

# Uebungen ■ Exercices

## 15. Packages

Die Gliederung dieses Kurses folgt in groben Zügen dem Buch von Nancy Blachman: *A Practical Approach...* Hinweis: Kapitel 15 lesen!

- L'articulation de ce cours correspond à peu près à celle du livre de Nancy Blachman: *A Practical Approach...*  
Indication: Lire le chapitre 15.

Run mit WIN+*Mathematica* Version 5.2

- Testé avec *Mathematica* version 5.2+WIN

WIR94/98/99/2000/2007 // Copyright Rolf Wirz

### Aufgabe 1 ■ Problème 1

Lade das Package "Colors.m" aus "Graphics". Liste alle Farben auf.

- Charger le Package "Colors.m" de "Graphics". Fais une liste des couleurs.

```
In[1]:= Needs["Graphics`Colors`"]
```

```
In[2]:= ?Graphics`Colors`*
```

#### Graphics`Colors`

AliceBlue	Goldenrod	PaleGreen
AlizarinCrimson	GoldenrodDark	PaleTurquoise
AllColors	GoldenrodLight	PaleVioletRed
Antique	GoldenrodPale	PapayaWhip
Apricot	GoldOchre	Peach
Aquamarine	GreenDark	PeachPuff
AquamarineMedium	GreenishUmber	Peacock
AureolineYellow	GreenPale	PermanentGreen
Azure	GreenYellow	PermanentRedViolet
Banana	HLSColor	Peru
Beige	Honeydew	PinkLight
Bisque	HotPink	Plum
BlanchedAlmond	HSBColor	PowderBlue
BlueLight	IndianRed	PrussianBlue
BlueMedium	Indigo	PurpleMedium
BlueViolet	Ivory	Raspberry
Brick	IvorvBlack	RawSienna

Brick	IvoryBlack	RawSienna
BrownMadder	Khaki	RawUmber
BrownOchre	KhakiDark	RoseMadder
Burlywood	LampBlack	RosyBrown
BurntSienna	Lavender	RoyalBlue
BurntUmber	LavenderBlush	SaddleBrown
Cadet	LawnGreen	Salmon
CadetBlue	LemonChiffon	SandyBrown
CadmiumLemon	LightBeige	SapGreen
CadmiumOrange	LightBlue	SeaGreen
CadmiumRedDeep	LightCadmiumRed	SeaGreenDark
CadmiumRedLight	LightCadmiumYellow	SeaGreenLight
CadmiumYellow	LightCoral	SeaGreenMedium
CadmiumYellowLight	LightGoldenrod	Seashell
Carrot	LightGray	Sepia
Cerulean	LightPink	Sienna
Chartreuse	LightSalmon	SkyBlue
Chocolate	LightSeaGreen	SkyBlueDeep
ChromeOxideGreen	LightSkyBlue	SkyBlueLight
CinnabarGreen	LightSlateBlue	SlateBlue
CMYColor	LightSlateGray	SlateBlueDark
Cobalt	LightSteelBlue	SlateBlueLight
CobaltGreen	LightViridian	SlateBlueMedium
CobaltVioletDeep	LightYellow	SlateGray
ColdGray	LimeGreen	SlateGrayDark
Coral	Linen	SlateGrayLight
CoralLight	MadderLakeDeep	Smoke
CornflowerBlue	ManganeseBlue	Snow
Cornsilk	Maroon	SpringGreen
CyanWhite	MarsOrange	SpringGreenMedium
DarkGoldenrod	MarsYellow	SteelBlue
DarkGreen	MediumAquamarine	SteelBlueLight
DarkKhaki	MediumBlue	TerreVerte
DarkOliveGreen	MediumOrchid	Thistle
DarkOrange	MediumPurple	Titanium
DarkOrchid	MediumSeaGreen	Tomato
DarkSeaGreen	MediumSlateBlue	Turquoise
DarkSlateBlue	MediumSpringGreen	TurquoiseBlue
DarkSlateGray	MediumTurquoise	TurquoiseDark
DarkTurquoise	MediumVioletRed	TurquoiseMedium
DarkViolet	Melon	TurquoisePale
DeepCadmiumRed	MidnightBlue	Ultramarine
DeepCobaltViolet	Mint	UltramarineViolet
DeepMadderLake	MintCream	VanDykeBrown
DeepNaplesYellow	MistyRose	VenetianRed
DeepOchre	Moccasin	Violet
DeepPink	NaplesYellowDeep	VioletDark
DeepSkyBlue	Navajo	VioletRed
DimGray	Navy	VioletRedMedium
DodgerBlue	NavyBlue	VioletRedPale
Eggshell	Oak	ViridianLight
EmeraldGreen	OldLace	WarmGray
EnglishRed	Olive	Wheat
Firebrick	OliveDrab	YellowBrown
Floral	OliveGreenDark	YellowGreen
ForestGreen	OrangeRed	YellowLight
Gainsboro	Orchid	YellowOchre
GeraniumLake	OrchidDark	YIQColor

Ghost	OrchidMedium	Zinc
Gold	PaleGoldenrod	

## Aufgabe 2 ■ Problème 2

Schreibe ein *Mathematica*-Package, das den Mittelwert (mean) und den Median einer Liste berechnet. In der folgenden Lösung wird die Standard-Lösung aus den mitgelieferten Packages kopiert und präsentiert.

