In diesem Heft

CEBIT Spezial:

Liebe Leserin, lieber Leser

Endlich liegt die neue MathPAD vor. Es war aber nicht Bequemlichkeit, die uns von der Publikation des Heftes abgehalten hat. Denn daß die Mitglieder der MathPAD-Arbeitsgruppe inzwischen nicht auf der faulen Haut gelegen haben, geht aus dem Inhalt hervor: Wir beschäftigen uns fast ausschließlich mit unserer eigenen Entwicklung, dem Computeralgebrasystem MuPAD. Im vorliegenden Heft, welches als Begleitmaterial für unsere Ausstellung auf der CEBIT gedacht ist, berichten einige der MuPAD-Entwickler in leicht lesbarer Form über die Ziele, die sie beim Entwurf des Systems verfolgt haben. Wer sich für mehr Details von MuPAD interessiert, der sei auf das 450-seitige MuPAD-Handbuch, welches gleichzeitig bei Birkhäuser erscheint, verwiesen. Aus Anlaß der Publikation des Handbuchs haben wir auch erstmals eine Anzeige aufgenommen, gewissermaßen als Werbung in eigener Sache.
Wegen des eigenen Produkts ist der redaktionelle Teil diesmal auf ein Minimum geschrumpft, gerade zwei Beiträge. In der Hoffnung, daß uns die Leser soviel Begeisterung in eigener Sache vergeben werden, danken wir wieder ganz herzlich allen Autoren des vorliegenden Heftes.

B.F. & W.W.

Redaktion:
Benno Fuchssteiner, Waldemar Wiwianka
Universität - GH Paderborn
Postfach 1621, D - 4790 Paderborn
E-mail: benno@uni-paderborn.de
waldemar@uni-paderborn.de

ISSN 0941 – 9187
V.i.S.d.P.: B. Fuchssteiner
Auflage: 2000

Das Computeralgebra System – MuPAD

Benno Fuchssteiner, Waldemar Wiwianka

MuPAD ist ein Computeralgebrasystem; der Name ist eine Abkürzung für: **Mu**lti **P**rocessing **A**lgebra **D**ata **T**ool. MuPAD ist eine Entwicklung der Mathematikgruppe im *Institut für Automatisierung und Instrumentelle Mathematik* (AUTOMATH) der Universität-Gesamthochschule Paderborn. Die Entwicklung von MuPAD wurde von der Deutschen Forschungsgemeinschaft und dem Heinz-Nixdorf Institut gefördert. Forschungsinstitutionen und Ausbildungseinrichtungen steht MuPAD kostenlos zur Verfügung.

Naturwissenschaftler und Ingenieure haben bei ihrer täglichen Arbeit mit einem Übermaß an mathematischen Formeln zu tun. Der Umgang damit und die akribische Sorgfalt, die dieser Umgang erfordert, gehören keineswegs zu den Lieblingsbeschäftigungen dieser Berufsgruppen. Eine Tatsache, die durch die sehr hohen Abbruchraten bei Studenten dieser Disziplinen an deutschen Hochschulen belegt wird; Abbruchraten, die meist auf einen Mangel an Routine im effizienten Umgang mit mathematischen Sachverhalten und Formeln zurückgehen.

MuPAD ist ein Softwareprodukt, das den Umgang mit Formeln erlaubt und mit intelligentem mathematischen Expertenwissen erleichtert. Neben numerischem Rechnen erlaubt MuPAD nicht nur den interaktiven, formelmäßigen Umgang mit mathematischen Objekten, sondern ermöglicht auch die problemlose Handhabung von Formeln - und damit von technischen Sachverhalten - die bisher wegen ihrer Kompliziertheit dem Anwender unzugänglich waren. Im Unterschied zur numerischen Behandlung mathematischer Sachverhalte manipuliert Computeralgebra also auch Zeichen und Symbole. Eine ganz natürliche Sache, da für den Computer, wie auch für den Menschen, zum Beispiel der Umgang mit dem Symbol π leichter ist, als der Umgang mit der dadurch bezeichneten *unendlich* langen Dezimalzahl. Daß Computer mit Zahlen und numerischen Daten eindrucksvoll umgehen können, ist bekannt; in den Forschungs- und Entwicklungszentren ist dies häufig ihre Hauptaufgabe. Daß sie mit Formeln noch viel besser und leichter umgehen können, ist der Öffentlichkeit bisher weitgehend verborgen geblieben. Daß Computeralgebra möglich sein würde, ist schon lange bekannt, daß sie heute die praktischen Möglichkeiten vieler Anwender bereichert, und daß sie morgen auf dem *Notebook* von Studenten und Schülern verfügbar sein wird, ist dem Fortschritt in Hard- und Softwaretechnik zu verdanken.

Um einen anschaulichen Einblick in die Möglichkeiten solcher Systeme zu geben, hier einige ganz einfache Beispiele:

- *Zur Berechnung der 800-stelligen Zahl* 341! *braucht man nur den Befehl* fact(341) *und etwa 33 Millisekunden Rechenzeit.*
- *Die Zerlegung der 34-stelligen Zahl (31!+1) in ihre vier Faktoren braucht weniger als fünf Sekunden und als Befehl* ifactor(fact(31)+1)*.*
- *Durch Eingabe eines halbzeiligen Befehls erhält man die explizite Lösung vieler der in der Praxis vorkommenden Gleichungen. Hat man noch etwas Geduld, so kann man gleich einen Plot der Lösung in druckfertiger Form mitnehmen.*

Über diese einfachen Beispiele hinaus erlaubt Computeralgebra einfachste Programmierung selbst anspruchsvollster mathematischer und formelmäßiger Sachverhalte. Gravierende Auswirkungen auf die mathematische Lehre an Universitäten, technischen Hochschulen und Fachhochschulen werden die Folge der Verbreitung von Computeralgebrasystemen sein.

Während die meisten kommerziellen Computeralgebrasysteme in Nordamerika entwickelt wurden, ist MuPAD das erste europäische universelle Computeralgebrasystem.

MuPAD hat eine außerordentlich komfortable Bedieneroberfläche, die selbst mathematischen Laien mit geringer Rechnererfahrung das spielerische Erlernen der Beherrschung des Systems ermöglicht. Ein interaktives digitales Handbuch, eine Systemkomponente zur interaktiven Fehlersuche (Debugger), sowie ein leistungsfähiges menügesteuertes Graphikmodul runden das System ab. Ein ca. 450-seitiges Handbuch wird vom Birkhäuser Verlag in Basel publiziert.

Im Gegensatz zu herkömmlichen Systemen erlaubt MuPAD die nutzergesteuerte Manipulation und Veränderung der eigenen Routinen; damit ist ein erster Schritt hin zu einem lernfähigen System getan.

ü ber Spezialaufgaben hinaus hat MuPAD die effiziente Erledigung allgemeiner mathematischer Aufgaben zum Ziel. Um Aufgaben und Probleme von ganz neuer Dimension Lösen zu könnenen, bietet MuPAD neben der Mögliglichkeit des sequentiellen Arbeitens, Versionen, die auf parallelen Rechnerarchitekturen aufbauen. Die grobe Parallelitätt von MuPAD basiert auf dem Modell der Shared-Memory-Maschine. MuPAD wurde jedoch auch für andere Architekturen entworfen, für die dann über spezielle Interfaces Shared-Memory-Maschinen emuliert werden.
MuPAD ist modular aufgebaut und um sein zentrales Element MAMMUT (Memory Allocation and Management Unit) gruppiert. MAMMUT ist leicht portierbar und hat neben der eindeutigen Datenhaltung und der Speicherverwaltung auch das Management der groben Parallelität zur Aufgabe. MAMMUT verleiht MuPAD die Schnelligkeit, die es anderen Systemen überlegen macht. Bei der feinkörnigennParallelität hat der Nutzer zusätzlich die Möglichkeitt, die Kommunikation gemäß der Architektur der zugrundeliegenden Rechnerplattform zu steuern.
Gegenwärtig steht MuPAD für UNIX Arbeitsplatzrechner und Rechner der Familie APPLE-Macintosh zur Verfügung; eine Portierung auf andere Rechnerplattformen wird durchgeführt.

Grafische Benutzerschnittstellen zum Computeralgebra-System MuPAD

Karsten Morisse

Um mit dem Computeralgebra-System **MuPAD** arbeiten zu können, benötigt man kein besonderes Ein- und Ausgabegerät. Es genügt eine Tastatur und ein ASCII-Terminal. Um jedoch den interaktiven Umgang mit **MuPAD** komfortabler zu gestalten, gibt es **XMuPAD**, eine grafische Benutzerschnittstelle für **MuPAD** unter X-Windows und OpenWindows. Darüber hinaus steht dem Benutzer mit dem Hypertext-System **HyTEX** die vollständige **MuPAD**-Dokumentation am Bildschirm zur Verfügung.

Bei der Entwicklung der Benutzerschnittstellen wurde Wert darauf gelegt, daß die für das Window-Systeme typischen Eigenschaften beibehalten wurden. Dazu zählen z.B. die Angabe von Optionen beim Aufruf, beispielsweise zur Festlegung der Fenstergröße oder Fensterpositionierung und auch der „Drag and Drop“-Mechanismus. Für die Beschreibung dieser Eigenschaften sei der Leser jedoch auf die Dokumentation von OpenWindows und X-Windows verwiesen.

In diesem Beitrag wird die grafische Schnittstelle beschrieben. Dabei wird neben dem Aussehen von und dem Umgang mit **XMuPAD** und **HyTEX** auch kurz auf die technische Realisierung eingegangen. Alle Angaben über zu verwendende Tasten, Maustasten etc. beziehen sich dabei auf die Sun SPARCstation.

XMuPAD

XMuPAD selbst ist ein völlig eigenständiges Programm. Es hat keinerlei Kenntnis von Computeralgebra. Es ist als Schnittstelle zum **MuPAD**-Kern mit Hilfe der XView-Library implementiert. Aufgerufen wird **XMuPAD** durch das Kommando

```
xmupad [options]
```

Die einzige für den Benutzer wichtige Option ist die `-L` Option. Damit kann er die Sprache der Online-Dokumentation festlegen. Als mögliche Parameter stehen `german` und `english` zur Verfügung.

Nach dem Aufruf erscheint auf dem Bildschirm ein *Basisfenster* (siehe Abbildung 1), welches aus zwei Komponenten besteht:

- eine Reihe von Buttons und
- ein Textfenster für die Ein- und Ausgabe.

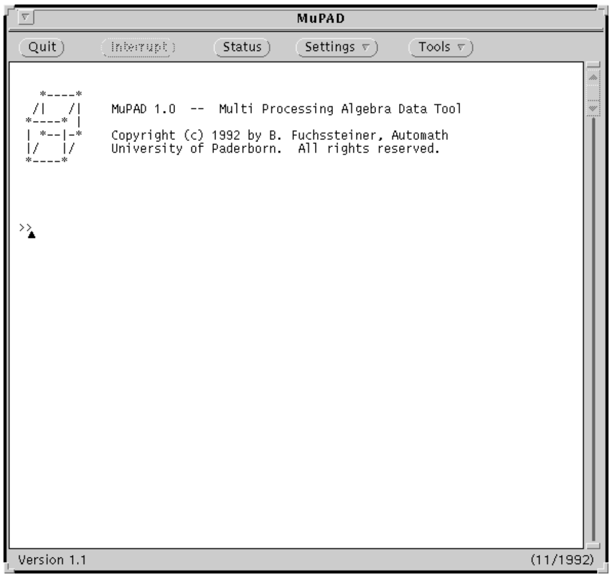


Abbildung 1: **XMuPAD**-Basisfenster

Weiterhin wird der Kern von **MuPAD** als Sohnprozeß auf UNIX-Ebene gestartet.

Die Arbeitsweise von **XMuPAD** läßt sich in drei, für ein interaktives System typische Phasen aufteilen:

1. In der *Eingabephase* gibt der Benutzer Kommandos in das Textfenster ein. Mit der Taste `<Return>` beendet man diese Phase.
2. In der *Evaluierungsphase* wird die Eingabe der ersten Phase ausgewertet.
3. Die dritte und letzte Phase ist die Ausgabe des von **MuPAD** bestimmten Ergebnisses.

Das Textfenster

Das Textfenster ist ein gewöhnliches Fenster zur Eingabe von Text, wie es beispielsweise auch der Editor `textedit` unter OpenWindows zur Verfügung stellt. An der rechten Seite ist eine Scrollbar angebracht, mit der man den sichtbaren Textausschnitt verschieben kann.

Eingabe

Die Eingabe von Kommandos erfolgt über die Tastatur an der aktuellen Einfügeposition, die durch den Cursor ▲ sichtbar gemacht ist. Diese Einfügeposition kann mit Hilfe der Maus (Maus-Cursor auf gewünschte Position und linke Maustaste drücken) oder den Cursor-Tasten ◄, ►, ▲ und ▼ (im rechten Ziffernblock) auf jede beliebige Position innerhalb des Textfensters versetzt werden.
Beendet wird die Eingabe eines Kommandos durch die Taste `<Return>`. Daraufhin wird der aktuelle Befehl gelesen und als Kommando an **MuPAD** zur Ausführung geschickt. In der Regel wird sich ein Befehl nur über eine Bildschirmzeile erstrecken. Es ist jedoch auch möglich, Kommandos über mehrere Zeilen verteilt einzugeben, ohne daß jede Zeile einzeln vom Parser interpretiert wird. Wie dies geschieht, ist weiter unten beschrieben.
Die Ausgabe des Ergebnisses erfolgt in dem Textfenster direkt unterhalb der Zeile, die die Ausgabe bewirkt hat. An die Ausgabe schließt sich das Prompt-Zeichen von **MuPAD** (">>") an. Die aktuelle Einfügeposition ist dann das Zeichen nach dem Prompt.
Nachdem das Kommando an **MuPAD** übergeben wurde, befindet sich **XMuPAD** in der Evaluierungsphase. Während dieser Zeit ist das Textfenster inaktiviert, und es werden keine Eingaben angenommen. Optisch wird dies durch eine kleine Uhr als Maus-Cursor verdeutlicht.

Besonderheiten

Gäbe es keine Besonderheiten für die Eingabe von Kommandos, so wäre **XMuPAD** überflüssig, man könnte dann auch ein normales Terminalfenster nehmen.

`<Linefeed>` Um Kommandos strukturiert über mehrere Zeilen eingeben zu können, kann man eine Zeile mit `<Linefeed>` beenden. Dadurch wird die aktuelle Zeile mit einem \ abgeschlossen, nachfolgend wird eine neue Zeile eingefügt, und die aktuelle Einfügeposition wandert an den Beginn dieser Zeile[1]. Das soeben eingegebene Kommando wird jedoch noch nicht an **MuPAD** zur Auswertung übergeben, also auch noch nicht ausgeführt. Auf diese Weise kann ein beliebig langes **MuPAD**-Kommando über beliebig viele Zeilen eingegeben werden. Zur Ausführung wird das Kommando durch Drücken von `<Return>` gebracht. Dabei ist es unwichtig, an welcher Stelle im Kommando die aktuelle Einfügeposition liegt. Es werden sowohl alle Zeilen oberhalb der Einfügeposition als auch alle Zeilen unterhalb zum Kommando gezählt. In anderen Applikationen hat das Drücken von `<Return>` für gewöhnlich einen Zeilenumbruch an der aktuellen Einfügeposition zur Folge. In **XMuPAD** erfolgt der Umbruch der Zeile jedoch erst am Ende der Eingabe.

Erneute Ausführung alter Eingaben Durch Versetzen der aktuellen Einfügeposition in eine zuvor gemachte Eingabe und erneutes Drücken von `<Return>` kann eine alte Eingabe noch einmal als Kommando ausgeführt werden. Dabei darf die Einfügeposition an beliebiger Stelle im Kommando sein, auch wenn die Eingabe über mehrere Zeilen ging.
Ist im Settings Pulldown Menü der Replace Mode auf `OFF` gesetzt, so bleibt die alte Ausgabe erhalten und die neue Ausgabe wird vor der alten eingefügt. Ist Replace Mode gleich `ON`, so wird die alte Ausgabe durch die neue überschrieben. In diesem Fall erscheint keine neue Eingabaufforderung durch **MuPAD**, sondern die aktuelle Einfügeposition wandert an das Ende der ersten Zeile des nächsten Kommandos. Ist Recalculate Mode auf `ALL` gesetzt, so wird nicht nur das Kommando an der aktuellen Einfügeposition erneut ausgeführt, sondern auch alle Kommandos bis zum Ende der kompletten **MuPAD**-Sitzung. Dabei ist die Reihenfolge der Kommandoausführung durch die textuelle Reihenfolge innerhalb des Textfensters festgelegt und nicht durch die Reihenfolge der Eingabe. Damit ist es auf sehr einfache Weise möglich, eine interaktiv eingegebene Folge von Anweisungen, gegebenenfalls mit unterschiedlichen Anfangswerten, auszuführen.

Einfügen neuer Eingaben zwischen alten Eingaben Wie oben beschrieben, kann die Eingabe eines Kommandos an jeder beliebigen Stelle im Textfenster erfolgen. Es ist also möglich, ein neues Kommando zwischen der Ausgabe eines alten Kommandos und der Eingabe des Nachfolgekommandos einzugeben. Dadurch wird die Reihenfolge aller eingegebe-

[1] Das Setzen des \ am Zeilenende ist notwendig, damit **XMuPAD** erkennt, was zu einer Eingabezeile gehört

nen Kommandos beeinflußt, die ja für die Realisierung des Recalculate Modus von Bedeutung ist.

Copy & Paste Das Textfenster bietet auch eine besondere Möglichkeit des *Copy & Paste*. Die mit `<Paste>` im Textfenster abgelegten Eingabezeilen werden jeweils mit einem \ abgeschlossen, jedoch noch nicht automatisch ausgeführt. Der Benutzer muß dies explizit durch Drücken von `<Return>` machen. Er hat dadurch aber die Möglichkeit, den Text noch abzuändern. Der Auswahlvorgang geschieht, so wie man es gewöhnt ist, mit der linken und mittleren Maustaste.

Buttons im Basisfenster

Die Buttons im Basisfenster sind horizontal angeordnet (siehe Abbildung 1). Es gibt Zeitpunkte, zu denen einige Buttons deaktiviert sind, d.h. ein Anklicken dieser Buttons bewirkt nichts. Optisch wird dies verdeutlicht, indem beim Button die ansonsten schwarze Umrandung und Beschriftung in hellgrau überwechseln. Im folgenden werden die einzelnen Buttons näher beschrieben.

Quit-Button

Der Quit-Button ist selbsterklärend, er beendet **XMuPAD**. Dabei werden alle zu **XMuPAD** gehörigen Fenster geschlossen, und der **MuPAD**-Prozeß wird ebenfalls beendet. Dieser Button ist zu jedem Zeitpunkt aktiv.

Interrupt-Button

Mit diesem Button wird an den **MuPAD**-Prozeß ein Signal geschickt, die gerade durchgeführte Berechnung zu terminieren. Dieser Button ist nur während der Evaluierungsphase aktiv. Während der Eingabe ist er deaktiviert.

Status-Button

Der Status-Button öffnet ein Popup-Fenster (siehe Abbildung 2), welches Statusinformationen über **MuPAD** liefert. Die Informationen entsprechen der Ausgabe der **MuPAD**-Befehle `bytes()` bzw. `time()`. Der Status-Button ist nur während der Eingabephase aktiviert.

Settings-Button

Durch Anklicken dieses Buttons wird ein Pulldown-Menü geöffnet (siehe Abbildung 3), mit dessen Hilfe

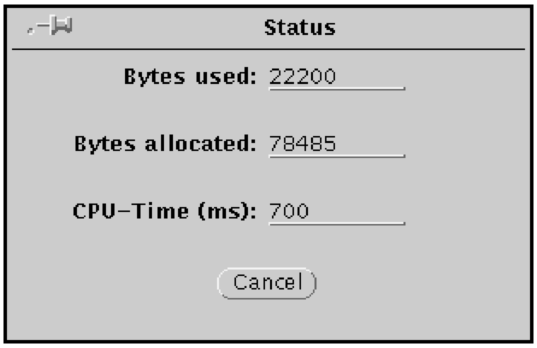


Abbildung 2: Status Popup-Fenster

der Benutzer die Möglichkeit hat, verschiedene Funktionen auszuführen und die Ein- und Ausgabefunktionalität modifizieren kann. Das Pulldown-Menü besitzt den OpenWindows typischen Pushpin, mit dessen Hilfe das Menü auf dem Bildschirm „festgeheftet" werden kann. Der Settings-Button ist ebenso wie alle Menü-Items nur während der Eingabephase aktiv.

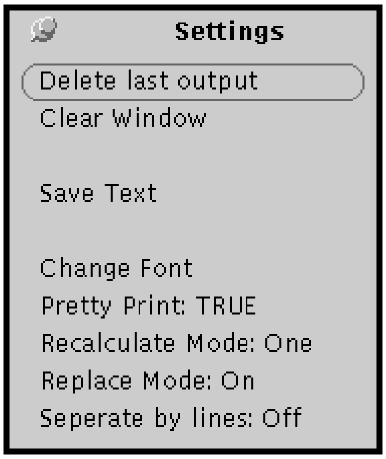


Abbildung 3: Settings Pulldown-Menü

Delete last output Dieses Item löscht die Ausgabe des letzten Befehls. Als erstes Item in dem Pulldown-Menü ist dies auch die Default-Aktion. Drückt man also den Settings-Button, ohne das Menü aufzuziehen, so wird automatisch diese Aktion ausgeführt.

Clear Window Dieses Item löscht den Inhalt des Textfensters. Nach Aktivierung steht als einziger Text das **MuPAD**-Prompt ">>" im Textfenster.

Save Text Der Menüpunkt Save Text öffnet ein weiteres Popup-Fenster, das Save Popup-Fenster (siehe Abbildung 4). Das obere Eingabefeld dient zur Eingabe eines Verzeichnisses. In das untere wird der eigentliche Dateiname eingegeben. Der unter „Directory“ eingetragene Name braucht nicht mit / beendet zu werden. Es wird automatisch zwischen die beiden Namen eingefügt. Drückt man im „Directory“-Eingabefeld `<Return>`, so springt der Eingabecursor in das „File“-Eingabefeld. Mit dem OK-Button wird der Inhalt des Textfensters in der spezifizierten Datei abgespeichert. Existiert eine Datei mit diesem Namen, so wird vor dem Überschreiben der Datei eine Sicherung der alten Datei vorgenommen. Dazu wird an den Namen der alten Datei die Endung % angehängt. Zwischen zwei Aufrufen des Status Popup-Fensters bleiben die eingetragenen Namen erhalten.

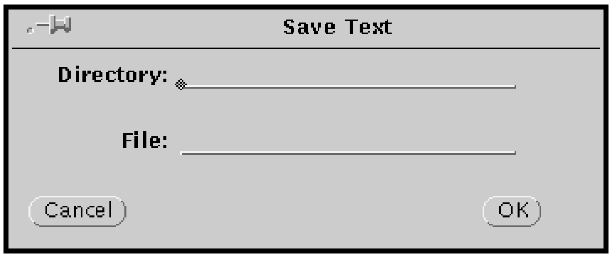


Abbildung 4: Save Popup-Fenster

Change Font Mit dem Menüpunkt Change Font wird ein weiteres Popup-Fenster geöffnet (siehe Abbildung 5). Darin kann der Benutzer den Zeichensatz

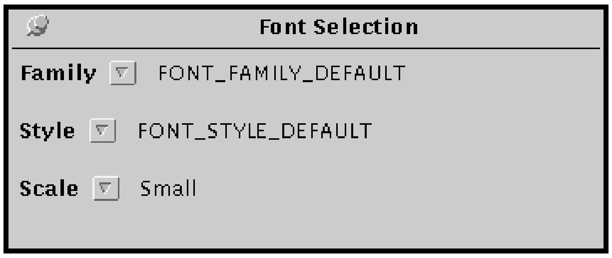


Abbildung 5: Fontselection-Fenster

für das Eingabefenster verändern. Er kann dabei die Font-Familie den Font-Stil und die Font-Größe festlegen.

Mit Hilfe der folgenden Menüeinträge können verschiedene Modi für die Ausgabe im Textfenster eingestellt werden. Der aktuelle Zustand wird im Menü angezeigt.

Pretty Print Mit Hilfe dieses Kommandos kann die formatierte Ausgabe, der „Pretty-Printer“, ein- und ausgeschaltet werden.

Recalculate Mode In **XMuPAD** ist es möglich, eine einmal gemachte Eingabe erneut auszuführen. Dazu wird die alte Eingabe mit dem Maus-Cursor angeklickt und durch Drücken von `<Return>` ausgeführt. Ist Recalculate Mode auf `ONE` gesetzt, so wird nur die ausgewählte Eingabe erneut ausgewertet. Ist Recalculate Mode auf `ALL` gesetzt, so wird nicht nur dieses Kommando erneut ausgeführt, sondern auch alle innerhalb des Eingabefensters textuell nachfolgenden Befehle.

Replace Mode Mit diesem Menü-Item wird der Replace Modus ein- und ausgeschaltet. Auf Seite 5 ist die Wirkungsweise dieses Modus beschrieben.

Seperate by lines Mit diesem Menü-Item hat der Benutzer die Möglichkeit, eine Ausgabe von der nächsten Eingabe optisch zu trennen. Dies geschieht durch eine Linie über die gesamte Fensterbreite.

Tools-Button

Der Tools-Button öffnet das Tools Pulldown-Menü (siehe Abbildung 6). Dieses Menü enthält Einträge zum Starten anderer Programme. Derzeitig sind die Anbindung an den **MuPAD**-Debugger, an das Help-System und an das Graphiktool **VCam** implementiert. Wählt man Help, so wird die Online-Dokumentation aktiviert. Diese enthält sowohl das vollständige **MuPAD**-Benutzerhandbuch als auch Help-Seiten zu allen **MuPAD**-Befehlen (siehe auch die Beschreibung von **HyTEX** ab Seite 8). Wird der Debugger ausgewählt, so wird der **MuPAD**-Debugger gestartet. Bislang ist es nicht möglich, auf die bis zu diesem Zeitpunkt an **MuPAD** gemachten Eingaben zurückzugreifen, da der Debugger mit einem neuen **MuPAD**-Prozeß gestartet wird. Der Tools-Button ist nur während der Eingabephase aktiviert.

Sonderfälle – Abweichungen von MuPAD

In einigen wenigen Punkten unterscheidet sich die Funktionalität von **XMuPAD** von der von **MuPAD**.

