Revised Report on the Algorithmic Language Scheme

Name : OECAL NACI

MatNr : 0327187

Kennzahl : 534

Universität : Technische Universität Wien

E-mail : naciocal@yahoo.com

INHALTVERZEICHNIS

EINFÜHRUNG

The Scheme ist eine funktionale Programmiersprache, die durch Guy L. Steele und Gerald Jay Sussman 1970 entwickelt wurde und über eine Reihe von Veröffentlichungen eingeführt, die jetzt als Sussman und Steeles Lambda Papiere gekennzeichnet sind. Scheme war eine der ersten Programmiersprachen, wie im Verfahren des Lambda Kalküls. Es wird die Verwendungsfähigkeit der statischen Bereichrichtlinien und Blockstruktur in einer dynamisch geschriebenen Sprache erprüft. Scheme war der erste Hauptdialekt von **LISP**. Jedoch gibt es drei wesentliche Merkmale, die Scheme von LISP unterscheidet. Zunächst, gilt in Scheme das Prinzip der lexikalischen Bindung ,während LISP dynamische Bindung verwendet. Zweitens, dass Scheme Standart schreibt „proper tail recursion" vor, dass bedeutet das Prozeduraufrufe, die in einer endrekursiven Position stattfinden, keinen Speicherplatz auf dem Stack verbrauchen dürfen. Eine weitere Unterscheidung ist, dass Makros in Scheme im Gegensatz zu LISP hygienisch ist. Nach einiger Zeit wurde Scheme die erste Programmiersprache, die hygienischen Makros unterstützt. Diese Makros haben die gleiche Syntax wie von einer blockierten-strukturieren Sprache. Man kann in Scheme unnötige Einschränkungen entfernen. Es gibt in Scheme Standart zum Beispiel keine Hilfsmittel zur OOP, aber durch Makros kann man sich solche in der Sprache programmieren. Scheme ist eine programmierbare Programmiersprache und kann dadurch von den Programmierern beliebig erweitert werden. Scheme stellte das Konzept der genauen und ungenauen Zahlen. Im Algemeinen repräsentiert diese Sprache die Verlängerung der generischen Arithmetik. Es gibt zwei Standards, die die Schemesprache definiert: der amtliche „***IEEE Standard***" und ein de Facto Standard benannten „***The Revised n-th Report on the Algorithmic Language Scheme***", fast immer abgekürztes RnRS, in dem n die Zahl der Neuausgabe ist. Derzeitige Spezifikation ist R5RS, an R6RS wird gearbeitet.

HAUPTDIALEKTEN VON LISP

SCHEME	XLISP
COMMON LISP	ISLISP
EMACS LISP	EULISP
MAC LISP	SML
INTER LISP	TCL LOGO
AUTO LISP	NET LISP
MULISP	

IMPLEMENTATIONS VON SCHEME

Gambit-C (Unix)
The DrScheme (Pc)
The Bigloo Scheme
MIT Scheme
Guile
The Scheme48 System
Pixie Scheme (Mac)

GRUNDLEGENDE ELEMENTE DER SCHEME

Kommentare

Kommentare werden durch ein Semikolon (;) eingeleitet und reichen bis zum Zeilenende.

Variablen

Variablen werden dynamisch geschrieben. Variablen werden durch entweder define oder let Aussage definiert. Variablen erklärten außerhalb jeder möglicher Funktion sind "im globalen" Bereich.

(define var1 value)

```
(let ((var2 value))
  ...)
```

Funktionen

Funktionen sind erstklassige Gegenstände im Scheme. Sie können Variablen zugewiesen werden. Z.B. kann eine Funktion mit zwei Argumenten arg1 und arg2 wie definiert werden

(define abc

```
   (lambda (arg1 arg2)
      ...))
```

welches abgekürzt werden kann, wie folgt:

(define (abc arg1 arg2)
  ...)

