleich sind, so dass sie selbst entscheiden können, ob Darstellungen äquivalenter Objekte durch Nutzung desselben Zeigers oder Bit-Musters verschmolzen werden sollen, um eine Darstellung für beide zu sein.

`eq?` *obj1 obj2* [Prozedur]

‘`Eq?`’ ist ähnlich wie ‘`Eqv?`’, außer dass es in manchen Fällen in der Lage ist, feinere Unterschiede zu erkennen als ‘`Eqv?`’.

Es wird garantiert, dass sich ‘`Eq?`’ und ‘`Eqv?`’ auf Symbolen, booleschen Werten, der leeren Liste, Paaren, Prozeduren und nicht leeren Zeichenketten und Vektoren gleich verhalten. Das Verhalten von ‘`Eq?`’ für Zahlen und Zeichen ist implementierungsabhängig, aber es wird immer entweder `#t` oder `#f` liefern und wird nur dann `#t` liefern, wenn ‘`Eqv?`’ auch `#t` liefern würde. ‘`Eq?`’ darf sich für leere Vektoren und leere Zeichenketten auch anders als ‘`Eqv?`’ verhalten.

```
(eq? 'a 'a)                         ==>  #t
(eq? '(a) '(a))                     ==>  unbestimmt
(eq? (list 'a) (list 'a))           ==>  #f
(eq? "a" "a")                       ==>  unbestimmt
(eq? "" "")                         ==>  unbestimmt
(eq? '() '())                       ==>  #t
(eq? 2 2)                           ==>  unbestimmt
(eq? #\A #\A)                       ==>  unbestimmt
(eq? car car)                       ==>  #t
(let ((n (+ 2 3)))
  (eq? n n))                        ==>  unbestimmt
(let ((x '(a)))
  (eq? x x))                        ==>  #t
(let ((x '#()))
  (eq? x x))                        ==>  #t
(let ((p (lambda (x) x)))
  (eq? p p))                        ==>  #t
```

Begründung: Gewöhnlich wird man '`Eq?`' deutlich effizienter implementieren können als '`Eqv?`', zum Beispiel als einfacher Vergleich von Zeigern statt einer komplizierteren Operation. Ein Grund ist, dass es unmöglich sein kann, '`Eqv?`' für zwei Zahlen in konstanter Zeit zu berechnen, während '`Eq?`', als Zeigervergleich implementiert, immer in konstanter Zeit abgeschlossen sein wird. '`Eq?`' darf wie '`Eqv?`' benutzt werden, wenn in einer Anwendung Prozeduren benutzt werden, um zustandsbehaftete Objekte zu implementieren, da es denselben Einschränkungen wie '`Eqv?`' genügt.

`equal?` *obj1 obj2* [Bibliotheksprozedur]

'`Equal?`' vergleicht den Inhalt von Paaren, Vektoren und Zeichenketten rekursiv, indem es auf andere Objekte wie Zahlen und Symbole '`Eqv?`' anwendet. Eine Daumenregel ist, dass Objekte im Allgemeinen '`Equal?`' sind, wenn sie gleich ausgegeben werden. '`Equal?`' muss nicht terminieren, wenn seine Argumente zirkuläre Datenstrukturen sind.

```
(equal? 'a 'a)                          ==>  #t
(equal? '(a) '(a))                      ==>  #t
(equal? '(a (b) c)
        '(a (b) c))                     ==>  #t
(equal? "abc" "abc")                    ==>  #t
(equal? 2 2)                            ==>  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))             ==>  #t
(equal? (lambda (x) x)
        (lambda (y) y))                 ==>  unbestimmt
```

6.2 Zahlen

Numerische Berechnung wurde traditionell von der Lisp-Gemeinschaft vernachlässigt. Bis Common Lisp gab es keine vorsichtig durchdachte Strategie zum Organisieren numerischer Berechnungen und abgesehen vom MacLisp-System [Pitman83] wurden kaum Anstrengungen unternommen, numerischen Code effizient auszuführen. Dieser Bericht erkennt die exzellente Arbeit des Common-Lisp-Komitees an und akzeptiert viele ihrer Empfehlungen. Auf manche Art vereinfacht und verallgemeinert dieser Bericht deren Vorschläge auf eine Art, die mit den Zwecken von Scheme im Einklang steht.

Es ist wichtig, zwischen mathematischen Zahlen, den Scheme-Zahlen, welche diese zu modellieren versuchen, und Notationen zum Aufschreiben von Zahlen zu unterscheiden. Dieser Bericht benutzt die Typen Zahl (*number*), komplexe Zahl (*complex*), reelle Zahl (*real*), rationale Zahl (*rational*) und ganze Zahl (*integer*), um sowohl die mathematischen Zahlen als auch die Scheme-Zahlen zu bezeichnen. Maschinendarstellungen wie Festkomma (fixed point) und Gleitkomma (floating point) werden [im englischsprachigen Scheme-Standard] durch Namen wie *fixnum* und *flonum* bezeichnet.

6.2.1 Numerische Typen

Mathematisch können Zahlen in einem Turm von Subtypen angeordnet werden, in dem jede Stufe eine Teilmenge der darüberliegenden Stufe ist:

number (Zahl)
complex (komplexe Zahl)
real (reelle Zahl)
rational (rationale Zahl)
integer (ganze Zahl)

Zum Beispiel ist 3 eine ganze Zahl. Daher ist 3 auch eine rationale, reelle und komplexe Zahl. Dasselbe gilt für die Scheme-Zahlen, die die 3 modellieren. Für Scheme-Zahlen sind diese Typen durch die Prädikate `Number?`, `Complex?`, `Real?`, `Rational?`, und `Integer?` definiert.

Es gibt keine einfache Beziehung zwischen dem Typ einer Zahl und ihrer Darstellung in einem Rechner. Obwohl die meisten Implementierungen von Scheme zumindest zwei verschiedene Darstellungen von 3 anbieten werden, bezeichnen diese unterschiedlichen Darstellungen dieselbe ganze Zahl.

Schemes numerische Operationen behandeln Zahlen als abstrakte Daten, die so unabhängig wie möglich von ihrer Darstellung sind. Obwohl eine Implementierung von Scheme Festkomma (fixnum), Gleitkomma (flonum) und unter Umständen andere Repräsentationen benutzen darf, sollte dies gegenüber dem Gelegenheitsprogrammierer, der einfache Programme schreibt, unauffällig sein.

Es ist allerdings notwendig, zwischen Zahlen zu unterscheiden, die exakt dargestellt werden, und denen, deren Darstellung inexakt sein darf. Zum Beispiel müssen Indizes auf Datenstrukturen exakt bekannt sein, genau wie manche Polynomialkoeffizienten eines Systems für symbolische Algebra exakt bekannt sein müssen. Andererseits sind Messergebnisse an sich schon inexakt und irrationale Zahlen können durch rationale Zahlen und somit inexakt angenähert werden. Damit abgefangen werden kann, wenn inexakte Zahlen benutzt werden, wo exakte Zahlen benötigt werden, unterscheidet Scheme ausdrücklich zwischen exakten und inexakten Zahlen. Diese Unterscheidung ist unabhängig von der Größendimension des Datentyps.

6.2.2 Exaktheit

Scheme-Zahlen sind entweder *exakt* oder *inexakt*. Eine Zahl ist exakt, wenn sie als exakte Konstante geschrieben wurde oder sich aus exakten Zahlen nur durch exakte Operationen ergeben hat. Eine Zahl ist inexakt, wenn sie als inexakte Konstante geschrieben wurde, sich aus inexakten Zutaten ergeben hat oder sich unter Benutzung inexakter Operationen ergeben hat. Daher ist Inexaktheit eine ansteckende Eigenschaft einer Zahl.

Wenn zwei Implementierungen exakte Ergebnisse für eine Berechnung liefern, die keine inexakten Zwischenergebnisse verwendet hat, werden die beiden Endergebnisse mathematisch äquivalent sein. Dies ist im Allgemeinen unwahr für Berechnungen, die inexakte Zahlen verwenden, denn dort dürfen Näherungsverfahren wie Gleitkommaarithmetik verwendet werden, aber es ist die Pflicht der jeweiligen Implementierung, die Ergebnisse so nah ans mathematische Ideal zu bringen, wie es praktikabel ist.

Rationale Operationen wie '`+`' sollten immer exakte Ergebnisse liefern, wenn ihnen exakte Argumente übergeben werden. Wenn die Operation keine exakten Ergebnisse liefern

kann, darf sie entweder die Verletzung einer Implementierungseinschränkung melden, oder das Ergebnis stillschweigend in einen inexakten Wert umwandeln. Siehe den Abschnitt Abschnitt 6.2.3 [Implementierungseinschränkungen], Seite 43.

Mit Ausnahme von `inexact->exact` müssen die in diesem Abschnitt beschriebenen Operationen im Allgemeinen inexakte Ergebnisse liefern, wenn ihnen auch beliebige inexakte Argumente gegeben wurden. Eine Operation darf allerdings ein exaktes Ergebnis liefern, wenn sie beweisen kann, dass der Wert des Ergebnisses nicht von der Inexaktheit ihrer Argumente betroffen ist. Zum Beispiel liefert die Multiplikation einer beliebigen Zahl mit einer exakten Null eine exakte Null als Ergebnis, selbst wenn das andere Argument inexakt ist.

6.2.3 Implementierungseinschränkungen

Implementierungen von Scheme müssen nicht den gesamten Turm von Untertypen, die im Abschnitt Abschnitt 6.2.1 [Numerische Typen], Seite 42, angegeben wurden, implementieren, aber sie müssen eine widerspruchsfreie Teilmenge davon implementieren, die sowohl mit den Zwecken der Implementierung als auch dem Geist der Scheme-Sprache konsistent ist. Zum Beispiel kann selbst eine Implementierung, in der alle Zahlen reell sind, dennoch ziemlich nützlich sein.

Implementierungen dürfen auch bloß einen eingeschränkten Wertebereich eines beliebigen Typs unterstützen, wenn er die Anforderungen dieses Abschnitts erfüllt. Der unterstützte Bereich exakter Zahlen eines beliebigen Typs darf sich von dem unterstützten Bereich inexakter Zahlen dieses Typs unterscheiden. Zum Beispiel darf eine Implementierung, die Gleitkommazahlen zur Darstellung all ihrer inexakten reellen Zahlen verwendet, einen praktisch unbeschränkten Wertebereich für exakte ganze und rationale Zahlen unterstützen, während sie den Wertebereich für inexakte reelle (und deswegen auch den Wertebereich für inexakte ganze und rationale Zahlen) auf den dynamischen Wertebereich des Gleitkommaformats einschränkt. Desweiteren sind die Lücken zwischen darstellbaren inexakten ganzen und rationalen Zahlen in einer solchen Implementierung wahrscheinlich sehr groß, wenn sich die Zahlen den Grenzen des Wertebereichs annähern.

Eine Implementierung von Scheme muss exakte ganze Zahlen in dem Wertebereich unterstützen, der als Index für Listen, Vektoren und Zeichenketten verwendet werden kann, oder der das Ergebnis der Berechnung der Länge einer Liste, eines Vektors oder eine Zeichenkette sein kann. Die Prozeduren `Length`, `Vector-length`, und `String-length` müssen eine exakte ganze Zahl liefern und es ist ein Fehler, etwas anderes als eine exakte ganze Zahl als Index zu benutzen. Desweiteren wird jede ganzzahlige Konstante, wenn sie durch eine exakte Syntax für ganze Zahlen ausgedrückt wurde, tatsächlich auch als eine exakte ganze Zahl eingelesen, ohne Rücksicht auf jegliche Implementierungseinschränkungen, die außerhalb dieses Bereichs gelten. Zuletzt werden die unten angeführten Prozeduren immer eine exakte ganze Zahl als Ergebnis liefern, sofern all ihre Argumente exakte ganze Zahlen sind und das mathematisch erwartete Ergebnis als exakte ganze Zahl innerhalb der Implementierung darstellbar ist:

```
+           -           *
quotient    remainder   modulo
max         min         abs
numerator   denominator gcd
lcm         floor       ceiling
```

```
truncate     round          rationalize
expt
```

Implementierungen werden ermutigt, sind aber nicht verpflichtet, exakte ganze und exakte rationale Zahlen praktisch unbeschränkter Größe und Genauigkeit zu unterstützen, und die oben genannten Prozeduren und die Prozedur '`/`' auf eine Art zu implementieren, so dass sie immer exakte Ergebnisse liefern, wenn ihnen exakte Argumente gegeben werden. Wenn eine dieser Prozeduren kein exaktes Ergebnis liefern kann, obwohl ihr exakte Argumente übergeben wurden, dann darf sie entweder eine Verletzung einer Implementierungseinschränkung melden oder sie darf stillschweigend das Ergebnis in einen inexakten Wert umwandeln. Eine solche Anpassung darf später zu einem Fehler führen.

Eine Implementierung darf Gleitkommazahlen und andere näherungsweise Darstellungsstrategien für inexakte Zahlen benutzen.

Dieser Bericht empfiehlt, setzt aber nicht voraus, dass Implementierungen, die Gleitkommadarstellungen benutzen, den IEEE-Standards für 32-Bit- und 64-Bit-Gleitkommazahlen folgen, und dass Implementierungen, die andere Darstellungen benutzen, dieselbe oder eine höhere Genauigkeit bieten, wie sie unter Benutzung dieser Gleitkommastandards erreicht werden kann [IEEE].

Insbesondere müssen Implementierungen, die eine Gleitkommadarstellung verwenden, diesen Regeln folgen: Ein Gleitkommaergebnis muss mit mindestens soviel Genauigkeit dargestellt werden, wie Genauigkeit benutzt wurde, um ein beliebiges Argument der Operation auszudrücken. Es ist wünschenswert (aber nicht notwendig), dass womöglich inexakte Operationen wie '`Sqrt`', wenn sie auf exakte Argumente angewandt werden, exakte Antworten liefern, wann immer das möglich ist (zum Beispiel sollte die Quadratwurzel einer exakten 4 eine exakte 2 sein). Wenn eine Operation jedoch anhand einer exakten Zahl ein inexaktes Ergebnis liefert (wie durch '`Sqrt`'), und wenn das Ergebnis als eine Gleitkommazahl dargestellt wird, dann muss dafür das verfügbare Gleitkommaformat mit der höchsten Genauigkeit benutzt werden, aber wenn das Ergebnis auf andere Art dargestellt wird, muss die Darstellung mindestens soviel Genauigkeit wie das genaueste verfügbare Gleitkommaformat haben.

Obwohl Scheme eine Vielfalt geschriebener Notationen für Zahlen unterstützt, darf eine Implementierung auch nur manche davon unterstützen. Zum Beispiel muss eine Implementierung nicht die kartesische und polare Notation für komplexe Zahlen unterstützen. Wenn eine Implementierung eine exakte numerische Konstante vorfindet, die sie nicht als eine exakte Zahl darstellen kann, darf sie entweder eine Verletzung einer Implementierungseinschränkung melden oder stillschweigend die Konstante durch eine inexakte Zahl darstellen.

6.2.4 Syntax numerischer Konstanter

Die Syntax der geschriebenen Darstellungen für Zahlen wird formal im Abschnitt Abschnitt 7.1.1 [Lexikalische Struktur], Seite 82, beschrieben. Beachten Sie, dass Groß- und Kleinschreibung für numerische Konstante nicht unterschieden wird.

Eine Zahl darf binär, oktal, dezimal oder hexadezimal durch Nutzung eines Radix-Präfixes geschrieben werden. Die Radix-Präfixe sind '`#b`' (binär), '`#o`' (oktal), '`#d`' (dezimal) und '`#x`' (hexadezimal). Ohne ein Radix-Präfix wird eine Zahl als dezimal angenommen.

Eine numerische Konstante darf durch ein Präfix als entweder exakt oder inexakt festgelegt werden. Die Präfixe sind '`#e`' für exakt und '`#i`' für inexakt. Ein Exaktheitspräfix darf vor oder nach einem beliebigen benutzten Radix-Präfix stehen. Wenn die geschriebene Darstellung einer Zahl kein Exaktheitspräfix hat, kann die Konstante entweder inexakt oder exakt sein. Sie ist inexakt, wenn sie einen Punkt als Dezimaltrennzeichen, einen Exponenten oder ein „#"-Zeichen anstelle einer Ziffer enthält, andernfalls ist sie exakt.

In Systemen mit inexakten Zahlen verschiedener Genauigkeiten kann es nützlich sein, die Genauigkeit einer Konstanten anzugeben. Zu diesem Zweck dürfen numerische Konstante mit einer Exponentenmarkierung geschrieben werden, die die gewünschte Genauigkeit der inexakten Darstellung anzeigt. Die Buchstaben '`s`', '`f`', '`d`' und '`l`' legen jeweils die Nutzung von kleiner („*short*"), einfacher („*single*"), doppelter („*double*") und großer („*long*") Genauigkeit fest. (Wenn weniger als vier interne inexakte Darstellungen existieren, werden die vier spezifizierten Größen auf die verfügbaren abgebildet. Zum Beispiel darf eine Implementierung mit zwei internen Darstellungen „short" und „single" zusammenlegen und „long" und „double" zusammenlegen.) Zusätzlich steht die Exponentenmarkierung '`e`' für die standardmäßige Genauigkeit der Implementierung. Die standardmäßige Genauigkeit ist mindestens so genau wie *double*, aber Implementierungen dürfen es der Nutzerin erlauben, die Standardgenauigkeit selbst festzulegen.

