Anhang B

Die Struktur des Repetitorium-Programmes

Das Programm des Repetitoriums liegt nun in der zweiten Version vor, stellt aber immer noch einen Prototyp dar. Zu seiner Realisierung wurde das Softwarprodukt Mathematica in der Version 4.0 benutzt. Seit Version 3.0 ist Mathematica mit einer graphischen Oberfläche ausgestattet und unterstützt eine besondere Dokumentform, in der (interaktive) Formeln und Textsatz vereint sind, die sogenannten *Notebooks*. Diese Notebooks sind aus Mathematica heraus programmierbar, woraus sich die realisierten neuen Möglichkeiten der Benutzerführung ergeben haben.

B.1 Realisierung der Oberfläche des Repetitoriums

Der Aufbau von Mathematica-Notebooks aus einzelnen Zellen, die je nach Typ Formeln, Text, Buttons oder anderes enthalten können, wird im Handbuch zu Mathematica detailliert erklärt. Damit sind Notebooks selber Ausdrücke, die mit den üblichen Mitteln von Mathematica manipuliert und verändert werden können. Benötigt werden nur noch eine Handvoll Funktionen, die Zellen, Zellinhalte oder auch ganze Notebooks in Variablen einlesen bzw. Notebooks oder Zellen schreiben können. Diese Funktionen sind (in Auswahl)

SelectionMove zum Navigieren innerhalb der Notebooks,

NotebookRead zum Einlesen von Notebooks oder Zellen,

NotebookWrite zum Schreiben und

NotebookPut zum Öffnen eines neuen Notebooks.

Diese Funktionen sind im Handbuch zwar beschrieben, ihr Einsatz zur Gestaltung einer Benutzeroberfläche wird aber nicht ausgeführt. Die folgenden Abschnitte sollen erläutern, wie man dabei vorgehen kann.

B.1.1 Auswahlmenü

Für die Gestaltung eines Auswahlmenüs findet sich im Mathematica-Handbuch ein Beispiel in Gestalt einer Palette, die ein Pulldown-Menü bereitstellt. Erläuterungen zur Arbeitsweise werden nicht gegeben, der Code ist unübersichtlich. Obwohl man denselben sicher hätte analysieren können, ist für das Auswahlmenü eine einfachere und direktere Konstruktion verwendet worden.

Die Grundidee besteht darin, einfach eine Zellengruppe zu verwenden, deren oberste Zelle den gewählten Menüpunkt anzeigt, während die anderen Zellen Buttons enthalten, welche bei Mausklick die Auswahl tätigen. Durch Öffnen bzw. Schließen der Zellgruppe werden die Auswahlbuttons sichtbar gemacht bzw. wieder verdeckt. Auf diese Weise muß nur die oberste Zelle, in der die Auswahl angezeigt wird, schreibend verändert werden, alle anderen Zellen sind statisch. Das Auswahlmenü in geschlossenem bzw. geöffnetem Zustand zeigen die Abbildungen A.2 bzw. A.3, ab Seite 38.

Interessant sind die Funktionen, die sich hinter den Auswahlbuttons verbergen. Die Funktion des Buttons *Auswahl* ist:

```
openChoices:=(Module[{nb,c,t},
        nb=ButtonNotebook[];
        SelectionMove[nb,All,CellGroup];
        c=NotebookRead[nb];
        c[[1,-1]]=Open;
        NotebookWrite[nb,c];
        ])&
```

Diese Funktion öffnet die Auswahlliste. Dies geschieht in der Zeile

```
  c[[1,-1]]=Open.
```

Verständlich wird dieser Schritt, wenn man sich die Zellgruppe, die das Auswahlmenü enthält, ansieht:

```
Cell[
  CellGroupData[..., Closed]
]
```

Die Details der Zellgruppe, durch „...“ angedeutet, sind hier nicht ausgeführt, wichtig ist, dass der letzte Parameter über „geöffnet“ oder „geschlossen“ entscheidet und durch `openChoices` umgeändert wird.

Die einzelnen Einträge der Auswahlbuttons sehen so aus (dargestellt am Beispiel der Übung 2.3):

```
Cell[
  BoxData[
    ButtonBox["Grundrechenarten (nur Zahlen)",
              Active->True,
              ButtonData->f2$3,
              ButtonSource->ButtonContents,
```

```
                ButtonEvaluator->Automatic,
                ButtonFunction->makeChoice
        ]
      ]
    ]
```

Dabei ist `f2$3` eine Funktion, die den Dialog für Übung 2.3 erstellt. Dazu wertet sie eine Liste mit den Beschreibungen der Buttons, die zum Dialog für Übung 2.3 gehören, aus. Die Funktion `makechoice` klappt die Auswahlliste wieder zu und erstellt dann mittels `f2$3` den Dialog:

```
     makeChoice:=(Module[{nb,c,t},
        nb=ButtonNotebook[];
        SelectionMove[nb,All,CellGroup];
        c=NotebookRead[nb];
(1)     t=#1[[1]];
(2)     c[[1,1,1]]=c[[1,1,1]]/.FrameBox[_]->FrameBox[t];
(3)     c[[1,-1]]=Closed;
        NotebookWrite[nb,c];
        ...
(4)     #2[];
        ])&
```

Hier wird in Zeile (1) der erste Parameter der Buttonfunktion in die Variable `t` gestellt. Dies ist[1] der Buttontext „Grundrechenarten (nur Zahlen)“. Zeile (2) schreibt diesen Text in das Textfeld des Auswahlmenüs. In Zeile (3) wird die Zellengruppe geschlossen. Zeile (4) ruft den zweiten Parameter der Buttonfunktion als Funktion auf, also die Funktion `f2$3`, die nun den Dialog für Übung 2.3 zeichnet.

Die Verwendung von `f2$3` mutet vielleicht etwas umständlich an, schließlich hätte man hier in Zeile (4) einfach mittels `NotebookWrite` jene Zellen explizit schreiben können, die den Dialog ausmachen. Ein kurzer Ausschnitt aus den Definitionen der Funktionen `fn$m`, nämlich

```
f2$3=(makeDialog[{
            {"Neue Aufgabe",DialogX["2.3"]&}
            }]&);

f2$4=(makeDialog[{
            {"Neue Aufgabe",DialogX["2.4"]&},
            {"Vorführen",
              VorfuehrungRechenaufgabe["2.4",aufgBaum]&}
            }]&);

f3$1=(makeDialog[{
            {"Neue Aufgabe",DialogX["3.1"]&},
            {"Vorführen",VorfuehrungFaktorAufgabe&},
            {"Kontrolle",KontrolleFaktorAufgabe&},
```

[1] siehe Mathematica-Handbuch, `ButtonFunction`, [WO99]

```
        {"Eigene Aufgabe",eigeneAufgabe&}
        }]&);
```

zeigt aber, dass auf die hier gewählte Weise die Übungsteile leicht und übersichtlich erweiterbar sind. Die Liste hinter `makeDialog` enthält Paare von Buttontexten und Buttonfunktionen. Man sieht auch die einfache Syntax: irgendein Symbol wird zu einer Funktion durch Anhängen von „&“.

