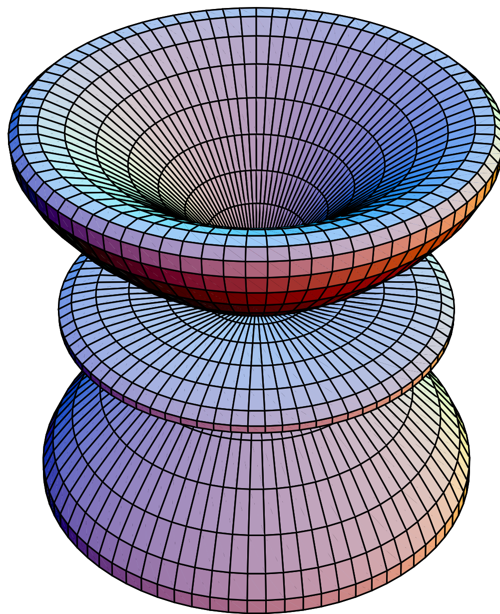
Institut für Theoretische Physik
LV-Nr. 132.456

Symbolische Mathematik in der Theoretischen Physik

Christoph Strnadl
Robert Lorenz
Werner Scholz



1. (korrigierte) Auflage
Juni 2000

Benutzungsbestimmungen

- Konzeption und Inhalt der Vorlesung:
  Christoph F Strnadl (c.strnadl@ieee.org)
- Übertragung/Zusammenfassung/Skriptum:
  Werner Scholz (e9426502@student.tuwien.ac.at)
- Jede kommerzielle Weiterverwendung ist untersagt.
- Die unentgeltliche Weitergabe (im Sinne des GNU Copyleft s.u.) ist möglich und erwünscht.

Copyright (c) Christoph F Strnadl, Robert Lorenz, Werner Scholz, Martin Ertl
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU Free Documentation License".

Inhaltsverzeichnis

1 Einleitung **4**

2 *Mathematica* **4**

2.1 Aufbau 4

2.2 User-Interfaces 5

2.3 Das Paradigma von *Mathematica*: *Pattern Matching* 5

3 Grundlagen **8**

3.1 *Everything is an expression* 8

3.2 Muster (*Patterns*) 10

3.3 *Evaluation* 20

4 Funktionale Methoden **24**

4.1 Grundlagen und Konzept 24

4.2 Fundamentaloperatoren: `Nest`, `Fold`, `Map`, `Apply`, `Thread` . . . 24

4.3 *Pure Functions* 28

4.4 *Advanced Operators*: `Inner`, `Outer` 31

5 *1-d Forest Fire Simulation* **32**

5.1 Problembeschreibung: Wachsen/Anzünden – Brennen – Absterben 32

5.2 Implementation der einzelnen Phasen 33

6 Objektorientierte Methoden **37**

6.1 Konzepte 37

6.2 Objekte 37

6.3 *Operator Overloading* 39

6.4 `Packages` und `Context` 41

7 GNU Free Documentation License **45**

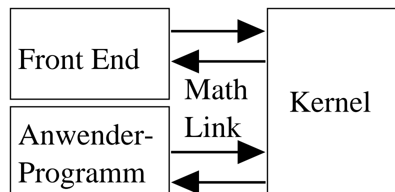
1 Einleitung

Stephen Wolfram, geboren 1959 in London, erhielt seine Schulausbildung in Eton. Im Alter von fünfzehn Jahren verfaßte er seine erste wissenschaftliche Publikation, mit siebzehn studierte er in Oxford und schloß seine Studien drei Jahre später am California Institute of Technology mit einem Ph.D. in Theoretischer Physik ab. In der Folge forschte er auf den Gebieten der Hochenergiephysik, Quantenfeldtheorie und Kosmologie. Daneben beschäftigte er sich mit der wissenschaftlichen Verwendung von Computern und entwickelte ein Computer Algebra System, das 1981 erstmals kommerziell vermarktet wurde. Seine wissenschaftliche Laufbahn führte Stephen Wolfram vom Caltech über das Institute for Advanced Study in Princeton schließlich zu einem Lehrstuhl für Physik, Mathematik und Computerwissenschaften an der Universität von Illinois. Seine Forschungstätigkeit auf dem Gebiet komplexer Systeme führte zur Gründung des ersten Forschungszentrums und der ersten Fachzeitschrift für dieses Wissenschaftsgebiet im Jahr 1986. Es ist auch das Geburtsjahr von *Mathematica*, in dem die Entwicklung dieses Computer Algebra Systems begann. 1988 kam die erste Version auf den Markt. Drei Jahre später erschien Version zwei und 1997 wurde die Entwicklung von Version drei abgeschlossen.

2 *Mathematica*

2.1 Aufbau

Das *Mathematica*-System besteht aus zwei Hauptteilen, dem Kernel und dem Front End. Als Benutzer sieht man nur das Front End, die Benutzerschnittstelle, die Eingaben entgegennimmt, Ausgaben auf ein entsprechendes Gerät leitet (Terminal, Grafikfenster, Lautsprecher, Drucker) und viele zusätzliche Informationen liefert. Der Kernel ist für die Berechnungen zuständig. Er kommuniziert über *MathLink* mit dem Front End, nimmt Befehle entgegen und liefert immer ein Ergebnis oder eine Fehlermeldung zurück. Außerdem können in einer höheren Programmiersprache geschriebene Programme mittels *MathLink* mit *Mathematica* kommunizieren.



2.2 User-Interfaces

Command Line

Die *command line* stellt – im Gegensatz zu den *notebooks* – eine auf allen Plattformen einheitliche Benutzerschnittstelle dar.

Nach dem *Mathematica*-Prompt `In[n]:=...` können beliebige Anweisungen eingegeben werden. Sobald die Zeile mit RETURN abgeschlossen wird, übergibt das Front End den Befehl an den Kernel, der gegebenenfalls geladen und gestartet wird. Dieser versucht die Anweisung auszuführen und liefert ein Ergebnis zurück. Dieses wird in einer Zeile, die mit `Out[n]=` ... beginnt, ausgegeben.

Notebooks

Ein spezielle Aufbereitungsform für *Mathematica*-Sessions stellen *notebooks* dar. Sie sind hierarchisch in Zellen organisiert, die ein einfaches Editieren von Input- und Kommentieren von Output-Zeilen ermöglichen. Zusätzlich kann einfacher Text eingefügt und formatiert werden. Auch Grafiken werden direkt in notebooks eingebunden. Die Reihenfolge der *Mathematica*-Befehle muß dabei nicht mehr mit jener, in der sie geschrieben wurden, übereinstimmen.

2.3 Das Paradigma von *Mathematica*: *Pattern Matching*

Die Grundlage der meisten Operationen, die *Mathematica* ausführt, ist der Vergleich von Mustern. Dem Kernel steht eine große Datenbank mit vordefinierten Mustern zur Verfügung, die beispielsweise zur Umformung und Vereinfachung mathematischer Ausdrücke verwendet werden. Neue Muster können dann der Datenbank hinzugefügt werden, um *Mathematica* neue Fähigkeiten beizubringen und spzielle Probleme zu lösen.

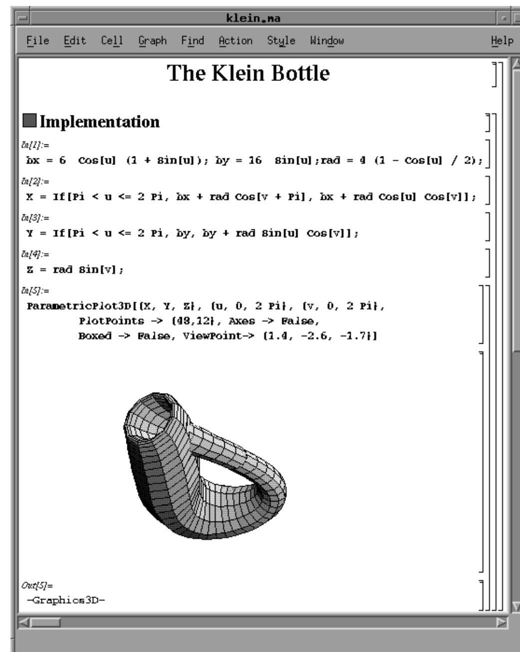


Abbildung 1: *Mathematica* Notebook

Ingredienzien

Kontrollstrukturen verschiedener Programmiersprachen

- FORTRAN:
```
      DO 100 i=1, 10
      ...
100   CONTINUE
```
- C: `for (i=1; i<100; i++){...}`
- LISP: Listen `(PLUS A(TIMES 4 B))`
- PROLOG: `parent[x|X]:-` (Regeln)
- Functional Programming: APL, J

Vereinheitlichung von 3 Konzepten

Die Syntax, der Befehle auf der command line oder in notebooks gehorchen müssen, beruht auf folgenden Konzepten:

- Mathematische Funktionen: $f(x)$ $\quad f: \mathbb{R} \to \mathbb{R} \quad x \mapsto x^2$
- Muster: `f[x_]:=x^2`

```
In[1]:= f[x_]:=x^2
```

- Funktionsaufrufe: $f(3) = 9$

```
In[2]:= f[3]
Out[2]= 9
```

Beispiel

```
In[3]:= f[1]:=0

In[4]:= f[a]:=b
```

Die Musterdatenbank hat nach obigen Eingaben folgende Einträge für das Symbol `f`:

```
f/:   f[1]  := 0
      f[a]  := b
      f[x_]  := x^2
g/:   ...
```

Der Befehl

```
In[5]:= ?f
Out[5]= Global`f
        f[1]  := 0
        f[a]  := b
        f[x_]  := x^2
```

zeigt die „Muster“, die in der Datenbank gespeichert sind, an.