`3.14159265358979F0`
: Runden auf single — `3.141593`

`0.6L0`
: Erweitern auf long — `.600000000000000`

6.2.5 Numerische Operationen

Der Leser sei verwiesen auf den Abschnitt Abschnitt 1.3.3 [Eintragsformat], Seite 5, für eine Zusammenfassung der Namenskonventionen, mit denen Einschränkungen der Argumenttypen numerischer Routinen angegeben werden.

Die in diesem Abschnitt benutzten Beispiele gehen davon aus, dass jede mit exakter Notation angegebene numerische Konstante tatsächlich als eine exakte Zahl dargestellt wird. Manche Beispiele gehen auch davon aus, dass bestimmte numerische Konstante, die mit inexakter Notation geschrieben wurden, ohne Verlust von Genauigkeit dargestellt werden können; die inexakten Konstanten wurden so gewählt, dass dies in Implementierungen, die Gleitkommazahlen benutzen, um inexakte Zahlen darzustellen, wahrscheinlich der Fall ist.

`number?` *obj* [Prozedur]
`complex?` *obj* [Prozedur]
`real?` *obj* [Prozedur]
`rational?` *obj* [Prozedur]
`integer?` *obj* [Prozedur]

Diese numerischen Typprädikate können auf jede Art von Argument angewandt werden, auch auf Nichtzahlen. Sie liefern `#t`, wenn das Objekt den genannten Typ hat, und sonst liefern sie `#f`. Im Allgemeinen ist, wenn ein Typprädikat für eine Zahl wahr ist, auch jedes höhere Typprädikat für diese Zahl wahr. Umgekehrt sind, wenn ein Typprädikat für eine Zahl falsch ist, alle niedrigeren Typprädikate für diese Zahl auch falsch.

Wenn *z* eine inexakte komplexe Zahl ist, dann ist '`(real? z)`' genau dann wahr, wenn '`(zero? (imag-part z))`' wahr ist. Wenn *x* eine inexakte reelle Zahl ist, dann ist '`(integer? x)`' genau dann wahr, wenn '`(= x (round x))`'.

```
(complex? 3+4i)                    ==>  #t
(complex? 3)                       ==>  #t
(real? 3)                          ==>  #t
(real? -2.5+0.0i)                  ==>  #t
(real? #e1e10)                     ==>  #t
(rational? 6/10)                   ==>  #t
(rational? 6/3)                    ==>  #t
(integer? 3+0i)                    ==>  #t
(integer? 3.0)                     ==>  #t
(integer? 8/4)                     ==>  #t
```

Anmerkung: Das Verhalten dieser Typprädikate ist auf inexakten Zahlen unverlässlich, denn jede Ungenauigkeit kann das Ergebnis beeinflussen.

Anmerkung: In vielen Implementierung wird die Prozedur `Rational?` dieselbe sein wie `Real?` und die Prozedur `Complex?` wird dieselbe sein wie `Number?`, aber ungewöhnliche Implementierungen könnten nur in der Lage sein, manche irrationale Zahlen exakt darzustellen, oder das Zahlensystem erweitern, so dass es auch eine Art nicht komplexer Zahlen unterstützt.

`exact?` *z* [Prozedur]
`inexact?` *z* [Prozedur]

Diese numerischen Prädikate bieten Exaktheitheitsprüfungen einer Größe an. Für jede beliebige Scheme-Zahl ist genau eines dieser Prädikate wahr.

`=` *z1 z2 z3 . . .*, [Prozedur]
`<` *x1 x2 x3 . . .*, [Prozedur]
`>` *x1 x2 x3 . . .*, [Prozedur]
`<=` *x1 x2 x3 . . .*, [Prozedur]
`>=` *x1 x2 x3 . . .*, [Prozedur]

Diese Prozeduren liefern `#t`, wenn ihre Argumente (jeweils) gleich, monoton steigend, monoton fallend, monoton nicht fallend oder monoton nicht steigend sind.

Diese Prädikate müssen transitiv sein.

Anmerkung: Die traditionelle Implementierung dieser Prädikate in Lisp-artigen Sprachen ist nicht transitiv.

Anmerkung: Auch wenn es kein Fehler ist, inexakte Zahlen mit diesen Prädikaten zu vergleichen, können die Ergebnisse unverlässlich sein, weil eine kleine Ungenauigkeit das Ergebnis beeinflussen kann; dies gilt besonders für `=` und `Zero?`. Fragen Sie im Zweifel einen Numeriker um Rat.

`zero?` *z* [Bibliotheksprozedur]
`positive?` *x* [Bibliotheksprozedur]
`negative?` *x* [Bibliotheksprozedur]
`odd?` *n* [Bibliotheksprozedur]

`even?` *n* [Bibliotheksprozedur]

Diese numerischen Prädikate prüfen, ob eine bestimmte Eigenschaft (gleich null, positiv, negativ, gerade bzw. ungerade) für eine Zahl gilt, und liefern `#t` oder `#f`. Siehe obige Anmerkung.

`max` *x1 x2 ...*, [Bibliotheksprozedur]
`min` *x1 x2 ...*, [Bibliotheksprozedur]

Diese Prozeduren liefern das Maximum oder Minimum ihrer Argumente.

```
(max 3 4)                    ==>  4    ; exakt
(max 3.9 4)                  ==>  4.0  ; inexakt
```

Anmerkung: Solange ein beliebiges Argument inexakt ist, wird auch das Ergebnis inexakt sein (außer die Prozedur kann beweisen, dass die Ungenauigkeit nicht groß genug ist, um das Ergebnis zu beeinflussen, was nur in ungewöhnlichen Implementierungen möglich ist). Wenn '`Min`' oder '`Max`' zum Vergleich von Zahlen unterschiedlicher Exaktheit benutzt wird und der numerische Wert des Ergebnisses nicht als eine inexakte Zahl dargestellt werden kann, ohne Genauigkeit zu verlieren, dann darf die Prozedur eine Verletzung einer Implementierungseinschränkung melden.

`+` *z1 ...*, [Prozedur]
`*` *z1 ...*, [Prozedur]

Diese Proceduren liefern die Summe oder das Produkt ihrer Argumente.

```
(+ 3 4)                      ==>  7
(+ 3)                        ==>  3
(+)                          ==>  0
(* 4)                        ==>  4
(*)                          ==>  1
```

`-` *z1 z2* [Prozedur]
`-` *z* [Prozedur]
`-` *z1 z2 ...*, [optionale Prozedur]
`/` *z1 z2* [Prozedur]
`/` *z* [Prozedur]
`/` *z1 z2 ...*, [optionale Prozedur]

Für zwei oder mehr Argumente liefern diese Prozeduren die Differenz oder den Quotienten ihrer Argumente, linksassoziativ. Mit einem Argument liefern sie jedoch das additiv oder multiplikativ Inverse ihres Arguments.

```
(- 3 4)                      ==>  -1
(- 3 4 5)                    ==>  -6
(- 3)                        ==>  -3
(/ 3 4 5)                    ==>  3/20
(/ 3)                        ==>  1/3
```

`abs` *x* [Bibliotheksprozedur]

'`Abs`' liefert den Absolutbetrag ihres Arguments.

```
(abs -7)                                ==>  7
```

`quotient` *n1 n2* [Prozedur]
`remainder` *n1 n2* [Prozedur]
`modulo` *n1 n2* [Prozedur]

Diese Prozeduren implementieren zahlentheoretische (ganzzahlige) Division. *n2* sollte nicht null sein. Alle drei Prozeduren liefern ganze Zahlen. Wenn *n1*/*n2* eine ganze Zahl ist:

```
(quotient n1 n2)                        ==> n1/n2
(remainder n1 n2)                       ==> 0
(modulo n1 n2)                          ==> 0
```

Wenn *n1*/*n2* keine ganze Zahl ist:

```
(quotient n1 n2)                        ==> n_q
(remainder n1 n2)                       ==> n_r
(modulo n1 n2)                          ==> n_m
```

wobei *n_q* der Wert von *n1*/*n2* gegen null gerundet ist, 0 < |*n_r*| < |*n2*|, 0 < |*n_m*| < |*n2*|, *n_r* und *n_m* sich von *n1* um ein Vielfaches von *n2* unterscheidet, *n_r* dasselbe Vorzeichen wie *n1* hat und *n_m* dasselbe Vorzeichen wie *n2* hat.

Daraus können wir schließen, dass für ganze Zahlen *n1* und *n2* mit *n2* ungleich 0 gilt, dass

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))
                                        ==>  #t
```

sofern alle in der Berechnung vorkommenden Zahlen exakt sind.

```
(modulo 13 4)                           ==>  1
(remainder 13 4)                        ==>  1

(modulo -13 4)                          ==>  3
(remainder -13 4)                       ==>  -1

(modulo 13 -4)                          ==>  -3
(remainder 13 -4)                       ==>  1

(modulo -13 -4)                         ==>  -1
(remainder -13 -4)                      ==>  -1
```

```
(remainder -13 -4.0)                    ==>  -1.0  ; inexakt
```

`gcd` *n1 ...*, [Bibliotheksprozedur]
`lcm` *n1 ...*, [Bibliotheksprozedur]

Diese Prozeduren liefern den größten gemeinsamen Teiler („greatest common divisor“) oder das kleinste gemeinsame Vielfache („least common multiple“) ihrer Argumente. Das Ergebnis ist nie negativ.

```
(gcd 32 -36)                            ==>  4
(gcd)                                   ==>  0
(lcm 32 -36)                            ==>  288
(lcm 32.0 -36)                          ==>  288.0  ; inexakt
(lcm)                                   ==>  1
```

`numerator` *q* [Prozedur]
`denominator` *q* [Prozedur]

Diese Prozeduren liefern den Zähler („numerator“) oder den Nenner („denominator“) ihres Arguments; das Ergebnis wird berechnet, als wäre das Argument als unkürzbarer Bruch dargestellt. Der Nenner ist immer positiv. Der Nenner von 0 ist auf 1 festgelegt.

```
(numerator (/ 6 4))                     ==>  3
(denominator (/ 6 4))                   ==>  2
(denominator
  (exact->inexact (/ 6 4)))             ==> 2.0
```

`floor` *x* [Prozedur]
`ceiling` *x* [Prozedur]
`truncate` *x* [Prozedur]
`round` *x* [Prozedur]

Diese Prozeduren liefern ganze Zahlen. ‘`Floor`’ („abrunden“) liefert die größte ganze Zahl, die nicht größer als *x* ist. ‘`Ceiling`’ („aufrunden“) liefert die kleinste ganze Zahl, die nicht kleiner als *x* ist. ‘`Truncate`’ („abschneiden“) liefert die am nächsten bei *x* liegende ganze Zahl, deren Absolutbetrag nicht größer als der Absolutbetrag von *x* ist. ‘`Round`’ („runden“) liefert die am nächsten bei *x* liegende ganze Zahl; sie rundet auf die nächste gerade ganze Zahl, wenn *x* genau in der Mitte zwischen zwei ganzen Zahlen liegt.

> *Begründung:* ‘`Round`’ rundet auf gerade Zahlen, um mit dem standardmäßigen Rundungsmodus, der vom IEEE-Gleitkommastandard festgelegt wurde, übereinzustimmen.
>
> *Anmerkung:* Wenn das Argument einer dieser Prozeduren inexakt ist, dann wird das Ergebnis auch inexakt sein. Wird ein exakter

Wert benötigt, sollte das Ergebnis an die Prozedur '`Inexact->exact`' weitergereicht werden.

```
(floor -4.3)                    ==>  -5.0
(ceiling -4.3)                  ==>  -4.0
(truncate -4.3)                 ==>  -4.0
(round -4.3)                    ==>  -4.0

(floor 3.5)                     ==>  3.0
(ceiling 3.5)                   ==>  4.0
(truncate 3.5)                  ==>  3.0
(round 3.5)                     ==>  4.0  ; inexakt

(round 7/2)                     ==>  4    ; exakt
(round 7)                       ==>  7
```

`rationalize` *x y* [Bibliotheksprozedur]

'`Rationalize`' liefert die *einfachste* rationale Zahl, die sich von *x* nicht um mehr als *y* unterscheidet. Eine rationale Zahl r_1 ist *einfacher* als eine andere rationale Zahl r_2, wenn r_1 = p_1/q_1 und r_2 = p_2/q_2 (unkürzbare Brüche) und |p_1|<= |p_2| und |q_1| <= |q_2|. Somit ist 3/5 einfacher als 4/7. Obwohl nicht alle rationalen Zahlen in dieser Ordnung vergleichbar sind (betrachten Sie 2/7 und 3/5), enthält jedes beliebige Intervall eine rationale Zahl, die einfacher ist als jede andere rationale Zahl in diesem Intervall (die einfachere 2/5 liegt zwischen 2/7 und 3/5). Beachten Sie, dass 0 = 0/1 die einfachste rationale Zahl von allen ist.

```
(rationalize
  (inexact->exact .3) 1/10)     ==> 1/3    ; exakt
(rationalize .3 1/10)           ==> #i1/3  ; inexakt
```

`exp` *z* [Prozedur]
`log` *z* [Prozedur]
`sin` *z* [Prozedur]
`cos` *z* [Prozedur]
`tan` *z* [Prozedur]
`asin` *z* [Prozedur]
`acos` *z* [Prozedur]
`atan` *z* [Prozedur]
`atan` *y x* [Prozedur]

Diese Prozeduren sind Teil jeder Implementierung, die allgemeine reelle Zahlen unterstützt; sie berechnen die üblichen transzendenten Funktionen. '`Log`' berechnet den natürlichen Logarithmus von *z* (nicht den Logarithmus zur Basis 10). '`Asin`', '`Acos`' und '`Atan`' berechnen jeweils den Arkussinus (sin^-1), Arkuskosinus (cos^-1) und Arkustangens (tan^-1). Die Variante mit zwei Argumenten von '`Atan`' berechnet `(angle`

`(make-rectangular` *x y*`))` (siehe unten), selbst in Implementierungen, die komplexe Zahlen nicht allgemein unterstützen.

Im Allgemeinen sind die mathematischen Funktionen log, arcsine, arccosine und arctangent verschiedenartig definiert. Der Wert von log z ist definiert als diejenige Zahl, deren Imaginärteil im Bereich von -Pi (nicht einschließlich) bis Pi (einschließlich) liegt. log 0 ist undefiniert. Mit dem so definierten log ergeben sich die Werte von sin^-1 z, cos^-1 z und tan^-1 z anhand der folgenden Formeln:

sin^-1 z = -i log (i z + sqrt(1 - z^2))

cos^-1 z = pi / 2 - sin^-1 z

tan^-1 z = (log (1 + i z) - log (1 - i z)) / (2 i)

Die obige Spezifikation folgt [CLtL], wo wiederum [Penfield81] zitiert wird; konsultieren Sie diese Quellen für detailliertere Diskussionen von Verzweigungsschnitten, Randbedingungen und die Implementierung dieser Funktionen. Wenn möglich liefern diese Prozeduren ein reelles Ergebnis für ein reelles Argument.

`sqrt` *z* [Prozedur]

Liefert die Haupt-Quadratwurzel von *z*. Das Ergebnis wird entweder einen positiven Realteil haben, oder null Realteil und nicht-negativen Imaginärteil haben.

`expt` *z1 z2* [Prozedur]

Liefert *z1* zur Potenz *z2*. Für z_1 ~= 0

z_1^z_2 = e^z_2 log z_1

0^z ist 1 wenn z = 0, und 0 sonst.

`make-rectangular` *x1 x2* [Prozedur]
`make-polar` *x3 x4* [Prozedur]
`real-part` *z* [Prozedur]
`imag-part` *z* [Prozedur]
`magnitude` *z* [Prozedur]
`angle` *z* [Prozedur]

Diese Prozeduren sind Teil jeder Implementierung, die allgemeine komplexe Zahlen unterstützt. Angenommen *x1*, *x2*, *x3* und *x4* sind reelle Zahlen und *z* ist eine komplexe Zahl, so dass

z = *x1* + *x2*i = *x3* . e^i *x4*

Dann