Abbildung 6: Tools Pulldown-Menü

`system()` und `!` – UNIX Shell Kommando Mit der Funktion `system` (Kurzform `!`) ist die Eingabe von Befehlen an die UNIX-Shell in **MuPAD** möglich. Die Ausführung des `system()`-Befehls erfordert jedoch eine gesonderte Behandlung, da es z.B. nicht möglich ist, den Editor `vi` innerhalb eines Textfensters zu Starten. In **XMuPAD** wird in diesem Fall ein neues Terminalfenster geöffnet, in dem die Default-Shell des Benutzers gestartet wird. Der Parameter des `system()`-Befehls wird dann in dieser Shell ausgeführt.
In diesem Fenster gibt es einen Button, den Done Button. Durch ihn kann man das Fenster verschwinden lassen. Dabei bleibt der Inhalt des Terminalfensters jedoch erhalten. Durch ein erneutes `system()`-Kommando wird das Terminalfenster mit dem alten Inhalt wieder sichtbar.
Die Ausführung des `system()`-Befehls ist in **XMuPAD** nicht ganz unproblematisch, da die XView-Library keinerlei Kontrollmechanismen über die in dem Terminalfenster ausgeführten Prozesse anbietet. Aus diesem Grund liefert in **XMuPAD** das `system()`-Kommando innerhalb des Textfensters kein Ergebnis zurück. Der Benutzer erhält also keine Informationen darüber, ob das abgesetzte Kommando auch erfolgreich ausgeführt wurde oder ob die Ausführung bereits beendet ist. In einer interaktiven Sitzung kann der Benutzer dies natürlich innerhalb des Terminalfensters sehen.
Weiterhin ist der Befehl `system()` in **XMuPAD** auch nicht blockierend. D.h. wird `system()` innerhalb einer Anweisungssequenz aufgerufen, so erfolgt die Ausführung das nachfolgenden Befehls nicht erst dann, wenn die Ausführung des Shell-Kommandos beendet ist, sondern die beiden Kommandos werden „gleichzeitig“ als zwei Prozesse bearbeitet (ein Kommando wird von der Shell, das andere von **MuPAD** ausgeführt).

User Interrupt In **MuPAD** hat der Benutzer die Möglichkeit mittels `<Ctrl C>` eine laufende Evaluierung zu unterbrechen. Es erscheint dann ein Menü, in dem er wählen kann zwischen

- abort – Abbruch der laufenden Berechnung
- continue – Fortführung der laufenden Berechnung
- quit – Beenden der **MuPAD** Sitzung

In **XMuPAD** kann der Benutzer durch Anklicken des Interrupt-Button ein Interrupt generieren. Er hat dann jedoch nicht die Möglichkeit in einem Menü den weiteren Verlauf der Berechnung zu bestimmen, sondern die laufende Berechnung wird sofort abgebrochen. Dies ist jedoch kein schwerwiegender Nachteil, da zwei der drei möglichen Menüpunkte durch die Buttons Quit und Interrupt abgedeckt werden.

Ausführung von `textinput()` Mit der Funktion `textinput()` kann der Benutzer während eines Programmablaufes interaktiv Text eingeben und diesen dann beispielsweise an einen Bezeichner zuweisen. Auch dieser Befehl erfordert in **XMuPAD** eine gesonderte Behandlung. In diesem Fall wird ein weiteres Fenster geöffnet, das dem Benutzer einen vollständigen, mausorientierten Texteditor bereitstellt. Nach der Eingabe des gewünschten Textes und dem Anklicken des Done Buttons wird der Text an den **MuPAD**-Kern übergeben.

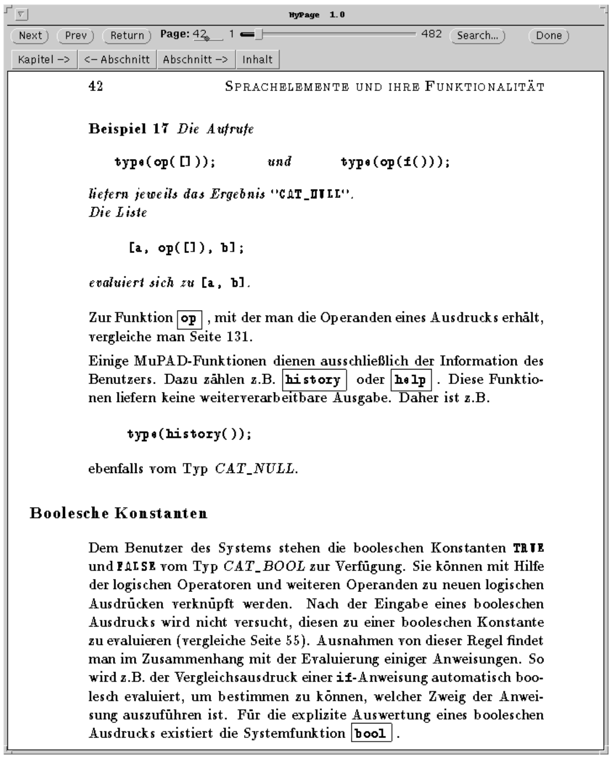
HyTeX

HyTeX ist ein Hypertext-System, welches an der Universität-GH-Paderborn, in der Arbeitsgruppe von N. Köckler, entwickelt wurde. In Verbindung mit **XMuPAD** wird eine auf unsere Bedürfnisse angepaßte und funktional eingeschränkte Version dieses Programmes verwendet.
Wie man es dem Namen **HyTeX** direkt entnehmen kann, basiert das System auf dem TeX- bzw. LaTeX-System von D.E. Knuth. Neben einem speziellen LaTeX-Previewer stellt es einige neue Befehle zur Verfügung. Die Dokumenterstellung ist also für einen TeXniker überhaupt kein Problem.
HyTeX wird für die vollständige Online-Dokumentation verwendet. Diese umfaßt zum einen das **MuPAD**-Benutzerhandbuch und zum anderen Help-Seiten für jeden **MuPAD**-Befehl.
HyTeX wird zusammen mit **XMuPAD** als Sohnprozeß desselbigen gestartet, ist jedoch zunächst nur als Icon auf dem Bildschirm zu sehen. Erst auf die explizite Anfrage des Benutzers hin wird ein neues Fenster geöffnet. Dies kann der Nutzer auf verschiedene Weisen machen. Einerseits kann er durch

Auswahl von Help im Tools Menü (siehe Seite 7) das Inhaltsverzeichnis des **MuPAD**-Handbuches aufschlagen. Eine zweite Möglichkeit bietet die Help-Funktion in **MuPAD**. Mit ihrer Hilfe wird die Help-Seite über ein vom Benutzer angegebenes Kommando aufgeschlagen.

HyTEX-Fenster

Das **HyTEX**-Fenster (siehe Abbildung 7) enthält einen LaTEX-Previewer, der mit Hypertext-Funktionen ausgestattet ist. Er bietet dem Anwender verschiedene Navigationsmöglichkeiten durch das Dokument. Diese werden im folgenden beschrieben.



42 SPRACHELEMENTE UND IHRE FUNKTIONALITÄT

Beispiel 17 *Die Aufrufe*

`type(op([]));` *und* `type(op(f()));`

liefern jeweils das Ergebnis "CAT_NULL".
Die Liste

`[a, op([]), b];`

evaluiert sich zu `[a, b]`.

Zur Funktion `op`, mit der man die Operanden eines Ausdrucks erhält, vergleiche man Seite 131.

Einige MuPAD-Funktionen dienen ausschließlich der Information des Benutzers. Dazu zählen z.B. `history` oder `help`. Diese Funktionen liefern keine weiterverarbeitbare Ausgabe. Daher ist z.B.

`type(history());`

ebenfalls vom Typ *CAT_NULL*.

Boolesche Konstanten

Dem Benutzer des Systems stehen die booleschen Konstanten `TRUE` und `FALSE` vom Typ *CAT_BOOL* zur Verfügung. Sie können mit Hilfe der logischen Operatoren und weiteren Operanden zu neuen logischen Ausdrücken verknüpft werden. Nach der Eingabe eines booleschen Ausdrucks wird nicht versucht, diesen zu einer booleschen Konstante zu evaluieren (vergleiche Seite 55). Ausnahmen von dieser Regel findet man im Zusammenhang mit der Evaluierung einiger Anweisungen. So wird z.B. der Vergleichsausdruck einer `if`-Anweisung automatisch boolesch evaluiert, um bestimmen zu können, welcher Zweig der Anweisung auszuführen ist. Für die explizite Auswertung eines booleschen Ausdrucks existiert die Systemfunktion `bool`.

Abbildung 7: **HyTEX**-Fenster

Statische Buttons

Zu dieser Art von Buttons gehören diejenigen, die unabhängig von der gerade dargestellten Seite immer im **HyTEX**-Fenster zu sehen sind. Sie sind alle in der obersten Kontrollzeile des Fensters untergebracht.

Next- und Prev-Button Mit Hilfe dieser beiden Buttons kann man seitenweise im Dokument blättern. Next blättert eine Seite vorwärts, Prev ein Seite zurück.

Return-Button Dieser Button ist im Zusammenhang mit den dynamischen Buttons von Bedeutung. Er wird daher erst später beschrieben.

Page-Panel und -Slider Das Page-Panel dient zur Angabe einer Seite, die man direkt anspringen möchte. Anstatt hier per Tastatur direkt eine Seitenzahl einzugeben, kann man die gewünschte Seite auch mittels des Sliders einstellen.

Search-Button Dieser Button, man errät es leicht, dient zum Suchen innerhalb des Dokumentes. Nach seiner Aktivierung erscheint das Search-Menü (siehe Abbildung 8), in dem der Benutzer ein Textmuster eingeben kann, auf welches hin der gesamte Text durchsucht wird. In dem Menü kann der Benutzer

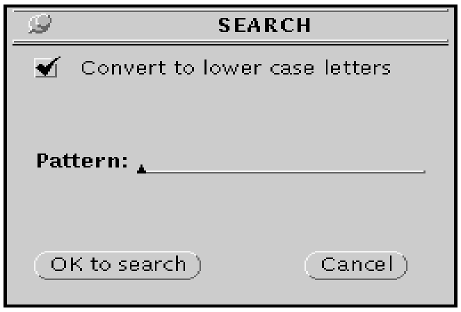


Abbildung 8: Search-Menü

angeben, ob zwischen Groß- und Kleinschreibung unterschieden werden soll.
Der Suchmechanismus ist noch nicht vollständig perfektioniert. So werden z.B. im Dokument getrennt vorkommende Wörter nicht gefunden.

Done-Button Klickt der Nutzer diesen Button an, so verschwindet das **HyTEX**-Fenster. Es verschwindet jedoch nicht völlig, sondern kann vom Benutzer mittels der Help-Funktion oder des Help-Buttons in **XMuPAD** wieder zum Vorschein gebracht werden.

Dynamische Buttons

Der Autor eines Textes hat die Möglichkeit, sein Dokument mit sogenannten Knoten und Verweisen auf diese Knoten zu versehen. Diese Knoten kann man sich als interne Marken im Text vorstellen, sie sind im Text selbst nicht sichtbar. Zu sehen sind jedoch die Verweise auf Knoten. Sie sind im **HyTEX**-Fenster als Rechtecke oder Boxen zu erkennen. Klickt man solch eine Box an, so wird automatisch zu dem entsprechenden Knoten gesprungen. Auf diese Weise hat der

Nutzer also die Möglichkeit, mittels Querverweise, gesteuert durch seine persönlichen Interessen, durch das Dokument zu „hüpfen“.
In der **MuPAD**-Dokumentation gibt es zweierlei Arten solcher *dynamischer Buttons*. Einerseits gibt es sie im Kontrollfeld über dem eigentlichen Textfenster. Mit diesen Buttons sind Sprünge zum nächsten und zum vorhergehenden Kapitel, Abschnitt u.a. möglich. Diese Buttons können sich von Textseite zu Textseite ändern. Andererseits gibt es sie im Text, wo beispielsweise Erläuterungen von Begriffen als Querverweise realisiert sind.
Hat der Benutzer solch einen Sprung ausgeführt, so möchte er natürlich, insbesondere bei den Sprüngen der zweiten Art, schnell wieder auf die Ausgangsseite zurückkehren. Zu diesem Zweck dient der Return-Button. Bei einem Sprung durch einen dynamischen Button auf eine andere Seite, wird die verlassene Seite intern auf einem Stack abgelegt. Ein Anklicken von Return nimmt nun das oberste Element von diesem Stack und springt auf die entsprechende Seite im Dokument zurück.

Help-Seiten

Wie bereits oben erwähnt, ist neben dem **MuPAD**-Benutzerhandbuch auch das Online-Help-System in **HyTEX** integriert. Beispielsweise kann der Nutzer in **MuPAD** bzw. **XMuPAD** Informationen über die Systemfunktion `igcd()` durch den Befehl

```
help("igcd");
```

oder

?igcd;

anfordern. In **HyTEX** wird dann die entsprechende Help-Seite aufgeschlagen (siehe Abbildung 9). Auf solch einer Help-Seite ist die exakte Syntax eines Befehls beschrieben. Es wird erläutert, wie der Befehl wirkt, einige Beispiele zu seiner Anwendung sind angegeben, und Sprünge zu Help-Seiten verwandter Funktionen bzw. Befehle sind möglich. Dabei ist jede Beispieleingabe wieder mit einer Box versehen, die ebenfalls maussensitiv ist, d.h. bei ihrem Anklicken passiert etwas. Im Unterschied zu den dynamischen Buttons im Text wird hier jedoch kein Querverweis vollzogen, sondern die Eingabe des Beispiels wird an **XMuPAD** übergeben und in das Textfenster von **XMuPAD** geschrieben. Der Benutzer kann sich die Eingabe nun noch einmal in Ruhe ansehen, eventuell auch modifizieren und dann per `<Return>` ausführen. Durch dieses „Learning by doing“ wird dem Anwender schnell einer Kenntnis der zu verwendenden Syntax vermittelt.

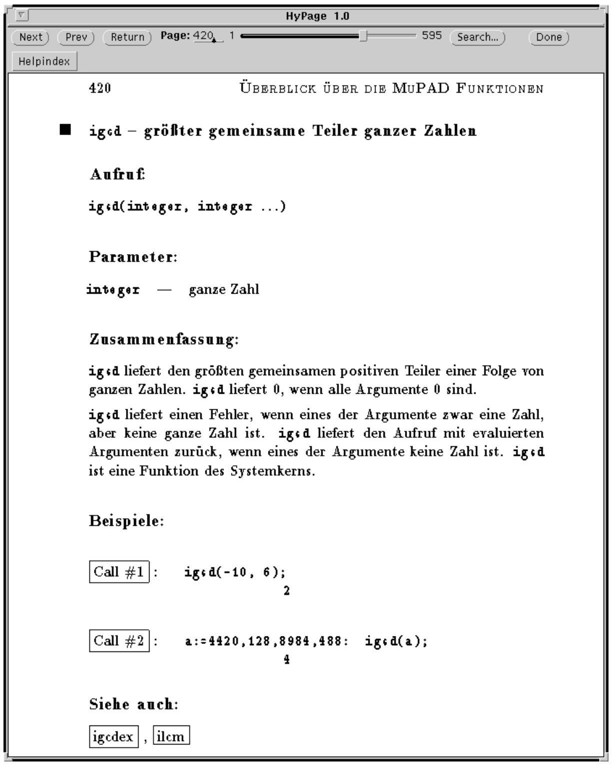


Abbildung 9: Help-Seite zu `igcd`

Implementationsdetails

XMuPAD ist mit Hilfe der XView-Library implementiert. Diese ist im Prinzip ein großer Baukasten mit verschiedenen Fenstertypen, Buttons, Menüs etc. sowie zahlreiche Funktionen zur Kontrolle und Modifikation dieser Objekte.
XMuPAD ist eine reine Schnittstelle für den Benutzer zum Computeralgebra-System **MuPAD**. Als eine der ersten Anweisungen führt **XMuPAD** einen Aufruf der Funktion `fork()` aus und startet **MuPAD** als neuen Prozeß. Die Ein- und Ausgabe dieses Prozesses werden auf einen Kommunikationskanal dupliziert, über den **XMuPAD** und **MuPAD** sich dann unterhalten. Der prinzipielle Ablauf ist in der nachfolgenden Abbildung dargestellt.
Die Online-Dokumentation **HyTEX** wird ebenfalls als Sohnprozeß durch **MuPAD** gestartet. Der Kommunikationsablauf funktioniert analog zur Verbindung **XMuPAD** und **MuPAD**: Die Ein- und Ausgabe von **HyTEX** sind auf einen Kommunikationskanal dupliziert, über den sich beide Programme unterhalten. Dabei werden an **HyTEX** die internen Marken übergeben, die es auf Anforderung des Benutzers hin anzuspringen hat.

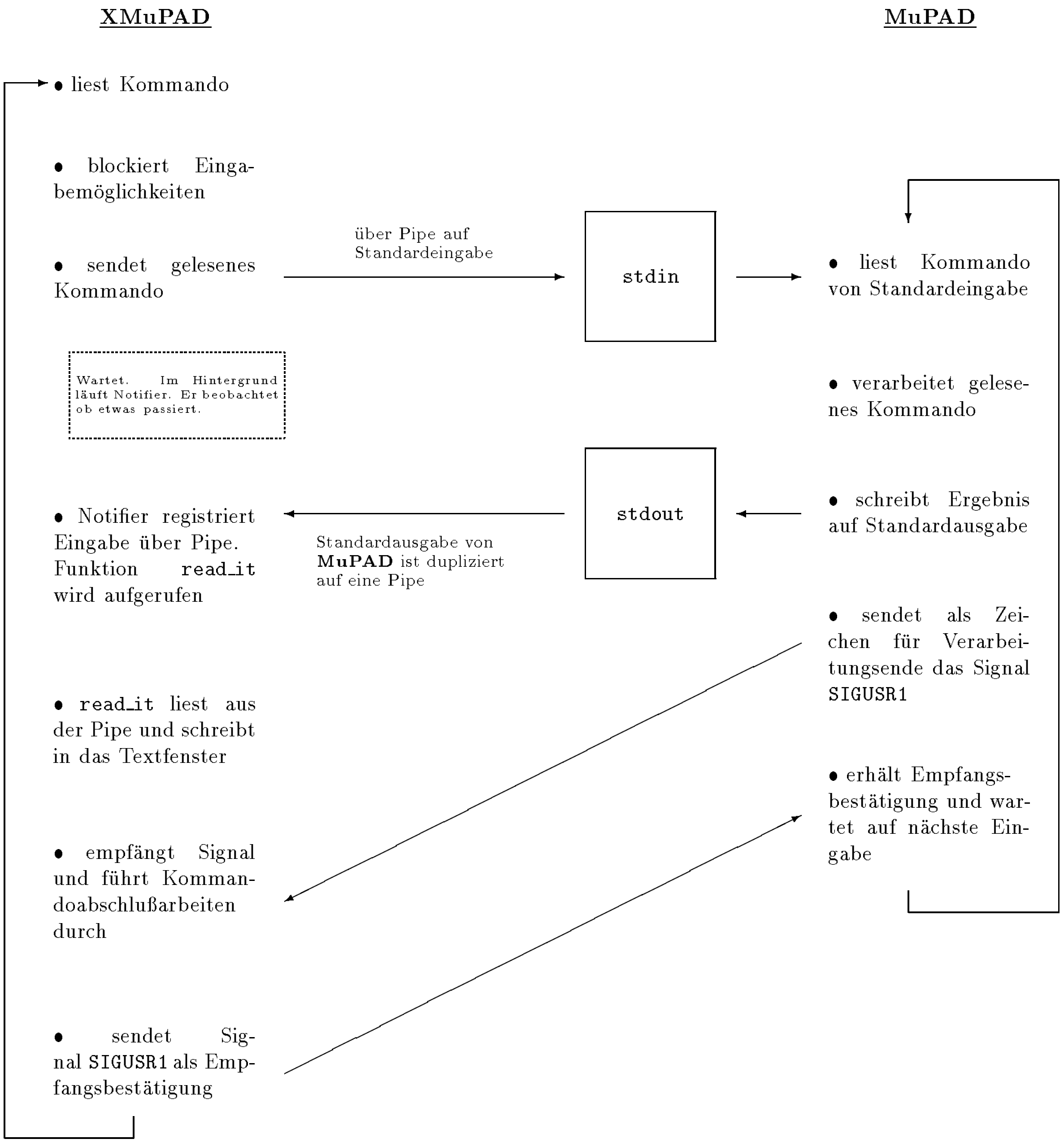


Abbildung 10: Kommunikation zwischen **XMuPAD** und **MuPAD**

MuPAD – Grafik

Thorsten Schulze

In diesem Artikel wollen wir das **MuPAD**-Grafiktool **VCam**— Virtual Camera — anhand einer kleinen Beispielsitzung erläutern. Zur Zeit existiert leider nur auf der Sun, wo sie mit Hilfe von **XView** implementiert wurde. Das Grafiktool wird aus dem Programm **XMuPAD**heraus aktiviert, indem man dort einen der **MuPAD**-Befehle `plot3d();` oder `plot2d();` eingibt. Diese Befehle können sowohl mit einer Reihe von Argumenten als auch ohne Argumente abgesetzt werden, wobei die letztere Alternative gerade für unerfahrene Benutzer gedacht ist. Bestätigt man dann den gerade eingegebenen Befehl `plot3d();` mit <Return>, so erscheint das sogenannte **Basisfenster** des Grafiktools.

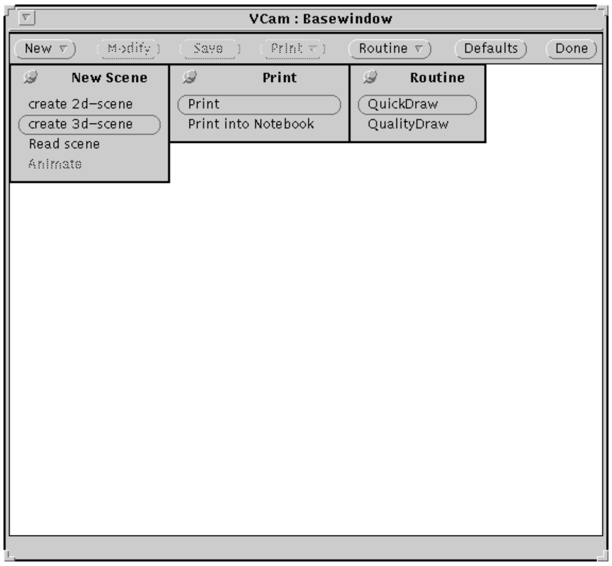


Abbildung 1: Basisfenster des Grafiktools

Dieses enthält die eigentliche Zeichenfläche zur Darstellung der Grafiken und eine Leiste mit Kontrollfunktionen zur Steuerung des Grafiktools, wie

- New:
  Menü zum Erzeugen neuer Grafiken.
- Modify:
  zum Modifizieren bestehender Grafiken.
- Save:
  zum Speichern bestehender Grafiken.
- Print:
  zum Ausdrucken von Grafiken.
- Routine:
  zur Auswahl von unterschiedlichen Zeichenroutinen.
- Defaults:
  zur Veränderung der Default-werte.
- Done:
  um den Grafikprozeß zu beenden.

Das aber wollen wir jetzt nicht, denn wir möchten eine Grafik erzeugen. Dieses geschieht, wie oben schon angedeutet, mit Hilfe des New-Menüs. Wählt man dort den Punkt create 3d-scene, so erscheint ein weiteres Fenster, das sogenannte **Grafikmanipulationsfenster**.

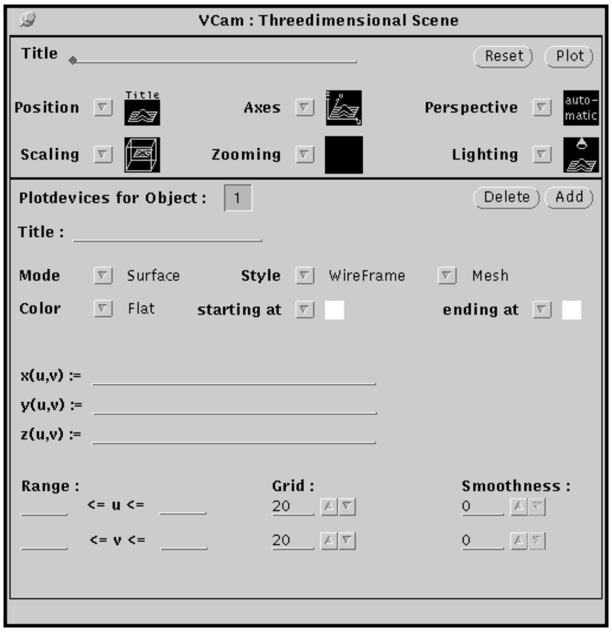


Abbildung 2: Grafikmanipulationsfenster einer 3D-Szene

Die Unterteilung dieses Fensters ergibt sich direkt aus dem zugrundeliegenden Benutzermodell. Und zwar ist die Erzeugung von Grafiken an den Prozeß des Fotografierens angelehnt. Zum besseren Verständnis stelle man sich vor, man wolle ein Foto mit einem bestimmten Informationsgehalt anfertigen. Dazu werden **Objekte** mit speziellen Eigenschaften zu einer **Szene** mit weiteren für alle Objekte geltenden Eigenschaften gruppiert. Genau dieses Benutzermodell reflektiert das **Grafikmanipulationsfenster**, d.h. im oberen Teil dieses Fensters werden die Szene-Eigenschaften angezeigt und im unteren Teil die zum jeweiligen Objekt gehörenden Spezifikationen. Um nun eine Grafik zu erzeugen, muß man zunächst den Modus des Objektes festlegen. Dieses geschieht, indem man mittels des Mode-Menüs die gewünschte Option wählt. Um beispielsweise ein Raumkurve darzustellen, wählt man die Option Curve. Nun braucht man nur noch die Parametrisierungen und die Parameterbereiche festzulegen. Dazu gibt man hinter den Textitems x(u), y(u), z(u) und $\leq$ u $\leq$ die entsprechenden Werte ein. Die Grafik wird dann gezeichnet, wenn man auf den Plot-Button im oberen Teil des Fensters drückt. Die Aktivierung des Plot-Buttons generiert nun einen neuen plot3d()-Befehl, der die im **Grafikmanipulationsfenster** angezeigten Werte als Argumente erhält. Dieser plot3d()-Befehl wird nun an das zugrundeliegende Computeralgebra System **MuPAD**gesendet und dort ausgewertet. Die evaluierten Daten werden an das Grafiktool zurückgeschickt und dort verarbeitet. Die Grafik erscheint dann im **Basisfenster** des Grafiktools.

