Seminar
Parallelisierung und verteilte Ausführung funktionaler Programme
bei **Prof. Dr. Manfred Schmidt-Schauß**

Parallel Haskell (pH)

Razi Lotfi-Tabrizi
Matrikel-Nr.: 1035430
razi@informatik.uni-frankfurt.de
Fachbereich Informatik(20)
J.W. Goethe-Universität Frankfurt am Main
Vorgelegt am 24. Oktober 1996

Inhaltsverzeichnis:

1 Einleitung

Parallel Haskell (pH) ist eine parallele Sprache, die einen Parallelisierungsversuch der Haskell-Sprache darstellt.

Haskell ist eine funktionale Sprache, die nach dem Logiker Haskell B. Curry benannt wurde. Sie wurde von einem internationalen Kommitee, das aus 15 Forschern[1] bestand, entworfen.

Haskell wertet die Programme „lazy“ aus. Sie besitzt außerdem folgende Eigenschaften:

- Statisch starkes Typsystem
- Polymorphe Funktionen höherer Ordnung
- Algebraische Datentypen
- List-Comprehensions
- Typklassensystem
- Behandlung von Überladung (overloading)
- Pattern-Matching
- Keine Seiteneffekte

Im August 1993 traf sich eine Gruppe[2] von Wissenschaftlern, um Haskell zu einer parallelen Sprache zu erweitern. Das Ergebnis nannten sie „pH“.

Die funktionale Sprache pH ist auch eine nicht strikte (lazy) Sprache, die alle Eigenschaften der Haskell-Sprache besitzt. Hier wurden aber zum Zweck der Parallelisierung (synchroner Zugriff) zwei neue Datenstrukturen, nämlich die I- und M-Struktur, eingefügt.

Zweck dieser Arbeit ist es, diese Sprache zu beschreiben und dabei die verschiedenen funktionalen und parallelen Eigenschaften dieser Sprache dem Leser näher zu bringen. Am Ende werden auch die neuen Datenstrukturen erwähnt.

---

[1] Die Mitglieder: Arvid (MIT), Brian Boutel (Victoria University of Wellington), Jon Fairbairn (Cambridge University), Josef Fasel (Los Alamos National Laboratory), Kevin Hammond (University of Glasgow), Paul Hudak (Yale University), John Hughes (University of Glasgow), Thomas Johnson (Chalmes Institute of Technology), Dick Kieburtz (Oregon Graduate Center), Rishuyar Nikhil (MIT), Simon Peyton Jones (University College London), Mike Reeve (Imperial College), Philip Wadler (University of Glasow), David Wise (Indiana University) und Jonathan Young (Yale und MIT).

[2] Die Mitglieder: Arvid (MIT), Lennart Augustsson (Chalmers), James Hicks (Motorola MCRC), Simon Peyton Jones (Glasgow), Rishiyur Nikhil (DEC CRL), Joe Stoy (Oxford) und John Williams (IBM Research).

2 Funktionen

Wie bereits erwähnt, ist pH eine funktionale Programmiersprache. Diese Sprachen zeichnen sich dadurch aus, daß die Programme Funktionen berechnen, die Eingabedaten in Ausgabedaten abbilden. Eine Berechnung ist die Anwendung einer Funktion auf einer Liste von Werten oder Ausdrücken. Applikation ist die Bezeichnung für solche Anwendung einer Funktion auf einen Ausdruck. Aus diesem Grunde spricht man auch von applikativen Programmiersprachen.

Funktionen kann man in zwei Formen darstellen: als Operatoren, wie z.B. „+ oder `*+*`“, oder als Funktionsnamen (identifier) wie „`sUMME_'Quad'`“.

Das nächste Beispiel zeigt, wie eine Funktion in pH definiert werden kann:

```
sUMME_'Quad' x y      = ((quadrat x) + (quadrat y))
```
(Identifier)

```
x *+* y          = ((quadrat x) + (quadrat y))
```
(Operator)

„`sUMME_'Quad'`“ und „ `*+*` “ summieren die Quadrate von x und y (wobei `quadrat` noch definiert sein muß). Wie man sieht, haben die beiden Programme die gleiche Aufgabe; der Unterschied liegt nur in der Art der Funktion. Für beide Funktionen gilt:

Identifier = `(a|...|z)(a|...|z|A|...|Z|'|_)`*

Operator = `(( symbol | presymbol )( symbol | : )`*`)`

Wobei für `sysmbol` und `presysmbol` gelten muß:

`sysmbol` → `!|#|$|%|&|*|+|.|/|<|=|>|?|@|\|^||`

`presymbol` → `- | ~`

Wie in den meisten Programmiersprachen gibt es auch bei pH reservierte Wörter, die man nicht als Funktionsnamen benutzen darf:

case	class	data	default	deriving	else	hiding
if	import	in	infix	infixl	infixr	instance
interface	let	module	of	renaming	then	to
type	where	next	while	for	finally	

2.1 Darstellungsmöglichkeiten der Operatoren und Identifier

In pH hat man die Möglichkeit, Identifier als Operator und umgekehrt Operator als Identifier zu benutzen:

Jeden Identifier kann man durch das Einschließen in Apostrophe (' ') als Operatoren benutzen. `sUMME_'Quad'` kann man zum Beispiel als Infix-Operator wie folgt definieren:

```
x 'sUMME_'Quad'' y    = ((quadrat x) + (quadrat y))
```

Andererseits kann man Operatoren durch das Einschließen in Klammern als Identifier benutzen:

```
(*+*) x  y           = ((quadrat x) + (quadrat y))
```

Wie man sieht, wird „`*+*`“ als ein Präfix-Identifier benutzt.

In der pH-Sprache hat man die Möglichkeit des „Currying“. Currying ist ein starkes Werkzeug, das dem Benutzer erlaubt, eine Funktion mit wenig Argumenten laufen zu lassen. Man kann z.B. `(2 +)` als eine Funktion benutzen, die 2 addiert (auch `(+ 2)`). `(/2)` erzeugt eine Funkton, die halbiert; `(2 /)` ist eine Funktion, die `2/x` realisiert (wobei `x` das neue Argument sei). `(- x)` ist eine Ausnahme, die statt einer Funktion eine Negation des Wertes `x` zur Folge hat.

2.2 Priorität und Gruppierung

Jeder Operator hat bei pH eine Priorität. Das ermöglicht, daß man Klammern weglassen darf. Eine Priorität wird durch eine Zahl zwischen 0 und 9 bestimmt, wobei die Priorität aufwärts „stärker“ wird.