```
(make-rectangular x1 x2)          ==> z
(make-polar x3 x4)                ==> z
(real-part z)                     ==> x1
(imag-part z)                     ==> x2
(magnitude z)                     ==> |x3|
(angle z)                         ==> x_Winkel
```

wobei -Pi < x_Winkel <= Pi mit x_Winkel = *x4* + 2Pi n für eine ganze Zahl n.

Begründung: '`Magnitude`' ist dasselbe wie `Abs` für ein reelles Argument, aber '`Abs`' muss in allen Implementierungen vorhanden sein, während

'`Magnitude`' nur in Implementierungen vorhanden sein muss, die allgemeine komplexe Zahlen unterstützen.

`exact->inexact` *z* [Prozedur]
`inexact->exact` *z* [Prozedur]

'`Exact->inexact`' liefert eine inexakte Darstellung von *z*. Der gelieferte Wert ist die inexakte Zahl, die numerisch dem Argument am nächsten liegt. Wenn ein exaktes Argument kein inexaktes Gegenstück in vernünftiger Nähe hat, dann darf eine Verletzung einer Implementierungseinschränkung gemeldet werden.

'`Inexact->exact`' liefert eine exakte Darstellung von *z*. Der gelieferte Wert ist die exakte Zahl, die numerisch dem Argument am nächsten liegt. Wenn ein inexaktes Argument kein exaktes Gegenstück in vernünftiger Nähe hat, dann darf eine Verletzung einer Implementierungseinschränkung gemeldet werden.

Diese Prozeduren implementieren die natürliche Eins-zu-eins-Korrespondenz zwischen exakten und inexakten ganzen Zahlen innerhalb eines implementierungsabhängigen Wertebereichs. Siehe den Abschnitt Abschnitt 6.2.3 [Implementierungseinschränkungen], Seite 43.

6.2.6 Numerische Ein- und Ausgabe

`number->string` *z* [Prozedur]
`number->string` *z radix* [Prozedur]

Radix muss eine exakte ganze Zahl sein, entweder 2, 8, 10 oder 16. Wenn sie nicht angegeben wird, nimmt *Radix* standardmäßig den Wert 10 an. Die Prozedur '`Number->string`' nimmt eine Zahl und eine Radix und liefert als Zeichenkette eine externe Darstellung der gegebenen Zahl zur angegebenen Basis (der Radix), so dass

```
(let ((zahl Zahl)
      (radix Radix))
  (eqv? zahl
        (string->number (number->string zahl
                                        radix)
                        radix)))
```

wahr ist. Es ist ein Fehler, wenn kein mögliches Ergebnis diesen Ausdruck wahr macht.

Wenn *z* inexakt ist, die Radix 10 beträgt und obiger Ausdruck durch ein Ergebnis erfüllt werden kann, das ein Dezimaltrennzeichen enthält, dann enthält das Ergebnis einen Dezimalpunkt als Dezimaltrennzeichen und wird mit der geringstmöglichen Anzahl Ziffern ausgedrückt (den Exponenten und folgende Nullen nicht eingeschlossen), die nötig sind, um obigen Ausdruck wahr zu machen [howtoprint], [howtoread], andernfalls ist das Format des Ergebnisses unbestimmt.

Das von '`Number->string`' gelieferte Ergebnis enthält niemals ein explizites Radix-Präfix.

Anmerkung: Der Fehlerfall kann nur eintreten, wenn *z* keine komplexe Zahl oder eine komplexe Zahl mit nicht-rationalem reellem oder imaginären Anteil ist.

Begründung: Wenn *z* eine inexakte Zahl ist, die mit Gleitkommazahlen dargestellt ist, und die Radix 10 ist, dann wird obiger Ausdruck normalerweise von einem Ergebnis mit Dezimaltrennzeichen erfüllt. Der unbestimmte Fall erlaubt Unendlichkeiten, NaNs und Nichtgleitkommadarstellungen.

`string->number` *string* [Prozedur]
`string->number` *string radix* [Prozedur]

Liefert eine Zahl mit der genauestmöglichen Darstellung, die von der gegebenen Zeichenkette *String* ausgedrückt wird. *Radix* muss eine exakte ganze Zahl, entweder 2, 8, 10 oder 16, sein. Wenn angegeben, ist *Radix* eine Standardbasis, gegenüber der ein ausdrücklich angegebenes Radix-Präfix in *String* Vorrang hat (z.B. `"#o177"`). Wenn *Radix* nicht angegeben wird, ist die Standardbasis 10. Wenn *String* keine syntaktisch gültige Notation einer Zahl ist, dann liefert '`String->number`' den Wert `#f`.

```
(string->number "100")                ==>  100
(string->number "100" 16)             ==>  256
(string->number "1e2")                ==>  100.0
(string->number "15##")               ==>  1500.0
```

Anmerkung: Die Definitionsmenge von '`String->number`' darf auf folgende Arten von Implementierungen eingeschränkt werden. '`String->number`' darf `#f` liefern, wann immer *String* ein ausdrückliches Radix-Präfix enthält. Wenn alle von einer Implementierung unterstützten Zahlen reell sind, dann darf '`String->number`' den Wert `#f` liefern, wann immer *String* polare oder kartesische Notationen für komplexe Zahlen benutzt. Wenn alle Zahlen ganze Zahlen sind, darf '`String->number`' den Wert `#f` liefern, wann immer die Bruchdarstellung benutzt wird. Wenn alle Zahlen exakt sind, dann darf '`String->number`' den Wert `#f` liefern, wann immer eine Exponentenmarkierung oder ein ausdrückliches Exaktheitspräfix benutzt wird, oder wenn ein `#` anstelle einer Ziffer vorkommt. Wenn alle inexakten Zahlen ganze Zahlen sind, dann darf '`String->number`' den Wert `#f` liefern, wann immer ein Dezimaltrennzeichen benutzt wird.

6.3 Andere Datentypen

Dieser Abschnitt beschreibt Operationen auf manchen von Schemes nicht-numerischen Datentypen: booleschen Werten („booleans“), Paaren („pairs“), Listen („lists“), Symbolen („symbols“), Zeichen („characters“), Zeichenketten („strings“) und Vektoren („vectors“).

6.3.1 Boolesche Werte

Die standardmäßigen booleschen Objekte für wahr und falsch werden geschrieben als `#t` und `#f`. Was aber wirklich wichtig ist, sind die Objekte, die die bedingten Ausdrücke von

Scheme (‘`If`’, ‘`Cond`’, ‘`And`’, ‘`Or`’, ‘`Do`’) als wahr oder falsch behandeln. Die Formulierung „ein wahrer Wert“ (oder manchmal kurz „wahr“) bedeutet ein beliebiges Objekt, das von den bedingten Ausdrücken als wahr behandelt wird, und die Formulierung „ein falscher Wert“ (oder „falsch“) bedeutet ein beliebiges Objekt, das von den bedingten Ausdrücken als falsch behandelt wird.

Von allen Standard-Scheme-Werten zählt nur `#f` als falsch in bedingten Ausdrücken. Außer `#f` zählen alle Standard-Scheme-Werte, einschließlich `#t`, Paaren, der leeren Liste, Symbolen, Zahlen, Zeichenketten, Vektoren und Prozeduren als wahr.

> *Anmerkung:* Programmierer, die an andere Dialekte von Lisp gewöhnt sind, sollten beachten, dass Scheme sowohl `#f` als auch die leere Liste vom Symbol `nil` unterscheidet.

Boolesche Konstante werden zu sich selbst ausgewertet, also müssen sie in Programmen nicht maskiert werden.

```
#t                                      ==>  #t
#f                                      ==>  #f
'#f                                     ==>  #f
```

`not` *obj* [Bibliotheksprozedur]

‘`Not`’ liefert `#t`, wenn *Obj* falsch ist, und liefert sonst `#f`.

```
(not #t)                                ==>  #f
(not 3)                                 ==>  #f
(not (list 3))                          ==>  #f
(not #f)                                ==>  #t
(not '())                               ==>  #f
(not (list))                            ==>  #f
(not 'nil)                              ==>  #f
```

`boolean?` *obj* [Bibliotheksprozedur]

‘`Boolean?`’ liefert `#t`, wenn *Obj* entweder `#t` oder `#f` ist, und liefert sonst `#f`.

```
(boolean? #f)                           ==>  #t
(boolean? 0)                            ==>  #f
(boolean? '())                          ==>  #f
```

6.3.2 Paare und Listen

Ein *Paar* (manchmal auch ein *gepunktetes Paar*, „dotted pair“) ist eine Verbundsstruktur mit zwei Komponenten, die als die Car- und Cdr-Komponenten bezeichnet werden (aus historischen Gründen). Paare werden durch die Prozedur ‘`Cons`’ erzeugt. Auf die Car- und Cdr-Komponenten wird durch die Prozeduren ‘`Car`’ und ‘`Cdr`’ zugegriffen. Die Car- und Cdr-Komponenten werden durch die Prozeduren ‘`Set-car!`’ und ‘`Set-cdr!`’ zugewiesen.

Paare werden vorrangig benutzt, um Listen darzustellen. Eine Liste kann rekursiv als entweder die leere Liste oder ein Paar, dessen Cdr eine Liste ist, definiert werden. Genauer ist die Menge der Listen definiert als die kleinste Menge X, so dass gilt:

- Die leere Liste ist in X.
- Wenn *Liste* in X ist, ist auch jedes Paar in X, dessen Cdr-Komponente *Liste* enthält.

Die Objekte in den Car-Komponenten aufeinanderfolgender Paare einer Liste sind die Elemente der Liste. Zum Beispiel ist eine zweielementige Liste ein Paar, deren Car das erste Element ist und deren Cdr ein Paar ist, deren Car das zweite Element und deren Cdr die leere Liste ist. Die Länge einer Liste ist die Anzahl der Elemente, was dasselbe ist wie die Anzahl der Paare.

Die leere Liste ist ein besonderes Objekt mit seinem eigenen Typ (sie ist kein Paar); sie hat keine Elemente und ihre Länge ist null.

> *Anmerkung:* Die obigen Definitionen haben zur Folge, dass alle Listen eine endliche Länge haben und auf die leere Liste enden.

Die allgemeinste Notation (externe Darstellung) für Scheme-Paare ist die „gepunktete“ Notation '`(`*c1* `.` *c2*`)`', wobei *c1* der Wert der Car-Komponente und *c2* der Wert der Cdr-Komponente ist. Zum Beispiel ist '`(4 . 5)`' ein Paar, deren Car 4 ist und deren Cdr 5 ist. Beachten Sie, dass '`(4 . 5)`' die externe Darstellung eines Paares ist, nicht ein Ausdruck, der zu einem Paar ausgewertet wird.

Eine optimiertere Notation kann für Listen benutzt werden: Die Elemente der Liste werden einfach von runden Klammern umschlossen und durch Leerzeichen getrennt. Die leere Liste wird als `()` geschrieben. Zum Beispiel sind

```
(a b c d e)
```

und

```
(a . (b . (c . (d . (e . ()))))))
```

äquivalente Notationen für eine Liste von Symbolen.

Eine Kette aus Paaren, die nicht auf die leere Liste endet, wird *unechte Liste* genannt. Beachten Sie, dass eine unechte Liste keine Liste ist. Die Listennotation und die gepunktete Notation können zur Darstellung unechter Listen kombiniert werden:

```
(a b c . d)
```

ist äquivalent zu

```
(a . (b . (c . d)))
```

Ob ein bestimmtes Paar eine Liste ist, hängt davon ab, was in der Cdr-Komponente gespeichert ist. Wenn die Prozedur `Set-cdr!` benutzt wird, kann ein Objekt im einen Moment eine Liste sein und im nächsten nicht mehr:

```
(define x (list 'a 'b 'c))
(define y x)
y                                       ==>  (a b c)
(list? y)                               ==>  #t
(set-cdr! x 4)                          ==>  unbestimmt
x                                       ==>  (a . 4)
(eqv? x y)                              ==>  #t
y                                       ==>  (a . 4)
(list? y)                               ==>  #f
(set-cdr! x x)                          ==>  unbestimmt
(list? x)                               ==>  #f
```

Innerhalb literaler Ausdrücke und in Darstellungen von mit der Prozedur `Read` gelesenen Objekten bezeichnen die Formen `'`<Datenelement>, `‘`<Datenelement>, `,`<Datenelement> und `,@`<Datenelement> zweielementige Listen, deren erstes Element jeweils die Symbole `Quote`, `Quasiquote`, `Unquote` und `Unquote-splicing` sind. Das zweite Element ist in allen Fällen <Datenelement>. Diese Konvention wird unterstützt, damit beliebige Scheme-Programme als Listen dargestellt werden können. Das heißt, dass laut der Grammatik von Scheme jeder <Ausdruck> auch ein <Datenelement> ist (siehe den Abschnitt siehe Abschnitt 7.1.2 [Externe Darstellung], Seite 84). Dies erlaubt unter Anderem, die Prozedur ‘`Read`’ zu benutzen, um Scheme-Programme grammatikalisch zu analysieren. Siehe den Abschnitt Abschnitt 3.3 [Externe Darstellungen], Seite 12.

`pair?` *obj* [Prozedur]

‘`Pair?`’ liefert `#t`, wenn *Obj* ein Paar ist, und liefert andernfalls `#f`.

```
(pair? '(a . b))                        ==>  #t
(pair? '(a b c))                        ==>  #t
(pair? '())                             ==>  #f
(pair? '#(a b))                         ==>  #f
```

`cons` *obj1 obj2* [Prozedur]

Liefert ein neu zugeteiltes Paar, dessen Car *Obj1* ist und dessen Cdr *Obj2* ist. Es wird garantiert, dass das Paar sich (im Sinn von ‘`Eqv?`’) von jedem existierenden Objekt unterscheidet.

```
(cons 'a '())                           ==>  (a)
(cons '(a) '(b c d))                    ==>  ((a) b c d)
(cons "a" '(b c))                       ==>  ("a" b c)
(cons 'a 3)                             ==>  (a . 3)
(cons '(a b) 'c)                        ==>  ((a b) . c)
```

`car` *paar* [Prozedur]

Liefert den Inhalt der Car-Komponente von *Paar*. Beachten Sie, dass es ein Fehler ist, das Car der leeren Liste zu nehmen.

```
(car '(a b c))                          ==>  a
```

```
(car '((a) b c d))                    ==>  (a)
(car '(1 . 2))                        ==>  1
(car '())                             ==>  Fehler
```

`cdr` *paar* [Prozedur]

Liefert den Inhalt der Cdr-Komponente von *Paar*. Beachten Sie, dass es ein Fehler ist, den Cdr der leeren Liste zu nehmen.

```
(cdr '((a) b c d))                    ==>  (b c d)
(cdr '(1 . 2))                        ==>  2
(cdr '())                             ==>  Fehler
```

`set-car!` *paar obj* [Prozedur]

Speichert *Obj* in der Car-Komponente von *Paar*. Der durch '`Set-car!`' gelieferte Wert ist unbestimmt.

```
(define (f) (list 'keine-konstante-liste))
(define (g) '(konstante-liste))
(set-car! (f) 3)                      ==>  unbestimmt
(set-car! (g) 3)                      ==>  Fehler
```

`set-cdr!` *paar obj* [Prozedur]

Speichert *Obj* in der Cdr-Komponente von *Paar*. Der durch '`Set-cdr!`' gelieferte Wert ist unbestimmt.

`caar` *paar* [Bibliotheksprozedur]
`cadr` *paar* [Bibliotheksprozedur]
`...` [...]
`cdddar` *paar* [Bibliotheksprozedur]
`cddddr` *paar* [Bibliotheksprozedur]

Diese Prozeduren sind Zusammensetzungen von '`Car`' und '`Cdr`', wobei zum Beispiel '`Caddr`' definiert werden könnte als

```
(define caddr (lambda (x) (car (cdr (cdr x)))))
```

.

Beliebige Zusammensetzungen, bis zur Tiefe vier, werden angeboten. Es gibt insgesamt achtundzwanzig solche Prozeduren.

`null?` *obj* [Bibliotheksprozedur]

Liefert `#t`, wenn *Obj* die leere Liste ist, und liefert sonst `#f`.

`list?` *obj* [Bibliotheksprozedur]

Liefert `#t`, wenn *Obj* eine Liste ist, und liefert sonst `#f`. Per Definition haben alle Listen endliche Länge und enden auf die leere Liste.

```
(list? '(a b c))                      ==>  #t
```

```
(list? '())                    ==>  #t
(list? '(a . b))               ==>  #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))                   ==>  #f
```

`list` *obj ...*, [Bibliotheksprozedur]

Liefert eine neu zugeteilte Liste ihrer Argumente.

```
(list 'a (+ 3 4) 'c)           ==>  (a 7 c)
(list)                         ==>  ()
```

`length` *liste* [Bibliotheksprozedur]

Liefert die Länge von *Liste*.

```
(length '(a b c))              ==>  3
(length '(a (b) (c d e)))      ==>  3
(length '())                   ==>  0
```

`append` *liste ...*, [Bibliotheksprozedur]

Liefert eine Liste, die aus den Elementen der ersten *Liste*, gefolgt von den Elementen der anderen *Liste*n, besteht.

```
(append '(x) '(y))             ==>  (x y)
(append '(a) '(b c d))         ==>  (a b c d)
(append '(a (b)) '((c)))       ==>  (a (b) (c))
```

Die sich so ergebende Liste wird immer neu zugeteilt, außer dass sie ihre Struktur mit dem letzten *Liste*nargument teilt. Das letzte Argument darf tatsächlich ein beliebiges Objekt sein; eine unechte Liste ergibt sich, wenn das letzte Argument keine echte Liste ist.

```
(append '(a b) '(c . d))       ==>  (a b c . d)
(append '() 'a)                ==>  a
```

`reverse` *liste* [Bibliotheksprozedur]

Liefert eine neu zugeteilte Liste, die aus den Elementen der *Liste* in umgekehrter Reihenfolge besteht.