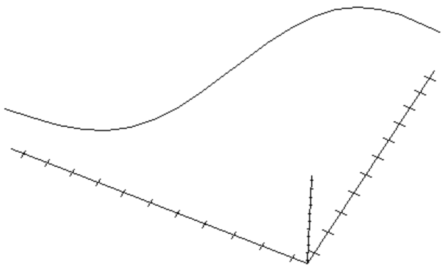


Abbildung 3: Beispielgrafik

Hier muß zunächst noch einmal ausdrücklich darauf hingewiesen werden, daß es in **MuPAD** im Gegensatz zu anderen Computeralgebra Systemen möglich ist, vom Grafiktool aus Grafikbefehle als neue **MuPAD**-Prozesse zu starten. Kommen wir nun zu den Veränderungen, die der Benutzer vornehmen kann. Im Bereich der Szene-Eigenschaften können folgende Optionen verändert bzw. neu eingegeben werden:

Title:
Jeder Szene kann ein Titel zugeordnet werden.

Position:
Die Position des Szenetitels kann anhand der Icons ober- bzw. unterhalb der Grafik gewählt werden.

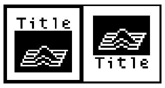


Abbildung 4: Icons zur Auswahl der Titelposition

Axes:
Wir stellen insgesamt vier verschiedene Achsenstile zur Verfügung. Diese sind anhand der Icons eindeutig wählbar.

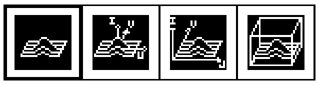


Abbildung 5: Icons zur Auswahl des Achsenstils

Scaling:
Mit Hilfe dieser Icons

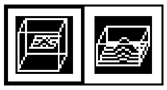


Abbildung 6: Icons zur Auswahl der Skalierungsroutine

kann aus zwei verschiedenen Skalierungroutinen gewählt werden. Im ersten Fall werden die grafischen Daten so skaliert, daß sich Verzerrungen nur aus den Perspektivtransformationen ergeben, d.h. beispielsweise erscheinen Kugeln als Kugeln und nicht als Ellipsoide. Im zweiten Fall werden die Daten so skaliert, daß der Canvas, also die eigentliche Zeichenfläche optimal ausgefüllt wird.

Zooming:
Mit Hilfe dieses Menüs stellen wir dem Benutzer

verschiedene Möglichkeiten zur Verfügung, das sichtbare Volumen — auch Viewing Box genannt — zu verändern. Wird der zweite Icon **Zooming in** gewählt, so wird die Viewing Box in allen Richtungen um einen vordefinierten Wert verkleinert. Der dritte Icon vergrößert dann die Viewing Box in allen Richtungen um einen vordefinierten Wert. Wird hingegen der vierte Icon ausgewählt, so erscheint das sogenannte **Zoom-Fenster**, welches später erläutert wird.

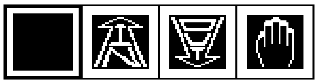


Abbildung 7: Icons des Zooming - Menüs

Perspective:
Dieses Menü dient dazu, von automatischer Perspektive auf manuelle Perspektive umzuschalten, was durch die Icons angedeutet werden soll. In diesem Fall wird dann das **Perspective-Fenster** aktiviert, aber dazu später mehr.

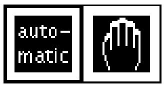


Abbildung 8: Icons des Perspective-Menüs

Lighting:
Mit Hilfe dieses Menüs soll die Eingabe eines Beleuchtungsmodells möglich gemacht werden. Dieses ist allerdings im jetzigen Entwicklungsstadium noch nicht implementiert.

Weitere Funktionen können mittels der zwei Buttons im oberen Teil des Fensters aufgerufen werden:

Reset:
Mit Hilfe dieses Buttons werden die Einträge des letzten Plot-Befehls wiederhergestellt, bzw. wenn es noch keinen gab, so werden die Default-Werte eingetragen.

Plot:
Dieser Button dient dazu, wie oben schon erwähnt, die aktuellen Grafikoptionen in einen neuen `plot3d()`-Befehl einzutragen, der dann ausgewertet werden kann. Die Grafik wird dann in das bereits existierende Basisfenster gezeichnet.

Von den im unteren Teil des Fensters liegenden Funktionen werden hier nur die wichtigsten erläutern. Zu allen anderen Funktionen gibt es weitere Informationen, die man entweder im **MuPAD**-Handbuch oder mittels des Help-Systems von **XMuPAD** erhalten kann. Angezeigt werden im unteren Teil des **Grafikmanipulationsfensters** jeweils die Spezifikationen des aktuellen Objektes. Mittels der Add bzw. Delete-Buttons können weitere Objekte zu einer bereits existierenden Szene hinzugefügt bzw. aus dieser Szene gelöscht werden. Dabei kann eine Szene zur Zeit maximal sechs Objekte enthalten. Weitere wichtige Optionen sind:

Mode:
der Modus des Objektes. Vorgesehen sind hier die vier Möglichkeiten

Curve:
Bei dem Objekt handelt es sich um eine Raumkurve.

Contour:
Es handelt sich um ein dreidimensionales Objekt, dessen Kontourlinien gezeichnet werden sollen. Diese Option ist in der aktuellen Version noch nicht verfügbar.

Surface:
Bei dem Objekt handelt es sich um eine Oberfläche.

List:
Hier handelt es sich um eine vom Benutzer anzugebende Liste von Punkten. Auch diese Option ist in der aktuellen Version noch nicht verfügbar.

Die nun folgenden Optionen, wie auch weitere Optionen sind jeweils vom Modus des Objektes abhängig.

Parametrisierungen:
Die Parametrisierungen der Objekte sind jeweils hinter den Textitems anzugeben. Dabei unterscheiden wir die Fälle:

Mode = Curve:
Da es sich hier um eine Raumkurve handelt, sind drei Parametrisierungen in Abhängigkeit von einer Variablen einzugeben. Um dieses zu verdeutlichen, heißen diese Textitems x(u), y(u) und z(u). Diese drei Parametrisierungen definieren dann die x-, y- bzw. z-Koordinaten.

Mode = Surface:
Bei diesem Objekt handelt es sich um eine

Oberfläche. Einzugeben sind in diesem Fall drei Parametrisierungen in Abhängigkeit von zwei Variablen, was durch die Benennung x(u,v), y(u,v) und z(u,v) angedeutet wird.

Range:
Hier müssen die Parameterbereiche eingeben werden. Dabei ist bei Mode = Curve nur die Angabe eines Bereiches und bei Mode = Surface die Angabe von zwei Parameterbereichen notwendig.

Alle weiteren Optionen, wie beispielsweise die Auswahl von unterschiedlichen Plot-Stilen oder Farbfunktionen sind optional. Hinweisen möchten wir hier nur auf einige Details, die das Grafiktool von anderen Computeralgebra Systemen unterscheiden. Im **MuPAD**-Grafiktool hat der Benutzer zusätzliche Plot-Stile zur Verfügung. Diese sind bei

Mode = Curve der Plot-Stil Impulses, d.h. es werden die Abszissen der Funktionswerte gezeichnet

und bei

Mode = Surface der Plot-Stil DepthCueing, mit dem man transparente Objekte simulieren kann.

Ferner haben wir bei Oberflächen die Möglichkeit vorgesehen, daß der Benutzer entscheiden kann, welche Parameterlinien gezeichnet werden sollen. Zudem können mittels des Smoothness-Wertes zusätzliche Stützstellen eingefügt werden, ohne daß die zugehörigen Parameterlinien gezeichnet werden. Dieses zeigt der folgende Beispielbefehl:

```
plot3d(CameraPoint = [0, 1, 30],
       Axes = NoAxes,
       [Mode = Surface,
             [sin(u)*cos(v)+1,
              sin(u)*sin(v),
              0],
             u = [0, PI/2],
             v = [-PI, PI],
             Grid = [10, 10],
             Smoothness = [0, 0]
       ],
       [Mode = Surface,
             [sin(u)*cos(v)-1,
              sin(u)*sin(v),
              0],
             u = [0, PI/2],
             v = [-PI ,PI],
             Grid = [10, 10],
             Smoothness = [0,2]
       ]);
```

Gibt man diesen Befehl in **MuPAD**ein, so wird die folgende Grafik erzeugt:

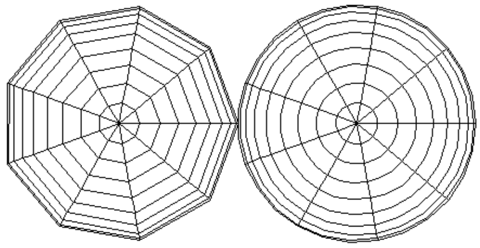


Abbildung 9: Demonstration von Smoothness

Dargestellt werden zwei Kreisscheiben, die beide mit derselben Anzahl an Stützstellen erzeugt wurden, wohingegen die Werte für Smoothness unterschiedlich gewählt wurden. Wollte man eine ähnliche Grafik wie die rechte ohne die Funktionalität von Smoothness erzeugen, so müßte man im obigen Fall die Anzahl der Stützstellen erhöhen, was aber dazu führen würde, daß weitere Parameterlinien gezeichnet würden. Der Befehl, der ein solches Objekt erzeugen würde, lautet:

```
plot3d(CameraPoint = [0, 1, 30],
       Axes = NoAxes,
       [Mode = Surface,
             [sin(u)*cos(v)+1,
              sin(u)*sin(v),
              0],
             u = [0, PI/2],
             v = [-PI, PI],
             Grid = [10, 28],
             Smoothness = [0, 0]
       ],
       [Mode = Surface,
             [sin(u)*cos(v)-1,
              sin(u)*sin(v),
              0],
             u = [0, PI/2],
             v = [-PI, PI],
             Grid = [10, 10],
             Smoothness = [0, 2]
       ]);
```

Kommen wir nun zu den oben bereits erwähnten weiteren Popupfenstern, die durch das Betätigen von einigen Buttons aktiviert werden, d.h. auf dem Bildschirm erscheinen.

Das **Zoom-Fenster** wird durch die Auswahl des vierten Icons im Menü Zooming aktiviert.

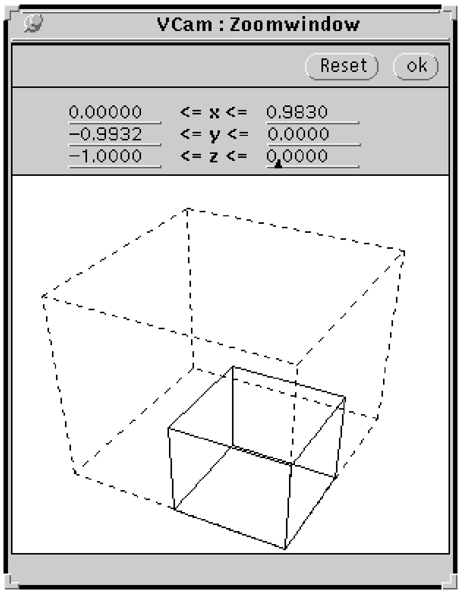


Abbildung 10: Zoom-Fenster

Mit Hilfe dieses Fensters kann der Benutzer die Viewing Box verändern. Dieses Fenster enthält zunächst einmal den üblichen Button Reset zur Wiederherstellung der ursprünglichen Werte und ok um die geänderten Werte zu übernehmen. Ferner gibt es einen Bereich, in dem die aktuellen Werte der Viewing Box angezeigt werden und einen Canvas, in dem die Viewing Box grafisch dargestellt wird. Veränderungen an der Viewing Box kann man dann folgendermaßen vornehmen:
Zum einem, indem man die expliziten Werte im mittleren Bereich des Fensters einträgt und diese Eingaben mit <Return> bestätigt. Dann wird die Viewing Box mit diesen neuen Werten gezeichnet. Zusätzlich zeichnen wir auch noch die alte Viewing Box, um dem Benutzer einen besseren Überblick über die Wirkung seiner Änderungen zu geben.
Eine weitere Möglichkeit, Änderungen vorzunehmen, besteht darin, die im Canvas dargestellte Viewing Box mit Hilfe der Maustasten zu verändern. Dazu muß man zunächst eine der Kanten der Viewing Box aktivieren, indem man auf sie klickt. Die aktivierte Kante wird dadurch gekennzeichnet, daß ihre Endpunkte mit kleinen Quadraten versehen werden. Von diesen Endpunkten sollte man einen auswählen, indem man auf ihn klickt. Die Viewing Box kann nun verändert werden, indem man die Maus verschiebt. Um eine andere Kante zu aktivieren, muß zunächst die aktivierte Kante deaktiviert werden. Dieses geschieht mit Hilfe der rechten Maustaste.

Das **Perspective-Fenster** wird aktiviert, indem man im **Grafikmanipulationsfenster** von automatischer Perspektive auf manuelle Perspektive umschaltet. Es erscheint das folgende Popupfenster:

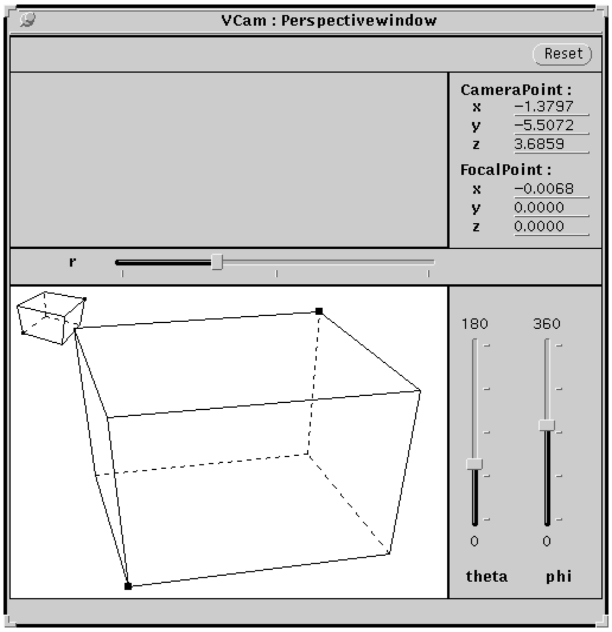


Abbildung 11: Perspective-Fenster

Dieses Fenster enthält wieder den üblichen Button Reset. Ferner enthält es

- einen Bereich zur Eingabe der expliziten Perspektivparameter in Objektraumkoordinaten,
- einen Bereich zur Eingabe der zugehörigen Perspektivparameter in Polarkoordinaten und
- zwei Canvases, von denen im unteren die Viewing Box in der gerade aktuellen Perspektive dargestellt ist.

Auch hier hat man zwei Möglichkeiten, die Perspektivparameter zu verändern:
Zum einen kann man die expliziten Werte für die Perspektivparameter, also für den CameraPoint und den FocalPoint angeben. Wieder muß diese Eingabe mit <Return> bestätigt werden. Dann wird die Viewing Box in dieser neuen Perspektive dargestellt. Um dem

Benutzer einen besseren Überblick über die Wirkung seiner Änderungen zu geben, zeigen wir zusätzlich die Viewing Box in der vorherigen Perspektive in der linken oberen Ecke des Canvases.
Natürlich können die Perspektivparameter auch mittels der Angabe der Polarkoordinaten verändert werden. Dazu stehen die drei Slider r, theta und phi zur Verfügung. Wieder wird die Viewing Box sowohl in der neuen als auch in der zuvor gültigen Perspektive gezeichnet.

Das **Save-Fenster** wird durch die Betätigung des Save-Buttons aus dem **Basisfenster** aufgerufen. Es dient dazu, bestehende Grafiken in verschiedenen Formaten zu speichern.

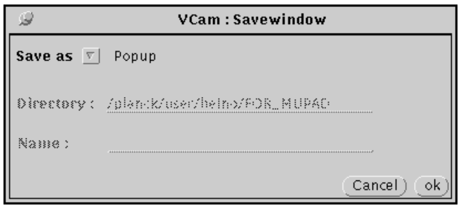


Abbildung 12: Save - Popupfenster

Die Buttons dieses Fensters dienen zur Auswahl des gewünschten Formats und zur Steuerung des Speichervorgangs, ferner existieren zwei Textitems zur Eingabe von Pfad- und Dateinamen. Im einzelnen haben wir

Save as:
Dieser Button dient zur Auswahl des zu speichernden Formats. Es sind bis jetzt drei Optionen vorgesehen:

Popup:
Die aktuelle Grafik des **Basisfensters** wird in einem weiteren Popupfenster gespeichert. Dem Benutzer wird hiermit die Möglichkeit gegeben, andere Grafiken mit dieser zu vergleichen. Mit Hilfe des Replot-Buttons kann er dann diese Grafik erneut berechnen lassen.

Raster-File:
Die im **Basisfenster** sichtbare Grafik wird als Raster-File gespeichert. In diesem Fall werden die beiden Textitems Directory und Name aktiviert. Hier muß der Benutzer den Pfad- und Dateinamen angeben.

Gif-File:
Die Grafik aus dem **Basisfenster** wird im Gif-Format gespeichert. Wieder sind zunächst der Pfad und der Name der Datei anzugeben.

Cancel:
der Speichervorgang wird abgebrochen.

ok:
Mit Hilfe dieses Buttons werden die ausgewählten Optionen bestätigt und die Grafik wird gespeichert.

Mit Hilfe des **Default-Fensters** kann der Benutzer einige der vorgegebenen Default-Werte verändern. Dieses Popupfenster wird aktiviert, indem der Defaults-Button im **Basisfenster** gedrückt wird.

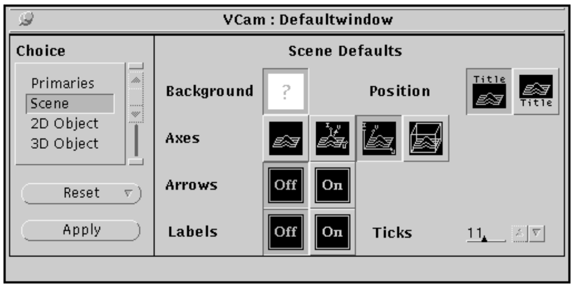


Abbildung 13: Default-Fenster

Dieses Popupfenster ist in zwei Bereiche unterteilt, und zwar in

einen Bereich mit einer Reihe von Kontrollfunktionen, in dem der Benutzer entscheiden kann, welche Default-Werte verändert werden sollen, und

einen Bereich, in dem die aktuellen Default-Werte angezeigt werden und die Veränderungen vereinbart werden können.

Der Bereich der Kontrollfunktionen setzt sich zusammen aus den Buttons und Menüs:

Choice:
Diese Liste enthält die übergeordneten Begriffe, nach denen die Default-Werte zusammengefaßt sind. Der aktuelle Wert ist dabei mit einem Rechteck umgeben.

Reset:
Dieses Menü dient zur Zurücknahme von Änderungen. Dabei kann man auswählen, ob nur die Änderungen zurückgenommen werden, die im unter Choice gewählten Bereich liegen, oder ob alle Änderungen zurückgenommen werden sollen.

Apply:
Mit Hilfe dieses Buttons werden die Änderungen bestätigt und stehen somit dem Benutzer zur Verfügung.

Der Bereich , in dem die expliziten Änderungen vorgenommen werden können, ist natürlich abhängig von dem unter Choice gewählten Begriff. In der obigen Abbildung sieht man beispielsweise die aktuellen Default-Werte für eine **Szene**. Diese Werte lassen sich dann leicht mit Hilfe der Maustasten verändern. Für alle weiteren Werte von Choice existieren ähnliche Teilfenster.

Zum Schluß möchten wir noch zwei Beispiele vorstellen, an denen man sehr schön die verschiedenen Grafik-Möglichkeiten sehen kann.

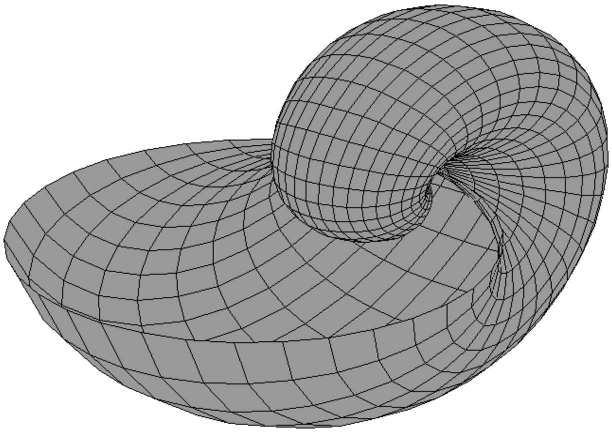


Abbildung 14: Beispiel einer Grafik

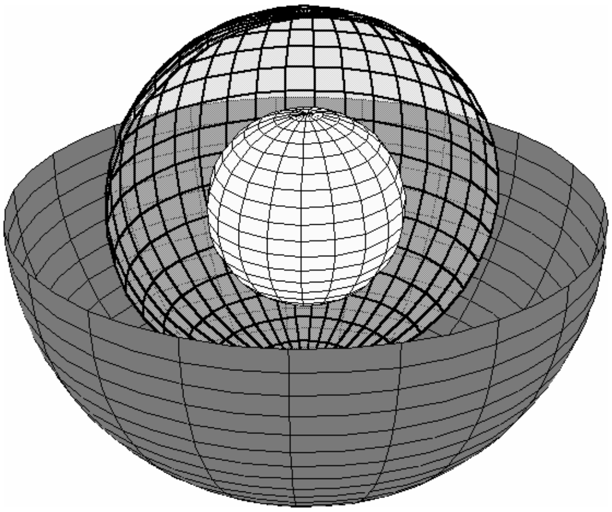


Abbildung 15: Beispiel einer Grafik

Abschließend sollte noch erwähnt werden, daß sich das Grafiktool in der ersten Entwicklungsphase befindet, was sich beispielsweise darin äußert, daß einige der hier erläuterten Optionen in dieser frühen Version noch nicht verfügbar sind. Deshalb möchten wir noch kurz auf die Erweiterungen hinweisen, die für das Grafiktool geplant sind:

- Ausweitung der dreidimensionalen Grafiken um Kontourmodelle und Punktlisten.
- Ausweitung der Farbfunktionalität.
- Implementation des Beleuchtungsmodells.
- Einarbeitung von Animationsroutinen.

Debugging in MuPAD

Oliver Kluge

1 Der MuPAD-Debugger

MuPAD enthält zwei Möglichkeiten, den Programmablauf im Detail zu verfolgen. Der **Trace**-Modus protokolliert den Ablauf eines Programms. Der Benutzer kann in diesem Modus keinen Einfluß auf die Programmausführung nehmen. Der **interaktive, zeilenorientierte Quelltext-Debugger** bietet dagegen folgende Möglichkeiten:

- Benutzerkontrollierbarer Programmablauf. (schrittweise oder durch Angabe von Haltepunkten)
- Bestimmung der Zeile im Quelltext, in der ein Laufzeitfehler aufgetreten ist.
- Anzeigen und Verändern von Variablenwerten.

Diese, im **MuPAD**-Systemkern integrierten, Debug-Mechanismen besitzen eine zeichen-/zeilenorientierte Benutzerschnittstelle und bieten selbst auf einem reinen Textausgabegerät (ASCII-Terminal) eine komfortable Bedienung und liefern sehr detaillierte Informationen in Textform. Eine auf OpenWindows basierende Schnittstelle, die einen noch höheren Komfort bietet, wird durch das vom **MuPAD**-Systemkern abgetrennte Programm *mdx* angeboten (siehe Abschnitt 3).

Der Debugger wird durch Optionen im **MuPAD**-Aufruf aktiviert. Hierzu kann das **MuPAD**-Programm mit drei verschiedenen Optionen aufgerufen werden. Alle drei Optionen können auch gleichzeitig benutzt werden (siehe Aufrufsyntax).

Ein Programm, das unter der Kontrolle des Benutzers ausgeführt (Debug-Modus) oder protokolliert werden soll (Trace-Modus), muß als Textdatei, die ein syntaktisch korrektes **MuPAD**-Programm enthält, vorliegen. Diese Textdatei wird wie gewohnt im **MuPAD**-Aufruf spezifiziert oder mittels der Funktion *read* eingelesen. Der Aufruf einer Prozedur, die innerhalb dieser Datei definiert wurde, gibt nun detaillierte Informationen bzgl. seines Programmablaufs aus.

Unter UNIX kann der Debugger mit Hilfe des folgenden Kommandos aktiviert werden.

mupad [–g] [–t] [–v] [weitere Optionen]

–t : Trace-Modus einschalten.

Während eines Programmlaufs werden die Zeilennummern im Quelltext der ausgeführten Anweisungen ausgegeben. Zudem werden die Stellen im Quelltext, aus der eine (benutzerdefinierte) Funktion aufgerufen wurde, und die Übergabeparameter an die Funktion angezeigt. Außerdem wird die Stelle, in der eine (benutzerdefinierte) Funktion verlassen wird, und der Rückgabewert der Funktion ausgegeben.

–v : Verbose-Modus einschalten.

Während der Ausführung einer `read`-Anweisung, also während des Einlesens einer **MuPAD**-Datei, werden die Zeilennummern des Beginns einer Prozedurdefinition angezeigt. Informationen über mögliche Stellen im Quelltext zum Setzen von Haltemarken werden bei der Ausgabe von benutzerdefinierten Funktionen in Form von # *line* # angezeigt.

–g : Debug-Modus einschalten.

In diesem Modus kann der Benutzer interaktiv per Tastatur Befehle zur Steuerung des Debuggers eingeben. Einen Überblick über die Steuerungsbefehle gibt folgende Tabelle.

Funktionen	Kommandos
Auflisten aktiver Prozeduren	(d)own, (u)p, (w)here
Anzeigen/Verändern von Variablen	(D)isplay, (p)rint, (e)xecute
Haltemarken setzen/Löschen	(g)oto proc, (S)top at, (C)lear, clear (a)ll
Steuerung des Programmablaufs	(c)ont, (n)ext, (q)uit, (s)tep, CTRL-C

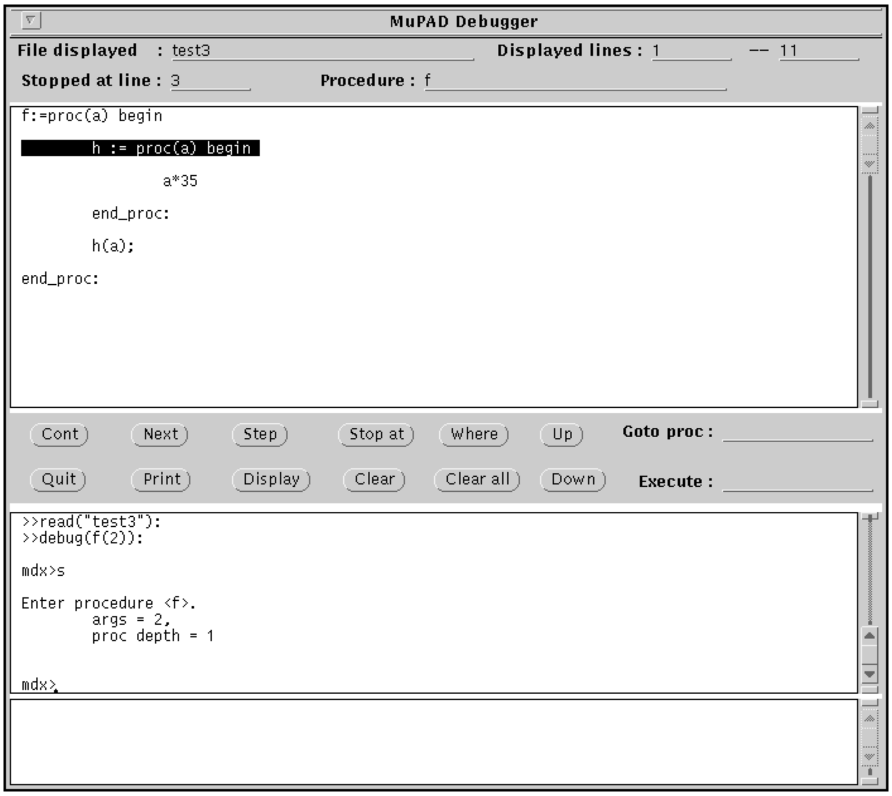


Abbildung 1: mdx-Fenster

Im Gegensatz zur graphischen Benutzerschnittstelle *mdx* stehen dem Benutzer keine Fenster zur Verfügung. Insbesondere fehlt hier das Quelltextfenster, das die Stelle im Quelltext anzeigt, in der sich das Programm befindet. Informationen bzgl. dieser Position im Quelltext können somit nicht visuell dargestellt werden, sondern werden textuell beschrieben.