■ Ecris un Package de *Mathematica*, qui calcule la valeur moyenne et la médiane d'une liste. Dans la solution suivante, la solution standard est copiée et présentée dans les Packages livrées.

Standard-Package einlesen:

■ Lire le Package standard:

```
In[3]:= Needs["DiscreteMath`Combinatorica`"]
```

```
In[4]:= ?DiscreteMath`Combinatorica`*
```

### DiscreteMath`Combinatorica`

AcyclicQ	MinimumChangePermutations
AddEdge	MinimumSpanningTree
AddEdges	MinimumVertexColoring
AddVertex	MinimumVertexCover
AddVertices	MultipleEdgesQ
Algorithm	MultiplicationTable
AllPairsShortestPath	MycielskiGraph
AlternatingGroup	NecklacePolynomial
AlternatingGroupIndex	Neighborhood
AlternatingPaths	NetworkFlow
AnimateGraph	NetworkFlowEdges
AntiSymmetricQ	NextBinarySubset
Approximate	NextComposition
ApproximateVertexCover	NextGrayCodeSubset
ArticulationVertices	NextKSubset
Automorphisms	NextLexicographicSubset
Backtrack	NextPartition
BellB	NextPermutation
BellmanFord	NextSubset
BiconnectedComponents	NextTableau
BiconnectedQ	NoMultipleEdges
BinarySearch	NonLineGraphs
BinarySubsets	NoPerfectMatchingGraph
BipartiteMatching	NormalDashed
BipartiteMatchingAndCover	NormalizeVertices
BipartiteQ	NoSelfLoops
BooleanAlgebra	NthPair
BreadthFirstTraversal	NthPermutation
Brelaz	NthSubset
BrelazColoring	NumberOf2Paths
Bridges	NumberOfCompositions
ButterflyGraph	NumberOfDerangements
CageGraph	NumberOfDirectedGraphs
CartesianProduct	NumberOfGraphs
ChangeEdges	NumberOfInvolutions

ChangeEdges	NumberOfInvolutions
ChangeVertices	NumberOfKPaths
ChromaticNumber	NumberOfNecklaces
ChromaticPolynomial	NumberOfPartitions
ChvatalGraph	NumberOfPermutationsByCycles
CirculantGraph	NumberOfPermutationsByInversions
CircularEmbedding	NumberOfPermutationsByType
CircularVertices	NumberOfSpanningTrees
CliqueQ	NumberOfTableaux
CoarserSetPartitionQ	OctahedralGraph
CodeToLabeledTree	OddGraph
Cofactor	One
CompleteBinaryTree	Optimum
CompleteGraph	OrbitInventory
CompleteKaryTree	OrbitRepresentatives
CompleteKPartiteGraph	Orbits
CompleteQ	Ordered
Compositions	OrientGraph
ConnectedComponents	OutDegree
ConnectedQ	PairGroup
ConstructTableau	PairGroupIndex
Contract	Parent
CostOfPath	ParentsToPaths
CoxeterGraph	PartialOrderQ
CubeConnectedCycle	PartitionLattice
CubicalGraph	PartitionQ
Cut	Partitions
Cycle	PathConditionGraph
CycleIndex	PerfectQ
Cycles	PermutationGraph
CycleStructure	PermutationGroupQ
Cyclic	PermutationQ
CyclicGroup	PermutationToTableaux
CyclicGroupIndex	PermutationType
DeBruijnGraph	PermutationWithCycle
DeBruijnSequence	Permute
Degrees	PermuteSubgraph
DegreeSequence	PetersenGraph
DegreesOf2Neighborhood	PlanarQ
DeleteCycle	PointsAndLines
DeleteEdge	Polya
DeleteEdges	PseudographQ
DeleteFromTableau	RadialEmbedding
DeleteVertex	Radius
DeleteVertices	RandomComposition
DepthFirstTraversal	RandomGraph
DerangementQ	RandomHeap
Derangements	RandomInteger
Diameter	RandomKSetPartition
Dihedral	RandomKSubset
DihedralGroup	RandomPartition
DihedralGroupIndex	RandomPermutation
Dijkstra	RandomPermutation1
DilateVertices	RandomPermutation2
Directed	RandomRGF
Distances	RandomSetPartition
DistinctPermutations	RandomSubset
Distribution	RandomTableau
DodecahedralGraph	RandomTree