Läßt man *Mathematica* den Befehl

```
In[6]:= f[3]
Out[6]= 9
```

ausführen, so kann man sich den Vorgang des Mustervergleichs wie folgt vorstellen:

Mathematica beginnt, den eingegebenen Befehl zeichenweise mit den gespeicherten Mustern zu vergleichen, und erkennt zunächst, daß es sich um ein `f` handelt. Unter jenen Mustern, die damit beginnen sucht es jenes, das von einer eckigen Klammer `[` gefolgt wird. Nun wird das dritte Zeichen `3`

untersucht. Offensichtlich paßt es weder auf den ersten, noch auf den zweiten Eintrag in der Datenbank. Der dritte Eintrag `f[x_]:=x^2` enthält ein spezielles Zeichen, das sogenannte „blank“ („`_`“), das für einen beliebigen Ausdruck (expression) steht. Dieses ist jedoch kein Leerzeichen, sondern wird durch einen Unterstrich dargestellt. Kapitel 3.2 beschäftigt sich eingehend mit blanks. Dem blank geht ein `x` voran, das lediglich der expression, die dem blank zugeordnet wurde, einen Namen gibt, um auf diese zugreifen zu können. Da das blank für eine beliebige expression steht, ist zweifellos auch `3` erlaubt, weshalb das dritte Muster als zutreffend erkannt wird.

Mathematica kann dann mit der Auswertung des Ausdrucks beginnen. In `f[x_]:=x^2` wird `3` an Stelle des beliebigen Ausdrucks eingesetzt: `x^2` wird ersetzt durch `3^2`. Das Symbol `^` für das Potenzieren wird durch seine `FullForm` ersetzt, `Power[3, 2]`, und der Ausdruck schließlich ausgewertet.

3 Grundlagen

3.1 *Everything is an expression*

Jeder Ausdruck, jede expression hat folgende allgemeine Form:

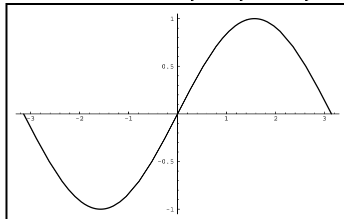
In[1]:= *head* [arg_1, arg_2,...]

Beispiel

```
In[2]:= Plot[Sin[x],{x,-Pi,Pi}]
```

Gibt die Funktion $\sin x$ im Intervall $[-\pi, \pi]$ graphisch aus.

Out[2]=



```
-Graphics-

In[3]:= {1, 2, a, 5} // FullForm
Out[3]= List[1, 2, a, 5]

In[4]:= a+b // FullForm
Out[4]= Plus[a, b]
```

Bedeutung des Kopfes	Bed. der Argumente	Beispiel
Funktion	Funktionsargumente	`Sin[x], f[x, 1]`
Befehl, Kommando, Prozedur	Parameter	`Plot[Cos[x],{x,0,2 Pi}]`
Operator	Operanden	`Plus[4, a]`
Objekte	innere Zustände	`{A,Sin[y],1,{5},U}`

Darstellung von Ausdrücken

Bezeichnung	Notation	Beispiel
Standard	*f*`[`*x*`, `*y*`]`	`Plus[5,7]` → `12`
Prefix-Notation	*f* `@` *x*	`Expand @ ((x+y)^2)` → `Expand[(x+y)^2]` → `(x^2+2xy+y^2)`
Postfix-Notation	*x* `//` *f*	`3/4 // N` → `0.75`
Infix	*x* ~ *f* ~ *y*	`5~Plus~7` → `5+7` → `12` `{1, 2}~Join~{3, 4}` → `{1, 2, 3, 4}`

Teile von Ausdrücken

Auf das n-te Argument eines Ausdrucks kann durch folgende Notation zugegriffen werden:

```
In[5]:= expr[[n]]
```

```
In[6]:= Plus[A, Times[4, b]][[2]]
Out[6]= 4 b
In[7]:= Plus[A, Times[4, b]][[0]]
Out[7]= Plus
In[8]:= {1, 2, A}[[-1]]
Out[8]= A
```

Ist der Index negativ, so beginnt die Zählung beim letzten Argument.

Es kann auch gleichzeitig auf mehrere Argumente zugegriffen werden:

```
In[9]:= expr[[{n1, n2,..., nm}]]
Out[9]= {expr[[n1]], expr[[n2]],..., expr[[nm]]}
```

Die wiederholte Indizierung ermöglicht den Zugriff auf einzelne Elemente in verschachtelten Ausdrücken oder Listen:

```
In[10]:= expr[[n]][[m]]
```

```
In[11]:= {{1, 2, 3}, {A, B, C}}[[2]][[3]]
Out[11]= C
```

3.2 Muster (*Patterns*)

Ein Muster steht für eine Klasse von Ausdrücken mit derselben internen Struktur.

- Definitionen, Zuweisungen

  `f[x_]:=x^3`

- Transformation Rules

  expr `/.\` *lhs* `->` *rhs*

  Jedes Auftreten von *lhs* in *expr* wird durch *rhs* ersetzt. Der Ausdruck wird jedoch nur einmal durchlaufen.

  expr `//.\` *lhs* `->` *rhs*

  Jedes Auftreten von *lhs* in *expr* wird durch *rhs* ersetzt. Und zwar so lange bis sich sich der resultierende Ausdruck nicht mehr ändert.

Beispiel

```
In[1]:= 3/a^2 + a/(2b^2)
Out[1]= 3/a^2 + a/(2b^2)
```

```
In[2]:= % /. x_^2 -> x
Out[2]= 3/a^2 + a/(2b^2)
```

Warum werden die quadratischen Terme nicht korrekt ersetzt?

```
In[3]:= %% // FullForm
Out[3]= Plus[
        Times[3, Power[a, -2]],
        Times[Rational[1, 2], a,
        Power[b, -2]]
        ]
```

a wird intern in den Zähler gehoben und mit der negativen Potenz versehen!

```
In[4]:= FullForm[x_^2]
Out[4]= Power[Pattern[x, Blank[]], 2]
```

Arten von Mustern

Beliebige Ausdrücke

Das Blank _ steht für einen einzigen, aber beliebigen Ausdruck. Um im weiteren auf diesen zugreifen zu können, muß er benannt werden:

*name_*

Ausdrücke mit einem bestimmten Kopf

Das Muster kann zusätzlich auf jene Ausdrücke eingeschränkt werden, die einen bestimmten Kopf (head) haben:

name_head.

Beispiel

```
In[5]:= Clear[f]
In[6]:= f[a_Integer]:= a-1
```

a muß den Kopf Integer haben, damit dieses Muster angewendet werden kann.

```
In[7]:= f[5]
Out[7]= 4
In[8]:= Head[u]
Out[8]= Symbol
In[9]:= f[u]
Out[9]= f[u]
```

u hat offenbar den falschen Kopf.

```
In[10]:= f[1, 4]
Out[10]= f[1, 4]
In[11]:= f[a]:=a-1
```

Hier fehlt das blank!

```
In[12]:= f[5]
Out[12]= 4
```

f[a] paßt nicht, aber das zuvor definierte f[a_Integer].

```
In[13]:= f[a]
Out[13]= -1 + a
```

Wichtige Köpfe

x_Integer	Head[5] → Integer
x_Real	Head[4.3] → Real
x_List	Head[{1, 4, 6}] → List
x_Symbol	Head[a] → Symbol
x_Automat	Automat[$expr_1$,...]