Man kann außerdem einem Operator eine Links- bzw. Rechtsassoziativität zuordnen. Die Assoziativität bestimmt die Reihenfolge der Auswertung.

In pH-Prelude werden die Operatoren folgendermaßen deklariert:

Priorität	Linksasso.	Nicht-asso.	Rechtsasso.
9	`!, !!, //, !., !^, !^^, //=, //^=, //^^=`		`.`
8			`**, ^,^^`
7	`*`	`%, /, ’div’, ’mod’, ’rem’, ’quot’`	
6	`+, -`	`:+`	
5		`\\`	`:, ++`
4		`/=, <, <=, ==, >, >=, ’elem’, ’notElem’`	
3			`&&`
2			`||`
1		`:=`	`>#>, >’., <>=`
0			`$`

Subtraktion wird z.B. linksassoziativ behandelt. Das heißt, daß „`1-2-3`“ als „`(1-2)-3`“ interpretiert wird.

Im Gegensatz dazu ergibt „`1/2/3`“ einen Syntaxfehler, weil „`/`“ keine Assoziativität besitzt (ist nicht gruppierbar).

Standardmäßig müssen alle Operatoren, die neu definiert werden, als nicht-assoziativ und mit der Priorität 9 behandelt werden. Mit Hilfe der INFIX-Befehle kann man jedoch diese Eigenschaften ändern:

`infixl` pr op	Mit diesem Befehl kann man den Operator (op) als linksassoziativ mit einer Priorität „pr“ definieren.
`infix` pr op	Der Operator ist nicht-assoziativ mit Priorität „pr“.
`infixr` pr op	Der Operator ist rechtsassoziativ mit Priorität „pr“.

In pH-Prelude steht z.B.:

`infixl`	`7`	`*`	
`infix`	`5`	`/, 'dev'`	
`infixr`	`5`	`++`	(List-Append)
`infix`	`4`	`'elem', 'notElem'`	(Ob (Argument $\in$ List))

3 Basis-Typen (Basic Types)

Wie bereits am Anfang erwähnt, besitzt pH ein statisch starkes Typsystem. Die starke Typisierung hat zur Folge, daß alle Ausdrücke getypt sein müssen und die Typfehler vom Compiler nicht toleriert werden. Durch dieses System wird sichergestellt, daß während der Laufzeit keine Fehler auftreten, die auf die Anwendung von Funktionen auf Argument des falschen Typs zurückzuführen sind. Der Ausdruck „ `2 + 'y'` “ wird schon zur Übersetzungszeit als fehlerhaft zurückgewiesen.

Eine Programmiersprache heißt „statisch typisiert“, wenn alle oder die meisten Typüberprüfungen zur Übersetzungszeit und nicht während der Laufzeit des Programms durchgeführt werden können.

In pH beginnen die Typnamen immer mit einem großen Buchstaben; formal ausgedrückt heißt es:

Typname = `(A|...|Z)(a|...|z|A|...|Z|'|_)*`

In der pH-Sprache existiert ein Typ-Operator „ `::` “, der jedem Ausdruck einen Typ zuordnet. In pH existieren verschiedene Typen. Die wichtigsten sind:

3.1 Bool

Ein Ausdruck vom Typ „`Bool`“ hat den Wert „`False`“ oder „`True`“. Zu diesem Typ sind die Standard-Funktionen `&&`, `||`, `not` und `otherwise` definiert. Es gilt:

```
data Bool =     False | True

(&&), (||)            :: Bool -> Bool -> Bool
True  && x            = x
False && _            = False
True  || _            = True
False || x            = x

not             :: Bool -> Bool
not True        = False
not False       = True

otherwise       :: Bool
otherwise       = True
```

Was die einzelnen Wörter wie z.B. „`data`“ bedeuten, wird in den folgenden Kapiteln erläutert; hier es geht darum zu sehen, wie solche Datentypen definiert werden.

3.2 Int und Float

„`Int`“ ist die Bezeichnung für den Integer-Typ; „`Float`“ bezeichnet den Typ für Gleitkomma-Zahlen. Es gilt:

integer → $(\text{digit})^+$

float → integer . integer [(e | E) [- | +] integer]

Wobei [x] bedeutet, daß der Inhalt (in diesem Fall x) optional gewählt wird.

3.3 Listen

Der Typ „ `[]` “ bezeichnet eine Liste. Die Liste ist eine algebraische Datentype[3] mit zwei Konstruktoren:

- Der Leere-Liste-Konstruktor „`[]`“
- Der Cons-Konstruktor „`cons`“

„ `[1,2,3]` “ ist z.B. eine Liste, die 1,2 und 3 als Elemente hat. Mit Hilfe von Cons-Konstruktor kann man am Anfang der Liste ein Element einfügen:

```
(4 : [1,2,3]) = [4,1,2,3]
```

Allgemein gilt:

$[e_1,...,e_n] \quad = \quad e_1 : (e_2 : (... (e_n : []) ...))$

In pH hat man die Möglichkeit, eine arithmetische Folge in Form von Listen zu erzeugen:

`[n..]` ist eine unendliche aufsteigende Liste, die mit `n` anfängt; z.B.:

```
[-2..] = [-2,-1,0,1,...
```

`[n..m]` Mit Hilfe dieser Liste erzeugt man eine Liste, die mit `n` beginnt und bei `m` endet (in aufsteigender Reihenfolge).

```
[-2..4] = [-2,-1,0,1,2,3,4]
[4..-2] = []
```

`[n,m..]` stellt eine unendliche Liste dar, die mit `n` und `m` anfängt. Die folgenden Elemente werden durch die Differenz aus `m` und `n` bestimmt.

```
[0,5..] = [0,5,10,15,20,...]
[5,0..] = [5,0,-5,-10,-15,...]
[9,9..] = [9,9,9,9,9,9,...]
```

`[n,i..m]` funktioniert genauso wie die vorherige Liste mit dem Unterschied, daß die Liste bei m endet:

```
[0,5..24] = [0,5,10,15,20]
[9,9..11] = [9,9,9,9,...]
```