2 Interaktion zwischen Debugger und Benutzer

Wird **MuPAD** mit eingeschaltetem Debugger gestartet, z.B. mittels `mupad -g`, so werden intern Zusatzinformationen für den Debugger erzeugt, die für den Benutzer allerdings transparent sind. Aus der Sicht des Benutzers gibt es keinen Unterschied zum Standardmodus von **MuPAD**. Der interaktive Debug-Modus wird erst durch die Systemfunktion *debug* aktiviert. Das Argument der Funktion *debug* kann eine beliebige Anweisung sein. Enthält diese Anweisung einen Prozeduraufruf, so erscheint `mdx>` als Prompt, und der Benutzer befindet sich im Eingabemodus des Debuggers, in dem Befehle zur Steuerung des Programmablaufs oder Befehle, die über den Programmzustand informieren, abgesetzt werden können (siehe Abschnitt 4). Solange die Anweisung ausgeführt wird, können nur Befehle an den Debugger gegeben werden. Nachdem die Anweisungen abgearbeitet wurde, meldet der Debugger `Execution completed`, und der Benutzer befindet sich wieder im Eingabemodus von **MuPAD**.

3 mdx: Das X-Frontend des MuPAD-Debuggers

Eine graphische Benutzerschnittstelle zum **MuPAD**-Debugger bildet das Programm *mdx*. Dieses Programm benötigt entweder das Fenster-System *X* oder OpenWindows. Es erweitert den Bedienungskomfort durch:

- Vereinfachung der Eingabe von Debug-Befehlen.
- Optische Aufarbeitung der vom Debugger gelieferten Informationen.
- Gleichzeitige Anzeige des Quelltextes und der Programmausgabe in voneinander getrennten Fenstern.

Aus diesen Gründen ist *mdx* in fünf Fenster unterteilt:

Status Enthält allgemeine Informationen über den aktuellen Zustand. Dies beinhaltet u.a. die Zeilennummer, in der das Programm angehalten wurde, und die Zeilennummer der Datei, die in dem Source-Fenster angezeigt wird.

Source Zeigt den Quelltext der bearbeiteten Datei an. Nach Aufruf des Debuggers zeigt das Fenster die ersten Zeilen der gelesenen Datei an, falls eine Datei im Aufruf spezifiziert wurde, oder das Fenster bleibt vorerst leer. Im Laufe der Programmausführung wird ein Textausschnitt angezeigt, der die Zeile beinhaltet, in der das Programm angehalten wurde. Welchen Textausschnitt das Fenster anzeigen soll, kann auch durch den Benutzer mittels des **goto proc**-Buttons festgelegt werden.

Buttons Häufig benötigte Befehle des Debuggers können durch einen Maus-Klick zur Ausführung gebracht werden. Befehle, die hierin nicht berücksichtigt werden, können direkt im Command-Fenster per Tastatur eingegeben werden.

Command Alle Ausgaben, die durch die Ausführung des **MuPAD**-Programms erzeugt werden, sowie einige Ausgaben, die durch Befehle des Debuggers hervorgerufen werden, erscheinen in diesem Fenster. Eingaben, sowohl an **MuPAD** als auch an den Debugger, müssen in diesem Fenster erfolgen.

Display Die Ausgabe der Variablenwerte, die permanent angezeigt werden sollen, erscheint in diesem Fenster.

Das Source-, Command- und Display-Fenster haben am rechten Rand eine Scrollbar, in jedem dieser Fenster den gesamten Text zu betrachten.

4 Befehlsübersicht

Next Ausführung aller Anweisungen innerhalb der aktuellen Zeile. Der Debugger hält danach in der nächsten auszuführenden Zeile. Enthält eine der ausgeführten Anweisungen einen Aufruf einer benutzerdefinierten Funktion, so wird die Funktion ausgeführt, ohne daß der Debugger innerhalb der Funktion hält.

Step Analog zu **Next**, jedoch hält der Debugger innerhalb benutzerdefinierter Funktionen.

Cont Das Programm wird solange ausgeführt, bis auf eine Anweisung getroffen wird, die in einer mit einer Haltemarke versehenen Zeile steht, oder bis das Programm abgearbeitet wurde. Siehe hierzu auch **Stop at**.

Hält der Debugger, so wird in den Statuszeilen angezeigt, in welcher Zeile der Debugger hält [`Stopped at:`], in welcher Datei diese Zeile steht [`Displayed file:`] und innerhalb welcher Prozedur gehalten wird [`Procedure:`]. Das Quelltextfenster enthält einen Ausschnitt der Datei, die die Zeile beinhaltet, in der der Debugger hält. Welche Zeilen dieser Ausschnitt umfaßt, wird durch [`Lines:` ... `--` ...] wiedergegeben. Die Zeile, in der der Debugger hält, wird zusätzlich invers dargestellt.

Goto proc Die in dem Eingabefeld spezifizierte Prozedur wird im Quelltextfenster angezeigt.

Quit Beendet die Ausführung des **MuPAD**-Programms, sofern der Debugger im Eingabemodus ist. Der Debugger meldet danach `Execution completed`, und der Benutzer befindet sich im **MuPAD**-Eingabemodus. Quit im **MuPAD**-Eingabemodus beendet die gesamte **MuPAD**-Sitzung. Ein laufendes Programm kann mittels *Ctrl-C* angehalten werden. Der Debugger befindet sich hiernach im Eingabemodus.

Execute Der in dem Eingabefeld spezifizierte **MuPAD**-Befehl wird ausgeführt. Der Befehl kann nicht nur aus einer einzelnen Anweisung, sondern aus einer beliebigen Anweisungsfolge bestehen. Eine Anwendung ist z.B. die interaktive Beeinflussung des Programmablaufs, indem der Benutzer Variablenwerte verändert.

Where Im Terminalfenster werden die Prozeduren, die von Beginn des Programmlaufs bis zum jetzigen Zeitpunkt aufgerufen wurden, angezeigt. Neben dem Prozedurnamen wird der Benutzer auch über die Aufrufstelle in Form einer Zeilennummer und eines Dateinamens informiert.

Up Im Quelltextfenster wird die Zeile angezeigt, von der aus die bisher angezeigte Prozedur aufgerufen wurde. Die Statuszeilen des Debuggers werden ebenfalls auf den aktuellen Stand gebracht. Ist bereits die Prozedur erreicht, die der Benutzer interaktiv eingegeben hat, so erscheint im Terminalfenster die Meldung `Top level reached`.

Down Kann nur angewendet werden, wenn zuvor mindestens ein **Up** erfolgte. Es wird die Prozedur im Quelltextfenster angezeigt, aus der **Up** aufgerufen wurde. Ist bereits die aktuelle Prozedur, d.h. die Prozedur in der der Debugger hält, erreicht, so erscheint im Terminalfenster die Meldung `Bottom level reached`.

Clear all Löschen aller Haltemarken.

Clear Der Benutzer spezifiziert mit einem Doppel-Maus-Klick im Quelltextfenster die Zeile, in der die Haltemarke gelöscht werden soll. Siehe **Stop at**.

Stop at Der Benutzer spezifiziert mit einem Doppel-Maus-Klick im Quelltextfenster die Zeile, in der eine Haltemarke gesetzt werden soll. Um in einer bestimmten Prozedur eine Haltemarke zu setzen, muß der Benutzer diese zuvor in das Quelltextfenster laden. Siehe hierzu **Goto proc**.

Print Der Benutzer spezifiziert den Ausdruck, dessen aktueller Wert angezeigt werden soll, indem er im Quelltextfenster den Ausdruck mit der Maus markiert. Der Wert der Variablen wird im Terminalfenster angezeigt. Siehe auch **Execute**.

Display Der Benutzer spezifiziert die Variable, dessen Wert angezeigt werden soll, indem er im Quelltextfenster den Variablennamen mit der Maus markiert. Der Wert der Variablen wird im Display-Fenster angezeigt. Außerdem wird der aktuelle Wert der Variablen ab diesem Zeitpunkt permanent angezeigt. Eine Aktualisierung erfolgt allerdings nur dann, wenn der Debugger im Eingabe-Modus ist, d.h. wenn der Programmlauf unterbrochen ist.

Jeder der zuvor beschriebenen Befehle, die durch einen Maus-Klick auf den entsprechenden **Button** ausgeführt wird, kann auch direkt im Terminalfenster per Tastatur eingegeben werden.

5 Interne Programmstruktur des Debuggers

Daten, die während der Programmausführung erzeugt werden, enthalten niemals relevante Debugger–Informationen. Daten, die relevante Informationen beinhalten, sind nur diejenigen, die vom Parser innerhalb eine read–Anweisung erzeugt werden. Aber selbst hier wird nur eine Information pro Zeile im Quelltext benötigt, unabhängig von der Anzahl der internen Daten, die durch diese Zeile erzeugt werden. Aus diesem Grund ist der MuPAD-Parser das einzige Modul, das Debugger–Informationen generiert, und stellt zudem sicher, daß keine unnötigen Informationen erzeugt werden.

Um die Informationen, die der Debugger benötigt, zu speichern, existiert ein eigenständiger Kategorietyp (CAT_DEBUG). Diese Kategorie speichert eine Debugger - Information. Eine Information betsteht aus einem 32-Bit Integer, der die Zeilennummer und den Dateinamen in kodierter Form speichert.

Da in MuPAD ein Datum aus Kategorien besteht, und der Evaluator auf Kategorien arbeitet, existiert aus der internen Sicht des MuPAD-Systems kein Unterschied zwischen algebraischen Daten, z.B. Zahlen, und Debugger-Informationen. Der Debugger ist daher weder ein Aufsatz, noch eine Ergänzung bestehender Datenstrukturen, sondern eine Erweiterung des Systems um einen neuen Kategorietyp, der die Struktur und Eigenschaften vorhandener Kategorietypen weder verändert, noch beeinflußt.

Wird eine Kategorie vom Typ CAT_DEBUG evaluiert, so wird der Debugger–Kern aktiviert, der die weitere Bearbeitung übernimmt. Im Normalmodus wird dieser Kategorietyp vom Evaluator ignoriert.

Hieraus ergibt sich, das der Evaluator im Normalmodus keinerlei Laufzeiteinbußen aufweist, und die Ausführungsgeschwindigkeit selbst beim Vorhandensein von Debugger-Informationen nur minimal herabgesetzt wird. Der Parser ist das einzige Modul, dessen Ausführungsgeschwindigkeit bei eingeschaltenem Debugger vermindert wird, aber dies beeinträchtigt die Ausführunggeschwindigkeit eines eingelesenen Programmes nicht, da nur der Evaluator für die Programmausführung zuständig ist.

Eigenschaften des Debuggers:

- Keine Beeinflussung bestehender Datenstrukturen und Kategorien.
- Keine Beeinflussung der Evaluierungsalgorithmen bestehender Kategorien.
- Es werden nur Informationen generiert, die auch benötigt werden. Es werden keine redundanten Daten erzeugt.
- Der Debugger verarbeitet Daten, auf die mehrfach referenziert wird (Stichwort “unique data representation”).

- Die Ausführungsgeschwindigkeit des Evaluators wird im Normal-Modus durch das Debug-Modul nicht herabgesetzt.

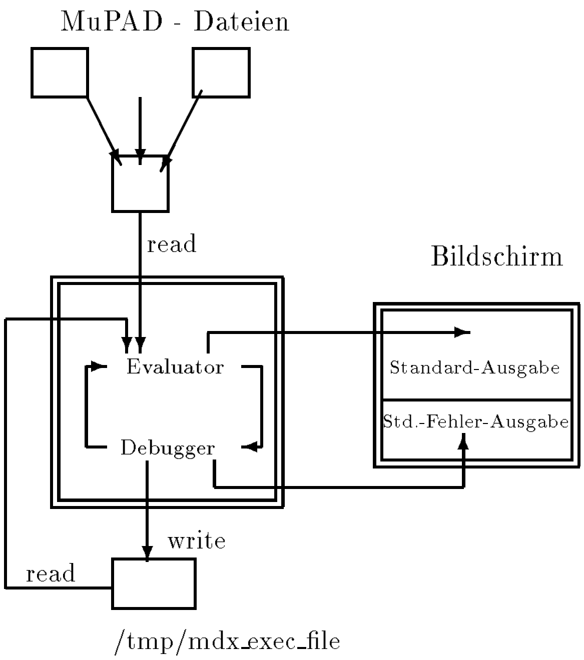
- Klare Struktur des Debuggers.

- Prinzipielle Möglichkeit auf Benutzerebene nachträglich Debug–Informationen in ein Programm einzufügen.

- Hohe Ausführungsgeschwindigkeit bei eingeschaltetem Debugger. (Wichtig für den Trace-Modus)

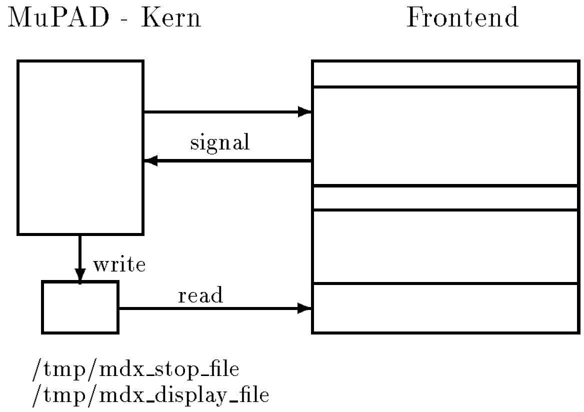
6 Kommunikationstrukturen

6.1 Ein– und Ausgabe des Debuggers



Die Datei */tmp/mdx_exec_file* wird benötigt, wenn der Debugger den Execute– oder den Print–Befehl ausführt. Der vom Benutzer spezifizierte MuPAD–Befehl wird in diese Datei geschrieben, die danach vom MuPAD–Parser gelesen wird, und vom MuPAD–Evaluator ausgeführt wird.

6.2 Grundlegende Kommunikation MuPAD – Kern $\Longleftrightarrow$ Frontend

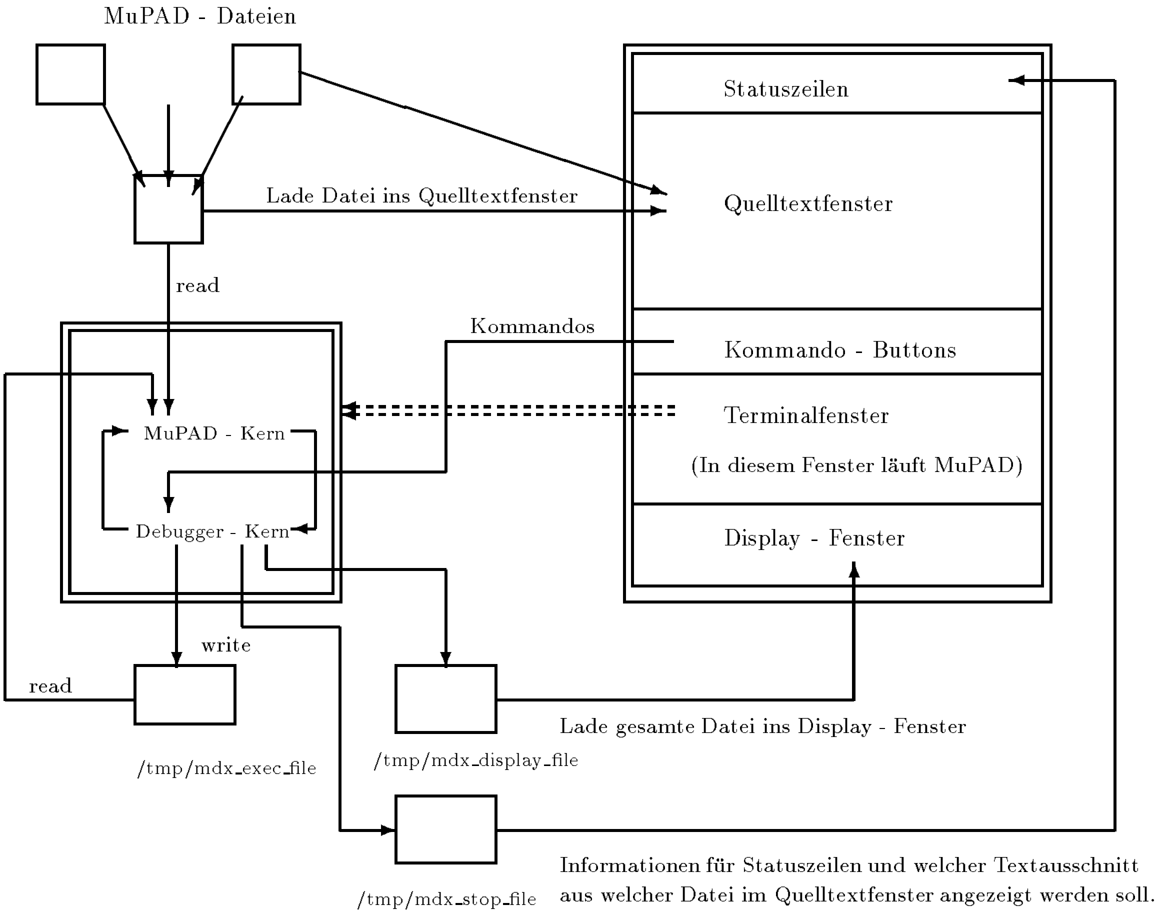


1. Der MuPAD – Kern schreibt Informationen, die dem Frontend mitgeteilt werden sollen, in eine Datei (/tmp/mdx_stop_file bzw. /tmp/mdx_display_file).
2. Nachdem alle Informationen geschrieben wurden, wird dem Frontend ein Signal (SIGCONT) gesendet.
3. Das Frontend liest alle Informationen aus den Dateien.
4. Nachdem alle Informationen gelesen wurden, wird dem MuPAD – Kern ein Signal (SIGCONT) gesendet.
5. Nachdem der MuPAD – Kern dieses Signal erhalten hat, beginnt dieser Zyklus von Neuem.

Wird der Debugger über das Frontend *mdx* aufgerufen, so werden nicht mehr alle Debugger–Informationen direkt auf dem Bildschirm ausgegeben, sondern werden in Dateien geschrieben. Diese Dateien dienen dem Datenaustausch zwischen MuPAD–Kern (mit integriertem Debugger) und dem Frontend. Es existieren zwei Dateien (*/tmp/mdx_stop_file* und */tmp/mdx_display_file*).

Die Datei *mdx_stop_file* beinhaltet Informationen, die das Quelltextfenster und die Statuszeilen des Frontends beeinflussen. Die Datei */tmp/mdx_display_file* enthält die Werte der permant anzuzeigenen Variablen.
Beide Dateien werden vom Frontend gelesen, wenn der Debugger hält, d.h im Eingabemodus ist.

6.3 Datei – Kommunikation MuPAD – Kern $\Longleftrightarrow$ Frontend



Entwurf und Implementation des MuPAD-Kernes

Andreas Kemper

Im folgenden Beitrag möchte ich einen Überblick über die Implementation des Kerns des Computeralgebra-Systems **MuPAD** geben. Zu diesem Zweck werden die einzelnen Komponenten des Systems, ihre Aufgaben und ihre technische Realisierung sowie das Zusammenspiel der einzelnen Teile untereinander beschrieben. Besonders ausführlich möchte ich auf die Datenstrukturen zur Darstellung algebraischer Objekte sowie die Mechanismen zur Evaluierung dieser Daten eingehen. Einen weiteren Schwerpunkt bildet die Darstellung assoziativer Speicher, die innerhalb des Kernes an den unterschiedlichsten Stellen zum Einsatz kommen, um die Geschwindigkeit des Systems zu erhöhen und die Größe der erzeugten Daten einzuschränken.

1 Allgemeiner Aufbau

Das MuPAD-Paket besteht unter UNIX aus mehreren weitestgehend unabhängigen Programmen, die gleichzeitig ablaufend unterschiedliche Aufgaben wahrnehmen. Zentrales Element dieser Programme ist der sogenannte **Kern**, in dem die eigentlichen algebraischen Operationen durchgeführt werden. Die übrigen Programme stellen Tools dar, die die Bedienung des Systems erleichtern und die Darstellung algebraischer Daten unterstützen. Alle diese Programme kommunizieren sowohl mit dem Kern wie auch teilweise untereinander.
Der Benutzer wird bei der Bedienung des Systems durch die graphische Benutzeroberfläche **XMuPAD** unterstützt. Um die Bedienung des im Kern vorhandenen zeilenorientierten Quelltextdebuggers zu vereinfachen, kann das speziell hierfür entwickelte Tool **mdx** verwendet werden, welches eine komfortable graphische Bedienung des Debuggers ermöglicht. Die Dokumentation des Systems steht Online in Form des Hypertextsystem **HyTeX**[1] zur Verfügung. Außerdem existiert ein **Graphik-Tool** zur graphischen Darstellung der im System behandelten Objekte.
Im folgenden möchte ich lediglich auf den MuPAD-Kern eingehen. Dieser steht auch ohne die soeben beschriebenen Oberflächen und zusätzlichen Tools in Form einer Terminalversion zur Verfügung. Bereits diese Minimal-Version erlaubt jedoch eine interaktive Benutzung des im Kern realisierten Debuggers. Ihre algebraischen Fähigkeiten sind durch das Fehlen der übrigen Komponenten in keiner Weise eingeschränkt.
Aus der Sicht des Benutzers besteht das System aus einem Kreislauf von Benutzereingabe, Evaluierung des erzeugten Datums und Ausgabe des Ergebnisses. Dies spiegelt sich auch in der internen Realisierung des Kernes wieder. Mit Ausnahme der Initialisierungsroutinen führt er eine Schleife aus, die zunächst zum Einlesen der Benutzereingabe den Parser aufruft, wobei die Eingabe syntaktisch überprüft und dann eine entsprechende Datenstruktur aufbaut wird. Hierbei werden gegebenenfalls Informationen für den Debugger bereitgestellt, wenn es sich um das Einlesen eines Programmes im Debug-Modus handelt. Der nächste und bei weitem aufwendigste Schritt umfaßt die Evaluierung der erzeugten Datenstruktur. Die hierbei verwendeten Techniken werden im folgenden detailliert beschrieben. Das Ergebnis der Evaluierung wird von diesem Modul in Form einer zumindest logisch neuen Datenstruktur zurückgegeben, welche anschließend im letzten Schritt durch das Ausgabemodul auf dem Bildschirm ausgegeben wird.

2 Die Speicherverwaltung

Die Grundlage für die Implementation des Kernes ist **Mammut**, die **M**emory **A**llocation **M**anage**m**ent **U**ni**t** . Hierbei handelt es sich um eine universell einsetzbare Speicherverwaltung, die jedoch die Anforderungen eines Computeralgebra-Systems optimal unterstützt. Mammut übernimmt innerhalb des Kernes die Bereitstellung, Manipulation und Freigabe jedes dynamischen Objektes. Mammut stellt damit die unterste Ebene des Systems dar.
Die einzelnen Mammut-Objekte bestehen aus einem statischen Teil von konstanter Größe und einem dynamischen Teil, der von der Speicherverwaltung selbstständig verschoben werden kann. Ziel dieses Verschiebens ist die Vermeidung der Fragmentierung des Speichers, einem für Computeralgebra-

[1] Eine Entwicklung der Arbeitsgruppe von Prof. Dr. N. Köckler an der Universität - Gesamthochschule Paderborn

Systeme typischen Problem. Hierbei entsteht, durch wiederholtes Anfordern und Freigeben von Speicherobjekten unterschiedlicher Größe, innerhalb des Speichers ein Gemisch von großen und kleinen, benutzten und unbenutzten Bereichen, sodaß nach einiger Zeit Speicheranforderungen für größere Objekte nicht mehr befriedigt werden können, obwohl insgesamt genügend Speicher zur Verfügung steht.
Durch die Verschiebung der einzelnen Objekte kann hier Abhilfe geschaffen werden. Viele kleine freie Bereiche werden zu wenigen großen Stücken zusammengeschoben, sodaß diese wieder genutzt werden können. Leider besitzt die Verschiebbarkeit dieses Datenteils auch den Nachteil, daß auf seinen Inhalt über einen Doppelverweis, einen sogenannten Handle, zugegriffen werden muß, da sich der Speicherort der entsprechenden Daten beliebig ändern kann. Mit Hilfe des Handles wird dieses Problem umgangen. Eine andere Möglichkeit, um jederzeit auf die richtige Speicherstelle zugreifen zu können, besteht darin, bestimmte Verweise in diesen dynamischen Bereich auszuzeichnen, sodaß ihr Wert bei einem Verschieben automatisch aktualisiert wird. Beide Möglichkeiten besitzen einen gewissen Overhead im Vergleich zum herkömmlichen Zugriff über einen einfachen Verweis. Welche Methode innerhalb des Kernes zur Anwendung kommt, hängt von der konkreten Situation ab. So ist es z.B. beim schrittweisen Durchlaufen eines verschiebbaren Bereiches mit Sicherheit günstiger, den hierzu verwendeten Pointer als Verweis in einen dynamischen Teil auszuzeichnen und dann wie gewohnt vorzugehen, indem der Pointer herauf- oder heruntergezählt wird. Der wiederholte Zugriff über einen Handle wäre in diesem Fall mit Sicherheit wesentlich zeitaufwendiger.
Aufgrund der einheitlichen Größe des statischen Teils (Header) eines Mammut-Objektes, muß dieser nicht verschoben werden, da die frei werdenden Bereiche dieser Art sofort wieder durch neue passende Header belegt werden können. Daher kann der Zugriff auf einen Header durch einen einfachen Verweis erfolgen. Innerhalb des Headers befindet sich dann der Verweis auf den dynamischen Teil des Objektes, sodaß sich für diesen Teil der bereits erwähnte Handle ergibt. Wird dieser dynamische Teil nun verschoben, so ist lediglich dieser Verweis innerhalb des zugehörigen Headers anzupassen. Daher kann auf das verschobene Objekt weiterhin jederzeit zugegriffen werden.
Der dynamische Teil eines Mammut-Objektes enthält unter anderem einen Bereich, der zur Speicherung einer beliebigen Anzahl von Verweisen auf andere Mammut-Objekte dient. Die Größe dieses Bereiches und damit die Anzahl der enthaltenen Verweise kann bei der Anforderung eines neuen Objektes frei bestimmt werden. Die Verweise werden innerhalb dieses Bereiches sequentiell angeordnet, sodaß auf einen konkreten Eintrag in konstanter Zeit zugegriffen werden kann. Hierdurch ist es möglich, die vom Kern als Datenstruktur verwendeten n-ären Bäume effizient zu realisieren.
Einen weiteren wichtigen Aspekt stellt die Möglichkeit, physikalisch einmal vorhandene Objekte logisch mehrfach darzustellen, dar. Hierdurch wird die **Unique Data Representation** unterstützt, bei der versucht wird, gleiche algebraische Ausdrücke oder Teile dieser nur einmal im Speicher abzulegen. Auf die Vorteile dieses Verfahrens möchte ich im folgenden noch genauer eingehen.
Für die parallele Version des Systems realisiert die Speicherverwaltung eine **Shared Memory Maschine**, sodaß die Algorithmen des sequentiellen Kernes in weiten Bereichen unverändert verwendet werden können. Größere Änderungen sind lediglich bei der Realisierung des Laufzeitkellers der parallelen Version sowie bei der Implementation der parallelen Elemente der Programmiersprache erforderlich.