```
(reverse '(a b c))             ==>  (c b a)
(reverse '(a (b c) d (e (f))))
          ==>  ((e (f)) d (b c) a)
```

`list-tail` *liste k* [Bibliotheksprozedur]

Liefert die Teilliste von *Liste*, die man durch Weglassen der ersten *k* Elemente erhält. Es ist ein Fehler, wenn *Liste* weniger als *k* Elemente hat. '`List-tail`' könnte definiert werden als

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))
```

`list-ref` *liste k* [Bibliotheksprozedur]

Liefert das Element *k* der *Liste*. (Dieses ist dasselbe wie das Car von `(list-tail` *liste k*`)`.) Es ist ein Fehler, wenn die *Liste* weniger als *k* Elemente hat.

```
(list-ref '(a b c d) 2)                ==>  c
(list-ref '(a b c d)
          (inexact->exact (round 1.8)))
          ==>  c
```

`memq` *obj liste* [Bibliotheksprozedur]
`memv` *obj liste* [Bibliotheksprozedur]
`member` *obj liste* [Bibliotheksprozedur]

Diese Prozeduren liefern die erste Teilliste der *Liste*, deren Car *Obj* ist, wobei die Teillisten der *Liste* die von `(list-tail` *Liste k*`)` gelieferten nicht-leeren Listen für *k* kleiner als die Länge der Liste sind. Wenn *Obj* in *Liste* nicht vorkommt, dann wird `#f` (nicht die leere Liste) geliefert. '`Memq`' benutzt '`Eq?`' zum Vergleichen von *Obj* mit den Elementen der *Liste*, während '`Memv`' die Prozedur '`Eqv?`' und '`Member`' die Prozedur '`Equal?`' benutzt.

```
(memq 'a '(a b c))                     ==>  (a b c)
(memq 'b '(a b c))                     ==>  (b c)
(memq 'a '(b c d))                     ==>  #f
(memq (list 'a) '(b (a) c))            ==>  #f
(member (list 'a)
        '(b (a) c))                    ==>  ((a) c)
(memq 101 '(100 101 102))              ==>  unbestimmt
(memv 101 '(100 101 102))              ==>  (101 102)
```

`assq` *obj aliste* [Bibliotheksprozedur]
`assv` *obj aliste* [Bibliotheksprozedur]
`assoc` *obj aliste* [Bibliotheksprozedur]

Aliste (für „assoziative Liste“) muss eine Liste von Paaren sein. Diese Prozeduren finden das erste Paar in *Aliste*, dessen Car-Komponente *Obj* ist, und liefern dieses

Paar. Wenn kein Paar in *Aliste* das *Obj* als sein Car hat, dann wird `#f` geliefert (nicht die leere Liste). '`Assq`' benutzt '`Eq?`' zum Vergleich von *Obj* mit den Car-Komponenten der Paare in der *Aliste*, während '`Assv`' die Prozedur '`Eqv?`' und '`Assoc`' die Prozedur '`Equal?`' benutzt.

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                              ==>  (a 1)
(assq 'b e)                              ==>  (b 2)
(assq 'd e)                              ==>  #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                                         ==>  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                                         ==>  ((a))
(assq 5 '((2 3) (5 7) (11 13)))
                                         ==>  unbestimmt
(assv 5 '((2 3) (5 7) (11 13)))
                                         ==>  (5 7)
```

Begründung: Obwohl sie normalerweise als Prädikate benutzt werden, haben '`Memq`', '`Memv`', '`Member`', '`Assq`', '`Assv`' und '`Assoc`' keine Fragezeichen in ihren Namen, weil sie nützliche Werte statt bloß `#t` oder `#f` liefern.

6.3.3 Symbole

Symbole sind Objekte, deren Nützlichkeit auf der Tatsache beruht, dass zwei Symbole genau dann identisch sind (im Sinn von '`Eqv?`'), wenn ihre Namen auf gleiche Weise geschrieben werden. Dies ist genau die Eigenschaft, die gebraucht wird, um Bezeichner in Programmen darzustellen, daher benutzen die meisten Implementierungen von Scheme intern Symbole als Bezeichner. Symbole sind nützlich für viele andere Anwendungen, zum Beispiel können sie so wie Aufzählungswerte in Pascal benutzt werden.

Die Regeln, wie ein Symbol geschrieben wird, sind genau die gleichen wie die Regeln, wie ein Bezeichner geschrieben wird, siehe die Abschnitte Abschnitt 2.1 [Bezeichner], Seite 8, und Abschnitt 7.1.1 [Lexikalische Struktur], Seite 82.

Es wird garantiert, dass jedes Symbol, was als Teil eines literalen Ausdrucks geliefert wurde oder was von der Prozedur '`Read`' gelesen und anschließend durch die Prozedur '`Write`' geschrieben wurde, wieder als dasselbe Symbol einlesbar ist (im Sinne von '`Eqv?`'). Die Prozedur '`String->symbol`' kann jedoch Symbole erstellen, für die diese Schreib-/Lese-Invarianz nicht gelten könnte, weil ihre Namen Sonderzeichen oder Buchstaben in nicht standardmäßiger Groß- und Kleinschreibung enthalten.

Anmerkung: Manche Implementierungen von Scheme haben eine Funktionalität, die als Slashifizierung („slashification“) bekannt ist, um Schreib-/Lese-Invarianz für alle Symbole zu garantieren, aber historisch war die wichtigste Anwendung dieser Funktionalität, das Fehlen eines Datentyps für Zeichenketten auszugleichen.

Manche Implementierungen haben auch „nicht internierte Symbole“, welche Schreib-/Lese-Invarianz unmöglich machen, selbst in Implementierungen mit

Slashifizierung, und auch Ausnahmen zu der Regel bilden, dass zwei Symbole genau dann dasselbe sind, wenn ihre Namen gleich geschrieben werden.

`symbol?` *obj* [Prozedur]

Liefert `#t`, wenn *Obj* ein Symbol ist, und liefert sonst `#f`.

```
(symbol? 'foo)                          ==>  #t
(symbol? (car '(a b)))                  ==>  #t
(symbol? "bar")                         ==>  #f
(symbol? 'nil)                          ==>  #t
(symbol? '())                           ==>  #f
(symbol? #f)                            ==>  #f
```

`symbol->string` *symbol* [Prozedur]

Liefert den Namen des *Symbol*s als eine Zeichenkette. Wenn das Symbol ein Teil eines Objekts war, das als der Wert eines literalen Ausdrucks geliefert wurde (Abschnitt siehe Abschnitt 4.1.2 [Literale Ausdrücke], Seite 17) oder von einem Aufruf der Prozedur '`Read`' geliefert wurde, und sein Name Buchstabenzeichen enthält, dann wird die gelieferte Zeichenkette Zeichen in der von der Implementierung bevorzugten Groß- und Kleinschreibung enthalten—manche Implementierungen werden Großbuchstaben bevorzugen, andere Kleinschreibung. Wenn das Symbol von '`String->symbol`' geliefert wurde, wird die Groß- und Kleinschreibung der Zeichen der gelieferten Zeichenkette dieselbe sein wie die, die an '`String->symbol`' übergeben wurde. Es ist ein Fehler, Veränderungsprozeduren wie `String-set!` auf von dieser Prozedur gelieferte Zeichenketten anzuwenden.

Die folgenden Beispiele nehmen an, dass die standardmäßige Groß-und-Kleinschreibung der Implementierung Kleinschreibung ist:

```
(symbol->string 'fliegende-Fische)
                                        ==>  "fliegende-fische"
(symbol->string 'Martin)                ==>  "martin"
(symbol->string
   (string->symbol "Malvina"))
                                        ==>  "Malvina"
```

`string->symbol` *string* [Prozedur]

Liefert das Symbol, dessen Name *String* ist. Diese Prozedur kann Symbole erzeugen, deren Namen Sonderzeichen enthalten oder Buchstaben in nicht standardmäßiger Groß- und Kleinschreibung enthalten, aber es ist meistens eine schlechte Idee, solche Symbole zu erzeugen, denn in manchen Implementierungen von Scheme können sie nicht als sie selbst eingelesen werden. Siehe '`Symbol->string`'.

Die folgenden Beispiele nehmen an, dass die standardmäßige Groß- und Kleinschreibung der Implementierung Kleinschreibung ist:

```
(eq? 'mISSISSIppi 'mississippi)
          ==>  #t
```

```
(string->symbol "mISSISSIppi")
          ==>
  das Symbol namens "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt"))
          ==>  #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))
          ==>  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
          ==>  #t
```

6.3.4 Zeichen

Zeichen sind Objekte, die gedruckte Zeichen wie Buchstaben und Ziffern darstellen. Zeichen werden mit der Notation #\<Zeichen> oder #\<Zeichenname> geschrieben. Zum Beispiel:

```
#\a         ; Kleinbuchstabe
#\A         ; Großbuchstabe
#\(         ; linke runde Klammer
#\          ; das Leerzeichen
#\space     ; die bevorzugte Art, ein Leerzeichen zu schreiben
#\newline
            ; das Zeilenvorschubs-Zeichen („newline“)
```

Groß- und Kleinschreibung hat Auswirkungen in #\<Zeichen>, aber nicht in #\<Zeichenname>.

Wenn <Zeichen> in #\<Zeichen> ein Buchstabe ist, dann muss das auf <Zeichen> folgende Zeichen ein Trennzeichen wie ein Leerzeichen oder eine runde Klammer sein. Diese Regel löst den mehrdeutigen Fall auf, dass zum Beispiel die Folge von Zeichen „`#\space`“ als entweder eine Darstellung des Leerzeichens oder als eine Darstellung des Zeichens „`#\s`“ gefolgt von einer Darstellung des Symbols „`pace`“ verstanden werden könnte.

Zeichen, die in der Notation #\ geschrieben werden, sind selbtauswertend. Das heißt, sie müssen in Programmen nicht maskiert werden.

Manche der Prozeduren, die auf Zeichen arbeiten, ignorieren den Unterschied zwischen Groß- und Kleinschreibung. Die Prozeduren, die Groß- und Kleinschreibung ignorieren, haben „`-ci`“ (für „case insensitive“) in ihre Namen eingebettet.

`char?` *obj* [Prozedur]
: Liefert `#t`, wenn *Obj* ein Zeichen ist, sonst liefert sie `#f`.

`char=?` *char1 char2* [Prozedur]
`char<?` *char1 char2* [Prozedur]

`char>?` *char1 char2* [Prozedur]
`char<=?` *char1 char2* [Prozedur]
`char>=?` *char1 char2* [Prozedur]

Durch diese Prozeduren wird der Menge der Zeichen eine totale Ordnung auferlegt. Es wird garantiert, dass gemäß dieser Ordnung gilt:

- Die Großbuchstaben sind in der richtigen Reihenfolge. Zum Beispiel liefert '`(char<? #\A #\B)`' den Wert `#t`.
- Die Kleinbuchstaben sind in der richtigen Reihenfolge. Zum Beispiel liefert '`(char<? #\a #\b)`' den Wert `#t`.
- Die Ziffern sind in der richtigen Reihenfolge. Zum Beispiel liefert '`(char<? #\0 #\9)`' den Wert `#t`.
- Entweder kommen alle Ziffern vor allen Großbuchstaben, oder umgekehrt.
- Entweder kommen alle Ziffern nach allen Großbuchstaben, oder umgekehrt.

Manche Implementierungen können diese Prozeduren so verallgemeinern, dass sie mehr als zwei Argumente nehmen, genau wie bei den entsprechenden numerischen Prädikaten.

`char-ci=?` *char1 char2* [Bibliotheksprozedur]
`char-ci<?` *char1 char2* [Bibliotheksprozedur]
`char-ci>?` *char1 char2* [Bibliotheksprozedur]
`char-ci<=?` *char1 char2* [Bibliotheksprozedur]
`char-ci>=?` *char1 char2* [Bibliotheksprozedur]

Diese Prozeduren sind ähnlich wie '`Char=?`' et cetera, aber sie behandeln Groß- und Kleinbuchstaben, als wären sie dasselbe. Zum Beispiel liefert '`(char-ci=? #\A #\a)`' den Wert `#t`. Manche Implementierungen können diese Prozeduren so verallgemeinern, dass sie mehr als zwei Argumente nehmen, genau wie bei den entsprechenden numerischen Prädikaten.

`char-alphabetic?` *char* [Bibliotheksprozedur]
`char-numeric?` *char* [Bibliotheksprozedur]
`char-whitespace?` *char* [Bibliotheksprozedur]
`char-upper-case?` *buchstabe* [Bibliotheksprozedur]
`char-lower-case?` *buchstabe* [Bibliotheksprozedur]

Diese Prozeduren liefern `#t`, wenn ihre Argumente jeweils Buchstaben, Zahlen, Leerraum, Großbuchstaben oder Kleinbuchstaben sind, sonst liefern sie `#f`. Die folgenden Anmerkungen, welche speziell für den ASCII-Zeichensatz gelten, sind nur als Orientierung gedacht: Die Buchstabenzeichen sind die 52 Groß- und Kleinbuchstaben. Die Ziffernzeichen sind die zehn Dezimalziffern. Die Leerraumzeichen sind das Leerzeichen, das Tabulatorzeichen, der Zeilenvorschub, der Seitenvorschub und der Wagenrücklauf.

`char->integer` *char* [Prozedur]
`integer->char` *n* [Prozedur]

Gegeben ein Zeichen liefert '`Char->integer`' eine exakte ganzzahlige Darstellung des Zeichens. Gegeben eine exakte ganze Zahl, die das Bild eines Zeichens unter '`Char->integer`' ist, liefert '`Integer->char`' dieses Zeichen. Diese Prozeduren implementieren ordnungserhaltende Isomorphismen zwischen der Menge an Zeichen unter

der Ordnung von `Char<=?` und einer der Teilmengen der ganzen Zahlen unter der Ordnung von '`<=`'. Das heißt, wenn

```
(char<=? a b) ⇒ #t  und  (<= x y) ⇒ #t
```

und x und y in der Definitionsmenge von '`Integer->char`' sind, dann

```
(<= (char->integer a)
    (char->integer b))                ==>  #t

(char<=? (integer->char x)
         (integer->char y))           ==>  #t
```

`char-upcase` *char* [Bibliotheksprozedur]
`char-downcase` *char* [Bibliotheksprozedur]

Diese Prozeduren liefern ein Zeichen *Char2*, für das '`(char-ci=? char char2)`' gilt. Zudem ist, wenn *Char* ein Buchstabe ist, das Ergebnis von '`Char-upcase`' ein Großbuchstabe und das Ergebnis von '`Char-downcase`' ein Kleinbuchstabe.

6.3.5 Zeichenketten

Zeichenketten sind Folgen von Zeichen. Zeichenketten werden als Folge von Zeichen geschrieben, die in doppelte Anführungszeichen eingeschlossen sind ('`"`'). Doppelte Anführungszeichen können nur in eine Zeichenkette geschrieben werden, indem man sie mit einem Backslash (\) maskiert, wie hier:

```
"Das Wort \"Rekursion\" hat viele Bedeutungen."
```

Ein Backslash kann nur in einer Zeichenkette geschrieben werden, indem man ihn mit einem anderen Backslash maskiert. Scheme legt die Wirkung eines Backslashs innerhalb einer Zeichenkette, auf den keine doppelten Anführungszeichen oder ein Backslash folgt, nicht fest.

Eine konstante Zeichenkette darf sich von einer Zeile bis auf die nächste erstrecken, aber der genaue Inhalt einer solchen Zeichenkette ist unbestimmt.

Die *Länge* einer Zeichenkette ist die Anzahl der Zeichen, die sie enthält. Diese Anzahl ist eine exakte, nicht negative ganze Zahl, die von der Erstellung einer Zeichenkette an fest ist. Die *gültigen Indizes* einer Zeichenkette sind die exakten, nicht negativen ganzen Zahlen, die kleiner als die Länge der Zeichenkette sind. Das erste Zeichen einer Zeichenkette hat den Index 0, das zweite hat den Index 1, und so weiter.

In Formulierungen wie „die Zeichen von *String* vom Index *Anfang* bis zum Index *Ende*“ ist damit gemeint, dass der Index *Anfang* dazugehört und der Index *Ende* nicht mehr dazugehört. Wenn also *Anfang* und *Ende* derselbe Index sind, wird auf eine leere Teilzeichenkette Bezug genommen, und wenn *Anfang* null und *Ende* die Länge von *String* ist, dann wird auf die gesamte Zeichenkette Bezug genommen.

Manche der Prozeduren, die auf Zeichenketten arbeiten, ignorieren den Unterschied zwischen Groß- und Kleinschreibung. Die Prozeduren, die Groß- und Kleinschreibung ignorieren, haben „‘`-ci`’“ (für „case insensitive“) in ihre Namen eingebettet.

`string?` *obj* [Prozedur]
: Liefert `#t`, wenn *Obj* eine Zeichenkette ist, und andernfalls `#f`.

`make-string` *k* [Prozedur]
`make-string` *k char* [Prozedur]
: ‘`Make-string`’ liefert eine neu zugeteilte Zeichenkette der Länge *k*. Wenn *Char* angegeben wurde, dann werden alle Elemente der Zeichenkette auf *Char* initialisiert, andernfalls ist der Inhalt der gelieferten Zeichenkette unbestimmt.

`string` *char ...*, [Bibliotheksprozedur]
: Liefert eine neu zugeteilte Zeichenkette, die sich aus den Argumenten zusammensetzt.

`string-length` *string* [Prozedur]
: Liefert die Anzahl der Zeichen in der gegebenen Zeichenkette *String*.

`string-ref` *string k* [Prozedur]
: *k* muss ein gültiger Index der Zeichenkette *String* sein. ‘`String-ref`’ liefert das Zeichen *k* von *String*, wobei Indizes von null an gezählt werden.

`string-set!` *string k char* [Prozedur]
: *k* muss ein gültiger Index der Zeichenkette *String* sein. . ‘`String-set!`’ speichert *Char* in das Element *k* von *String* und liefert einen unbestimmten Wert.

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)               ==>  unbestimmt
(string-set! (g) 0 #\?)               ==>  Fehler
(string-set! (symbol->string 'unveränderlich)
             0
             #\?)                     ==>  Fehler
```

`string=?` *string1 string2* [Bibliotheksprozedur]
`string-ci=?` *string1 string2* [Bibliotheksprozedur]
: Liefert `#t`, wenn die beiden Zeichenketten dieselbe Länge haben und dieselben Zeichen an denselben Stellen haben, sonst liefert sie `#f`. ‘`String-ci=?`’ behandelt Groß- und Kleinbuchstaben als wären sie dasselbe Zeichen, aber ‘`String=?`’ behandelt Groß- und Kleinbuchstaben als verschiedene Zeichen.