In der pH-Sprache ist außerdem die List-Comprehension erlaubt. Sie ist eine hervorragend Technik zur Verarbeitung der Listen. Eine der Formen der List-Comprehension ist die „Generation“. Ein Generator hat die allgemeine Form „`[exp | pat <- exp ]`“. Hier benutzt man „`pat <- exp`“, um aus der Liste `exp` alle Elemente zu extrahieren, die das Argument `pat` ansprechen, zum Beispiel:

```
[ x + 2 | x <- [1..5] ] = [3,4,5,6,7]
```

oder

```
[ x * x | x <- [1..5] ] = [1,4,9,16,25]
```

[3] Siehe Kapitel 3.7.1 „Der algebraische Datentyp“

Die Listen sind linear definierte Datenstrukturen; daher ist es schwierig, auf die Listen parallele Funktionen zu definieren. Eine der parallelen Funktionen für die Listen ist „`reduce`“ (diese Funktion arbeitet wie die Funktionen `foldl` oder `foldr`, die man von Haskell oder Gofer kennt):

```
reduce                :: (a -> a -> a) -> a -> [a] -> a
reduce f z (a ++ b)   == (reduce f z a) 'f' (reduce f z b)
reduce f z [e]        == e
reduce f z []         == z
```

Wie man sieht, basiert diese Funktion auf „divide and conquer“-Algorithmus. Das Gesamtproblem „`reduce f z (a ++ b)`“ wird in zwei Teilprobleme „`(reduce f z a)`“ und „`(reduce f z b)`“ aufgeteilt, wobei die Teilprobleme dann parallel zueinander behandelt werden.

`z` soll das „neutrale Element“ der Funktion `f` sein. Mit Hilfe dieser Funktion kann man die Summation-Funktion „`sum`“, die die Elemente einer Liste miteinander addierte, folgendermaßen definieren:

```
sum = reduce (+) 0
```

3.4 Tupels

Tupels sind algebraische Datentypen, die mit Klammern „`()`“ erzeugt werden.

`(Float,Float)`	sind ein Paar aus Fließkomma-Zahlen.
`(Int, Char)`	ist ein Tupeltyp für den Tupel `(1,'a')`
`(Int, [Char], Bool)`	ist ein Trippel, der aus einem Integer, einem String (siehe Kapitel 3.5) und einem Booleschen Wert besteht; `(23,„Hilfe‟,True)` wäre ein solcher Trippel.

3.5 Zeichen (characters) und Strings

„`Char`“ ist der Zeichentyp und beinhaltet 256 Werte, von denen 128 die ASCII-Zeichen sind. Man stellt die Zeichenkonstanten durch Einschließen in Apostrophe (`''`) dar.

Die Strings werden in pH als eine Liste von Zeichen dargestellt. Es gilt:

```
type String    = [Char]
```

Die Zuweisung einer Zeichenkette erfolgt durch Einschließen in doppelte Anführungsstriche.

`‟ein Baum‟` ist die Abkürzung für `['e','i','n',' ','B','a','u','m']`.

3.6 Funktionsoperator (->)

Der Funktionsoperator wird zur Darstellung des Typs von Funktionen benutzt. Nimmt man das Beispiel „`sUMME_'Quad'`“ aus dem Kapitel 2, dann definiert man die Funktion folgendermaßen:

```
sUMME_'Quad' :: Float -> Float -> Float
```

`sUMME_'Quad'` ist eine Funktion, die zwei Argumente vom Typ `Float` erhält und als Ergebnis auch einen `Float`-Wert liefert.

„`map`" ist eine Funktion, die über die Liste definiert ist:

```
map            :: (a -> b) -> [a] -> [b]
map f []       =  []
map f (x:xs)   =  f x : map f xs
```

Das erste Argument der map-Funktion ist selber eine Funktion; das kann man an dem Funktionsoperator „`(a -> b)`" sehen. Das zweite Argument ist eine Liste; auch das Ergebnis ist eine Liste. Das Endergebnis der map-Funktion (`map f xs`) ist eigentlich synonym mit „`[f x | x <- xs]`" (siehe Kapitel 3.3 „List-Comprehension").

Beim genaueren Betrachten der map-Funktion erkennt man zwei wichtige Punkte:

- Die map-Funktion kann auf Listen beliebigen Typs arbeiten. In pH kann man „polymorphe Funktionen" definieren. Das sind Funktionen, die nicht auf einen bestimmten Argument- und Ergebnistyp festgelegt sind. Die map-Funktion ist ein gutes Beispiel für diese Art von Funktionen.
- In pH dürfen die Funktionen wie alle anderen Werte behandelt werden. Man setzt sie sogar als Argumente oder als Ergebnis einer Funktion ein. Man sagt: Man benutzt „Funktionen höherer Ordnung".

3.7 Benutzerdefinierte Datentypen

In diesem Kapitel werden die zwei Formen der benutzerdefinierten Datentypen eingeführt; der algebraische Datentyp (`data`) und das Typ-Synonym (`type`).

3.7.1 Der algebraische Datentyp (`data`)

Mit Hilfe des algebraischen Datentyps „`data`" kann man in pH einen neuen Datentyp und über diesen die entsprechenden Konstruktoren einführen. In pH-Prelude werden z.B. der Boolesche Typ und der List-Typ folgendermaßen eingeführt:

```
data Bool =    False | True          (Boolesche Typ)
data [a]  =    []    | a : [a]       (List-Typ)
```

Die allgemeine Form lautet:

data c => Datentyp u_1 ... u_n = $Konstr_1$ | ... | $Konstr_p$

wobei `Datentyp` der Name des neuen n-stelligen Typkonstruktors ist, $u_1 \ldots u_n$ verschiede Typvariablen sind und $Konstr_1$ ... $Konstr_p$ die Art beschreiben, in der die Elemente des neuen Datentyps konstruiert werden. `Datentyp` fängt immer mit einem großen Buchstaben an. Falls `Konstr` eine Konstruktorfunktion ist, schreibt man den Namen mit einem großen Buchstaben am Anfang; im Falle eines Konstruktoroperators beginnt der Name mit einem Doppelpunkt.

„`c =>`" ist ein Kontext, den man optional benutzt:

```
data Eq a => Set a    =     NilSet      | ConSet a (Set a)
```

Hier stellt „`Eq a =>`" sicher, daß `ConsSet` nur auf Werte angewendet wird, deren Typ ein Instanz der Klasse „`Eq`" ist.