Beispiel

```
In[14]:= aut=Automat[{1, 2, 3, 4}, {1 -> 2, 2 -> 3}]
Out[14]= Automat[{1, 2, 3, 4}, {1 -> 2, 2 -> 3}]
```

`{1, 2, 3, 4}` könnte etwa eine Menge von vier verschiedenen Zuständen repräsentieren.

```
In[15]:= States[a_]:=Length[a[[1]]]

In[16]:= States[5]
Out[16]= Part::partd:  Part specification 5[[1]] is longer
         than depth of object.
         2

In[17]:= Clear[States]

In[18]:= States[a_Automat]:=Length[a[[1]]]

In[19]:= States[5]
Out[19]= States[5]

In[20]:= States[aut]
Out[20]= 4
```

Ausdrücke mit bestimmten Eigenschaften

name_?testf

testf ist eine Funktion, deren Ergebnis `True` oder `False` sein kann.

Beispiel

```
In[21]:= IntegerQ[1]
Out[21]= True

In[22]:= IntegerQ[a]
Out[22]= False

In[23]:= IntegerQ[{a, 1}]
Out[23]= False

In[24]:= NumberQ[1]
Out[24]= True
```

```
In[25]:= NumberQ[3.5]
Out[25]= True

In[26]:= NumberQ[4/5]
Out[26]= True

In[27]:= NumberQ[Pi]
Out[27]= False

In[28]:= Pi > E
Out[28]= Pi > E
```

Symbolische Konstanten können nicht direkt miteinander verglichen werden.

```
In[29]:= Pi > E //N
Out[29]= True
```

Weitere „Testfunktionen“:

`EvenQ[]`	Argument gerade ?
`OddQ[]`	Argument ungerade ?
`PrimeQ[]`	Primzahl ?
`VectorQ[]`	Vektor / Liste ?
`MatrixQ[]`	Matrix / Liste von Listen geeigneter Länge ?

`PrimeQ[]` arbeitet mit einem probabilistischen Algorithmus. Daher wird bei großen Argumenten möglicherweise `False` geliefert, obwohl es sich um eine Primzahl handelt. Der umgekehrte Fall kann nicht auftreten.

Beispiele verschiedener Kombinationen

```
In[30]:= Clear[f]

In[31]:= f[x_?syma]:=1
```

Da die Funktion nur eine Konstante liefert, kann die Benennung `x` der „beliebigen expression“ auch unterbleiben.

```
In[32]:= f[_?syma]:=1

In[33]:= syma[x_]:=x===a
```

Selbstdefinierte Testfunktion. `===` vergleicht, ob die beiden Ausdrücke identisch sind.

```
In[34]:= f[u]
Out[34]= f[u]

In[35]:= syma[u]
Out[35]= False

In[36]:= f[a]
Out[36]= 1

In[37]:= Clear[f]

In[38]:= f[x_Symbol?syma]:=x

In[39]:= f[1]
Out[39]= f[1]

In[40]:= Head[1]
Out[40]= Integer

In[41]:= f[u]
Out[41]= f[u]

In[42]:= f[a]
Out[42]= a

In[43]:= f[{u_List, n_Integer}]:=u[[n]]

In[44]:= f[{{A}, 1}]
Out[44]= A

In[45]:= f[A, 1]
Out[45]= f[A, 1]
```

Hier werden zwei Argumente übergeben. Es wird jedoch eine Liste mit zwei Elementen erwartet.

```
In[46]:= f[{A, 1}, 2]
Out[46]= f[{A, 1}, 2]
```

Abermals zwei Argumente statt einer Liste.

```
In[47]:= f[{A, 5}]
Out[47]= f[{A, 5}]
```

Es wird zwar eine Liste übergeben, aber `A` ist keine Liste.

```
In[48]:= f[{{A, B}, 3}]
Out[48]= Part::partw:  Part 3 of A, B does not exist.
         {A, B}[[3]]
```

Alternative Schreibweise

$pattern_1 \,|\, pattern_2 \,|\, \ldots$

Beispiel

```
In[49]:= h[a|b]:=2
In[50]:= h[c]
Out[50]= h[c]
In[51]:= h[a]
Out[51]= 2
```

Eine bessere, übersichtlichere Schreibweise ist:

```
In[52]:= h[a]:=2
In[53]:= h[b]:=2
```

Transformation Rules

```
In[54]:= {a, b, c, d} /. {(a|c)->A}
Out[54]= {A, b, A, d}
In[55]:= {a[2],b[3],c[4],d[5]} /. {(f:(a|b))[x_]->r[f, x]}
Out[55]= {r[a, 2], r[b, 3], c[4], d[5]}
```

Nebenbedingungen (Constraints)

pattern `/;` *cond*
lhs `:=` *rhs* `/;` *cond*

Die Nebenbedingung *cond* muß `True` liefern, damit das Muster paßt bzw. die Definition angewendet wird.

Beispiel

Fakultät händisch nachprogrammiert:

```
In[56]:= f[1]:=1
In[57]:= f[n_ /; n>1]:=n f[n-1]
```

oder

```
In[58]:= f[n_]:=n f[n-1] /; n>1
```

oder

```
In[59]:= f[n_Integer]:=n f[n-1] /; n>1
```

oder

```
In[60]:= f[n_]:=n f[n-1] /; IntegerQ[n] && n>1
```

oder

```
In[61]:= f[n_Integer /; n>1]:=n f[n-1]
```

Default-Werte

name_ : value
name_head: value

Bisher mußten immer alle Argumente übergeben werden, damit ein Muster erkannt oder ein Definition verwendet wurde. Durch die Festsetzung von Default-Werten kann man mehrere Argumente als optional deklarieren und bestimmte Voreinstellungen vorgeben.

Beispiel

```
In[62]:= get[x_, n_:1]:=x[[n]]
Out[62]= General::spell1:  Possible spelling error:  new
         symbol name "get" is similar to existing symbol
         "Get".

In[63]:= get[{1, 2}]
Out[63]= 1
```

Da n nicht übergeben wurde wird dafür 1 eingesetzt.

```
In[64]:= get[{a, b}, 2]
Out[64]= b

In[65]:= get[{a, b}, 3]
Out[65]= Part::partw:  Part 3 of
         a, b does not exist.
         {a, b}[[3]]
```

Die Liste enthält nur zwei Elemente.

```
In[66]:= Clear[get]

In[67]:= get[x_, n_:1]:=x[[n]] /; Length[x]>=n

In[68]:= get[{a, b}, 3]
Out[68]= get[{a, b}, 3]

In[69]:= get[{a, b}, E]
Out[69]= get[{a, b}, E]

In[70]:= get[{a, b, c}, -2]
Out[70]= b

In[71]:= get[h[u, v], -3]
Out[71]= Part::partw:  Part -3 of h[u, v] does not exist.
         h[u, v][[-3]]
```

```
In[72]:= get[h[u, v], -2.6]
Out[72]= Part::pspec:  Part specification -2.6 is neither an
         integer nor a list of non-zero integers.
         h[u, v][[-2.6]]

In[73]:= get[x_, n_Integer:1]:=x[[n]] /; Positive[n] &&
         Length[x]>=n

In[74]:= get[x_, n_Integer?Positive:1]:=x[[n]] /;
         Length[x]>=n

In[75]:= get[x_, n:(_Integer?Positive):1]:=x[[n]] /;
         Length[x]>=n
```

Mehrere beliebige Ausdrücke

Das double blank „__" steht für einen oder mehrere beliebige Ausdrücke. Analog zum einfachen blank kann das Muster näher spezifiziert werden:

*name*__
*name*__*head*
*name*__?*testf*

Beispiel

```
In[76]:= Clear[h]

In[77]:= h[x__]:={x} /; Length[{x}]>2

In[78]:= h[a]
Out[78]= h[a]

In[79]:= h[a, 1, b]
Out[79]= {a, 1, b}
```

Fehlerhafte Beispiele

```
In[80]:= Clear[h]

In[81]:= h[x__]:={x} /; Length[x]>2
```

```
In[82]:= h[1, a, 2, b]
Out[82]= Length::argx:  Length called with 4 arguments; 1
         argument is expected.
         h[1, a, 2, b]

In[83]:= Clear[h]

In[84]:= h[x__Integer]:=x

In[85]:= h[a]
Out[85]= h[a]

In[86]:= h[5]
Out[86]= 5

In[87]:= h[5, 9]
Out[87]= Sequence[5, 9]
```

Muster für viele (beliebige) Ausdrücke

`___` „Triple Blank"
name`___`
name`___`*head*
name`___?`*testf*

Beispiel

```
In[88]:= h[{a_Integer, b_Integer, c_Integer,
         r___Integer}]:={a, b, c, r}

In[89]:= h[{1, 2}]
Out[89]= h[{1, 2}]
```

Das dritte Argument `c` fehlt. Daher kann obiges Muster nicht verwendet werden

```
In[90]:= h[{1, 2, 3}]
Out[90]= {1, 2, 3}

In[91]:= h[{1, 2, 3, 4, 5, 6, 7}]
Out[91]= {1, 2, 3, 4, 5, 6, 7}

In[92]:= h[{a_Integer, b_Integer, c_Integer, r___Integer}]:=r
```

```
In[93]:= h[{1, 2, 3, 4, 5, 6, 7}]
Out[93]= Sequence[4, 5, 6, 7]
```

r wird intern nicht als Liste, sondern als Sequence gespeichert.