3 Datenstrukturen

Von grundlegender Bedeutung bei der Implementation eines Computeralgebra-Systems ist die Auswahl geeigneter Datenstrukturen. Nahezu alle Algorithmen des Kernes sind damit beschäftigt, diese Objekte zu erzeugen, zu analysieren oder auf irgend eine Art und Weise zu verändern. Wie elegant und damit einfach und schnell dies erfolgen kann, ist in großem Maße vom Aufbau der Datenstrukturen abhängig. Damit muß der Entwickler vor dem Design dieser Strukturen klären, welche Art von Algorithmen später auf diesen Objekten arbeiten und wie die Arbeit dieser Algorithmen durch die Datenstrukturen optimal unterstützt werden kann.
Einen wichtigen Gesichtspunkt stellt in diesem Zusammenhang ein möglichst gleichförmiger Aufbau aller Datentypen dar. Hierdurch können viele Vorgänge stark vereinfacht und somit u.a. eine größere Lesbarkeit und Wartbarkeit der Programme erzielt werden. Dies ist gerade bei einem Projekt, das einer ständigen Weiterentwicklung unterworfen ist, von größter Bedeutung.
Einen ersten Schritt in diese Richtung stellt die konsequente Verwendung des Mammut-Modules dar. Hierdurch besitzen alle dynamischen Datenobjekte eine uniforme Struktur, die in vielen Fällen eine genauere Unterscheidung der Datenobjekte unnötig macht und

dadurch viele Algorithmen vereinfacht und beschleunigt.
Ein anderer wichtiger Punkt ist die Anlehnung der Datenstrukturen an die innere Logik der dargestellten algebraischen Objekte. Unter diesem Gesichtspunkt wurden n-äre Bäume als Grundstruktur gewählt, wobei die Knoten der Bäume aus den Mammut-Objekten gebildet werden und die bereits beschriebenen von diesen Objekten ausgehenden Verweise zur Verbindung der einzelnen Knoten genutzt werden. Diese Struktur kommt der mathematischen Interpretation eines algebraischen Ausdrucks als Operator mit einer beliebigen Anzahl von Operanden sehr nahe. Hierbei werden sowohl der Operator wie auch die Operanden als Söhne eines Knotens dargestellt. In diesem Fall ist der erste Sohn als Operator zu interpretieren. Um die bereits angesprochene Gleichförmigkeit zu gewährleisten, wird versucht, nahezu alle algebraischen Daten in solch einer funktionalen Gestalt zu speichern. So werden sämtliche Operatoren des Systems funktional gespeichert. Der Ausdruck

$$a + b + c$$

wird dabei durch den Funktionsaufruf

$$\_plus(a, b, c)$$

repräsentiert, also durch einen Knoten mit den vier Söhnen

$$\_plus, a, b \text{ und } c.$$

Die Arbeit des Kernes und insbesondere des Evaluierers besteht zum einen aus der Evaluierung von Ausdrücken beliebiger Form, zum anderen aus der Ausführung von Anweisungen wie Zuweisungen, Schleifen und bedingten Verzweigungen. Ebenso werden Prozeduren durch den in den Evaluierer integrierten Interpreter ausgeführt. Dieser tritt bei der interaktiven Eingabe von Anweisungen bzw. dem Aufruf von Programmen automatisch in Aktion.
Da aus logischer Sicht kein wesentlicher Unterschied zwischen algebraischen Daten und Programmkonstrukten wie Prozedurdefinitionen und Anweisungen besteht, wird auch auf der Ebene der Datenstrukturen zwischen diesen Objekten nicht unterschieden. So werden auch Anweisungen und Prozeduren funktional gespeichert, sodaß eine Zuweisung der Form

$$a := b$$

als

$$\_assign(a, b)$$

und somit als ein Knoten mit den Söhnen

$$\_assign, a \text{ und } b$$

dargestellt wird. Durch diese Vorgehensweise wird es ermöglicht, Anweisungen und Programme ebenso wie Ausdrücke zu manipulieren. So können diese an Funktionen übergeben, zerlegt und beliebig zusammengesetzt werden.
Die interne Darstellung ist dem Benutzer des Systems auch direkt zugänglich. Zu diesem Zweck können die sogenannten Underline-Funktionen wie zum Beispiel *_plus* oder *_assign* in der Eingabe benutzt werden.
Von der funktionalen Speicherung abgewichen wird im Fall von Listen, Mengen, Tabellen und Feldern. Mengen werden mit Hilfe von assoziativen Speichern realisiert, die selbstverständlich ebenfalls aus Mammut-Objekten aufgebaut werden. Listen bestehen aus einem Objekt, dessen Verweise sämtlich als Listenelemente interpretiert werden. Felder verschiedener Dimension werden ebenfalls aus solch listenähnlichen Strukturen in eventuell mehreren Stufen gebildet.
Ein Knoten innerhalb der Datenstruktur eines algebraischen Ausdrucks trägt jedoch nicht nur Verweise auf andere Knoten. Es werden auch Informationen gehalten, die innerhalb jeden Knotens einen Speicherplatz von gleicher Größe beanspruchen. Hierzu gehört zum Beispiel die sogenannte Signatur, die einen Hashwert für den von diesem Knoten und seinen Söhnen gebildeten Teilbaum darstellt. Auf die Nützlichkeit solcher Signaturen werde ich im Zusammenhang mit der Beschreibung des Evaluierers näher eingehen. Da es sich hier um einen Datenbereich konstanter Größe handelt, der in jedem Datenobjekt vorhanden sein muß, ist es sinnvoll ihn im statischen Teil eines Knotens, dem Header, anzulegen. Da der Header, wie bereits beschreiben, nicht verschoben werden kann und im Gegensatz zu den anderen Teilen eines Objektes durch einen einfachen Verweis erreichbar ist, kann der Zugriff auf die Signatur sehr effizient erfolgen.
Die Objekte der Speicherverwaltung unterliegen einer strengen Typisierung. Bei jedem Erzeugen eines neuen Objektes ist dessen interner Typ anzugeben. So existiert zum Beispiel der Typ **MMM_PARI**, der zur Speicherung eines numerischen Objektes des **Pari**[2]-Paketes dient, welches die Arithmetik innerhalb des Systems realisiert. Ein anderer wichtiger Typ, **MMM_STRING**, dient zur Darstellung einer beliebigen Zeichenfolge, die je nach darüberhängendem Knoten vom System als Bezeichner oder String interpretiert wird. Die meisten Objekte besitzen den Mammut-Typ **MMM_CATEGORY**. Diese Objekte stellen aus Sicht des Kernes vollständige

[2]Pari ist ein Arithmetikpaket von C. Batut, D. Bernardi, H.Cohen und M. Olivier

algebraischen Daten dar und sind wiederum typisiert. Alle Objekte des Typs **MMM_CATEGORY** besitzen auch in ihrem Datenteil einen uniformen Aufbau und unterscheiden sich nur durch die Anzahl der an ihnen hängenden Verweise. Außer diesen Verweisen und einem Eintrag, der der Typisierung dient, enthalten Kategorieobjekte ein Bitfeld, in dem die Vorzeichen des Datums sowie verschiedene Flags zur Unterstützung des Evaluierers gespeichert werden. Alle wesentlichen Funktionen des Evaluierers arbeiten auf der Ebene von Objekten des Typs **MMM_CATEGORY**, das heißt, daß sowohl Eingabe als auch Ausgabe der Funktionen aus Objekten diesen Typs bestehen. Um diese Regel nicht zu durchbrechen, existiert eine eigene Kategorie **CAT_NULL**, die logisch dem NULL-Pointer entspricht.

Eine besondere Rolle spielt auch die Kategorie **CAT_DEBUG**. Sie dient zur Speicherung von Informationen für den Debugger und erlaubt diesem die Zuordnung von Anweisungen eines Programmes zu einer bestimmten Textstelle innerhalb des Quelltextfiles. Zu diesem Zweck werden Daten wie die Zeilennummer und der Filename der jeweiligen Datei gespeichert. Diese Kategorien werden vom Evaluierer im Normalbetrieb, also bei ausgeschaltetem Debugger, vollständig ignoriert bzw. erst gar nicht erzeugt. Lediglich im Debug-Modus führt die Evaluierung einer solchen Kategorie zu einer Reaktion des integrierten Debuggers bzw. zu einer Kommunikation mit dem **mdx**-Tool, falls das Debugging mit seiner Hilfe durchgeführt wird.

Durch den geschickten Aufbau der Datenstrukturen kann die Realisierung der Manipulationsfunktionen wie **op** und **subsop** stark vereinfacht werden. Der i-te Operand ist in der Regel durch den Zugriff auf den i-ten Sohn eines Knotens zu erhalten. Hierbei ist es im allgemeinen nicht nötig, das gegebene Datum näher zu untersuchen. Auch der Typ eines Ausdruckes wird immer durch den ersten Sohn bestimmt. An dieser Stelle wird besonders deutlich, wie eine geschickt ausgewählte Datenstruktur die benötigten Algorithmen vereinfachen kann. Jeder Speziallfall innerhalb der Datenstrukturen, der von der oben beschriebenen Regel abweicht, führt zu einem zusätzlichen Aufwand bei der Realisierung der Manipulationsfunktionen. Ähnliches gilt für viele Algorithmen innerhalb des Kernes.

Der Evaluierer

Aufgabe des Evaluierers ist die Substitution von Bezeichnern, die Ausführung von Systemfunktionen sowie die Vereinfachung von Ausdrücken. Die Ausführung von benutzerdefinierten Funktionen, wird ebenso vom Evaluierer veranlaßt. Die eigentliche Ausführung der Programme übernimmt der Interpreter, der hierfür jedoch wiederum den Evaluierer benötigt. Beide Module sind darher nicht klar voneinander zu trennen. Auch das Lesen und Schreiben aus bzw. in Tabellen und Feldern wird vom Evaluierer durchgeführt. Eine weitere Hauptaufgabe des Evaluierers besteht darin algebraische Daten in eine Normalform zu bringen, da nur diese das effiziente Vergleichen von algebraischen Objekten ermöglicht. Zu diesem Zweck wird der interne Simplifier eingesetzt.

Zur Unterstützung des Evaluierers werden mehrere assoziative Speicher verwendet, die das mehrmalige Evaluieren ein und desselben Datums verhindern und auf diese Art und Weise die Geschwindigkeit des Systems erheblich steigern.

Das evaluierte Resultat eines algebraischen Ausdruckes stellt im allgemeinen einen vollständig neuen Ausdruck dar. Können jedoch Teile des alten Datums weiterverwendet werden, da sie bereits evaluiert vorliegen, das heißt durch die Evaluierung nicht verändert worden sind, so werden die entsprechenden Teilbäume durch den Evaluierer lediglich logisch kopiert und in den das Resultat darstellenden Baum eingefügt. So wird gewährleistet, daß ein und dasselbe Datum möglichst selten mehrmals gespeichert wird. Das logische Kopieren wird durch die Speicherverwaltung durchgeführt. Im allgemeinen wird vom System nicht versucht, die Unique Data Representation strikt einzuhalten. Sollte dies geschehen, so müßte bei jeder Veränderung eines Datums sowie bei jeder Generierierung überprüft werden, ob ein entsprechendes Datum bereits als Datenstruktur vorliegt bzw. ob ein Baum exisitiert, der einen entsprechenden Teilbaum enthält. Dieser Aufwand würde sicher den Nutzen eines solchen Vorgehens bei weitem übertreffen. Die Strategie innerhalb des Kernes ist es hingegen, die Gleichheit von Daten dort auszunutzen, wo sie ohne zusätzlichen Aufwand erkannt werden kann. Eine dieser Stellen ist der soeben beschriebene Vorgang der Evaluierung von Operanden. Ebenso wird der Wert eines Bezeichners bei seiner Substitution aus dem Stack gelesen und eine logische Kopie dieses Datums anstelle des Bezeichners eingesetzt. Gegebenenfalls ist hier natürlich, in Abhängigkeit von der Substitutionstiefe, eine weitere Evaluierung des neuen Wertes erforderlich.

Der Vorgang des Evaluierens eines Ausdruckes erfolgt rekursiv. Zum Evaluieren einer Summe wird zunächst der Evaluierer auf jeden Summanden ange-

wand. Die evaluierten Söhne werden sodann sortiert, wenn es sich um die Operanden eines kommutativen Operators handelt, um im nächsten Schritt Vereinfachungen leichter durchführen zu können. Zu solchen Vereinfachungen gehört in diesem Fall zum Beispiel das Zusammenziehen gleicher Summanden zu einem Produkt. Gleiche Summanden können nach der Sortierung leicht identifiziert werden, da sie anschließend benachbarte Operanden sein müssen. Ebenso werden numerische Konstanten addiert und das Ergebnis als letzter Summand der Summe abgelegt. Die bereits beschriebenen Signaturen definieren auf natürliche Art und Weise eine Ordnung auf der Menge der algebraischen Ausdrücke. Die Sortierung der einzelnen Operanden eines Ausdruckes erfolgt auf der Grundlage dieser Ordnung.
Eine Evaluierung kann aber auch die Auswertung eines indizierten Bezeichners sein, wobei ein Wert aus einer Tabelle, einem Feld oder einer Liste gelesen werden muß. Auch Tabellen, Felder, Listen oder Mengen selbst sind vor ihrer weiteren Verarbeitung zu evaluieren. Im Falle von Listen und Mengen geschieht dies durch die Evaluierung der enthaltenen Elemente. Das Ergebnis wird, wenn mindestens ein Operand wirklich verändert worden ist, in einem neuen Objekt gespeichert. Anderenfalls ist auch hier lediglich ein logisches Kopieren erforderlich.

4 Evaluierung von Systemfunktionen und Operatoren

Die Evaluierung von Systemfunktionen und Operatoren geschieht in MuPAD durch einen besonderen Mechanismus. Der Grundgedanke hierbei ist, daß das die Funktionalität einer Systemfunktion oder eines Operators speichernde Objekt durch eine einfache Substitution des entsprechenden Bezeichners erhalten werden kann. Zu diesem Zweck wird diese Funktionalität mit Hilfe einer sogenannten Funktionsumgebung gespeichert. Hierbei handelt es sich um einen eigenen Kategorietypen, der drei Söhne besitzt. Der erste ist für die Evaluierung zuständig, der zweite steuert die Ausgabe der jeweiligen Systemfunktion bzw. des algebraischen Ausdrucks. Der dritte Eintrag kann aus einer Tabelle bestehen, in der weitere Eigenschaften gespeichert werden können. Die beiden ersten Söhne bestehen aus Objekten des Tys **CAT_EXEC**, deren Operanden aus gewöhnlichen algebraischen Ausdrücken wie auch aus C-Funktionen bestehen. Zu diesem Zweck bietet die Speicherverwaltung die Möglichkeit, C-Funktionen direkt in besondere Objekte zu integrieren, sodaß diese als Mammut-Objekte verwendet werden können. Diese C-Funktionen dienen zur eigentlichen Ausführung der Evaluierung oder Ausgabe. Zur Unterstützung dieser Aufgabe enthalten die übrigen Operanden der **CAT_EXEC**-Knoten Informationen wie z.B. die Priorität des Operators, den Namen der Systemfunktion oder das Aussehen des Operatorsymbols für die Ausgabe. Der Name einer Systemfunktion wird vom Evaluierer zum Beispiel benötigt, um nicht weiter auswertbare Funktionsaufrufe zurückgeben zu können. Alternativ kann der erste oder der zweite Sohn einer Funktionsumgebung auch aus einer benutzerdefinierten Prozedur bestehen, die dann die Evaluierung bzw. Ausgabe übernimmt.
Die Funktionsumgebungen werden im Stack unter dem jeweiligen Funktionsnamen gespeichert. Im Verlauf der Evaluierung einer solchen Systemfunktion wird nun also zunächst der linke Operand zu einer solchen Funktionsumgebung evaluiert. Hierzu ist lediglich eine gewöhnliche Evaluierung des entsprechenden Operanden nötig. Aus der resultierenden Struktur wird dann die aufzurufende C-Funktion bzw. Prozedurdefinition ermittelt und mit dem ursprünglichen Funktionsaufruf als Parameter ausgeführt. Handelt es sich um eine C-Funktion, so werden als weitere Parameter die an dem jeweiligen Sohn der Funktionsumgebung hängenden Operandenen mit übergeben. Das Ergebnis dieses Aufrufes ist zugleich das Ergebnis der Evaluierung.
Durch diesen Mechanismus ist es unter anderem Möglich, die Funktionaliät eines Operators oder einer Systemfunktion global zu verändern, indem dem Funktionsnamen eine andere oder veränderte Funktionsumgebung zugewiesen wird. Auch die Funktionsumgebung selbst kann so auf MuPAD-Ebene verändert werden, um zum Beispiel lediglich die Ausgabe eines Operators zu verändern.

5 Signaturen

Ein grundlegendes Konzept innerhalb des Kernes bilden Signaturen. Hierbei handelt es sich, wie bereits beschrieben, um eine Art Hashwert, der jedem Teilausdruck eines Datums innerhalb des Kernes zugewiesen wird. Die Signaturen stellen sicher, daß zwei identische Datenstrukturen den gleichen Signaturwert besitzen. Umgekehrt müssen gleiche Signaturen jedoch nicht auf gleichen Ausdrücken beruhen. Außerdem definieren die Signaturen eine einfache, wenn auch künstliche, Ordnung auf der Menge aller algebraischen Objekte. Diese Ordnung ist im allgemeinen keine Verallgemeinerung der lexikographi-

schen Ordnung für Bezeichner.
Die Berechnung eines Signaturwertes für ein beliebiges algebraisches Datum wird im Verlauf der Evaluierung bzw. Vereinfachung durchgeführt, da er in MuPAD nur für Daten in Normalform die oben beschriebenen Eigenschaften besitzt. Hierbei werden ausgehend von den Blättern des entsprechenden Baumes die Signaturen in die Knoten eingetragen. Die Signaturen der Blätter (Bezeichner, Strings und numerische Konstanten) werden aus der Folge der ASCII-Zeichen bzw. aus den Signaturen der Pari-Datenstruktur errechnet. Die Signaturen von Ausdrücken, die aus mehreren Teilausdrücken bestehen, werden mit Hilfe der Signaturen der einzelnen Teilausdrücke errechnet.
Eine wesentliche Hilfe stellen die Signaturen beim Vergleichen von Ausdrücken dar. Hier kann zunächst der Signaturwert verglichen werden. Stimmt dieser nicht überein, können die Ausdrücke nicht identisch sein. Erst im Fall identischer Signaturen ist ein wirklicher Vergleich der Daten durch ein rekursives Vergleichen der Söhne des Knotens erforderlich. Eine gute Streuung der Signaturen sorgt dafür, daß dieser Fall nur sehr selten eintritt und somit ein wirkliches Vergleichen kaum erforderlich ist, wenn die Ausdrücke nicht identisch sind.
Da das Vergleichen von algebraischen Daten in vielen Modulen des Kernes benötigt wird, so im Vereinfacher bei der Sortierung von Operanden und beim Zusammenfassen gleicher Söhne, aber auch bei der Realisierung der assoziativen Speicher, kommt der Verwendung von Signaturen innerhalb des Kernes eine zentrale Bedeutung zu. Die hier verwendeten assoziativen Speicher werden mit Hilfe von Hashtabellen realisiert und müssen daher zur endgültigen Identifikation eines Objektes immer einen Vergleich durchführen.

6 Assoziative Speicher

Ein eigenes Modul stellt auf der Ebene des Kernes assoziative Speicher mit unterschiedlichen Assoziationsmechanismen zur Verfügung. Hierbei können in einem assoziativen Speicher beliebige Daten miteinander verbunden werden, wodurch sich ein weites Anwendungsgebiet ergibt.
Einen Einsatzbereich dieser Speicher stellen Mengen und Tabellen dar. Mengen sind in Form eines assoziativen Speichers realisiert, in dem jedes Element der Menge mit sich selbst assoziiert wird. Beim Eintrag eines neuen Elementes in eine Menge wird nun zunächst überprüft, ob mit dem entsprechenden Element bereits ein Wert assoziiert ist; ist dies nicht der Fall, so kann das neue Datum eingetragen werden. Ein analoges Vorgehen stellt die Implementation von Tabellen dar. Hier wird der Index eines Eintrages mit dem zugehörigen Wert assoziiert. Gleichzeitig besitzen diese Speicher die Möglichkeit, sich selbst, in Abhängigkeit ihrer Einträge, eine Signatur zuzuordenen. Dies ist erforderlich da eine Menge oder Tabelle ein gültiges algebraisches Datum darstellt und daher natürlich auch mit anderen Daten verglichen werden können muß.
Ein anderes weites Einsatzfeld dieser Datenstruktur bildet der Evaluierer. Hier sorgen mehrere assoziative Speicher für eine beträchtliche Beschleunigung. Ein Phänomen innerhalb der Computeralgebra besteht darin, daß gleiche Teilausdrücke durch Substitution und Evaluierung wiederholt vorkommen und damit auch wiederholt evaluiert werden müssen. Um hier das mehrfache Ausführen ein und desselben Vorganges zu verhindern, werden innerhalb des Evaluierers auf mehreren Ebenen assoziative Speicher eingesetzt, die sich bereits evaluierte algebraische Daten zusammen mit dem Ergebnis der Evaluierung merken. Wird ein beliebiger algebraischer Teilausdruck evaluiert, so wird zunächst mit Hilfe eines solchen Speichers überpüft, ob der entsprechende Ausdruck bereits evaluiert worden ist. Ist dies der Fall, so kann das Ergebnis direkt entnommen werden und die Evaluierung ist beendet. Im anderen Fall wird die Evaluierung durchgeführt und anschließend der zu evaluierende Ausdruck zusammen mit dem Ergebnis in den assoziativen Speicher eingefügt. Leider muß der Inhalt eines solchen assoziativen Speichers nach der Durchführung einer Zuweisung gelöscht werden, da sich hierdurch die Werte von Variablen verändern und somit die vorhandenen Assoziationen ungültig werden können. Dies betrifft zwar im allgemeinen nur einen geringen Teil der Einträge, aber eine genauere Analyse ist hier aufgrund der komplexen Abhängigkeiten in vertretbarer Zeit kaum durchzuführen. Jedoch wird nach der Evaluierung einer Zuweisung mit Hilfe eines Zeitstempels überprüft, wann die Variable, an die ein Wert zuzuweisen ist, das letzte Mal vom Evaluierer substituiert worden ist. Ist dies vor dem letzten Löschen des Speichers erfolgt, so ist ein erneutes Löschen nicht erforderlich.
Eine weitere Besonderheit bilden die unterschiedlichen Assoziationsmethoden, die die assoziativen Speicher zur Verfügung stellen. Im ersten Fall wird beim Lesen aus dem assoziativen Speicher ein äquivalenter Eintrag gesucht, um den hiermit verbundenen Wert zu erhalten. Dabei ist es lediglich erforderlich, daß das angegebene Datum und der entsprechende Wert im Speicher dasselbe Datum repräsentieren. Im

Gegensatz hierzu müßen bei der zweiten Assoziationsmethode diese Daten auch physikalisch übereinstimmen, d.h. hier wird lediglich ein Vergleich der physikalischen Adressen beim Suchen eines Eintrages durchgeführt. Dies bedeutet selbstverständlich einen beträchtlichen Geschwindigkeitszuwachs, führt jedoch auch dazu, daß bereits vorliegende Assoziationen nicht in jedem Fall erkannt werden. Damit ist diese Methode nicht für alle Bereiche einsetzbar. Eine Implementation von Mengen mithilfe dieser Methode würde unweigerlich zu Fehlern führen. Hingegen kann ihr Einsatz innerhalb des Evaluierers durchaus sinnvoll sein, da hier, wenn keine Assoziation gefunden worden ist, der gewöhnliche Evaluierungsprozeß angestoßen wird und somit das Ergebnis letztlich nicht beeinflußt wird, lediglich die Ausführungszeit verlängert sich.
Innerhalb des Evaluierers ist zwischen beiden Methoden gründlich abzuwägen. Zwar müssen bei der Benutzung der zweiten Methode mehr Daten wirklich evaluiert werden, dafür ist jedoch auch der Aufwand des Suchens innerhalb des assoziativen Speichers kleiner. Welche Methode im Evaluierer zum Einsatz kommen sollte, wird auch durch den Grad der Einhaltung der Unique Data Representation beeinflußt. Eine strikte Einhaltung würde dazu führen, daß zwei gleiche Objekte immer die gleiche physikalische Adresse besitzen und somit bei der Verwendung der zweiten Assoziationsmethode ebenfalls alle Einträge gefunden würden. Ein weiteres Beispiel für den sinnvollen Einsatz assoziativer Speicher stellt die im nächsten Abschnitt beschriebene Behandlung von Strings dar.