B.1.2 Dialogsteuerung

Die Erstellung der Dialoge zu den Übungen ist kein besonderes Problem. Man kann sie fast wie mit einem GUI-Werkzeug entwerfen, indem man einfach in einem Hilfs-Notebook mit den üblichen Menüfunktionen von Mathematica den Dialog aus Texten und Buttons zusammenstellt, mit Hintergrundfarben, Schriftarten, etc. versieht und dann mittels `NotebookRead` in eine Variable einliest. Dann hat man kein Problem mehr, mit Hilfe dieser Variablen diesen Dialog dynamisch zu erzeugen. Man sehe sich noch einmal Abb. A.4 an: der Button *Neue Aufgabe* wird durch `makeDialog` geschrieben, der eigentliche Ein-/Ausgabebereich hingegen ist eine vorgefertigte Zellgruppe:

```
einAusCellGroup =Cell[CellGroupData[{
        Cell[BoxData[""],
          "Text",
          CellTags->"anfangsdummy",
          Editable->False,
          CellOpen->False],
        Cell["",
          "Text",
          FormatType->TextForm,
          Editable->False,
          CellTags->"aufgabentext"],
        Cell[BoxData[""],
          "Print",
          Background->RGBColor[1,1,0.5],
          ZeroWidthTimes->True,
          Editable->False,
          CellTags->"aufgabenfeld"],
        Cell["und tragen Sie ihre Antwort hier ein:",
          "Text",
          Editable->False,
          FormatType->TextForm],
        Cell[BoxData[""],
          "Print",
          Background->RGBColor[0.7,0.8,1],
          ZeroWidthTimes->True,
          Editable->True,
          CellTags->"eingabefeld"],
        Cell[BoxData[ButtonBox[
              "Prüfen",
```

```
            Active->True,
            ButtonStyle->"Evaluate",
            ButtonEvaluator->Automatic,
            ButtonFunction->Null
            ]],
        "Input",
        Editable->False,
        CellTags->"fertigbutton"],
      Cell[""",
        "Text",
        FormatType->TextForm,
        CellTags->"bemerkung",
        Editable->False,
        CellOpen->True],
      Cell[BoxData[""],
        "Print",
        Background->RGBColor[0.5,1,0.5],
        ZeroWidthTimes->True,
        CellTags->"antwortfeld",
        Editable->False,
        CellOpen->False],
      Cell[BoxData[""],
        "Text",
        CellTags->"analysefeld",
        Editable->False,
        CellOpen->True],
      Cell[BoxData[ButtonBox[
            "^",
            Active->True,
            ButtonStyle->"Evaluate",
            ButtonEvaluator->Automatic,
            ButtonFunction:>
              (SelectionMove[ButtonNotebook[],Before,Notebook]&)
            ]],
        "Input",
        Editable->False,
        CellOpen->False,
        CellTags->"topbutton"],
      Cell[BoxData[""],
        "Text",
        CellTags->"abschlussdummy",
        Editable->False,
        CellOpen->False]
      },
    Open]];
```

Hierzu einige Kommentare:

- Jede Zelle hat ein `CellTag` bekommen. Damit sind diese Zellen leicht und eindeutig mit `SelectionMove` ansteuerbar, können gelesen, verändert und

wieder neu geschrieben werden.

- Fast alle Zellen tragen das Attribut `Editable->False`. Dadurch wird der Dialog vor versehentlicher Veränderung geschützt. Eingabefelder (`CellTag` „eingabefeld“) sind natürlich ausgenommen. Das Beschreiben der ebenfalls schreibgeschützten Ausgabefelder geschieht durch einfache Funktionen wie z.B.

```
printToCell[inhalt_,tag_]:=
  Module[{nb},
    nb=SelectedNotebook[];
    cellEditable[tag];
    If[NotebookFind[nb,tag,All,CellTags]===$Failed,
      SelectionMove[nb,After,Notebook]];
    SelectionMove[nb,All,CellContents];
    NotebookWrite[nb,ToBoxes[inhalt,TraditionalForm]];
    cellNotEditable[tag];
    showCell[tag];
    ]

cellEditable[tag_]:=
  Module[{nb,cc},
    nb=SelectedNotebook[];
    If[NotebookFind[nb,tag,All,CellTags]===$Failed,Return[]];
    SelectionMove[nb,All,Cell];
    cc=NotebookRead[nb];
    cc=cc/.Rule[Editable,_]->Rule[Editable,True];
    NotebookWrite[nb,cc];
    ]

cellNotEditable[tag_]:=
  Module[{nb,cc},
    nb=SelectedNotebook[];
    If[NotebookFind[nb,tag,All,CellTags]===$Failed,Return[]];
    SelectionMove[nb,All,Cell];
    cc=NotebookRead[nb];
    cc=cc/.Rule[Editable,_]->Rule[Editable,False];
    NotebookWrite[nb,cc];
    ]
```

- Das Attribut `CellOpen` entscheidet über die Sichtbarkeit der Zelle. Durch Ändern von `False` auf `True` und umgekehrt kann die Zelle sichtbar gemacht bzw. verdeckt werden.

- Der Button *Prüfen* hat `Null` als Buttonfunktion. Diese wird beim Schreiben der Zellgruppe durch die richtige Funktion ersetzt:

```
c=c/.Null->((hideCell["antwortfeld"];
             hideCell["analysefeld"];
             hideCell["topbutton"];
             setFocus["eingabefeld"];
```

```
DialogIn[typ])&);
```

Funktionen wie `hideCell` haben einen `CellTag` als Parameter und tun genau das, was ihr Name sagt. Die eigentliche Prüfung der Eingabe übernimmt `DialogIn[typ]`, `typ` enthält die Übungsnummer.

Ein wenig komplexer ist es, Radiobuttons zu erstellen, wie sie im Dialog zur Kontrolle der Termumformungen bei Gleichungen verwendet wurden, siehe Abb. A.15. Hier bilden mehrere Buttons, verteilt auf mehrere Zellen, eine Gruppe von Radiobuttons. Zur Realisierung wurden die Zellen zu einer Zellgruppe zusammengefaßt. Nun können alle Radiobuttons in eine Variable eingelesen werden, indem diese Zellgruppe als Ganzes gelesen wird. Die Buttonfunktion eines Radiobuttons heißt `buttonSelected` und bekommt als Parameter den Buttontext:

```
ButtonBox["+",
  ButtonFunction:>(buttonSelected["+"]&),
  ButtonEvaluator->Automatic,
  Active->True,
  ButtonStyle->"Evaluate",
  Background->GrayLevel[0.75]],
```

Die Buttonfunktion selber sieht so aus:

```
buttonSelected[caption_]:=Module[{nb,cb,cg,mb},
    nb=ButtonNotebook[];
    SelectionMove[nb,Before,Notebook];
    SelectionMove[nb,Next,CellGroup];
    cg=NotebookRead[nb];
    cg=cg/.ButtonBox[b___,Rule[ButtonStyle,_],c___]->
          ButtonBox[b,Rule[ButtonStyle,"Evaluate"],c];
    cg=cg/.ButtonBox[b___,Rule[Background,_],c___]->
          ButtonBox[b,Rule[Background,GrayLevel[0.75]],c];
    cg=cg/.ButtonBox[caption,b___,Rule[ButtonStyle,_],c___]->
          ButtonBox[caption,b,Rule[ButtonStyle,"Hyperlink"],c];
    cg=cg/.ButtonBox[caption,b___,Rule[Background,_],c___]->
          ButtonBox[caption,b,Rule[Background,RGBColor[0.6,1.,0.6]],c];
    NotebookWrite[nb,cg];
    radio$Selection=caption;
    ]
```

Der Ablauf:

1. Einlesen aller Buttons in die Variable `cg`,
2. durch mehrfache Regelanwendung
   `cg=cg/.ButtonBox[b___,...]->ButtonBox[b___,...]`
   werden alle Buttons deselektiert (kenntlich durch `Style->"Evaluate"`, Farbe=Grau),
3. durch mehrfache Regelanwendung
   `cg=cg/.ButtonBox[caption,...]->ButtonBox[caption,...]`
   wird der ausgewählte Button selektiert (`Style->"Hyperlink"`, Farbe=Grün),

4. die Selektion wird in der globalen Variablen `radio$Selection` festgehalten.