3.7.2 Das Typ-Synonym (`type`)

Die Typ-Synonyme werden benutzt, um einen neuen Datentyp einzuführen, der äquivalent zu einem älteren (vordefinierten) Typ ist.

```
type Name a1 ... an   =     Ausdruck
```

`Name` beginnt immer mit einem großen Buchstaben. Ein Beispiel für das Typ-Synonym wurde in Kapitel 3.5 erwähnt; dort wurde die Zeichenkette (String) über die Liste der Zeichen definiert:

```
type String  =  [Char]
```

In pH existieren noch drei Datenstrukturen („Array“ , „I-Struktur“ und „M-Struktur“), die eine wichtige Rolle in pH spielen. Diese Datenstrukturen werden in Kapitel 5.2 behandelt.

4 Eigenschaften der pH-Sprache

Bisher wurden einige Eigenschaften (Polymorphismus, Funktionen höherer Ordnung, List-Comprehension, Algebraische Datentypen, Currying, statisch und starke Typsystem) der pH-Sprache erklärt. In diesem Kapitel wird auf die anderen wichtigen Eigenschaften der pH-Sprache - wie lazy Auswertung, keine Seiteneffekte, Pattern-Matching, Klassensystem und die Überladung - eingegangen.

4.1 Auswertungsmethoden

pH ist eine „lazy“ Sprache. Diese mächtige Eigenschaft erlaubt pH z.B., unendliche Datenstrukturen oder Listen zu verarbeiten. Es gibt zwei wichtige Auswertungsmethoden:

- Strikte Auswertung
- Lazy Auswertung

Um die beiden Methoden zu verstehen, muß man sich zuerst mit der LI- und LO-Reduktion (evaluation) vertraut machen. Im Zusammenhang mit der Auswertung tauchen häufig die zwei Begriffe Redex und Normalform auf. Redex (reducible expression) ist ein noch reduzierbarer Ausdruck. Enthält ein Ausdruck keinen Redex, so liegt der Ausdruck in der Normalform vor. Das Ziel einer Auswertung ist es, einen Ausdruck in seine Normalform zu überführen. Die beiden Funktionen `double` und `first` sind folgendermaßen definiert:

```
double    n      =      n + n
first           (a,b) =     a
```

Der Ausdruck „`double (double 4)`“ enthält zwei Redexe: Der gesamte Ausdruck ist ein Redex und der Teilausdruck `double 4` ist ein Redex. Man kann diesen Ausdruck in zwei verschiedenen Reihenfolgen auswerten:

double (double 4)	
LI-Reduktion:	LO-Reduktion:
double (4 + 4) double 8 8 + 8 16	double 4 + double 4 (4 + 4) + double 4 8 + double 4 8 + (4 + 4) 8 + 8 16

Bei der LI-Reduktion (leftmost innermost reduction) wird jeweils der am weitesten links stehende Redex, der keinen anderen Redex enthält, reduziert. In imperativen Sprachen spricht man auch von „call by value“. Diese Reduktionsmethode wird auch von der Funktionen einer strikten Programmiersprache verwendet. (Eine Funktion heißt strikt in einem bestimmten Argument, wenn die Berechnung eines Funktionsaufrufes immer dann nicht terminiert, wenn die Berechnung des Arguments nicht terminiert.)

Im Gegensatz dazu wird bei der LO-Reduktion (leftmost outermost reduction) der am weitesten links stehende Redex, der in keinem anderen Redex enthalten ist, ausgewählt. Hier spricht man bei den imperativen Sprachen vom „call by name".

Die LO-Reduktion ist gegenüber der LI-Reduktion im Nachteil, wenn ein Funktionsparameter wie bei double mehrfach im Funktionsrumpf auftritt. In einem solchen Fall wird der aktuelle Parameter, der unausgewertet in den Rumpf eingesetzt wird, mehrfach berechnet.

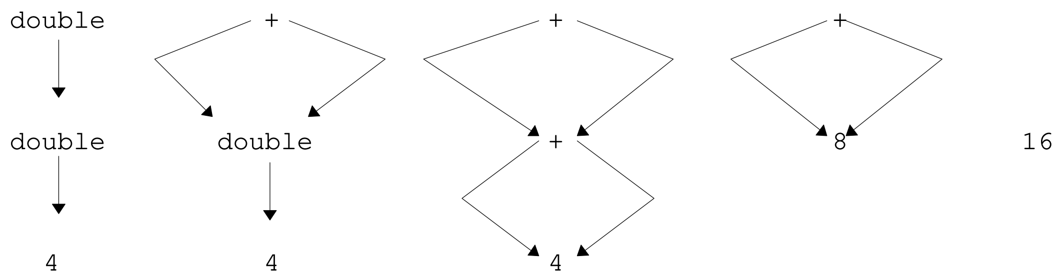
Tritt hingegen ein Funktionsparameter im Rumpf einer Funktion nicht auf (siehe folgendes Beispiel), so erweist sich die LO-Reduktion als vorteilhaft, da der aktuelle Parameter überhaupt nicht ausgewertet wird.

`first (5 , double 4)`	
LI-Reduktion:	LO-Reduktion:

`first (5 , 4 + 4)`
`first (5 , 8)`
`5`
	`5`

Der Unterschied in der Anzahl der Reduktionsschritte kann erheblich sein, besonders falls man mit sehr großen Listen oder Datenstrukturen zu tun hat.

Der Nachteil der Mehrfachauswertung bei der LO-Reduktion kann vermieden werden, indem man von der Termrepäsentation zu einer Graphenrepräsentation übergeht. Hier kann ein Teilausdruck mehrere „Vorgänger" haben. Wenn dieser Teilausdruck vereinfacht wird, erfahren alle Ausdrücke, die auf den Ausdruck verweisen, von der Vereinfachung. Für das vorherige Beispiel würde es heißen:



Die Pfeile verweisen jeweils auf die Argumente der Funktionen. LO-Reduktion angewendet auf eine Graphrepräsentation bezeichnet man als „Lazy-Reduktion" (nichtstrikte Auswertung). Sie wird „call by need" genannt, weil bei dieser Reduktion ein Ausdruck nur dann und nur einmal berechnet wird, wenn es unbedingt nötig ist und sie zum Endergebnis beiträgt.