DodecahedralGraph	RandomTree
DominatingIntegerPartitionQ	RandomVertices
DominationLattice	RankBinarySubset
DurfeeSquare	RankedEmbedding
Eccentricity	RankGraph
Edge	RankGrayCodeSubset
EdgeChromaticNumber	RankKSetPartition
EdgeColor	RankKSubset
EdgeColoring	RankPermutation
EdgeConnectivity	RankRGF
EdgeDirection	RankSetPartition
EdgeLabel	RankSubset
EdgeLabelColor	ReadGraph
EdgeLabelPosition	RealizeDegreeSequence
Edges	ReflexiveQ
EdgeStyle	RegularGraph
EdgeWeight	RegularQ
EmptyGraph	RemoveMultipleEdges
EmptyQ	RemoveSelfLoops
EncroachingListSet	ResidualFlowGraph
EquivalenceClasses	RevealCycles
EquivalenceRelationQ	ReverseEdges
Equivalences	RGFQ
Euclidean	RGFs
Eulerian	RGFToSetPartition
EulerianCycle	RobertsonGraph
EulerianQ	RootedEmbedding
ExactRandomGraph	RotateVertices
ExpandGraph	Runs
ExtractCycles	SamenessRelation
FerrersDiagram	SelectionSort
FindCycle	SelfComplementaryQ
FindSet	SelfLoopsQ
FiniteGraphs	SetEdgeLabels
FirstLexicographicTableau	SetEdgeWeights
FolkmanGraph	SetGraphOptions
FranklinGraph	SetPartitionListViaRGF
FromAdjacencyLists	SetPartitionQ
FromAdjacencyMatrix	SetPartitions
FromCycles	SetPartitionToRGF
FromInversionVector	SetVertexLabels
FromOrderedPairs	SetVertexWeights
FromUnorderedPairs	ShakeGraph
FruchtGraph	ShortestPath
FunctionalGraph	ShortestPathSpanningTree
GeneralizedPetersenGraph	ShowGraph
GetEdgeLabels	ShowGraphArray
GetEdgeWeights	ShowLabeledGraph
GetVertexLabels	ShuffleExchangeGraph
GetVertexWeights	SignaturePermutation
Girth	Simple
Graph	SimpleQ
GraphCenter	Small
GraphComplement	SmallestCyclicGroupGraph
GraphDifference	Spectrum
GraphicQ	SpringEmbedding
GraphIntersection	StableMarriage
GraphJoin	Star

GraphOptions	StirlingFirst
GraphPolynomial	StirlingSecond
GraphPower	Strings
GraphProduct	Strong
GraphSum	StronglyConnectedComponents
GraphUnion	SymmetricGroup
GrayCode	SymmetricGroupIndex
GrayCodeKSubsets	SymmetricQ
GrayCodeSubsets	TableauClasses
GrayGraph	TableauQ
Greedy	Tableaux
GreedyVertexCover	TableauxToPermutation
GridGraph	TetrahedralGraph
GrotztschGraph	Thick
HamiltonianCycle	ThickDashed
HamiltonianPath	Thin
HamiltonianQ	ThinDashed
Harary	ThomassenGraph
HasseDiagram	ToAdjacencyLists
Heapify	ToAdjacencyMatrix
HeapSort	ToCanonicalSetPartition
HeawoodGraph	ToCycles
HerschelGraph	ToInversionVector
HideCycles	ToOrderedPairs
Highlight	TopologicalSort
HighlightedEdgeColors	ToUnorderedPairs
HighlightedEdgeStyle	TransitiveClosure
HighlightedVertexColors	TransitiveQ
HighlightedVertexStyle	TransitiveReduction
Hypercube	TranslateVertices
IcosahedralGraph	TransposePartition
IdenticalQ	TransposeTableau
IdentityPermutation	TravelingSalesman
IncidenceMatrix	TravelingSalesmanBounds
InDegree	Tree
IndependentSetQ	TreeIsomorphismQ
Index	TreeQ
InduceSubgraph	TreeToCertificate
InitializeUnionFind	TriangleInequalityQ
InsertIntoTableau	Turan
IntervalGraph	TutteGraph
Invariants	TwoColoring
InversePermutation	Type
InversionPoset	Undirected
Inversions	UndirectedQ
InvolutionQ	UnionSet
Involutions	Uniquely3ColorableGraph
IsomorphicQ	UnitransitiveGraph
Isomorphism	UnrankBinarySubset
IsomorphismQ	UnrankGrayCodeSubset
Josephus	UnrankKSetPartition
KnightsTourGraph	UnrankKSubset
KSetPartitions	UnrankPermutation
KSubsetGroup	UnrankRGF
KSubsetGroupIndex	UnrankSetPartition
KSubsets	UnrankSubset
LabeledTreeToCode	UnweightedQ
Large	UpperLeft

LastLexicographicTableau	UpperRight
LeviGraph	V
LexicographicPermutations	VertexColor
LexicographicSubsets	VertexColoring
LineGraph	VertexConnectivity
ListGraphs	VertexConnectivityGraph
ListNecklaces	VertexCover
LNorm	VertexCoverQ
LongestIncreasingSubsequence	VertexLabel
LoopPosition	VertexLabelColor
LowerLeft	VertexLabelPosition
LowerRight	VertexNumber
M	VertexNumberColor
MakeDirected	VertexNumberPosition
MakeGraph	VertexStyle
MakeSimple	VertexWeight
MakeUndirected	Vertices
MaximalMatching	WaltherGraph
MaximumAntichain	Weak
MaximumClique	WeaklyConnectedComponents
MaximumIndependentSet	WeightingFunction
MaximumSpanningTree	WeightRange
McGeeGraph	Wheel
MeredithGraph	WriteGraph
MinimumChainPartition	Zoom