`string<?` *string1 string2* [Bibliotheksprozedur]
`string>?` *string1 string2* [Bibliotheksprozedur]
`string<=?` *string1 string2* [Bibliotheksprozedur]
`string>=?` *string1 string2* [Bibliotheksprozedur]
`string-ci<?` *string1 string2* [Bibliotheksprozedur]
`string-ci>?` *string1 string2* [Bibliotheksprozedur]

`string-ci<=?` *string1 string2* [Bibliotheksprozedur]
`string-ci>=?` *string1 string2* [Bibliotheksprozedur]

Diese Prozeduren sind die alphabetisch geordnete Erweiterung auf Zeichenketten von der entsprechenden Ordnung auf Zeichen. Zum Beispiel ist '`String<?`' die alphabetische Ordnung auf Zeichenketten, die von der Ordnung '`Char<?`' auf Zeichen induziert wird. Wenn zwei Zeichenketten verschiedene Längen haben, aber bis zur Länge der kürzeren Zeichenkette identisch sind, wird die kürzere Zeichenkette als alphabetisch kleiner angesehen als die längere Zeichenkette.

Implementierungen dürfen diese Prozeduren ebenso wie die Prozeduren '`String=?`' und '`String-ci=?`' verallgemeinern, so dass sie mehr als zwei Argumente annehmen, wie bei den entsprechenden Prädikaten auf Zahlen.

`substring` *string anfang ende* [Bibliotheksprozedur]

String muss eine Zeichenkette und *Anfang* und *Ende* müssen exakte ganze Zahlen sein, so dass gilt

0 <= *anfang* <= *ende* <= (`string-length` *string*).

'`Substring`' liefert eine neu zugeteilte Zeichenkette, die aus den Zeichen von *String* zusammengesetzt wird, angefangen mit dem Index *Anfang* (einschließlich) und bis zum Index *Ende* (nicht einschließlich).

`string-append` *string* ..., [Bibliotheksprozedur]

Liefert eine neu zugeteilte Zeichenkette, deren Zeichen die Verkettung der übergebenen Zeichenketten darstellt.

`string->list` *string* [Bibliotheksprozedur]
`list->string` *liste* [Bibliotheksprozedur]

'`String->list`' liefert eine neu zugeteilte Liste der Zeichen, die die übergebene Zeichenkette bilden. '`List->string`' liefert eine neu zugeteilte Zeichenkette, die aus den Zeichen in der *Liste* gebildet wird, welche eine Liste von Zeichen sein muss. '`String->list`' und '`List->string`' sind jeweils Umkehrfunktionen voneinander bezüglich '`Equal?`'.

`string-copy` *string* [Bibliotheksprozedur]

Liefert eine neu zugeteilte Kopie der übergebenen Zeichenkette *String*.

`string-fill!` *string char* [Bibliotheksprozedur]

Speichert *Char* in jedem Element der übergebener Zeichenkette *String* und liefert einen unbestimmten Wert.

6.3.6 Vektoren

Vektoren sind heterogene Strukturen, deren Elements durch ganze Zahlen indiziert werden. Ein Vektor belegt typischerweise weniger Speicherplatz als eine Liste derselben Länge und die durchschnittliche Zeit, die nötig ist, um auf ein zufällig gewähltes Element zuzugreifen, ist typischerweise weniger für den Vektor als für die Liste.

Die *Länge* eines Vektors ist die Anzahl der Elemente, die er enthält. Diese Zahl ist eine nicht negative ganze Zahl, die von der Erstellung eines Vektors an fest ist. Die *gültigen Indizes* eines Vektors sind die exakten, nicht negativen ganzen Zahlen, die kleiner als die Länge des

Vektors sind. Das erste Element eines Vektors hat den Index null und das letzte Element hat als Index eins weniger als die Länge des Vektors.

Vektoren werden mit der Notation `#(`*obj* `...,)` geschrieben. Zum Beispiel kann ein Vektor der Länge 3, der die Zahl null in Element 0, die Liste '`(2 2 2 2)`' in Element 1 und die Zeichenkette '`"Anna"`' in Element 2 hat, wie folgt geschrieben werden:

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

Beachten Sie, dass dies die externe Darstellung eines Vektors ist, kein Ausdruck, der zu einem Vektor ausgewertet wird. Wie bei konstanten Listen müssen auch konstante Vektoren maskiert werden:

```
'#(0 (2 2 2 2) "Anna")
          ==>  #(0 (2 2 2 2) "Anna")
```

`vector?` *obj* [Prozedur]
: Liefert `#t`, wenn *Obj* ein Vektor ist, und liefert sonst `#f`.

`make-vector` *k* [Prozedur]
`make-vector` *k fill* [Prozedur]
: Liefert einen neu zugeteilten Vektor mit *k* Elementen. Wenn ein zweites Argument übergeben wurde, dann wird jedes Element auf *Fill* initialisiert. Sonst ist der anfängliche Inhalt jedes Elements unbestimmt.

`vector` *obj ...*, [Bibliotheksprozedur]
: Liefert einen neu zugeteilten Vektor, dessen Elemente die übergebenen Argumente enthalten. Analog zu '`List`'.

```
(vector 'a 'b 'c)                    ==>  #(a b c)
```

`vector-length` *vektor* [Prozedur]
: Liefert die Anzahl der Elemente im *Vektor* als exakte ganze Zahl.

`vector-ref` *vektor k* [Prozedur]
: *k* muss ein gültiger Index des Vektors *Vector* sein. '`Vector-ref`' liefert den Inhalt des Elements *k* des *Vektor*s.

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

`vector-set!` *vektor k obj* [Prozedur]

k muss ein gültiger Index des *Vektor*s sein. '`Vector-set!`' speichert *Obj* im Element *k* des *Vektor*s. Der von '`Vector-set!`' gelieferte Wert ist unbestimmt.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
          ==>  #(0 ("Sue" "Sue") "Anna")

(vector-set! '#(0 1 2) 1 "doe")
          ==>  Fehler  ; konstanter Vektor
```

`vector->list` *vektor* [Bibliotheksprozedur]
`list->vector` *liste* [Bibliotheksprozedur]

'`Vector->list`' liefert eine neu zugeteilte Liste der Objekte, die in den Elementen des *Vektor*s enthalten sind. '`List->vector`' liefert einen neu erstellten Vektor, der auf die Elemente der *Liste* initialisiert ist.

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

`vector-fill!` *vektor fill* [Bibliotheksprozedur]

Speichert *Fill* in jedem Element des *Vektor*s. Der von '`Vector-fill!`' gelieferte Wert ist unbestimmt.

6.4 Programmflussfunktionalitäten

Dieses Kapitel beschreibt verschiedene elementare Prozeduren, die den Fluss der Programmausführung auf besondere Arten steuern. Das Prädikat '`Procedure?`' wird hier auch beschrieben.

`procedure?` *obj* [Prozedur]

Liefert `#t`, wenn *Obj* eine Prozedur ist, und sonst `#f`.

```
(procedure? car)                          ==>  #t
(procedure? 'car)                         ==>  #f
(procedure? (lambda (x) (* x x)))
                                          ==>  #t
(procedure? '(lambda (x) (* x x)))
                                          ==>  #f
(call-with-current-continuation procedure?)
                                          ==>  #t
```

`apply` *proz arg1 ... args* [Prozedur]

Proz muss eine Prozedur und *Args* eine Liste sein. '`Apply`' (deutsch „anwenden“) ruft *Proz* mit den Elementen der Liste '`(append (list` *arg1* `...,)` *args*`)`' als die tatsächlichen Argumente auf.

```
(apply + (list 3 4))                    ==>  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)                ==>  30
```

`map` *proz liste1 liste2 ...,* [Bibliotheksprozedur]

Die *Liste*n müssen Listen sein und *Proz* muss eine Prozedur sein, die ebensoviele Argumente nimmt, wie es Listenargumente *Liste* gibt, und einen einzelnen Wert liefert. Wenn mehr als eine Liste *Liste* übergeben wird, dann müssen sie alle dieselbe Länge haben. '`Map`' (deutsch „abbilden“) wendet *Proz* elementweise auf die Elemente der *Liste*n an und liefert eine Liste der Ergebnisse in derselben Reihenfolge. Die Reihenfolge, mit der *Proz* auf die Elemente der *Liste*n zur Laufzeit angewandt wird, ist unbestimmt.

```
(map cadr '((a b) (d e) (g h)))
          ==>  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
          ==>  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6))               ==>  (5 7 9)

(let ((count 0))
  (map (lambda (ignoriert)
         (set! count (+ count 1))
         count)
       '(a b)))                         ==>  (1 2) oder (2 1)
```

`for-each` *proz liste1 liste2 ...,* [Bibliotheksprozedur]

Die Argumente von '`For-each`' sind wie die Argumente von '`Map`', aber '`For-each`' ruft *Proz* für ihre Nebenwirkungen statt für ihre Werte auf. Anders als '`Map`' ruft '`For-each`' die Prozedur *Proz* garantiert der Reihe nach auf die Elemente der *Liste*n

auf, vom ersten Element bzw. den ersten Elementen bis zu dem bzw. den letzten, und der von ‘`For-each`’ gelieferte Wert ist unbestimmt.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                                    ==>  #(0 1 4 9 16)
```

`force` *promise* [Bibliotheksprozedur]

Erzwingt („force“) den Wert vom Versprechen *Promise* (siehe `Delay`, Abschnitt siehe Abschnitt 4.2.5 [Verzögerte Auswertung], Seite 27). Wenn noch kein Wert für das Versprechen berechnet wurde, dann wird ein Wert berechnet und geliefert. Der Wert des Versprechens wird zwischengespeichert („gecachet“ bzw. „memoisiert“), so dass, wenn es ein weiteres Mal erzwungen wird, der vormals berechnete Wert geliefert wird.

```
(force (delay (+ 1 2)))                 ==>  3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
                                        ==>  (3 3)

(define ein-strom ; Strom = englisch Stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))

(head (tail (tail ein-strom)))
                                        ==>  2
```

‘`Force`’ und ‘`Delay`’ sind hauptsächlich für im funktionalen Stil geschriebene Programme gedacht. Die folgenden Beispiele sollten nicht als Vorzeigebeispiele guten Programmierstils angesehen werden, sondern lediglich die Eigenschaft aufzeigen, dass für ein Versprechen nur ein Wert berechnet wird, egal wie oft es erzwungen wird.

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                                       ==>  ein Versprechen
```

```
(force p)                                ==>  6
p                                        ==>  ein Versprechen, immer noch
(begin (set! x 10)
       (force p))                        ==>  6
```

Hier ist eine mögliche Implementierung von '`Delay`' und '`Force`'. Versprechen werden hier als Prozeduren ohne Argumente implementiert und '`Force`' ruft einfach sein Argument auf:

```
(define force
  (lambda (object)
    (object)))
```

Wir definieren dazu den Ausdruck

`(delay <Ausdruck>)`

als gleichbedeutend mit dem Prozeduraufruf

`(make-promise (lambda () <Ausdruck>))`

wie folgt

```
(define-syntax delay
  (syntax-rules ()
    ((delay ausdruck)
     (make-promise (lambda () ausdruck)))),
```

wobei '`Make-promise`' wie folgt definiert ist:

```
(define make-promise
  (lambda (proz)
    (let ((ergebnis-bereit? #f)
          (ergebnis #f))
      (lambda ()
        (if ergebnis-bereit?
            ergebnis
            (let ((x (proz)))
              (if ergebnis-bereit?
                  ergebnis
                  (begin (set! ergebnis-bereit? #t)
                         (set! ergebnis x)
                         ergebnis)))))))))
```

Begründung: Ein Versprechen darf seinen eigenen Wert referenzieren, wie das letzte Beispiel oben zeigt. Das Erzwingen eines solchen Versprechens kann dazu führen, dass das Versprechehen ein weiteres Mal erzwungen wird, bevor der Wert des ersten Erzwingens berechnet wurde. Dies verkompliziert die Definition von '`Make-promise`'.

Manche Implementierungen unterstützen eine Vielzahl von Erweiterungen dieser Semantik von '`Delay`' und '`Force`':

- '`Force`' auf ein Objekt aufzurufen, das kein Versprechen ist, könnte einfach dieses Objekt liefern.
- Es kann der Fall sein, dass es unmöglich ist, ein Versprechen von seinem erzwungenen Wert operativ zu unterscheiden. Das heißt, Ausdrücke wie die folgenden dürfen entweder zu `#t` oder zu `#f` ausgewertet werden, abhängig von der Implementierung:

```
(eqv? (delay 1) 1)                ==>  unbestimmt
(pair? (delay (cons 1 2)))        ==>  unbestimmt
```

- Manche Implementierungen könnten „implizites Erzwingen“ implementieren, wobei der Wert eines Versprechens durch elementare Prozeduren wie '`Cdr`' und '`+`' erzwungen wird:

```
(+ (delay (* 3 7)) 13)            ==>  34
```

`call-with-current-continuation` *proz* [Prozedur]

Proz muss eine Prozedur mit einem Argument sein. Die Prozedur '`Call-with-current-continuation`' verpackt die aktuelle Fortsetzung (siehe die Begründung unten) als eine „Ausstiegsprozedur“ („escape procedure“) und übergibt sie als Argument an *Proz*. Die Ausstiegsprozedur ist eine Scheme-Prozedur, die, wenn sie später aufgerufen wird, egal welche Fortsetzung zu diesem späteren Zeitpunkt aktiv ist, mit dieser Fortsetzung aufhört und stattdessen mit der Fortsetzung fortfährt, die aktiv war, als die Ausstiegsprozedur erzeugt wurde. Ein Aufruf der Ausstiegsprozedur kann einen Aufruf von mit `Dynamic-wind` installierten *Before*- und *After*-Thunks verursachen.

Die Ausstiegsprozedur nimmt dieselbe Anzahl an Argumenten an, wie die Fortsetzung bis zum ursprünglichen Aufruf von `Call-with-current-continuation`. Abgesehen von Fortsetzungen, die durch die Prozedur '`Call-with-values`' erzeugt wurden, nehmen alle Fortsetzungen genau einen Wert an. Wie es sich auswirkt, wenn kein oder mehr als ein Wert an nicht durch `Call-with-values` erzeugte Fortsetzungen übergeben wird, ist unbestimmt.

Die Ausstiegsprozedur, die an *Proz* übergeben wird, hat einen unbeschränkten Gültigkeitsbereich, genau wie jede andere Prozedur in Scheme. Sie darf in Variablen oder Datenstrukturen gespeichert werden und so oft aufgerufen werden, wie es gewünscht ist.

Die folgenden Beispiele zeigen nur die häufigsten Arten, auf die '`Call-with-current-continuation`' benutzt wird. Wenn alle wirklichen

Anwendungen so einfach wären wie diese Beispiele, bräuchte man keine Prozedur, die so mächtig ist wie ‘`Call-with-current-continuation`’.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                                ==>  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))                ==>  4

(list-length '(a b . c))                ==>  #f
```

Begründung:

Eine übliche Nutzung von ‘`Call-with-current-continuation`’ sind strukturierte, nicht-lokale Sprünge aus Schleifen oder Prozedurrümpfen heraus, aber tatsächlich ist ‘`Call-with-current-continuation`’ für die Implementierung einer große Vielfalt fortgeschrittener Programmflussstrukturen von extremem Nutzen.