Die lazy-Eigenschaft ermöglicht z.B. das Arbeiten mit undefinierten Eingaben; der Ausdruck „`first 10 (1/0)`" liefert `10`, obwohl das zweite Argument nicht terminiert wird. Möglich ist auch das Manipulieren der unendlichen Liste, wie das folgende Beispiel zeigt:

```
head ( tail [1..] )   =    2
```

4.2 Der Seiteneffekt

Die funktionalen Sprachen unterscheiden sich in einem Punkt völlig von den anderen Programmiersprachen (besonders von den imperativen Sprachen): Variablen sind keine Behälter (Speicherplätze), denen Werte zugewiesen werden, sondern Platzhalter für Werte und Funktionen, mit denen im mathematischen Sinne symbolisch gerechnet wird. In den imperativen Sprachen (Von-Neumann-Architektur) besteht ein Programm aus einer Folge der Anweisungen, mit der Aufgabe die Werte der Zellen im Hauptspeicher zu ändern. Bei den funktionalen Programmiersprachen bezeichnen Variablen Werte und nicht Speicherplätze.

In pH ist außerdem die dauerhafte Änderung des Speichers während einer Funktion nicht erlaubt. Man sagt, pH ist „seiteneffektfrei". Diese Eigenschaft hat zwar den Nachteil, daß sie nicht unbedingt von der Hardware unterstützt wird, aber den Vorteil, daß sie keine Sequentialisierung erzwingt. In Sprachen mit Seiteneffekt müssen die Speicherzellen nach einer bestimmten Reihenfolge verändert werden; dies verlangt stets eine Sequentialisierung des Programms. Die Sprachen, die keine Seiteneffekte besitzen, können daher einfacher parallelisiert werden.

4.3 Pattern und Pattern-Matching

In pH werden die Funktionen durch eine oder mehrere Gleichungen definiert:

```
f   pat1 pat2 ... patn      =     ausdruck
```

`pat1 pat2 ... patn` repräsentieren die Argumente der Funktion `f`; sie werden „Pattern" genannt. Die Anzahl der Pattern bestimmt damit die Stelligkeit der Funktion. Falls die Funktionen durch mehrere Gleichungen definiert werden, müssen alle Gleichungen bei der Definition hintereinander stehen und die gleiche Stelligkeit haben. Bei der Ausführung solcher Funktionen (mit mehreren Gleichungen) findet ein „Pattern-Matching" statt. Bei diesem Vorgang werden die Argumente der Funktion mit den Pattern der Definitionsgleichung verglichen (immer von oben nach unten). Falls ein Patter der Definitionsgleichung zu dem aktuellen Argument paßt (to match), wird diese Gleichung ausgeführt. Mit Hilfe des Pattern-Matching kann man die Fakultät-Funktion folgendermaßen definieren:

```
fak 0      =     1
fak n      =     n * fak(n - 1)
```

Man muß beachten, daß die Reihenfolge der Definition wichtig ist. Wenn man die Definition von oben wie folgt ändern würde, bekäme man bei Eingabe „`fak 1`" keine Antwort; die Funktion würde nicht terminieren.

```
fak n      =     n * fak(n - 1)
fak 0      =     1
```

Die Fakultät-Funktion wurde rekursiv definiert; man kann aber diese Funktion iterativ definieren:

```
fak n      =     fakul n 1

fakul      0 ergebnis  =     ergebnis
fakul      z ergenbis  =     fakul    (z-1)  (ergebnis*z)
```

„`z`" ist der Zähler

Ein weiteres Beispiel wurde bereits in Kapitel 3.1 behandelt:

```
True  && x          = x
False && _          = False

True  || _          = True
False || x          = x
```

Man kann z.B. die append-Funktion, die zwei Listen in einer Liste vereint, wie folgt definieren:

```
append_2  []     y      =     y

append_2  (a:xs)       y     =     a : append_2 xs y
```

Der Patter `[]` bzw. `a:xs` dient zur Fallunterscheidung zwischen der leeren Liste und einer Liste bestehend aus einem Kopfelement und einer Restliste.

Man kann Konstanten als Pattern verwenden; diese Eigenschaft wurde auch in Kapitel 3.1 benutzt, um die not-Funktion zu definieren:

```
not True        = False

not False       = True
```

Pattern müssen nicht immer Variablen oder Konstanten sein:

`_` oder `otherwise`	Beide sind äquivalent, und stimmen mit jedem Wert überein.
`n + k`	`k` ist eine natürliche Zahl. Der Patter matcht alle ganzzahligen Werte `m` mit `m>=k`; „`n+1`“ bezeichnet z.B. eine Integer-Zahl, die größer gleich 1 ist.
`[]`	bezeichnet eine leere Liste.
`a : x`	Der Listenkonstruktor matcht jede mindestens einelementige Liste. Die Muster `a` und `x` werden rekursiv gegen den Listenkopf bzw. gegen den Listenrest gematcht.
`[10,_,_]`	bezeichnet eine Liste mit genau drei Elementen, wobei das erste Element gleich `10` sein muß.
`(x:y:z)`	bezeichnet eine Liste mit mindestens zwei Elementen.
`p@(x,y)`	bezeichnet ein Paar; gestattet zusätzlich den Zugriff auf das gesamte Paar mit dem Namen p (p bezieht sich auf das gesamte Paar und nicht auf einzelne Elemente).

4.4 Typklassen und Überladung

Die class-Deklaration führt eine neue Typklasse ein und überlädt Funktionen und Operatoren, wobei die Bedeutung dieser überladenen Funktionen (oder Operatoren) vom Typ ihrer Argumente abhängt. `+, *, -` und `/` werden z.B. in pH-Prelude für verschiedene Arten von Zahlen, wie Integer, Fließkomma-Zahl, rational usw. verwendet. Man kann dann z.B. den Gleichheitsoperator (==) sowohl auf Integer als auch auf Zeichen (Char) definieren. Ein Auszug aus dem pH-Prelude verdeutlicht die Arbeitsweise der class-Deklaration:

```
class Eq a where

          (==), (/=)  ::    a -> a -> Bool
          x /= y           =     not (x == y)
```

Alle Typen, für die ein überladener Operator existiert, werden in Typklassen zusammengefaßt. Die Klasse `Eq` enthält z.B. alle Typen, die auf Gleichheit getestet werden können.

Am Anfang der Deklaration steht immer der Name der Klasse (in unserem Beispiel ist es `Eq`). Im zweiten Teil stehen die Funktionen und die Operatoren, die überladen werden müssen (== und /=), und deren Definitionstyp; das heißt, „==“ und „/=“ bekommen zwei Variablen (vom gleichen Typ); als Ergebnis erhält man einen Wert vom Typ „Bool“.