In[5]:= **?Grid**

System`Grid

```
MakeBoxes[
  Grid[BoxForm`matrix_ /; TensorRank[Unevaluated[BoxForm`matrix], 2] == 2, BoxForm`opts___],
  BoxForm`fmt_] ^= Function[BoxForm`e, TagBox[GridBox[BoxForm`e, BoxForm`opts], Grid]][
  Map[Function[BoxForm`entry, MakeBoxes[BoxForm`entry, BoxForm`fmt], HoldAllComplete],
    Unevaluated[BoxForm`matrix], {2}]]
```

In[6]:= **?Vertices**

Vertices[g] gives the embedding of graph g, that is, the coordinates of each vertex in the plane. Vertices[g, All] gives the embedding of the graph along with graphics options associated with each vertex. Mehr...

In[7]:= **<<Graphics`Arrow`**

In[8]:= **?Graphics`Arrow`\***

## Graphics`Arrow`

Absolute HeadCenter HeadScaling HeadWidth ZeroShape  
Arrow HeadLength HeadShape Relative

In[9]:= **?Arrow**

Arrow[start, finish, (opts)] is a graphics primitive representing an arrow starting at start and ending at finish. Mehr...

Standard-Package anschauen:

■ Regarde le Package standard:

```
In[10]:= !!Graphics`Arrow`

(* :Title: Arrow Graphics Primitives *)

(* :Context: Graphics`Arrow` *)

(* :Author: John M. Novak *)

(* :Summary:
This package introduces the Arrow[start, finish] graphics
primitive and various style directives.
*)

(* :Package Version: 1.0.3 *)

(* :Mathematica Version: 2.2 *)

(* :Copyright: Copyright 1992-2005, Wolfram Research, Inc.*)

(* :History:
V 0.9 June 1992 by John M. Novak.
V 1.0 October 1992 by John M. Novak--substantial revisions.
V 1.0.1 March 1994 by John M. Novak -- bug fixes, including zero check
in the PostScript.
V 1.0.2 February 1997 by John M. Novak -- bug fix for DisplayString.
V 1.0.3 February 1998 by John M. Novak -- fix to allow Arrow objects
in Epilog or Prolog of any graphics object
*)

(* :Keywords:
Arrow, Vector, PostScript, Graphics
*)

(* :Sources:
PostScript Language Reference Manual, Adobe Systems
*)

(* :Limitations: Size of arrowhead cannot be taken into account
for autoscaling of plot ranges, since determination of
plot ranges are done in Mathematica, and the arrow heads are
generated in pure PostScript. *)

(* :Discussion: *)

BeginPackage["Graphics`Arrow`"]

Arrow::usage =
"Arrow[start, finish, (opts)] is a graphics primitive representing an
arrow starting at start and ending at finish.";

HeadShape::usage =
"HeadShape is an option to the Arrow primitive; it specifies
the shape of the arrow's head by Automatic, which specifies that the
shape is described by the parameters HeadLength, HeadCenter, and
HeadWidth, or it can be a list of a subset of the Mathematica graphics
primitives, drawn in the coordinate system scaled by HeadScaling. The
coordinate system is centered at the head of the arrow, with the negative
direction moving towards the tail of the arrow.";

HeadScaling::usage =
"HeadScaling is an option to the Arrow primitive; it specifies
the scaling used in the coordinate system for drawing the
arrowhead. Automatic scales the system to the graphic, where
{0,0} is at the head of the arrow, and the system is rotated along
the arrow, and the distance between 0 and 1 is equivalent to the width of
the graphic. Relative scales the coordinates of the arrowhead so that {0,0}
is at the head of the arrow, {-1,0} at the tail. Absolute scales to
the same coordinate system used in the device coordinate
```

```

system, rotated along the arrow, with {0,0} at the head.";

ZeroShape::usage =
"ZeroShape is an option to the Arrow primitive; it specifies
the shape of an arrow with no length (and hence no direction) in a form
similar to that of the HeadShape option. Note that the
parameterized form of HeadShape is not available. The coordinate system
is not rotated, but is scaled to HeadScaling. Automatic sets
the default zero arrow (a point.);";

HeadLength::usage =
"HeadLength is an option to the Arrow primitive. It is used when
HeadShape -> Automatic. It describes the length of the arrowhead, scaled
according to HeadScaling.";

HeadWidth::usage =
"HeadWidth is an option to the Arrow primitive. It is used when
HeadShape -> Automatic. It describes the width of the arrowhead, relative
to the length of the arrowhead (specified by HeadLength.);";

HeadCenter::usage =
"HeadCenter is an option to the Arrow primitive. It is used when
HeadShape -> Automatic. It describes the location of the center of the
base of the arrowhead along the length of the arrow, as a factor of the
length of the arrowhead. That is, if HeadCenter -> 0, the arrow will be
two lines; if HeadCenter -> 1, the arrowhead will be a perfect triangle;
otherwise, the arrowhead will be four-sided.";

Relative::usage =
"Relative is a possible value for the HeadScaling option to Arrow.
It specifies that the coordinate system in which the arrowhead is rendered
should be scaled to the length of the arrow, where {0,0} is at the head
of the arrow and {-1,0} is at the tail of the arrow.";

Absolute::usage =
"Absolute is a possible value for the HeadScaling option to Arrow.
It specifies that the device scaling should be used for the
arrowhead.";

Begin["`Private`"]

(* some global (but in private context) variables for caching the arrow
styles. *)
{ $$HeadDescriptions, $$HeadRoutines, $$ZeroDescriptions, $$ZeroRoutines};

(* A little utility function for checking numeric values. *)

numberQ[x_] := NumberQ[N[x]]

(* putting numbers in PostScript -
To make it easy to put numbers into PostScript, this utility hacks the
PostScript operator. Any number followed by a string is joined to the
string in the proper format; a number at the end is turned into a string.
*)

numtostring[num_];Abs[num] > 10^38 := numtostring[Sign[num] * 10^38]

numtostring[num_];Abs[num] < 10^-38 && Abs[num] > 0 :=
numtostring[Sign[num] * 9.9999999^-37]

numtostring[num_] :=
ToString[NumberForm[N[num], 8,
ExponentFunction -> (If[Abs[#] > 7, #, Null] &),
NumberFormat -> (If[#3 != "", StringJoin[#1, "e", #3], #1] &)]

]]

Unprotect[PostScript];

PostScript[x___, y_?numberQ] :=
PostScript[x,
numtostring[y]]

PostScript[a___, y_?numberQ, z_String, b___] :=