Grundregeln für die Verwendung von „___“ (aus der Praxis gewonnen):

___ sollte auf beiden Seiten begrenzt sein:

- durch ein anderes Muster mit mindestens einem Ausdruck
  z.B. ..., x___, y__, ...
- durch ein „strukturelles Ende“
  z.B. ..., y___] oder ..., y___}

Belegungsregeln für Mehrfachmuster

Mit folgendem Befehl kann man die Belegung herausfinden

```
In[94]:= h[x__, y__]:={x, y} /; Print["x=", x, "y=", y]
```

Das Ergebnis einer Nebenbedingung muß True ergeben, damit sie als zutreffend erkannt wird. Print wird nie True zurückliefern, erlaubt uns jedoch, die Belegung der Argumente zu untersuchen. Wird eine sinnvolle Nebenbedingung verwendet, dann werden alle Möglichkeiten, die Argumente zu verteilen, durchprobiert, bis die Nebenbedingung erfüllt ist. Alle weiteren Möglichkeiten, auch wenn sie die Nebenbedingung erfüllen, werden ignoriert.

h[x__, y__]	x	y
h[a, b]	a	b
h[a, b, c]	a	b, c
	a, b	c
h[a, b, c, d]	a	b, c, d
	a, b	c, d
	a, b, c	d

h[x__, y___]	x	y
h[a]	a	
h[a, b]	a	b
	a, b	
h[a, b, c]	a	b, c
	a, b	c
	a, b, c	

3.3 *Evaluation*

1. Der Benutzer gibt eine expression ein.
2. Das Front End übergibt die expression an den Kernel.
3. Der Kernel evaluiert die expression und
4. liefert eine expression als Ergebnis zurück.
5. Das Front End stellt das Ergebnis in geeigneter Form dar.

Eigenschaften der „Evaluation Seqence“

Die Evaluation geschieht durch die Anwendung von Definitionen

- explicit definitions
  `h[x_]:=x^3; a=7.4; f[1]:=4`
- built-in definitions
  `5+2` → `Plus[5, 2]` → `7; Print["x=",x]`

Mathematica ist ein „infinite evaluation system“
Das bedeutet, daß es

- *alle* Definitionen verwendet und zwar
- so lange, bis sich das Ergebnis nicht mehr ändert

Beispiel

```
In[1]:= x1=x2+7; x2=7
Out[1]= 7
```

```
In[2]:= x1
Out[2]= 14
```

Obwohl `x2` erst nach `x1` definiert wurde, erhält man das korrekte Ergebnis.

Beispiel

```
In[3]:= t:=0

In[4]:= x:=0 /; (t = t+1 ; t > 3)
```

`In[5]:= x` `Out[5]= x`	t=1
`In[6]:= x` `Out[6]= x`	t=2
`In[7]:= x` `Out[7]= x`	t=3
`In[8]:= x` `Out[8]= 0`	t=4: *Mathematica* ist doch kein „infinite evaluation system“.

Obwohl (oberflächlich betrachtet) keine Regel geändert wurde, erhält man plötzlich ein anderes Ergebnis. Zugegeben, `t` wird hochgezählt und ändert damit den Inhalt der Datenbank von *Mathematica*, an der Definition von `x` ändert sich aber nichts.

Standard Evaluation Sequence

Wie untersucht *Mathematica* eine Eingabe, eine expression

`In[9]:=` h `[`e_1`,` e_2`,...,` e_n`]`

die dem Kernel vom Front End übergeben wird?

1. Handelt es sich bei der expression um eine raw expression? Eine raw expression ist eine durch Anführungszeichen (`"..."`) begrenzte Zeichenkette oder eine Zahl (Integer oder Real). Ist dies der Fall, so wird die evaluation sequence abgebrochen, da nichts weiter zu tun ist, und die expression unverändert zurückgegeben.

2. Eine raw expression ist auch ein Symbol ohne weitere Regeln. (In *Mathematica*-Sprech OwnValues genannt.) Folglich gibt es nichts in die Datenbank einzutragen und die expression wird wieder unverändert zurückgegeben.

3. Evaluiere die Regeln des Symbols (OwnValues)
   z.B.: `a /; a:=b`

4. Hat sich etwas seit der letzten Evaluierung geändert? Falls nicht, wird abgebrochen und die expression zurückgegeben.

5. Evaluiere den Kopf h der expression. Auch der Kopf selbst kann eine expression sein: `a[1][A, {1, 2}]`

6. Evaluiere jedes Element e_i der expression unter Beachtung der Attribute `HoldFirst`, `HoldRest` und `HoldAll`.

7. Transformiere die expression entsprechen den Attributen von h.

   - Flat ($\hat{=}$ assoziativ): *h*`[` *h*`[x, y],` *h*`[z] ]` $\rightarrow$ *h*`[x, y, z]`
   - Listable: *h*`[{x, y}]` $\rightarrow$ `{`*h*`[x],` *h*`[y]}`
   - Orderless ($\hat{=}$ kommutativ): *h*`[z, a, y]` $\rightarrow$ *h*`[a, y, z]`

   Attribute können mit `SetAttributes[`*h*`,` *Attribut*`]` gesetzt und mit `ClearAttributes[`*h*`,` *Attribut*`]` gelöscht werden.

8. Wende UpValues (betreffend die Argumente e_i) an, zuerst user-defined, dann built-in. (siehe Kap.6.3 Operator overloading)
   Bsp.: `f/:  g[f[x_]]^:=gf[x]`
   (Diese Definition wird bei `f` gespeichert, nicht bei `g`!)

9. Wende DownValues (*h*`[.]`) und SubValues (*h*`[.][.]...`) bezüglich h an, zuerst user-defined, dann built-in. DownValues entsprechen den üblichen Transformation Rules.
   Bsp.: `f/:  f[{a_, b_}]:=a[f[b]]`

Beispiel

```
In[10]:= Clear[f]; Clear[g]

In[11]:= f[a_]:=a^2

In[12]:= g:=f

In[13]:= g[5]
Out[13]= 25
```

In Schritt 3 wird `g` durch `f` ersetzt, die Evaluation Sequence von neuem begonnen und in Schritt 6 das Ergebnis berechnet.

Non-Standard Evaluation Sequence

Assignments

Immediate Assignment: *lhs* = *rhs*
interne Speicherung: Set[*symbol*, *rhs*]

lhs kann eine beliebige expression oder ein Muster sein. Verschiedene Zuweisungen, die einem bestimmten Symbol zugeordnet sind, werden in der Reihenfolge der Eingabe gespeichert, außer sie ist spezifischer als die vorhergehenden. In diesem Fall wird das Muster vorangestellt. Neue Zuweisungen mit identischer *lhs* überschreiben alte Einträge.

Ist die Eingabe von der Form

In[14]:= f[arg_1, ..., arg_n]=rhs
Out[14]= rhs

so werden die Argumente arg_1 bis arg_n ausgewertet. f und f[arg'_1, ..., arg'_n] bleiben unverändert im Gegensatz zu *rhs*, das sofort ausgewertet wird.

Delayed Assignment: *lhs* := *rhs*
interne Speicherung: SetDelayed[*lhs*, *rhs*] mit dem Attribut HoldAll.

Der Unterschied zum Immediate Assignment besteht in der Behandlung von *rhs*. Diese wird beim Delayed Assignment **nicht** ausgewertet. Sobald im weiteren *lhs* auftritt, wird es durch *rhs* ersetzt. Erst dann wird *rhs* ausgewertet.

In[15]:= f[arg_1, ..., arg_n]:=rhs

Analoges gilt in diesem Fall: Alle Argumente arg_i werden evaluiert. f und f [arg'_1, ..., arg'_n] werden nicht weiter ausgewertet. Außerdem bleibt *rhs* unverändert stehen.

Immediate Assignment		Delayed Assignment
	In[16]:= a:=5	
In[17]:= f[x_] = a x		In[17]:= f[x_] := ax
Out[17]= 5x		keine Ausgabe
f/: f[x_] = 5*x	Musterdatenbank	f/: f[x_] := a*x
	In[18]:= a:=9; f[4]	
5 · 4 →		a · 4 → 9 · 4 →
Out[18]= 20		Out[18]= 36

4 Funktionale Methoden

4.1 Grundlagen und Konzept

Grundidee

- Jedes Programm besteht aus (vielen) verschachtelten Funktionen (z.B. $f(g(h(x^2+5)-\Gamma(x))))$, die jeweils auf das Resultat der vorigen Funktion angewendet werden.
- Eine Folge von Funktionen wird auf eine *einzige aktuelle* Expression, die den momentanen Zustand des Programms beschreibt, angewendet.
- Es gibt keine lokalen Variablen. Jede Information wird sofort an die nächste (übergeordnete) Funktion weitergegeben. Ein grundsätzliches Ziel ist es daher, Variablen mit Namen zu vermeiden. Dadurch kann aber die Übersichtlichkeit der Programme beeinträchtigt werden.
- Der aktuelle Zustand hat (meistens) eine *nicht-triviale* Struktur. (In einer Liste sind die Elemente beispielsweise in einer bestimmten Reihenfolge angeordnet, sodaß jeder Position eine bestimmte Bedeutung zugeordnet werden kann. Vergleiche im Gegensatz dazu eine Menge.)