B.1.3 Generierung von Notebooks

Die verschiedenen Notebooks für Vorführung, Kontrolle, Fehlermeldung etc. lassen sich genauso einfach erstellen und benutzen wie die Dialoge. Das Notebook *Vorführung* sei als Beispiel dargestellt:

```
vorfuehrNb=Notebook[{Cell[BoxData[
    ButtonBox["Schließen",
      Active->True,
      ButtonStyle->"Evaluate",
      ButtonEvaluator->Automatic,
      ButtonFunction:>(NotebookClose[ButtonNotebook[]]&)]
    ],
    "Input",
    CellTags->"closebutton"]},
    CellGrouping->Manual,
    Visible->False,
    ShowCellBracket->True,
    CellBracketOptions->{"Color"->GrayLevel[1]},
    ScreenStyleEnvironment->Working,
    WindowTitle->"Vorführung"];
```

Hier ist die Liste der Optionen interessant. Da solche Notebooks mehrfach geöffnet werden und der Benutzer gerne vergisst, sie wieder zu schließen, wurde dafür gesorgt, dass offene Notebooks wiederverwendet werden. Dazu wurde die Funktion `NotebookPut` umgeschrieben:

```
notebookRePut[hNb_,notebook_]:=
  Module[{hMargins,nNb,eOpt},
    hMargins=$Failed;
    If[Head[hNb]===NotebookObject,
      hMargins=Options[hNb,WindowMargins];
      NotebookClose[hNb];
    ];
    nNb=NotebookPut[notebook];
    eOpt=Options[nNb,Editable];
    SetOptions[nNb,Editable->True];
    If[hMargins=!=$Failed,
      SetOptions[nNb,#]&/@hMargins
    ];
    SetOptions[nNb,Visible->True];
    SetOptions[nNb,#]&/@eOpt;
    Return[nNb];
    ]
```

Die Variable `hNb` ist ein Handle, das beim vorigen Öffnen eines Notebooks mit `notebookRePut` zurückgegeben wird. Anhand dieses Handles kann die Position des Windows bestimmt und in `hMargins` abgelegt werden. Falls das Fenster

noch nie geöffnet war oder wieder geschlossen wurde – das ist feststellbar anhand des Inhalts von `hNb` bzw. an der Rückgabe `$Failed` für `hMargins` – wird das Notebook einfach neu geöffnet, sonst aber vorher geschlossen. Wenn vorhanden wird die alte Position aus `hMargins` wieder eingenommen. Man beachte den kleinen Trick mit der Option `Visible`: jedes Notebook ist zunächst unsichtbar und wird erst kurz vor Schluss der Funktion sichtbar gemacht, damit der Benutzer eine evtl. Positionsänderung nicht wahrnimmt. Ebenfalls wurde sorgfältig darauf geachtet, dass die Option `Editable` genau den Wert behält, den es vor dem Aufruf dieser Funktion hatte (das ist derjenige, der im vorbereiteten Notebook `notebook` eingetragen ist).

B.2 Realisierung der Übungsteile

In diesem Abschnitt soll nicht im Detail erklärt werden, wie die Übungen, deren Vorführung und Kontrolle realisiert wurden. Dazu muss auf den Quellcode verwiesen werden. Stattdessen soll hier auf spezielle Probleme und deren Lösung eingegangen werden.

Mathematica wendet sich ja an den erfahrenen Anwender, der Ergebnisse in Standardnotation und übersichtlicher Form, möglichst in irgendeiner einfachen Normalform, sehen möchte. Deshalb führt Mathematica viele Umformungen und Vereinfachungen im Hintergrund durch. Im Repetitorium sollen aber ja gerade Zwischenschritte sichtbar werden bzw. Eingaben auf ihre genaue Syntax hin untersucht werden. Die an sich auch im Repetitorium erwünschten Fähigkeiten von Mathematica müssen also manchmal umgangen oder ausgeschaltet werden.

Insbesondere ist es schwierig,

1. die Eingabestruktur von Termen mit ausschließlich Zahlen als Operanden zu erhalten,
2. solche Terme als Ausgabe zu generieren,
3. die genaue Struktur der Klammerung einer Eingabe zu erhalten,
4. Ausgaben mit einer definierten Klammerung zu generieren,

Verschiedene Lösungsmöglichkeiten sind denkbar:

- Die automatische Umformung von Eingaben dadurch zu unterdrücken, dass Funktionen wie `Times` und `Plus` alle Attribute wie `Flatten` und `Listable` entzogen werden.

  Allerdings bleibt dann auch von den erwünschten Fähigkeiten von Mathematica nicht viel übrig.

- Terme als Zeichenketten (*strings*) zu generieren und zu verarbeiten.

  So wurde in den Übungen vorgegangen, in denen mit Zahlen gerechnet wird.

- Verzicht auf die Kenntnis der genauen Termstruktur.

  So wurde in den späteren Übungen, in denen einfachere Umformungen bereits beherrscht werden sollten, vorgegangen.

- Verwendung von `HoldForm` und Varianten davon. Die Funktion `HoldForm` verhindert die Auswertung von Ausdrücken.

Darüber hinaus erwies sich die Überprüfung der Richtigkeit von Benutzereingaben als aufwendig. So erkennt Mathematica z.B. nicht ohne weiteres, dass die Ausdrücke

$$\frac{1}{2}(1+\sqrt{3}) \text{ und } \frac{1}{2}+\frac{1}{2}\sqrt{3}$$

äquivalent sind. Genauer: Es gilt nicht

```
1/2(1+Sqrt[3])===1/2+1/2 Sqrt[3].
```

Ebenso gilt nicht

```
MemberQ[{1/2(1+Sqrt[3])},1/2+1/2 Sqrt[3]]==True.
```

Daher konnten an vielen Stellen die von Mathematica bereitgestellten Funktionen nicht unmittelbar benutzt werden.

B.2.1 Grundrechenarten

Das Hauptproblem bei Übungen mit Zahlenrechnungen besteht darin, dass Mathematica jeden Ausdruck, der nur Zahlen enthält, sofort „ausrechnet“. Deshalb werden alle Rechenaufgaben als Baumstruktur generiert. Die folgende kleine Grammatik zeigt den einfachen Aufbau:

```
aufgBaum := List[operation, operand, operand]

operation := "+" | "-" | "*"

operand := aufgBaum | integer | rational

integer := digits | "(-" <> digits <> ")"

rational := digits <> "/" <> digits

digits := digit digits | (leeres Wort)

digit := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Ein typisches Beispiel wäre

```
aBaum = {"*", "(-23)", {"-", "2/3", "7"}},
```

welches natürlich für

$$(-23)\cdot(\frac{2}{3}-7)$$

stehen soll.

Ein solcher Aufgabenbaum kann in einen Zahlausdruck umgewandelt werden:

```
BaumZuString[x_String]:=x

BaumZuString[{p_,a_,b_}]:=If[p==="*",
    BaumZuString[a]<>p<>BaumZuString[b],
    "("<>BaumZuString[a]<>p<>BaumZuString[b]<>")"
    ]
```

Im Beispiel wäre

```
BaumZuString[aBaum] = "(-23)*(2/3-7)"
```

Der Aufruf `ToExpression[BaumZuString[aBaum]]` liefert dann die Zahl, die sich bei der Auswertung des Zahlausdrucks ergibt.

Zur Anzeige von Aufgaben für den Benutzer muss der Aufgabenbaum mittels `HoldForm` aufbereitet werden, damit Auswertungen verhindert werden. Gleichzeitig wird eine möglichst korrekte Klammerung angestrebt:

```
BaumZuStringHold[x_]:=
  If[Head[x]===String,
(* then *)
    "HoldForm[" <> x <> "]",
(* else *)
    "HoldForm[(" <>
      BaumZuStringHold[x[[2]]] <>
      x[[1]]<>"HoldForm[" <>
      BaumZuStringHold[x[[3]]] <>
      "])]"]
```

Wieder muss `ToExpression` auf das Resultat angewendet werden, um einen Ausdruck zu erhalten, der angezeigt werden kann. Durch mehrfaches Anwenden von `ReleaseHold` kann der Ausdruck auch ausgewertet werden.

Die Schachtelung von `HoldForm` ist nötig, weil ein Ausdruck wie