Wann immer ein Scheme-Ausdruck ausgewertet wird, gibt es eine *Fortsetzung* („continuation“), die das Ergebnis des Ausdrucks haben will. Die Fortsetzung stellt die gesamte (standardmäßige) Zukunft der Berechnung dar. Wenn der Ausdruck zum Beispiel auf oberster Ebene ausgewertet wird, dann kann die Fortsetzung das Ergebnis entgegennehmen, auf dem Bildschirm ausgeben, nach der nächsten Eingabe fragen, und ewig so weiter. Die meiste Zeit schließt die Fortsetzung vom Nutzer-Code festgelegte Aktionen mit ein, wie zum Beispiel eine Fortsetzung, die das Ergebnis nimmt, mit dem in einer lokalen Variablen gespeicherten Wert multipliziert, sieben addiert und die Antwort an die Fortsetzung auf oberster Ebene zum Ausgeben weitergibt. Normalerweise sind diese allgegenwärtigen Fortsetzungen im Hintergrund verborgen und Programmierer denken nicht viel über sie nach. In seltenen Fällen

könnte ein Programmierer jedoch mit Fortsetzungen ausdrücklich umgehen müssen. '`Call-with-current-continuation`' erlaubt es Scheme-Programmierern, dazu eine Prozedur zu erstellen, die sich genau wie die momentane Fortsetzung verhält.

Die meisten Programmiersprachen enthalten eines oder mehr Ausstiegskonstrukte für ganz bestimmte Zwecke mit Namen wie `Exit`, '`Return`' oder sogar `Goto`. 1965 hat Peter Landin [Landin65] jedoch einen für allgemeine Zwecke bestimmten Ausstiegsoperator namens J-Operator erfunden. John Reynolds [Reynolds72] hat 1972 ein einfacheres, aber ebenso mächtiges Konstrukt beschrieben. Die Sonderform '`Catch`', die Sussman und Steele im Scheme-Bericht von 1975 beschrieben haben, ist genau dasselbe wie Reynolds' Konstrukt, doch ihr Name kommt von einem weniger allgemeinen Konstrukt in MacLisp. Mehrere Scheme-Implementierer bemerkten, dass die ganze Macht des `Catch`-Konstrukts durch eine Prozedur statt eines besonderen syntaktischen Konstrukts bereitgestellt werden könnte, und der Name '`Call-with-current-continuation`' wurde 1982 geprägt. Dieser Name ist beschreibend, aber die Meinungen gehen über die Vorzüge eines derart langen Namens auseinander und manche Leute benutzen stattdessen den Namen `Call/cc`.

`values` *obj ...* [Prozedur]

Liefert all ihre Argumente an ihre Fortsetzung. Abgesehen von mit der Prozedur `Call-with-values` erstellten Fortsetzungen nehmen alle Fortsetzungen genau einen Wert entgegen. `Values` kann wie folgt definiert werden:

```
(define (values . dinge)
  (call-with-current-continuation
    (lambda (cont) (apply cont dinge))))
```

`call-with-values` *produzent konsument* [Prozedur]

Ruft ihr *Produzent*en-Argument mit keinen Werten und einer Fortsetzung auf, die, wenn ihr Werte übergeben werden, die *Konsument*en-Prozedur mit diesen Werten als Argumente aufruft. Die Fortsetzung des *Konsument*enaufrufs ist die Fortsetzung des Aufrufs von `Call-with-values`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                                              ==>  5

(call-with-values * -)                        ==>  -1
```

`dynamic-wind` *before thunk after* [Prozedur]

Ruft *Thunk* ohne Argumente auf und liefert das Ergebnis oder die Ergebnisse dieses Aufrufs. *Before* und *After* werden, ebenso ohne Argumente, wie von den folgenden Regeln vorausgesetzt aufgerufen (beachten Sie, dass ohne Aufrufe von mit

`Call-with-current-continuation` gefangenen Fortsetzungen die drei Argumente jeweils einmal der Reihe nach aufgerufen werden). *Before* wird immer dann aufgerufen, wenn die Ausführung den dynamischen Bereich des Aufrufs von *Thunk* betritt, und *After* wird immer dann aufgerufen, wenn sie diesen dynamischen Bereich verlässt. Der dynamische Bereich eines Prozeduraufrufs ist die Zeit zwischen dem Verursachen des Aufrufs und dem Liefern seines Wertes. In Scheme kann, wegen '`Call-with-current-continuation`', der dynamische Bereich etwas anderes als eine einzelne, zusammenhängende Zeitspanne sein. Er ist wie folgt definiert:

- Der dynamische Bereich wird betreten, wenn die Ausführung des Rumpfs der aufgerufenen Prozedur beginnt.
- Der dynamische Bereich wird auch betreten, wenn die Ausführung sich nicht im dynamischen Bereich befindet und eine Fortsetzung aufgerufen wird, die (mit '`Call-with-current-continuation`') innerhalb des dynamischen Bereichs eingefangen wurde.
- Er wird verlassen, wenn die aufgerufene Prozedur ihr Ergebnis oder ihre Ergebnisse liefert.
- Er wird auch verlassen, wenn sich die Ausführung innerhalb des dynamischen Bereichs befindet und eine Fortsetzung, die außerhalb des dynamischen Bereichs gefangen wurde, aufgerufen wird.

Wenn ein zweiter Aufruf von '`Dynamic-wind`' innerhalb des dynamischen Bereichs des Aufrufs von *Thunk* stattfindet und eine Fortsetzung auf eine Art aufgerufen wird, dass die *After*s dieser beiden Aufrufe von '`Dynamic-wind`' beide aufzurufen sind, dann wird die *After*, die mit dem zweiten (inneren) Aufruf von '`Dynamic-wind`' assoziiert ist, zuerst aufgerufen.

Wenn der zweite Aufruf von '`Dynamic-wind`' innerhalb des dynamischen Bereichs des Aufrufs von *Thunk* stattfindet und dann eine Fortsetzung auf eine Art aufgerufen wird, dass die *Before*s dieser beiden Aufrufe von '`Dynamic-wind`' beide auszuführen sind, dann wird die *Before*, die mit dem ersten (äußeren) Aufruf von '`Dynamic-wind`' assoziiert ist, zuerst aufgerufen.

Wenn das Aufrufen einer Fortsetzung das Aufrufen der *Before* des einen Aufrufs von '`Dynamic-wind`' und der *After* eines anderen Aufrufs erfordert, dann wird die *After* zuerst aufgerufen.

Die Auswirkung, eine eingefangene Fortsetzung zu benutzen, um aus dem dynamischen Bereich eines Aufrufs von *Before* oder *After* auszusteigen, ist undefiniert.

```
(let ((pfad '())
      (c #f))
  (let ((add (lambda (s)
               (set! pfad (cons s pfad)))))
    (dynamic-wind
      (lambda () (add 'verbinde))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'spreche1))))
```

```
      (lambda () (add 'trenne)))
    (if (< (length pfad) 4)
        (c 'spreche2)
        (reverse pfad))))

      ==> (verbinde spreche1 trenne
           verbinde spreche2 trenne)
```

6.5 Eval

`eval` *ausdruck umgebungsangabe* [Prozedur]

Wertet *Ausdruck* in der angegebenen Umgebung aus und liefert seinen Wert. *Ausdruck* muss ein gültiger Scheme-Ausdruck sein, der als Daten dargestellt ist, und *Umgebungsangabe* muss ein von einer der drei unten beschriebenen Prozeduren gelieferter Wert sein. Implementierungen dürfen '`Eval`' erweitern, um Programme, die keine Ausdrücke sind (Definitionen) als das erste Argument zu erlauben und um andere Werte als Umgebungen zuzulassen, mit der Einschränkung, dass '`Eval`' keine neuen Bindungen in den mit '`Null-environment`' oder '`Scheme-report-environment`' assoziierten Umgebungen erzeugen darf.

```
(eval '(* 7 3) (scheme-report-environment 5))
                                                    ==>  21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
                                                    ==>  20
```

`scheme-report-environment` *version* [Prozedur]

`null-environment` *version* [Prozedur]

Version muss die exakte ganze Zahl '`5`' sein, entsprechend dieser Fassung des Scheme-Berichts (des Revised^5 Report on Scheme). '`Scheme-report-environment`' liefert eine Angabe einer Umgebung, die leer ist bis auf alle in diesem Bericht angegebenen Bindungen, die entweder notwendig oder sowohl optional als auch von der Implementierung unterstützt sind. '`Null-environment`' liefert eine Angabe der Umgebung, die leer ist außer den (syntaktischen) Bindungen aller syntaktischen Schlüsselwörtern, die in diesem Bericht definiert werden und die entweder notwendig oder sowohl optional als auch von der Implementierung unterstützt sind.

Andere Werte von *Version* können benutzt werden, um zu vergangenen Fassungen passende Umgebungen zu definieren, aber ihre Unterstützung wird nicht vorausgesetzt. Eine Implementierung wird einen Fehler signalisieren, wenn *Version* weder '`5`' noch ein anderer von der Implementierung unterstützter Wert ist.

Die Auswirkung davon, eine in einer '`Scheme-report-environment`' gebundene Variable (durch Nutzung von '`Eval`') zuzuweisen, ist unbestimmt. Die

durch '`Scheme-report-environment`' angegebenen Umgebungen dürfen also unveränderlich sein.

`interaction-environment` [optionale Prozedur]
Diese Prozedur liefert eine Angabe der Umgebung, die implementierungsdefinierte Bindungen enthält, typischerweise eine Teilmenge der im Bericht aufgelisteten Bindungen. Die Absicht ist, dass diese Prozedur die Umgebung liefert, in der die Implementierung von der Nutzerin dynamisch eingetippte Ausdrücke auswertet.

6.6 Ein- und Ausgabe

6.6.1 Ports

Ports stellen Ein- und Ausgabegeräte dar. Gegenüber Scheme ist ein Eingabe-Port ein Scheme-Objekt, das auf Anfrage hin Zeichen liefern kann, während ein Ausgabe-Port ein Scheme-Objekt ist, das Zeichen annehmen kann.

`call-with-input-file` *string proz* [Bibliotheksprozedur]
`call-with-output-file` *string proz* [Bibliotheksprozedur]
String sollte eine Zeichenkette sein, die eine Datei benennt, und *Proz* sollte eine Prozedur sein, die ein Argument nimmt. Für '`Call-with-input-file`' sollte die Datei bereits existieren; für '`Call-with-output-file`' ist die Auswirkung unbestimmt, wenn die Datei bereits existiert. Diese Prozeduren rufen *Proz* mit einem Argument auf: dem Port, der durch das Öffnen der benannten Datei zur Ein- oder Ausgabe erhalten wurde. Wenn die Datei nicht geöffnet werden kann, wird ein Fehler signalisiert. Wenn *Proz* einen Wert liefert, dann wird der Port automatisch geschlossen und der Wert oder die Werte, die *Proz* geliefert hat, geliefert. Wenn *Proz* keinen Wert liefert, wird der Port nicht automatisch geschlossen, solange es nicht möglich ist, zu beweisen, dass der Port niemals wieder für eine Lese- oder Schreiboperation benutzt werden wird.

> *Begründung:* Weil Schemes Ausstiegsprozeduren einen unbegrenzten Gültigkeitsbereich haben, ist es möglich, aus der aktuellen Fortsetzung auszusteigen, aber später wieder hinein zurückzukehren. Wenn Implementierungen den Port bei jedem Ausstieg aus der aktuellen Fortsetzung schließen dürften, wäre es unmöglich, portablen Code zu schreiben, der sowohl '`Call-with-current-continuation`' als auch '`Call-with-input-file`' oder '`Call-with-output-file`' benutzt.

`input-port?` *obj* [Prozedur]
`output-port?` *obj* [Prozedur]
Liefert `#t`, wenn *Obj* jeweils ein Eingabe-Port oder Ausgabe-Port ist, und liefert sonst `#f`.

`current-input-port` [Prozedur]
`current-output-port` [Prozedur]
Liefert den aktuellen standardmäßigen Eingabe- oder Ausgabe-Port.

`with-input-from-file` *string thunk* [optionale Prozedur]
`with-output-to-file` *string thunk* [optionale Prozedur]

String sollte eine Zeichenkette sein, die eine Datei benennt, und *Thunk* sollte eine Prozedur ohne Argumente sein. Bei '`With-input-from-file`' sollte die Datei bereits existieren, bei '`With-output-to-file`' ist die Auswirkung unbestimmt, wenn die Datei bereits existiert. Die Datei wird zur Eingabe oder Ausgabe geöffnet, ein Eingabe- oder Ausgabe-Port mit ihr verbunden, zum standardmäßigen Wert gemacht, der von '`Current-input-port`' oder '`Current-output-port`' geliefert (und von `(read)`, `(write obj)`, und so weiter) benutzt wird, und der *Thunk* wird ohne Argumente aufgerufen. Wenn der *Thunk* einen Wert liefert, wird der Port geschlossen und der vorherige standardmäßige Wert wiederhergestellt. '`With-input-from-file`' und '`With-output-to-file`' liefern den Wert oder die Werte, die *Thunk* geliefert hat. Wenn eine Ausstiegsprozedur benutzt wird, um aus der Fortsetzung dieser Prozeduren auszusteigen, ist das Verhalten der beiden implementierungsabhängig.

`open-input-file` *dateiname* [Prozedur]

Nimmt eine Zeichenkette entgegen, die eine existierende Datei benennt, und liefert einen Eingabe-Port, der in der Lage ist, Zeichen aus dieser Datei zu liefern. Wenn die Datei nicht geöffnet werden kann, wird ein Fehler signalisiert.

`open-output-file` *dateiname* [Prozedur]

Nimmt eine Zeichenkette entgegen, die eine Ausgabedatei benennt, die zu erstellen ist, und liefert einen Ausgabe-Port, der in der Lage ist, Zeichen in eine neue Datei dieses Namens zu schreiben. Wenn die Datei nicht geöffnet werden kann, wird ein Fehler signalisiert. Wenn eine Datei mit dem angegebenen Namen bereits existiert, ist die Auswirkung unbestimmt.

`close-input-port` *port* [Prozedur]
`close-output-port` *port* [Prozedur]

Schließt die mit *Port* assoziierte Datei, was den *Port* nicht mehr in der Lage belässt, Zeichen zu liefern oder zu akzeptieren.

Diese Routinen haben keine Auswirkung, wenn die Datei bereits geschlossen wurde. Der gelieferte Wert ist unbestimmt.

6.6.2 Eingabe

`read` [Bibliotheksprozedur]
`read` *port* [Bibliotheksprozedur]

'`Read`' („lesen“) wandelt externe Darstellungen von Scheme-Objekten in die Objekte selbst um. Das heißt, sie dient zur grammatikalischen Analyse („Parsing“) des nichtterminalen <Datenelement>s (siehe die Abschnitte siehe Abschnitt 7.1.2 [Externe Darstellung], Seite 84, und siehe Abschnitt 6.3.2 [Paare und Listen], Seite 54).

'`Read`' liefert das nächste analysierbare Objekt des angegebenen Eingabe*port*s und aktualisiert dabei den *Port*, so dass er auf das erste Zeichen nach dem Ende der externen Darstellung des Objekts zeigt.

Wenn '`Read`' auf ein Dateiende in der Eingabe trifft, bevor jegliche Zeichen gefunden wurden, die der Anfang eines Objekts sein können, dann wird ein Dateiende-Objekt geliefert. Der Port bleibt offen und weitere Versuche zu lesen, werden auch ein Dateiende-Objekt liefern. Wenn auf ein Dateiende nach dem Anfang der externen Darstellung eines Objekts getroffen wird, aber die externe Darstellung unvollständig und daher nicht analysierbar ist, wird ein Fehler signalisiert.

Das *Port*-Argument darf weggelassen werden und nimmt in diesem Fall standardmäßig den Wert an, der von '`Current-input-port`' geliefert wird. Es ist ein Fehler, von einem geschlossenen Port zu lesen.

`read-char` [Prozedur]
`read-char` *port* [Prozedur]

Liefert das nächste vom Eingabe*port* verfügbare Zeichen und aktualisiert dabei den *Port*, so dass er auf das darauf folgende Zeichen zeigt. Wenn keine Zeichen mehr verfügbar sind, wird ein Dateiende-Objekt geliefert. *Port* darf weggelassen werden und nimmt in diesem Fall standardmäßig den Wert an, der durch '`Current-input-port`' geliefert wird.

`peek-char` [Prozedur]
`peek-char` *port* [Prozedur]

Liefert das nächste vom Eingabe-*Port* verfügbare Zeichen, *ohne* den *Port* zu aktualisieren, also ohne ihn auf das nachfolgende Zeichen zeigen zu lassen. Wenn keine Zeichen mehr verfügbar sind, wird ein Dateiende-Objekt geliefert. *Port* darf weggelassen werden, in diesem Fall nimmt er standardmäßig den Wert an, der von '`Current-input-port`' geliefert wird.

> *Anmerkung:* Der vom Aufruf von '`Peek-char`' gelieferte Wert ist derselbe wie der von einem Aufruf von '`Read-char`' mit demselben *Port* geliefert wird. Der einzige Unterschied ist, dass der direkt folgende Aufruf von '`Read-char`' oder '`Peek-char`' auf diesem *Port* den Wert liefern wird, den der vorhergehende Aufruf von '`Peek-char`' geliefert hat. Insbesondere wird ein Aufruf von '`Peek-char`' auf einem interaktiven Port hängen, während er auf eine Eingabe wartet, wann immer ein Aufruf von '`Read-char`' gehangen hätte.

`eof-object?` *obj* [Prozedur]

Liefert `#t`, wenn *Obj* ein Dateiende-Objekt ist, und liefert sonst `#f`. Die genaue Menge von Dateiende-Objekten wird sich je nach Implementierung unterscheiden, jedenfalls wird ein Dateiende-Objekt aber niemals ein durch Benutzung von '`Read`' lesbares Objekt sein.

`char-ready?` [Prozedur]
`char-ready?` *port* [Prozedur]

Liefert `#t`, wenn ein Zeichen auf dem Eingabe*port* bereit ist, und liefert sonst `#f`. Wenn '`Char-ready`' den Wert `#t` liefert, dann ist garantiert, dass die nächste '`Read-char`'-Operation auf dem übergebenen *Port* nicht hängen wird. Wenn der *Port* am Dateiende

ist, dann liefert '`Char-ready?`' den Wert `#t`. *Port* darf weggelassen werden, in diesem Fall nimmt er standardmäßig den von '`Current-input-port`' gelieferten Wert an.

> *Begründung:* '`Char-ready?`' existiert, um es einem Programm zu ermöglichen, Zeichen von interaktiven Ports zu akzeptieren, ohne im Warten auf Eingaben festzustecken. Jeder Eingabebearbeiter, der mit solchen Ports assoziiert ist, muss sicherstellen, dass Zeichen, deren Existenz durch '`Char-ready?`' zugesichert wurde, nicht mehr ausradiert werden können. Wenn '`Char-ready?`' den Wert `#f` am Dateiende liefern würde, könnte man einen Port am Dateiende nicht von einem interaktiven Port unterscheiden, der keine bereitstehenden Zeichen hat.

6.6.3 Ausgabe

`write` *obj* [Bibliotheksprozedur]
`write` *obj port* [Bibliotheksprozedur]

Schreibt eine geschriebene Darstellung von *Obj* auf den übergebenen *Port*. Zeichenketten, die in der geschriebenen Darstellung vorkommen, werden in doppelte Anführungszeichen eingeschlossen, und innerhalb dieser Zeichenketten wird jeder Backslash und jedes doppelte Anführungszeichen durch einen Backslash maskiert. Zeichenobjekte werden durch Nutzung der Notation '`#\`' geschrieben. '`Write`' liefert einen unbestimmten Wert. Das *Port*-Argument darf weggelassen werden, in diesem Fall nimmt es standardmäßig den von '`Current-output-port`' gelieferten Wert an.

`display` *obj* [Bibliotheksprozedur]
`display` *obj port* [Bibliotheksprozedur]

Schreibt eine Darstellung von *Obj* auf den übergebenen *Port*. Zeichenketten, die in der geschriebenen Darstellung auftauchen, werden nicht mit doppelten Anführungszeichen umschlossen und keine Zeichen werden innerhalb dieser Zeichenketten maskiert. Zeichenobjekte kommen in der Darstellung so vor, als würden sie mit '`Write-char`' statt mit '`Write`' geschrieben. '`Display`' liefert einen unbestimmten Wert. Das *Port*-Argument darf weggelassen werden, in diesem Fall nimmt es standardmäßig den Wert an, den '`Current-output-port`' liefert.

> *Begründung:* Die Absicht hinter '`Write`' ist, maschinenlesbare Ausgaben zu produzieren, während '`Display`' dazu dienen soll, menschenlesbare Ausgaben zu produzieren. Implementierungen, die „Slashifizierung“ innerhalb von Symbolen zulassen, werden wahrscheinlich wollen, dass '`Write`', aber nicht '`Display`', komische Zeichen in Symbolen slashifiziert.

`newline` [Bibliotheksprozedur]
`newline` *port* [Bibliotheksprozedur]

Schreibt ein Zeilenende auf den *Port*. Wie genau dies umgesetzt wird, unterscheidet sich je nach Betriebssystem. Liefert einen unbestimmten Wert. Das *Port*-Argument

darf weggelassen werden, in diesem Fall nimmt es standardmäßig den durch ‘`Current-output-port`’ gelieferten Wert an.

`write-char` *char* [Prozedur]
`write-char` *char port* [Prozedur]
Schreibt das Zeichen *Char* (nicht eine externe Darstellung dieses Zeichens) auf den übergebenen *Port* und liefert einen unbestimmten Wert. Das *Port*-Argument darf weggelassen werden, in diesem Fall nimmt es standardmäßig den von ‘`Current-output-port`’ gelieferten Wert an.

6.6.4 Systemschnittstelle

Fragen der Systemschnittstelle fallen im allgemeinen nicht in den Zuständigkeitsbereich dieses Berichts. Die folgenden Operationen sind jedoch wichtig genug, um eine Beschreibung hier zu verdienen.

`load` *dateiname* [optionale Prozedur]
Dateiname sollte eine Zeichenkette sein, die eine existierende Datei benennt, welche Scheme-Quellcode enthält. Die Prozedur ‘`Load`’ liest Ausdrücke und Definitionen aus der Datei und wertet sie der Reihe nach aus. Es ist unbestimmt, ob die Ergebnisse der Ausdrücke ausgegeben werden. Die Prozedur ‘`Load`’ beeinflusst den von ‘`Current-input-port`’ und ‘`Current-output-port`’ gelieferten Wert nicht. ‘`Load`’ liefert einen unbestimmten Wert.

> *Begründung:* Der Portabilität wegen wird vorausgesetzt, dass ‘`Load`’ auf Quelldateien arbeiten muss. Wie es mit anderen Arten von Dateien umgeht, unterscheidet sich notwendigerweise zwischen Implementierungen.

`transcript-on` *dateiname* [optionale Prozedur]
`transcript-off` [optionale Prozedur]
Dateiname muss eine Zeichenkette sein, die eine Ausgabedatei benennt, die erstellt werden soll. Die Auswirkung von ‘`Transcript-on`’ ist, die benannte Datei zur Ausgabe zu öffnen und eine Abschrift der nachfolgenden Interaktion zwischen der Nutzerin und dem Scheme-System in die Datei schreiben zu lassen. Die Abschrift endet mit einem Aufruf von ‘`Transcript-off`’, welche die Datei der Abschrift schließt. Nur eine Abschrift darf gleichzeitig angefertigt werden, wobei manche Implementierungen diese Einschränkung lockern könnten. Die Werte, die von diesen Prozeduren geliefert werden, sind unbestimmt.

7 Formale Syntax und Semantik

Dieses Kapitel bietet formale Beschreibungen dessen, was in vorherigen Kapiteln dieses Berichts bereits informell beschrieben wurde.

7.1 Formale Syntax

Dieser Abschnitt bietet eine formale Syntax für Scheme, die in einer erweiterten BNF geschrieben ist.

Jeglicher Leerraum in der Grammatik dient nur der Lesbarkeit. Groß- und Kleinschreibung wird nicht unterschieden, zum Beispiel sind '`#x1A`' und '`#X1a`' äquivalent. <leer> steht für die leere Zeichenkette.

Die folgenden Erweiterungen der BNF werden benutzt, um eine knappere Beschreibung zu ermöglichen: <Ding>* bedeutet null oder mehr Vorkommen von <Ding> und <Ding>+ bedeutet mindestens ein <Ding>.

7.1.1 Lexikalische Struktur

Dieser Abschnitt beschreibt, wie einzelne Tokens („Marken“, d.h. Bezeichner, Zahlen, etc.) aus Folgen von Zeichen gebildet werden. Die danach folgenden Abschnitte beschreiben, wie Ausdrücke und Programme aus Folgen von Tokens gebildet werden.

<Leerraum zwischen Tokens> darf auf beiden Seiten eines beliebigen Tokens auftreten, aber nicht innerhalb eines Tokens.

Tokens, die ein implizites Ende erfordern (Bezeichner, Zahlen, Zeichen und der Punkt) dürfen von jedem <Trennzeichen> beendet werden, aber nicht unbedingt von irgendetwas anderem.

Die folgenden fünf Zeichen werden für spätere Erweiterungen der Sprache reserviert: `[ ] { } |`