```

```

    PostScript[a,
      numtostring[y] <> " " <> z,
      b
    ]

Protect[PostScript]

(* define PostScript operators *)

(* math to ps coordinates; note that this implementation is
used in part because of the sparsity of Mathematica PostScript
(e.g., transform does not exist.) The MBeginOrig/MEndOrig
transform needs to be done every time, because this can be
different in a subgraph (i.e., I can't cache the transform
matrix once at the beginning.) Because the transform operator
doesn't exist, I use this moveto method; unfortunately, this
requires the gsave/grestore, which undoubtedly leads to some
performance hit. *)

mathtops =
  PostScript[
    "/mathtops {", (* stack: mathx mathy *)
      "gsave",
      "MBeginOrig",
      "moveto", (* stack: - *)
      "MEndOrig",
      "currentpoint", (* stack : psx psy *)
      "grestore",
    "} bind def"
  ];

tocoords =
  PostScript[
    "/MATocoords {", (* stack: beginx beginy endx endy *)
      "mathtops 4 2 roll mathtops", (* x2 y2 x1 y1 *)
      "4 copy pop pop", (* x2 y2 x1 y1 x2 y2 *)
      "3 -1 roll sub", (* x2 y2 x1 x2 y2-y1 *)
      "/array exch def",
      "exch sub", (* x2 y2 x2-x1 *)
      "/array exch def",
      "array dup mul", (* x2 y2 (x2-x1)^2 *)
      "array dup mul", (* x2 y2 (x2-x1)^2 (y2-y1)^2 *)
      "add sqrt", (* x2 y2 (sqrt((x2-x1)^2+(y2-y1)^2) *)
      "/arrl exch def", (* x2 y2 *)
      "translate", (* - *)
    "} bind def"
  ];

(* The following sets up the call to the doarrow routine. *)

Arrow::bad =
"Arguments `1` to Arrow are not valid.";

Options[Arrow] =
{HeadScaling -> Automatic,
HeadLength -> Automatic,
HeadCenter -> 1,
HeadWidth -> .5,
HeadShape -> Automatic,
ZeroShape -> Automatic};

evalarrow[{bx_?numberQ,by_?numberQ}, {ex_?numberQ,ey_?numberQ},
  opts:((_Rule | _RuleDelayed)...)] :=
Module[{head, zero, scale, len, cent, width,
  nbx = N[bx], nby = N[by], nex = N[ex], ney = N[ey]},
{head, zero, scale, len, cent, width} = {HeadShape, ZeroShape,
  HeadScaling, HeadLength, HeadCenter, HeadWidth}/.
{opts}/.Options[Arrow];
If[numtostring[nbx] === numtostring[nex] &&
  numtostring[nby] === numtostring[ney],
  arrow = generatezero[zero, scale, nbx, nby],
  arrow = generatehead[head, scale, len, cent, width,
  nbx, nby, nex, ney]