Funktionales Programmieren ist ein Programmierstil, der die *Evolution von Ausdrücken* gegenüber der Ausführung von Befehlen bevorzugt. Ausdrücke (innerhalb der Progammiersprache) werden durch das Anwenden von Funktionen auf Basisobjekte geformt.

Eine ausführliche Diskussion dieser Thematik kann in der Newsgroup `comp.lang.functional` verfolgt werden.

4.2 Fundamentaloperatoren: `Nest`, `Fold`, `Map`, `Apply`, `Thread`

Nest

`Nest[` f`,` x`,` n `]` $\rightarrow$ f`[` f`[` f`[` $\cdots$ f`[` x `]` $\cdots$ `] ] ]`

- f muß eine Funktion mit genau einem Argument sein.
- x ist der Startwert.

- n ist die Anzahl der Iterationen

In FORTRAN (prozeduraler Stil) könnte man folgendes Konstrukt verwenden:

```
      Res=x
      DO 100 I=1, n
        Res=F(Res)
100   CONTINUE
```

NestList[f,x,n] $\rightarrow$ {x,f[x],f[f[x]],..., f[f[f[$\cdots f$[x]$\cdots$]]]}

NestList[f,x,0] $\rightarrow$ {x}
NestList[f,x,1] $\rightarrow$ {x, f[x]}

`Nest` und `NestList` eignen sich besonders für die Iteration von Funktionen mit nur einem Argument. Diese müssen ihr eigenes Ergebnis wieder als Argument akzeptieren. Grundsätzlich lassen sich alle tail-recursive-Algorithmen mit `Nest` formulieren.

Fehlerhaftes Beispiel

```
In[1]:= Clear[f]

In[2]:= f[{x_,y_}]:=x+y

In[3]:= Nest[f,{1,2},3]
Out[3]= f[f[3]]
```

`f` erwartet eine Liste mit zwei Elementen als Argument, liefert aber eine Zahl als Ergebnis.

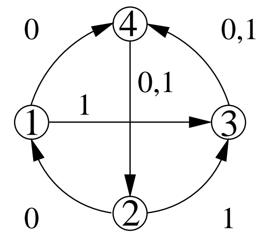
Fold

Fold[f,x,{a,b,...}] $\rightarrow$ f[$\cdots$ f[f[f[x, a], b], c] $\cdots$]

FoldList[f,x,{a,b,...}] $\rightarrow$ {x, f[x,a], f[f[x,a],b],...}

- f muß eine Funktion mit genau zwei Argumenten sein.
- f[x,a] muß als Ergebnis eine expression zurückgeben, die als erstes Argument von f akzeptiert wird.

Transition Diagram



Beispiel

Finite Automata

Moore Automaten kennen

- 2 Input Symbole: „0“ und „1“
- 4 interne Zustände
- keine Output Symbole

Die diskrete Entwicklung des Automaten bei gegebenem Input von einem bestimmten Anfangszustand kann mit folgendem Programm untersucht werden:

```
In[4]:= Automat[1,0]:=4

In[5]:= Automat[1,1]:=3

In[6]:= Automat[2,0]:=1

In[7]:= Automat[2,1]:=3

In[8]:= Automat[3,_]:=4

In[9]:= Automat[4,_]:=2

In[10]:= evolve[aut_,initstate_,input_List]:=
         Fold[aut,initstate,input]

In[11]:= evolve[Automat,1,{0,0,1,1}]
Out[11]= 4
```

Map

`Map[`f`, {`a`,`b`,`c`,...}]` $\rightarrow$ `{`f`[`a`], `f`[`b`], `f`[`c`],...}`

Spezielle Eingabeform: f `/@ {`a`,`b`,`c`,...}`

`Map[`f`, `*expr*`, `*levelspec*`]`

levelspec ist eine Liste, die angibt, auf welche Stufen von *expr* die Funktion f angewendet werden soll. (z.B. `{`n`}` wodurch f nur auf die $n$te Stufe oder `{`n`, `m`}` wodurch f auf die $n$te bis einschließlich $m$te Stufe angewendet wird)

expression	f	[	g	[	h	[···]],	u	[···]]
Stufe	0		1		2	3	1	2

Beispiel

```
In[12]:= Clear[f]

In[13]:= sq[x_]:=x^2

In[14]:= sq /@ Range[5]           → sq /@ {1,2,3,4,5}
                                  → {sq[1], sq[2], sq[3],
                                  sq[4], sq[5]}
Out[14]= {1,4,9,16,25}

In[15]:= f /@ {{1,2},{a,b}}
Out[15]= {f[{1,2}], f[{a,b}]}

In[16]:= Map[f,{{1,2},{a,b}},{2}]
Out[16]= {{f[1],f[2]},{f[a],f[b]}}

In[17]:= Map[f,{{1,2},{a,b}},2]
Out[17]= {f[{f[1],f[2]}],f[{f[a],f[b]}]}
```

Verwandte Operatoren: MapAll, MapAt, MapIndexed, MapThread.

```
In[18]:= MapThread[f,{{a,b,c,...},{1,2,3,...}}]
         f[a,1],f[b,2],f[c,3],...}
```

Apply

`Apply[`f`, `*expr*`]`
Apply f to the head of *expr*. = Ersetze den Kopf von *expr* durch f.

Kurzschreibweise: `f @@ expr`

Beispiel

```
In[19]:= Plus @@ Range[100]          → Plus @@ List[1,2,3,...,100]
Out[19]= 5050                        → Plus[1,2,3,...,100]

In[20]:= And @@ {True, False, True}
Out[20]= False

In[21]:= Or @@ {True, False, True}
Out[21]= True
```

Thread

`Thread[`f`[`arg_1`, `arg_2`,...]]`
Threads f over any lists that appear in arguments arg_1, arg_2, ...

Beispiel

```
In[22]:= Thread[f[{1,2,3}]]          ≡ f /@ {1,2,3}
Out[22]= {f[1], f[2], f[3]}          (wäre ca. 5 % schneller)
```

Wenn mehrere Listen als Argumente übergeben werden, müssen sie dieselbe Länge besitzen.

```
In[23]:= Thread[f[{a,b,c},{1,2,3}]]
Out[23]= {f[a,1], f[b,2], f[c,3]}

In[24]:= Thread[f[{A,B,C},{a,b,c},{1,2,3}]]
Out[24]= {f[A,a,1], f[B,b,2], f[C,c,3]}

In[25]:= Thread[f[a,{1,2,3},A]]
Out[25]= {f[a,1,A], f[a,2,A], f[a,3,A]}
```

`Thread` ist somit in gewisser Weise „verwandt“ mit dem Attribut `Listable`. Wenn f dieses Attribut besitzt wird `Thread` sozusagen automatisch ausgeführt.

4.3 *Pure Functions*

Pure Functions sind Funktionen, die keinen eigenen Namen besitzen. Ihre Definition ist ihre einzige Bezeichnung. In der formalen Logik und in LISP bezeichnet man sie als λ-Ausdrücke oder anonyme Funktionen. Wenn eine Funktion nur an einer einzigen bestimmten Stelle gebraucht wird, erspart man sich dadurch eine separate Funktionsdefinition und das Suchen eines geeigneten Namens.

Beispiel

```
In[1]:= sq[x_]:=x^2
```
als *Pure Function*: `(#^2)&`

```
In[2]:= sq /@ {a,b}
Out[2]= {a^2, b^2}

In[3]:= (#^2)& /@ {a,b}
Out[3]= {a^2, b^2}
```

Bei funktionalen Operatoren (`Map`, `Apply`, `Fold`, ...) muß immer eine Funktion angeben werden:

```
In[4]:= Nest[f, x, 3]
Out[4]= f[f[f[0]]]
```

Ist die Funktion in der Form `f[x_]:=`*defintionbody* definiert, so ensteht ein Defintions-Overhead, sowohl syntaktisch als auch semantisch.

Pointe: Die *Definition* einer Pure Function ist gleichzeitig ihr *Name*.