7 Der Interpreter

Um die Konstrukte der MuPAD-Programmiersprache zu verabeiten, verfügt das System, wie bereits beschrieben, über einen Interpreter zur Ausführung dieser Anweisungen und Prozeduren. Die wesentliche Aufgabe dieses Interpreters ist der Aufbau eines Laufzeitkellers für die verschiedenen globalen und lokalen Variablen sowie die formalen Parameter der benutzerdefinierten Prozeduren. Außerdem muß das Zurückspringen aus Prozeduren sowie das Abbrechen von Schleifenkonstrukten geregelt werden. Da sich die Programmkonstrukte in ihrer internen Darstellung nicht von anderen Systemfunktionen unterscheiden, kann zu ihrer Ausführung der gleiche Mechanismus verwendet werden.
Eine zentrale Rolle spielt die Organisation des Laufzeitkellers. Dieser wird im wesentlichen durch den geschachtelten Aufruf der entsprechenden Funktionen des Evaluierers gebildet. Hierdurch wird zum Beispiel das Speichern der Rücksprungadresse beim Verlassen einer Prozedur überflüssig.
Zur Implementation der dynamischen Scope-Regeln muß ein Stack aufgebaut werden, der die Werte für alle zugewiesenen und lokalen Variablen innerhalb jeder aktiven Prozedur speichert. Der Laufzeitkeller jeder Variablen wird hierzu an die Datenstruktur des entsprechenden Bezeichner bzw. den hierunter befindlichen String angehängt. Dies ist nur möglich, da durch die strikte Anwendung der Unique Data Representation für Strings zwei identische Objekte des Typs **MMM_STRING** niemals physikalisch unterschiedlich gespeichert werden. Um dies zu erreichen wird beim Systemstart ein weiterer assoziativer Speicher angelegt, der alle diese Objekte enthält. Da neue Strings nur durch die Konkatenation von Bezeichnern und Strings sowie durch den Parser generiert werden können, ist es relativ einfach die Einhaltung dieser Unique Data Representation zu überwachen, indem jeder neue String zunächst gegen den Inhalt des entsprechenden Assoziativen Speichers getestet wird.
Der Laufzeitkeller eines Bezeichners selbst besteht aus einer einfach verketteten Liste die als Stack genutzt wird, indem Objekte an den Anfang der Liste gehängt oder wieder entfernt werden. Dies geschieht gerade beim Eintritt in eine Prozedur bzw. bei deren Verlassen. Die einzelnen Einträge enthalten den jeweiligen gültigen Wert des Bezeichners. Das Lesen dieses Wertes kann somit immer durch einen einfachen Zugriff auf das oberste Listenelement erfolgen. Dieses ist immer schnell greifbar, da im Fall des Lesens eines Wertes der entsprechende Bezeichner vorliegt unter dem sich der zugehörige String und damit der Laufzeitkeller befindet.
Eine zusätzliche Numerierung der einzelnen Schachteln ermöglicht das kontrollierte Abbauen von Teilen des Stacks, wie es zum Beispiel im Fall eines Laufzeitfehlers erforderlich ist.
In der parallelen Version des Kernes mußte dieses Konzept erweitert werden, da unterschiedliche Tasks unterschiedliche Werte für ein und dieselbe Variable besitzen. Dabei können unterschiedliche Tasks dennoch auf gemeinsame globale Variablen zugreifen. Um dies zu realisieren, hängen an jeder Variablen mehrere Stacks, einer für jeden Prozessor. Durch eine geschickte Speicherung ihrer eigenen Stacktiefe und der ihrer Vorgänger zusammen mit der Nummer der entsprechenden Prozessoren kann jede Task dann effizient auf den Inhalt einer Variablen zugreifen.

Die MuPAD-Sprache kurzgefasst

Klaus Gottheil

In diesem Artikel wird die **MuPAD**-Sprache beschrieben — auf etwa 5 Seiten. Das ist natürlich etwas knapp, die wesentlichen Konzepte der Sprache sollten aber deutlich werden. Die parallelen Sprachkonstrukte werden allerdings nicht vorgestellt. Die Ausführung der **MuPAD**-Sprache wird im Artikel „Evaluierung in **MuPAD**" beschrieben.

Kommentare sind eines der wichtigsten Sprachelemente. In **MuPAD** werden sie in `#` geklammert, wie bei `# Kommentar #`.

Konstanten sind numerische Konstanten, die booleschen Konstanten `TRUE` und `FALSE` sowie Zeichenketten. Numerische Konstanten sind ganze Zahlen wie `175`, rationale Zahlen wie `-17/3`, Gleitkommazahlen wie `12.3e-4` und komplexe Zahlen wie `2/5+4*I`. (Bei der Exponentialschreibweise muß ein kleines `e` verwendet werden.) Zeichenketten werden in `"` eingeschlossen, wie bei `"Otto"`.
Numerische Werte können beliebig viele Stellen haben. Die Anzahl der Stellen von Gleitkommazahlen wird mit der Systemvariablen `DIGITS` festgelegt.

Spezielle Konstanten sind `I` für die imaginäre Einheit, sowie `E` und `PI` für e und π. Die Konstante `NIL` dient dazu, Werte von Bezeichnern zu entfernen.

Bezeichner bestehen aus einem alphabetischem Zeichen, gefolgt von beliebig vielen alphabetischem Zeichen oder Ziffern, wie bei `deriv` oder `x_1`. (Der Unterstrich `_` ist ein alphabetisches Zeichen.) Einem Bezeichner kann mit der Zuweisung `:=` ein Wert zugeordnet werden; anfangs besitzt ein Bezeichner keinen Wert. Ein Bezeichner hat keinen Typ, jedem Bezeichner kann ein beliebiger Wert zugeordnet werden.

Funktionsaufrufe haben die Form *Prozedurname*(*Parameter*) oder *Prozedurname*(), wobei die Parameter Folgen von Ausdrücke sind, die durch `,` getrennt werden. Prozedurnamen sind meistens Bezeichner, wie bei `reset()`, `sin(1)` oder `append(l,x)`. Prozedurnamen können aber auch durch Ausdrücke angegeben werden.

Ausdruckssequenzen sind Folgen von Ausdrükke, die durch `,` getrennt werden, wie `2, "x", y, 3/4`.

Listen sind in `[ ]` eingeschlossene geordnete Folgen von Ausdrücke, die durch `,` getrennt werden, wie `[a,b,3/7]`, `[x]` oder `[]`, die leere Liste. Auf die Einträge einer Liste kann mit `[ ]` zugegriffen werden, so ist `l[3]` das dritte Element der Liste `l`. Mit `append(l,7)` wird `7` an `l` angehängt und die neue Liste als Ergebnis geliefert.

Mengen sind wie Listen Folgen von Ausdrücken, die in { } eingeschlossen werden, wie bei `{a, 2.23}`, `{"Lisbet"}` oder `{}`. In einer Menge gibt es keine identischen Elemente, die Elemente sind nicht geordnet.

Tabellen bestehen aus beliebig vielen Gleichungen der Form *Index*=*Wert*, wobei *Index* und *Wert* beliebige Ausdrücke sind. Tabellen werden mit der Funktion `table` erzeugt, wie bei `t:=table()`. Auf die Einträge einer Tabelle kann mit `[ ]` zugegriffen werden, wie bei `t[x]`, `t[7]` oder `t["z"]`. (Hierbei sind `x`, `7` und `"z"` die Indexwerte der jeweiligen Einträge.)

Felder haben beliebig viele Dimensionen und dienen zur Speicherung von Vektoren, Matrizen, Tensoren etc. Felder werden mit der Funktion `array` erzeugt, wie in `v:=array(1..4)` (der Vektor `v[1]` ... `v[4]`), `m:=array(2..6, 3..4)` (die Matrix `m[2,3]`, `m[2,4]` ... `m[6,4]`) oder `t:=array(1..3, 1..3, 1..3)` (ein 3-dimensionales Feld mit den Elementen `t[1,1,1]` ... `t[3,3,3]`). Felder können nachträglich nicht vergrößert werden. Die Elemente von Feldern haben anfangs keinen Wert.

Ausdrücke sind in **MuPAD** Datenstrukturen, keine „Berechnungsanweisungen" wie in den meisten anderen Programmiersprachen. Ausdrücke werden nicht immer direkt *evaluiert* (ausgewertet), die Evaluierung hängt vom Kontext ab. Bei der Evaluierung werden Bezeichner durch ihre Werte *substituiert*, Funktionsaufrufe werden durch ihre Ergebnisse ersetzt und arithmetische Ausdrücke werden

vereinfacht und *sortiert*, damit sie leichter verglichen werden können.
Bezeichner werden bei der Evaluierung eines Ausdrucks normalerweise so weit wie möglich rekursiv substituiert (so daß im Ausdruck keine Bezeichner mehr vorhanden sind, die einen Wert haben). Dies gilt allerdings nicht in Prozeduren: In Prozeduren werden Bezeichner nur einmal durch ihren Wert ersetzt.
Ein Ausdruck besteht aus einem *primitiven Ausdruck* oder aus einem *Operanden* und dessen *Operatoren*. Ein primitiver Ausdruck ist eine Konstante oder ein Bezeichner, ein Funktionsaufruf, eine Ausdruckssequenz, eine Liste oder eine Menge. Operanden sind die **MuPAD**-Operanden wie `+`, `*` und `$`. Die Operanden eines Ausdrucks sind wieder beliebige Ausdrücke. Beispiele für Ausdrücke sind `a+3*PI+sin(x)` oder `[ f(i) $ i=1..3 ]`.
Tabellen und Felder können in Ausdrücken nicht direkt auftreten. Es gibt nur Bezeichner, die ganze Tabellen oder Felder als Werte haben, oder indizierte Bezeichner, wie `t[3]`, die einzelne Elemente als Werte haben. (Solch ein indizierter Bezeichner wird intern als Operator `t` mit Operand `3` aufgefasst.)
Ein Funktionsaufruf mit seinen aktuellen Parametern wird intern genau wie ein Operator mit seinen Operanden aufgefasst. Umgekehrt gibt es zu jedem Operator eine Systemfunktion, die den Operator realisiert. Ein Operator wird ausgeführt, indem die zugehörige Systemfunktion mit den Operanden als Parametern aufgerufen wird.
Durch die obige rekursive Definition werden Bäume definiert: Jeder Ausdruck ist ein Baum, Operatoren und Funktionsaufrufe sind innere Knoten, Operanden und Parameter sind Kinder, primitive Ausdrücke sind Blätter.

Arithmetische Operatoren sind `+`, `-`, `*`, `/`, `^` sowie `div` und `mod`. `div` und `mod` sind ganzzahlige Division bzw. ganzzahliger Rest. `1/a mod b` liefert die modulare Inverse von a modulo b.

Relationale Operatoren sind `=`, `<>` (ungleich), `<`, `<=`, `>` und `>=`.

Logische Operatoren sind `and`, `or` und `not`.

Mengenoperatoren sind `union`, `minus` (Differenz) und `intersect`.

Spezielle Operatoren sind der Bereichsoperator `..`, der Sequenzoperator `$` und der Konkatenationsoperator `.`.

Ein Bereichsausdruck hat die Form *Untergrenze* `..` *Obergrenze*, wobei Unter- und Obergrenze Ausdrücke sind, die ganze Zahlen liefern. Solch ein Bereich repräsentiert die ganzen Zahlen zwischen der Untergrenze und der Obergrenze.
Die einfachste Form des Sequenzausdrucks ist *Ausdruck* `$` *Anzahl*, hierdurch wird eine Ausdruckssequenz mit *Anzahl* Elementen definiert, jedes Element darin ist gleich dem *Ausdruck*. Der Ausdruck `a $ 3` liefert die Sequenz `a,a,a`.
Durch *Ausdruck* `$` *Variable* `=` *Bereich* wird eine Ausdruckssequenz mit den Elementen *Ausdruck* definiert, bei der allerdings die Laufvariable in *Ausdruck* durch den jeweiligen aktuellen Wert aus dem Bereich ersetzt wird. Der Ausdruck `i*PI $ i=0..3` liefert die Sequenz `0,PI,2*PI,3*PI`.
Durch `$` *Bereich* schließlich wird eine Ausdruckssequenz definiert, die aus sämtlichen ganzen Zahlen zwischen der Untergrenze und der Obergrenze des Bereichs besteht. Der Ausdruck `$ 2..5` liefert die Sequenz `2,3,4,5`, `$ -2..-2` die Zahl `-2`. Der Ausdruck `$ 1..0` liefert die *leere Folge* als Ergebnis.
Der Konkatenationsoperator dient zur Konkatenation von Listen und Zeichenketten sowie zur Definition neuer Bezeichner. Durch `[a,b].[c,d]` wird die Liste `[a,b,c,d]` erzeugt, durch `"Mensa"."essen"` die Zeichenkette `"Mensaessen"`. Durch `force."_x"` wird der Bezeichner `force_x` definiert. (Bei der Definition neuer Bezeichner muß das Ergebnis natürlich wieder ein syntaktisch gültiger Bezeichner sein.)

Der Datentyp eines Ausdrucks ist praktisch der interne Typ, in dem ein Ausdruck in **MuPAD** gespeichert wird. Dieser ist bei primitiven Ausdrücken mit dem Ausdruckstyp identisch. Zusammengesetzte Ausdrücke haben den Datentyp `"CAT_EXPR"`. Der Datentyp kann mit der Funktion `cattype` erfragt werden.

Der Ausdruckstyp ist bei primitiven Ausdrücken der Typ des entsprechenden Objektes und bei Operatoren der Typ des Operators. Ausdruckstypen werden als Zeichenketten angegeben und mit den Funktionen `type` und `testtype` erfragt. `type(expr)` liefert den Typ von `expr`, `testtype(expr,string)` liefert genau dann `TRUE`, wenn `expr` den Typ `string` hat. Die Ausdrücke `type(a*b)`, `type(x)` bzw. `type(a+b+c)` liefern `"MULT"` (Produkt), `"CAT_IDENT"` (Bezeichner) bzw. `"PLUS"` (Summe). Der Ausdruck `testtype(x, "CAT_IDENT")` liefert `TRUE`.

Die Operanden eines Ausdrucks können mit der Funktion `op` erfragt werden. `op(expr)` liefert eine Sequenz mit allen Operanden von `expr`.

`op(expr, n)` liefert den n-ten Operanden von `expr`. `op(expr, 0)` liefert den Operator von `expr`. Sei `e:= a+b+sin(3*c)`, dann ist `op(e)` die Sequenz `a, b, sin(c*3)`, `op(e, 1)` ist `a` und `op(e, 0)` ist `_plus`.
Mit `op` können auch rekursiv Operanden von Operanden erfragt werden, Sei `e` wie oben definiert, dann liefert `op(e, [3,1])` den Ausdruck `c*3`, `op(e, [3,1,2])` ist `3` und `op(e, [3,0])` ist `sin`.
Weiter kann mit `op` auch auf die Elemente komplexer Datenstrukturen wie Mengen, Listen und Tabellen zugegriffen werden. So liefert `op([a,b,c], 2)` das zweite Element `b` der Liste.

Die Anzahl der Operanden eines Ausdrucks kann mit der Funktion `nops` erfragt werden. Der Ausdruck `nops(a+17+3*c)` liefert den Wert 3 (für die 3 Operanden `a`, `17` und `3*c`).

Die Substitution von Teilausdrücken geschieht mit den Funktionen `subs`, `subsex` und `subsop`. Mit einem Aufruf der Form `subs(expr, old=new)` wird der Teilausdruck `old` in `expr` durch `new` ersetzt. Der Teilausdruck `old` muß dabei vollständig sein, es können nicht einzelne Operanden eines Operators ersetzt und andere beibehalten werden. So kann im obigen Ausdruck `e` durch `subs(e, 3*c=d)` zwar `3*c` durch `d` ersetzt werden, mit `subs` kann aber nicht der Teilausdruck `a+b` ersetzt werden.
Mit `subsex` können auch unvollständige Teilausdrücke ersetzt werden, der Aufruf ist dabei derselbe wie bei `subs`. So liefert `subs(e, a+b=d)` den Ausdruck `d+sin(3*c)`.
Mit `subsop(expr, n=new)` wird der n-te Operand von `expr` durch `new` ersetzt. `subsop(e, 3=d)` liefert den Ausdruck `a+b+d`. Statt einer einfachen Zahl n können wie bei `op` auch Pfade zu „inneren“ Teilausdrücken angegeben werden.

Anweisungen sind Ausdrücke, Zuweisungen oder bedingte Anweisungen. Bei interaktiver Eingabe muß eine Anweisung immer durch `;` oder `:` abgeschlossen werden, damit sie ausgeführt wird. Bei `:` wird das Ergebnis der Anweisung nicht am Bildschirm ausgegeben.

Anweisungssequenzen sind Folgen von Anweisungen, die durch `;` oder `:` getrennt werden. (Bei `:` wird das Ergebnis einer Anweisung auf interaktiver Ebene nicht ausgegeben.)

Zuweisungen binden normalerweise Werte an Bezeichner, wie bei `a:= 12+x`. Sie haben die Form *Bezeichner* `:=` *Ausdruck*. Nach einer Zuweisung von `NIL`, wie bei `a:= NIL`, hat ein Bezeichner dagegen keinen Wert mehr.
Statt eines einfachen Bezeichners kann auf der rechten Seite einer Zuweisung auch ein *indizierter Bezeichner* wie `l[2]` (ein Listen-, Tabellen- oder Feldelement) oder ein durch den Konkatenations-Operator erzeugter Bezeichner wie `x.2` stehen.
Das Ergebnis der Zuweisung ist normalerweise die linke Seite der Zuweisung. Bei einer Zuweisung auf `NIL` ist das Ergebnis dagegen der Bezeichner selber. Das Ergebnis von `a[3]:= 12+x*0+4;` ist `16`, bei `b:= NIL` ist das Ergebnis `b`.
Bei System-Variablen bewirkt eine Zuweisung von `NIL` das „Zurücksetzen“ der Variablen auf ihren Default-Wert, diese Bezeichner haben also immer einen Wert.

If-Anweisungen haben die Form

```
if Bedingung then
    Anweisungssequenz
elif Bedingung then
    Anweisungssequenz
...
else
    Anweisungssequenz
end_if
```

Die Bedingungen müssen boolesche Ausdrücke sein. Liefert die erste Bedingung `TRUE`, so werden die Anweisungen im Then-Teil ausgeführt, andernfalls wird die zweite Bedingung im Elif-Teil geprüft. Liefert diese `TRUE`, so werden die zugehörigen Anweisungen im Elif-Teil ausgeführt. Es können beliebig viele Elif-Teile folgen, deren Bedingungen nacheinander überprüft werden. Liefert keine der Bedingungen `TRUE`, so werden die Anweisungen im Else-Teil ausgeführt. Die Elif-Teile und der Else-Teil können auch fehlen. Beispiele sind `if x > y then x else y end_if` oder `if a mod 3 = 0 then a:= a+1; return(a) end_if`.

For-Schleifen haben zwei unterschiedliche Formen. Bei der gewohnten Form wird ein Zahlenbereich durchlaufen:

```
for Bezeichner from Anfang to Ende
step Schrittweite do
    Anweisungssequenz
end_for
```

Der Bereich *Anfang* bis *Ende* wird mit der Schrittweite *Schrittweite* durchlaufen. Bei jedem Durchlauf wird die Laufvariable *Bezeichner* auf den aktuellen Wert gesetzt und die Anweisungssequenz

ausgeführt. Mit `for i from 2 to 8 step 2 do print(i) end_for` werden die Werte `2, 4, 6, 8` ausgegeben.
Die Schrittweite kann auch fehlen, in diesem Fall wird 1 als Schrittweite verwendet. Mit `for i from 1 to 3 do a.i:= i end_for` werden die Zuweisungen `a1:=1`, `a2:=2` und `a3:=3` ausgeführt.
Statt der Form *Anfang* `to` *Ende* kann auch die Form *Ende* `downto` *Anfang* verwendet werden, in diesem Fall wird die Schleife rückwärts durchlaufen und die Laufvariable jeweils um die Schrittweite vermindert. Mit `for j from 4 downto 2 do t[j]:= t[j-1] end_for` werden die Zuweisungen `t[4]:= t[3]`, `t[3]:= t[2]` und `t[2]:= t[1]` in der angegebenen Reihenfolge ausgeführt.
Bei der zweiten Form der For-Schleife werden alle Operanden eines Ausdrucks durchlaufen:

`for` *Bezeichner* `in` *Ausdruck* `do`
    Anweisungssequenz
`end_for`

Hier wird dem Bezeichner der Reihe nach jeder Operand des Ausdrucks zugewiesen, damit wird dann jeweils der Schleifenrumpf ausgeführt. Mit `for t in a+b+c do print(t) end_for` werden der Reihe nach die Werte von `a`, `b` und `c` ausgegeben.

While-Schleifen haben die Form

`while` *Bedingung* `do`
    Anweisungssequenz
`end_while`

Hier wird die Anweisungssequenz ausgeführt, solange die Bedingung `TRUE` liefert. Die Bedingung wird vor der Anweisungssequenz ausgewertet.

Repeat-Schleifen haben die Form

`repeat`
    Anweisungssequenz
`until` *Bedingung* `end_repeat`

Hier wird die Anweisungssequenz ausgeführt, solange die Bedingung `FALSE` liefert. Die Bedingung wird erst am Ende der Anweisungssequenz ausgewertet.

Schleifen beenden kann man mit den Anweisungen `next` und `break`. Bei `next` wird die Abarbeitung einer Anweisungssequenz beendet und der nächste Schleifendurchlauf gestartet, bei `break` wird eine Schleife ganz abgebrochen.

Case-Anweisungen haben die Form

`case` *Ausdruck*
    `of` *Wert1* `do` *Anweisungssequenz*
    `of` *Wert2* `do` *Anweisungssequenz*
    ...
    `otherwise` *Anweisungssequenz*
`end_case`

Hierbei wird der Ausdruck der Reihe nach mit den Ausdrücken *Wert1*, *Wert2* etc. verglichen. Sind die Ausdrücke gleich, so werden alle nachfolgenden Anweisungssequenzen der Case-Anweisung ausgeführt. (Also auch die Sequenzen der restlichen Of-Teile und des Otherwise-Teiles!) Hat der Ausdruck keinen der Werte der Of-Teile, so werden nur die Anweisungen des Otherwise-Teiles ausgeführt. Der Otherwise-Teil kann auch fehlen.
Mit dem Kommando `break` kann man an das Ende einer Case-Anweisung springen, die restlichen Anweisungen in der Case-Anweisung werden dann nicht mehr ausgeführt. Mit dem Kommando `next` kann der Vergleich des Ausdrucks mit den Werten der Of-Teile fortgeführt werden: Wie zum Beginn der Case-Anweisung wird dann ab der `next`-Anweisung der Ausdruck weiter mit den Werten der Of-Teile verglichen und die Anweisungen nach einem passenden Wert ausgeführt.

Prozeduren haben die Form

`proc (` *Parameter* `)`
    `local` *lokale Variablen* ;
    `option` *Optionen* ;
`begin`
    Anweisungssequenz
`end_proc`

Die formalen Parameter sind eine (evtl. leere) Folge von Bezeichnern, die durch `,` getrennt werden. Die lokalen Variablen sind ebenfalls eine Folge von Bezeichnern, die mit `,` getrennt werden. Die Optionen sind durch `,` getrennte Schlüsselworte; möglich sind `hold` und `remember`. Die Deklaration der lokalen Variablen und Optionen kann auch fehlen.
Eine Prozedur hat keinen Namen per se. Erst durch eine Zuweisung wird einem Bezeichner eine Prozedur als Wert zugewiesen. Dieser Bezeichner ist dann der „Name“ der Prozedur, so wie man dies z.B. aus Pascal kennt.

Durch

```
max:= proc(x,y)
      begin
        if x > y then
          x
        else y
        end_if
      end_proc
```

wird die Prozedur `max` definiert, die das Maximum zweier Zahlen berechnet. Durch die Zuweisung `f:= max` erhält `f` dieselbe Prozedur wie `max` zugewiesen. (Manchmal werden vorteilhaft auch Prozeduren ohne Namen verwendet.) Prozeduren sind in **MuPAD** Daten wie gewöhnliche Ausdrücke oder Listen auch. Beim Aufruf einer Prozedur werden zuerst den formalen Parametern Kopien der aktuellen Parameter als Werte zugewiesen. (Es wird also immer ein „*call by value*" durchgeführt, ein „call by reference" gibt es nicht.) Danach wird die Anweisungssequenz des Prozedurrumpfes ausgeführt. Die lokalen Variablen haben dabei Anfangs noch keine Werte.
Eine *freie Variable* innerhalb einer Prozedur (also eine Variable, die weder Parameter noch lokale Variable ist) ist automatisch eine globale Variable. Eine Prozedur kann auch innerhalb einer anderen Prozedur definiert werden. In diesem Fall können die lokalen Variablen der äußeren Prozedur globale Variablen der inneren sein, wie bei

```
g:= proc(x)
      local y, f;
    begin y:= x;
      f:= proc(x)
          begin
            x*y
          end_proc;
      f(x)
    end_proc
```

Hier ist die innere Prozedur `f` lokale Variable der äußeren Prozedur `g`, `y` ist lokale Variable von `g` und globale Variable von `f`. `g` berechnet (recht umständlich) `x^2`.
Bezüglich der freien Variablen gilt bei *dynamisches Scoping*, globale Variablen werden also zur Laufzeit gebunden. (In einer Prozedur wird diejenige globale Variable verwendet, die zur Laufzeit zuletzt in einer der aufrufenden Prozeduren definiert wurde.)
Normalerweise liefert eine Prozedur den Wert der zuletzt ausgeführten Anweisung als Rückgabewert. Eine Prozedur kann aber auch mit der Funktion `return` verlassen werden. Dabei sind die Parameter von `return` dann die Rückgabewerte der Prozedur.

Die obige Prozedur `max` kann auch wie folgt definiert werden:

```
max:= proc(x,y)
      begin
        if x > y then
          return(x)
        end_if;
        y
      end_proc
```

Der Rückgabewert einer Prozedur kann eine Ausdruckssequenz sein, damit kann eine Prozedur also mehrere Werte gleichzeitig zurückgeben.

Optionen für Prozeduren sind wie gesagt `hold` und `remember`. Normalerweise werden die aktuellen Parameter von Prozeduren vor der Ausführung der Prozedur evaluiert, dies wird mit der Option `hold` verhindert. Die Option `remember` bewirkt, daß eine Prozedur sich alte Ergebnisse „merkt" und bei Bedarf einfach diese Resultate zurückliefert, statt die Prozedur erneut auszuführen. Dadurch können insbesondere Rekursionen sehr beschleunigt werden. Der Preis dafür ist der zusätzliche Speicherplatz für die Ergebnisse.