```

```

];
{Line[{{bx,by},{ex,ey}}, arrow]
]

evalarrow[args___] :=
  (Message[Arrow::bad, {args}]; {})

(* This routine checks whether the arguments correspond to a cached
arrowhead; if so, it returns the PostScript call to the routine. If not,
it build the PostScript string, caches the arrow description and the
PostScript, and returns the PostScript call to the cached routine. Note
that the preparearrows[] routine is the one to emit the cached routines.
*)

generatehead[head_, scale_, len_, cent_, width_, bx_, by_, nex_, ney_] :=
  Module[{pos},
    If[(pos = Position[$$HeadDescriptions, {head, scale, len, cent, width},
      {1}, Heads -> False]) != {},
      PostScript[bx, by, nex, ney,
        "MAarrowhead"<>ToString[ pos[[1,1]] ]],
    (* else *)
      buildhead[head, scale, len, cent, width];
      PostScript[bx, by, nex, ney,
        "MAarrowhead"<>ToString[Length[$$HeadRoutines]]]
  ]

]

generatezero[zero_, scale_, bx_, by_] :=
  Module[{pos},
    If[(pos = Position[$$ZeroDescriptions, {zero, scale},
      {1}, Heads -> False]) != {},
      PostScript[bx, by, "MAarrowzero"<>ToString[ pos[[1,1]] ]],
    (* else *)
      buildzero[zero, scale];
      PostScript[bx, by, "MAarrowzero"<>ToString[Length[$$ZeroRoutines]]]
  ]

]

(* These build the PostScript descriptions, and add descriptions to the
proper caching variables. *)

buildhead[head_, scale_, len_, cent_, width_] :=
  Module[{routine, pshead},
    AppendTo[$$HeadDescriptions, {head, scale, len, cent, width}];
    If[head === Automatic,
      pshead = fromparams[scale, len, cent, width],
      pshead = fromdescription[head]
    ];
    routine = {PostScript["gsave", "MATocoords", "arrl 0. eq",
      "{ 0 0 Mdot }", "{", arrowscale[scale],
      rotatesystem, pshead, PostScript["} ifelse", "grestore"]];
    AppendTo[$$HeadRoutines, routine]
  ]

]

buildzero[head_, scale_] :=
  Module[{routine, zhead, zscale},
    AppendTo[$$ZeroDescriptions, {head, scale}];
    If[head === Automatic,
      zhead = PostScript["0 0 Mdot"],
      zhead = fromdescription[head]
    ];
    If[scale === Absolute,
      zscale = arrowscale[scale],
      zscale = {}
    ];
    routine = {PostScript["gsave", "mathtops translate"], zscale,
      zhead, PostScript["grestore"]];
    AppendTo[$$ZeroRoutines, routine]
  ]

]

(* PostScript for rotating coordinate system *)

rotatesystem =

```

```

PostScript[
  "[ arrx arrl div", (* [ cos(t) *)
  "arry arrl div", (* [ cos(t) sin(t) *)
  "-1 arry mul arrl div",
  "arrx arrl div",
  "0 0 ]", (* [cos(t) sin(t) -sin(t) cos(t) 0 0] *)
  "concat" (* - *)
];

(* This generate the primitives describing a parameterized arrowhead. *)
(* fromparams determines the default length parameter depending on scaling; it
calls fromsparams *)

fromparams[Automatic, Automatic, c_, w_] :=
  fromsparams[.05, c, w]

fromparams[Relative, Automatic, c_, w_] :=
  fromsparams[.2, c, w]

fromparams[Absolute, Automatic, c_, w_] :=
  fromsparams[15, c, w]

fromparams[_ , l_, c_, w_] := fromsparams[l,c,w]

fromsparams[l_, _?(#==0&), w_] :=
  PostScript[
    -l, (w l)/2, " moveto 0 0 lineto ",
    -l, -(w l)/2, " lineto stroke"
  ]

(* note that filled arrowheads are outlined by a line to hopefully cause a
more graceful resizing... *)
fromsparams[l_, _?(#==1&), w_] :=
  Module[{nls = -l, ws = (w l)/2, nws = -(w l)/2},
    PostScript[
      nls, ws, "moveto 0 0 lineto",
      nls, nws, "lineto fill",
      nls, ws, "moveto 0 0 lineto",
      nls, nws, "lineto",
      nls, ws, "lineto stroke"
    ]
  ]

fromsparams[l_, c_, w_] :=
  Module[{nls = -l, ncs = -c l, ws = (w l)/2, nws = -(w l)/2},
    PostScript[
      nls, ws, "moveto 0 0 lineto",
      nls, nws, "lineto",
      ncs, "0 lineto fill",
      nls, ws, "moveto 0 0 lineto",
      nls, nws, "lineto",
      ncs, "0 lineto",
      nls, ws, "lineto stroke"
    ]
  ]

(* Create PostScript from a Mathematica-like syntax for describing head *)

fromdescription[head_] := head/.{Polygon[g_] :> frompoly[g],
  Line[l_] :> fromline[l],
  Point[p_] :> frompoint[p],
  Thickness[t_] :> PostScript[t, "w"]}

frompoly[g_] :=
  Module[{str = listtostring[g]},
    PostScript[
      str,
      "fill"
    ]
  ]

fromline[g_] :=