```
HoldForm[2*3-4*5+(-3)*(-3-6)]
```

zwar nicht ausgewertet wird, der Rechenausdruck innerhalb des `HoldForm`-Aufrufs aber umgeordnet oder anders geklammert werden kann. Will man solche Details kontrollieren, so muss man innere Teile ebenfalls mit `HoldForm` schützen.

Besonders sorgfältig muss man vorgehen, wenn Aufgaben mit geschachtelten Brüchen wie in Übung 4.3 vorgeführt werden sollen. Dabei müssen außerdem Zähler und Nenner eines Bruches aus den Strings isoliert werden.

B.2.2 Termumformungen

Die Behandlung algebraischer Terme ist weniger problematisch. Mathematica wandelt Terme nicht automatisch in Normalformen um. So werden z.B. Klammerausdrücke nicht ausmultipliziert. Erst die Anwendung von Funktionen wie

`Expand` führt Termumformungen durch. Daher ist die Generierung von Aufgaben unproblematisch und kann durch explizite Vorgabe von Mathematica-Ausdrücken erfolgen.

Größere Sorgfalt erfordert die Prüfung der Eingaben und die Vorführung von Aufgaben. Wenn die genaue Gestalt der Eingabe wichtig ist, wird die Eingabe beim Einlesen mit einer `HoldForm`-Anweisung umhüllt.

Bei der Vorführung von Aufgaben soll es möglich sein, Zwischenschritte sichtbar zu machen. Deshalb sollte eine `Expand`-Anweisung nicht sofort vollständig ausgeführt werden. Oft hilft ebenfalls `HoldForm`, in manchen Fällen muss anders vorgegangen werden. Als Beispiel sei der Code für die schrittweise Faktorisierung eines Terms („Ausklammern“, Übung 2.3) gezeigt:

```
VorfuehrungExpandAufgabe0[aufg_]:=
  Module[{fakts,n,l,letzter,m},
    fakts=FactorList[aufg];
    l=Length[fakts];
    fakts=Apply[Power,fakts,{1}];
    fakts=Sort[fakts,Depth[#1]<Depth[#2]&];
    If[fakts[[-1,0]]===Power&&
        fakts[[-1,2]]===2,n=l,
      n=l-1];
    printLineNb[vNb,"Fortschreitendes Ausklammern
      ergibt nacheinander"];
    Do[printLineNb[
        vNb,(Times@@Take[fakts,{1,m}])(Expand[
              Times@@Take[fakts,{m+1,l}]])],{m,1,n}]]
```

Bei den Vorführungen von Umformungen ergab sich generell das Problem, dass die Terme einer längeren Formel nach internen Sortierkriterien von Mathematica angeordnet werden. Nach einem Umformungsschritt ist dann irritierenderweise die Reihenfolge der Terme anders als erwartet. Beispielsweise wird aus

$$(a+x)(x+y)$$

nach dem Ausmultiplizieren mit `Expand`

$$ax+ay+xy+x^2,$$

anders, als dies ein menschlicher Schreiber tun würde. Auf die Sortierreihenfolge von Mathematica kann man keinen Einfluss nehmen. Die gewünschte Reihenfolge wäre nur durch massiven Einsatz von `HoldForm` zu erzwingen. Da damit behandelte Terme aber nur schwer weiterverarbeitet werden können, wurde davon Abstand genommen. Dieser didaktisch unschöne Umstand bleibt also unbefriedigend.

B.3 Fehlererkennung und -behandlung

Erklärtes Ziel des Repetitoriums ist es, die Antworten des Benutzers nicht nur mit „richtig“ oder „falsch“ zu bewerten, sondern auch Hinweise auf den

gemachten Fehler zu geben. Diesem Ziel sind dadurch Grenzen gesetzt, dass die meisten Fehler schlicht Rechenfehler sein dürften und nur die wenigsten auf einem expliziten Verstoß gegen Rechenregeln beruhen werden. Reine Rechenfehler wie $2 + 2 = 5$ oder Ansammlungen von Unachtsamkeiten wie in $(2a+7b)(6a^2+9b) = 12a^3+42ab+27ab+63b^2$ ergeben so viele Möglichkeiten von falschen Ergebnissen, dass eine Analyse des Fehlers in vielen Fällen nicht möglich ist. Es wird deshalb vor allem nach Fehlern durch Regelverstöße ohne zusätzliche Flüchtigkeitsfehler gesucht. Weiterhin sollte das Programm möglichst tolerant gegenüber sytaktischen Fehlern bei der Eingabe sein.

B.3.1 Fehler bei der Eingabe

Das Repetitorium, welches ja vor allem für Benutzer gedacht ist, welche nicht im Umgang mit Mathematica geübt sind, muss Fehler bei der Eingabe möglichst gut tolerieren. Der Benutzer wird ja im Dialog geführt, die Zellen der Notebooks sind schreibgeschützt, so dass Fehler nur bei der Eingabe von Antworten in die vorgesehenen Eingabefelder auftreten können. Man kann zwei Arten falscher Eingaben unterscheiden:

- Eingaben, die nicht zur aktuellen Übung passen, ansonsten aber korrekt sind, und
- Eingaben, die syntaktische Fehler enthalten und keine gültigen Mathematica-Ausdrücke darstellen.

Die erste Art von falschen Eingaben wird durch die jeweiligen Übungsteile abgefangen. Im allgemeinen erhält der Benutzer einen Hinweis darauf, welche Antworten erwartet werden, und wenn möglich, wird ein direkter Hinweis darauf gegeben, was nicht passend ist. Für den Zweck der Untersuchung der Eingaben stehen innerhalb Mathematica genügend viele Funktionen bereit, z.B.

NumberQ zum Test, ob eine Eingabe numerisch ist;

PolynomialQ zum Test, ob ein Polynom eingeben wurde;

UserSymbols – eine einfache Funktion, die nicht zum Standardvorrat von Mathematica gehört – liefert alle Variablen eines Ausdrucks[2];

Cases mit geeigneten Mustern kann z.B. alle Wurzelexponenten eines Ausdrucks liefern und so die Eingabe von dritten Wurzeln erkennen und ablehnen;

...

Mit ihrer Hilfe sind die nötigen Prüfungen relativ leicht zu erledigen und beliebig sorgfältig durchzuführen.

Schwieriger zu behandeln sind Syntaxfehler. Mathematica reagiert nämlich auf den Versuch, einen fehlerhaften Ausdruck einzulesen, immer mit einer Fehlermeldung, die das optische Bild eines Dialogs stört. Es wurde folgendes Verfahren angewendet:

[2] siehe [MA01]

1. Eingabezellen werden zunächst per `NotebookRead` eingelesen. Man erhält eine Zeichenkette, die die Boxstruktur der eingelesenen Zelle enthält. Diese Boxstruktur an sich ist stets syntaktisch korrekt, ist aber kein auswertbarer Ausdruck.

2. Die Funktion `ToExpression` versucht, eine Zeichenkette in einen Mathematica-Ausdruck umzuwandeln. Genau hier treten unvermeidbar Fehlermeldungen auf. Deshalb werden unmittelbar vor dem Aufruf von `ToExpression` alle Fehlermeldungen aus-, und danach wieder eingeschaltet.

3. Da im Fehlerfalle der Rückgabewert auf das spezielle Symbol `$Failed` gesetzt ist, läßt sich eine syntaktisch falsche Eingabe danach zweifelsfrei erkennen und eine entsprechende Meldung ausgeben.

Die Funktion `InputExpressionHold` führt diesen Ablauf durch:

```
InputExpressionHold[tag_]:=
  Module[{input},
    input=readFromCell[tag];
    messagesOff;
    input=ToExpression[input,TraditionalForm,HoldForm];
    messagesOn;
    If[input===$Failed,
      printMessage["Syntaxfehler, bitte Eingabe korrigieren!"];
      Return[False]];
    If[input=!={}&&input[[1,0]]==Set,input[[1,0]]=Equal];
    expr=ReleaseHold[input];
    If[expr==ComplexInfinity||expr==Indeterminate,expr=$Failed;
      printMessage["Division durch Null ist nicht erlaubt!"];
      Return[False]];
    Return[input]
    ]
```

In dieser Funktion sind noch weitere Dinge miterledigt worden:

- Durch den dritten Parameter `HoldForm` im Aufruf von `ToExpression` wird erreicht, dass der erhaltene Ausdruck nicht sofort ausgewertet wird. Dadurch wird eine genaue Untersuchung des eingegebenen Terms möglich. Andernfalls erhielte man eine Normalform desselben, die manche Prüfung unmöglich macht.

- Die Zeile