Environment-Variablen sind Systemvariablen, die nach der Ausführung einer Prozedur automatisch wieder auf ihren alten Wert gesetzt werden. Die Prozedur

```
pi:= proc()
       local DIGITS;
     begin
       DIGITS:=200;
       float(PI)
     end_proc
```

gibt π auf 200 Stellen genau aus. Nach dem Aufruf `pi()` hat die Systemvariable `DIGITS` den gleichen Wert wie vor dem Aufruf, sie wird also nur innerhalb der Prozedur `pi` verändert. Dazu muß die Environment-Variable allerdings als `local` deklariert werden!

Evaluierung in MuPAD

Klaus Gottheil

Zusammenfassung

Evaluierung ist in der **MuPAD**-Terminologie die Ausführung eines **MuPAD**-Ausdrucks durch den **MuPAD**-Interpreter. Man kann mit einigem Recht also sagen, daß die Evaluierung der „eigentliche Zweck“ des Interpreters ist. Was bedeutet „Ausführung“ in diesem Zusammenhang?

1 MuPAD-Ausdrücke

Bevor die Evaluierung angegangen werden kann, muß der Begriff des *Ausdrucks* präzisiert werden: Ein **MuPAD**-Ausdruck ist — stark vereinfacht ausgedrückt — ein Baum.

Die Blätter eines „Ausdrucks-Baumes“ sind im wesentlichen Identifier (d.h. Bezeichner von Variablen), Konstanten (Zahlen, Boolesche Konstanten, Strings) und spezielle Konstrukte, die zur Identifikation von Systemfunktionen und Prozeduren benötigt werden.

Identifier haben außer der Tatsache, daß sie einen Namen haben, noch eine überaus nützliche Eigenschaft: Es können Werte an Identifier *gebunden* werden, d.h. ein Ausdruck kann einem Identifier zugeordnet werden. (Dies geschieht u.a. durch die Zuweisung „`:=`“.)

Durch die Evaluierung kann ein Identifier, an den ein Wert gebunden ist, durch diesen Wert (d.h. durch diesen Baum) ersetzt werden. Durch die Anweisungen

```
a:=b; b:=2;
```

z.B. wird an die Variable `a` der Wert `b` und an `b` der Wert `2` gebunden.

Die inneren Knoten des Baumes stellen Funktionsaufrufe oder komplexe Datenstrukturen wie Listen, Mengen oder Tabellen dar. Auf die komplexen Datenstrukturen wird im folgenden nicht weiter eingegangen, um die Darstellung nicht zu unübersichtlich werden zu lassen.

Bei Funktionsaufrufen identifiziert der erste Teilbaum die Funktion und die restlichen Teilbäume die aktuellen Parameter des Aufrufes. Ein solcher Knoten kann in „funktionaler Schreibweise“ also geschrieben werden als

$$F(P_1, \ldots, P_n)$$

wobei die Funktion F und die Parameter P_i für Teilbäume stehen können (nicht nur für Blätter). In Lisp würde man diesen Knoten als $(F P_1 ... P_n)$ schreiben.

Die Anweisungen der **MuPAD**-Sprache sind übrigens sämtlichst Funktionen und sind dem Benutzer auch als Funktionen zugänglich (z.B. als `_assign` für die Zuweisung „`:=`“ oder als `_if` für die „`if then else end_if`“-Anweisung). Zu jeder **MuPAD**-Anweisung gibt es eine sogenannte *„Underline“-Funktion* (so genannt, weil die Namen dieser Funktionen per Konvention alle mit einem Underline „`_`“ beginnen). Die **MuPAD**Anweisung

```
if a < b then a+2 else b end_if
```

etwa lässt sich genauso schreiben als

```
_if(_less(a,b), _plus(a,2), b)
```

wobei allerdings die Lesbarkeit sichtlich leidet (Lisp-gewohnte Klammer-Künstler werden keine Probleme haben).

2 Evaluierung von Ausdrücken

Durch die *Evaluierung* wird ein **MuPAD**-Ausdruck in einen anderen überführt, es wird also ein Baum durch einen anderen ersetzt. Evaluierung ist in **MuPAD** ein rekursiver Prozess:

Ein Blatt, welches kein Identifier ist, wird durch die Evaluierung nicht verändert. Ist an einen Identifier kein Wert gebunden, so wird dieser ebenfalls nicht verändert. Ist an einen Identifier ein Wert gebunden, so wird dieser Wert (d.h. dieser Ausdrucks-Baum) evaluiert und als Ergebnis zurückgeliefert. Nach den Anweisungen

```
a:=b; b:=c; c:=2;
```

etwa wird `a` bei interaktiver Eingabe normalerweise zu `2` evaluiert. (Auf die Einschränkungen „interaktiv“ und „normalerweise“ wird später noch genauer eingegangen.)

Ein innerer Knoten eines Ausdrucks stellt wie gesagt einen Funktionsaufruf dar. Zuerst wird der erste Teilbaum des Knotens evaluiert. Ergibt die Evaluierung des ersten Teilbaumes eine Funktionsdefinition, so wird die entsprechende Funktion mit den restlichen Teilbäumen des Knotens als Parameter ausgeführt. Bei der Evaluierung des Aufrufs

```
sin(a+b+2)
```

etwa wird der Identifier `sin` zur Funktionsdefinition für die Systemfunkion „Sinus“ evaluiert und die Funktion dann mit dem Ausdruck `a+b+2` (oder äquivalent mit `_plus(a,b,2)`) als Parameter aufgerufen. Ergibt die Evaluierung des ersten Teilbaumes eines Funktionsaufrufes einen Identifier, so liegt eine Funktion vor, an die keine Definition gebunden ist. In diesem Fall werden auch die restlichen Teilbäume evaluiert und der neue Baum in Form eines „formalen Funktionsaufrufes“ zurückgeliefert. Ist `f` z.B. ein Identifier, an den kein Wert gebunden ist, so wird

```
f(4+3)
```

zu `f(7)` evaluiert, da `_plus(4,3)` zu `7` evaluiert wird. Wird der erste Teilbaum des Knotens weder zu einer Funktionsdefinition noch zu einem Identifier evaluiert, so wird die Evaluierung mit einem Fehler abgebrochen. So ist z.B. der Ausdruck

```
"_plus"(2,3)
```

ungültig, da der String `"_plus"` nicht zur Funktionsdefinition für die Summe evaluiert wird.

3 Ausführung von MuPAD-Prozeduren

Eine Funktion wird wie gesagt mit den Teilbäumen des Aufrufs als Parameter ausgeführt. Ist die Funktion eine Prozedur, die in der **MuPAD**-Sprache definiert wurde, so gibt es zwei Formen für den Aufruf: Normalerweise werden die Parameter-Ausdrücke evaluiert und die Prozedur mit diesen Parametern ausgeführt. Die Anweisungen

```
f:= proc(x) begin
   x*2
end_proc;
a:=2;
```

etwa definieren die Funktion `f` und weisen danach `a` den Wert `2` zu. Der Aufruf `f(a)` liefert dann wie erwartet den Wert `4`.
Wird dagegen bei der Definition der Prozedur die Option „`hold`“ angegeben, so werden ihre Parameter vor der Ausführung nicht evaluiert. Die Anweisungen

```
g:= proc(x) option hold; begin
   x*2
end_proc;
a:=2;
```

definieren die Funktion `g` und weisen `a` wieder den Wert `2` zu. Der Aufruf `g(a)` liefert dann den Ausdruck `a*2` als Ergebnis: Der Parameter `a` wird nicht evaluiert, die Funktion `g` liefert als Ergebnis den Identifier `a` multipliziert mit `2`.
Die Ausführung der Prozedur geschieht in drei Schritten: Zuerst werden die aktuellen Parameter an die Identifier der formalen Parameter gebunden. Danach werden die mit „`local`“ deklarierten lokalen Variablen der Prozedur eingerichtet. Als letzter Schritt wird die Prozedur wie gehabt evaluiert. Der zuletzt evaluierte Wert ist dann der Rückgabewert des Aufrufes.
Die lokalen Variablen sind praktisch „frische“ Identifier, an die zunächst keine Werte gebunden sind. Sind vor dem Aufruf der Prozedur Identifier mit gleichem Namen vorhanden, so sind deren Werte in der Prozedur nicht mehr sichtbar. Bei der Evaluierung der Identifier werden stattdessen die Werte der lokalen Variablen verwendet. Alle anderen Identifier in der Prozedur werden wie globale Variablen behandelt, wobei auf die aktuell zur Laufzeit gültigen Werte der Identifier zugegriffen wird.
In folgender Prozedur `f` z.B. ist `x` der Parameter und `y` eine lokale Variable, während `z` eine globale Variable ist, da der Identifier `z` weder als Parameter noch als lokale Variable deklariert ist:

```
f:= proc(x) local y; begin
   y:=2; x*y*z
end_proc;
```

Nach den Zuweisungen

```
y:=3; z:=NIL;
```

ist an `y` der Wert `3` und an `z` (durch die Zuweisung `z:=NIL`) kein Wert gebunden. Da im Ausdruck `x*y*z` die lokale Variable `y` zu `2` evaluiert wird, liefert der Ausdruck `f(2)` danach den Wert `z*4`. Nach der Zuweisung

```
z:=3;
```

liefert `f(2)` dagegen den Wert `12`.
Die beschriebene Vorgehensweise hat eine Konsequenz, die man auf neudeutsch *dynamic scoping* nennt: Bei globalen Variablen wird derjenige Identifier verwendet, der zur *Ausführungszeit* sichtbar ist. In einer „normalen“ prozeduralen Programmiersprache wie Pascal wird dagegen diejenige Variable verwendet, die statisch durch den Programmtext (durch

die äußere Prozedur) vorgegeben ist (hierbei spricht man von *lexical scoping*). In der folgenden Prozedur g etwa wird die lokale Variable z definiert, die ja global bezüglich der obigen Funktion f war:

```
g:= proc() local z; begin
   z:= 4; f(2)
end_proc;
```

Der Ausdruck g() liefert damit immer den Wert 16, wobei der Wert von z beim Aufruf von g gleichgültig ist. Während der Evaluierung von f(2) in der Prozedur g wird die lokale Variable z aus g verwendet (die ja bei der Definition von f noch garnicht „bekannt“ war).

4 Ausführung von Systemfunktionen

Bei den System-Funktionen hängt die Evaluierung der Parameter von der jeweiligen Funktion ab. Die meisten Funktionen evaluieren alle ihre Parameter vor der Ausführung der „eigentlichen“ Funktion. Einige Funktionen evaluieren dagegen ihre Parameter nicht oder nur zum Teil.
Die Zuweisung (_assign) z.B. evaluiert ihren ersten Parameter (die rechte Seite der Zuweisung) nicht, falls es sich um einen Identifier handelt. Ansonsten würde die Variable b nach den Anweisungen

```
a:=b; a:=2;
```

den Wert 2 haben — eine recht verwirrende Situation. Die linke Seite der Zuweisung wird dagegen evaluiert. Durch die Anweisungen

```
b:=2; a:=b;
```

wird bei interaktiver Eingabe an a der Wert 2 gebunden, nicht der Wert b.
Ein anderes Beispiel für eine unvollständige Evaluierung von Parametern ist die Prozedur-Definition mittels „proc ... end_proc“ (intern _procdef genannt). Hierbei wird die Prozedur-Definition nicht evaluiert. Es wäre schon sehr verwirrend, wenn durch die Anweisungen

```
x:=13;
f:= proc(x) begin x*2 end_proc;
```

die (natürlich fehlerhafte) Prozedurdefinition

```
proc(13) begin 13*2 end_proc;
```

erzeugt würde.

Die Ausführung der System-Funktionen hängt natürlich von der jeweiligen Funktion ab. Im folgenden werden nur einige Hinweise zur Ausführung von Funktionen wie „+“, „*“ und „^“ gegeben, die algebraische Operatoren repräsentieren:
Bei diesen Funktionen werden die evaluierten Parameter zuerst algebraisch vereinfacht und danach in eine kanonische Form gebracht, damit der Ausdruck leichter mit anderen Ausdrücken verglichen werden kann.
Die Vereinfachung geschieht nach einfachen Regeln. Zuerst wird ggf. die Assoziativität der Operation ausgenutzt, um den Ausdrucks-Baum zu „*flatten*“, d.h. flacher zu machen. Geschachtelte Ausdrücke der Form

```
_plus(a,_plus(b,3))
```

können dabei wegen der Assoziativität der Addition zu

```
_plus(a,b,3)
```

umgeformt werden.
Danach werden Regel wie z.B. $a+0 = a$ oder $a \times 0 = 0$ angewandt, es werden also 0'en aus Summen entfernt und eine Multiplikation mit 0 gibt 0. Die Anwendung dieser Regeln ist zum Teil — diese Schattenseite kann leider nicht verschwiegen werden — recht gefährlich. Eine solche Regel der gefährlichen Art ist z.B. $a/a = 1$; denn stellt sich später heraus, daß a den Wert 0 annehmen kann, so ist die Ersetzung durch 1 natürlich falsch. Wird z.B. die Variable a definiert durch

```
a:= b*(c+1)/b;
```

wobei b noch keinen Wert hat, so erhält a durch die Vereinfachung den Wert c+1. Wird danach

```
b:=0;
```

definiert, so ist der Wert von a fälschlicherweise weiterhin wohldefiniert c+1, die Variable a „weiß“ nichts mehr von der Variablen b aus ihrer Definition.
Man muß hier leider mit der Gefahr leben, sonst würden Ausdrücke schnell „explodieren“. (Ein explodierender Ausdruck ist zwar für den Benutzer physisch ungefährlich, kann aber das **MuPAD**-System unbrauchbar machen und somit psychische Schäden hervorrufen.) Man sollte sich als Benutzer aber über die Gefahr der Vereinfachungs-Regeln klar sein.
Mit zur Vereinfachung gehört weiter die Berechnung von Konstanten, also Umformungen wie

```
_plus(3,a,4) ⟶ _plus(a,7)
```

Nach der Vereinfachung wird der Ausdruck normalisiert. Dazu werden z.B. bei Summen und Produkten Konstanten an das Ende des Ausdrucks „geschoben“ und die restlichen Teilausdrücke sortiert. Der Ausdruck

```
z*3*a1*a
```

wird durch die Sortierung z.B. umgeformt in

```
a*z*a1*3
```

Die Sortier-Reihenfolge ist dabei für den Benutzer schwer vorhersagbar, aber zum Glück meist auch nicht relevant.
Der Ausdruck `b*a` wird durch die Sortierung der Terme z.B. immer in `a*b` umgeformt. Daher muß eine Summe oder ein Produkt daher als kommutativ aufgefaßt werden. Will man Matrizen oder nicht-kommutative Operatoren symbolisch multiplizieren, so muß man dafür eigene Funktionen verwenden.
Der so berechnte Ausdruck wird dann als Ergebnis der Evaluierung zurückgeliefert.

5 Die Substitutionstiefe

Bei der Evaluierung von Identifiern spielt die *Substitutionstiefe* eine wichtige Rolle. Was ist darunter zu verstehen?
Betrachtet man den ursprünglichen Ausdruck vor der Evaluierung, so ist die Substitutionstiefe 0. Wird nun während der Evaluierung ein Identifier durch seinen Wert ersetzt, so wird die Substitutionstiefe um 1 erhöht und der Wert des Identifiers mit dieser neuen Substitutionstiefe evaluiert. Nach der Evaluierung des Wertes wird die alte Substitutionstiefe wieder angenommen.
Die Substitutionstiefe gibt also die Rekursionstiefe der Evaluierung von Identifiern an. Die Evaluierung kann dabei durch die Vorgabe einer maximalen Substitutionstiefe gesteuert werden. Wird die maximale Substitutionstiefe erreicht, so wird ein Identifier nicht weiter evaluiert.
Interaktiv eingegebene Ausdrücke werden vollständig evaluiert, wobei „vollständig“ heißen soll, daß der Benutzer die maximale Substitutionstiefe vorgeben kann. Der Normalfall wird hierbei aber sein, daß Ausdrücke so tief evaluiert werden, bis keine „ungebundenen“ Identifier mehr vorhanden sind. Nach den Zuweisungen

```
a:=b; b:=c; c:=13;
```

wird die Variable `a` bei interaktiver Eingabe zu `13` evaluiert, ganz wie erwartet.

In **MuPAD**-Proceduren beträgt die maximale Substitutionstiefe 1, d.h. Identifier werden durch ihre Werte ersetzt, diese Werte werden aber nicht weiter evaluiert. In folgender Prozedur `f` z.B. wird die lokale Variable `z` zu `y` evaluiert und *nicht* zum Parameter `x`:

```
f:= proc(x) local y, z; begin
  z:=y; y:=x; z
end_proc;
```

Der Aufruf `f(13)` liefert also das Ergebnis `y` und nicht `13`, wie man vielleicht erwarten würde.

6 Steuerung der Substitutionstiefe

Die maximale Substitutionstiefe kann vom Benutzer mit den beiden Variablen `LEVEL` und `MAXLEVEL` explizit gesetzt werden.
Bei der Evaluierung wird (soweit nötig) bis zur Tiefe `LEVEL` substituiert. Nach den Anweisungen

```
a:=b; b:=c; c:=13;
```

wird `a` bei interaktiver Eingabe normalerweise zu `13` evaluiert. Wird dagegen `LEVEL` auf `2` gesetzt, so wird `a` zu `c` und `b` zu `13` evaluiert.
Die Variable `LEVEL` kann übrigens in Prozeduren auch auf einen anderen Wert als 1 gesetzt werden. In der Prozedur

```
f:= proc(x) local y, z; begin
  LEVEL:=2; z:=y; y:=x; z
end_proc;
```

wird `z` zum Parameter `x` evaluiert und nicht zu `y`. Der Aufruf `f(13)` liefert also `13` und nicht `y`.
Die Variable `MAXLEVEL` dient zur Erkennung unendlicher Rekursionen, wie bei der scheinbar harmlosen Anweisungsfolge

```
a:= a+1; a;
```

Falls `a` vor der Zuweisung keinen Wert hatte, so wird in der zweiten Anweisung `a` zu `a+1` evaluiert, dies wiederum zu `a+2`, dies zu `a+3` und so weiter ad infimum. Um solche Situationen zu erkennen, wird bei der Evaluierung eine Substitutionstiefe von `MAXLEVEL` als Fehler angenommen und die Evaluierung mit der Meldung „Recursive Definition“ abgebrochen. Das ist natürlich nur eine Heuristik, die im Einzelfall falsche Fehlermeldungen liefern kann, sie wird aber in den allermeisten Fällen richtig liegen.

7 Feinsteuerung der Evaluierung

Bisher wurden bereits die Prozedur-Option `hold` sowie die Variablen `LEVEL` und `MAXLEVEL` zur Steuerung der Evaluierung vorgestellt. Eine noch genauere Kontrolle über die Evaluierung gestatten die Systemfunktionen „`hold`“, „`val`“ und „`level`“.
Die Funktion `hold` kann einen beliebigen Ausdruck als Parameter erhalten. Dieser wird nicht evaluiert und als Ergebnis des Aufrufs zurückgeliefert. Mit `hold` wird also die Evaluierung eines Ausdrucks verhindert. Der Ausdruck

```
hold(a+2+0)
```

wird z.B. zu

```
a+2+0
```

evaluiert und nicht weiter vereinfacht.
Die Funktion `val` hat ebenfalls einen Ausdruck als Parameter, der genau wie bei `hold` nicht evaluiert wird. Vielmehr werden in dem Ausdruck alle Identifier, an die ein Wert gebunden ist, durch diesen Wert substituiert. Der so erzeugte Ausdruck wird dann nicht weiter evaluiert sondern direkt als Ergebnis von `val` zurückgeliefert. Nach den Zuweisungen

```
a:=b; b:=2;
```

wird z.B. der Ausdruck

```
val(a+3+0)
```

zu `b+2+0` und der Ausdruck

```
val(b+4)
```

zu `2+4` evaluiert und nicht weiter vereinfacht. Die wichtigste Anwendung von `val` ist der schnelle, direkte Zugriff auf den Wert einer Variablen, ohne daß eine zeitraubende Evaluierung des Wertes erfolgt.
Die Funktion `level` schließlich bekommt einen Ausdruck als ersten Parameter und optional eine ganze Zahl als zweiten Parameter. Ein Ausdruck der Form `level(`a`,`n`)` wird evaluiert, indem der Ausdruck a mit der maximalen Substitutionstiefe n statt mit der „aktuellen“ durch `LEVEL` gegebenen Tiefe evaluiert wird. Die Funktion `level` überschreibt also praktisch temporär die Variable `LEVEL`. Die Variable `MAXLEVEL` bleibt davon unberührt, bei Erreichen der Substitutionstiefe `MAXLEVEL` wird nach wie vor ein Fehler gemeldet. Fehlt der zweite Parameter, so wird als maximale Substitutionstiefe fast unendlich (genauer 2147483647) angenommen.
Auf interaktiver Ebene etwa wird nach den Anweisungen

```
a:=b; b:=c; c:=3;
```

der Ausdruck

```
level(a,1)
```

zu `b` und der Ausdruck

```
level(a,2)
```

zu `c` evaluiert. Die Funktion `level` kann insbesondere auch verwendet werden, um eine Variable in einer Funktion vollständig zu evaluieren.

Ziele der Parallelität in MuPAD

Holger Naundorf

In diesem Beitrag möchte ich einen Überblick über die Forderungen geben, die wir bei der Entwicklung von MuPAD an die Parallelität gestellt haben. In diesem Zusammenhang werde ich dann auch überblicksmäßig die gewählten Lösungen beschreiben. Ich werde jedoch weder auf die derzeitige Version auf der Sequent-Symmetry noch auf konkrete parallele Sprachkonstrukte in MuPAD eingehen.

1 Lösung speicherintensiver Aufgaben

MuPAD wurde von Anfang an für den Zweck konzipiert, Probleme zu lösen, die mit anderen Computeralgebra-Systemen (sowohl parallelen als auch sequentiellen) nicht zu lösen sind, weil bei diesen der Speicher nicht ausreicht und der Hintergrundspeicher zu langsam ist.

Die Idee ist hierbei, die Speicher einer Vielzahl von (vernetzten) Rechnern zu kombinieren, so daß insgesamt ein wesentlich größerer Speicher zur Verfügung steht und nur die Größe dieses gesamten Speichers die speicherbare Datenmenge begrenzt.

Dieses Ziel wird nicht immer mit einem guten Speed up zu verbinden sein, insbesondere, wenn der Speicher des Systems stark ausgelastet ist. Aber es gibt viele Probleme, die ein sequentielles Programm gut lösen könnte, wenn es einige hundert Megabyte echten Hauptspeicher zur Verfügung hätte. In einem solchen Fall würde es uns reichen, wenn die Geschwindigkeit unseres Parallelrechners einer solchen sequentiellen Maschine ebenbürtig wäre.

Um keinen falschen Eindruck aufkommen zu lassen, möchte ich darauf hinweisen, daß ein guter Speed up natürlich auch von uns eine Forderung mit hoher Priorität ist (schon weil nur so einige sehr speicherintensive Aufgaben in vertretbarer Zeit zu lösen sind), sie hat bei uns allerdings nicht höchste Priorität.

Das von uns gesteckte Ziel spaltet sich natürlicherweise wie folgt auf:

1. Ein Rechner darf mehr Daten produzieren, als er in seinem eigenen Speicher halten kann. Das System wird erst dann überfordert, wenn die von allen Prozessoren zusammen produzierten Daten den gesamten Speicher aller Rechner überschreiten. Dies bedeutet also, daß ein Rechner seine eigenen Daten auch in den Speichern anderer Rechner halten kann.

2. Es besteht die Möglichkeit einer globalen unique data representation. Diese wird benötigt, da in einem Computeralgebra-System häufig das exakt gleiche Datum an verschiedenen Stellen benötigt wird, z.B. weil es Teildatum von verschiedenen Daten ist. In der Regel ist es nicht akzeptabel, dieses mehrfach benötigte Datum auch physikalisch mehrfach im Speicher zu halten, da sich der Speicherbedarf eines Systems sonst leicht vervielfachen kann.

   Diese unique data representation wird von den heutigen sequentiellen Computeralgebra-Systemen durchgeführt. An MuPAD stellen wir aber die Forderung, daß auch von mehreren Rechnern benötigte Daten physikalisch nur einmal im Gesamtspeicher gehalten werden dürfen.

Offensichtlich wären diese beiden Forderungen wie auf einem sequentiellen System zu erfüllen, wenn die verwendeten Rechner eine shared memory Maschine bildeten. Aus diesem Grund wurde von uns die Speicherverwaltung Mammut entwickelt, die auf einer beliebigen parallelen Architektur eine shared memory Maschine simulieren kann. Außerdem stellt Mammut die Möglichkeit der globalen unique data representation zur Verfügung.

Durch diesen Lösungsansatz ergeben sich folgende Eigenschaften:

- Es besteht die Möglichkeit, sehr speicherintensive Aufgaben zu lösen, d.h. unsere Hauptforderung ist erfüllt.

- Jede Anpassung von MuPAD an eine Architektur ist nur eine Veränderung von Mammut. Der Kern von MuPAD bleibt immer unverändert. So kann MuPAD z.B., obwohl es für Netzwerke konzipiert ist, leicht auf eine echte shared memory Maschine portiert werden. Die erste echt parallele Version von MuPAD wird auch voraussichtlich auf einer solchen Maschine laufen.

- Mammut, und nicht der Kernprogrammierer, sorgt dafür, daß der vorhandene Speicher immer optimal ausgenutzt wird. Dies geschieht auf einem Netzwerk z.B. dadurch, daß jeder Rechner eine Kopie von einem möglichst großen Teil der Daten, die er für die Rechnung benötigt, in seinem lokalen Speicher hält. Mit steigender Speicherauslastung sinkt natürlich die Anzahl der möglichen Kopien und die Rechenzeit steigt. Wegen der Parallelität ist MuPAD dann aber immer noch wesentlich schneller als ein sequentieller Prozeß, der seinen Sekundärspeicher benutzen muß.

- Daten, die von einem Rechner A zu einem Rechner B geschickt werden, müssen zunächst nur logisch kopiert werden, wofür fast keine Kommunikation benötigt wird. Physikalisch kopiert werden müssen sie erst, wenn B wirklich auf sie zugreift. Da während eines Algorithmus häufig nur auf einen kleinen Teil der vorhandenen Daten zugegriffen wird, müssen möglicherweise große Teile der logisch kopierten Daten niemals übertragen werden. Durch diese Eigenschaft sinkt die Kommunikationsrate im Netz, was einen besseren Speed up ermöglicht.