- `Function[`x`, `*body*`]` ... mit nur einem Argument
- *body* `&` (`#` für das Argument)
- `Function[{`x_1`, `x_2`, `x_3`, ..., `x_n` }, `*body*` ]`
- *body* `&` (`#1`, `#2`, `#n` für das erste, zweite, $n$te Argument)

Anwendung:

- `Function[`x`, `*body*`][`*arg*`]`
- `Function [{`x_1`, `x_2`, `x_3`, ..., `x_n` }, `*body*` ] [`arg_1`, ..., `arg_n`]`
- *body* `& [`arg_1`, ..., `arg_n`]`

Evaluierung: An jeder Stelle in *body*, an der x_i vorkommt, wird der Wert von arg_i buchstäblich ersetzt („replaced"), es erfolgt keine vorherige Evaluierung von *body* und x_i.

`Function[`x`, `*body*`][`*arg*`]` $\equiv$ *body* `/. `x` -> `*arg* (λ-Conversion)
body und x werden in letzterem Fall nicht evaluiert, sondern bleiben bestehen, d.h. der eigentliche *Mathematica*-Ausdruck lautet:

`(Hold[`*body*`] /. HoldPattern[`x`] -> `*arg*`) // ReleaseHold`

`Function[`x`, `*body*`]` $\equiv$ `Function [`y`, `*body*`]` (α-Conversion)
`Function[`x`, `h`[`x`]]` $\equiv$ h`[`x`]]` (η-Conversion)

Beispiel

```
In[5]:= Clear[x]

In[6]:= Function[x, x/2][4]
Out[6]= 2

In[7]:= Function[x, x/2][4, 6]
Out[7]= 2

In[8]:= Function[x, x/2][x]
Out[8]= x/2

In[9]:= x=2; Function[x, x/2][x]
Out[9]= 1

In[10]:= #/2 & [4]
Out[10]= 2

In[11]:= #/2 & 6
Out[11]= 6 (#1/2 &)
```

Eine Funktion, die eine Funktion zurückliefert:

```
In[12]:= pow[n_Integer] := Function[x, x^n]

In[13]:= pow[2]
Out[13]= Function[x$, x$^2]

In[14]:= % [4]
Out[14]= 16

In[15]:= pow[3][3]
Out[15]= 27
```

Redundanz:

```
In[16]:= f[#]& /@ {a, b, c}          kürzer: f /@ {a, b, c}
Out[16]= {f[a], f[b], f[c]}
```

4.4 *Advanced Operators*: Inner, Outer

Inner

```
In[1]:= Inner[f, {a1, b1,..., n1 }, {a2, b2,..., n2},g]
Out[1]= g[ f[a1, a2], f[b1, b2],..., f[n1, n2]]
```

Euklidsches Skalarprodukt:

```
In[2]:= Inner[Times, {a1,a2,a3},     kürzer:
        {b1,b2,b3}, Plus]            Dot[{a1,a2,a3}, {b1,b2,b3}]
                                     oder {a1,a2,a3}.{b1,b2,b3}
Out[2]= a1 b1 + a2 b2 + a3 b3
```

Die allgemeine Schreibweise lautet `Inner[`f`, `k`_List, `l`_List, `g`]`, wobei die Funktion f an Stelle der Multiplikation und g an Stelle der Addition ausgeführt wird.

Eine äquivalente Formulierung ist g `@@ Thread [`f` [`k`, `l`]]`, deren Ausführungsgeschwindigkeit jedoch ca. 10 % langsamer ist. k und l können Tensoren beliebiger Stufe sein, wobei `Inner` den letzten Index des ersten Tensors mit dem ersten des zweiten Tensors überschiebt.

Outer

`Outer[f, `$List_1$`, `$List_2$`,...]`
berechnet das verallgemeinerte äußere Produkt mehrerer Tensoren $List_i$ unter Verwendung der Funktion f.
„alle möglichen Kombinationen von Elemeten der Listen“

```
In[3]:= Outer[f,{a, b, c}]           ≡ f /@ {a, b, c}
Out[3]= {f[a], f[b], f[c]}
In[4]:= Outer[f, {a,b}, {1, 2,       ≡ Thread[ f[#, {1, 2,
        3}]                          3}] ] & /@ {a,b}
Out[4]= {{f[a,1], f[a,2],
        f[a,3]}, {f[b,1], f[b,2],
        f[b,3]}}
In[5]:= Outer[f,{A,B},{1,2,3},{a,b}]
Out[5]= {{{f[A, 1, a], f[A, 1, b]},
          {f[A, 2, a], f[A, 2, b]},
          {f[A, 3, a], f[A, 3, b]}},
         {{f[B, 1, a], f[B, 1, b]},
          {f[B, 2, a], f[B, 2, b]},
          {f[B, 3, a], f[B, 3, b]}}}
```

5 Ein realistisches Beispiel: *1-d Forest Fire Simulation*

5.1 Problembeschreibung: Wachsen/Anzünden – Brennen – Absterben

Bäume

Jeder Baum kennt 3 verschiedene Zustände.

- 0 ... kein Baum
- 1 ... ein Baum
- 2 ... ein brennender Baum

Natur

- Sie läßt mit einer Wahrscheinlichkeit p Bäume wachsen.
- Sie läßt mit einer Wahrscheinlichkeit f Bäume spontan Feuer fangen.

Feuer

- Ein brennender Baum zündet beide Nachbarbäume an.
- Das Feuer breitet sich im Vergleich zum Wachstum der Bäume rasend schnell aus.
- Brennende Bäume sterben immer ab.

Physik

Da das simulierte System nur endlich sein kann, treten Randeffekte auf. Diese können zwar durch zunehmende Größe des Systems reduziert werden, die Berechnungen werden jedoch aufwendiger und langsamer.

Eine andere Möglichkeit, diese unerwünschten Effekte zu vermeiden, stellen periodische (toroidale) Randbedingungen dar.

5.2 Implementation der einzelnen Phasen

Datenrepräsentation

1-dimensionaler Wald: `{0, 1,...}`

Erzeugen

```
In[1]:= n:=12

In[2]:= forestList=Table[Random[Integer],{n}]
Out[2]= {1,0,1,1,1,0,1,1,1,0,0,1}
```

Wachsen/Anzünden

```
In[3]:= treeGrowIgnite[0]:=Floor[1+p-Random[]]

In[4]:= treeGrowIgnite[1]:=Floor[2+f-Random[]]

In[5]:= f:=0.3; p:=0.75

In[6]:= forestList=treeGrowIgnite
        /@ forestList
Out[6]= {1,0,2,1,1,0,1,2,1,0,0,2}
```

Wald mit jungen Bäumen, der „ein bißchen“ brennt.

Ausbreitung des Feuers

Steppe	=	{0,0,0,0,0,0,0,0,0,0,0,0}
Wald mit jungen Bäumen	=	{1,0,1,1,1,0,1,1,1,0,0,1}
Anzünden	→	{1,0,2,1,1,0,1,2,1,0,0,2}
Brennen	→	{2,0,2,2,1,0,2,2,2,0,0,2}
Brennen	→	{2,0,2,2,2,0,2,2,2,0,0,2}
Absterben	→	{0,0,0,0,0,0,0,0,0,0,0,0}

Idee:
`{...2,1...}` → `{...2,2...}`
`{...1,2...}` → `{...2,2...}`

```
In[7]:= forestBurn[{a___,2,1,b___}]:=forestBurn[{a,2,2,b}]
```

```
In[8]:= forestBurn[{a___,1,2,b___}]:=forestBurn[{a,2,2,b}]
```

```
In[9]:= forestBurn[{1,a___,2}]:=forestBurn[{2,a,2}]
```

```
In[10]:= forestBurn[{2,a___,1}]:=forestBurn[{2,a,2}]
```

Eine alternative Formulierung mit Transformation Rules:
`forestList /. {a___,2,1,b___} -> {a,2,2,b}`
Mit `/.` würde jedoch nur eine Ersetzung vorgenommen. Daher muß man `ReplaceRepeated` = `//.` verwenden:
`forestList //. {a___,2,1,b___} -> {a,2,2,b}`

Ende des Feuers

`{2,0,2,2,2,0,2,2,2,0,0,2}` → `{0,0,0,0,0,0,0,0,0,0,0,0}`

```
In[11]:= forestBurn[forestList]
Out[11]= forestBurn[{2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 0, 2}]
```

```
In[12]:= %[[1]] /. 2->0
Out[12]= {0,0,0,0,0,0,0,0,0,0,0,0}
```

Neuer Durchgang

Bäume wachsen von neuem und entzünden sich:

```
In[13]:= treeGrowIgnite /@ %
Out[13]= {1,0,1,0,1,1,1,1,1,1,0,1}
```

Algorithmus

- Anfangswald erzeugen: `forestList=Table[Random[Integer],{n}]`
- Wachsen/Anzünden: `treeGrowIgnite /@ forestList`
- Ausbreitung des Feuers: `forestBurn[%]`
- Absterben der brennenden Bäume: `%[[1]] /. {2->0}`

Mehrere Zyklen

Um einen kontinuierlichen Kreislauf zu erzeugen, muß ein Anfangszustand erzeugt werden, der Wald wachsen und sich entzünden, das Feuer sich ausbreiten und die brennenden Bäume schließlich absterben. Dann kann der Wald neuerlich wachsen und sich entzünden, ...