```

```

Module[{str = listtostring[g]},
  PostScript[
    str,
    "stroke"
  ]
]

frompoint[{x_,y_}] :=
  PostScript[
    x, y, "Mdot"
  ]

listtostring[l_] :=
  With[{initial = numtostring[#1]<>" "<>numtostring[#2]<>
    " moveto " & @@ First[l]},
    Fold[#1<>numtostring[First[#2]]<>" "<>numtostring[Last[#2]]<>
      " lineto "&,initial, Rest[l]
    ]
  ]

(* PostScript for coordinate scaling for arrowhead *)

HeadScaling::bad =
"Arguments `1` to HeadScaling are not Automatic, Relative, or Absolute; using
default of Automatic.";

arrowsscale[] := arrowsscale[Automatic]

arrowsscale[Automatic] := {}

arrowsscale[Relative] :=
  PostScript[
    "arrl arrl scale"
  ]

arrowsscale[Absolute] :=
  PostScript[
    "currentlinewidth 1 Mabswid",
    "currentlinewidth dup scale setlinewidth"
  ]

arrowsscale[x___] :=
  (Message[HeadScaling::bad, {x}]; arrowsscale[Automatic])

(* generate PostScript from cached descriptions, set up routines for preparing
to do arrows (This gets dumped into the start of the Prolog. *)

preparearrows[] :=
Module[{heads, zeros},
  heads = MapIndexed[
    {PostScript["/MAarrowhead"<>ToString[First[#2]]<>" {"},
      #1, PostScript["} def"]}&, $$HeadRoutines];
  zeros = MapIndexed[
    {PostScript["/MAarrowzero"<>ToString[First[#2]]<>" {"},
      #1, PostScript["} def"]}&, $$ZeroRoutines];
  Flatten[{mathtops, tocoords, heads, zeros}]/|.
    {x___, PostScript[a___], PostScript[b___], y___} :=>
      {x,PostScript[a,b], y}
  ]

(* remember that all the $$etc. are global... *)
(* was originally coded to only operate on Graphics objects, but it
turns out that Arrow objects can work in Epilog or Prolog, so at
cost of poorer error checking, the code was trivially modified to
check anything being passed to Display for an Arrow. This may
mean trouble for design of a 3D arrow, if it shares the head Arrow. *)
Unprotect[Display];

Display[stream_,
  ((ghead_)[igarg_, igopts___])?
  (!FreeQ[#,Arrow]&),
  opts___] :=
Module[{oldgropts, oldgaopt, prolog, p},

```

```

    $$HeadDescriptions = {}; $$HeadRoutines = {}; $$ZeroDescriptions = {};
    $$ZeroRoutines = {};
    Unprotect[Graphics, GraphicsArray];
    oldgropts = Options[Graphics];
    oldgaopts = Options[GraphicsArray];
    Options[Graphics] = Options[Graphics]/.
      Arrow[p___] :> evalarrow[p];
    Options[GraphicsArray] = Options[GraphicsArray]/.
      Arrow[p___] :> evalarrow[p];
    prolog = (Prolog/.Flatten[{igopts}]/.Options[ghead])/.
      Arrow[p___] :> evalarrow[p];
    garg = igarg/.Arrow[p___] :> evalarrow[p];
    gopts = {igopts}/.Arrow[p___] :> evalarrow[p];
    If[Head[prolog] === List,
      prolog = Join[preparearrows[],
        prolog],
      prolog = Append[preparearrows[], prolog]
    ];
    Display[stream,
      ghead[garg, Flatten[{Prolog -> prolog, Sequence @@ gopts}]],
      opts
    ];
    Unprotect[Graphics, GraphicsArray];
    Options[Graphics] = oldgropts;
    Options[GraphicsArray] = oldgaopts;
    Protect[Graphics, GraphicsArray];
    ghead[igarg, igopts]
  ]

Protect[Display];

Unprotect[DisplayString];

DisplayString[
  ((ghead_)[igarg_, igopts___])?
  (!FreeQ[#, Arrow]&),
  opts___] :=
Module[{oldgropts, oldgaopt, prolog, p, grout},
  $$HeadDescriptions = {}; $$HeadRoutines = {}; $$ZeroDescriptions = {};
  $$ZeroRoutines = {};
  Unprotect[Graphics, GraphicsArray];
  oldgropts = Options[Graphics];
  oldgaopts = Options[GraphicsArray];
  Options[Graphics] = Options[Graphics]/.
    Arrow[p___] :> evalarrow[p];
  Options[GraphicsArray] = Options[GraphicsArray]/.
    Arrow[p___] :> evalarrow[p];
  prolog = (Prolog/.Flatten[{igopts}]/.Options[ghead])/.
    Arrow[p___] :> evalarrow[p];
  garg = igarg/.Arrow[p___] :> evalarrow[p];
  gopts = {igopts}/.Arrow[p___] :> evalarrow[p];
  If[Head[prolog] === List,
    prolog = Join[preparearrows[],
      prolog],
    prolog = Append[preparearrows[], prolog]
  ];
  grout = DisplayString[
    ghead[garg, Flatten[{Prolog -> prolog, Sequence @@ gopts}]],
    opts
  ];
  Unprotect[Graphics, GraphicsArray];
  Options[Graphics] = oldgropts;
  Options[GraphicsArray] = oldgaopts;
  Protect[Graphics, GraphicsArray];
  grout
]

Protect[DisplayString];

(* OK, this is an ugly hack for several "graphic characteristic determination"
functions (FullOptions, FullGraphics, FullAxes, PlotRange).
This is not a complete solution, since it doesn't spot Arrows that are
set in default options. Also, FullGraphics is (for the time being) not