- Die Gleichheit von Daten x und y kann sehr schnell festgestellt werden, wenn x und y in derselben Speicherzelle gespeichert sind. Diese Art des Vergleichens kann schon bei sequentiellen Systemen einen erheblichen Geschwindigkeitszuwachs ergeben, auf parallelen Systemen kommt noch hinzu, daß dadurch die gesamte Kommunikation entfällt, die entstanden wäre, wenn die zu vergleichenden Daten erst zum entsprechenden Rechner übertragen worden wären.

2 Einfache Parallelisierung

Ein Hauptaspekt von MuPAD ist die Parallelität; Aufgaben, die viel Speicherplatz benötigen, können mit MuPAD nur dann vernünftig gelöst werden, wenn parallel programmiert wird. Um auch den MuPAD-Benutzern, die nicht an paralleles Programmieren gewöhnt sind, die Möglichkeit zu geben, parallele MuPAD-Programme zu schreiben, sollte es kaum einen Unterschied zwischen sequentiellen und parallelen Programmen geben. Insbesondere bedeutet dies, daß sich der Benutzer weder um das Problem, daß ihm nur endlich viele Prozessoren zur Verfügung stehen, noch um die Zuweisung von Aufgaben an Prozessoren zu kümmern braucht.

Aus diesem Grund werden dem Benutzer in der MuPAD-Programmiersprache Sprachkonstrukte angeboten, mit denen er eine beliebige Anzahl von Tasks erzeugen kann. Der Kern von MuPAD übernimmt dann die Aufgabe, diese Tasks auf den tatsächlich vorhandenen Prozessoren zu verteilen.
Diese Konstrukte haben gleichzeitig die Eigenschaft, daß sie sich automatisch synchronisieren. Hierdurch stehen dem Benutzer zwar nicht alle denkbaren Formen der Parallelität zur Verfügung, das Programmieren vereinfacht sich aber erheblich.

3 Anpassung der Parallelität an die Architektur

Bei der obigen Form der Parallelität kümmert sich MuPAD um das schwierige Problem der Verteilung von Aufgaben auf Prozessoren. Solange es sich bei der zugrundeliegenden Architektur um eine echte shared memory Maschine oder ein Netzwerk mit sehr hoher Kommunikationsleistung handelt, kann MuPAD dies auch durchführen, ohne den möglichen Speed up stark zu verschlechtern.
Sobald die zugrundeliegende Architektur aber ein Netzwerk mit einem großen Durchmesser oder mit schlechter Kommunikationsleistung ist, wird MuPAD diese Aufgabe oft nur sehr unbefriedigend lösen.
Deshalb gibt es für den MuPAD-Benutzer eine weitere Möglichkeit der Parallelität. Der Benutzer hat hierbei eine während einer Rechnung konstante Anzahl von normalen MuPAD-Prozessen zu seiner Verfügung, denen er gezielt Aufgaben und Daten zuweisen kann. Die MuPAD-Prozesse bilden für den MuPAD-Benutzer eine shared memory Maschine, bei der jeder Prozeß normalerweise auf seinen lokalen Daten arbeitet, zusätzlich aber auf einen für alle Prozesse zugänglichen globalen Speicher zugreifen kann. Weiterhin stehen dem Benutzer komfortable Befehle zur Verfügung, mit denen ein Prozeß anderen Prozessen gezielt Daten übermitteln kann.
Der Benutzer hat dann selbst die Aufgabe, dafür zu sorgen, daß alle Prozessoren arbeiten und die richtigen Daten erhalten, so daß das Programmieren auf dieser Ebene wesentlich schwieriger ist.
Die beiden beschriebenen Formen der Parallelität können sinnvoll gemeinsam genutzt werden, z.B. wenn ein Netzwerk von echten shared memory Maschinen vorliegt. Dann kann die Parallelisierung auf den shared memory Maschinen mit Hilfe der einfachen Parallelität stattfinden, während das Verteilen der Aufgaben unter den einzelnen Maschinen vom MuPAD-Benutzer übernommen wird.

Software Preis

Am 6. 10. 1992 wurde in Berlin Angehörigen zweier Arbeitsgruppen der Uni-GH Paderborn von Prof. Dr. Rainer Ortleb, dem Bundesminister für Bildung und Wissenschaft, der Deutsch-Österreichische Hochschul-Software-Preis überreicht. Dieser Softwarepreis wird von der Akademischen Software Kooperation (ASK) in Karlsruhe alljährlich für qualitativ hochwertige Software im Bereich der Lehre vergeben. In diesem Jahr gab es insgesamt 15 Preisträger (bei 185 eingereichten Programmen) aus den verschiedensten Fachgebieten; zwei der ausgezeichneten Programme kommen aus Paderborn: *1st Grade* und SIMEX.

1st Grade

Den beiden Studenten Aurel Balmoŝan und Curd Bergmann und den wissenschaftlichen Mitarbeiten Franz Höfting und Dr. Egon Wanke (alle Fachbereich 17, Arbeitsgruppe Lengauer) wurde diese Auszeichnung im Bereich Informatik für das Programm *1st Grade* verliehen.
Das *1st Grade* System wurde entwickelt zur Unterstützung der Ausbildung im Bereich Algorithmen, speziell Graphalgorithmen. Das Programm besteht im wesentlichen aus zwei Komponenten, einem Graphen-Editor und der *1st Grade* Programmiersprache.
Der Grapheditor bietet die Möglichkeit, beliebige Graphen aus Knoten und gerichteten oder ungerichteten Kanten zu generieren und zu modifizieren. An alle Graphenobjekte können beliebige Informationen geschrieben werden, die in den Algorithmen verarbeitet werden sollen. Das Grapheditor-Fenster dient zugleich als Standardausgabe für Applikationen. Die *1st Grade* Sprache ist eine Programmiersprache, deren Syntax sich eng an der bekannter Sprachen wie C, Modula oder Oberon orientiert, die jedoch zahlreiche Konstrukte enthält, die üblicherweise in den recht abstrakten Algorithmensprachen bekannter Lehrbücher verwendet werden. So wird etwa mit dem Ausdruck “`FOR EACH DEDGE e FROM u DO etwas(e) END;`” für jede vom Knoten `u` ausgehende gerichtete Kante `etwas` gemacht. Wegen ihrer Nähe zu den in Büchern angegeben Algorithmenbeschreibungen ist in der Sprache recht leicht zu programmieren. Die Oberfläche stellt einen Texteditor zur Verfügung, in dem gleichzeitig Programme “per Knopfdruck” übersetzt werden können.
Das Zusammenspiel der Komponenten Programmiersprache und Grapheditor macht jedoch erst die besondere Stärke von *1st Grade* aus: die selbstgeschriebenen Algorithmen werden auf den selbstgenerierten Beispielgraphen angewendet, und man hat dabei vielfältige Möglichkeiten, sich die algorithmischen Abläufe im Detail anzuschauen. Dadurch erst ist ein tiefes algorithmisches Verständnis zu erzielen. Aber auch die reine Ergebnisanimation in Verbindung mit einer komfortablen Ablaufkontrolle ist für viele im echten Anwendungsbereich angesiedelte Einsatzmöglichkeiten eine echte Bereicherung.

SIMEX

Mechatronik ist eine der neuen integrativen Disziplinen, die ihr Entstehen der rasanten Entwicklung der Mikroelektronik verdanken. Das Kunstwort Mechatronik wurde aus den beiden Anteilen Mechanik und Elektronik gebildet.
Elektronik wird bei der Integration mit mechanischen Strukturen in Form von digitaler Mikroelektronik zur Informationsübertragung und -verarbeitung von Sensorsignalen, zur Aufbereitung der Sensorsignale in Regelfiltern und zur Ansteuerung von aktiven Komponenten (elektrischen oder hydraulischen Stellgliedern) eingesetzt. So entstehen besonders leistungsfähige Aggregate, die aufgrund ihrer lokalen ”Intelligenz” weitgehend selbständig reagieren und sich sehr schnell und effizient an geänderte Umgebungsbedingungen anpassen können.
Dabei ist wichtig, daß zur Schaffung neuer Produkte von besonderer Leistungsfähigkeit und Flexibilität die Integration mechanischer Strukturen mit der digitalen Informationsverarbeitung vom Anfang einer Systementwicklung an erfolgen muß. Voraussetzung dafür sind neuartige Entwurfsverfahren und -werkzeuge, die sich grundsätzlich des Rechners bedienen.
Vor diesem Hintergrund hat die Paderborner Forschungsgruppe **MLaP** (Mechatronik Laboratorium

Paderborn, Leitung Professor Lückel) in mehr als zehnjähriger Arbeit das Softwaresystem CAMeL (Computer Aided Mechatronic Laboratory) entwickelt. Zentrale Vorstellung bei dieser Entwicklung ist die möglichst anschauliche Realisierung eines Versuchslabors im Rechner, mit allen wesentlichen dem Ingenieur geläufigen Geräten. Eine Hauptschwierigkeit bei dieser Arbeit bestand darin, eine funktionsorientierte Beschreibung von Systemen in einer normierten Standardform zu entwickeln, die eine aufwendige und umfangreiche algorithmische Computerverarbeitung erlaubt und die dennoch die in den verschiedenen Fachdisziplinen der Ingenieuerwissenschaften entstandenen häufig sehr unterschiedlichen Systembeschreibungsformen nachbildet oder in vereinfachter Form wiedergibt.

CAMeL selbst besteht neben *Simex* (Softwarepaket zur Modellierung und Simulation nichtlinearer Systeme) aus den Hauptgruppen *Lintools* (Werkzeuge zur Analyse linearer Systeme) und *Linex* (Softwarepaket zur Parameteroptimierung von Systemen mit Vektorkriterium)

Der Modul *Simex* des Softwaresystems CAMeL, in einer etwas eingeschränkten Installation für PC-Rechner, wurde vom ASK (Akademische Software Kooperation) Jurorenteam aus insgesamt 185 eingereichten Programmen ausgewählt und mit dem deutsch-österreichischen Software-Preis 1992 als herausragende Lehrsoftware aus dem Fachbereich Maschinenbau ausgezeichnet.

Als Preis wurde eine HP-900/700 RISC-Workstation im Wert von rd. 30 000.-DM vergeben

Durchbruch digitaler Bildtechnologien

Jan Felix Fuchssteiner

Die Photokina 1992 stand ganz im Zeichen der Markteinführung neuer digitaler Bildsysteme.

Vor gut zwei Jahren, am 18. September 1990 erblickte die Kodak Photo-CD in New York, Tokio und London fast gleichzeitig das Licht der Welt. Ihr wird eine goldene Zukunft bevorstehen, zumindest wenn man den Prophezeihungen beim "gelben Riesen" Kodak aus Rochester Glauben schenken darf. Dort wurde nämlich in Kooperation mit Philips und Sun Microsystems eine kleine, goldschimmernde Scheibe entwickelt, die wie eine simple Musik-CD aussieht, aber in der Lage ist den gesamten Fotosektor zu revolutionieren. Die Umstellung von der herkömmlichen, chemischen Fotografie zur digitalen Technik mit vielen neuen Anwendungsmöglichkeiten, wird weit gemächlicher ablaufen, als einige Hersteller konventioneller Filme zunächst befürchtet haben. Denn mit der Photo-CD sieht alles nach einer friedlichen Koexistenz, ja sogar nach einer erfolgversprechenden Zusammenarbeit beider Medien aus. Kodak ist es gelungen, "die Vorzüge der konventionellen Silberhalogenid-Fotografie mit den attraktiven Seiten digitaler Bilder zu verschmelzen". Profitieren soll das System von der weltweiten Verbreitung der 35mm Kleinbildkamera mit chemischem Film, der für hohe Qualität bei einfacher und preiswerter Aufnahmetechnik sorgt. Den ca. 250 Millionen Anwendern wird die elektronische Bildverarbeitung, sowie das Versenden der Fotos über Funk oder Telefon erschlossen, ohne daß eine neue Ausrüstung notwendig wird: es bleibt die gesamte Aufnahmetechnik mit Kameras, Objektiven und Filmen erhalten, und auch an den liebgewonnenen Fotografiergewohnheiten der Anwender braucht sich nichts zu ändern.

Während Negative und Bilder schon nach wenigen Jahren unwiederbringlich verblassen, an Brillanz verlieren und die Gelatineschicht ständig der Gefahr eines zerstörerischen Pilzbefalls ausgesetzt ist, glänzt die CD mit einer Datensicherheit von mindestens 30 Jahren. Auch ist das Kopieren auf andere digitale Medien ohne Qualitätsverluste möglich. Alle diese Vorzüge könnten auch der, mehr als 150 Jahre alten, chemischen Fotografie neuen Auftrieb geben. Das Kodak Photo-CD-System setzt nämlich erst *nach* den bisherigen Verarbeitungsschritten, nach der Entwicklung des Dia-, Negativ- oder Schwarzweißfilmes, ein. Der Fotograf gibt seinen Film wie gewohnt zum Entwickeln, kann aber auf der Auftragstüte des entsprechend ausgestatteten Foto-Finishers ankreuzen, ob er seine Bilder gleich auf eine Photo-CD überspielt haben möchte. Auch ausgewählte Einzelbilder kann man, gegen einen kleinen Preisaufschlag, digitalisieren lassen.

Die überaus hohe Bildqualität, welche die Auflösung der Still-Video-Kameras um weit mehr als das 10-fache übertrifft, wird durch den Kodak DCD-Film-Scanner realisiert. Dieser arbeitet mit einer neuartigen Scantechnologie, bei der drei parallel angeordnete CCD-Einheiten (jede mit bis zu 2048 Pixeln) das Bild abtasten. Die Bilddaten werden dann an eine Sun SPARCStation weitergeleitet, an der dann noch Korrekturen von kleineren Aufnahme- und Verarbeitungsfehlern vorgenommen werden: Helligkeit, Kontrast, Bilddichte und Farbcharakteristik werden optimiert. Die Datenflut von über 20 Megabyte wird durch eine neuartige Kompressionsmethode auf nur 6,5 MB reduziert. Mithilfe eines Laserstrahls werden diese Informationen dann auf eine CD-ROM eingebrannt – WORM-Prinzip: Write Once, Read Multiple. Es ist zwar nicht möglich die Daten zu löschen, wohl aber können auf eine halbvolle CD noch nachträglich weitere Bilder gespeichert werden. Insgesamt finden 100 Farbfotos in digitalisierter und komprimierter Form auf einer Photo-CD Platz. Um dem Kunden die Ordnung in seinem CD-Archiv zu erleichtern, wird ein Index-Print für die CD-Hülle angefertigt, auf dem alle Bilder im Kleinstformat mitsamt den zugehörigen Suchnummern ausgedruckt sind.

Mit der bespielten Photo-CD stehen dem Anwender dann alle Möglichkeiten der modernen Verarbeitung und Präsentation offen. Digitales Bildvergnügen auf dem Fernsehschirm erwartet ihn, wenn er das goldene Scheibchen in einen von Philips modifizierten CD-Player gibt, der selbstverständlich auch Audio-CD's abspielen kann. Doch wehe dem, der die musikalischen Qualitäten der Photo CD in einem normalen CD-Player auf die Probe stellen will. Er wird mit einem dröhnendem, nicht sehr melodischen Ton bestraft.

Will man die vielen kreativen, komfortablen und zeitsparenden Möglichkeiten der Bildbearbeitung am Computer nutzen, muß die CD in einem CD-ROM-XA (eXtended Architecture) Laufwerk abgespielt werden. Dabei dürfte es in Zukunft auch keine Probleme mehr geben, da viele namhafte Hersteller der Hard- und Softwarebranche die Übernahme des neuen Standards bereits angekündigt haben, was ihnen durch eine offene Lizenzpolitik der Entwickler erleichtert wird. So hat zum Beispiel Apple dem neuen Betriebssystem 7.1 eine verbesserte Version von QuickTime mit auf den Weg gegeben, die das Kodak-Format umwandeln und die Bilder entkomprimieren kann.

Doch damit sind nicht alle Probleme aus der Welt geschafft: die älteren CD-Rom Laufwerke sind meistens noch nicht "multisession-fähig". Das heißt sie können keine Daten lesen, die in "mehreren-Sitzungen" überspielt worden sind, da sie die verschiedenen Inhaltsverzeichnisse, die dabei angelegt werden nicht verstehen. Das ist kein Grund zum Verzweifeln: die Bilder des ersten Arbeitsganges werden meistens ohne Klagen gelesen. Doch wer läßt schon 100 Fotos auf einmal überspielen?

Auf dem Macintosh-Bildschirm erscheint eine kleine, symbolisierte Photo-CD. Führt man mit dem Mauszeiger den berühmten Mac-Doppelklick aus, kommen neue Symbole ans Licht, für die QuickTime verantwortlich ist. Zum einen wäre da das "Slide-Show"-Bildchen. Wird es aktiviert, sieht man eine Dia-Show, die alle Bilder der CD in geringer Auflösung der Reihe nach abspielt. In dem "Photos-Ordner" befinden sich weitere Ordner, welche die Bilder in den verschiedenen Auflösungen enthalten. Den Programmen stehen die Bilder im PICT Format zur Verfügung. Dieses Format ist für den Computer weitaus einfacher zu handhaben als das Kodak YCC-Format, welches speziell für die Anwendung im Fernsehbereich (HDTV, CD-I) gedacht ist. Doch nun ist die CD auch für dem Computer gut lesbar: ein erster Erfolg der Apple-Strategie, die erreichen will, daß "Fotografien für den Computeranwender genauso einfach zu benutzen sind, wie Buchstaben" (Eric Zarakov, zuständig für die Integration der Photo-CD bei Apple).

Bis auf die Bilder im 3072 * 2048-Ordner (deren 6 Millionen Pixel aufwendiger entkomprimiert werden müssen) lassen sich alle problemlos öffnen. Um sie bearbeiten zu können, werden sie in ein Bildbearbeitungsprogramm wie z.B. Photoshop eingelesen. Die Bilder sind fast allesamt etwas zu hell und zu flau geraten. Also vermindert man die Helligkeit und erhöht den Kontrast. Leider können die verbesserten Bilder nicht auf eine neue Photo CD überspielt werden. Etwas derartiges ist aber für Ende dieses Jahres geplant.

Die Photo-CD ist der Mittelpunkt eines umfassenden Systems, das auf der Photokina noch ausgebaut wurde. So gibt es nun eine Profi-CD, die Bilddaten von 4*5 Inch Dias (90 MB) aufnehmen kann.

Zum Entdigitalisieren von Computerbildern in hochauflösender Foto-Qualität, dient der XC 7700 Digital Halbtondrucker von Kodak. Dieser arbeitet nach dem Thermo-Transfer Prinzip, bei dem Farbband und Emfängermaterial auf einer Transporttrommel zusammengeführt werden. Wird ein Thermo-Element aktiviert, überträgt sich der Farbstoff vom Band auf ein Spezialpapier oder eine Folie. Die Bilder können bis zum Format 28*28 ausgegeben werden, und sind in ihrer Qualität von herkömmlich hergestellten Prints kaum zu unterscheiden.

Gänzlich anderer Natur ist das Agfa Digital-Print-System, das bereits jetzt in einigen Labors zur Herstellung hochwertiger Abzüge vom Dia eingesetzt wird. Während die sorgfältige Verarbeitung von Negativfilmen fast immer für gute Bilder sorgt, bereiten Prints vom Dia den Foto-Finishern ein schwer zu lösendes Problem: der große Kontrastumfang hat Einbußen bei den Licht- und Schattendetails zur Folge. Auch sind die in der Projektion hell leuchtenden Farben auf der Diadirektvergrößerung oft nur unbefriedigend blaß und stumpf. Dank Agfa's langjährigen Erfahrungen in der Scannertechnologie werden diese Schwierigkeiten nun geschickt eliminiert. Ein CCD-Element tastet zunächst das Dia in den drei Grundfarben mit je 6 Millionen Bildpunkten ab. Die Farbauszüge werden dann, ähnlich wie bei dem Kodak-System, in 0,16 Sekunden digital optimiert. Anschließend werden sie in ein Negativ umgewandelt, das über eine Kathodenstrahlröhre auf normales Farbnegativpapier der Größen 9*13 bis 20*30 belichtet wird. Zusätzlich ist es beim Agfa Digital-Print-System auch möglich, digitale Vorlagen von Photo-CD, Wechselplatte oder Diskette zu verarbeiten. Damit bekommt der Computernutzer erstmals die Möglichkeit, seine digital veränderten Bilder kostengünstig in ein konventionelles Photo zurückzuverwandeln. Diesen Service sollen die Großlabors bereits nächstes Jahr anbieten können.

Neben dieser Chemie-Elektronik Ehe, gibt es aber auch einige Kameras, die ganz auf Film verzichten können. Von Kodak waren auf der Photokina digitale Rückteile für Nikon Kameras zu sehen. Das DCS (Digital Camera System), welches sich aus einer unmodifizierten Nikon F3, einer speziellen Rückwand und einer externen Speichereinheit zusammensetzt, richtet sich besonders an Bildjournalisten, für die Geschwindigkeit an erster Stelle steht. Deshalb bietet sie auch die, für eine elektronische Kamera hohe, Aufnahmefrequenz von 2,5 Bildern pro Sekunde. Zusätzlich ist ein Modemanschluß in der Recheneinheit integriert und der Bildausschnitt kann auf einem eingebauten SW-Monitor begutachtet und verändert werden. Der CCD-Chip zur Bildaufzeichnung liefert eine Auflösung von 1,3 Millionen Bildpunkten. Die Bilddaten werden in der Digital Storage Unit verwaltet und verarbeitet. Die DSU verfügt uber eine 200 Megabyte Festplatte, auf der in komprimiertem Zustand bis zu 600 Bilder maximaler Auflösung Platz finden. Beim Fotografieren wird die Kamera wie vom Standardmodell gewohnt bedient. Berücksichtigen muß man allerdings, daß sich der Bildwinkel des verwendeten Nikon-Objektivs verengt, da der Bildsensor nur die Hälfte des Kleinbildformates aufweist. Aus einem 28mm Weitwinkel wird so ein Objektiv mit Normalbrennweite. Im Sucher wird zur Ausschnittwahl ein Rahmen in der Größe des Chips eingeblendet. Man kann, vom Blitz bis hin zu sämtlichen Objektiven, das schon vorhandene Zubehör der F3 weiterverwenden. Das System arbeitet in der Farbversion mit frei wählbaren Empfindlichkeiten von 100 bis 1600 ASA, bei SW-Einstellung von 200 bis 3200 ASA. Leider verschlechtert sich die Bildqualität bei zunehmender Empfindlichkeit merklich. Das System ist ausgesprochen unhandlich. Schon das stattliche Gewicht der mit der speziellen Rückwand ausgestatteten F3 fällt unangenehm auf. Die überaus schwere, um den Hals gehängte Speichereinheit von mehr als 6 Kilo, fordert einen sportlich durchtrainierten Fotografen, insbesondere wenn dieser auch noch auf einige Wechseloptiken Wert legt. Kompakter und preisgünstiger ist hingegen die DCS 200, welcher eine Nikon F-801s zugrunde liegt, die ohne Monitor, Rechner, Modemanschluß und mit 50 MB Festplatte auskommen muß. Die maximale Bildfrequenz von nur einem Bild alle drei Sekunden schließt einige Einsatzbereiche, wie etwa die Sportfotografie aus.

Einen ähnlichen Weg wie Kodak ging Minolta mit der Entwicklung der MS-C1100, die auf der konventionellen Dynax SPxi basiert, jedoch mit einer fest angesetzten Rückwand ausgestattet wurde. Diese Kamera speichert die Bilddaten nicht auf einer Festplatte sondern auf einer abgewandelten Form des Digital Audio Tape (DAT). Der Unterschied zur herkömmlichen DAT-Kassette besteht in erster Linie im besonderen, 4mm breiten, Bandmaterial. Die Aufzeichnung geschieht in einem speziellen DAT-Recorder, der keinen mobilen Einsatz ermöglicht. Fur die Aufzeichnung auf DAT-Material spricht insbesondere die hohe Speicherkapazität bei günstigem Preis, denn auf eine ca. 25 DM teure Kassette passen 2000 Bilder à 360 000 Bildpunkten. Diese Kamera soll, mit einem Macro-Objektiv und Ringblitz ausgestattet, insbesondere für die medizinische Dokumentation geeignet sein.
Der Rollei Digital ScanPack, ein an die Rollei 6008 als Rückteil anschließbarer Scanner ermöglicht das Digitalisieren dreidimensionaler Objekte in der hohen Auflösung von 31 Millionen Bildpunkten. Allerdings dauert das Scannen mehrere Minuten, und verbietet daher die Verewigung bewegter Objekte. Der Rollei-Prospekt zeigt eine, für das Scann-Magazin typische Aufnahmesituation: Ein Stilleben, die Kamera und einen Computer, auf dessen Bildschirm das Motiv in voller Farbigkeit leuchtet. Da jedoch bei der Katalogproduktion offenbar kein Mac II zur Verfügung stand, stellte man kurzerhand einen (damals wohl schwarzweißen!) Mac-Classic neben die Szenerie. Über den SW-Monitor wurde nach der Aufnahme digital ein Farbbild kopiert - eine unfreiwillige Demonstration der durch die neue Technik gegebenen Manipulationsmöglichkeiten!
Eine SW-Aufnahme bewegter Objekte erlaubt der von Scitex hergestellte Kameraansatz für Hasselblad und Mamiya Kameras mit 4,2 Millionen Pixel. Für Farbaufnahmen müssen dagegen 3 Einzelbelichtungen mit vorgesetzten Farbfiltern angefertigt, und die Farbauszüge dann digital zusammengesetzt werden. Für die Einzelbelichtungen ist, im Gegensatz zum Rollei System, welches eine weitaus höhere Bildqualität liefert, Blitzlicht einsetzbar.
Alles in allem bieten diese Exponate interessante neue Möglichkeiten mit immer noch eingeschränkter Beweglichkeit. Das Ende der traditionellen Silberhalogenid-Fotografie ist trotzdem noch nicht unmittelbar abzusehen, denn schon ein Kleibilddia bietet bei einfacher und preiswerter Verarbeitung die höhe Auflösung von 15-30 Millionen Pixeln. Die digitalen Kameras sind auch vorerst noch in Preisregionen angesiedelt (zwischen 20- und 100-tausend DM), in die sich ein Hobby-Anwender kaum wagen wird. Die digitale Aufnahmetechnik ist jedoch in der Lage in den Bereichen der professionellen Fotografie Fuß zu fassen, in denen die Chemie durch zu lange Verarbeitungszeit hinderlich ist. Wer seine Bilder unmittelbar nach der Aufnahme an seine Tante in Kanada, Sibirien oder anderswo elektronisch übermitteln will, wird deshalb noch eine Weile warten mussen. Wer aber den rasanten Preisverfall auf dem Elektroniksektor kennt, der weiß, daß der Wunsch von heute schon morgen finanzierbare Realität ist.