Mit „`instance`“ kann man in pH den Typ, auf dem die Operatoren einer Typklasse arbeiten sollen, definieren. Wenn man z.B. zwei Boolesche Werte miteinander mit Hilfe von „==“ und „/=“ vergleichen möchte, muß man den Typ `Bool` in die Typklasse `Eq` einfügen:

```
instance Eq Bool where
     True  == True     =     True
     False == False    =     True
     _        == _            =     False
```

Der Operator „/=“ muß nicht definiert werden, weil er bereits bei der Klassendefinition definiert wurde.

Man kann die Definition noch auf die Zeichen erweitern. Zu diesem Zweck braucht man die Funktion „`ord`“, die Zeichen auf Integer-Zahlen (ASCII-Code) abbildet:

```
instance Eq Char where
     c == d            =     ord c = ord d
```

Bei den selbstdefinierten algebraischen Datentypen, wie z.B. Tupel oder Liste, ist das Benutzen von `instance` etwas anders:

```
instance (Eq a , Eq b)      =>    Eq (a,b) where
     (x,y) == (u,v)    =     x==u && y==v
```

„`(Eq a, Eq b) => Eq (a,b)`“ bedeutet: Wenn Typ `a` und Typ `b` zu `Eq` gehören, sollen auch Paare aus Elementen vom Typ `a` und `b` auf Gleichheit prüfbar sein.

Die Listen werden folgendermaßen deklariert:

```
instance Eq a => Eq [a] where
     []     == []            =     True
     []     == (y:ys)        =     False
     (x:xs)== []             =     False
     (x:xs)== (y:ys)         =     x==y && xs==ys
```

„`Eq a => Eq [a]`“ bedeutet: Falls der Typ `a` der Klasse `Eq` angehört, soll auch der Typ `[a]` der Klasse `Eq` angehören. Diese Gleichheit ist explizit für alle Konstruktor-Möglichkeiten definiert. Diese Methode ist zwar legal, aber in pH benutzt man eine einfachere Methode. Gleich bei der Definition des Typs (mit Hilfe von algebraischen Datentypen) erledigt man mit Hilfe von `instance` diese Aufgabe. Man schreibt:

```
data [a]  =     [] | a:[a]        deriving (Eq, Ord)
```

Mit Hilfe von „`deriving`“ werden für den Datentyp „`[a]`“ in den Klassen „`Eq, Ord`“ die instance-Deklarationen erzeugt.

In der Ordnungsklasse „`Ord`“ werden die Variablen auf ihre Ordnung geprüft.

```
class (Eq a) => Ord a where
    (<), (<=), (>=), (>)    ::    a -> a -> Bool
    max, min                ::    a -> a -> a

    x <  y                  =     x <= y && x /= y

    x >= y                  =     y <= x

    x >  y                  =     y < x

    max x y       | x >= y    =     x
                  | y >= x    =     y
                  |otherwise  =     error

    min x y       | x <= y    = x
                  | y <= x    = y
                  |otherwise  = error
```

Hier verwendet man Wächter „|“, die in Kapitel 5.1.2 behandelt werden.

5 Ausdrücke und Datenstrukturen

Im ersten Teil dieses Kapitels wird versucht, die wichtigsten Ausdrücke, wie if-then-else-Ausdruck, lambda-Ausdruck usw., zu erläutern. Im zweiten Abschnitt werden die Datenstrukturen, wie Array, I- und M-Struktur, behandelt.

5.1 Ausdrücke

5.1.1 lambda-Ausdrücke

In pH kann man mit Hilfe von lambda-Ausdrücken namenlose Funktionen definieren:

\ *p1 ... pn* -> *ausdruck*

Die *p1...pn* demonstrieren Pattern. Das folgende Beispiel verdoppelt die Eingabe:

```
(\x ->    2 * x)
```
oder
```
(\x ->    x + x)
```

5.1.2 Wächter

Jede Gleichung einer Funktionsdefinition darf „Wächter" (guards) haben, die Bedingungen angeben, welche von den Argumenten der Funktion erfüllt werden müssen:

f *x1 x2 ... xn* | *bedingung1* = *e1*
| *bedingung2* = *e2*
...
| *bedingungm* = *em*

Man kann dann z.B. die Fakultät-Funktion folgendermaßen definieren:

```
fak n    |   n = 0       =    1
         |   n > 0       =    n * fak(n-1)
         |   otherwise   =    error
```

5.1.3 Lokale Definitionen

Man hat in pH die Möglichkeit der lokalen Definition; zu diesem Zweck werden entweder let- oder where-Ausdrücke benutzt:

5.1.3.1 let

Der let-Ausdruck hat die Form „`let` {*d1* ; ... ; *dn*} `in` *Ausdruck*". *d1...dn* soll Deklarationen (es dürfen auch Funktionen sein) darstellen, wobei pH die Anweisungen *d1...dn* parallel ausführt.

Das nächste Beispiel berechnet den Umfang eines Kreises mit Radius `r`:

```
umfang r  =     let
                pi = 3.14
                a  = 2
          in r*a*pi
```

5.1.3.2 where

Der where-Ausdruck hat die gleiche Aufgabe wie der let-Ausdruck. Er hat aber die allgemeine Form „*Ausdruck* `where {`*d1* `;...;` *dn*`}`“. Das Beispiel von oben kann man mit Hilfe des where-Ausdrucks so schreiben:

```
umfang r =      r*a*pi
                where
                      pi = 3.14
                      a  = 2
```

Man muß beachten, daß pH das Einrücken einzelner Programmteile in den Ausdrükken gegenüber dem Rest des Programms zur Erkennung der Struktur des Programms verwendet (es sind die let-, where-, case-of-, do-, seq- und par-Ausdrücke gemeint). Normalerweise benutzt man „`{ ;...; }`“ anstelle des Einrückens der Programmteile. Dann sieht die `umfang`-Funktion wie folgt aus:

```
umfang r =      r*a*pi
                where { pi=3.14 ; a=2}
```

5.1.4 Bedingte Ausdrücke

In pH existieren zwei Wege, bedingte Ausdrücke zu definieren:

5.1.4.1 if-then-else

if-then-else-Ausdrücke werden genauso wie in anderen Sprachen benutzt:

`if` *bedingung* `then` *ausdruck1* `else` *ausdruck2*

Beispiel:

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

if-then-else-Ausdruck ist in seinem Bedingungsbereich (*bedingung*) strikt.