  ```
  If[input=!={}&&input[[1,0]]==Set,input[[1,0]]=Equal];
  ```

  ermöglicht es, einen typischen Eingabefehler, nämlich die Eingabe einer Bestimmungsgleichung mit dem einfachen Gleichheitszeichen „=“ statt korrekt mit dem doppelten „==“, für den Benutzer folgenlos zu lassen. Eine Eingabe der Form `a+x=0` wird nämlich zunächst als `HoldForm[Set[a+x,0]]` eingelesen. Durch die obige Zeile wird die falsche Zuweisung `Set` durch den richtigen Vergleich auf Gleichheit `Equal` ersetzt, und es wird `a+x==0` weiterverarbeitet, ohne dass der Benutzer irgendetwas davon bemerkt.

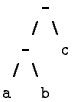
- Eingaben, die bei ihrer Auswertung auf eine Division durch Null – und damit zu unschönen Fehlermeldungen – führen würden, werden ebenfalls abgefangen.

In ähnlicher Weise wurde der Fall behandelt, dass die Lösungsmenge einer Gleichung, insbesondere wenn es nur eine Lösung gibt, ohne Mengenklammern eingegeben wurde. Dies dürfte auch ein typischer Eingabefehler sein. Hier werden die Klammern automatisch ergänzt und es wird ein Hinweis darauf gegeben. Leider können die Klammern nicht automatisch ergänzt werden, wenn es mehrere Lösungen gibt und der Benutzer z.B. statt „$\{2,3\}$“ nur „$2,3$“ eingibt. Die Eingabe „$2,3$“ wird nämlich von Mathematica als unvollständige Eingabe angesehen und wird auf keine Weise von `ToExpression` akzeptiert. Auf die mögliche Untersuchung der Boxstruktur daraufhin, ob sie eine Liste von durch Kommas getrennten Zahlen enthält, wurde verzichtet, da Boxstrukturen sehr unübersichtlich sein können.

B.3.2 Fehlerhafte Antworten

Ein von Anfängern häufig gemachter Fehler ist die Umformung

$$a - (b - c) = a - b - c.$$

Stellt man Terme als Bäume dar, so sieht man, dass hier der Baum  fälschlich durch den Baum  ersetzt wird. Diese beiden Bäume gehen durch eine sog. *Linksrotation* auseinander hervor. Viele typische Fehler lassen sich auf eine Linksrotation oder die analoge Rechtsrotation des Termbaumes zurückführen.

Eine Linksrotation ist in Mathematica leicht direkt zu programmieren:

```
LinksRotation[t_]:=Apply[t[[2,0]],
       {Apply[t[[0]],{t[[1]],t[[2,1]]}],
        t[[2,2]]}]
```

Besser und übersichtlicher ist es, die Linksrotation als Ersetzungsregel zu formulieren:

```
LinksRotationsRegel=p_[a_,q_[b_,c_]]->q[p[a,b],c]
```

Man wendet diese beiden Möglichkeiten wie folgt an:

```
In[]  = LinksRotation[a-(b-c)]
Out[] = a-b-c
```

bzw.

```
In[]  = a-(b-c) /. LinksRotationsregel
Out[] = a-b-c
```

Die beiden Varianten wirken in gleicher Weise auf *alle* Ausdrücke, die entweder entsprechend viele Teile haben oder auf das Muster passen. Das ist aber manchmal nicht der Fall:

```
In[]  = a-(b-c)+d /. LinksRotationsregel
Out[] = a-(b-c)+d
```

Die Ursache liegt in der Eigenschaft *Listable* von Symbolen wie *Plus*, die dazu führen, dass `a-(b-c)+d` die interne Darstellung

```
Plus[a,Times[-1,Plus[b,Times[-1,c]]],d]
```

hat, auf welche das Muster `p_[a_,_]` nicht passt. Eine gewisse Abhilfe bietet

```
LinksRotationsRegel=p_[a_,q_[b_,c_,r___],s___]->q[p[a,b,r],c,s]
```

mit den Platzhaltern `r___` und `s___` für beliebig viele oder auch keine weiteren Parameter.

Alternativ läßt sich der obige Fehler durch eine spezielle Regel für genau diesen Fall darstellen:

```
a_-(b_-c_) -> a-b-c
```

Wenn ein Benutzer den Term `a-(b-c)` fälschlich in `a-b-c` umgeformt hat, so kann man entweder die Linksrotation oder die Regel auf den Term `a-(b-c)` anwenden, das Ergebnis evaluieren und mit der Eingabe `a-b-c` vergleichen. Bei Übereinstimmung ist der Fehler erkannt und ein entsprechender Hinweis kann ausgegeben werden.

Die Rotationsregeln sind, da sie auf alle möglichen Ausdrücke passen, in manchen Fällen zu allgemein und ergeben „Fehler"-Terme, die weit jenseits dessen liegen, was ein Benutzer an falschen Eingaben produzieren würde:

```
In[] =  a-b /. LinksRotationsRegel
Out[] = (a-1)b
```

Zur Erklärung bedenke man, dass $a - b$ intern als `Plus[a,Times[-1,b]]` dargestellt wird.

In dieser Arbeit wurden beide Ansätze weiterverfolgt. Für das Ausmultiplizieren von Termen (Übung 3.1) wurden z.B. folgende Regeln verwendet:

```
x_ (y_+z_)          ->    x y - x z,
x_ (y_+z_)          ->   -x y - x z,
(a_+b_) (c_+d_) e_. ->  ( a c + b d)e,
(a_+b_) (c_+d_) e_. ->  ( a d + b c + b d)e,
(a_+b_) (c_+d_) e_. ->  (-a c + a d + b c + b d)e,
(a_+b_) (c_+d_) e_. ->  (-a c - a d + b c + b d)e,
(a_+b_) (c_+d_) e_. ->  (-a c - a d - b c + b d)e,
(a_+b_) (c_+d_) e_. ->  (-a c + a d + b c - b d)e,
a_ (b_+c_)          ->    a b + c
```

Die linken Seiten der Regeln sind stets für die Addition formuliert und nicht für die Subtraktion, was ja eigentlich näher liegen würde. Weil Mathematica grundsätzlich die Subtraktion auf die Multiplikation mit -1 und anschließende Addition zurückführt, decken die Regeln so alle gewünschten und noch weitere Fälle zusätzlich ab.

Die Regeln decken auch unvollständiges Ausmultiplizieren ab (3., 4. und letzte Regel).

Für die Rechenoperationen mit Zahlen (Übung 2.4) wurden hingegen Rotationsregeln eingesetzt:

```
{q_,a_,{p_,b_,c_}}          ->  {p,{q,a,b},c}
{p_,{q_,a_,b_},c_}          ->  {q,a,{p,b,c}}
{r_,a_,{q_,b_,{p_,c_,d_}}}  ->  {p,{q,{r,a,b},c},d}
{{p_,{q_,{r_,a_,b_},c_},d_} ->  {r,a,{q,b,{p,c,d}}}
```

Diese Regeln beschreiben sukzessive eine Links-, eine Rechts-, eine doppelte Links- und eine doppelte Rechtsrotation. Dass hier die Ausdrücke als geschachtelte Listen formuliert sind liegt daran, dass für die Rechenoperationen mit Zahlen – wie weiter oben beschrieben – Ausdrücke mit Zahlen explizit als Termbäume modelliert wurden, um deren Evaluation durch Mathematica kontrollieren zu können.

Sämtliche obigen Regeln scheinen das vorhin angesprochene Problem mit der *Listable*-Eigenschaft nicht zu beachten, da es keine Platzhalter wie `r___` gibt. In Wirklichkeit tritt dieses Problem deshalb nicht auf, weil die Regeln stets auf Teilterme angewendet werden, die eben nicht zu viele Argumente enthalten. Solche Teilterme können mit der Funktion `Position` gefunden werden:

```
In[] =  Position[a(b+c)+d(e+f),x_+y_]
Out[] = {{},{1,2},{2,2}}
```

Anschaulicher ist die Funktion `Cases`, die die gefundenen Teilterme zurückgibt:

```
In[] =  Cases[a(b+c)+d(e+f),x_+y_]
Out[] = {a(b+c)+d(e+f),b+c,e+f}
```

Seit der Version 3.0 von Mathematica ist es auch möglich, nicht nur die erste mögliche, sondern alle Belegungen eines Musters zu finden, die zu einem Term passen:

```
In[] =  (a+b)(c+d) /. x_(y_+z_)->{x,y,z}
Out[] = {a+b,c,d}

In[] =  ReplaceList[(a+b)(c+d), x_(y_+z_)->{x,y,z}]
Out[] = {{a+b,c,d},
         {a+b,d,c},
         {c+d,a,b},
         {c+d,b,a}}
```

Erst `ReplaceList` in Verbindung mit `Cases` und `Position` macht eine vollständige Suche nach falschen Termumformungen mit Regeln möglich.

Rechenfehler, die sich nur in falschen Koeffizienten äußern, lassen sich kaum sinnvoll durch Regeln abdecken, da man sich ja auf alle erdenklichen Arten „verrechnen“ kann. Es ist aber über Funktionen wie `CoefficientList`, die die Koeffizienten eines Polynoms – auch in mehreren Veränderlichen – zurückgibt, wenigstens zu prüfen, ob alle erwarteten Terme überhaupt vorhanden sind. Dann kann immerhin noch der allgemeine Hinweis auf einen Rechenfehler gegeben werden.

B.4 Generierung von zufälligen Aufgaben

Es wurde besonderer Wert darauf gelegt, den Benutzer mit einer reichlichen und vielfältigen Menge von Aufgaben zu versorgen, die sich nur sehr selten wiederholen und alle denkbaren Sonderfälle mit enthalten sollen.

B.4.1 Termskelette

Da sich jeder algebraische Term – auch Gleichungen – als (binärer) Baum mit Operationen in den Knoten und Operanden in den Blättern darstellen läßt, ist die naheliegendste Idee, einen Aufgabentyp durch Bedingungen an den Baum zu charakterisieren und dann zufällig Bäume, die diesen Bedingungen genügen, zu generieren. Die Programmiersprache von Mathematica bietet gute Möglichkeiten zur Listenverarbeitung – ähnlich wie in LISP. Es ist daher kein Problem, Terme als Baum zu generieren und unmittelbar in Mathematica weiter zu verwenden.

Für diesen Prototyp wurde ein anderer Weg eingeschlagen: Für jeden Aufgabentyp wurden mehrere Untertypen als Termskelett in einer Liste abgelegt. Die Operanden und evtl. auch die Operationen in diesem Termskelett sind durch Variable belegt, hinter denen sich eine Funktion zur zufälligen Generierung eines Operanden oder einer Operation verbirgt. Typische Vertreter sind etwa

```
RandomSign:=2Random[Integer]-1
ZahlOderNull:=RandomSign Random[Integer,10]
positiveZahl:=Random[Integer,{1,10}]
Zahl:=RandomSign positiveZahl
linearerTerm:=Zahl x+ZahlOderNull
positiverLinearerTerm:=positiveZahl x + ZahlOderNull
Wurzelterm:=Sqrt[positiverLinearerTerm]
```

Diese Variablen können z.B. als Termskelett

```
Wurzelterm+linearerTerm==linearerTerm,
```

für Wurzelgleichungen verwendet werden.

Diese Methode eignet sich auch für jene Übungsteile, die Aufgaben mit Zahlen enthalten. Die dort verwendeten Listen mit Baumstrukturen für Ausdrücke

lassen sich auf die gleiche Weise mit zufälligen Inhalten bei vorgegebener Termstruktur füllen.

Die Erzeugung von zufälligen Termen über Zufallsbäume von Ausdrücken wurde nicht weiter verfolgt. Die gestellten Aufgaben dürfen nämlich nicht zu umfangreich sein, wenn der Benutzer sich nicht in zu langwierigen Rechnungen verstricken soll. Für Aufgaben, die eine gewisse Länge nicht überschreiten, lassen sich aber die möglichen Terme relativ rasch vollständig aufzählen und in einer Liste von Termskeletten ablegen. Man gewinnt dadurch die Möglichkeit, jeden Term auf eine auf ihn speziell zugeschnittene Weise zu behandeln. Dieser Weg der individuellen Behandlung wurde z.B. für Potenzaufgaben mit Zahlen gewählt. Damit die Potenzgesetze sinnvoll anwendbar sind, muss nicht nur der Term die richtige Struktur haben, er muss auch mit passenden Zahlen gefüllt werden, damit interessante Aufgaben entstehen. Jeder Termtyp hat außerdem seinen eigenen spezifischen Lösungsweg. Dies alles würde bei zufällig gewählten Termstrukturen erheblich aufwendiger zu realisieren sein.

B.4.2 Glatte Lösungen

Manche Aufgabentypen können bei der Lösung schnell zu großen Zahlen und komplizierten Lösungen führen. Dies gilt besonders für Gleichungen 2. Grades, vor allem dann, wenn sie noch Wurzelterme enthalten. Der Lerneffekt ist nicht besonders hoch, wenn der Benutzer sich mit Lösungsmengen wie

$$x \in \left\{ \frac{29}{33} + \sqrt{\frac{1258}{363}}, \frac{29}{33} - \sqrt{\frac{1258}{363}} \right\}$$

herumschlagen muss. Es ist deshalb sinnvoll, die Gleichungen so auszuwählen, dass glatte, also ganzzahlige Lösungen, oder zumindest nur Brüche mit kleinen Nennern vorkommen. Eine Gleichung auf eine bestimmte Lösung hin zu konstruieren ist nicht ganz einfach und macht eine Zufallsauswahl schwierig. Deshalb wurde ein anderer Ansatz implementiert, der darin besteht, eine zufällig ausgewählte Gleichung im Nachhinein so zu verändern, dass eine ganzzahlige Lösung herauskommt. Zufällige Gleichungen haben meist irrationale Lösungen. Eine davon kann man auf eine ganze Zahl runden und in die Gleichung einsetzen. Dann kommt nicht mehr Null heraus, sondern etwas anderes. Addiert man diese Zahl zur rechten Seite der Gleichung, so hat man was man will. Die Funktion `GlatteLoesung` leistet dies:

```
GlatteLoesung[aufg_]:=
  Module[{xs,korr,nenner},
    xs=Solve[aufg];
    If[xs===={},Return[aufg]];
    xs=x/.(xs//N);
    xs=xs[[1]];
    If[Head[xs]===Complex,Return[aufg]];
    If[Head[xs]===Rational,Return[aufg]];
    If[Head[xs]===Integer,Return[aufg]];
    xs=xs//Round;
    nenner=Denominator[Subtract@@aufg[[1]]//Together];
```