`Nest` bietet sich an, um eine geeignete Funktion, die das Wachsen, Entzünden, Ausbreiten des Feuers und Absterben der Bäume herbeiführt, immer wieder auf den Wald anzuwenden (zu iterieren).

Die zu iterierende Funktion:

```
In[14]:= doForestCycle[l_List]:= forestBurn[treeGrowIgnite /@
         l][[1]] /. 2->0
```

Der Kreislauf schließt sich:

```
In[15]:= woodlife[n_Integer, m_Integer]:= Nest[doForestCycle,
         Table[Random[Integer],{n}],m]
```

Ästhetische Verbesserungen

`forestBurn` verbleibt als Kopf nach dem Anzünden der Nachbarbäume. Durch eine zusätzliche Regel läßt sich das verhindern.

```
In[16]:= forestBurn[l_List]:=l /. 2->0
```

Die Reihenfolge der Regeln von `forestBurn` ist hier jedoch entscheidend! Diese kann einfach kontrolliert werden:

```
In[17]:= ?forestBurn
Out[17]= Global`forestBurn
         forestBurn[ {a___, 2, 1, b___} ] := forestBurn[{a,
         2, 2, b}]
         forestBurn[ {a___, 1, 2, b___} ] := forestBurn[{a,
         2, 2, b}]
         forestBurn[{1, a___, 2}] := forestBurn[{2, a, 2}]
         forestBurn[{2, a___, 1}] := forestBurn[{2, a, 2}]
         forestBurn[l_List] := l /. 2 -> 0
```

`doForestCycle` läßt sich jetzt vereinfachen:

```
In[18]:= doForestCycle[l_List]:=
         forestBurn[treeGrowIgnite /@ l]
```

Eliminieren des Maps `/@` in `doForestCycle`:

```
In[19]:= SetAttributes[treeGrowIgnite, Listable]
```

Beispiel

```
In[20]:= Clear[x]

In[21]:= SetAttributes[x, Listable]

In[22]:= x[{a,b,c}]
Out[22]= {x[a],x[b],x[c]}
```

langsamer als `Map`: `x /@ {a,b,c}`

aber:

```
In[23]:= x[{a,{b,c}}]
Out[23]= {x[a],{x[b],x[c]}}
```

Ergebnis:

```
In[24]:= doForestCycle[l_List]:=forestBurn[treeGrowIgnite[l]]
```

„One-Liner"

```
In[25]:= woodLife[n_Integer, m_Integer]:=
            Nest[
              ( # /. {0:>Floor[1+p-Random[]],
                      1:>Floor[2+f-Random[]]}
                  //. {{a___,1,2,b___} :> {a,2,2,b},
                      {a___, 2, 1, b___} :> {a, 2, 2, b},
                      {1, a___, 2} :> {2, a, 2},
                      {2, a___, 1} :> {2, a, 2}}
                  /. 2 -> 0
              ) &,
              Table[Random[Integer],{n}],
              m
            ]
Out[25]= General::spell1:  Possible spelling error:  new
         symbol name "woodLife" is similar to existing symbol
         "woodlife".
```

:> kennzeichnet eine „Delayed Rule", die nicht bei der Definition, sondern erst bei der Auswertung des Ausdrucks angewendet wird.

6 Objektorientierte Methoden

6.1 Konzepte

Konzept	Beispiel	unterstützt
Objekte	`{a,A,1}, Automat[...],...`	ja
Vererbung		nein
encapsulation, information hiding	`$Context, Packages, Module[], Block[]`	teilweise
operator overloading	`4.3 + 7.9` UpValues (`^:=`) `Op[]+Op[]`	teilweise
Klassen (abstract types)	keine Typen in *Mathematica* (umgehbar)	nein

6.2 Objekte

Es gibt keine Datentypen („everything is an expression“).

`Head[.]` fungiert als „Pseudotyp“.

```
In[1]:= Head[5]
Out[1]= Integer

In[2]:= Head[4.2]
Out[2]= Real

In[3]:= Head[{A,1}]
Out[3]= List

In[4]:= Head[Pi]
Out[4]= Symbol
```

Objekte existieren auch innerhalb von *Mathematica*:
`{...}` oder `Graphics[...]`

Ein Objekt ist

- ein Container,
- statisch,
- produziert nichts (im Gegensatz zu: `Plot[]`, `Plus[]`).

Warum dann überhaupt Objekte?

Man definiert Methoden, die

- den inneren Zustand des Objekts ändern,
- Objekte kombinieren,
- Objekte transformieren,
- die Eigenschaften des Objekts extrahieren.

Beispiel

Lineare Operatoren

Hilbertraum $\mathbb{R}^3$ ($\mathbb{C}^3$),
Orthonormalbasis $\{|a_1\rangle, |a_2\rangle, |a_3\rangle\}$,
$\langle a_i|a_j\rangle = \delta_{ij}$
Operatoren über diesem Hilbertraum
(Operatoren = lineare Abbildungen $\mathbb{R}^3 \to \mathbb{R}^3$)

Operator A:
$A|a_1\rangle \mapsto |a_2\rangle$
$A|a_2\rangle \mapsto |a_1\rangle + |a_3\rangle$
$A|a_3\rangle \mapsto |a_2\rangle$

```
In[5]:= Op[{a1, a2, a3}, {a2, a1+a3, a2}]
Out[5]= Op[{a1, a2, a3}, {a2, a1+a3, a2}]
```

`Op[]` alleine ist sinnlos, daher sind „sinnvolle Methoden“ gesucht.

Matrixdarstellung des Operators

```
In[6]:= ToMatrix[op:Op[basic_List,image_List]] :=
        Outer[project, basis,image]

In[7]:= project[a_, b_ + c__] := project[a,b] + project[a,c]

In[8]:= project[a_, n_ a_ ] := n
```

($\langle a|\lambda a\rangle = \lambda\langle a|a\rangle = \lambda$)

```
In[9]:= project[a_, a_] := 1

In[10]:= project[a_, b_] := 0
```

$\langle b + c|a\rangle$ funktioniert noch nicht.

```
In[11]:= project[b_ +c_, a_]
         := project[b,a] +
         project[c,a]
```

oder in reellen Hilberträumen (keine komplexe Konjugation notwendig) `SetAttributes[project, Orderless]`

```
In[12]:= Outer [project, {a1, a2,
         a3}, {a2, a1+a3, a2}]
Out[12]= {{0, 1, 0}, {1, 0, 1},
         {0, 1, 0}}
```

Ist der Operator hermitesch ($A = A^T$)?

```
In[13]:= HermiteQ[ Op[ b_List, i_List] ] :=
         Module[ m=ToMatrix[ Op[b, i] ],
         Conjugate[Transpose[m]] == m ]
```

Komplizierte Ausdrücke müssen unter Umständen noch „händisch“ vereinfacht werden (`Expand`, `Simplify`).

6.3 *Operator Overloading*

Beispiel

Lineare Operatoren bilden selbst einen Vektorraum :

$\mathbb{1}$, 0, $A + B \mapsto C$, $\lambda A \mapsto (\lambda A)$, $A + 0 = A$, $(A + B)|x\rangle = A|x\rangle + B|x\rangle$

```
In[1]:= Op[ b_, i_] + Op[b_, j_] := Op[b, i+j]
Out[1]= SetDelayed::write:  Tag Plus in Op[b_, i_] + Op[b_,
        j_] is Protected.
        $Failed
```

Dieser Fehler tritt auf, da die Definition des Operators beim Head (`Plus`) gespeichert wird, „+“ jedoch ein geschütztes Symbol ist.

Lösung:

```
In[2]:= Unprotect[Plus]
Out[2]= { Plus }

In[3]:= Op[b_, i_] + Op[b_, j_ ] := Op[b, i+j]

In[4]:= Protect[Plus]
Out[4]= { Plus }
```

Diese Lösung ist zwar korrekt aber langsam, da bei jedem „+“ in der Datenbank nachgesehen werden muß, wie der Operator zu interpretieren ist.

```
In[5]:= ??Plus
Out[5]= x + y + z represents a sum of terms.
        Attributes[Plus] = {Flat, Listable, OneIdentity,
        Orderless, Protected}
        Op[b_, i_] + Op[b_, j_] := Op[b, i + j]
        Default[Plus] := 0
```

Implementierbare Lösung : UpValues (im Gegensatz zu DownValues, „`:=`“)

DownValue:

```
In[6]:= Dimension [Op [b_, _]] := Length [b]
```

```
In[7]:= ?Dimension
Out[7]= Global`Dimension
        Dimension[Op[b_, _]] := Length[b]
```

UpValue:

```
In[8]:= Clear[Dimension]
```

```
In[9]:= Dimension[Op[b_,_]]^:= Length [b]
```

```
In[10]:= ?Dimension
Out[10]= Global`Dimension
```

```
In[11]:= ?Op
Out[11]= Global`Op
         Dimension[Op[b_, _]] ^:=
         Length[b]
```

Die Definition ist mit `Op` assoziiert (und nicht mit `Dimension`).