Funktionen können mit der folgenden Syntax benannt werden:

(abc value1 value2)

Merken Sie, daß die Funktion, die benannt wird, in der ersten Position der Liste ist, während der Rest der Liste die Argumente enthalten. The apply Funktion nimmt das erste Argument und apply den

Rest der Argumente zum ersten Argument, also kann der vorhergehende Funktion Anruf wie auch geschrieben werden

(apply abc (list value1 value2))

Listen

Listen werden in Scheme-Programmen relativ häufig gebraucht. Eine leere Liste wird mit (quote ()) gebildet.
Beispiel eine Liste,

(cons 1 (cons 3 (cons 5 (cons 7 '()))))

oder

```
 (list 1 3 5 7)
```

```
oder als
```

```
 '(1 3 5 7)
```

Listen in Scheme sind einfach verkettet. Das vordereste Element wird mit *car* extrahiert, der Rest der Liste mit *cdr* (sprich: „cudder“).

Beispiel:

```
(car (list 1 2 3 4))  ;ergibt  als Ausgabe: 1

(cdr (list 1 2 3 4))  ;ergibt als Ausgabe: (2 3 4)
```

Merken Sie den Anführungsstrich ('), diesen erklärt Entwurf, den Ausdruck nicht zu deuten, der dem Anführungsstrich folgt.

Vektoren

Vektoren sind Reihenfolgen wie Zeichenketten, aber ihre Elemente können alles, nicht gerechte Buchstaben sein. In der Tat können die Elemente Vektoren selbst sein, das eine gute Weise ist, mehrdimensionale Vektoren zu erzeugen.

Ist hier eine Weise, einen Vektor der ersten fünf Ganzzahlen zu verursachen:

```
(vector 0 1 2 3 4)
=>  #(0 1 2 3 4)
```

Anmerkung von Scheme eines vektorwertes: a # Buchstabe folgten vom Inhalt des Vektors, der in Klammern umgeben wurde. In der Analogie mit Bildenzeichenkette, bildet der Verfahren Bildenvektor einen Vektor von einer spezifischen Länge:

```
(define v (make-vector 5))
```

Weitere Datentypen

Weitere Datentypen sind unter anderem:

- integer (ganze Zahlen, beliebige Stellenzahl)
- rational (Brüche, beliebige Genauigkeit)
- real (Dezimalzahlen)
- complex (komplexe Zahlen)
- symbol
- string (Zeichenkette)
- port
- boolean

Wahr und *falsch* werden durch #t und #f dargestellt, wobei Scheme jedoch nur #f (in veralteten Implemantionen nur ein Synonym für leere Liste "()") als wirklich *falsch* interpretiert; alles andere gilt als *wahr*.

Conditionals

Cond

Mit *Cond* ist es möglich mehrere Fälle abzufangen. Trifft keiner dieser Fälle ein, so wird eine **else**-Behandlung für alle sonstigen Möglichkeiten eingeleitet:

```
(cond ((= wert 1) (display "Der Wert ist 1"))
      ((= wert 2) (display "Der Wert ist 2"))
      (else (display "Der Wert ist weder 1 noch 2)))
```

Darüber hinaus gibt es u. a. noch *when*, *unless*, *case* als weitere Möglichkeit mit Bedingungen zu arbeiten.

Scheme hat auch ***If***

If wertet einen Ausdruck aus, und führt je nach dessen Wahrheitswert (#t oder #f) eine entsprechende Anweisung aus:

```
(if (eq? wert #t)
    (display "Der Wert ist wahr")
  (display "Der Wert ist falsch"))
```

Schleifen

Schleifen werden in Scheme für gewöhnlich durch eine Rekursion erreicht. Eine Endlosschleife sieht im einfachsten Fall so aus:

```
(define (loop)
 (loop))
```

Ein häufig gezeigtes Beispiel, um dies zu demonstrieren, ist die Berechnung der faktorial

```
(define (fact n)
   (cond ((= n 0) 1)
         (else (* n (fact (- n 1))))))

 (fact 6)
 ;; => 720
```

VORTEILEN

Sie hat folgende entscheidende Vorteile gegenüber anderen Sprachen:

Scheme ist sehr klein: die gesamte Ausbildung kommt mit nur 6-8 Sprachelementen aus, während in jeder anderen Sprache Dutzende notwendig sind. Damit sind die Elemente der Sprache leicht zu merken; lästiges Nachschlagen entfällt weitgehend.

Scheme ist eine *funktionale Sprache*, die auf den Regeln der Schulalgebra basiert.

Scheme-Programme sind sehr kurz, ca. 3-10 mal kürzer als die entsprechenden Programme in anderen Sprachen.

Scheme ist geeignet, auch komplexe, interaktive Programme mit grafischer Benutzeroberfläche zu

schreiben. Konzepte anderer Programmiersprachen lassen sich grundsätzlich auch in Scheme formulieren: damit haben es Scheme-Könner erfahrungsgemäß leicht, sich im Selbststudium andere Sprachen anzueignen.

NACHTEILEN

Anders als scripting Sprachen wie Perl oder Pythons, wird Scheme nicht über seinem Kern hinaus standardisiert. Funktionen, die in einer Schemeimplementierung bestehen, brauchen nicht, in einer anderen Schemeimplementierung zu bestehen oder können einen vollständig anderen Namen und/oder eine Schnittstelle haben.

BEISPIELEN

Beispiel 1: This example evaluates the sum of the given list, and tested in EdScheme for Windows program.

```
(define (list-sum lis)
  (if (null? lis)   ; if empty list?
      0             ; then sum is zero
      (+ (car lis)  ; else sum is car plus the
        (list-sum (cdr lis))))) ; sum of rest of list
#void

=>(list-sum `( 1 3 4 5 6 7 8 9 0 11 ))
54
```

[2] Beispiel 2:

```
(define checkbook (lambda ()

                ; This check book balancing program was written to illustrate
```

```
; i/o in Scheme. It uses the purely functional part of Scheme.

; These definitions are local to checkbook
(letrec

  ; These strings are used as prompts

 ((IB "Enter initial balance: ")
  (AT "Enter transaction (- for withdrawal): ")
  (FB "Your final balance is: ")

  ; This function displays a prompt then returns
  ; a value read.

  (prompt-read (lambda (Prompt)

                 (display Prompt)
                 (read)))

  ; This function recursively computes the new
  ; balance given an initial balance init and
  ; a new value t.  Termination occurs when the
  ; new value is 0.

  (newbal (lambda (Init t)
          (if (= t 0)
             (list FB Init)
             (transaction (+ Init t)))))

  ; This function prompts for and reads the next
  ; transaction and passes the information to newbal

  (transaction (lambda (Init)
                 (newbal Init (prompt-read AT)))))

 ; This is the body of checkbook;  it prompts for the
 ; starting balance

 (transaction (prompt-read IB)))))
```

```
>(checkbook)
Enter initial balance: 1000
Enter transaction (- for withdrawal): -55
Enter transaction (- for withdrawal): +32
Enter transaction (- for withdrawal): -68
Enter transaction (- for withdrawal): +5
Enter transaction (- for withdrawal): 0
("Your final balance is: " 914)
```

ZUSAMMENFASSUNG:

Die Progrmmiersprache Scheme ist eine funktionale Programmierung und wird als LISP Dialekt bezeichnet. Unnötiges in Scheme kann sehr einfach entfernt werden. Diese Sprache ist eine programmierbare Programmiersprache und kann somit von Programmierern flexibel erweitert werden. Es gibt in dieser Sprache mehrere Implementationen und wird in allen Betriebsystemen unterstützt. Eine Implementation von Scheme genannt nach GUILE, wird als Skriptsprache, in der graphischen Oberfläche GNOME verwendet.

REFERENCES

[1] MIT/GNU Scheme
http://swiss.csail.mit.edu/projects/scheme/index.html

[2] Scheme Tutorial
http://cs.wwc.edu/~cs_dept/KU/PR/Scheme.html

[3] Dybvig, R. Kent.
The Scheme Programming Language, Third Edition
Copyright © 2003 The MIT Press. Electronically reproduced by permission.
http://www.scheme.com/tspl3/

[4] Springer, G. and Friedman, D.,
Scheme and the Art of Programming. The MIT Press, 1989.

[5] EdScheme for Windows : http://www.schemers.com/

[6] Revised(5) Report on the Algorithmic Language Scheme

http://download.plt-scheme.org/doc/103p1/html/r5rs/node3.htm

[7] Sitaram, Dorai.
Teach Yourself Scheme in Fixnum Days
http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-1.html#node_toc_start

[8] Sperber, Michael. Programmieren für Alle, 2002.

http://www.deinprogramm.de/overview/overview.html