```
While[nenner/.x->xs==0,xs+=1];
korr=(Subtract@@aufg[[1]])/.x->xs;
ReplacePart[aufg,aufg[[1,2]]+korr,{1,2}]
]
```

Hierin ist `aufg` eine zuvor zufällig erzeugte Gleichung[3]. Diese wird numerisch gelöst, eine Lösung gerundet, der nötige Summand für die rechte Seite (in der Variablen `korr`) bestimmt und mit `ReplacePart` in die Gleichung eingefügt. Etliche Sonderfälle müssen beachtet werden: falls die Gleichung keine Lösung hat oder diese komplex ist, kann das Verfahren nicht angewendet werden. Das Verfahren ist andererseits nicht nötig, wenn eine Lösung bereits ganzzahlig oder rational ist. Die Polstellen bei Aufgaben mit rationalen Funktionen müssen besonders beachtet werden: die gerundete Lösung darf nicht auf einen Pol zu liegen kommen. Hier wird in der `While`-Schleife die gewünschte Lösung so lange um 1 erhöht, bis man nicht mehr auf einer Polstelle steht.

Gleichungen mit Wurzeltermen können leider nicht so behandelt werden, weil die Korrektursummanden im allgemeinen selbst irrational sind. Die geänderte Gleichung hat dann meist einen wesentlich komplizierteren Lösungsweg, der sich nicht mehr automatisch vorführen läßt. Deshalb wurde hier doch versucht, Aufgaben mit einer vorgegebenen Lösung zu konstruieren. Zu diesem Zweck wurde das weiter oben genannte einfache Termskelett für Wurzelterme anders gestaltet:

```
Wurzelterm[xs_] :=
  Module[{a, c},
    If[Head[xs] === Rational,
      a = 2 kleineZahl,
      a = positiveZahl;
      c = kleineZahl;
      Sqrt[a x + c^2 - a xs]
  ]
```

Hier ist `xs` die gewünschte Lösung, die auch halbzahlig sein kann. Es wird ein zufälliger Wurzelterm zurückgeliefert, der die kleine ganze Zahl `c` als Wert hat, wenn man $x = xs$ setzt. Die entstehende zufällige Gleichung muss am Ende noch korrigiert werden, damit auch `xs` Lösung wird, denn durch `Wurzelterm` sind nur ganzrationale Wurzeln garantiert.

Allerdings gestaltet sich die Korrektur der Gleichung etwas schwieriger, da dafür Sorge getragen werden muss, dass die neue Gleichung höchstens vom 2. Grade ist. Weiterhin stellt sich heraus, dass die erzeugte Gleichung zwar eine einfache Lösung besitzt (die vorgegebene), die zweite Lösung ist jedoch oft ein Bruch mit großem Nenner, und bei der Lösung der Gleichung treten große Zahlen auf.

[3] eingehüllt in eine Liste

Anhang C

Fazit

Die Arbeit am Repetitorium kann sicher noch beliebig weitergeführt werden. So könnten weitere Aufgabentypen aufgenommen werden. Eine Erweiterung des Stoffes auf Themen aus der Analysis wäre denkbar. In diesem Bereich gibt es allerdings schon ansprechende Tutorials, auch auf der Basis von Mathematica. Unter der Internetadresse [UIUC] findet sich ein Programm, welches schrittweise Funktionen differenziert. Auf ähnliche Weise wird dort die Berechnung unbestimmter Integrale vorgeführt, dieser Teil - und einige andere - ist allerdings gebührenpflichtig.

Die Verwendung von Mathematica für dieses Tutorial hat sich letztlich doch bewährt. Die Notebooks bieten genügend Komfort für den Benutzer, Dialoge lassen sich damit recht bequem erstellen, sobald die grundlegenden Aufgaben für Menüsteuerung und Dialoggestaltung gelöst sind. Positiv zu bemerken ist auch die Möglichkeit des schnellen Prototyping. Bequem ist das Testen von neuen Funktionen: bei laufendem Repetitorium können neue Funktionen interaktiv erstellt, getestet und ins Repetitoriumsprogramm eingestellt werden. Das reibungslose Zusammenspiel ist sofort erkennbar. Allerdings: die häufigen, aus heiterem Himmel stattfindenden Abstürze in der Testphase – meist im Zusammenhang mit Aktionen in den generierten Notebooks – konnten sehr lästig werden.

Die bereits an früherer Stelle erwähnten Schwierigkeiten mit der automatischen Evaluierung von Ausdrücken lassen sich letztlich doch gut in den Griff bekommen. Es ist nicht notwendig, auf direkte Programmierung in anderen Programmiersprachen zurückzugreifen.

Letztlich erweist sich der Funktionsumfang von Mathematica als so groß und reichhaltig, dass sich für jedes auftretende Problem eine Lösung finden ließ, und oft sogar eine recht elegante.

Literaturverzeichnis

[GU92] Charlie Gunn: Remarks on Mathematical Courseware. in: S. Cunningham, R. J. Hubbold [Eds.]: Interactive Learning Through Visualization. Berlin 1992, p. 115-127

[MA01] Emily Martin: Mathematica 4.1, Standard Add-on Packages. Cambridge 2001

[MA99] Kurt Marti, Detlef Groeger: Brückenkurs Mathematik. Wittenberg 1999

[TA92] Jonathan P. Taylor: Prospero, A System for Representing the Lazy Evaluation of Functions. in: S. Cunningham, R. J. Hubbold [Eds.]: Interactive Learning Through Visualization. Berlin 1992, p. 145-157

[WO99] Stephen Wolfram: The Mathematica Book. Cambridge 1999

[UIUC] http://mtl.math.uiuc.edu/classroom_resources.htm

In der Reihe FINAL sind bisher erschienen:

1. Jahrgang 1991:

1. Hinrich E. G. Bonin; Softwaretechnik, Heft 1, 1991 (ersetzt durch Heft 2, 1992).
2. Hinrich E. G. Bonin (Herausgeber); Konturen der Verwaltungsinformatik, Heft 2, 1991 (überarbeitet und erschienen im Wissenschaftsverlag, Bibliographisches Institut & F. A. Brockhaus AG, Mannheim 1992, ISBN 3-411-15671-6).

2. Jahrgang 1992:

1. Hinrich E. G. Bonin; Produktionshilfen zur Softwaretechnik --- Computer-Aided Software Engineering --- CASE, Materialien zum Seminar 1992, Heft 1, 1992.
2. Hinrich E. G. Bonin; Arbeitstechniken für die Softwareentwicklung, Heft 2, 1992 (3. überarbeitete Auflage Februar 1994), PDF-Format (Passwort: arbeiten).
3. Hinrich E. G. Bonin; Object-Orientedness --- a New Boxologie, Heft 3, 1992.
4. Hinrich E. G. Bonin; Objekt-orientierte Analyse, Entwurf und Programmierung, Materialien zum Seminar 1992, Heft 4, 1992.
5. Hinrich E. G. Bonin; Kooperative Produktion von Dokumenten, Materialien zum Seminar 1992, Heft 5, 1992.

3. Jahrgang 1993:

1. Hinrich E. G. Bonin; Systems Engineering in Public Administration, Proceedings IFIP TC8/ WG8.5: Governmental and Municipal Information Systems, March 3--5, 1993, Lüneburg, Heft 1, 1993 (überarbeitet und erschienen bei North-Holland, IFIP Transactions A-36, ISSN 0926-5473).
2. Antje Binder, Ralf Linhart, Jürgen Schultz, Frank Sperschneider, Thomas True, Bernd Willenbockel; COTEXT --- ein Prototyp für die kooperative Produktion von Dokumenten, 19. März 1993, Heft 2, 1993.
3. Gareth Harries; An Introduction to Artificial Intelligence, April 1993, Heft 3, 1993.
4. Jens Benecke, Jürgen Grothmann, Mark Hilmer, Manfred Hölzen, Heiko Köster, Peter Mattfeld, Andre Peters, Harald Weiss; ConFusion --- Das Produkt des AWÖ-Projektes 1992/93, 1. August 1993, Heft 4, 1993.
5. Hinrich E. G. Bonin; The Joy of Computer Science --- Skript zur Vorlesung EDV ---, September 1993, Heft 5, 1993 (4. ergänzte Auflage März 1995).
6. Hans-Joachim Blanke; UNIX to UNIX Copy --- Interactive application for installation and configuration of UUCP ---, Oktober 1993, Heft 6, 1993.

4. Jahrgang 1994:

1. Andre Peters, Harald Weiss; COMO 1.0 --- Programmierumgebung für die Sprache COBOL --- Benutzerhandbuch, Februar 1994, Heft 1, 1994.

2. Manfred Hölzen; UNIX-Mail --- Schnelleinstieg und Handbuch ---, März 1994, Heft 2, 1994.
3. Norbert Kröger, Roland Seen; EBrain --- Documentation of the 1994 AWÖ-Project Prototype ---, June 11, 1994, Heft 3, 1994.
4. Dirk Mayer, Rainer Saalfeld; ADLATUS --- Documentation of the 1994 AWÖ-Project Prototype -- -, July 26, 1994, Heft 4, 1994.
5. Ulrich Hoffmann; Datenverarbeitungssystem 1, September 1994, Heft 5, 1994. (2. überarbeitete Auflage Dezember 1994)
6. Karl Goede; EDV-gestützte Kommunikation und Hochschulorganisation, Oktober 1994, Heft 6 (Teil 1), 1994.
7. Ulrich Hoffmann; Zur Situation der Informatik, Oktober 1994, Heft 6 (Teil 2), 1994.

5. Jahrgang 1995:

1. Horst Meyer-Wachsmuth; Systemprogrammierung 1, Januar 1995, Heft 1, 1995.
2. Ulrich Hoffmann; Datenverarbeitungssystem 2, Februar 1995, Heft 2, 1995.
3. Michael Guder / Kersten Kalischefski / Jörg Meier / Ralf Stöver / Cheikh Zeine; OFFICE-LINK --- Das Produkt des AWÖ-Projektes 1994/95, März 1995, Heft 3, 1995.
4. Dieter Riebesehl; Lineare Optimierung und Operations Research, März 1995, Heft 4, 1995.
5. Jürgen Mattern / Mark Hilmer; Sicherheitsrahmen einer UTM-Anwendung, April 1995, Heft 5, 1995.
6. Hinrich E. G. Bonin; Publizieren im World-Wide Web --- HyperText Markup Language und die Kunst der Programmierung ---, Mai 1995, Heft 6, 1995
7. Dieter Riebesehl; Einführung in Grundlagen der theoretischen Informatik, Juli 1995, Heft 7, 1995
8. Jürgen Jacobs; Anwendungsprogrammierung mit Embedded-SQL, August 1995, Heft 8, 1995
9. Ulrich Hoffmann; Systemnahe Programmierung, September 1995, Heft 9, 1995 (ersetzt durch Heft 1, 1999).
10. Klaus Lindner; Neuere statistische Ergebnisse, Dezember 1995, Heft 10, 1995

6. Jahrgang 1996:

1. Jürgen Jacobs / Dieter Riebesehl; Computergestütztes Repetitorium der Elementarmathematik, Februar 1996, Heft 1, 1996
2. Hinrich E. G. Bonin; "Schlanker Staat" & Informatik, März 1996, Heft 2, 1996
3. Jürgen Jacobs; Datenmodellierung mit dem Entity-Relationship-Ansatz, Mai 1996, Heft 3, 1996
4. Ulrich Hoffmann; Systemnahe Programmierung, (2. überarbeitete Auflage von Heft 9, 1995), September 1996, Heft 4, 1996 (ersetzt durch Heft 1, 1999).
5. Dieter Riebesehl; Prolog und relationale Datenbanken als Grundlagen zur Implementierung einer NF2-Datenbank (Sommer 1995), November 1996, Heft 5, 1996

7. Jahrgang 1997:

1. Jan Binge, Hinrich E. G. Bonin, Volker Neumann, Ingo Stadtsholte, Jürgen Utz; Intranet-/Internet- Technologie für die Öffentliche Verwaltung --- Das AÖW-Projekt im WS96/97 --- (Anwendungen in der Öffentlichen Verwaltung), Februar 1997, Heft 1, 1997
2. Hinrich E. G. Bonin; Auswirkungen des Java-Konzeptes für Verwaltungen, FTVI'97, Oktober 1997, Heft 2, 1997

8. Jahrgang 1998:

1. Hinrich E. G. Bonin; Der Java-Coach, Oktober 1998, Heft 1, 1998 (CD-ROM, PDF-Format; aktuelle Fassung)
2. Hinrich E. G. Bonin (Hrsg.); Anwendungsentwicklung WS 1997/98 --- Programmierbeispiele in COBOL & Java mit Oracle, Dokumentation in HTML und tcl/tk, September 1998, Heft 2, 1998 (CD-ROM)
3. Hinrich E. G. Bonin (Hrsg); Anwendungsentwicklung SS 1998 --- Innovator, SNiFF+, Java, Tools, Oktober 1998, Heft 3, 1998 (CD-ROM)
4. Hinrich E. G. Bonin (Hrsg); Anwendungsentwicklung WS 1998 --- Innovator, SNiFF+, Java, Mail und andere Tools, November 1998, Heft 4, 1998 (CD-ROM)
5. Hinrich E. G. Bonin; Persistente Objekte --- Der Elchtest für ein Java-Programm, Dezember 1998, Heft 5, 1998 (CD-ROM)

9. Jahrgang 1999:

1. Ulrich Hoffmann; Systemnahe Programmierung (3. überarbeitete Auflage von Heft 9, 1995), Juli 1999, Heft 1, 1999 (CD-ROM und Papierform), Postscript-Format, zip-Postscript-Format, PDF- Format und zip-PDF-Format.

10. Jahrgang 2000:

1. Hinrich E. G. Bonin; Citizen Relationship Management, September 2000, Heft 1, 2000 (CD-ROM und Papierform), PDF-Format --- Password: arbeiten
2. Hinrich E. G. Bonin; WI>DATA --- Eine Einführung in die Wirtschaftsinformatik auf der Basis der Web_Technologie, September 2000, Heft 2, 2000 (CD-ROM und Papierform), PDF-Format --- Password: arbeiten
3. Ulrich Hoffmann; Angewandte Komplexitätstheorie, November 2000, Heft 3, 2000 (CD-ROM und Papierform), PDF-Format
4. Hinrich E. G. Bonin; Der kleine XMLer, Dezember 2000, Heft 4, 2000 (CD-ROM und Papierform), PDF-Format, aktuelle Fassung --- Password: arbeiten

11. Jahrgang 2001:

1. Hinrich E. G. Bonin (Hrsg.): 4. SAP-Anwenderforum der FHNON, März 2001, (CD-ROM und Papierform), Downloads & Videos.
2. J. Jacobs / G. Weinrich; Bonitätsklassifikation kleiner Unternehmen mit multivariater linear Diskriminanzanalyse und Neuronalen Netzen; Mai 2001, Heft 2, 2001, (CD-ROM und Papierform), PDF-Format und MS Word DOC-Format --- Password: arbeiten
3. K. Lindner; Simultanttestprozedur für globale Nullhypothesen bei beliebiger Abhängigkeitsstruktur der Einzeltests, September 2001, Heft 3, 2001 (CD-ROM und Papierform).

12. Jahrgang 2002:

1. Hinrich E. G. Bonin: Aspect-Oriented Software Development. März 2002, Heft 1, 2002 (CD-ROM und Papierform), PDF-Format --- Password: arbeiten.
2. Hinrich E. G. Bonin: WAP & WML --- Das Projekt Jagdzeit ---. April 2002, Heft 2, 2002 (CD-ROM und Papierform), PDF-Format --- Password: arbeiten.
3. Ulrich Hoffmann; Ausgewählte Kapitel der Theoretischen Informatik. August 2002, Heft 3, 2002 (CD-ROM und Papierform), PDF-Format ---

Herausgeber:

Prof. Dr. Dipl.-Ing. Dipl.-Wirtsch.-Ing. Hinrich E. G. Bonin Fachhochschule Nordostniedersachsen (FH NON), Volgershall 1, D-21339 Lüneburg, email: bonin@fhnon.de

Verlag:

Eigenverlag (Fotographische Vervielfältigung), FH NON

Erscheinungsweise:

ca. 4 Hefte pro Jahr Für unverlangt eingesendete Manuskripte wird nicht gehaftet. Sie sind aber willkommen.