Beispiel

```
In[12]:= Op[b_, i_] + Op[b_, j_] ^:= Op [b, i+j ]
```

```
In[13]:= Lambda Op [b_, i_] ^:= Op[b, Lambda i]
```

```
In[14]:= EigenValues[op:Op[b_, i_]] ^:=
         Eigenvalues[ToMatrix[op]]
```

Nachteil: `UpValues[]` geht nur einen Level tief.

$e^{A+B} = e^A \cdot e^B$ `/;` $[A,B]$`==0`

```
In[15]:= Exp[ Op[]+Op[] ]^:= Exp[Op[]] Exp[Op[]] /;
         Inner[Times,A,B,Plus]==0
Out[15]= SetDelayed::write:  Tag Times in Exp[2 Op[]] is
         Protected.
```

Evaluation

Reihenfolge von Up/DownValues

```
In[16]:= Clear[f]; Clear[g]; Clear[h]
```

`In[17]:= g[ f[x_] ] ^:= gfup[x]`	UpValue von `f`
`In[18]:= g[ f[x_] ] := gfdown[x]`	DownValue von `g`
`In[19]:= f[ g[x_] ] := fg[x]`	DownValue von `f`

```
In[20]:= f[g[f[1]]]
Out[20]= f[gfup[1]]

In[21]:= f[g[h[1]]]
Out[21]= fg[h[1]]

In[22]:= g[f[1]]
Out[22]= gfup[1]

In[23]:= g[f[g[1]]]
Out[23]= g[fg[1]]
```

Der DownValue von `g` wurde somit nie verwendet.

Sinnvolle Kombination:

`In[24]:= g[f[x_]] ^:= gfup[x]`	UpValue von `f` für Spezialfälle
`In[25]:= g[h_[x_]] := gdown[h, x]`	DownValue von `g` für den allgemeinen Fall

6.4 `Packages` und `Context`

Wo speichert *Mathematica*Transformation rules ?

Mechanismen

Sie bestimmen, wo etwas abgespeichert und gesucht wird.

- `$Context` - current context (aktuelle Suchumgebung)
  `$ContextPath` - list of contexts to be searched z.B. `{Global`, System`}`
- Packages - Mechanismen zum Verändern von `$Context` und `$ContextPath`

Suchalgorithmus für Symbole

Wenn *Mathematica* einem Symbol begegnet, geht es wie folgt vor:

1. Ist das Symbol im current context `$Context` definiert? Bei ja nimmt *Mathematica* die gespeicherte Regel, sonst...
2. *Mathematica* durchsucht alle Kontexte im `$ContextPath`. Bei ja nimmt *Mathematica* die gespeicherte Information, sonst...
3. *Mathematica* erzeugt das Symbol im current context `$Context`

Beispiel

```
In[1]:= $ContextPath
Out[1]= {Global`, System`}

In[2]:= $Context
Out[2]= Global`

In[3]:= h[x_]:= 2x
```

Ist `h` in `$Context` ? Nein
Ist `h` im `$ContextPath` ? Nein
Dann erzeuge Symbol („Tag“ in der Datenbank) im `$Context = Global``.

```
In[4]:= ?h
Out[4]= Global`h
        h[x_]:=2*x
```

Wie verändert man `$Context` und `$ContextPath`?

- `BeginPackage["`*context*`` `"]`
  Die alten Werte von `$Context` und `$ContextPath` werden gesichert.
  `$context` wird *context* gesetzt.
  `$ContextPath` wird `{`*context*`` `, System`}`` gesetzt.
  Achtung: kein `Global`` in `$ContextPath`
- `End Package[]`
  Die ursprünglichen Werte von `$Context` und `$ContextPath` werden wiederhergestellt.
  `$ContextPath` ← `{`*context*`` `}`` ∪ „gemerkter“ `$ContextPath`

- `Begin["`*context*`"]`
  Mathematica merkt sich den aktuellen `$Context` und wechselt in den neuen (absoluten) *context*.
  Anmerkung: relativer *context* mit `Begin["`*context*`"]`, d.h. der neue `$Context` lautet *oldContext`context`*

- `End[]`
  Der ursprüngliche Wert von `$Context` vor `Begin[]` wird wiederhergestellt.
  Anmerkung: `$ContextPath` wird durch `Begin[]` und `End[]` nie geändert.

Befehl	Symbolname	$Context	$ContextPath
Prime[10^6]	System`Prime	Global`	{Global`,System`}
<<LinOp.m			
BeginPackage ["LinOp`"]	System`BeginPackage	LinOp`	{LinOp`, System`}
ToMatrix ::usage:...	LinOp`ToMatrix	LinOp`	{LinOp`, System`}
Begin["`Private`"]		LinOp`Private`	{LinOp`, System`}
project[a_,n_,a_]:=n	LinOp`Private`project	LinOp`Private`	{LinOp`, System`}
End[]		LinOp`	{LinOp`, System`}
EndPackage[]		Global`	{LinOp`, Global`, System`}
ToMatrix[Op[{a}, {4a}]	LinOp`ToMatrix		

Probleme

Wenn das Symbol aus einem Package schon vorher definiert wurde.

```
In[5]:= ToMatrix[l_List]:=l

In[6]:= ?ToMatrix
Out[6]= Global`ToMatrix
        ToMatrix[op:Op[basic_List, image_List]] :=
        Outer[project, basis, image] ToMatrix[l_List] :=
        l

In[7]:= <<LinOp.m
Out[7]= ToMatrix::shdw:  Warning:  Symbol ToMatrix appears
        in multiple contexts {LinOp`, Global`}; definitions
        in context LinOp` may shadow or be shadowed by other
        definitions.
```

Da nach `EndPackage[]` `$Context` wieder auf `Global`` gesetzt wird, wird beim nächsten Aufruf von `ToMatrix` `Global`ToMatrix` verwendet.

Ein Package benötigt ein anderes Package

- `BeginPackage["context`", {`*needed*$_1$, *needed*$_2$`,...}]`

  Beispiel

  `QM.m` benötigt `LinOp``

  `$ContextPath = {Global`, System`}`
  `BeginPackage["QM`", "LinOp`"]`
  falls `LinOp`` bereits geladen wurde: no action
  falls nicht: `<<LinOp.m` einlesen
  `$ContextPath = {QM`, LinOp`, System`}`

- *Hidden Import*
  User soll auf dessen Funktion nicht zugreifen können
  `Needs["context`"]` (nach `BeginPackage[]`)

Grundgerüst für Packages

```
BeginPackage["Package`"]
Package::usage = "This package implements..."
f::usage = "f[list]..." (* erzeugt erst Package`f *)
Begin["`Private`"]
f[...]:= ...
End[]           (* Package`Private *)
EndPackage[]    (* Package *)
```

7 GNU Free Documentation License

GNU Free Documentation License
Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of

the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the

Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the

substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single

copy. If there are multiple Invariant Sections with the same name but
different contents, make the title of each such section unique by
adding at the end of it, in parentheses, the name of the original
author or publisher of that section if known, or else a unique number.
Make the same adjustment to the section titles in the list of
Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History"
in the various original documents, forming one section entitled
"History"; likewise combine any sections entitled "Acknowledgements",
and any sections entitled "Dedications". You must delete all sections
entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents
released under this License, and replace the individual copies of this
License in the various documents with a single copy that is included in
the collection, provided that you follow the rules of this License for
verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute
it individually under this License, provided you insert a copy of this
License into the extracted document, and follow this License in all
other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate
and independent documents or works, in or on a volume of a storage or
distribution medium, does not as a whole count as a Modified Version
of the Document, provided no compilation copyright is claimed for the
compilation. Such a compilation is called an "aggregate", and this
License does not apply to the other self-contained works thus compiled
with the Document, on account of their being thus compiled, if they
are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these
copies of the Document, then if the Document is less than one quarter
of the entire aggregate, the Document's Cover Texts may be placed on
covers that surround only the Document within the aggregate.
Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
```

```
      under the terms of the GNU Free Documentation License, Version 1.1
      or any later version published by the Free Software Foundation;
      with the Invariant Sections being LIST THEIR TITLES, with the
      Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
      A copy of the license is included in the section entitled "GNU
      Free Documentation License".
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.