```
<Token> --> <Bezeichner> | <boolescher Wert> | <Zahl>
     | <Zeichen> | <Zeichenkette>
     | ( | ) | #( | ’ | ‘ | , | ,@ | .
<Trennzeichen> --> <Leerraum> | ( | ) | " | ;
<Leerraum> --> <Leerzeichen oder Zeilenvorschub>
<Kommentar> --> ;  <alle nachfolgenden Zeichen bis zu einem
                   Zeilenvorschub>
<Atmosphäre> --> <Leerraum> | <Kommentar>
<Leerraum zwischen Tokens> --> <Atmosphäre>*

<Bezeichner> --> <Anfangselement> <Folgeelement>*
     | <besonderer Bezeichner>
<Anfangselement> --> <Buchstabe> | <spezielles Anfangselement>
<Buchstabe> --> a | b | c | ... | z

<spezielles Anfangselement> --> ! | $ | % | & | * | / | :
     | < | = | > | ? | ^ | _ | ~
<Folgeelement> --> <Anfangselement> | <Ziffer>
```

```
     | <spezielles Folgeelement>
<Ziffer> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<spezielles Folgeelement> --> + | - | . | @
<besonderer Bezeichner> --> + | - | ...
<syntaktisches Schlüsselwort> --> <Ausdrucksschlüsselwort>
     | else | => | define
     | unquote | unquote-splicing
<Ausdrucksschlüsselwort> --> quote | lambda | if
     | set! | begin | cond | and | or | case
     | let | let* | letrec | do | delay
     | quasiquote
```

‘<Variable> ⇒ <’beliebiger Bezeichner, der nicht
auch ein <syntaktisches Schlüsselwort> ist>

```
<boolescher Wert> --> #t | #f
<Zeichen> --> #\ <beliebiges Zeichen>
     | #\ <Zeichenname>
<Zeichenname> --> space | newline

<Zeichenkette> --> " <Zeichenkettenelement>* "
<Zeichenkettenelement> --> <beliebiges Zeichen außer " oder \>
     | \" | \\

<Zahl> --> <num 2>| <num 8>
     | <num 10>| <num 16>
```

Die folgenden Regeln für <num R>, <complex R>, <real R>, <ureal R>, <uinteger R> und <prefix R> sollten für R = 2, 8, 10, und 16 nachgebildet werden. Es gibt keine Regeln für <decimal 2>, <decimal 8> und <decimal 16>, was bedeutet, dass Zahlen, die Dezimaltrennzeichen oder Exponenten enthalten, Radix 10 (d.h. Basis 10) haben müssen.

```
<num R> --> <Präfix R> <complex R>
<complex R> --> <real R> | <real R> @ <real R>
    | <real R> + <ureal R> i | <real R> - <ureal R> i
    | <real R> + i | <real R> - i
    | + <ureal R> i | - <ureal R> i | + i | - i
<real R> --> <Vorzeichen> <ureal R>
<ureal R> --> <uinteger R>
    | <uinteger R> / <uinteger R>
    | <decimal R>
<decimal 10> --> <uinteger 10> <Suffix>
    | . <Ziffer 10>+ #* <Suffix>
    | <Ziffer 10>+ . <Ziffer 10>* #* <Suffix>
    | <Ziffer 10>+ #+ . #* <Suffix>
```

```
<uinteger R> --> <digit R>+ #*
<Präfix R> --> <Radix R> <Exaktheit>
    | <Exaktheit> <Radix R>



<Suffix> --> <leer>
    | <Exponentenmarkierung> <Vorzeichen> <Ziffer 10>+
<Exponentenmarkierung> --> e | s | f | d | l
<Vorzeichen> --> <leer>  | + |  -
<Exaktheit> --> <leer> | #i | #e
<Radix 2> --> #b
<Radix 8> --> #o
<Radix 10> --> <empty> | #d
<Radix 16> --> #x
<Ziffer 2> --> 0 | 1
<Ziffer 8> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<Ziffer 10> --> <Ziffer>
<Ziffer 16> --> <Ziffer 10> | a | b | c | d | e | f
```

7.1.2 Externe Darstellungen

<Datenelement> ist, was die Prozedur `Read` (siehe den Abschnitt siehe Abschnitt 6.6.2 [Eingabe], Seite 78) erfolgreich grammatikalisch analysiert. Beachten Sie, dass jede Zeichenkette, die als ein <Ausdruck> grammatikalisch analysiert werden kann, auch als ein <Datenelement> grammatikalisch analysiert werden kann>.

```
<Datenelement> --> <einfaches Datenelement>
     | <zusammengesetztes Datenelement>
<einfaches Datenelement> --> <boolescher Wert> | <Zahl>
     | <Zeichen> | <Zeichenkette> | <Symbol>
<Symbol> --> <Bezeichner>
<zusammengesetztes Datenelement> --> <Liste> | <Vektor>
<Liste> --> (<Datenelement>*)
       | (<Datenelement>+ . <Datenelement>) | <Abkürzung>
<Abkürzung> --> <Abk-Präfix> <Datenelement>
<Abk-Präfix> --> ' | ` | , | ,@
<Vektor> --> #(<Datenelement>*)
```

7.1.3 Ausdrücke

```
<Ausdruck> --> <Variable>
     | <Literal>
     | <Prozeduraufruf>
     | <lambda-Ausdruck>
     | <Bedingung>
     | <Zuweisung>
```

```
     | <abgeleiteter Ausdruck>
     | <Makronutzung>
     | <Makroblock>

<Literal> --> <Maskierung> | <Selbstauswertung>
<Selbstauswertung> --> <boolescher Wert> | <Zahl>
     | <Zeichen> | <Zeichenkette>
<Maskierung> --> '<Datenelement> | (quote <Datenelement>)
<Prozeduraufruf> --> (<Operator> <Operand>*)
<Operator> --> <Ausdruck>
<Operand> --> <Ausdruck>

<lambda-Ausdruck> --> (lambda <Formale> <Rumpf>)
<Formale> --> (<Variable>*) | <Variable>
     | (<Variable>+ . <Variable>)
<Rumpf> --> <Definition>* <Folge>
<Folge> --> <Befehl>* <Ausdruck>
<Befehl> --> <Ausdruck>

<Bedingung> --> (if <Test> <Folgerung> <Alternative>)
<Test> --> <Ausdruck>
<Folgerung> --> <Ausdruck>
<Alternative> --> <Ausdruck> | <leer>

<Zuweisung> --> (set! <Variable> <Ausdruck>)

<abgeleiteter Ausdruck> -->
       (cond <cond-Klausel>+)
     | (cond <cond-Klausel>* (else <Folge>))
     | (case <Ausdruck>
         <case-Klausel>+)
     | (case <Ausdruck>
         <case-Klausel>*
         (else <Folge>))
     | (and <Test>*)
     | (or <Test>*)
     | (let (<Bindungsspezifikation>*) <Rumpf>)
     | (let <Variable> (<Bindungsspezifikation>*) <Rumpf>)
     | (let* (<Bindungsspezifikation>*) <Rumpf>)
     | (letrec (<Bindungsspezifikation>*) <Rumpf>)
     | (begin <Folge>)
     | (do (<Iterationsspezifikation>*)
           (<Test> <do-Ergebnis>)
         <Befehl>*)
     | (delay <Iterationsspezifikation>)
     | <Quasimaskierung>
```

```
<cond-Klausel> --> (<Test> <Folge>)
      | (<Test>)
      | (<Test> => <Empfänger>)
<Empfänger> --> <Ausdruck>
<case-Klausel> --> ((<Datenelement>*) <Folge>)
<Bindungsspezifikation> --> (<Variable> <Ausdruck>)
<Iterationsspezifikation> --> (<Variable> <Anfang> <Schritt>)
    | (<Variable> <Anfang>)
<Anfang> --> <Ausdruck>
<Schritt> --> <Ausdruck>
<do-Ergebnis> --> <Folge> | <leer>

<Makronutzung> --> (<Schlüsselwort> <Datenelement>*)
<Schlüsselwort> --> <Bezeichner>

<Makroblock> -->
     (let-syntax (<Syntaxspezifikation>*) <Rumpf>)
     | (letrec-syntax (<Syntaxspezifikation>*) <Rumpf>)
<Syntaxspezifikation> --> (<Schlüsselwort> <Umwandlerspezifikation>)
```

7.1.4 Quasimaskierungen

Die folgende Grammatik für Quasimaskierungsausdrücke ist nicht kontextfrei. Sie wird als ein Rezept zur Generierung unendlich vieler Produktionsregeln vorgestellt. Stellen Sie sich eine Kopie der folgenden Regeln für jedes D = 1, 2,3, ... vor. D misst die Verschachtelungstiefe.