5.1.4.2 case

Der case-Ausdruck hat die allgemeine Form

`case` *ausdruck* `of {`*p1* `->` *match1* `;...;` *pn* `->` *matchn*`}.`

Hier wird der Ausdruck (*ausdruck*) mit *p1* bis *pn* verglichen; bei der ersten Übereinstimmung mit *pi* wird *matchi* ausgeführt. Die Fakultät-Funktion definiert mit „case“ sieht so aus:

```
fak n    =      case  n  of
                      0           ->    1
                      otherwise   ->    n * fak (n-1)
```

(Hier wird davon ausgegangen, daß $n \in N^0$ sei).

Falls bei der Auswertung des case-Ausdruckes keine Alternative zutrifft, erhält man eine Fehlermeldung.

Es ist erlaubt, jeden *matchi*-Ausdruck durch zusätzliche Wächter zu ersetzen. Dann steht für *matchi*:

pat | *bedingung1* -> *e1*
| *bedingung2* -> *e2*
...
| *bedingungn* -> *en*

5.1.5 Schleifen-Ausdrücke

Ein Schleifen-Ausdruck besteht aus zwei Bereichen, nämlich aus dem Schleifenkopf und aus dem Körper. Der Schleifenkopf enthält eine Bedingung, die immer wieder auf Richtigkeit geprüft wird. Im Schleifenkörper (Rumpf) stehen der auszuführende Teil des Programms und die Anweisungen.

5.1.5.1 while-do-finally

Diese bedingte Schleife hat die allgemeine Form:

`while` *bedingung* `do {`*d1 ;...; dn*`}` `finally` *ausdruck*

Immer dann wenn Bedingung „WAHR" ist, wird der Rumpf des Programms neu berechnet, sonst springt man zum finally-Ausdruck. Man kann mit Hilfe des while-Ausdrucks einen Zähler bauen, der bis „100" zählt, und am Ende „100" ausgibt:

```
let s=0
in
    while (s < 100) do
          next s = s +1
    finally s
```

Mit Hilfe des nächsten Beispiels kann man die Funktion f im Interval [`x1,x2`] integrieren:

```
let s = 0.0
    x = x1
in
    while (x <= x2) do
       next s = s + f(x)
       next x = x + delta_x
    finally s
```

Wie man sieht, ist dieser while-Ausdruck eine iterativ definierte Funktion. Es ist möglich, sie auch mit Hilfe des if-then-else-Ausdrucks rekursiv zu definieren:

```
h x1 x2    =     g 0.0  x1 x1 x2
g s x n m =     if (x>m)
                      then s
                      else g (s + f(x)) (x + delta_x) n m
```

Im while-Ausdruck taucht auch der Begriff „`next`" auf. Eine next-Variable darf nur im Rumpf (`{`*d1 ;...; dn*`}`) einer Schleife (while- oder for-Ausdruck) auftreten. Ihr Geltungbereich ist der Rumpf der Schleife. Sie darf in der Schleife wie eine normale Variable benutzt werden. Mit Hilfe des Befehls „`next s = s + f(x)`" weist man der Variable `s` einen neuen Wert zu. Der Grund für diese Markierung der Variable durch „`next`“ ist, daß man keinen Seiteneffekt zulassen möchte.

5.1.5.2 for-do-finally

Die for-Schleife hat die allgemeine Form:

`for` *pat* `<-` *ausdruck1* `do` {*d1 ;...;dn*} `finally` *ausdruck2*

Die Fakultät-Funktion läßt sich mit der for-Schleife auf folgende Weise beschreiben:

```
fak n     =     let   f = 1
                in
                      for j <- [1..n] do
                            next f = f * j
                      finally f
```

5.1.6 seq und par

pH ist eine parallele Sprache, die die Programme synchron ausführt. Hin und wieder muß man Programme schreiben, die eine Sequenzialisierung verlangen. Da hilft das Schlüsselwort „`seq`“. „`par`“ arbeitet genauso wie „`seq`“ mit dem Unterschied, daß man „`par`“ zur Parallelisierung einsetzt.

`seq` und `par` sind einfache Kontrollstrukturen; sie beeinflussen den Geltungsbereich der einzelnen Programme überhaupt nicht. Sie kommen im Deklarationsbereich „{*d1;...;dn*}“ der let-, where- und Schleifenausdrücke vor. Falls im Deklarationsbereich keines dieser Schlüsselwörter `seq` oder `par` steht, wird automatisch `par` genommen; `d1` bis `dn` werden parallel ausgeführt.

Das nächste Beispiel zeigt, wie man `seq` einsetzen kann:

```
fak n     =     let   ergebnis = 1
                      zaehler  = 1
                in
                      while ( zaehler <= n) do
                            seq   next ergbnis      = ergebnis * zaehler
                                  next zaehler      = zaehler + 1
                      finally ergebnis
```

5.2 Datenstrukturen

In pH existiert die „array“-Datenstruktur. Darüber hinaus wurden noch zwei neue Feldertypen, „I-Struktur“ und „M-Struktur“-Felder, in die Sprache eingegliedert. Im Gegensatz zur anderen Sprachen, hat man hier beim Schreib- und Lese-Befehl die Möglichkeit des synchronen Zugriffs. Das ermöglicht eine parallele Arbeitsweise. Das Einsetzen der I- und M-Felder hat auch Nachteile: Durch die Verwendung der I- und M-Felder können Seiteneffekte entstehen.

In diesem Kapitel wird versucht, die Eigenschaften der I- und M-Felder zu erläutern und die verschiedenen Zugriffs- und Manipulationsbefehle aufzuzeigen.

Für die I- und M-Felder existieren zwei verschiedene Zustände: „leer“ und „voll“.

Ein I-Feld ist ursprünglich im Zustand „leer“. Wenn eine Zuweisung auf dieses Feld unternommen wird, geht es zum Zustand „voll“ über. Wenn eine zweite Zuweisung erfolgt, ergibt das gesamte Programm einen Fehler (error). Ein Ausdruck, der den Inhalt eines Feldes liest, muß warten, bis das Feld voll wird; erst dann darf der Ausdruck den Inhalt des Feldes ausgeben.

Das M-Feld ist genauso wie das I-Feld am Anfang im Zustand „leer“. Nach einer Zuweisung geht das Feld auch in den Zustand „voll“ über. Bei einem Lesevorgang passieren zwei Dinge:

- Es wird gewartet, bis das Feld voll wird. Dann wird der Wert des Feldes ausgegeben.
- Der Zustand des Feldes wird auf „leer“ gesetzt.