```

```

    being handled, since that would involve generating the proper PostScript
    primitives. Hopefully, this will all be able to be supersceded by some
    kernel design, so I am only making this temporary fix. --JMN, 10.93
*)

Unprotect[FullAxes];

FullAxes[gr_?(!FreeQ[#,Arrow]&), rest___] :=
  FullAxes[gr/.Arrow[beg_, end_, ___] :> Line[{beg, end}], rest]

Protect[FullAxes];

Unprotect[PlotRange];

PlotRange[gr_?(!FreeQ[#,Arrow]&), rest___] :=
  PlotRange[gr/.Arrow[beg_, end_, ___] :> Line[{beg, end}], rest]

Protect[PlotRange];

Unprotect[FullOptions]

FullOptions[gr_?(!FreeQ[#,Arrow]&), rest___] :=
  FullOptions[gr/.Arrow[beg_, end_, ___] :> Line[{beg, end}], rest]

Protect[FullOptions];

End[]

EndPackage[]

```

**In[11]:= ?ReadList**

```

ReadList["file"] reads all the remaining expressions in a file, and returns
a list of them. ReadList["file", type] reads objects of the specified type
from a file, until the end of the file is reached. The list of objects read
is returned. ReadList["file", {type1, type2, ...}] reads objects with a
sequence of types, until the end of the file is reached. ReadList["file",
types, n] reads only the first n objects of the specified types. Mehr...

```

**In[12]:= Options[Put]**

**Out[12]= {}**

**In[13]:= ?NullRecords**

```

NullRecords is an option for Read and related functions which specifies whether null
records should be taken to exist between repeated record separators. Mehr...

```

```

In[14]:= readIn = ReadList[
  "Graphics`Arrow`",
  Record,RecordSeparators -> {"\n","\t"}];
readIn >> C:\work\MathematicaData\DataRec;
readIn >> C:\work\MathematicaData\DataRec.ma;
readIn = ReadList[
  "Graphics`Arrow`",
  String];
readIn >> C:\work\MathematicaData\DataOuts;
readIn >> C:\work\MathematicaData\DataOuts.ma;

```

Diese Files können nun editiert und von Hand manipuliert werden. Z.B. wie folgt:

■ Ces fichiers peuvent être édités maintenant et manipulés à la main. P. ex. de la façon suivante:

```

In[20]:= OpenWrite["Statisti.m"];
WriteString["Statisti.m",
"\n",
"Du sollst das Package von Hand in das File","\n",
"Statisti.m kopieren."
]; (* End WriteString *)

Close["Statisti.m"];

(* Author: Student Heimlich Fileklau *)
(* History: Original version ... 1989. *)
(* Revised 1995. *)
(* Summary: This package computes descriptive *)
(* statistics *)
(* Keywords: mean, median *)

BeginPackage["AAAStatistics`"];
EndPackage[ ];

mean::usage =
"mean[list] gives the mean of the entries \
in list.";
median::usage =
"median[list] gives the median of the \
entries in list.";

Begin["`Private`"];
mean[list_] := Apply[Plus, list]/Length[list] /;
VectorQ[list] && Length[list] > 0;
median[list_] := Sort[list][[
(Length[list]+1)/2]] /;
VectorQ[list] &&
OddQ[Length[list]];
median[list_] := Block[{s, n},
s =Sort[list] ;
n = Length[list] ;
(s[[n/2]] + s[[n/2 + 1]]) / 2] /;
VectorQ[list] && EvenQ[Length[list]];
End[ ];

(* Protect[mean, median];
SetAttributes[ mean, ReadProtected];
SetAttributes[ median,ReadProtected]; *)

General::spell1 :
Possible spelling error: new symbol name "mean" is similar to existing symbol "Mean". Mehr...

General::spell1 :
Possible spelling error: new symbol name "median" is similar to existing symbol "Median". Mehr...

In[40]:= Needs["AAAStatistics`"]

In[41]:= ?AAAStatistics`*

Information::nomatch : No symbol matching AAAStatistics`* found. Mehr...

In[42]:= ?mean

mean[list] gives the mean of the entries in list.

```

```
In[43]:= ?median
```

median[list] gives the median of the entries in list.

```
In[44]:= mean[{1,2,3,4,5,6,7,8,9}]
```

```
Out[44]= 5
```

```
In[45]:= mean[{1,2,3,4,5,6,7,8}]
```

```
Out[45]=  $\frac{9}{2}$ 
```

```
In[46]:= median[{1,2,3,4,5,6,7,8}]
```

```
Out[46]=  $\frac{9}{2}$ 
```

```
In[47]:= median[{1,2,3,4,5,6,7,8,9}]
```

```
Out[47]= 5
```

Zum Beispielprogramm: Funktioniert es immer gut und somit tadellos? - Wenn nicht, so korrigiere es! Wenn Du Erfolg hast und es besser machen kannst, so hast Du ein grosses Ziel erreicht!

■ Quant au programme exemple: Fonctionne-t-il toujours bien et donc sans erreurs? - Si non, corrige-le. Si tu as du succès et que tu peux le faire mieux, tu as atteint un grand but!

---

## "Putzmaschine" einsetzen

### ■ Employer la "machine de nettoyage"

```
In[48]:= (* Old Form: Remove["Global`*"] *)
```

```
In[49]:= Remove["Global`*"]
```