```
<Quasimaskierung> --> <Quasimaskierung 1>
<qq-Schablone 0> --> <Ausdruck>
<Quasimaskierung D> --> `<qq-Schablone D>
       | (quasiquote <qq-Schablone D>)
<qq-Schablone D> --> <einfaches Datenelement>
       | <Listen-qq-Schablone D>
       | <Vektor-qq-Schablone D>
       | <Demaskierung D>
<Listen-qq-Schablone D> --> (<qq-Schablone oder Spleißung D>*)
       | (<qq-Schablone oder Spleißung D>+ . <qq-Schablone D>)
       | '<qq-Schablone D>
       | <Quasimaskierung D+1>
<Vektor-qq-Schablone D> --> #(<qq-Schablone oder Spleißung D>*)
<Demaskierung D> --> ,<qq-Schablone D-1>
       | (unquote <qq-Schablone D-1>)
<qq-Schablone oder Spleißung D> --> <qq-Schablone D>
       | <spleißende Demaskierung D>
```

```
<spleißende Demaskierung D> --> ,@<qq-Schablone D-1>
       | (unquote-splicing <qq-Schablone D-1>)
```

In <Quasimaskierung>en kann eine <Listen-qq-Schablone D> manchmal verwechselt werden mit entweder einer <Demaskierung D> oder einem Vorkommen von <spleißende Demaskierung D>. Die Interpretation als eine <Demaskierung> oder <spleißende Demaskierung D> hat Vorrang.

7.1.5 Umwandler

```
<Umwandlerspezifikation> -->
    (syntax-rules (<Bezeichner>*) <Syntaxregel>*)
<Syntaxregel> --> (<Muster> <Schablone>)
<Muster> --> <Musterbezeichner>
     | (<Muster>*)
     | (<Muster>+ . <Muster>)
     | (<Muster>* <Muster> <Auslassungspunkte>)
     | #(<Muster>*)
     | #(<Muster>* <Muster> <Auslassungspunkte>)
     | <Musterdatenelement>
<Musterdatenelement> --> <Zeichenkette>
     | <Zeichen>
     | <boolescher Wert>
     | <Zahl>
<Schablone> --> <Musterbezeichner>
     | (<Schablonenelement>*)
     | (<Schablonenelement>+ . <Schablone>)
     | #(<Schablonenelement>*)
     | <Schablonendatenelement>
<Schablonenelement> --> <Schablone>
     | <Schablone> <Auslassungspunkte>
<Schablonendatenelement> --> <Musterdatenelement>
<Musterbezeichner> --> <beliebiger Bezeichner außer '...'>
<Auslassungspunkte> --> <der Bezeichner '...'>
```

7.1.6 Programme und Definitionen

```
<Programm> --> <Befehl oder Definition>*
<Befehl oder Definition> --> <Befehl>
    | <Definition>
    | <Syntaxdefinition>
    | (begin <Befehl oder Definition>+)
<Definition> --> (define <Variable> <Ausdruck>)
      | (define (<Variable> <Def-Formale>) <Rumpf>)
      | (begin <Definition>*)
```

```
<Def-Formale> --> <Variable>*
      | <Variable>* . <Variable>
<Syntaxdefinition> -->
     (define-syntax <Schlüsselwort> <Umwandlerspezifikation>)
```

7.2 Formale Semantik

Dieser Abschnitt bietet eine formale, denotationelle Semantik für die elementaren Ausdrücke von Scheme und ausgewählte eingebaute Prozeduren. Das hier benutzte Konzept und die Notation werden in [STOY77] beschrieben.

> *Anmerkung:* Der Abschnitt zur formalen Semantik wurde in LaTeX geschrieben, das mit TeXinfo inkompatibel ist. Siehe den Abschnitt zur formalen Semantik im Originaldokument, von dem dieses hier eine Bearbeitung ist.

7.3 Abgeleiteter Ausdruckstyp

Dieser Abschnitt gibt Makrodefinitionen für abgeleitete Ausdruckstypen aufgebaut aus den elementaren Ausdruckstypen an (Literal, Variable, Aufruf, '`Lambda`', '`If`', '`Set!`'). Siehe den Abschnitt Abschnitt 6.4 [Programmflussfunktionalitäten], Seite 68, für eine mögliche Definition von '`Delay`'.

```
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else ergebnis1 ergebnis2 ...))
     (begin ergebnis1 ergebnis2 ...))
    ((cond (test => ergebnis))
     (let ((temp test))
       (if temp (ergebnis temp))))
    ((cond (test => ergebnis) klausel1 klausel2 ...)
     (let ((temp test))
       (if temp
           (ergebnis temp)
           (cond klausel1 klausel2 ...))))
    ((cond (test)) test)
    ((cond (test) klausel1 klausel2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond klausel1 klausel2 ...))))
    ((cond (test ergebnis1 ergebnis2 ...))
     (if test (begin ergebnis1 ergebnis2 ...)))
    ((cond (test ergebnis1 ergebnis2 ...)
           clause1 klausel2 ...)
     (if test
```

```
          (begin ergebnis1 ergebnis2 ...)
          (cond klausel1 klausel2 ...)))))
```

Das folgende Beispiel nimmt an, dass die benutzte Scheme-Implementierung Umlaute auch als Buchstabenzeichen ansieht, als Erweiterung zur in diesem Bericht spezifizierten Syntax:

```
(define-syntax case
  (syntax-rules (else)
    ((case (schlüssel ...)
       klauseln ...)
     (let ((atom-schlüssel (schlüssel ...)))
       (case atom-schlüssel klauseln ...)))
    ((case schlüssel
       (else ergebnis1 ergebnis2 ...))
     (begin ergebnis1 ergebnis2 ...))
    ((case schlüssel
       ((atome ...) ergebnis1 ergebnis2 ...))
     (if (memv schlüssel '(atome ...))
         (begin ergebnis1 ergebnis2 ...)))
    ((case schlüssel
       ((atome ...) ergebnis1 ergebnis2 ...)
       klausel klauseln ...)
     (if (memv schlüssel '(atome ...))
         (begin ergebnis1 ergebnis2 ...)
         (case schlüssel klausel klauseln ...)))))


(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #f))))


(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))


(define-syntax let
  (syntax-rules ()
```

```
    ((let ((name wert) ...) rumpf1 rumpf2 ...)
     ((lambda (name ...) rumpf1 rumpf2 ...)
      wert ...))
    ((let tag ((name wert) ...) rumpf1 rumpf2 ...)
     ((letrec ((tag (lambda (name ...)
                      rumpf1 rumpf2 ...)))
        tag)
      wert ...))))


(define-syntax let*
  (syntax-rules ()
    ((let* () rumpf1 rumpf2 ...)
     (let () rumpf1 rumpf2 ...))
    ((let* ((name1 wert1) (name2 wert2) ...)
       rumpf1 rumpf2 ...)
     (let ((name1 wert1))
       (let* ((name2 wert2) ...)
         rumpf1 rumpf2 ...)))))
```

Das folgende Makro für '`Letrec`' benutzt das Symbol '`<undefiniert>`' für einen Ausdruck, der etwas liefert, was, wenn es an einer Stelle gespeichert ist, zu einem Fehler führt, wenn man versucht, den Wert an der Stelle zu ermitteln (kein solcher Ausdruck ist in Scheme definiert). Ein Trick wird benutzt, um temporäre Namen zu generieren, damit die Reihenfolge, in der die Werte ausgewertet werden, nicht festgelegt wird. Dies könnte auch über ein Hilfsmakro erreicht werden.

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec ((var1 anfang1) ...) rumpf ...)
     (letrec "erzeuge temp. Namen"
       (var1 ...)
       ()
       ((var1 anfang1) ...)
       rumpf ...))
    ((letrec "erzeuge temp. Namen"
       ()
       (temp1 ...)
       ((var1 anfang1) ...)
       rumpf ...)
     (let ((var1 <undefiniert>) ...)
       (let ((temp1 anfang1) ...)
         (set! var1 temp1)
         ...
         rumpf ...)))
    ((letrec "erzeuge temp. Namen"
```

```
        (x y ...)
        (temp ...)
        ((var1 anfang1) ...)
        rumpf ...)
      (letrec "erzeuge temp. Namen"
        (y ...)
        (neuetemp temp ...)
        ((var1 anfang1) ...)
        rumpf ...))))


(define-syntax begin
  (syntax-rules ()
    ((begin ausdruck ...)
     ((lambda () ausdruck ...)))))
```

Der folgende alternative Ausdruck für '`Begin`' benutzt die Möglichkeit nicht, mehr als einen Ausdruck in den Rumpf eines lambda-Ausdrucks zu schreiben. Beachten Sie jedenfalls, dass diese Regeln nur anwendbar sind, wenn der Rumpf des '`Begin`' keine Definitionen enthält.

```
(define-syntax begin
  (syntax-rules ()
    ((begin ausdruck)
     ausdruck)
    ((begin ausdruck1 ausdruck2 ...)
     (let ((x ausdruck1))
       (begin ausdruck2 ...)))))
```

Die folgende Definition von '`Do`' benutzt einen Trick, um die Variablenklauseln zu entfalten. Wie bei obigem '`Letrec`' würde ein Hilfsmakro auch funktionieren. Der Ausdruck '`(if #f #f)`' wird benutzt, um einen nicht näher bestimmten Wert zu erhalten.

```
(define-syntax do
  (syntax-rules ()
    ((do ((var anfang schritt ...) ...)
         (test ausdruck ...)
         befehl ...)
     (letrec
       ((loop
         (lambda (var ...)
           (if test
               (begin
                 (if #f #f)
                 ausdruck ...)
               (begin
                 befehl
```

```
              ...
              (loop (do "schritt" var schritt ...)
                    ...))))))
   (loop anfang ...)))
((do "schritt" x)
 x)
((do "schritt" x y)
 y)))
```

Bemerkungen

Sprachänderungen

Dieser Abschnitt zählt die Änderungen auf, die an Scheme vorgenommen wurde, seit der „Revised^4 report“ [R4RS] veröffentlicht wurde.

- Dieser Bericht ist nun eine Obermenge des IEEE-Standards für Scheme [IEEEScheme]: Implementierungen, die diesem Bericht hier genügen, werden auch dem Standard genügen. Dies machte folgende Änderungen erforderlich:
  - Es wird nun vorausgesetzt, dass die leere Liste als wahr zählt.
  - Die Klassifikation von Funktionalitäten als essenziell oder nicht essenziell wurde entfernt. Es gibt nun drei Klassen von eingebauten Prozeduren: elementare Prozeduren, Bibliotheksprozeduren und optionale Prozeduren. Die optionalen Prozeduren sind ‘`Load`’, ‘`With-input-from-file`’, ‘`With-output-to-file`’, ‘`Transcript-on`’, ‘`Transcript-off`’ und ‘`Interaction-environment`’, sowie ‘`-`’ und ‘`/`’ mit mehr als zwei Argumenten. Keine von diesen gibt es im IEEE-Standard.
  - Programme dürfen eingebaute Prozeduren umdefinieren. Dies wird das Verhalten anderer eingebauter Prozeduren nicht verändern.
- *Port* wurde zur Liste untereinander typfremder Typen hinzugefügt.
- Der Makro-Anhang wurde entfernt. Hochsprachliche Makros sind nun im Hauptteil des Berichts zu finden. Die Umschreiberegeln für abgeleitete Ausdrücke wurden durch Makrodefinitionen ersetzt. Es gibt keine reservierten Bezeichner.
- ‘`Syntax-rules`’ erlaubt nun Vektormuster.
- Liefern mehrerer Werte, ‘`Eval`’ und ‘`Dynamic-wind`’ wurden hinzugefügt.
- Die Aufrufe, die endständig implementiert werden müssen, werden ausdrücklich definiert.
- ‚‘`@`’‘ kann innerhalb von Bezeichnern verwendet werden. ‚‘`|`’‘ ist für mögliche zukünftige Erweiterungen reserviert.

Weiteres Material

Das Internet Scheme Repository unter

`http://www.cs.indiana.edu/scheme-repository/`

enthält eine umfassende Scheme-Bibliographie, sowie wissenschaftliche Arbeiten, Programme, Implementierungen und anderes Material, das mit Scheme zu tun hat.

Beispiel

'`Integrate-system`' integriert das System

$$y\_k\hat{}\hat{} = f\_k(y\_1, y\_2, \ldots, y\_n),\ k = 1, \ldots, n$$

von Differentialgleichungen mit der Methode von Runge-Kutta.

Der Parameter `System-derivative` ist eine Funktion, die einen Systemzustand nimmt (einen Vektor von Werten für die Zustandsvariablen y_1, ..., y_n) und produziert eine System-Ableitung (die Werte y_1^^, ...,y_n^^). Der Parameter `Initial-state` liefert einen initialen Systemzustand und `h` ist eine erste Schätzung für die Länge des Integrationsschritts.

Der von '`Integrate-system`' gelieferte Wert ist ein unendlicher Strom von Systemzuständen.

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

'`Runge-Kutta-4`' nimmt eine Funktion, `f`, die eine Systemableitung aus einem Systemzustand produziert. '`Runge-Kutta-4`' produziert eine Funktion, die einen Systemzustand nimmt und einen neuen Systemzustand produziert.

```
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
          (*2 (scale-vector 2))
          (*1/2 (scale-vector (/ 1 2)))
          (*1/6 (scale-vector (/ 1 6))))
      (lambda (y)
        ;; y ist ein Systemzustand
        (let* ((k0 (*h (f y)))
               (k1 (*h (f (add-vectors y (*1/2 k0)))))
               (k2 (*h (f (add-vectors y (*1/2 k1)))))
               (k3 (*h (f (add-vectors y k2)))))
          (add-vectors y
            (*1/6 (add-vectors k0
                               (*2 k1)
                               (*2 k2)
                               k3)))))))))

(define elementwise
```

```
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))

(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

‘`Map-streams`’ ist analog zu ‘`Map`’: Sie wendet ihr erstes Argument (eine Prozedur) auf alle Elemente ihres zweiten Arguments (ein Strom) an.

```
(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (delay (map-streams f (tail s))))))
```

Unendliche Ströme sind als Paare implementiert, deren Car das erste Element des Stroms enthält und deren Cdr ein Versprechen den Rest des Stroms zu liefern enthält.

```
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
```

Im Folgenden wird die Nutzung von ‘`Integrate-system`’ illustriert, um das System

$$C\ dv\_C / dt = -i\_L - v\_C / R$$
$$L\ di\_L / dt = v\_C$$

zu integrieren, welches einen gedämpften Oszillator modelliert.

```
(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
                (/ Vc L))))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

Bibliographie

- [SICP] Harold Abelson und Gerald Jay Sussman mit Julie Sussman. *Structure and Interpretation of Computer Programs, second edition.* MIT Press, Cambridge, 1996.
- [Bawden88] Alan Bawden und Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Seiten 86–95.
- [howtoprint] Robert G. Burger und R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Seiten 108–116.
- [RRRS] William Clinger, Herausgeber. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Auch veröffentlicht als Computer Science Department Technical Report 174, Indiana University, Juni 1985.
- [howtoread] William Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, Seiten 92–101. Sammelband veröffentlicht als *SIGPLAN Notices* 25(6), Juni 1990.
- [R4RS] William Clinger und Jonathan Rees, Herausgeber. The revised^4 report on the algorithmic language Scheme. In *ACM Lisp Pointers* 4(3), Seiten 1–55, 1991.
- [macrosthatwork] William Clinger und Jonathan Rees. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, Seiten 155–162.
- [propertailrecursion] William Clinger. Proper Tail Recursion and Space Efficiency. Zu erscheinen in *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, Juni 1998.
- [syntacticabstraction] R. Kent Dybvig, Robert Hieb und Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5(4):295–326, 1993.
- [Scheme311] Carol Fessenden, William Clinger, Daniel P. Friedman und Christopher Haynes. Scheme 311 version 4 reference manual. Indiana University Computer Science Technical Report 137, Februar 1983. Abgelöst von [Scheme84].
- [Scheme84] D. Friedman, C. Haynes, E. Kohlbecker und M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, Januar 1985.
- [IEEE] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 1985.
- [IEEEScheme] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.
- [Kohlbecker86] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* Doktorarbeit (PhD), Indiana University, August 1986.
- [hygienic] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen und Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, Seiten 151–161.
- [Landin65] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101, Februar 1965.

- [MITScheme] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.
- [Naur63] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, Januar 1963.
- [Penfield81] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings,* Seiten 248–256. ACM SIGAPL, San Francisco, September 1981. Sammelband veröffentlicht als *APL Quote Quad* 12(1), ACM, September 1981.
- [Pitman83] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, Mai 1983.
- [Rees82] Jonathan A. Rees und Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, Seiten 114–122.
- [Rees84] Jonathan A. Rees, Norman I. Adams IV und James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, Januar 1984.
- [R3RS] Jonathan Rees und William Clinger, editors. The revised^3 report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), Seiten 37–79, Dezember 1986.
- [Reynolds72] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, Seiten 717–740. ACM, 1972.
- [Scheme78] Guy Lewis Steele Jr. und Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, Januar 1978.
- [Rabbit] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, Mai 1978.
- [CLtL] Guy Lewis Steele Jr. *Common Lisp: The Language, second edition.* Digital Press, Burlington MA, 1990.
- [Scheme75] Gerald Jay Sussman und Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, Dezember 1975.
- [Stoy77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, 1977.
- [TImanual85] Texas Instruments, Inc. TI Scheme Language Reference Manual. Preliminary version 1.0, November 1985.

Alphabetisches Stichwortverzeichnis der Definitionen von Konzepten, Schlüsselwörtern und Prozeduren

Der Haupteintrag für jeden Begriff, jede Prozedur bzw. jedes Schlüsselwort wird zuerst aufgeführt, von den anderen Einträgen getrennt.

Konzepte

Prozeduren

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

V

W

Z

Inhaltsverzeichnis