Wenn zwei Ausdrücke versuchen sollten, gleichzeitig den Inhalt des Feldes zu lesen, wird nur einer von beiden Erfolg haben. Der Erfolglose muß so lange warten, bis das Feld wieder voll ist. Zuweisungen sind nur auf ein leeres Feld erlaubt; die Zuweisung auf ein volles Feld führt zum Fehler.

Die M-Felder haben zusätzlich noch zwei Befehle: `examine` und `replace`. Sie unterstüzen auch das synchrone Bearbeiten der Felder.

Der `examine`-Befehl liest den Inhalt des Feldes, läßt aber das Feld im Zustand „voll“. Er hat die gleiche Aufgabe wie der herkömmliche Lese-Befehl. Der Befehl hat intern zwei Aufgaben: Er liest das Feld und gibt den Inhalt aus; dann schreibt er das Feld mit dem gleichen Inhalt und setzt den Zustand auf „voll“.

Der zweite Befehl „`replace`“ ersetzt den Inhalt des Felds (im Zustand „voll“) mit einem anderen Wert. Dieser Befehl ist eine Zusammensetzung des Lese- und Schreib-Befehls, wobei beim Schreiben der neue Wert eingefügt wird.

Mit Hilfe der nächsten Beispiele werden die wichtigsten Funktionen für I- und M-Felder erklärt.

Das folgende Beispiel beschreibt eine Funktion, die als Eingabe ein Iarray „`a`“ erhält. Das Iarray „`a`“ enthält nur Integer-Zahlen. Als Ergebnis liefert die Funktion das Iarray „`b`“. Am Anfang des Feldes „`b`“ stehen die negativen Elemente des Iarrays „`a`“, gefolgt von den Positiven.

```
let
    (l,u) = ibounds a
    b     = iarray (l,u) []
    kl    = l
    ku    = u
    for j <- [l..u] do
          (j1, next kl, next ku)  = if (a!.j < 0)   then (kl, kl+1, ku)
                                                    else (ku, kl, ku-1)
          b!.j1                   = a!.j
    finally ()
in
    b
```

„`ibounds`“ ist eine Funktion, die ein Iarray erhält, und die die Index-Grenze des Iarrays ausgibt (`iBounds :: (Ix a) => Iarray a b -> (a,a)`). Die Index-Grenze eines Feldes besteht aus einem Zahlenpaar, in dem der Anfang- und der End-Index des Feldes festgelegt sind. Mit dem Befehl „`(l,u)    = ibounds a`“ weist man dem Paar `(l,u)` die Index-Grenze des Feldes „a“ zu.

Mit dem Befehl „`iarray`“ erzeugt man ein Iarray. „`iarray`“ hat zwei Argumente: Eine Index-Grenze und eine Liste. Diese Liste besteht aus Paaren.

„`iarray (1,2) [(1,999),(2,10100)]`“ ist z.B. ein Iarray mit zwei Elementen „999“ und „10100“. Das Element „999“ hat den Index „1“ und „1000“ den Index „2“. Mit dem Befehl „`b = iarray (l,u) []`“ erzeugt man ein leeres Feld mit dem Namen „b“ (Initialisierung eines Feldes).

Man kann auch bei der Erzeugung des Feldes die Eigenschaften der Listen ausnutzen, z.B. „List-Comprehension“:

„`iarray (1,9) ((1,1):[(i , i*(a!.(i-1))) | i <- [2..9]]`“ enthält Elemente „`(1,1),(2,2),(3,6),(4,24),(5,120)(6,720)(7,5040)(8,40320)(9,362880)`“

Mit dem Befehl „`a!.j`“ erhält man das Element des Iarrays „`a`“ mit dem Index „`j`“ (siehe Zeile sieben des Programms auf Seite 21). Man kann den !.-Operator auch zum Zweck der Zuweisung benutzen. „`b!.j1 = a!.j`“ bedeutet: Speichere das j-te Element des Iarrays „`a`“ unter dem Index „`j1`“ des Iarrays „`b`“. Zum Speichern gibt es auch den Befehl „`iastore`“, der als Eingabe ein Element und den entsprechenden Index erhält.

Die bisher erwähnten Funktionen für Iarray existieren auch für das Marray. Natürlich müssen diese Funktionen sich an die Eigenschaften der M-Felder halten; z.B.:

`!^`	Der Operator arbeitet wie „`!.`“; er löst aber das gelesene Feld (siehe Lese-Befehl der „M-Struktur“.
`!^^`	Der Operator ist der examine-Befehl für Marray.
`mastore`	Die Funktion speichert ein Element unter einem Index.
`mareplace`	Ersetzt ein Element des Marray durch ein neues Element.

In pH gilt allgemein für alle Array-Typen:

- Wenn bei der Konstruktion eines Arrays zwei Elemente den gleichen Index beanspruchen, ergibt das gesamte Programm einen Widerspruch („ $\top$ “).
- Wenn bei der Konstruktion eines Arrays ein Element einen Index erhält, der nicht in der Index-Grenze des Arrays liegt, dann ist diese Zuweisung undefiniert („$\perp$“).

Zusammenfassung

Die funktionale Sprache „pH“ gehört zur Klasse der höheren Sprachen. Sie bietet darüber hinaus die Möglichkeit der parallelen Programmierung. Sie ist eine einfache und elegante Programmiersprache. Die Definition der Funktionen in pH ähnelt sehr ihren mathematischen Definitionen. Diese Eigenschaft der pH-Sprache macht es den meisten Programmieren sehr einfach, sie zu lernen und zu bedienen.

Literaturverzeichnis

Brian Boutel, Jon Fairbairn, Josef Fasel, Kevin Hammond, Paul Hudak, John Hughes, Thomas Johnson, Dick Kieburtz, Rishuyar Nikhil, Simon Peyton Jones, Philip Wadler, Maria M. Guzmán, Will Partain, John Peterson*:* Report on the Programming Language Haskell, Version 1.2, USA: März 1992.

Ralf Hinze: Einführung in die funktionale Programmierung mit Miranda, B.G. Teubner Stuttgart: 1992

Rishiyur S. Nikhil, Arvind, James Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, Yuli Zhou: pH Language Reference Manual, Version 1.0, USA: Januar 1995.

Manfred Schmidt-Schauß: Einführung in die funktionale Programmierung: Vorlesung WS 1995/96

ERROR: undefined
OFFENDING COMMAND:

STACK: